

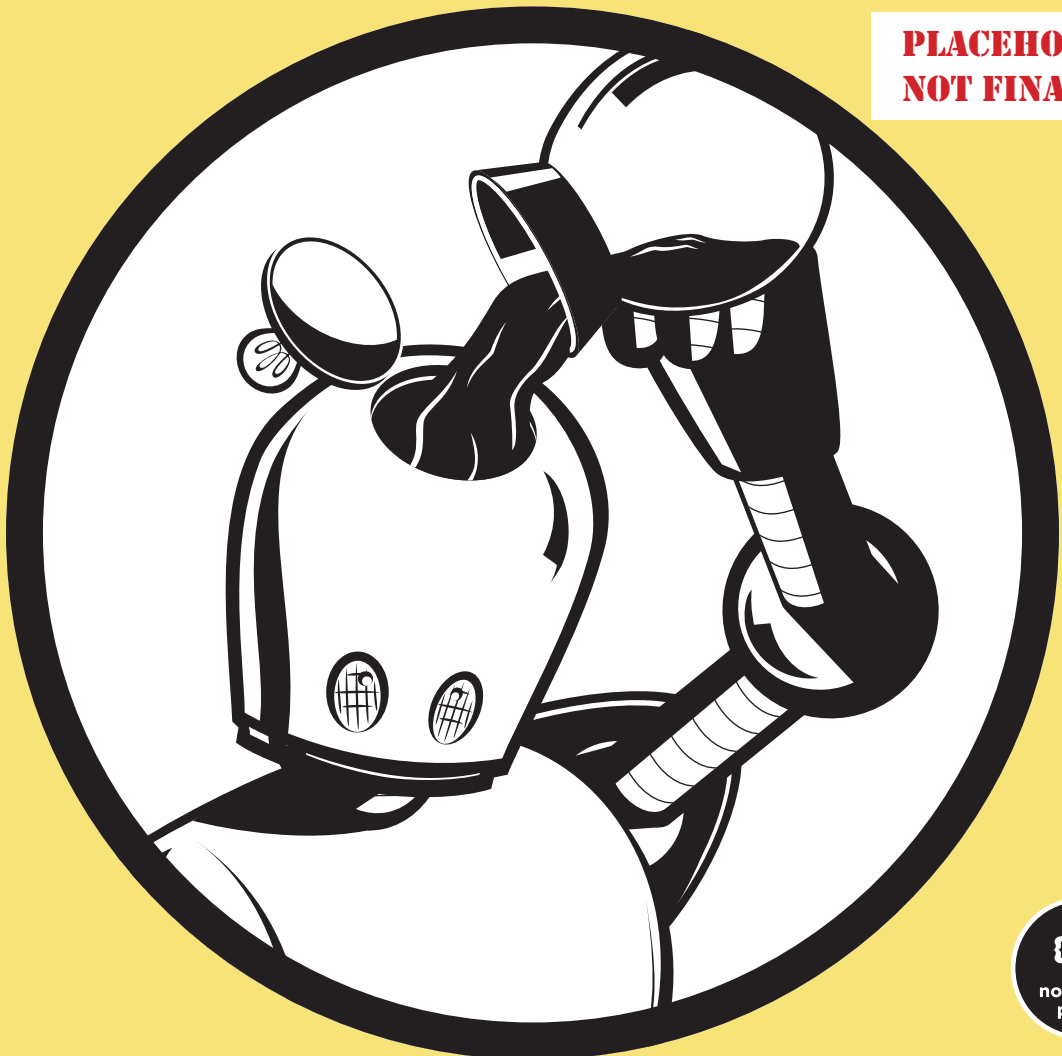
**EARLY
ACCESS**

The Linux Memory Manager (Early Access) © 2025 by Lorenzo Stoakes

THE LINUX MEMORY MANAGER

LORENZO STOAKES

**PLACEHOLDER
NOT FINAL**



NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best dis-cretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes. Please note that any online resources may not be available until the book is complete.

If you encounter any errors (“errata”) you’d like to report, please fill out this [Google form](#) so we can review your comments.

THE LINUX MEMORY MANAGER.

Lorenzo Stoakes

Early Access edition, 2/7/25

Copyright © 2025 by Lorenzo Stoakes.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-7185-0446-2(print)

ISBN-13: 978-1-7185-0447-9 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock

Managing Editor: Jill Franklin

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author(s) nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

For Sabina, my lovely wife, without whose support this book would not have been possible.
For Cherry, Mary, Linus and Jim the cats, who were no help at all*.

*. Their emotional support, however, was indispensable.

BRIEF CONTENTS

Chapter 1: Introduction	1
Chapter 2: Physical Memory	5
Chapter 3: Virtual Memory	123
Chapter 4: Process Memory	209
Chapter 5: Memory Mapping	271
Chapter 6: Page Faults	345
Chapter 7: Reverse Mappings	435
Chapter 8: Manipulating Userland Memory	537
Chapter 9: The Page Cache	625
Chapter 10: Writeback	795
Chapter 11: Reclaim and Memory Pressure	939
Chapter 12: Swap Memory	1113
Chapter 13: The Out Of Memory (OOM) Killer	1177
Chapter 14: Practical Memory Management	1217
Index	1287

CONTENTS IN DETAIL

1		
INTRODUCTION		1
1.1	Approach.....	1
1.2	Who Is This Book For?	2
1.3	Book Overview	2
2		
PHYSICAL MEMORY		5
2.1	struct page	8
2.1.1	Metadata	11
2.1.2	struct slab	24
2.1.3	Page flags	28
2.2	struct folio	32
2.3	Physical memory model	39
2.3.1	Sections.....	40
2.3.2	PFN validity	42
2.3.3	Converting between PFN and section	42
2.3.4	Page block flags	43
2.3.5	Looking up struct page/struct folio objects	45
2.4	Nodes and Zones	46
2.4.1	Low memory reserve	53
2.4.2	Total reserved pages	57
2.5	Migrate types	59
2.6	GFP flags	61
2.6.1	Physical address zone modifiers	61
2.6.2	Page mobility and placement hints	62
2.6.3	Watermark modifiers	63
2.6.4	Reclaim modifiers	63
2.6.5	Action modifiers	64
2.6.6	Predefined GFP flag combinations	65
2.6.7	Memalloc Flags	66
2.7	Buddy allocator	67
2.7.1	Algorithm	67
2.7.2	Free lists	69
2.7.3	Per-CPU free Pages (PCPs)/Pagesets	70
2.8	Allocator implementation	72
2.8.1	Visualising the allocator	72
2.8.2	__alloc_pages()	79
2.8.3	rmqueue()	91

2.8.4	rmqueue_buddy()	92
2.8.5	rmqueue_pcplist()	93
2.8.6	Migrate type fallback	96
2.8.7	New page preparation	109
2.9	Freeing pages	110
2.9.1	__free_one_page()	117

3 VIRTUAL MEMORY 123

3.1	Page Tables	125
3.1.1	Page Table Operations	128
3.1.2	Page Table Flags	135
3.1.3	Page Flag Combinations	139
3.1.4	Page Table Traversal	141
3.1.5	Page Table Locking	142
3.2	The Address Space	143
3.3	Direct Mapping	147
3.3.1	Bootstrapping	148
3.3.2	Direct Mapping Initialization	149
3.4	An Introduction to the Transaction Lookaside Buffer	175
3.5	The Kernel Virtual Memory Allocator (vmalloc)	176
3.5.1	Finding a Free Block	182
3.5.2	Inserting a Newly Allocated Block	195
3.5.3	Physical Allocation	197
3.5.4	Virtual Mapping	203

4 PROCESS MEMORY 209

4.1	Overcommit	211
4.2	Userland memory from 50,000 feet	214
4.2.1	Describing userland memory	214
4.3	The process address space	216
4.3.1	mm_struct reference counting	221
4.3.2	The initial process address space	222
4.3.3	Kernel PGD maintenance	224
4.3.4	Process address space locking	228
4.3.5	Process address space flags	240
4.4	Virtual Memory Areas (VMAs)	243
4.4.1	VMA flags	247
4.4.2	Allocation and freeing	251
4.4.3	VMA layout	252
4.4.4	VMA insertion/removal	253
4.4.5	VMA traversal	259
4.5	An introduction to the page cache	263
4.6	Per-process memory management statistical counters	267

5	MEMORY MAPPING	271
5.0.1	Program break	272
5.0.2	mmap()	278
5.0.3	mmap map flags	279
5.0.4	mmap kernel implementation	283
5.0.5	Choosing where to map	300
5.0.6	Unmapping memory	310
5.1	VMA merge and split	316
5.1.1	VMA merge	316
5.1.2	Mergeability	323
5.1.3	VMA adjust	325
5.1.4	VMA split	338
6	PAGE FAULTS	345
6.1	Hardware page fault handling	346
6.1.1	Kernel page faults	348
6.1.2	Userland page faults	356
6.1.3	Success conditions	363
6.1.4	Informational	364
6.1.5	Error conditions	364
6.2	Page fault handling	367
6.2.1	Edge cases	377
6.2.2	Page flags	378
6.3	Anonymous page fault	378
6.3.1	Zero page	381
6.3.2	anon_vma preparation	382
6.3.3	Physical page allocation	383
6.3.4	Setting up the PTE entry	384
6.3.5	Adding to the reverse mapping and LRU and setting the PTE	385
6.4	Non-anonymous page fault	385
6.4.1	Folio locks	387
6.5	Non-anonymous read page fault	388
6.6	Non-anonymous MAP_PRIVATE Copy on Write page fault	393
6.7	Non-anonymous shared write fault	394
6.8	Shared non-anonymous fault logic	397
6.9	Write-protected page fault	402
6.9.1	Non-anonymous folios	406
6.9.2	Anonymous folios	406
6.9.3	Folio/PFN sharing	408
6.9.4	Page reuse	413
6.9.5	Folio copying	414
6.10	Stack expansion	420
6.11	Userland bad area handling	424
6.12	Special mappings	427

7**REVERSE MAPPINGS****435**

7.0.1	Anonymous reverse mappings	436
7.0.2	Key points	442
7.0.3	File reverse mappings	443
7.0.4	Anonymous reverse mapping types	445
7.0.5	Anonymous reverse mapping initialisation	449
7.0.6	Reusing adjacent VMA's anon_vma objects	454
7.0.7	Connecting anon_vma objects	457
7.0.8	Cloning anon_vma objects	458
7.0.9	Reusing anon_vma objects on fork	460
7.0.10	Forking anon_vma objects	461
7.0.11	VMA split and merge	463
7.0.12	Folio anon_vma operations	465
7.0.13	File-backed reverse mapping folio operations	472
7.0.14	Unlinking anon_vma objects	474
7.0.15	Walking the reverse mapping	476
7.0.16	Walking the VMA	485
7.0.17	Summary	493
7.1	Freeing userland memory and the TLB	493
7.1.1	Unmapping memory mapped regions	494
7.1.2	MMU gather initialisation	495
7.1.3	VMA unmapping	499
7.1.4	Zapping memory ranges	501
7.1.5	Freeing page tables	516
7.1.6	Flushing the TLB	524
7.1.7	Freeing pages	532
7.1.8	Lazy TLB mode	535

8**MANIPULATING USERLAND MEMORY****537**

8.1	Accessing User Memory from the Kernel	537
8.1.1	User Access	538
8.1.2	Get User Pages (GUP)	544
8.1.3	Pinning Folios	546
8.1.4	Follow Flags	549
8.1.5	GUP Functions	551
8.1.6	Walking Page Tables	575
8.1.7	Faulting in Pages	585
8.1.8	Fast GUP Functions	587
8.1.9	GUP Helper Functions	603
8.2	Userland Memory Manipulation APIs	606
8.2.1	mlock()	606
8.2.2	mprotect()	612
8.2.3	mremap()	612
8.2.4	madvise()	613

9**THE PAGE CACHE****625**

9.1	The Virtual File System (VFS)	626
9.2	A Brief Digression: eXtensible Arrays (xarrays)	629
9.2.1	API	641
9.2.2	The Rest	645
9.3	Reading From a File	646
9.4	File-Backed Read Faults	656
9.5	Reading Page Cache Entries	670
9.5.1	Reading a Batch of Folios From the Page Cache	671
9.5.2	Reading a Batch of Folios Into the Page Cache	676
9.5.3	Reading a Single Folio From the Page Cache	684
9.5.4	Reading a Single Folio Into the Page Cache	692
9.5.5	Adding Folios to the Page Cache	695
9.6	Reading Folios From Disk	698
9.7	Readahead	701
9.7.1	Synchronous Readahead	706
9.7.2	Asynchronous Readahead	713
9.7.3	On-Demand Readahead	715
9.7.4	Common Readahead Code	724
9.7.5	Physical Readahead	729
9.8	Fault-Around	731
9.9	Removing Page Cache Entries and Truncation	738
9.9.1	Removing Folios from the Page Cache	739
9.9.2	Folio Truncation	743
9.9.3	Unmapping Folios	754
9.9.4	Dropping Caches	759
9.9.5	Folio Eviction	765
9.10	Buffers and Block I/O	766
9.10.1	An Introduction to Buffer Heads	766
9.10.2	blockdev	769
9.10.3	Accessing Blocks	772
9.10.4	Block Writes	777
9.10.5	Block I/O (BIO) Operations	779
9.10.6	Buffer Head I/O Operations	780
9.11	Folio Locking and Waiting	781

10**WRITEBACK****795**

10.1	Dirty Tracking in the Kernel	796
10.2	Marking the Folio Dirty	798
10.3	Marking the inode Dirty	799
10.4	Page Fault Dirty Tracking	803
10.5	File Write Dirty Tracking	810
10.6	Synchronising to Disk	816
10.7	sync	817

10.8 syncfs	819
10.9 fsync	821
10.10 Writing Back to Disk	824
10.11 File System and Background Writeback	842
10.12 Flusher Thread Operation	855
10.13 File Writeback	889
10.14 Dirty Throttling	897
10.15 Rate-Limited Dirty Throttling	902
10.16 Dirty Throttle Statistics	905
10.17 Bandwidth Updates	907
10.18 Dirty Position Control Ratio	911
10.19 Dirty Limits	916
10.20 Dirty Poll Interval	920
10.21 Dirty Rate Limit	921
10.22 Maximum and Minimum Pause Time	926
10.23 Core Dirty Throttling	928
10.24 Writeback Chunk Size	935
10.25 Background Writeback Threshold	936

11

RECLAIM AND MEMORY PRESSURE

939

11.1 Physical Allocation Slow Path	941
11.2 LRU Vectors	956
11.2.1 struct lruvec	961
11.2.2 lruvec Operations	963
11.3 Direct Reclaim	967
11.4 Indirect Reclaim	976
11.4.1 Initialisation	976
11.4.2 kswapd Thread	979
11.4.3 kswapd Sleeping	983
11.4.4 Node Balancing	987
11.5 The Reclaim Mechanism	998
11.5.1 The Scan Control Object	999
11.5.2 A Brief Overview of the Working Set	1004
11.5.3 Shrinking the Node	1007
11.5.4 Determining Scan Balance	1020
11.5.5 Shrinking LRU Vectors	1029
11.5.6 Shrinking an Individual LRU List	1037
11.5.7 Shrinking the Active List	1038
11.5.8 Isolating LRU Folios	1042
11.5.9 Shrinking the Inactive List	1048
11.5.10 Reclaim Folio Reference Checking	1065
11.5.11 Reclaim Page Out	1067
11.5.12 Reclaim Mapping Removal	1070
11.6 Reclaim Throttling	1072
11.6.1 General Reclaim Throttling	1072

11.6.2	Direct Reclaim Throttling	1077
11.7	Folio Batches	1082
11.7.1	CPU Folio Batches	1083
11.7.2	LRU Rotation	1084
11.7.3	mlock	1084
11.7.4	Folio Operations	1085
11.7.5	Adding Folios to a Batch	1088
11.7.6	Folio Activation	1093
11.7.7	Folio Rotation	1094
11.7.8	Folio Deactivation	1096
11.7.9	File Folio Deactivation	1097
11.7.10	Folio Lazy Free	1099
11.7.11	mlock Folio Batch	1101
11.7.12	Folio Batch Drain	1106

12

SWAP MEMORY

1113

12.1	The Swap Cache	1114
12.1.1	Swapper Initialisation	1116
12.1.2	Assigning Folios to the Swap Cache	1118
12.1.3	Page Table Mappings	1119
12.2	Swapping Out	1122
12.2.1	Adding a Folio to the Swap Cache	1127
12.2.2	Setting Page Table Swap Entries	1132
12.2.3	Swapping Out to Disk	1137
12.2.4	Freeing a Swapped Out Folio	1140
12.3	Swapping In	1144
12.3.1	Page Fault on a Swapped Out Page	1145
12.3.2	Looking Up a Folio in the Swap Cache	1155
12.3.3	Reading Swapped Out Folios From Disk	1157
12.3.4	Swap Cluster Readahead	1163
12.3.5	Swap VMA Readahead	1168

13

THE OUT OF MEMORY (OOM) KILLER

1177

13.1	Causes of Out of Memory Conditions	1178
13.1.1	Memory Allocation	1178
13.1.2	sysrq-f	1180
13.2	OOM Killer Score Adjustment	1180
13.2.1	OOM Score	1181
13.3	OOM Killer Alternative Behaviours	1182
13.3.1	vm.panic_on_oom	1182
13.3.2	vm.oom_kill_allocating_task	1182
13.4	Kernel Interface	1182
13.4.1	Allocating Memory at Risk of OOM	1185

13.4.2	Manual OOM Killer Invocation Via sysrq-f	1188
13.4.3	The Out of Memory Killer	1189
13.4.4	Victim Selection	1196
13.4.5	Victim Killing	1201
13.4.6	OOM Reaper	1207

14

PRACTICAL MEMORY MANAGEMENT

1217

14.1	Measuring free memory	1219
14.1.1	Kernel implementation of MemAvailable	1221
14.1.2	Higher order folios	1223
14.1.3	Summary	1224
14.2	Measuring the memory usage of a process	1224
14.2.1	Proportional Set Size	1226
14.2.2	Summary	1226
14.3	Memory mapping using mmap()	1226
14.3.1	Mapping anonymous memory	1227
14.3.2	Mapping a file	1228
14.3.3	Private file mapping	1230
14.3.4	Fixed mappings	1232
14.4	Interpreting Out Of Memory reports	1233
14.4.1	Failed allocation statistics	1237
14.4.2	Stack trace	1238
14.4.3	Global free page statistics	1239
14.4.4	Per-node statistics	1242
14.4.5	Zone-specific statistics	1243
14.4.6	Zone buddy page statistics	1246
14.4.7	Global statistics	1248
14.4.8	Out of memory killer report	1249
14.5	procfs memory interfaces	1251
14.5.1	A quick tour: Physical memory	1254
14.5.2	A quick tour: Virtual memory	1259
14.5.3	A quick tour: Page table introspection	1265
14.6	Memory tunables	1270
14.7	Sharing memory	1276
14.7.1	System V shared memory	1276
14.7.2	POSIX shared memory	1279
14.7.3	Anonymous shared memory across forked processes	1281
14.7.4	Sharing memory via memfd	1283

INDEX

1287

1

INTRODUCTION

The memory management subsystem is arguably the core of the Linux kernel, forming the foundation upon which the rest of the kernel and all of userspace operates. Despite its fundamental nature, it is surprisingly complicated and intricate.

The Linux Memory Manager explores the Linux kernel’s memory management subsystem in detail. In this chapter we will explore the approach taken, who the book is aimed at,

1.1 Approach

Many kernel books take the approach of providing an overview and then hand-waving away details, that is a top-down approach..

The key motivating philosophy of the Linux Memory Manager was to do entirely the opposite—explore each part of the memory management subsystem starting with basic principles, and then work from the bottom-up to form a broader understanding of the topic at hand.

Within the kernel there can be no better source of truth than the source code itself, so we explore kernel functionality by exploring its code, which is reproduced in extensive snippets.

In effect, this book tries to combine a source commentary with in depth analysis of concepts explored in both extensive discussions of the concepts

at hand along side a large number of diagrams which aim to put each concept into perspective.

As the kernel is constantly evolving and this book necessarily must target a static kernel version, the intent is that by taking this approach, not only will the reader acquire understanding of fundamentals which are unlikely to change, but more importantly develop their skills at exploring kernel source so that they can adapt and update their knowledge to the latest version of the code.

1.2 Who Is This Book For?

This book is aimed at developers who already have some fundamental understanding of the C programming language and operating system basics, and have an interest in exploring under the hood to see how Linux manages memory in as much detail as they may want to do so.

This spans from that most wonderful of things—a genuinely curious (perhaps aspiring?) kernel hacker to a professional kernel developer, either working with or making use of the memory management subsystem.

The intent is to bring out the hidden tribal knowledge within this code-base as much as possible and in the spirit of open source software make it more widely available to curious hackers all over the world!

1.3 Book Overview

Chapter 1: Introduction A description of the contents of the book.

Chapter 2: Physical Memory A description of how a system's memory is managed on its most fundamental level—the allocation of its physically installed RAM.

Chapter 3: Virtual Memory The why, what and how, page tables, page table flags, page table sizes, virtual memory layout, direct mapping, kernel/userland split and a detailed description of `vmalloc()`.

Chapter 4: Process Memory An overview of how userland memory is structured—`mm_struct`, Process VMAs, Copy-on-Write and more.

Chapter 5: Memory Mapping A description of how memory mapping is performed within the kernel - `mmap`, `brk` (used by `malloc()`) and how mappings (via VMAs) are split/merged.

Chapter 6: Page Faults A detailed examination of how page faults are handled and propagated within the kernel.

Chapter 7: The Reverse Mapping A detailed look at how the kernel maps raw physical pages of userland memory back to the abstract VMA representation, in addition to how this is used to free memory.

Chapter 8: Manipulating Userland Memory A detailed examination of how the kernel accesses userland memory. Also a brief look at how memory ranges can be manipulated by `madvise()`.

- Chapter 9: The Page Cache** A detailed look into the page cache, how it interacts with the Linux virtual file system, read-ahead, read-behind and generally a very focused discussion of how the memory subsystem interacts with VFS.
- Chapter 10: Writeback** A description of how writeback proceeds both originating from write() operations and memory-mapped files, how dirty data is tracked, how synchronisation functions within the kernel and how dirty throttling is applied.
- Chapter 11: Reclaim and Memory Pressure** A detailed explanation of how reclaim operations, what direct and indirect reclaim are, how it interacts with demand paging, higher order page starvation, etc.
- Chapter 12: Swap Memory** A detailed description of how the swap operations in the kernel and how pages are paged out and paged back in again.
- Chapter 13: The Out of Memory (OOM) Killer** A deep dive into how it works, how to tune it, the what, why and how.
- Chapter 14: Practical Memory Management** A brief overview of practical memory management techniques - procfs interfaces, tuneables, decoding out of memory reports and more. Generally a practical 'how to' for sysadmins/developers.

2

PHYSICAL MEMORY

Physical memory describes all of the memory on a system which can be addressed arbitrarily by at least one core. Typically this will be in the form of RAM modules but could include other forms of random-access stores. Over the course of this chapter we will explore how the kernel manages, allocates and abstracts this resource.

Physical addresses tell the CPU where to find a particular byte of memory. For example, DRAM stores each individual byte on a specific row and column in a specific array known as a bank, accessed by a specific channel on a DIMM stored in a specific slot. The physical address encodes all of this information.

Within the kernel, physical addresses are assigned the `phys_addr_t` type which is equal to the word size, e.g. for a 64-bit system this is simply an unsigned 64-bit integer.

If we were to try to manage every byte of memory individually the overhead associated with trying to keep a track of it would exceed the amount of available memory in the system. This is obviously intractable so we have to compromise and subdivide memory into aggregate blocks, which we call *pages*.

The *base page* size of an architecture is the hardware-configured minimum supported page size and thus the fundamental unit of memory. For x86-64 this is 2^{12} or 4,096 bytes, for arm64 and other architectures however this can vary. In the Linux kernel the base page size is defined to be `PAGE_SIZE` bytes (This is architecture-specific and so declared in the appropriate `arch/` header file).

We subdivide pages further into *nodes* and *zones*. Let's examine what each of these mean:

Large server boxes may support more than once CPU socket with some memory modules attached to one CPU, some attached to another and so on, with some slow means of interconnect between remote blocks of memory.

Systems with memory arranged in this configuration are said to have a Non-Uniform Memory Access (NUMA) memory architecture.

NOTE

In addition, systems may even have blocks of memory that don't have a CPU associated with them and are slow for any core to access, or are attached to a CPU with locality to other memory also attached to the same CPU, but possess other characteristics which reduces the speed of access to them.

Even single-CPU systems on modern hardware are typically treated as NUMA, only with a single NUMA node defined which owns all physical memory.

In order for the kernel to take these heterogeneous memory blocks into account it subdivides them into physically contiguous domains known as *nodes*, with each page of physical memory belonging to one and only one node.

Within each node memory is further divided based on physical address *ranges* into areas known as *zones*. This is either because that range of memory naturally possesses an important characteristic or because a range of memory has been arbitrarily assigned to a particular purpose. Zones overlap nodes.

It is important to note that for a modern system which has NUMA enabled, physical memory allocation will always be parameterised by node and zone (even if they may have been automatically determined by a function higher up the call stack).

In order to efficiently allocate memory the kernel allocates memory in power-of-two numbers of base pages at a time with the *order* of an allocated page being equal this power-of-two e.g. an order-3 allocation consists of $2^3 = 8$ base pages of memory.

Each base page of physical memory has metadata associated with it which is stored in struct `page` objects, with aggregate blocks of 1 or more base pages represented as a struct `folio` object.

For convenience we define the useful notion of a *Page Frame Number* (PFN) which is simply the physical address divided by the page size, in other words if you were to conceive of physical pages as elements in an array, the PFN is the index of a page in that array. `PFN_PHYS()` and `PHYS_PFN()` are used to convert between the two.

Memory is further divided into *page blocks* (the order of which is defined by `pageblock_order`, 9 on x86-64, i.e. 512 pages). These are the smallest number of pages which can have a *migrate type* applied to them. A migrate type determines whether pages can be *migrated* i.e. moved around, described by `enum migratetype`. More on this later.

Let's examine an example physical memory layout in Figure 2-1.

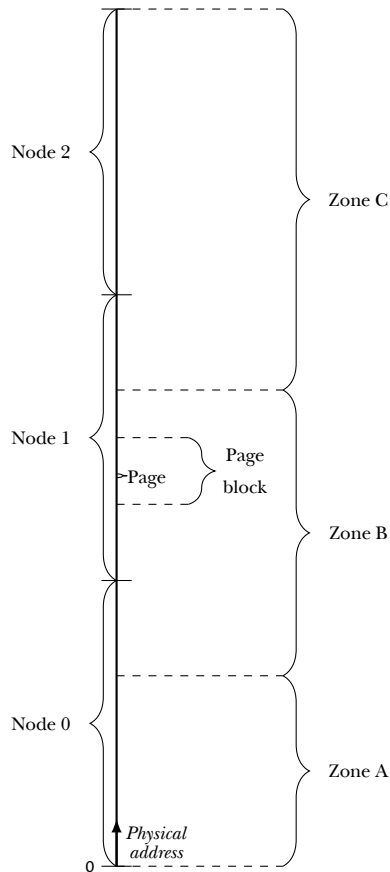


Figure 2-1: Example physical memory layout

The line represents physical memory from zero to the maximum available to the system.

Note that this diagram excludes memory *holes*, (blocks of physical memory inaccessible for use as ordinary memory—either empty, reserved or used by devices) for simplicity.

Each node and zone subdivides the physical address space into contiguous blocks of memory, with zones overlapping nodes. Each node and zone are page-aligned so base pages belong definitely to a node and a zone.

NOTE

It is actually possible for distinct nodes and zones to overlap one another, for instance in cases where access times for memory in the overlapping region are equivalent in each node, however for the sake of brevity we will assume these ranges are distinct.

2.1 struct page

In order to manage pages of memory we must store metadata about them—flags to identify what characteristics the page has, reference count to determine whether we can free the page or not and a whole host of other metadata specific to how they are used.

NOTE

Rather than confuse matters and include every permutation of kernel configuration and platform we focus on modern 64-bit, little-endian architectures with sensible configurations. This might seem like heresy, but trying to cover everything would render this book unreadable and my sanity unknowable.

The data type that encapsulates this data is `struct page` which, for modern 64-bit systems contains:

- A `flags` field which specifies attributes of the page and additionally encodes zone and optionally node information (it can also encode ‘section’ information under certain configuration options, this is out of scope at this point.) set via `set_page_zone()` and `set_page_node()` and accessed via `page_zonenum()` and `page_to_nid()`.
- A 40-byte *metadata union*, a union of a series of data structures which varies depending on what the page is being used for. This is used to store pertinent metadata to the page depending on type, or if used for ‘slab’ pages (see the slab chapter for more on this) the whole `struct page` is cast to a `struct slab` object, which shares `flags`, `_refcount` and `memcg_data` fields but overwrites the metadata union and `_mapcount/page_type` fields.
- If not a slab page—a union of either a `_mapcount` variable (which should be accessed via `page_mapcount()`) counting page table mappings if the page is mappable in userland (or a `page_type` field giving further details on the page type).
- A reference count `_refcount` field (which should be accessed via `page_ref_count()`) used to determine if this page can be freed typically incremented by `get_page()` and decremented by `put_page()`.
- A `memcg_data` field used to store data pertinent to memory cgroups, typically a pointer to a cgroup data structure.

In total the size of the structure on a modern x86-64 system is 64 bytes (the typical size of a L1 cache line) and is typically laid out in such a way that it is accessed *contiguously* in memory meaning that `struct page` objects can be accessed very efficiently.

Each `struct page` represents the smallest page size the kernel (and typically the system) supports (e.g. for x86-64 this is 4 KiB), however physically contiguous pages can be compounded together resulting in *compound*

pages allocated as a block. In order to square this circle we designate the first struct page of a compound page a *head* page and the remainder *tail* pages.

Compound pages will always consist of a power-of-2 number of base pages equal to 2^{order} and referred to as ‘order-N’ pages, e.g. an order-0 page is $2^0 = 1$ base page in size, an order-1 page is $2^1 = 2$ base pages in size, an order-2 page is $2^2 = 4$ base pages in size and so forth.

To add to the confusion, it is possible to allocate higher order pages without them being compound – in which case it is assumed the owner of the page simply knows which struct page represents the higher-order page. In order to obtain a compound page, the `__GFP_COMP` flag must be set (more on GFP flags later).

The terminology is confusing here – both higher order pages and base pages are referred to simply as *pages*. A good rule of thumb is, unless qualified as a ‘compound’ page or an ‘order-N’ page, it is usually safe to assume that the term ‘page’ refers to a page of base size, especially when expressed as a unit e.g. ‘137 pages’.

Overall the philosophy of struct page is to utilise the space taken up by each object as efficiently as possible, squeezing out the available space for all its worth. As a result there is a heavy use of unions overloading the use of each offset within the struct as well as the aforementioned casting of struct page to struct slab (this object is used for slab allocation, see the slab chapter for more on this).

Historically when a struct page has been referred to (for example as a function parameter), it has not been possible to guarantee that it wasn’t a tail page. The solution until recently has been to always invoke `compound_head()` which, if it was indeed a tail page exchanged it for its head.

However rather sensibly the kernel has moved towards explicitly indicating that these parameters are *not* tail pages so that this uncertainty can be eliminated. This is achieved by using the struct folio type which maps one-to-one to a struct page but is *always* either non-compound or a head page. Helper functions exist to ease the transition as interfaces are updated, and for the moment struct folio contains a top-level union with a struct page so it is very easy to convert between the two. More on this later.

After excising fields that aren’t relevant to a modern x86-64 system the definition is as follows, as shown in Listing 2-1.

```

72 struct page {
73     unsigned long flags;           /* Atomic flags, some possibly
74                                   * updated asynchronously */
75     /*
76      * Five words (20/40 bytes) are available in this union.
77      * WARNING: bit 0 of the first word is used for PageTail(). That
78      * means the other users of this union MUST NOT use the bit to
79      * avoid collision and false-positive PageTail().
80      */
81     union {
82         :                          /* ... (metadata union) ... */
186    };

```

```

187
188     union {          /* This union is 4 bytes in size. */
189         /*
190          * If the page can be mapped to userspace, encodes the number
191          * of times this page is referenced by a page table.
192          */
193         atomic_t _mapcount;
194
195         /*
196          * If the page is neither PageSlab nor mappable to userspace,
197          * the value stored here may help determine what this page
198          * is used for. See page-flags.h for a list of page types
199          * which are currently stored here.
200          */
201         unsigned int page_type;
202     };
203
204     /* Usage count. *DO NOT USE DIRECTLY*. See page_ref.h */
205     atomic_t _refcount;
206
207 #ifdef CONFIG_MEMCG
208     unsigned long memcg_data;
209 #endif
210 :
229 } _struct_page_alignment;

```

Listing 2-1: include/linux/mm_types.h: struct page definition

The `_struct_page_alignment` define specifies that the page should be aligned to double system word size (e.g. 16 bytes for 64-bit systems) if `CONFIG_HAVE_ALIGNED_STRUCT_PAGE` is set which it typically will be on modern systems. This permits the SLUB allocator to perform atomic operations at this size for better performance. Ultimately for x86-64 and arm64 this should have no impact as in both cases struct page will be of cache line size – i.e. 64 bytes, as shown in Listing 2-2.

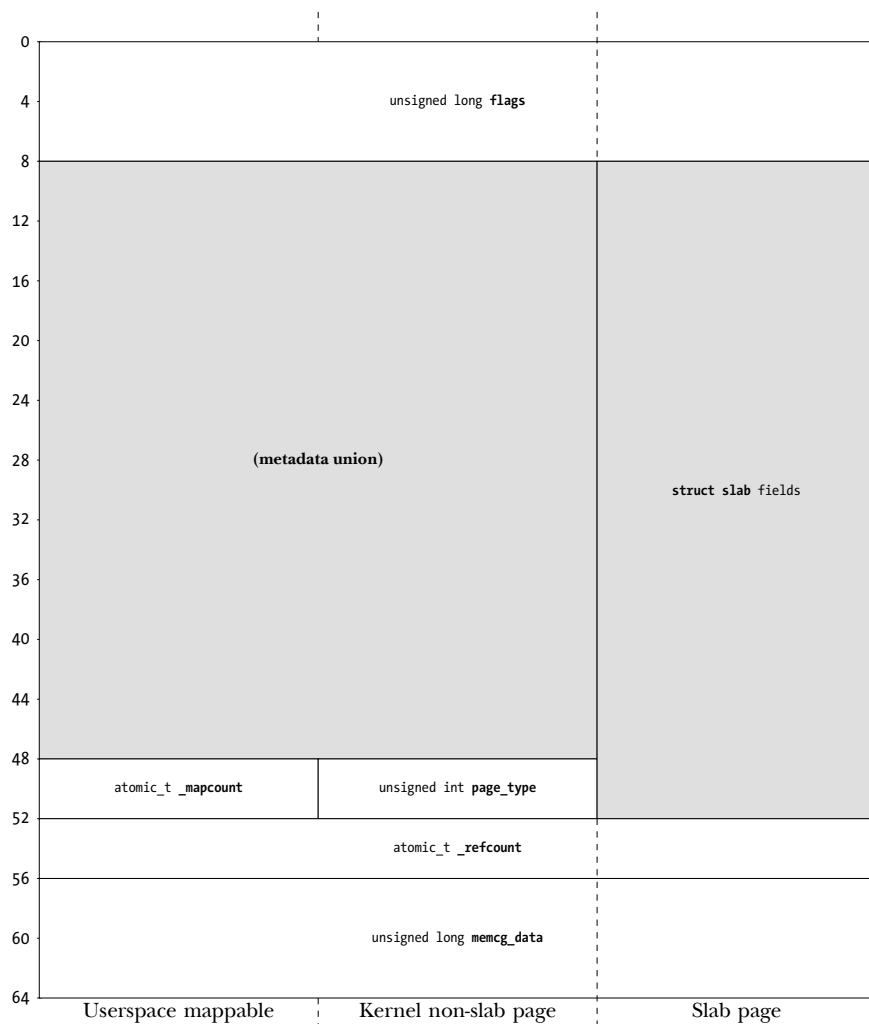
```

66 #ifdef CONFIG_HAVE_ALIGNED_STRUCT_PAGE
67 #define _struct_page_alignment __aligned(2 * sizeof(unsigned long))
68 #else
69 #define _struct_page_alignment
70 #endif

```

Listing 2-2: include/linux/mm_types.h: _struct_page_alignment

In graphical form (the digits on the left indicate byte offset, the vertical divisions indicate either different entries in a union or being cast to another type) as shown in Figure 2-2.

Figure 2-2: *struct page* layout for a typical 64-bit architecture

2.1.1 Metadata

2.1.1.1 Anonymous/page cache pages

If a page is either allocated as a simple ‘anonymous’ page (i.e. not tied to a file and not used for any special purpose requiring more metadata) or is a part of the page cache (a page containing data for a memory-mapped file, either explicitly mapped in or cached automatically, see later chapter on this topic) then the metadata union used is, as shown in Listing 2-3.

```

82      struct {          /* Page cache and anonymous pages */
83          /**
84           * @lru: Pageout list, eg. active_list protected by
85           * lruvec->lru_lock. Sometimes used as a generic list

```

```

86         * by the page owner.
87     */
88     union {
89         struct list_head lru;
90
91         /* Or, for the Unevictable "LRU list" slot */
92         struct {
93             /* Always even, to negate PageTail */
94             void *__filler;
95             /* Count page's or folio's mlocks */
96             unsigned int mlock_count;
97         };
98
99         /* Or, free page */
100        struct list_head buddy_list;
101        struct list_head pcpl_list;
102    };
103    /* See page-flags.h for PAGE_MAPPING_FLAGS */
104    struct address_space *mapping;
105    pgoff_t index; /* Our offset within mapping.
106
107    /**
108     * @private: Mapping-private opaque data.
109     * Usually used for buffer_heads if PagePrivate.
110     * Used for swp_entry_t if PageSwapCache.
111     * Indicates order in the buddy system if PageBuddy.
112     */
113    unsigned long private;
114    };

```

Listing 2-3: include/linux/mm_types.h: Anonymous page/page cache metadata union struct

This consists of:

- `lru/mlock_count` – A union of either a list head field, `lru`, which is used to connect the page to a list maintained in possibly Least Recently Used (LRU) order or, if the page contains LRU list metadata itself, then the field is `mlock_count` a count of the number of times this memory has been `mlock()`d.
- `mapping` – A rather deceptively typed and named field – it points to a page cache-specific struct `address_space` `mapping` field if this page is part of the page cache (as one might expect), however the lower 2 bits are given over to flags and if the `PAGE_MAPPING_ANON` flag is set *alone* it will instead point to a struct `anon_vma` object (see later chapter on this topic). If the page does not appear on LRU lists (termed ‘non-LRU’) it may set the `PAGE_MAPPING_MOVABLE` flag in which case the mapping points at a struct `movable_operations` object. Finally, if the flags are set to `PAGE_MAPPING_KSM` (a combination of the prior two) it points to a private

KSM data structure. KSM is *Kernel Samepage Merging*, which allows for memory pages to be de-duplicated, typically used by virtual machines. All rather confusing!

- `index` – If this page is within a page (or swap) cache this field indicates the offset of this page within the mapped file.
- `private` – A field containing private data that the filesystem which owns the page can use for its own purposes (see the `PG_private` comment for more details).

NOTE

A list head is a structure containing pointers to previous and next nodes in a doubly-linked ring list, implemented neatly by keeping nodes within structures and using the statically known offset of the entry from the start of the structure to be able to retrieve node contents at compile-time, see `include/linux/list.h` to see the implementation.

We examine this in graphical form in Figure 2-3.

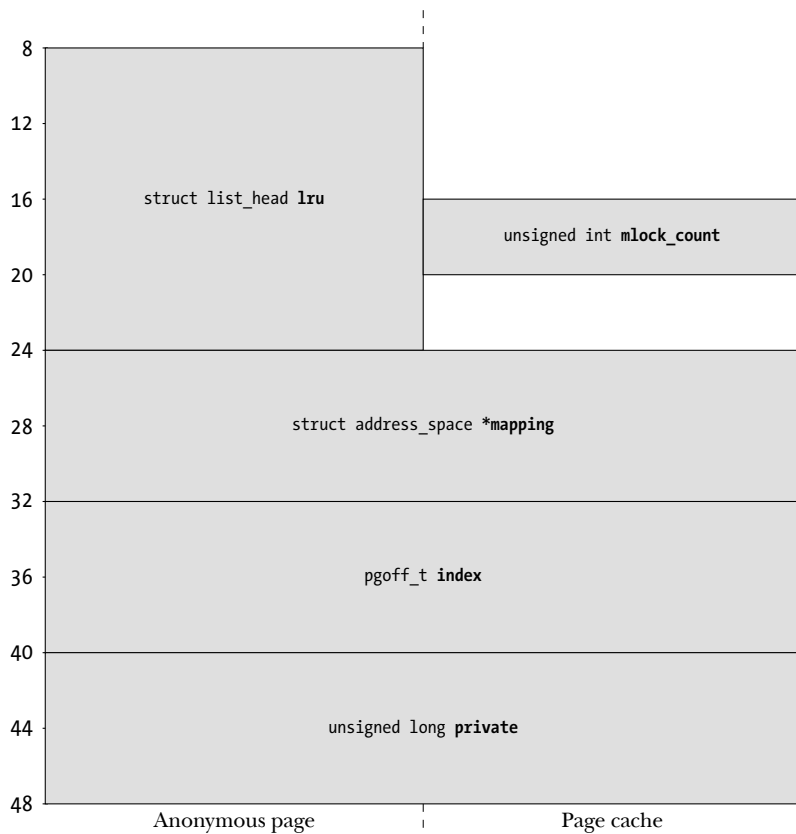


Figure 2-3: Anonymous page/page cache metadata union struct

2.1.1.2 Tail pages

Tail page metadata is represented in a slightly less clear fashion – two unions containing overlapping information from the following structs, as shown in Listing 2-4.

```

136         struct {          /* Tail pages of compound page */
137             unsigned long compound_head;    /* Bit zero is set */
138
139             /* First tail page only */
140             unsigned char compound_dtor;
141             unsigned char compound_order;
142             atomic_t compound_mapcount;
143             atomic_t compound_pincount;
144 #ifdef CONFIG_64BIT
145             unsigned int compound_nr; /* 1 << compound_order */
146 #endif
147         };
148         struct {          /* Second tail page of compound page */
149             unsigned long _compound_pad_1; /* compound_head */
150             unsigned long _compound_pad_2;
151             /* For both global and memcg */
152             struct list_head deferred_list;
153         };

```

Listing 2-4: include/linux/mm_types.h: Tail page metadata union structs

Every tail page has the `compound_head` field set which is a pointer to the head page with the lower bit additionally set (this is how we identify tail pages via `PageTail()`). This is set by `set_compound_head()`, as shown in Listing 2-5.

```

806 static __always_inline void set_compound_head(struct page *page, struct page *
      head)
807 {
808     WRITE_ONCE(page->compound_head, (unsigned long)head + 1);
809 }

```

Listing 2-5: include/linux/page-flags.h: `set_compound_head()`

The call stack for this is `prep_compound_page()` which calls `prep_compound_tail()`, as shown in Listing 2-6.

```

801 static void prep_compound_tail(struct page *head, int tail_idx)
802 {
803     struct page *p = head + tail_idx;
804
805     p->mapping = TAIL_MAPPING;
806     set_compound_head(p, head);
807 }

```

Listing 2-6: mm/page_alloc.c: `prep_compound_tail()`

Note that this sets the mapping field to the known ‘poisoned’ value `TAIL_MAPPING` to mark these pages as being tail pages to make them more easily identifiable when debugging.

The *first* tail page contains additional information on the compound page (since a page is compound if it contains more than one of the minimum page size, we know for sure there will be at least one tail page to place this in).

We set this additional data in `prep_compound_head()` which is also invoked by `prep_compound_page()` which sets the `compound_dtor`, `compound_order`, `compound_mapcount`, `compound_pincount` and `compound_nr` fields as shown in Listing 2-7.

```

793 static void prep_compound_head(struct page *page, unsigned int order)
794 {
795     set_compound_page_dtor(page, COMPOUND_PAGE_DTOR);
796     set_compound_order(page, order);
797     atomic_set(compound_mapcount_ptr(page), -1);
798     atomic_set(compound_pincount_ptr(page), 0);
799 }
```

Listing 2-7: mm/page_alloc.c: prep_compound_head()

Note that we set the initial mapcount field to -1 to indicate that no mapping has yet been established. Examining the setters and getters referenced here in Listing 2-8.

```

891 static inline void set_compound_page_dtor(struct page *page,
892     enum compound_dtor_id compound_dtor)
893 {
894     VM_BUG_ON_PAGE(compound_dtor >= NR_COMPOUND_DTORS, page);
895     page[1].compound_dtor = compound_dtor;
896 }
```

Listing 2-8: include/linux/mm.h: set_compound_page_dtor()

The compound order is set in `set_compound_order()` as shown in Listing 2-9.

```

905 static inline void set_compound_order(struct page *page, unsigned int order)
906 {
907     page[1].compound_order = order;
908 #ifdef CONFIG_64BIT
909     page[1].compound_nr = 1U << order;
910 #endif
911 }
```

Listing 2-9: include/linux/mm.h: set_compound_order()

We obtain the compound mapcount via `compound_mapcount_ptr()` as shown in Listing 2-10.

```

309 static inline atomic_t *compound_mapcount_ptr(struct page *page)
```

```

310 {
311     return &page[1].compound_mapcount;
312 }

```

Listing 2-10: include/linux/mm_types.h: compound_mapcount_ptr()

Finally, we obtain the compound pincount via `compound_pincount_ptr()` as shown in Listing 2-11.

```

314 static inline atomic_t *compound_pincount_ptr(struct page *page)
315 {
316     return &page[1].compound_pincount;
317 }

```

Listing 2-11: include/linux/mm_types.h: compound_pincount_ptr()

The *order* referred to here defines the number of pages compounded together – the number of base pages contained in a compound page is equal to 2^{order} . using a power-of-2 allows us to use a *buddy allocator* to efficiently allocate (more on that later!) The maximum order is equal to `MAX_ORDER - 1` (typically 10, i.e. $2^{10} = 1024$ pages, on x86-64 this is 4 MiB).

You can see that each field being set is offset by 1 from the head page – confirming that this data is stored in the *first* tail page.

We can see how these are invoked by `prep_compound_page()` in Listing 2-12.

```

809 void prep_compound_page(struct page *page, unsigned int order)
810 {
811     int i;
812     int nr_pages = 1 << order;
813
814     __SetPageHead(page);
815     for (i = 1; i < nr_pages; i++)
816         prep_compound_tail(page, i);
817
818     prep_compound_head(page, order);
819 }

```

Listing 2-12: mm/page_alloc.c: prep_compound_page()

Here `__SetPageHead()` sets a flag to indicate that the page is a head page, before invoking the functions we have explored above. These flag options are generated using macros, e.g. `__SETPAGEFLAG()` and ultimately set the flags field in the struct page object.

The `deferred_list` field is used by the transparent huge page (see later chapter on this topic) logic to store individual constituent pages for deferred processing, typically accessed via `page_deferred_list()`.

Huge pages (page sizes larger than the smallest supported page size) must consist of a compound page so we know will have at least two tail pages and therefore can safely reference a second tail page, as shown in Listing 2-13.

```

294 static inline struct list_head *page_deferred_list(struct page *page)
295 {
296     /*
297      * See organization of tail pages of compound page in
298      * "struct page" definition.
299      */
300     return &page[2].deferred_list;
301 }

```

Listing 2-13: include/linux/huge_mm.h: *page_deferred_list()*

We examine this in graphical form in Figure 2-4.

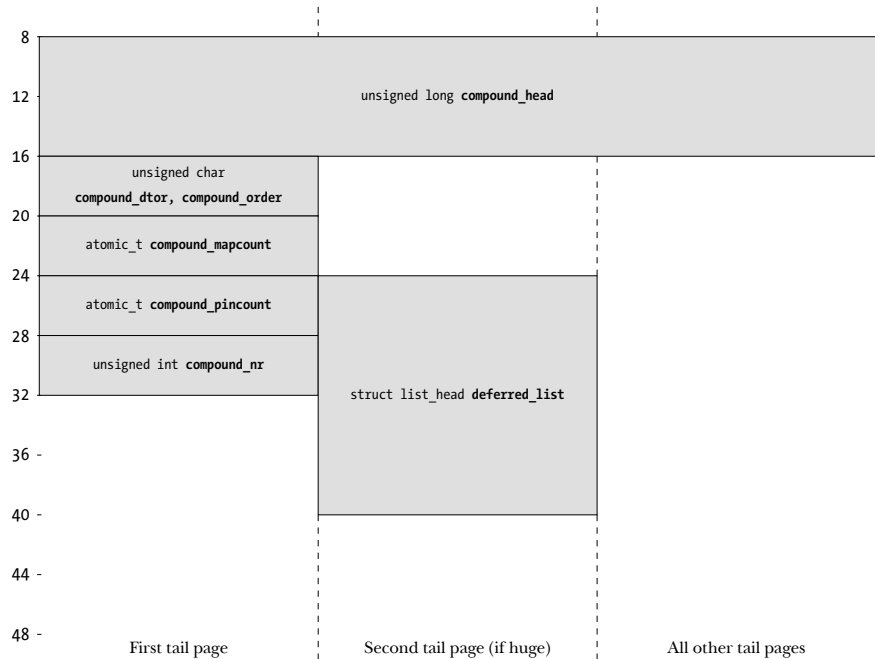


Figure 2-4: Tail page metadata union fields

Reviewing what each field is used for:

- `compound_head` – Pointer to the *head page* of the compound page with its high bit set (e.g. if the head page resided at `0xf00ba100` then this value would be `0xf00ba101` and require the lower bit to be masked off). A common task for a function that takes a `struct page` as an argument is to check whether the page is in fact a tail page and in that case obtain its head. This is achieved through the `compound_head()` macro (more on this later!) this field allows us to determine if a page is a tail page via `PageTail()` or more generally `compound` via `PageCompound()`.

- `compound_dtor` – This determines the *destructor* for the compound page, i.e. the function to invoke on freeing of the page. This is one of enum `compound_dtor_id` (see Listing 2-14) which is used to index into the `compound_page_dtors` array (see Listing 2-15) to invoke the appropriate destructor in `destroy_large_folio()`.

```

877 /* Keep the enum in sync with compound_page_dtors array in mm/page_alloc.c
      */
878 enum compound_dtor_id {
879     NULL_COMPOUND_DTOR,
880     COMPOUND_PAGE_DTOR,
881 #ifdef CONFIG_HUGETLB_PAGE
882     HUGETLB_PAGE_DTOR,
883 #endif
884 #ifdef CONFIG_TRANSPARENT_HUGEPAGE
885     TRANSHUGE_PAGE_DTOR,
886 #endif
887     NR_COMPOUND_DTORS,
888 };

```

Listing 2-14: `include/linux/mm.h`: `enum compound_dtor_id`

We examine the compound page destructor array in Listing 2-15.

```

410 compound_page_dtor * const compound_page_dtors[NR_COMPOUND_DTORS] = {
411     [NULL_COMPOUND_DTOR] = NULL,
412     [COMPOUND_PAGE_DTOR] = free_compound_page,
413 #ifdef CONFIG_HUGETLB_PAGE
414     [HUGETLB_PAGE_DTOR] = free_huge_page,
415 #endif
416 #ifdef CONFIG_TRANSPARENT_HUGEPAGE
417     [TRANSHUGE_PAGE_DTOR] = free_transhuge_page,
418 #endif
419 };

```

Listing 2-15: `mm/page_alloc.c`: `compound_page_dtors[]`

- `compound_order` – As previously discussed, this defines the order of the possibly-compound page, expressed as a power-of-two – an order-1 page is $2^1 = 2$ base pages in size (e.g. 8 KiB for x86-64), an order-2 page is $2^2 = 4$ base pages in size and so on.
- `compound_mapcount` – A count of the number of times the compound page as a whole has been mapped into virtual memory. This supersedes the `_mapcount` field in any of individual struct page components of the the compound page and is referenced instead by functions such as `compound_mapcount()/folio_entire_mapcount()` and `page_mapcount()` if the page is compound.
- `compound_pincount` – When the compound page constitutes user memory and these pages are ‘pinned’ in place (i.e. made such that they cannot

be moved or paged out, more on user pages and pinning later), this field is used to store a precise count of the number of times the compound page has been pinned.

- `compound_nr` – A field added for convenience in 64-bit architectures which is equal to $2^{\text{compound\_order}}$ i.e. the number of base pages this compound page occupies.
- `deferred_list` – As described previously, this is used as part of the transparent huge page functionality, see later chapter on this topic.

2.1.1.3 Page table pages

Page tables (see later chapter on this topic) require a lock to synchronise changes, and typically it is efficient to ‘split’ these locks at the lowest two page table levels (designated ‘PTEs’ and ‘PMDs’ – we go into more detail on what these mean in the next chapter) so there are separate locks for each individual page. Additionally there are some huge page and architecture-specific metadata that is convenient to store in the essentially ‘free’ data store that is a struct page in use. Examining the struct as shown in Listing 2-16.

NOTE

Note as with the struct page code listing we are assuming `ALLOC_SPLIT_PTLOCKS` is not set which is the case for modern 64-bit architectures as this parameter is determined by whether the spinlock size exceeds the number of bytes in an unsigned long which will not be the case unless lock debugging is configured.

```

154         struct {           /* Page table pages */
155             unsigned long _pt_pad_1;           /* compound_head */
156             pgtable_t pmd_huge_pte; /* protected by page->ptl */
157             unsigned long _pt_pad_2;           /* mapping */
158             union {
159                 struct mm_struct *pt_mm; /* x86 pgds only */
160                 atomic_t pt_frag_refcount; /* powerpc */
161             };
162             :
163             :
164             spinlock_t ptl;
165             :
166             :
167         };

```

Listing 2-16: `include/linux/mm_types.h`: Page table metadata union struct

Note that the padding in this struct ensures the tail page bit in `compound_head` is clear so `PageTail()` will return false, and as a non-tail page efforts space is left to ensure the mapping field from the page cache struct is null (you will note that the metadata union structs each go to pains to ensure this is the case – the struct page will be zeroed other than fields in use). Similarly, where the mapping would be for the anonymous/page cache entry is kept zeroed to avoid mistaking the page for either an anonymous or page cache one.

Considering each field:

- `pmd_huge_pte` – This is used by the transparent hugepage functionality (see later chapter on this topic) to tie a potentially huge *PMD* to an underlying fallback *PTE*. Its type is `pgtable_t` which for x86-64 and arm64 is typedef'd to a struct `page*`. If `USE_SPLIT_PMD_PTLOCKS` is set (typically the case, see below) then `pmd_huge_pte()` references this field.
- `pt_mm` – This is used by the x86 architecture to store a pointer from the PGD page table to its relevant struct `mm_struct` (more on this later!)
- `pt_frag_refcount` – This is a PowerPC-specific field (for sake of brevity I examine only x86-64 in this book).
- `ptl` – This lock is used for *PTE* page tables when `USE_SPLIT_PTE_PTLOCKS` is set (when the *maximum* number of cores defined by `CONFIG_NR_CPUS`, which typically defaults to at least 64, is greater than or equal to `CONFIG_SPLIT_PTLOCK_CPUS`, which typically defaults to 4, so this is almost certainly the case for any modern system) and used for *PMD* page tables when `USE_SPLIT_PMD_PTLOCKS` is set (when `USE_SPLIT_PTE_PTLOCKS` and `CONFIG_ARCH_ENABLE_SPLIT_PMD_PTLOCK` are set which is typically the case), otherwise a shared lock is used.

We examine this in graphical form in Figure 2-5.

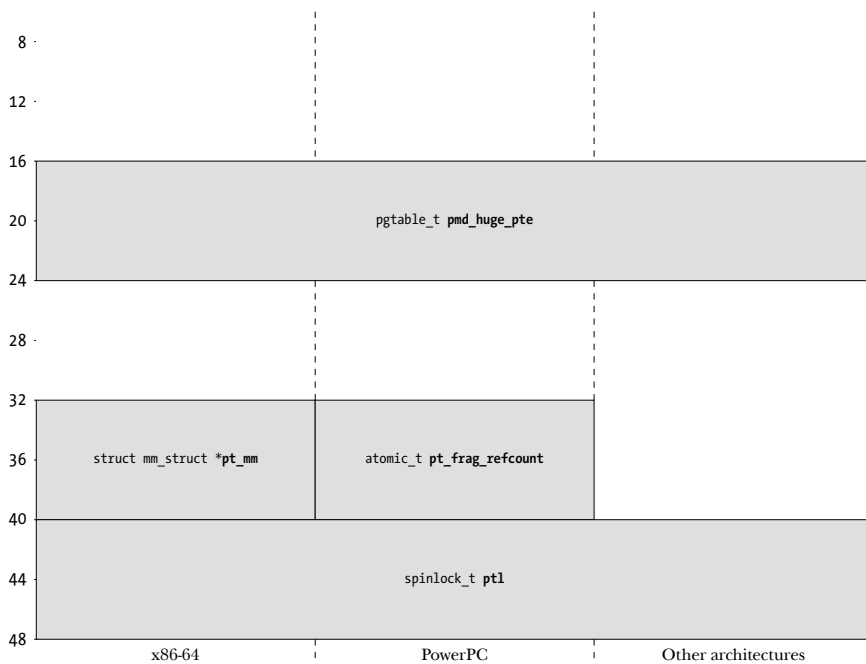


Figure 2-5: Page table metadata union struct

2.1.1.4 Other metadata

There are additional structs contained in the struct `page` metadata union that are very specific and less core to the functioning of the physical memory allocator so I won't go over them in detail, however for completeness let's examine them, starting with the `page_pool` metadata union struct as shown in Listing 2-17.

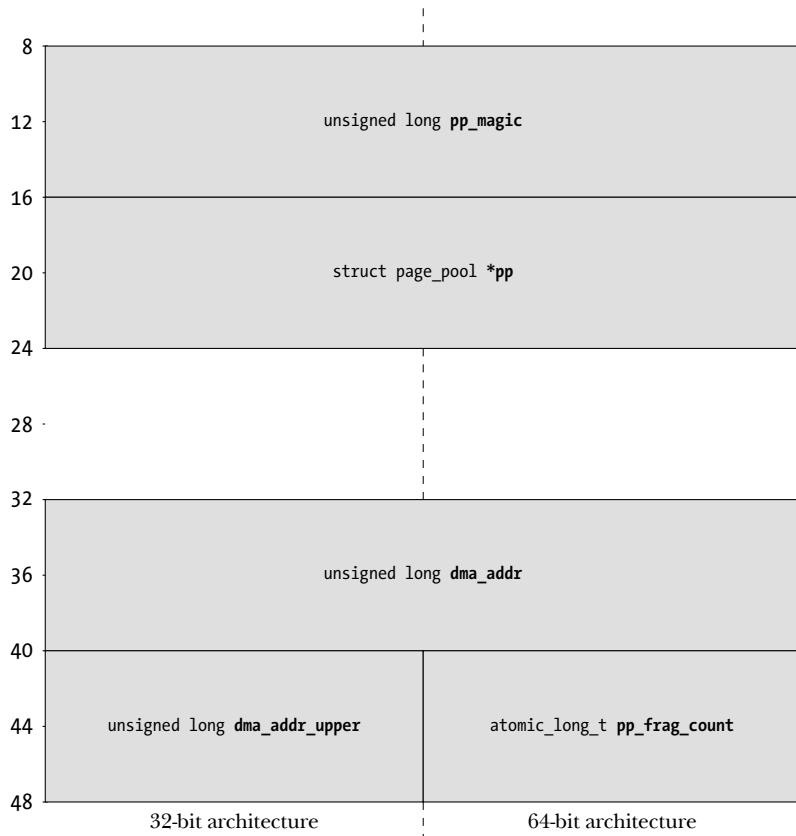
```

114         struct {           /* page_pool used by netstack */
115             /**
116              * @pp_magic: magic value to avoid recycling non
117              * page_pool allocated pages.
118              */
119             unsigned long pp_magic;
120             struct page_pool *pp;
121             unsigned long _pp_mapping_pad;
122             unsigned long dma_addr;
123             union {
124                 /**
125                  * dma_addr_upper: might require a 64-bit
126                  * value on 32-bit architectures.
127                  */
128                 unsigned long dma_addr_upper;
129                 /**
130                  * For frag page support, not supported in
131                  * 32-bit architectures with 64-bit DMA.
132                  */
133                 atomic_long_t pp_frag_count;
134             };
135         };

```

Listing 2-17: `include/linux/mm_types.h`: `page_pool` metadata union struct

We examine the `page_pool` metadata union struct in graphical form in Figure 2-6.

Figure 2-6: `page_pool` metadata union struct

Note that `pp_magic` maintains clear lower bits to ensure it isn't misidentified as a compound tail page via `PageTail()` and equally the mapping offset is kept zeroed to ensure that the page is not inadvertently mistaken for a page cache entry.

We examine the `ZONE_DEVICE` metadata union struct in Listing 2-18.

```

168         struct {           /* ZONE_DEVICE pages */
169             /** @pgmap: Points to the hosting device page map. */
170             struct dev_pagemap *pgmap;
171             void *zone_device_data;
172             /*
173              * ZONE_DEVICE private pages are counted as being
174              * mapped so the next 3 words hold the mapping, index,
175              * and private fields from the source anonymous or
176              * page cache page while the page is migrated to
177              device
178              * private memory.
179              * ZONE_DEVICE MEMORY_DEVICE_FS_DAX pages also
180              * use the mapping, index, and private fields when

```

```

180             * pmem backed DAX files are mapped.
181             */
182         };

```

Listing 2-18: include/linux/mm_types.h: `ZONE_DEVICE` metadata union struct

We examine the `ZONE_DEVICE` metadata union struct in graphical form in Figure 2-7.

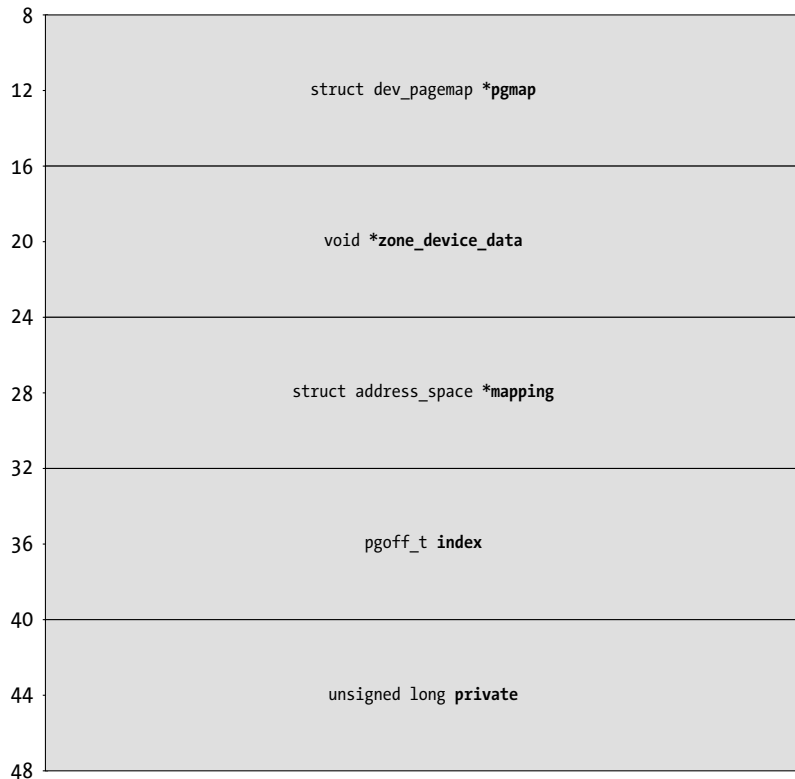


Figure 2-7: `ZONE_DEVICE` metadata union struct

While the struct defines the upper two parameters (again noting that alignment of the `pgmap` pointer will ensure that lower bits are clear to avoid misidentification of the page) the lower three are adapted from the Anonymous page/page cache metadata union struct.

2.1.1.5 RCU head

Finally, the metadata union contains a *Read, Copy, Update (RCU)* data structure – `struct rcu_head` – for use in freeing pages from the slab allocator efficiently. This appears to be unnecessary as `struct slab` declares this separately and shares bits with the `struct page` object.

2.1.2 *struct slab*

The `struct slab` object is declared separately from `struct page` and does not form part of its metadata union but is however cast to/from it, converting from pages to slabs via `page_slab()` and back again via `slab_page()`. It is therefore effectively a metadata union entry, though maintained in a separate data structure and as per figure 2-2.

This object is used to track state relating to the kernel *slab allocator* caching objects for allocation in this page, see later chapter on this topic.

I won't dwell too long on what each field refers to as going into detail on this is better suited to the slab chapter however, briefly, each of the three different slab allocators (SLAB, SLUB and SLOB) typically share the following fields:

- `slab_list` – The doubly-linked list node that allows slab pages to be placed on free lists as deemed appropriate by the specific slab algorithm used. Alignment of this field ensures the lowest bit will be clear and so it won't be mistaken by `PageTail()` for a tail page.
- `rcu_head` – This forms the fundamental data structure permitting the use of *Read, Copy, Update (RCU)* for `struct slab` objects. I will cover the specifics of this in more detail in the slab chapter. Similar to `slab_list` alignment ensures this won't be mistaken for a tail page.
- `slab_cache` – The slab cache object this particular slab belongs to. Note that this coincides with the anonymous page/page cache mapping field so we rely on alignment here to ensure the lower bits will not be set falsely indicating an anonymous or movable page.

We examine the `struct slab` data structure in Listing 2-19.

```

 8  /* Reuses the bits in struct page */
 9  struct slab {
10      unsigned long __page_flags;
11
12  #if defined(CONFIG_SLAB)
13
14      union {
15          struct list_head slab_list;
16          struct rcu_head rcu_head;
17      };
18      struct kmem_cache *slab_cache;
19      void *freelist; /* array of free object indexes */
20      void *s_mem;    /* first object */
21      unsigned int active;
22
23  #elif defined(CONFIG_SLUB)
24
25      union {
26          struct list_head slab_list;
27          struct rcu_head rcu_head;

```

```

28 #ifdef CONFIG_SLUB_CPU_PARTIAL
29     struct {
30         struct slab *next;
31         int slabs;      /* Nr of slabs left */
32     };
33 #endif
34 };
35 struct kmem_cache *slab_cache;
36 /* Double-word boundary */
37 void *freelist;        /* first free object */
38 union {
39     unsigned long counters;
40     struct {
41         unsigned inuse:16;
42         unsigned objects:15;
43         unsigned frozen:1;
44     };
45 };
46 unsigned int __unused;
47
48 #elif defined(CONFIG_SLOB)
49
50     struct list_head slab_list;
51     void *__unused_1;
52     void *freelist;     /* first free block */
53     long units;
54     unsigned int __unused_2;
55
56 #else
57 #error "Unexpected slab allocator configured"
58 #endif
59
60     atomic_t __page_refcount;
61 #ifdef CONFIG_MEMCG
62     unsigned long memcg_data;
63 #endif
64 };

```

Listing 2-19: mm/slab.h: struct slab

There are different slab allocators available and the structure varies according to which one is used (and if SLUB is chosen, whether the CONFIG_SLUB_CPU_PARTIAL configuration options is set).

We examine each class of slab allocator in turn.

2.1.2.1 SLAB allocator

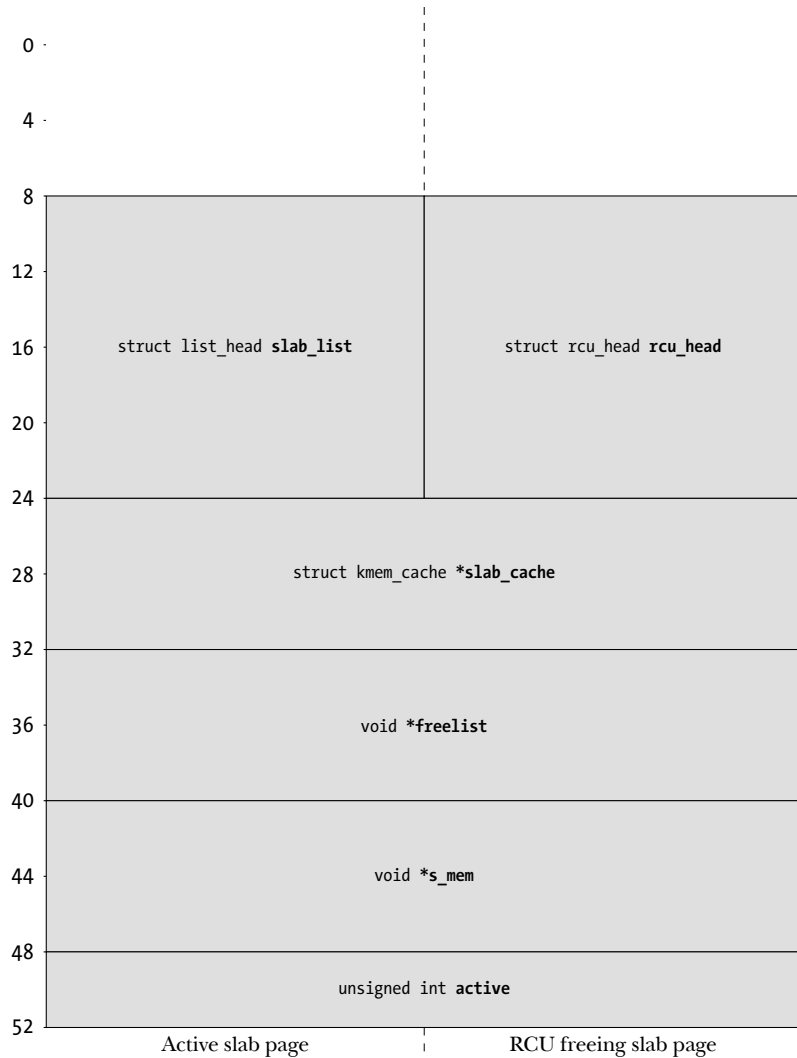


Figure 2-8: CONFIG_SLAB struct slab

Each different allocator has slightly different fields, examining fields specific to the SLAB allocator (unfortunately slab allocation is out of scope for the book):

- **freelist** – A First In-First Out (FIFO) queue of indexes of free objects indexing into **s_mem**.
- **s_mem** – A contiguous array of slab objects which **freelist** indexes into.
- **active** – This is a count of the number of slab objects in use (i.e. that have been allocated from this page).

2.1.2.2 SLUB allocator

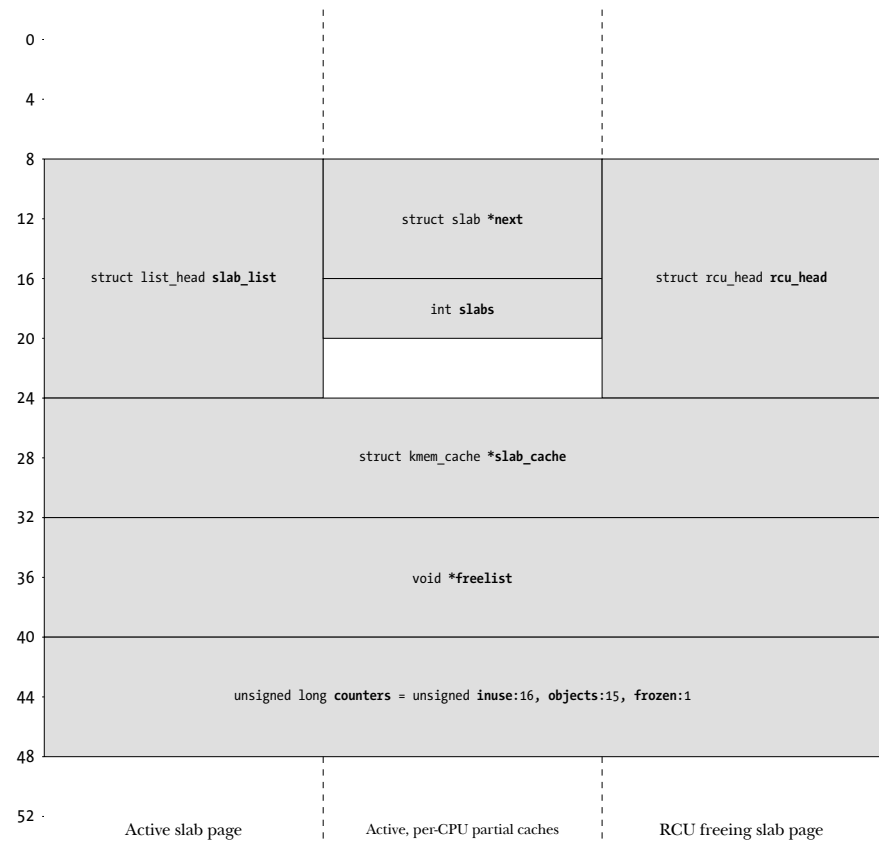


Figure 2-9: `CONFIG_SLUB struct slab`

Note that the ‘active, per-CPU partial caches’ fields here are enabled by setting the `CONFIG_SLUB_CPU_PARTIAL` configuration option.

Briefly examining each SLUB-specific field:

- `next` – (only if `CONFIG_SLUB_CPU_PARTIAL` is set) If the current slab object only represents a partial portion of the overall cache, this points at the next slab.
- `slabs` – (only if `CONFIG_SLUB_CPU_PARTIAL` is set) The number of slabs in this particular cache.
- `freelist` – Thread-safe accessible list of free objects.
- `counters` – This consists of the bitfields `inuse`, `objects` and `frozen`, with the `counters` variable provided for convenience to enable copying all of them at once.

2.1.2.3 SLOB allocator

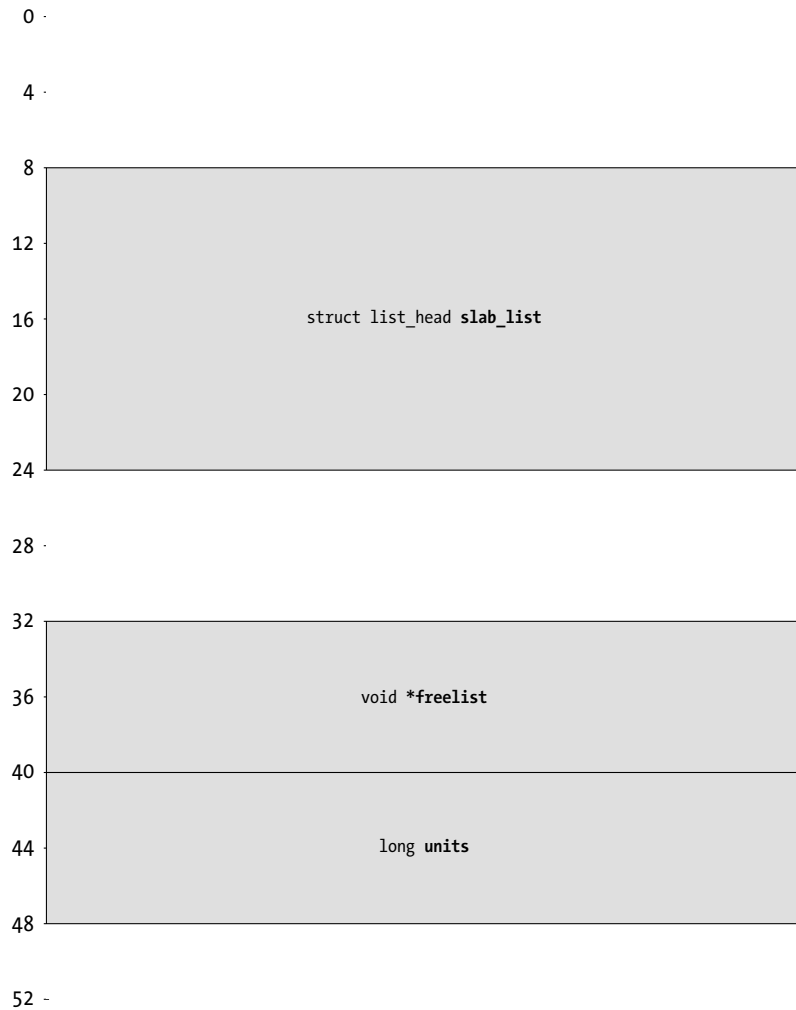


Figure 2-10: CONFIG_SLOB struct slab

Briefly examining fields unique to the SLOB:

- freelist – Pointer to the first free struct slob_block.
- units – If positive, indicates the size of this block, if negative indicates the offset of the next block.

2.1.3 Page flags

Page flags describing their attributes are kept in three separate fields in struct page:

- `struct page->flags` – This field specialises any of a large number of flags each of which are defined in `enum pageflags`. There are 28 basic flags as well as additional flags which overload these.
- `struct page->page_type` – This field defines additional fields for non-slab, non-user mappable pages, declared as preprocessor macros but still maintaining the same `PG_` prefix as flags fields.
- `struct page->mapping` – The lower two bits of this field can be set to either `PAGE_MAPPING_ANON`, `PAGE_MAPPING_MOVABLE`, or the combination of both – `PAGE_MAPPING_KSM`. These indicate whether the page is anonymous, movable or part of the Kernel Samepage Merging (KSM) functionality. These of course are prefixed with the `PAGE_MAPPING_` prefix.

The majority of page flags and related functions are defined in the `include/linux/page-flags.h` header. Flags are accessed by test/set/clear functions of the form `xxxPageyyy`. The majority are generated by macros, but a few which require special handling are manually declared. Regardless they all follow the same naming convention:

- **`Pagexxx()`** – This is a boolean function that simply tests whether the flag is set or not.
- **`SetPagexxx()`** – This sets the flag and guarantees this is done *atomically*.
- **`__SetPagexxx()`** – This sets the flag but is not *guaranteed* to be atomic.
- **`TestSetPagexxx()`** – This sets the flag and returns the old state as a boolean, guaranteed to be done atomically.
- **`ClearPagexxx()`** – This clears the flag and guarantees this is done atomically.
- **`__ClearPagexxx()`** – This clears the flag but is not *guaranteed* to be atomic.
- **`TestClearPagexxx()`** – This clears the flag and returns the old state as a boolean, guaranteed to be done atomically.

In most cases, flags specify a *policy* which performs a check on the page and potentially transforms which page the flags are accessed on:

- **`PF_POISONED_CHECK`** – Checks whether the `flags` field is set to a known ‘poison’ value meaning that the flags are invalid via `PagePoisoned()`. This is *always* enforced and checked by *all* policies below in addition to their own checks.
- **`PF_ANY`** – Can be used on *any* page, regardless of whether it is compound, head or tail.
- **`PF_HEAD`** – Transform page to a head page if its compound or do nothing if not via `compound_head()`.
- **`PF_ONLY_HEAD`** – *Always* check to ensure this isn’t a tail page.
- **`PF_NO_TAIL`** – Check to ensure this is not a tail page if setting/clearing.
- **`PF_NO_COMPOUND`** – Check to ensure it isn’t a compound page if setting/-clearing.

- `PF_SECOND` – *Always* check to ensure it is a head page, accesses flags on second page (i.e. first tail page).

Note that these checks are performed via `VM_BUG_ON_PGFLAGS()` which will only do something meaningful at runtime if `CONFIG_DEBUG_VM_PGTABLE` is configured.

We examine all available page flag helpers in Table 2-1.

Table 2-1: Page flag helpers

Test	Set	Clear	Policy	Flag
PageActive	SetActive	(__,Test)ClearPageActive	PF_HEAD	PG_active
PageAnon†	-	-	-	PAGE_MAPPING_ANON
PageAnonExclusive**	SetActiveAnonExclusive**	(__)ClearPageAnonExclusive**	PF_ANY	←PG_anon_exclusive
PageBuddy*	__SetPageBuddy*	__ClearPageBuddy*	-	PG_buddy
PageChecked	SetPageChecked	ClearPageChecked	PF_NO_COMPOUND	←PG_checked
PageCompound‡	-	ClearPageCompound‡	-	-
PageDirty	(Test)SetPageDirty	(__,Test)ClearPageDirty	PF_HEAD	PG_dirty
PageDoubleMap	(Test)SetPageDoubleMap	(Test)ClearPageDoubleMap	PF_SECOND	←PG_double_map
PageError	SetPageError	(Test)ClearPageError	PF_NO_TAIL	PG_error
PageForeign	SetPageForeign	ClearPageForeign	PF_NO_COMPOUND	←PG_foreign
PageGuard*	__SetPageGuard*	__ClearPageGuard*	-	PG_guard
PageHashHWPoisoned	(Test)SetPageHashHWPoisoned	(Test)ClearPageHashHWPoisoned	PF_SECOND	←PG_has_hwp poisoned
PageHead**	__SetPageHead	(__)ClearPageHead	PF_ANY	PG_head
PageHeadHuge‡	-	-	-	-
PageHuge‡	-	-	-	-
PageHWPoison	(Test)SetPageHWPoison	(Test)ClearPageHWPoison	PF_ANY	PG_hwp poison
PageIdle	SetPageIdle	ClearPageIdle	PF_ANY	PG_idle
PageIsolated	SetPageIsolated	ClearPageIsolated	PF_ANY	←PG_isolated
PageKsm†	-	-	-	PAGE_MAPPING_KSM
PageLocked	__SetPageLocked	__ClearPageLocked	PF_NO_TAIL	PG_locked
PageLRU	SetPageLRU	(__,Test)ClearPageLRU	PF_HEAD	PG_lru
PageMappedToDisk	SetPageMappedToDisk	ClearPageMappedToDisk	PF_NO_TAIL	PG_mappedtodisk
PageMappingFlags†	-	-	-	PAGE_MAPPING_FLAGS
PageMlocked	(Test)SetPageMlocked	(__,Test)ClearPageMlocked	PF_NO_TAIL	PG_mlocked
(__)PageMovable†	__SetPageMovable†	__ClearPageMovable†	-	PAGE_MAPPING_MOVABLE
PageOffline*	__SetPageOffline*	__ClearPageOffline*	-	PG_offline
PageOwnerPriv1	SetPageOwnerPriv1	(Test)ClearPageOwnerPriv1	PF_ANY	PG_owner_priv_1
PagePinned	(Test)SetPagePinned	(Test)ClearPagePinned	PF_NO_COMPOUND	PG_pinned
PagePoisoned‡	-	-	-	-
PagePrivate	SetPagePrivate	ClearPagePrivate	PF_ANY	PG_private
PagePrivate2	(Test)SetPagePrivate2	(Test)ClearPagePrivate2	PF_ANY	PG_private_2
PageReadahead	SetPageReadahead	(Test)ClearPageReadahead	PF_NO_COMPOUND	←PG_readahead
PageReclaim	SetPageReclaim	(Test)ClearPageReclaim	PF_NO_TAIL	PG_reclaim
PageReferenced	(__)SetPageReferenced	(__,Test)ClearPageReferenced	PF_HEAD	PG_referenced
PageReported	__SetPageReported	__ClearPageReported	PF_NO_COMPOUND	←PG_reported
PageReserved	(__)SetPageReserved	(__)ClearPageReserved	PF_NO_COMPOUND	PG_reserved
PageSavePinned	SetPageSavePinned	ClearPageSavePinned	PF_NO_COMPOUND	←PG_savepinned
PageSkipKASanPoison	SetSkipKASanPoison	ClearSkipKASanPoison	PF_HEAD	PG_skip_kasan_poison
PageSlab	__SetPageSlab	__ClearPageSlab	PF_NO_TAIL	PG_slab
PageSlobFree	SetPageSlobFree	ClearPageSlobFree	PF_NO_TAIL	←PG_slob_free
PageSwapBacked	(__)SetPageSwapBacked	(__)ClearPageSwapBacked	PF_NO_TAIL	PG_swapbacked
PageSwapCache**	SetPageSwapCache	ClearPageSwapCache	PF_NO_TAIL	←PG_swapcache
PageTable*	__SetPageTable*	__ClearPageTable*	-	PG_table
PageTail‡	-	-	-	-
PageTransCompound‡	-	-	-	-
PageTransHuge‡	-	-	-	-
PageTransTail‡	-	-	-	-
PageUncached	SetPageUncached	ClearPageUncached	PF_NO_COMPOUND	PG_uncached
PageUnevictable	SetPageUnevictable	(__,Test)ClearPageUnevictable	PF_HEAD	PG_unevictable
PageUptodate**	(__)SetPageUptodate**	ClearPageUptodate	PF_NO_TAIL	PG_uptodate
PageVmemmapSelfHosted	SetPageVmemmapSelfHosted	ClearPageVmemmapSelfHosted	PF_ANY	←PG_owner_priv_1
PageWaiters	SetPageWaiters	ClearPageWaiters	PF_ONLY_HEAD	PG_waiters
PageWorkingset	SetPageWorkingset	(Test)ClearPageWorkingset	PF_HEAD	PG_workingset
PageWriteback	(Test)SetPageWriteback	(Test)ClearPageWriteback	PF_NO_TAIL	PG_writeback
PageXenRemapped	SetPageXenRemapped	ClearPageXenRemapped	PF_NO_COMPOUND	←PG_xen_remapped
PageYoung	SetPageYoung	TestClearPageYoung	PF_ANY	PG_young

Key:

- *(no symbol)* – Denotes that the flag is stored in `struct page->flags` and the function was generated by a macro.
- *** – Denotes that the flag is stored in `struct page->page_type` and the function was generated by a macro.
- **** – Denotes that the function refers to a `struct page->flags` flag but is implemented manually rather than being generated by a macro.
- *†* – Denotes that the flag is stored in `struct page->mapping` and is implemented manually rather than being generated by a macro.
- *‡* – Denotes that the function is implemented manually but has no flag associated with it but follows the same naming convention.
- *←* – Denotes a flag that overloads an existing flag rather than having its own bit assigned.

The `PG_arch_1`, `PG_arch_2` and `PG_fscache` flags from `enum pageflags` do not have functions associated with them so they were excluded from the preceding table. Additionally, a number of these flags are dependent on config options being set to be available.

We will examine each flag individually, however before we do so let's examine the key `struct folio` data structure.

2.2 struct folio

While compound pages are a useful abstraction they cause an unfortunate ambiguity for any code working with a pointer to a `struct page` – you might always be dealing with an arbitrary tail page. This is a problem because compound page metadata is located either on the head page or one of the tail pages immediately following it.

As a result, any function that must process a `struct page` has historically passed it through `compound_head()` which takes a page as input and is guaranteed to return either the head page if it is compound or the page itself if it is not.

This is clearly suboptimal and results in functions being unintentionally weakly typed – there is no way to know whether a function parameterised by `struct page` is intended to interact with head (or non-compound) pages or tail pages without examining the code carefully.

`struct folio` addresses these shortcomings by explicitly *guaranteeing* that it is either non-compound or the head of a compound page. It is binary-compatible with `struct page` and can be used in place of it with the use of some wrapper functions which translate between `struct page` and `struct folio` – `page_folio()` and `folio_page()`.

These wrapper functions are intended to be removed the kernel over time as it moves towards using `struct folio`'s directly in more and more places.

The `struct folio` declaration also provides a wonderful description:

“A folio is a physically, virtually and logically contiguous set of bytes. It is a power-of-two in size, and it is aligned to that same power-of-two. It is at

least as large as `PAGE_SIZE`. If it is in the page cache, it is at a file offset which is a multiple of that power-of-two. It may be mapped into userspace at an address which is at an arbitrary page offset, but its kernel virtual address is aligned to its size.”

We will examine allocation of physical memory later in this chapter and virtual memory in the next, but suffice to say the two are aligned precisely as described here.

We examining the struct `folio` data structure in Listing 2-20.

```

231 /**
232  * struct folio - Represents a contiguous set of bytes.
233  * @flags: Identical to the page flags.
234  * @lru: Least Recently Used list; tracks how recently this folio was used.
235  * @mlock_count: Number of times this folio has been pinned by mlock().
236  * @mapping: The file this page belongs to, or refers to the anon_vma for
237  *   anonymous memory.
238  * @index: Offset within the file, in units of pages. For anonymous memory,
239  *   this is the index from the beginning of the mmap.
240  * @private: Filesystem per-folio data (see folio_attach_private()).
241  *   Used for swp_entry_t if folio_test_swapcache().
242  * @mapcount: Do not access this member directly. Use folio_mapcount() to
243  *   find out how many times this folio is mapped by userspace.
244  * @_refcount: Do not access this member directly. Use folio_ref_count()
245  *   to find how many references there are to this folio.
246  * @memcg_data: Memory Control Group data.
247  *
248  * A folio is a physically, virtually and logically contiguous set
249  * of bytes. It is a power-of-two in size, and it is aligned to that
250  * same power-of-two. It is at least as large as PAGE_SIZE. If it is
251  * in the page cache, it is at a file offset which is a multiple of that
252  * power-of-two. It may be mapped into userspace at an address which is
253  * at an arbitrary page offset, but its kernel virtual address is aligned
254  * to its size.
255  */
256 struct folio {
257     /* private: don't document the anon union */
258     union {
259         struct {
260             /* public: */
261             unsigned long flags;
262             union {
263                 struct list_head lru;
264                 /* private: avoid cluttering the output */
265                 struct {
266                     void *__filler;
267                     /* public: */
268                     unsigned int mlock_count;
269                     /* private: */

```

```

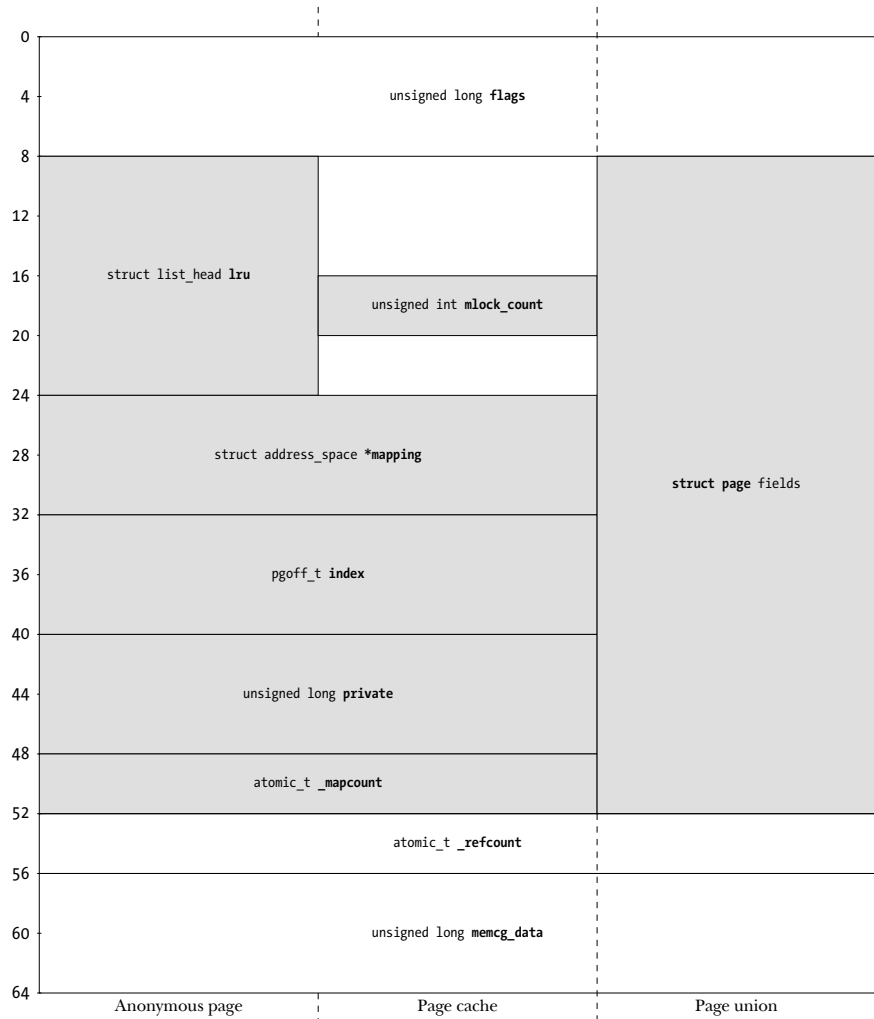
270                                     };
271      /* public: */
272                                     };
273      struct address_space *mapping;
274      pgoff_t index;
275      void *private;
276      atomic_t _mapcount;
277      atomic_t _refcount;
278 #ifdef CONFIG_MEMCG
279      unsigned long memcg_data;
280 #endif
281      /* private: the union with struct page is transitional */
282      };
283      struct page page;
284      };
285 };

```

Listing 2-20: include/linux/mm_types.h: struct folio

This mirrors the struct page structure for anonymous/page cache pages, and is unioned with the struct page type for convenience (this is used by folio_page()).

We examine this in graphical form in Figure 2-11.

Figure 2-11: *struct folio*

As the majority of use cases of page structures are anonymous pages or page cache pages, the folio data structure directly contains these fields which are binary-compatible with `struct page` (i.e. fields are placed precisely at the same offsets as the same fields in `struct page`).

Let's examine the code of helpers which convert between `struct page` and `struct folio` code.

`folio_page()` and its sister function `page_folio()` are key amongst these as they allow for the transition between `struct page` and `struct folio`.

We examine `page_folio()` in Listing 2-21.

```

262 /**
263  * page_folio - Converts from page to folio.
264  * @p: The page.
```

```

265 *
266 * Every page is part of a folio. This function cannot be called on a
267 * NULL pointer.
268 *
269 * Context: No reference, nor lock is required on @page. If the caller
270 * does not hold a reference, this call may race with a folio split, so
271 * it should re-check the folio still contains this page after gaining
272 * a reference on the folio.
273 * Return: The folio which contains this page.
274 */
275 #define page_folio(p)      (_Generic((p), \
276      const struct page *: (const struct folio *)_compound_head(p), \
277      struct page *:      (struct folio *)_compound_head(p)))

```

Listing 2-21: include/linux/page-flags.h: `page_folio()`

We examine `folio_page()` in Listing 2-22.

```

279 /**
280 * folio_page - Return a page from a folio.
281 * @folio: The folio.
282 * @n: The page number to return.
283 *
284 * @n is relative to the start of the folio. This function does not
285 * check that the page number lies within @folio; the caller is presumed
286 * to have a reference to the page.
287 */
288 #define folio_page(folio, n)    nth_page(&(folio)->page, n)

```

Listing 2-22: include/linux/page-flags.h: `folio_page()`

As the transition from page to folio can be taken from any base page `page_folio()` implicitly applies `_compound_head()` which is the untyped version of `compound_head()`, and thus this function need only be parameterised only by the page.

`folio_page()` is a little trickier, as the caller may either wish to access the head page or a specific tail page. This is addressed by providing an index parameter so the caller can indicate which page is required.

As some flags are located on tail pages, it is convenient to have a helper function which retrieves page flags located on a specific page, which is provided by `folio_flags()` as shown in Listing 2-23.

```

315 static unsigned long *folio_flags(struct folio *folio, unsigned n)
316 {
317     struct page *page = &folio->page;
318
319     VM_BUG_ON_PGFLAGS(PageTail(page), page);
320     VM_BUG_ON_PGFLAGS(n > 0 && !test_bit(PG_head, &page->flags), page);
321     return &page[n].flags;
322 }

```

Listing 2-23: `include/linux/page-flags.h`: `folio_flags()`

Which does what you'd expect, with debug assertions to check sanity (enabled only if `CONFIG_DEBUG_VM_PGTABLE` is set). The assertions ensure that the folio page isn't a tail page (this would imply something had gone *horribly* wrong) and if a tail page is being tested, that the head page has `PG_head` set asserting that this is a compound page under examination.

The struct `folio` fields have the same meaning as struct `page` and similarly `_mapcount` and `_refcount` fields should not be accessed directly but via `folio_mapcount()` and `folio_ref_count()` respectively (though the reference count will typically be handled by `folio_get()` and `folio_put()`).

Flag helpers are generated in the same way they are for struct `page`'s in the `include/linux/page-flags.h` header, only with the convention that they have a `folio_` prefix rather than a `Page_` one. Additionally policies are not enforced in the same way, rather the macros prefix the specified policy with `FOLIO_` and defines are set for each policy, e.g. `FOLIO_PF_ANY`. This selects the appropriate page to check via `folio_flags()` but skips any `CONFIG_DEBUG_VM_PGTABLE` checks. Examining equivalent folio prefixes:

- `folio_test_xxx()` (equivalent of `Pagexxx()`)
- `folio_set_xxx()` (equivalent of `SetPagexxx()`)
- `__folio_set_xxx()` (equivalent of `__SetPagexxx()`)
- `folio_test_set_xxx()` (equivalent of `TestSetPagexxx()`)
- `folio_clear_xxx()` (equivalent of `ClearPagexxx()`)
- `__folio_clear_xxx()` (equivalent of `__ClearPagexxx()`)
- `folio_test_clear_xxx()` (equivalent of `TestClearPagexxx()`)

We Examine all folio flag helpers in Table 2-2.

Table 2-2: Folio flag helpers

Test	Set	Clear	Page	Flag
folio_test_active	folio_set_active	(_)folio_(test)_clear_active	0	PG_active
folio_test_anon†	-	-	0	PAGE_MAPPING_ANON
folio_test_checked	folio_set_checked	folio_clear_checked	0	←PG_checked
folio_test_dirty	folio_(test)_set_dirty	(_)folio_(test)_clear_dirty	0	PG_dirty
folio_test_double_map	folio_(test)_set_double_map	folio_(test)_clear_double_map	1	←PG_double_map
folio_test_error	folio_set_error	folio_(test)_clear_error	0	PG_error
folio_test_foreign	folio_set_foreign	folio_clear_foreign	0	←PG_foreign
folio_test_has_hwpoisoned	folio_(test)_set_has_hwpoisoned	folio_(test)_clear_has_hwpoisoned	1	←PG_has_hwpoisoned
folio_test_head**	_ folio_set_head	(_)folio_clear_head	0	PG_head
folio_test_hugetlb‡	-	-	0	-
folio_test_hwpoisson	folio_(test)_set_hwpoisson	folio_(test)_clear_hwpoisson	0	PG_hwpoisson
folio_test_idle	folio_set_idle	folio_clear_idle	0	PG_idle
folio_test_isolated	folio_set_isolated	folio_clear_isolated	0	←PG_isolated
folio_test_ksm†	-	-	0	PAGE_MAPPING_KSM
folio_test_large**	-	-	0	PG_head
folio_test_locked	_ folio_set_locked	_ folio_clear_locked	0	PG_locked
folio_test_lru	folio_set_lru	(_)folio_(test)_clear_lru	0	PG_lru
folio_test_mappedtodisk	folio_set_mappedtodisk	folio_clear_mappedtodisk	0	PG_mappedtodisk
folio_mapping_flags†	-	-	0	PAGE_MAPPING_FLAGS
folio_test_mlocked	folio_(test)_set_mlocked	(_)folio_(test)_clear_mlocked	0	PG_mlocked
folio_test_movable	-	-	0	PAGE_MAPPING_MOVABLE
folio_test_owner_priv_1	folio_set_owner_priv_1	folio_(test)_clear_owner_priv_1	0	PG_owner_priv_1
folio_test_pinned	folio_(test)_set_pinned	folio_(test)_clear_pinned	0	PG_pinned
folio_test_private	folio_set_private	folio_clear_private	0	PG_private
folio_test_private_2	folio_(test)_set_private_2	folio_(test)_clear_private_2	0	PG_private_2
folio_test_readahead	folio_set_readahead	folio_(test)_clear_readahead	0	←PG_readahead
folio_test_reclaim	folio_set_reclaim	folio_(test)_clear_reclaim	0	PG_reclaim
folio_test_referenced	(_)folio_set_referenced	(_)folio_(test)_clear_referenced	0	PG_referenced
folio_test_reported	_ folio_set_reported	_ folio_clear_reported	0	←PG_reported
folio_test_reserved	(_)folio_set_reserved	(_)folio_clear_reserved	0	PG_reserved
folio_test_savepinned	folio_set_savepinned	folio_clear_savepinned	0	←PG_savepinned
folio_test_skip_kasan_poison	folio_set_skip_kasan_poison	folio_clear_skip_kasan_poison	0	PG_skip_kasan_poison
folio_test_slab	_ folio_set_slab	_ folio_clear_slab	0	PG_slab
folio_test_slob_free	folio_set_slob_free	folio_clear_slob_free	0	←PG_slob_free
folio_test_swapbacked	(_)folio_set_swapbacked	(_)folio_clear_swapbacked	0	PG_swapbacked
folio_test_swapcache**	folio_set_swapcache	folio_clear_swapcache	0	←PG_swapcache
folio_test_transhuge‡	-	-	0	-
folio_test_uncached	folio_set_uncached	folio_clear_uncached	0	PG_uncached
folio_test_unevictable	folio_set_unevictable	(_)folio_(test)_clear_unevictable	0	PG_unevictable
folio_test_uptodate**	(_)folio_mark_uptodate**	folio_clear_uptodate	0	PG_uptodate
folio_test_vmemmap_self_hosted	folio_set_vmemmap_self_hosted	folio_clear_vmemmap_self_hosted	0	←PG_owner_priv_1
folio_test_waiters	folio_set_waiters	folio_clear_waiters	0	PG_waiters
folio_test_workingset	folio_set_workingset	folio_(test)_clear_workingset	0	PG_workingset
folio_test_writeback	folio_(test)_set_writeback	folio_(test)_clear_writeback	0	PG_writeback
folio_test_xen_remapped	folio_set_xen_remapped	folio_clear_xen_remapped	0	←PG_xen_remapped
folio_test_young	folio_set_young	folio_test_clear_young	0	PG_young

As before the key is as follows:

- *(no symbol)* – Denotes that the flag is stored in struct folio->flags and the function was generated by a macro.
- ** – Denotes that the function refers to a struct folio->flags flag but is implemented manually rather than being generated by a macro.
- † – Denotes that the flag is stored in struct folio->mapping and is implemented manually rather than being generated by a macro.
- ‡ – Denotes that the function is implemented manually but has no flag associated with it but follows the same naming convention.
- ← – Denotes a flag that overloads an existing flag rather than having its own bit assigned.

Notes:

- As no policies are actually applied for the folio flag helpers, the policy column is replaced with a struct page offset indicating which struct page actually contains the flags field referenced by the helper.
- No folio flag helpers refer to struct page->page_type.
- folio_mapping_flags() (the equivalent of PageMappingFlags()) breaks the convention of folio flag tests being prefixed with folio_test_.

2.3 Physical memory model

While we have discussed struct page (and equivalently struct folio objects), we have not explored where exactly these are kept and how they are maintained. How this is done is termed the *physical memory model* of a system. It describes how struct page objects are laid out and, as this layout is designed to represent physical memory, it is also a representation of how memory is laid out in the system.

Converting from physical addresses (very often conveniently represented as Page Frame Numbers (PFNs), physical addresses shifted by page size to become an index into physical memory as if it were one giant array) to struct page or struct folio objects, or vice-versa, is a key task within the kernel and performed whenever we need to access page metadata from an address or vice-versa.

Doing so is achieved via the page_to_pfn() and pfn_to_page() macros whose ultimate implementations differ depending on memory model.

There are two different memory models available – a ‘flat’ memory model specified by CONFIG_FLATMEM and a ‘sparse’ memory model specified by CONFIG_SPARSEMEM.

The flat memory model declares a global variable mem_map which is simply an array of struct page entries. There are redundant entries for memory holes and memory hot-plugging is unavailable. This is not practical in a modern system and therefore we will not examine it any further.

The sparse memory model maintains *sections* which represent disparate, ‘sparse’ blocks of contiguous physical memory. The sparse memory model can be further modified by additional configuration options:

- CONFIG_SPARSEMEM_VMEMMAP – Provides a virtual mapping global vmemmap variable which implements a virtual equivalent to the mem_map variable for fast implementations of page_to_pfn() and pfn_to_page(). Virtual memory is discussed in Chapter 3—briefly, it allows for physical memory to be mapped to arbitrary addresses.
- CONFIG_SPARSEMEM_EXTREME – Permits for dynamic allocation of *sections* (described) below, allowing for complete flexibility in memory layout.

A modern system will use a sparse memory model and define both of these configuration options so we examine only this configuration.

2.3.1 Sections

In a sparse memory model contiguous blocks of physical memory are represented by *sections*. Each section is represented by a struct `mem_section` object in a dynamically allocated (since we assume `CONFIG_SPARSEMEM_EXTREME` is set) array struct `mem_section **mem_section`.

The `mem_section` array is two dimensional, with the first index being termed a section *root* index and the second an index into the root. Each root consists of a base page containing an array of a `SECTIONS_PER_ROOT` count of struct `mem_section` objects. More on this below.

Each section spans `SECTION_SIZE_BITS` bits (therefore $2^{\text{SECTION\_SIZE\_BITS}}$ bytes). By default this is 27 bits or 128 MiB (this is the case for `SECTION_SIZE_BITS` for x86-64 and `SECTION_SIZE_BITS` for arm64 with page size below 64 KiB).

Examining struct `mem_section` as shown in Listing 2-24 (since we assume a sparse memory model we elide the `CONFIG_PAGE_EXTENSION` portion of this struct).

```

1424 struct mem_section {
1425     /*
1426      * This is, logically, a pointer to an array of struct
1427      * pages. However, it is stored with some other magic.
1428      * (see sparse.c::sparse_init_one_section())
1429      *
1430      * Additionally during early boot we encode node id of
1431      * the location of the section here to guide allocation.
1432      * (see sparse.c::memory_present())
1433      *
1434      * Making it a UL at least makes someone do a cast
1435      * before using it wrong.
1436      */
1437     unsigned long section_mem_map;
1438
1439     struct mem_section_usage *usage;
1440     :
1441     /*
1442      * WARNING: mem_section must be a power-of-2 in size for the
1443      * calculation and use of SECTION_ROOT_MASK to make sense.
1444      */
1445 };
1446 
```

Listing 2-24: `include/linux/mmzone.h: struct mem_section`

The `section_mem_map` field contains a pointer to the struct page array and takes advantage of the fact that it will be aligned to at least 6 bits to store some `SECTION_*` flags to store various attributes of the section.

The page array can be accessed via `__section_mem_map_addr()`. However, as we are only considering systems which have `CONFIG_SPARSEMEM_VMEMMAP` set, we

generally have no need to access pages this way as we can simply offset into the architecture-specific `vmemmap` array to access these pages.

The `usage` field is an `struct mem_section_usage` object which contains a bitmap indicating whether a PFN is a *valid* area of memory or not (i.e. whether we can actually use it) and flags describing page blocks contained within the section.

NOTE

A bitmap is a series of bytes which treat individual bits within the bytes as booleans at offset equal to the bit offset.

Section flags stored in `section_mem_map` are as follows:

- `SECTION_MARKED_PRESENT` – Indicates that a section of memory is actually physically present in the system. This is tested by `present_section()`, `present_section_nr()` and `pfm_in_present_section()`.
- `SECTION_HAS_MEM_MAP` – Determines whether the `section_mem_map` field actually refers to an array of `struct page` objects or not, i.e. whether the section itself is valid. This is checked by `valid_section()` and `valid_section_nr()` and more importantly `pfm_valid()`.
- `SECTION_IS_ONLINE` – Determines whether hotpluggable memory is actually online and checked by `online_section()` and `online_section_nr()`.
- `SECTION_IS_EARLY` – Determines whether the section is undergoing early memory initialisation. Checked by `early_section()`.
- `SECTION_TAINT_ZONE_DEVICE` – Used in conjunction with `SECTION_IS_ONLINE` to indicate that a section is part of a `ZONE_DEVICE`. Checked by `online_device_section()`. This topic is out of scope for the book.

We examine `struct mem_section_usage` in Listing 2-25.

```

1412 struct mem_section_usage {
1413     #ifdef CONFIG_SPARSEMEM_VMEMMAP
1414         DECLARE_BITMAP(subsection_map, SUBSECTIONS_PER_SECTION);
1415     #endif
1416     /* See declaration of similar field in struct zone */
1417     unsigned long pageblock_flags[0];
1418 };

```

Listing 2-25: `include/linux/mmzone.h`: `struct mem_section_usage`

The `subsection_map` field is a bitmap indicating whether a *subsection* is *valid*, i.e. whether we are actually able to use it. A PFN can be converted to a subsection via `subsection_map_index()` and checked for validity via `pfm_section_valid()`.

Each section is divided into subsections of `SUBSECTION_SIZE` bytes size (`SUBSECTION_SHIFT` bits), hard coded to 2 MiB. There are `SUBSECTIONS_PER_SECTION` subsections in each section, assuming 27 bits section size this results in 64 subsections per section.

Note that we track only whether a subsection is valid, not a page, so the granularity of tracking whether pages are actually valid or not is 2 MiB. In

the early stages of memory initialisation when memory ranges are identified, the entire subsection containing valid pages are marked valid.

2.3.2 PFN validity

Determining if a Page Frame Number (PFN) is valid therefore reduces to a process of determining a PFN's section/subsection then checking this subsection_map bitmap. This is performed by `pfn_valid()` which checks validity of the section via `valid_section()` and the validity of a PFN's subsection via `pfn_section_valid()` as shown in Listing 2-26.

```

1602 static inline int pfn_section_valid(struct mem_section *ms, unsigned long pfn)
1603 {
1604     int idx = subsection_map_index(pfn);
1605
1606     return test_bit(idx, ms->usage->subsection_map);
1607 }
```

Listing 2-26: include/linux/mmzone.h: pfn_section_valid()

We determine the subsection index *within* the section via `subsection_map_index()`, which we use to index into the `usage->subsection_map` field of the section as shown in Listing 2-27.

```

1596 static inline int subsection_map_index(unsigned long pfn)
1597 {
1598     return (pfn & ~(PAGE_SECTION_MASK)) / PAGE_SIZE;
1599 }
```

Listing 2-27: include/linux/mmzone.h: subsection_map_index()

Which simply finds the subsection offset by taking the lower bits of the PFN offsetting into the section and dividing by the number of pages per subsection.

2.3.3 Converting between PFN and section

Working with section necessitates converting between PFNs and `struct mem_section` objects. This is achieved via a number of functions. Firstly, `pfn_to_section_nr()`, which simply determines the *section number*, i.e. the index of the section (imagining physical memory being a simple array) as shown in Listing 2-28.

```

1384 static inline unsigned long pfn_to_section_nr(unsigned long pfn)
1385 {
1386     return pfn >> PFN_SECTION_SHIFT;
1387 }
```

Listing 2-28: include/linux/mmzone.h: pfn_to_section_nr()

A more useful function is `__pfn_to_section()` which converts a PFN directly to a `struct mem_section` object as shown in Listing 2-29.

```

1589 static inline struct mem_section *__pfn_to_section(unsigned long pfn)
1590 {
1591     return __nr_to_section(pfn_to_section_nr(pfn));
1592 }

```

Listing 2-29: include/linux/mmzone.h: __pfn_to_section()

This invokes `pfn_to_section_nr()` to obtain the PFN's section number, then `__nr_to_section()` to convert this to a struct `mem_section` as shown in Listing 2-30.

```

1475 static inline struct mem_section *__nr_to_section(unsigned long nr)
1476 {
1477     unsigned long root = SECTION_NR_TO_ROOT(nr);
1478
1479     if (unlikely(root >= NR_SECTION_ROOTS))
1480         return NULL;
1481
1482 #ifdef CONFIG_SPARSEMEM_EXTREME
1483     if (!mem_section || !mem_section[root])
1484         return NULL;
1485 #endif
1486     return &mem_section[root][nr & SECTION_ROOT_MASK];
1487 }

```

Listing 2-30: include/linux/mmzone.h: __nr_to_section()

This determines which root the section belongs to via `SECTION_NR_TO_ROOT()` (which simply divides the section number by `SECTIONS_PER_ROOT`) and looks up the section in the `mem_section[][]` array by placing this in the first index and determining the second array index via `SECTION_ROOT_MASK` which masks the lower bits.

2.3.4 Page block flags

The `pageblock_flags[]` array in struct `mem_section_usage`, also termed the section's *use map*, contains flags which set attributes for page blocks contained within the section.

It is of `SECTION_BLOCKFLAGS_BITS` bits (and thus `usemap_size(). bytes`) in size as shown in Listing 2-31

```

1377 #define SECTION_BLOCKFLAGS_BITS \
1378     ((1UL << (PFN_SECTION_SHIFT - pageblock_order)) * NR_PAGEBLOCK_BITS)

```

Listing 2-31: include/linux/mmzone.h: SECTION_BLOCKFLAGS_BITS

This essentially multiplies the number of page blocks each section can contain by `NR_PAGEBLOCK_BITS`. This latter value is the last entry in `enum_pageblock_bits` denoting its size as shown in Listing 2-32.

```

18 enum pageblock_bits {

```

```

19     PB_migrate,
20     PB_migrate_end = PB_migrate + PB_migratetype_bits - 1,
21     /* 3 bits required for migrate types */
22     PB_migrate_skip, /* If set the block is skipped by compaction */
23
24     /*
25      * Assume the bits will always align on a word. If this assumption
26      * changes then get/set pageblock needs updating.
27      */
28     NR_PAGEBLOCK_BITS
29 };

```

Listing 2-32: include/linux/pageblock-flags.h: enum pageblock_bits

This ‘reserves’ sufficient space for each of the migrate types declared in enum migratetype as well as adding space for an additional compaction flag.

To obtain this ‘use map’ the section_to_usemap() function can be used to simply obtains the section’s ->usage->pageblock_flags field. This is, in turn, used by get_pageblock_bitmap().

This value is described as a bitmap but returns NR_PAGEBLOCK_BITS at a time rather than single bits.

Accessing these values is performed by get_pfnblock_flags_mask() which in turn invokes __get_pfnblock_flags_mask() as shown in Listing 2-33.

```

550 static __always_inline
551 unsigned long __get_pfnblock_flags_mask(const struct page *page,
552                                         unsigned long pfn,
553                                         unsigned long mask)
554 {
555     unsigned long *bitmap;
556     unsigned long bitidx, word_bitidx;
557     unsigned long word;
558
559     bitmap = get_pageblock_bitmap(page, pfn);
560     bitidx = pfn_to_bitidx(page, pfn);
561     word_bitidx = bitidx / BITS_PER_LONG;
562     bitidx &= (BITS_PER_LONG-1);
563     /*
564      * This races, without locks, with set_pfnblock_flags_mask(). Ensure
565      * a consistent read of the memory array, so that results, even though
566      * racy, are not corrupted.
567      */
568     word = READ_ONCE(bitmap[word_bitidx]);
569     return (word >> bitidx) & mask;
570 }

```

Listing 2-33: mm/page_alloc.c: __get_pfnblock_flags_mask()

This simply looks up the index within the bitmap determined by `pfn_to_bitidx()` and applies the supplied mask. Examining `pfn_to_bitidx()` as shown in Listing 2-34.

```

540 static inline int pfn_to_bitidx(const struct page *page, unsigned long pfn)
541 {
542     #ifdef CONFIG_SPARSEMEM
543         pfn &= (PAGES_PER_SECTION-1);
544     #else
545         pfn = pfn - round_down(page_zone(page)->zone_start_pfn,
546                                pageblock_nr_pages);
547     #endif /* CONFIG_SPARSEMEM */
548     return (pfn >> pageblock_order) * NR_PAGEBLOCK_BITS;
549 }
```

Listing 2-34: `mm/page_alloc.c: pfn_to_bitidx()`

This takes the lower bits of the PFN by masking with the lower bits of `PAGES_PER_SECTION`, shifts by pageblock order (note this order is expressed in base pages so we neatly handle PFN being expressed in page units) and multiplies by `NR_PAGEBLOCK_BITS` which is the size of data we are retrieving.

NOTE

As this is a power-of-2 the classic bitwise trick of simply subtracting one for all bits below it to be set to use as a mask is applied. E.g. `0b1000 - 1` is `0b0111`.

These values are set by `set_pfnblock_flags_mask()` which performs a similar operation only taking care to handle data races.

An important use of the ‘use map’ is to obtain a page block’s migrate type via `get_pageblock_migratetype()` which simply invokes `get_pfnblock_flags_mask()` with `MIGRATETYPE_MASK`.

The compaction skip flag is also retrieved from `get_pfnblock_flags_mask()` in `get_pageblock_skip()`.

2.3.5 Looking up struct page/struct folio objects

As `CONFIG_SPARSEMEM_VMEMMAP` is set, struct page objects (and thus struct folio objects) are looked up via the `vmemmap` global (x86-64’s `vmemmap` is simply an offset into the kernel’s established virtual memory map, as is arm64’s `vmemmap`).

Converting from PFN to page is achieved via `pfn_to_page()` which is a macro that invokes `__pfn_to_page()` as shown in Listing 2-35.

```

25 #define __pfn_to_page(pfn)      (vmemmap + (pfn))
```

Listing 2-35: `include/asm-generic/memory_model.h: __pfn_to_page()`

Equally converting the other way, as shown in Listing 2-36.

```

26 #define __page_to_pfn(page)     (unsigned long)((page) - vmemmap)
```

Listing 2-36: `include/asm-generic/memory_model.h: __page_to_pfn()`

These are incredibly simple as the virtual memory map has already been set up and designed to simply be offset into, and of course this is a very efficient means of looking up pages.

We will speak more about virtual memory in the next chapter but broadly it is a means by which disparate pages of physical memory can be mapped to arbitrary ‘virtual’ addresses.

2.4 Nodes and Zones

The physical layout of a computer may result in different regions of memory having different access times depending from which core they are accessed. For example consider a dual-socket machine – these typically have banks of RAM attached to each CPU socket. Accesses to memory attached to a given socket will be fast, however accesses to memory belonging to another socket will have to pass over some kind of interconnect, and will therefore be slow relative to ‘local’ memory.

In order to take this into account, the kernel subdivides cores and memory into NUMA nodes. Memory attached to a NUMA node has uniform memory access time for cores which also reside within that node.

The architecture of a real system can be complicated and interconnects between different nodes can have varying ‘distances’ between them. The architecture of these can be examined on a system by invoking `numactl --hardware`.

Each node represents a physically contiguous range of memory, though within that range there may be ‘holes’ (either memory that is not present at all or ranges that are reserved for devices or system).

A *zone* describes a physically contiguous range of memory which is categorised by that range having a distinct attribute – for example `ZONE_DMA32` describes the zone over which 32-bit devices can perform Direct Memory Access (DMA) operations.

A given physical memory address belongs to one and only one zone and one and only one node. However, zones can and typically do overlap nodes. Each node is represented by the `pg_data_t` type. Each of these contain an array of `struct zone` objects which represent the portion of the zone that is present in that node.

Each zone has *watermarks* specified which determine *reclaim* (the effort to free up, i.e. ‘reclaim’ memory) behaviour, as shown in Figure 2-12.

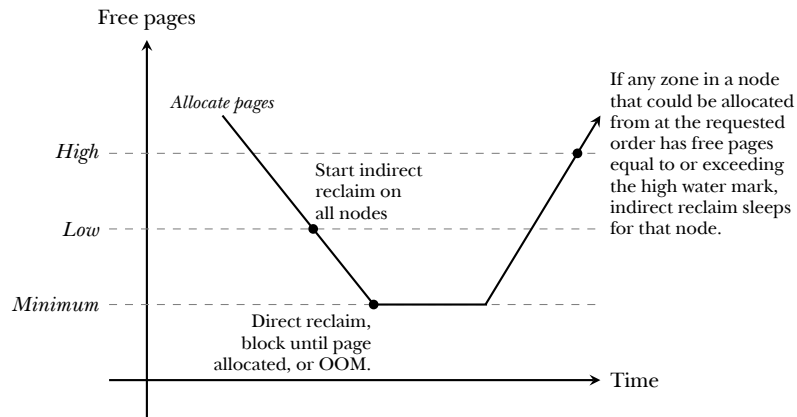


Figure 2-12: Zone Watermarks

As seen above, when free pages in a zone pass these watermarks, then the following happens:

- **WMARK_MIN** – When free base pages in the zone are at or below the sum of **WMARK_MIN** and the lowmem reserve (see below) then the allocation blocks while *direct reclaim* (see section 11.3 for more details) is performed – the kernel will try to free up memory sufficient to satisfy the allocation. If it cannot, an Out Of Memory (OOM) condition may arise (see the OOM chapter on this for more details).
- **WMARK_LOW** – When free base pages in the zone are at or below **WMARK_LOW** the kswapd kernel process is woken up to perform *indirect reclaim* (see section 11.4 for more details) to attempt to free up pages in the background for all nodes in the allocation nodemask via `wake_all_kswaps()`.
- **WMARK_HIGH** – When free base pages in any zone that could be allocated from by the failing allocation in the node at the requested order, which had dipped below **WMARK_LOW** is at or above **WMARK_HIGH**, the kswapd process for that node is put to sleep.
- **WMARK_PROMO** – A watermark used in NUMA balancing memory tiering mode. This has a very specific purpose, see the `sysctl/kernel` documentation for more details.

Examining a simplified version of `pg_data_t` stripping out all but the core fields as shown in Listing 2-37.

```

905 typedef struct pglist_data {
906     /*
907      * node_zones contains just the zones for THIS node. Not all of the
908      * zones may be populated, but it is the full list. It is referenced
909      by
910      * this node's node_zonelists as well as other node's node_zonelists.
911      */
912     struct zone node_zones[MAX_NR_ZONES];

```

```

912
913     /*
914     * node_zonelists contains references to all zones in all nodes.
915     * Generally the first zones will be references to this node's
916     * node_zones.
917     */
918     struct zonelist node_zonelists[MAX_ZONELISTS];
919
920     int nr_zones; /* number of populated zones in this node */
921     :
922
923     unsigned long node_start_pfn;
924     unsigned long node_present_pages; /* total number of physical pages */
925     unsigned long node_spanned_pages; /* total size of physical page
926                                     range, including holes */
927
928     int node_id;
929     :
930
931     /*
932     * This is a per-node reserve of pages that are not available
933     * to userspace allocations.
934     */
935     unsigned long          totalreserve_pages;
936     :
937
938     unsigned long          flags;
939     :
1015 } pg_data_t;

```

Listing 2-37: include/linux/mmzone.h: Simplified pg_data_t

Each node's `pg_data_t` object contains all of the struct zone objects associated with that node within it. It is often referred to as a `pgdat`.

Examining each of these fields:

- `node_zones` – An array of struct zone objects each of which describing the portion of that zone that overlap this node. `MAX_NR_ZONES` is generated at build time based on kernel configuration and indicates the maximum number of zones the system supports.
- `node_zonelists` – An array of struct zonelist objects describing zones allocations should try to allocate from, in order. Examining it as shown in Listing 2-38.

```

878 struct zonelist {
879     struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
880 };

```

Listing 2-38: include/linux/mmzone.h: struct zonelist

There are `MAX_ZONELISTS` of these struct zonelist, which on a NUMA system are `ZONELIST_FALLBACK` (index 0) and `ZONELIST_NOFALLBACK` (index 1). The default zonelist used is `ZONELIST_FALLBACK` which permits fallback to nodes other than the one requested one, The `ZONELIST_NOFALLBACK` option

is to support the `__GFP_THISNODE` allocation flag, and the zonelist is chosen by `gfp_zonelist()`.

Examining struct `zoneref` as shown in Listing 2-39.

```

859 struct zoneref {
860     struct zone *zone;      /* Pointer to actual zone */
861     int zone_idx;           /* zone_idx(zoneref->zone) */
862 };

```

Listing 2-39: `include/linux/mmzone.h`: struct `zoneref`

This references struct `zone` with a zone index stored for convenience.

Accessors are provided to access these fields – `zonelist_zone()` to access the struct `zone` object, `zonelist_zone_idx()` to access the index and for convenience `zonelist_node_idx()` provides the index of the node associated with this zone via `zone_to_nid()`.

There are zone entries here even for zones that are not populated (a zone is populated if it contains at least one *present* page, i.e. a page that is within the node/zone memory range and actually allocatable).

All zones on online nodes can be iterated through via `for_each_zone()`, all populated zones on online nodes can be iterated through via `for_each_populated_zone()`.

- `nr_zones` – Count of the number of *populated* zones in this node (zones which have at least one present page). A zone is determined to be populated via `populated_zone()`.
- `node_start_pfn` – The Page Frame Number (PFN) of the first page spanned by this node. Note that this might not be a *present* page. The last PFN is not stored but rather deduced via `node_end_pfn()` which in turn invokes `pgdat_end_pfn()` which obtains an *exclusive* bound by simply adding `node_spanned_pages` to `node_start_pfn`.
- `node_present_pages` – Determines the number of *present* pages, i.e. pages that are allocatable within the node (not part of a memory hole, online and available). A memory hole is a range of physical memory that is unavailable for ordinary use due to either not physically existing, or being mapped to devices or reserved for systems use. Accessed via `node_present_pages()`.
- `node_spanned_pages` – The range of physical memory *spanned* by this node, including PFNs that are not allocatable. This is accessed via `node_spanned_pages()`.
- `node_id` – Also known as the node's `nid`, an index identifying the node.
- `totalreserve_pages` – Counts the number of pages reserved per-node in order to service high priority allocations and prevent a total exhaustion of RAM. This is determined by the `vm.lowmem_reserve_ratio` and `vm.min_free_kbytes` sysctl tunables and determined by `calculate_totalreserve_pages()`.

- **flags** – Specifies flags that indicate various characteristics of the node used to modify reclaim behaviour. The bits used for the flags are declared in `enum pgdat_flags`:
 - **PGDAT_DIRTY** – Indicates that there are an excess of dirty pages on the node which are at the end of the LRU and thus not otherwise queued for reclaim soon. This is set in `shrink_node()`.
 - **PGDAT_WRITEBACK** – Indicates that there are an excess of pages in write-back (i.e. writing back their contest to disk) for the purposes of reclaim. This is also set in `shrink_node()`.
 - **PGDAT_RECLAIM_LOCKED** – Indicates that reclaim is taking place on the node and used to prevent concurrent reclaim attempts. Set and cleared in `node_reclaim()`.

Accessing nodes is performed via the `NODE_DATA()` helper function by node index (or `nid`). The ID of a node can be determined via `page_to_nid()` (it is encoded into the `struct page` flags field).

Zones are described by `enum zone_type`. Available zones are determined by kernel configuration but for a modern 64-bit system typically consist of:

- **ZONE_DMA** – On the x86-64 architecture this spans the range a museum-worthy legacy device is able to perform Direct Memory Access (DMA) upon (DMA is a means by which hardware can transfer data to/from memory independently of the CPU). For a typical x86-system this spans the first 24 bits of memory, i.e. 16 MiB. However, for other architectures, notably arm64, this may span *all* of the memory aside from those put into optional zones. This is the case when devices can access memory anywhere and there is no need to separate.
- **ZONE_DMA32** – This spans the range a 32-bit DMA device is able to perform DMA upon, except in the case of all physical memory (aside from those put in optional zones) residing in **ZONE_DMA**.
- **ZONE_NORMAL** – This comprises the remainder of memory not located in other zones.
- **ZONE_MOVABLE** – This is a special zone that has to be enabled via the `kernelcore` and/or `movablecore` command line parameters. The `kernelcore` designated memory, will exist in zones other than **ZONE_MOVABLE**, the remainder will, if it would have otherwise been placed in **ZONE_NORMAL**, be contained within **ZONE_MOVABLE**. The purpose of doing this is to maintain allocations that can be moved around by the kernel in a physically contiguous region rather than mixing movable and unmovable memory together. This helps reduce fragmentation and improves the availability of higher order pages.

Examining a simplified version of `struct zone` stripping out all but the core fields as shown in Listing 2-40.

```

515 struct zone {
516     /* Read-mostly fields */
517
```

```

518      /* zone watermarks, access with *_wmark_pages(zone) macros */
519      unsigned long _watermark[NR_WMARK];
520      unsigned long watermark_boost;
521
522      unsigned long nr_reserved_highatomic;
523
524      /*
525       * We don't know if the memory that we're going to allocate will be
526       * freeable or/and it will be released eventually, so to avoid totally
527       * wasting several GB of ram we must reserve some of the lower zone
528       * memory (otherwise we risk to run OOM on the lower zones despite
529       * there being tons of freeable ram on the higher zones). This array
530       * is
531       * recalculated at runtime if the sysctl_lowmem_reserve_ratio sysctl
532       * changes.
533       */
534      long lowmem_reserve[MAX_NR_ZONES];
535      :
536      int node;
537      :
538      struct pglist_data      *zone_pgdat;
539      struct per_cpu_pages    __percpu *per_cpu_pageset;
540      struct per_cpu_zonestat __percpu *per_cpu_zonestats;
541      /*
542       * the high and batch values are copied to individual pagesets for
543       * faster access
544       */
545      int pageset_high;
546      int pageset_batch;
547      :
548      /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
549      unsigned long          zone_start_pfn;
550      :
551      atomic_long_t          managed_pages;
552      unsigned long          spanned_pages;
553      unsigned long          present_pages;
554      :
555      const char             *name;
556      :
557      /* free areas of different sizes */
558      struct free_area        free_area[MAX_ORDER];
559
560      /* zone flags, see below */
561      unsigned long          flags;
562
563      /* Primarily protects free_area */
564      spinlock_t             lock;

```

```

:
677     bool                contiguous;
:
680     /* Zone statistics */
681     atomic_long_t        vm_stat[NR_VM_ZONE_STAT_ITEMS];
682     atomic_long_t        vm_numa_event[NR_VM_NUMA_EVENT_ITEMS];
683 } ____cacheline_internodealigned_in_smp;

```

Listing 2-40: include/linux/mmzone.h: Simplified struct zone

Looking at each field:

- `_watermark` – An array of `NR_WMARK` watermark values expressed in base pages for each of the watermarks – `WMARK_MIN`, `WMARK_LOW`, `WMARK_HIGH` and `WMARK_PROMO`. These should not be accessed directly but rather via `wmark_pages()`, `min_wmark_pages()`, `low_wmark_pages()` or `high_wmark_pages()`.
- `watermark_boost` – A value which is added to watermarks in order to reclaim early. set in `boost_watermark()` with the magnitude being scaled by the `vm.watermark_boost_factor` tuneable. This is set when a pageblock mixes migrate types of different types and is a heuristic designed to reduce fragmentation, and therefore `boost_watermark()` is only set from `steal_suitable_fallback()`.
- `nr_reserved_highatomic` – The number of pages reserved for exclusive high-order atomic allocations.
- `lowmem_reserve` – Specifies the number of pages which should be held in reserve when pages are allocated from zones other than the one specified by the allocation. This is to prevent exhaustion of pages by allocations which do not *have* to allocate pages from each zone. Discussed in more detail below. Set in `setup_per_zone_lowmem_reserve()` and controlled by the `vm.lowmem_reserve_ratio` tuneable.
- `node` – Indicates the node ID (or *nid*) of the zone to which this struct `zone` belongs. Accessed by `zone_to_nid()` and set by `zone_set_nid()`.
- `zone_pgdat` – A pointer to the `pg_data_t` object representing the node this zone belongs to.
- `per_cpu_pageset` – Pointer to a per-CPU array of struct `per_cpu_pages` which describes Per-CPU-Pages (or page sets) which act as a per-CPU cache of free pages to avoid locks on page allocation/freeing. See section 2.7.3 for more details. Per-CPU arrays (with *sparse* tag `__percpu`) have an entry for each core and when accessed are offset according to the current core. Care is taken to avoid false sharing between cache lines.
- `per_cpu_zonestats` – Per-CPU zone statistics contained in struct `per_cpu_zonestat` objects.
- `pageset_high` – The value to set each struct `per_cpu_pages->high` specifying the maximum number of base pages each page set should contain. Set by `pageset_update()` (which is set by `__zone_set_pageset_high_and_batch()`).

- `pageset_batch` – The value to set each struct `per_cpu_pages->batch` specifying the batch size of pages to retrieve when refilling PCP lists. Set by `pageset_update()` (which is set by `__zone_set_pageset_high_and_batch()`).
- `zone_start_pfn` – The Page Frame Number (PFN) of the first physical base page spanned by this zone.
- `spanned_pages` – The number of base pages over which this zone object spans, e.g. `zone_end_pfn - zone_start_pfn`, this includes all ‘holes’ and reserved pages.
- `present_pages` – The number of physically available base pages – equal to the number of spanned pages minus any unavailable pages (i.e. ‘holes’).
- `managed_pages` – The number of page available to the buddy allocator (this is effectively present base pages minus reserved pages).
- `name` – A pointer to a string describing the zone (e.g. “ZONE_NORMAL”), taken from the `zone_names[]` static array.
- `free_area` – Array of struct `free_area` free lists indexed by order (each struct `free_area` contains per-migrate type lists of struct `page`).
- `flags` – Flags which impact zone behaviour as defined by enum `zone_flags`.
- `lock` – Spin lock for the zone, used for access to free lists.
- `contiguous` – A boolean value indicating that there are no holes in this zone, which is used as an optimisation in `pageblock_pfn_to_page()`.
- `vm_stat` – Statistics for this zone.
- `vm_numa_event` – NUMA-specific statistics for this zone.

Whether a PFN is spanned by a zone can be determined via `zone_spans_pfn()` which simply checks whether a PFN is less than or equal to `struct zone->zone_start_pfn` and less than `zone_end_pfn()`.

2.4.1 Low memory reserve

The low memory reserve mechanism is designed to prevent exhaustion of pages in zones with fewer resources by allocations which could be serviced by zones with more. This only comes into play if an allocation is attempted in a particular zone but is unable to be fulfilled by that zone. If so, the allocator attempts to allocate from the zone *below* it.

As discussed previously, a modern 64-bit system will consist of the following zones (though `ZONE_MOVABLE` and `ZONE_DEVICE` are optional):

1. `ZONE_DMA`
2. `ZONE_DMA32`
3. `ZONE_NORMAL`
4. `ZONE_MOVABLE`
5. `ZONE_DEVICE`

This is also the order in which the zones are declared in `enum zone_type`. This means that `ZONE_DMA32` sits *below* `ZONE_NORMAL` and `ZONE_DEVICE` sits *above* `ZONE_MOVABLE` and so on.

Therefore, an allocation designated to be taken from `ZONE_MOVABLE` could, if no memory were available there, be taken from `ZONE_NORMAL`, `ZONE_DMA32` or `ZONE_DMA`.

In order to decide how to apply the low memory reserve to these allocations, the tunable `vm.lowmem_reserve_ratio` is used. This specifies a range of integers which align with available zones. For example on my system I observe:

```
[~]$ sysctl vm.lowmem_reserve_ratio
vm.lowmem_reserve_ratio = 256    128    32    0    0
```

Each integer relates to each zone in ascending order, so ratios are assigned as shown in Table 2-3.

Table 2-3: Low memory zone ratios

ZONE_DMA	ZONE_DMA32	ZONE_NORMAL	ZONE_MOVABLE	ZONE_DEVICE
1/256	1/128	1/32	0	0

Note that non-zero ratios are inverted to determine the additional reserve multiplier for each zone. Zero indicates no adjustment.

This is a rather confusing subject so it's important to be clear – the ratio we use is specified by which zone we *actually* end up allocating in, and is applied to the sum of the managed pages of the ones we *could* have allocated in but did not. For example if we requested `ZONE_NORMAL` but ended up allocating in `ZONE_DMA` we will establish a reserve of 1/256 of the sum of managed pages in `ZONE_NORMAL` and `ZONE_DMA32`.

The heuristic is implemented this way in order for the penalty to be proportional to the available pages that *could* have been allocated from but were not.

We visualise this in table 2-4.

Table 2-4: Low memory zone penalties

Actual zone	Requested zone				
	ZONE_DMA	ZONE_DMA32	ZONE_NORMAL	ZONE_MOVABLE	ZONE_DEVICE
ZONE_DMA	•	1/256	1/256	1/256	1/256
ZONE_DMA32		•	1/128	1/128	1/128
ZONE_NORMAL			•	1/32	1/32
ZONE_MOVABLE				•	0
ZONE_DEVICE					•

The low memory reserve values can be observed in `/proc/zoneinfo` listed under 'protection', providing low memory base page reserve counts for each requested zone sequentially, for example on my machine:

```

Node 0, zone      DMA
        managed 3977
        protection: (0, 2991, 10163, 30084, 30084)
Node 0, zone      DMA32
        managed 765917
        protection: (0, 0, 14344, 54185, 54185)
Node 0, zone      Normal
        managed 1836032
        protection: (0, 0, 0, 159364, 159364)
Node 0, zone      Movable
        managed 5099663
        protection: (0, 0, 0, 0, 0)
Node 0, zone      Device
        managed 0
        protection: (0, 0, 0, 0, 0)

```

We examine summed managed pages in Table 2-5.

Table 2-5: Bypassed zone pages

Actual zone	Requested zone				
	ZONE_DMA	ZONE_DMA32	ZONE_NORMAL	ZONE_MOVABLE	ZONE_DEVICE
ZONE_DMA	•	765,917	2,601,949	7,701,612	7,701,612
ZONE_DMA32		•	1,836,032	6,935,695	6,935,695
ZONE_NORMAL			•	5,099,663	5,099,663
ZONE_MOVABLE				•	0
ZONE_DEVICE					•

If we apply the ratios we can observe the reserve page requirements in Table 2-6.

Table 2-6: Reserve page requirements

Actual zone	Requested zone				
	ZONE_DMA	ZONE_DMA32	ZONE_NORMAL	ZONE_MOVABLE	ZONE_DEVICE
ZONE_DMA	•	2,991	10,163	30,084	30,084
ZONE_DMA32		•	14,344	54,185	54,185
ZONE_NORMAL			•	159,364	159,364
ZONE_MOVABLE				•	0
ZONE_DEVICE					•

And you can see that these values align with those reported by `/proc/zoneinfo` correctly.

These reserve values are added to the minimum watermark value as determined by `__zone_watermark_ok()` as shown in Listing 2-59.

```

3995      /*
3996      * Check watermarks for an order-0 allocation request. If these
3997      * are not met, then a high-order request also cannot go ahead

```

```

3998      * even if a suitable page happened to be free.
3999      */
4000      if (free_pages <= min + z->lowmem_reserve[highest_zoneidx])
4001          return false;

```

Listing 2-41: mm/page_alloc.c: Excerpt from __zone_watermark_ok()

This is invoked by `zone_watermark_fast()` – if free pages exceed this value the allocation is not permitted as shown in Listing 2-58.

```

4060          if (usable_free > mark + z->lowmem_reserve[highest_zoneidx])
4061              return true;

```

Listing 2-42: mm/page_alloc.c: Excerpt from zone_watermark_fast()

In both cases the `lowmem_reserve[]` value is referenced using the `highest_zoneidx` value, indicating the zone that the allocation was *requested* at.

These page counts are established by `setup_per_zone_lowmem_reserve()` where you can see this algorithm in action. Note that this is triggered by the early memory setup function `init_per_zone_wmark_min()` and on changes in the tunable via `lowmem_reserve_ratio_sysctl_handler()`.

Examining `setup_per_zone_lowmem_reserve()` as shown in Listing 2-43.

```

8642  /*
8643   * setup_per_zone_lowmem_reserve - called whenever
8644   *     sysctl_lowmem_reserve_ratio changes. Ensures that each zone
8645   *     has a correct pages reserved value, so an adequate number of
8646   *     pages are left in the zone after a successful __alloc_pages().
8647   */
8648  static void setup_per_zone_lowmem_reserve(void)
8649  {
8650      struct pglist_data *pgdat;
8651      enum zone_type i, j;
8652
8653      for_each_online_pgdat(pgdat) {
8654          for (i = 0; i < MAX_NR_ZONES - 1; i++) {
8655              struct zone *zone = &pgdat->node_zones[i];
8656              int ratio = sysctl_lowmem_reserve_ratio[i];
8657              bool clear = !ratio || !zone_managed_pages(zone);
8658              unsigned long managed_pages = 0;
8659
8660              for (j = i + 1; j < MAX_NR_ZONES; j++) {
8661                  struct zone *upper_zone = &pgdat->node_zones[j];
8662
8663                  managed_pages += zone_managed_pages(upper_zone);
8664              }
8665
8666              if (clear)

```

```

8666                                     zone->lowmem_reserve[j] = 0;
8667                                     else
8668                                     zone->lowmem_reserve[j] =
                                     managed_pages / ratio;
8669                                     }
8670                                     }
8671                             }
8672
8673                             /* update totalreserve_pages */
8674                             calculate_totalreserve_pages();
8675     }

```

Listing 2-43: mm/page_alloc.c: setup_per_zone_lowmem_reserve()

Here we can see that we iterate through each zone of each online node, cumulatively summing managed pages for each zone above our own and applying the specified ratio to these as described above.

This also triggers a call to `calculate_totalreserve_pages()` which calculates *total* reserved pages combining the lowmem reserve and high watermark of each node and zone.

2.4.2 Total reserved pages

We keep track of the total number of reserved pages within each node in `struct pglist_data->totalreserve_pages` and the sum of these across *all* nodes and zones in the global value `totalreserve_pages`.

What does it mean for pages to be reserved in this context? Simply that we consider them to be unavailable for userland allocation.

These values are calculated in `calculate_totalreserve_pages()` which is invoked in `setup_per_zone_lowmem_reserve()` when the `vm.lowmem_reserve_ratio` tunable is changed and `__setup_per_zone_wmarks()` which is invoked when the `vm.min_free_kbytes` tunable is changed or memory is hot-added or hot-removed.

We examine `calculate_totalreserve_pages()` in Listing 2-44.

```

8603 /*
8604  * calculate_totalreserve_pages - called when sysctl_lowmem_reserve_ratio
8605  *      or min_free_kbytes changes.
8606  */
8607 static void calculate_totalreserve_pages(void)
8608 {
8609     struct pglist_data *pgdat;
8610     unsigned long reserve_pages = 0;
8611     enum zone_type i, j;
8612
8613     for_each_online_pgdat(pgdat) {
8614
8615         pgdat->totalreserve_pages = 0;
8616

```

```

8617         for (i = 0; i < MAX_NR_ZONES; i++) {
8618             struct zone *zone = pgdat->node_zones + i;
8619             long max = 0;
8620             unsigned long managed_pages = zone_managed_pages(zone)
;
8621
8622             /* Find valid and maximum lowmem_reserve in the zone
*/
8623             for (j = i; j < MAX_NR_ZONES; j++) {
8624                 if (zone->lowmem_reserve[j] > max)
8625                     max = zone->lowmem_reserve[j];
8626             }
8627
8628             /* we treat the high watermark as reserved pages. */
8629             max += high_wmark_pages(zone);
8630
8631             if (max > managed_pages)
8632                 max = managed_pages;
8633
8634             pgdat->totalreserve_pages += max;
8635
8636             reserve_pages += max;
8637         }
8638     }
8639     totalreserve_pages = reserve_pages;
8640 }

```

Listing 2-44: mm/page_alloc.c: calculate_totalreserve_pages()

We obtain the most conservative value possible – a combination of the maximum low memory reserve described above in section 2.4.1 with the *high* water mark of each zone.

Recall that at the low water mark indirect reclaim is initiated (a background process which tries to ‘reclaim’, i.e. evict pages from memory, which prioritises freeing memory used to cache file data), and at the minimum water mark this is done rather more forcefully, as a blocking call.

The high water mark denotes the point at which indirect reclaim is halted. When free memory passes through this level, nothing happens, it is only meaningful when performing reclaim (see the reclaim chapter for a detailed examination of how this functions).

However, it is a useful heuristic for a zone being in good health – if its free pages sit above the high water mark, then no reclaim is necessary under *any* circumstances.

This algorithm sums the high water mark with the maximum possible low memory zone protection – this is to account for the worst case scenario allocation in each zone, i.e. an allocation from the zone which bypasses the most pages in order to allocate from a lower one.

2.5 Migrate types

Within the kernel physical memory is subdivided into node, zone, order and migrate type. Nodes divide memory by memory locality (i.e. whether accessing some memory involves obtaining it from a long ‘distance’), zones divide memory by memory range, which can be of critical importance to hardware (if a device cannot DMA to/from memory allocated for that purpose it’s useless) or meaningful for the purposes of grouping like allocations together (as is the case for `ZONE_MOVABLE` and `ZONE_DEVICE`). Order determines the amount of *physically contiguous* memory allocated. But what about migrate type?

Migrate types, as the name implies, categorises pages based on their *mobility*. Some memory is able to be *moved*, some is able to be *reclaimed* and some is entirely *unmovable*. What does ‘moving’ memory in this context mean? When memory is virtually mapped (more on this in the next chapter), users of the memory are oblivious to the actual physical pages being referenced. As a result, these mappings can be moved around by the kernel (termed *migration*, covered in detail in the chapter on this subject).

These three cases form the fundamental migrate types, though there are additional types for specific use case. These are defined by enum `migratetype`:

- `MIGRATE_UNMOVABLE` – These pages cannot be migrated at all. This is typically the case for kernel allocations which are accessed essentially directly indexed by their physical address and thus no abstraction exists to permit page movement.
- `MIGRATE_MOVABLE` – These pages can be migrated without restriction. Typically the case for userland allocations.
- `MIGRATE_RECLAIMABLE` – Rather confusingly this refers specifically to slab-allocated memory that is otherwise unmovable but might be *shrunk* under memory pressure by invoking ‘page shrinkers’ to remove unused pages. This is typically specified by the `SLAB_ACCOUNT` flag when invoking `kmem_cache_create()`.
- `MIGRATE_HIGHATOMIC` – A special migrate type used to reserve memory for atomic allocations (ones that are performed in an uninterruptible state) of higher order.
- `MIGRATE_CMA` – These pages are used for allocating memory using the Contiguous Memory Allocator (CMA). This mechanism reserves memory early in boot which is then added back to the buddy allocator under `MIGRATE_CMA`. These pages can be used for `MIGRATE_MOVABLE` allocations or CMA-specific ones but nothing else. This way, pages can be moved as necessary to provide physically contiguous blocks of memory on demand. The details of how CMA is implemented are out of scope for this book.
- `MIGRATE_ISOLATE` – Pages marked with this migrate type are not permitted to be allocated at all. This is used by the CMA mechanism early on to transfer memory from other migrate types to `MIGRATE_CMA`, as well as memory hot-plugging. The details of this are out of scope for this book.

The purpose of categorising memory in this way is in furtherance of the never ending and inevitably heuristic quest to reduce memory fragmentation. In the context of a buddy allocator (as described in the last chapter), we *coalesce* smaller pages into larger ones when they are freed. If we find that allocated memory sits between blocks that could otherwise be coalesced with their buddies and those pages are movable, we can *migrate* them.

However, if we mix movable and unmovable pages together, we can end up in painful situations where this fragmentation prevents us from being able to do so. Equally, for slab allocations which are otherwise unmovable, we can shrink memory freeing up contiguous space which can then be coalesced. Again if unmovable pages sit between shrinkable memory our ability to do so is impeded.

It is therefore sensible to group these different migrate types together. The kernel does this by assigning migrate types to *page blocks*. These are contiguous blocks of physical memory of order `pageblock_order` which is typically order-9 (i.e. 2 MiB of memory). These blocks are naturally aligned, so are able to be coalesced to page block order.

How is this actually implemented in the kernel? All free lists are subdivided by migrate type (more on free lists later) and allocations will specify which migrate type is required (more on this later also), and so pages can be allocated for the desired migrate type.

The migrate type of a page is determined via `get_pageblock_migratetype()` which looks up the migrate type assigned to a page block (see section 2.3.4 for details of how and where this is stored). This is used when freeing pages to know which free lists to place them in. This is not the case for Per-CPU-Page (PCP) lists which stores migrate type separately for fast access, but PCP pages derive their migrate type from the page block in the first instance. See section 2.7.3 for more details on this.

During early boot all available page blocks are marked `MIGRATE_MOVABLE` by `memmap_init_zone_range()` which invokes `memmap_init_range()` and finally `set_pageblock_migratetype()` which actually sets this.

When allocations are attempted but no pages exist of the required migrate type, migrate type *stealing* occurs. This is where free pages are taken from another migrate type and placed in the free list of the required migrate type to be able to fulfil the request. This is implemented in `__rmqueue_fallback()`. This is covered in section 2.8.6 in considerable detail. `__rmqueue_fallback()` will try to fallback to another migrate type as specified by a list of fallbacks in priority order of *mergeable* migrate types as determined by `find_suitable_fallback()` (only unmovable, movable and reclaimable migrate types are mergeable – as specified by `migratetype_is_mergeable()`).

Crucially, except in one very specific case, this effort will try to steal the *largest* order page possible. This helps reduce fragmentation by reducing the degree to which these efforts will cause page blocks to end up mixing up migrate types.

The fact we are able to ‘steal’ pages from other migrate types means it is possible for a page block to contain a mix of migrate types. This is un-

desirable but sometimes necessary under memory pressure. As a result, `get_pageblock_migratetype()` is merely heuristic and a page block's migrate type is set to the migrate type *most* represented within the page block. Note that if migrate type fragmentation occurs, watermarks are subject to a *boost*, reserving more pages in order to offset the increased fragmentation.

Importantly, the fact that this stealing mechanism attempts to steal the largest possible order page from another migrate type means it is often the case that page block order pages or above can be stolen, which are then simply reassigned to the required migrate type. This means that during early allocations of the buddy allocator page blocks can be transferred from the default of `MIGRATE_MOVABLE` to other migrate types as required without incurring fragmentation (once one higher order page is transferred it can be split into smaller pages as needed).

Also it's important to note that `can_steal_fallback()` will simply permit page blocks to be transferred from one type to another if the target migrate type is `MIGRATE_RECLAIMABLE` or `MIGRATE_UNMOVABLE`. As both of these cannot be migrated (`MIGRATE_RECLAIMABLE` pages can only be shrunk relative to one another), so stealing actually migratable pages that could otherwise later be coalesced together is considered more of a sin than simply taking a whole block at a time.

There couldn't be a greater example of how migrate types are a heuristic mechanism for the reduction of fragmentation which is itself by its nature a heuristic task.

2.6 GFP flags

When allocating memory GFP ('Get Free Pages') flags are used to specify how memory is to be allocated. All kernel allocations must specify these flags which are defined in the `include/linux/gfp.h` header.

The flags are comprised of a series of modifiers which are prefixed with `__GFP` e.g. `__GFP_DMA`, and useful combinations which are prefixed `GFP_`, e.g. `GFP_KERNEL`. The flags can be bitwise-or combined as needed. The kernel divides modifiers into different categories, which we will examine in turn.

2.6.1 Physical address zone modifiers

These specify which zone the allocation should originate from:

- `__GFP_DMA` – Indicates that the allocation should originate from `ZONE_DMA`.
- `__GFP_HIGHMEM` – Indicates that the allocation should originate from `ZONE_HIGHMEM`. Not relevant for 64-bit systems.
- `__GFP_DMA32` – Indicates that the allocation should originate from `ZONE_DMA32`.

The `GFP_ZONEMASK` values provides a mask for all of these values in addition to `__GFP_MOVABLE` which indicates that a page *can* be placed in `ZONE_MOVABLE`, if configured and available.

NOTE

ZONE_MOVABLE is a zone that exists if the `kernelcore` and/or `movablecore` command line parameters are specified which provides a physically contiguous range of memory where movable pages can be placed.

2.6.2 Page mobility and placement hints

These flags indicate how mobile pages can be (e.g. to what degree the kernel can move them around):

- `__GFP_MOVABLE` – Indicates that the page should have a migrate type of `MIGRATE_MOVABLE` (determined by `gfp_migratetype()`) and whose pages can be moved by the kernel as needed (the process of moving underlying pages of memory is termed *migration*). Additionally, it indicates that if `ZONE_MOVABLE` spans sufficient pages memory can be allocated from this zone. This flag straddles both the mobility and zone modifier categories.
- `__GFP_RECLAIMABLE` – Indicates that the page should have a migrate type of `MIGRATE_RECLAIMABLE` (again, determined by `gfp_migratetype()`). For slab allocations it refers to memory which has the `SLAB_RECLAIM_ACCOUNT` flag set which can have unused pages freed via ‘shrinkers’ on reclaim (not to be confused with `__GFP_RECLAIM`). More on that in the slab chapter.
- `__GFP_WRITE` – A hint indicating that file-backed memory will be written to. This is a placement hint as it sets the `spread_dirty_pages` flag in the allocation context (more on allocation context later) in `prepare_alloc_pages()`. This will be used to distribute potentially dirty pages amongst nodes in order that no node has its dirty page limit exceeded (dirty pages are pages that have changes which need to be flushed back to disk). This is only respected on the allocator’s fast path, if no node is available with a sufficiently low dirty page count, then this flag will be ignored.
- `__GFP_HARDWALL` – A hint indicating that the process’s cgroup `cpuset.mems` setting should be honoured, i.e. that allocations should occur on the specified nodes only (cpusets are a feature of cgroups which allows users to limit processes to specific CPUs and memory nodes). See the cgroup v2 admin guide on cpuset settings. This is set in `prepare_alloc_pages()`. Note that cgroups are out of scope for the book.
- `__GFP_THISNODE` – Strictly requires that pages must be allocated from the requested node. This is used in `gfp_zonelist()` to restrict available zones to those only on the specified node.
- `__GFP_ACCOUNT` – Indicates that, if the cgroups `kmemcg` controller is enabled, this page should be ‘charged’ to it, and if not permitted by the controller to allocate then no allocation will occur. Referenced directly in `__alloc_pages()`.

2.6.3 Watermark modifiers

These flags determine how watermark limits are enforced on allocation and relatedly how zone emergency reserves can be made use of:

- `__GFP_ATOMIC` – This indicates that the caller can't sleep (and as a result direct reclaim) due to being in an atomic context such as an interrupt handler. A number of tweaks are performed on allocation to suit this scenario.
- `__GFP_HIGH` – This indicates that the allocation is high-priority and must be performed in order for the system to make forward progress. This will reduce watermarks making more reserved memory available to allocators, as `__GFP_HIGH` implies the allocator flag `ALLOC_HIGH`.
- `__GFP_MEMALLOC` – This is an extreme flag to use as it makes all memory within a zone available for allocation regardless of watermarks. It is intended for cases where the caller knows that memory is on the verge of being made available immediately after this allocation. This flag should generally only be used by the memory manager subsystem internally as it risks exhausted memory entirely otherwise.
- `__GFP_NOMEMALLOC` – Expressly forbids access to emergency memory reserves altogether. Overrides `__GFP_MEMALLOC` if both flags are set.

2.6.4 Reclaim modifiers

These flags indicate what reclaim can be performed on allocation:

NOTE

Reclaim is the process by which physical memory is reclaimed by discarding file-backed pages and swapping out to disk, discussed in far more detail in Chapter 11.

- `__GFP_IO` – Indicates that I/O may be performed. This is used by the reclaim logic in `may_enter_fs()` – if the memory is in the swap buffer (see later chapters on this) and `__GFP_FS` is not set. It is also used in compaction logic (again, see the chapter on this) to determine if I/O can be performed during this process, as checked in `try_to_compact_pages()`. Clearing this flag is used when no I/O *at all* can be permitted. Will typically be used via the helper flag `GFP_NOIO`.
- `__GFP_FS` – Indicates that low-level filesystem calls should be performed (whether I/O is performed at all is determined by `__GFP_IO`). This is typically cleared when filesystem locks are acquired and an allocation resulting in filesystem operations might cause deadlock. This is used in reclaim via the `may_enter_fs()` check – if `__GFP_FS` is set, then this is set true unconditionally. Will typically be invoked via the helper flag `GFP_NOFS`.
- `__GFP_DIRECT_RECLAIM` – Indicates that *direct* reclaim can be performed – this is where, under severe memory pressure, the allocator tries to reclaim memory *immediately* upon allocation. This should be cleared if failing to allocate is acceptable (e.g. there are fallback options). Direct

reclaim might sleep. If this flag is cleared then the OOM killer will not be invoked (checked in `__alloc_pages_slowpath()`, see the reclaim chapter for more on this).

- `__GFP_KSWAPD_RECLAIM` – Indicates that *indirect* reclaim can be performed by the kswapd kernel thread when the low watermark level is reached for a zone. See the reclaim chapter for more details.
- `__GFP_RECLAIM` – A helper flag which can be used to determine whether reclaim is set *at all* or to enable/disable reclaim altogether. Equal to `__GFP_DIRECT_RECLAIM | __GFP_KSWAPD_RECLAIM`.
- `__GFP_RETRY_MAYFAIL` – If a page has a *costly order* (i.e. has order greater than `PAGE_ALLOC_COSTLY_ORDER`) then we typically don't retry a direct reclaim attempt. This overrides this behaviour. Additionally, if set, the OOM killer will not be invoked (checked in `__alloc_pages_may_oom()`).
- `__GFP_NORETRY` – Only attempt to direct reclaim once then give up, even if forward progress could have been made in another attempt. This also does not invoke the OOM killer if insufficient memory is available to satisfy the request.
- `__GFP_NOFAIL` – A drastic flag that should be used with great care, this does the diametric opposite of `__GFP_NORETRY` and direct reclaim instead loops indefinitely until a page is successfully obtained. Sanity prevails in that a kernel warning will be generated if you attempt this with pages of order 2 or greater.

2.6.5 Action modifiers

Action modifiers either enable or disable specific actions taken by the allocator:

- `__GFP_NOWARN` – Prevent the kernel from outputting warnings about allocation failures, most pertinently in `warn_alloc()`. Typically used when performing allocations for which there are fallbacks or otherwise whose failures are not abnormal events.
- `__GFP_COMP` – Indicates that, if the allocation is of greater than order 0, the page should be configured to be compound via `prep_compound_page()`.
- `__GFP_ZERO` – After allocating a page, zero the memory. Used in e.g. `get_zeroed_page()`. Checked in `want_init_on_alloc()` and performed in `post_alloc_hook()` which invokes `kernel_init_pages()` to do the heavy lifting. If `__GFP_SKIP_ZERO` is set this is ignored.
- `__GFP_ZEROTAGS` – Zeros memory tags (an arm64-specific memory security feature), ultimately implemented in `tag_clear_highpage()` which is in turn invoked by `post_alloc_hook()`. Only applicable if `__GFP_ZERO` is set and `__GFP_SKIP_ZERO` is not.
- `__GFP_SKIP_ZERO` – If set, prevents page/tag zeroing even if the appropriate GFP flags are also set. Available and relevant only if `CONFIG_KASAN_HW_TAGS` is set.

- `__GFP_SKIP_KASAN_UNPOISON` – Causes KASAN to skip unpoisoning on page allocation. KASAN is out of scope for this book. Available and relevant only if `CONFIG_KASAN_HW_TAGS` is set.
- `__GFP_SKIP_KASAN_POISON` – Causes KASAN to skip poisoning on page free. KASAN is out of scope for this book. Available and relevant only if `CONFIG_KASAN_HW_TAGS` is set.
- `__GFP_NOLOCKDEP` – If lock dependency tracking is enabled (via `CONFIG_LOCKDEP`) this appears to disable direct reclaim in order to avoid lock dependency checking there in `__need_reclaim()`.

2.6.6 *Predefined GFP flag combinations*

Rather than specifying individual flags it is far more convenient to use a sensible predefined combination of them. Examining each of these:

- `GFP_KERNEL` – `__GFP_RECLAIM`, `__GFP_IO`, `__GFP_FS` – Standard kernel allocation, both direct and indirect reclaim is permitted, as is disk I/O. May sleep.
- `GFP_HIGHUSER_MOVABLE` – `GFP_HIGHUSER`, `__GFP_MOVABLE`, `__GFP_SKIP_KASAN_POISON` – The typical userland allocation which is allocated from `do_anonymous_page()` which invokes `alloc_zeroed_user_highpage_movable()` though more typically you have an arch-specific one, i.e. x86-64 has `alloc_zeroed_user_highpage_movable()` and arm64 has `alloc_zeroed_user_highpage_movable()` also. These both combine this flag with `__GFP_ZERO` to perform the zeroing. The reference to high memory is not relevant to modern 64-bit systems so is not meaningful here.
- `GFP_HIGHUSER` – `GFP_USER`, `__GFP_HIGHMEM` – Equivalent to `GFP_USER` for modern 64-bit systems.
- `GFP_USER` – `__GFP_RECLAIM`, `__GFP_IO`, `__GFP_FS`, `__GFP_HARDWALL` – Used for allocations that are typically to be accessed by both the kernel and userland (because it does not mark the memory movable). Also provides the basis for flag sets for other userland allocations. Identical to kernel except that hardwall is enabled, permitting cpuset usage.
- `GFP_ATOMIC` – `__GFP_ATOMIC`, `__GFP_HIGH`, `__GFP_KSWAPD_RECLAIM` – Allocation under conditions where the caller cannot sleep (e.g. in interrupt context) but indirect reclaim is permissible and the allocation should be marked high-priority.
- `GFP_KERNEL_ACCOUNT` – `GFP_KERNEL`, `__GFP_ACCOUNT` – Kernel allocation with `kmemcg` accounting.
- `GFP_NOWAIT` – `__GFP_KSWAPD_RECLAIM` – Only permits indirect reclaim, disables I/O and filesystem access altogether on direct reclaim and is to be used when absolutely no delays can be tolerated on allocation.
- `GFP_NOIO` – `__GFP_RECLAIM` – The same as `GFP_KERNEL` except both `__GFP_IO` and `__GFP_FS` flags are not set meaning no I/O at all is permitted on direct reclaim. Typically this should not be used directly but rather set by `memalloc_noio_save()` and unset by `memalloc_noio_restore()`.

- `GFP_NOFS` – `__GFP_RECLAIM`, `__GFP_IO` – Will direct reclaim and potentially perform I/O (and thus possibly sleep) but will not use low level filesystem interfaces. This is typically applicable where locks are held in the filesystem and reclaim using filesystem interfaces might deadlock. As with `GFP_NOIO` this should not be used directly but rather set via `memalloc_nofs_save()` and unset by `memalloc_nofs_restore()`.
- `GFP_DMA` – `__GFP_DMA` – No reclaim whatsoever and allocate from `ZONE_DMA`. Used for *very* legacy device memory allocation. In the case of architectures that place all memory only in `ZONE_DMA` this designation is less meaningful.
- `GFP_DMA32` – `__GFP_DMA32` – No reclaim whatsoever and allocate from `ZONE_DMA32`. Used for device memory allocation where the device has a 32-bit address bus width.
- `GFP_TRANSHUGE_LIGHT` – `GFP_HIGHUSER_MOVABLE`, `__GFP_COMP`, `__GFP_NOMEMALLOC`, `__GFP_NOWARN`, with `__GFP_RECLAIM` masked – Used for Transparent Huge Page allocations (THP), see huge page chapter for more details on this. This light version attempts no reclaim and is designed to fail fast if the allocation fails.
- `GFP_TRANSHUGE` – `GFP_TRANSHUGE_LIGHT`, `__GFP_DIRECT_RECLAIM` – Another THP flag which additionally enables direct reclaim.

2.6.7 Memalloc Flags

It is possible to specify process-specific flags which, while they are set, adjust allocation GFP flags. These are used for situations where the process cannot perform certain actions. The GFP modifications are applied by `current_gfp_context()` except for `PF_MEMALLOC`. The process flags are as follows:

- `PF_MEMALLOC_NOIO` – This clears `__GFP_IO` and `__GFP_FS` flags disabling all direct reclaim I/O or filesystem access. Set by `memalloc_noio_save()` and restored by `memalloc_noio_restore()`.
- `PF_MEMALLOC_NOFS` – This clears `__GFP_FS` disabling all direct reclaim filesystem access. Set by `memalloc_nofs_save()` and restored by `memalloc_nofs_restore()`.
- `PF_MEMALLOC_PIN` – This clears `__GFP_MOVABLE` ensuring that the migrate type will not be `MIGRATE_MOVABLE`. Set by `memalloc_pin_save()` and restored by `memalloc_pin_restore()`.
- `PF_MEMALLOC` – This prevents reclaim from occurring (as checked in `__need_reclaim()`) and ignores watermarks (setting the allocation flag `ALLOC_NO_WATERMARKS` in `__gfp_pfmemalloc_flags()`). This is set by `memalloc_noreclaim_save()` and restored by `memalloc_noreclaim_restore()`.

2.7 Buddy allocator

2.7.1 Algorithm

The key algorithm underpinning physical memory allocation in the kernel is a *buddy* allocator. This is a conceptually simple algorithm for allocating memory in power-of-2 chunks and is able to very quickly coalesce blocks of memory when they are freed, as described in the previous chapter on allocators.

Briefly recapping – the idea is to initially divide available memory into the largest possible power-of-two blocks up to a maximum limit (`MAX_ORDER - 1`). The kernel's buddy allocator allocates base pages of memory (the *slab* allocator allow for allocations of smaller size, see the following chapter on slab allocators for details on this). Each power-of-two block size is termed the block's *order*, which equates to 2^{order} base pages.

A separate free list is maintained for each node, zone, order and migrate type. When memory is allocated, the allocator tries allocating from the order requested. If no block is available at this size, it checks each free list for orders greater than requested until it finds an available block. If this fails, then different migrate types are tried (see section 2.8.6 for more details on this).

At this point the block is split into two parts to add 2 pages to the free list for order - 1. Each block that is split is called the *buddy* of the other. This process is repeated until pages at the required order are available.

The real 'trick' of the buddy allocator is being able to quickly determine the PFN of the buddy of a block given its PFN. To do this, we ensure that each block of memory is aligned to its block size. By doing so, it becomes very simple to determine the address of the two blocks that a larger block was split into e.g. if the order-3 block at PFN 0b01011000 were split into two order-2 blocks, this would result in buddies at 0b01011000 and 0b01011100.

This makes it very easy to determine a block's 'buddy' in constant time – simply flip the bit at position *order*, e.g. in the example given above, flip bit 2. This would map 0b01011000 to buddy 0b01011100 and 0b01011100 to buddy 0b01011000 as expected.

Another nice property of using a buddy allocator is that we can obtain the combined PFN of a coalesced block and its buddy by performing a bitwise and between them - this will clear the order bit leaving the lower of the two addresses which now comprises both at order + 1, e.g. 0b01011100 and 0b01011000 have a combined PFN of 0b01011000.

To achieve this programmatically we need to use the *exclusive or* binary operation to achieve the bit flip which is done in `__find_buddy_pfn()` as shown in Listing 2-45.

```

307 static inline unsigned long
308 __find_buddy_pfn(unsigned long page_pfn, unsigned int order)
309 {
310     return page_pfn ^ (1 << order);
311 }
```

Listing 2-45: mm/internal.h: `__find_buddy_pfn()`

When freeing pages, we also try to coalesce them. This is the beauty of the buddy allocator algorithm – since each block is either of maximum order or has been divided into two, we simply try to coalesce with our buddy, then try to coalesce *that* block with its buddy and so on until we obtain the largest block possible, thereby doing the inverse of the splitting performed on allocation.

Since looking up our buddy is incredibly simple, this can be done very quickly and the whole operation takes $O(\text{MAX\_ORDER})$ time.

The actual process of obtaining a buddy page involves a little more effort as shown in Listing 2-46.

```

327 static inline struct page *find_buddy_page_pfn(struct page *page,
328         unsigned long pfn, unsigned int order, unsigned long *
        buddy_pfn)
329 {
330     unsigned long __buddy_pfn = __find_buddy_pfn(pfn, order);
331     struct page *buddy;
332
333     buddy = page + (__buddy_pfn - pfn);
334     if (buddy_pfn)
335         *buddy_pfn = __buddy_pfn;
336
337     if (page_is_buddy(page, buddy, order))
338         return buddy;
339     return NULL;
340 }
```

Listing 2-46: mm/internal.h: `find_buddy_page_pfn()`

This function obtains both the buddy PFN *and* its struct page. Note that `__buddy_pfn - pfn` might underflow (all values are unsigned), however this is intentional – the buddy PFN might be greater than or less than the page’s PFN. What is key here is the invocation of `page_is_buddy()` which checks that the buddy page is actually present and available as shown in Listing 2-47.

```

256 /*
257  * This function checks whether a page is free && is the buddy
258  * we can coalesce a page and its buddy if
259  * (a) the buddy is not in a hole (check before calling!) &&
260  * (b) the buddy is in the buddy system &&
261  * (c) a page and its buddy have the same order &&
262  * (d) a page and its buddy are in the same zone.
263  *
264  * For recording whether a page is in the buddy system, we set PageBuddy.
265  * Setting, clearing, and testing PageBuddy is serialized by zone->lock.
266  *
```

```

267  * For recording page's order, we use page_private(page).
268  */
269  static inline bool page_is_buddy(struct page *page, struct page *buddy,
270                                unsigned int order)
271  {
272      if (!page_is_guard(buddy) && !PageBuddy(buddy))
273          return false;
274
275      if (buddy_order(buddy) != order)
276          return false;
277
278      /*
279       * zone check is done late to avoid uselessly calculating
280       * zone/node ids for pages that could never merge.
281       */
282      if (page_zone_id(page) != page_zone_id(buddy))
283          return false;
284
285      VM_BUG_ON_PAGE(page_count(buddy) != 0, buddy);
286
287      return true;
288  }

```

Listing 2-47: mm/internal.h: page_is_buddy()

Examining the logic:

1. Guard pages are out of scope for this chapter (and only relevant if CONFIG_DEBUG_PAGEALLOC is set in any case) so disregarding these we start by checking whether the page is managed by the buddy page allocator via PageBuddy(). This should handle all cases of buddy pages not being available.
2. Next we check to ensure the candidate buddy page is actually of the same order as the page whose buddy we are seeking via buddy_order(). Buddy pages store their order in the private field of struct page and this retrieves it.
3. Finally, we ensure that the buddies do not cross a zone boundary.

See the previous chapter on allocators for a detailed examination of the buddy allocator algorithm.

2.7.2 Free lists

Free lists are subdivided by node, zone, order and migrate type.

Node metadata is stored in pg_data_t objects which contains per-zone data in the struct zone pg_data_t->node_zones[MAX_NR_ZONES] field.

Each struct zone contains an array of free lists for each page order size in the struct free_area struct zone->free_area[MAX_ORDER] field protected by the spinlock_t lock field.

It's important to note that one zone (which is fundamentally a physical address range) might be subdivided into several struct zone objects, as each node's portion of a zone will have its own object.

Examining the struct free_area type as shown in Listing 2-48.

```

105 struct free_area {
106     struct list_head    free_list[MIGRATE_TYPES];
107     unsigned long       nr_free;
108 };

```

Listing 2-48: include/linux/mmzone.h: struct free_area

This simply contains the heads of the free lists themselves per-migrate type and a count of free blocks. The free lists consist of struct page objects which use the struct page->lr field to store list nodes.

Pages are added to free lists via add_to_free_list() and add_to_free_list_tail(), moved to a different free list via move_to_free_list() and deleted from a free list via del_page_from_free_list().

2.7.3 Per-CPU free Pages (PCPs)/Pagesets

Aside from the core free lists there is an additional 'cache' of per-CPU free lists for pages of lower orders to allow efficient allocation without acquiring a lock to do so, stored in struct per_cpu_pages objects in the struct zone->per_cpu_pageset field. Per-CPU Pages are abbreviated *PCPs*, and the set of each CPU's PCPs is referred to as its *pageset*.

Each physical allocation first tries to allocate from pageset pages, and each free tries to free to pageset pages. These pages are essentially in limbo (they are not actually freed) and act as a cache between allocations and the actual free lists as an optimisation.

NOTE

You can obtain per-zone pagestat statistics via /proc/zoneinfo.

We examine the struct per_cpu_pages type in Listing 2-49.

```

383 /* Fields and list protected by pagesets local_lock in page_alloc.c */
384 struct per_cpu_pages {
385     spinlock_t lock;           /* Protects lists field */
386     int count;                 /* number of pages in the list */
387     int high;                  /* high watermark, emptying needed */
388     int batch;                 /* chunk size for buddy add/remove */
389     short free_factor;         /* batch scaling factor during free */
390 #ifdef CONFIG_NUMA
391     short expire;              /* When 0, remote pagesets are drained */
392 #endif
393
394     /* Lists of pages, one per migrate type stored on the pcp-lists */
395     struct list_head lists[NR_PCP_LISTS];
396 } ____cacheline_aligned_in_smp;

```

Listing 2-49: include/linux/mmzone.h: struct per_cpu_pages

This is stored in the struct zone type with an entry for each system CPU. Per-CPU data is essentially an array of data (aligned to cache line size to avoid false sharing), indexed by CPU. They are usually allocated by alloc_percpu() and referenced by per_cpu_ptr() or, for convenience, this_cpu_ptr(). We examine the struct zone per-CPU pageset field in Listing 2-50.

```

515 struct zone {
    :
539     struct per_cpu_pages    __percpu *per_cpu_pageset;
    :
683 } ____cacheline_internodealigned_in_smp;
```

Listing 2-50: include/linux/mmzone.h: struct zone per_cpu_pageset field

Entries are initialised by build_all_zonelists_init() during early boot which uses boot_pageset initially until the memory manager is initialised.

After the memory manager is initialised, setup_zone_pageset() is used to allocate pagesets overriding the initial boot values via alloc_percpu() and initialises them via per_cpu_pages_init() and zone_set_pageset_high_and_batch().

Pagesets get populated if the page order is not too costly on free (see next section on freeing where we go into more detail), up to *and including* order of PAGE_ALLOC_COSTLY_ORDER, which is currently set to 3, meaning that pages of order-0, order-1, order-2 and order-3 can form part of pagesets. pcp_allowed_order() is used to determine whether a page order is permitted in a PCP.

NR_PCP_LISTS specifies the number of lists per pageset, which is equal to the number of migrate types which are permitted on PCPs, MIGRATE_PCPTYPES, multiplied by PAGE_ALLOC_COSTLY_ORDER + 1 (i.e. the count of orders supported), plus NR_PCP_THP which is set to 1 if CONFIG_TRANSPARENT_HUGEPAGE is set. This provides free lists for all supported migrate types and page orders (plus an additional ‘order’ for transparent huge pages, more on this in the chapter on huge pages). Examining each field:

- **lock** – Lock to protect the lists field typically acquired via pcp_spin_lock_irqsave() or pcp_spin_trylock_irqsave() and released by pcp_spin_unlock_irqrestore(). Note that we disable IRQs because free_pcppages_bulk() requires it (see the comment in drain_zone_pages() for more explanation).
- **count** – This is a simple count of the number of *base pages* in the pageset (when a page is freed to/allocated from a pageset, the count field is incremented/decremented by 2^{order}).
- **high** – Indicates the *high* watermark of pages permitted to be on this pageset. If exceeded, pages are *drained* back to the ordinary free lists. This value is not read directly but rather via nr_pcp_high() which either returns the high value, or a lesser value depending on whether the zone

is undergoing reclaim/too many higher order pages are present on the pageset. Updated by `pageset_update()`. This is set either to a default—the zone ‘low watermark’, which is the number of free pages allowed in a zone before reclaim starts, divided by the number of CPUs— via `zone_highsize()` or via the `vm.percpu_pagelist_high_fraction` tunable.

- `batch` – Determines the minimum number of base pages to allocate into a list for a specific migrate type/order when it is empty, and how many base pages to free back to free lists when the high watermark is exceeded. The actual number of pages to allocate is determined by `__rmqueue_pcp_list()` and to free back by `nr_pcp_free()`. This field is set by `zone_batchsize()` to the smaller of 1 MiB/0.1% of the managed pages of the zone.
- `free_factor` – A value which is divided by 2 on allocation from a pageset and multiplied by 2 on pages being freed to a pageset. `batch` is multiplied by $2^{\text{free\_factor}}$ when freeing pageset pages back to the free lists. If this value is greater than zero, then we always free pages back to free lists when freeing an order-1 or greater page to a pageset to reduce higher order fragmentation. See the following section for more details on this. This is a form of exponential back-off.
- `expire` – This field is used by `refresh_cpu_vm_stats()` to drain a batch of pagesets for remote zones (i.e. on node in which the core executing the refresh does not reside) after a delay.
- `lists` – Contains PCP free lists for every supported migrate type and order. The lists are of struct `page` objects tied together using their `lru` field.

2.8 Allocator implementation

2.8.1 Visualising the allocator

The functions which ultimately invoke the physical page allocator are as follows. The key function underlying the allocator is `__alloc_pages()`, with `__alloc_pages_bulk()` being a version of the same logic designed for efficient allocation of *multiple* pages in bulk as shown in Figure ??.

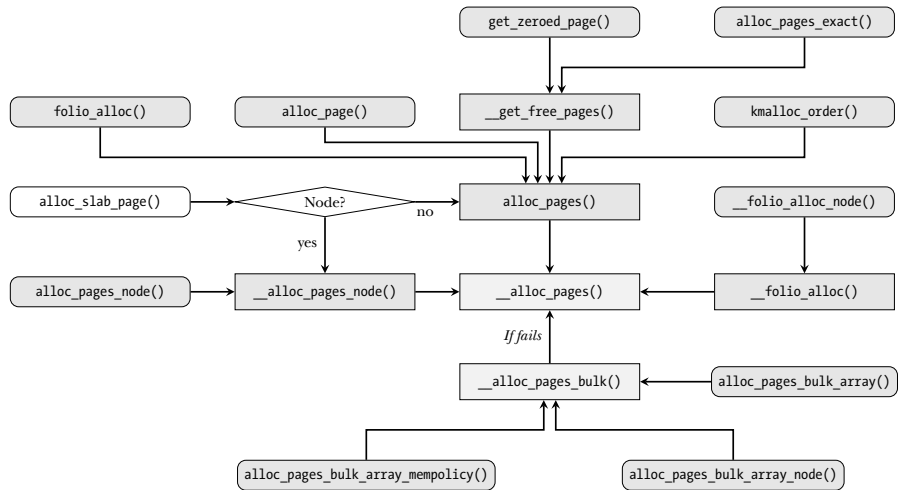


Figure 2-13: Physical page allocation in the kernel

Note that:

- Denotes functions that are exported.
- Denotes the key functions which all others ultimately arrive at.

As this demonstrates, all non-bulk physical memory allocations are ultimately performed by `__alloc_pages()`. This is therefore the key function to analyse in order to understand physical allocation within the kernel as shown in Listing 2-51.

```

5513 struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
5514                             nodemask_t *nodemask)

```

Listing 2-51: `mm/page_alloc.c: __alloc_pages()` prototype

The `gfp` field specifies GFP flags which includes any zone specification, order specifies the required page order (if greater than 0 then the function returns a compound page), `preferred_nid` specifies a *preferred* node for the allocation and `nodemask` specifies which nodes we are able to allocate memory on. The `nodemask_t` type is a bitmap of size equal to `MAX_NUMNODES`.

It then uses these to generate a base set of parameters used throughout the allocation via `prepare_alloc_pages()` and `current_gfp_context()`. This results in initial values of:

- `order = order`
- `alloc_gfp = gfp`
 - `& ~__GFP_FS` if `PF_MEMALLOC_NOIO/PF_MEMALLOC_NOFS` is set.
 - `& ~__GFP_IO` if `PF_MEMALLOC_NOIO` is set.
 - `& ~__GFP_MOVABLE` if `PF_MEMALLOC_PIN` is set.

- `alloc_flags = ALLOC_WMARK_LOW`
 - | `ALLOC_KSWAPD` *if* `__GFP_KSWAPD_RECLAIM` is set.
 - | `ALLOC_NOFRAGMENT` *if* `ZONE_NORMAL` and `ZONE_DMA32` populated.
 - | `ALLOC_CPUSET` *if* cpusets enabled and interrupt context/nodemask set.
- `ac = struct alloc_context {`
 - . `zonelist = node_zonelist(preferred_nid, gfp)`
 - . `migratetype = gfp_migratetype(gfp)`
 - . `highest_zoneidx = gfp_zone(gfp)`
 - . `spread_dirty_pages = gfp & __GFP_WRITE`
 - . `preferred_zoneref = first_zones_zonelist(zonelist, highest_zoneidx, nodemask)`
 - . `nodemask = nodemask` [*or if* cpusets enabled, not in interrupt context and no nodemask specified = `cpuset_current_mems_allowed`]

Let's go on a whistle-stop tour of `__alloc_pages()` and examine the entire call stack involved in allocating physical memory as shown in Figure 2-14.

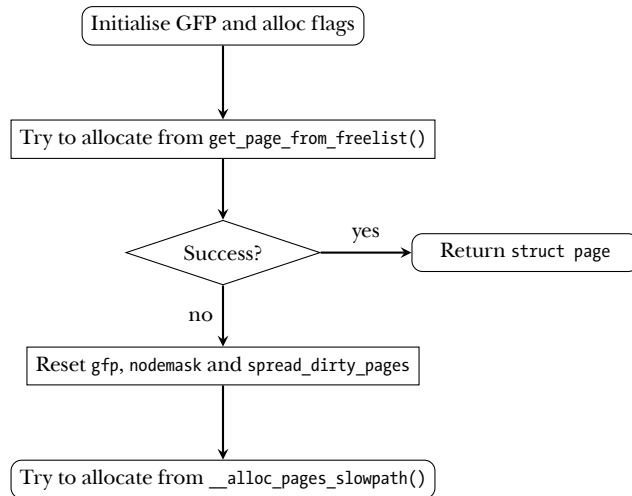
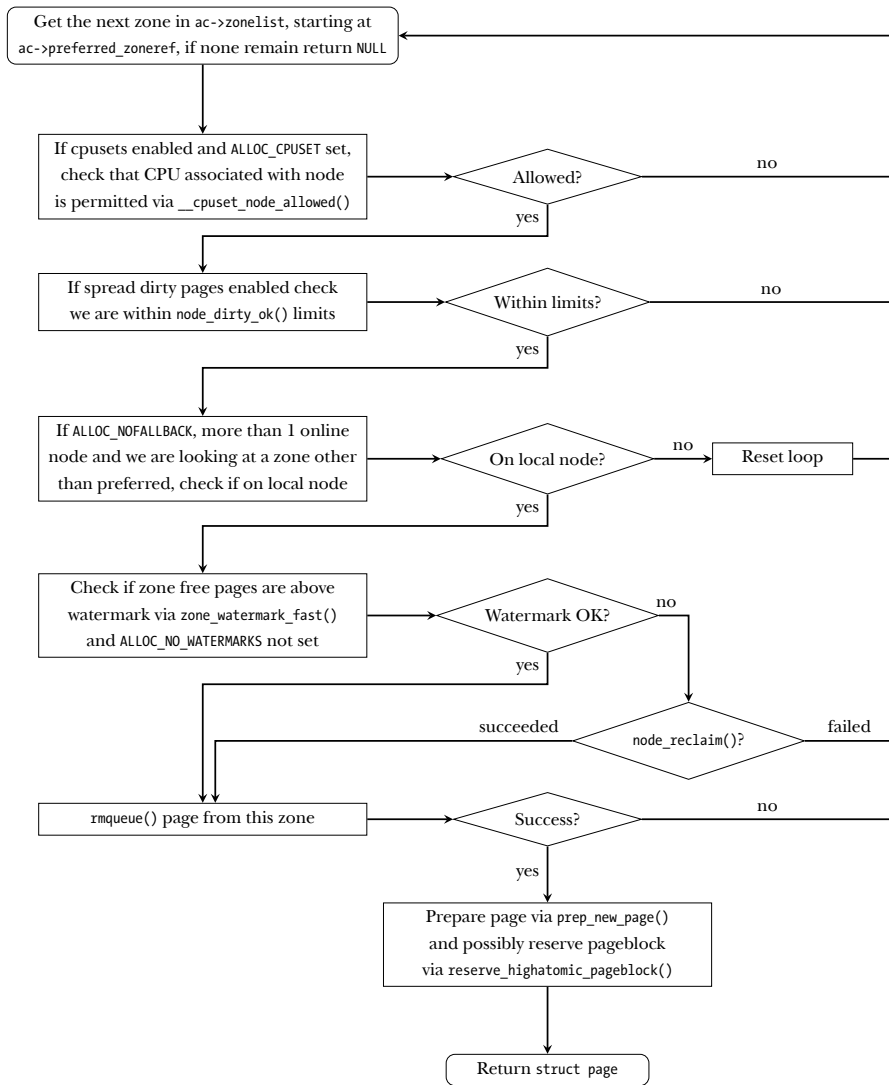


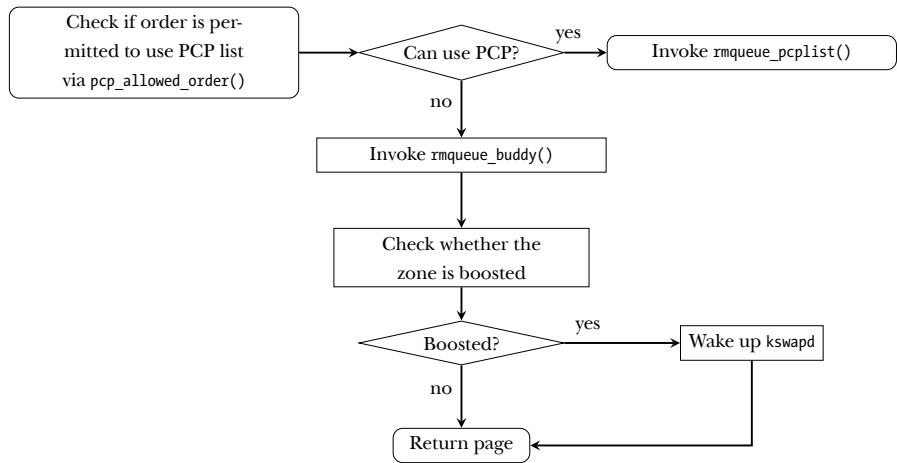
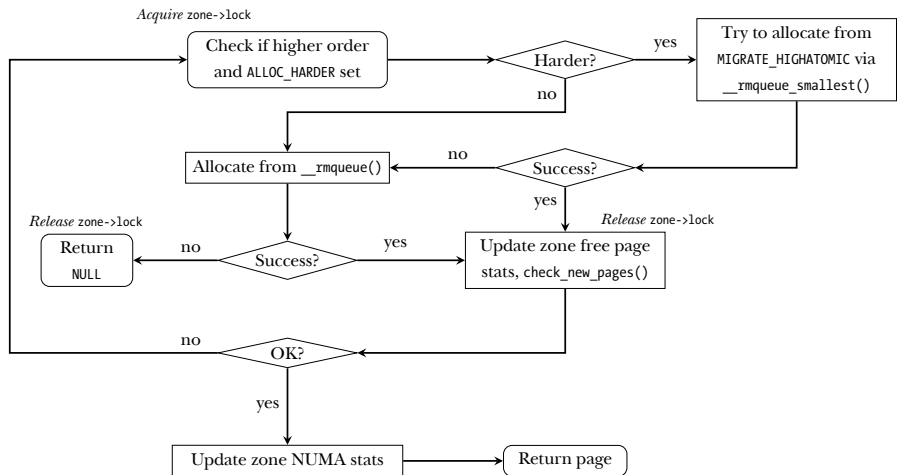
Figure 2-14: Visualisation of `__alloc_pages()`

Initialisation of flags consists of applying the global GFP mask `gfp_allowed_mask` (typically only applicable during boot), obtaining any process-specific memalloc flag settings via `current_gfp_context()`, invoking `prepare_alloc_pages()` to set up an `struct alloc_context` object and possibly modify the alloc flags before finally applying `alloc_flags_nofragment()` to avoid fragmentation in `ZONE_DMA32`.

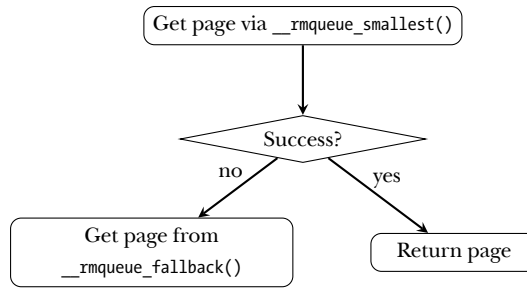
The net result of these flags and their initial values are described on the previous page.

Note that we do not explore `__alloc_pages_slowpath()` here but rather examine this in Listing 11-1 in Chapter 11 on reclaim, as this is where direct reclaim takes place.

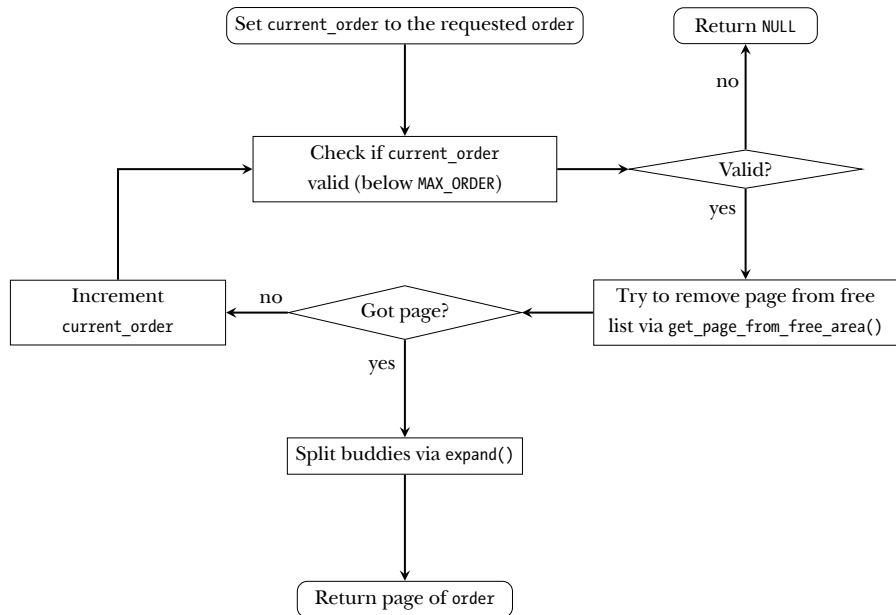
Figure 2-15: Visualisation of `get_page_from_freelist()`

Figure 2-16: Visualisation of `rmqueue()`Figure 2-17: Visualisation of `rmqueue_buddy()`

`__rmqueue()` is a simple wrapper around `__rmqueue_smallest()` with the added logic of falling back to `__rmqueue_fallback()` to steal from another migrate type if this fails as shown in Figure 2-18.

Figure 2-18: Visualisation of `__rmqueue()`

`rmqueue_pcplist()` is a simple mechanism for retrieving a page from the PCP lists so there is not much benefit in visualising it. Let's examine `__rmqueue_smallest()` as this is where pages are actually retrieved from free lists as shown in Figure 2-19.

Figure 2-19: Visualisation of `__rmqueue_smallest()`

We visualise `expand()`, whose logic is rather simple, in Figure ??.

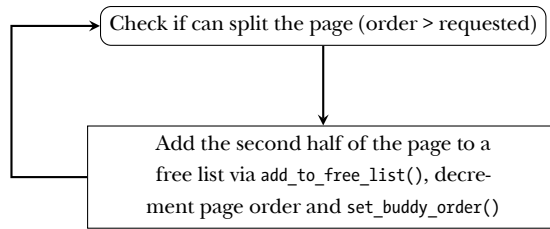
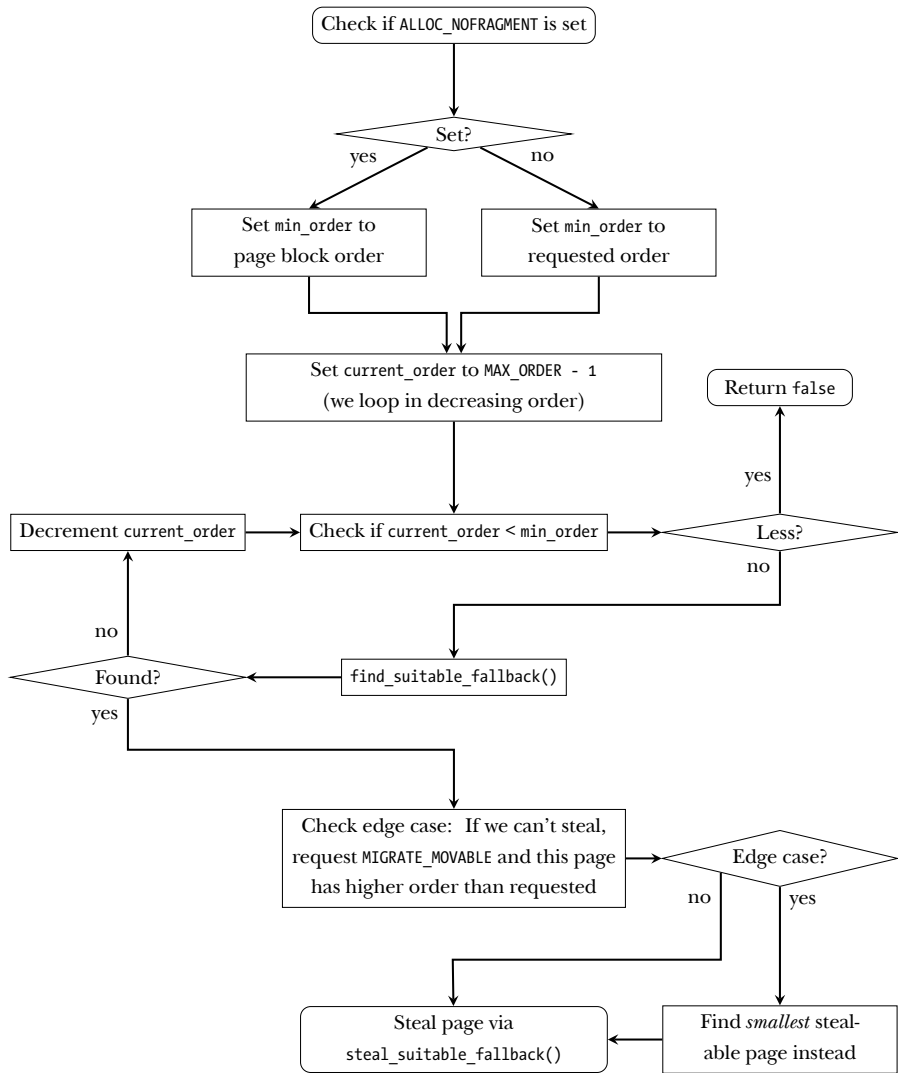


Figure 2-20: Visualisation of `expand()`

Finally, let's examine `__rmqueue_fallback()`, this is invoked when no page of the required migrate type is available and a page needs to be 'stolen' from another migrate type as shown in Figure 2-21.

Figure 2-21: Visualisation of `__rmqueue_fallback()`

A key point to note here is that under ordinary circumstances the *largest possible* page is ‘stolen’ from another migrate type. We can grab a page at a page block size or more than we simply convert the entire page block with zero fragmentation.

2.8.2 `__alloc_pages()`

Examining the actual function (stripping out of scope memcg and trace point logic) as shown in Listing 2-52.

```

5510 /*
5511  * This is the 'heart' of the zoned buddy allocator.

```

```

5512  */
5513  struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
5514                             nodemask_t *nodemask)
5515  {
5516      struct page *page;
5517      unsigned int alloc_flags = ALLOC_WMARK_LOW;
5518      gfp_t alloc_gfp; /* The gfp_t that was actually used for allocation */
5519      struct alloc_context ac = { };
5520
5521      /*
5522       * There are several places where we assume that the order value is
5523       * sane
5524       * so bail out early if the request is out of bound.
5525       */
5526      if (WARN_ON_ONCE_GFP(order >= MAX_ORDER, gfp))
5527          return NULL;
5528
5529      gfp &= gfp_allowed_mask;
5530      /*
5531       * Apply scoped allocation constraints. This is mainly about GFP_NOFS
5532       * resp. GFP_NOIO which has to be inherited for all allocation
5533       requests
5534       * from a particular context which has been marked by
5535       * memalloc_nofs_io_{save,restore}. And PF_MEMALLOC_PIN which
5536       ensures
5537       * movable zones are not used during allocation.
5538       */
5539      gfp = current_gfp_context(gfp);
5540      alloc_gfp = gfp;
5541      if (!prepare_alloc_pages(gfp, order, preferred_nid, nodemask, &ac,
5542                              &alloc_gfp, &alloc_flags))
5543          return NULL;
5544
5545      /*
5546       * Forbid the first pass from falling back to types that fragment
5547       * memory until all local zones are considered.
5548       */
5549      alloc_flags |= alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp)
5550      ;
5551
5552      /* First allocation attempt */
5553      page = get_page_from_freelist(alloc_gfp, order, alloc_flags, &ac);
5554      if (likely(page))
5555          goto out;
5556
5557      alloc_gfp = gfp;
5558      ac.spread_dirty_pages = false;

```

```

5555
5556     /*
5557      * Restore the original nodemask if it was potentially replaced with
5558      * &cpuset_current_mems_allowed to optimize the fast-path attempt.
5559      */
5560     ac.nodemask = nodemask;
5561
5562     page = __alloc_pages_slowpath(alloc_gfp, order, &ac);
5563
5564 out:
5565     :
5573     return page;
5574 }

```

Listing 2-52: mm/page_alloc.c: Simplified `__alloc_pages()`

The top level logic is as follows:

1. Check to ensure the specified order is within the permitted range (0 to `MAX_ORDER - 1`).
2. Mask gfp against `gfp_allowed_mask` – this is only relevant during early boot and power state transition.
3. Invoke `current_gfp_context()` to adjust GFP flags according to whether the current thread has memalloc flags set. These are set when the thread is performing allocations in a context where certain types of allocations cannot be permitted and are set in the thread's flags field:
 - `PF_MEMALLOC_NOIO` – This clears `__GFP_IO` and `__GFP_FS` meaning no I/O or low-level filesystem operations can occur during allocation. This is important when filesystem locks are held during an allocation to avoid deadlock. Set by `memalloc_noio_save()` and cleared by `memalloc_noio_restore()`.
 - `PF_MEMALLOC_NOFS` – Similar to `PF_MEMALLOC_NOIO` but a weaker constraint – This clears `__GFP_FS` meaning no low-level filesystem operations can occur during allocation. Set by `memalloc_nofs_save()` and cleared by `memalloc_nofs_restore()`.
 - `PF_MEMALLOC_PIN` – This prevents allocated memory from being movable by clearing the `__GFP_MOVABLE` GFP flag. Set by `memalloc_pin_save()` and cleared by `memalloc_pin_restore()`. This is currently only used in `__gup_longterm_locked()`.
4. After taking a copy of the gfp flags in `alloc_gfp` invoke `prepare_alloc_pages()` which sets `alloc_gfp`, `alloc_flags` and the struct `alloc_context` object `ac`. These three values parameterise the *fast path* allocation attempt. Under certain circumstances this step can fail.
5. Update `alloc_flags` to set `ALLOC_NOFRAGMENT` in order to avoid memory fragmentation between zones if `alloc_flags_nofragment()` deems this necessary.

6. Attempt fast path allocation via `get_page_from_freelist()`, parameterised by `alloc_gfp`, `order`, `alloc_flags` and `ac`.
7. If and only if this fails – restore the original `gfp` flags to `alloc_gfp`, disable dirty page spreading, reset the mask of allowable nodes and invoke `__alloc_pages_slowpath()` to perform *slow path* allocation. The bulk of this logic forms part of *direct* memory reclaim, which will be examined in the reclaim chapter.

For most allocations we can simplify this to – generate `struct alloc_context` based on input parameters, try fast path, if failed, try slow path.

Let's examine `struct alloc_context` as shown in Listing 2-53.

```

196 /*
197  * Structure for holding the mostly immutable allocation parameters passed
198  * between functions involved in allocations, including the alloc_pages*
199  * family of functions.
200  *
201  * nodemask, migratetype and highest_zoneidx are initialized only once in
202  * __alloc_pages() and then never change.
203  *
204  * zonelist, preferred_zone and highest_zoneidx are set first in
205  * __alloc_pages() for the fast path, and might be later changed
206  * in __alloc_pages_slowpath(). All other functions pass the whole structure
207  * by a const pointer.
208  */
209 struct alloc_context {
210     struct zonelist *zonelist;
211     nodemask_t *nodemask;
212     struct zoneref *preferred_zoneref;
213     int migratetype;
214
215     /*
216      * highest_zoneidx represents highest usable zone index of
217      * the allocation request. Due to the nature of the zone,
218      * memory on lower zone than the highest_zoneidx will be
219      * protected by lowmem_reserve[highest_zoneidx].
220      *
221      * highest_zoneidx is also used by reclaim/compaction to limit
222      * the target zone since higher zone than this index cannot be
223      * usable for this allocation request.
224      */
225     enum zone_type highest_zoneidx;
226     bool spread_dirty_pages;
227 };

```

Listing 2-53: `mm/internal.h: struct alloc_context`

This data structure is threaded through all allocation paths. Examining each field:

- **zonelist** – A struct `zonelist` is a list of zones that will be tried for allocation. The zone list used during allocation is the one which determine which zones will be tried. There are two possible zonelists – all available zones, and those available only on the preferred `nid` node. In fast-path allocation this is determine via `node_zonelist()`, selecting the `ZONELIST_FALLBACK` zonelist (i.e. the one containing the zones of *all* nodes) by default or selecting the `ZONELIST_NOFALLBACK` (i.e. the one containing only the zones of the selected node) if the `__GFP_THISNODE` flag has been specified.
- **nodemask** – Simply passes through the `nodemask` parameter passed to `__alloc_pages()`. If the `cgroups cpusets` feature is enabled and no `nodemask` is provided in task context (i.e. not interrupt context) then this to the `cpuset_current_mems_allowed` global.
- **highest_zoneidx** – The index of the highest zone *type* (e.g. `ZONE_DMA32`) that will be examined as determined by GFP flags via `gfp_zone()`. This is determined via the physical address zone modifiers described in section 2.6.1. This is the *requested zone* for the allocation – i.e. the zone the allocator was intended to be taken from.
- **preferred_zoneref** – This field is set last and determines the zone which we first look at when iterating through each zone. This uses `first_zones_zonelist()` to find the first zone in `ac->zonelist` that is at or below `ac->highest_zoneidx` which exists on `ac->nodemask`.
- **migratetype** – The `migratetype` of the allocation as determined by `gfp_migratetype()` determined by GFP page mobility flags described in section 2.6.2. In early boot there is a global variable which may also impact mobility `—page_group_by_mobility_disabled—` but for ordinary allocations this isn't a consideration..
- **spread_dirty_pages** – Set true if `__GFP_WRITE` is set.

We can see this in action in `prepare_alloc_pages()` (excluding out-of-scope CMA code) as shown in Listing 2-54.

```

5297 static inline bool prepare_alloc_pages(gfp_t gfp_mask, unsigned int order,
5298                                     int preferred_nid, nodemask_t *nodemask,
5299                                     struct alloc_context *ac, gfp_t *alloc_gfp,
5300                                     unsigned int *alloc_flags)
5301 {
5302     ac->highest_zoneidx = gfp_zone(gfp_mask);
5303     ac->zonelist = node_zonelist(preferred_nid, gfp_mask);
5304     ac->nodemask = nodemask;
5305     ac->migratetype = gfp_migratetype(gfp_mask);
5306
5307     if (cpusets_enabled()) {
5308         *alloc_gfp |= __GFP_HARDWALL;
```

```

5309         /*
5310          * When we are in the interrupt context, it is irrelevant
5311          * to the current task context. It means that any node ok.
5312          */
5313         if (in_task() && !ac->nodemask)
5314             ac->nodemask = &cpuset_current_mems_allowed;
5315         else
5316             *alloc_flags |= ALLOC_CPUSET;
5317     }
5318
5319     might_alloc(gfp_mask);
5320
5321     if (should_fail_alloc_page(gfp_mask, order))
5322         return false;
5323     :
5324
5325     /* Dirty zone balancing only done in the fast path */
5326     ac->spread_dirty_pages = (gfp_mask & __GFP_WRITE);
5327
5328     /*
5329      * The preferred zone is used for statistics but crucially it is
5330      * also used as the starting point for the zonelist iterator. It
5331      * may get reset for allocations that ignore memory policies.
5332      */
5333     ac->preferred_zoneref = first_zones_zonelist(ac->zonelist,
5334                                                 ac->highest_zoneidx, ac->nodemask);
5335
5336     return true;
5337 }
5338

```

Listing 2-54: mm/page_alloc.c: *prepare_alloc_pages()*

The logic largely follows the description of struct `alloc_context` fields above, with some nuances – if the cgroup cpusets are enabled, then `__GFP_HARDWALL` is set in the GFP flags and `ALLOC_CPUSET` set in allocation flags (see later for more on allocation flags) if not in task context or a nodemask is specified.

Additionally, `should_fail_alloc_page()` is called to determine whether a page fault should fail under `CONFIG_FAIL_PAGE_ALLOC` (ultimately invoking `__should_fail_alloc_page()`). This is enabled if fault injection is enabled (via `CONFIG_FAULT_INJECTION`) but this is beyond the scope of the book (see the *fault injection documentation* for more details).

Before invoking `prepare_alloc_pages()`, `__alloc_pages()` defaults `alloc_flags` to `ALLOC_WMARK_LOW`. `prepare_alloc_pages()` also possibly sets the `ALLOC_CPUSET` flag.

These are additional flags beyond GFP flags which determine how allocations should be performed – let’s examine them now (all declared in `mm/internal.h`).

Firstly, considering the watermark flags:

- `ALLOC_WMARK_MIN` – Only allocate if the zone watermark is at or above the *minimum* watermark (taking into account reserves, and checking that at least one page exists if allocating higher order).
- `ALLOC_WMARK_LOW` – Same as above, for the *low* watermark. Default behaviour.
- `ALLOC_WMARK_HIGH` – Same as above, for the *high* watermark.
- `ALLOC_NO_WATERMARKS` – Ignore watermarks on allocation.

These flags are kept in sync with the equivalent enum entries in `enum zone_watermarks` so the index can simply be extracted by applying the `ALLOC_WMARK_MASK`. This is used for example in `get_page_from_freelist()`, (the fast-path allocation logic).

The remaining flags are set at higher bits and can be combined:

- `ALLOC_OOM` – Indicates that the task is being targeted as an Out-Of-Memory (OOM) victim and watermarks should be reduced to permit allocations as the process is about to die anyway.
- `ALLOC_HARDER` – Try to allocate harder than usual. Reduces watermarks to permit allocation even in low memory scenarios.
- `ALLOC_HIGH` – Indicates high-priority allocation, which further reduces watermarks. Implied by `__GFP_HIGH`.
- `ALLOC_CPUSET` – cpuset-specific allocation flag out of scope for this book.
- `ALLOC_CMA` – Indicates a CMA allocation, out of scope for this book.
- `ALLOC_NOFRAGMENT` – Indicates that allocations should not steal pages from other migratetype pageblocks.
- `ALLOC_KSWAPD` – Indicates that `kswapd` processes should be woken to perform indirect reclaim. These are kernel threads that work to reclaim free pages back to zones, covered further in the reclaim chapter.

Let's examine `get_page_from_freelist()` (simplified to remove some comments and deferred struct page initialisation). We can divide the function up into three parts. Firstly the initial portion and the beginning of the zone loop as shown in Listing 2-55.

```

4165 static struct page *
4166 get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
4167                        const struct alloc_context *ac
4168                        )
4169 {
4170     struct zoneref *z;
4171     struct zone *zone;
4172     struct pglist_data *last_pgdat = NULL;
4173     bool last_pgdat_dirty_ok = false;
4174     bool no_fallback;
4175     retry:

```



```

4176      /*
4177       * Scan zonelist, looking for a zone with enough free.
4178       * See also __cpuset_node_allowed() comment in kernel/cgroup/cpuset.c.
4179       */
4180      no_fallback = alloc_flags & ALLOC_NOFRAGMENT;
4181      z = ac->preferred_zoneref;
4182      for_next_zone_zonelist_nodemask(zone, z, ac->highest_zoneidx,
4183                                     ac->nodemask) {

```

Listing 2-55: mm/page_alloc.c: Preface of get_page_from_freelist()

We are iterating through each zone specified by the node mask to find a candidate from which we can obtain pages. The loop code firstly examines edge cases as shown in Listing 2-56.

```

4184      struct page *page;
4185      unsigned long mark;
4186
4187      if (cpusets_enabled() &&
4188          (alloc_flags & ALLOC_CPUSET) &&
4189          !__cpuset_zone_allowed(zone, gfp_mask))
4190          continue;
4191      :
4192
4193      if (ac->spread_dirty_pages) {
4194          if (last_pgdat != zone->zone_pgdat) {
4195              last_pgdat = zone->zone_pgdat;
4196              last_pgdat_dirty_ok = node_dirty_ok(zone->
4197              zone_pgdat);
4198          }
4199
4200          if (!last_pgdat_dirty_ok)
4201              continue;
4202      }
4203
4204      if (no_fallback && nr_online_nodes > 1 &&
4205          zone != ac->preferred_zoneref->zone) {
4206          int local_nid;
4207
4208          /*
4209           * If moving to a remote node, retry but allow
4210           * fragmenting fallbacks. Locality is more important
4211           * than fragmentation avoidance.
4212           */
4213          local_nid = zone_to_nid(ac->preferred_zoneref->zone);
4214          if (zone_to_nid(zone) != local_nid) {
4215              alloc_flags &= ~ALLOC_NOFRAGMENT;
4216              goto retry;
4217          }
4218      }
4219  }

```

Listing 2-56: mm/page_alloc.c: *get_page_from_freelist()* edge cases

Here we perform a few checks:

- If `ALLOC_CPUSET` is set and the CPU is not allowed under the cgroup cpuset functionality then we reject the zone.
- If the `spread_dirty_pages` allocation flag has been set (via the `__GFP_WRITE` GFP flag having been passed), we check each new node to see if it has exceeded dirty page limits via `node_dirty_ok()`. If it has then we do not allocate from it as the `__GFP_WRITE` hints that this will be a file-backed allocation and thus we would only be exceeding the dirty limit by even more.
- If the `ALLOC_NOFRAGMENT` flag has been set and we are not on the preferred node, then we clear this flag and retry the operation.

Finally we perform the vital step of the zone watermark check and the actual allocation of the requested page as shown in Listing 2-57.

```

4237             if (!zone_watermark_fast(zone, order, mark,
4238                                     ac->highest_zoneidx, alloc_flags,
4239                                     gfp_mask)) {
4240                 :
4241             }
4242
4243 try_this_zone:
4244     page = rmqueue(ac->preferred_zoneref->zone, zone, order,
4245                  gfp_mask, alloc_flags, ac->migratetype);
4246     if (page) {
4247         prep_new_page(page, order, gfp_mask, alloc_flags);
4248
4249         /*
4250          * If this is a high-order atomic allocation then
4251          *
4252          * if the pageblock should be reserved for the future
4253          */
4254         if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
4255             reserve_highatomic_pageblock(page, zone, order
4256 );
4257
4258     return page;
4259 } else {
4260     :
4261 }
4262 }
4263
4264 /*
4265  * It's possible on a UMA machine to get through all zones that are

```

```

4306         * fragmented. If avoiding fragmentation, reset and try again.
4307         */
4308         if (no_fallback) {
4309             alloc_flags &= ~ALLOC_NOFRAGMENT;
4310             goto retry;
4311         }
4312
4313         return NULL;
4314     }

```

Listing 2-57: mm/page_alloc.c: get_page_from_freelist() watermark check and page retrieval

The key steps here are the fast-path zone watermark check via `zone_watermark_fast()` (we exclude the ‘slow’ watermark check case if this function fails, as this is covered in the reclaim chapter) and the actual retrieval of pages via `rmqueue()` with the page being prepared via `prep_new_page()`. We will examine each of these in turn.

In order to check whether the zone watermark has been exceeded we use `zone_watermark_fast()` as shown in Listing 2-58.

```

4039 static inline bool zone_watermark_fast(struct zone *z, unsigned int order,
4040                                       unsigned long mark, int highest_zoneidx,
4041                                       unsigned int alloc_flags, gfp_t gfp_mask)
4042 {
4043     long free_pages;
4044
4045     free_pages = zone_page_state(z, NR_FREE_PAGES);
4046
4047     /*
4048      * Fast check for order-0 only. If this fails then the reserves
4049      * need to be calculated.
4050      */
4051     if (!order) {
4052         long usable_free;
4053         long reserved;
4054
4055         usable_free = free_pages;
4056         reserved = __zone_watermark_unusable_free(z, 0, alloc_flags);
4057
4058         /* reserved may over estimate high-atomic reserves. */
4059         usable_free -= min(usable_free, reserved);
4060         if (usable_free > mark + z->lowmem_reserve[highest_zoneidx])
4061             return true;
4062     }
4063
4064     if (__zone_watermark_ok(z, order, mark, highest_zoneidx, alloc_flags,
4065                             free_pages))
4066         return true;

```

```

4067      /*
4068       * Ignore watermark boosting for GFP_ATOMIC order-0 allocations
4069       * when checking the min watermark. The min watermark is the
4070       * point where boosting is ignored so that kswapd is woken up
4071       * when below the low watermark.
4072       */
4073      if (unlikely(!order && (gfp_mask & __GFP_ATOMIC) && z->watermark_boost
4074                  && ((alloc_flags & ALLOC_WMARK_MASK) == WMARK_MIN))) {
4075          mark = z->watermark[WMARK_MIN];
4076          return __zone_watermark_ok(z, order, mark, highest_zoneidx,
4077                                     alloc_flags, free_pages);
4078      }
4079
4080      return false;
4081  }

```

Listing 2-58: mm/page_alloc.c: `zone_watermark_fast()`

The `__zone_watermark_unusable_free()` function returns how many base pages are designated ‘unusable’ within the zone – this defaults to the page order minus 1 (or 0 if order-0). This ensures that we always leave at least one page of the specified order in the zone at all times except for order-0. If neither `ALLOC_HARDER` or `ALLOC_OOM` are set also includes the pages kept reserved for high atomic page blocks.

In the fast path (order-0 pages) this is all we need to determine whether the zone watermark check has passed (also taking into account lowmem reserves as discussed in section 2.4 above).

For higher-order allocators or cases where this fast check failed, we invoke the slow path function `__zone_watermark_ok()` (Stripping out-of-scope CMA code) as shown in Listing 2-59.

```

3968  bool __zone_watermark_ok(struct zone *z, unsigned int order, unsigned long
3969                          mark,
3970                          int highest_zoneidx, unsigned int alloc_flags,
3971                          long free_pages)
3972  {
3973      long min = mark;
3974      int o;
3975      const bool alloc_harder = (alloc_flags & (ALLOC_HARDER|ALLOC_OOM));
3976
3977      /* free_pages may go negative - that's OK */
3978      free_pages -= __zone_watermark_unusable_free(z, order, alloc_flags);
3979
3980      if (alloc_flags & ALLOC_HIGH)
3981          min -= min / 2;
3982
3983      if (unlikely(alloc_harder)) {
3984          /*
3985           * OOM victims can try even harder than normal ALLOC_HARDER

```

```

3985         * users on the grounds that it's definitely going to be in
3986         * the exit path shortly and free memory. Any allocation it
3987         * makes during the free path will be small and short-lived.
3988         */
3989         if (alloc_flags & ALLOC_OOM)
3990             min -= min / 2;
3991         else
3992             min -= min / 4;
3993     }
3994
3995     /*
3996     * Check watermarks for an order-0 allocation request. If these
3997     * are not met, then a high-order request also cannot go ahead
3998     * even if a suitable page happened to be free.
3999     */
4000     if (free_pages <= min + z->lowmem_reserve[highest_zoneidx])
4001         return false;
4002
4003     /* If this is an order-0 request then the watermark is fine */
4004     if (!order)
4005         return true;
4006
4007     /* For a high-order request, check at least one suitable page is free
4008     */
4009     for (o = order; o < MAX_ORDER; o++) {
4010         struct free_area *area = &z->free_area[o];
4011         int mt;
4012
4013         if (!area->nr_free)
4014             continue;
4015
4016         for (mt = 0; mt < MIGRATE_PCPTYPES; mt++) {
4017             if (!free_area_empty(area, mt))
4018                 return true;
4019         }
4020         :
4021         if (alloc_harder && !free_area_empty(area, MIGRATE_HIGHATOMIC)
4022     )
4023         return true;
4024     }
4025     return false;
4026 }

```

Listing 2-59: mm/page_alloc.c: `__zone_watermark_ok()`

As before, `__zone_watermark_unusable_free()` is used to take into account reserved pages.

After this, we examine the impact of allocation flags that reduce the minimum required number of free pages (ALLOC_OOM supersedes ALLOC_HARDER so if both are set only ALLOC_OOM takes effect.):

Table 2-7: Alloc flag watermark impact

ALLOC_HIGH	ALLOC_OOM	ALLOC_HARDER	Δ Req'd pages
		•	–25%
•			–50%
	•		–50%
•		•	–62.5%
•	•		–75%

The motivation for reducing watermark requirements even further under OOM conditions is that a process that is a target for OOM (and thus allocating with ALLOC_OOM set) will soon be killed so allowing it to allocate some more memory is immaterial – the memory will be freed very soon.

Low mem reserves are taken into account (as discussed in section 2.4). For order-0 pages these checks are sufficient.

For higher-order pages additional checks are performed – at least one page at or above the required order must exist for PCP-usable migrate types (MIGRATE_UNMOVABLE, MIGRATE_MOVABLE, or MIGRATE_RECLAIMABLE). If either ALLOC_HARDER or ALLOC_OOM are set, then the MIGRATE_HIGHATOMIC migratetype can also be used.

Looking back to `zone_watermark_fast()` we cover a very specific edge case – for order-0 pages, allocated via `__GFP_ATOMIC`, with zone watermark boost and allocation flags set to `ALLOC_WMARK_MIN` we invoke `__zone_watermark_ok()` again with the watermark set to `WMARK_MIN`, i.e. removing the watermark boost altogether under these specific circumstances.

2.8.3 *rmqueue()*

Once we have established the zone to allocate from, we need to actually perform the allocation from a freelist. This is achieved via `rmqueue()` (stripping out-of-scope CMA code) as shown in Listing 2-60.

```

3820 static inline
3821 struct page *rmqueue(struct zone *preferred_zone,
3822                     struct zone *zone, unsigned int order,
3823                     gfp_t gfp_flags, unsigned int alloc_flags,
3824                     int migratetype)
3825 {
3826     struct page *page;
3827
3828     /*
3829      * We most definitely don't want callers attempting to
3830      * allocate greater than order-1 page units with __GFP_NOFAIL.
3831      */
3832     WARN_ON_ONCE((gfp_flags & __GFP_NOFAIL) && (order > 1));

```

```

3833
3834     if (likely(pcp_allowed_order(order))) {
3835         :
3841         page = rmqueue_pcplist(preferred_zone, zone, order,
3842                                gfp_flags, migratetype, alloc_flags);
3843         if (likely(page))
3844             goto out;
3845         :
3846     }
3847
3848     page = rmqueue_buddy(preferred_zone, zone, order, alloc_flags,
3849                          migratetype);
3850
3851 out:
3852     /* Separate test+clear to avoid unnecessary atomics */
3853     if (unlikely(test_bit(ZONE_BOOSTED_WATERMARK, &zone->flags))) {
3854         clear_bit(ZONE_BOOSTED_WATERMARK, &zone->flags);
3855         wakeup_kswapd(zone, 0, 0, zone_idx(zone));
3856     }
3857
3858     VM_BUG_ON_PAGE(page && bad_range(zone, page), page);
3859     return page;
3860 }

```

Listing 2-60: mm/page_alloc.c: `rmqueue()`

The logic is as follows:

1. If the order of the allocation is such that Per-CPU-Pages (PCPs) can be used as indicated by `pcp_allowed_order()` (see section 2.7.3 for more details), then we allocate from these using `rmqueue_pcplist()`. This avoids the need for acquiring a zone lock.
2. If we are unable to retrieve a page from the PCP, invoke `rmqueue_buddy()` to retrieve the page directly from the buddy allocator.
3. If the watermark for the zone has been boosted, then `kswapd` is woken up to perform indirect reclaim. We go into detail about this in the reclaim chapter. This boosting occurs when migrate type fragmentation occurs (see section 2.8.6) in order to reduce the impact of fragmented page blocks.

2.8.4 `rmqueue_buddy()`

We examine `rmqueue_buddy()` in Listing 2-61 (eliding statistical updates).

```

3699 static __always_inline
3700 struct page *rmqueue_buddy(struct zone *preferred_zone, struct zone *zone,
3701                            unsigned int order, unsigned int alloc_flags,
3702                            int migratetype)
3703 {

```

```

3704     struct page *page;
3705     unsigned long flags;
3706
3707     do {
3708         page = NULL;
3709         spin_lock_irqsave(&zone->lock, flags);
3710         /*
3711          * order-0 request can reach here when the pcplist is skipped
3712          * due to non-CMA allocation context. HIGHATOMIC area is
3713          * reserved for high-order atomic allocation, so order-0
3714          * request should skip it.
3715          */
3716         if (order > 0 && alloc_flags & ALLOC_HARDER)
3717             page = __rmqueue_smallest(zone, order,
MIGRATE_HIGHATOMIC);
3718         if (!page) {
3719             page = __rmqueue(zone, order, migratetype, alloc_flags
);
3720             if (!page) {
3721                 spin_unlock_irqrestore(&zone->lock, flags);
3722                 return NULL;
3723             }
3724         }
3725         :
3726         spin_unlock_irqrestore(&zone->lock, flags);
3727     } while (check_new_pages(page, order));
3728     :
3729     return page;
3730 }

```

Listing 2-61: mm/page_alloc.c: *rmqueue_buddy()*

Here we enter a loop where we either allocate a page from MIGRATE_HIGHATOMIC if ALLOC_HARDER is set via *__rmqueue_smallest()* or via *__rmqueue()* if either the former is not the case or it fails. It then updates zone statistics. The loop invokes *check_new_pages()* which ensures the page returned from the free list is correctly set up as a free page and prints an alert in the kernel ring buffer. This should nearly never loop (typically this would occur if a use-after-free bug had arisen in the kernel).

2.8.5 *rmqueue_pcplist()*

Examining *rmqueue_pcplist()* as shown in Listing 2-62.

```

3777 /* Lock and remove page from the per-cpu list */
3778 static struct page *rmqueue_pcplist(struct zone *preferred_zone,
3779                                     struct zone *zone, unsigned int order,
3780                                     gfp_t gfp_flags, int migratetype,
3781                                     unsigned int alloc_flags)

```



```

3782 {
3783     struct per_cpu_pages *pcp;
3784     struct list_head *list;
3785     struct page *page;
3786     unsigned long flags;
3787     unsigned long __maybe_unused UP_flags;
3788
3789     /*
3790      * spin_trylock may fail due to a parallel drain. In the future, the
3791      * trylock will also protect against IRQ reentrancy.
3792      */
3793     pcp_trylock_prepare(UP_flags);
3794     pcp = pcp_spin_trylock_irqsave(zone->per_cpu_pageset, flags);
3795     if (!pcp) {
3796         pcp_trylock_finish(UP_flags);
3797         return NULL;
3798     }
3799
3800     /*
3801      * On allocation, reduce the number of pages that are batch freed.
3802      * See nr_pcp_free() where free_factor is increased for subsequent
3803      * frees.
3804      */
3805     pcp->free_factor >>= 1;
3806     list = &pcp->lists[order_to_pindex(migratetype, order)];
3807     page = __rmqueue_pcp(list, zone, order, migratetype, alloc_flags, pcp,
3808                          list);
3809     pcp_spin_unlock_irqrestore(pcp, flags);
3810     pcp_trylock_finish(UP_flags);
3811     if (page) {
3812         __count_zid_vm_events(PGALLOC, page_zonenum(page), 1);
3813         zone_statistics(preferred_zone, zone, 1);
3814     }
3815     return page;
3816 }

```

Listing 2-62: mm/page_alloc.c: `rmqueue_pcp()`

The logic is as follows:

1. Setup PCP lock via `pcp_trylock_prepare()` (disabling IRQs to avoid reentry), acquiring the PCP lock via `pcp_spin_trylock_irqsave()`.
2. Retrieve the `struct per_cpu_pages` object specific to this zone and CPU. If lock cannot be acquired fails, restore IRQs via `pcp_trylock_finish()` and return `NULL`.
3. Divide the ‘free factor’ by 2. This value is used to scale the number of PCP pages that are freed back to ordinary free lists. For each page that

is freed *to* a PCP list the value is doubled, for each page that is allocated *from* a PCP list it is halved. As we are allocating here, we halve.

4. Determine the correct PCP list to retrieve a page from using `order_to_pindex()` which simply determines the array index of that a PCP list for specific order and migrate type.
5. Actually retrieve the page via `__rmqueue_pcplist()`.
6. Release the PCP lock via `pcp_spin_unlock_irqrestore()` and re-enable IRQs via `pcp_trylock_finish()`.
7. Update some statistics if the page was correctly retrieved and return the result.

Examining `__rmqueue_pcplist()` as shown in Listing 2-63.

```

3736 /* Remove page from the per-cpu list, caller must protect the list */
3737 static inline
3738 struct page *__rmqueue_pcplist(struct zone *zone, unsigned int order,
3739                               int migratetype,
3740                               unsigned int alloc_flags,
3741                               struct per_cpu_pages *pcp,
3742                               struct list_head *list)
3743 {
3744     struct page *page;
3745
3746     do {
3747         if (list_empty(list)) {
3748             int batch = READ_ONCE(pcp->batch);
3749             int allocated;
3750
3751             /*
3752              * Scale batch relative to order if batch implies
3753              * free pages can be stored on the PCP. Batch can
3754              * be 1 for small zones or for boot pagesets which
3755              * should never store free pages as the pages may
3756              * belong to arbitrary zones.
3757              */
3758             if (batch > 1)
3759                 batch = max(batch >> order, 2);
3760             allocated = rmqueue_bulk(zone, order,
3761                                     batch, list,
3762                                     migratetype, alloc_flags);
3763
3764             pcp->count += allocated << order;
3765             if (unlikely(list_empty(list)))
3766                 return NULL;
3767         }
3768
3769         page = list_first_entry(list, struct page, pcpl_list);

```

```

3770         list_del(&page->pcp_list);
3771         pcp->count -= 1 << order;
3772     } while (check_new_pcp(page, order));
3773
3774     return page;
3775 }

```

Listing 2-63: mm/page_alloc.c: `__rmqueue_pcplist()`

The logic of this is as follows:

1. If the PCP list is empty, refill it using `rmqueue_bulk()`. We refill based on the `per_cpu_pages->batch` field, refilling $\max(\lfloor \frac{\text{batch}}{2^{\text{order}}} \rfloor, 2)$ of pages of order equal to specified order. If this fails to allocate pages, we abort and return NULL.
2. Retrieve the first entry from the PCP list, delete it from the list, update the stats.
3. Check the page via `check_new_pcp()` if `CONFIG_DEBUG_PAGEALLOC` is enabled. If it fails, repeat page retrieval. If success, return page.

Examining `__rmqueue()` (out-of-scope CMA logic removed for clarity) as shown in Listing 2-64.

```

3079 static __always_inline struct page *
3080 __rmqueue(struct zone *zone, unsigned int order, int migratetype,
3081           unsigned int alloc_flags)
3082 {
3083     struct page *page;
3084     :
3099 retry:
3100     page = __rmqueue_smallest(zone, order, migratetype);
3101     if (unlikely(!page)) {
3102         :
3105         if (!page && __rmqueue_fallback(zone, order, migratetype,
3106                                         alloc_flags))
3107             goto retry;
3108     }
3109     return page;
3110 }

```

Listing 2-64: mm/page_alloc.c: `__rmqueue()`

This is simply a wrapper around `__rmqueue_smallest()` with fallback page allocation provided via `__rmqueue_fallback()`. The fallback will try to move pages from a fallback migrate type to the migrate type we require, then retries the operation.

2.8.6 Migrate type fallback

Examining `__rmqueue_fallback()` as shown in Listing 2-65.

```

2987 /*
2988  * Try finding a free buddy page on the fallback list and put it on the free
2989  * list of requested migratetype, possibly along with other pages from the
2990  * same
2991  * block, depending on fragmentation avoidance heuristics. Returns true if
2992  * fallback was found so that __rmqueue_smallest() can grab it.
2993  *
2994  * The use of signed ints for order and current_order is a deliberate
2995  * deviation from the rest of this file, to make the for loop
2996  * condition simpler.
2997  */
2998 static __always_inline bool
2999 __rmqueue_fallback(struct zone *zone, int order, int start_migratetype,
3000                  unsigned int alloc_flags)
3001 {
3002     struct free_area *area;
3003     int current_order;
3004     int min_order = order;
3005     struct page *page;
3006     int fallback_mt;
3007     bool can_steal;
3008
3009     /*
3010      * Do not steal pages from freelists belonging to other pageblocks
3011      * i.e. orders < pageblock_order. If there are no local zones free,
3012      * the zonelists will be reiterated without ALLOC_NOFRAGMENT.
3013      */
3014     if (alloc_flags & ALLOC_NOFRAGMENT)
3015         min_order = pageblock_order;
3016
3017     /*
3018      * Find the largest available free page in the other list. This
3019      * roughly
3020      * approximates finding the pageblock with the most free pages, which
3021      * would be too costly to do exactly.
3022      */
3023     for (current_order = MAX_ORDER - 1; current_order >= min_order;
3024          --current_order) {
3025         area = &(zone->free_area[current_order]);
3026         fallback_mt = find_suitable_fallback(area, current_order,
3027                                             start_migratetype, false, &can_steal);
3028         if (fallback_mt == -1)
3029             continue;
3030     }
3031
3032     /*
3033      * We cannot steal all free pages from the pageblock and the

```

```

3031         * requested migratetype is movable. In that case it's better
           to
3032         * steal and split the smallest available page instead of the
3033         * largest available page, because even if the next movable
3034         * allocation falls back into a different pageblock than this
3035         * one, it won't cause permanent fragmentation.
3036         */
3037         if (!can_steal && start_migratetype == MIGRATE_MOVABLE
3038             && current_order > order)
3039             goto find_smallest;
3040
3041         goto do_steal;
3042     }
3043
3044     return false;
3045
3046 find_smallest:
3047     for (current_order = order; current_order < MAX_ORDER;
3048         current_order++) {
3049         area = &(zone->free_area[current_order]);
3050         fallback_mt = find_suitable_fallback(area, current_order,
3051             start_migratetype, false, &can_steal);
3052         if (fallback_mt != -1)
3053             break;
3054     }
3055
3056     /*
3057     * This should not happen - we already found a suitable fallback
3058     * when looking for the largest page.
3059     */
3060     VM_BUG_ON(current_order == MAX_ORDER);
3061
3062 do_steal:
3063     page = get_page_from_free_area(area, fallback_mt);
3064
3065     steal_suitable_fallback(zone, page, alloc_flags, start_migratetype,
3066         can_steal);
3067
3068     trace_mm_page_alloc_extfrag(page, order, current_order,
3069         start_migratetype, fallback_mt);
3070
3071     return true;
3072
3073 }

```

Listing 2-65: mm/page_alloc.c: __rmqueue_fallback()

If `__rmqueue_fallback()` is called we have failed to find a page of the required migrate type at or above the required order so we fallback to moving pages from another migrate type to the desired one.

A page's migrate type is determined from the page block it belongs to via `get_pageblock_migratetype()`, however this is only a heuristic as we can 'steal' pages from one migrate block for another at a granularity less than this. When a page is freed it will end up returning to a free list associate with migrate type of its page block. If `ALLOC_NOFRAGMENT` is set then this will not be permitted.

NOTE

Page blocks are aligned sets of physical pages of a specific size. Page block granularity for x86-64 is order 9 or 512 base pages for example.

The logic is as follows:

1. If `ALLOC_FRAGMENT` is set the minimum order (of page to be moved from a fallback migrate type to the one we require) to set is the pageblock order, otherwise set this to the required order.
2. Iterating from largest allowable order (`MAX_ORDER - 1`, typically 10) to the minimum order, if `find_suitable_fallback()` indicates a suitable fallback migrate type can't be found, then loop again. If none can be found, returning false indicating that no fallback could be found. This iteration is an approximate means of finding the migrate type with the most free pages.
3. If `find_suitable_fallback()` indicated that we cannot steal the *whole* page block the required migrate type is `MIGRATE_MOVABLE` and we are considering moving an order larger than required we prefer to move the smallest possible page. Exit the loop and perform the `find_smallest` branch.
4. Otherwise, perform the `do_steal` branch.
5. The `find_smallest` branch repeats the top-level iteration only from smallest to largest page size. This is used when we would get less fragmentation than trying to find the migrate type with approximately the most free pages. Once this is complete, we fall through to the `do_steal` branch.
6. The `do_steal` branch invokes `steal_suitable_fallback()` to perform the actual stealing of pages, updates statistics and returns true to indicate that the steal was successful and `__rmqueue()` should be retried.

It's important to note here that, as discussed in section 2.5, as a result of this stealing mechanism migrate types at the page block level are merely heuristic – we can end up with page blocks containing a mix of different migrate types. This is something we try to avoid by 'stealing' the largest possible pages such that we can ideally simply change a page block's migrate type with no fragmentation.

However, if we are forced to mix migrate types, we reserve more memory to prevent further fragmentation – we achieve this through zone *boosting* whenever we encounter this situation. This increases watermarks, thereby causing reclaim to occur sooner.

The watermark boost is applied by `boost_watermark()` which is called only in `steal_suitable_fallback()`.

Examining `find_suitable_fallback()` as shown in Listing 2-66.

```

2832 /*
2833  * Check whether there is a suitable fallback freepage with requested order.
2834  * If only_stealable is true, this function returns fallback_mt only if
2835  * we can steal other freepages all together. This would help to reduce
2836  * fragmentation due to mixed migratetype pages in one pageblock.
2837  */
2838 int find_suitable_fallback(struct free_area *area, unsigned int order,
2839                          int migratetype, bool only_stealable, bool *can_steal)
2840 {
2841     int i;
2842     int fallback_mt;
2843
2844     if (area->nr_free == 0)
2845         return -1;
2846
2847     *can_steal = false;
2848     for (i = 0;; i++) {
2849         fallback_mt = fallbacks[migratetype][i];
2850         if (fallback_mt == MIGRATE_TYPES)
2851             break;
2852
2853         if (free_area_empty(area, fallback_mt))
2854             continue;
2855
2856         if (can_steal_fallback(order, migratetype))
2857             *can_steal = true;
2858
2859         if (!only_stealable)
2860             return fallback_mt;
2861
2862         if (*can_steal)
2863             return fallback_mt;
2864     }
2865
2866     return -1;
2867 }

```

Listing 2-66: `mm/page_alloc.c: find_suitable_fallback()`

And `can_steal_fallback()` which it invokes as shown in Listing 2-67.

```

2678 /*
2679  * When we are falling back to another migratetype during allocation, try to
2680  * steal extra free pages from the same pageblocks to satisfy further
2681  * allocations, instead of polluting multiple pageblocks.

```

```

2682  *
2683  * If we are stealing a relatively large buddy page, it is likely there will
2684  * be more free pages in the pageblock, so try to steal them all. For
2685  * reclaimable and unmovable allocations, we steal regardless of page size,
2686  * as fragmentation caused by those allocations polluting movable pageblocks
2687  * is worse than movable allocations stealing from unmovable and reclaimable
2688  * pageblocks.
2689  */
2690 static bool can_steal_fallback(unsigned int order, int start_mt)
2691 {
2692     /*
2693      * Leaving this order check is intended, although there is
2694      * relaxed order check in next check. The reason is that
2695      * we can actually steal whole pageblock if this condition met,
2696      * but, below check doesn't guarantee it and that is just heuristic
2697      * so could be changed anytime.
2698      */
2699     if (order >= pageblock_order)
2700         return true;
2701
2702     if (order >= pageblock_order / 2 ||
2703         start_mt == MIGRATE_RECLAIMABLE ||
2704         start_mt == MIGRATE_UNMOVABLE ||
2705         page_group_by_mobility_disabled)
2706         return true;
2707
2708     return false;
2709 }

```

Listing 2-67: mm/page_alloc.c: `can_steal_fallback()`

The logic depends on the `fallbacks[]` array which encodes the primary and secondary fallbacks for each migrate type, as shown in Table 2-8.

Table 2-8: Migrate type fallbacks

Migrate type	Primary fallback	Secondary fallback
MIGRATE_UNMOVABLE	MIGRATE_RECLAIMABLE	MIGRATE_MOVABLE
MIGRATE_MOVABLE	MIGRATE_RECLAIMABLE	MIGRATE_UNMOVABLE
MIGRATE_RECLAIMABLE	MIGRATE_UNMOVABLE	MIGRATE_MOVABLE

Note that other migrate types do not have fallbacks.

The logic is as follows:

1. If the free list contains no pages, return -1 indicating that no suitable fallback could be found.
2. Try the primary fallback, checking whether pages exist at the required order and migrate type via `free_area_empty()`. If not, try the secondary. If both fail, return -1 indicating that no suitable fallback could be found.

3. Set the `can_steal` output parameter to the result of `can_steal_fallback()`. This checks whether the order is greater than or equal to half of a page block (at which point we can treat the page block as being majority the required migrate type once stolen), or if the allocation is `MIGRATE_RECLAIMABLE` or `MIGRATE_UNMOVABLE` we steal regardless rather than mix unmovable pages in movable page blocks. `page_group_by_mobility_disabled` will also cause a steal to always occur, however this is only relevant to systems with very low memory.
4. If `only_stealable` is not set (the case for any `page_alloc.c` allocation invocation) then we return the chosen migrate type, otherwise we only return it if `can_steal` has been set.

Examining `steal_suitable_fallback()` as shown in Listing 2-68.

```

2748 /*
2749  * This function implements actual steal behaviour. If order is large enough,
2750  * we can steal whole pageblock. If not, we first move freepages in this
2751  * pageblock to our migratetype and determine how many already-allocated pages
2752  * are there in the pageblock with a compatible migratetype. If at least half
2753  * of pages are free or compatible, we can change migratetype of the pageblock
2754  * itself, so pages freed in the future will be put on the correct free list.
2755  */
2756 static void steal_suitable_fallback(struct zone *zone, struct page *page,
2757                                   unsigned int alloc_flags, int start_type, bool whole_block)
2758 {
2759     unsigned int current_order = buddy_order(page);
2760     int free_pages, movable_pages, alike_pages;
2761     int old_block_type;
2762
2763     old_block_type = get_pageblock_migratetype(page);
2764
2765     /*
2766      * This can happen due to races and we want to prevent broken
2767      * highatomic accounting.
2768      */
2769     if (is_migrate_highatomic(old_block_type))
2770         goto single_page;
2771
2772     /* Take ownership for orders >= pageblock_order */
2773     if (current_order >= pageblock_order) {
2774         change_pageblock_range(page, current_order, start_type);
2775         goto single_page;
2776     }
2777
2778     /*
2779      * Boost watermarks to increase reclaim pressure to reduce the
2780      * likelihood of future fallbacks. Wake kswapd now as the node
2781      * may be balanced overall and kswapd will not wake naturally.

```

```

2782     */
2783     if (boost_watermark(zone) && (alloc_flags & ALLOC_KSWAPD))
2784         set_bit(ZONE_BOOSTED_WATERMARK, &zone->flags);
2785
2786     /* We are not allowed to try stealing from the whole block */
2787     if (!whole_block)
2788         goto single_page;
2789
2790     free_pages = move_freepages_block(zone, page, start_type,
2791                                     &movable_pages);
2792     /*
2793      * Determine how many pages are compatible with our allocation.
2794      * For movable allocation, it's the number of movable pages which
2795      * we just obtained. For other types it's a bit more tricky.
2796      */
2797     if (start_type == MIGRATE_MOVABLE) {
2798         alike_pages = movable_pages;
2799     } else {
2800         /*
2801          * If we are falling back a RECLAIMABLE or UNMOVABLE
2802          allocation
2803          * to MOVABLE pageblock, consider all non-movable pages as
2804          * compatible. If it's UNMOVABLE falling back to RECLAIMABLE
2805          or
2806          * vice versa, be conservative since we can't distinguish the
2807          * exact migratetype of non-movable pages.
2808          */
2809         if (old_block_type == MIGRATE_MOVABLE)
2810             alike_pages = pageblock_nr_pages
2811                          - (free_pages + movable_pages);
2812         ;
2813     }
2814     else
2815         alike_pages = 0;
2816 }
2817
2818 /* moving whole block can fail due to zone boundary conditions */
2819 if (!free_pages)
2820     goto single_page;
2821
2822 /*
2823  * If a sufficient number of pages in the block are either free or of
2824  * comparable migratability as our allocation, claim the whole block.
2825  */
2826 if (free_pages + alike_pages >= (1 << (pageblock_order-1)) ||
2827     page_group_by_mobility_disabled)
2828     set_pageblock_migratetype(page, start_type);
2829

```

```

2826         return;
2827
2828     single_page:
2829         move_to_free_list(page, zone, current_order, start_type);
2830 }

```

Listing 2-68: mm/page_alloc.c: steal_suitable_fallback()

Examining the logic:

1. If the existing page block migrate type is set to MIGRATE_HIGHATOMIC due to race conditions, then do not attempt to steal the whole page block and instead take the single_page branch.
2. If the required order is equal to or exceeds the page block order then we simply change the migrate type of each page block in range, since for the allocation to be permitted they must all contain only free pages, then we invoke the single_page branch to move the pages to the new migrate type free list.
3. If neither of the above cases apply and we are able to do so we invoke boost_watermark() to add a watermark boost, then if ALLOC_KSWAPD is set we set the ZONE_BOSTED_WATERMARK bit to ensure kswapd, a kernel process which performs indirect reclaim, is woken up. See Chapter 11 for details.
4. If we are not permitted to steal the whole page block to the new migrate type, simply skip ahead to the single_page branch.
5. We are now at the point where we need to steal a block but the page block potentially consists of a mix of allocated and free pages of the old migrate type. Free pages within the page block containing the page are moved using move_freepages_block() (and move_freepages()), being careful not to cross zone boundaries. The number of allocated, but movable pages is assigned to movable_pages. The number of free pages moved is returned and assigned to free_pages.
6. Next we calculate alike_pages, which is a count of the number of allocated pages within the pageblock are *compatible* with the desired migrate type:
 - If the target migrate type is movable, the number of pages is simply equal to the number of allocated, movable pages within the page-block, i.e. movable_pages.
 - Otherwise if the source page block is MIGRATE_MOVABLE, this is equal to all *unmovable* pages within the page block.
 - Otherwise if neither the source nor the target migrate type is MIGRATE_MOVABLE we consider no pages to be alike. We do this to be conservative as we can't easily distinguish between different non-movable migrate types.
7. If no free pages were moved in move_freepages_block() we jump to the single_page branch.
8. Otherwise, we determine whether to change the page block migrate type to the target by summing free_pages and alike_pages and seeing if

this is equal to or exceeds half the page block size (i.e. if the pages which are alike and the free pages that will have been moved between migrate types dominate the page block). If so we set the page block migratetype via `set_pageblock_migratetype()`.

9. The `single_page` branch invokes `move_to_free_list()` to ‘steal’ the page to the target migrate type.

Examining `__rmqueue_smallest()` as shown in Listing 2-69.

```

2553 static __always_inline
2554 struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
2555                                int migratetype)
2556 {
2557     unsigned int current_order;
2558     struct free_area *area;
2559     struct page *page;
2560
2561     /* Find a page of the appropriate size in the preferred list */
2562     for (current_order = order; current_order < MAX_ORDER; ++current_order
2563 ) {
2564         area = &(zone->free_area[current_order]);
2565         page = get_page_from_free_area(area, migratetype);
2566         if (!page)
2567             continue;
2568         del_page_from_free_list(page, zone, current_order);
2569         expand(zone, page, order, current_order, migratetype);
2570         set_pcppage_migratetype(page, migratetype);
2571         trace_mm_page_alloc_zone_locked(page, order, migratetype,
2572                                         pcp_allowed_order(order) &&
2573                                         migratetype < MIGRATE_PCPTYPES);
2574         return page;
2575     }
2576     return NULL;
2577 }
```

Listing 2-69: `mm/page_alloc.c: __rmqueue_smallest()`

This functions assumes the zone spinlock is being held and contains the allocation part of the buddy algorithm:

1. Iterate through each order from the target page order to maximum.
2. Try to obtain a page for the target order and migrate type, using `get_page_from_free_area()` to obtain the page and `del_page_from_free_list()` to delete it from the free list.
3. If successful invoke `expand()` which, if an order higher than required was allocated, places the unneeded pages back on the appropriate freelists.

4. Invoke `set_pcppage_migratetype()` to set `struct page->index` to the migrate type for convenience when this page is freed to a PCP list in future. Additionally we register a *trace event* for this allocation.

Examining `expand()` as shown in Listing 2-70.

```

2340 static inline void expand(struct zone *zone, struct page *page,
2341                          int low, int high, int migratetype)
2342 {
2343     unsigned long size = 1 << high;
2344
2345     while (high > low) {
2346         high--;
2347         size >>= 1;
2348         VM_BUG_ON_PAGE(bad_range(zone, &page[size]), &page[size]);
2349
2350         /*
2351          * Mark as guard pages (or page), that will allow to
2352          * merge back to allocator when buddy will be freed.
2353          * Corresponding page table entries will not be touched,
2354          * pages will stay not present in virtual address space
2355          */
2356         if (set_page_guard(zone, &page[size], high, migratetype))
2357             continue;
2358
2359         add_to_free_list(&page[size], zone, high, migratetype);
2360         set_buddy_order(&page[size], high);
2361     }
2362 }
```

Listing 2-70: `mm/page_alloc.c: expand()`

As each page order represents a power of 2 splitting up a higher order page simply consists of dividing a page into two of the order below, placing one of these on the appropriate free list and repeating the process for the other until we end up with the page we want to return and its buddy which is the last we place on a free list.

This will be invoked with `high` set to the order that was allocated and `low` set to the required order and we loop `high - low` times, placing `high - low` pages onto freelists. Each loop starts by decrementing `high` meaning this field specifies the order of the page *after* splitting. The invocation of `set_page_guard()` is only relevant if `CONFIG_DEBUG_PAGEALLOC` is set so for brevity we will disregard it.

Throughout the iteration the `size` variable tracks the size, expressed in base pages, of the split pages. We choose to free the latter portion of the split page back to the freelist via `add_to_free_list()` by referencing `&page[size]`, setting its buddy order correctly using `set_buddy_order()`.

This ultimately leaves a page of the required order at page. Since we are about to remove this from its free list and provide it to a process we do not need to set buddy order.

The kernel contains a mechanism for reserving a page block's worth of memory exclusively for high-order *atomic* allocations (allocations performed where the kernel cannot sleep, e.g. under interrupt context) as shown in Listing 2-71.

```

2869 /*
2870  * Reserve a pageblock for exclusive use of high-order atomic allocations if
2871  * there are no empty page blocks that contain a page with a suitable order
2872  */
2873 static void reserve_highatomic_pageblock(struct page *page, struct zone *zone,
2874                                         unsigned int alloc_order)
2875 {
2876     int mt;
2877     unsigned long max_managed, flags;
2878
2879     /*
2880      * Limit the number reserved to 1 pageblock or roughly 1% of a zone.
2881      * Check is race-prone but harmless.
2882      */
2883     max_managed = (zone_managed_pages(zone) / 100) + pageblock_nr_pages;
2884     if (zone->nr_reserved_highatomic >= max_managed)
2885         return;
2886
2887     spin_lock_irqsave(&zone->lock, flags);
2888
2889     /* Recheck the nr_reserved_highatomic limit under the lock */
2890     if (zone->nr_reserved_highatomic >= max_managed)
2891         goto out_unlock;
2892
2893     /* Yoink! */
2894     mt = get_pageblock_migratetype(page);
2895     /* Only reserve normal pageblocks (i.e., they can merge with others)
2896     */
2897     if (migratetype_is_mergeable(mt)) {
2898         zone->nr_reserved_highatomic += pageblock_nr_pages;
2899         set_pageblock_migratetype(page, MIGRATE_HIGHATOMIC);
2900         move_freepages_block(zone, page, MIGRATE_HIGHATOMIC, NULL);
2901     }
2902 out_unlock:
2903     spin_unlock_irqrestore(&zone->lock, flags);
2904 }

```

Listing 2-71: mm/page_alloc.c: *reserve_highatomic_pageblock()*

This is invoked by `get_page_from_freelist()` (see listing ??) if the allocation is higher than order-0 and the `ALLOC_HARDER` flag is set (typically set for atomic allocations under memory pressure.) The page block is only moved to `MIGRATE_HIGHATOMIC` if `migratetype_is_mergeable()` which indicates whether a migrate type can have fallbacks.

The use of this migrate type is covered in more detail in the reclaim chapter. Examining `rmqueue_bulk()` (With out-of-scope CMA logic removed) as shown in Listing 2-72.

```

3112 /*
3113  * Obtain a specified number of elements from the buddy allocator, all under
3114  * a single hold of the lock, for efficiency. Add them to the supplied list.
3115  * Returns the number of new pages which were placed at *list.
3116  */
3117 static int rmqueue_bulk(struct zone *zone, unsigned int order,
3118                        unsigned long count, struct list_head *list,
3119                        int migratetype, unsigned int alloc_flags)
3120 {
3121     int i, allocated = 0;
3122
3123     /* Caller must hold IRQ-safe pcp->lock so IRQs are disabled. */
3124     spin_lock(&zone->lock);
3125     for (i = 0; i < count; ++i) {
3126         struct page *page = __rmqueue(zone, order, migratetype,
3127                                     alloc_flags);
3128         if (unlikely(page == NULL))
3129             break;
3130
3131         if (unlikely(check_pcp_refill(page, order)))
3132             continue;
3133
3134         /*
3135          * Split buddy pages returned by expand() are received here in
3136          * physical page order. The page is added to the tail of
3137          * caller's list. From the callers perspective, the linked
3138          * is ordered by page number under some conditions. This is
3139          * useful for IO devices that can forward direction from the
3140          * head, thus also in the physical page order. This is useful
3141          * for IO devices that can merge IO requests if the physical
3142          * pages are ordered properly.
3143          */
3144         list_add_tail(&page->pcp_list, list);
3145         allocated++;
3146     }
3147
3148     /*
3149      *
3150      */
3151 
```

```

3152      * i pages were removed from the buddy list even if some leak due
3153      * to check_pcp_refill failing so adjust NR_FREE_PAGES based
3154      * on i. Do not confuse with 'allocated' which is the number of
3155      * pages added to the pcp list.
3156      */
3157      __mod_zone_page_state(zone, NR_FREE_PAGES, -(i << order));
3158      spin_unlock(&zone->lock);
3159      return allocated;
3160 }

```

Listing 2-72: mm/page_alloc.c: *rmqueue_bulk()*

rmqueue_bulk() attempts to remove count pages of order from zone's migratetype free list with *alloc_flags* set and adds them to list, returning the number of pages actually allocated.

The heavy lifting is performed by the previously examined *__rmqueue()*, with page state being checked by *check_pcp_refill()* which performs checks to ensure returned pages are valid typically to avoid use-after-free bugs.

2.8.7 New page preparation

When a page is ready to be returned from the buddy allocator it needs to have some final preparations applied. This is done via *prep_new_page()* which is called from *get_page_from_freelist()* and *__alloc_pages_bulk()*. Examining the code as shown in Listing 2-73.

```

2529 static void prep_new_page(struct page *page, unsigned int order, gfp_t
      gfp_flags,
2530                                     unsigned int
      alloc_flags)
2531 {
2532     post_alloc_hook(page, order, gfp_flags);
2533
2534     if (order && (gfp_flags & __GFP_COMP))
2535         prep_compound_page(page, order);
2536
2537     /*
2538      * page is set pfmemalloc when ALLOC_NO_WATERMARKS was necessary to
2539      * allocate the page. The expectation is that the caller is taking
2540      * steps that will free more memory. The caller should avoid the page
2541      * being used for !PFMEMALLOC purposes.
2542      */
2543     if (alloc_flags & ALLOC_NO_WATERMARKS)
2544         set_page_pfmemalloc(page);
2545     else
2546         clear_page_pfmemalloc(page);
2547 }

```

Listing 2-73: mm/page_alloc.c: *prep_new_page()*

All pages invoke `post_alloc_hook()` which performs basic initialisation which always includes:

- Clear `struct page->private` via `set_page_private()`.
- Set the ref count to 1 via `set_page_refcounted()`.
- Zero pages if configured to do so via `kernel_init_pages()` (e.g. if `__GFP_ZERO` is set).

Compound pages also invoke `prep_compound_page()` which has already been discussed in section 2.1.1.

2.9 Freeing pages

Pages allocated by the kernel are reference-counted as they can be used by multiple different processes (for instance memory shared between forked processes or multiple references to a file mapping). The reference count is stored in the `struct page->_refcount` field (or equivalently `struct folio->_refcount`). This field tracks the number of ‘users’ of the page which is ultimately freed once this count reaches zero.

`get_page()` or `folio_get()` increments a page’s reference count and `put_page()` or `folio_put()` decrements it (the `_refcount` field should never be directly referenced).

`put_page_testzero()` atomically decrements a page’s reference counter and checks whether it has reached zero. This is used by all publicly exported page free functions.

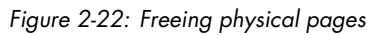
Freeing of kernel pages is typically performed by `__free_pages()`, whether via the slab allocator or directly. Userland pages are freed using `folio` which invokes `release_pages()` (see section 7.1 for more details).

Ultimately all frees end up in one of two places – `free_unref_page_commit()` if freeing from PCP lists or `__free_one_page()` otherwise. These comprise the heart of the allocator’s freeing mechanism.

The logic for freeing pages is simpler than that of allocating them – allocation might incur reclaim or require special handling (for example allocating in an atomic context or with recursive file system locks held) – whereas freeing pages merely consists of deciding where to put them and coalescing them with free buddy pages if possible.

As the underlying mechanisms are not complicated there is no need to visualise them in diagrammatic form, rather we examine the relevant functions directly.

There is, however, complexity around how exactly underlying functions are invoked – there are many entry points to the freeing mechanism which take into account that pages being freed may be of order-0 or compound, may be being freed in bulk or not as well as the aforementioned reference counting mechanism. We visualise this in Figure 2-22.



- Denotes functions that are exported.
- Denotes the key functions which all others ultimately arrive at.

```

3435 static void free_unref_page_commit(struct zone *zone, struct per_cpu_pages *
        pcp,
3436                                     struct page *page, int migratetype,
3437                                     unsigned int order)
3438 {
3439     int high;
3440     int pindex;

```

```

3441     bool free_high;
3442
3443     __count_vm_event(PGFREE);
3444     pindex = order_to_pindex(migratetype, order);
3445     list_add(&page->pcp_list, &pcp->lists[pindex]);
3446     pcp->count += 1 << order;
3447
3448     /*
3449      * As high-order pages other than THP's stored on PCP can contribute
3450      * to fragmentation, limit the number stored when PCP is heavily
3451      * freeing without allocation. The remainder after bulk freeing
3452      * stops will be drained from vmstat refresh context.
3453      */
3454     free_high = (pcp->free_factor && order && order <=
PAGE_ALLOC_COSTLY_ORDER);
3455
3456     high = nr_pcp_high(pcp, zone, free_high);
3457     if (pcp->count >= high) {
3458         int batch = READ_ONCE(pcp->batch);
3459
3460         free_pcppages_bulk(zone, nr_pcp_free(pcp, high, batch,
free_high), pcp, pindex);
3461     }
3462 }

```

Listing 2-74: mm/page_alloc.c: free_unref_page_commit()

The initial part of the code is straightforward – we determine which PCP list to place the page on, place it there and update stats. The only complexity that arises here is in determining when to free or pages from the PCP lists to the underlying free lists.

We determine when to flush PCP pages to free lists via `nr_pcp_high()`. This calculates the effective high watermark of the current CPU's PCP free pages.

When we have determined that PCP pages need to be flushed back to free lists, we do so via `free_pcppages_bulk()` which determines how many pages we free via `nr_pcp_free()`.

We set the `free_high` variable if the PCP `free_factor` is set, the page is compound and the page order is such that it is eligible to be present on PCP lists. While you might think this wouldn't be necessary in a function specifically designed for PCP page freeing, huge pages can also be present on PCP lists if transparent huge pages are enabled.

We examine `nr_pcp_high()` in Listing 2-75.

```

3417 static int nr_pcp_high(struct per_cpu_pages *pcp, struct zone *zone,
3418                      bool free_high)
3419 {
3420     int high = READ_ONCE(pcp->high);
3421

```

```

3422         if (unlikely(!high || free_high))
3423             return 0;
3424
3425         if (!test_bit(ZONE_RECLAIM_ACTIVE, &zone->flags))
3426             return high;
3427
3428         /*
3429          * If reclaim is active, limit the number of pages that can be
3430          * stored on pcp lists
3431          */
3432         return min(READ_ONCE(pcp->batch) << 2, high);
3433     }

```

Listing 2-75: mm/page_alloc.c: nr_pcp_high()

We can see that, if no PCP high watermark is set (which should only be the case in early kernel stages) or `free_high` is set, we indicate that flushing of PCP lists should go ahead no matter what.

As the comment in listing 2-74 indicates, `free_high` is set in cases where an allocation has been performed from a PCP list without a subsequent free and the page is higher order. This is because higher order pages on PCP lists are rather prone to causing fragmentation (higher order pages can always be broken down into lower order ones, the reverse is not so). You'll note that the memory management subsystem contains a great many heuristics. These are the results of optimisations based on real world experience.

Next, if reclaim is not indicated within the zone, we simply use the PCP list's designated high watermark level. If it is, then we select whichever is the smaller of the batch size multiplied by 4 or the high water mark (typically the former is considerably smaller). This is, again, a carefully tuned heuristic.

We can see that the PCP pages act as a per-core cache where pages are preferentially allocated from and freed to. By doing so we avoid expensive zone locks and thus improve performance of allocating and freeing memory.

Examining `nr_pcp_free()` as shown in Listing 2-76.

```

3388 static int nr_pcp_free(struct per_cpu_pages *pcp, int high, int batch,
3389                      bool free_high)
3390 {
3391     int min_nr_free, max_nr_free;
3392
3393     /* Free everything if batch freeing high-order pages. */
3394     if (unlikely(free_high))
3395         return pcp->count;
3396
3397     /* Check for PCP disabled or boot pageset */
3398     if (unlikely(high < batch))
3399         return 1;
3400
3401     /* Leave at least pcp->batch pages on the list */
3402     min_nr_free = batch;

```

```

3403         max_nr_free = high - batch;
3404
3405         /*
3406          * Double the number of pages freed each time there is subsequent
3407          * freeing of pages without any allocation.
3408          */
3409         batch <= pcp->free_factor;
3410         if (batch < max_nr_free)
3411             pcp->free_factor++;
3412         batch = clamp(batch, min_nr_free, max_nr_free);
3413
3414         return batch;
3415     }

```

Listing 2-76: mm/page_alloc.c: *nr_pcp_free()*

In line with its affect on *nr_pcp_high()*, if *free_high* is set we clear *all* PCP pages rather than a batch. Also, if the PCP is disabled or we are in the boot phase, we free a page a time.

In most cases however, we observe the following process:

- Set bounds: *min_nr_free* set to the batch size and *max_nr_free* set to the PCP high watermark value minus a batch. This means we always free *at least* a batch, but always leave at least one batch size of pages on the PCP.
- Multiply the batch size by $2^{\text{free\_factor}}$ and increase *free_factor* if this increase does not exceed the maximum permissible number of pages to drain. As discussed in section 2.7.3, this is a heuristic which ensures that pages stored on PCP lists are proportionate to pages allocated from them, exponentially increasing the number of pages drained back to free lists if intervening allocations do not occur.
- Finally, we clamp the number of pages to drain to a range within *min_nr_free* to *max_nr_free*.

free_pcppages_bulk(), as the name suggests, frees a number of PCP pages in bulk as shown in Listing 2-77.

```

1535 static void free_pcppages_bulk(struct zone *zone, int count,
1536                               struct per_cpu_pages *pcp,
1537                               int pindex)
1538 {
1539     int min_pindex = 0;
1540     int max_pindex = NR_PCP_LISTS - 1;
1541     unsigned int order;
1542     bool isolated_pageblocks;
1543     struct page *page;
1544
1545     /*
1546      * Ensure proper count is passed which otherwise would stuck in the
1547      * below while (list_empty(list)) loop.

```

```

1548     */
1549     count = min(pcp->count, count);
1550
1551     /* Ensure requested pindex is drained first. */
1552     pindex = pindex - 1;
1553
1554     /* Caller must hold IRQ-safe pcp->lock so IRQs are disabled. */
1555     spin_lock(&zone->lock);
1556     isolated_pageblocks = has_isolate_pageblock(zone);
1557
1558     while (count > 0) {
1559         :
1602     }
1603
1604     spin_unlock(&zone->lock);
1605 }

```

Listing 2-77: mm/page_alloc.c: free_pcppages_bulk() top level

This initially:

- Performs a sanity check to ensure that we only attempt to free as many pages as are available.
- Decrements the pindex by one (the index into the PCP list determined by order and migrate type) in order that the inner loop which initially increments this causes the freed page's order/migrate type to be drained first.
- Acquires a zone lock around the inner loop.
- Determines whether the zone contains any MIGRATE_ISOLATE pageblocks via `has_isolate_pageblock()` which checks the zone's `nr_isolate_pageblock` counter to do so.

Before invoking the inner loop until count has reached zero (this counts the name of base pages to be drained) as shown in Listing 2-78.

```

1559     struct list_head *list;
1560     int nr_pages;
1561
1562     /* Remove pages from lists in a round-robin fashion. */
1563     do {
1564         if (++pindex > max_pindex)
1565             pindex = min_pindex;
1566         list = &pcp->lists[pindex];
1567         if (!list_empty(list))
1568             break;
1569
1570         if (pindex == max_pindex)
1571             max_pindex--;
1572         if (pindex == min_pindex)

```

```

1573             min_pindex++;
1574         } while (1);
1575
1576         order = pindex_to_order(pindex);
1577         nr_pages = 1 << order;
1578         BUILD_BUG_ON(MAX_ORDER >= (1<<NR_PCP_ORDER_WIDTH));
1579         do {
1580             int mt;
1581
1582             page = list_last_entry(list, struct page, pcp_list);
1583             mt = get_pcppage_migratetype(page);
1584
1585             /* must delete to avoid corrupting pcp list */
1586             list_del(&page->pcp_list);
1587             count -= nr_pages;
1588             pcp->count -= nr_pages;
1589
1590             if (bulkfree_pcp_prepare(page))
1591                 continue;
1592
1593             /* MIGRATE_ISOLATE page should not go to pcplists */
1594             VM_BUG_ON_PAGE(is_migrate_isolate(mt), page);
1595             /* Pageblock could have been isolated meanwhile */
1596             if (unlikely(isolated_pageblocks))
1597                 mt = get_pageblock_migratetype(page);
1598
1599             __free_one_page(page, page_to_pfn(page), zone, order,
1600                             mt, FPI_NONE);
1601             trace_mm_page_pcpu_drain(page, order, mt);
1602         } while (count > 0 && !list_empty(list));

```

Listing 2-78: mm/page_alloc.c: *free_pcppages_bulk()* inner loop

Examining the logic:

- As the comment suggests, we begin by traversing PCP lists in a round-robin fashion, beginning with the order and migrate type of the page being freed. Since PCP lists are laid out by migrate type for each order, this means we free pages of each migrate type in order from unmovable, movable, and reclaimable at each increasing order until we wrap around to the lowest order and migrate type.
- We take the *last* page in the PCP list and use `get_pcppage_migratetype()` to retrieve its migrate type. This was placed in the index field of the PCP page's struct page to optimise lookup.
- The page is deleted from the PCP free list and stats are updated. `bulkfree_pcp_prepare()` is invoked which in turn calls `check_free_page()` (more if `CONFIG_DEBUG_VM` is set) to ensure the page is as expected.

- If there are isolated pageblocks within the zone, the PCP page might have been isolated since being freed so to be safe we retrieve the page block designated migrate type in this instance.
- Finally, we invoke `__free_one_page()` to perform the heavy lifting of actually freeing the pages. This function assumes a zone lock is held.

2.9.0.1 FPI flags

Free Page Internal (FPI) flags adjust page free behaviour.

Examining the signature of `__free_one_page()` as shown in Listing 2-79.

```

1102 static inline void __free_one_page(struct page *page,
1103                                   unsigned long pfn,
1104                                   struct zone *zone, unsigned int order,
1105                                   int migratetype, fpi_t fpi_flags)

```

Listing 2-79: mm/page_alloc.c: `__free_one_page()` signature

We can see that there is a `fpi_flags` parameter of type `fpi_t`. This declares flags which impact page free behaviour:

- `FPI_NONE` – No impact on freeing behaviour.
- `FPI_SKIP_REPORT_NOTIFY` – This interacts with the *free page reporting* mechanism to indicate that no such report should be generated for this page.
- `FPI_TO_TAIL` – Forces pages to be freed to the back of the relevant free list. This is an optimisation used in specific instances where shuffling pages (the usual behaviour) is *known* to be unnecessary or inappropriate.
- `FPI_SKIP_KASAN_POISON` – The linux kernel address sanitiser (*KASAN*) is a mechanism that is intended to catch errors in dynamic memory allocation. One part of its operation is to ‘poison’, i.e. uniquely mark unused pages with a recognisable pattern that can be detected and highlighted as a use-after-free. This flag causes this not to be done for pages where this is not appropriate.

These flags all specify bits in a bit field so they can be combined as needed.

2.9.1 `__free_one_page()`

Examining the top-level logic of `__free_one_page()` as shown in Listing 2-80.

```

1102 static inline void __free_one_page(struct page *page,
1103                                   unsigned long pfn,
1104                                   struct zone *zone, unsigned int order,
1105                                   int migratetype, fpi_t fpi_flags)
1106 {
1107     struct capture_control *capc = task_capc(zone);
1108     unsigned long buddy_pfn;
1109     unsigned long combined_pfn;

```

```

1110     struct page *buddy;
1111     bool to_tail;
1112
1113     VM_BUG_ON(!zone_is_initialized(zone));
1114     VM_BUG_ON_PAGE(page->flags & PAGE_FLAGS_CHECK_AT_PREP, page);
1115
1116     VM_BUG_ON(migratetype == -1);
1117     if (likely(!is_migrate_isolate(migratetype)))
1118         __mod_zone_freepage_state(zone, 1 << order, migratetype);
1119
1120     VM_BUG_ON_PAGE(pfn & ((1 << order) - 1), page);
1121     VM_BUG_ON_PAGE(bad_range(zone, page), page);
1122
1123     while (order < MAX_ORDER - 1) {
1124         :
1161     }
1162     :
1181 }

```

Listing 2-80: mm/page_alloc.c: __free_one_page() top-level logic

The code performs the following steps:

1. Initial setup and sanity checks.
2. Merge loop – This is the part of the code that implements the coalescing part of the buddy allocator, i.e. checking whether the page’s buddy is free, if so coalescing into an order-order + 1 page and looping round and trying to coalesce that block. This is why the loop iterates up to but *not* including the maximum allowed order of MAX_ORDER - 1, since each iteration may result in coalescing to a order + 1 page.
3. Add page to free list – Finally the logic adds the page to the appropriate free list.

Note that the code refers to compaction via struct `capture_control`, however this is out of scope for this chapter, see the chapter on compaction and migration for more on this.

Examining the `__free_one_page()` merge loop as shown in Listing 2-81.

```

1124     if (compaction_capture(capc, page, order, migratetype)) {
1125         __mod_zone_freepage_state(zone, -(1 << order),
1126                                     migratetype);
1127         return;
1128     }
1129
1130     buddy = find_buddy_page_pfn(page, pfn, order, &buddy_pfn);
1131     if (!buddy)
1132         goto done_merging;
1133
1134     if (unlikely(order >= pageblock_order)) {

```

```

1135         /*
1136         * We want to prevent merge between freepages on
1137         pageblock
1138         * without fallbacks and normal pageblock. Without
1139         this,
1140         * pageblock isolation could cause incorrect freepage
1141         or CMA
1142         * accounting or HIGHATOMIC accounting.
1143         */
1144         int buddy_mt = get_pageblock_migratetype(buddy);
1145         if (migratetype != buddy_mt
1146             && (!migratetype_is_mergeable(
1147                 migratetype) ||
1148                 !migratetype_is_mergeable(
1149                     buddy_mt)))
1150             goto done_merging;
1151     }
1152     /*
1153     * Our buddy is free or it is CONFIG_DEBUG_PAGEALLOC guard
1154     page,
1155     * merge with it and move up one order.
1156     */
1157     if (page_is_guard(buddy))
1158         clear_page_guard(zone, buddy, order, migratetype);
1159     else
1160         del_page_from_free_list(buddy, zone, order);
1161     combined_pfn = buddy_pfn & pfn;
1162     page = page + (combined_pfn - pfn);
1163     pfn = combined_pfn;
1164     order++;

```

Listing 2-81: mm/page_alloc.c: __free_one_page() merge loop

This code is invoked while `order < MAX_ORDER - 1`. Examining the logic (noting that, again, we defer discussion of compaction code to the relevant chapter):

1. Find buddy page – This is the *key* step in the merging process – we need to determine if there is a page we can coalesce this one with via `find_buddy_page_pfn()` (this makes use of the buddy allocator algorithm described in section 2.7.1 and the previous chapter). If no such page exists we exit the loop as no further coalescing is possible.
2. Check if cross a page block boundary – If we are page block order or above then we are merging pages across two page blocks. As we heuristically assign migrate type by page block, we need to check to ensure that the buddy's migrate type is mergeable with the page we are coalescing

it into (ultimately we will place the coalesced page on a free list of the original page's migrate type). We do this via `migratetype_is_mergeable()`.

3. Remove the buddy page from its free list – Finally we remove the buddy page from its free list, then update the PFN, page and order accordingly before looping to see if we can coalesce again.

Finally after any coalescing has taken place we add our newly merged page to the appropriate free list as shown in Listing 2-82.

```

1163 done_merging:
1164     set_buddy_order(page, order);
1165
1166     if (fpi_flags & FPI_TO_TAIL)
1167         to_tail = true;
1168     else if (is_shuffle_order(order))
1169         to_tail = shuffle_pick_tail();
1170     else
1171         to_tail = buddy_merge_likely(pfn, buddy_pfn, page, order);
1172
1173     if (to_tail)
1174         add_to_free_list_tail(page, zone, order, migratetype);
1175     else
1176         add_to_free_list(page, zone, order, migratetype);
1177
1178     /* Notify page reporting subsystem of freed page */
1179     if (!(fpi_flags & FPI_SKIP_REPORT_NOTIFY))
1180         page_reporting_notify_free(order);

```

Listing 2-82: `mm/page_alloc.c: __free_one_page()` add page to free list

Other than free page reporting (out of scope for this book), the only complexity here is the decision of whether to add a page to the head or the tail of the free list. If `FPI_TO_TAIL` is set the algorithmic mechanism is overridden, otherwise we start by determining if a memory shuffle need occur via `is_shuffle_order()`. This is only applicable for order-10 pages and only if `CONFIG_SHUFFLE_PAGE_ALLOCATOR` is set (this is a feature that is designed to make the page allocator more efficient in the presence of a direct-mapping memory-side-cache. This topic is out of scope for the book). If so, `shuffle_pick_tail()` is called to randomly select whether to place to page on the tail of the free list or not.

Under ordinary circumstances whether to place the page on the tail of the free list is determined by `buddy_merge_likely()`. As the name suggests this determines whether pages are likely to be merged soon. If they are, it is a good idea to put the page on the tail of the list so it doesn't get allocated soon (and thus made unavailable for merging).

After choosing where to put the page, `add_to_free_list()` or `add_to_free_list_tail()` is called as appropriate.

Examining `buddy_merge_likely()` as shown in Listing 2-83.

```

1053 /*
1054  * If this is not the largest possible page, check if the buddy
1055  * of the next-highest order is free. If it is, it's possible
1056  * that pages are being freed that will coalesce soon. In case,
1057  * that is happening, add the free page to the tail of the list
1058  * so it's less likely to be used soon and more likely to be merged
1059  * as a higher order page
1060  */
1061 static inline bool
1062 buddy_merge Likely(unsigned long pfn, unsigned long buddy_pfn,
1063                   struct page *page, unsigned int order)
1064 {
1065     unsigned long higher_page_pfn;
1066     struct page *higher_page;
1067
1068     if (order >= MAX_ORDER - 2)
1069         return false;
1070
1071     higher_page_pfn = buddy_pfn & pfn;
1072     higher_page = page + (higher_page_pfn - pfn);
1073
1074     return find_buddy_page_pfn(higher_page, higher_page_pfn, order + 1,
1075                               NULL) != NULL;
1076 }

```

Listing 2-83: mm/page_alloc.c: `buddy_merge Likely()`

The key thing to note here is that we are examining pages at order + 1 which might be merged into a page of order + 2 (if pages of order order were mergeable to order + 1 we would have already merged them).

In effect we are checking whether, should this page's buddy get freed, would that result in a merge to order + 2? We achieve this by simply invoking `find_buddy_page_pfn()` for order + 1 to see if its buddy is available.

3

VIRTUAL MEMORY

Arguably the most important memory feature provided by hardware is the ability to abstract memory addresses altogether. This forms the basis of modern memory management by allowing *separation* of the address spaces of each running process as well as permitting the kernel to decide the permissions of memory ranges and how to handle invalid accesses. Over the course of this chapter we'll examine how the kernel initializes and manages virtual memory.

In the past, hardware placed no restriction on software being able to access all available physical memory on a machine. This is problematic in a number of respects:

Stability A program can access memory regardless of whose it is—its own, another program's or the system's. This means that a programming error resulting in the access of unintended memory can result in unrecoverable memory corruption, a system hang if you are lucky, and/or data loss if you are not.

Complexity Loading programs is significantly complicated. The OS has to ensure that it loads programs in such a position in memory that it not

only doesn't overlap that of other programs but also doesn't overlap the memory they are *yet* to use.

Security Every process is as privileged as any other and can access data no matter how sensitive at will, which means that in effect there is no security between programs at all.

Contiguity A contiguous block of memory is one that spans an address range without gaps. If you expose physical memory directly, then obtaining contiguous blocks of memory becomes a real problem; you don't control the physical memory layout so fragmentation is an ever-present threat and memory cannot be moved because any program referencing it holds a pointer to the *actual* location in memory, which would be invalidated on any attempt to move it. A sensible program will expect to be allocated contiguous memory, so as a result, memory exhaustion due to fragmentation becomes likely.

Capacity When physical memory is exposed directly, you can access *only* the physical memory available to the system. This might seem a moot point, but facilities like paging are unavailable rendering running out of memory a critical system failure.

Different approaches to solving these problems have been tried throughout the history of computing, but the one hardware developers have converged upon is virtual memory.

The concept is simple. Rather than expose physical memory directly, use *virtual addresses* that can be mapped arbitrarily to physical addresses at page granularity and alignment. Once this is in place, you can have each process believe it has all of the memory available to itself, which at a stroke eliminates all of the aforementioned issues.

NOTE

I differentiate between programs and processes here because without virtual memory, the concept of a process is rather meaningless.

The concept of virtual memory really does necessitate a separation between “privileged” kernel code and “unprivileged” user space as the ability to map virtual memory to physical implies access to *all* memory and thus only the kernel can be permitted to do this.

Ultimately support for virtual memory has to come from hardware, which provides the ability to change virtual memory mappings and determines whether currently executed code is privileged or not (thus whether or not the mapping can be changed).

Using this facility, a kernel can switch out virtual mappings for user space processes so each has their own independent virtual *address space*. The kernel provides system calls to allocate memory into their address space at a page granularity thus abstracting the process of obtaining memory altogether. From the kernel's point of view, this is a two-step process: allocating physical memory, then mapping it into the process.

When user space processes access unmapped memory or memory they are not permitted to, or try to perform an operation that's not permitted

in that virtual memory range (for example, write in a read-only mapping or execute code in memory marked non-executable), the CPU generates a *page fault*, a hardware exception that the kernel can trap and respond to.

When accessing arbitrary memory that it should not, the kernel can *seg-fault* the process in a fashion every application programmer is familiar with. However, page faults can *also* be used to facilitate other useful functionality (a non-exhaustive list):

Demand paging We said earlier that userland memory allocation from the kernel’s perspective is a two-step process: physical allocation and virtual mapping. However, the kernel can also choose to be “lazy” and instead make this a *zero-step* operation—simply updating an internal data structure signifying that the allocated memory range is a valid virtual memory range. Doing so allows it to defer the actual allocation and mapping to when the memory is *accessed* rather than allocated (and a page fault is raised). This is a powerful concept that we’ll revisit in Chapter 4 on process memory, and Chapter 6 page faults.

Copy-On-Write (CoW) semantics An important facility for a Unix operating system is the ability to *fork* processes. This is where a process makes an exact copy of itself that executes independently as a new *child* process. Virtual memory permits a very efficient means of doing so: copy the virtual memory mappings (but not the underlying physical memory) and mark them read-only and “copy on write.” As the name suggests, this instructs the kernel to copy the underlying physical memory only when it is written to. By marking the mapping read-only, we can use the resultant page fault to copy only those pages that are written to.

NUMA balancing In a NUMA system a core may take longer to access some memory that is not “local”—that is, physically near it. It may not be obvious which core will ultimately access a particular memory page, so in order to ensure that memory allocations are efficiently distributed, userland memory can be periodically unmapped, allowing for *migrate-on-fault* semantics—that is, if the current core accessing the memory on fault is not local to it, migrate and re-map.

As the hardware mechanics virtual memory relies upon are inevitably architecture-specific, I have, for brevity, chosen to focus on *x86-64* for the those parts of the kernel code which interface with the hardware directly (as I have done so across the book). This applies to only a minority of the code examined here, so the majority of the contents of this chapter remain relevant regardless of architecture.

3.1 Page Tables

In the previous chapter on physical memory allocation, I remarked that memory must be divided into *pages* in order to be managed sensibly. The same applies to virtual memory mappings that also have to be mapped at a page granularity to remain manageable.

This leaves us with a question: how should we map virtual pages to physical ones? Subdividing memory into pages *alone* doesn't save us from memory exhaustion, as a 64-bit system has up to 2^{64} bytes of available address space. If memory were subdivided into 4KiB pages and directly mapped, this would result in 4,503,599,627,370,496 pages, and the mapping metadata alone would require 32 petabytes of RAM. Increasing page size to anything within a sensible range would do little to resolve this.

Of course no realistic system would have any use for mapping all possible virtual memory addresses at once, so we need some means of denoting that some of the virtual address space is *not* mapped. Given that the majority of virtual address space will remain unmapped at all times, the solution is to use a *sparse* mapping—that is, mapping only what we need in such a way as to eliminate as much metadata as possible for empty mappings.

We need to store metadata in memory, and since the smallest unit of memory that can be allocated is a single base page, the most efficient means of obtaining a sparse mapping is to subdivide each virtual address into a page's worth of possible mappings at a time. We denote each of these mapping tables *page tables*.

Page tables are arranged in a hierarchy of *page table levels*; for a page to exist at a lower level in the hierarchy, at least one entry has to exist at a higher level pointing to it. This means we only ever allocate metadata if we have at least one mapping in it and minimize wasted space on nil mappings.

The page table hierarchy begins with a global page table that's always accessible to a given process, called the *page global directory* (PGD). The PGD provides physical addresses for page tables at the next level of granularity, which provides physical addresses for the next lower page table level, and so on, until finally we provide the physical address of the mapped data.

How do we map a virtual address to entries in page tables? We treat the address as a series of adjacent indices into each page table level and finally into the page itself.

For clarity, let's examine concrete examples. We'll consider page table semantics for x86-64 here, but the concepts remain the same for other architectures. With a page size of 4KiB and 8 bytes per address, this means the minimum number of mappings we can set is 512 pages at a time (2MiB of address space).

Since $512 = 2^9$ and $4,096 = 2^{12}$, we require 12 bits for the data page offset that the virtual address points at and 9 bits for each page table level. Modern x86-64 hardware can support up to 57 bits of *physical* memory. Since virtually addressing memory that cannot exist physically is pointless, this implies that only the lower 57 bits of a virtual memory address are meaningful, and thus we require *up to* five page table levels to map virtual memory.

NOTE

In x84-64, virtual addresses must be of canonical form, which means that all upper bits must be equal to the highest permissible bit. For example, if bit-56 is 0, then bits 57 through 63 must be 0 also. If it is 1, they must also be 1.

It turns out five-level x86-64 virtual addresses are the maximum any architecture supports. Most consumer x86-64 systems support only 48 bits

(four levels), and other architectures or configurations might support significantly less. The kernel deals with this by *defining* five page table levels and cleverly using macros to ignore levels that don't exist (*folding* them into the level above).

Table 3-1 shows the page tables within the kernel (note that type declarations are architecture-specific).

Table 3-1: Page Table Levels

Level	Entry type	Count	Bit offset	Description
PGD	pgd_t	PTRS_PER_PGD	PGDIR_SHIFT	Page global directory
P4D	p4d_t	PTRS_PER_P4D	P4D_SHIFT	Page fourth-level directory
PUD	pud_t	PTRS_PER_PUD	PUD_SHIFT	Page upper directory
PMD	pmd_t	PTRS_PER_PMD	PMD_SHIFT	Page middle directory
PTE	pte_t	PTRS_PER_PTE	PAGE_SHIFT	Page table entry directory

Each of these entry types defines an individual entry in these tables that contains the physical address of a page table of the next level in the hierarchy (or in the case of a PTE, the address of the data page containing the data referenced by the virtual address), so PGDs contain physical addresses of P4Ds, P4Ds contain physical addresses of PUDs, and so on.

As a form of rudimentary type safety, these are implemented as bare structs wrapping their underlying values. The kernel defines the underlying fundamental data types in, for x86-64, 64-bit specific wrappers shown in Listing 3-1.

```

14 typedef unsigned long pteval_t;
15 typedef unsigned long pmdval_t;
16 typedef unsigned long pudval_t;
17 typedef unsigned long p4dval_t;
18 typedef unsigned long pgdval_t;
19 :
21 typedef struct { pteval_t pte; } pte_t;
```

Listing 3-1: arch/x86/include/asm/pgtable_64_types.h: page table entry value types

These value types are then wrapped by the actual page table type wrappers shown in Listing 3-2.

```

265 typedef struct { pgdval_t pgd; } pgd_t;
266 :
312 typedef struct { p4dval_t p4d; } p4d_t;
313 :
338 typedef struct { pudval_t pud; } pud_t;
339 :
364 typedef struct { pmdval_t pmd; } pmd_t;
```

Listing 3-2: arch/x86/include/asm/pgtable_types.h: page table entry types

Declaring these this way means the compiler checks to ensure that a PGD entry is never supplanted for a P4D, a P4D for a PUD, and so on.

3.1.1 Page Table Operations

There are myriad, mostly architecture-specific, helper functions for page table operations. We'll examine these briefly to provide an idea of the scope of kernel operations.

First, we'll consider the page table flags specific to the PGD and P4D page table levels shown in Table 3-2. As these levels don't map pages directly and normally don't have their flags altered, fewer operations are defined for them.

Table 3-2: Page Table Helper Functions: PGD and P4D

	PGD	P4D
Get raw value	<code>pgd_val()</code>	<code>p4d_val()</code>
Cast raw value	<code>__pgd()</code>	<code>__p4d()</code>
Get index	<code>pgd_index()</code>	<code>p4d_index()</code>
Get next level	<code>pgd_offset()</code> [†]	<code>p4d_offset()</code>
Get flags	<code>pgd_flags()</code>	<code>p4d_flags()</code>
Get PFN	<code>pgd_pfn()</code>	<code>p4d_pfn()</code>
Get page table struct	<code>page</code> <code>pgd_page()</code>	<code>p4d_page()</code>
Get next entry address	<code>pgd_addr_end()</code>	<code>p4d_addr_end()</code>
Get fine-grained lock (split)	-	-
Get fine-grained lock (shared)	-	-
Acquire fine-grained lock	-	-
In bad state?	<code>pgd_bad()</code>	<code>p4d_bad()</code>
Is empty?	<code>pgd_none()</code>	<code>p4d_none()</code>
Are the same?	<code>pgd_same()</code>	<code>p4d_same()</code>
From PFN/flags	-	-
From struct page/flags	-	-
From entry/flags	-	-
Set flags	-	-
Clear flags	-	-
Set entry	<code>set_pgd()</code>	<code>set_p4d()</code>
Set entry with check	-	-
Clear	<code>pgd_clear()</code>	<code>p4d_clear()</code>
Get or allocate next level	<code>pgd_alloc()</code>	<code>p4d_alloc()</code>
Get/allocate w/tracking	-	<code>p4d_alloc_track()</code>
Allocate	-	<code>__p4d_alloc()</code>
Free	<code>pgd_free()</code>	<code>p4d_free()</code>
Set from struct page	<code>pgd_populate()</code>	<code>p4d_populate()</code>

We then examine page table helper functions for the PUD, PMD, and PTE page table levels in Table 3-3. *Key:*

★ – The PTE lock is applied at the same time as obtaining the PTE entry via `pte_offset_map_lock()`.

★★ – For kernel allocations, retrieving or allocationg a PTE can be performed via `pte_alloc_kernel_track()`.

† – We can also use `pgd_offset_k()` for kernel mappings.

Table 3-3: Page Table Helper Functions: PUD, PMD, PTE

	PUD	PMD	PTE
Get raw value	<code>pud_val()</code>	<code>pmd_val()</code>	<code>pte_val()</code>
Cast raw value	<code>__pud()</code>	<code>__pmd()</code>	<code>__pte()</code>
Get index	<code>pud_index()</code>	<code>pmd_index()</code>	<code>pte_index()</code>
Get next level	<code>pud_offset()</code>	<code>pmd_offset()</code>	<code>pte_offset_map()</code>
Get flags	<code>pud_pgprot()</code>	<code>pmd_pgprot()</code>	<code>pte_pgprot()</code>
Get PFN	<code>pud_pfn()</code>	<code>pmd_pfn()</code>	<code>pte_pfn()</code>
Get page table struct page	<code>pud_page()</code>	<code>pmd_page()</code>	<code>pte_page()</code>
Get next entry address	<code>pud_addr_end()</code>	<code>pmd_addr_end()</code>	-
Get fine-grained lock (split)	-	<code>pmd_lockptr()</code>	<code>pte_lockptr()</code>
Get fine-grained lock (shared)	<code>pud_lockptr()</code>	<code>pmd_lockptr()</code>	<code>pte_lockptr()</code>
Acquire fine-grained lock	<code>pud_lock()</code>	<code>pmd_lock()</code>	*
In bad state?	<code>pud_bad()</code>	<code>pmd_bad()</code>	-
Is empty?	<code>pud_none()</code>	<code>pmd_none()</code>	<code>pte_none()</code>
Are the same?	<code>pud_same()</code>	<code>pmd_same()</code>	<code>pte_same()</code>
From PFN/flags	<code>pfn_pud()</code>	<code>pfn_pmd()</code>	<code>pfn_pte()</code>
From struct page/flags	-	<code>mk_pmd()</code>	<code>mk_pte()</code>
From entry/flags	-	<code>pmd_modify()</code>	<code>pte_modify()</code>
Set flags	<code>pud_set_flags()</code>	<code>pmd_set_flags()</code>	<code>pte_set_flags()</code>
Clear flags	<code>pud_clear_flags()</code>	<code>pmd_clear_flags()</code>	<code>pte_clear_flags()</code>
Set entry	<code>set_pud()</code>	<code>set_pmd()</code>	<code>set_pte()</code>
Set entry with check	<code>set_pud_at()</code>	<code>set_pmd_at()</code>	<code>set_pte_at()</code>
Clear	<code>pud_clear()</code>	<code>pmd_clear()</code>	<code>pte_clear()</code>
Get or allocate next level	<code>pud_alloc()</code>	<code>pmd_alloc()</code>	<code>pte_alloc()</code>
Get/allocate w/tracking	<code>pud_alloc_track()</code>	<code>pmd_alloc_track()</code>	**
Allocate	<code>__pud_alloc()</code>	<code>__pmd_alloc()</code>	<code>__pte_alloc()</code>
Free	<code>pud_free()</code>	<code>pmd_free()</code>	<code>pte_free()</code>
Set from struct page	<code>pud_populate()</code>	<code>pmd_populate()</code>	-

These helper functions are used through the memory manager for page table allocation, initialisation, modification, clean up, and freeing among many other tasks.

x86-64 pages can be set to three different sizes: 4KiB, 2MiB, and 1GiB. The kernel exposes the ability to change naturally aligned mappings of 4KiB pages to mappings at the larger pages sizes. These mappings are then termed *huge pages*. This is done for cache efficiency.

At the time of writing, most commercially available x86-64 hardware doesn't support five page table levels. The concept of a P4D page table level still exists, only it is “folded” into the page table above and assigned size of zero, so in effect, it's ignored and the compiler eliminates any redundant code.

Additionally, x86-64 implements huge pages in rather a straightforward fashion, simply dropping a page table and setting the appropriate page flag. Two huge page sizes are available: 2 MiB and 1 GiB.

We start by examining the layout of a five-level, 4 KiB page size configuration in Figure 3-1.

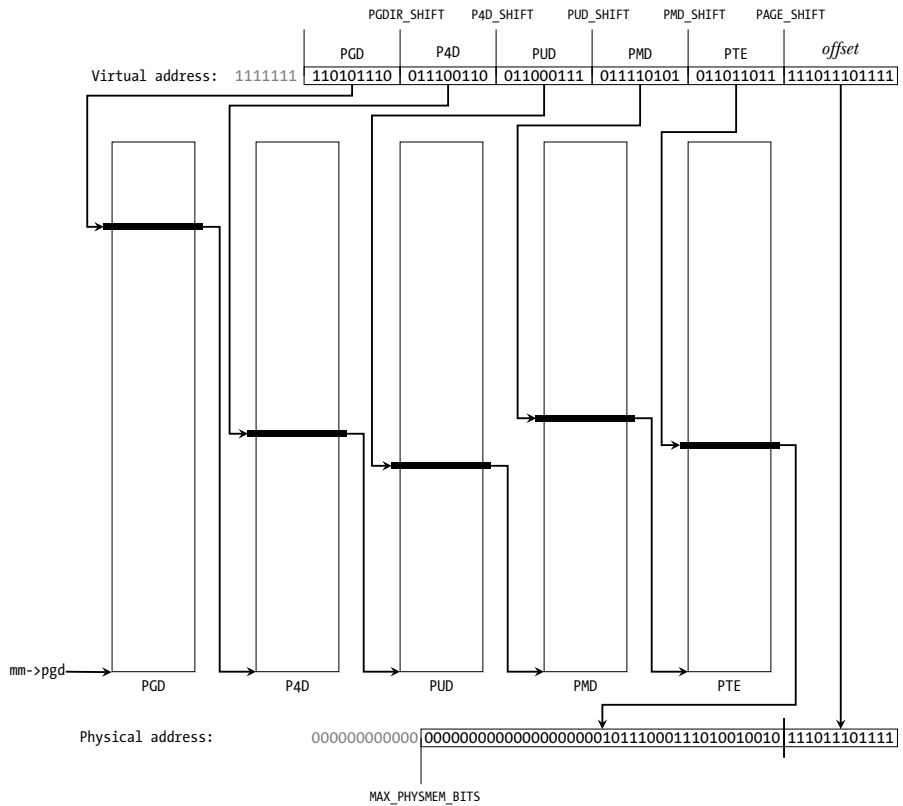
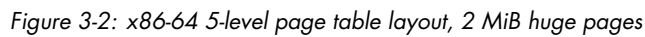


Figure 3-1: x86-64 five-level page table layout, 4 KiB pages

Note that the PGD's address is determined using the process's `struct mm_struct` object, via the `pgd` field (we examine this data type in considerable detail in Chapter 4). For x86-64, the high bits must be equal to the upper most meaningful bit to be valid (termed “canonical form”). Additionally, there are maximum `MAX_PHYSMEM_BITS` available in physical memory, which for 5-level is 52 bits or 4 PiB of memory.

We can remove the PTE page table level to obtain 2 MiB huge page sizes, as shown in Figure 3-2.



As you can see, the data page offset simply extended to consume the next page table level and a page table level is dropped. We can do this again to obtain 5-level x86-64 1 GiB huge page tables, as shown in Figure 3-3.

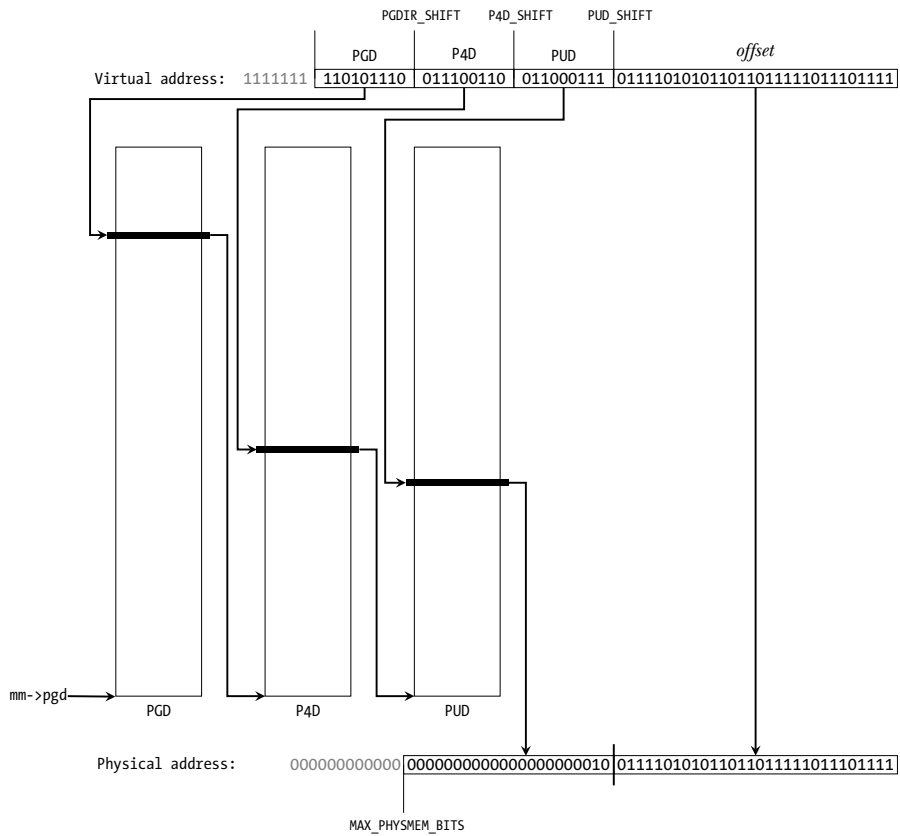


Figure 3-3: x86-64 five-level page table layout, 1 GiB huge pages

Again we drop a page table level and expand the data page offset to accommodate the difference.

The four-level page table configuration is very similar, only with the P4D level elided. Figure 3-4 shows the 4KiB page case.

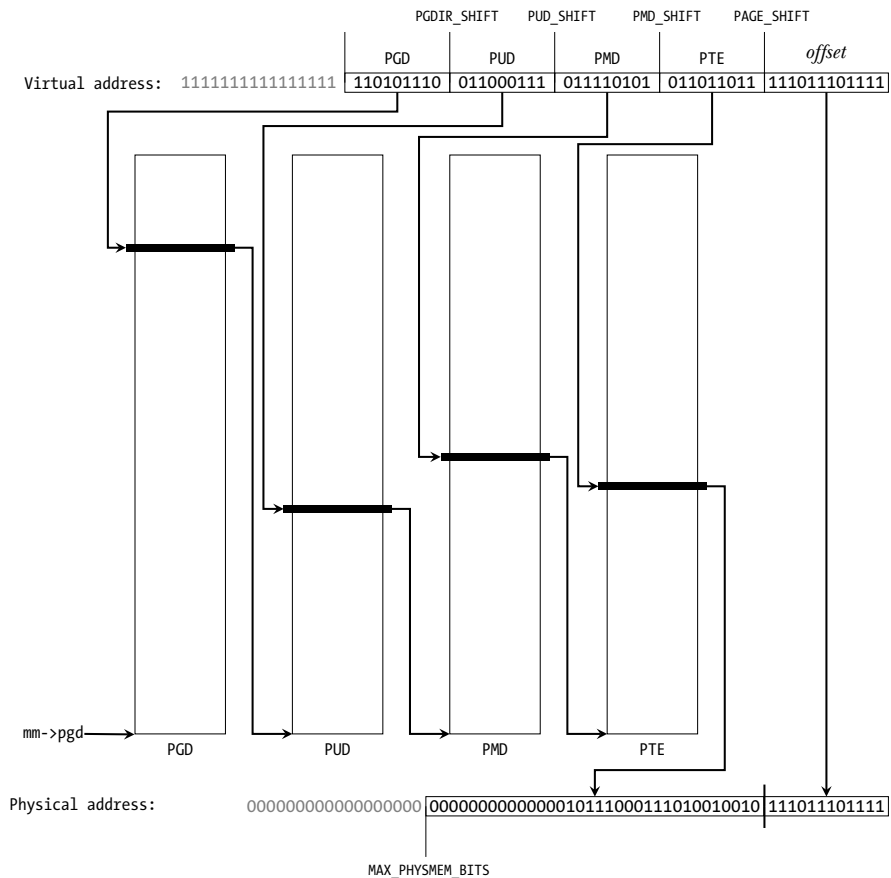


Figure 3-4: x86-64 4-level page table layout, 4 KiB pages

We drop the P4D page table level, but also note that `MAX_PHYSMEM_BITS` goes to 46 bits from 52, and `PGDIR_SHIFT` drops to a 39-bit offset to accommodate the reduction in address space. As before, all higher bits outside of the virtual address range must be set to the same value as the most significant valid bit for the virtual address to be canonical.

As with five-level page tables, we can achieve 2 MiB and 1 GiB huge page tables by dropping the lowest page table levels. Figure 3-5 shows the the 4-level 2 MiB huge page case.

Finally, similar to the five-level case, we can obtain 2GiB huge pages by also dropping the PMD level, as shown in Figure 3-6.



Since page tables must be aligned to the page size, this leaves lower bits of page table entries clear (12 lower bits in the case of x86-64 at a 4KiB page size), and since physical page address size is limited (57 bits in five-level, 48 bits in four-level), we have additional redundancy at the higher bits of the entries. As a result, we (and the hardware) can mask these bits when looking up physical addresses and free them up to be able to store *page table flags* that describe attributes of the mapping (for example, read-only/read-write, executable, kernel/user, and so on).

Page flags vary by architecture. As mentioned previously, we'll focus on x86-64, but the concepts remain similar for all architectures, so it's not a major task to adapt these to whatever architecture you are using.

Figure 3-7 shows how page table flags are extracted from page table entries.

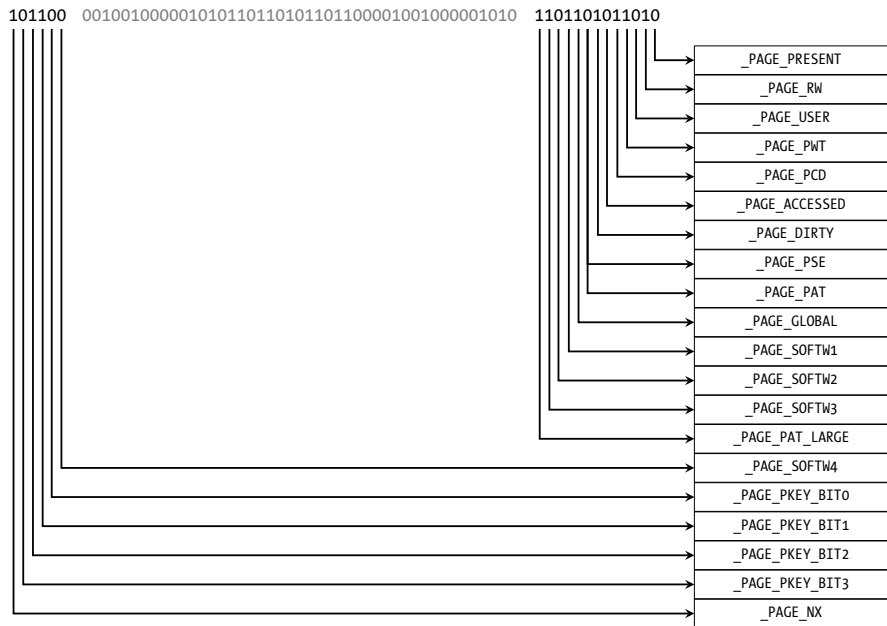


Figure 3-7: x86-64 page table flags

Table 3-4 shows which page tables each of these flags relate to.

Table 3-4: x86-64 page table flags

Flag	Bit	Description	PGD	P4D	PUD	PMD	PTE
_PAGE_PRESENT	0	Is the page mapped?	•	•	•	•	•
_PAGE_RW	1	Page writable if set	•	•	•	•	•
_PAGE_USER	2	Unprivileged user page	•	•	•	•	•
_PAGE_PWT	3	Write-through cache	•	•	•	•	•
_PAGE_PCD	4	Cache disabled	•	•	•	•	•
_PAGE_ACCESSED	5	Has page been accessed?	•	•	•	•	•
_PAGE_DIRTY	6	Has page been written to?			1 GiB	2 MiB	•
_PAGE_PSE	7	Is this a huge page?			1 GiB	2 MiB	
_PAGE_PAT	7	Page Attribute Table applies			×	×	•
_PAGE_GLOBAL	8	Mapping survives context switch			1 GiB	2 MiB	•
_PAGE_SOFTW1	9	Software-defined	•	•	•	•	•
_PAGE_SOFTW2	10	Software-defined	•	•	•	•	•
_PAGE_SOFTW3	11	Software-defined	•	•	•	•	•
_PAGE_PAT_LARGE	12	PAT flag for huge pages			1 GiB	2 MiB	
_PAGE_SOFTW4	58	Software-defined	•	•	•	•	•
_PAGE_PKEY_BIT0	59	Protection key			1 GiB	2 MiB	•
_PAGE_PKEY_BIT1	60	Protection key			1 GiB	2 MiB	•
_PAGE_PKEY_BIT2	61	Protection key			1 GiB	2 MiB	•
_PAGE_PKEY_BIT3	62	Protection key			1 GiB	2 MiB	•
_PAGE_NX	63	Non-executable	•	•	•	•	•

Note that, while the hardware might support these flags (in this instance x86-64), the kernel may not explicitly expose nor use these at every page table level.

In addition a number of flags overload others for specific purposes (mostly the software-defined flags), as shown in Table 3-5.

Table 3-5: x86-64 Overloaded Page Table Flags

Flag	Overloads	Bit	Description
<code>_PAGE_PROTNONE</code>	<code>_PAGE_GLOBAL</code>	8	<code>PROT_NONE</code> mapped page
<code>_PAGE_SPECIAL</code>	<code>_PAGE_SOFTW1</code>	9	Has no struct page
<code>_PAGE_CPA_TEST</code>	<code>_PAGE_SOFTW1</code>	9	PAT test flag
<code>_PAGE_UFFD_WP</code>	<code>_PAGE_SOFTW2</code>	10	<i>userfaultfd</i> managed page
<code>_PAGE_SOFT_DIRTY</code>	<code>_PAGE_SOFTW3</code>	11	Software-defined dirty flag
<code>_PAGE_DEVMAP</code>	<code>_PAGE_SOFTW4</code>	58	<code>ZONE_DEVICE</code> page

The following list describes each flag:

- `_PAGE_PRESENT`** Indicates that the page directory/data page referred to is actually resident in memory. If this flag is not set, any attempt to access memory at an address described by this entry will fault.
Checked by `pte_present()` (also returns true if `_PAGE_PROTNONE` is set, see description later in this list), `pmd_present()` (also returns true if either `_PAGE_PROTNONE` or `_PAGE_PSE` are set—the latter to avoid an edge-case race condition), `pud_present()`, `p4d_present()` and `pgd_present()`.
- `_PAGE_RW`** Indicates that a page can be written to as well as read from. If this flag is clear, attempts to write to the page will fault.
Checked by `pte_write()`, `pmd_write()`, and `pud_write()`. Set by `pte_mkwrite()`, `pmd_mkwrite()`, and `pud_mkwrite()`. Cleared by `pte_wrprotect()`, `pmd_wrprotect()`, and `pud_wrprotect()`.
- `_PAGE_USER`** If set, indicates that this memory is accessible when the CPU is set to ring 3 privilege level (that is, user mode). If this flag is cleared and access to the memory is attempted from user mode, a fault will be raised.
No helper methods for x86-64 check this flag directly, but, in addition to checking `_PAGE_PRESENT` and, if appropriate, `_PAGE_RW`, it is checked by `pte_access_permitted()`, `pmd_access_permitted()`, and `pud_access_permitted()`.
- `_PAGE_PWT`** Indicates that memory accesses should be *write-through*, that is, writes should not be cached but immediately written back to the physical bus. Typically used for memory-mapped devices where writes must not be cached.
- `_PAGE_PCD`** Indicates that cache should be disabled when *reading* from this page. Again this is typically used for memory-mapped devices, for example, registers which must be read from without delay. Can be combined with `_PAGE_PWT` when no caching whatsoever can be permitted.
- `_PAGE_ACCESSED`** A “sticky” flag, set but never cleared by hardware when the page is read from (this is what makes it sticky). The kernel can clear it,

which means it can be used to determine if a page is in active use or not (an important trick!)

This flag is checked via `pte_young()`, `pmd_young()`, and `pud_young()`. Set by `pte_mkyoung()`, `pmd_mkyoung()`, and `pud_mkyoung()`. Cleared by `pte_mkold()`, `pmd_mkold()`, and `pud_mkold()`.

The “young” and “old” refer to the age of the mapping in the sense that newer “younger” pages are likely to be regularly accessed, whereas older pages are not. More on how this flag is used in the chapter on reclaim.

_PAGE_DIRTY Another ‘sticky’ flag, set but never cleared by hardware when the page is written to. Again, the kernel can clear it, which can be used in order to determine when to flush memory-mapped pages to disk.

Checked by `pte_dirty()`, `pmd_dirty()`, and `pud_dirty()`. Set by `pte_mkdirty()`, `pmd_mkdirty()`, and `pud_mkdirty()` (note that these *also* set the `_PAGE_SOFT_DIRTY` flag described later). Cleared by `pte_mkclean()`, `pmd_mkclean()`, and `pud_mkclean()`.

_PAGE_PSE Determines whether a PMD points at a 2MiB data *huge page* rather than a PTE, or whether a PUD points at a 1GiB data *huge page* rather than a PMD. If this is set, the hardware will stop decoding page tables at this level and use the appropriately aligned data page contained within the entry to determine the resultant data page.

Checked by `pte_huge()`, `pmd_large()`, and `pud_large()`, set by `pte_mkhuge()`, `pmd_mkhuge()`, and `pud_mkhuge()` and cleared by `pte_clrhuge()`.

Note that the functions which reference PTE here will be using cast PMD/PUD entries as PTE entries cannot be marked huge.

_PAGE_PAT/_PAGE_PAT_LARGE Indicates that this page has a Page Attribute Table (PAT) associated with it (the `_PAGE_PAT_LARGE` variant is for huge page sizes). Discussion of this topic is out of scope for this book.

_PAGE_GLOBAL If set, then context switching does not result in this address’s Translation Lookaside Buffer (TLB), a cache of virtual address to physical, discussed in more detail later) being cleared. Useful for kernel mappings that should remain mapped regardless of which process is running. This needs to be treated with care given concerns around meltdown/spectre vulnerabilities, however.

Checked by `pte_global()`, set by `pte_mkglobal()`, and cleared by `pte_clrglobal()`.

_PAGE_SOFTW[1-4] These flags are explicitly software-only and overloaded for various other flags described next.

_PAGE_PKEY_BIT[0-3] These flags implement Intel memory protection keys, a topic which is out of scope for this book.

_PAGE_NX If set, code *cannot* be executed in this page. This is a vital security tool as it prevents exploit code from being able to execute arbitrary code in regions of memory used for storing data such as stack or heap allocations. This flag should only be cleared for explicit code mappings (and those pages should be marked read-only).

Checked by `pte_exec()` and set by `pte_mkexec()`.

_PAGE_PROTNONE Overloads **_PAGE_GLOBAL**, and is only set if **_PAGE_PRESENT** is *not* set. Indicates that the page is mapped but is intended to always result in page faults should it be accessed. Useful for guard pages placed after regions of memory that risk buffer overruns where it is desired that a segfault should be raised when this occurs. Set by `mmap()` as a `PROT_NONE` parameter.

Checked by `pte_protnone()` and `pmd_protnone()`. Both assert that the **_PAGE_PRESENT** flag is not set. `pte_present()` and `pmd_present()` assert that either the **_PAGE_PRESENT** or **_PAGE_PROTNONE** flags are set.

Note that the physical address encoded in the entry on modification via `pte_modify()` or `pmd_modify()` is inverted each time **_PAGE_PROTNONE** is either set or cleared. This is to prevent unwanted speculative memory reads for memory that is intended not to be read.

_PAGE_SPECIAL Overloads **_PAGE_SOFTW1**. This indicates that the mapping should not be associated with a struct `page`; either it is device memory or the mapping is not intended to interact with physical page tracking. Checked by `pte_special()` and set by `pte_mkspecial()`.

_PAGE_CPA_TEST Overloads **_PAGE_SOFTW1**. A flag used by PAT for self-test, out of scope for this book.

_PAGE_UFFD_WP Overloads **_PAGE_SOFTW2**. Used to indicate a *userfaultfd* managed page, out of scope for this book.

_PAGE_SOFT_DIRTY Overloads **_PAGE_SOFTW3**. *Soft-dirty PTEs* are an added layer of dirty page tracking which can be used by userland to keep track of pages which have been written to by a process since an arbitrary point in time. A user can clear these flags for all pages used by a process by writing to `/proc/$pid/clear_refs`, then read back the page table flags using `/proc/$pid/pagemap`. The clearing logic is performed by `clear_soft_dirty()`. Checked by `pte_soft_dirty()`, `pmd_soft_dirty()`, and `pud_soft_dirty()`. Set by `pte_mksoft_dirty()`, `pmd_mksoft_dirty()`, and `pud_mksoft_dirty()`. Cleared by `pte_clear_soft_dirty()`, `pmd_clear_soft_dirty()`, and `pud_clear_soft_dirty()`.

_PAGE_DEVMAP Overloads **_PAGE_SOFTW4** – Used to indicate a **ZONE_DEVICE** page, out of scope for this book.

3.1.3 Page Flag Combinations

A number of useful page table flag combinations are defined for specific purposes such as setting flags for user and kernel pages (excluding out of scope encrypted memory flags), as shown in Table 3-6.

Table 3-6: x86-64 Page Flag Combinations

Flag	__PP	__RW	__USR	__A	__NX	__D	__PSE	__G	__WP	__NC
__PAGE_KERNEL	•	•		•	•	•		•		
__PAGE_KERNEL_IO	•	•		•	•	•		•		
__PAGE_KERNEL_EXEC	•	•		•		•		•		
__PAGE_KERNEL_RO	•			•	•	•		•		
__PAGE_KERNEL_ROX	•			•		•		•		
__PAGE_KERNEL_VVAR	•		•	•	•	•		•		
__PAGE_KERNEL_LARGE	•	•		•	•	•	•	•		
__PAGE_KERNEL_LARGE_EXEC	•	•		•		•	•	•		
__PAGE_KERNEL_WP	•	•		•	•	•		•	•	
__PAGE_KERNEL_NOCACHE	•	•		•	•	•		•		•
__PAGE_KERNEL_IO_NOCACHE	•	•		•	•	•		•		•
__PAGE_TABLE	•	•	•	•		•				
__KERNPG_TABLE	•	•		•		•				
PAGE_NONE				•				•		
PAGE_SHARED	•	•	•	•	•					
PAGE_SHARED_EXEC	•	•	•	•						
PAGE_COPY_EXEC	•		•	•						
PAGE_READONLY_EXEC	•		•	•						
PAGE_COPY_NOEXEC	•		•	•	•					
PAGE_COPY	•		•	•	•					
PAGE_READONLY	•		•	•	•					
PAGE_KERNEL	•	•		•	•	•		★		
PAGE_KERNEL_RO	•			•	•	•		★		
PAGE_KERNEL_EXEC	•	•		•		•		★		
PAGE_KERNEL_ROX	•			•		•		★		
PAGE_KERNEL_NOCACHE	•	•		•	•	•		★		•
PAGE_KERNEL_LARGE	•	•		•	•	•	•	★		
PAGE_KERNEL_LARGE_EXEC	•	•		•		•	•	★		
PAGE_KERNEL_VVAR	•		•	•	•	•		★		

Key:

__PP is an abbreviation of __PAGE_PRESENT.

__RW is an abbreviation of __PAGE_RW.

__USR is an abbreviation of __PAGE_USER.

__A is an abbreviation of __PAGE_ACCESSED.

__NX is an abbreviation of __PAGE_NX.

__D is an abbreviation of __PAGE_DIRTY.

__PSE is an abbreviation of __PAGE_PSE.

__G is an abbreviation of __PAGE_GLOBAL.

__WP is an abbreviation of __PAGE_PWT.

__NC is an abbreviation of __PAGE_PWT and __PAGE_PCD.

- * - __PAGE_GLOBAL is only set if hardware mitigation for the *meltdown* vulnerability is available. Otherwise *Page Table Isolation (PTI)* (X86_FEATURE_PTI) is enabled and this flag is *not* set. This flag is set up in `probe_page_size_mask()` on boot. Note that, while this is a CPU feature flag, it is software-only and does not reflect a hardware mitigation.

Kernel page mappings will use one of the `PAGE_KERNEL_*` flag variants based on requirements. Page tables are mapped using either `__PAGE_TABLE` for user page tables and `__KERNPG_TABLE` for kernel page tables.

Userland page flags are determined by the `VM_` flags (`VM_SHRED`, `VM_READ`, `VM_WRITE`, and `VM_EXEC`) and mapped in `vm_get_page_prot()`, as shown in Table 3-7.

Table 3-7: x86-64 VM Flag to Page Flag Mappings

VM_SHARED	VM_READ	VM_WRITE	VM_EXEC	Flag
				PAGE_NONE
•				PAGE_NONE
	•			PAGE_READONLY
		•		PAGE_COPY
	•	•		PAGE_COPY
			•	PAGE_READONLY_EXEC
	•		•	PAGE_READONLY_EXEC
		•	•	PAGE_COPY_EXEC
	•	•	•	PAGE_COPY_EXEC
•				PAGE_NONE
•	•			PAGE_READONLY
•		•		PAGE_SHARED
•	•	•		PAGE_SHARED
•			•	PAGE_READONLY_EXEC
•	•		•	PAGE_READONLY_EXEC
•		•	•	PAGE_SHARED_EXEC
•	•	•	•	PAGE_SHARED_EXEC

Chapter 4 on Process Memory will go into more detail as to how these `VM_` flags are determined.

3.1.4 Page Table Traversal

We will go into detail as to how page table entries are managed in Chapter 6 on Page Faults. However, to get a general idea as to how page tables are traversed manually, consider `follow_pte()` as shown in Listing 3-3.

```

5269 int follow_pte(struct mm_struct *mm, unsigned long address,
5270                pte_t **ptep, spinlock_t **ptlp)
5271 {
5272     pgd_t *pgd;
5273     p4d_t *p4d;
```



```

5274     pud_t *pud;
5275     pmd_t *pmd;
5276     pte_t *ptep;
5277
5278     pgd = pgd_offset(mm, address);
5279     if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
5280         goto out;
5281
5282     p4d = p4d_offset(pgd, address);
5283     if (p4d_none(*p4d) || unlikely(p4d_bad(*p4d)))
5284         goto out;
5285
5286     pud = pud_offset(p4d, address);
5287     if (pud_none(*pud) || unlikely(pud_bad(*pud)))
5288         goto out;
5289
5290     pmd = pmd_offset(pud, address);
5291     VM_BUG_ON(pmd_trans_huge(*pmd));
5292
5293     if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
5294         goto out;
5295
5296     ptep = pte_offset_map_lock(mm, pmd, address, ptlp);
5297     if (!pte_present(*ptep))
5298         goto unlock;
5299     *ptepp = ptep;
5300     return 0;
5301 unlock:
5302     pte_unmap_unlock(ptep, *ptlp);
5303 out:
5304     return -EINVAL;
5305 }

```

Listing 3-3: mm/memory.c: follow_pte()

This uses the address to determine indexes of each page table entry, using `pgd_offset()` to determine the PGD entry, `p4d_offset()` to determine P4D entry, `pud_offset()` to determine PUD entry, `pmd_offset()` to determine PMD entry, and `pte_offset_map_lock()` to obtain the PTE entry while also acquiring a fine-grained PTE spin lock.

Each retrieval entry is checked to ensure it is not empty via `pXX_offset()` and is a valid state via `pXX_bad()`.

3.1.5 Page Table Locking

When page table entries are modified, a lock must be acquired. There is a per-process spin lock, `struct mm_struct->page_table_lock`, which is used for all page levels other than PMD and PTE.

For example, `__p4d_alloc()` allocates a P4D page and installs it in a PGD entry, acquiring `page_table_lock` along the way, `__pud_alloc()` and `__pmd_alloc()` do the same, installing a PUD in a P4D entry and a PMD in a PUD entry, respectively. However, `__pte_alloc()` does things differently by calling `pmd_install()` which acquires `pmd_lock()`, possibly fine-grained. Finally, when obtaining a PTE entry for the purposes of modifying it, `pte_offset_map_lock()` calls `pte_lockptr()` to obtain a possibly fine-grained PTE lock.

Why a *possibly* fine-grained lock? That's because the kernel determines whether to use fine-grained spinlocks for PTEs based on the `USE_SPLIT_PTE_PTLOCKS` define and for PMDs via `USE_SPLIT_PMD_PTLOCKS`. Fine-grained locks are stored in the struct `page->ptl` field (see previous chapter for more on struct `page` layout), and are either a pointer to a dynamically allocated spin lock via `ptlock_ptr()`, or are an inline spin lock via the identically named `ptlock_ptr()`. This is determined by `ALLOC_SPLIT_PTLOCKS`. Let's examine these defines in Listing 3-4.

```

22 #define USE_SPLIT_PTE_PTLOCKS    (NR_CPUS >= CONFIG_SPLIT_PTLOCK_CPUS)
23 #define USE_SPLIT_PMD_PTLOCKS    (USE_SPLIT_PTE_PTLOCKS && \
24     IS_ENABLED(CONFIG_ARCH_ENABLE_SPLIT_PMD_PTLOCK))
25 #define ALLOC_SPLIT_PTLOCKS      (SPINLOCK_SIZE > BITS_PER_LONG/8)

```

Listing 3-4: include/linux/mm_types_task.h: Page table locking defines

The splitting is based on a heuristic where if the system has sufficient CPUs for potential contention to be enough for the benefit of acquiring fine-grained locks at PTE (and possibly PMD) levels to exceed the cost of acquiring/releasing more locks. The number of cores where this comes into play and whether to do the same for PMDs is configurable.

Whether we allocate these fine-grained locks is based purely on whether we can fit the spin lock into the single long that struct `page->ptl` occupies, as determined by `ALLOC_SPLIT_PTLOCKS`.

See Chapter 4 on Process Memory for some discussion of the nuances of the use of `page_table_lock`.

3.2 The Address Space

Virtual address space describes all of the virtual addresses that could possibly be used in a system. Since virtual memory is by its nature arbitrary, the layout of virtual address space is entirely up to the kernel (as is what virtual addresses are permitted to be used by userland processes).

We must reserve a portion of the address space for the kernel. Given the vast size of the 64-bit address space, we simply divide the address space in half with the lower addresses assigned to user space and the higher addresses assigned to the kernel.

As discussed previously, x86-64 must have all unusable bits of virtual addresses set to the most-significant *valid* bit in order to be in canonical form. Therefore, if the most significant bit is 0, then so are all the remaining higher bits and equally so if it is 1. Therefore, userland addresses will always

have the most significant bit cleared, and kernel addresses will have the most significant bit set, making it easy to differentiate between the two.

Note that we are constrained in which addresses we can use by this canonical form. For 4-level page tables this limits us to use of the lowest 48 bits of virtual addresses (covering 256TiB), with 128TiB assigned to the kernel; for 5-level the lowest 57 bits (covering 128PiB) with 64PiB assigned to the kernel.

We maintain a portion of the kernel virtual address space for the *direct mapping*, which contains all non-device physical addresses mapped into virtual such that the kernel can access physical memory quickly and conveniently (more on this topic in the next section).

In order for this to fit in the kernel virtual address space the direct mapping cannot span all possible physical memory in its entirety (otherwise all of the virtual address space would be taken up just by the direct mapping), nor can it take up half as this would occupy *all* kernel mappings. The *maximum* sensible amount of space for it to occupy is therefore one quarter of the available address space, and this is what is reserved.

This, therefore, also constrains the maximum physical memory which is limited to `MAX_PHYSMEM_BITS`. For 4-level there are 48 bits of physical memory available; if we take a quarter of this, then we are left with 46 bits (64TiB). For 5-level we are limited by hardware, so while we have reserved 55 bits of 57 for the direct mapping (32PiB), we are in fact currently limited to using 52 bits (4PiB) of physical address space.

Userland address space is always constrained to the first half of available address space and is entirely dedicated to the current userland process. The *kernel space* (in the upper half of available address space), however, has specific regions mapped. Kernel Address Space Layout Randomisation (KASLR), a security mitigation mechanism (the details of which are out of scope for the book), results in offsets to these regions, but the subdivision remains the same.

NOTE

Kernel Address Space Layout Randomisation (KASLR) is a technique whereby kernel regions of memory are offset in order to prevent exploits from reliably targeting specific memory.

The exact addresses vary depending on whether the hardware supports 4-level or 5-level page tables. Let's examine the 5-level case first in Figure 3-8.

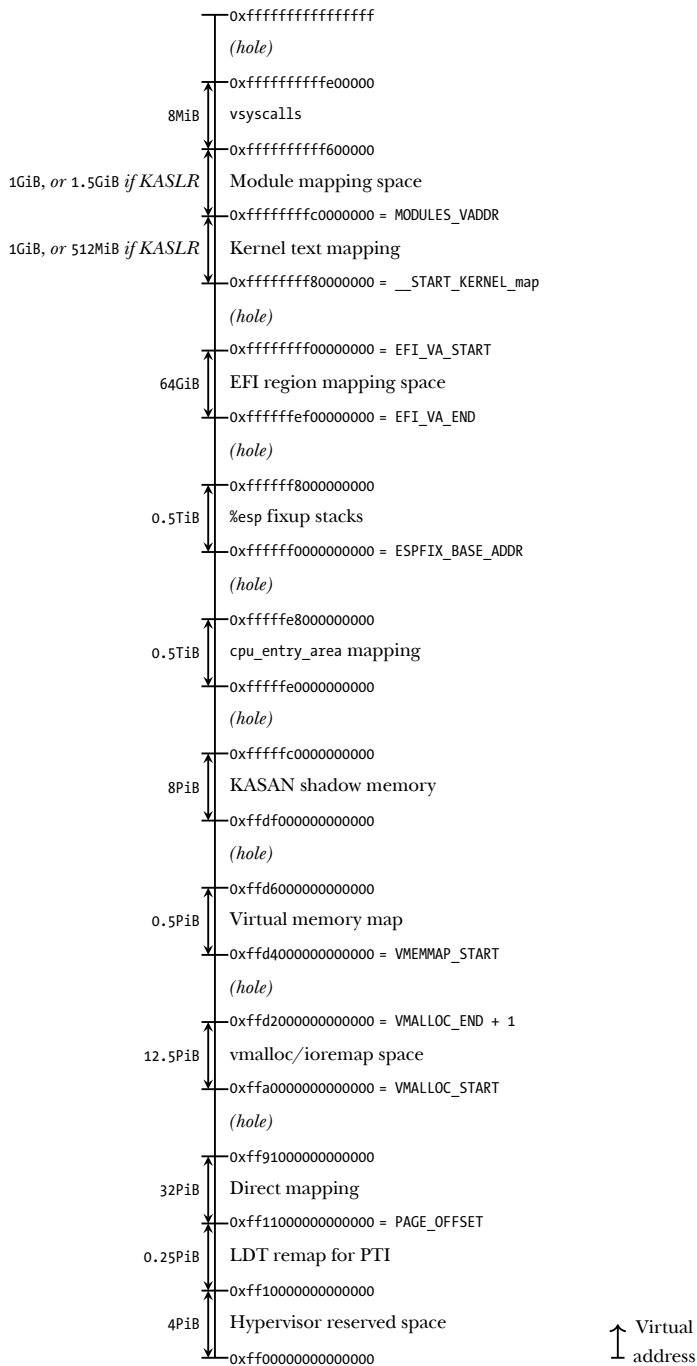


Figure 3-8: x86-64 5-level kernel virtual address layout

Next, we examine the 4-level case, which is similar but restricted to 48 bits, shown in Figure 3-9.

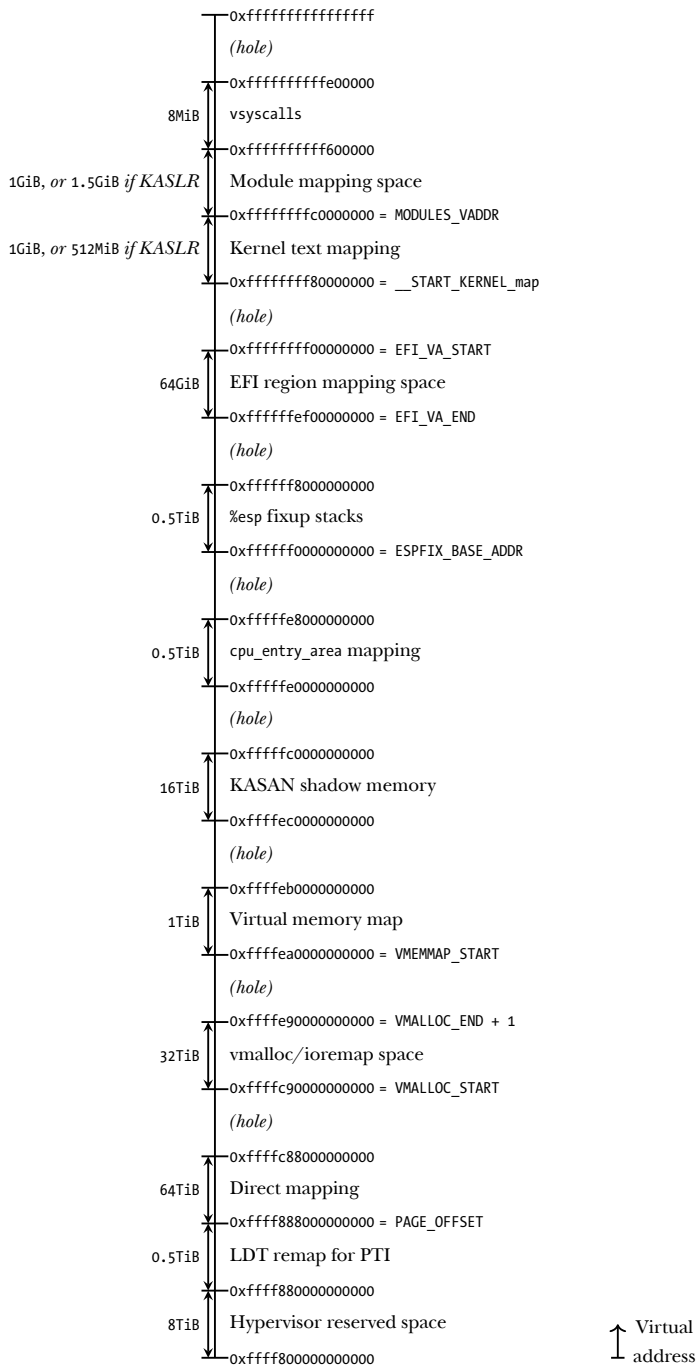


Figure 3-9: x86-64 4-level kernel virtual address layout

3.3 Direct Mapping

As we limit maximum physical memory while at the same time possessing a truly vast 64-bit address space, it becomes feasible for the kernel to simply map *all* accessible physical memory into the kernel address space, which it does in what is known as the *direct mapping*.

This is hugely convenient as it makes converting between virtual and physical addresses incredibly simple and obviates the need to manipulate page table mappings for the vast majority of kernel memory tasks.

We'll examine the direct mapping but also how direct mappings are bootstrapped in the early phases of kernel initialisation. We examine this in such detail not only because it is interesting and important to understand how we establish it, but also because it gives us insight into the interaction between virtual and physical memory and how one is mapped to another in specific detail.

The direct mapping is placed at `PAGE_OFFSET` which on x86-64 maps to `__PAGE_OFFSET`. When `CONFIG_DYNAMIC_MEMORY_LAYOUT` is set this maps to an externally defined variable `page_offset_base`, that can vary according to KASLR.

Addresses are translated from physical to virtual via `__va()` (the macro version) or `phys_to_virt()` (an inline function equivalent which simply invokes `__va()`). Addresses are converted the other way via `__pa()` or `virt_to_phys()`, as shown in Listing 3-5.

```

42 #define __pa(x)          __phys_addr((unsigned long)(x))
   :
59 #define __va(x)          ((void *)((unsigned long)(x)+PAGE_OFFSET))

```

Listing 3-5: `arch/x86/include/asm/page.h: __pa()` and `__va()`

The translation from physical to virtual is, due to the direct mapping, simplicity itself—simply offset by `PAGE_OFFSET`, and you have your answer.

The translation from virtual to physical, however, is more involved. This function permits the conversion from *kernel* addresses either from the direct mapping virtual range *or* from the kernel text mapping (where the kernel ELF image is loaded) so must differentiate between the two.

The logic is applied in `__phys_addr_nodebug()`, which starts by subtracting `__START_KERNEL_map` from the address. This is the start of the kernel virtual address space which contains the kernel text mapping (without taking into account KASLR) as shown in Listing 3-6.

```

19 static __always_inline unsigned long __phys_addr_nodebug(unsigned long x)
20 {
21     unsigned long y = x - __START_KERNEL_map;
22
23     /* use the carry flag to determine if x was < __START_KERNEL_map */
24     x = y + ((x > y) ? phys_base : (__START_KERNEL_map - PAGE_OFFSET));
25
26     return x;
27 }

```

Listing 3-6: arch/x86/include/asm/page_64.h: `__phys_addr_nodebug()`

Since the kernel places the text mapping at a higher address than the direct mapping, this means `y` is either negative (meaning this is not located within the kernel text mapping) or positive and equal to the offset from `__START_KERNEL_map`. We then use this to determine the offset into one of the two regions. If the region is the kernel text mapping, then we offset by any KASLR adjustment set in `phys_base`.

3.3.1 Bootstrapping

In this early stage of x86-64 memory initialisation we find ourselves in something of a chicken-and-egg situation. In order to access page tables, we need to use the direct mapping to look up physical addresses. However, to do that, we need page tables mapped.

To resolve this, the kernel reserves a small area of memory in the kernel ELF image itself: `early_pgt_alloc`. This is a buffer in which the kernel places the very first page tables allocated for the direct mapping. The PGD is already specified as part of `init_mm` as `swapper_pg_dir` (which in turn points at `init_top_pgt`), so we need only generate page tables below this.

This space has to be sufficient to handle the worst case of huge pages not being available, which for each aligned 2MiB of physical memory requires a 4KiB page to be allocated for each of the P4D, PUD, PMD, and PTE directories.

`early_pgt_alloc` reserves a portion of the early kernel ELF image in the `brk` section, but in order to access this, it must be requested via `extend_brk()`. This is done by `early_alloc_pgt_buf()`, which sets up the `pgt_buf` containing these early page tables, with `pgt_buf_start` set to the start of the buffer, `pgt_buf_end` set to the *current* exclusive end of the used portion of the buffer, and `pgt_buf_top` specifying capacity as shown in Listing 3-7.

```

187 RESERVE_BRK(early_pgt_alloc, INIT_PGT_BUF_SIZE);
188 void __init early_alloc_pgt_buf(void)
189 {
190     unsigned long tables = INIT_PGT_BUF_SIZE;
191     phys_addr_t base;
192
193     base = __pa(extend_brk(tables, PAGE_SIZE));
194
195     pgt_buf_start = base >> PAGE_SHIFT;
196     pgt_buf_end = pgt_buf_start;
197     pgt_buf_top = pgt_buf_start + (tables >> PAGE_SHIFT);
198 }
```

Listing 3-7: arch/x86/mm/init.c: `early_alloc_pgt_buf()`

In x86-64 we absolutely must have access to the ‘ISA’ memory range (that is, the first megabyte of physical memory) before doing anything else for

architecture-specific reasons, so this reserved space must have sufficient room for page tables to map this, as well as sufficient space for page tables to map wherever future page table allocations will go.

3.3.2 Direct Mapping Initialization

The number of page tables is determined by `INIT_PGD_PAGE_TABLES`: 3 for 4-level and 4 for 5-level. The actual number of pages needed to be reserved is then doubled and set as `INIT_PGD_PAGE_COUNT` to cover the ISA and page table regions. *However*, if KASLR is applied (meaning the direct mapping location in virtual memory is offset by a random amount), we have to take into account a worse-case scenario where the offset causes these mappings to cross a PGD entry boundary, meaning we need to allocate *twice* as many pages. In that case `INIT_PGD_PAGE_COUNT` is equal to four times the number of page table levels below PGD rather than double.

The direct mapping is set up during boot in `init_mem_mapping()` along with other initialisation tasks. This function uses the early boot memory allocator *memblocks* abstraction to determine the physical memory that needs to be mapped (omitting the 32-bit case and some code not relevant to the discussion), as shown in Listing 3-8.

```

738 void __init init_mem_mapping(void)
739 {
740     unsigned long end;
741
742     pti_check_boottime_disable();
743     probe_page_size_mask();
744     setup_pcid();
745     :
746     end = max_pfn << PAGE_SHIFT;
747     :
748     /* the ISA range is always mapped regardless of memory holes */
749     init_memory_mapping(0, ISA_END_ADDRESS, PAGE_KERNEL);
750
751     /* Init the trampoline, possibly with KASLR memory offset */
752     init_trampoline();
753
754     /*
755      * If the allocation is in bottom-up direction, we setup direct
756      * mapping
757      * in bottom-up, otherwise we setup direct mapping in top-down.
758      */
759     if (memblock_bottom_up()) {
760         unsigned long kernel_end = __pa_symbol(_end);
761
762         /*
763          * we need two separate calls here. This is because we want to
764          * allocate page tables above the kernel. So we first map

```



```

768         * [kernel_end, end) to make memory above the kernel be mapped
769         * as soon as possible. And then use page tables allocated
       above
770         * the kernel to map [ISA_END_ADDRESS, kernel_end).
771         */
772         memory_map_bottom_up(kernel_end, end);
773         memory_map_bottom_up(ISA_END_ADDRESS, kernel_end);
774     } else {
775         memory_map_top_down(ISA_END_ADDRESS, end);
776     }
       :
793 }

```

Listing 3-8: arch/x86/mm/init.c: Simplified `init_mem_mapping()`

The `pti_check_boottime_disable()`, `setup_pcid()`, and `init_trampoline()` functions are all used to mitigate the meltdown vulnerability and are out of scope for this discussion.

However, `probe_page_size_mask()` is relevant. This determines which page flags can be used via a mask as well as whether 2MiB and 1GiB huge pages are enabled. Let's examine this in Listing 3-9, stripping the irrelevant page flag mask code.

```

225 static void __init probe_page_size_mask(void)
226 {
227     /*
228      * For pagealloc debugging, identity mapping will use small pages.
229      * This will simplify cpa(), which otherwise needs to support
       splitting
230      * large pages into small in interrupt context, etc.
231      */
232     if (boot_cpu_has(X86_FEATURE_PSE) && !debug_pagealloc_enabled())
233         page_size_mask |= 1 << PG_LEVEL_2M;
234     else
235         direct_gbpages = 0;
236
237     /* Enable PSE if available */
238     if (boot_cpu_has(X86_FEATURE_PSE))
239         cr4_set_bits_and_update_boot(X86_CR4_PSE);
240     :
241
242     /* Enable 1 GB linear kernel mappings if available: */
243     if (direct_gbpages && boot_cpu_has(X86_FEATURE_GBPAGES)) {
244         printk(KERN_INFO "Using GB pages for direct mapping\n");
245         page_size_mask |= 1 << PG_LEVEL_1G;
246     } else {
247         direct_gbpages = 0;
248     }
249 }
250
251 }

```

Listing 3-9: arch/x86/mm/init.c: Simplified probe_page_size_mask()

First, we determine whether 2MiB pages are available by checking both the CPU flag which indicates this (X86_FEATURE_PSE) and the `debug_pagealloc_enabled()` function which indicates whether both `CONFIG_DEBUG_PAGEALLOC` is set and whether this debug mode is enabled for early memory initialization. Since any even vaguely modern x86-64 CPU will support 2MiB huge pages, the debug flag really is the deciding factor as to whether these huge pages will be available.

Next, if 2MiB pages are available (a prerequisite to having 1GiB huge pages available as well) and `X86_FEATURE_GBPAGES` is set (that is, the CPU supports 1GiB huge pages—again, any modern x86-64 CPU will), then we enable this page size.

Note that this function sets global variables `direct_gbpages` and `page_size_mask`, which are used in the rest of the early direct mapping logic.

We'll see where these pages are allocated from later. In the meantime, let's examine how mappings are performed. We have two choices:

1. Allocate bottom-up using `memory_map_bottom_up()`, starting from the end of the kernel image mapping, determined from the global variable `_end`, to the end of available physical memory determined from the global variable `max_pfn`, then from the end of the ISA mapping, `ISA_END_ADDRESS`, up to the end of the kernel mapping.
2. Allocate top-down using `memory_map_top_down()`, starting from the end of available physical memory to the end of the ISA mapping.

NOTE

The `max_pfn` global explicitly indicates the end of available physical memory, and sets the upper bound for the direct mapping. This is determined very early in the boot process.

The choice of which approach is used is based on *where we want page tables to be allocated*. If we allocate bottom-up, they end up immediately after the kernel image. If top-down, which is the default, then they are allocated as high as possible in memory. Which way we go is determined by `memblock_bottom_up()`.

NOTE

An example of a situation where we would allocate bottom-up is memory hotplug being enabled and the `movable_node` kernel command line parameter being set. This marks hot-pluggable memory as being entirely within `ZONE_MOVABLE`. So, in order to avoid unmovable page table allocations being placed in hotpluggable memory (which will be physically higher than the kernel image), we would then prefer to allocate page tables as low as possible. Note that the point at which the kernel image is loaded is the lowest point above which physical memory is broadly available without overlap).

Let's examine the bottom-up approach performed by `memory_map_bottom_up()` in Listing 3-10.

```

677 static void __init memory_map_bottom_up(unsigned long map_start,
678                                         unsigned long map_end)
679 {
680     unsigned long next, start;
681     unsigned long mapped_ram_size = 0;
682     /* step_size need to be small so pgd_buf from BRK could cover it */
683     unsigned long step_size = PMD_SIZE;
684
685     start = map_start;
686     min_pfn_mapped = start >> PAGE_SHIFT;
687     :
688
689     while (start < map_end) {
690         if (step_size && map_end - start > step_size) {
691             next = round_up(start + 1, step_size);
692             if (next > map_end)
693                 next = map_end;
694         } else {
695             next = map_end;
696         }
697
698         mapped_ram_size += init_range_memory_mapping(start, next);
699         start = next;
700
701         if (mapped_ram_size >= step_size)
702             step_size = get_new_step_size(step_size);
703     }
704 }

```

Listing 3-10: arch/x86/mm/init.c: *memory_map_bottom_up()*

We initialize direct mappings from `map_start` to the exclusive bound `map_end`. We allocate *up to* `step_size` of memory at a time. If our starting point is not aligned to `step_size`, then we start by allocating a block up to the next physical address aligned to this step size, and allocate `step_size` blocks up until we are a step size away from the end of the mapping (noting that step size can be increased over time!).

We start with a step size equal to `PMD_SIZE`, which is 2MiB. As the comment suggests, this is chosen to ensure the previously described `pgd_buf` has sufficient space for page table allocation. Since, in the worst case, we can only map a PTE directory's worth of *aligned* mappings before running out of space, and noting that we allocate up to the next address aligned to the step size, then this should be 2MiB (the amount of address space a PTE directory can map, with 512 4KiB entries).

You can see this logic in place at the beginning of the loop, with each block of memory allocated by `init_range_memory_mapping()`, and the next step size determined by `get_new_step_size()`. We'll come to the actual allocation shortly.

Next, let's examine how we determine the next step in `get_new_step_size()` in Listing 3-11.

```

583 static unsigned long __init get_new_step_size(unsigned long step_size)
584 {
585     /*
586      * Initial mapped size is PMD_SIZE (2M).
587      * We can not set step_size to be PUD_SIZE (1G) yet.
588      * In worse case, when we cross the 1G boundary, and
589      * PG_LEVEL_2M is not set, we will need 1+1+512 pages (2M + 8k)
590      * to map 1G range with PTE. Hence we use one less than the
591      * difference of page table level shifts.
592      *
593      * Don't need to worry about overflow in the top-down case, on 32bit,
594      * when step_size is 0, round_down() returns 0 for start, and that
595      * turns it into 0x100000000ULL.
596      * In the bottom-up case, round_up(x, 0) returns 0 though too, which
597      * needs to be taken into consideration by the code below.
598      */
599     return step_size << (PMD_SHIFT - PAGE_SHIFT - 1);
600 }

```

Listing 3-11: `arch/x86/mm/init.c: get_new_step_size()`

As the comment suggests, in the worst case of allocating 1GiB of address space with no huge page support where we are a crossing a page boundary, we will need to allocate 512 PTE pages or 2MiB, in addition to 4KiB for each higher level page table. The comment is incorrect for 5-level, as in this case crossing a 512 GiB boundary requires a new P4D, PUD, and PMD, which amounts to 12KiB of additional space.

Regardless, the solution is to simply divide by 2, requiring half as much as space as the worst case and ensuring that we have sufficient direct mapping to accommodate it.

We examine the opposite direction performed by `memory_map_top_down()` in Listing 3-12.

```

612 static void __init memory_map_top_down(unsigned long map_start,
613                                       unsigned long map_end)
614 {
615     unsigned long real_end, last_start;
616     unsigned long step_size;
617     unsigned long addr;
618     unsigned long mapped_ram_size = 0;
619
620     /*
621      * Systems that have many reserved areas near top of the memory,
622      * e.g. QEMU with less than 1G RAM and EFI enabled, or Xen, will
623      * require lots of 4K mappings which may exhaust pgt_buf.
624      * Start with top-most PMD_SIZE range aligned at PMD_SIZE to ensure

```

```

625     * there is enough mapped memory that can be allocated from
626     * memblock.
627     */
628     addr = memblock_phys_alloc_range(PMD_SIZE, PMD_SIZE, map_start,
629                                     map_end);
630     memblock_phys_free(addr, PMD_SIZE);
631     real_end = addr + PMD_SIZE;
632
633     /* step_size need to be small so pgt_buf from BRK could cover it */
634     step_size = PMD_SIZE;
635     max_pfn_mapped = 0; /* will get exact value next */
636     min_pfn_mapped = real_end >> PAGE_SHIFT;
637     last_start = real_end;
638
639     /*
640     * We start from the top (end of memory) and go to the bottom.
641     * The memblock_find_in_range() gets us a block of RAM from the
642     * end of RAM in [min_pfn_mapped, max_pfn_mapped) used as new pages
643     * for page table.
644     */
645     while (last_start > map_start) {
646         unsigned long start;
647
648         if (last_start > step_size) {
649             start = round_down(last_start - 1, step_size);
650             if (start < map_start)
651                 start = map_start;
652         } else
653             start = map_start;
654         mapped_ram_size += init_range_memory_mapping(start,
655                                                     last_start);
656         last_start = start;
657         min_pfn_mapped = last_start >> PAGE_SHIFT;
658         if (mapped_ram_size >= step_size)
659             step_size = get_new_step_size(step_size);
660     }
661
662     if (real_end < map_end)
663         init_range_memory_mapping(real_end, map_end);
664 }

```

Listing 3-12: arch/x86/mm/init.c: memory_map_top_down()

This is significantly more complicated than `memory_map_bottom_up()`, though it broadly follows the same approach, only working backwards.

First, this function does something quite unusual—it allocates physical memory from `memblock` (an early memory allocator, out of scope for the book), calling `memblock_phys_alloc_range()` for `PMD_SIZE` size and align-

ment, retrieves the allocated physical address and immediately frees it via `memblock_phys_free()`.

This is done not to actually allocate memory but to obtain the last *physically contiguous* block of `PMD_SIZE` (2MiB) size and alignment. One important thing to note here is that this, similar to the direct mapping allocation, will allocate top-down unless `memblock_bottom_up()` returns true, but if it returned true, we would be allocating bottom up, so we can safely make the assumption that this is what we'll get.

We do this, as the comment suggests, because there might be many reserved blocks of memory at the high end of physical memory. If this is so, we may end up trying to allocate a block of memory and only being able to allocate some small proportion of it, exhausting `pgt_buf` without sufficient space to store further page tables. By reserving an aligned 2MiB of space for page tables, we should have more than enough space.

From here, things proceed as they do with `memory_map_bottom_up()`, only in reverse, with one additional step of invoking `init_range_memory_mapping()` to map all memory past the `memblock` allocated range that was previously skipped.

Now that we have looked at how we determine which ranges of memory to allocate and in what order, along with how the direct memory mapping is bootstrapped, we need to look at how the mapping is actually performed. This is via `init_range_memory_mapping()`, which in turn invokes `init_memory_mapping()`.

Let's examine `init_range_memory_mapping()` in Listing 3-13.

```

542 /*
543  * We need to iterate through the E820 memory map and create direct mappings
544  * for only E820_TYPE_RAM and E820_KERN_RESERVED regions. We cannot simply
545  * create direct mappings for all pfns from [0 to max_low_pfn) and
546  * [4GB to max_pfn) because of possible memory holes in high addresses
547  * that cannot be marked as UC by fixed/variable range MTRRs.
548  * Depending on the alignment of E820 ranges, this may possibly result
549  * in using smaller size (i.e. 4K instead of 2M or 1G) page tables.
550  *
551  * init_mem_mapping() calls init_range_memory_mapping() with big range.
552  * That range would have hole in the middle or ends, and only ram parts
553  * will be mapped in init_range_memory_mapping().
554  */
555 static unsigned long __init init_range_memory_mapping(
556                                     unsigned long r_start,
557                                     unsigned long r_end)
558 {
559     unsigned long start_pfn, end_pfn;
560     unsigned long mapped_ram_size = 0;
561     int i;
562
563     for_each_mem_pfn_range(i, MAX_NUMNODES, &start_pfn, &end_pfn, NULL) {
564         u64 start = clamp_val(PFN_PHYS(start_pfn), r_start, r_end);

```

```

565         u64 end = clamp_val(PFN_PHYS(end_pfn), r_start, r_end);
566         if (start >= end)
567             continue;
568
569         /*
570          * if it is overlapping with brk pgt, we need to
571          * alloc pgt buf from memblock instead.
572          */
573         can_use_brk_pgt = max(start, (u64)pgt_buf_end<<PAGE_SHIFT) >=
574                             min(end, (u64)pgt_buf_top<<PAGE_SHIFT);
575         init_memory_mapping(start, end, PAGE_KERNEL);
576         mapped_ram_size += end - start;
577         can_use_brk_pgt = true;
578     }
579
580     return mapped_ram_size;
581 }

```

Listing 3-13: arch/x86/mm/init.c: *init_range_memory_mapping()*

This is the core method for mapping ranges of memory from *r_start* to the exclusive bound *r_end*. The key point here is that, by using *for_each_mem_pfn_range()*, this function only examines memory that is physically available in the specified range, excluding memory holes.

The start and end ranges of each block are *clamped* to the input range via *clamp_val()*, that is, kept between *r_start* and *r_end*. Since this clamps to an *inclusive* range and also because clamping could cause the start to exceed the end, that the range is still valid is checked immediately afterwards.

The global variable *can_use_brk_pgt* is set according to whether this space is exhausted, before the core memory mapping function *init_memory_mapping()* is called to perform the mapping itself, as shown in Listing 3-14.

```

519 unsigned long __ref init_memory_mapping(unsigned long start,
520                                         unsigned long end, pgprot_t prot)
521 {
522     struct map_range mr[NR_RANGE_MR];
523     unsigned long ret = 0;
524     int nr_range, i;
525
526     pr_debug("init_memory_mapping: [mem %#010lx-%#010lx]\n",
527             start, end - 1);
528
529     memset(mr, 0, sizeof(mr));
530     nr_range = split_mem_range(mr, 0, start, end);
531
532     for (i = 0; i < nr_range; i++)
533         ret = kernel_physical_mapping_init(mr[i].start, mr[i].end,
534                                             mr[i].page_size_mask,

```

```

535                                     prot);
536
537     add_pfn_range_mapped(start >> PAGE_SHIFT, ret >> PAGE_SHIFT);
538
539     return ret >> PAGE_SHIFT;
540 }

```

Listing 3-14: arch/x86/mm/init.c: *init_memory_mapping()*

This is the fundamental function which performs the mappings. Note that *init_mem_mapping()* starts by calling this directly for the ISA memory range because we know on the x86-64 architecture this whole range should be accessible.

Since we can map ranges using huge pages when a portion of the range is aligned to 2MiB or 1GiB, and that the range is not guaranteed to be aligned, we therefore might have to map at multiple page sizes. Considering the worst case, we may have to allocate 4KiB, 2MiB, 1GiB, 2MiB, and 4KiB pages, or five different page sizes, as shown in Figure 3-10.

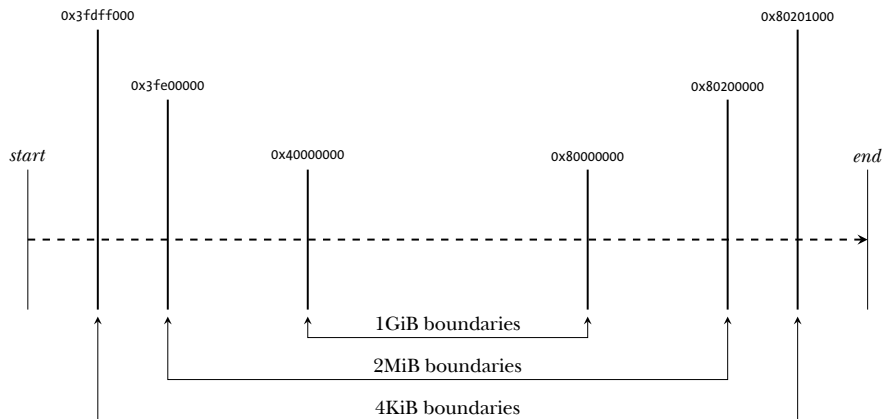


Figure 3-10: Worst-case direct mapping page size combination

We store this maximum number of *map ranges* in *NR_RANGE_MR*. We keep track of these memory ranges using the struct *map_range* data type as shown in Listing 3-15.

```

204 struct map_range {
205     unsigned long start;
206     unsigned long end;
207     unsigned page_size_mask;
208 };

```

Listing 3-15: arch/x86/mm/init.c: *struct map_range*

We maintain *NR_RANGE_MR* of these objects for every possible page size, populating these via *split_mem_range()*. These ranges will then be used for the actual page mapping. This therefore subdivides the input range into up to five memory ranges, saving each via *save_mr()* as shown in Listing 3-16.

```

311 static int __meminit save_mr(struct map_range *mr, int nr_range,
312                             unsigned long start_pfn, unsigned long end_pfn,
313                             unsigned long page_size_mask)
314 {
315     if (start_pfn < end_pfn) {
316         if (nr_range >= NR_RANGE_MR)
317             panic("run out of range for init_memory_mapping\n");
318         mr[nr_range].start = start_pfn<<PAGE_SHIFT;
319         mr[nr_range].end   = end_pfn<<PAGE_SHIFT;
320         mr[nr_range].page_size_mask = page_size_mask;
321         nr_range++;
322     }
323
324     return nr_range;
325 }

```

Listing 3-16: arch/x86/mm/init.c: save_mr()

The check to determine whether the range is actually valid is performed here, and the memory range is simply not saved if it is not valid. This means that we can calculate the range of memory up to the next aligned address and call `save_mr()` directly rather than having to check separately each time.

Additionally, range checking is performed, so a programming error which would otherwise cause a buffer overflow is avoided. The possibly incremented `nr_range` index is returned so the caller can iterate through the array for all valid ranges.

The `split_mem_range()` function starts by moving through page sizes from small to large, designated *head* pages, then large to small, designated *tail* pages, calling `save_mr()` for each to save if such a range exists. We will examine the head and tail portions of the function separately, starting with the head in Listing 3-17.

```

386 static int __meminit split_mem_range(struct map_range *mr, int nr_range,
387                                     unsigned long start,
388                                     unsigned long end)
389 {
390     unsigned long start_pfn, end_pfn, limit_pfn;
391     unsigned long pfn;
392     int i;
393
394     limit_pfn = PFN_DOWN(end);
395
396     /* head if not big page alignment ? */
397     pfn = start_pfn = PFN_DOWN(start);
398     :
410     end_pfn = round_up(pfn, PFN_DOWN(PMD_SIZE));
411
412     if (end_pfn > limit_pfn)

```

```

413         end_pfn = limit_pfn;
414     if (start_pfn < end_pfn) {
415         nr_range = save_mr(mr, nr_range, start_pfn, end_pfn, 0);
416         pfn = end_pfn;
417     }
418
419     /* big page (2M) range */
420     start_pfn = round_up(pfn, PFN_DOWN(PMD_SIZE));
421     :
422
423     end_pfn = round_up(pfn, PFN_DOWN(PUD_SIZE));
424     if (end_pfn > round_down(limit_pfn, PFN_DOWN(PMD_SIZE)))
425         end_pfn = round_down(limit_pfn, PFN_DOWN(PMD_SIZE));
426     :
427
428     if (start_pfn < end_pfn) {
429         nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
430                             page_size_mask & (1<<PG_LEVEL_2M));
431         pfn = end_pfn;
432     }
433     :
434
435     /* big page (1G) range */
436     start_pfn = round_up(pfn, PFN_DOWN(PUD_SIZE));
437     end_pfn = round_down(limit_pfn, PFN_DOWN(PUD_SIZE));
438     if (start_pfn < end_pfn) {
439         nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
440                             page_size_mask &
441                             ((1<<PG_LEVEL_2M)|(1<<PG_LEVEL_1G)));
442         pfn = end_pfn;
443     }
444

```

Listing 3-17: arch/x86/mm/init.c: *split_mem_range()* head pages

Examining the head page logic:

4KiB head pages Set pfn to start_pfn, that is, the PFN of the start argument and end_pfn to the next 2MiB aligned PFN (or the end PFN). If the range is valid, we save it, setting pfn to the beginning of the 2MiB page range.

2MiB head pages Set start_pfn to the first 2MiB aligned PFN and end_pfn to the first 1GiB aligned PFN in the range. If this is invalid then set end_pfn to the last 2MiB aligned PFN instead. If the range is valid, save it and update pfn to end_pfn.

1GiB head pages Set start_pfn to the first 1GiB aligned PFN and end_pfn to last 1GiB aligned PFN of limit_pfn, noting that this is an *exclusive* bound. This can therefore encompass multiple gigabyte-sized pages. If the range is valid, it is saved and pfn set to end_pfn. Both 2MiB and 1Gib page size flags are set.

Let's examine the remainder of *split_mem_range()* in Listing 3-18.

```

446     /* tail is not big page (1G) alignment */

```

```

447     start_pfn = round_up(pfn, PFN_DOWN(PMD_SIZE));
448     end_pfn = round_down(limit_pfn, PFN_DOWN(PMD_SIZE));
449     if (start_pfn < end_pfn) {
450         nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
451                             page_size_mask & (1<<PG_LEVEL_2M));
452         pfn = end_pfn;
453     }
454     :
455     /* tail is not big page (2M) alignment */
456     start_pfn = pfn;
457     end_pfn = limit_pfn;
458     nr_range = save_mr(mr, nr_range, start_pfn, end_pfn, 0);
459
460
461     if (!after_bootmem)
462         adjust_range_page_size_mask(mr, nr_range);
463
464     /* try to merge same page size and continuous */
465     for (i = 0; nr_range > 1 && i < nr_range - 1; i++) {
466         unsigned long old_start;
467         if (mr[i].end != mr[i+1].start ||
468             mr[i].page_size_mask != mr[i+1].page_size_mask)
469             continue;
470         /* move it */
471         old_start = mr[i].start;
472         memmove(&mr[i], &mr[i+1],
473                 (nr_range - 1 - i) * sizeof(struct map_range));
474         mr[i--].start = old_start;
475         nr_range--;
476     }
477
478     for (i = 0; i < nr_range; i++)
479         pr_debug(" [mem %#010lx-%#010lx] page %s\n",
480                 mr[i].start, mr[i].end - 1,
481                 page_size_string(&mr[i]));
482
483     return nr_range;
484 }

```

Listing 3-18: arch/x86/mm/init.c: *split_mem_range()* tail pages and merge

The tail page logic is as follows:

2MiB tail pages We set `start_pfn` to the next 2MiB aligned page, and `end_pfn` to the last 2MiB aligned page and save the range if valid, setting `pfn` to the end of the range if so.

4KiB tail pages We set `start_pfn` to where we left off previously and `end_pfn` to `limit_pfn` to pick up all remaining pages, saving the range if it is valid.

Note that `PFN_DOWN()` simply determines the PFN for the input physical address, rounding down. `round_up()` rounds up to the next multiple of the specified power-of-2 argument, that is aligns to it. Equally, `round_down()` rounds down to the next aligned value.

The technique of invoking `round_up()` or `round_down()` aligning to `PFN_DOWN(PMD_SIZE)` or `PFN_DOWN(PUD_SIZE)` causes the value to be aligned to either 2MiB or 1GiB, rounding up or down. This is a key part of how `split_mem_range()` operates, as it is subdividing into *aligned* blocks of pages at each possible size.

Also note that each time we invoke `save_mr()` we apply the global `page_size_mask`, so we may actually end up with pages aligned to huge page sizes but with the `page_size_mask` field set to a lower page size.

Once we've extracted all of the page size-aligned memory ranges, we need to perform a few more steps. The first step involves invoking `adjust_range_page_size_mask()`, which we'll examine in Listing 3-19.

NOTE

The `adjust_range_page_size_mask()` function is only called if the global flag `after_bootmem` is not set. This flag denotes that the buddy allocator is available and the early stage memory allocator is no longer being used—however, this is not the case when the direct mapping is being initialized, which we're examining, so we can assume not.

```

331 static void __ref adjust_range_page_size_mask(struct map_range *mr,
332                                             int nr_range)
333 {
334     int i;
335
336     for (i = 0; i < nr_range; i++) {
337         if ((page_size_mask & (1<<PG_LEVEL_2M)) &&
338             !(mr[i].page_size_mask & (1<<PG_LEVEL_2M))) {
339             unsigned long start = round_down(mr[i].start, PMD_SIZE
340
341         );
342
343             unsigned long end = round_up(mr[i].end, PMD_SIZE);
344             :
345
346             if (memblock_is_region_memory(start, end - start))
347                 mr[i].page_size_mask |= 1<<PG_LEVEL_2M;
348         }
349
350         if ((page_size_mask & (1<<PG_LEVEL_1G)) &&
351             !(mr[i].page_size_mask & (1<<PG_LEVEL_1G))) {
352             unsigned long start = round_down(mr[i].start, PUD_SIZE
353
354         );
355
356             unsigned long end = round_up(mr[i].end, PUD_SIZE);
357             :
358
359             if (memblock_is_region_memory(start, end - start))
360                 mr[i].page_size_mask |= 1<<PG_LEVEL_1G;
361         }
362     }
363 }
```

359 }

Listing 3-19: arch/x86/mm/init.c: `adjust_range_page_size_mask()`

This checks each memory range to see if it is contiguous to physical memory, possibly beyond the input range, which allows us to map it as a larger huge page even if it has not been flagged as such so far. This check is performed by `memblock_is_region_memory()`.

If so, then we re-flag it at the appropriate huge page size and assume the low-level mapping logic will utilize existing huge page mappings if they already exist.

For example, in the case where we adjust a 4KiB page range to 2MiB, then the next memory range tries to map the portion of the original that has now been remapped to 2MiB will note that a huge page mapping already exists.

Finally, we *merge* any adjacent memory ranges of the same page size mask. This can happen when the global page size mask disallows certain page sizes, and head and tail page ranges end up next to each other. For example, if 1GiB pages are not available, then we might map head 2MiB pages to the first 1GiB aligned PFN and tail 2MiB pages to the last 2MiB aligned PFN.

Going back to `init_memory_mapping()` in Listing 3-14), once we have split the memory ranges and set up our struct `map_range` array, we then need to create the actual mappings via `kernel_physical_mapping_init()`, as shown in Listing 3-20.

```

781 unsigned long __meminit
782 kernel_physical_mapping_init(unsigned long paddr_start,
783                             unsigned long paddr_end,
784                             unsigned long page_size_mask, pgprot_t prot)
785 {
786     return __kernel_physical_mapping_init(paddr_start, paddr_end,
787                                         page_size_mask, prot, true);
788 }
```

Listing 3-20: arch/x86/mm/init_64.c: `kernel_physical_mapping_init()`

This simply invokes `__kernel_physical_mapping_init()` with the `init` argument set to true, with this function doing the heavy lifting of the actual mapping.

Note that all of these kernel mappings utilize the struct `mm_struct` global value `init_mm` for keeping track of the mappings. We'll come onto struct `mm_struct` in Chapter 4 on Process Memory in considerable detail but for now consider it to be where mapping state is contained.

Next let's examine `__kernel_physical_mapping_init()` in Listing 3-21.

```

724 static unsigned long __meminit
725 __kernel_physical_mapping_init(unsigned long paddr_start,
726                               unsigned long paddr_end,
727                               unsigned long page_size_mask,
```

```

728         pgprot_t prot, bool init)
729 {
730     bool pgd_changed = false;
731     unsigned long vaddr, vaddr_start, vaddr_end, vaddr_next, paddr_last;
732
733     paddr_last = paddr_end;
734     vaddr = (unsigned long)__va(paddr_start);
735     vaddr_end = (unsigned long)__va(paddr_end);
736     vaddr_start = vaddr;
737
738     for (; vaddr < vaddr_end; vaddr = vaddr_next) {
739         pgd_t *pgd = pgd_offset_k(vaddr);
740         p4d_t *p4d;
741
742         vaddr_next = (vaddr & PGDIR_MASK) + PGDIR_SIZE;
743
744         if (pgd_val(*pgd)) {
745             p4d = (p4d_t *)pgd_page_vaddr(*pgd);
746             paddr_last = phys_p4d_init(p4d, __pa(vaddr),
747                                     __pa(vaddr_end),
748                                     page_size_mask,
749                                     prot, init);
750             continue;
751         }
752
753         p4d = alloc_low_page();
754         paddr_last = phys_p4d_init(p4d, __pa(vaddr), __pa(vaddr_end),
755                                 page_size_mask, prot, init);
756
757         spin_lock(&init_mm.page_table_lock);
758         if (pgtable_l5_enabled())
759             pgd_populate_init(&init_mm, pgd, p4d, init);
760         else
761             p4d_populate_init(&init_mm, p4d_offset(pgd, vaddr),
762                             (pud_t *) p4d, init);
763
764         spin_unlock(&init_mm.page_table_lock);
765         pgd_changed = true;
766     }
767
768     if (pgd_changed)
769         sync_global_pgds(vaddr_start, vaddr_end - 1);
770
771     return paddr_last;
772 }

```

Listing 3-21: arch/x86/mm/init_64.c: __kernel_physical_mapping_init()

We iterate over the virtual address range of the input physical range that the direct mapping is intended to provide, incrementing the PGD index on each iteration.

We invoke `pgd_offset_k()` to retrieve the PGD entry containing the address of the P4D directory for each virtual address. If this entry is already populated, then instead of allocating, we simply retrieve the virtual address of this page via `pgd_page_vaddr()`, and populate it for our address range via `phys_p4d_init()`.

Otherwise, we allocate a new page via `alloc_low_page()`, initialize as discussed previously, and assign via `p4d_populate_init()`, which invokes the `_safe` variant of `p4d_populate()`, `p4d_populate_safe()` during memory initialization. This achieves the same as the “unsafe” version except it checks to ensure that a page table entry is not being overwritten with something different. The populations differs based on whether there are 5 page table levels or 4, because with 4 the level can be treated as a PUD rather than a P4D.

Finally, if a PGD entry is changed, `sync_global_pgds()` is invoked.

For kernel mappings, PGD changes are a problem as each process has its own separate PGD table so changes which affect PGD mappings must be synced between them all.

Additionally, if PTI is enabled then each PGD will be a separate kernel page table, if not then they are each shared with userspace mappings.

NOTE

PGD synchronisation is only a concern during kernel initialization. Once the direct mapping is set up and the memory management subsystem is established, PGDs do not need to be synchronized again.

While setting up mappings, we move through each page level performing similar actions. We’ll look at each level in detail, but let’s take some time to examine `sync_global_pgds()` and `alloc_low_page()`, starting with the former, shown in Listing 3-22.

```

208 /*
209  * When memory was added make sure all the processes MM have
210  * suitable PGD entries in the local PGD level page.
211  */
212 static void sync_global_pgds(unsigned long start, unsigned long end)
213 {
214     if (pgtable_15_enabled())
215         sync_global_pgds_15(start, end);
216     else
217         sync_global_pgds_14(start, end);
218 }
```

Listing 3-22: arch/x86/mm/init_64.c: `sync_global_pgds()`

There is separate logic for 5-level and 4-level. As they are very similar, we’ll examine the 5-level logic only. All PGDs are stored on the `pgd_list` linked list, which is added to via `pgd_list_add()` and removed from via `pgd_list_del()`.

PGDs are added to these lists when they are constructed in `pgd_ctor()` (which is invoked in turn by `pgd_alloc()`) and removed when they are destructed in `pgd_dtor()` (which is invoked in turn by `pgd_free()`).

Let's examine `sync_global_pgds_15()` in Listing 3-23.

```

127 static void sync_global_pgds_15(unsigned long start, unsigned long end)
128 {
129     unsigned long addr;
130
131     for (addr = start; addr <= end; addr = ALIGN(addr + 1, PGDIR_SIZE)) {
132         const pgd_t *pgd_ref = pgd_offset_k(addr);
133         struct page *page;
134
135         /* Check for overflow */
136         if (addr < start)
137             break;
138
139         if (pgd_none(*pgd_ref))
140             continue;
141
142         spin_lock(&pgd_lock);
143         list_for_each_entry(page, &pgd_list, lru) {
144             pgd_t *pgd;
145             spinlock_t *pgt_lock;
146
147             pgd = (pgd_t *)page_address(page) + pgd_index(addr);
148             /* the pgt_lock only for Xen */
149             pgt_lock = &pgd_page_get_mm(page)->page_table_lock;
150             spin_lock(pgt_lock);
151
152             if (!pgd_none(*pgd_ref) && !pgd_none(*pgd))
153                 BUG_ON(pgd_page_vaddr(*pgd) != pgd_page_vaddr
154                       (*pgd_ref));
155
156             if (pgd_none(*pgd))
157                 set_pgd(pgd, *pgd_ref);
158
159             spin_unlock(pgt_lock);
160         }
161         spin_unlock(&pgd_lock);
162     }

```

Listing 3-23: `arch/x86/mm/init_64.c: sync_global_pgds_15()`

This function takes the *current* kernel PGD and synchronizes all mappings from it to all other PGDs in the range `start` to `end`, *inclusive*. The current reference PGD entry is stored in `pgd_ref`. The global `pgd_lock` is used to protect accesses to `pgd_list`.

For present entries, we retrieve the address of the PGD via `page_address()`, which in turn invokes `page_to_virt()` and `__va()`. In addition, we can determine the offset at which a virtual address lies within the PGD via `pgd_index()`.

After taking appropriate locks, we check the edge case where both the source and target PGD entry are populated. In this instance, we cannot permit replacing the referenced P4D with another (and thus possibly leaking the existing mapping). We therefore raise a kernel bug (and under these circumstances, likely a panic) if anything but the flags are changed.

Finally, we set the entry via `set_pgd()` and continue the process for all other entries in the range.

Returning to `__kernel_physical_mapping_init()`, we see that it invokes `alloc_low_page()` to perform the actual allocation of pages, which in turn itself invokes `alloc_low_pages()` with `num` set to 1, as shown in Listing 3-24.

```

123 _ref void *alloc_low_pages(unsigned int num)
124 {
125     unsigned long pfn;
126     int i;
127
128     if (after_bootmem) {
129         unsigned int order;
130
131         order = get_order((unsigned long)num << PAGE_SHIFT);
132         return (void *)__get_free_pages(GFP_ATOMIC | __GFP_ZERO, order
133     );
134     }
135
136     if ((pgt_buf_end + num) > pgt_buf_top || !can_use_brk_pgt) {
137         unsigned long ret = 0;
138
139         if (min_pfn_mapped < max_pfn_mapped) {
140             ret = memblock_phys_alloc_range(
141                 PAGE_SIZE * num, PAGE_SIZE,
142                 min_pfn_mapped << PAGE_SHIFT,
143                 max_pfn_mapped << PAGE_SHIFT);
144         }
145         if (!ret && can_use_brk_pgt)
146             ret = __pa(extend_brk(PAGE_SIZE * num, PAGE_SIZE));
147
148         if (!ret)
149             panic("alloc_low_pages: can not alloc memory");
150
151         pfn = ret >> PAGE_SHIFT;
152     } else {
153         pfn = pgt_buf_end;
154         pgt_buf_end += num;
155     }

```

```

156     for (i = 0; i < num; i++) {
157         void *adr;
158
159         adr = __va((pfn + i) << PAGE_SHIFT);
160         clear_page(adr);
161     }
162
163     return __va(pfn << PAGE_SHIFT);
164 }

```

Listing 3-24: arch/x86/mm/init.c: alloc_low_pages()

Again, we can assume that `after_bootmem` is not set as we are initializing the direct mapping. We then see whether we can use the PGT buffer. If not, we allocate using `memblock` from within the range specified by the globals `min_pfn_mapped` and `max_pfn_mapped`. These are updated and expanded as the direct mapping expands. If no memory can be allocated, then the kernel panics.

If memory is allocated from the `pgt_buf`, then it's as simple as expanding `pgt_buf_end`. After this we clear pages via `clear_page()` and return the virtual address of the first.

Now let's examine the next stage of this process in `phys_p4d_init()`, shown in Listing 3-25.

```

673 static unsigned long __meminit
674 phys_p4d_init(p4d_t *p4d_page, unsigned long paddr, unsigned long paddr_end,
675              unsigned long page_size_mask, pgprot_t prot, bool init)
676 {
677     unsigned long vaddr, vaddr_end, vaddr_next, paddr_next, paddr_last;
678
679     paddr_last = paddr_end;
680     vaddr = (unsigned long)__va(paddr);
681     vaddr_end = (unsigned long)__va(paddr_end);
682
683     if (!pgtable_l5_enabled())
684         return phys_pud_init((pud_t *) p4d_page, paddr, paddr_end,
685                             page_size_mask, prot, init);
686
687     for (; vaddr < vaddr_end; vaddr = vaddr_next) {
688         p4d_t *p4d = p4d_page + p4d_index(vaddr);
689         pud_t *pud;
690
691         vaddr_next = (vaddr & P4D_MASK) + P4D_SIZE;
692         paddr = __pa(vaddr);
693
694         if (paddr >= paddr_end) {
695             paddr_next = __pa(vaddr_next);
696             if (!after_bootmem &&
697                 !e820__mapped_any(paddr & P4D_MASK, paddr_next,

```

```

698                                     E820_TYPE_RAM) &&
699                                     !e820__mapped_any(paddr & P4D_MASK, paddr_next,
700                                     E820_TYPE_RESERVED_KERN))
701                                     set_p4d_init(p4d, __p4d(0), init);
702                                     continue;
703                                     }
704
705                                     if (!p4d_none(*p4d)) {
706                                         pud = pud_offset(p4d, 0);
707                                         paddr_last = phys_pud_init(pud, paddr, __pa(vaddr_end)
708                                     ,
709                                     page_size_mask, prot, init);
710                                     continue;
711                                     }
712
713                                     pud = alloc_low_page();
714                                     paddr_last = phys_pud_init(pud, paddr, __pa(vaddr_end),
715                                     page_size_mask, prot, init);
716
717                                     spin_lock(&init_mm.page_table_lock);
718                                     p4d_populate_init(&init_mm, p4d, pud, init);
719                                     spin_unlock(&init_mm.page_table_lock);
720                                     }
721
722                                     return paddr_last;
723     }

```

Listing 3-25: arch/x86/mm/init_64.c: phys_p4d_init()

This is broadly similar to `__kernel_physical_mapping_init()` but has a number of differences.

If 5-level page tables are disabled, we simply skip to the PUD level. Otherwise, we iterate through the virtual address range, with each step moving to the next entry in the P4D. Similar to `__kernel_physical_mapping_init()`, we reuse the existing mapping if it is in place. Otherwise, we allocate a new PUD via `alloc_low_page()` and populate the P4D entry while holding the shared `page_table_lock`.

However, a key difference here is we explicitly check for the situation where, somehow, the physical address is out of range. If it is, we clear the P4D entry if and only if there is no RAM or kernel reserved memory as specified by the E820 PC architecture memory map via `e820__mapped_any()`—this is used to explicitly clear page table entries for memory holes (places where no accessible physical memory exists).

NOTE

E820 refers to a BIOS call, which is used on PCs early in boot to get a physical memory map. e820 refers to what to place in the AX register to invoke the call.

The `phys_p4d_init()` function returns the exclusive bound of physically mapped memory in any case, which is updated when `phys_pud_init()` is invoked to initialize the underlying PUD.

One thing to note with the P4D initialization that differs from others is the use of virtual addresses to iterate through the memory. This is because the `PAGE_OFFSET` addressed used to offset physical addresses in the direct mapping to find virtual ones is not guaranteed to be P4D aligned due to restrictions on the maximum KASLR offset.

The code therefore uses a conversion to and from virtual addresses to ensure that when dealing with physical addresses they are always aligned and to remove any implicit assumption about alignment from the code. This was changed in commit 432c833218dd.

We'll examine the initial part of `phys_pud_init()` in Listing 3-26.

```

586 static unsigned long __meminit
587 phys_pud_init(pud_t *pud_page, unsigned long paddr, unsigned long paddr_end,
588              unsigned long page_size_mask, pgprot_t _prot, bool init)
589 {
590     unsigned long pages = 0, paddr_next;
591     unsigned long paddr_last = paddr_end;
592     unsigned long vaddr = (unsigned long)__va(paddr);
593     int i = pud_index(vaddr);
594
595     for (; i < PTRS_PER_PUD; i++, paddr = paddr_next) {

```

Listing 3-26: arch/x86/mm/init_64.c: phys_pud_init() preamble and loop

Note that here we loop through all entries in the PUD starting at the beginning of the range, even if we extend beyond the range, for which we have special handling.

Next we initialize some values and check to see if we're beyond the specified range—if so, we clear any entries that may exist for ranges which do not have any accessible memory, or memory holes. As we are initializing memory ranges sequentially, we are opportunistically ensuring that we clear any potentially incorrect page table entries spanning memory holes as we go, which is shown in Listing 3-27.

```

596     pud_t *pud;
597     pmd_t *pmd;
598     pgprot_t prot = _prot;
599
600     vaddr = (unsigned long)__va(paddr);
601     pud = pud_page + pud_index(vaddr);
602     paddr_next = (paddr & PUD_MASK) + PUD_SIZE;
603
604     if (paddr >= paddr_end) {
605         if (!after_bootmem &&
606             !e820_mapped_any(paddr & PUD_MASK, paddr_next,
607                             E820_TYPE_RAM) &&

```

```

608             !e820__mapped_any(paddr & PUD_MASK, paddr_next,
609                               E820_TYPE_RESERVED_KERN))
610             set_pud_init(pud, __pud(0), init);
611         continue;
612     }

```

Listing 3-27: arch/x86/mm/init_64.c: phys_pud_init() loop initialisation and PA check

Next, if we are reusing an existing PUD entry, we have to consider 1GiB huge pages and the ordinary 2MiB/4KiB case. If the PUD entry isn't large, we proceed as before. If it is, we have to check to see if the page size mask permits 1GiB huge pages. If so, we simply continue as normal. If not, we enter into an edge case where we need to clear the huge page entry. We examine this in Listing 3-28.

```

614         if (!pud_none(*pud)) {
615             if (!pud_large(*pud)) {
616                 pmd = pmd_offset(pud, 0);
617                 paddr_last = phys_pmd_init(pmd, paddr,
618                                           paddr_end,
619                                           page_size_mask,
620                                           prot, init);
621                 continue;
622             }
623             /*
624              * If we are ok with PG_LEVEL_1G mapping, then we will
625              * use the existing mapping.
626              *
627              * Otherwise, we will split the gbpge mapping but use
628              * the same existing protection bits except for large
629              * page, so that we don't violate Intel's TLB
630              * Application note (317080) which says, while
631              *
632              * the page sizes, new and old translations should
633              * not differ with respect to page frame and
634              * attributes.
635              */
636             if (page_size_mask & (1 << PG_LEVEL_1G)) {
637                 if (!after_bootmem)
638                     pages++;
639                 paddr_last = paddr_next;
640                 continue;
641             }
642             prot = pte_pgprot(pte_clrhuge(*(pte_t *)pud));

```

Listing 3-28: arch/x86/mm/init_64.c: phys_pud_init() existing entry

If there is no existing entry, we have to either mark the page as huge or allocate a new PMD and assign it. We try to use a 1GiB huge page if specified to do so by the `page_size_mask` argument, resetting the `paddr_last` value to take this into account. If we are not permitted to use a huge page, we allocate and assign as before, calling into `phys_pmd_init()`, shown in Listing 3-31.

Next let's examine the case of a new entry in `phys_pud_init()` shown in Listing 3-29.

```

644         if (page_size_mask & (1<<PG_LEVEL_1G)) {
645             pages++;
646             spin_lock(&init_mm.page_table_lock);
647
648             prot = __pgprot(pgprot_val(prot) | _PAGE_PSE);
649
650             set_pte_init((pte_t *)pud,
651                          pfn_pte((paddr & PUD_MASK) >> PAGE_SHIFT,
652                                prot),
653                          init);
654             spin_unlock(&init_mm.page_table_lock);
655             paddr_last = paddr_next;
656             continue;
657         }
658
659         pmd = alloc_low_page();
660         paddr_last = phys_pmd_init(pmd, paddr, paddr_end,
661                                  page_size_mask, prot, init);
662
663         spin_lock(&init_mm.page_table_lock);
664         pud_populate_init(&init_mm, pud, pmd, init);
665         spin_unlock(&init_mm.page_table_lock);
666     }

```

Listing 3-29: arch/x86/mm/init_64.c: phys_pud_init() new entry

Finally, we keep a track of the number of 1GiB huge pages allocated via the `pages` variable, which we use to update stats with before returning the last mapped physical address, `paddr_last`, as shown in Listing 3-30.

```

668         update_page_count(PG_LEVEL_1G, pages);
669
670         return paddr_last;
671     }

```

Listing 3-30: arch/x86/mm/init_64.c: phys_pud_init() after loop

The structure of `phys_pmd_init()` is very similar to `phys_pud_init()` as shown in Listing 3-31.

```

501 static unsigned long __meminit
502 phys_pmd_init(pmd_t *pmd_page, unsigned long paddr, unsigned long paddr_end,

```

```

503         unsigned long page_size_mask, pgprot_t prot, bool init)
504     {
505         unsigned long pages = 0, paddr_next;
506         unsigned long paddr_last = paddr_end;
507
508         int i = pmd_index(paddr);
509
510         for (; i < PTRS_PER_PMD; i++, paddr = paddr_next) {
511             pmd_t *pmd = pmd_page + pmd_index(paddr);
512             pte_t *pte;
513             pgprot_t new_prot = prot;
514
515             paddr_next = (paddr & PMD_MASK) + PMD_SIZE;
516             if (paddr >= paddr_end) {
517                 if (!after_bootmem &&
518                     !e820__mapped_any(paddr & PMD_MASK, paddr_next,
519                                         E820_TYPE_RAM) &&
520                     !e820__mapped_any(paddr & PMD_MASK, paddr_next,
521                                         E820_TYPE_RESERVED_KERN))
522                     set_pmd_init(pmd, __pmd(0), init);
523                 continue;
524             }
525
526             if (!pmd_none(*pmd)) {
527                 if (!pmd_large(*pmd)) {
528                     spin_lock(&init_mm.page_table_lock);
529                     pte = (pte_t *)pmd_page_vaddr(*pmd);
530                     paddr_last = phys_pte_init(pte, paddr,
531                                                 paddr_end, prot,
532                                                 init);
533                     spin_unlock(&init_mm.page_table_lock);
534                     continue;
535                 }
536                 /*
537                  * If we are ok with PG_LEVEL_2M mapping, then we will
538                  * use the existing mapping,
539                  *
540                  * Otherwise, we will split the large page mapping but
541                  * use the same existing protection bits except for
542                  * large page, so that we don't violate Intel's TLB
543                  * Application note (317080) which says, while
544
545                  changing
546
547                  * the page sizes, new and old translations should
548                  * not differ with respect to page frame and
549                  * attributes.
550                  */
551                 if (page_size_mask & (1 << PG_LEVEL_2M)) {

```

```

549             if (!after_bootmem)
550                 pages++;
551             paddr_last = paddr_next;
552             continue;
553         }
554         new_prot = pte_pgprot(pte_clrhuge(*(pte_t *)pmd));
555     }
556
557     if (page_size_mask & (1<<PG_LEVEL_2M)) {
558         pages++;
559         spin_lock(&init_mm.page_table_lock);
560         set_pte_init((pte_t *)pmd,
561                     pfn_pte((paddr & PMD_MASK) >> PAGE_SHIFT,
562                             __pgprot(pgprot_val(prot) |
563                                     _PAGE_PSE)),
564                     init);
565         spin_unlock(&init_mm.page_table_lock);
566         paddr_last = paddr_next;
567         continue;
568     }
569
570     pte = alloc_low_page();
571     paddr_last = phys_pte_init(pte, paddr, paddr_end, new_prot,
572                               init);
573
574     spin_lock(&init_mm.page_table_lock);
575     pmd_populate_kernel_init(&init_mm, pmd, pte, init);
576     spin_unlock(&init_mm.page_table_lock);
577 }
578
579     update_page_count(PG_LEVEL_2M, pages);
580     return paddr_last;
581 }

```

Listing 3-31: arch/x86/mm/init_64.c: phys_pmd_init()

The only difference is that 2MiB huge pages are considered and the populate command is named `pmd_populate_kernel_init()` with `_kernel_` explicitly labelled.

Finally, let's consider the far simpler `phys_pte_init()`, shown in Listing 3-32.

```

447 static unsigned long __meminit
448 phys_pte_init(pte_t *pte_page, unsigned long paddr, unsigned long paddr_end,
449              pgprot_t prot, bool init)
450 {
451     unsigned long pages = 0, paddr_next;
452     unsigned long paddr_last = paddr_end;
453     pte_t *pte;
454     int i;

```



```

455
456     pte = pte_page + pte_index(paddr);
457     i = pte_index(paddr);
458
459     for (; i < PTRS_PER_PTE; i++, paddr = paddr_next, pte++) {
460         paddr_next = (paddr & PAGE_MASK) + PAGE_SIZE;
461         if (paddr >= paddr_end) {
462             if (!after_bootmem &&
463                 !e820__mapped_any(paddr & PAGE_MASK, paddr_next,
464                                     E820_TYPE_RAM) &&
465                 !e820__mapped_any(paddr & PAGE_MASK, paddr_next,
466                                     E820_TYPE_RESERVED_KERN))
467                 set_pte_init(pte, __pte(0), init);
468             continue;
469         }
470
471         /*
472          * We will re-use the existing mapping.
473          * Xen for example has some special requirements, like mapping
474          * pagetable pages as RO. So assume someone who pre-setup
475          * these mappings are more intelligent.
476          */
477         if (!pte_none(*pte)) {
478             if (!after_bootmem)
479                 pages++;
480             continue;
481         }
482
483         if (0)
484             pr_info("    pte=%p addr=%lx pte=%016lx\n", pte, paddr,
485                     pfn_pte(paddr >> PAGE_SHIFT, PAGE_KERNEL).pte);
486
487         ;
488         pages++;
489         set_pte_init(pte, pfn_pte(paddr >> PAGE_SHIFT, prot), init);
490         paddr_last = (paddr & PAGE_MASK) + PAGE_SIZE;
491     }
492
493     update_page_count(PG_LEVEL_4K, pages);
494
495     return paddr_last;
496 }

```

Listing 3-32: arch/x86/mm/init_64.c: *phys_pte_init()*

This is similar to the other functions, only with no need to consider huge pages or populating/allocating any further page tables.

In each case of 1GiB, 2MiB huge pages, and 4KiB non-huge pages we initialize the final entry via `set_pte_init()` (this and the other `set_*_init` func-

tions being generated by macros), using `pfn_pte()` to encode the data page physical address and page flags.

While we went into considerable detail in this section to explain the initialization of direct mapping, it is a very useful exercise not only in understanding how this is initialized but also to see page table manipulation in action.

3.4 An Introduction to the Transaction Lookaside Buffer

Traversing page tables is expensive, so the hardware provides caches to speed up translation of virtual addresses. The most important of these is *Transaction Lookaside Buffer (TLB)*, which is in effect a hash between virtual and physical addresses, permitting the hardware to avoid a page table traversal when looking up a virtual address.

As we modify page tables it is critical to ensure that any pre-existing TLB entries are cleared in order that incorrect address translations do not occur. In x86-64, the current PGD is set in the CR3 control register. When this is changed the TLB is automatically flushed, so context switching between processes achieves a TLB flush along side it, except for mappings marked with the `_PAGE_GLOBAL` flag.

When a single page's mapping is invalidated, however, we would rather simply clear any existent TLB entry for it. In x86-64 this is done via the `invlpg` instruction and ultimately performed by `native_flush_tlb_one_user()`, shown in Listing 3-33.

NOTE

Note that this makes reference to the x86-64 feature Process Context Identifiers (PCIDs), which is used by the Page Table Isolation (PTI) functionality for mitigation against the meltdown vulnerability. Discussion of this is out of scope here.

```

1120 STATIC_NOPV void native_flush_tlb_one_user(unsigned long addr)
1121 {
1122     u32 loaded_mm_asid = this_cpu_read(cpu_tlbstate.loaded_mm_asid);
1123
1124     asm volatile("invlpg (%0)" :: "r" (addr) : "memory");
1125
1126     if (!static_cpu_has(X86_FEATURE_PTID))
1127         return;
1128
1129     /*
1130      * Some platforms #GP if we call invpcid(type=1/2) before CR4.PCIDE=1.
1131      * Just use invalidate_user_asid() in case we are called early.
1132      */
1133     if (!this_cpu_has(X86_FEATURE_INVPCID_SINGLE))
1134         invalidate_user_asid(loaded_mm_asid);
1135     else
1136         invpcid_flush_one(user_pcid(loaded_mm_asid), addr);
1137 }
```

Listing 3-33: arch/x86/mm/tlb.c: *native_flush_tlb_one_user()*

The `native_flush_tlb_one_user()` function is called by `__flush_tlb_one_user()`, which is called in turn by `flush_tlb_one_user()`. This is invoked by many different call paths for TLB cache invalidation for both kernel and userland. The alternative, global flush, is ultimately performed by `native_flush_tlb_global()` that either uses PCID invalidation or writes CR3 to flush globally.

The use of huge pages can make a very big difference in TLB performance—since huge pages span a far wider range of memory, the entries cached in the TLB face far less contention. For example, 2MiB huge pages span 512 4KiB pages, meaning a potentially huge reduction in the number of page table walks required. 1GiB huge pages span 262,144 4KiB pages with a further order of magnitude reduction in TLB contention under ideal circumstances.

The discussion is kept intentionally brief here as we will go into more detail on TLB maintenance in Chapter 7 on the Reverse Mapping.

3.5 The Kernel Virtual Memory Allocator (`vmalloc`)

Allocating memory in the kernel is typically performed using the slab allocator via `kmalloc()` (slab allocation is out of scope for the book), but this acts solely as a *physical* memory allocator and uses the direct mapping to return a virtual address. Since contiguous physical memory is in short supply, trying to allocate larger blocks of memory using the slab allocator may not be entirely useful or wise.

The kernel provides a mechanism for the allocation of *virtually* contiguous memory, *vmalloc*, allocating via `vmalloc()` and freeing via `vfree()`.

The `vmalloc()` function accepts an arbitrary size parameter in bytes for which sufficient underlying pages will be allocated, with page tables generated to span the range.

Additionally, existing physical memory can be mapped and unmapped via `vmap()` and `vunmap()`.

Let's examine `vmalloc()` in Listing 3-34.

```

3280 /**
3281  * vmalloc - allocate virtually contiguous memory
3282  * @size:    allocation size
3283  *
3284  * Allocate enough pages to cover @size from the page level
3285  * allocator and map them into contiguous kernel virtual space.
3286  *
3287  * For tight control over page level allocator and protection flags
3288  * use __vmalloc() instead.
3289  *
3290  * Return: pointer to the allocated memory or %NULL on error
3291  */

```

```

3292 void *vmalloc(unsigned long size)
3293 {
3294     return __vmalloc_node(size, 1, GFP_KERNEL, NUMA_NO_NODE,
3295                          __builtin_return_address(0));
3296 }

```

Listing 3-34: mm/vmalloc.c: vmalloc()

If a caller wishes to allocate memory that is automatically zeroed, `vzalloc()` provides this functionality, as shown in Listing 3-35.

```

3332 void *vzalloc(unsigned long size)
3333 {
3334     return __vmalloc_node(size, 1, GFP_KERNEL | __GFP_ZERO, NUMA_NO_NODE,
3335                          __builtin_return_address(0));
3336 }

```

Listing 3-35: mm/vmalloc.c: vzalloc()

Each of these invoke `__vmalloc_node()`, which is shown in Listing 3-36.

```

3239 /**
3240  * __vmalloc_node - allocate virtually contiguous memory
3241  * @size:          allocation size
3242  * @align:         desired alignment
3243  * @gfp_mask:      flags for the page level allocator
3244  * @node:          node to use for allocation or NUMA_NO_NODE
3245  * @caller:        caller's return address
3246  *
3247  * Allocate enough pages to cover @size from the page level allocator with
3248  * @gfp_mask flags. Map them into contiguous kernel virtual space.
3249  *
3250  * Reclaim modifiers in @gfp_mask - __GFP_NORETRY, __GFP_RETRY_MAYFAIL
3251  * and __GFP_NOFAIL are not supported
3252  *
3253  * Any use of gfp flags outside of GFP_KERNEL should be consulted
3254  * with mm people.
3255  *
3256  * Return: pointer to the allocated memory or %NULL on error
3257  */
3258 void *__vmalloc_node(unsigned long size, unsigned long align,
3259                    gfp_t gfp_mask, int node, const void *caller)
3260 {
3261     return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
3262                                gfp_mask, PAGE_KERNEL, 0, node, caller);
3263 }

```

Listing 3-36: mm/vmalloc.c: __vmalloc_node()

You can see that `vmalloc()` invokes this with alignment set to 1, the GFP mask set to `GFP_KERNEL`, the node mas set to `NUMA_NO_NODE` (specifying that no

specific node is required to be allocated on), and caller set to the return address of the function, which is used for reporting. Equally, `vzalloc()` passes the same parameters, only additionally specifying that the memory range be zeroed via `__GFP_ZERO`.

There are a number of different allocation functions which can be used to perform a kernel virtual memory allocation, but all ultimately invoke `__vmalloc_node_range()`. Figure 3-11 illustrates these allocators.

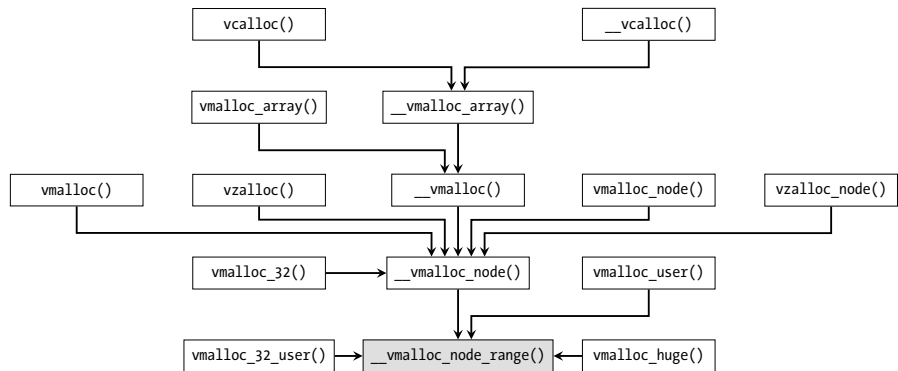


Figure 3-11: *vmalloc* allocators

Each of these end up invoking `__vmalloc_node_range()`, the “heart” of the virtual allocator. Let’s examine the beginning of this function in Listing 3-37 (stripping out of scope KASAN and kmemleak logic).

```

3111 void *__vmalloc_node_range(unsigned long size, unsigned long align,
3112                             unsigned long start, unsigned long end, gfp_t gfp_mask
3113                             ,
3114                             pgprot_t prot, unsigned long vm_flags, int node,
3115                             const void *caller)
3116 {
3117     struct vm_struct *area;
3118     void *ret;
3119     :
3120     unsigned long real_size = size;
3121     unsigned long real_align = align;
3122     unsigned int shift = PAGE_SHIFT;
3123     if (WARN_ON_ONCE(!size))
3124         return NULL;
3125
3126     if ((size >> PAGE_SHIFT) > totalram_pages()) {
3127         warn_alloc(gfp_mask, NULL,
3128                   "vmalloc error: size %lu, exceeds total pages",
3129                   real_size);
3130         return NULL;
3131     }
  
```

```

3132
3133     if (vmap_allow_huge && (vm_flags & VM_ALLOW_HUGE_VMAP)) {
3134         unsigned long size_per_node;
3135
3136         /*
3137          * Try huge pages. Only try for PAGE_KERNEL allocations,
3138          * others like modules don't yet expect huge pages in
3139          * their allocations due to apply_to_page_range not
3140          * supporting them.
3141          */
3142
3143         size_per_node = size;
3144         if (node == NUMA_NO_NODE)
3145             size_per_node /= num_online_nodes();
3146         if (arch_vmap_pmd_supported(prot) && size_per_node >= PMD_SIZE
3147     )
3148             shift = PMD_SHIFT;
3149         else
3150             shift = arch_vmap_pte_supported_shift(size_per_node);
3151
3152         align = max(real_align, 1UL << shift);
3153         size = ALIGN(real_size, 1UL << shift);
3154     }
3155 }

```

Listing 3-37: mm/vmalloc.c: __vmalloc_node_range() preface

Note that the start and (exclusive) end bound parameters specify the range in which the allocation *can* be obtained from.

Typically these are set to VMALLOC_START and VMALLOC_END respectively, indicating that any portion of the vmalloc space can be used.

After ensuring the specified size parameter is of a reasonable size, the VM_ALLOW_HUGE_VMAP flag is checked to see if huge pages can be used (and whether the size is large enough for this). If no specific node is specified, then we anticipate the worst case NUMA memory policy of interleaved pages per node—therefore requiring size to be equal to the huge page size times the number of nodes in this instance.

The function then checks arch_vmap_pmd_supported() to see if huge pages are supported at all and arch_vmap_pte_supported_shift() is used for a Power PC architecture edge case. Size and alignment are adjusted accordingly.

Listing 3-38 shows the remainder of the __vmalloc_node_range() function.

```

3155 again:
3156     area = __get_vm_area_node(real_size, align, shift, VM_ALLOC |
3157                             VM_UNINITIALIZED | vm_flags, start, end,
3158                             node,
3159                             gfp_mask, caller);
3160     if (!area) {
3161         bool nofail = gfp_mask & __GFP_NOFAIL;
3162         warn_alloc(gfp_mask, NULL,

```

```

3162             "vmalloc error: size %lu, vm_struct allocation failed%
           s",
3163             real_size, (nofail) ? ". Retrying." : "");
3164         if (nofail) {
3165             schedule_timeout_uninterruptible(1);
3166             goto again;
3167         }
3168         goto fail;
3169     }
3170     :
3171     /* Allocate physical pages and map them into vmalloc space. */
3172     ret = __vmalloc_area_node(area, gfp_mask, prot, shift, node);
3173     if (!ret)
3174         goto fail;
3175     :
3176     /*
3177      * In this function, newly allocated vm_struct has VM_UNINITIALIZED
3178      * flag. It means that vm_struct is not fully initialized.
3179      * Now, it is fully initialized, so remove this flag here.
3180      */
3181     clear_vm_uninitialized_flag(area);
3182
3183     size = PAGE_ALIGN(size);
3184     :
3185     return area->addr;
3186
3187 fail:
3188     if (shift > PAGE_SHIFT) {
3189         shift = PAGE_SHIFT;
3190         align = real_align;
3191         size = real_size;
3192         goto again;
3193     }
3194
3195     return NULL;
3196 }

```

Listing 3-38: mm/vmalloc.c: `__vmalloc_node_range()` core

A virtual memory area node of type struct `vm_struct` (more on this data type shortly) is initialized via `__get_vm_area_node()` (see Listing 3-40). This tries to find a suitable range of available vmalloc address space of the required size and updates vmalloc state to track it. If this fails, the same operation is tried again if the `__GFP_NO_FAIL` flag was specified, otherwise the operation fails.

Next, the actual allocation and mapping is performed by `__vmalloc_area_node()`, before clearing the `VM_UNINITIALIZED` flag in the generated `vm_struct` (this structure is still referenced within vmalloc).

On failure, if huge pages were used, we try again only this time using an ordinary page size.

We examine struct `vm_struct` in Listing 3-39.

```

48 struct vm_struct {
49     struct vm_struct    *next;
50     void                *addr;
51     unsigned long       size;
52     unsigned long       flags;
53     struct page         **pages;
54 #ifdef CONFIG_HAVE_ARCH_HUGE_VMALLOC
55     unsigned int        page_order;
56 #endif
57     unsigned int        nr_pages;
58     phys_addr_t         phys_addr;
59     const void          *caller;
60 };

```

Listing 3-39: `include/linux/vmalloc.h: struct vm_struct`

Let's go over each field:

- `next` – Used by early initialization functions to maintain a list of early `vmalloc` areas with a head entry at `vm_list`, manipulated via `vm_area_add_early()`, `vm_area_register_early()`, and `vmalloc_init()`.
- `addr` – The virtual address of the start of the allocated block.
- `size` – The size, in bytes, of the allocated block.
- `flags` – `vmalloc` flags associated with the block.
- `pages` – An array of pointers to the `struct page` objects describing the physical memory backing the `vmalloc` area.
- `page_order` – Either 0 or if huge pages are used, the huge page order.
- `nr_pages` – The number of base pages spanning the allocated block.
- `phys_addr` – Ostensibly a physical address associated with the mapped block, but doesn't appear to be used.
- `caller` – A pointer containing the return address of the function that invoked the entrypoint into `vmalloc`. This is used for debug output (for example, via `/proc/vmallocinfo`) to indicate what function requested each block.

The `vmalloc` flags determine how `vmalloc` allocations are performed and are either set by the function the user has called into, or if calling via something like `get_vm_area()`, these can be specified. Flags can be combined:

- `VM_IOREMAP` – This indicates that the allocation will be used for I/O operations in the vein of `ioremap()`. This is out of scope for this discussion.
- `VM_ALLOC` – Indicates that this is an ordinary `vmalloc` allocation.

- **VM_MAP** – Indicates the memory is to be mapped. This field is used as the default flag in invocations of `vmap()`, however it does not appear to have any impact on actual `vmalloc` behaviour.
- **VM_USERMAP** – Indicates that the `vmalloc` block can be remapped into userland. `remap_vmalloc_range_partial()` (which is invoked by `remap_vmalloc_range()`) checks that flag is set (as well as **VM_DMA_COHERENT**) or otherwise aborts. Set by `vmalloc_user()` and `vmalloc_32_user()`.
- **VM_DMA_COHERENT** – Indicates that the memory is to be used for coherent DMA allocations. Out of scope of discussion here.
- **VM_UNINITIALIZED** – This flag is set when the area is first created, and cleared once it is set up via `clear_vm_uninitialized_flag()`. It is only used by `show_numa_info()`, invoked via `s_show()`, which is part of the `vmalloc_op` object used for `/proc/vmallocinfo`, causing no data to be output when the block is not initialized.
- **VM_NO_GUARD** – Used to *prevent* the addition of a guard page on allocation. By default an additional page is marked as used by `vmalloc` but not mapped in `__get_vm_area_node()`. This prevents any subsequent `vmalloc` allocations from allocating the page, so a buffer overrun won't just blend into another region of `vmalloc` and an overrun will definitely trigger a page fault. A notable use of `vmalloc` guard pages are kernel stacks (when **CONFIG_VMAP_STACK** is set), for example, in `page_fault_oops()`, where a stack overflow can be detected and dealt with rather than causing a kernel oops due to it being a page fault in kernel mode. Setting this flag to disable guard pages is not advised.
- **VM_KASAN** – Makes use of the kernel address sanitizer, out of scope for this discussion.
- **VM_FLUSH_RESET_PERMS** – Used to ensure that the TLB is entirely flushed for the `vmalloc` block memory range, and any lazy references to the memory range are cleared also (`vmalloc` lazily flushes the TLB cache on unmap to amortise the time taken for this slow operation). The direct mapping range that points at the pages is also invalidated.
- **VM_MAP_PUT_PAGES** – If set in the invocation of `vmap()`, then ownership of the pages is passed to `vmalloc` and the underlying pages will be put (and possibly freed) on `vfree()`.
- **VM_ALLOW_HUGE_VMAP** – Allows for huge pages to be used in the `vmalloc` allocation. If this is set, the requested allocation size is sufficiently large, and the system supports it, then huge pages will be used.
- **VM_DEFER_KMEMLEAK** – Defers `kmemleak` checks, out of scope for this discussion.

3.5.1 Finding a Free Block

The `__vmalloc_node_range()` function calls `__get_vm_area_node()` to find a free memory block, which we examine in Listing 3-40 (stripping out of scope KASAN and I/O remap logic).

```

2459 static struct vm_struct *__get_vm_area_node(unsigned long size,
2460                                             unsigned long align, unsigned long shift, unsigned long flags,
2461                                             unsigned long start, unsigned long end, int node,
2462                                             gfp_t gfp_mask, const void *caller)
2463 {
2464     struct vmmap_area *va;
2465     struct vm_struct *area;
2466     unsigned long requested_size = size;
2467
2468     BUG_ON(in_interrupt());
2469     size = ALIGN(size, 1ul << shift);
2470     if (unlikely(!size))
2471         return NULL;
2472     :
2477     area = kzalloc_node(sizeof(*area), gfp_mask & GFP_RECLAIM_MASK, node);
2478     if (unlikely(!area))
2479         return NULL;
2480
2481     if (!(flags & VM_NO_GUARD))
2482         size += PAGE_SIZE;
2483
2484     va = alloc_vmmap_area(size, align, start, end, node, gfp_mask);
2485     if (IS_ERR(va)) {
2486         kfree(area);
2487         return NULL;
2488     }
2489
2490     setup_vmmap_vm(area, va, flags, caller);
2491     :
2504     return area;
2505 }

```

Listing 3-40: mm/vmalloc.c: Simplified __get_vm_area_node()

The function starts by asserting we are not in an interrupt context, aligning the size to the input page shift (possibly huge) and ensuring that the alignment didn't overflow the size.

Next, we allocate memory for the `vm_struct` from the slab allocator via `kzalloc_node()`, masking the `gfp_mask` against `GFP_RECLAIM_MASK`, which limits input GFP flags to those that impact watermark checking and reclaim behaviour. If this fails, the function returns `NULL`.

As mentioned previously in the description of `vmalloc` flags, if `VM_NO_GUARD` is *not* set, then the size is extended by `PAGE_SIZE`, or 1 extra page. This will result in an additional page being marked as used by `vmalloc`, and thus unavailable for other `vmalloc` allocations, but the page will not actually be mapped or allocated so any attempt to access it (like a buffer overrun) will result in a page fault.

You can see this is the case as the additional page is added here where `vmalloc` metadata is being setup but not in `__vmalloc_area_node()` where the actual backing page allocation and mapping is performed, and `get_vm_area_size()` explicitly excludes the guard page on backing page allocation.

We then allocate a struct `vmap_area` via `alloc_vmap_area()`. If this fails, we free the previously allocated `vm_struct` and return `NULL`.

Otherwise, we connect the two objects using `setup_vmalloc_vm()`, which acquires the `vmap_area_lock` spin lock and invokes `setup_vmalloc_vm_locked()` to assign the initial `vm_struct` fields, shown in Listing 3-41.

```

2430 static inline void setup_vmalloc_vm_locked(struct vm_struct *vm,
2431      struct vmap_area *va, unsigned long flags, const void *caller)
2432 {
2433     vm->flags = flags;
2434     vm->addr = (void *)va->va_start;
2435     vm->size = va->va_end - va->va_start;
2436     vm->caller = caller;
2437     va->vm = vm;
2438 }
2439
2440 static void setup_vmalloc_vm(struct vm_struct *vm, struct vmap_area *va,
2441      unsigned long flags, const void *caller)
2442 {
2443     spin_lock(&vmap_area_lock);
2444     setup_vmalloc_vm_locked(vm, va, flags, caller);
2445     spin_unlock(&vmap_area_lock);
2446 }
```

Listing 3-41: mm/vmalloc.c: `setup_vmalloc_vm()` and `setup_vmalloc_vm_locked()`

As you can see, there is a direct relationship between the `vm_struct` and the `vmap_area` objects, with the former containing metadata for the allocated block and the latter acting as a node in a red-black tree and a linked list representing the `vmalloc` space.

We construct things this way in order to make it efficient to look up `vmalloc` metadata and to establish which portions of `vmalloc` space are available for allocation.

Returning to `__get_vm_area_node()`, note that it allocates a `vmap_area` via `alloc_vmap_area()`. Let's examine the initial portion of the function in Listing 3-42 (removing out of scope KASAN and `kmemleak` logic).

```

1569 static struct vmap_area *alloc_vmap_area(unsigned long size,
1570      unsigned long align,
1571      unsigned long vstart, unsigned long vend,
1572      int node, gfp_t gfp_mask)
1573 {
1574     struct vmap_area *va;
1575     unsigned long freed;
```

```

1576     unsigned long addr;
1577     int purged = 0;
1578     int ret;
1579
1580     BUG_ON(!size);
1581     BUG_ON(offset_in_page(size));
1582     BUG_ON(!is_power_of_2(align));
1583
1584     if (unlikely(!vmap_initialized))
1585         return ERR_PTR(-EBUSY);
1586
1587     might_sleep();
1588     gfp_mask = gfp_mask & GFP_RECLAIM_MASK;
1589
1590     va = kmem_cache_alloc_node(vmap_area_cachep, gfp_mask, node);
1591     if (unlikely(!va))
1592         return ERR_PTR(-ENOMEM);

```

Listing 3-42: mm/vmalloc.c: alloc_vmap_area() preface

Here, we ensure that the inputs are sane, ensuring that vmalloc is initialized before invoking `kmem_cache_alloc_node()` to allocate memory from a slab cache set up expressly to provide struct `vmap_area` objections. If this allocation fails, we return an error.

At this point we've allocated the `vmap_area` but have yet to insert it into vmalloc state. Let's examine the core logic in Listing 3-43.

```

1600 retry:
1601     preload_this_cpu_lock(&free_vmap_area_lock, gfp_mask, node);
1602     addr = __alloc_vmap_area(&free_vmap_area_root, &free_vmap_area_list,
1603         size, align, vstart, vend);
1604     spin_unlock(&free_vmap_area_lock);
1605
1606     /*
1607      * If an allocation fails, the "vend" address is
1608      * returned. Therefore trigger the overflow path.
1609      */
1610     if (unlikely(addr == vend))
1611         goto overflow;
1612
1613     va->va_start = addr;
1614     va->va_end = addr + size;
1615     va->vm = NULL;
1616
1617     spin_lock(&vmap_area_lock);
1618     insert_vmap_area(va, &vmap_area_root, &vmap_area_list);
1619     spin_unlock(&vmap_area_lock);
1620
1621     BUG_ON(!IS_ALIGNED(va->va_start, align));

```

```

1622     BUG_ON(va->va_start < vstart);
1623     BUG_ON(va->va_end > vend);
1624     :
1631     return va;

```

Listing 3-43: mm/vmalloc.c: alloc_vmap_area() core

This logic is divided into two parts:

1. Find and allocate a free block of vmalloc memory via `__alloc_vmap_area()` (with `free_vmap_area_lock` held) using the red/black binary search tree rooted in `free_vmap_area_root` and the free list headed by `free_vmap_area_list`.
2. Insert the newly allocated block into the vmalloc tree rooted in `vmap_area_root` and headed by `vmap_area_list` (with `vmap_area_lock` held).

We'll examine each of these shortly.

The invocation of `__alloc_vmap_area()` can fail, in which case it returns `vend` to indicate this and we jump to the overflow branch described in Listing 3-44. Otherwise, we set up the `vmap_area` fields appropriately, insert the area, perform some sanity checks, and return the area object.

Finally, we handle overflow in `alloc_vmap_area()`, shown in Listing 3-44.

```

1633 overflow:
1634     if (!purged) {
1635         purge_vmap_area_lazy();
1636         purged = 1;
1637         goto retry;
1638     }
1639
1640     freed = 0;
1641     blocking_notifier_call_chain(&vmap_notify_list, 0, &freed);
1642
1643     if (freed > 0) {
1644         purged = 0;
1645         goto retry;
1646     }
1647
1648     if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit())
1649         pr_warn("vmap allocation for size %lu failed: use vmalloc=<
size> to increase size\n",
1650             size);
1651
1652     kmem_cache_free(vmap_area_cachep, va);
1653     return ERR_PTR(-EBUSY);
1654 }

```

Listing 3-44: mm/vmalloc.c: alloc_vmap_area() overflow handling

This code attempts to free up vmalloc space, first by invoking `purge_vmap_area_lazy()` if this has not previously been tried, then retrying the operation. Next, `blocking_notifier_call_chain()` is tried to invoke any shrinkers registered by vmalloc users. If memory is freed, the operation is retried. If not, it is aborted and an `EBUSY` error code returned.

NOTE

Shrinkers are a means by which kernel slab memory can be freed under memory pressure (this is out of scope for the book). `blocking_notifier_call_chain()` references the `vmap_notify_list` blocking notifier. Users of vmalloc can register or unregister shrinkers via `register_vmap_purge_notifier()` and `unregister_vmap_purge_notifier()`, respectively, though this only appears to be used by a couple of GPU drivers at present.

By default the vmalloc address space area is 32TiB in size for 4-level or 12.5PiB for 5-level x86-64 and similar for other modern architectures, so possessing insufficient space is unlikely to occur except on highly constrained hardware.

In order to avoid flushing the TLB too often (an expensive operation), when vmalloc memory is freed it is typically instead added to the `purge_vmap_area_list` and only freed when either the number of entries on the list reaches the `lazy_max_pages()` threshold, or vmalloc space is exhausted. Both ultimately invoke `__purge_vmap_area_lazy()` to actually purge the blocks.

We examine the core `__alloc_vmap_area()` in Listing 3-45.

```

1471 /*
1472  * Returns a start address of the newly allocated area, if success.
1473  * Otherwise a vend is returned that indicates failure.
1474  */
1475 static __always_inline unsigned long
1476 __alloc_vmap_area(struct rb_root *root, struct list_head *head,
1477                  unsigned long size, unsigned long align,
1478                  unsigned long vstart, unsigned long vend)
1479 {
1480     bool adjust_search_size = true;
1481     unsigned long nva_start_addr;
1482     struct vmap_area *va;
1483     int ret;
1484
1485     /*
1486      * Do not adjust when:
1487      *   a) align <= PAGE_SIZE, because it does not make any sense.
1488      *      All blocks(their start addresses) are at least PAGE_SIZE
1489      *      aligned anyway;
1490      *   b) a short range where a requested size corresponds to exactly
1491      *      specified [vstart:vend] interval and an alignment > PAGE_SIZE.
1492      *      With adjusted search length an allocation would not succeed.
1493      */

```

```

1494         if (align <= PAGE_SIZE || (align > PAGE_SIZE && (vend - vstart) ==
size))
1495             adjust_search_size = false;
1496
1497         va = find_vmap_lowest_match(root, size, align, vstart,
adjust_search_size);
1498         if (unlikely(!va))
1499             return vend;
1500
1501         if (va->va_start > vstart)
1502             nva_start_addr = ALIGN(va->va_start, align);
1503         else
1504             nva_start_addr = ALIGN(vstart, align);
1505
1506         /* Check the "vend" restriction. */
1507         if (nva_start_addr + size > vend)
1508             return vend;
1509
1510         /* Update the free vmap_area. */
1511         ret = adjust_va_to_fit_type(root, head, va, nva_start_addr, size);
1512         if (WARN_ON_ONCE(ret))
1513             return vend;
1514
1515 #if DEBUG_AUGMENT_LOWEST_MATCH_CHECK
1516         find_vmap_lowest_match_check(size, align);
1517 #endif
1518
1519         return nva_start_addr;
1520     }

```

Listing 3-45: mm/vmalloc.c: __alloc_vmap_area()

This function starts by determining whether the search size can be adjusted, as determined by `adjust_search_size`. As the comment says, only a sub-page-sized alignment or one that fits precisely in the specified range would result in this being disabled.

Next, it invokes `find_vmap_lowest_match()`, which is the workhorse of this function. Note that we invoke this function without specifying the upper bound, so we'll need to check this afterwards. This finds the first free block that fulfils size and alignment requirements. If we can't, we exit early by returning `vend`.

Once we have determined the first free block, we set `nva_start_addr` to the largest of `vstart` and the returned `va->va_start` aligned as needed. Note that we ensure that the returned block will have an address greater than or equal to `vstart` because `is_within_this_va()`, called by `find_vmap_lowest_match()`, explicitly asserts this.

We then perform the deferred check against `vend` using this start address and exit returning `vend` to indicate failure if we exceed this.

The only remaining task (excluding an out of scope debug check) is `adjust_va_to_fit_type()`, which adjusts the free `vmalloc` space to account for the allocation.

We examine `find_vmap_lowest_match()` in Listing 3-46.

```

/*
 * Find the first free block(lowest start address) in the tree,
 * that will accomplish the request corresponding to passing
 * parameters. Please note, with an alignment bigger than PAGE_SIZE,
 * a search length is adjusted to account for worst case alignment
 * overhead.
 */
static __always_inline struct vmap_area *
find_vmap_lowest_match(struct rb_root *root, unsigned long size,
                      unsigned long align, unsigned long vstart, bool adjust_search_size)
{
    struct vmap_area *va;
    struct rb_node *node;
    unsigned long length;

    /* Start from the root. */
    node = root->rb_node;

    /* Adjust the search size for alignment overhead. */
    length = adjust_search_size ? size + align - 1 : size;

    while (node) {
        va = rb_entry(node, struct vmap_area, rb_node);

        if (get_subtree_max_size(node->rb_left) >= length &&
            vstart < va->va_start) {
            node = node->rb_left;
        } else {
            if (is_within_this_va(va, size, align, vstart))
                return va;

            /*
             * Does not make sense to go deeper towards the right
             * sub-tree if it does not have a free block that is
             * equal or bigger to the requested search length.
             */
            if (get_subtree_max_size(node->rb_right) >= length) {
                node = node->rb_right;
                continue;
            }
        }

        /*
         * OK. We roll back and find the first right sub-tree,

```



```

        * that will satisfy the search criteria. It can happen
        * due to "vstart" restriction or an alignment overhead
        * that is bigger then PAGE_SIZE.
        */
while ((node = rb_parent(node))) {
    va = rb_entry(node, struct vmmap_area, rb_node);
    if (is_within_this_va(va, size, align, vstart))
        return va;

    if (get_subtree_max_size(node->rb_right) >= length &&
        vstart <= va->va_start) {
        /*
         * Shift the vstart forward. Please note, we update it
         * parent's start address adding "1" because we do not
         * to enter same sub-tree after it has already been
         * and no suitable free block found there.
         */
        vstart = va->va_start + 1;
        node = node->rb_right;
        break;
    }
}

return NULL;
}

```

Listing 3-46: mm/vmalloc.c: `find_vmap_lowest_match()`

This walks the red/black search tree using the twin workhorses of `get_subtree_max_size()` (which safely accesses the `subtree_max_size` field of the node's `struct vmmap_area` object) and `is_within_this_va()` which determines whether the node can contain our aligned size requirements, which we examine in Listing 3-47.

```

1209 static __always_inline bool
1210 is_within_this_va(struct vmmap_area *va, unsigned long size,
1211                  unsigned long align, unsigned long vstart)
1212 {
1213     unsigned long nva_start_addr;
1214
1215     if (va->va_start > vstart)
1216         nva_start_addr = ALIGN(va->va_start, align);
1217     else
1218         nva_start_addr = ALIGN(vstart, align);

```

```

1219
1220     /* Can be overflowed due to big size or alignment. */
1221     if (nva_start_addr + size < nva_start_addr ||
1222         nva_start_addr < vstart)
1223         return false;
1224
1225     return (nva_start_addr + size <= va->va_end);
1226 }

```

Listing 3-47: mm/vmalloc.c: *is_within_this_va()*

NOTE

As the comment states, this function takes special care to avoid underflow or overflow in the case of being supplied an overly large size or alignment.

The `find_vmap_lowest_match()` function walks through the tree, finding the left-most node whose subtree max size is greater than or equal to the required length. Thinking back to the binary search tree invariant, all nodes on the left will be located at a lower address than all of those on the right. Therefore, we are locating the smallest possible entry in the address space.

If we don't have a smaller left-hand subtree, we first check to see if *this* node can satisfy our requirements. If not, we check to see if the right node can satisfy us. In most cases it should, as logically it will certainly have sufficient subtree size, however it may fail to do so because of alignment reasons (potentially causing us to underflow `vstart`).

In this case we have to perform some backtracking, going through each parent node, checking to see if it fulfils our requirements and if not, we try to previously unexamined right sub-node, resetting `vstart` to set a lower-most bound to indicate that we have examined all values at or below it.

It is clear then that we allocate using a simple *first-fit* algorithm. This keeps `vmalloc` simple and relatively fast to use, and on modern hardware is a entirely reasonable trade-off as the amount of available virtual address space is far in excess of what you would ever need to allocate from it so fragmentation should not present a significant problem (for modern 64-bit hardware virtual address space is abundant; it is the physical memory that is scarce).

Finally, once we have discovered a `struct vmap_area` free block that is sufficiently large to contain our requirements from `find_vmap_lowest_match()`, `__alloc_vmap_area()` calls `adjust_va_to_fit_type()` to extract only what is required from the free block and adjust it as required, as shown in Listing 3-48.

```

static __always_inline int
adjust_va_to_fit_type(struct rb_root *root, struct list_head *head,
                     struct vmap_area *va, unsigned long nva_start_addr,
                     unsigned long size)
{
    struct vmap_area *lva = NULL;
    enum fit_type type = classify_va_fit_type(va, nva_start_addr, size);

    if (type == FL_FIT_TYPE) {

```

```

/*
 * No need to split VA, it fully fits.
 */
/*
 *      |
 * V      NVA      V
 * /-----/
 */
unlink_va_augment(va, root);
kmem_cache_free(vmap_area_cachep, va);
} else if (type == LE_FIT_TYPE) {
/*
 * Split left edge of fit VA.
 */
/*
 *      |
 * V NVA V R
 * /-----/-----/
 */
va->va_start += size;
} else if (type == RE_FIT_TYPE) {
/*
 * Split right edge of fit VA.
 */
/*
 *      |      |
 *      L V NVA V
 * /-----/-----/
 */
va->va_end = nva_start_addr;
} else if (type == NE_FIT_TYPE) {
/*
 * Split no edge of fit VA.
 */
/*
 *      |      |
 *      L V NVA V R
 * /---/-----/---/
 */
lva = __this_cpu_xchg(ne_fit_preload_node, NULL);
if (unlikely(!lva)) {
/*
 * For percpu allocator we do not do any pre-allocation
 * and leave it as it is. The reason is it most likely
 * never ends up with NE_FIT_TYPE splitting. In case of
 * percpu allocations offsets and sizes are aligned to
 * fixed align request, i.e. RE_FIT_TYPE and FL_FIT_TYPE
 * are its main fitting cases.
 *
 * There are a few exceptions though, as an example it is
 * a first allocation (early boot up) when we have "one"

```

```

        * big free space that has to be split.
        *
        * Also we can hit this path in case of regular "vmap"
        * allocations, if "this" current CPU was not preloaded.
        * See the comment in alloc_vmap_area() why. If so, then
        * GFP_NOWAIT is used instead to get an extra object for
        * split purpose. That is rare and most time does not
        * occur.
        *
        * What happens if an allocation gets failed. Basically,
        * an "overflow" path is triggered to purge lazily freed
        * areas to free some memory, then, the "retry" path is
        * triggered to repeat one more time. See more details
        * in alloc_vmap_area() function.
        */
    lva = kmem_cache_alloc(vmap_area_cachep, GFP_NOWAIT);
    if (!lva)
        return -1;
}

/*
 * Build the remainder.
 */
lva->va_start = va->va_start;
lva->va_end = nva_start_addr;

/*
 * Shrink this VA to remaining size.
 */
va->va_start = nva_start_addr + size;
} else {
    return -1;
}

if (type != FL_FIT_TYPE) {
    augment_tree_propagate_from(va);

    if (lva) /* type == NE_FIT_TYPE */
        insert_vmap_area_augment(lva, &va->rb_node, root, head);
}

return 0;
}

```

Listing 3-48: mm/vmalloc.c: *adjust_va_to_fit_type()*

The comments here are excellent and give a good idea as to what is going on. First, they classify the *type* of fit via enum *fit_type*, shown in Listing 3-49.

```

1337 enum fit_type {
1338     NOTHING_FIT = 0,
1339     FL_FIT_TYPE = 1,      /* full fit */
1340     LE_FIT_TYPE = 2,      /* left edge fit */
1341     RE_FIT_TYPE = 3,      /* right edge fit */
1342     NE_FIT_TYPE = 4       /* no edge fit */
1343 };

```

Listing 3-49: mm/vmalloc.c: enum fit_type

These are:

NOTHING_FIT No fit at all.

FL_FIT_TYPE The required block fits perfectly into the free one (that is, they are of identical size and alignment).

LE_FIT_TYPE The free block has some excess space to its ‘right’ (at higher addresses), but the left fits perfectly.

RE_FIT_TYPE The free block has some excess space to its ‘left’ (at lower addresses), but the right fits perfectly.

NE_FIT_TYPE The required block fits within the free block, but it leaves “gaps” at either side.

The logic that performs this classification is defined in `classify_va_fit_type()`, shown in Listing 3-50.

```

1345 static __always_inline enum fit_type
1346 classify_va_fit_type(struct vmmap_area *va,
1347     unsigned long nva_start_addr, unsigned long size)
1348 {
1349     enum fit_type type;
1350
1351     /* Check if it is within VA. */
1352     if (nva_start_addr < va->va_start ||
1353         nva_start_addr + size > va->va_end)
1354         return NOTHING_FIT;
1355
1356     /* Now classify. */
1357     if (va->va_start == nva_start_addr) {
1358         if (va->va_end == nva_start_addr + size)
1359             type = FL_FIT_TYPE;
1360         else
1361             type = LE_FIT_TYPE;
1362     } else if (va->va_end == nva_start_addr + size) {
1363         type = RE_FIT_TYPE;
1364     } else {
1365         type = NE_FIT_TYPE;
1366     }
1367

```

```

1368     return type;
1369 }

```

Listing 3-50: mm/vmalloc.c: classify_va_fit_type()

Once we've classified the kind of fit, this indicates what we need to do. If it is a `NOTHING_FIT`, we simply error out: for `LE_FIT_TYPE` or `RE_FIT_TYPE` we need only adjust the `va_start` and `va_end` fields of the free block, respectively. In each of these cases, no nodes need be removed or added.

In the case of `FL_FIT_TYPE` we simply remove the node from the tree via `unlink_va_augment()` and free the area altogether via `kmem_cache_free()`.

In the case of `NE_FIT_TYPE` we need to allocate a new block to describe one of the two separate free blocks, which we achieve using an optimization—trying to obtain the block from a per-CPU value `ne_fit_preload_node`, which is populated via `preload_this_cpu_lock()`, which in turn is invoked by `alloc_vmap_area()` on each allocation attempt. The newly allocated block is assigned as the 'left' block and the existing one is truncated to be the 'right' block.

Finally, for all cases other than `FL_FIT_TYPE`, the tree is updated such that `subtree_max_size` values are recalculated via `augment_tree_propagate_from()`, and if a new node was added, it is inserted via `insert_vmap_area_augment()`.

3.5.2 Inserting a Newly Allocated Block

Finally, we can look to the final step in `alloc_vmap_area()`, invoking `insert_vmap_area()` on the newly allocated vmap area, shown in Listing 3-51.

```

1065 static void
1066 insert_vmap_area(struct vmap_area *va,
1067                 struct rb_root *root, struct list_head *head)
1068 {
1069     struct rb_node **link;
1070     struct rb_node *parent;
1071
1072     link = find_va_links(va, root, NULL, &parent);
1073     if (link)
1074         link_va(va, root, parent, link, head);
1075 }

```

Listing 3-51: mm/vmalloc.c: insert_vmap_area()

We'll gloss over `link_va()` here as it essentially uses the kernel's red/black tree library code to insert a node into the tree (as well as adding it to the node list). Instead we'll focus on `find_va_links()`, which finds the location at which to insert the node, returning a pointer to the parent node's free left or right node as well as the parent node itself, shown in Listing 3-52.

```

839 /*
840  * This function returns back addresses of parent node
841  * and its left or right link for further processing.

```

```

842  *
843  * Otherwise NULL is returned. In that case all further
844  * steps regarding inserting of conflicting overlap range
845  * have to be declined and actually considered as a bug.
846  */
847  static __always_inline struct rb_node **
848  find_va_links(struct vmmap_area *va,
849               struct rb_root *root, struct rb_node *from,
850               struct rb_node **parent)
851  {
852     struct vmmap_area *tmp_va;
853     struct rb_node **link;
854
855     if (root) {
856         link = &root->rb_node;
857         if (unlikely(!*link)) {
858             *parent = NULL;
859             return link;
860         }
861     } else {
862         link = &from;
863     }
864
865     /*
866      * Go to the bottom of the tree. When we hit the last point
867      * we end up with parent rb_node and correct direction, i name
868      * it link, where the new va->rb_node will be attached to.
869      */
870     do {
871         tmp_va = rb_entry(*link, struct vmmap_area, rb_node);
872
873         /*
874          * During the traversal we also do some sanity check.
875          * Trigger the BUG() if there are sides(left/right)
876          * or full overlaps.
877          */
878         if (va->va_end <= tmp_va->va_start)
879             link = &(*link)->rb_left;
880         else if (va->va_start >= tmp_va->va_end)
881             link = &(*link)->rb_right;
882         else {
883             WARN(1, "vmalloc bug: 0x%lx-0x%lx overlaps with 0x%lx
884             -0x%lx\n",
885                  va->va_start, va->va_end, tmp_va->va_start,
886             tmp_va->va_end);
887
888             return NULL;
889         }
890     } while (1);

```

```

887         }
888     } while (*link);
889
890     *parent = &tmp_va->rb_node;
891     return link;
892 }

```

Listing 3-52: mm/vmalloc.c: find_va_links()

If the root argument is specified, then we begin our search here, check that there is at least one node, exiting if not. Otherwise, we begin from from.

The logic then simply loops through nodes, moving left if that node is after the end of the one being inserted, or right otherwise (with a check to ensure that no buggy vmalloc implementation has led to unexpected overlaps) and ending once we have a pointer to an empty pointer either in left or right of the parent before returning it.

3.5.3 Physical Allocation

The final part of `__vmalloc_node_range()` allocates and maps underlying physical memory via `__vmalloc_area_node()`. This function is parameterized by `gfp_mask`, which defines how physical pages will be allocated, and `prot`, which defines the page table flags, so the caller has complete control over how these pages are mapped in.

Let's examine the first part of this function in Listing 3-53.

```

2988 static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp_mask,
2989                                pgprot_t prot, unsigned int page_shift,
2990                                int node)
2991 {
2992     const gfp_t nested_gfp = (gfp_mask & GFP_RECLAIM_MASK) | __GFP_ZERO;
2993     bool nofail = gfp_mask & __GFP_NOFAIL;
2994     unsigned long addr = (unsigned long)area->addr;
2995     unsigned long size = get_vm_area_size(area);
2996     unsigned long array_size;
2997     unsigned int nr_small_pages = size >> PAGE_SHIFT;
2998     unsigned int page_order;
2999     unsigned int flags;
3000     int ret;
3001
3002     array_size = (unsigned long)nr_small_pages * sizeof(struct page *);
3003     gfp_mask |= __GFP_NOWARN;
3004     if (!(gfp_mask & (GFP_DMA | GFP_DMA32)))
3005         gfp_mask |= __GFP_HIGHMEM;
3006
3007     /* Please note that the recursion is strictly bounded. */
3008     if (array_size > PAGE_SIZE) {
3009         area->pages = __vmalloc_node(array_size, 1, nested_gfp, node,
3010                                     area->caller);

```



```

3011         } else {
3012             area->pages = kmalloc_node(array_size, nested_gfp, node);
3013         }
3014
3015         if (!area->pages) {
3016             warn_alloc(gfp_mask, NULL,
3017                 "vmalloc error: size %lu, failed to allocated page
array size %lu",
3018                     nr_small_pages * PAGE_SIZE, array_size);
3019             free_vm_area(area);
3020             return NULL;
3021         }

```

Listing 3-53: mm/vmalloc.c: `__vmalloc_area_node()` preface

We start by determining how many base pages will be need to be allocated via `get_vm_area_size()`, which obtains the area's size less the guard page, shown in Listing 3-54.

```

198 static inline size_t get_vm_area_size(const struct vm_struct *area)
199 {
200     if (!(area->flags & VM_NO_GUARD))
201         /* return actual size without guard page */
202         return area->size - PAGE_SIZE;
203     else
204         return area->size;
205 }
206

```

Listing 3-54: include/linux/vmalloc.h: `get_vm_area_size()`

Returning to `__vmalloc_area_node()`, we then focus on allocating the array that contains the pointers to struct pages to be placed in the struct `vm_struct->pages` field. If we require less than a page of them, we use the slab allocator via `kmalloc_node()`, otherwise we perform a *nested* vmalloc allocation for the pages with the `__GFP_ZERO` flag set to zero them via `__vmalloc_node()`. If either of these fail, we raise an error, clean up, and return NULL. We also specify that no warnings should be reported on allocation via `__GFP_NOWARN`, and set `__GFP_HIGHMEM` if no DMA flags are set, however this is not relevant to 64-bit architectures. Next, we physically allocate the pages, as shown in Listing 3-55.

```

3023         set_vm_area_page_order(area, page_shift - PAGE_SHIFT);
3024         page_order = vm_area_page_order(area);
3025
3026         area->nr_pages = vm_area_alloc_pages(gfp_mask | __GFP_NOWARN,
3027             node, page_order, nr_small_pages, area->pages);
3028
3029         atomic_long_add(area->nr_pages, &nr_vmalloc_pages);
3030         if (gfp_mask & __GFP_ACCOUNT) {

```

```

3031             int i;
3032
3033             for (i = 0; i < area->nr_pages; i++)
3034                 mod_memcg_page_state(area->pages[i], MEMCG_VMALLOC, 1)
3035         ;
3036     }
3037     /*
3038     * If not enough pages were obtained to accomplish an
3039     * allocation request, free them via __vfree() if any.
3040     */
3041     if (area->nr_pages != nr_small_pages) {
3042         warn_alloc(gfp_mask, NULL,
3043             "vmalloc error: size %lu, page order %u, failed to
3044             allocate pages",
3045             area->nr_pages * PAGE_SIZE, page_order);
3046         goto fail;
3047     }

```

Listing 3-55: mm/vmalloc.c: __vmalloc_area_node() physical allocation

We set the struct `vm_struct->page_order` field via `set_vm_area_page_order()`, which asserts that the order is either zero unless if huge allocations are permitted. The actual allocation is performed via `vm_area_alloc_pages()` (to be covered in Listing 3-57) after which we perform some cgroup accounting and check to ensure that the allocation succeeded, exiting and indicating failure if not.

Finally, we map the memory, as shown in Listing 3-56.

```

3048     /*
3049     * page tables allocations ignore external gfp mask, enforce it
3050     * by the scope API
3051     */
3052     if ((gfp_mask & (__GFP_FS | __GFP_IO)) == __GFP_IO)
3053         flags = memalloc_nofs_save();
3054     else if ((gfp_mask & (__GFP_FS | __GFP_IO)) == 0)
3055         flags = memalloc_noio_save();
3056
3057     do {
3058         ret = vmap_pages_range(addr, addr + size, prot, area->pages,
3059             page_shift);
3060         if (nofail && (ret < 0))
3061             schedule_timeout_uninterruptible(1);
3062     } while (nofail && (ret < 0));
3063
3064     if ((gfp_mask & (__GFP_FS | __GFP_IO)) == __GFP_IO)
3065         memalloc_nofs_restore(flags);
3066     else if ((gfp_mask & (__GFP_FS | __GFP_IO)) == 0)
3067         memalloc_noio_restore(flags);

```

```

3068
3069     if (ret < 0) {
3070         warn_alloc(gfp_mask, NULL,
3071             "vmalloc error: size %lu, failed to map pages",
3072             area->nr_pages * PAGE_SIZE);
3073         goto fail;
3074     }
3075
3076     return area->addr;
3077
3078 fail:
3079     __vfree(area->addr);
3080     return NULL;
3081 }

```

Listing 3-56: mm/vmalloc.c: __vmalloc_area_node() virtual mapping

We start by ensuring that `__GFP_FS` and `__GFP_IO` flags are honored throughout mapping before performing the actual mapping itself via `vmap_pages_range()`, retry the mapping if `__GFP_NOFAIL` was set by the user, warning and failing if the allocation fails and returning the newly mapped virtual address if it did not.

Let's dive into how physical allocation is performed in `vm_area_alloc_pages()`, starting with examining the initial portion of this function in Listing 3-57.

```

static inline unsigned int
vm_area_alloc_pages(gfp_t gfp, int nid,
    unsigned int order, unsigned int nr_pages, struct page **pages)
{
    unsigned int nr_allocated = 0;
    struct page *page;
    int i;

    /*
     * For order-0 pages we make use of bulk allocator, if
     * the page array is partly or not at all populated due
     * to fails, fallback to a single page allocator that is
     * more permissive.
     */
    if (!order) {
        gfp_t bulk_gfp = gfp & ~__GFP_NOFAIL;

        while (nr_allocated < nr_pages) {
            unsigned int nr, nr_pages_request;

            /*
             * A maximum allowed request is hard-coded and is 100
             * pages per call. That is done in order to prevent a

```

```

        * long preemption off scenario in the bulk-allocator
        * so the range is [1:100].
        */
nr_pages_request = min(100U, nr_pages - nr_allocated);

/* memory allocation should consider mempolicy, we can't
 * wrongly use nearest node when nid == NUMA_NO_NODE,
 * otherwise memory may be allocated in only one node,
 * but mempolicy wants to alloc memory by interleaving.
 */
if (IS_ENABLED(CONFIG_NUMA) && nid == NUMA_NO_NODE)
    nr = alloc_pages_bulk_array_mempolicy(bulk_gfp,
                                           nr_pages_request,
                                           pages + nr_allocated);

else
    nr = alloc_pages_bulk_array_node(bulk_gfp, nid,
                                     nr_pages_request,
                                     pages + nr_allocated);

nr_allocated += nr;
cond_resched();

/*
 * If zero or pages were obtained partly,
 * fallback to a single page allocator.
 */
if (nr != nr_pages_request)
    break;
}
}

```

Listing 3-57: mm/vmalloc.c: *vm_area_alloc_pages()* preface and order-0 allocation

If the allocations are of order-0 pages, then we simply allocate the pages in bulk using the physical memory allocate via `alloc_pages_bulk_array_mempolicy()` if NUMA is enabled and no specific node is specified, or `alloc_pages_bulk_array_node()` otherwise. These allow for a lock to be shared for the whole operation and therefore allocates pages more efficiently.

We choose to allocate in batches of up to 100 pages to avoid spending too long in the page allocator and thus risk blocking.

If the allocation fails (we are not allocated the number of pages we need), then we fall through to the code used for higher order pages where we allocate a page at a time as a fallback, shown in Listing 3-58.

```

2951     /* High-order pages or fallback path if "bulk" fails. */
2952
2953     while (nr_allocated < nr_pages) {

```

```

2954         if (fatal_signal_pending(current))
2955             break;
2956
2957         if (nid == NUMA_NO_NODE)
2958             page = alloc_pages(gfp, order);
2959         else
2960             page = alloc_pages_node(nid, gfp, order);
2961         if (unlikely(!page))
2962             break;
2963         /*
2964          * Higher order allocations must be able to be treated as
2965          * independent small pages by callers (as they can with
2966          * small-page vmallocs). Some drivers do their own refcounting
2967          * on vmalloc_to_page() pages, some use page->mapping,
2968          * page->lru, etc.
2969          */
2970         if (order)
2971             split_page(page, order);
2972
2973         /*
2974          * Careful, we allocate and map page-order pages, but
2975          * tracking is done per PAGE_SIZE page so as to keep the
2976          * vm_struct APIs independent of the physical/mapped size.
2977          */
2978         for (i = 0; i < (1U << order); i++)
2979             pages[nr_allocated + i] = page + i;
2980
2981         cond_resched();
2982         nr_allocated += 1U << order;
2983     }
2984
2985     return nr_allocated;
2986 }

```

Listing 3-58: mm/vmalloc.c: vm_area_alloc_pages() higher order allocation

After checking to see if we are possibly blocking a fatal signal via `fatal_signal_pending()`, we allocate a single page directly either via `alloc_pages()` if no node is specified, or `alloc_pages_node()` if one is.

Since we do not set the `__GFP_COMP` flag when allocating, the page is not marked compound even if higher order. Therefore, we use `split_page()` to set the reference count for each individual page and thus establish them as single pages.

The pages array is then filled with the allocated pages and the number of allocations returned.

3.5.4 Virtual Mapping

The key `vmalloc` mapping code is implemented in `vmap_pages_range()` as invoked by `__vmalloc_area_node()` (shown in Listing 3-56), as well as `vm_map_ram()` and `vmap()`.

We examine this function in Listing 3-59.

```

604 /**
605  * vmap_pages_range - map pages to a kernel virtual address
606  * @addr: start of the VM area to map
607  * @end: end of the VM area to map (non-inclusive)
608  * @prot: page protection flags to use
609  * @pages: pages to map (always PAGE_SIZE pages)
610  * @page_shift: maximum shift that the pages may be mapped with, @pages must
611  * be aligned and contiguous up to at least this shift.
612  *
613  * RETURNS:
614  * 0 on success, -errno on failure.
615  */
616 static int vmap_pages_range(unsigned long addr, unsigned long end,
617                             pgprot_t prot, struct page **pages, unsigned int page_shift)
618 {
619     int err;
620
621     err = vmap_pages_range_noflush(addr, end, prot, pages, page_shift);
622     flush_cache_vmap(addr, end);
623     return err;
624 }
```

Listing 3-59: `mm/vmalloc.c: vmap_pages_range()`

This performs the core mapping logic in `vmap_pages_range_noflush()` before flushing memory caches for the range via `flush_cache_vmap()` if the architecture requires it (this is a no-op on x86-64 and arm64). We examine `vmap_pages_range_noflush()` in Listing 3-60.

```

578 int vmap_pages_range_noflush(unsigned long addr, unsigned long end,
579                             pgprot_t prot, struct page **pages, unsigned int page_shift)
580 {
581     unsigned int i, nr = (end - addr) >> PAGE_SHIFT;
582
583     WARN_ON(page_shift < PAGE_SHIFT);
584
585     if (!IS_ENABLED(CONFIG_HAVE_ARCH_HUGE_VMAP) ||
586         page_shift == PAGE_SHIFT)
587         return vmap_small_pages_range_noflush(addr, end, prot, pages);
588
589     for (i = 0; i < nr; i += 1U << (page_shift - PAGE_SHIFT)) {
590         int err;
591     }
```

```

592         err = vmap_range_noflush(addr, addr + (1UL << page_shift),
593                                __pa(page_address(pages[i])), prot,
594                                page_shift);
595         if (err)
596             return err;
597
598         addr += 1UL << page_shift;
599     }
600
601     return 0;
602 }

```

Listing 3-60: mm/vmalloc.c: vmap_pages_range_noflush()

If the architecture doesn't support huge pages or huge pages have not been used, then the mapping is performed by `vmap_small_pages_range_noflush()`. Otherwise, each individual page is mapped via `vmap_range_noflush()`.

For brevity, we'll examine `vmap_small_pages_range_noflush()` only in Listing 3-61.

```

542 static int vmap_small_pages_range_noflush(unsigned long addr, unsigned long
      end,
543                                           pgprot_t prot, struct page **pages)
544 {
545     unsigned long start = addr;
546     pgd_t *pgd;
547     unsigned long next;
548     int err = 0;
549     int nr = 0;
550     pgtbl_mod_mask mask = 0;
551
552     BUG_ON(addr >= end);
553     pgd = pgd_offset_k(addr);
554     do {
555         next = pgd_addr_end(addr, end);
556         if (pgd_bad(*pgd))
557             mask |= PGTBL_PGD_MODIFIED;
558         err = vmap_pages_p4d_range(pgd, addr, next, prot, pages, &nr,
      &mask);
559         if (err)
560             return err;
561     } while (pgd++, addr = next, addr != end);
562
563     if (mask & ARCH_PAGE_TABLE_SYNC_MASK)
564         arch_sync_kernel_mappings(start, end);
565
566     return 0;
567 }

```

Listing 3-61: mm/vmalloc.c: `vmap_small_pages_range_noflush()`

This follows a similar pattern through each page table level, keeping track of which page table entries have been updated via `mask`. Each invocation of the `pXd_addr_end()` function retrieves the virtual address associated with the next boundary at that level (in this case, the next PGD entry).

One difference at this level is that, for architectures that require it, `arch_sync_kernel_mappings()` is invoked when an update has occurred that requires it. For x86-64 this isn't necessary because during initialization `preallocate_vmalloc_pages()` is called to ensure that the shared page table entries that span the `vmalloc` space are correctly allocated from which all process page tables are derived.

We examine `vmap_pages_p4d_range()` in Listing 3-62.

```

524 static int vmap_pages_p4d_range(pgd_t *pgd, unsigned long addr,
525                                unsigned long end, pgprot_t prot, struct page **pages, int *nr
526                                ,
527                                pgtbl_mod_mask *mask)
528 {
529     p4d_t *p4d;
530     unsigned long next;
531
532     p4d = p4d_alloc_track(&init_mm, pgd, addr, mask);
533     if (!p4d)
534         return -ENOMEM;
535     do {
536         next = p4d_addr_end(addr, end);
537         if (vmap_pages_pud_range(p4d, addr, next, prot, pages, nr,
538                                 mask))
539             return -ENOMEM;
540     } while (p4d++, addr = next, addr != end);
541     return 0;
542 }
```

Listing 3-62: mm/vmalloc.c: `vmap_pages_p4d_range()`

This follows a similar pattern except that `p4d_alloc_track()` is used, which performs the same role as `p4d_alloc()`, keeping track of whether the page table previously shown was modified (as is the case where a new P4D had to be allocated and thus the PGD entry modified).

The `pXd_alloc_track()` functions are used for this purpose for all of the page table levels in order that the `arch_sync_kernel_mappings()` logic at the top level can be applied regardless of which level updates trigger this for the architecture.

The equivalent function for PUD page table entries is `vmap_pages_pud_range()`, shown in Listing 3-63.

```

506 static int vmap_pages_pud_range(p4d_t *p4d, unsigned long addr,
```

```

507         unsigned long end, pgprot_t prot, struct page **pages, int *nr
508         ,
509         pgtbl_mod_mask *mask)
510 {
511     pud_t *pud;
512     unsigned long next;
513
514     pud = pud_alloc_track(&init_mm, p4d, addr, mask);
515     if (!pud)
516         return -ENOMEM;
517     do {
518         next = pud_addr_end(addr, end);
519         if (vmap_pages_pmd_range(pud, addr, next, prot, pages, nr,
520                                 mask))
521             return -ENOMEM;
522     } while (pud++, addr = next, addr != end);
523     return 0;
524 }

```

Listing 3-63: mm/vmalloc.c: vmap_pages_pud_range()

The equivalent for PMDs is vmap_pages_pmd_range(), shown in Listing 3-64.

```

488 static int vmap_pages_pmd_range(pud_t *pud, unsigned long addr,
489                                unsigned long end, pgprot_t prot, struct page **pages, int *nr
490                                ,
491                                pgtbl_mod_mask *mask)
492 {
493     pmd_t *pmd;
494     unsigned long next;
495
496     pmd = pmd_alloc_track(&init_mm, pud, addr, mask);
497     if (!pmd)
498         return -ENOMEM;
499     do {
500         next = pmd_addr_end(addr, end);
501         if (vmap_pages_pte_range(pmd, addr, next, prot, pages, nr,
502                                 mask))
503             return -ENOMEM;
504     } while (pmd++, addr = next, addr != end);
505     return 0;
506 }

```

Listing 3-64: mm/vmalloc.c: vmap_pages_pmd_range()

These are each broadly equivalent to one another, however vmap_pages_pte_range() does more, as shown in Listing 3-65.

```

457 static int vmap_pages_pte_range(pmd_t *pmd, unsigned long addr,

```

```

458             unsigned long end, pgprot_t prot, struct page **pages, int *nr
459             ,
460             pgtbl_mod_mask *mask)
461 {
462     pte_t *pte;
463     /*
464      * nr is a running index into the array which helps higher level
465      * callers keep track of where we're up to.
466      */
467
468     pte = pte_alloc_kernel_track(pmd, addr, mask);
469     if (!pte)
470         return -ENOMEM;
471     do {
472         struct page *page = pages[*nr];
473
474         if (WARN_ON(!pte_none(*pte)))
475             return -EBUSY;
476         if (WARN_ON(!page))
477             return -ENOMEM;
478         if (WARN_ON(!pfn_valid(page_to_pfn(page))))
479             return -EINVAL;
480
481         set_pte_at(&init_mm, addr, pte, mk_pte(page, prot));
482         (*nr)++;
483     } while (pte++, addr += PAGE_SIZE, addr != end);
484     *mask |= PGTBL_PTE_MODIFIED;
485     return 0;
486 }

```

Listing 3-65: mm/vmalloc.c: `vmap_pages_pte_range()`

This function differs from the others in that it both allocates the PTE directory *and* assigns the PTE entry. We make sure that PTE entries are empty, that the backing page is assigned, that the proposed PFN is valid, and we keep track of the number of pages mapped via `nr`.

The purpose of examining the `vmalloc` implementation in this level of detail was not only to provide understanding of how it is implemented, but also to examine at length how the virtual as well as physical memory is managed.

As the `vmalloc` uses a simple first-fit allocator (as opposed to the physical allocator), we will not examine `vfree()`, which more or less follows similar patterns.

4

PROCESS MEMORY

We've examined how the kernel manages physical and maps virtual memory. This provides the foundation upon which to explore arguably the most important area of the linux memory manager – *process* memory. This is the means by which processes allocate, free, map and unmap memory and importantly, the *entire* means by which userland processes interact with memory.

Given that the kernel's reason for being is to provide abstractions for userland, which is another way of saying: provide abstractions for userland processes, these are therefore the *key* interfaces to understand in order to gain a deep understand of memory management as a whole.

We examine a simple userland program which allocates and frees system memory in Listing 4-1.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
```

```
int main(void)
{
```

```

char *ptr = mmap(NULL, 20000, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

if (ptr == MAP_FAILED || munmap(ptr, 20000) != 0) {
    perror("mmap");
    return EXIT_FAILURE;
}

return EXIT_SUCCESS;
}

```

Listing 4-1: Simple use of `mmap` and `munmap`

Note that we use `mmap()` rather than `malloc()` as the latter uses a userland allocator occasionally interacting with the kernel to obtain memory, whereas the former directly maps memory into the userland process which may be backed by *anonymous** pages allocated by the kernel’s physical allocator. We specify `MAP_PRIVATE` and `MAP_ANONYMOUS` flags so we expect to be allocated anonymous physical pages.

We can use the `/proc/$pid/smmaps`[†] procfs interface to observe virtual memory layout. Examining this for Listing 4-1 after the `mmap()` call which in this instance has returned a pointer to `0x7ffff7bf000`, as shown in Listing 4-4.

```

$ cat /proc/$PID/smmaps

7ffff7bf000-7ffff7fc4000 rw-p 00000000 00:00 0
Size:                20 kB

Rss:                  0 kB

```

Listing 4-2: `/proc/$PID/smmaps` output after `mmap()` of `0x7ffff7bf000`

We can see that the memory region containing our pointer is present, but there is something odd going on here – ‘RSS’ is short for ‘Resident Set Size’ which indicates how much memory is actually allocated, but it reports 0 bytes, i.e. *nothing* has been allocated (note that the requested 20,000 bytes was page-aligned to 20 KiB i.e. 20,480 bytes). What is going on?

Answering in two parts – firstly what exactly are `/proc/$pid/smmaps` referring to if not actual page tables and physical pages? The answer are Virtual Memory Areas (VMAs) – these are blocks of contiguous memory which share characteristics (e.g. permissions, flags, etc.) described by the `struct vm_area_struct` type.

Secondly, and more importantly, how can memory be allocated from userland’s perspective but neither allocated nor mapped from the kernel’s? The answer arises from the fact that the kernel gets to decide what happens when hardware *page faults* occurs. Page faults are triggered when either unmapped memory is accessed or operations prohibited by page flags occur

*. ‘Anonymous’ pages are those that are backed by physical memory, not files.

†. We will go over memory-specific procfs interfaces in great detail in a later chapter.

(e.g. writing into memory marked read-only) and crucially, the kernel can catch and handle these, correcting the issue and resuming the process if appropriate.

As a result, it becomes possible to *defer* memory allocation until the first instance a process tries to access it. When they do, a page fault occurs and the kernel can access the process's struct `mm_struct` object (this describes its address space and contains all of its VMAs) and check whether there is a VMA containing the accessed address and then allocate the physical backing memory and any page tables that are needed, pointing them at the backing page and resuming the process. This technique is termed *demand paging*.

This concept of demand paging is one which the kernel heavily relies upon. Another is *Copy-on-Write (CoW)*. This is where a read-write memory mapping is mapped read-only, and when a page fault arises on write, the underlying page is copied, hence 'copy on write'.

This is useful for speeding up forking of processes as we can simply establish CoW mappings to the parent process and avoid having to copy any underlying memory at all, but is also useful when allocating memory.

Early on the kernel establishes a *zero page*, which is simply a page with all of its contents set to zero. When mapping private memory, the kernel can simply map the memory as CoW to the zero page – when read this memory will read zero, when written to it will be copied, establishing the actual allocated memory.

A memory mapping can map underlying physical memory alone mapping) or an underlying file. We describe the former as *anonymous-backed* and the latter as *file-backed*.

Anonymous memory is capable of being 'swapped out' if it possesses the `PG_swapbacked` folio flag, at which point no folio will be associated with the mapping and the data will be written to disk, 'swapped in' again on page fault. More on this in the chapter on swap memory.

4.1 Overcommit

The ability to allocate more memory than is actually installed in the system (deferring the *actual* mapping and allocation to access time) is termed **overcommit**. There are three different overcommit modes permitted in the kernel set via the `vm.overcommit_memory` tuneable and checked by the kernel in `__vm_enough_memory()`:

- `OVERCOMMIT_GUESS` (0) – *default* – If a process tries to allocate an obviously silly amount of memory i.e. one that exceeds the sum of total installed RAM and swap then the allocation is disallowed, otherwise it is permitted.
- `OVERCOMMIT_ALWAYS` (1) – No checks are performed whatsoever, all allocations are permitted. This might be useful for situations where it is known that huge allocations will be made but remain mostly sparse.
- `OVERCOMMIT_NEVER` (2) – Attempt to avoid overcommit altogether. The amount of *virtual* memory permitted to be allocated *by the system as a*

whole is calculated in `vm_commit_limit()` and is specified by either the `vm.overcommit_kbytes` or `vm.overcommit_ratio` *VM tuneables*. The maximum allocation permitted at any one time is the total available swap plus either `vm.overcommit_kbytes` or `vm.overcommit_ratio%` of all non-hugetlb RAM. This limit can be observed in `/proc/meminfo` as `CommitLimit`. Note that additional space is reserved for root operations (via the `vm.admin_reserve_kbytes` tuneable) and also for userland processes to ensure system recovery can always occur via `vm.user_reserve_kbytes`.

A key difference here is that in `OVERCOMMIT_GUESS` mode the check is performed against the amount of memory being allocated, however in `OVERCOMMIT_NEVER` mode the check is made against the system's total *committed* memory size, i.e. the amount of virtual memory allocated by all process but not necessarily physically allocated/mapped by the kernel.

For example, invoking `mmap()` or `malloc()` to allocate 1 GiB of RAM will increase committed memory by 1 GiB while not necessarily increasing physical memory usage at all. This value can be observed in `/proc/meminfo` as `Committed_AS` and the value is stored in the per-CPU counter `vm_committed_as`.

NOTE

Per-CPU counters are a mechanism for efficiently accruing counters between CPUs while permitting some small degree of error equal to the counter batch size, see `lib/percpu_counter.c` for more details.

An important caveat here – any memory that is allocated such that it cannot trigger demand paging, e.g. `PROT_NONE` or `PROT_READ` (without `PROT_WRITE`), will *not* contribute towards any commit accounting as the memory, if written to, would result in a segfault.

NOTE

Additionally, `mmap()`'d memory with the `MAP_NORESERVE` flag set does not contribute towards commit accounting, however this is not honoured in `OVERCOMMIT_NEVER` mode, so is only meaningfully useful in `OVERCOMMIT_GUESS` where even the basic total RAM and swap check is not performed (an exception to this rule are hugetlb mappings but these are out of scope for this discussion).

However memory mapped this way *will* show up in `/proc/$PID/status` `VmSize` and `VmPeak` statistics. This is checked via `security_vm_enough_memory_mm()` (which in turn calls `__vm_enough_memory()`) which is typically either invoked via `accountable_mapping()` or `shmem_acct_size()`.

Note that no mode prevents demand paging, which is a *vital* component of linux systems – when processes fork, the memory of the forking process is shared with the child process using a Copy-on-Write (CoW) mechanism which marks the copied pages as read-only while still pointing at underlying physical pages, allowing for child process memory to only be separately allocated if changed. If the memory actually had to be copied each time this fundamental operation would be considerably slower.

Let's take listing 4-1 and update it to write to the first page as shown in Listing 4-3.

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <sys/mman.h>

int main(void)
{
    char *ptr = mmap(NULL, 20000, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    ptr[0] = 'x';

    if (ptr == MAP_FAILED || munmap(ptr, 20000) != 0) {
        perror("mmap");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

```

Listing 4-3: mmap touch page then munmap

We can now see that the first page of memory has now been allocated and observation of the binary `/proc/$PID/pagemap` interface indicates that it has also been mapped as expected as shown in Listing 4-4.

```

$ cat /proc/$PID/smmaps

7ffff7fbf000-7ffff7fc4000 rw-p 00000000 00:00 0
Size:                20 kB

Rss:                  4 kB

```

Listing 4-4: /proc/\$PID/smmaps output after mmap() and touch 1st page

So to recap – each process has a struct `mm_struct` object describing its address space which contains a collection of struct `vm_area_struct` objects describing distinct virtual allocations (known as VMAs). When memory is allocated virtually it does not necessarily allocate physical memory or map it into the virtual address space, as this can be deferred to the time of access via *page fault* in a process known as *demand paging*.

Let's examine each of these concepts in more detail, but firstly taking a step back and looking at userland allocation as a whole, as shown in Figure 4-1.

4.2 Userland memory from 50,000 feet

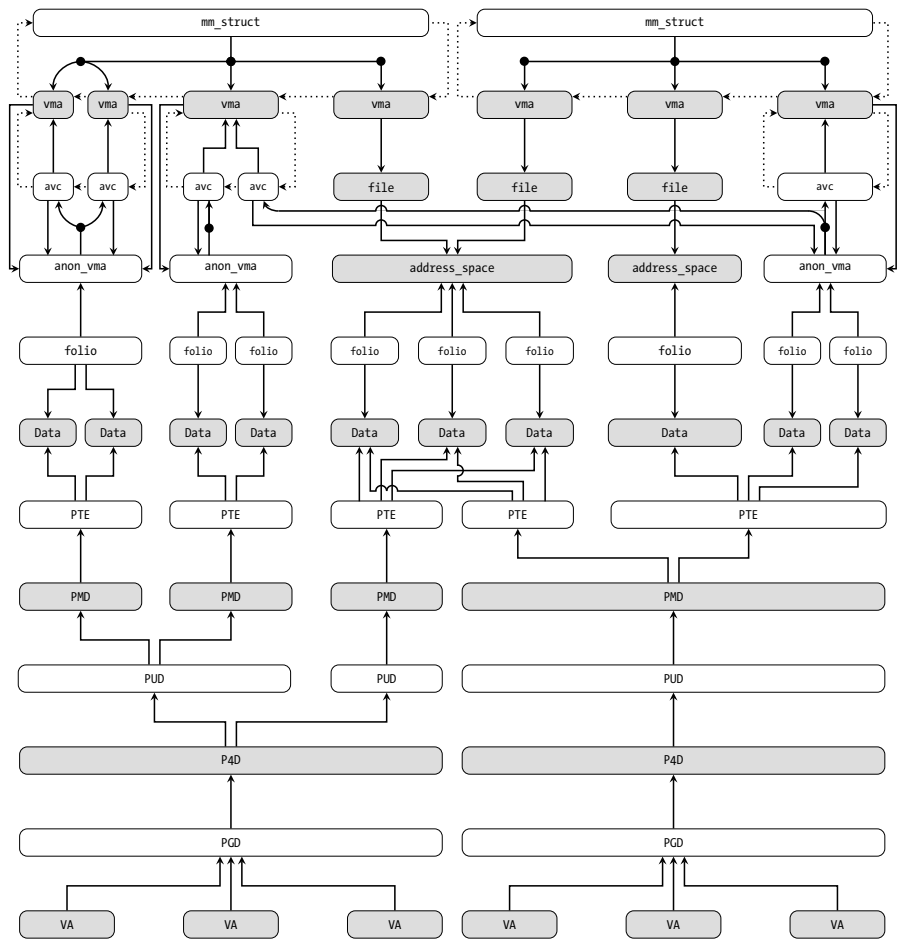
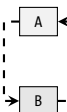


Figure 4-1: Overview of userland memory allocation

Note that:

-  Denotes that B is a member of a list declared in A.
-  Denotes that B is a node in a red/black tree rooted in A.

4.2.1 Describing userland memory

This diagram may seem a little overwhelming at first, so let's examine each element in turn:

`struct mm_struct` – This is the core data structure describing a process’s entire address space, most notably containing a list and an *interval tree** of Virtual Memory Areas or VMAs which describe each valid range of userland memory. Described in section 4.3.

`struct vm_area_struct` (VMAs) – This is one of the most important data structures in the entire memory management subsystem and describes valid userland virtual memory ranges and their attributes. Accessing memory within a range described by a VMA is valid, accessing memory outside this range is not. Described in section 4.4.

`struct anon_vma_chain` – Virtual memory is in a state of constant flux – VMAs are often split and expanded, processes are forked and take CoW mappings of their parent’s folios and yet throughout this we must track to which VMA folios belong and vice-versa.

There is, as a result, a many-to-many mapping between VMAs and underlying folios. Give that each anonymous (i.e. non-file backed) block of memory is described by an `anon_vma`, we need an object to ‘glue’ these and VMAs together – the `anon_vma_chain` object is this glue. This is described in detail in Chapter 7.

`struct file` – A single file can be mapped from multiple different processes. When a file is open in a process, a file object describes the open file and usefully for the purposes of file-backed memory mapping, references the underlying `address_space` which describes the folios which back the file.

The file object therefore resembles the `anon_vma_chain` object in that it provides ‘glue’ between file folios and VMAs in a many-to-many mapping, however it suffers less complexity as each process possesses one and only one file object for each open file and file-backed VMAs are not subject to the same splitting and expansion as anonymous ones. Described in section 4.5.

`struct anon_vma` – Each folio which forms part of an anonymous memory mapping needs to be able to trace back to the VMAs associated with it, which it does via the folio’s `mapping` field. The object to which they refer is the `anon_vma` which groups all such folios together. This object, along with `anon_vma_chain` forms the *reverse mapping* from folios to VMAs. Described in detail in Chapter 7.

`struct address_space` – Describes an element that resides within the *page cache*, which is the in-memory cached representation of files or other ‘cacheable’ objects such as block device buffers.

The folios which make up entries in the page cache must be able to refer back to their owning VMAs, and this object provides the ‘glue’ for them to be able to do so. As with `anon_vma` objects, these refer to their `address_mapping` via the `mapping` field. Described in section 4.5.

*. An interval tree is a red/black tree designed to efficiently locate elements containing a value within intervals.

struct folio – The metadata describing either a single base page of memory or a compound aggregate of them as provided by the physical allocator. Described in section 2.2.

Data pages – These are the underlying base pages of physical memory which contain the actual data referenced by the virtual mappings and kernel metadata structures.

Page tables (PGD - PTE) – These are sparse data structures, each occupying a base page of memory in x86-64, providing a mapping between virtual and physical addresses. See the previous chapter for a detailed examination of page table structure.

Virtual Addresses (VAs) – These are the memory addresses used in userland familiar to all programmers and the net result of all the rest of the memory management machinery.

We will spend the rest of this chapter describing each of the process memory-specific metadata structures referenced above in considerable detail.

4.3 The process address space

Each process has its memory state stored in `struct mm_struct` objects. These contain all the information the kernel needs to describe a *userland* process's virtual memory address space and are referenced by the process state object's `struct task_struct`'s `mm` field.

Kernel threads do not require userspace mappings, but it would be inefficient to flush the TLB each time you scheduled a kernel thread, so the kernel uses a trick known as a *lazy TLB* which has existed in the kernel for a very long time.

This stores the *previously* scheduled task's `struct mm_struct` object in the `active_mm` field rather than the `mm` field, which is set to `NULL`*. Since the hardware will have the page table mappings and PGD loaded from this process, it's vital that we increment a reference counter and mark that we are relying on this mapping remaining in place. This is performed by the scheduler in `context_switch()` as shown in Listing 4-5.

```

5130 static __always_inline struct rq *
5131 context_switch(struct rq *rq, struct task_struct *prev,
5132               struct task_struct *next, struct rq_flags *rf)
5133 {
5134     :
5143     /*
5144      * kernel -> kernel lazy + transfer active
5145      * user -> kernel lazy + mmgrab() active
5146      */

```

*. This is not always the case, as kernel threads can actually 'borrow' a `struct mm_struct` altogether via `kthread_use_mm()`, but most of the time kernel threads do not do this.

```

5147      * kernel -> user  switch + mmdrop() active
5148      * user -> user  switch
5149      */
5150      if (!next->mm) {                                // to kernel
5151          enter_lazy_tlb(prev->active_mm, next);
5152
5153          next->active_mm = prev->active_mm;
5154          if (prev->mm)                                // from user
5155              mmgrab(prev->active_mm);
5156          else
5157              prev->active_mm = NULL;
5158      } else {                                        // to user
5159          :
5170          if (!prev->mm) {                            // from kernel
5171              /* will mmdrop() in finish_task_switch(). */
5172              rq->prev_mm = prev->active_mm;
5173              prev->active_mm = NULL;
5174          }
5175      }
5176      :
5185      return finish_task_switch(prev);
5186  }

```

Listing 4-5: kernel/sched/core.c: *context_switch()*

The struct `mm_struct` reference count (maintained in the atomic `mm->mm_count` field) is incremented by `mmgrab()` and decremented by `mmdrop()`. The architecture-specific means of indicating a lazy TLB state is performed by `enter_lazy_tlb()`.

The key purpose of `mm_struct` however is to store the state of a process's virtual memory address space, most fundamentally a pointer to its PGD table (a virtual equivalent of the assignment of the physical address as the root of the virtual memory mapping in hardware), a collection of all its VMAs (discussed in detail shortly) and important locks.

Since the data structure encompasses a great many fields and to keep things focused, we examine only the core fields as shown in Listing 4-6.

```

486  struct mm_struct {
487      struct {
488          struct vm_area_struct *mmap;                /* list of VMAs */
489          struct rb_root mm_rb;
490          u64 vmacache_seqnum;                        /* per-thread vmacache
491      */
492      :
493      unsigned long (*get_unmapped_area) (struct file *filp,
494          unsigned long addr, unsigned long len,
495          unsigned long pgoff, unsigned long flags);
496      :
497      unsigned long mmap_base;                        /* base of mmap area */

```

```

:
503         unsigned long task_size;          /* size of task vm space */
504         unsigned long highest_vm_end;      /* highest vma end address */
505         pgd_t * pgd;
:
517     /**
518      * @mm_users: The number of users including userspace.
519      *
520      * Use mmget()/mmget_not_zero()/mmput() to modify. When this
521      * drops to 0 (i.e. when the task exits and there are no other
522      * temporary reference holders), we also release a reference
on
523      * @mm_count (which may then free the &struct mm_struct if
524      * @mm_count also drops to 0).
525      */
526     atomic_t mm_users;
527
528     /**
529      * @mm_count: The number of references to &struct mm_struct
530      * (@mm_users count as 1).
531      *
532      * Use mmgrab()/mmdrop() to modify. When this drops to 0, the
533      * &struct mm_struct is freed.
534      */
535     atomic_t mm_count;
536
537 #ifndef CONFIG_MMU
538         atomic_long_t pgtables_bytes;      /* PTE page table pages */
539 #endif
540         int map_count;                     /* number of VMAs */
541
542         spinlock_t page_table_lock; /* Protects page tables and some
543                                     * counters
544                                     */
545     /**
546      * With some kernel config, the current mmap_lock's offset
547      * inside 'mm_struct' is at 0x120, which is very optimal, as
548      * its two hot fields 'count' and 'owner' sit in 2 different
549      * cachelines, and when mmap_lock is highly contended, both
550      * of the 2 fields will be accessed frequently, current layout
551      * will help to reduce cache bouncing.
552      *
553      * So please be careful with adding new fields before
554      * mmap_lock, which can easily push the 2 fields into one
555      * cacheline.
556      */
557     struct rw_semaphore mmap_lock;

```

```

558
559         struct list_head mmlist; /* List of maybe swapped mm's. These
560                                 * are globally strung together off
561                                 * init_mm.mmlist, and are protected
562                                 * by mmlist_lock
563                                 */
564         :
575         unsigned long def_flags;
576
577         /**
578          * @write_protect_seq: Locked when any thread is write
579          * protecting pages mapped by this mm to enforce a later COW,
580          * for instance during page table copying for fork().
581          */
582         seqcount_t write_protect_seq;
583         :
603         unsigned long flags; /* Must use atomic bitops to access */
604         :
610         /*
611          * "owner" points to a task that is regarded as the canonical
612          * user/owner of this mm. All of the following must be true in
613          * order for it to be changed:
614          *
615          * current == mm->owner
616          * current->mm != mm
617          * new_owner->mm == mm
618          * new_owner->alloc_lock is held
619          */
620         struct task_struct __rcu *owner;
621         :
675     } __randomize_layout;
676
677     /*
678     * The mm_cpumask needs to be at the end of mm_struct, because it
679     * is dynamically sized based on nr_cpu_ids.
680     */
681     unsigned long cpu_bitmap[];
682 };

```

Listing 4-6: include/linux/mm_types.h: Simplified struct mm_struct

Examining each field:

- **mmmap** – Points to the first VMA in the address space, which are connected together as a linked list via the `vm_area_struct->vm_prev`, `vm_next` pointers. VMAs within an `mm_struct` are kept in sorted order so this will also be the VMA with the lowest address in the process address space.
- **mm_rb** – The root node of a red/black tree used to efficiently navigate through VMAs during search operations.

- `vmacache_seqnum` – A ‘sequence number’ used to keep track of whether the current VMA cache is valid. These caches are kept per-thread in the `struct task_struct->vmacache` field whose validity is tied to matching their own sequence number to this one. See section 4.4.5 for more on VMA caches.
- `get_unmapped_area` – The default method used to find unmapped memory within the process address space used for file-backed mappings if no VMA-specific one is specified and for all non-shared anonymous mappings. For x86-64 this is set in `arch_pick_mmap_layout()` which typically sets this to `arch_get_unmapped_area_topdown()`. We examine this in more detail in section 4.4.
- `mmap_base` – The minimum address from which `mmap()` allocations will be performed *if no fixed address is specified*. This is set up in `setup_new_exec()` which in turn invokes `arch_pick_mmap_layout()` and then in the case of x86-64 `arch_pick_mmap_base()`. This is a value which is set relatively high in the address space to avoid the data section of a process (expanded by the `brk()` system call), but low enough to avoid the stack which is allocated from a high address and grows downwards. It is also typically randomised for security reasons.
- `task_size` – The *maximum* virtual memory size the process can occupy. For x86-64 with 4 page table levels this is 128 TiB (less 4 KiB), with 5 page tables it is 64 PiB (less 4 KiB), i.e. equal to `TASK_SIZE`. See the virtual memory chapter for more on page table mappings.
- `highest_vm_end` – The maximum exclusive upper bound of VMA contained in the address space, i.e. the ‘end’ of the virtual mapping. Updated in functions such as `__vma_link_rb()`, `__vma_adjust()` and `detach_vmas_to_be_unmapped()`.
- `pgd` – A virtual mapping of the highest order page table for the address space, the Page Global Directory (PGD). All page table operations are performed relative to this.
- `mm_users` – See section 4.3.1 for a description of `mm_struct` reference counts.
- `mm_count` – See section 4.3.1 for a description of `mm_struct` reference counts.
- `pgtables_bytes` – Atomic value indicating the number of bytes occupied by PTE page tables in the address space. Accessed via `mm_pgtables_bytes()`, incremented via `mm_inc_nr_ptes()` and decremented by `mm_dec_nr_ptes()`. This is incremented by e.g. `pmd_install()` and decremented by e.g. `free_pte_range()`.
- `map_count` – A count of the number of VMAs contained within the `mm_struct`. This is incremented in `vma_link()`, `__insert_vm_struct()` and `dup_mmap()` and decremented in `__vma_adjust()` and `detach_vmas_to_be_unmapped()`.

- `page_table_lock` – One of the two most important userspace memory locks, ostensibly protecting page tables that do not possess fine-grained locks, but in reality protecting a number of other mechanisms. Discussed in considerably more detail below.
- `mmap_lock` – The other of the two most important userspace locks, which is in actual fact a read/write semaphore. This is discussed in substantial detail, along with `page_table_lock`, in the below section on this topic.
- `mmlist` – A struct `list_head` node used to maintain a linked list of `mm_struct` objects which are maintained on the `init_mm->mmlist` list and protected by `mmlist_lock`. Used by the swap logic and discussed in more detail in the swap chapter.
- `def_flags` – Specifies default VMA flags to be applied to all VMAs. This is used most notably by the `mlock()` functionality to specify `VM_LOCKED` when the user invokes `mlockall()` with the `MCL_FUTURE` flag set to lock all newly allocated mappings (this is performed in `apply_mlockall_flags()`, see section 8.2.1 for more details on this specific use of default flags). This flag is applied in the core mapping functions `do_brk_flags()` and `do_mmap()`.
- `write_protect_seq` – Lock used to protect the operation of write-protecting pages for e.g. a Copy-on-Write (COW) operation (used in `copy_page_range()` for copying page table mappings) and checked by GUP in the lockless path in `lockless_pages_from_mm()`.
- `flags` – Contains a bitmap of flags describing attributes of the `mm_struct`, discussed below in section 4.3.5.
- `owner` – Indicates which process, represented by a struct `task_struct`, ‘owns’ this virtual address space. This is only available if the memory cgroup, `memcg`, is enabled, via `CONFIG_MEMCG`, and is initialised by `mm_init_owner()` (set to a process in `mm_init()`), cleared by `mm_clear_owner()` (used in an error branch in `copy_process()`) and assigned to the next owner via `mm_update_next_owner()` (invoked on `execve()` in `exec_mmap()` and on exit in `exit_mm()`).
- `cpu_bitmap` – Used by `mm_cpumask()` to provide a bitmap of type `cpumask_t` indicating to which CPUs this `mm_struct` is affinited. Notably used by TLB logic to determine which cores actually need to have TLB cache actions performed in relation to a process.

4.3.1 *mm_struct* reference counting

As described above, there are *two* reference counts present in the struct `mm_struct` object – `mm_users` and `mm_count`. Examining each in turn:

4.3.1.1 *mm_users*

Incremented by `mmget()` (or, if we need to be cautious about whether or not we have already reached zero users, `mmget_not_zero()`) and decremented by `mmaput()` (or, optionally, asynchronously via `mmaput_async()`.)

Counts the number of *userland* references to the process address space. If this value is non-zero then `mm_count` will also be non-zero (i.e. userspace as a whole counts as 1 `mm_count`).

When this reference count reaches zero, all userland-specific metadata is torn down as are all userland mappings via `__mmapput()`. Importantly, the kernel mappings remain and so does the object as a whole, with `mm_count` simply being decremented, only if this reaches zero does the object as a whole get torn down (see below).

Other users which increment this reference counter might include a forked process (via `copy_process()` which invokes `copy_mm()` which invokes `mmget()` in turn), the `move_pages()` syscall for moving pages between nodes via `kernel_move_pages()`, among many other users, often via the `get_task_mm()` and `access_process_vm()` functions.

4.3.1.2 `mm_count`

Incremented by `mmgrab()` and decremented by `mmdrop()`.

Counts the number of *kernel* references to the process address space (importantly, including the process itself i.e. its `struct task_struct` object).

The reference count is incremented when lazy TLB mode is entered, either by a kernel process reusing the existing process address space mappings without swapping out its PGD (which involves expensive TLB invalidation) or a task such as exiting a process that needs to tear things down but keep the `mm_struct` around for a while, e.g. in `exit_mm()`. See the virtual memory chapter for more on the TLB. Any non-zero number of `mm_users` count as 1 `mm_count`.

If the reference count drops to zero, this invokes `__mmdrop()` which frees the `mm_struct`'s PGD and the `mm_struct` itself among other housekeeping tasks. By this point all user mappings will already have been freed.

4.3.2 *The initial process address space*

Each processor has an *idle task* which represents the task that the scheduler switches in when there is no other processor work to perform*. It is, ostensibly, the process from which all other processes are derived.

The progenitor of each CPU's idle task is created very early on in kernel initialisation and is declared statically as `init_task`. Its `active_mm` field is assigned to `init_mm` which represents the shared `mm_struct` of each idle task. The `mm` field is set to `NULL` because this is a kernel process and thus defined by not possessing its own individual `mm_struct` as shown in Listing 4-7.

*. we use 'task' and 'process' interchangeably – the data structure representing a process is `struct task_struct` and from a kernel point of view a process is defined by possessing an individual task object, however in the common vernacular we would describe the same entity as a process. Additional confusion can arise because the kernel treats each individual thread as a separate process whereas a userland programmer might consider each to be part of the same process, however from the kernel point of view, we see these as processes with shared `mm_struct` objects.

```

30 struct mm_struct init_mm = {
31     .mm_rb          = RB_ROOT,
32     .pgd            = swapper_pg_dir,
33     .mm_users       = ATOMIC_INIT(2),
34     .mm_count       = ATOMIC_INIT(1),
35     .write_protect_seq = SEQCNT_ZERO(init_mm.write_protect_seq),
36     MMAP_LOCK_INITIALIZER(init_mm)
37     .page_table_lock = __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
38     .arg_lock        = __SPIN_LOCK_UNLOCKED(init_mm.arg_lock),
39     .mmlist          = LIST_HEAD_INIT(init_mm.mmlist),
40     .user_ns         = &init_user_ns,
41     .cpu_bitmap      = CPU_BITS_NONE,
42 #ifdef CONFIG_IOMMU_SVA
43     .pasid           = INVALID_IOASID,
44 #endif
45     INIT_MM_CONTEXT(init_mm)
46 };

```

Listing 4-7: mm/init-mm.c: *init_mm*

Note that the `mm_users` field is initialised to 2. This ensures that no spurious `mmap()` attempts to tear down the object.

The global struct `mm_struct` object `init_mm` is used for a number of different purposes (this is not an exhaustive list):

- As the *active* `mm_struct` for the CPU idle tasks as described above and during CPU initialisation e.g. in `cpu_init()` where it is the very first active `mm_struct` assigned to a CPU and in `finish_cpu()` invoked when a CPU goes offline in a hotplug system for example.
- During kernel initialisation, where we need to have both a PGD established to perform early memory mappings (e.g. in `fill_p4d()`) as well as access to a shared `page_table_lock` in order to synchronise page table updates across CPUs (e.g. in `__kernel_physical_mapping_init()`).
- As the `mm_struct` parameter for kernel-specific page table functions like `pgd_offset_k()` (which simply uses it to access the kernel PGD), or `__pte_alloc_kernel()` which uses it to obtain the `page_table_lock` for kernel PTE allocations.
- As the head of the list of `mm_struct` objects for process address spaces used for the swap logic in the `init_mm.mmlist` field, manipulated in e.g. `try_to_unuse()`, `drain_mmlist()` and `try_to_unmap_one()`.
- In x86-64, when flushing the TLB via `flush_tlb_func()` and lazy TLB is in force (this is cheaper than a minimum flush), or ‘leaving’ the current `mm_struct` via `leave_mm()` used to force a TLB flush by switching to the `init_mm` if the CPU idle task is set up that way. This function is also used by the x86-64 specific `use_temporary_mm()` which is used by the text poke functionality via `__text_poke()` for live kernel patching (e.g. via KGDB).

- When a ‘spurious fault’ occurs in x86-64 in `spurious_kernel_fault()` where, for efficiency reasons, a TLB entry may be permitted to remain stale with the fault handled here. The `init_mm` object is used simply to obtain the kernel-specific PGD to check to see if the mapping is valid.

The key purpose is to both have an early kernel process address space available for kernel initialisation and to also maintain global kernel (i.e. non-userland) state like the head of the `mmlist` object and the kernel PGD, locks and other global memory state.

The PGD used by `init_mm` is very important, as it contains the top-level of *all* kernel mappings. This is assigned to the static object, `swapper_pg_dir` (in x86-64 this is an alias for `init_top_pgt`).

4.3.3 Kernel PGD maintenance

Kernel page table mappings are rooted in the PGD defined in `swapper_pg_dir` and assigned to the `init_mm`.

Userland mappings are always present whether in userland or kernel mode. On x86-64, which kernel mappings are present is dependent on whether or not an important security mitigation, termed *Page Table Isolation (PTI)*, is enabled. This was implemented in order to address the ‘meltdown’ security vulnerability (a detailed discussion of which is outside the scope of this book).

The kernel needs to have userland mappings in place for process-specific transactions such as system calls in order that it can interact with userland memory, e.g. to copy memory between kernel and userland so these must be maintained between each.

4.3.3.1 Page Table Isolation

When the PTI mitigation is *not* in place, then the mappings are simple – both kernel and userland contain kernel and userland mappings, with the kernel mappings maintaining page flags rendering them unusable in userland, and kernel mappings maintained such that the TLB is not flushed between context switches (see the virtual memory chapter for more on these flags).

The feature is *made available* via `CONFIG_PAGE_TABLE_ISOLATION` but even if this configuration option is set, the mitigation will only be enabled if the `X86_FEATURE_PT` flag is set, which is only enabled if the CPU is vulnerable to meltdown, checked in `pti_check_boottime_disable()`.

When PTI is enabled, then PGDs are allocated as order-1, i.e. 8 KiB, folios and contain two separate PGD tables, one for userland mode and one for kernel mode.

In kernel mode (the first 4 KiB of the order-1 PGD) all kernel and userland mappings are present, however userland mappings are marked non-executable in order to avoid any mistake in page table assignment in userland mode.

In userland mode, however (the last 4 KiB of the order-1 GPD), only a bare minimum set of kernel mappings are present, the minimum required

to permit kernel transition. By using order-1 PGD pages, we need only flip a bit to switch between the two, which is what `kernel_to_user_pgdp()` and `user_to_kernel_pgdp()` do.

Importantly, note that the `struct mm_struct->pgd` field is set to the *kernel* portion of the order-1 PGD, i.e. the lower portion, with the userland mappings kept in the adjacent base page. This means initial transitions to another process address space, performed in the kernel, maintain kernel mappings.

When a PGD entry needs to be updated for userland, the function `pti_set_user_pgtbl()` is invoked which, if the PTI mitigation is in place, invokes `__pti_set_user_pgtbl()`. This uses `kernel_to_user_pgdp()` to update the userland portion of the PGD, and adds an equivalent in the kernel portion of the PGD with execution disabled as previously discussed.

Switching between processes is ultimately performed by `switch_mm_irqs_off()`, typically on context switch via `context_switch()` (shown in listing 4-5) but also in other code paths via `switch_mm()` or `activate_mm()`.

These functions change the PGD the CPU is currently using in the hardware, which for x86-64 is specified by the `cr3` control register whose value is determined by `build_cr3()` and set by `load_new_mm_cr3()*`.

Note however that we remain in kernel mode as other actions need to be performed before actually switching into user mode if that is what is required. As a result, as discussed above, we use the `struct mm_struct->pgd` field which is pointing at the kernel PGD table, and thus this logic remains largely unmodified.

The real change arises in the *entry* code that returns from a system call or interrupt (including scheduler interrupts), for example `common_interrupt_return()` and `entry_SYSCALL_64()` which both invoke the `SWITCH_TO_KERNEL_CR3()` and `SWITCH_TO_USER_CR3_STACK()` assembly macros which fixup the PGDs accordingly before loading them into `cr3`.

I have focused on the PTI logic in a fair bit of detail here as a result of it complicating the otherwise fairly clear logic around kernel and userland page table mappings, which without this mitigation are simply stored in the same page tables, only with the kernel mappings explicitly made unavailable to userland.

4.3.3.2 Maintaining kernel mappings between processes

Early kernel code updates `swapper_pg_dir` to include the direct mapping, the `vmalloc` area, and other crucial kernel-wide mappings. As a result, a fairly vast swathe of kernel memory addresses are already established in `init_mm` and thus inherited by the proceeding processes which each copy these mappings from the former on fork and execution.

When it comes to examining how kernel mappings are propagated when new userland processes are created we need to consider the two classic execution operations in a Unix system – `fork()` and `execve()`:

*. There is some additional logic here relating to PCIDs, however this is out of scope for the book.

On *fork*, page tables are copied via `copy_page_range()` which was invoked in turn by `dup_mmap()`, `dup_mm()`, `copy_mm()`, `copy_process()` and `kernel_clone()`. This copies the entire contents of the page tables and thus new processes receive a copy of all existing kernel mappings in the PGD as well as all existing userland mappings.

On *execve* either `kernel_execve()` will be invoked or userland will call one of the `exec_ve()` variants which invoke `do_execveat_common()`. Both of these routes ultimately invoke `mm_alloc()` which initialises the newly allocated struct `mm_struct` object via `mm_init()`.

This function allocates the PGD via `mm_alloc_pgd()` which ultimately invokes the architecture-specific `pgd_alloc()`, here examining the x86-64 implementation as shown in Listing 4-8.

```

424 pgd_t *pgd_alloc(struct mm_struct *mm)
425 {
426     pgd_t *pgd;
427     :
430     pgd = _pgd_alloc();
431
432     if (pgd == NULL)
433         goto out;
434
435     mm->pgd = pgd;
436     :
446     /*
447      * Make sure that pre-populating the pmds is atomic with
448      * respect to anything walking the pgd_list, so that they
449      * never see a partially populated pgd.
450      */
451     spin_lock(&pgd_lock);
452
453     pgd_ctor(mm, pgd);
454     :
457     spin_unlock(&pgd_lock);
458
459     return pgd;
460     :
467 out:
468     return NULL;
469 }
```

Listing 4-8: `arch/x86/mm/pgtable.c: pgd_alloc()`

We exclude pre-population logic here because it is typically not relevant in a modern system. What is relevant, however, is `pgd_ctor()` as shown in Listing 4-9.

```

123 static void pgd_ctor(struct mm_struct *mm, pgd_t *pgd)
124 {
```

```

125      /* If the pgd points to a shared pagetable level (either the
126      ptes in non-PAE, or shared PMD in PAE), then just copy the
127      references from swapper_pg_dir. */
128      if (CONFIG_PGTABLE_LEVELS == 2 ||
129          (CONFIG_PGTABLE_LEVELS == 3 && SHARED_KERNEL_PMD) ||
130          CONFIG_PGTABLE_LEVELS >= 4) {
131          clone_pgd_range(pgd + KERNEL_PGD_BOUNDARY,
132                          swapper_pg_dir + KERNEL_PGD_BOUNDARY,
133                          KERNEL_PGD_PTRS);
134      }
135
136      /* list required to sync kernel mapping updates */
137      if (!SHARED_KERNEL_PMD) {
138          pgd_set_mm(pgd, mm);
139          pgd_list_add(pgd);
140      }
141  }

```

Listing 4-9: arch/x86/mm/pgtable.c: pgd_ctor()

This directly copies from the swapper_pg_dir global kernel PGD object for all addresses in the kernel range, as defined by KERNEL_PGD_BOUNDARY (providing the index of the first PGD entry within the kernel address range) and KERNEL_PGD_PTRS (indicating the number of entries of kernel mappings. This is done by clone_pgd_range() as shown in Listing 4-10.

```

1232 /*
1233  * clone_pgd_range(pgd_t *dst, pgd_t *src, int count);
1234  *
1235  * dst - pointer to pgd range anywhere on a pgd page
1236  * src - ""
1237  * count - the number of pgds to copy.
1238  *
1239  * dst and src can be on the same page, but the range must not overlap,
1240  * and must not cross a page boundary.
1241  */
1242 static inline void clone_pgd_range(pgd_t *dst, pgd_t *src, int count)
1243 {
1244     memcpy(dst, src, count * sizeof(pgd_t));
1245 #ifdef CONFIG_PAGE_TABLE_ISOLATION
1246     if (!static_cpu_has(X86_FEATURE_PTI))
1247         return;
1248     /* Clone the user space pgd as well */
1249     memcpy(kernel_to_user_pgdp(dst), kernel_to_user_pgdp(src),
1250           count * sizeof(pgd_t));
1251 #endif
1252 }

```

Listing 4-10: arch/x86/include/asm/pgtable.h: clone_pgd_range()

The fundamental task here is very simple – copy a single PGD entry from `swapper_pg_dir` to the target PGD, however there is additional complexity arising from PTI – if PTI is active, then we also need to copy the minimal user space kernel mappings from the user space portion of the initial PGD to this new PGD.

Note that `pgd_alloc()` holds the global `pgd_lock` over the operation, which protects `pgd_list` containing all PGDs in the system and permits this new PGD to be added via `pgd_list_add()`.

This list is used during early initialisation in x86-64 (most notably of the direct mapping, see the virtual memory chapter for a detailed examination of this) via `sync_global_pgds()` to ensure that all kernel mappings are kept updated, however importantly, after the kernel mappings are initialised, no region of virtual memory that can be mapped into is large enough that the PGD need be updated. Note that most kernel virtual addresses use the direct mapping therefore this is really not all that limiting.

Since all of the PGD entries that point to P4D mappings are copied, then all of the kernel mappings are *shared* between each process (for PTI, in kernel mode only, as described above). Thus all changes at P4D or below are propagated, since all kernel mappings point at the same P4Ds.

On kernel start the init process, PID 1, is initialised in `rest_init()` (invoked ultimately from `start_kernel()`) which invokes `user_mode_thread()` setting this process to execute `kernel_init()`. This ultimately calls `run_init_process()` which, in turn, invokes `kernel_execve()` as described above.

4.3.4 Process address space locking

The two key locks at the process address space level are the spinlock `struct mm_struct->page_table_lock` and `struct mm_struct->mmap_lock` which is a read/write semaphore.

These locks are absolutely critical as they are very heavily utilised and thus it is useful to examine their usage in detail:

4.3.4.1 Page table lock

As described in the virtual memory chapter in section 3.1.4, this lock is used for page table locking at the less-contended higher page table levels, typically all levels above PMD. There are some nuances to this – operations on kernel mappings typically use this lock for PMDs unconditionally, e.g. in `__pte_alloc_kernel()` which is invoked by `vmalloc` via `pte_alloc_kernel_track()` and of course, if the architecture doesn't use split locks for PMD/PTEs at all then this lock will be used for *all* page table updates.

However, there are cases where the lock is used for non-page table operations. This used to be more common in the distant kernel past but has now been narrowed down to only a few instances:

- Protecting *stack expansion* – In `expand_downwards()` (typically used with a stack) and `expand_upwards()` VMA expansion takes place which results in thread unsafe fields being updated, specifically the VMA gap calculation. Since the `mmap_lock` semaphore permits simultaneous read access,

`page_table_lock` is used to place the entire update process in a critical section. This is simply an overloading of an available lock for an entirely different use.

- *Protecting VMA gap calculation – (only applicable if `CONFIG_DEBUG_VM_RB` has been set)* – A calculation of the VMA subtree gap (see the next section on VMAs to see what this means) is performed in `vma_compute_subtree_gap()` which is invoked in turn by `browse_rb()` (via `validate_mm()`). This maintains a `page_table_lock` over the operation, introduced in commit `acf128d048c7: mm: validate_mm browse_rb SMP race condition`. This was added because, if this function were to race with `vma_gap_update()`, data corruption could occur. Since this update function is invoked in `expand_downwards()` and `expand_upwards()`, and might also hold a *read* semaphore at the time (rendering the semaphore useless), using this lock resolves the data race.
- *Protecting the update of the `anon_vma` field and chain linkage in `__anon_vma_prepare()`* – Here there may be multiple threads attempting to perform the same action rendering the update a critical section. Again, `page_table_lock` is a convenient lock to use at this stage. See the section below on reverse mappings for more details on the use of this operation.

4.3.4.2 mmap lock

The `struct mm_struct->mmap_lock` field specifies a read/write semaphore object that is used to synchronise VMAs and any other `mm_struct` state within the address space.

It is a *very* heavily contended lock and used in an enormous number of places, being arguably one of the most important locks in the entire kernel. It is therefore worth exploring in some detail:

The `struct rw_semaphore` object is a lock type which allows concurrent access to critical sections for multiple readers, but importantly only permits writes when no readers are present. It is designed such that readers cannot starve writers*, though in theory the reverse could occur (however in practice this should never happen).

The lock is designed such that, if threads are simply reading fields, they do not contend one another (the state is in theory immutable and thus there is no reason that they should, though this isn't quite the case, see the above note about the use of `page_table_lock` on VMA expansion).

There are a number of helper functions which wrap common operations:

4.3.4.3 General mmap_lock functions

- `mmap_init_lock()` – Initialises the lock, used in `mm_init()` on initialisation of an `mm_struct`.

*. Under circumstances where locks are acquired and released without deadlock, livelock or programming error. If a read lock failed to be released correctly, then it could indeed starve a waiting writer, but this is a circumstance that should never occur in a correctly functioning kernel.

- `mmap_assert_locked()` – Invokes a *lockdep* assert that a lock is held* and a `VM_BUG_ON_MM()` check which only has in impact if `CONFIG_DEBUG_VM` is set.
- `mmap_assert_write_locked()` – Invokes a *lockdep* assert that a *write* lock is held and a `VM_BUG_ON_MM()` check which only has in impact if `CONFIG_DEBUG_VM` is set.
- `mmap_lock_is_contended()` – A wrapper around `rwsem_is_contended()` which checks to see whether the wait list of threads that wish to perform an operation which is currently unavailable is non-empty. This is used by `show_smaps_rollup()` to avoid egregious contention on access to the `/proc/$pid/smmaps_rollup` interface. This was introduced in mm: proc: smmaps_rollup: do not stall write attempts on `mmap_lock` (commit `ff9f47f6f00c`) to prevent stalling on use of this interface for processes with large memory footprints. Similar `smmaps_rollup` functionality was ported to BPF bringing along this usage with it, so this function is also invoked in `task_vma_seq_get_next()`.

4.3.4.4 `mmap_lock` write functions

Note that we describe the *use* of these in more detail below.

- `mmap_write_lock()` – Acquires a write lock on the `mm_struct`. Wraps `down_write()` on `mmap_lock`. Ultimately invokes `__down_write()` which calls `__down_write_common()` setting state to `TASK_UNINTERRUPTIBLE`, which indicates that while waiting for the lock the thread attempting to acquire the lock should be marked uninterruptible.
- `mmap_write_lock_nested()` – Acquires a *nested* lock on the `mm_struct`. Wraps `down_write_nested()` on `mmap_lock`, which only differs from `down_write()` if `CONFIG_DEBUG_LOCK_ALLOC` is specified. Used to specify that there is `SINGLE_DEPTH_NESTING` in `dup_mmap()`.
- `mmap_write_lock_killable()` – Acquires a write lock on the `mm_struct`. Wraps `down_write_killable()` on `mmap_lock`. Ultimately invokes `__down_write_killable()` which calls `__down_write_common()` setting state to `TASK_KILLABLE`, which indicates that while waiting for the lock the thread attempting to acquire the lock should respond to deadly signals, but otherwise be uninterruptible.
- `mmap_write_trylock()` – Tries to acquire a write lock on the `mm_struct`. Wraps `down_write_trylock()` on `mmap_lock`. Ultimately invokes the function `__down_write_trylock()`.
- `mmap_write_downgrade()` – Converts a write lock on `mm_struct` into a read lock, i.e. ‘downgrading’ it. Wraps `downgrade_write()` on `mmap_lock`. Ultimately invokes the function `__downgrade_write()`.
- `mmap_write_unlock()` – Releases a write lock on the `mm_struct`. Wraps `up_write()` on `mmap_lock`. Ultimately invokes the function `__up_write()`.

*. *lockdep* is a means by which lock dependencies can be checked and locking bugs discovered. It is out of scope for the book.

4.3.4.5 `mmap_lock` read functions

Note that we describe the *use* of these in more detail below.

- `mmap_read_lock()` – Acquires a read lock on the `mm_struct`. Wraps `down_read()` on `mmap_lock`. Ultimately invokes `__down_read()` which calls `__down_read_common()` setting state to `TASK_UNINTERRUPTIBLE`, which indicates that while waiting for the lock the thread attempting to acquire the lock should be marked uninterruptible.
- `mmap_read_lock_killable()` – Acquires a read lock on the `mm_struct`. Wraps `down_read_killable()` on `mmap_lock`. Ultimately invokes `__down_read_killable()` which calls `__down_read_common()` setting state to `TASK_KILLABLE`, which indicates that while waiting for the lock the thread attempting to acquire the lock should respond to deadly signals, but otherwise be uninterruptible.
- `mmap_read_trylock()` – Tries to acquire a read lock on the `mm_struct`. Wraps `down_read_trylock()` on `mmap_lock`. Ultimately invokes the function `__down_read_trylock()`.
- `mmap_read_unlock()` – Releases a read lock on the `mm_struct`. Wraps `up_read()` on `mmap_lock`. Ultimately invokes the function `__up_read()`.
- `mmap_read_unlock_non_owner()` – Releases a read lock on the `mm_struct`. Wraps `up_read_non_owner()` which asserts that the thread that will release it is not the ‘owner’ of the lock. Only has an impact if `CONFIG_DEBUG_LOCK_ALLOC` is defined. Ultimately invokes the function `__up_read()`.

4.3.4.6 Use of `mmap` locks

For the purposes of examining the lifetime and users of a process address space’s `mmap_lock` the differences between these variants is of less importance than whether a. a lock is being acquired/released and b. the lock is read-/write.

In order to get a sense of just how important this lock is in the kernel and how often it is used, let’s visualise the places where the read/write lock is acquired/released.

This lock is quite *so* contended that covering every single usage of it would be impractical. Therefore, the scope has been limited to:

- Exclude initialisation of struct `mm_struct` objects, as you would expect the lock to be held here and it is unlikely to see much contention in this phase.
- Exclude all driver usage, huge page functionality, system V-style shared memory functionality, cgroup and ptrace logic.
- The x86-64 architecture, if logic varies depending on architecture.
- ELF, if examining binary format-specific logic.

Focusing therefore on the *core* users of this lock let’s examine where they are acquired and released, relatively exhaustively. This gives us a sense

of both where these locks are acquired as well as the breadth of functions which contend them.

The subsequent diagrams use this key:

Denotes that an `mmio_lock` is acquired (and typically released) in this function. If an ^r is present this denotes a read lock is acquired, if a ^w is present this denotes that a write lock is acquired. The meaning of the suffix can vary, as specified by accompanying text, e.g. (*Releases*) indicates a lock is released, not acquired.

Denotes that an `mmap_lock` is held on invocation of this function, but acquired/released elsewhere.

syscall: Denotes that this function is a system call rather than a function invocation.

Denotes that this function is an interrupt handler, CPU exception handler, or sits at a transition between user/kernel, rather than a function invocation.

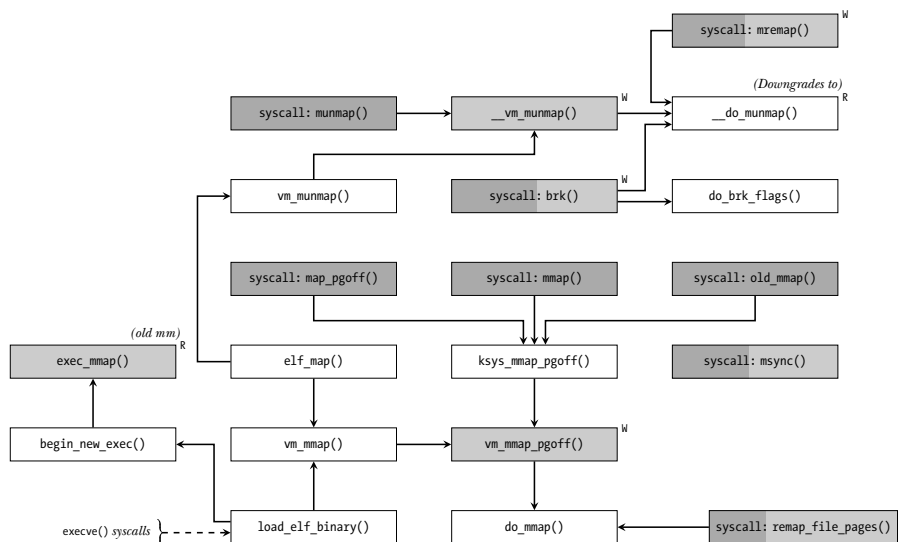


Figure 4-2: `mmap` `mmap_lock` invocations

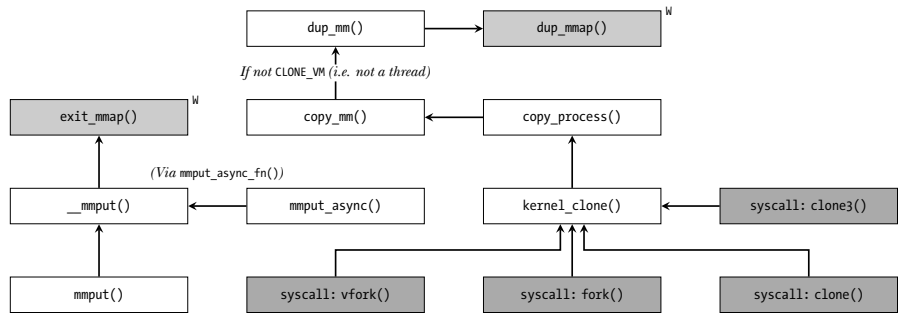
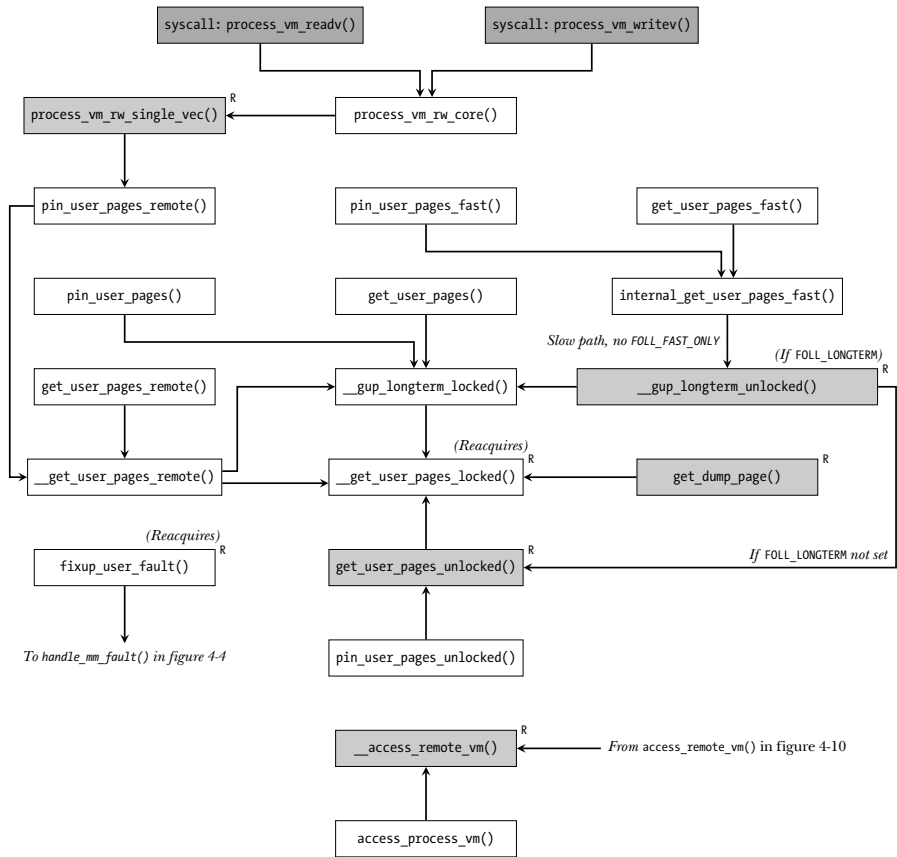
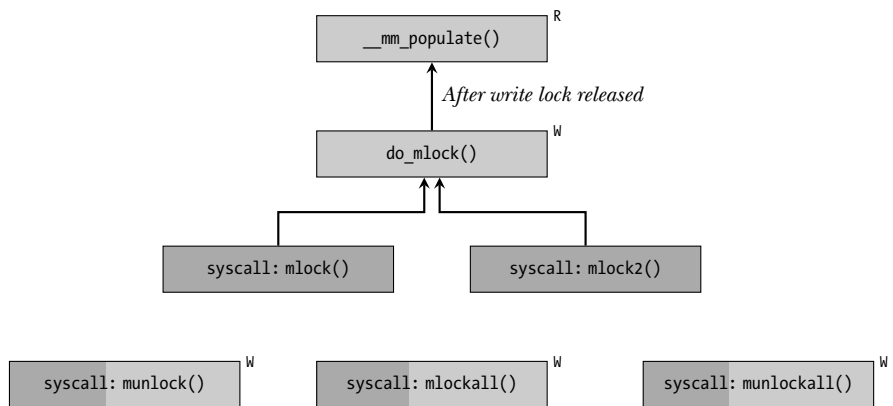


Figure 4-3: Fork `mmap_lock` invocations



Figure 4-4: Page fault `mmap_lock` invocations

Note: If a fault releases the `mmap_lock` then the fault handler returns `VM_FAULT_RETRY` or `VM_FAULT_COMPLETED`.


Figure 4-5: GUP `mmap_lock` invocations

Figure 4-6: `mlock()` `mmap_lock` invocations

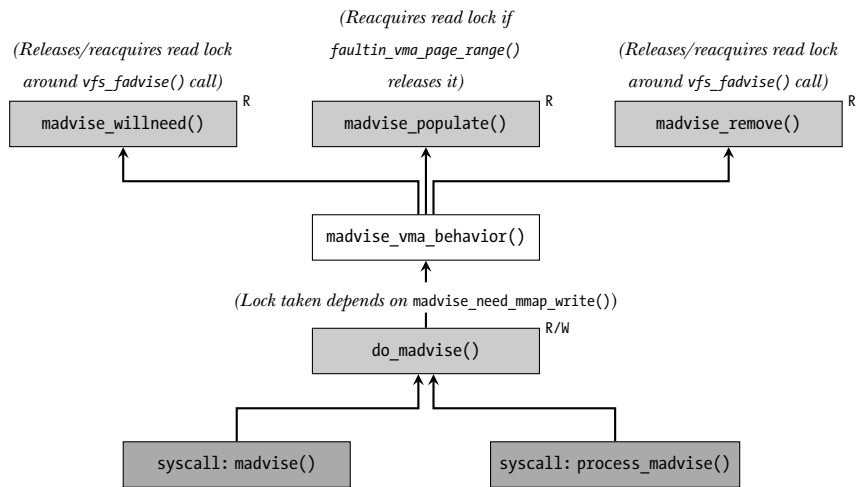
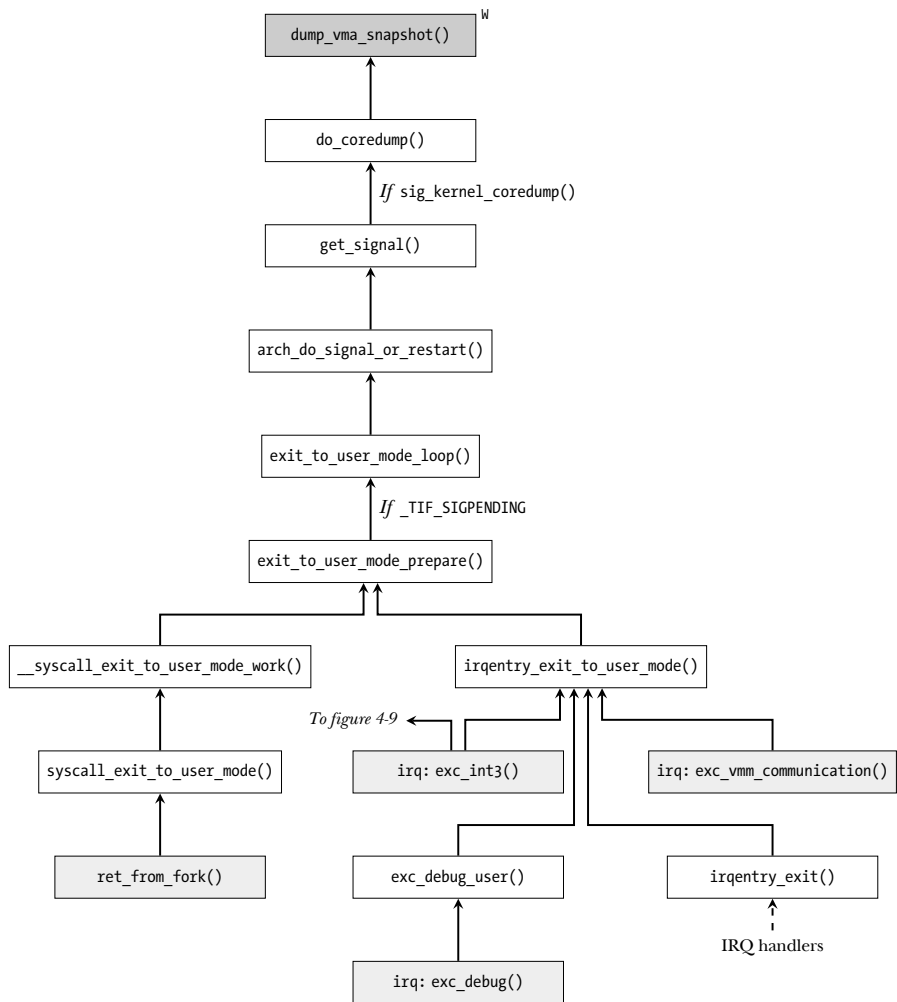


Figure 4-7: `madvise()` `mmap_lock` invocations

Figure 4-8: Core dump (x86-64) `mmap_lock` invocation

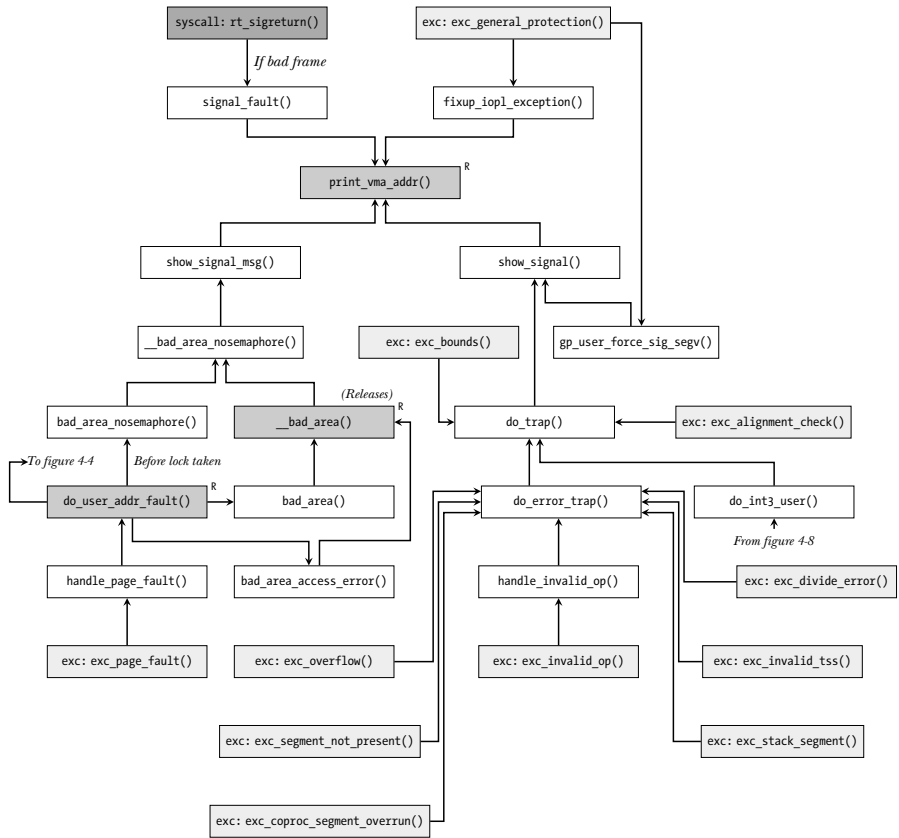


Figure 4-9: CPU exception (x86-64) `mmap_lock` invocations

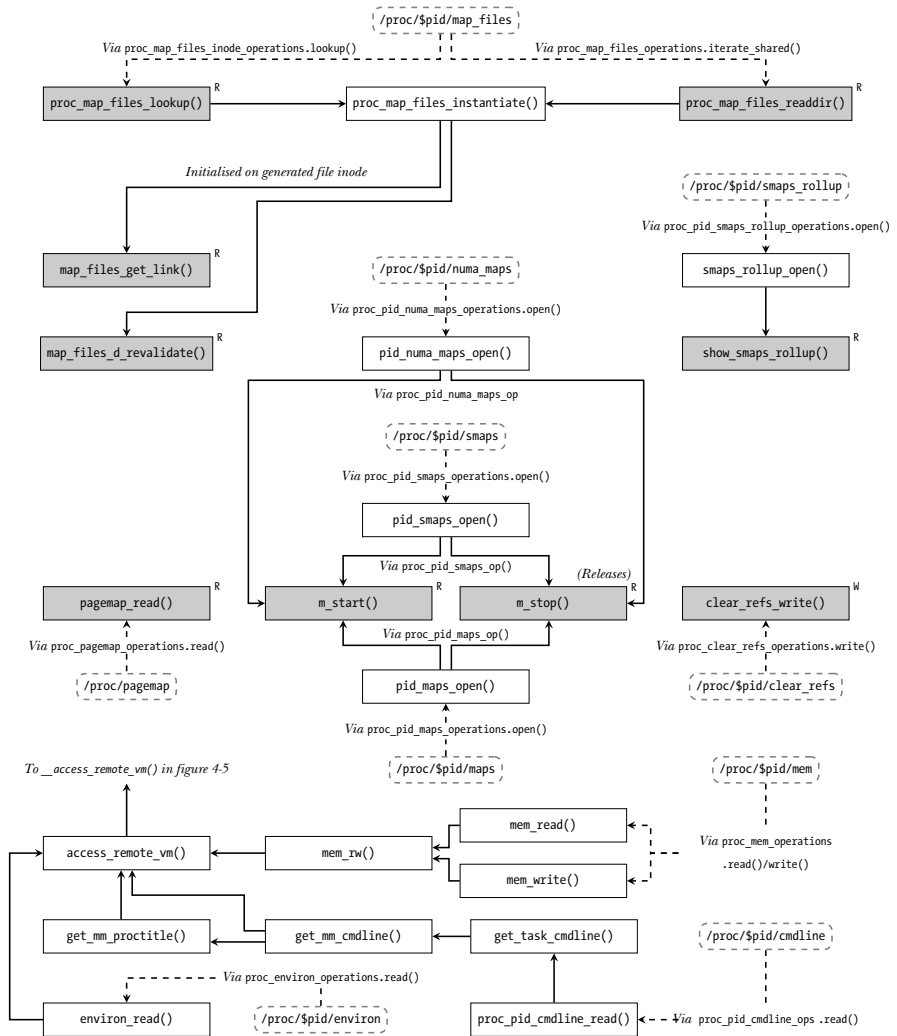


Figure 4-10: procfs mmap_lock invocations

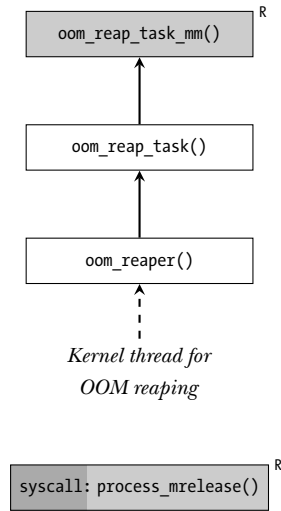


Figure 4-11: Out Of Memory (OOM) killer `mmap_lock` invocations

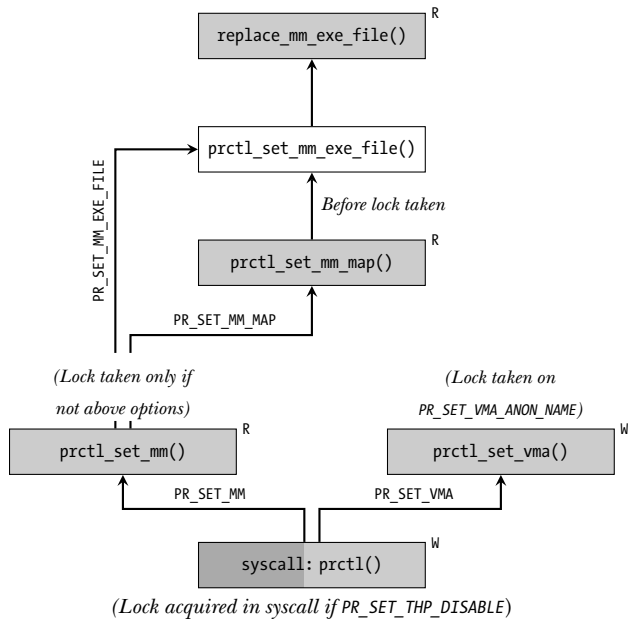


Figure 4-12: `prctl()` `mmap_lock` invocations

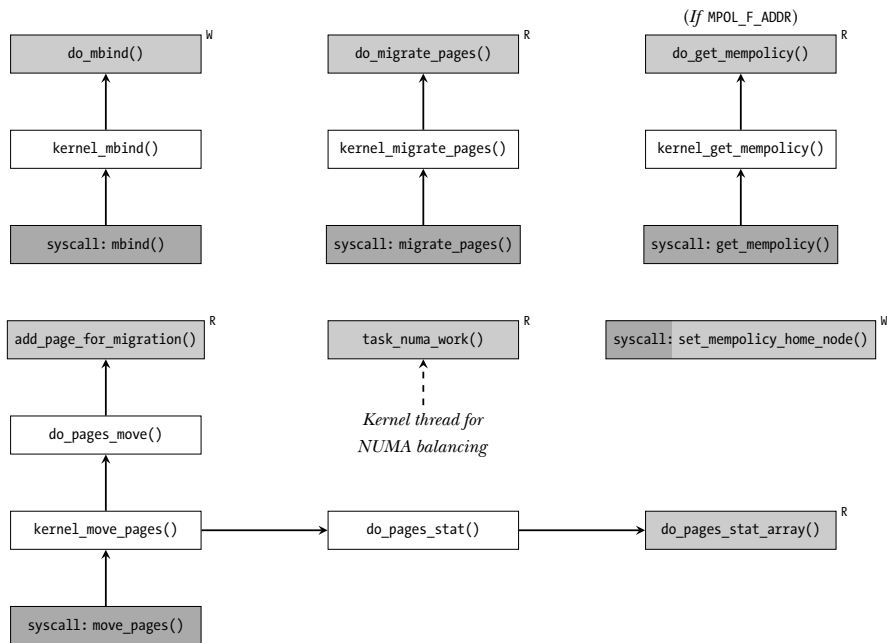


Figure 4-13: NUMA `mmap_lock` invocations

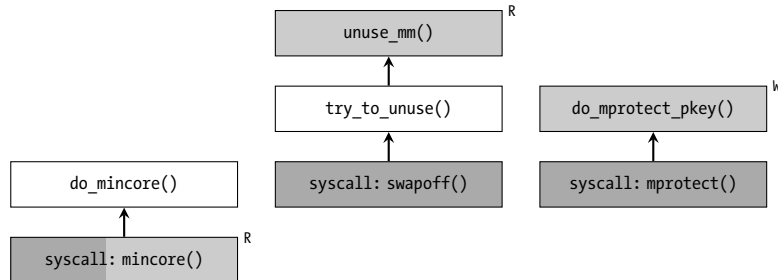


Figure 4-14: Miscellaneous `mmap_lock` invocations

4.3.5 Process address space flags

The struct `mm_struct->flags` field contains a bitmap of fields that describe attributes of a process address space. These are defined in `include/linux/sched/coredump.h`, (despite being located here not all relate to core dumps). Examining each flag:

4.3.5.1 Core dump user flags

A core dump is the means by which, typically on a segfault or otherwise abnormal exit, the memory state of a process is written to disk to a ‘core file’ (specified in `/proc/sys/kernel/core_pattern`) which can be examined after the

fact for debugging purposes (see the */proc/sys/kernel documentation* for more on this).

Note that these indicate *values* not bit indexes. These occupy the first MMF_DUMPABLE_BITS (i.e. 2) bits of the flags field:

- SUID_DUMP_DISABLE – Core dumping is disabled altogether.
- SUID_DUMP_USER – Core dumping will be performed by the process user – this is the default mode, unless the binary being dumped is setuid and */proc/sys/fs/suid_dumpable* indicates a different one.
- SUID_DUMP_ROOT – Core dumping will be performed by the root user.

These are accessed via `__get_dumpable()` (parameterised by the flags themselves) and `get_dumpable()` (parameterised by a `mm_struct`) and set by `set_dumpable()`.

Userland can adjust a process's dump mode via `prctl()` (specifying the `PR_SET_DUMPABLE` mode), though it is now prohibited to set `SUID_DUMP_ROOT` this way. The mode defaults to `SUID_DUMP_USER` if not otherwise specified.

If a binary is setuid then the mode will be determined by */proc/sys/fs/suid_dumpable* – the index matching each of the modes specified above (see the */proc/sys/fs documentation* for more on this). This is determined in `begin_new_exec()` and modified if process credentials change in `commit_creds()`.

4.3.5.2 Core dump filter flags

Which classes of memory that get core dumped are also stored in the `struct mm_struct->flags` fields and specified in */proc/\$pid/coredump_filter* (or via the `coredump_filter` kernel parameter), see the *proc documentation* for more details on how to read and set these.

Note that each VMA can override the dumping process altogether by specifying the `VM_DONTDUMP` flag. And on the other hand, certain kinds of VMAs *always* get dumped, as determined via `always_dump_vma()` – any VMA that has a name assigned to it is assumed to always be important, as well as the `vsyscall` mapping.

Examining each of these flags, which are processed by `vma_dump_size()`:

- MMF_DUMP_ANON_PRIVATE – Dump anonymous, non-huge private VMAs.
- MMF_DUMP_ANON_SHARED – Dump anonymous, non-huge shared VMAs.
- MMF_DUMP_MAPPED_PRIVATE – Dump file-mapped, private VMAs.
- MMF_DUMP_MAPPED_SHARED – Dump file-mapped, shared VMAs (if the file is anonymous, e.g. created by `memfd()`, then this will degrade to `MMF_DUMP_ANON_SHARED`).
- MMF_DUMP_ELF_HEADERS – Dump ELF headers. Requires the `VM_READ` flag to be set and the VMA to not be offset.
- MMF_DUMP_HUGETLB_PRIVATE – Dump hugetlb private mapping, as determined via `is_vm_hugetlb_page()`, i.e. requiring the `VM_HUGETLB` VMA flag to be set. Requires `VMA_SHARED` to not be set.

- `MMF_DUMP_HUGETLB_SHARED` – Dump hugetlb shared mapping, determined as above, only with `VMA_SHARED` set.
- `MMF_DUMP_DAX_PRIVATE` – Dumps a DAX (Direct Access for files system, discussed in *DAX documentation*) private VMA block. Out of scope for the book.
- `MMF_DUMP_DAX_SHARED` – Dumps a DAX shared VMA block. Out of scope for the book.

The default flags are determined by `MMF_DUMP_FILTER_DEFAULT`, setting `MMF_DUMP_ANON_PRIVATE`, `MMF_DUMP_ANON_SHARED`, `MMF_DUMP_HUGETLB_PRIVATE` and `MMF_DUMP_MASK_DEFAULT_ELF` which enables `MMF_DUMP_ELF_HEADERS` if `CONFIG_CORE_DUMP_DEFAULT_ELF_HEADERS` is configured.

4.3.5.3 Other flags

There are a number of other non-coredump flags which `mm_struct` objects can possess. Examining them:

- `MMF_VM_MERGEABLE` – Used by *Kernel Samepage Merging (KSM)* logic to mark a VMA as mergeable. Out of scope for the book.
- `MMF_VM_HUGEPAGE` – Used by *Transparent Huge Pages (THP)* (more on this in the huge pages chapter) to indicate that folios in this process can be converted into huge pages. Checked by the `khugepaged` kernel thread on `khugepaged_fork()` (called on fork from `dup_mmap()`). If this flag is set then `__khugepaged_enter()` is invoked which marks the `mm_struct` as available for use. Checked again on exit in `khugepaged_exit()` which is called when an `mm_struct` is torn down in `__mmaput()`.
- `MMF_HAS_UPROBES` – Used to indicate that the `mm_struct` has *uprobes* present in its address space. Out of scope for the book.
- `MMF_RECALC_UPROBES` – Indicates that uprobes need to be recalculated for this `mm_struct`. Out of scope for the book.
- `MMF_OOM_SKIP` – Indicates that memory of the process belonging to the `mm_struct` should be ignored by the Out Of Memory (OOM) killer. Set in `exit_mmap()` when a process is identified to already be an OOM victim via `mm_is_oom_victim()` (which checks the `MMF_OOM_VICTIM` flag), in `oom_reap_task()` after reaping is complete to achieve the same ends and in `__oom_kill_process()` if the process is the global init process (checked by `is_global_init()`).
- `MMF_UNSTABLE` – Set by the OOM killer in `__oom_reap_task_mm()` to indicate that the address space being reaped is no longer stable, as indicated by `check_stable_address_space()` and checked by `do_anonymous_page()`, `finish_fault()`, huge page equivalents and memory migration code.
- `MMF_HUGE_ZERO_PAGE` – Indicates that a huge zero page has been obtained for the address space via `mm_get_huge_zero_page()`.
- `MMF_DISABLE_THP` – Disables Transparent Huge Pages (THP) altogether. Set by `prctl()` (specifying the `PR_SET_THP_DISABLE` mode), and can be queried

either via `prctl()` using `PR_GET_THP_DISABLE` or from observing the status in `proc/$pid/status`. Checked in `hugepage_vma_check()` (alongside checking the VMA flag `VM_NOHUGEPAGE`).

- `MMF_OOM_VICTIM` – Set to indicate that a process is a victim of the OOM killer in `mark_oom_victim()` and checked via `mm_is_oom_victim()` which determines whether to set `MMF_OOM_SKIP` in `exit_mmap()` as described above.
- `MMF_OOM_REAP_QUEUED` – Both tested and set in `queue_oom_reaper()` to mark that the process will be the subject of OOM repeating and to set up the timer to do just that.
- `MMF_MULTIPROCESS` – Indicates that the `mm_struct` is referenced by multiple processes. Set when a process is copied via `copy_process()` in `copy_oom_score_adj()` and used in `__set_oom_adj()` to ensure that the `mm_struct` gets pinned (i.e. `mmgrab()` applied to it to increment its `mm_count` reference count).
- `MMF_HAS_PINNED` – Set by the Get User Pages (GUP) logic in `mm_set_has_pinned_flag()` to indicate that `FOLL_PIN` has been set and thus the invocation of GUP is intended to pin folios in the process. Once set, this is maintained for the process lifetime. Checked in `page_needs_cow_for_dma()` to determine if folios might be pinned for DMA (which would have set this pinned flag) and in `pte_is_pinned()` which is used by `clear_soft_dirty()` to clear the soft dirty flag, but explicitly not when pinned folios could be present.

4.4 Virtual Memory Areas (VMAs)

Each contiguous range of virtual memory with similar characteristics (more specifically ranges which handle *page faults** in the same fashion) are represented by Virtual Memory Areas (VMAs).

These are ultimately the fundamental building block upon which user-land memory allocations are built – a process has memory mapped at an address if it has a VMA spanning the base page containing it.

Since we can ‘fault in’ memory on demand, we don’t actually need to have physical memory allocated or even page table mappings established for a memory range to be valid, rather we simply mark that it is valid in its containing VMA.

Examining a simplified struct `vm_area_struct` (retaining the key fields) as shown in Listing 4-11.

```

397 /*
398  * This struct describes a virtual memory area. There is one of these
399  * per VM-area/task. A VM area is any part of the process virtual memory
400  * space that has a special rule for the page-fault handlers (ie a shared

```

*. A page fault occurs when memory is either not virtually mapped or is not permitted to be accessed in the fashion the user is attempting to (e.g. writing to memory with page table flags indicating that it is read-only), we shall describe this in considerably more detail shortly.

```

401  * library, the executable area etc).
402  */
403  struct vm_area_struct {
404      /* The first cache line has the info for VMA tree walking. */
405
406      unsigned long vm_start;          /* Our start address within vm_mm. */
407      unsigned long vm_end;           /* The first byte after our end
address                                     within vm_mm. */
408
409
410      /* linked list of VM areas per task, sorted by address */
411      struct vm_area_struct *vm_next, *vm_prev;
412
413      struct rb_node vm_rb;
414
415      /*
416       * Largest free memory gap in bytes to the left of this VMA.
417       * Either between this VMA and vma->vm_prev, or between one of the
418       * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
419       * get_unmapped_area find a free area of the right size.
420       */
421      unsigned long rb_subtree_gap;
422
423      /* Second cache line starts here. */
424
425      struct mm_struct *vm_mm;         /* The address space we belong to. */
426
427      /*
428       * Access permissions of this VMA.
429       * See vmf_insert_mixed_prot() for discussion.
430       */
431      pgprot_t vm_page_prot;
432      unsigned long vm_flags;          /* Flags, see mm.h. */
433
434      /*
435       * For areas with an address space and backing store,
436       * linkage into the address_space->i_mmap interval tree.
437       *
438       * For private anonymous mappings, a pointer to a null terminated
string
439       * containing the name given to the vma, or NULL if unnamed.
440       */
441
442      union {
443          struct {
444              struct rb_node rb;
445              unsigned long rb_subtree_last;

```

```

446         } shared;
447     /*
448      * Serialized by mmap_sem. Never use directly because it is
449      * valid only when vm_file is NULL. Use anon_vma_name instead.
450      */
451     struct anon_vma_name *anon_name;
452 };
453
454 /*
455  * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
456  * list, after a COW of one of the file pages. A MAP_SHARED vma
457  * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
458  * or brk vma (with NULL file) can only be in an anon_vma list.
459  */
460 struct list_head anon_vma_chain; /* Serialized by mmap_lock &
461                                  * page_table_lock */
462 struct anon_vma *anon_vma;      /* Serialized by page_table_lock */
463
464 /* Function pointers to deal with this struct. */
465 const struct vm_operations_struct *vm_ops;
466
467 /* Information about our backing store: */
468 unsigned long vm_pgoff;          /* Offset (within vm_file) in
PAGE_SIZE
469                                  units */
470 struct file * vm_file;           /* File we map to (can be NULL). */
471 void * vm_private_data;          /* was vm_pte (shared mem) */
472 :
473 :
480 struct mempolicy *vm_policy;     /* NUMA policy for the VMA */
481 :
482 } __randomize_layout;

```

Listing 4-11: include/linux/mm_types.h: Simplified struct vm_area_struct

Examining each field:

- `vm_start` – The first address mapped by this region. Must be page-aligned.
- `vm_end` – The *exclusive* upper bound of memory mapped by this region, i.e. 1 byte beyond the memory it maps.
- `vm_next` – The next VMA in the process address space, sorted by address, or NULL if this is the last.
- `vm_prev` – The previous VMA in the process address space, sorted by address, or NULL if this is the first.
- `vm_rb` – Node object with linkage for this VMA within the binary red-black tree of VMAs contained within the process address space and rooted on struct `mm_struct->mm_rb`.

- `rb_subtree_gap` – The largest free gap in all of the nodes to the left of us in the binary tree, i.e., the largest difference between the `vm_end` and `vm_start` of adjacent VMAs at or below this one. This is used to more efficiently determine an appropriate free area in which to place a mapping in `get_unmapped_area()`. This is calculated via `vma_gap_callbacks_compute_max()` via the macro generator `RB_DECLARE_CALLBACKS_MAX()`. This is reference in `vma_gap_callbacks_propagate()`, generated by the same macro and invoked by `vma_gap_update()` which performs the subtree gap update. The field is used to determine an unmapped area in which to place a mapping in `unmapped_area()`.
- `vm_mm` – A pointer to the process address space's struct `mm_struct` to which this VMA belongs.
- `vm_page_prot` – The `pgprot_t` protection bits to be applied to PTEs (or higher order page tables if huge page) within the range defined by the VMA. This is applied in the various page fault handlers, for example in `do_anonymous_page()` where the field is referenced via `entry = mk_pte(page, vma->vm_page_prot)`. This field is initially determined from the architecture-specific `vm_get_page_prot()` function, which for x86-64 are determined by the `protection_map` array, as described in the virtual memory chapter in figure 3.13 in which the chosen bits are determined from the `vm_flags` field described below. Note that this can be further modified via `vma_set_page_prot()`, specifically when a non-anonymous mapping's underlying VMA requires that it is notified when a write first occurs, as determined by `vma_wants_writenotify()`.
- `vm_flags` – Flags describing the VMA. This is a broad topic see below in section 4.4.1 for a detailed discussion.
- `shared.rb` – If file-backed (i.e. referencing a struct `address_space` object), a node object used for placement into the interval tree hung off the address space's `i_mmap` field. Discussed in detail in section 4.5.
- `shared.rb_subtree_last` – Used in conjunction with the `rb` node described above as part of the address space interval tree placement. Again, see section 4.5 for a discussion of this.
- `anon_name` – This is held in union with the `shared` object described above, so this field is only present in the case of VMAs mapping anonymous memory. This contains a name describing the mapping, e.g. `heap`, `stack`, etc. Access to this field is serialised by the `mmap_lock` semaphore which must be held when this function is invoked. It should be obtained via the `anon_vma_name()` function which asserts that this lock is held via `mmap_assert_locked()` as described in section 4.3.4.3. This is represented by a dynamically allocated struct `anon_vma_name` object which wraps a reference count using the generic kernel reference counter type struct `kref`, as well as the string containing the name itself. It is allocated by `anon_vma_name_alloc()`, a reference taken on it via `anon_vma_name_get()` and released by `anon_vma_name_put()` (which, if the reference count

reaches zero, frees it via `anon_vma_name_free()`). Anonymous VMA names are only available if `CONFIG_ANON_VMA_NAME` is specified.

- `anon_vma` – Represents an object which links folios with VMAs. Since VMAs are ephemeral and multiple different VMAs can reference the same regions in memory, it is not practical to attempt to reference the VMAs in a folio or the folios in a VMA. Instead, an intermediate object, the `struct anon_vma`, is used to tie the two together. The use of this object is termed a memory range’s *reverse mapping* as described in Chapter 7. This field references this object if such a mapping exists for this VMA. Access to this field is serialised by the `struct mm_struct->page_table_lock` as described in section 4.3.4.1.
- `anon_vma_chain` – Forking means that an `anon_vma` object can become tied to multiple different processes. To tie them together, `struct anon_vma_chain` objects are hung off this field using the `same_vma` field. See section 4.3.4.1 for more.
- `vm_ops` – Contains operations defined in the `struct vm_operations_struct` which provide callbacks for a number of different operations, arguably the most notable of which is the page fault handler specified in its `fault` field (and huge page equivalent in its `huge_fault` field). This field is intimately tied into page fault behaviour, and we will therefore discuss it detail in the dedicated section 6.
- `vm_pgoff` – If file-backed, the offset into the underlying file specified in base pages. If anonymous, the ‘virtual page offset’ at which the memory was *originally* mapped (i.e. before any `mremap()`) which is the virtual address of its mapping specified in base pages. In both cases the page offset of an address within a VMA is determined by `linear_page_index()`. There are special cases where this might be different, such as kernel memory mapped into userland via `remap_pfn_range()`, see section 6.12 for more details on such special mappings.
- `vm_file` – If file-backed, this references the `struct file` object which describes the file which this memory maps. `NULL` if the VMA describes an anonymous mapping.
- `vm_private_data` – A field available for private use by drivers and ‘special’ mappings.
- `vm_policy` – A field used to keep track of the NUMA ‘memory policy’ that applies to this memory range as defined by an `struct mempolicy` object. This is superseded by the `vm_ops->get_policy()` callback if it is specified as looked up by the `__get_vma_policy()` function (via `get_vma_policy()`). See chapter 10 for more details on NUMA.

4.4.1 VMA flags

A VMA’s `struct vm_area_struct->vm_flags` field contains a bit map of flags which describe its properties and determine what happens on page fault. Each unique subset of flags defines a distinct VMA, even if two regions

with different flags are immediately adjacent to one another (ignoring `VM_SOFDITY` which is considered ephemeral). There are other characteristics which determine whether one VMA is equivalent to (or *mergeable* with) another, as determined by `is_mergeable_vma()`.

Examining each of the flags, all of which are declared in the `include/linux/mm.h` header. Note that where architecture-specific behaviour is defined, we describe x86-64 for brevity:

- `VM_NONE` – (*not a flag but rather the zero value*) – Indicates that no flags are specified at all. Will result in page table flags equal to `PAGE_NONE`. This is equivalent to having absolutely no flags set and is useful for the `protection_map` mappings between VMA flags and page table flags to designate a case where neither `VM_SHARED`, `VM_READ`, `VM_WRITE` nor `VM_EXEC` is set *when masked against those fields alone*.
- `VM_READ` – Set when the memory range a VMA describes has permission to be read from. When memory is mapped with the `PROT_READ` flag then this field is set.
- `VM_WRITE` – Set when the memory range a VMA describes has permission to be written to. When memory is mapped with the `PROT_WRITE` flag then this field is set.
- `VM_EXEC` – Set when the memory range a VMA describes has permission to be executed. When memory is mapped with the `PROT_EXEC` flag then this field is set.
- `VM_SHARED` – Set when the memory range a VMA describes is memory will be shared between processes. With this set, Copy-on-Write semantics will not apply, otherwise they always will with the memory mapped to the zero page*. When memory is mapped with the `MAP_SHARED` flag then this field is set.
- `VM_MAYREAD` – This indicates that the memory range a VMA describes is *capable* of being read from, even if `VM_READ` is not currently set, implying that this might be changed by something like `mprotect()`.
- `VM_MAYWRITE` – This indicates that the memory range a VMA describes is *capable* of being written to, even if `VM_WRITE` is not currently set, implying that this might be changed by something like `mprotect()`. If this is set but `VM_SHARED` is not, then this mapping will be a Copy-on-Write mapping, as the fact that it *can* be written to but is not shared implies that it later becoming will be as a Copy-on-Write mapping. This condition is checked via `is_cow_mapping()`.
- `VM_MAYEXEC` – This indicates that the memory range a VMA describes is *capable* of code being executed within it, even if `VM_EXEC` is not currently set, implying that this might be changed by something like `mprotect()`.

*. The *zero page* is an area of memory which is kept zeroed. By pointing Copy-on-Write mappings to this page initially, reads will be zeroed and writes will actually result a dedicated page of memory being allocated for the mapping.

- **VM_MAYSHARE** – A slightly odd flag, as VMAs cannot transition between shared and unshared in the same way they can with the protection flags (read, write and execute). It appears to only be meaningful for architectures without MMU where it seems necessary to associate the capability of being shared with a specific VMA. For modern architectures this is interchangeable with **VM_SHARED** as it can and is only set when this flag is also set.
- **VM_GROWSDOWN** – Indicates that the VMA describes this process's stack which grows downwards. For architectures which have upward growing stacks or both, the **VM_GROWSUP** flag is also defined.
- **VM_PFNMAP** – Indicates that the memory range a VMA describes have no struct folios to describe them. This is typically used for mapping device memory into userland, often via `remap_pfn_range()`.
- **VM_LOCKED** – Indicates that the memory described by the VMA has been locked via `mlock()`. It is set before locking memory within the VMA and used both by the `mlock` logic itself to determine whether it is locking or unlocking (in `mlock_pte_range()`) as well as kernel logic which needs to know this, most notably reclaim where a folio may have become subject to reclaim before it was able to be taken off LRU lists (see section 11.2 for more on this). Memory locked in this way is also faulted in.
- **VM_LOCKONFAULT** – Defer `mlock()`ing folios until page fault within the memory range described by this VMA. This is set by the user either passing `MLOCK_ONFAULT` to `mlock2()` or `MCL_ONFAULT` to `mlockall()`. If a `mlock()` operation is performed *without* this flag set, then the range specified is faulted into memory. If the flag is set then the memory is not faulted in, as enforced by `populate_vma_page_range()`.
- **VM_IO** – Indicates that the VMA range maps memory which is backed by an I/O interface rather than physical memory. This indicates that the memory should not be accessed by the kernel as doing so can cause unexpected side effects. Typically this is backed by memory-mapped I/O device registers but doesn't necessarily have to be. This is often set by `remap_pfn_range()` (which in turn calls `remap_pfn_range_notrack()`) which sets this flag, **VM_PFNMAP**, **VM_DONTEXPAND** and **VM_DONTDUMP**. Typically checks within the kernel avoiding accessing this memory check either this flag or **VM_PFNMAP** and equally avoid both.
- **VM_SEQ_READ** – Indicates that the memory (this is really only meaningful if a file is mapped) by this VMA will be consumed sequentially – increase readahead pages (via `do_sync_mmap_readahead()`) and be more aggressive in reclaim after they are read (checked for in `folio_referenced_one()`). This is set via `madvise()` using the **MADV_SEQUENTIAL** flag.
- **VM_RAND_READ** – Indicates that the memory (this is really only meaningful if a file is mapped) by this VMA will be consumed non-sequentially, i.e. with random access – readahead is considerably less useful so avoid it (as enforced in `do_sync_mmap_readahead()` and `do_async_mmap_readahead()`). This is set via `madvise()` using the **MADV_RANDOM** flag.

- `VM_DONTCOPY` – Indicates that this VMA should *not* be copied when a process forks. This is checked in `dup_mmap()` and can be set by `madvise()` using `MADV_DONTFORK` and cleared by `MADV_DOFORK`, as well as being set by the kernel where forking certain memory ranges would be inappropriate.
- `VM_DONTEXPAND` – Indicates that the VMA must not be expanded by `mremap()`. This is useful for cases where it simply would not make any sense to expand such mappings, such as device-mapped memory and indeed is set by `remap_pfn_range()`.
- `VM_DONTDUMP` – Indicates that the memory range described by the VMA should not appear in core dumps. This can be set by `madvise()` using `MADV_DONTDUMP` (and cleared by `MADV_DODUMP`) but is also set by the kernel for memory which would not be appropriate in a core dump such as device or secret memory (and again this is set by `remap_pfn_range()`).
- `VM_WIPEONFORK` – Indicates that the memory range described by the VMA should be cleared on fork rather than placed in a Copy-on-Write relationship with its child process. This can be set by `madvise()` using `MADV_WIPEONFORK` (and cleared by `MADV_KEEPONFORK`), but is also used by the kernel to prevent forking of memory where doing so would be inappropriate. This flag is only valid for private anonymous memory.
- `VM_NORESERVE` – Specifies that no swap space should be reserved for the memory range described by this VMA, though only meaningful if the overcommit mode (see section 4.1) is *not* set to `OVERCOMMIT_NEVER`. This is specified on `mmap()` by specifying the `MAP_NORESERVE` flag. See the swap chapter for more on how swap functions in the kernel.
- `VM_SOFTDIRTY` – An ephemeral flag which indicates that memory within the VMA has been written to since last cleared using the *soft-dirty* functionality within the kernel. Unlike other VMA flags, this flag has no bearing as to whether one VMA is independent from another – if two VMAs are identical other than this flag, then they can be merged (as enforced by `is_mergeable_vma()`).
- `VM_ACCOUNT` – Indicates that the memory range described by this VMA is accounted for the purposes of overcommit (see section 4.1 for more on this), set in `vm_acct_memory()`. Whether memory is accountable or not is determined by `accountable_mapping()` – if the VMA has the `VM_WRITE` flag but has neither `VM_NORESERVE` nor `VM_SHARED` set, then it is accountable. Accountable memory updates the `Committed_AS` statistics observable in `/proc/meminfo`.
- `VM_MIXEDMAP` – The memory range described by this VMA contains a mix of both `struct folio`-described memory as well as `VM_PFNMAP`-style ‘pure PFNs’ mappings. This is used to enable Copy-on-Write of ‘special’ mappings the discussion of which is out of scope here.
- `VM_HUGEPAGE` – Indicates that the memory range described by this VMA has been marked as `MADV_HUGEPAGE` by `madvise()` in order to mark the range as suitable for coalescing into *Transparent Huge Pages (THP)*. This is only meaningful if the `/sys/kernel/mm/transparent_hugepages/enabled` sysfs

tunable is set to `madvise`. See the huge page chapter for more details on THP.

- `VM_NOHUGEPAGE` – Similar to `VM_HUGEPAGE`, this indicates that the memory range described by this VMA has been marked as `MADV_NOHUGEPAGE` by `madvise()`, indicating that this memory is *not* suited to being coalesced into a huge page by the THP functionality. See the huge page chapter for more details on THP.
- `VM_HUGETLB` – Indicates that the memory range described by this VM comprises the backing for a *hugetlb* huge page region of memory. see the huge page chapter for more details on this.
- `VM_MERGEABLE` – Used by *Kernel Samepage Merging (KSM)* logic to mark a VMA as mergeable. Out of scope for this book.
- `VM_UFFD_MISSING` – Used for tracking missing pages within `userfaultfd` memory ranges. This is out of scope for the book.
- `VM_UFFD_WP` – Used for tracking write-protected pages within `userfaultfd` memory ranges. This is out of scope for the book.
- `VM_SYNC` – Used by the *Direct Access for files (DAX)* logic to indicate synchronous page faults should occur. Discussion of DAX is out of scope for the book.

4.4.2 Allocation and freeing

VMA objects are allocated via `vm_area_alloc()` which simply allocates from a slab cache via `kmem_cache_alloc()` (more on this in the slab chapter) as shown in Listing 4-12.

```

455 struct vm_area_struct *vm_area_alloc(struct mm_struct *mm)
456 {
457     struct vm_area_struct *vma;
458
459     vma = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
460     if (vma)
461         vma_init(vma, mm);
462     return vma;
463 }
```

Listing 4-12: `kernel/fork.c: vm_area_alloc()`

This calls `vma_init()` to perform initialisation as shown in Listing 4-13.

```

614 static inline void vma_init(struct vm_area_struct *vma, struct mm_struct *mm)
615 {
616     static const struct vm_operations_struct dummy_vm_ops = {};
617
618     memset(vma, 0, sizeof(*vma));
619     vma->vm_mm = mm;
620     vma->vm_ops = &dummy_vm_ops;
```

```

621     INIT_LIST_HEAD(&vma->anon_vma_chain);
622 }

```

Listing 4-13: include/linux/mm.h: `vma_init()`

This zeroes the object, assigns the process's struct `mm_struct` object to which the VMA belongs, assigns some zeroed operations `dummy_vm_ops` and initialises the `anon_vma_chain` list.

The means by which the kernel decides where to place allocated VMAs and what triggers this forms the subject of memory mapping, discussed in Chapter 5.

Often, the kernel will attempt to manipulate existing VMAs to accommodate memory mappings, this is a broad enough subject that it has its own dedicated section to discuss it – section 5.1.

VMAs are freed via `vm_area_free()` as shown in Listing 5-45.

```

484 void vm_area_free(struct vm_area_struct *vma)
485 {
486     free_anon_vma_name(vma);
487     kmem_cache_free(vm_area_cachep, vma);
488 }

```

Listing 4-14: kernel/fork.c: `vm_area_free()`

This ultimately frees the VMA memory to the cache via the slab function `kmem_cache_free()`, freeing any allocated anonymous memory mapping name via `free_anon_vma_name()`.

4.4.3 VMA layout

VMAs are tied together in two different ways – connected to their owning struct `mm_struct` in a doubly-linked list (though not via struct `list_head`), headed by `mm_struct`'s `mmap` field, and joined by each VMA's `vm_prev` and `vm_next` field. Importantly, this list is maintained in virtual address order.

This list permits linear traversal of VMAs and the ability to determine each VMA's predecessor and successor, however it is not efficient for looking up a specific address. Efficient means of doing so is achieved via a red/black binary search tree rooted in the `mm_struct`'s `mm_rb` field, with each VMA hosting a node in their `vm_rb` field.

The tree is keyed on the `vm_end` field of VMAs, which are guaranteed to be non-overlapping (we check this when linking them). When searching, we traverse the tree looking for a node which possesses a `vm_end` value greater than our start address, then in each case test whether the `vm_start` is less than our start address.

Let's consider an example VMA tree with VMAs between `0x0 - 0x500`, `0x500 - 0x1000`, `0x2000 - 0x3000`, `3250 - 3750`, `0x4000 - 0x4500` and `0x6000 - 0x7000` in Figure 4-15.

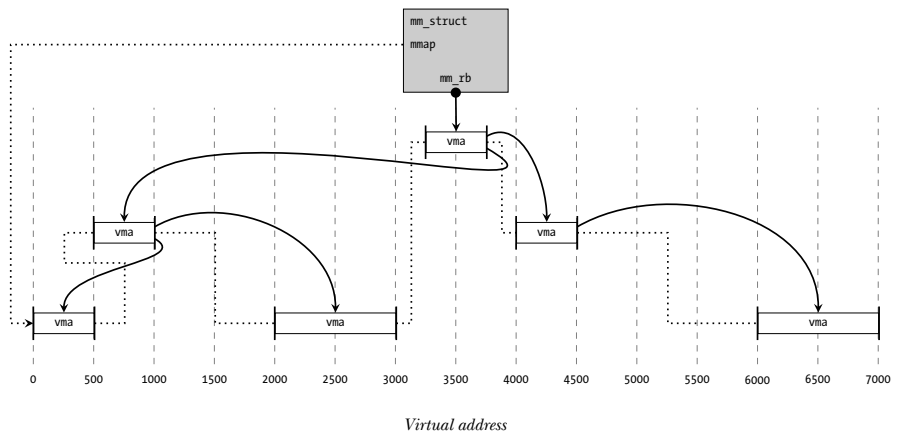


Figure 4-15: Example VMA layout

Note that here each VMA is shown proportionate to the range they represent and are arranged such that virtual address increases from left to right.

The VMAs are linked in a binary tree, keyed on `struct vm_area_struct->vm_end` while also being connected to one another in a linked list via `vm_prev` and `vm_next`.

4.4.4 VMA insertion/removal

We examine the red/black binary tree node type in the kernel struct `rb_node` in Listing 4-15.

```

5 struct rb_node {
6     unsigned long __rb_parent_color;
7     struct rb_node *rb_right;
8     struct rb_node *rb_left;
9 } __attribute__((aligned(sizeof(long))));

```

Listing 4-15: `include/linux/rbtree_types.h`: `struct rb_node`

The `__rb_parent_color` field is used as part of the red/black tree algorithm (out of scope for this book, but any good algorithm book will cover it), with `rb_left` and `rb_right` pointing to the subtrees containing VMAs with `vm_end` less than it to the left, and those with greater `vm_end` to the right. The root object struct `rb_root` simply wraps a `rb_node`.

In order to insert new VMAs into a process address space, we need to obtain a pointer to the parent node (this is in order to set the `__rb_parent_color` field) and a pointer to either the `rb_left` or `rb_right` field to know where to put the new node.

In order to maintain the linked list we also need to know an adjacent VMA object, which happens to be the previous one. From this we can find the next VMA before we were inserted, update it and point ourselves to the prior and next VMA.

All of these are provided by the `find_vma_links()` function as shown in Listing 4-16.

```

488 static int find_vma_links(struct mm_struct *mm, unsigned long addr,
489                          unsigned long end, struct vm_area_struct **pprev,
490                          struct rb_node ***rb_link, struct rb_node **rb_parent)
491 {
492     struct rb_node **__rb_link, *__rb_parent, *rb_prev;
493
494     mmap_assert_locked(mm);
495     __rb_link = &mm->mm_rb.rb_node;
496     rb_prev = __rb_parent = NULL;
497
498     while (*__rb_link) {
499         struct vm_area_struct *vma_tmp;
500
501         __rb_parent = *__rb_link;
502         vma_tmp = rb_entry(__rb_parent, struct vm_area_struct, vm_rb);
503
504         if (vma_tmp->vm_end > addr) {
505             /* Fail if an existing vma overlaps the area */
506             if (vma_tmp->vm_start < end)
507                 return -ENOMEM;
508             __rb_link = &__rb_parent->rb_left;
509         } else {
510             rb_prev = __rb_parent;
511             __rb_link = &__rb_parent->rb_right;
512         }
513     }
514
515     *pprev = NULL;
516     if (rb_prev)
517         *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
518     *rb_link = __rb_link;
519     *rb_parent = __rb_parent;
520     return 0;
521 }

```

Listing 4-16: `mm/mmap.c: find_vma_links()`

This function accepts the owning `mm_struct`, the *start* address of the VMA we want to link, `addr`, the exclusive end bound of the VMA we want to link, `end`, and we output:

- `pprev` – A pointer to the `struct vm_area_struct` (VMA) immediately prior to this one, or `NULL` if the inserted VMA would have the lowest virtual address in the process address space.
- `rb_link` – A pointer to either the `rb_left` or `rb_right` field in the parent node in which a pointer to the newly inserted node must be placed.

- `rb_parent` – A pointer to the parent node, or NULL if this would be the first VMA in this address space.

The function works by traversing through each node, determining whether the containing VMA has a `vm_end` bound greater than the start address of the VMA to be inserted, `addr` – if so this indicates that we should traverse *left*. In this case, the VMA could overlap ours – we check this by seeing if it starts after us, if not we raise an error.

Otherwise we traverse *right* and in either case we set `__rb_link` to point to the node to which we traverse. Since we predicate the loop on this being non-NULL, the final result will be that this points to NULL and thus is where we should insert our VMA.

An important nuance here is how we track the *previous* VMA – this is only updated when we traverse *right*. This is correct, as until we traverse right, we have no predecessor, and when we do so the parent must be the most immediate predecessor as the parent's `vm_end` must be less than ours and its left subtree must be less than itself.

The users of `find_vma_links()` are a little eclectic:

- `__insert_vm_struct()` – This is used as a helper by `__vma_adjust()` as part of VMA splitting (covered in detail in section 5.1).
- `insert_vm_struct()` – This is a helper function used to wrap insertion of VMAs, used by some initialisation functions and the out of scope ‘special’ mapping handling function `__install_special_mapping()`.
- `copy_vma()` – This is used by the `mremap()` logic when VMAs need to be copied around and thus inserted elsewhere.
- `munmap_vma_range()` – Ironically, this is one of the most common means by which `find_vma_links()` is invoked for VMA traversal – it is used by `mmap_region()` and `do_brk_flags()` to unmap any existing mappings while also obtaining the fields required for inserting the new VMA at the same time.

Regardless of the method used to obtain the values returned by `find_vma_links()`, a VMA is ultimately ‘linked’ into a process address space via `__vma_link()` as shown in Listing 5-51.

```

636 static void
637 __vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
638           struct vm_area_struct *prev, struct rb_node **rb_link,
639           struct rb_node *rb_parent)
640 {
641     __vma_link_list(mm, vma, prev);
642     __vma_link_rb(mm, vma, rb_link, rb_parent);
643 }
```

Listing 4-17: `mm/mmap.c: __vma_link()`

This links the VMA being inserted into the VMA linked list and red/black tree. Note the parameters matching those provided by `find_vma_links()` – `prev`, `rb_link` and `rb_parent`.

We first insert into the linked list via `__vma_link_list()`, examining this as shown in Listing 4-18.

```

275 void __vma_link_list(struct mm_struct *mm, struct vm_area_struct *vma,
276                     struct vm_area_struct *prev)
277 {
278     struct vm_area_struct *next;
279
280     vma->vm_prev = prev;
281     if (prev) {
282         next = prev->vm_next;
283         prev->vm_next = vma;
284     } else {
285         next = mm->mmap;
286         mm->mmap = vma;
287     }
288     vma->vm_next = next;
289     if (next)
290         next->vm_prev = vma;
291 }
```

Listing 4-18: `mm/util.c: __vma_link_list()`

This handles the case where this VMA possesses the earliest address in the process address space (indicated by `prev` being `NULL`), correctly updating the `mm_struct->mmap` field accordingly.

Both the inserted VMA, its predecessor (or the `mm_struct` object if first) and successor (if it exists) are updated.

The node is inserted into the red/black tree via `__vma_link_rb()` as shown in Listing 4-19.

```

595 void __vma_link_rb(struct mm_struct *mm, struct vm_area_struct *vma,
596                   struct rb_node **rb_link, struct rb_node *rb_parent)
597 {
598     /* Update tracking information for the gap following the new vma. */
599     if (vma->vm_next)
600         vma_gap_update(vma->vm_next);
601     else
602         mm->highest_vm_end = vm_end_gap(vma);
603
604     /*
605      * vma->vm_prev wasn't known when we followed the rbtree to find the
606      * correct insertion point for that vma. As a result, we could not
607      * update the vma vm_rb parents rb_subtree_gap values on the way down.
608      * So, we first insert the vma with a zero rb_subtree_gap value
609      * (to be consistent with what we did on the way down), and then

```

```

610      * immediately update the gap to the correct value. Finally we
611      * rebalance the rbtree after all augmented values have been set.
612      */
613      rb_link_node(&vma->vm_rb, rb_parent, rb_link);
614      vma->rb_subtree_gap = 0;
615      vma_gap_update(vma);
616      vma_rb_insert(vma, &mm->mm_rb);
617 }

```

Listing 4-19: mm/mmap.c: `__vma_link_rb()`

This performs an update of the VMA's subtree gap and propagates changes to other nodes (stored in their `rb_subtree_gap` field) via `vma_gap_update()`. This field is used to determine the largest gap between VMA regions less at or below this one and used by the memory mapping logic when determining where to place a new VMA. We discuss how this initial mapping is done in detail in Chapter 5.

We actually insert the node in the tree via `rb_link_node()` which is the function for doing this in the kernel's generic red/black tree implementation.

Finally we invoke `vma_rb_insert()` which kicks off `rb_insert_augmented()` to balance the tree and also passes it the `vma_gap_callbacks` callbacks object to maintain the subtree gap calculations as it does so.

Coming back to `__vma_link()` – this function will either be invoked by `__insert_vm_struct()` or `vma_link()`. Let's examine the former, which is somewhat simplistic as shown in Listing 5-50.

```

666 /*
667  * Helper for vma_adjust() in the split_vma insert case: insert a vma into the
668  * mm's list and rbtree. It has already been inserted into the interval tree.
669  */
670 static void __insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *
        vma)
671 {
672     struct vm_area_struct *prev;
673     struct rb_node **rb_link, *rb_parent;
674
675     if (find_vma_links(mm, vma->vm_start, vma->vm_end,
676                       &prev, &rb_link, &rb_parent))
677         BUG();
678     __vma_link(mm, vma, prev, rb_link, rb_parent);
679     mm->map_count++;
680 }

```

Listing 4-20: mm/mmap.c: `__insert_vm_struct()`

This simply invokes `find_vma_links()`, then links what it finds via `__vma_link()` before incrementing the `mm_struct->map_count` field to indicate that a new mapping exists there.

Now examining `vma_link()` as shown in Listing 4-21.

```

645 static void vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
646                     struct vm_area_struct *prev, struct rb_node **rb_link,
647                     struct rb_node *rb_parent)
648 {
649     struct address_space *mapping = NULL;
650
651     if (vma->vm_file) {
652         mapping = vma->vm_file->f_mapping;
653         i_mmap_lock_write(mapping);
654     }
655
656     __vma_link(mm, vma, prev, rb_link, rb_parent);
657     __vma_link_file(vma);
658
659     if (mapping)
660         i_mmap_unlock_write(mapping);
661
662     mm->map_count++;
663     validate_mm(mm);
664 }

```

Listing 4-21: mm/mmap.c: vma_link()

This similarly invokes `__vma_link()` but additionally includes logic for handling file-backed VMAs, invoking `__vma_link_file()` and acquiring the appropriate lock over the struct `address_space->i_mmap` interval tree containing VMAs linked to it. See the chapter on the page cache for more on this.

Finally, `validate_mm()` is called, which only does something if `CONFIG_DEBUG_VM_RB` is set, validating the subtree gaps set by the VMA logic.

Removal of VMAs is a little more involved, see Chapter 5 for an in-depth treatment of unmapping memory, however we will examine the function that is ultimately invoked, `__vma_unlink()` as shown in Listing 5-41.

```

682 static __always_inline void __vma_unlink(struct mm_struct *mm,
683                                         struct vm_area_struct *vma,
684                                         struct vm_area_struct *ignore)
685 {
686     vma_rb_erase_ignore(vma, &mm->mm_rb, ignore);
687     __vma_unlink_list(mm, vma);
688     /* Kill the cache */
689     vmacache_invalidate(mm);
690 }

```

Listing 4-22: mm/mmap.c: __vma_unlink()

This starts by erasing the red/black tree node from the VMA tree via `vma_rb_erase_ignore()` – in turn this invokes `__vma_rb_erase()` which invokes the generic red/black tree erase function `rb_erase_augmented()`.

Next, it releases the VMA from the linked list via `__vma_unlink_list()` as shown in Listing 5-42.

```

293 void __vma_unlink_list(struct mm_struct *mm, struct vm_area_struct *vma)
294 {
295     struct vm_area_struct *prev, *next;
296
297     next = vma->vm_next;
298     prev = vma->vm_prev;
299     if (prev)
300         prev->vm_next = next;
301     else
302         mm->mmap = next;
303     if (next)
304         next->vm_prev = prev;
305 }
```

Listing 4-23: mm/util.c: `__vma_unlink_list()`

This performs the inverse of what `__vma_link_list()` does, taking a copy of what comes before and after the VMA before linking them together excluding the VMA being unlinked, while also taking into account that this might be the first VMA in the process address space and handling the update of `mm_struct->mmap` appropriately.

Finally, the VMA cache is invalidated via `vmacache_invalidate()` which simply increments the `mm_struct->vmacache_seqnum` sequence number to do so. See below for a discussion of the VMA cache as a whole.

4.4.5 VMA traversal

Traversing the linked list is straightforward – the `struct vm_area_struct` fields `vm_prev` and `vm_next` provide easy means of walking these, and each `mm_struct` specifies the VMA with the earliest virtual address in its `mmap` field.

Finding VMAs by address however is more involved – we have, after all, established an entire red/black tree structure in order to do so efficiently. The principle means of doing so is via `find_vma()` as shown in Listing 4-24.

```

2252 /* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
2253 struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)
2254 {
2255     struct rb_node *rb_node;
2256     struct vm_area_struct *vma;
2257
2258     mmap_assert_locked(mm);
2259     /* Check the cache first. */
2260     vma = vmacache_find(mm, addr);
2261     if (likely(vma))
2262         return vma;
2263 }
```

```

2264         rb_node = mm->mm_rb.rb_node;
2265
2266         while (rb_node) {
2267             struct vm_area_struct *tmp;
2268
2269             tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
2270
2271             if (tmp->vm_end > addr) {
2272                 vma = tmp;
2273                 if (tmp->vm_start <= addr)
2274                     break;
2275                 rb_node = rb_node->rb_left;
2276             } else
2277                 rb_node = rb_node->rb_right;
2278         }
2279
2280         if (vma)
2281             vmacache_update(addr, vma);
2282         return vma;
2283     }

```

Listing 4-24: mm/mmap.c: *find_vma()*

As discussed above, VMAs are maintained in a tree keyed on their exclusive upper bound, `vm_end`, and this function simply returns the VMA with the earliest address (i.e. *first*) which satisfies the condition that the input address is less than `vm_end`, or if it cannot find such a VMA, it returns `NULL`.

This necessitates an additional check being required if you are searching for an address range, i.e. checking that the start of your range is greater than the returned VMA's `vm_start`.

Returning a VMA this way allows for maximum flexibility – we can find the *closest* VMA greater than the specified address and via its `vm_prev` field determine the closest VMA less than it if we need to do so.

This function requires that a read lock is held on the `mm_struct->mmap_lock` semaphore.

The first stage is to checking the VMA cache to see whether it contains the VMA object for the struct `vm_area_struct` (VMA) at the specified address via `vmacache_find()`. If so, then we simply return this VMA. We will return to VMA caches shortly.

If no VMA cache entry exists, we walk the red/black tree. This simply traverses the tree from `mm_struct->mm_rb`, examining each node – if the node under examination ends after our target address, we examine lower addressed VMAs, otherwise we examine higher addressed ones.

When examining nodes which have an `vm_end` greater than our address, these *could* actually contain the target address, so we do the additional check to see whether `vm_start` is less than or equal to it – if so we can exit early.

When we are done, we update the VMA cache via `vmacache_update()`.

4.4.5.1 VMA cache

The VMA cache is simply a hash between addresses and VMAs stored in each thread's `struct task_struct->vmacache` field.

It is kept in sync with updated VMA mappings using a 'sequence number' – This is a simple integer which is maintained in `mm_struct->vmacache_seqnum`, which is incremented every time VMA mappings are updated. If a VMA cache's sequence number does match this, then it is invalidated and not used.

The key data structure used in the VMA cache is `struct vmacache` as shown in Listing 4-25.

```

34 struct vmacache {
35     u64 seqnum;
36     struct vm_area_struct *vmas[VMACACHE_SIZE];
37 };

```

Listing 4-25: include/linux/mm_types_task.h: struct vmacache

This maintains a very simple hash of VMA objects and the aforementioned sequence number. The number of entries in this hash is defined by `VMACACHE_SIZE` which is in turn determined by `VMACACHE_BITS` and hardcoded to 2 bits, i.e. 4 entries. `VMACACHE_MASK` specifies the valid hash table array entries as a bitmask.

The hash used in the lookup is determined by `VMACACHE_HASH()` as shown in Listing 4-26.

```

19 #define VMACACHE_HASH(addr) ((addr >> VMACACHE_SHIFT) & VMACACHE_MASK)

```

Listing 4-26: mm/vmacache.c: VMACACHE_HASH()

This simply shifts by the `VMACACHE_SHIFT` value which is set to `PMD_SHIFT`, which on x86-64 is 21. This means that address in 2 megabyte aligned blocks will all resolve to the same array entry. This has been heuristically determined to still be a useful granularity.

The lookup of VMA cache entries is performed via `vmacache_find()` as shown in Listing 4-27.

```

61 struct vm_area_struct *vmacache_find(struct mm_struct *mm, unsigned long addr)
62 {
63     int idx = VMACACHE_HASH(addr);
64     int i;
65
66     count_vm_vmacache_event(VMACACHE_FIND_CALLS);
67
68     if (!vmacache_valid(mm))
69         return NULL;
70
71     for (i = 0; i < VMACACHE_SIZE; i++) {
72         struct vm_area_struct *vma = current->vmacache.vmas[idx];
73

```

```

74         if (vma) {
75 #ifdef CONFIG_DEBUG_VM_VMACACHE
76             if (WARN_ON_ONCE(vma->vm_mm != mm))
77                 break;
78 #endif
79             if (vma->vm_start <= addr && vma->vm_end > addr) {
80                 count_vm_vmacache_event(VMACACHE_FIND_HITS);
81                 return vma;
82             }
83         }
84         if (++idx == VMACACHE_SIZE)
85             idx = 0;
86     }
87
88     return NULL;
89 }

```

Listing 4-27: mm/vmacache.c: *vmacache_find()*

We first determine whether the cache is valid via `vmacache_valid()`, after which we iterate through *all* entries, using the hash value as a starting point only. If we find a VMA that *contains* the address (we could not reasonably cache VMAs that come after the target address) we return it otherwise we return `NULL`.

We examine `vmacache_valid()` in Listing 4-28.

```

41 static bool vmacache_valid(struct mm_struct *mm)
42 {
43     struct task_struct *curr;
44
45     if (!vmacache_valid_mm(mm))
46         return false;
47
48     curr = current;
49     if (mm->vmacache_seqnum != curr->vmacache.seqnum) {
50         /*
51          * First attempt will always be invalid, initialize
52          * the new cache for this task here.
53          */
54         curr->vmacache.seqnum = mm->vmacache_seqnum;
55         vmacache_flush(curr);
56         return false;
57     }
58     return true;
59 }

```

Listing 4-28: mm/vmacache.c: *vmacache_valid()*

This first checks if the VMA belongs to the current task's `mm_struct` and that it is not a kernel thread via `vmacache_valid_mm()`, before checking to en-

sure that the sequence numbers of this cache and the `mm_struct` are equal to one another. If not, the sequence number is reset, and the cache is flushed via `vmacache_flush()` which simply zeroes the `vmas` array altogether.

Finally, updates made to the VMA cache are performed via `vmacache_update()` as shown in Listing 4-29.

```

35 void vmacache_update(unsigned long addr, struct vm_area_struct *newvma)
36 {
37     if (vmacache_valid_mm(newvma->vm_mm))
38         current->vmacache.vmas[VMACACHE_HASH(addr)] = newvma;
39 }
```

Listing 4-29: mm/vmacache.c: vmacache_update()

4.5 An introduction to the page cache

Within the kernel, all files, block I/O buffers and other related ‘cacheable’ objects are kept within a key cache in memory – the *page cache*. We will briefly examine the fundamentals of this topic and how they relate to process memory here, but for more detail on this topic, refer to the dedicated page cache chapter.

Typically all file are mediated by this cache (the exceptions being cases where either direct I/O or direct access (DAX) are used, both topics outside the scope of the book), with both memory-mapped and stream-read operations reading from this cache (see `filemap_read()` for the latter).

Each of the objects present in the page cache is described by a `struct address_space` object, pointed to by the underlying folios in the mapping field as shown in Listing 4-30.

```

402 /**
403  * struct address_space - Contents of a cacheable, mappable object.
404  * @host: Owner, either the inode or the block_device.
405  * @i_pages: Cached pages.
406  * @invalidate_lock: Guards coherency between page cache contents and
407  *   file offset->disk block mappings in the filesystem during invalidates.
408  *   It is also used to block modification of page cache contents through
409  *   memory mappings.
410  * @gfp_mask: Memory allocation flags to use for allocating pages.
411  * @i_mmap_writable: Number of VM_SHARED mappings.
412  * @nr_thps: Number of THPs in the pagecache (non-shmem only).
413  * @i_mmap: Tree of private and shared mappings.
414  * @i_mmap_rwsem: Protects @i_mmap and @i_mmap_writable.
415  * @nrpages: Number of page entries, protected by the i_pages lock.
416  * @writeback_index: Writeback starts here.
417  * @a_ops: Methods.
418  * @flags: Error bits and flags (AS_*).
419  * @wb_err: The most recent error which has occurred.
420  * @private_lock: For use by the owner of the address_space.
```

```

421  * @private_list: For use by the owner of the address_space.
422  * @private_data: For use by the owner of the address_space.
423  */
424  struct address_space {
425      struct inode          *host;
426      struct xarray         i_pages;
427      struct rw_semaphore   invalidate_lock;
428      gfp_t                 gfp_mask;
429      atomic_t              i_mmap_writable;
430      :
431
434      struct rb_root_cached i_mmap;
435      struct rw_semaphore   i_mmap_rwsem;
436      unsigned long         nrpages;
437      pgoff_t               writeback_index;
438      const struct address_space_operations *a_ops;
439      unsigned long         flags;
440      errseq_t              wb_err;
441      spinlock_t            private_lock;
442      struct list_head       private_list;
443      void                  *private_data;
444  } __attribute__((aligned(sizeof(long)))) __randomize_layout;

```

Listing 4-30: include/linux/fs.h: struct address_space

Examining each field:

- **host** – If this object refers to a file, this field references the file’s struct inode object*. This field can be NULL for non-file objects present within the page cache.
- **i_pages** – An *eXtensible array*[†] of pointers to struct page objects for page cache-mapped folios. This is inextricably linked to the struct folio->index field which specifies the page offset of the folio within the page cache mapping (which it references via its mapping field).
- **invalidate_lock** – A read/write semaphore used to synchronise modifications to the mapping such as reading into the page cache (e.g. via the function read_cache_page_gfp()), adding folios to it (e.g. via the filemap_create_folio() function), or on truncation (acquired by a filesystem using filemap_invalidate_lock()).
- **gfp_mask** – The GFP mask (see section 2.6 in the physical memory chapter for more details on these) applied to folio allocations within this

*. An inode is a data structure which contains a file system object’s metadata such as file size, access times, and other related information. It is independent of a file’s name or location within the filesystem.

†. The *eXtensible array* is an efficient data structure for storing a large set of possibly-sparse pointers with the ability to efficiently traverse to next and previous elements. The kernel documentation on this is extensive.

page cache entry. Typically accessed via `mapping_gfp_mask()`. Notably referenced by `filemap_create_folio()` to allocate page cache folios.

- `i_mmap_writable` – An atomic count used to determine whether the object is referenced by at least one writable memory mapping (i.e. a shared mapping). If only copy-on-write mappings have been performed (or it is not memory-mapped) this will be set to 0, for the more usual case of an `MAP_SHARED` file mapping (e.g. under a VMA with the `VM_SHARED` flag set) this is positive, set by `mapping_map_writable()` in `mmap_region()`. When a shared mapping is removed this is decremented by `mapping_unmap_writable()`.

This is implemented as a counter as it can be referenced by multiple VMAs and thus some mappings may be private and others not, with the overall purpose of this being the need to determine whether the underlying folios can be arbitrarily written to by userland (checked by `mapping_writably_mapped()` which simply determines if this value is greater than zero).

This value can be negative as it is possible to deny writable access via `mapping_deny_writable()` (which fails if the value is not already less than or equal to zero). This is used by the `memfd` functionality when sealing/unsealing via the `fcntl()` specifying the `F_SEAL_WRITE` option. This denial is reversed via `mapping_allow_writable()`.

- `i_mmap` – The root of a red/black interval tree of mappings to this cacheable object. This tree consists of `struct vm_area_struct` nodes (i.e. VMAs). It is used most notably by VMA interval tree operations via `__vma_link_file()`. Anonymous mappings equivalently hang their VMAs from a `struct anon_vma` object. See section 5.1 for more on this.
- `i_mmap_rwsem` – A read/write semaphore used to protect the `i_mmap` field. Manipulated by the helper functions `i_mmap_lock_write()`, `i_mmap_trylock_write()`, `i_mmap_unlock_write()`, `i_mmap_trylock_read()`, `i_mmap_lock_read()` and `i_mmap_unlock_read()`. A read lock must be held when traversing this interval tree, and a write lock must be held when manipulating it. This is typically invoked by a memory operation which performs an operation across VMAs, such as `unmap_mapping_range_tree()`.
- `nrpages` – The number of pages of this page cache entry currently present in the page cache. Importantly, this is not equal to the size of the file (contained in `struct inode->i_size`). For instance, a newly opened file which does not have pages in the page cache yet will have zero `nrpages`, and each time pages are read into the page cache, this is incremented. This therefore gives a count of the pages contained in `i_pages`. This is protected by `i_pages`'s xarray locks and manipulated in functions such as `__filemap_add_folio()` and `page_cache_delete()`.
- `writeback_index` – Specifies the page offset at which *writeback* (i.e. the writing back of modified data in the page cache to its backing store such as a disk) should commence. Used by filesystem implementations and also in `write_cache_pages()` to trace a writeback operation through to completion. See the page cache chapter for more on writeback.

- `a_ops` – A struct `address_space_operations` object which specifies functions to call for operations on this page cache object and specified by the underlying filesystem. Generic functions are often used where the filesystem does not need specific behaviour. See the page cache chapter for more on this.
- `flags` – A bitmap of flags which describes the page cache object using bits described by `enum mapping_flags`. See the page cache chapter for more details on this.
- `wb_err` – The last writeback error that occurred for the object. See the page cache chapter for more on this.
- `private_lock` – A spinlock reserved for use by the filesystem within which this object resides.
- `private_list` – A list head field reserved for use by the filesystem within which this object resides.
- `private_data` – A void pointer field reserved for use by the filesystem within which this object resides.

This object acts as an intermediary between underlying page cache folios and VMAs as both of these are ephemeral. It may be referenced by VMAs from different processes as equally a file can be mapped from different folios.

The underlying page cache folios which store the data reference the page cache object via their `mapping` field. Shared mappings (i.e. `MAP_SHARED`) *directly* map the virtual address to the underlying page cache folios.

Private mappings of file-backed memory (which are strange beasts, see Chapter 5 for more details) either reference a read-only Copy-on-Write mapping to the page cache folio or an entirely separate anonymous folio, but whose VMA is retained on the `i_mmap` tree.

If a VMA is backed by a file then a reference to that file is contained in its struct `vm_area_struct->vm_file` field. This is what is used by the page fault machinery to obtain the underlying struct `address_space` object, and is set in `mmap_region()` when the memory is mapped.

The object referenced by a VMA's `vm_file` field is a struct `file` object which contains file metadata for a file opened by a process, the most pertinent of which is the `f_mapping` field which references the `address_space` object.

A file in the linux virtual file system has an *inode* associated with it, which describes a file's metadata (but notably not its path). The kernel representation of this is the struct `inode` object, but an open file is described by struct `file` which *does* refer to a specific path.

Since a file-backed VMA will have explicitly been established via a file descriptor (and thus a file at a specific path) it makes sense to use the file object here. The `f_mapping` field duplicates the `f_inode`'s `i_mapping` field, but since the file's reference to the inode is cached, it is useful to also cache the mapping. Typically this mapping is simply equal to `&f_inode->i_data` but this can't always be assumed to be the case.

The more interesting operations involving the page cache occur on page fault (see section 6) which invoke the VMA's `struct vm_area_struct->vm_ops->fault()` handler, either customised by the file system or, more typically, having been set to the `filemap_fault()` function.

This performs the actual ‘faulting in’ of files, utilising the `a_ops` set of functions (a `struct address_space_operations` object) to perform the underlying work required to actually read from and write back to the underlying file.

An interesting aspect of the page cache is that, when reading data from disk, it is likely that the user will want to access more than the first page of data, nor will the underlying block device necessarily be optimised for retrieving data at this granularity. Therefore, it makes sense to *readahead*, i.e. read more data in the page cache than is requested.

It is also useful to keep page cache entries around even after they are finished with, as reading file data from the page cache is *significantly* faster than reading it from a hard disk and *markedly* faster than reading from a faster store such as an SSD or NVMe device.

Therefore the kernel very much favours caching as much file data as possible and storing it in the page cache, while making it easy for the memory to equally be freed under memory pressure via reclaim if the memory is required for something else.

The freeing of the memory is controlled by reclaim (see the reclaim chapter for more on this), and precisely how we prioritise what gets freed first is determined by the position of folios on the appropriate LRU list (see section 11.2 for more on this).

The address space object is allocated by a file system's *super block* (meta-data describing a filesystem as a whole), described by `struct super_block` and implemented via the `struct super_operations->alloc_inode()`.

The address space, including its `a_ops` is set by the file system itself, for example `ext4` initialises inodes in `__ext4_iget()`, setting `a_ops` via `ext4_set_aops()`.

This has been a whistle-stop tour as the page cache is large enough of a topic to deserve its own chapter, so refer to that for a significantly more detailed analysis.

4.6 Per-process memory management statistical counters

Memory management statistics are vitally important for users as they are the primary means to determine the status of memory within the system. These counters track a number of different things, each of which are accessible via interfaces like `procfs` (see section 14.5 for more details).

The key *per-process* memory management counters are:

- `MM_FILEPAGES` – Counts *resident* pages containing data backed by files, excluding those backed by `shmem` memory (see below).
- `MM_ANONPAGES` – Counts *resident* anonymous memory.

- `MM_SWAPENTS` – Counts the number of anonymous swap entries, i.e. the number of pages which are currently swapped out.
- `MM_SHMEMPAGES` – Counts *resident* pages containing data backed by shmem files, e.g. tmpfs and shared ‘anonymous’ mappings.

There are `NR_MM_COUNTERS` of these values maintained in an anonymous enum.

The values themselves are stored in two places – per-thread `struct task_rss_stat` state stored in `struct task_struct->rss_stat` and per-process address space specific `struct mm_rss_stat` state stored in `mm_struct->rss_stat`.

Why are there two different locations? This is to account for the fact that these counters belong to the virtual address space, but may be updated by multiple threads, each of which may do so concurrently.

NOTE

In linux, each individual thread is treated as if it were a different process, each with independent `struct task_struct` objects all sharing the same `mm_struct`.

In order to avoid heavy lock contention, the per-thread values are updated and only ‘flushed’ to the per-address space values every `TASK_RSS_EVENTS_THRESH` events.

We examine the `struct task_rss_stat` type in Listing 4-31.

```

53 /* per-thread cached information, */
54 struct task_rss_stat {
55     int events;      /* for synchronization threshold */
56     int count[NR_MM_COUNTERS];
57 };

```

Listing 4-31: include/linux/mm_types_task.h: `struct task_rss_stat`

The counters are simply stored in the count array, with events tracking the number of events which have occurred in order to rate limit into batches of `TASK_RSS_EVENTS_THRESH` as shown in Listing 4-32.

```

60 struct mm_rss_stat {
61     atomic_long_t count[NR_MM_COUNTERS];
62 };

```

Listing 4-32: include/linux/mm_types_task.h: `struct mm_rss_stat`

Here an atomic counter is maintained, as individual threads will update count, so the atomic value synchronises between these.

The counters are incremented and decremented by `inc_mm_counter_fast()` and `dec_mm_counter_fast()` respectively, both of which delegate the actual operate to `add_mm_counter_fast()` as shown in Listing 4-33.

```

194 static void add_mm_counter_fast(struct mm_struct *mm, int member, int val)
195 {
196     struct task_struct *task = current;
197

```

```

198         if (likely(task->mm == mm))
199             task->rss_stat.count[member] += val;
200         else
201             add_mm_counter(mm, member, val);
202     }

```

Listing 4-33: mm/memory.c: `add_mm_counter_fast()`

Note that, should the memory access be to a `mm_struct` object which does not belong to the current process, then this function falls back to `add_mm_counter()`, which simply increments the `mm_struct->rss_stat` value.

The function which checks whether the per-thread values are ready to be flushed to the per-address space values is `check_sync_rss_stat()`, which is invoked in `handle_mm_fault()`, i.e. at the start of the page fault process.

NOTE

This only occurs if the `SPLIT_RSS_COUNTING` directive is specified, which it is only if `USE_SPLIT_PTE_PTLOCKS` is also specified. This is the case for x86-64. Not all means by which statistics might get updated result in a synchronisation of RSS statistics, for instance the zero-copy TCP receive implementation in the kernel uses `vm_insert_pages()` to place data into memory, which does not update RSS statistics at all.

Examining `check_sync_rss_stat()` as shown in Listing 4-34.

```

208 static void check_sync_rss_stat(struct task_struct *task)
209 {
210     if (unlikely(task != current))
211         return;
212     if (unlikely(task->rss_stat.events++ > TASK_RSS_EVENTS_THRESH))
213         sync_mm_rss(task->mm);
214 }

```

Listing 4-34: mm/memory.c: `check_sync_rss_stat()`

This simply increments the current task's `struct task_rss_stat.event` value, if it exceeds the threshold value `TASK_RSS_EVENTS_THRESH` then it invokes `sync_mm_rss()` as shown in Listing 4-35.

```

181 void sync_mm_rss(struct mm_struct *mm)
182 {
183     int i;
184
185     for (i = 0; i < NR_MM_COUNTERS; i++) {
186         if (current->rss_stat.count[i]) {
187             add_mm_counter(mm, i, current->rss_stat.count[i]);
188             current->rss_stat.count[i] = 0;
189         }
190     }
191     current->rss_stat.events = 0;
192 }

```

Listing 4-35: `mm/memory.c: sync_mm_rss()`

This simply iterates through the MM counters and sets the values in the `mm_struct->rss_stat` via `add_mm_counter()`.

These counters are used to output memory usage statistics to users via interfaces such as `/proc/$pid/status`. It's important to note that due to the throttling described here, these values will be inaccurate by up to `TASK_RSS_EVENTS_THRESH` pages, e.g. on x86-64 up to 256 KiB, *per thread*.

Therefore, for processes with a great many threads, these values can be rather unreliable, and an alternative such as `/proc/$pid/smaps_rollup` should be used if absolute accuracy is required. See section 14.5 for a detailed description of the `procfs` interfaces which read these values.

NOTE

The inaccuracy of these values has been improved since kernel 6.2, in `mm: convert mm's rss stats into percpu_counter` (commit `f1a7941243c1`).

5

MEMORY MAPPING

As explored in Chapter 3 on Virtual Memory, a key capacity of a modern operating system is the ability to map virtual memory as it pleases and with which the properties it desires. Doing so is termed memory mapping, a topic we explore in detail here.

Examining the layout of a process in memory as shown in Figure 5-1.

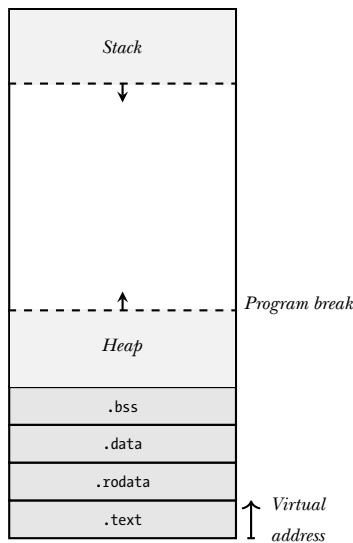


Figure 5-1: Simplified ELF image layout in memory

There are two principle means by which a process such as this allocates memory – the `brk()` system call and the `mmap()` system call.

The code of the process is stored in the `.text` segment, read-only static data in `.rodata`, initialised static data in `.data` and uninitialised data in `.bss`.

Any non-trivial program requires dynamically allocated memory, which comes in two forms – the automatically expanding **stack**, an area of memory that for most architectures grows downwards (including x86-64). Expansion and contraction of the stack happens automatically by page fault (see section 6.10) without the program author having to do anything explicitly*.

However, if memory is required that is too large for the stack or if its life-time extends beyond that of the currently running function it must be allocated from the **heap**.

The two main functions that a programmer will be familiar with for allocating from the heap are `malloc()` and `mmap()`, and more so the former of the two.

5.0.1 Program break

The classic means of expanding the dynamic memory used by a process is to expand the *program break*. This begins at (and is initialised to) the upper bound of the `.bss` section, and represents the portion of process memory referred to as the *heap*.

The `malloc()` allocator expands and shrinks this region in response to userland allocation requests which is typically used for smaller allocations, with larger ones deferred to `mmap()`.

*. Although it may be manipulated at runtime via Variable Length Arrays (VLAs) or the use of functions like `alloca()`.

The program break is typically manipulated by the standard C library function `sbrk()` which increments or decrements the program break by the specified offset, with the current program break available by invoking `sbrk(0)`. A less useful version of this is the `brk()` function which sets the program break to a specific address.

Underlying both functions is the `brk()` system call which adjusts the program break and adjusts VMAs to mark the new heap as valid. We examine it in Listing 5-1.

```

153 SYSCALL_DEFINE1(brk, unsigned long, brk)
154 {
155     unsigned long newbrk, oldbrk, origbrk;
156     struct mm_struct *mm = current->mm;
157     struct vm_area_struct *next;
158     unsigned long min_brk;
159     bool populate;
160     bool downgraded = false;
161     LIST_HEAD(uf);
162
163     if (mmap_write_lock_killable(mm))
164         return -EINTR;
165
166     origbrk = mm->brk;
167     :
168     min_brk = mm->start_brk;
169     :
170     if (brk < min_brk)
171         goto out;
172
173     /*
174      * Check against rlimit here. If this check is done later after the
175      test
176      * of oldbrk with newbrk then it can escape the test and let the data
177      * segment grow beyond its set limit the in case where the limit is
178      * not page aligned -Ram Gupta
179      */
180     if (check_data_rlimit(rlimit(RLIMIT_DATA), brk, mm->start_brk,
181                          mm->end_data, mm->start_data))
182         goto out;
183
184     newbrk = PAGE_ALIGN(brk);
185     oldbrk = PAGE_ALIGN(mm->brk);
186     if (oldbrk == newbrk) {
187         mm->brk = brk;
188         goto success;
189     }
190
191     :
192     :
193     :
194     :
195     :
196     :
197     :
198     :
199     :
200     :
201     :

```

```

202      * Always allow shrinking brk.
203      * __do_munmap() may downgrade mmap_lock to read.
204      */
205      if (brk <= mm->brk) {
206          int ret;
207
208          /*
209           * mm->brk must to be protected by write mmap_lock so update
it
210           * before downgrading mmap_lock. When __do_munmap() fails,
211           * mm->brk will be restored from origbrk.
212           */
213           mm->brk = brk;
214           ret = __do_munmap(mm, newbrk, oldbrk-newbrk, &uf, true);
215           if (ret < 0) {
216               mm->brk = origbrk;
217               goto out;
218           } else if (ret == 1) {
219               downgraded = true;
220           }
221           goto success;
222       }
223
224       /* Check against existing mmap mappings. */
225       next = find_vma(mm, oldbrk);
226       if (next && newbrk + PAGE_SIZE > vm_start_gap(next))
227           goto out;
228
229       /* Ok, looks good - let it rip. */
230       if (do_brk_flags(oldbrk, newbrk-oldbrk, 0, &uf) < 0)
231           goto out;
232       mm->brk = brk;
233
234   success:
235       populate = newbrk > oldbrk && (mm->def_flags & VM_LOCKED) != 0;
236       if (downgraded)
237           mmap_read_unlock(mm);
238       else
239           mmap_write_unlock(mm);
240       userfaultfd_unmap_complete(mm, &uf);
241       if (populate)
242           mm_populate(oldbrk, newbrk - oldbrk);
243       return brk;
244
245   out:
246       mmap_write_unlock(mm);
247       return origbrk;

```

248 }

Listing 5-1: mm/mmap.c: syscall: brk()

We acquire a write lock from the struct `mm_struct->mmap_lock` semaphore around the system call, as we are potentially manipulating VMAs.

We then perform a series of sanity checks – ensuring the requested program break is not less than the beginning of the heap (`mm->start_brk`) and that the `RLIMIT_DATA` resource limit is not exceeded (via `check_data_rlimit()`).

We page align the existing and candidate page break values, then perform a check for the simple case of the break remaining the same as it was. After these preliminary checks are complete we are able to perform the actual operation:

- If the caller is attempting to shrink the program break, this always succeeds and utilises the shared `munmap()` code path via `__do_munmap()`. This may downgrade the write `mmap_lock` to a read lock in doing so.
- Otherwise, we invoke the key VMA function `find_vma()` to find the first VMA which follows or contains this address (if any) and check whether we leave at least a page between us and any following VMA via `vm_start_gap()` (which takes into account the need for a stack guard gap if the next VMA grows downward), if not we abort.
- Finally we perform the actual program break adjustment via `do_brk_flags()` which we examine in detail below.

After a successful program break adjustment has been performed we perform some cleanup tasks and ‘populate’ (i.e. fault in) the memory range via `mm_populate()` if the page break was extended, except if `mlockall()` has been invoked with the `MCL_FUTURE` flag set, indicating that all future memory should be `mlock’d` and thus must be allowed to fault in when the memory is used.

The core function which does the heavy lifting of page break extension is `do_brk_flags()` as shown in Listing 5-2.

```

2968 /*
2969  * this is really a simplified "do_mmap". it only handles
2970  * anonymous maps. eventually we may be able to do some
2971  * brk-specific accounting here.
2972  */
2973 static int do_brk_flags(unsigned long addr, unsigned long len, unsigned long
                flags, struct list_head *uf)
2974 {
2975     struct mm_struct *mm = current->mm;
2976     struct vm_area_struct *vma, *prev;
2977     struct rb_node **rb_link, *rb_parent;
2978     pgoff_t pgoff = addr >> PAGE_SHIFT;
2979     int error;
2980     unsigned long mapped_addr;
2981
```

```

2982     /* Until we need other flags, refuse anything except VM_EXEC. */
2983     if ((flags & (~VM_EXEC)) != 0)
2984         return -EINVAL;
2985     flags |= VM_DATA_DEFAULT_FLAGS | VM_ACCOUNT | mm->def_flags;
2986
2987     mapped_addr = get_unmapped_area(NULL, addr, len, 0, MAP_FIXED);
2988     if (IS_ERR_VALUE(mapped_addr))
2989         return mapped_addr;
2990
2991     error = mlock_future_check(mm, mm->def_flags, len);
2992     if (error)
2993         return error;
2994
2995     /* Clear old maps, set up prev, rb_link, rb_parent, and uf */
2996     if (munmap_vma_range(mm, addr, len, &prev, &rb_link, &rb_parent, uf))
2997         return -ENOMEM;
2998
2999     /* Check against address space limits *after* clearing old maps... */
3000     if (!may_expand_vm(mm, flags, len >> PAGE_SHIFT))
3001         return -ENOMEM;
3002
3003     if (mm->map_count > sysctl_max_map_count)
3004         return -ENOMEM;
3005
3006     if (security_vm_enough_memory_mm(mm, len >> PAGE_SHIFT))
3007         return -ENOMEM;
3008
3009     /* Can we just expand an old private anonymous mapping? */
3010     vma = vma_merge(mm, prev, addr, addr + len, flags,
3011                     NULL, NULL, pgoff, NULL, NULL_VM_UFFD_CTX, NULL);
3012     if (vma)
3013         goto out;
3014
3015     /*
3016      * create a vma struct for an anonymous mapping
3017      */
3018     vma = vm_area_alloc(mm);
3019     if (!vma) {
3020         vm_unacct_memory(len >> PAGE_SHIFT);
3021         return -ENOMEM;
3022     }
3023
3024     vma_set_anonymous(vma);
3025     vma->vm_start = addr;
3026     vma->vm_end = addr + len;
3027     vma->vm_pgoff = pgoff;
3028     vma->vm_flags = flags;

```

```

3029         vma->vm_page_prot = vm_get_page_prot(flags);
3030         vma_link(mm, vma, prev, rb_link, rb_parent);
3031     out:
3032         perf_event_mmap(vma);
3033         mm->total_vm += len >> PAGE_SHIFT;
3034         mm->data_vm += len >> PAGE_SHIFT;
3035         if (flags & VM_LOCKED)
3036             mm->locked_vm += (len >> PAGE_SHIFT);
3037         vma->vm_flags |= VM_SOFTDIRTY;
3038         return 0;
3039     }

```

Listing 5-2: mm/mmap.c: `do_brk_flags()`

In order to avoid duplication we'll examine the logic here but not dive too deep as the approach is broadly the same as `mmap()` and we will examine the underlying details in section 5.0.2.

We start with a check – if the `flags` argument is set, only the `VM_EXEC` `vmap` flag is valid. In the usual `brk()` system call code path this value is zero to 0 so we can disregard this.

The flags are then defaulted to the default flags specified by `struct mm_struct->def_flags`, `VM_ACCOUNT` to indicate that the memory should be accounted as being in use by the userland portion of this process, and `VM_DATA_DEFAULT_FLAGS` specifying the default flags set for all data pages.

For x86-64 this is aliased to `VM_DATA_FLAGS_TSK_EXEC`, which sets `VM_READ`, `VM_WRITE`, `VM_MAYREAD`, `VM_MAYWRITE`, `VM_MAYEXEC` (it can, if the user specifies an executable heap, be set to `VM_EXEC` but this would be unusual!).

We then perform the following steps:

1. Confirm that the memory we wish to expand the page break into is *valid* by invoking `get_unmapped_area()` (see section 5.0.5), specifying `MAP_FIXED` to indicate that only the specified address is acceptable.
Note that on x86-64, despite the name of the function, it only checks that the size increase does not exceed the maximum permissible and performs no further checks, so the address may in fact be mapped (though performing a `brk()` system call this should not be the case as we have already asserted that the range is clear).
2. Check that we don't exceed the `RLIMIT_MEMLOCK` limit if `mlockall()` has been called with `MCL_FUTURE` set via `mlock_future_check()`.
3. Unmap all existing mappings in the range via `munmap_vma_range()` – this will have no effect for a `brk()` system call as we will have already asserted, under lock, that no such mappings exist. However `vm_brk_flags()`, which also invokes `do_brk_flags()`, makes no such guarantees so this check is more pertinent to that caller.
4. Check we are permitted to expand our vm space based on the address space limit `RLIMIT_AS` and the data limit `RLIMIT_DATA` via `may_expand_vm()`.
5. Check that the total map count set by the `vm.max_map_count` tunable is not exceeded.

6. Check to ensure we have sufficient memory and our memory usage does not violate any security rules in the system via `security_vm_enough_memory_mm()`.
7. Attempt to simply expand a preceding or following VMA to encompass the new area via `vma_merge()`.
8. If this is not possible, we allocate a new VMA via `vm_area_alloc()`, marking it anonymous via `vma_set_anonymous()`, setting its core flags and linking it to surrounding VMAs via `vma_link()`.

Whether merged or not, we update `mmap` statistics and set the VMA's `VM_SOFTDIRTY` flag.

Note that we will examine some of these invoked functions in more detail below as this borrows liberally from the broader `mmap()` functionality.

It's important to note that both `brk()` and `sbrk()` are not functions which users should use directly, rather they should only be used by the userland allocator in `malloc()` and `free()` implementations.

In fact, if you try to use these functions yourself you will likely end up disrupting the userland allocator's operation and of course you can have no expectation that the program break won't be moved by it.

In the vast majority of cases a user need not concern themselves with any kernel memory interfaces at all. However there are cases where a file needs to be memory-mapped, or more fine-grained control is required over the mapping, which is where `mmap()` comes into play:

5.0.2 `mmap()`

`mmap()` is arguably *the* key interface between userland and the kernel for mapping memory and thus worth examining in extensive detail. Let's begin by examining the userland interface itself in Listing 5.0.2.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

The function maps memory into the virtual address space as defined by these parameters, returning either the page-aligned virtual address of the mapped region or `MAP_FAILED` (i.e. `(void *)-1`), setting `errno` to the error that arose.

Examining each of the parameters:

- `addr` – This is either `NULL` if the returned address is of no importance (which is the typical means of invoking this function), or the virtual address the mapping should start from.
If the `MAP_FIXED` flag is set then mapping at this address is a *hard* requirement (and the function will return an error if it cannot map here). Otherwise it is a hint which the kernel will make a best effort to fulfil, but may not be able to.

All memory mapped by the kernel will be page-aligned, so if a hint address is specified which is not, the kernel will page-align it. If `MAP_FIXED` is specified and the address is not page-aligned, an error will occur.

- `length` – The size of the memory to be mapped, in bytes. The actual size of the mapping will be page-aligned, so this value will be rounded up to the nearest base page.
- `prot` – This determines how the memory is intended to be accessed and maps onto VMA flags. `PROT_NONE` (set as a single value not a flag) indicates that the memory cannot be accessed at all*, `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` can be combined to indicate whether the memory can be read from, written to or executed.
Each of `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` are converted to `VM_READ`, `VM_WRITE` and `VM_EXEC` respectively via `calc_vm_prot_bits()`.
- `flags` – These determine the characteristics of the allocation, which we will examine below.
- `fd` – If the mapping is anonymous, then this should be set to `-1`. Otherwise it specifies the file descriptor of the file to be mapped. This file descriptor can be safely closed after the mapping has been performed.
- `offset` – This is only meaningful if a file is being mapped and determines the offset within the file, in bytes, from which to start the mapping. It must be page-aligned, or an `-EINVAL` error will be returned. The offset can be outside of the size of the file at the time it is mapped, as the file size might change in future rendering the mapping valid. However if the file is accessed at a time when the offset is invalid a page fault will arise.

5.0.3 *mmap map flags*

Flags which alter mapping behaviour are passed in the `flags` field (each of which can be bitwise-combined).

These are either referenced directly in the kernel, or converted directly to VMA flags via `calc_vm_flag_bits()` (applicable to `MAP_GROWSDOWN`, `MAP_LOCKED` and `MAP_SYNC`) (note that `_calc_vm_trans()` is simply an optimised means of determining if a flag is set) as shown in Listing 5-3.

```

146 /*
147  * Combine the mmap "flags" argument into "vm_flags" used internally.
148  */
149 static inline unsigned long
150 calc_vm_flag_bits(unsigned long flags)
151 {
```

*. While it might seem somewhat pointless, `PROT_NONE` can be useful reserving virtual memory ranges that are later overwritten using `MAP_FIXED` (see below for more details on this) or for implementing guard pages in userland, i.e. memory that, if touched, generates a page fault. These can be placed at the end of valid memory ranges to protect against access of buffer overflows or otherwise out of range memory accesses to mitigate exploits or buggy code.

```

152         return _calc_vm_trans(flags, MAP_GROWSDOWN, VM_GROWSDOWN ) |
153                _calc_vm_trans(flags, MAP_LOCKED,    VM_LOCKED    ) |
154                _calc_vm_trans(flags, MAP_SYNC,      VM_SYNC      ) |
155                arch_calc_vm_flag_bits(flags);
156     }

```

Listing 5-3: include/linux/mman.h: *calc_vm_flag_bits()*

Examining each of the possible flags that can be passed in the flags field:

- **MAP_PRIVATE** – Mutually exclusive with **MAP_SHARED**. Indicates that a Copy On Write (COW) mapping should be obtained. This can be used with both anonymous *and* file-backed mappings. Typically this is combined with **MAP_ANONYMOUS** to indicate that an unshared, anonymous mapping is required. However, it can also be specified for file-backed mappings, in which case it gains very specific semantics – becoming a Copy-on-Write read-only mapping to the underlying page cache entry. If this mapping is written to, then the Copy-on-Write triggers and the mapping behaves as if it were anonymous, only initialised with the contents of the mapped file. However, if after this pages within the file are *truncated* (which can occur if the file is deleted, overwritten or shrunk in size) the anonymous pages are freed and it reverts back to being a Copy-on-Write page cache mapping (triggering SIGBUS if data does not exist in the file at the accessed page). See section 6.6 for more details on how this functions.
- **MAP_SHARED** – Indicates that more than one process might map the same underlying physical memory. This can be used both with anonymous and file-backed mappings, with the latter meaning that each process which maps the same file also maps the same page cache pages. Shared anonymous memory is useful for sharing memory between forked processes, as it prevents the mapping from adopting Copy-on-Write semantics. Behind the scenes, the memory is actually mapped as an deleted copy of /dev/zero effectively residing in tmpfs (setup via `shmem_zero_setup()`), however for all intents and purposes it is an anonymous mapping.
- **MAP_SHARED_VALIDATE** – The **MAP_SHARED** flag, for legacy compatibility reasons, does not confirm that the flags parameter contains only valid flags. If this needs to be checked, use **MAP_SHARED_VALIDATE** instead.
- **MAP_ANON**/**MAP_ANONYMOUS** – Indicates that the underlying mapping should be backed by RAM only and when faulted in, physical memory should be allocated to back it.
- **MAP_POPULATE** – Fault in the mapped memory immediately using the kernel's GUP functionality (see section 8.1.2). This invokes the `mm_populate()` function (and `__mm_populate()` in turn). Note that this faulting in is best effort, if an error occurs during faulting it is ignored, there-

fore it is possible that memory mapped this way may *still* result in page faults.

- **MAP_FIXED** – Indicates that the `addr` parameter is not a hint but rather a hard requirement. The specified address must be suitable for a virtual mapping for the architecture, which typically means it must be page-aligned, failure to do so will result in an error.
It's important to note that this flag will cause `mmap()` to unmap any mappings which already exist at the specified address, so this is a rather dangerous flag to use, and in typical usage should only be used to overwrite an already existing mapping (perhaps a `PROT_NONE` mapping) to ensure mappings are not unexpectedly overwritten.
- **MAP_FIXED_NOREPLACE** – This is equivalent to the **MAP_FIXED** flag, only avoiding the dangerous unmapping (i.e. replacing) behaviour, returning an error if the mapping would do so.
- **MAP_GROWSDOWN** – Indicates that this mapping is a stack and should grow downwards (this is a user-defined stack and thus not tied to the architecture-dependent direction of program stack growth).
Memory mapped this way has the special quality of stacks, which is that the valid VMA range is not necessarily static. When unmapped memory below the start of the stack is accessed, rather than a segfault arising, `expand_downwards()` is called which extends the start of the VMA to encompass the range up to and including the faulting address (see section 6.10 for details on this).
Stack expansion is controlled by the `RLIMIT_STACK` limit which is checked on page fault, this can be set to being unlimited at which case stack expansion is limited only by overcommit settings (see section 4.1) and the `RLIMIT_AS` and `RLIMIT_DATA` user-specified limits.
Clashes with other mappings are avoided as the kernel maintains a gap between the bottom of the stack and the immediate prior mapping* of `stack_guard_gap` bytes. This can be configured by the kernel command line parameter `stack_guard_gap`, but defaults to 256 base pages, which for x86-64 implies 1 MiB.
- **MAP_LOCKED** – Indicates that all memory faulted in for this mapping should be `mlock()`'d (see section 8.2.1 for more on this). This is equivalent to setting the `VM_LOCKED` VMA flag.
The purpose of locking memory like this is to prevent it from being reclaimed or swapped out. This is necessary for regions of memory which the user requires to remain strictly resident in memory, though doing so adds memory pressure.
If this flag is specified, then **MAP_POPULATE** is implied (as checked in `do_mmap()`). Since this function ignores any errors which occur when

*. For architectures which grow program stacks upwards, they maintain this gap *above* the stack, however this is not relevant to the discussion as an `mmap()`'d stack can only grow downwards. The gap is enforced in both directions regardless of architecture.

faulting in memory this does not guarantee that further page faults will not occur.

As the `mmap()` documentation indicates, if such faulting behaviour is not desired, then `mlock()` should be used separately to *guarantee* that memory is faulted in.

- **MAP_NORESERVE** – Indicates that no swap space should be reserved for a mapping. The practical impact of this is:
 - The containing VMA will have the `VM_NORESERVE` flag set, except if overcommit mode (see section 4.1) `OVERCOMMIT_NEVER` is set, which overrules this*.
 - Otherwise, the containing VMA is determined to be *non-accountable* by `accountable_mapping()` which results in the VMA flag `VM_ACCOUNT` *not* being set.
 - As a result, when mapping non-stack memory in `OVERCOMMIT_GUESS` overcommit mode, the free memory check in `__vm_enough_memory()` is not performed (which in this mode simply checks that the mapping size does not exceed the sum of the total physical memory installed and swap space). Security hooks also invoked by `security_vm_enough_memory_mm()` are not run either.
 - This is true of `OVERCOMMIT_ALWAYS`, but this mode always permits overcommit so it typically has no bearing, unless a security hook would have otherwise disallowed the mapping.
 - As a result, for overcommit modes other than `OVERCOMMIT_NEVER`, the `Committed_AS` statistic seen in `/proc/meminfo` is not updated (`vm_acct_memory()` is not invoked on mapping/unmapping).
 - Only writable, private mappings are accountable in any case, so this mapping flag is only meaningful for private anonymous mappings or `MAP_PRIVATE` file-backed mappings.
- **MAP_SYNC** – This is only meaningful for shared mappings, and requires `MAP_SHARED_VALIDATE` to also be specified as if only `MAP_SHARED` is this flag is explicitly masked out by applying `LEGACY_MAP_MASK` in `do_mmap()`. The flag is only meaningful for DAX (which provides Direct Access not mediated by the page cache to memory-like devices), causing dirty metadata to be flushed on write faults as checked by `dax_fault_is_synchronous()`. DAX is out of scope for this book. This flag is mapped to `VM_SYNC` by `calc_vm_flag_bits()`.
- **MAP_UNINITIALIZED** – Anonymous pages mapped by `mmap()` are always zeroed for security reasons. However, on no-MMU systems alone and only if `CONFIG_MMAP_ALLOWED_UNINITIALIZED` is set then uninitialised anonymous mappings permitted when the `MMAP_UNINITIALIZED` flag is set. This is only used on embedded devices due to the huge security hole this introduces, and is in any case limited in practice to those devices which do not possess an MMU so on modern architectures such as x86-64 this will simply not be permitted.

*. Except for `hugetlb` but this is out of scope for this discussion.

- **MAP_32BIT** – Only implemented for the x86 architecture. Indicates that mappings should only be mapped into the first 32-bits of virtual address space.
This is checked by `arch_get_unmapped_area()` which uses the legacy bottom-up `mmap` layout, with start and end addresses constrained by `find_start_end()` to fit within the 32-bit range. If the mapping was not specified to be legacy, `arch_get_unmapped_area_topdown()` is called, however this explicitly checks for **MAP_32BIT** and falls back to the legacy mode if so.
- **MAP_HUGETLB**, **MAP_HUGE_2MB**, **MAP_HUGE_1GB** – **MAP_HUGETLB** specifies that a *hugetlb* huge page should be used, with the size of the huge page specified by the latter two flags. This is handled by the *hugetlb* logic which is out of scope here.
- **MAP_NONBLOCK** – A legacy flag which currently results in the bizarre behaviour of, if **MAP_POPULATE** is also set, cancelling the **MAP_POPULATE** operation. This briefly did more than this in the distant past.
- **MAP_STACK** – *Does nothing*. May do something in the future.
- **MAP_FILE** – *Deprecated* and ignored.
- **MAP_DENYWRITE** – *Deprecated* and ignored.
- **MAP_EXECUTABLE** – *Deprecated* and ignored.

5.0.4 *mmap* kernel implementation

The `mmap()` system call implementation is architecture-specific. We examine the x86-64 version in Listing 5-4.

```

86 SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,
87                 unsigned long, prot, unsigned long, flags,
88                 unsigned long, fd, unsigned long, off)
89 {
90     if (off & ~PAGE_MASK)
91         return -EINVAL;
92
93     return ksys_mmap_pgoff(addr, len, prot, flags, fd, off >> PAGE_SHIFT);
94 }
```

Listing 5-4: `arch/x86/kernel/sys_x86_64.c: syscall:mmap()`

This invokes `ksys_mmap_pgoff()` which has the same parameters as the `mmap()` function, only the offset is expressed in pages, not bytes* (excluding out of scope huge page handling) as shown in Listing 5-5.

```

1548 unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len,
1549                               unsigned long prot, unsigned long flags,
```

*. This is utilised by the `mmap2()` system call which permits the avoidance of overflow for systems whose `off_t` is 32-bits and who need to be able to offset further. This system call is aliased to `mmap_pgoff()` which calls `ksys_mmap_pgoff()` directly.

```

1550                                unsigned long fd, unsigned long pgoff)
1551 {
1552     struct file *file = NULL;
1553     unsigned long retval;
1554
1555     if (!(flags & MAP_ANONYMOUS)) {
1556         audit_mmap_fd(fd, flags);
1557         file = fget(fd);
1558         if (!file)
1559             return -EBADF;
1560         :
1561     }
1562
1563     retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
1564     :
1565     if (file)
1566         fput(file);
1567     return retval;
1568 }

```

Listing 5-5: mm/mmap.c: ksys_mmap_pgoff()

This is essentially pass-through to `vm_mmap_pgoff()`, obtaining the `struct file` for file-backed mappings and incrementing its reference count via `fget()`. Examining `vm_mmap_pgoff()` (eliding out of scope `userfaultfd` logic) as shown in Listing 5-6.

```

539 unsigned long vm_mmap_pgoff(struct file *file, unsigned long addr,
540                             unsigned long len, unsigned long prot,
541                             unsigned long flag, unsigned long pgoff)
542 {
543     unsigned long ret;
544     struct mm_struct *mm = current->mm;
545     unsigned long populate;
546     LIST_HEAD(uf);
547
548     ret = security_mmap_file(file, prot, flag);
549     if (!ret) {
550         if (mmap_write_lock_killable(mm))
551             return -EINTR;
552         ret = do_mmap(file, addr, len, prot, flag, pgoff, &populate,
553                     &uf);
554         mmap_write_unlock(mm);
555         :
556         if (populate)
557             mm_populate(ret, populate);
558     }
559     return ret;
560 }

```

Listing 5-6: mm/util.c: vm_mmap_pgoff()

This starts by invoking `security_mmap_file()` to provide a security hook for memory mapping of files, before acquiring the process address space's `struct mm_struct->mmap_lock` semaphore for writing, performing the heavy lifting in `do_mmap()`.

The function finishes by populating (i.e. pre-faulting) the range via `mm_populate()` if specified (e.g. the user having specified `MAP_POPULATE`). This ultimately calls `__mm_populate()` which is described in section 8.1.9.

We examine the key `do_mmap()` function starting in Listing 5-7.

```

1366 /*
1367  * The caller must write-lock current->mm->mmap_lock.
1368  */
1369 unsigned long do_mmap(struct file *file, unsigned long addr,
1370                      unsigned long len, unsigned long prot,
1371                      unsigned long flags, unsigned long pgoff,
1372                      unsigned long *populate, struct list_head *uf)
1373 {
1374     struct mm_struct *mm = current->mm;
1375     vm_flags_t vm_flags;
1376     int pkey = 0;
1377
1378     *populate = 0;
1379
1380     if (!len)
1381         return -EINVAL;
1382
1383     /*
1384      * Does the application expect PROT_READ to imply PROT_EXEC?
1385      *
1386      * (the exception is when the underlying filesystem is noexec
1387      * mounted, in which case we dont add PROT_EXEC.)
1388      */
1389     if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
1390         if (!(file && path_noexec(&file->f_path)))
1391             prot |= PROT_EXEC;
1392
1393     /* force arch specific MAP_FIXED handling in get_unmapped_area */
1394     if (flags & MAP_FIXED_NOREPLACE)
1395         flags |= MAP_FIXED;

```

Listing 5-7: mm/mmap.c: do_mmap() parameter filtering

We start with some simple initialisation and filtering of parameters. If no length is specified we exit with an error, otherwise if `PROT_READ` implies `PROT_EXEC`, as it does on some architectures (or otherwise specified via the process's `personality()`).

Finally, we ensure `MAP_FIXED` is always set if `MAP_FIXED_NOPLACE` is to ensure architecture-specific logic run in `get_unmapped_area()` which is run only if the former flag is present is correctly invoked.

We examine the next part of the function in Listing 5-8.

```

1397     if (!(flags & MAP_FIXED))
1398         addr = round_hint_to_min(addr);
1399
1400     /* Careful about overflows.. */
1401     len = PAGE_ALIGN(len);
1402     if (!len)
1403         return -ENOMEM;
1404
1405     /* offset overflow? */
1406     if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
1407         return -E_OVERFLOW;
1408
1409     /* Too many mappings? */
1410     if (mm->map_count > sysctl_max_map_count)
1411         return -ENOMEM;
1412
1413     /* Obtain the address to map to. we verify (or select) it and ensure
1414      * that it represents a valid section of the address space.
1415      */
1416     addr = get_unmapped_area(file, addr, len, pgoff, flags);
1417     if (IS_ERR_VALUE(addr))
1418         return addr;
1419
1420     if (flags & MAP_FIXED_NOPLACE) {
1421         if (find_vma_intersection(mm, addr, addr + len))
1422             return -EEXIST;
1423     }
1424     :
1431     /* Do simple checking here so the lower-level routines won't have
1432      * to. we assume access permissions have been handled by the open
1433      * of the memory object, so we don't do any here.
1434      */
1435     vm_flags = calc_vm_prot_bits(prot, pkey) | calc_vm_flag_bits(flags) |
1436               mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
1437
1438     if (flags & MAP_LOCKED)
1439         if (!can_do_mlock())
1440             return -EPERM;
1441
1442     if (mlock_future_check(mm, vm_flags, len))
1443         return -EAGAIN;

```

Listing 5-8: `mm/mmap.c: do_mmap()` initial checks and setup

We start by ensuring that any specified hint address passed as `addr` (i.e. where the `MAP_FIXED` flag has not been specified) is correctly page-aligned and greater than the minimum permitted by `mmap_min_addr` via `round_hint_to_min()` as shown in Listing 5-9.

```

1304 /*
1305  * If a hint addr is less than mmap_min_addr change hint to be as
1306  * low as possible but still greater than mmap_min_addr
1307  */
1308 static inline unsigned long round_hint_to_min(unsigned long hint)
1309 {
1310     hint &= PAGE_MASK;
1311     if (((void *)hint != NULL) &&
1312         (hint < mmap_min_addr))
1313         return PAGE_ALIGN(mmap_min_addr);
1314     return hint;
1315 }
```

Listing 5-9: mm/mmap.c: round_hint_to_min()

We then ensure that both the `len` and `pgoff` parameters are checked to ensure they are not so close to the maximum value for their types that they might overflow.

Next, we test to see if the `struct mm_struct->map_count` parameter (counting the number of virtual mappings in the process address space) exceeds the `vm.max_map_count` tunable. If so we error out.

Where precisely this mapping will be situated in virtual memory is determined by the *critical* function `get_unmapped_area()`. As this function is rather a large topic, we defer discussion of it to section 5.0.5.

If `MAP_FIXED_NO_REPLACE` is specified, we determine whether any existing VMA intersects with the input address range via `find_vma_intersection()`, erroring out if so.

VMA flags always default to `VM_MAYREAD`, `VM_MAYWRITE` and `VM_MAYEXEC` – we will unset these flags should any be determined not to be applicable. In addition, further VMA flags are implied by `prot` and `flags`, this being mapped by via `calc_vm_prot_bits()` (simply mapping `PROT_xxx` to `VM_xxx`) and `calc_vm_flag_bits()` (as described in listing 5-3 above), combined with the default process address spaces flags specified in `struct mm_struct->def_flags`.

We then apply two `mlock()`-specific checks. The first is `can_do_mlock()` which checks to ensure the `RLIMIT_MEMLOCK` is non-zero (or the thread possesses the `CAP_IPC_LOCK` capability* which overrides this limit). The second is `mlock_future_check()` which, if `VM_LOCKED` is set (i.e. via `MAP_LOCKED`), ensures that locking the length of this mapping would not cause the `RLIMIT_MEMLOCK` user limit to be exceeded (again, unless the thread possesses the `CAP_IPC_LOCK` capability).

*. Capabilities are a fine-grained means of assigning permissions on a per-thread basis.

Once these checks are complete, we go ahead and check the flags field is valid and adjust VMA flags as necessary. There is separate logic for both file-backed and anonymous mappings, beginning with the file-backed case as shown in Listing 5-10.

```

1445     if (file) {
1446         struct inode *inode = file_inode(file);
1447         unsigned long flags_mask;
1448
1449         if (!file_mmap_ok(file, inode, pgoff, len))
1450             return -EOVERFLOW;
1451
1452         flags_mask = LEGACY_MAP_MASK | file->f_op->
mmap_supported_flags;
1453
1454         switch (flags & MAP_TYPE) {
1455         case MAP_SHARED:
1456             /*
1457              * Force use of MAP_SHARED_VALIDATE with non-legacy
1458              * flags. E.g. MAP_SYNC is dangerous to use with
1459              * MAP_SHARED as you don't know which consistency
1460              * you will get. We silently ignore unsupported flags
1461              * with MAP_SHARED to preserve backward compatibility.
1462              */
1463             flags &= LEGACY_MAP_MASK;
1464             fallthrough;
1465         case MAP_SHARED_VALIDATE:
1466             if (flags & ~flags_mask)
1467                 return -EOPNOTSUPP;
1468             if (prot & PROT_WRITE) {
1469                 if (!(file->f_mode & FMODE_WRITE))
1470                     return -EACCES;
1471                 if (IS_SWAPFILE(file->f_mapping->host))
1472                     return -ETXTBSY;
1473             }
1474             /*
1475              * Make sure we don't allow writing to an append-only
1476              * file..
1477              */
1478             if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
1479                 return -EACCES;
1480
1481             vm_flags |= VM_SHARED | VM_MAYSHARE;
1482             if (!(file->f_mode & FMODE_WRITE))
1483                 vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
1484             fallthrough;
1485

```

```

1486         case MAP_PRIVATE:
1487             if (!(file->f_mode & FMODE_READ))
1488                 return -EACCES;
1489             if (path_noexec(&file->f_path)) {
1490                 if (vm_flags & VM_EXEC)
1491                     return -EPERM;
1492                 vm_flags &= ~VM_MAYEXEC;
1493             }
1494
1495             if (!file->f_op->mmap)
1496                 return -ENODEV;
1497             if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
1498                 return -EINVAL;
1499             break;
1500
1501         default:
1502             return -EINVAL;
1503     }

```

Listing 5-10: mm/mmap.c: do_mmap() file-backed flag checks

We start by checking the size of the mapped file is reasonable via `file_mmap_ok()` as shown in Listing 5-11.

```

1353 static inline bool file_mmap_ok(struct file *file, struct inode *inode,
1354                                unsigned long pgoff, unsigned long len)
1355 {
1356     u64 maxsize = file_mmap_size_max(file, inode);
1357
1358     if (maxsize && len > maxsize)
1359         return false;
1360     maxsize -= len;
1361     if (pgoff > maxsize >> PAGE_SHIFT)
1362         return false;
1363     return true;
1364 }

```

Listing 5-11: mm/mmap.c: file_mmap_ok()

This checks that the length of the mapping (the `len` parameter) is less than or equal to the maximum permissible size and the offset within the mapping (the `pgoff` parameter, expressed in base pages) would not result in an offset into the file that would exceed the maximum size.

The maximum size is determined by `file_mmap_size_max()`. The key aim of this function is to avoid overflow for 32-bit systems as the file size is 64-bit but we must ensure that indexes into this region are correct even for drivers which might not behave correctly. For 64-bit systems this is not a concern. This logic was added in commit `be83bbf80682: mmap: introduce sane default mmap limits`.

After this check is performed, we establish a mask of accepted mapping flags by combining `LEGACY_MAP_MASK` with flags additionally supported by the file system, as determined by the struct `file_operations` handler in struct `file->f_op->mmap_supported_flags`.

The `LEGACY_MAP_MASK` consists of the key mapping flags. Excluding deprecated and hugetlb flags this consists of `MAP_SHARED`, `MAP_PRIVATE`, `MAP_FIXED`, `MAP_ANONYMOUS`, `MAP_UNINITIALIZED`, `MAP_GROWSDOWN`, `MAP_LOCKED`, `MAP_NORESERVE`, `MAP_POPULATE` and `MAP_32BIT`.

The *type* of mapping is masked by `MAP_TYPE` which masks `MAP_PRIVATE`, `MAP_SHARED` and `MAP_SHARED_VALIDATE` and we process these flags as follows:

- `MAP_SHARED` – For legacy reasons, when this flag is specified, we permit unrecognised or invalid flags to be specified which are silently ignored by simply masking against `LEGACY_MAP_MASK`. By doing so, we both maintain the legacy behaviour of not validating flags, while simultaneously preventing the use of flags which might be dangerous if used unintentionally (e.g. `MAP_SHARED`).

With flags now masked to a valid subset, we can simply fall through to the `MAP_SHARED_VALIDATE` case and share the proceeding logic between two cases.

- `MAP_SHARED_VALIDATE` – Having already established the set of valid flags as being those specified by `LEGACY_MAP_MASK` and `file->f_op->mmap_supported_flags`, so we simply return an error should flags specify any flags other than these.

If the mapping is read/write (i.e. `PROT_WRITE` is specified), then we ensure that the file is open with write permissions and that the file is not designated a swap file, if either of these are not the case, we return an error. Additionally, if the file is append-only, we cannot permit the caller to memory map the file in write mode, as this would emphatically not be append-only, so equally we raise an error should this be attempted.

Finally, we specify shared mapping-specific VMA flags – `VM_MAYSHARE` and if the file is writable `VM_SHARED`, otherwise we clear the `VM_MAYWRITE` flag. Again, we share logic between different modes by falling through to the `MAP_PRIVATE` case at this point.

- `MAP_PRIVATE` – If the file is not opened in read mode, then naturally we error out. Equally, if the file is mounted on a file system which does not permit execution (checked via `path_noexec()`), we either error out if `VM_EXEC` is set (implied by `PROT_EXEC`) or otherwise clear the `VM_MAYEXEC` VMA flag.

We then perform an absolutely key check – if the struct `file_operations` handler in struct `file->f_op->mmap` is not specified (something that must be provided by the filesystem), then memory-mapping this file is simply not supported so we must error out.

Finally, if this memory would be mapped as a stack (implied by `VM_GROWSDOWN*`), then we error out, as mapping memory file-backed and a stack simply makes no sense.

Having examined the file-backed logic, let's consider the far simpler anonymous mapping logic in Listing 5-12

```

1504     } else {
1505         switch (flags & MAP_TYPE) {
1506             case MAP_SHARED:
1507                 if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
1508                     return -EINVAL;
1509                 /*
1510                  * Ignore pgoff.
1511                  */
1512                 pgoff = 0;
1513                 vm_flags |= VM_SHARED | VM_MAYSHARE;
1514                 break;
1515             case MAP_PRIVATE:
1516                 /*
1517                  * Set pgoff according to addr for anon_vma.
1518                  */
1519                 pgoff = addr >> PAGE_SHIFT;
1520                 break;
1521             default:
1522                 return -EINVAL;
1523         }
1524     }

```

Listing 5-12: `mm/mmap.c: do_mmap()` anonymous flag checks

If the mapping is shared, then we check to ensure this mapping is not a stack (a shared stack makes no sense). If it is shared, then we reset the offset into the mapping specified by `pgoff`, since the shared anonymous mapping is actually achieved by mapping a deleted inode of a tmpfs copy of `/dev/zero`, and offsetting into this makes no sense.

We additionally set `VM_SHARED` and `VM_MAYSHARE` to indicate that this is a shared mapping.

If the mapping is private, then importantly, we specify a *virtual* page offset (again ignoring any that might have erroneously been specified by the user) – which is simply the virtual address shifted by the base page size. This is useful for forking, see the discussion of the reverse mapping (Chapter 7) for more on how this is utilised.

Finally, if neither `MAP_SHARED` or `MAP_PRIVATE` are specified, we error out. Note that `MAP_SHARED_VALIDATE` is not a valid flag for anonymous mappings.

Examining the final part of `do_mmap()` in Listing 5-13.

*. Some architectures may specify the `VM_GROWSUP` flag to indicate a stack which grows upward, however we do not consider such architectures here.

```

1526     /*
1527      * Set 'VM_NORESERVE' if we should not account for the
1528      * memory use of this mapping.
1529      */
1530     if (flags & MAP_NORESERVE) {
1531         /* We honor MAP_NORESERVE if allowed to overcommit */
1532         if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
1533             vm_flags |= VM_NORESERVE;
1534
1535         /* hugetlb applies strict overcommit unless MAP_NORESERVE */
1536         if (file && is_file_hugepages(file))
1537             vm_flags |= VM_NORESERVE;
1538     }
1539
1540     addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
1541     if (!IS_ERR_VALUE(addr) &&
1542         ((vm_flags & VM_LOCKED) ||
1543          (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
1544         *populate = len;
1545     return addr;
1546 }

```

Listing 5-13: mm/mmap.c: do_mmap() shared mapping logic

We start by handling the MAP_NORESERVE case. If the overcommit mode (see section 4.1) is *not* OVERCOMMIT_NEVER or if this is huge page file, then the VM_NORESERVE flag is set (see the discussion of map flags above for more discussion on this).

Finally, with all flags set and checks performed, the actual mapping is performed by mmap_region(), which we will examine below.

If this succeeds, MAP_POPULATE semantics are checked – if the region is not locked (which would already have been populated), MAP_POPULATE is specified but MAP_NONBLOCK is not then the population is performed.

We start examining mmap_region() in Listing 5-14.

```

1681 unsigned long mmap_region(struct file *file, unsigned long addr,
1682                          unsigned long len, vm_flags_t vm_flags, unsigned long pgoff,
1683                          struct list_head *uf)
1684 {
1685     struct mm_struct *mm = current->mm;
1686     struct vm_area_struct *vma, *prev, *merge;
1687     int error;
1688     struct rb_node **rb_link, *rb_parent;
1689     unsigned long charged = 0;
1690
1691     /* Check against address space limit. */
1692     if (!may_expand_vm(mm, vm_flags, len >> PAGE_SHIFT)) {
1693         unsigned long nr_pages;

```

```

1694
1695         /*
1696          * MAP_FIXED may remove pages of mappings that intersects with
1697          * requested mapping. Account for the pages it would unmap.
1698          */
1699         nr_pages = count_vma_pages_range(mm, addr, addr + len);
1700
1701         if (!may_expand_vm(mm, vm_flags,
1702                             (len >> PAGE_SHIFT) - nr_pages))
1703             return -ENOMEM;
1704     }
1705
1706     /* Clear old maps, set up prev, rb_link, rb_parent, and uf */
1707     if (munmap_vma_range(mm, addr, len, &prev, &rb_link, &rb_parent, uf))
1708         return -ENOMEM;
1709
1710     /*
1711      * Private writable mapping: check memory availability
1712      */
1713     if (accountable_mapping(file, vm_flags)) {
1714         charged = len >> PAGE_SHIFT;
1715         if (security_vm_enough_memory_mm(mm, charged))
1716             return -ENOMEM;
1717         vm_flags |= VM_ACCOUNT;
1718     }
1719
1720     /*
1721      * Can we just expand an old mapping?
1722      */
1723     vma = vma_merge(mm, prev, addr, addr + len, vm_flags,
1724                     NULL, file, pgoff, NULL, NULL_VM_UFFD_CTX, NULL);
1725     if (vma)
1726         goto out;

```

Listing 5-14: mm/mmap.c: mmap_region() initialisation and early exit

We start by checking the address space limit via `may_expand_vm()`, which ensures `RLIMIT_AS` and `RLIMIT_DATA` limits are respected. If this is apparently violated, we perform the expensive task of accounting for any overlapping mappings using `count_vma_pages_range()` and subtract this from the checked total, before trying again.

Note that any overlap of VMAs within the mapped range must imply `MAP_FIXED` was specified, as otherwise the calling code would have errored out by now, so we can simply go ahead and unmap any memory already within the range via `munmap_vma_range()`. This, for convenience, invokes `find_vma_links()` to find the previous VMA, parent node, and pointer to where the newly inserted VMA should be referenced from (i.e. a pointer to its parent's left or right node field).

See section 5.0.6 below for a detailed examination of how memory is unmapped.

Finally, if we can simply merge the candidate new VMA with an existing VMA either immediately prior to succeeding it, then we do so via `vma_merge()` and avoid having to allocate and set up a new VMA.

We examine the code which allocates and initialises a new VMA in Listing 5-15.

```

/*
 * Determine the object being mapped and call the appropriate
 * specific mapper. the address has already been validated, but
 * not unmapped, but the maps are removed from the list.
 */
vma = vm_area_alloc(mm);
if (!vma) {
    error = -ENOMEM;
    goto unacct_error;
}

vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = vm_get_page_prot(vm_flags);
vma->vm_pgoff = pgoff;

if (file) {
    if (vm_flags & VM_SHARED) {
        error = mapping_map_writable(file->f_mapping);
        if (error)
            goto free_vma;
    }

    vma->vm_file = get_file(file);
    error = call_mmap(file, vma);
    if (error)
        goto unmap_and_free_vma;

    /* Can addr have changed??
     *
     * Answer: Yes, several device drivers can do it in their
     * f_op->mmap method. -DaveM
     * Bug: If addr is changed, prev, rb_link, rb_parent should
     * be updated for vma_link()
     */
    WARN_ON_ONCE(addr != vma->vm_start);

    addr = vma->vm_start;

```

```

/* If vm_flags changed after call_mmap(), we should try merge vma again
 * as we may succeed this time.
 */
if (unlikely(vm_flags != vma->vm_flags && prev)) {
    merge = vma_merge(mm, prev, vma->vm_start, vma->vm_end, vma->vm_flags,
        NULL, vma->vm_file, vma->vm_pgoff, NULL, NULL_VM_UFFD_CTX, NULL
);
    if (merge) {
        /* ->mmap() can change vma->vm_file and fput the original file.

        So
        file
        * fput the vma->vm_file here or we would add an extra fput for

        * and cause general protection fault ultimately.
        */
        fput(vma->vm_file);
        vm_area_free(vma);
        vma = merge;
        /* Update vm_flags to pick up the change. */
        vm_flags = vma->vm_flags;
        goto unmap_writable;
    }

    vm_flags = vma->vm_flags;
} else if (vm_flags & VM_SHARED) {
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
} else {
    vma_set_anonymous(vma);
}

/* Allow architectures to sanity-check the vm_flags */
if (!arch_validate_flags(vma->vm_flags)) {
    error = -EINVAL;
    if (file)
        goto unmap_and_free_vma;
    else
        goto free_vma;
}

vma_link(mm, vma, prev, rb_link, rb_parent);

```

Listing 5-15: mm/mmap.c: `mmap_region()` VMA allocation and initialisation

We allocate and initialise a new VMA via `vm_area_alloc()` (for more details on this see section 4.4.2), before assigning the VMA's `vm_start`, `vm_end`, `vm_flags` and `vm_pgoff` with input parameters.

The protection flags applicable to the VMA are determined via the architecture-specific `vm_get_page_prot()` function, described in detail in the virtual memory chapter in section 3.1.3.

As usual, the file-backed mapping case is more complicated than the anonymous one – we start by indicating that we have a new writable mapping via `mapping_map_writable()` (the kernel questionably assumes that it is implicit that `VM_SHARED` implies that the mapping is writable) if shared.

Next, we increment the reference count on the struct file associated with the mapping, before invoking the filesystem’s customised struct file_operations handler in `file->f_op->mmap` via `call_mmap()`, erroring out if this fails.

We assert that we don’t have buggy drivers doing insane things with the address, before checking for an edge case – the call to the file system’s memory mapping hook above may have altered VMA flags which may cause a previously failed merge to now succeed, so we try the merge again via `vma_merge()`, cleaning up after ourselves and exiting if so.

Skipping the anonymous shared case for now which we shall return to shortly – if the VMA is private and anonymous, we invoke `vma_set_anonymous()` which clears the `vm_area_alloc()->vm_ops` field to explicitly indicate that the VMA is anonymous (note that, on allocation the VMA has its fields initialised by `vma_init()` which defaults `vm_ops` to a static empty dummy `vm_ops` object).

In the case of a shared anonymous mapping (i.e. one which the user has specified both `MAP_ANONYMOUS` and `MAP_SHARED`), special handling is required, as the kernel does not truly support shared anonymous mappings, but rather must work around this with RAM-based file-backed handling. This is done in `shmem_zero_setup()` as shown in Listing 5-16.

```

4222 /**
4223  * shmem_zero_setup - setup a shared anonymous mapping
4224  * @vma: the vma to be mmapped is prepared by do_mmap
4225  */
4226 int shmem_zero_setup(struct vm_area_struct *vma)
4227 {
4228     struct file *file;
4229     loff_t size = vma->vm_end - vma->vm_start;
4230
4231     /*
4232      * Cloning a new file under mmap_lock leads to a lock ordering
4233      * conflict
4234      * between XFS directory reading and selinux: since this file is only
4235      * accessible to the user through its mapping, use S_PRIVATE flag to
4236      * bypass file security, in the same way as shmem_kernel_file_setup().
4237      */
4238     file = shmem_kernel_file_setup("dev/zero", size, vma->vm_flags);
4239     if (IS_ERR(file))
4240         return PTR_ERR(file);
4241 
```

```

4241     if (vma->vm_file)
4242         fput(vma->vm_file);
4243     vma->vm_file = file;
4244     vma->vm_ops = &shmem_vm_ops;
4245
4246     return 0;
4247 }

```

Listing 5-16: mm/shmem.c: shmem_zero_setup()

This establishes a copy of `/dev/zero` of the designated size as an unlinked tmpfs file* (i.e. one tied to a newly generated tmpfs inode which is in effect deleted from the file system but which through reference count we keep around), setting up `vm_file` and `vm_ops` accordingly. The heavy lifting of this is performed by `shmem_kernel_file_setup()`, the description of which is out of scope here.

Once this setup is established for each of the file-backed, anonymous shared and anonymous private cases, we invoke the `arch_validate_flags()` function which provides a hook for some architectures which need to validate VMA flags when VMAs are created.

We then link this newly allocated and now set up VMA into the VMA linked list and red/black tree via `vma_link()` (see listing 4-21).

Finally, let's examine `mmap_region()`'s error handling and shared code (eliding out of scope huge page, uprobe and performance event logic) in Listing 5-17.

```

1806
1807     :
1813     /* Once vma denies write, undo our temporary denial count */
1814     unmap_writable:
1815         if (file && vm_flags & VM_SHARED)
1816             mapping_unmap_writable(file->f_mapping);
1817         file = vma->vm_file;
1818     out:
1819         :
1821         vm_stat_account(mm, vm_flags, len >> PAGE_SHIFT);
1822         if (vm_flags & VM_LOCKED) {
1823             if ((vm_flags & VM_SPECIAL) || vma_is_dax(vma) ||
1824                 is_vm_hugetlb_page(vma) ||
1825                 vma == get_gate_vma(current->mm))
1826                 vma->vm_flags &= VM_LOCKED_CLEAR_MASK;
1827             else
1828                 mm->locked_vm += (len >> PAGE_SHIFT);
1829         }
1830         :
1834     /*

```

*. Note that 'shmem' implements the RAM-based file systems, most notable of which is tmpfs.

```

1835      * New (or expanded) vma always get soft dirty status.
1836      * Otherwise user-space soft-dirty page tracker won't
1837      * be able to distinguish situation when vma area unmapped,
1838      * then new mapped in-place (which must be aimed as
1839      * a completely new data area).
1840      */
1841      vma->vm_flags |= VM_SOFTDIRTY;
1842
1843      vma_set_page_prot(vma);
1844
1845      return addr;
1846
1847  unmap_and_free_vma:
1848      fput(vma->vm_file);
1849      vma->vm_file = NULL;
1850
1851      /* Undo any partial mapping done by a device driver. */
1852      unmap_region(mm, vma, prev, vma->vm_start, vma->vm_end);
1853      if (vm_flags & VM_SHARED)
1854          mapping_unmap_writable(file->f_mapping);
1855  free_vma:
1856      vm_area_free(vma);
1857  unacct_error:
1858      if (charged)
1859          vm_unacct_memory(charged);
1860      return error;
1861  }

```

Listing 5-17: mm/mmap.c: mmap_region() error handling and shared logic

Each of the error handling cases effectively undo existing actions as expected, so we will not examine these closely and instead focus on the out case, i.e. the shared logic for all cases.

We start by updating statistics via `vm_stat_account()`, before performing some checks around `mlock()` logic – if the mapping is special (e.g. instances where `vm_normal_page()` would return `NULL`, i.e. those mappings which have no struct `page` (and thus no struct `folio`) associated with them), or are special by being one of the (out scope) instances of being a direct access, `hugetlb` or gate mapping, the `VM_LOCKED` is silently cleared.

If the locked mapping is permitted the process address space's struct `mm_struct->locked_vm` field is updated accordingly.

We always set the *soft-dirty* VMA flag `VM_SOFTDIRTY`, as otherwise as the comment indicates, there is no means by which userland can correctly account for memory being unmapped then freshly mapped in place.

Finally, the struct `vm_area_struct->vm_page_prot` field of the VMA, used to determine the protection bits for the PTE entries used to map in this mapping are set by `vma_set_page_prot()` as shown in Listing 5-18.

```

89  /* Update vma->vm_page_prot to reflect vma->vm_flags. */

```

```

90 void vma_set_page_prot(struct vm_area_struct *vma)
91 {
92     unsigned long vm_flags = vma->vm_flags;
93     pgprot_t vm_page_prot;
94
95     vm_page_prot = vm_pgprot_modify(vma->vm_page_prot, vm_flags);
96     if (vma_wants_writenotify(vma, vm_page_prot)) {
97         vm_flags &= ~VM_SHARED;
98         vm_page_prot = vm_pgprot_modify(vm_page_prot, vm_flags);
99     }
100     /* remove_protection_ptes reads vma->vm_page_prot without mmap_lock */
101     WRITE_ONCE(vma->vm_page_prot, vm_page_prot);
102 }

```

Listing 5-18: mm/mmap.c: `vma_set_page_prot()`

This establishes page protection flags as determined by `vm_pgprot_modify()`, which again sets the flags according to the architecture-specific `vm_get_page_prot()` function (again, described in detail in the virtual memory chapter in section 3.1.3), but also establishes that a read-only page fault will occur for file systems which require write notification for shared, writable pages, checked via `vma_wants_writenotify()`.

This is vital, as the majority of physically-backed file systems *must* be aware of a dirty page being written to so they know to write them back to disk at some later point. This process is known as **dirty tracking**, and is established here by setting flags as if `VM_SHARED` was not specified, which defaults to a read only mapping (this is intended for Copy-on-Write mappings but doubles up usefully here for write notify shared ones).

Examining `vma_wants_writenotify()` as shown in Listing 5-19.

```

1624 /*
1625  * Some shared mappings will want the pages marked read-only
1626  * to track write events. If so, we'll downgrade vm_page_prot
1627  * to the private version (using protection_map[] without the
1628  * VM_SHARED bit).
1629  */
1630 int vma_wants_writenotify(struct vm_area_struct *vma, pgprot_t vm_page_prot)
1631 {
1632     vm_flags_t vm_flags = vma->vm_flags;
1633     const struct vm_operations_struct *vm_ops = vma->vm_ops;
1634
1635     /* If it was private or non-writable, the write bit is already clear
1636      */
1637     if ((vm_flags & (VM_WRITE|VM_SHARED)) != ((VM_WRITE|VM_SHARED)))
1638         return 0;
1639
1640     /* The backer wishes to know when pages are first written to? */
1641     if (vm_ops && (vm_ops->page_mkwrite || vm_ops->pfn_mkwrite))
1642         return 1;

```

```

1642
1643     /* The open routine did something to the protections that
pgprot_modify
1644     * won't preserve? */
1645     if (pgprot_val(vm_page_prot) !=
1646         pgprot_val(vm_pgprot_modify(vm_page_prot, vm_flags)))
1647         return 0;
1648
1649     /*
1650     * Do we need to track softdirty? hugetlb does not support softdirty
1651     * tracking yet.
1652     */
1653     if (vma_soft_dirty_enabled(vma) && !is_vm_hugetlb_page(vma))
1654         return 1;
1655
1656     /* Specialty mapping? */
1657     if (vm_flags & VM_PFNMAP)
1658         return 0;
1659
1660     /* Can the mapping track the dirty pages? */
1661     return vma->vm_file && vma->vm_file->f_mapping &&
1662         mapping_can_writeback(vma->vm_file->f_mapping);
1663 }

```

Listing 5-19: mm/mmap.c: *vma_wants_writenotify()*

Write notify is only applicable to write-shared mappings, so we only proceed with the check if the mapping is write-shared. In all instances of a write-shared mapping possessing the struct `vm_operations_struct` `page_mkwrite` or `pfn_mkwrite` (for special mappings) handlers provided for by a file system, then write notify naturally must be enabled.

The rest of the function deals with various edge cases – if the read-only flag simply won't be preserved, then we cannot enable write-notify, otherwise we check to see if *soft-dirty* is enabled via `vma_soft_dirty_enabled()` (which confusingly checks to see whether `VM_SOFTDIRTY` is *cleared* – because that indicates that the page is not soft-dirty and we need write notification to determine when it becomes dirtied). We do not do this in the hugetlb case as this does not support soft-dirty.

If the mapping is special, i.e. the underlying memory it maps is not described by struct `page` objects, and `pfn_mkwrite` has not been specified, then we have no need for write notification.

Finally, if the mapping is file-backed and the mapping's associated struct `file` object is capable of writeback (see the page cache chapter for more on this), we assume write notification is required.

5.0.5 Choosing where to map

In order to determine which unmapped area in which to place a new mapping the function `get_unmapped_area()` is invoked in Listing 5-20.

```

2208 unsigned long
2209 get_unmapped_area(struct file *file, unsigned long addr, unsigned long len,
2210                  unsigned long pgoff, unsigned long flags)
2211 {
2212     unsigned long (*get_area)(struct file *, unsigned long,
2213                              unsigned long, unsigned long, unsigned long)
2214     ;
2215
2216     unsigned long error = arch_mmap_check(addr, len, flags);
2217     if (error)
2218         return error;
2219
2220     /* Careful about overflows.. */
2221     if (len > TASK_SIZE)
2222         return -ENOMEM;
2223
2224     get_area = current->mm->get_unmapped_area;
2225     if (file) {
2226         if (file->f_op->get_unmapped_area)
2227             get_area = file->f_op->get_unmapped_area;
2228     } else if (flags & MAP_SHARED) {
2229         /*
2230          * mmap_region() will call shmem_zero_setup() to create a file
2231          * so use shmem's get_unmapped_area in case it can be huge.
2232          * do_mmap() will clear pgoff, so match alignment.
2233          */
2234         pgoff = 0;
2235         get_area = shmem_get_unmapped_area;
2236     }
2237
2238     addr = get_area(file, addr, len, pgoff, flags);
2239     if (IS_ERR_VALUE(addr))
2240         return addr;
2241
2242     if (addr > TASK_SIZE - len)
2243         return -ENOMEM;
2244     if (offset_in_page(addr))
2245         return -EINVAL;
2246
2247     error = security_mmap_addr(addr);
2248     return error ? error : addr;
2249 }

```

Listing 5-20: mm/mmap.c: `get_unmapped_area()`

This function either returns a virtual address to place a mapping targeting address `addr` (which will be equal to zero should no specific address hint

be provided), of size `len` bytes, at a page offset within the mapping of `pgoff`, with `mmap()` flags supplied in the `flags` parameter, or an error value.

An architecture-specific check is performed via `arch_mmap_check()`, however for x86-64 this is a no-op. After checking to ensure that the specified length is sane, we determine which function to use:

- If file-backed and the struct `file->f_op` (of type struct `file_operations`) file operations object contains a `get_unmapped_area` function, we use this. This is usually a function which ultimately calls the general unmapped area handler, but augments it for the specific purposes of the file system.
- If not file-backed, but `MAP_SHARED` is specified (i.e. this is an anonymous, shared mapping), the `shmem*`-specific function `shmem_get_unmapped_area()` is called, which ultimately invokes the general unmapped area handler below with some additional handling for huge pages.
- Otherwise, the general unmapped area handler is invoked, which is specified in the struct `mm_struct->get_unmapped_area` field.

The general unmapped area handler is determined early in a process's life in `setup_new_exec()` which invokes `arch_pick_mmap_layout()`. As the name implies, this is architecture-specific. For x86-64, this specifies the use of `arch_get_unmapped_area_topdown()` unless it is a legacy `mmap()` invocation, determined by `mmap_is_legacy()`, in which case `arch_get_unmapped_area()` is used instead.

Examining `mmap_is_legacy()` as shown in Listing 5-21.

```

62 static int mmap_is_legacy(void)
63 {
64     if (current->personality & ADDR_COMPAT_LAYOUT)
65         return 1;
66
67     return sysctl_legacy_va_layout;
68 }
```

Listing 5-21: arch/x86/mm/mmap.c: `mmap_is_legacy()`

This will result in a legacy `mmap` if the personality of the process specifies `ADDR_COMPAT_LAYOUT`, or if the `sysctl` tunable `vm.legacy_va_layout` is set. This can be set using the `setarch` CLI tool.

Examining the general unmapped area handler used for all but legacy cases, `arch_get_unmapped_area_topdown()` (this is architecture-specific, so as usual we limit ourselves to the x86-64 case for brevity) as shown in Listing 5-22.

```

160 unsigned long
161 arch_get_unmapped_area_topdown(struct file *filp, const unsigned long addr0,
162                               const unsigned long len, const unsigned long pgoff,
```

*. `shmem` is the filesystem which underlies `tmpfs` and `ramfs`.

```

163             const unsigned long flags)
164 {
165     struct vm_area_struct *vma;
166     struct mm_struct *mm = current->mm;
167     unsigned long addr = addr0;
168     struct vm_unmapped_area_info info;
169
170     /* requested length too big for entire address space */
171     if (len > TASK_SIZE)
172         return -ENOMEM;
173
174     /* No address checking. See comment at mmap_address_hint_valid() */
175     if (flags & MAP_FIXED)
176         return addr;
177
178     /* for MAP_32BIT mappings we force the legacy mmap base */
179     if (!in_32bit_syscall() && (flags & MAP_32BIT))
180         goto bottomup;
181
182     /* requesting a specific address */
183     if (addr) {
184         addr &= PAGE_MASK;
185         if (!mmap_address_hint_valid(addr, len))
186             goto get_unmapped_area;
187
188         vma = find_vma(mm, addr);
189         if (!vma || addr + len <= vm_start_gap(vma))
190             return addr;
191     }
192 get_unmapped_area:
193
194     info.flags = VM_UNMAPPED_AREA_TOPDOWN;
195     info.length = len;
196     info.low_limit = PAGE_SIZE;
197     info.high_limit = get_mmap_base(0);
198
199     /*
200      * If hint address is above DEFAULT_MAP_WINDOW, look for unmapped area
201      * in the full address space.
202      *
203      * !in_32bit_syscall() check to avoid high addresses for x32
204      * (and make it no op on native i386).
205      */
206     if (addr > DEFAULT_MAP_WINDOW && !in_32bit_syscall())
207         info.high_limit += TASK_SIZE_MAX - DEFAULT_MAP_WINDOW;
208
209     info.align_mask = 0;

```

```

210         info.align_offset = pgoff << PAGE_SHIFT;
211         if (filp) {
212             info.align_mask = get_align_mask();
213             info.align_offset += get_align_bits();
214         }
215         addr = vm_unmapped_area(&info);
216         if (!(addr & ~PAGE_MASK))
217             return addr;
218         VM_BUG_ON(addr != -ENOMEM);
219
220     bottomup:
221         /*
222          * A failed mmap() very likely causes application failure,
223          * so fall back to the bottom-up function here. This scenario
224          * can happen with large stack limits and large mmap()
225          * allocations.
226          */
227         return arch_get_unmapped_area(filp, addr0, len, pgoff, flags);
228     }

```

Listing 5-22: x86-64-specific arch/x86/kernel/sys_x86_64.c: x86-64-specific arch_get_unmapped_area_topdown()

We start by performing a basic sanity check – if the length exceeds the entire permitted virtual address range, then this mapping is clearly invalid and we error out. This kind of check is largely useful for corrupted inputs, as a functioning user would never specify such a significantly incorrect parameter.

If the user has specified `MAP_FIXED`, there is no work to do, so we simply exit. Equally, if this is a mapping that is limited to 32-bits via `MAP_32BIT`, we force the legacy bottom-up approach.

We then handle the case of a hint being provided (i.e. the `mmap()` function being invoked with a non-NULL `addr` parameter, but with `MAP_FIXED` *not* having been specified). Firstly, we page-align the address as this is a strict requirement of any `mmap()` mapping.

Next, `mmap_address_hint_valid()` is called to ensure the hint is sensible – this checks to ensure the mapping would not exceed the maximum addressable virtual range and that a mapping does not straddle the 4-level and 5-level x86-64 mapping ranges as this could cause difficult bugs to arise.

Finally, we check to see if the hint would overlap a VMA (including any stack guard gap) – if it does we simply obtain the appropriate address using the unmapped area logic as if it were not specified. Otherwise, if no overlap, we use the (page-aligned) hint as supplied.

Ultimately, we specify a valid range – from a minimum of 1 page into the virtual address space (to permit NULL pointers to address 0 to function) to the base `mmap` address obtained by `get_mmap_base()` which simply retrieves the `struct mm_struct->mmap_base` value for the process address space.

This `mmap_base` parameter is determined by `arch_pick_mmap_layout()`, invoked by `setup_new_exec()` when a process is initially executed. The

`arch_pick_mmap_base()` function is called to obtain this base, which ultimately invokes `mmap_base()` as shown in Listing 5-23.

```

82 static unsigned long mmap_base(unsigned long rnd, unsigned long task_size,
83                               struct rlimit *rlim_stack)
84 {
85     unsigned long gap = rlim_stack->rlim_cur;
86     unsigned long pad = stack_maxrandom_size(task_size) + stack_guard_gap;
87     unsigned long gap_min, gap_max;
88
89     /* Values close to RLIM_INFINITY can overflow. */
90     if (gap + pad > gap)
91         gap += pad;
92
93     /*
94      * Top of mmap area (just below the process stack).
95      * Leave an at least ~128 MB hole with possible stack randomization.
96      */
97     gap_min = SIZE_128M;
98     gap_max = (task_size / 6) * 5;
99
100    if (gap < gap_min)
101        gap = gap_min;
102    else if (gap > gap_max)
103        gap = gap_max;
104
105    return PAGE_ALIGN(task_size - gap - rnd);
106 }

```

Listing 5-23: `arch/x86/mm/mmap.c: x86-64 mmap_base()`

This establishes a gap below the top of the maximum virtual address which will determine the maximum address at which we will assign unmapped space to a new mapping.

This is always clamped between 128 MiB and $\frac{5}{6}$ of the maximum virtual address space, in order to provide a sensible gap between the end of the mapping and the top of the available address space.

This base value is supplemented by a random value for security purposes for both the mapping itself (represented by `rnd`) and the stack which is offset by the maximum size this could be as determined by `stack_maxrandom_size()` (for 64-bit systems this is 1 GiB).

The final result is determined to be the combination of the maximum stack size and randomisation, the stack guard gap (a gap maintained between stacks and other mappings in order to ensure one cannot bump up against the other) and the random bytes up to the maximum amount of entropy as determined by `mmap_rnd_bits`, specified by `CONFIG_ARCH_MMAP_RAND_BITS` which for x86-64 defaults to 28 bits (256 MiB), but which can also be adjusted via the `vm.mmap_rnd_bits` tunable.

With this value set, an struct `vm_unmapped_area_info` object is populated with these bounds and alignment set according to architecture requirements, and `vm_unmapped_area()` is called to do the heavy lifting of determining where to place the mapping. For the top-down case (the only one we are considering here), `unmapped_area_topdown()` is ultimately called (eliding an align mask check that is not relevant to x86-64) as shown in Listing 5-24.

```

1966 static unsigned long unmapped_area_topdown(struct vm_unmapped_area_info *info)
1967 {
1968     struct mm_struct *mm = current->mm;
1969     struct vm_area_struct *vma;
1970     unsigned long length, low_limit, high_limit, gap_start, gap_end;
1971     :
1972     /*
1973      * Adjust search limits by the desired length.
1974      * See implementation comment at top of unmapped_area().
1975      */
1976     gap_end = info->high_limit;
1977     if (gap_end < length)
1978         return -ENOMEM;
1979     high_limit = gap_end - length;
1980
1981     if (info->low_limit > high_limit)
1982         return -ENOMEM;
1983     low_limit = info->low_limit + length;
1984
1985     /* Check highest gap, which does not precede any rbtree node */
1986     gap_start = mm->highest_vm_end;
1987     if (gap_start <= high_limit)
1988         goto found_highest;
1989
1990     /* Check if rbtree root looks promising */
1991     if (RB_EMPTY_ROOT(&mm->mm_rb))
1992         return -ENOMEM;
1993     vma = rb_entry(mm->mm_rb.rb_node, struct vm_area_struct, vm_rb);
1994     if (vma->rb_subtree_gap < length)
1995         return -ENOMEM;
1996
1997     while (true) {
1998         /* Visit right subtree if it looks promising */
1999         gap_start = vma->vm_prev ? vm_end_gap(vma->vm_prev) : 0;
2000         if (gap_start <= high_limit && vma->vm_rb.rb_right) {
2001             struct vm_area_struct *right =
2002                 rb_entry(vma->vm_rb.rb_right,
2003                     struct vm_area_struct, vm_rb);
2004             if (right->rb_subtree_gap >= length) {
2005                 vma = right;
2006                 continue;

```

```

2012         }
2013     }
2014
2015     check_current:
2016         /* Check if current node has a suitable gap */
2017         gap_end = vm_start_gap(vma);
2018         if (gap_end < low_limit)
2019             return -ENOMEM;
2020         if (gap_start <= high_limit &&
2021             gap_end > gap_start && gap_end - gap_start >= length)
2022             goto found;
2023
2024         /* Visit left subtree if it looks promising */
2025         if (vma->vm_rb.rb_left) {
2026             struct vm_area_struct *left =
2027                 rb_entry(vma->vm_rb.rb_left,
2028                     struct vm_area_struct, vm_rb);
2029             if (left->rb_subtree_gap >= length) {
2030                 vma = left;
2031                 continue;
2032             }
2033         }
2034
2035         /* Go back up the rbtrees to find next candidate node */
2036         while (true) {
2037             struct rb_node *prev = &vma->vm_rb;
2038             if (!rb_parent(prev))
2039                 return -ENOMEM;
2040             vma = rb_entry(rb_parent(prev),
2041                 struct vm_area_struct, vm_rb);
2042             if (prev == vma->vm_rb.rb_right) {
2043                 gap_start = vma->vm_prev ?
2044                     vm_end_gap(vma->vm_prev) : 0;
2045                 goto check_current;
2046             }
2047         }
2048     }
2049
2050     found:
2051         /* We found a suitable gap. Clip it with the original high_limit. */
2052         if (gap_end > info->high_limit)
2053             gap_end = info->high_limit;
2054
2055     found_highest:
2056         /* Compute highest gap address at the desired alignment */
2057         gap_end -= info->length;
2058     :

```

```

2060     VM_BUG_ON(gap_end < info->low_limit);
2061     VM_BUG_ON(gap_end < gap_start);
2062     return gap_end;
2063 }

```

Listing 5-24: mm/mmap.c: unmapped_area_topdown()

Firstly it's worth examining the comment referenced by the code from the bottom-up version of the function, `unmapped_area()` as shown in Listing 5-25.

```

1865     /*
1866      * We implement the search by looking for an rbtree node that
1867      * immediately follows a suitable gap. That is,
1868      * - gap_start = vma->vm_prev->vm_end <= info->high_limit - length;
1869      * - gap_end   = vma->vm_start   >= info->low_limit + length;
1870      * - gap_end - gap_start >= length
1871      */

```

Listing 5-25: mm/mmap.c: unmapped_area() comment

So it's important to remember that rather than looking for an *address* which has sufficient space for the VMA we wish to insert, we are instead looking for a node with a large enough *gap* to place it.

We therefore start by establishing `high_limit` as being equal to the maximum address (`gap_end`) less the length of the mapping and `low_limit` as being equal to the minimum address (`info->low_limit`) plus the length of the mapping.

We use the cached value `struct mm_struct->highest_vm_end` to see if we can simply slot in at the end of mapped values, if so we can proceed to the `found_highest` label (note that this value takes into account the stack guard gap determined by `vm_end_gap()`).

If this check fails, then this means there is not sufficient space at the end of the address space, and therefore we must find a place to slot in *behind* another VMA.

We therefore have to traverse the tree to find where to place the new mapping. In each VMA we traverse, we rely entirely on the metadata we store in each VMA in the `struct vm_area_struct->rb_subtree_gap` field. This maintains a track of the largest gap either between this VMA and the VMA previous or any of the VMAs below it in the tree.

Throughout the traversal, we maintain `gap_start` and `gap_end` as the bounds of each gap we have found, adjusting them as we go. We offset these using `vm_start_gap()` and `vm_end_gap()` (on the VMA prior to the one under examination) which obtain the start/end of the VMA respectively, offsetting by the stack guard gap if the VMAs are stacks, being careful about overflow.

We start by ensuring the root red/black tree node is valid and that at least one node in the tree has a sufficient gap (by examining the root's `rb_subtree_gap` we can determine this immediately).

As we are looking for the largest possible value, we start by always traversing the right node (as a balanced search tree, this will always result in VMAs

at higher addresses) if there is sufficient subtree gap until we reach a node without any right sub tree, where the `check_current` label is declared.

If `gap_end` is below the lower limit, this means that the gap before the VMA we are examining is insufficient to contain the candidate mapping so we must error out. Otherwise, we check to see whether there is sufficient gap before this VMA.

Note that we are looking to place an entry *before* the highest node (since otherwise we would be placing the VMA at the end of the process address space and we have already checked for this), so we establish `gap_start` and `gap_end` to be equal to the (stack guard adjusted) end of the previous VMA and the (also stack guard adjusted) beginning of this one – if this can fit the VMA then we proceed to the `found` label.

Otherwise, we must traverse any present left nodes, and loop around again – when we resume the loop, we will prefer to traverse right nodes at each instance of left node we have traversed so will always trend towards the VMA at the maximum address with sufficient gap for the mapping.

If the current VMA under examination is a leaf node in the tree and has an insufficient gap to store the VMA, we traverse back up the tree, looking for the first instance of a parent VMA for which this node was greater (i.e. on the right of the tree), then proceeding to examine the left hand nodes of this one. If this fails to find a parent, then the search fails and we error out.

When we finally find a sufficient gap, we use `gap_end` to store the end of the gap (taking into account the stack guard gap), clamped to the original maximum limit specified, less the length of the mapping, and return this value.

For the purposes of illustrating this process, let's examine a tree which ultimately does not contain a sufficient gap and the order in which nodes will be traversed as shown in Figure 5-2.

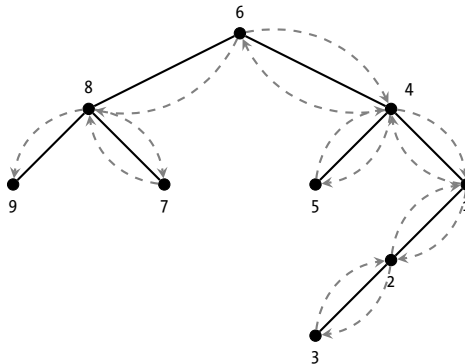


Figure 5-2: VMA tree gap traversal example

As you can see, the algorithm implemented in `unmapped_area_topdown()` results in a traversal from ‘rightmost’ node (i.e. the one with the largest address) to the leftmost (i.e. the one with the smallest address).

5.0.6 Unmapping memory

Unmapping memory allocated by `mmap()` (or otherwise) can be performed by `munmap()` which accepts a virtual address and length, over which range all VMAs will be removed and reference count dropped on all underlying pages. This is implemented in the `munmap` system call as shown in Listing 5-26.

```

2881 SYSCALL_DEFINE2(munmap, unsigned long, addr, size_t, len)
2882 {
2883     addr = untagged_addr(addr);
2884     return __vm_munmap(addr, len, true);
2885 }

```

Listing 5-26: mm/mmap.c: `munmap()` system call

This defers the unmapping operation to `__vm_munmap()`* which acquires a write lock on `struct mm_struct->mmap_lock` before invoking `__do_munmap()` to do the heavy lifting (eliding out of scope `userfaultfd` logic) as shown in Listing 5-27.

```

2850 static int __vm_munmap(unsigned long start, size_t len, bool downgrade)
2851 {
2852     int ret;
2853     struct mm_struct *mm = current->mm;
2854     LIST_HEAD(uf);
2855
2856     if (mmap_write_lock_killable(mm))
2857         return -EINTR;
2858
2859     ret = __do_munmap(mm, start, len, &uf, downgrade);
2860     /*
2861      * Returning 1 indicates mmap_lock is downgraded.
2862      * But 1 is not legal return value of vm_munmap() and munmap(), reset
2863      * it to 0 before return.
2864      */
2865     if (ret == 1) {
2866         mmap_read_unlock(mm);
2867         ret = 0;
2868     } else
2869         mmap_write_unlock(mm);
2870     :
2872     return ret;
2873 }

```

Listing 5-27: mm/mmap.c: `__vm_munmap()`

Examining `__do_munmap()` as shown in Listing 5-28.

*. The `untagged_addr()` invocation is only relevant to specific architectures which tag virtual addresses for specific purposes and is out of scope here.

```

2754 int __do_munmap(struct mm_struct *mm, unsigned long start, size_t len,
2755                 struct list_head *uf, bool downgrade)
2756 {
2757     unsigned long end;
2758     struct vm_area_struct *vma, *prev, *last;
2759
2760     if ((offset_in_page(start)) || start > TASK_SIZE || len > TASK_SIZE -
start)
2761         return -EINVAL;
2762
2763     len = PAGE_ALIGN(len);
2764     end = start + len;
2765     if (len == 0)
2766         return -EINVAL;
2767
2768     /*
2769      * arch_unmap() might do unmaps itself. It must be called
2770      * and finish any rbtree manipulation before this code
2771      * runs and also starts to manipulate the rbtree.
2772      */
2773     arch_unmap(mm, start, end);
2774
2775     /* Find the first overlapping VMA where start < vma->vm_end */
2776     vma = find_vma_intersection(mm, start, end);
2777     if (!vma)
2778         return 0;
2779     prev = vma->vm_prev;
2780
2781     /*
2782      * If we need to split any vma, do it now to save pain later.
2783      *
2784      * Note: mremap's move_vma VM_ACCOUNT handling assumes a partially
2785      * unmapped vm_area_struct will remain in use: so lower split_vma
2786      * places tmp vma above, and higher split_vma places tmp vma below.
2787      */
2788     if (start > vma->vm_start) {
2789         int error;
2790
2791         /*
2792          * Make sure that map_count on return from munmap() will
2793          * not exceed its limit; but let map_count go just above
2794          * its limit temporarily, to help free resources as expected.
2795          */
2796         if (end < vma->vm_end && mm->map_count >= sysctl_max_map_count
)
2797             return -ENOMEM;
2798

```

```

2799         error = __split_vma(mm, vma, start, 0);
2800         if (error)
2801             return error;
2802         prev = vma;
2803     }
2804
2805     /* Does it split the last one? */
2806     last = find_vma(mm, end);
2807     if (last && end > last->vm_start) {
2808         int error = __split_vma(mm, last, end, 1);
2809         if (error)
2810             return error;
2811     }
2812     vma = vma_next(mm, prev);
2813     :
2829     /* Detach vmas from rbtree */
2830     if (!detach_vmas_to_be_unmapped(mm, vma, prev, end))
2831         downgrade = false;
2832
2833     if (downgrade)
2834         mmap_write_downgrade(mm);
2835
2836     unmap_region(mm, vma, prev, start, end);
2837
2838     /* Fix up all other VM information */
2839     remove_vma_list(mm, vma);
2840
2841     return downgrade ? 1 : 0;
2842 }

```

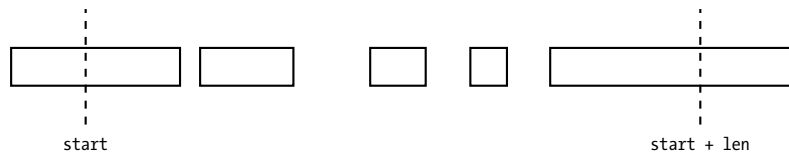
Listing 5-28: mm/mmap.c: `__do_munmap()`

We start by checking that the start value is both page-aligned (via `offset_in_page()`) and a valid virtual address (via comparison to `TASK_SIZE`), erroring out if invalid, before checking to ensure no overflow occurs on summing start and len.

Some architectures might hook the unmap operation via `arch_unmap()`, however this is currently only meaning for powerpc.

We identify the first struct `vm_area_struct` (VMA) which sits within the designated range via `find_vma_intersection()` (see 4.4.5 for more on VMA traversal), exiting early if we fail to do so.

While we free memory in the range, we have provided the user with a free hand as to the address range they've specified – there might be multiple VMAs contained within the range, all of which must be freed, and in addition the range might *overlap* VMAs as shown in Figure 5-3.

Figure 5-3: `munmap()` VMA cases

We deal with the overlapping VMAs using `__split_vma()` (see section 5.1.4 for a detailed exploration of VMA splitting), being careful to check to ensure splitting this way does not result in us exceeding the `vm.max_map_count` tunable if the specified range contains only one VMA, which we are now splitting (this net, incrementing the number of mapped VMAs by one).

Now we have split any overlapping VMAs, we must isolate the VMAs in the range in order to iterate through them and remove each of them which we do via `detach_vmas_to_be_unmapped()` (described below). If this returns true, this indicates that we no longer require a write lock held against the `struct mm_struct->mmap_lock` semaphore, so we downgrade to a read lock.

Finally, we perform TLB invalidation and reference count deduction and possible freeing of underlying physical pages via `unmap_region()` (see section 7.1.1 for an in depth description of this process) before invoking `remove_vma_list()` to update statistics and accounted memory (which we will examine shortly).

Examining `detach_vmas_to_be_unmapped()` as shown in Listing 5-29.

```

2628 /*
2629  * Create a list of vma's touched by the unmap, removing them from the mm's
2630  * vma list as we go..
2631  */
2632 static bool
2633 detach_vmas_to_be_unmapped(struct mm_struct *mm, struct vm_area_struct *vma,
2634                          struct vm_area_struct *prev, unsigned long end)
2635 {
2636     struct vm_area_struct **insertion_point;
2637     struct vm_area_struct *tail_vma = NULL;
2638
2639     insertion_point = (prev ? &prev->vm_next : &mm->mmap);
2640     vma->vm_prev = NULL;
2641     do {
2642         vma_rb_erase(vma, &mm->mm_rb);
2643         if (vma->vm_flags & VM_LOCKED)
2644             mm->locked_vm -= vma_pages(vma);
2645         mm->map_count--;
2646         tail_vma = vma;
2647         vma = vma->vm_next;
2648     } while (vma && vma->vm_start < end);
2649     *insertion_point = vma;
2650     if (vma) {

```

```

2651         vma->vm_prev = prev;
2652         vma_gap_update(vma);
2653     } else
2654         mm->highest_vm_end = prev ? vm_end_gap(prev) : 0;
2655     tail_vma->vm_next = NULL;
2656
2657     /* Kill the cache */
2658     vmacache_invalidate(mm);
2659
2660     /*
2661      * Do not downgrade mmap_lock if we are next to VM_GROWSDOWN or
2662      * VM_GROWSUP VMA. Such VMAs can change their size under
2663      * down_read(mmap_lock) and collide with the VMA we are about to unmap
2664      */
2665     if (vma && (vma->vm_flags & VM_GROWSDOWN))
2666         return false;
2667     if (prev && (prev->vm_flags & VM_GROWSUP))
2668         return false;
2669     return true;
2670 }

```

Listing 5-29: mm/mmap.c: detach_vmas_to_be_unmapped()

We start by determining the ‘insertion point’, i.e. the pointer which we must update to reflect the removal of these VMAs. If the range being remove does not start at the very beginning of the process address space, this will be a pointer to the previous VMA’s the struct `vm_area_struct->vm_next` field, otherwise it is a pointer to the struct `mm_struct->mmap` field which which is the beginning of the list of VMAs in the address space.

Next we start the process of detaching the VMAs in the range by setting the first VMA’s `vm_prev` to `NULL` and keeping track of the last VMA in the range via `tail_vma` in order to eventually set its `vm_next` field to `NULL` also.

We then loop through each VMA in the range, removing it from the process address space’s red/black tree via `vma_rb_erase()` (see section 4.4.4 for more on this), and decrementing the `map_count` of the process address space.

When this is complete, we place the VMA immediately proceeding the removed range in the insertion point, update the gap metadata associated with the red/black tree and isolate the tail VMA from the rest of the tree.

Now we have in removed the VMAs from both the red/black tree and the VMA linked list, they effectively no longer exist. We therefore invalidate the VMA cache via `vmacache_invalidate()` (see section 4.4.5.1 for more on the VMA cache).

Finally, we determine whether we can downgrade from a write to a read lock on the struct `mm_struct->mmap_lock` semaphore – if the immediately proceeding VMA is a stack that grows down the preceding one is a stack which grows up, we do not downgrade as stacks can alter their `vm_start` (if a downwards growing stack) or `vm_end` (if an upward growing stack) which could

cause inadvertent accesses to TLB cache entries not yet invalidated for the removed range.

Finally, examining `remove_vma_list()` as shown in Listing 5-30.

```

2583  /*
2584  * Ok - we have the memory areas we should free on the vma list,
2585  * so release them, and do the vma updates.
2586  *
2587  * Called with the mm semaphore held.
2588  */
2589  static void remove_vma_list(struct mm_struct *mm, struct vm_area_struct *vma)
2590  {
2591      unsigned long nr_accounted = 0;
2592
2593      /* Update high watermark before we lower total_vm */
2594      update_hiwater_vm(mm);
2595      do {
2596          long nrpages = vma_pages(vma);
2597
2598          if (vma->vm_flags & VM_ACCOUNT)
2599              nr_accounted += nrpages;
2600          vm_stat_account(mm, vma->vm_flags, -nrpages);
2601          vma = remove_vma(vma);
2602      } while (vma);
2603      vm_unacct_memory(nr_accounted);
2604      validate_mm(mm);
2605  }

```

Listing 5-30: `mm/mmap.c: remove_vma_list()`

We start by updating the `struct mm_struct->hiwater_vm` statistic (as we are about to reduce `total_vm` so we ensure that keep this consistent) via `update_hiwater_vm()`.

Then, for each VMA to be removed, we update the `mm_struct`'s `exec_vm`, `stack_vm` and `data_vm` statistics via `vm_stat_account()`, counting the number of accounted pages for VMAs which are, in fact accounted. Finally we close and free each `struct vm_area_struct` object via `remove_vma()` (which we will examine shortly).

Once each VMA is processed we update per-CPU Committed AS statistics via `vm_unacct_memory()` and, if `CONFIG_DEBUG_VM_RB` is specified, validate the process address space via `validate_mm()`.

Examining `remove_vma()` as shown in Listing 5-31.

```

134  /*
135  * Close a vm structure and free it, returning the next.
136  */
137  static struct vm_area_struct *remove_vma(struct vm_area_struct *vma)
138  {
139      struct vm_area_struct *next = vma->vm_next;

```

```

140
141     might_sleep();
142     if (vma->vm_ops && vma->vm_ops->close)
143         vma->vm_ops->close(vma);
144     if (vma->vm_file)
145         fput(vma->vm_file);
146     mpol_put(vma_policy(vma));
147     vm_area_free(vma);
148     return next;
149 }

```

Listing 5-31: mm/mmap.c: *remove_vma()*

If the VMA is file-backed and the file system has specified a close function to be invoked when a VMA is removed, this is called. Equally, if a struct file object is associated with this VMA, its reference count is reduced, as is that of any associated NUMA memory policy, before the VMA itself is freed via `vm_area_free()` (see listing 5-45).

5.1 VMA merge and split

5.1.1 VMA merge

Virtual regions of userland memory, represented by struct `vm_area_struct` (VMAs), are separated both by their location in memory and their attributes (as determined by `can_vma_merge_before()` and `can_vma_merge_after()` - both of which will examine shortly). When VMAs are mapped/remapped via `mmap()`, `mremap()` or `brk()` system calls or when their attributes are changed by e.g. `mprotect()` we may be able to merge them. This merge operation is performed by `vma_merge()` (with a write lock on the struct `mm_struct->mmap_lock` semaphore held), whose *primary** parameters are:

- `mm` – The `mm_struct` associated with the process.
- `prev` – The VMA immediately preceding the virtual range to be merged (or NULL if the inserted VMA will be the first VMA in the address space).
- `addr` – The first address of either the VMA to be inserted in the address space or the range whose attributes are being altered.
- `end` – The exclusive upper bound of this range (i.e. the first virtual address which sits *above* the range but is not a part of it).
- `vm_flags` – The VMA flags of the region being merged.

There are eight possible cases in which a merge can occur (where `prev`, `area` and `next` are the previous, adjacent and next VMAs respectively):

1. *New mapping, merge both* – `prev->vm_end` is equal to `addr`, `end` is equal to `next->vm_start` and both are mergeable.

*. There are further parameters which we will examine when we look at the function in detail.

2. *New mapping, merge previous* – `prev->vm_end` is equal to `addr` and `prev` is mergeable.
3. *New mapping, merge next* – `end` is equal to `next->vm_start` and `next` is mergeable.
4. *Adjust mapping, merge next, shrink previous* – `addr` is equal to `area->vm_start`, `end` is equal to both `prev->vm_end` and `next->vm_start` but only `next` is mergeable.
5. *Adjust mapping, merge previous, shrink current* – `addr` is equal to both `area->vm_start` and `prev->vm_end`, but only `prev` is mergeable.
6. *Adjust mapping, merge both* – `addr` is equal to both `area->vm_start` and `prev->vm_end`, `end` is equal to `next->vm_start` and both `prev` and `next` are mergeable.
7. *Adjust mapping, merge previous* – `addr` is equal to both `area->vm_start` and `prev->vm_end`, `end` is equal to `area->vm_end` and only `prev` is mergeable.
8. *Adjust mapping, merge next* – `addr` is equal to `area->vm_start`, `end` is equal to `next->vm_start` and only `next` is mergeable.

We visualise this in Figure 5-4.

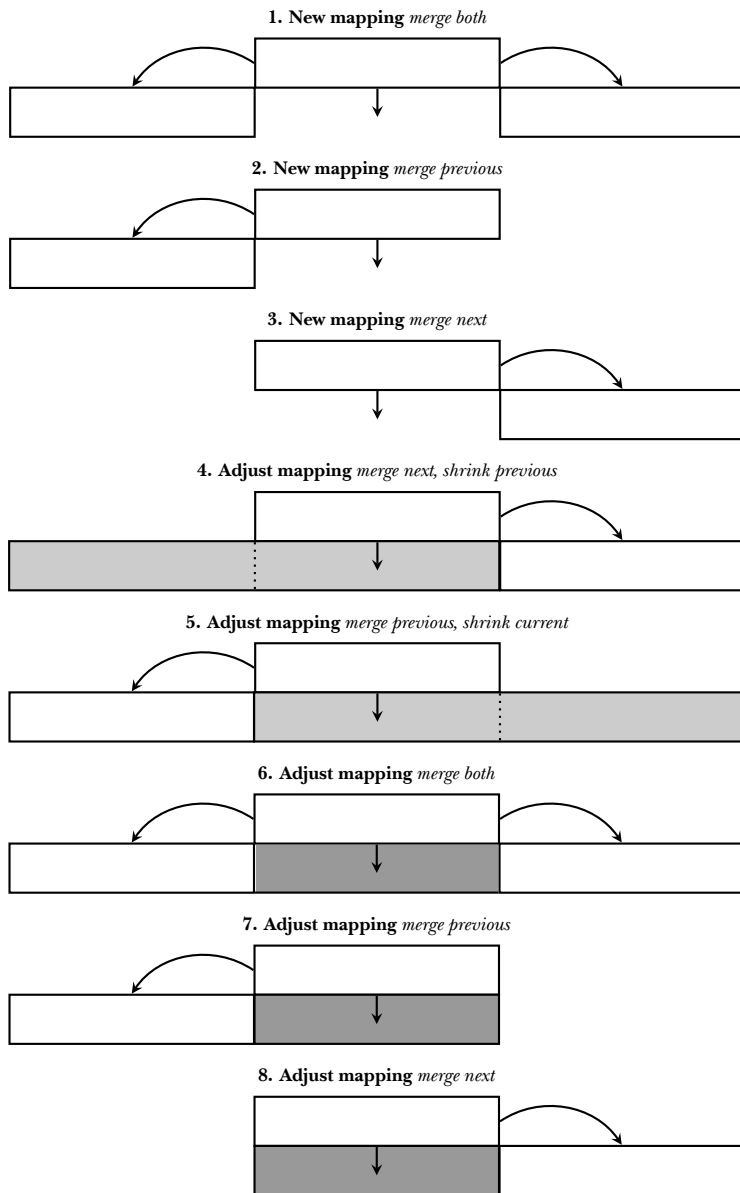


Figure 5-4: `vma_merge()` merge cases

These cases are each explicitly referenced in the code numbered as above. Note that the input range simply specifies the range over which we try to either expand existing VMAs to cover, either spanning an empty memory ‘hole’ (cases 1 - 3) or after adjusting the attributes of an existing VMA (cases 4-8).

VMA merging is performed by `vma_merge()`. Eliding out of scope user-faultfd, it is invoked from:

- `mmap_region()` – When memory is first mapped via `mmap()` then we attempt to expand existing VMAs to cover the mapped range. This saves us having to allocate a new VMA and maintains the invariant that immediately adjacent VMAs must possess different attributes (cases 1 - 3).
- `do_brk_flags()` – Similar to the `mmap()` case, we try to merge the mapped range with existing VMAs rather than allocate a new one if possible (cases 1 - 3).
- `copy_vma()` – Used by `remap()` to copy a VMA to a new location. We determine whether the new VMA can be merged in the location to which it has been relocated (cases 1 - 3).
- `mprotect_fixup()` – Used by `mprotect()` to change the protection flags of a portion of a VMA. It attempts to merge the newly adjusted memory range with those adjacent, if it cannot then the existing VMA will be split (cases 4 - 8).
- `madvise_update_vma()` – Invoked by `madvise()` in cases where the VMA flags might have changed, in which case an attempt must be made to merge the existing VMA with those adjacent, if it cannot then the existing VMA will be split (cases 4 - 8).
- `mlock_fixup()` – When the `mlock()` system call might result in the `VM_LOCKED` flag either being set or cleared (either via `apply_vma_lock_flags()` or `apply_mlockall_flags()`), we must attempt to merge the adjusted range with adjacent VMAs, or otherwise split the adjusted region (cases 4 - 8).
- `mbind_range()` – Part of what determines the attributes of a VMA is its NUMA memory policy (for more on NUMA as a whole, see the NUMA chapter). Given this is the case, when the `mbind()` system call is invoked we must, similar to the other VMA adjustment cases, check whether we can merge the adjusted region with adjacent VMAs, or otherwise we must split it (cases 4 - 8).

An important subtlety to note here for functions which adjust a mapping (i.e. specify an address range which spans an existing VMA) is that the input address (`addr`) will either be equal to `va_start` of a VMA (which will be designated area), in which case `prev` will be the VMA prior to the spanned one, or if offset, `prev` will specify the VMA which is spanned and can be merged in case 4 only.

Now we have examined each of the merge cases, let's examine the `vma_merge()` function itself as shown in Listing 5-32.

```

1122 struct vm_area_struct *vma_merge(struct mm_struct *mm,
1123                                struct vm_area_struct *prev, unsigned long addr,
1124                                unsigned long end, unsigned long vm_flags,
1125                                struct anon_vma *anon_vma, struct file *file,
1126                                pgoff_t pgoff, struct mempolicy *policy,
1127                                struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
1128                                struct anon_vma_name *anon_name)
1129 {

```

```

1130     pgoff_t pglen = (end - addr) >> PAGE_SHIFT;
1131     struct vm_area_struct *area, *next;
1132     int err;
1133
1134     /*
1135      * We later require that vma->vm_flags == vm_flags,
1136      * so this tests vma->vm_flags & VM_SPECIAL, too.
1137      */
1138     if (vm_flags & VM_SPECIAL)
1139         return NULL;
1140
1141     next = vma_next(mm, prev);
1142     area = next;
1143     if (area && area->vm_end == end)                /* cases 6, 7, 8 */
1144         next = next->vm_next;
1145
1146     /* verify some invariant that must be enforced by the caller */
1147     VM_WARN_ON(prev && addr <= prev->vm_start);
1148     VM_WARN_ON(area && end > area->vm_end);
1149     VM_WARN_ON(addr >= end);
1150
1151     /*
1152      * Can it merge with the predecessor?
1153      */
1154     if (prev && prev->vm_end == addr &&
1155         mpol_equal(vma_policy(prev), policy) &&
1156         can_vma_merge_after(prev, vm_flags,
1157                             anon_vma, file, pgoff,
1158                             vm_userfaultfd_ctx, anon_name)) {
1159         /*
1160          * OK, it can. Can we now merge in the successor as well?
1161          */
1162         if (next && end == next->vm_start &&
1163             mpol_equal(policy, vma_policy(next)) &&
1164             can_vma_merge_before(next, vm_flags,
1165                                 anon_vma, file,
1166                                 pgoff+pglen,
1167                                 vm_userfaultfd_ctx,
1168                                 anon_name) &&
1169             is_mergeable_anon_vma(prev->anon_vma,
1170                                   next->anon_vma, NULL)) {
1171             /* cases 1, 6 */
1172             err = __vma_adjust(prev, prev->vm_start,
1173                               next->vm_end, prev->vm_pgoff, NULL,
1174                               prev);
1175         } else                                /* cases 2, 5, 7 */
1176             err = __vma_adjust(prev, prev->vm_start,

```

```

1176                                     end, prev->vm_pgoff, NULL, prev);
1177             if (err)
1178                 return NULL;
1179             khugepaged_enter_vma(prev, vm_flags);
1180             return prev;
1181     }

```

Listing 5-32: mm/mmap.c: vma_merge() preface and predecessor merge

Note that the callers of this function will hold a `struct mm_struct->mmap_lock` write lock. The callers of this function guarantee that the address range specified by `addr` and `end` will fulfil one of the following criteria:

- `addr - end` will span a hole, i.e. unmapped memory (if mergeable, this will be one of cases 1 - 3).
- `addr - end` will span a portion of `prev` (if `end` is equal to `prev->end` then case 4 *may* apply if another VMA is immediately adjacent).
- `addr` will be equal to `area->vm_start` in all other cases (if mergeable then this will be one of cases 5 - 8).

We obtain the VMA after `prev` using `vma_next()`. If `prev` is `NULL` (meaning that the input memory range spans either the first VMA in the address space or the memory hole prior to the first VMA) then it simply obtains the first VMA via `mm_struct->mmap`.

We then assign `area` to this VMA, updating `next` to point at the VMA after it only if the entire `area` VMA is spanned by `addr` and `end` (cases 6 - 8).

Finally before checking mergeability, we assert some fundamentals – if the input `addr` starts prior to `prev`, ends after `area->vm_end` or nonsensically `addr` and `end` do not represent a valid range of memory then something has gone horribly wrong.

We start by checking whether we can merge with `prev` – this covers cases 1, 2, 5 - 7. The general mergeability of `prev` is determined by `mpol_equal()` which determines whether NUMA memory policy is equal (see the NUMA chapter for more on memory policies), deferring all other mergeability checks to `can_vma_merge_after()`.

We then need to differentiate between cases 1 and 6 (where both `prev` and `next` are merged) and all others by checking whether `next` is mergeable in the same fashion – checking memory policy and mergeability, only this time via `can_vma_merge_before()`.

One difference is that `struct anon_vma` mergeability is checked between `prev` and `next` via `is_mergeable_anon_vma()`. We will examine this function in detail later (and you can read more about `anon_vma` objects and the reverse mapping in Chapter 7), but this is the only criteria that isn't guaranteed to be compatible between `prev` and `next` when both have been determined to be mergeable with the input address range.

Finally, the heavy lifting of the merge is deferred to `__vma_adjust()`. We will examine this function shortly as well as how it is parameterised.

If an error occurs when adjusting, we return NULL to indicate no merge was possible otherwise, since we will have folded the address range into prev we return this as the newly merged VMA.

Let's examine the cases where we cannot merge with prev as shown in Listing 5-33.

```

1183      /*
1184      * Can this new request be merged in front of next?
1185      */
1186      if (next && end == next->vm_start &&
1187          mpol_equal(policy, vma_policy(next)) &&
1188          can_vma_merge_before(next, vm_flags,
1189                              anon_vma, file, pgoff+pglen,
1190                              vm_userfaultfd_ctx, anon_name)) {
1191          if (prev && addr < prev->vm_end)          /* case 4 */
1192              err = __vma_adjust(prev, prev->vm_start,
1193                                addr, prev->vm_pgoff, NULL, next);
1194          else {                                     /* cases 3, 8 */
1195              err = __vma_adjust(area, addr, next->vm_end,
1196                                next->vm_pgoff - pglen, NULL, next);
1197          /*
1198           * In case 3 area is already equal to next and
1199           * this is a noop, but in case 8 "area" has
1200           * been removed and next was expanded over it.
1201           */
1202          area = next;
1203      }
1204      if (err)
1205          return NULL;
1206      khugepaged_enter_vma(area, vm_flags);
1207      return area;
1208  }
1209
1210      return NULL;
1211  }

```

Listing 5-33: mm/mmap.c: *vma_merge()* successor merge only

We start by checking whether we can merge with next at all as above (only without performing an additional `is_mergeable_anon_vma()` check) – if we cannot then no merge is possible at all and we indicate this by returning NULL.

Otherwise, we check to see whether the address range spans the end of prev (case 4), in which case we return area which will have been merged with next and comprises the newly merged VMA carved out of the end of prev and the entirety of next.

Otherwise, in case 3 we will already have assigned area to next and in case 8 we will eliminate area and adjust next to envelop it so in both instances we return next.

In both cases we again perform the heavy lifting in `__vma_adjust()`, which we examine in section 5.1.3.

5.1.2 Mergeability

Once we have established that VMAs are adjacent, we determine whether a VMA is mergeable with another via `can_vma_merge_after()` when checking whether a candidate VMA can be merged after `prev`, or `can_vma_merge_before()` when checking whether a candidate VMA can be merged before `next` as shown in Listing 5-34.

```

1055 /*
1056  * Return true if we can merge this (vm_flags,anon_vma,file,vm_pgoff)
1057  * beyond (at a higher virtual address and file offset than) the vma.
1058  *
1059  * We cannot merge two vmAs if they have differently assigned (non-NULL)
1060  * anon_vmas, nor if same anon_vma is assigned but offsets incompatible.
1061  */
1062 static int
1063 can_vma_merge_after(struct vm_area_struct *vma, unsigned long vm_flags,
1064                    struct anon_vma *anon_vma, struct file *file,
1065                    pgoff_t vm_pgoff,
1066                    struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
1067                    struct anon_vma_name *anon_name)
1068 {
1069     if (is_mergeable_vma(vma, file, vm_flags, vm_userfaultfd_ctx,
1070                          anon_name) &&
1071         is_mergeable_anon_vma(anon_vma, vma->anon_vma, vma)) {
1072         pgoff_t vm_pglen;
1073         vm_pglen = vma_pages(vma);
1074         if (vma->vm_pgoff + vm_pglen == vm_pgoff)
1075             return 1;
1076     }
1077     return 0;
1078 }
```

Listing 5-34: `mm/mmap.c: can_vma_merge_after()`

This defers the heavy lifting to the helper functions `is_mergeable_vma()` and `is_mergeable_anon_vma()` which we will examine shortly.

Other than these checks, we perform one final one – the virtual page offset of the candidate VMA must be equal to that of the previous one (passed as `vma`).

`can_vma_merge_before()` is similar but with a simplified `vm_pgoff` check as we need only check equality as shown in Listing 5-35.

```

1029 /*
1030  * Return true if we can merge this (vm_flags,anon_vma,file,vm_pgoff)
1031  * in front of (at a lower virtual address and file offset than) the vma.
```

```

1032  *
1033  * We cannot merge two vmAs if they have differently assigned (non-NULL)
1034  * anon_vmas, nor if same anon_vma is assigned but offsets incompatible.
1035  *
1036  * We don't check here for the merged mmap wrapping around the end of
        pagecache
1037  * indices (16TB on ia32) because do_mmap() does not permit mmap's which
1038  * wrap, nor mmAs which cover the final page at index -1UL.
1039  */
1040  static int
1041  can_vma_merge_before(struct vm_area_struct *vma, unsigned long vm_flags,
1042                      struct anon_vma *anon_vma, struct file *file,
1043                      pgoff_t vm_pgoff,
1044                      struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
1045                      struct anon_vma_name *anon_name)
1046  {
1047      if (is_mergeable_vma(vma, file, vm_flags, vm_userfaultfd_ctx,
        anon_name) &&
1048          is_mergeable_anon_vma(anon_vma, vma->anon_vma, vma)) {
1049          if (vma->vm_pgoff == vm_pgoff)
1050              return 1;
1051      }
1052      return 0;
1053  }

```

Listing 5-35: mm/mmap.c: `can_vma_merge_before()`

The mergeability of a VMA is fundamentally determined by the predicate function `is_mergeable_vma()` as shown in Listing 5-36.

```

985  /*
986  * If the vma has a ->close operation then the driver probably needs to
        release
987  * per-vma resources, so we don't attempt to merge those.
988  */
989  static inline int is_mergeable_vma(struct vm_area_struct *vma,
990                                   struct file *file, unsigned long vm_flags,
991                                   struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
992                                   struct anon_vma_name *anon_name)
993  {
994      /*
995       * VM_SOFTDIRTY should not prevent from VMA merging, if we
996       * match the flags but dirty bit -- the caller should mark
997       * merged VMA as dirty. If dirty bit won't be excluded from
998       * comparison, we increase pressure on the memory system forcing
999       * the kernel to generate new VMAs when old one could be
1000       * extended instead.
1001       */
1002      if ((vma->vm_flags ^ vm_flags) & ~VM_SOFTDIRTY)

```

```

1003         return 0;
1004     if (vma->vm_file != file)
1005         return 0;
1006     if (vma->vm_ops && vma->vm_ops->close)
1007         return 0;
1008     if (!is_mergeable_vm_userfaultfd_ctx(vma, vm_userfaultfd_ctx))
1009         return 0;
1010     if (!anon_vma_name_eq(anon_vma_name(vma), anon_name))
1011         return 0;
1012     return 1;
1013 }

```

Listing 5-36: mm/mmap.c: *is_mergeable_vma()*

Note that this check is essentially determining whether two VMAs are effectively *identical*. We maintain this state in each VMA so that we can describe virtual memory ranges using as little space as possible, which is why we merge identical VMAs in the first instance.

Examining each of these criteria (other than out of scope userfaultfd handling):

1. `vm_flags` must be equal, excluding `VM_SOFTDIRTY` which, as the comment describes, is not worth the added memory pressure of treating it as the basis for otherwise identical VMAs being separate from one another. For example, when initially mapping memory in `mmap_region()` or `do_brk_flags()` the `VM_SOFTDIRTY` flag is set unconditionally, so the flag is maintained correctly.
2. `vm_file` – If file-backed, the VMAs must be backed by the same file.
3. `vm_ops->close` – If VMA operations are provided (i.e. this is a non-anonymous mapping), the existence of a `close()` handler implies that, as per the comment, the driver implementing this probably has distinct per-VMA state, so merging is not safe.
This function, if specified, is invoked in `remove_vma()` (which is ultimately invoked on `unmap` or when the process memory is torn down).
4. `anon_vma_name` – Anonymous VMAs can be specifically named when the region serves a specific purpose, e.g. `stack`, `heap` (shown in square brackets in `/proc/$pid/maps` and `/proc/$pid/smmaps`). Naturally, anonymous VMAs are not mergeable with one another if the names differ.

Finally, we have specific handling to ensure that reverse-mapping `struct anon_vma` objects (see Chapter 7) associated with each VMA are equivalent, via `is_mergeable_anon_vma()` (see listing 7-19). This is discussed in detail in the reverse mapping section 7.0.11.

5.1.3 VMA adjust

The `_vma_adjust()` function is parameterised as shown in Table 5-1.

Table 5-1: `__vma_adjust()` parameters

Parameter	Case							
	1	2	3	4	5	6	7	8
vma	prev		next	prev				area
start	prev->vm_start		addr	prev->vm_start				addr
end	next->vm_end	end	next->vm_end	addr	end	next->vm_end	area->vm_end	next->vm_end
pgoff	prev->vm_pgoff		*	prev->vm_pgoff				*
insert	NULL							
expand	prev		next		prev			next

* `next->vm_pgoff - (end - addr) >> PAGE_SHIFT`

Examining each parameter:

- `vma` – The VMA which will be adjusted to become the newly merged VMA.
- `start` – The address at which the newly merged mapping will begin (i.e. the merged VMA's `vm_start`).
- `end` – The exclusive bound of the newly merged mapping (i.e. the merged VMA's `vm_end`).
- `pgoff` – The merged VMA's `vm_pgoff`.
- `insert` – The VMA to insert. Not used in the merging logic.
- `expand` – The VMA to expand (either forwards or backwards) in order to obtain the newly merged VMA.

The `__vma_adjust()` function is huge, so let's look at it piece-by-piece (eliding out of scope uprobe and huge page handling).

We begin by determining what to do with the VMA that follows `vma`, in the cases where we are not inserting a VMA (i.e. merge cases). Let's start by taking a look at the code, then we'll try to unpack it as best we can as shown in Listing 5-37.

```

692 /*
693  * We cannot adjust vm_start, vm_end, vm_pgoff fields of a vma that
694  * is already present in an i_mmap tree without adjusting the tree.
695  * The following helper function should be used when such adjustments
696  * are necessary. The "insert" vma (if any) is to be inserted
697  * before we drop the necessary locks.
698  */
699 int __vma_adjust(struct vm_area_struct *vma, unsigned long start,
700                unsigned long end, pgoff_t pgoff, struct vm_area_struct *insert,
701                struct vm_area_struct *expand)
702 {
703     struct mm_struct *mm = vma->vm_mm;
704     struct vm_area_struct *next = vma->vm_next, *orig_vma = vma;
705     struct address_space *mapping = NULL;
706     struct rb_root_cached *root = NULL;
707     struct anon_vma *anon_vma = NULL;

```

```

708     struct file *file = vma->vm_file;
709     bool start_changed = false, end_changed = false;
710     long adjust_next = 0;
711     int remove_next = 0;
712
713     if (next && !insert) {
714         struct vm_area_struct *exporter = NULL, *importer = NULL;
715
716         if (end >= next->vm_end) {
717             /*
718              * vma expands, overlapping all the next, and
719              * perhaps the one after too (mprotect case 6).
720              * The only other cases that gets here are
721              * case 1, case 7 and case 8.
722              */
723             if (next == expand) {
724                 /*
725                  * The only case where we don't expand "vma"
726                  * and we expand "next" instead is case 8.
727                  */
728                 VM_WARN_ON(end != next->vm_end);
729                 /*
730                  * remove_next == 3 means we're
731                  * removing "vma" and that to do so we
732                  * swapped "vma" and "next".
733                  */
734                 remove_next = 3;
735                 VM_WARN_ON(file != next->vm_file);
736                 swap(vma, next);
737             } else {
738                 VM_WARN_ON(expand != vma);
739                 /*
740                  * case 1, 6, 7, remove_next == 2 is case 6,
741                  * remove_next == 1 is case 1 or 7.
742                  */
743                 remove_next = 1 + (end > next->vm_end);
744                 VM_WARN_ON(remove_next == 2 &&
745                             end != next->vm_next->vm_end);
746                 /* trim end to next, for case 6 first pass */
747                 end = next->vm_end;
748             }
749
750             exporter = next;
751             importer = vma;
752
753             /*

```

```

754         * If next doesn't have anon_vma, import from vma
      after
755         * next, if the vma overlaps with it.
756         */
757         if (remove_next == 2 && !next->anon_vma)
758             exporter = next->vm_next;
759
760     } else if (end > next->vm_start) {
761         /*
762          * vma expands, overlapping part of the next:
763          * mprotect case 5 shifting the boundary up.
764          */
765         adjust_next = (end - next->vm_start);
766         exporter = next;
767         importer = vma;
768         VM_WARN_ON(expand != importer);
769     } else if (end < vma->vm_end) {
770         /*
771          * vma shrinks, and !insert tells it's not
772          * split_vma inserting another: so it must be
773          * mprotect case 4 shifting the boundary down.
774          */
775         adjust_next = -(vma->vm_end - end);
776         exporter = vma;
777         importer = next;
778         VM_WARN_ON(expand != importer);
779     }

```

Listing 5-37: mm/mmap.c: `__vma_adjust()` initialisation

Within this function we declare the following local variables (frustratingly `next` is declared here and differs from `vma_merge()` in some cases, in all descriptions below `next` refers to `__vma_adjust()`'s `next`:

- `next` – The VMA immediately after `vma` (importantly, note that this is not always equal to the next VMA declared in `vma_merge()`, we will address this below).
- `adjust_next` – The number of bytes by which we offset `next->vm_start` (reducing this value in case 4 and increasing it in case 5), as well equivalently updating the page delta in `next->vm_pgoff`. Only relevant in cases 4 and 5.
- `remove_next` denotes the following:
 - 0 – No VMA removal will occur (cases 2 -5).
 - 1 – `next` will be removed (cases 1, 7)
 - 2 – `next` and `next->vm_next` will be removed (case 6).
 - 3 – `vma` and `next` have been swapped. The original `next` will be removed (case 8).

This is rather intricate and confusing, not helped by the fact that `next` is defined differently from the `next` within `vma_merge()` and that in case 8 we swap `vma` and `next`.

It's a good idea when trying to understand this code to constantly refer back to figure 5-4 as the code (and consequently the book) repeatedly refers to the numbered cases.

To make life easier, let's look at an expanded version of figure 5-1, removing the `insert` parameter which is always `NULL` and `pgoff` and adding `next`, `adjust_next` and `remove_next` (note that in case 8 `vma` and `next` are swapped. we show the post-swap state). We examine this in table 5-2.

Table 5-2: `__vma_adjust()` internal variables (relative to `vma_merge()`)

Variable	Case							
	1	2	3	4	5	6	7	8
vma	prev		next	prev				next *
Internal next	next		next->vm_next	next	area			area *
adjust_next	0			**	†	0		
remove_next	1	0			2		1	3
start	prev->vm_start		addr	prev->vm_start				addr
end	next->vm_end	end	next->vm_end	addr	end	next->vm_end ‡	area->vm_end	next->vm_end
expand	prev		next		prev			next
exporter	next	NULL		prev	area	area ††	area	
importer	prev	NULL		next	prev			next

Note that the references to `next` within the table refer to `next` as declared in `vma_merge()`.

Key:

- * - case 8 - Indicates that the `vma` (originally `area`) and `next` (originally `next`) parameters have been swapped.
- ** - case 4 - The **negative** value `addr - prev->vm_end` indicating the number of bytes `next->vm_start` must be offset by to expand the next VMA. `next->vm_pgoff` is updated accordingly too.
- † - case 5 - The **positive** value `end - area->vm_start` indicating the number of bytes `area->vm_start` must be offset by (i.e. shrunk) to accommodate the expanded `prev` VMA. `next->vm_pgoff` is updated accordingly too.
- ‡ - case 6 - The operation is performed in two passes - the first pass sets `end` to `area->vm_end` and the second sets it to `next->vm_end`.
- †† - case 6 - If `area` has no `anon_vma` field, then `exporter` is set to `next`.

The purpose of the code above is to initialise these values in the case where no insertion is occurring (which is true of merge) and the internal `next` value is not `NULL`.

The `exporter` and `importer` variables are used to handle an edge case as shown in Listing 5-38.

```

781                                     /*
782                                     * Easily overlooked: when mprotect shifts the boundary,
```

```

783         * make sure the expanding vma has anon_vma set if the
784         * shrinking vma had, to cover any anon pages imported.
785         */
786         if (exporter && exporter->anon_vma && !importer->anon_vma) {
787             int error;
788
789             importer->anon_vma = exporter->anon_vma;
790             error = anon_vma_clone(importer, exporter);
791             if (error)
792                 return error;
793         }
794     }

```

Listing 5-38: mm/mmap.c: `__vma_adjust()` importer/exporter

This ensures that the struct `anon_vma` object state is propagated from a merging source (the exporter) to a merged destination (the importer). See Chapter 7 for more on these objects and how they relate to the reverse mapping.

Now we have established the initial status of the VMA adjustment, it's time to perform the heavy lifting as shown in Listing 5-39.

```

795 again:
796 :
797     if (file) {
798         mapping = file->f_mapping;
799         root = &mapping->i_mmap;
800         :
806         i_mmap_lock_write(mapping);
807         if (insert) {
808             /*
809              * Put into interval tree now, so instantiated pages
810              * are visible to arm/parisc __flush_dcache_page
811              * throughout; but we cannot insert into address
812              * space until vma start or end is updated.
813              */
814             __vma_link_file(insert);
815         }
816     }
817
818     anon_vma = vma->anon_vma;
819     if (!anon_vma && adjust_next)
820         anon_vma = next->anon_vma;
821     if (anon_vma) {
822         VM_WARN_ON(adjust_next && next->anon_vma &&
823                   anon_vma != next->anon_vma);
824         anon_vma_lock_write(anon_vma);
825         anon_vma_interval_tree_pre_update_vma(vma);
826         if (adjust_next)

```

```

827             anon_vma_interval_tree_pre_update_vma(next);
828         }
829
830         if (file) {
831             flush_dcache_mmap_lock(mapping);
832             vma_interval_tree_remove(vma, root);
833             if (adjust_next)
834                 vma_interval_tree_remove(next, root);
835         }
836
837         if (start != vma->vm_start) {
838             vma->vm_start = start;
839             start_changed = true;
840         }
841         if (end != vma->vm_end) {
842             vma->vm_end = end;
843             end_changed = true;
844         }
845         vma->vm_pgoff = pgoff;
846         if (adjust_next) {
847             next->vm_start += adjust_next;
848             next->vm_pgoff += adjust_next >> PAGE_SHIFT;
849         }
850
851         if (file) {
852             if (adjust_next)
853                 vma_interval_tree_insert(next, root);
854             vma_interval_tree_insert(vma, root);
855             flush_dcache_mmap_unlock(mapping);
856         }

```

Listing 5-39: mm/mmap.c: __vma_adjust() VMA update

This is prefixed with the `again` label which is used in case 6 to perform multiple passes to merge both the previous and next VMAs.

Let's examine what is happening here:

- If `vma` is file-backed, we place the file's struct `address_space` page cache object into `mapping`, the root of the VMA interval tree in `root` and then acquire its lock via `i_mmap_lock_write()` (acquiring a write lock on the `mapping->i_mmap_rwsem` semaphore).
If this is an insertion rather than a merge we cover a corner case for specific architectures.
- When VMAs are adjusted, we must reconstruct the interval trees associated with them which are used to look up VMAs from the folio as part of the reverse mapping between folios and VMAs (see Chapter 7 for a deeper discussion of this).

This is necessary, as these are keyed on `vm_start` which is about to change. The struct `anon_vma` object tracks anonymous mappings, and

thus, prior to adjustment, we remove all entries in the `anon_vma` interval tree via `anon_vma_interval_tree_pre_update_vma()`. A write lock is also acquired over the `anon_vma` tree (maintained to account for forked Copy on Write mappings) via `anon_vma_lock_write()`.

If `adjust_next` is set by the previous code, indicating that the (internal) next VMA will be adjusted to, then we do the same with this VMA as well.

We also cover a corner case where `vma` lacks any target `anon_vma` object but `adjust_next` indicates the (internal) next VMA will have its updated and possesses a non-null `anon_vma` – in this instance we set `anon_vma` to this object.

- Equally, we must do the equivalent for the `address_space` page cache object associated with a file-backed VMA (note that `MAP_PRIVATE` mappings can be present in *both* interval trees). This is performed via `vma_interval_tree_remove()`*. Similar to the anonymous case, we perform the same operation on the (internal) next VMA if `adjust_next` is non-zero.
- At this point we go ahead and make the changes to `vma` – if `vm_start` is updated we set `start_changed`, equally if `vm_end` is changed we set `end_changed`. We also update `vm_pgoff`.
- If `adjust_next` has been set, we perform this adjustment by simply offsetting `next->vm_start` and `next->vm_pgoff`.
- Finally, we reinsert `vma` (and `next` if `adjust_next` indicates we are adjusting the VMA too) into the page cache struct `address_space` object's `i_mmap` interval tree.

Note the sporadic `*dcache*()` functions are for architectures which require the hardware cache to be updated manually when mappings change. This is not the case for x86-64 so we do not consider it, however note that this informs some of the ordering decisions made here.

Once this process is complete, we handle VMA removal or (non-merge) insertion as shown in Listing 5-40.

```

858         if (remove_next) {
859             /*
860              * vma_merge has merged next into vma, and needs
861              * us to remove next before dropping the locks.
862              */
863             if (remove_next != 3)
864                 __vma_unlink(mm, next, next);
865             else
866                 /*
867                  * vma is not before next if they've been
868                  * swapped.
869                  */

```

*. Note that this function is generated via the `INTERVAL_TREE_DEFINE()` macro in `mm/interval_tree.c`.

```

870         * pre-swap() next->vm_start was reduced so
871         * tell validate_mm_rb to ignore pre-swap()
872         * "next" (which is stored in post-swap())
873         * "vma").
874         */
875         __vma_unlink(mm, next, vma);
876         if (file)
877             __remove_shared_vm_struct(next, file, mapping);
878     } else if (insert) {
879         /*
880         * split_vma has split insert from vma, and needs
881         * us to insert it before dropping the locks
882         * (it may either follow vma or precede it).
883         */
884         __insert_vm_struct(mm, insert);
885     } else {
886         if (start_changed)
887             vma_gap_update(vma);
888         if (end_changed) {
889             if (!next)
890                 mm->highest_vm_end = vm_end_gap(vma);
891             else if (!adjust_next)
892                 vma_gap_update(next);
893         }
894     }
895
896     if (anon_vma) {
897         anon_vma_interval_tree_post_update_vma(vma);
898         if (adjust_next)
899             anon_vma_interval_tree_post_update_vma(next);
900         anon_vma_unlock_write(anon_vma);
901     }
902
903     if (file) {
904         i_mmap_unlock_write(mapping);
905         :
906     }
907 }

```

Listing 5-40: mm/mmap.c: __vma_adjust() VMA removal/insertion

We use `__vma_unlink()` to perform the actual removal of the next VMA* as shown in Listing 5-41.

```

682 static __always_inline void __vma_unlink(struct mm_struct *mm,
683                                         struct vm_area_struct *vma,

```

*. Note that the special case handling for `remove_next` equal to 3 (i.e. case 8 with the swapped `vma` and `next` parameters) varies what it passes to the `ignore` parameter, which is only relevant if `CONFIG_DEBUG_VM_RB` is specified.


```

684                                     struct vm_area_struct *ignore)
685 {
686     vma_rb_erase_ignore(vma, &mm->mm_rb, ignore);
687     __vma_unlink_list(mm, vma);
688     /* Kill the cache */
689     vmacache_invalidate(mm);
690 }

```

Listing 5-41: mm/mmap.c: __vma_unlink()

The `vma_rb_erase_ignore()` invocation ultimately performs the removal from the VMA red/black tree via `__vma_rb_erase()`. `vmacache_invalidate()` invalidates the current VMA cache and `__vma_unlink_list()` removes the VMA from the linked list as shown in Listing 5-42.

```

293 void __vma_unlink_list(struct mm_struct *mm, struct vm_area_struct *vma)
294 {
295     struct vm_area_struct *prev, *next;
296
297     next = vma->vm_next;
298     prev = vma->vm_prev;
299     if (prev)
300         prev->vm_next = next;
301     else
302         mm->mmap = next;
303     if (next)
304         next->vm_prev = prev;
305 }

```

Listing 5-42: mm/util.c: __vma_unlink_list()

This handles the case where either `vm_next` or `vm_prev` is NULL correctly.

If the VMA is file-mapped, the VMA is removed from the interval tree via `__remove_shared_vm_struct()` as shown in Listing 7-82.

```

104 /*
105  * Requires inode->i_mapping->i_mmap_rwsem
106  */
107 static void __remove_shared_vm_struct(struct vm_area_struct *vma,
108                                     struct file *file, struct address_space *mapping)
109 {
110     if (vma->vm_flags & VM_SHARED)
111         mapping_unmap_writable(mapping);
112
113     flush_dcache_mmap_lock(mapping);
114     vma_interval_tree_remove(vma, &mapping->i_mmap);
115     flush_dcache_mmap_unlock(mapping);
116 }

```

Listing 5-43: mm/mmap.c: __remove_shared_vm_struct()

The `*dcache*()` operations are for architectures which require manual data cache maintenance on mapping, however for architectures like x86-64 this is a no-op. `mapping_unmap_writable()` is called to track writable mappings of the page cache object (on the assumption that any shared mapping might be written to at any time*).

The actual removal from the `struct address_space->i_mmap` interval tree is performed by `vma_interval_tree_remove()`.

If the operation is an insertion rather than a merge, `__insert_vm_struct()` is called instead. We examine this function in section 5.1.4 as part of the discussion around splitting VMAs.

If we neither remove nor insert a VMA, we perform some accounting – `vma_gap_update()` updates internal state used within the red/black tree to keep track of gaps between a VMA and the nearest subtree VMA using common augmented red/black tree kernel code, and the `struct mm_struct->highest_vm_end` field is updated as necessary.

Finally if we have a `struct anon_vma` object we have updated we reestablish interval tree links via `anon_vma_interval_tree_post_update_vma()` for `vma` and next if `adjust_next` is non-zero before unlocking the `anon_vma` via `anon_vma_unlock_write()`.

Finally, if `vma` is file-backed, we need to unlock the page cache object's interval tree via `i_mmap_unlock_write()`.

After any removal performed here is complete, there are some final cleanup tasks we must take care before finally invoking `validate_mm()` to ensure that the `struct mm_struct` object is in a valid state as shown in Listing 5-44.

```

911         if (remove_next) {
912             :
913
914             if (next->anon_vma)
915                 anon_vma_merge(vma, next);
916             mm->map_count--;
917             mpol_put(vma_policy(next));
918             vm_area_free(next);
919             /*
920              * In mprotect's case 6 (see comments on vma_merge),
921              * we must remove another next too. It would clutter
922              * up the code too much to do both in one go.
923              */
924             if (remove_next != 3) {
925                 /*
926                  * If "next" was removed and vma->vm_end was
927                  * expanded (up) over it, in turn
928                  * "next->vm_prev->vm_end" changed and the
929                  * "vma->vm_next" gap must be updated.
930                  */
931             }

```

*. This functionality is largely used as part of cache maintenance for architectures which require it but is also used for write sealing of memfd objects created via `memfd_create()`.

```

932         */
933         next = vma->vm_next;
934     } else {
935         /*
936          * For the scope of the comment "next" and
937          * "vma" considered pre-swap(): if "vma" was
938          * removed, next->vm_start was expanded (down)
939          * over it and the "next" gap must be updated.
940          * Because of the swap() the post-swap() "vma"
941          * actually points to pre-swap() "next"
942          * (post-swap() "next" as opposed is now a
943          * dangling pointer).
944          */
945         next = vma;
946     }
947     if (remove_next == 2) {
948         remove_next = 1;
949         end = next->vm_end;
950         goto again;
951     }
952     else if (next)
953         vma_gap_update(next);
954     else {
955         /*
956          * If remove_next == 2 we obviously can't
957          * reach this path.
958          *
959          * If remove_next == 3 we can't reach this
960          * path because pre-swap() next is always not
961          * NULL. pre-swap() "next" is not being
962          * removed and its next->vm_end is not altered
963          * (and furthermore "end" already matches
964          * next->vm_end in remove_next == 3).
965          *
966          * We reach this only in the remove_next == 1
967          * case if the "next" vma that was removed was
968          * the highest vma of the mm. However in such
969          * case next->vm_end == "end" and the extended
970          * "vma" has vma->vm_end == next->vm_end so
971          * mm->highest_vm_end doesn't need any update
972          * in remove_next == 1 case.
973          */
974         VM_WARN_ON(mm->highest_vm_end != vm_end_gap(vma));
975     }
976 }
977 :
980 validate_mm(mm);

```

```

981
982     return 0;
983 }
```

Listing 5-44: mm/mmap.c: `__vma_adjust()` VMA post-removal housekeeping and VMA validation

We start by merging the (now unlinked) next VMA's struct anon_vma object with the current one via `anon_vma_merge()`. See Chapter 7 for a deep dive on this and the reverse mapping as a whole.

The struct `mm_struct->map_count` field is decremented to reflect the now-removed VMA, the reference count on the NUMA VMA policy is decremented via (see the NUMA chapter for more on this) and the VMA is actually freed via `vm_area_free()` as shown in Listing 5-45.

```

484 void vm_area_free(struct vm_area_struct *vma)
485 {
486     free_anon_vma_name(vma);
487     kmem_cache_free(vm_area_cachep, vma);
488 }
```

Listing 5-45: kernel/fork.c: `vm_area_free()`

This frees up any existent anonymous VMA name via `free_anon_vma_name()` and then simply invokes the slab cache free function `kmem_cache_free()` to release the VMA object itself.

Next there's something of a dance to cover case 8 (where `remove_next` is 3) – in this instance we want to ensure we update the internal VMA sub-tree gap value correctly which we do shortly afterwards for all cases accordingly.

In case 6, signified by `remove_next` being equal to 2, we have to loop around to the `again` label and go again, decrementing `remove_next` to indicate that we are looping.

You will recall from figure 5-2 (specifically discussed in the key section below) that we initially set `end` to `area->vm_end` on first pass, because the internal variable `next` is in fact set to `area` in `vma_merge()`. Here, having just removed the `area` VMA, `end` is reset to `next->vm_end` and we then merge the newly merged `prev` and `area` VMAs with `next` on the next iteration, which we examine in Figure 5-5.

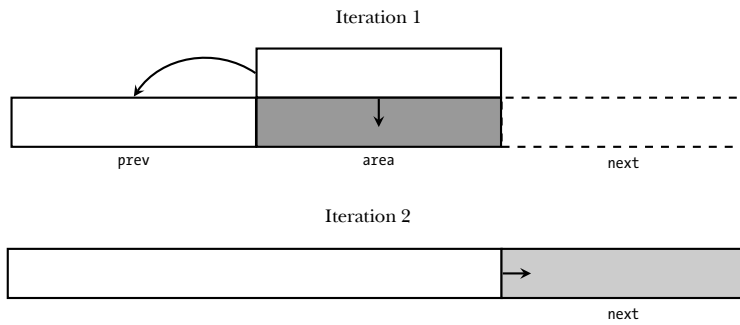


Figure 5-5: Case 6 iterations

After updating the VMA gap (a previously discussed internal implementation detail), we also perform an additional check in the instance where no internal next variable is specified in order to ensure that, if we reached the last VMA in the process address space, the `mm_struct->highest_vm_end` field is set correctly.

Finally, we perform some integrity checks via `validate_mm()` if and only if `CONFIG_DEBUG_VM_RB` is defined.

It's important to note that in the above we do not actually adjust the position of the VMA within the red/black tree (other than the case where we remove or insert a VMA of course). This is because any such change of bounds by its nature does not change the ordering, only the gap between VMAs which we do account for.

5.1.4 VMA split

We have now examined how VMAs are merged together, however we also have to examine the case where a portion of a VMA is *removed* which results in a VMA *split*.

Eliding out of scope `userfaultfd`, it is invoked from:

- `mbind_range()` – When changing the NUMA policy over an address range via `mbind()` this may result in a VMA split as well as a merge, depending on the range being adjusted.
- `mlock_fixup()` – When `mlock()`'ing ranges of memory, again we might split as well as merge depending on the range being locked/unlocked.
- `mprotect_fixup()` – When invoking `mprotect()` over an address range similarly we may need either to merge or to split the VMA to account for the change in protection.
- `madvise_update_vma()` – When performing a `madvise()` over an address range to adjust that range's attributes, this may result in a VMA split. Note that we invoke the internal `__split_vma()` function in this instance.
- `__do_munmap()` – When unmapping memory ranges, this might result in a 'hole' within a VMA and thus the need to split it. This uses the internal `__split_vma()` function.

One means of doing this is via the wrapper function `split_vma()`, which the first 3 cases above call as shown in Listing 5-46.

```

2736 /*
2737  * Split a vma into two pieces at address 'addr', a new vma is allocated
2738  * either for the first part or the tail.
2739  */
2740 int split_vma(struct mm_struct *mm, struct vm_area_struct *vma,
2741              unsigned long addr, int new_below)
2742 {
2743     if (mm->map_count >= sysctl_max_map_count)
2744         return -ENOMEM;
2745
2746     return __split_vma(mm, vma, addr, new_below);
2747 }

```

Listing 5-46: `mm/mmap.c: split_vma()`

While the heavy lifting is deferred to `__split_vma()`, this usefully adds a check against the `vm.max_map_count` tunable to ensure that a split would not exceed this limit.

Examining `__split_vma()` as shown in Listing 5-47.

```

2672 /*
2673  * __split_vma() bypasses sysctl_max_map_count checking. We use this where it
2674  * has already been checked or doesn't make sense to fail.
2675  */
2676 int __split_vma(struct mm_struct *mm, struct vm_area_struct *vma,
2677               unsigned long addr, int new_below)
2678 {
2679     struct vm_area_struct *new;
2680     int err;
2681
2682     if (vma->vm_ops && vma->vm_ops->may_split) {
2683         err = vma->vm_ops->may_split(vma, addr);
2684         if (err)
2685             return err;
2686     }
2687
2688     new = vm_area_dup(vma);
2689     if (!new)
2690         return -ENOMEM;
2691
2692     if (new_below)
2693         new->vm_end = addr;
2694     else {
2695         new->vm_start = addr;
2696         new->vm_pgoff += ((addr - vma->vm_start) >> PAGE_SHIFT);
2697     }

```

```

2698
2699     err = vma_dup_policy(vma, new);
2700     if (err)
2701         goto out_free_vma;
2702
2703     err = anon_vma_clone(new, vma);
2704     if (err)
2705         goto out_free_mpol;
2706
2707     if (new->vm_file)
2708         get_file(new->vm_file);
2709
2710     if (new->vm_ops && new->vm_ops->open)
2711         new->vm_ops->open(new);
2712
2713     if (new_below)
2714         err = vma_adjust(vma, addr, vma->vm_end, vma->vm_pgoff +
2715             ((addr - new->vm_start) >> PAGE_SHIFT), new);
2716     else
2717         err = vma_adjust(vma, vma->vm_start, addr, vma->vm_pgoff, new)
2718     ;
2719
2720     /* Success. */
2721     if (!err)
2722         return 0;
2723
2724     /* Clean everything up if vma_adjust failed. */
2725     if (new->vm_ops && new->vm_ops->close)
2726         new->vm_ops->close(new);
2727     if (new->vm_file)
2728         fput(new->vm_file);
2729     unlink_anon_vmas(new);
2730 out_free_mpol:
2731     mpol_put(vma_policy(new));
2732 out_free_vma:
2733     vm_area_free(new);
2734     return err;
2735 }

```

Listing 5-47: mm/mmap.c: `__split_vma()`

This begins by checking whether the struct `vm_area_struct` (VMA)'s struct `vm_operations_struct` `vm_ops` field has a `may_split()` predicate defined, which might be the case for a non-anonymous mapping. This gives file systems the chance to determine whether a split is permitted.

Next we duplicate the VMA, which will form the splitcopy of the VMA caused by the split in `vm_area_dup()` as shown in Listing 5-48.

```

465 struct vm_area_struct *vm_area_dup(struct vm_area_struct *orig)

```

```

466 {
467     struct vm_area_struct *new = kmem_cache_alloc(vm_area_cachep,
468         GFP_KERNEL);
469
470     if (new) {
471         ASSERT_EXCLUSIVE_WRITER(orig->vm_flags);
472         ASSERT_EXCLUSIVE_WRITER(orig->vm_file);
473         /*
474          * orig->shared.rb may be modified concurrently, but the clone
475          * will be reinitialized.
476          */
477         *new = data_race(*orig);
478         INIT_LIST_HEAD(&new->anon_vma_chain);
479         new->vm_next = new->vm_prev = NULL;
480         dup_anon_vma_name(orig, new);
481     }
482     return new;

```

Listing 5-48: kernel/fork.c: `vm_area_dup()`

This principally copies the existing VMA by assignment, but also resets the `anon_vma_chain` field (it will need to be re-added to this), and clears `vm_prev` and `vm_next` ready to be reinserted in right place. Any anonymous VMA name is also duplicated via `dup_anon_vma_name()`.

Note that the `vm_rb` node entry is not altered, this will be set correctly when we link this VMA into the tree.

The `new_below` parameter determines whether the new, duplicated VMA starts at `addr` or ends at it, as shown in Figure 5-6.

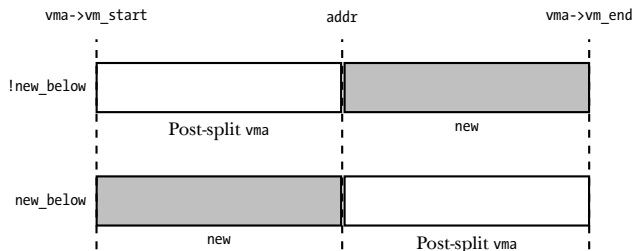


Figure 5-6: Split VMA cases

We duly update the new VMA's `vm_start`/`vm_end` and `vm_pgoff` fields.

We duplicate the VMA policy via `vma_dup_policy()` (see the NUMA chapter for details), duplicate the struct `anon_vma` object via `anon_vma_clone()` (see Chapter 7 for details on these objects within the reverse mapping), increment the reference count on the struct `file` object associated with the VMA if file-backed and invoke the struct `vm_operations_struct` `vm_ops->open()` callback if one is specified (i.e. a non-anonymous mapping which wishes to be notified of this).

Once this housekeeping is done, we are ready to perform the actual split. This is done via `vma_adjust()` as shown in Listing 5-49.

```

2581 static inline int vma_adjust(struct vm_area_struct *vma, unsigned long start,
2582                             unsigned long end, pgoff_t pgoff, struct vm_area_struct *insert)
2583 {
2584     return __vma_adjust(vma, start, end, pgoff, insert, NULL);
2585 }
```

Listing 5-49: include/linux/mm.h: `vma_adjust()`

Which ultimately invokes `__vma_adjust()` as described in section 5.1.3 described above, only specifying the insert parameter and setting the start and end according to `new_below`.

The insert version of `__vma_adjust()`, after adjusting the existing VMA's bounds, invokes `__insert_vm_struct()` to perform the insertion of new as shown in Listing 5-50.

```

666 /*
667  * Helper for vma_adjust() in the split_vma insert case: insert a vma into the
668  * mm's list and rbtree. It has already been inserted into the interval tree.
669  */
670 static void __insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *
        vma)
671 {
672     struct vm_area_struct *prev;
673     struct rb_node **rb_link, *rb_parent;
674
675     if (find_vma_links(mm, vma->vm_start, vma->vm_end,
676                       &prev, &rb_link, &rb_parent))
677         BUG();
678     __vma_link(mm, vma, prev, rb_link, rb_parent);
679     mm->map_count++;
680 }
```

Listing 5-50: mm/mmap.c: `__insert_vm_struct()`

This uses `find_vma_links()` to find the VMA prior to the one we are inserting as well as red/black tree nodes for the parent in `rb_parent` and the position the new node will reside in set in `rb_link`. We use these to perform the link in `__vma_link()` and increment the struct `mm_struct->mmap_count` field to indicate that a new VMA has been added.

Examining `__vma_link()` as shown in Listing 5-51.

```

636 static void
637 __vma_link(struct mm_struct *mm, struct vm_area_struct *vma,
638            struct vm_area_struct *prev, struct rb_node **rb_link,
639            struct rb_node *rb_parent)
640 {
641     __vma_link_list(mm, vma, prev);
```

```
642     __vma_link_rb(mm, vma, rb_link, rb_parent);  
643 }
```

Listing 5-51: mm/mmap.c: __vma_link()

This links the new VMA into the linked list via `__vma_link_list()` and the red/black tree via `__vma_link_rb()`.

Finally, `__split_vma()` performs clean up in the case where the split failed.

Overall the split is considerably simpler than the merge – each instance of splitting is the same, the only variable factor is whether we place the newly duplicated split VMA above or below the specified address.

6

PAGE FAULTS

In linux, the actual physical allocation of userland memory occurs not on mapping but when a *page fault* occurs on access to that memory. A page fault arises when a user attempts to access virtual memory that either does not have a valid page table mapping or for which the page table mapping does not allow the attempted action (e.g. writing to a read-only mapping). In this chapter we examine this mechanism in detail.

The page faults themselves arise due to the computer's Memory Management Unit (MMU) detecting an invalid access. The kernel can tell the hardware to invoke its handler when this occurs and modern hardware permits the mapping to be corrected, i.e. 'fixing up' the page fault, if the kernel deems the access to have been valid.

When page faults occur at an address contained within a valid mapping, the kernel is then able to 'back' this memory by performing the physical allocation of that memory and updating the page table mapping to correctly reference it. Doing so is termed *demand paging* because the actual allocation of memory occurs *on demand* rather than on mapping.

When this occurs, the pages are said to be 'faulted in' (which we can further subdivide into 'read faulting' and 'write faulting'). Note that a page

fault can be pre-triggered when using `mmap()` via the `MAP_POPULATE` flag. This is, from the kernel's point of view, equivalent to write faulting them in.

The page fault mechanism permits a disconnect between mapping userland memory and both allocating physical memory and establishing page table mappings to it – all that is required for userland memory to be valid is for the range to be contained within a VMA with the appropriate flags.

Broadly speaking there are two types of page faults – those arising for *anonymous* mappings and those arising for *file-backed* ones. They each have different semantics, with the former allocating physical memory to back them and the latter checking the page cache to see if the file is already available and mapping to that if so and if not invoke filesystem functionality to place the data into the page cache before mapping it.

Complexity arises around swap, page migration and NUMA balancing, each of which result specific actions being taken on page faults, however discussion of these are deferred to their respective chapters.

When it comes to architecture-specific functionality we must speak about a specific architecture as otherwise discussing the topic would quickly become impractical.

6.1 Hardware page fault handling

Implementations differ in how they handle hardware page fault handling, but after doing so they each share common fault handling code.

We examine how x86-64 invokes the general fault handling logic from an x86-64 hardware fault in Figure 6-1.

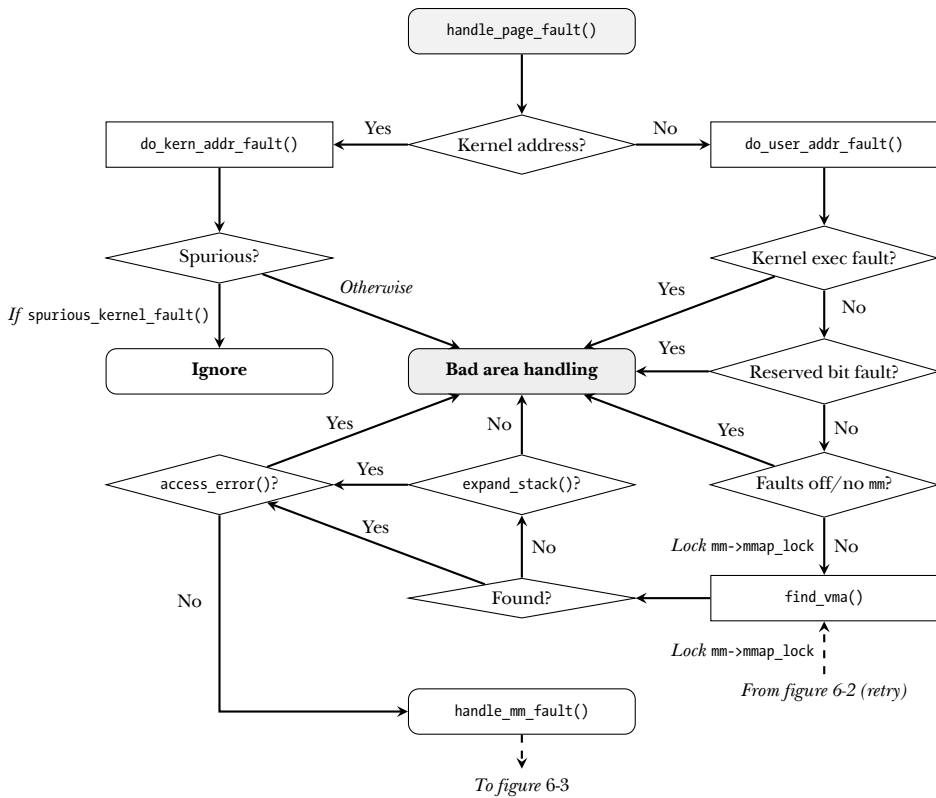


Figure 6-1: Overview of x86-64 hardware page faulting

The x86-64 architecture defines a handler for the hardware page fault at `exc_page_fault()` which in turn invokes `handle_page_fault()` as shown in Listing 6-1.

```

1475 static __always_inline void
1476 handle_page_fault(struct pt_regs *regs, unsigned long error_code,
1477                  unsigned long address)
1478 {
1479     :
1480     /* Was the fault on kernel-controlled part of the address space? */
1481     if (unlikely(fault_in_kernel_space(address))) {
1482         do_kern_addr_fault(regs, error_code, address);
1483     } else {
1484         do_user_addr_fault(regs, error_code, address);
1485     }
1486     /*
1487     * User address page fault handling might have reenabled
1488     * interrupts. Fixing up all potential exit points of
1489     * do_user_addr_fault() and its leaf functions is just not
1490     * doable w/o creating an unholy mess or turning the code
1491     * upside down.
1492     */
1493 
```

```

1495         */
1496         local_irq_disable();
1497     }
1498 }

```

Listing 6-1: arch/x86/mm/fault.c: `handle_page_fault()`

We start by determining whether the fault occurred in kernel space via `fault_in_kernel_space()` and then using this to determine which of `do_kern_addr_fault()` or `do_user_addr_fault()` to invoke.

6.1.1 Kernel page faults

Firstly, we will examine how page faults that occur in the kernel portion of virtual address space are handled. The usual functionality available to userspace such as faulting in pages are not available to the kernel, and therefore most of the time a page fault in a kernel address indicates an error state, however there are occasions where this is not the case.

Regardless, this necessitates entirely separate handling from userland page faults, which is handled by `fault_in_kernel_space()` as shown in Listing 6-2.

```

1128 bool fault_in_kernel_space(unsigned long address)
1129 {
1130     /*
1131      * On 64-bit systems, the vsyscall page is at an address above
1132      * TASK_SIZE_MAX, but is not considered part of the kernel
1133      * address space.
1134      */
1135     if (IS_ENABLED(CONFIG_X86_64) && is_vsyscall_vaddr(address))
1136         return false;
1137
1138     return address >= TASK_SIZE_MAX;
1139 }

```

Listing 6-2: arch/x86/mm/fault.c: `fault_in_kernel_space()`

This first checks if the page fault is for the legacy vsyscall page via `is_vsyscall_vaddr()`. This is a page of memory ostensibly mapped into userspace to improve performance of some system calls, however this functionality is typically disabled or emulated (via `CONFIG_X86_VSYSCALL_EMULATION`) due to security concerns.

Otherwise, it simply checks to see if the virtual address is equal to or exceeds `TASK_SIZE_MAX` which is the first address outside of the userland memory range.

Faults in kernel address space is handled by `do_kern_addr_fault()` (eliding irrelevant 32-bit and kprobe cases) as shown in Listing 6-3.

```

1141 /*
1142  * Called for all faults where 'address' is part of the kernel address

```

```

1143  * space. Might get called for faults that originate from *code* that
1144  * ran in userspace or the kernel.
1145  */
1146  static void
1147  do_kern_addr_fault(struct pt_regs *regs, unsigned long hw_error_code,
1148                   unsigned long address)
1149  {
1150      :
1151      /* Was the fault spurious, caused by lazy TLB invalidation? */
1152      if (spurious_kernel_fault(hw_error_code, address))
1153          return;
1154      :
1155      /*
1156       * Note, despite being a "bad area", there are quite a few
1157       * acceptable reasons to get here, such as erratum fixups
1158       * and handling kernel code that can fault, like get_user().
1159       *
1160       * Don't take the mm semaphore here. If we fixup a prefetch
1161       * fault we could otherwise deadlock:
1162       */
1163      bad_area_nosemaphore(regs, hw_error_code, address);
1164  }

```

Listing 6-3: arch/x86/mm/fault.c: `do_kern_addr_fault()`

Firstly we determine whether the kernel fault was *spurious* – that is, due to lazy TLB, a kernel TLB entry is permitted to go stale, meaning that an invalid access to a write-protected or execution-protected region of memory may occur due to the TLB entry (see section 7.1).

NOTE

The Transaction Lookaside Buffer (TLB) is a key hardware cache between virtual and physical addresses used to avoid having to walk the page table for every memory access.

Therefore, if the kernel tries to write to read-only memory or execute in NX (no execute) memory we must check whether the page table entry actually does permit this despite the fault. We do this in `spurious_kernel_fault()` as shown in Listing 6-4.

```

985  /*
986  * Handle a spurious fault caused by a stale TLB entry.
987  *
988  * This allows us to lazily refresh the TLB when increasing the
989  * permissions of a kernel page (RO -> RW or NX -> X). Doing it
990  * eagerly is very expensive since that implies doing a full
991  * cross-processor TLB flush, even if no stale TLB entries exist
992  * on other processors.
993  *
994  * Spurious faults may only occur if the TLB contains an entry with

```



```

995  * fewer permission than the page table entry. Non-present (P = 0)
996  * and reserved bit (R = 1) faults are never spurious.
997  *
998  * There are no security implications to leaving a stale TLB when
999  * increasing the permissions on a page.
1000  *
1001  * Returns non-zero if a spurious fault was handled, zero otherwise.
1002  *
1003  * See Intel Developer's Manual Vol 3 Section 4.10.4.3, bullet 3
1004  * (Optional Invalidation).
1005  */
1006  static ninline int
1007  spurious_kernel_fault(unsigned long error_code, unsigned long address)
1008  {
1009      pgd_t *pgd;
1010      p4d_t *p4d;
1011      pud_t *pud;
1012      pmd_t *pmd;
1013      pte_t *pte;
1014      int ret;
1015
1016      /*
1017       * Only writes to R0 or instruction fetches from NX may cause
1018       * spurious faults.
1019       *
1020       * These could be from user or supervisor accesses but the TLB
1021       * is only lazily flushed after a kernel mapping protection
1022       * change, so user accesses are not expected to cause spurious
1023       * faults.
1024       */
1025      if (error_code != (X86_PF_WRITE | X86_PF_PROT) &&
1026          error_code != (X86_PF_INSTR | X86_PF_PROT))
1027          return 0;
1028
1029      pgd = init_mm.pgd + pgd_index(address);
1030      if (!pgd_present(*pgd))
1031          return 0;
1032
1033      p4d = p4d_offset(pgd, address);
1034      if (!p4d_present(*p4d))
1035          return 0;
1036
1037      if (p4d_large(*p4d))
1038          return spurious_kernel_fault_check(error_code, (pte_t *) p4d);
1039
1040      pud = pud_offset(p4d, address);
1041      if (!pud_present(*pud))

```

```

1042         return 0;
1043
1044     if (pud_large(*pud))
1045         return spurious_kernel_fault_check(error_code, (pte_t *) pud);
1046
1047     pmd = pmd_offset(pud, address);
1048     if (!pmd_present(*pmd))
1049         return 0;
1050
1051     if (pmd_large(*pmd))
1052         return spurious_kernel_fault_check(error_code, (pte_t *) pmd);
1053
1054     pte = pte_offset_kernel(pmd, address);
1055     if (!pte_present(*pte))
1056         return 0;
1057
1058     ret = spurious_kernel_fault_check(error_code, pte);
1059     if (!ret)
1060         return 0;
1061
1062     /*
1063      * Make sure we have permissions in PMD.
1064      * If not, then there's a bug in the page tables:
1065      */
1066     ret = spurious_kernel_fault_check(error_code, (pte_t *) pmd);
1067     WARN_ONCE(!ret, "PMD has incorrect permission bits\n");
1068
1069     return ret;
1070 }

```

Listing 6-4: arch/x86/mm/fault.c: `spurious_kernel_fault()`

This simply walks the page tables, checking whether each page table level is present, and if so if it is potentially a huge page (see huge page chapter for more on this topic) checking whether the huge page entry (equivalent to a PTE) is valid, otherwise (if the page table mappings are present) reaching the PTE itself and performing this check there.

There is an additional check for PTE in case the PMD has incorrect permissions set as an additional bug check.

Each of these checks is performed by `spurious_kernel_fault_check()` as shown in Listing 6-5.

```

974 static int spurious_kernel_fault_check(unsigned long error_code, pte_t *pte)
975 {
976     if ((error_code & X86_PF_WRITE) && !pte_write(*pte))
977         return 0;
978
979     if ((error_code & X86_PF_INSTR) && !pte_exec(*pte))
980         return 0;

```

```

981
982     return 1;
983 }

```

Listing 6-5: arch/x86/mm/fault.c: `spurious_kernel_fault_check()`

This simply checks whether the PTE (or PTE equivalent) page table entry does indeed match the page fault, where `X86_PF_WRITE` and `X86_PF_INSTR` are page fault error codes that represent a write to read-only memory and execution of code in NX memory respectively, checked by `pte_write()` and `pte_exec()`.

Coming back to `do_kern_addr_fault()`, if the fault was not spurious, we invoke `bad_area_nosemaphore()`, which is used to handle invalid memory accesses when no `struct mm_struct->mmap_lock` semaphore is held. This in turn invokes `__bad_area_nosemaphore()` with the signal error set to `SEGV_MAPERR` as shown in Listing 6-6.

```

799 static void
800 __bad_area_nosemaphore(struct pt_regs *regs, unsigned long error_code,
801                       unsigned long address, u32 pkey, int si_code)
802 {
803     struct task_struct *tsk = current;
804
805     if (!user_mode(regs)) {
806         kernelmode_fixup_or_oops(regs, error_code, address,
807                                  SIGSEGV, si_code, pkey);
808         return;
809     }
810     :
849 }

```

Listing 6-6: arch/x86/mm/fault.c: Kernel handling in `__bad_area_nosemaphore()`

Note that we elide the user-mode handling of this function which we discuss in section 6.11. This passes through to `kernelmode_fixup_or_oops()` as shown in Listing 6-7.

```

711 static ninline void
712 kernelmode_fixup_or_oops(struct pt_regs *regs, unsigned long error_code,
713                          unsigned long address, int signal, int si_code,
714                          u32 pkey)
715 {
716     WARN_ON_ONCE(user_mode(regs));
717
718     /* Are we prepared to handle this kernel fault? */
719     if (fixup_exception(regs, X86_TRAP_PF, error_code, address)) {
720         /*
721          * Any interrupt that takes a fault gets the fixup. This makes
722          * the below recursive fault logic only apply to a faults from

```

```

723         * task context.
724         */
725         if (in_interrupt())
726             return;
727
728         /*
729          * Per the above we're !in_interrupt(), aka. task context.
730          *
731          * In this case we need to make sure we're not recursively
732          * faulting through the emulate_vsyscall() logic.
733          */
734         if (current->thread.sig_on_uaccess_err && signal) {
735             sanitize_error_code(address, &error_code);
736
737             set_signal_archinfo(address, error_code);
738
739             if (si_code == SEGV_PKUERR) {
740                 force_sig_pkuerr((void __user *)address, pkey)
741             ;
742                 } else {
743                     /* XXX: hwpoison faults will set the wrong
744                     code. */
745                     force_sig_fault(signal, si_code, (void __user
746                     *)address);
747                 }
748             }
749
750             /*
751              * Barring that, we can do the fixup and be happy.
752              */
753             return;
754         }
755
756         /*
757          * AMD erratum #91 manifests as a spurious page fault on a PREFETCH
758          * instruction.
759          */
760         if (is_prefetch(regs, error_code, address))
761             return;
762
763         page_fault_oops(regs, error_code, address);
764     }

```

Listing 6-7: arch/x86/mm/fault.c: `kernelmode_fixup_or_oops()`

We first check to see if we are fixing up an exception from the kernel uaccess interface via `fixup_exception()`. Section 8.1.1 covers this in detail, but briefly – this is a means by which the kernel can access userland mappings

directly, i.e. by simply accessing a userland virtual address, but it can only be performed in certain designated places in the kernel in case of *unhandled* page faults.

If we indeed we were fixing up (not in the case we are examining here – as the fault is in a kernel address, not userland), then no oops will be caused and we perform one more specific check – if we are not in interrupt context, and the current struct task_struct->thread architecture-specific thread information state struct thread_struct has its sig_on_uaccess_err flag set, and a signal has been specified, then this signal is sent.

Finally, we cover an AMD spurious page fault edge case, before accepting that this was an invalid access and thus a kernel oops must be raised and invoking page_fault_oops().

NOTE

A kernel oops means that something has gone terribly wrong in the kernel. It will, under ordinary circumstances, simply cause the running userland process to be SIGKILL'd with a message reported to the kernel log, however if this is a kernel thread or under other circumstances (such as the tunable kernel.panic_on_oops being set) then this results in the kernel panicking, i.e. the kernel being halted altogether.

Examining page_fault_oops() as shown in Listing 6-8.

```

631 static inline void
632 page_fault_oops(struct pt_regs *regs, unsigned long error_code,
633                unsigned long address)
634 {
635     #ifdef CONFIG_VMAP_STACK
636         struct stack_info info;
637     #endif
638     unsigned long flags;
639     int sig;
640
641     if (user_mode(regs)) {
642         /*
643          * Implicit kernel access from user mode? Skip the stack
644          * overflow and EFI special cases.
645          */
646         goto oops;
647     }
648
649     #ifdef CONFIG_VMAP_STACK
650         /*
651          * Stack overflow? During boot, we can fault near the initial
652          * stack in the direct map, but that's not an overflow -- check
653          * that we're in vmalloc space to avoid this.
654          */
655         if (is_vmalloc_addr((void *)address) &&
656             get_stack_guard_info((void *)address, &info)) {
657             /*

```

```

658         * We're likely to be running with very little stack space
659         * left. It's plausible that we'd hit this condition but
660         * double-fault even before we get this far, in which case
661         * we're fine: the double-fault handler will deal with it.
662         *
663         * We don't want to make it all the way into the oops code
664         * and then double-fault, though, because we're likely to
665         * break the console driver and lose most of the stack dump.
666         */
667         call_on_stack(__this_cpu_ist_top_va(DF) - sizeof(void*),
668                     handle_stack_overflow,
669                     ASM_CALL_ARG3,
670                     , [arg1] "r" (regs), [arg2] "r" (address), [arg3
        ] "r" (&info));
671
672         unreachable();
673     }
674 #endif
675
676     /*
677     * Buggy firmware could access regions which might page fault. If
678     * this happens, EFI has a special OOPS path that will try to
679     * avoid hanging the system.
680     */
681     if (IS_ENABLED(CONFIG_EFI))
682         efi_crash_gracefully_on_page_fault(address);
683
684     /* Only not-present faults should be handled by KFENCE. */
685     if (!(error_code & X86_PF_PROT) &&
686         kfence_handle_page_fault(address, error_code & X86_PF_WRITE, regs)
687     )
688         return;
689
690     oops:
691     /*
692     * Oops. The kernel tried to access some bad page. We'll have to
693     * terminate things with extreme prejudice:
694     */
695     flags = oops_begin();
696
697     show_fault_oops(regs, error_code, address);
698
699     if (task_stack_end_corrupted(current))
700         printk(KERN_EMERG "Thread overran stack, or stack corrupted\n"
701
702
703     sig = SIGKILL;

```

```

702         if (__die("Oops", regs, error_code))
703             sig = 0;
704
705         /* Executive summary in case the body of the oops scrolled away */
706         printk(KERN_DEFAULT "CR2: %016lx\n", address);
707
708         oops_end(flags, regs, sig);
709     }

```

Listing 6-8: arch/x86/mm/fault.c: page_fault_oops()

Note this function is explicitly invoked for invalid accesses to *kernel* mappings. It performs a few specific checks if we are in kernel context, if we are in userland trying to access kernel memory then these are skipped:

- Was this was a stack overflow due to hitting a ‘guard page’? (i.e. a page mapped such that it will fault just below the stack so this condition is detectable) if we use vmalloc kernel stacks (this makes it easy to detect this situation, whereas kernel stacks using the direct mapping make it harder to track this condition).
- If the kernel is EFI-enabled via CONFIG_EFI, then perform specific handling for buggy firmware.
- Finally, if *kfence* handling is enabled (via CONFIG_KFENCE), then let this logic handle the fault if it arose due to the memory not being present.

After these checks are complete, we perform the oops itself, updating the kernel log and performing ordinary kernel oops handling via `oops_begin()`, `show_fault_oops()`, `__die()` and `oops_end()` (this topic is out of scope for the book).

6.1.2 Userland page faults

Now let’s examine the userland case – note that this is invoked both in userland and kernel mode when accessing userland addresses. This is handled in `do_user_addr_fault()`.

Let’s start by examining the initial checks performed by this function (eliding out of scope *kprobe* and *SMAP* handling) as shown in Listing 6-9.

```

1219 static inline
1220 void do_user_addr_fault(struct pt_regs *regs,
1221                        unsigned long error_code,
1222                        unsigned long address)
1223 {
1224     struct vm_area_struct *vma;
1225     struct task_struct *tsk;
1226     struct mm_struct *mm;
1227     vm_fault_t fault;
1228     unsigned int flags = FAULT_FLAG_DEFAULT;
1229

```

```

1230         tsk = current;
1231         mm = tsk->mm;
1232
1233         if (unlikely((error_code & (X86_PF_USER | X86_PF_INSTR)) ==
1234             X86_PF_INSTR)) {
1235             /*
1236              * Whoops, this is kernel mode code trying to execute from
1237              * user memory. Unless this is AMD erratum #93, which
1238              * corrupts RIP such that it looks like a user address,
1239              * this is unrecoverable. Don't even try to look up the
1240              * VMA or look for extable entries.
1241              */
1242             if (is_errata93(regs, address))
1243                 return;
1244
1245             page_fault_oops(regs, error_code, address);
1246             return;
1247         }
1248         :
1249
1250         /*
1251          * Reserved bits are never expected to be set on
1252          * entries in the user portion of the page tables.
1253          */
1254         if (unlikely(error_code & X86_PF_RSVD))
1255             pgtable_bad(regs, error_code, address);
1256         :
1257
1258         /*
1259          * If we're in an interrupt, have no user context or are running
1260          * in a region with pagefaults disabled then we must not take the
1261          fault
1262          */
1263         if (unlikely(faulthandler_disabled() || !mm)) {
1264             bad_area_nosemaphore(regs, error_code, address);
1265             return;
1266         }
1267
1268         /*
1269          * It's safe to allow irq's after cr2 has been saved and the
1270          * vmalloc fault has been handled.
1271          *
1272          * User-mode registers count as a user access even for any
1273          * potential system fault or CPU buglet:
1274          */
1275         if (user_mode(regs)) {
1276             local_irq_enable();
1277             flags |= FAULT_FLAG_USER;

```



```

1296         } else {
1297             if (regs->flags & X86_EFLAGS_IF)
1298                 local_irq_enable();
1299         }
1300
1301         perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS, 1, regs, address);
1302
1303         if (error_code & X86_PF_WRITE)
1304             flags |= FAULT_FLAG_WRITE;
1305         if (error_code & X86_PF_INSTR)
1306             flags |= FAULT_FLAG_INSTRUCTION;
1307
1308 #ifdef CONFIG_X86_64
1309     /*
1310      * Faults in the vsyscall page might need emulation. The
1311      * vsyscall page is at a high address (>PAGE_OFFSET), but is
1312      * considered to be part of the user address space.
1313      *
1314      * The vsyscall page does not have a "real" VMA, so do this
1315      * emulation before we go searching for VMAs.
1316      *
1317      * PKRU never rejects instruction fetches, so we don't need
1318      * to consider the PF_PK bit.
1319      */
1320     if (is_vsyscall_vaddr(address)) {
1321         if (emulate_vsyscall(error_code, regs, address))
1322             return;
1323     }
1324 #endif

```

Listing 6-9: arch/x86/mm/fault.c: do_user_addr_fault() Initial checks

The logic is as follows:

- Check whether the fault is arising from the kernel (indicated by the error code not having X86_PF_USER set) is trying to *execute* code (indicated via X86_PF_INSTR). This is simply not permitted – under no circumstances should kernel mode attempt to execute code from userland, least of all when that causes a page fault.
- If the error code indicates that a page table entry flipped reserved bits, raise an oops via `pgtable_bad()`.
- If page fault handling is disabled or we are in atomic context (for example in an IRQ handler) as determined by `faulthandler_disabled()` (which checks these two cases via `in_atomic()` and `pagefault_disabled()`) or there is no struct `mm_struct` context (e.g. this is a classical kernel thread), then we do not process the page fault and instead invoke `bad_area_nosemaphore()`. We discuss the various userland address bad address scenarios below.

- If the fault occurred in user mode, we set the `FAULT_FLAG_USER` flag to indicate this (more on fault flags below). We additionally conditionally enable local IRQs. Interrupt handling is out of scope for the book.
- After updating statistics, specify `FAULT_FLAG_WRITE` if the fault was a write fault or `FAULT_FLAG_INSTRUCTION` was on execution.
- If the fault occurred in the legacy `vsyscall` page, then provide emulation for this if appropriate. Discussion of this is out of scope for the book.

Next, we acquire the struct `mm_struct->mmap_lock` and invoke the generic fault handling logic as shown in Listing 6-10.

```

1326      /*
1327      * Kernel-mode access to the user address space should only occur
1328      * on well-defined single instructions listed in the exception
1329      * tables. But, an erroneous kernel fault occurring outside one of
1330      * those areas which also holds mmap_lock might deadlock attempting
1331      * to validate the fault against the address space.
1332      *
1333      * Only do the expensive exception table search when we might be at
1334      * risk of a deadlock. This happens if we
1335      * 1. Failed to acquire mmap_lock, and
1336      * 2. The access did not originate in userspace.
1337      */
1338      if (unlikely(!mmap_read_trylock(mm))) {
1339          if (!user_mode(regs) && !search_exception_tables(regs->ip)) {
1340              /*
1341               * Fault from code in kernel from
1342               * which we do not expect faults.
1343               */
1344              bad_area_nosemaphore(regs, error_code, address);
1345              return;
1346          }
1347      retry:
1348          mmap_read_lock(mm);
1349      } else {
1350          /*
1351           * The above down_read_trylock() might have succeeded in
1352           * which case we'll have missed the might_sleep() from
1353           * down_read():
1354           */
1355          might_sleep();
1356      }
1357
1358      vma = find_vma(mm, address);
1359      if (unlikely(!vma)) {
1360          bad_area(regs, error_code, address);
1361          return;

```

```

1362     }
1363     if (likely(vma->vm_start <= address))
1364         goto good_area;
1365     if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
1366         bad_area(regs, error_code, address);
1367         return;
1368     }
1369     if (unlikely(expand_stack(vma, address))) {
1370         bad_area(regs, error_code, address);
1371         return;
1372     }
1373
1374     /*
1375      * Ok, we have a good vm_area for this memory access, so
1376      * we can handle it..
1377      */
1378     good_area:
1379     if (unlikely(access_error(error_code, vma))) {
1380         bad_area_access_error(regs, error_code, address, vma);
1381         return;
1382     }
1383
1384     /*
1385      * If for any reason at all we couldn't handle the fault,
1386      * make sure we exit gracefully rather than endlessly redo
1387      * the fault. Since we never set FAULT_FLAG_RETRY_NOWAIT, if
1388      * we get VM_FAULT_RETRY back, the mmap_lock has been unlocked.
1389      *
1390      * Note that handle_userfault() may also release and reacquire
1391      * mmap_lock
1392      * (and not return with VM_FAULT_RETRY), when returning to userland to
1393      * repeat the page fault later with a VM_FAULT_NOPAGE retval
1394      * (potentially after handling any pending signal during the return to
1395      * userland). The return to userland is identified whenever
1396      * FAULT_FLAG_USER|FAULT_FLAG_KILLABLE are both set in flags.
1397      */
1398     fault = handle_mm_fault(vma, address, flags, regs);

```

Listing 6-10: arch/x86/mm/fault.c: `do_user_addr_fault()` fault handling

If we can't immediately acquire the struct `mm_struct->mmap_lock`, we enter a very specific edge case – to avoid a deadlock occurring in kernel uaccess handling when this lock is already held we perform the expensive kernel `search_exception_tables()` operation here – if this fails then we are done. If we *can* obtain the lock, then this check will only be performed if the userland fault handler is unable to handle the fault. See section 8.1.1 for more on uaccess.

We now find ourselves in the core page fault handling logic with the `mm_struct` read lock held and ready to proceed with the generic page fault logic. We start by finding the VMA (we cover this in section 4.4.5) via `find_vma()`, which determines the first VMA whose `vm_end` exceeds the faulting address.

Once we have determined which VMA fulfils this requirement, we must determine whether the faulting address sits within it by seeing if it sits above its `vm_start`. If so, we're good and move on to `good_area` to perform the generic page fault handling.

If not, we check to see if this is a stack that grows downwards – which it would need to be for the address to be valid and for it to be less than this VMA's `vm_start`. We perform this check via `expand_stack()` which in turn invokes `expand_downwards()`. See section 6.10 for a detailed analysis of this.

Prior to handing over to the generic page table handler, we invoke `access_error()` to ensure that access is indeed permitted for the VMA (for example, a write fault might occur on a Copy on Write mapping, but it also might occur on a readonly one and we have to differentiate between the two):

Prior to handing over to the generic page fault handler `handle_mm_fault()`, we check to ensure that it is a valid access via `access_error()`, returning 0 if the access is OK, 1 otherwise (eliding out of scope protection keys and SGX checks) as shown in Listing 6-11.

```

1075 static inline int
1076 access_error(unsigned long error_code, struct vm_area_struct *vma)
1077 {
1078     /* This is only called for the current mm, so: */
1079     bool foreign = false;
1080     :
1110     if (error_code & X86_PF_WRITE) {
1111         /* write, present and write, not present: */
1112         if (unlikely(!(vma->vm_flags & VM_WRITE)))
1113             return 1;
1114         return 0;
1115     }
1116
1117     /* read, present: */
1118     if (unlikely(error_code & X86_PF_PROT))
1119         return 1;
1120
1121     /* read, not present: */
1122     if (unlikely(!vma_is_accessible(vma)))
1123         return 1;
1124
1125     return 0;
1126 }
```

Listing 6-11: `arch/x86/mm/fault.c: access_error()`

We indicate an access error if:

- The fault was a write fault but the VMA does not permit writes (so this is confirmed not to be a Copy on Write error).
- The fault was a protection fault, i.e. the memory is present, but not permitted to be read.
- The VMA is designated as not being accessible via `vma_is_accessible()` (i.e., none of the flags specified in `VM_ACCESS_FLAGS` are set – neither `VM_READ`, `VM_WRITE` nor `VM_EXEC`).

If at any point a bad access is detected, we handle this in `bad_area()`. We will examine this in detail in section 6.11. After all these checks are complete, we are ready to perform the generic page table handling that's shared between all architectures via `handle_mm_fault()`, see section 6.2 for a very detailed analysis of this!

Finally, after the page fault handling is complete, we handle any post-fault actions as determined by the results of the page fault as shown in Listing 6-12.

```

1399     if (fault_signal_pending(fault, regs)) {
1400         /*
1401          * Quick path to respond to signals. The core mm code
1402          * has unlocked the mm for us if we get here.
1403          */
1404         if (!user_mode(regs))
1405             kernelmode_fixup_or_oops(regs, error_code, address,
1406                                     SIGBUS, BUS_ADRERR,
1407                                     ARCH_DEFAULT_PKEY);
1408         return;
1409     }
1410
1411     /* The fault is fully completed (including releasing mmap lock) */
1412     if (fault & VM_FAULT_COMPLETED)
1413         return;
1414
1415     /*
1416      * If we need to retry the mmap_lock has already been released,
1417      * and if there is a fatal signal pending there is no guarantee
1418      * that we made any progress. Handle this case first.
1419      */
1420     if (unlikely(fault & VM_FAULT_RETRY)) {
1421         flags |= FAULT_FLAG_TRIED;
1422         goto retry;
1423     }
1424
1425     mmap_read_unlock(mm);
1426     if (likely(!(fault & VM_FAULT_ERROR)))
1427         return;

```

```

1428
1429     if (fatal_signal_pending(current) && !user_mode(regs)) {
1430         kernelmode_fixup_or_oops(regs, error_code, address,
1431                                 0, 0, ARCH_DEFAULT_PKEY);
1432         return;
1433     }
1434
1435     if (fault & VM_FAULT_OOM) {
1436         /* Kernel mode? Handle exceptions or die: */
1437         if (!user_mode(regs)) {
1438             kernelmode_fixup_or_oops(regs, error_code, address,
1439                                     SIGSEGV, SEGV_MAPERR,
1440                                     ARCH_DEFAULT_PKEY);
1441             return;
1442         }
1443         /*
1444          * We ran out of memory, call the OOM killer, and return the
1445          * userspace (which will retry the fault, or kill us if we got
1446          * oom-killed):
1447          */
1448         pagefault_out_of_memory();
1449     } else {
1450         if (fault & (VM_FAULT_SIGBUS|VM_FAULT_HWPOISON|
1451                   VM_FAULT_HWPOISON_LARGE))
1452             do_sigbus(regs, error_code, address, fault);
1453         else if (fault & VM_FAULT_SIGSEGV)
1454             bad_area_nosemaphore(regs, error_code, address);
1455         else
1456             BUG();
1457     }
1458 }
1459 }

```

Listing 6-12: arch/x86/mm/fault.c: do_user_addr_fault() post-fault handling

This logic behaves according to the enum `vm_fault_reason` returned by `handle_mm_fault()`, with special handling for signals. These form a bitmap and thus can be combined:

6.1.3 Success conditions

- `VM_FAULT_COMPLETED` – The page fault is complete and has been entirely resolved with `struct mm_struct->mmap_lock` released. This occurs in a specific situation – on the first fault attempt with retry allowed (via `FAULT_FLAG_ALLOW_RETRY` and tested in `fault_flag_allow_retry_first()`) on a shared page write fault which dirties the page in `fault_dirty_shared_page()` and checks whether the unlock can occur in `maybe_unlock_mmap_for_io()` after balancing dirty pages via `balance_dirty_pages_ratelimited()`. We discuss the dirty shared page

faulting logic and `FAULT_FLAG_ALLOW_RETRY` logic below and writeback in the page cache chapter.

- `~VM_FAULT_ERROR` – The page fault is complete and has been resolved, however the `mmap_lock` is still held and must be released. `VM_FAULT_ERROR` is a bitmask of the error conditions described below.

6.1.4 Informational

- `VM_FAULT_MAJOR` – Indicates that a *major* fault has occurred. A *major* fault is one that occurs when the memory being faulted is not present in RAM – typically this means an expensive disk access is required – either by swapping in memory or loading a page into the page cache. A *minor* page fault is any other page fault.
- `VM_FAULT_NOPAGE` – Indicates that the PTE has been populated but we cannot determine to which folio it points, i.e. the `struct vm_fault->page` field is unspecified (we will take a closer look at this data structure in section 6.2, but broadly it contains page fault state). Typically used when a race condition replaces the PTE entry or the PTE was never intended to point to a data page in the first place.
- `VM_FAULT_WRITE` – Indicates that a write fault occurred. This is only used by the out-of-scope *Kernel Samepage Merging (KSM)* logic, which was removed shortly after kernel v6 (in commit `cb8d86331343: mm: remove VM_FAULT_WRITE`).
- `VM_FAULT_LOCKED` – Indicates that the faulted-in folio has been marked locked, i.e. had its `PG_locked` folio flag set.
- `VM_FAULT_DONE_COW` – Indicates the VMA-specific fault handler has already handled the Copy-on-Write operation. Used only by the *Direct Access for files (DAX)* which is out of scope for the book.
- `VM_FAULT_NEEDSYNC` – Indicates a DAX mapping requires synchronisation. Discussion of DAX is out of scope for the book.

6.1.5 Error conditions

- `VM_FAULT_RETRY` – *mmap_lock released* – Indicates that an attempt has been made to acquire locks but failed, so rather than block on them, the function has exited early to permit a retry. This only occurs once, and only if `FAULT_FLAG_ALLOW_RETRY` was set. On the second attempt, `FAULT_FLAG_TRIED` is set to indicate that it has been tried once.
- `VM_FAULT_OOM` – *mmap_lock released* – The system lacks sufficient memory to allocate the memory and any direct reclaim performed has been insufficient to change this fact – the Out of Memory (OOM) killer must be invoked to free some up (see the chapter on this for more details).
- `VM_FAULT_SIGBUS` – Indicates that an access was made to an invalid address in a mapping (e.g. reading past the extant pages of a memory-mapped file). the process will be sent the `SIGBUS` signal.

- `VM_FAULT_SIGSEGV` – Indicates that an access was made to either unmapped memory or memory for which the user does not have permission. The process will be sent the `SIGSEGV` signal.
- `VM_FAULT_HWPOISON`, `VM_FAULT_HWPOISON_LARGE` – Indicates that the memory being accessed has been marked as hardware ‘poisoned’ as part of the `CONFIG_MEMORY_FAILURE` functionality. Discussion of this is out of scope for the book.
- `VM_FAULT_FALLBACK` – Indicates that a huge page fault failed and should fallback to using ordinary base pages. We defer discussion of huge pages to the huge page chapter so won’t examine this case in detail here.

We examine the post-fault logic in Figure 6-2.

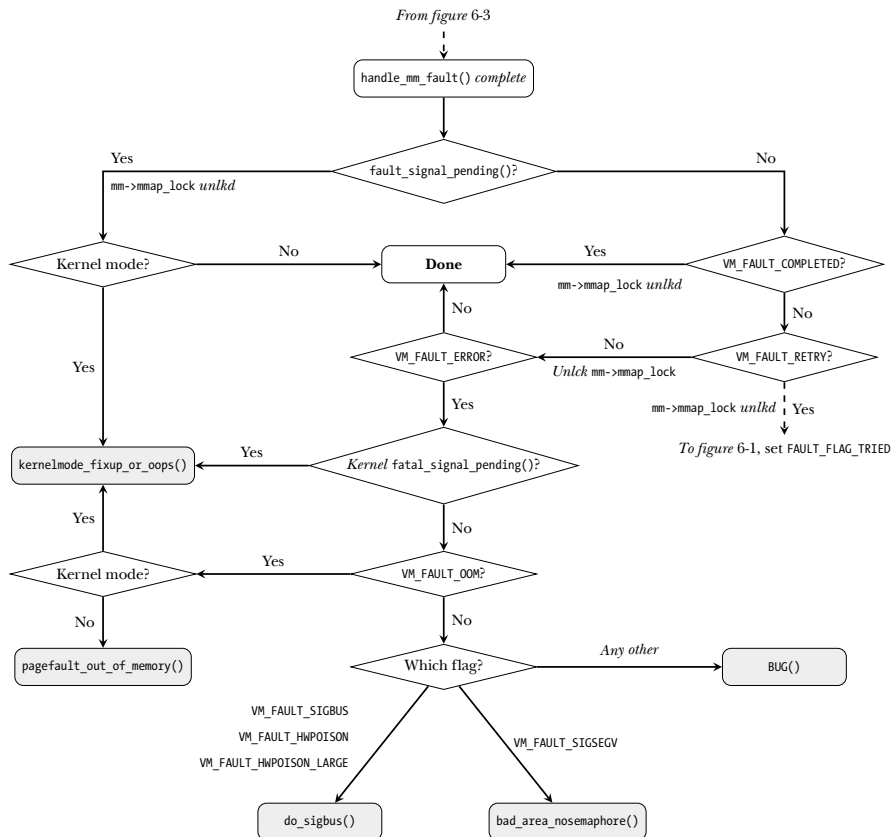


Figure 6-2: Overview of post-fault handling in `do_user_addr_fault()`

Note that after the page fault handler returns and the process can still be scheduled (i.e. is not subject to a segfault) the access will be tried again. The page fault handler will, on a successful page fault, have updated the page tables and thus the access will now succeed.

We start by checking whether a fault-specific signal is pending for the process via `fault_signal_pending()` – this is specific to the case where the generic page fault handler has returned `VM_FAULT_RETRY` and intended to break the process of retrying the page fault if a signal is pending as shown in Listing 6-13.

```

423 /*
424  * This should only be used in fault handlers to decide whether we
425  * should stop the current fault routine to handle the signals
426  * instead, especially with the case where we've got interrupted with
427  * a VM_FAULT_RETRY.
428  */
429 static inline bool fault_signal_pending(vm_fault_t fault_flags,
430                                         struct pt_regs *regs)
431 {
432     return unlikely((fault_flags & VM_FAULT_RETRY) &&
433                     (fatal_signal_pending(current) ||
434                      (user_mode(regs) && signal_pending(current))));
435 }
```

Listing 6-13: include/linux/sched/signal.h: `fault_signal_pending()`

If we are in kernel mode and a *fatal* signal (i.e. `SIGKILL`) is pending for this process or if in user mode and *any* signal is pending, then we should break out to handle this. In the kernel case, we ultimately trigger `kernelmode_fixup_or_oops()` as the page fault must be handled for forward progress, user mode can simply be allowed to page fault again.

If there is no fault signal pending, we check to see whether `handle_mm_fault()` reports that the page fault handling is complete via `VM_FAULT_COMPLETED` – this also means the read lock taken on the `struct mm_struct->mmap_lock` semaphore will have been released so we can simply exit.

If not, we check to see whether the `VM_FAULT_RETRY` flag was set, if so this indicates both that the page faulting must be retried and that the `mmap_lock` has been released. This sets the `FAULT_FLAG_TRIED` flag on the next `handle_mm_fault()` to indicate to the generic handler that we have retried already. See section 6.2 for details on why and when we specify this.

Next we release the lock unconditionally and determine whether we are dealing with an error or not by comparing the fault handler return value with `VM_FAULT_ERROR` – if not, then we simply return.

Otherwise we examine the different possible error states:

- A fatal signal (i.e. `SIGKILL`) is pending and we are in kernel mode – if we are in user mode we can simply carry on and defer this, however in kernel we must resolve it immediately as we can't risk returning from where the kernel instantiated the user memory access with it still pending. In this case we invoke `kernelmode_fixup_or_oops()` to handle the issue.
- `VM_FAULT_OOM` – There is insufficient memory to service the request, even with direct reclaim. If in kernel mode we have no choice but to invoke

the `kernelmode_fixup_or_oops()` handler, however in usermode we can invoke the Out of Memory (OOM) Killer via `pagefault_out_of_memory()`. See the OOM killer chapter for more on this.

- `VM_FAULT_SIGBUS`, `VM_FAULT_HWPOISON`, `VM_FAULT_HWPOISON_LARGE` – Raise a SIGBUS signal to the user process via `do_sigbus()`.
- `VM_FAULT_SIGSEGV` – Raise a SIGSEGV signal to the user process via `bad_area_nosemaphore()`.
- *Any other flag* – This should not occur, the only means by which it could other than a corrupted return value would be for handler to return `VM_FAULT_FALLBACK`, however this is should not be possible here. We catch either case with an oops-causing `BUG()`.

6.2 Page fault handling

We now reach the core of page fault handling in the kernel – the generic page fault handler used on all architectures – `handle_mm_fault()`. This is one of the most important functions in the entire kernel, and the place which userland memory is actually mapped.

We have explored the enum `vm_fault_reason` return types above – these are used both internally within the handler as well as to indicate to the caller what the outcome of the page fault was.

Let's examine the enum `fault_flag` bitmap flags that are passed *in* to the function (and threaded through the fault logic itself) to indicate both the type of page fault to be handled and properties of it:

- `FAULT_FLAG_WRITE` – Indicates that the fault was a write protection fault, i.e. the user attempted to write to a read-only mapping (which may or may not be a Copy on write page).
- `FAULT_FLAG_MKWRITE` – *internal* – Indicates that the fault has resulted in the need for the page table mapping to be made writeable for a VMA where `struct vm_area_struct->vm_ops->page_mkwrite()` or `struct vm_area_struct->vm_ops->pfn_mkwrite()` have been specified for the VMA and invoked by the fault handler. Set internally by `do_page_mkwrite()` and `wp_pfn_shared()`.
- `FAULT_FLAG_ALLOW_RETRY` – Indicates that, if a folio lock cannot be obtained, to simply abort the faulting logic and to return `VM_FAULT_RETRY` to indicate that the fault should be tried again, this time with `FAULT_FLAG_TRIED` set to indicate that the fault has already been tried. Setting this on retry means that a retry will only occur once, as checked by `fault_flag_allow_retry_first()`.

This is checked by `__folio_lock_or_retry()`, invoked in turn by `lock_page_or_retry()` as well as `maybe_unlock_mmap_for_io()` which itself is invoked in turn by `lock_folio_maybe_drop_mmap()`.

This is only applicable in places where a folio lock is obtained, i.e. file-backed folios when making the disk and memory representation consistent with one another, either in `do_swap_page()` (covered in the swap

chapter), DAX logic (out of scope), or most likely, on file-backed page fault via `filemap_fault()` (see the page cache chapter for more on this.). These will release the read lock on the semaphore `struct mm_struct->mmap_lock` unless `FAULT_FLAG_RETRY_NOWAIT` is specified (described below). Use of this flag can help avoid `mmap_lock` contention while attempting to lock the folio.

- **FAULT_FLAG_RETRY_NOWAIT** – A flag used by the Get User Pages (GUP) logic when `FOLL_NOWAIT` is specified (see section 8.1.2 for more on this) as well as a number of drivers and the s390 architecture. This further modifies the `FAULT_FLAG_ALLOW_RETRY` flag to indicate that a retry should be permitted but also for `__folio_lock_or_retry()` and `lock_folio_maybe_drop_mmap()` to not drop the `mmap_lock` nor wait to release the folio lock. This prevents further sleeping when trying to unlock the folio (and later reacquire the `mmap_lock`) while waiting on the lock hence ‘no wait’. This is not used as part of the core faulting in logic described in the previous section but rather in other invocations of the page fault handler such as GUP.
- **FAULT_FLAG_KILLABLE** – Indicates that the process which is faulting is in a state where it can be sent the `SIGKILL` signal if necessary. The net result is that the lock functions described above used in the retry case invoke `__folio_lock_killable()` instead of `__folio_lock()` and `folio_wait_locked_killable()` instead of `folio_wait_locked()` and the GUP function `fixup_user_fault()` explicitly checks for a `SIGKILL` signal.
- **FAULT_FLAG_TRIED** – Set when `FAULT_FLAG_RETRY` has been specified and the fault is being retried. This ensures that a retry only occurs once. See the description of `FAULT_FLAG_ALLOW_RETRY` above.
- **FAULT_FLAG_USER** – Indicates that the fault originates from userland. This flag not being set implies the fault originates from the kernel. Importantly, this does not refer to the *address* which is faulting, which is assumed to be userland. Kernel address faulting is handled separately by `do_kern_addr_fault()`.
- **FAULT_FLAG_REMOTE** – Indicates that we are faulting in memory for a process address space other than ours. This is used by the Get User Pages (GUP) interface and some drivers which are explicitly accessing remote process address spaces and does not form part of ordinary page faulting. See section 8.1.2 for more on GUP. The net result of this is for `handle_mm_fault()` (and for GUP, `vma_permits_fault()`) to check access via `arch_vma_access_permitted()`.
- **FAULT_FLAG_INSTRUCTION** – Indicates that the fault occurred when fetching an instruction, i.e. executing code.
- **FAULT_FLAG_INTERRUPTIBLE** – Indicates that the faulting can be interrupted by non-fatal signals, this is only relevant to `userfaultfd`, which is out of scope for the book which checks for

this in `userfaultfd_get_blocking_state()` which is referenced by `handle_userfault()`.

- `FAULT_FLAG_UNSHARE` – Indicates that the faulting *anonymous* page should be ‘unshared’ if it is a Copy on Write (CoW) page and marked exclusive. This is used by GUP in `faultin_page()`. Note that this flag is invalid if this is a write fault (indicated by `FAULT_FLAG_WRITE`) which would be copying the page anyway.

That this is required is ultimately determined by `gup_must_unshare()`, invoked by `follow_page_pte()`. See section 8.1.2 for a detailed discussion of GUP.

`c89357e27f20`: `mm`: support GUP-triggered unsharing of anonymous pages introduces this feature, explicitly to avoid a situation where a forked folio might get CoW’d, meaning the userland mapping would point at a different folio.

- `FAULT_FLAG_ORIG_PTE_VALID` – Indicates whether the `orig_pte` field in struct `vm_fault` field contains a valid entry. If this flag is set then it does, otherwise it does not.

This is set or cleared in `handle_pte_fault()` depending on whether the struct `vm_fault->pmd` field is empty (as determined by `pmd_none()`).

The default set of flags are set to `FAULT_FLAG_DEFAULT` – this consists of the `FAULT_FLAG_ALLOW_RETRY`, `FAULT_FLAG_KILLABLE` and `FAULT_FLAG_INTERRUPTIBLE` flags. These are set on hardware page fault handling.

The generic page handling function `handle_mm_fault()` is invoked both by the architecture-specific logic described in section 6.1, and by a number of other parts of the kernel too – the out of scope AMD IOMMU driver, HMM, KSM and ftrace components, but more pertinently – GUP via `faultin_page()` and `fixup_user_fault()` (see section 8.1.2 for a detailed description of GUP).

Now, let’s examine `handle_mm_fault()` as shown in Listing 6-14.

```

5123 /*
5124  * By the time we get here, we already hold the mm semaphore
5125  *
5126  * The mmap_lock may have been released depending on flags and our
5127  * return value. See filemap_fault() and __folio_lock_or_retry().
5128  */
5129 vm_fault_t handle_mm_fault(struct vm_area_struct *vma, unsigned long address,
5130                          unsigned int flags, struct pt_regs *regs)
5131 {
5132     vm_fault_t ret;
5133
5134     __set_current_state(TASK_RUNNING);
5135
5136     count_vm_event(PGFAULT);
5137     count_memcg_event_mm(vma->vm_mm, PGFAULT);
5138
5139     /* do counter updates before entering really critical section. */

```

```

5140     check_sync_rss_stat(current);
5141
5142     if (!arch_vma_access_permitted(vma, flags & FAULT_FLAG_WRITE,
5143                                     flags & FAULT_FLAG_INSTRUCTION,
5144                                     flags & FAULT_FLAG_REMOTE))
5145         return VM_FAULT_SIGSEGV;
5146
5147     /*
5148      * Enable the memcg OOM handling for faults triggered in user
5149      * space. Kernel faults are handled more gracefully.
5150      */
5151     if (flags & FAULT_FLAG_USER)
5152         mem_cgroup_enter_user_fault();
5153
5154     if (unlikely(is_vm_hugetlb_page(vma)))
5155         ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
5156     else
5157         ret = __handle_mm_fault(vma, address, flags);
5158
5159     if (flags & FAULT_FLAG_USER) {
5160         mem_cgroup_exit_user_fault();
5161         /*
5162          * The task may have entered a memcg OOM situation but
5163          * if the allocation error was handled gracefully (no
5164          * VM_FAULT_OOM), there is no need to kill anything.
5165          * Just clean up the OOM state peacefully.
5166          */
5167         if (task_in_memcg_oom(current) && !(ret & VM_FAULT_OOM))
5168             mem_cgroup_oom_synchronize(false);
5169     }
5170
5171     mm_account_fault(regs, address, flags, ret);
5172
5173     return ret;
5174 }

```

Listing 6-14: mm/memory.c: `handle_mm_fault()`

This starts by setting the task to the TASK_RUNNING state, as the process is considered to be running as it services the page fault, no matter its prior scheduler state.

There's numerous statistical updates, out of scope for this section cgroup handling (see the cgroup chapter for a detailed examination), leaving us with three core steps:

- `arch_vma_access_permitted()` – We perform architecture-specific checks to ensure the VMA is accessible parameterised by the fault being write, execute and/or remote.

- `__handle_mm_fault()` – This is the core of the generic page fault handler, which we will examine shortly.
- After the fault is complete, we perform *accounting* via `mm_account_fault()` – this tracks major (the page was not present in RAM at all and likely had to be read from disk) and minor faults (the page was present in memory and at worst had to be allocated by the physical allocator) as well as tracking performance statistics.

Throughout the generic page fault handler, `__handle_mm_fault()` threads state through a struct `vm_fault` object as shown in Listing 6-15.

```

481 struct vm_fault {
482     const struct {
483         struct vm_area_struct *vma;      /* Target VMA */
484         gfp_t gfp_mask;                  /* gfp mask to be used for
allocations */
485         pgoff_t pgoff;                    /* Logical page offset based
on vma */
486         unsigned long address;            /* Faulting virtual address -
masked */
487         unsigned long real_address;       /* Faulting virtual address -
unmasked */
488     };
489     enum fault_flag flags;                /* FAULT_FLAG_xxx flags
* XXX: should really be 'const' */
490     pmd_t *pmd;                           /* Pointer to pmd entry matching
* the 'address' */
491     pud_t *pud;                           /* Pointer to pud entry matching
* the 'address'
* */
492     union {
493         pte_t orig_pte;                  /* Value of PTE at the time of fault
*/
494         pmd_t orig_pmd;                  /* Value of PMD at the time of fault,
* used by PMD fault only.
* */
495     };
496     struct page *cow_page;                /* Page handler may use for COW fault
*/
497     struct page *page;                    /* ->fault handlers should return a
* page here, unless VM_FAULT_NOPAGE
* is set (which is also implied by
* VM_FAULT_ERROR).
* */
498     /* These three entries are valid only while holding ptl lock */
499     pte_t *pte;                           /* Pointer to pte entry matching
* the 'address'. NULL if the page

```

```

512                                     * table hasn't been allocated.
513                                     */
514     spinlock_t *ptl;                /* Page table lock.
515                                     * Protects pte page table if 'pte'
516                                     * is not NULL, otherwise pmd.
517                                     */
518     pgtable_t prealloc_pte;         /* Pre-allocated pte page table.
519                                     * vm_ops->map_pages() sets up a page
520                                     * table from atomic context.
521                                     * do_fault_around() pre-allocates
522                                     * page table to avoid allocation from
523                                     * atomic context.
524                                     */
525 };

```

Listing 6-15: include/linux/mm.h: struct vm_fault

This consists of a set of constant fields which define the page fault and do not change throughout – the `vma` field containing the struct `vm_area_struct` (VMA) that we are faulting pages for, `gfp_mask` specifying the GFP mask to be applied to *file-backed* allocations made during the page fault (see the chapter on physical memory for a deep dive on GFP flags) as obtained from `__get_fault_gfp_mask()`.

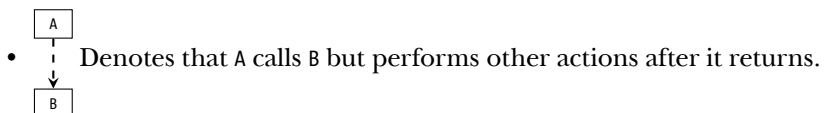
This defaults to `GFP_KERNEL`, but will otherwise be set to the struct `address_space->gfp_mask` field bitwise combined with `__GFP_FS` and `__GFP_IO`. Next, `pgoff` specifies the page offset of this faulting page within the VMA, as determined by `linear_page_index()` (see listing 7-23 and discussion around it for details as to how). `address` contains the page-aligned faulting address and `real_address` the unaligned one.

The rest of the fields can vary throughout the page fault – `flags` contains the previously described fault flags, `pmd` and `pud` containing pointers to the PMD and PUD entries which may or may not need to be allocated for the mapping (see the virtual memory chapter for more on page table structure), and `pte` containing a reference to the PTE entry if the page table lock `ptl` has been acquired.

Finally, either `orig_pte` or `orig_pmd` contain the original PTE or PMD at the time of fault if required, `page` contains the physical struct page allocated or assigned to the mapping, `cow_page` contains the newly CoW'd page if a Copy-on-Write operation has occurred and `prealloc_pte` contains a pre-allocated PTE page table if one has been allocated.

We examine the generic page fault mechanism from top-down starting in Figure 6-3.

Note that:



- This diagram elides huge page, device mapping and other non-core functionality, assumes x86-64 architecture and simplifies aspects of the logic for clarity.

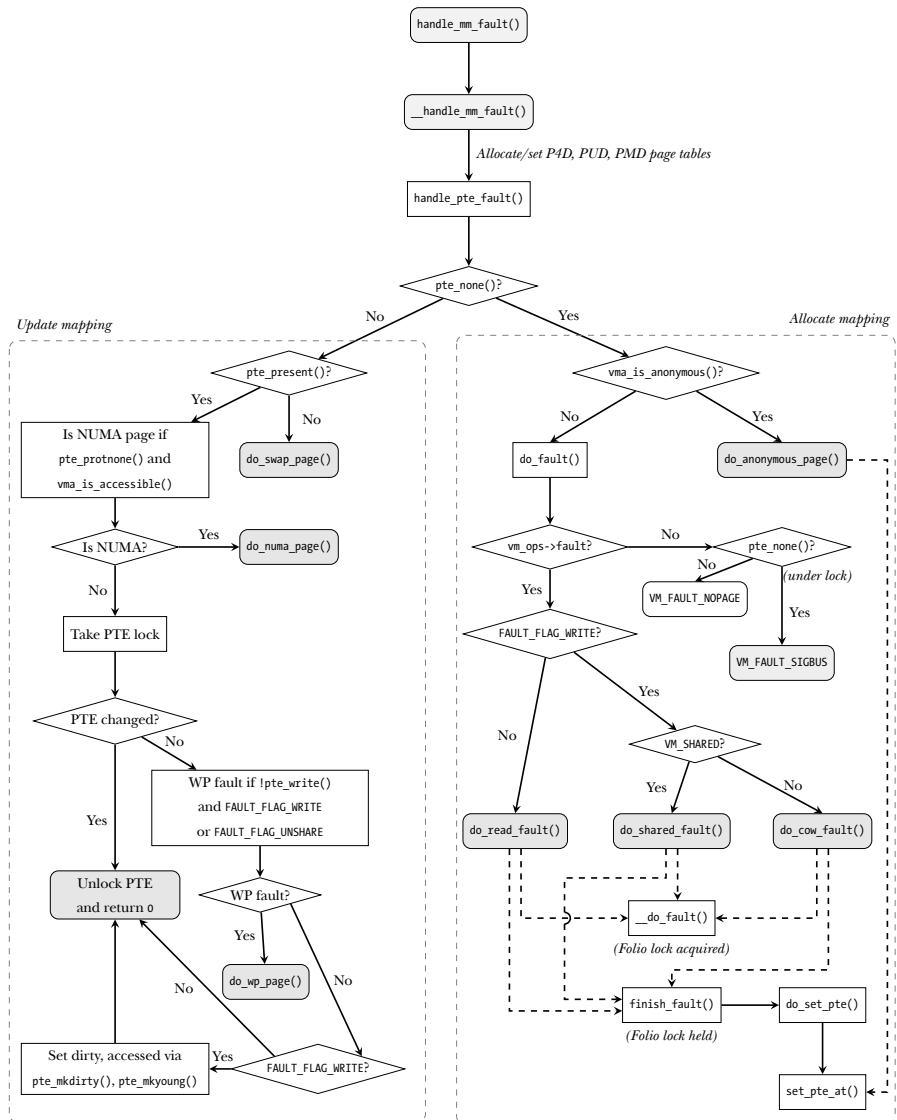


Figure 6-3: Overview of simplified non-huge generic page fault logic

Examining the core function, `_handle_mm_fault()`, eliding huge page handling which we address in the huge page chapter as shown in Listing 6-16.

4961 /*

4962 * By the time we get here, we already hold the mm semaphore


```

4963  *
4964  * The mmap_lock may have been released depending on flags and our
4965  * return value. See filemap_fault() and __folio_lock_or_retry().
4966  */
4967  static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
4968                                     unsigned long address, unsigned int flags)
4969  {
4970      struct vm_fault vmf = {
4971          .vma = vma,
4972          .address = address & PAGE_MASK,
4973          .real_address = address,
4974          .flags = flags,
4975          .pgoff = linear_page_index(vma, address),
4976          .gfp_mask = __get_fault_gfp_mask(vma),
4977      };
4978      struct mm_struct *mm = vma->vm_mm;
4979      unsigned long vm_flags = vma->vm_flags;
4980      pgd_t *pgd;
4981      p4d_t *p4d;
4982      vm_fault_t ret;
4983
4984      pgd = pgd_offset(mm, address);
4985      p4d = p4d_alloc(mm, pgd, address);
4986      if (!p4d)
4987          return VM_FAULT_OOM;
4988
4989      vmf.pud = pud_alloc(mm, p4d, address);
4990      if (!vmf.pud)
4991          return VM_FAULT_OOM;
4992      :
5019      vmf.pmd = pmd_alloc(mm, vmf.pud, address);
5020      if (!vmf.pmd)
5021          return VM_FAULT_OOM;
5022      :
5033      vmf.orig_pmd = *vmf.pmd;
5034      :
5059      return handle_pte_fault(&vmf);
5060  }

```

Listing 6-16: mm/memory.c: __handle_mm_fault()

This sets up the struct `vm_fault` object as previously described, then walks the page tables to the address, allocating new page tables if necessary via `p4d_alloc()`, `pud_alloc()` and `pmd_alloc()`. We examine page table structure and allocation in detail in the virtual memory chapter, however to be explicit about how we are allocating page tables here (if allocation is required) it is done so as follows (for x86-64):

P4D – Allocated via `p4d_alloc_one()` with the `GFP_KERNEL_ACCOUNT` GFP flags set (see the physical memory chapter for a discussion of GFP flags) obtaining a physical page via `get_zeroed_page()`. Sets the relevant PGD entry to point at it.

PUD – Allocated via `__pud_alloc_one()` with the `GFP_PGTABLE_USER` GFP flags set obtaining a physical page via `get_zeroed_page()`. Sets the relevant P4D entry to point at it.

PMD – Allocated via `pmd_alloc_one()` with the `GFP_PGTABLE_USER` GFP flags set obtaining a physical page via `alloc_pages()`. Sets the relevant PUD entry to point at it.

Note that in each potential page table allocation we explicitly check to see whether the allocation succeeded, if not we return `VM_FAULT_OOM` to indicate that we have insufficient memory to fault this memory in.

The remainder of the fault handling logic is deferred to `handle_pte_fault()` (eliding huge page logic) as shown in Listing 6-17.

```

4860 static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
4861 {
4862     pte_t entry;
4863
4864     if (unlikely(pmd_none(*vmf->pmd))) {
4865         :
4871         vmf->pte = NULL;
4872         vmf->flags &= ~FAULT_FLAG_ORIG_PTE_VALID;
4873     } else {
4874         :
4894         vmf->pte = pte_offset_map(vmf->pmd, vmf->address);
4895         vmf->orig_pte = *vmf->pte;
4896         vmf->flags |= FAULT_FLAG_ORIG_PTE_VALID;
4897
4898         /*
4899          * some architectures can have larger ptes than wordsize,
4900          * e.g. ppc44x-defconfig has CONFIG_PTE_64BIT=y and
4901          * CONFIG_32BIT=y, so READ_ONCE cannot guarantee atomic
4902          * accesses. The code below just needs a consistent view
4903          * for the ifs and we later double check anyway with the
4904          * ptl lock held. So here a barrier will do.
4905          */
4906         barrier();
4907         if (pte_none(vmf->orig_pte)) {
4908             pte_unmap(vmf->pte);
4909             vmf->pte = NULL;
4910         }
4911     }
4912
4913     if (!vmf->pte) {

```

```

4914         if (vma_is_anonymous(vmf->vma))
4915             return do_anonymous_page(vmf);
4916         else
4917             return do_fault(vmf);
4918     }
4919
4920     if (!pte_present(vmf->orig_pte))
4921         return do_swap_page(vmf);
4922
4923     if (pte_protnone(vmf->orig_pte) && vma_is_accessible(vmf->vma))
4924         return do_numa_page(vmf);
4925
4926     vmf->ptl = pte_lockptr(vmf->vma->vm_mm, vmf->pmd);
4927     spin_lock(vmf->ptl);
4928     entry = vmf->orig_pte;
4929     if (unlikely(!pte_same(*vmf->pte, entry))) {
4930         update_mmu_tlb(vmf->vma, vmf->address, vmf->pte);
4931         goto unlock;
4932     }
4933     if (vmf->flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) {
4934         if (!pte_write(entry))
4935             return do_wp_page(vmf);
4936         else if (likely(vmf->flags & FAULT_FLAG_WRITE))
4937             entry = pte_mkdirty(entry);
4938     }
4939     entry = pte_mkyoung(entry);
4940     if (ptep_set_access_flags(vmf->vma, vmf->address, vmf->pte, entry,
4941                             vmf->flags & FAULT_FLAG_WRITE)) {
4942         update_mmu_cache(vmf->vma, vmf->address, vmf->pte);
4943     } else {
4944         /* Skip spurious TLB flush for retried page fault */
4945         if (vmf->flags & FAULT_FLAG_TRIED)
4946             goto unlock;
4947
4948         /*
4949          * This is needed only for protection faults but the arch code
4950          * is not yet telling us if this is a protection fault or not.
4951          * This still avoids useless tlb flushes for .text page faults
4952          * with threads.
4953          */
4954         if (vmf->flags & FAULT_FLAG_WRITE)
4955             flush_tlb_fix_spurious_fault(vmf->vma, vmf->address);
4956     }
4957 unlock:
4958     pte_unmap_unlock(vmf->pte, vmf->ptl);
4959     return 0;

```

Listing 6-17: mm/memory.c: `handle_pte_fault()`

We start by checking whether the PMD is newly allocated (and thus empty), if so, we mark the struct `vm_fault->pte` field to NULL and clear the `FAULT_FLAG_ORIG_PTE_VALID` flag to indicate that no PTE is yet allocated.

Otherwise, the mapping is valid up to the PTE level, and we set the `pte` field to the point at the PTE entry, via `pte_offset_map()` (note that for a 64-bit architecture no actual ‘mapping’ takes place), take a copy of it at this point in time in `orig_pte` and set the `FAULT_FLAG_ORIG_PTE_VALID` flag.

If the PTE entry is empty (as determined via `pte_none()`), a PTE entry might be empty but not be fully zeroed as a mask is applied), we set the PTE field to NULL.

We are now at a significant point in the page fault handling – if there is no PTE entry this means nothing is yet mapped (not even a Copy on Write mapping) and the fault is about allocating memory or looking up a page cache entry, as determined by `vma_is_anonymous()` as shown in Listing 6-18.

```

629 static inline bool vma_is_anonymous(struct vm_area_struct *vma)
630 {
631     return !vma->vm_ops;
632 }
```

Listing 6-18: include/linux/mm.h: `vma_is_anonymous()`

This is a *key* definition – if a struct `vm_area_struct` (VMA) lacks customised struct `vm_operations_struct` `vm_ops`, then it is anonymous, i.e. only non-anonymous (typically file-backed) memory mappings require customised VMA operation handling.

If this function determines that the VMA is indeed anonymous, then the fault is handled by `do_anonymous_page()`, if non-anonymous then it is handled by `do_fault()`. We examine both shortly

Otherwise we check to see whether the PTE entry is non-empty but not present (i.e. lacking the `_PAGE_BIT_PRESENT` bit), if so then it is a swapped out page, handled by `do_swap_page()` which is discussed in the swap chapter.

Next we check to see if the page has been unmapped as part of NUMA balancing, which is handled by `do_numa_page()` (see the NUMA chapter for a detailed explanation of this).

Finally, we reach the Copy on Write handler – `do_wp_page()` – this handles write-protect faults, including the vitally important case of handling a Copy on Write (CoW) page faulting.

6.2.1 Edge cases

We have moved very quickly over the key logic here, but there are some edge cases we must consider. After checking for the NUMA page case, we acquire the PTE page table lock via `pte_lockptr()`, then check to see if the fault has been handled by a racing fault handler – if so we simply unlock the PTE and exit (`update_mmu_tlb()` is not relevant for non-MIPS/xtensa architectures).

Next, we have logic dealing with the case where either a read fault or a spurious write fault has occurred. In this case, if a write fault has arisen,

the entry we are building based on the original PTE is marked dirty via `pte_mkdirty()`.

In any case, this entry is then marked accessed via `pte_mkyoung()` and then the flags are possibly changed via `ptep_set_access_flags()` which indicates whether they have changed and, if dirty, sets them (it is not worth updating should other flags have changed).

We then reach specific handling for a spurious fault caused by an out-dated TLB entry* – on x86-64 this requires no additional handling and `flush_tlb_fix_spurious_fault()` is therefore a no-op.

6.2.2 Page flags

We will examine each of these in turn. Note that in each case, the *page table flags* are determined via `vm_get_page_prot()` (initially at least, possibly modified if a non-anonymous VMA requires write notification, we will discuss this in section 6.9) and stored in the VMA's `vm_page_prot` field – see figure 3.13 for details as to how these are mapped but broadly these are determined by the presence or lack of `VM_SHARED`, `VM_READ`, `VM_WRITE` and `VM_EXEC`.

Importantly – All mappings which are not `VM_SHARED` are marked read-only by default and are thus copy-on-write mappings. This applies both to anonymous mappings and `MAP_PRIVATE` file-backed mappings.

6.3 Anonymous page fault

At this point, the page fault has occurred in a memory range described by an anonymous VMA, and that VMA has no Copy on Write mapping or any other established and requires a page to be allocated.

This is handled via `do_anonymous_page()` (eliding out of scope huge page, userfaultfd and cgroup logic) as shown in Listing 6-19.

```

4026 /*
4027  * We enter with non-exclusive mmap_lock (to exclude vma changes,
4028  * but allow concurrent faults), and pte mapped but not yet locked.
4029  * We return with mmap_lock still held, but pte unmapped and unlocked.
4030  */
4031 static vm_fault_t do_anonymous_page(struct vm_fault *vmf)
4032 {
4033     struct vm_area_struct *vma = vmf->vma;
4034     struct page *page;
4035     vm_fault_t ret = 0;
4036     pte_t entry;
4037
4038     /* File mapping without ->vm_ops ? */
4039     if (vma->vm_flags & VM_SHARED)

```

*. The TLB is the Transaction Lookaside Buffer – a cache that maps virtual addresses to physical ones with all relevant page table flags. This being outdated can cause faults to occur when the mapping in the page table is in fact valid.

```

4040         return VM_FAULT_SIGBUS;
4041
4042     /*
4043      * Use pte_alloc() instead of pte_alloc_map(). We can't run
4044      * pte_offset_map() on pmds where a huge pmd might be created
4045      * from a different thread.
4046      *
4047      * pte_alloc_map() is safe to use under mmap_write_lock(mm) or when
4048      * parallel threads are excluded by other means.
4049      *
4050      * Here we only have mmap_read_lock(mm).
4051      */
4052     if (pte_alloc(vma->vm_mm, vmf->pmd))
4053         return VM_FAULT_OOM;
4054     :
4055
4056     /* Use the zero-page for reads */
4057     if (!(vmf->flags & FAULT_FLAG_WRITE) &&
4058         !mm_forbids_zeropage(vma->vm_mm)) {
4059         entry = pte_mkspecial(pfn_pte(my_zero_pfn(vmf->address),
4060             vma->vm_page_prot));
4061         vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd,
4062             vmf->address, &vmf->ptl);
4063         if (!pte_none(*vmf->pte)) {
4064             update_mmu_tlb(vma, vmf->address, vmf->pte);
4065             goto unlock;
4066         }
4067         ret = check_stable_address_space(vma->vm_mm);
4068         if (ret)
4069             goto unlock;
4070     }
4071     goto setpte;
4072 }
4073
4074 /* Allocate our own private page. */
4075 if (unlikely(anon_vma_prepare(vma)))
4076     goto oom;
4077 page = alloc_zeroed_user_highpage_movable(vma, vmf->address);
4078 if (!page)
4079     goto oom;
4080 :
4081
4082 /*
4083  * The memory barrier inside __SetPageUptodate makes sure that
4084  * preceding stores to the page contents become visible before
4085  * the set_pte_at() write.
4086  */
4087 __SetPageUptodate(page);
4088

```

```

4099     entry = mk_pte(page, vma->vm_page_prot);
4100     entry = pte_sw_mkyoung(entry);
4101     if (vma->vm_flags & VM_WRITE)
4102         entry = pte_mkwwrite(pte_mkdirty(entry));
4103
4104     vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, vmf->address,
4105                                   &vmf->ptl);
4106     if (!pte_none(*vmf->pte)) {
4107         update_mmu_cache(vma, vmf->address, vmf->pte);
4108         goto release;
4109     }
4110
4111     ret = check_stable_address_space(vma->vm_mm);
4112     if (ret)
4113         goto release;
4114     :
4122     inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
4123     page_add_new_anon_rmap(page, vma, vmf->address);
4124     lru_cache_add_inactive_or_unevictable(page, vma);
4125 setpte:
4126     set_pte_at(vma->vm_mm, vmf->address, vmf->pte, entry);
4127
4128     /* No need to invalidate - it was non-present before */
4129     update_mmu_cache(vma, vmf->address, vmf->pte);
4130 unlock:
4131     pte_unmap_unlock(vmf->pte, vmf->ptl);
4132     return ret;
4133 release:
4134     put_page(page);
4135     goto unlock;
4136 oom_free_page:
4137     put_page(page);
4138 oom:
4139     return VM_FAULT_OOM;
4140 }

```

Listing 6-19: mm/memory.c: `do_anonymous_page()`

We start by checking whether the struct `vm_area_struct` (VMA) has the `VM_SHARED` flag set – if so, something has gone wrong, as this flag should not be set if the VMA is anonymous.

This might seem counter-intuitive, as memory can indeed be mapped as both ‘anonymous’ and shared via `MAP_SHARED` and `MAP_ANONYMOUS`, however doing so does not result in an anonymous mapping, rather it memory-maps

an unlinked tmpfs clone of `/dev/zero` which is assigned its own inode. (see Chapter 5 for more on memory mapping)*.

Next we go right ahead and allocate a new PTE (if necessary) via `pte_alloc()`, assigning it to the PMD entry referenced in `struct vm_fault->pmd` if so. If this fails, we report out of memory via `VM_FAULT_OOM`.

6.3.1 Zero page

A very useful concept within the page faulting mechanism is the idea of the ‘zero page’. This is a page of pre-allocated and zeroed memory that is mapped as a Copy on Write page upon *read* fault.

This way, no actual physical memory need be allocated for anonymous read faults and the actual allocation can be deferred until it is written to.

This is determined by the fact that a write fault has not occurred (i.e. `FAULT_FLAG_WRITE` was not set), noting that we know this is not a shared mapping as we have checked this above. It is also mediated by `mm_forbids_zeropage()`, however this is only prohibited on the s390 architecture, so not relevant to our focused upon architecture, x86-64.

The PFN of the zero page is determined via `my_zero_pfn()` which simply references an external variable `zero_pfn` as shown in Listing 6-20.

```

1221 static inline unsigned long my_zero_pfn(unsigned long addr)
1222 {
1223     extern unsigned long zero_pfn;
1224     return zero_pfn;
1225 }
```

Listing 6-20: include/linux/pgtable.h: `my_zero_pfn()`

this PFN is set up in `init_zero_pfn()` which obtains the actual zero page via `ZERO_PAGE()`, then uses `page_to_pfn()` to convert it to a PFN as shown in Listing 6-21.

```

164 /*
165  * CONFIG_MMU architectures set up ZERO_PAGE in their paging_init()
166  */
167 static int __init init_zero_pfn(void)
168 {
169     zero_pfn = page_to_pfn(ZERO_PAGE(0));
170     return 0;
171 }
172 early_initcall(init_zero_pfn);
```

Listing 6-21: mm/memory.c: `init_zero_pfn()`

*. `mmap_region()` invokes the functions `shmem_zero_setup()`, `shmem_kernel_file_setup()` and `__shmem_file_setup()` in turn to do so. Note that this memory is still swap-backed so will be added to the anonymous LRU list

Using `early_initcall()` to declare that it must be run during early initialisation of the kernel.

The x86-64 architecture establishes `ZERO_PAGE()` in reference to a page of memory reserved in the static `.bss` (therefore zeroed) section of the kernel image, `empty_zero_page` as shown in Listing 6-22.

```

50 /*
51  * ZERO_PAGE is a global shared page that is always zero: used
52  * for zero-mapped memory areas etc..
53  */
54 extern unsigned long empty_zero_page[PAGE_SIZE / sizeof(unsigned long)]
55     __visible;
56 #define ZERO_PAGE(vaddr) ((void)(vaddr),virt_to_page(empty_zero_page))

```

Listing 6-22: `arch/x86/include/asm/pgtable.h: ZERO_PAGE()`

This uses `virt_to_page()` to convert this kernel image address to a struct page.

We convert the PFN obtained from `my_zero_pfn()` to a PTE via `pfn_pte()` and then marked ‘special’ via `pte_mkspecial()`, which simply sets the `_PAGE_SPECIAL` to indicate that this is not an ordinary mapping.

This will be allocated on Copy on Write in `wp_page_copy()` which uses `alloc_zeroed_user_highpage_movable()` to zero it (we touch on Copy on Write pages below in section 6.9).

The PTE table’s lock is then acquired and PTE entry address retrieved via `pte_offset_map_lock()` (note for 64-bit architectures no ‘mapping’ is required).

The MMU TLB is updated for architectures that require this to be done manually via `update_mmu_tlb()`, however this is not needed in the x86-64 architecture.

Finally, we check to see whether the struct `mm_struct` is stable via `check_stable_address_space()` – an ‘unstable’ address space being one where the Out of Memory (OOM) killer has started to reap the address space. If so, a mapping can be pulled out from underneath us, so simply unlock and exit, otherwise we proceed to setting the PTE entry.

6.3.2 *anon_vma preparation*

We must set up the struct `anon_vma` object associated with the VMA if one does not already exist via `anon_vma_prepare()` – this checks the struct `vm_area_struct` (VMA)’s `anon_vma` field, if `NULL` then it allocates a new object or merges with an existing adjacent one via `__anon_vma_prepare()`, and allocates and connects a struct `anon_vma_chain` object to bind the `anon_vma` to the VMA.

See Chapter 7 for a very in-depth explanation of the reverse mapping as a whole.

Note that, while we are setting up an `anon_vma` object here, we have yet to point a folio at it, this is performed later, as described below.

6.3.3 Physical page allocation

The physical page for the mapping is allocated via the physical allocation macro `alloc_zeroed_user_highpage_movable()` (examining the x86-64 specific version) as shown in Listing 6-23.

```
37 #define alloc_zeroed_user_highpage_movable(vma, vaddr) \
38     alloc_page_vma(GFP_HIGHUSER_MOVABLE | __GFP_ZERO, vma, vaddr)
```

Listing 6-23: `arch/x86/include/asm/page.h: alloc_zeroed_user_highpage_movable()`

This invokes `alloc_page_vma()` with `GFP_HIGHUSER_MOVABLE` and `__GFP_ZERO` set, the former being a set of other physical allocation flags specifically intended for userland memory allocation which, for a 64-bit system amount to:

- `__GFP_ZERO` – Zero the memory after allocating it. This is critical for user memory, as failing to do so could result in highly insecure information leakage, not to mention that userland entirely relies upon this.
- `__GFP_MOVABLE` – A key defining characteristic of userland memory is that it can be migrated, i.e. moved around as necessary. Userland does not care which physical address a virtual mapping points at, so we can move as is convenient. See the chapter on migration/compaction for more on when and how we do this.
The net result is that the memory is assigned the migrate type `MIGRATE_MOVABLE`, a heuristic to try to prevent movable and unmovable memory residing in the same page block. See section 2.6.2 for more on this.
- `__GFP_SKIP_KASAN_POISON`, `__GFP_SKIP_KASAN_UNPOISON` – Out of scope hints for the kernel's *KASAN* functionality.
- `__GFP_DIRECT_RECLAIM` – Indicates that the memory can be *directly* reclaimed if memory pressure is so high that free pages in the zone from which it is being allocated falls below the minimum permissible level. See section 2.4 for more on this.
- `__GFP_KSWAPD_RECLAIM` – Indicates that *indirect* reclaim, performed by the `kswapd` kernel thread, will be triggered if memory pressure causes the zone from which this memory is allocated to fall below its low watermark. Again, see section 2.4 for more on this.
- `__GFP_IO` – Indicates that I/O can be performed as a part of this allocation.
- `__GFP_FS` – Indicates that file systems can be utilised as part of the allocation.
- `__GFP_HARDWALL` – Indicates that, when used within a cgroup, the memory's `cpuset.mems` setting should be honoured. See the cgroup chapter for more details on this.

See section 2.6 for an in-depth explanation of GFP flags and chapter 2 in general for a broad examination of physical memory allocation as a whole.

Examining `alloc_page_vma()` itself as shown in Listing 6-24.

```

287 static inline struct page *alloc_page_vma(gfp_t gfp,
288      struct vm_area_struct *vma, unsigned long addr)
289 {
290     struct folio *folio = vma_alloc_folio(gfp, 0, vma, addr, false);
291
292     return &folio->page;
293 }

```

Listing 6-24: include/linux/gfp.h: *alloc_page_vma()*

This invokes `vma_alloc_folio()` which takes into account the NUMA memory policy specified by the user for this VMA and allocates accordingly (see the NUMA chapter for a detailed analysis of this), before ultimately performing the allocation via `__folio_alloc()` which in turn invokes the core physical memory allocation function `__alloc_pages()`.

After the memory is allocated we set the folio flag `PG_uptodate` via `__SetPageUptodate()`, a flag indicating whether memory is synchronised with disk. For anonymous memory, this is trivially true.

6.3.4 Setting up the PTE entry

We then move on to setting up the PTE entry, starting by setting the flags inferred from the `struct vm_area_struct` (VMA)'s `vm_flags` stored in its `vm_page_prot` field via `mk_pte()`, noting that these mappings *always* map Copy on Write (i.e. readonly) by default unless the mapping is shared.

We invoke `pte_sw_mkyoung()`, which sets the accessed page table flag for architectures which do not set this in hardware. However, sensible modern architectures including the focused upon x86-64 do so, so in this case this does nothing.

Next, we adjust the PTE flags if this is a write fault, as in this case we actually do not wish this page to be Copy on Write, since we have just allocated a page for it (a read-only fault should hit the zero page logic described above), so mark the page table mapping both writable, dirty and *soft-dirty* (the page has indeed been ‘touched’ by a write fault) via `pte_mkwrite()` and `pte_mkdirty()`.

A pointer to the PTE entry itself is then placed in the `struct vm_fault->pte` field, with its lock acquired and placed in the `ptl` field, using the `pmd` field to obtain the address of the PTE page which we may have allocated previously.

Because we only just acquired a lock for this PTE, we have to check for race conditions, and thus now check to see if we were beaten to it via `pte_none()`. If so we simply release the physical page and return.

Otherwise, we check whether we are in a ‘stable’ address space (i.e. one not targeted by the OOM killer) as we did in the zero page code path via `check_stable_address_space()`. If not we release the page and abort the operation.

We then increment the anonymous page statistics via `inc_mm_counter_fast()`.

6.3.5 Adding to the reverse mapping and LRU and setting the PTE

As this is user memory, we have a couple of additional tasks to perform – Firstly, we must point the folio at the struct anon_vma object which represents this via page_add_new_anon_rmap() (see listing 7-20). See section 7.0.12 for a detailed examination of this.

Secondly, we must add the folio to the ‘Least Recently Used’ LRU caching mechanism via lru_cache_add_inactive_or_unevictable() – this is a means by which the kernel keeps a track of how recently folios have been accessed when making decisions about which folios to swap out or evict from the page cache. See section 11.2 for a detailed analysis of this.

We are now ready to actually set the PTE entry via set_pte_at() (which calls set_pte() in turn).

Afterwards, we unlock the PTE lock and return – the page has been faulted in, with all page tables correctly set up and the memory added to both the reverse mapping and LRU mechanisms as appropriate.

6.4 Non-anonymous page fault

Non-anonymous page faults are handled by do_fault(), which determines which handler to invoke as shown in Listing 6-25.

```

4609 /*
4610  * We enter with non-exclusive mmap_lock (to exclude vma changes,
4611  * but allow concurrent faults).
4612  * The mmap_lock may have been released depending on flags and our
4613  * return value. See filemap_fault() and __folio_lock_or_retry().
4614  * If mmap_lock is released, vma may become invalid (for example
4615  * by other thread calling munmap()).
4616  */
4617 static vm_fault_t do_fault(struct vm_fault *vmf)
4618 {
4619     struct vm_area_struct *vma = vmf->vma;
4620     struct mm_struct *vm_mm = vma->vm_mm;
4621     vm_fault_t ret;
4622
4623     /*
4624      * The VMA was not fully populated on mmap() or missing VM_DONTEXPAND
4625      */
4626     if (!vma->vm_ops->fault) {
4627         /*
4628          * If we find a migration pmd entry or a none pmd entry, which
4629          * should never happen, return SIGBUS
4630          */
4631         if (unlikely(!pmd_present(*vmf->pmd)))
4632             ret = VM_FAULT_SIGBUS;
4633         else {
4634             vmf->pte = pte_offset_map_lock(vmf->vma->vm_mm,
```

```

4635                                     vmf->pmd,
4636                                     vmf->address,
4637                                     &vmf->ptl);
4638                                     /*
4639                                     * Make sure this is not a temporary clearing of pte
4640                                     * by holding ptl and checking again. A R/M/W update
4641                                     * of pte involves: take ptl, clearing the pte so that
4642                                     * we don't have concurrent modification by hardware
4643                                     * followed by an update.
4644                                     */
4645                                     if (unlikely(pte_none(*vmf->pte)))
4646                                         ret = VM_FAULT_SIGBUS;
4647                                     else
4648                                         ret = VM_FAULT_NOPAGE;
4649
4650                                     pte_unmap_unlock(vmf->pte, vmf->ptl);
4651                                 }
4652     } else if (!(vmf->flags & FAULT_FLAG_WRITE))
4653         ret = do_read_fault(vmf);
4654     else if (!(vma->vm_flags & VM_SHARED))
4655         ret = do_cow_fault(vmf);
4656     else
4657         ret = do_shared_fault(vmf);
4658
4659     /* preallocated pagetable is unused: free it */
4660     if (vmf->prealloc_pte) {
4661         pte_free(vm_mm, vmf->prealloc_pte);
4662         vmf->prealloc_pte = NULL;
4663     }
4664     return ret;
4665 }

```

Listing 6-25: mm/memory.c: do_fault()

This starts with specific handling for an edge case – non-anonymous mappings are assumed to provide their own fault-handling logic (which is often simply delegated to `filemap_fault()`) within the `fault` field of the struct `vm_area_struct` (VMA)’s struct `vm_operations_struct` `vm_ops` field.

If this is not present, something has gone wrong – either we have encountered something broken, in which case we return `VM_FAULT_SIGBUS` to indicate that the SIGBUS signal must be sent to the faulting process, or we have encountered a race condition, in which case we must return `VM_FAULT_NOPAGE` to indicate that we do not know which folio this fault belongs to (i.e. it has been faulted in elsewhere).

If the PMD is not present, then things are certainly broken and thus in this case we unconditionally return `VM_FAULT_SIGBUS`, otherwise we perform the same `pte_none()` check we performed in `handle_pte_fault()`, only this time

having acquired the PTE lock – either returning `VM_FAULT_SIGBUS` if the entry is still empty or `VM_FAULT_NOPAGE` otherwise.

This allows us to account for the situation where the PTE has been read by another fault, cleared before then being being written to as described in the commit `ff09d7ec9786: mm/memory.c: recheck page table entry with page table lock held.`

What remains are the various types of faults which are delegated:

Read fault – `do_read_fault()` is invoked when a read fault arises (i.e. `FAULT_FLAG_WRITE` was not set).

`MAP_PRIVATE` file-backed mapping write fault – `do_cow_fault()` is invoked when a file-backed `MAP_PRIVATE` mapping experiences a page write fault (see Chapter 5 for more on memory mapping in general). In this case, we must treat this as a Copy on Write fault, despite no existing read-only mapping existing, since the semantics of this type of mapping are that the page cache entry must be CoW'd on write.

Shared write fault – `do_shared_fault()` is invoked when a write fault has occurred for a shared mapping (i.e. within a VMA which has the `VM_SHARED` flag set).

After invoking the handlers to service the fault, the `struct vm_fault->prealloc_pte` field is checked to see if any pre-allocated PTE page table allocation remains – if so this is freed before returning the result obtained from the appropriate page fault handler.

Each of these handlers invokes `__do_fault()` to fault-in the memory via the `struct vm_area_struct` (VMA)'s `struct vm_operations_struct vm_ops->fault` handler, which returns a *locked* folio. This populates the appropriate fields in the `struct vm_fault` object, which is then actually applied via `finish_fault()` and the PTE installed (and added to the reverse mapping and LRU lists) via `do_set_pte()`.

6.4.1 Folio locks

When performing these operations, we enter states where a folio might be updated and thus must be considered unavailable for any further operations until the lock is released – this is so the folio remains *stable* throughout the operation, i.e. is not modified in a way that races with the ongoing operation such as removing the mapping or similar changes.

In each of the below operations, the shared `__do_fault()` handler acquires the folio lock, and the shared logic for completing the fault in `finish_fault()` assumes this lock is held throughout.

These locks are acquired via `folio_lock()` or the `struct page` equivalent `lock_page()` and unlocked via `folio_unlock()` or `unlock_page()`.

Page fault handlers may indicate that they have acquired this lock via `VM_FAULT_LOCKED`, and on `fault` or `page_mkwrite` are expected to do so, with special handling for cases where this is not so.

It's important to note that these functions are not simply setting or clearing the folio `PG_locked` flag but sleep and wait on the flag to be cleared.

There is a lot of intricate machinery around these locks and their interaction with page cache operations like writeback, however we defer the discussion of this to the page cache chapter where we can examine it in detail.

We will examine the aforementioned `__do_fault()` and `finish_fault()` shared logic in section 6.8, meanwhile let's examine each of these handlers individually:

6.5 Non-anonymous read page fault

Non-anonymous read faults are handled by `do_read_fault()` as shown in Listing 6-26.

```

4509 static vm_fault_t do_read_fault(struct vm_fault *vmf)
4510 {
4511     vm_fault_t ret = 0;
4512
4513     /*
4514      * Let's call ->map_pages() first and use ->fault() as fallback
4515      * if page by the offset is not ready to be mapped (cold cache or
4516      * something).
4517      */
4518     if (should_fault_around(vmf)) {
4519         ret = do_fault_around(vmf);
4520         if (ret)
4521             return ret;
4522     }
4523
4524     ret = __do_fault(vmf);
4525     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)
4526 ))
4527         return ret;
4528
4529     ret |= finish_fault(vmf);
4530     unlock_page(vmf->page);
4531     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)
4532 ))
4533         put_page(vmf->page);
4534     return ret;
4535 }
```

Listing 6-26: `mm/memory.c: do_read_fault()`

This for the most part simply invokes `__do_fault()` to populate the struct `vm_fault` object and `finish_fault()` to apply the faulted in memory, taking care to handle error states.

The key difference however is in the *fault-around* logic – when reading in from a file, it is useful to try to map *easily* obtained pages if we can on the usually reasonable assumption that subsequent pages are likely to be read next (i.e. only performing *minor* faults).

A typical example of this would be where a number of folios of a file exist in the page cache but are not yet mapped in the faulting VMA. In this case it is relatively cheap to simply map these, but if they had to be retrieved from disk this would not be the case.

This is distinct from read-ahead, where folios are loaded from disk and into the page cache in anticipation of further reads, which is an expensive operation. The fault-around is specified by the filesystem via the struct `vm_area_struct` (VMA)'s struct `vm_operations_struct` `vm_ops->map_pages` handler which is often delegated to the generic `filemap_map_pages()`.

Whether fault-around can occur is determined via `should_fault_around()` and mediated by the static variable `fault_around_bytes`, which defaults to 64 KiB but is configurable via the debugfs tunable `/sys/kernel/debug/fault_around_bytes`.

This value must be equal to a power-of-2, i.e. a page order and if specified otherwise is rounded down to the nearest such value (this is done in `fault_around_bytes_set()`). It must also be no larger than the area which can be mapped by a PTE, i.e. `PTES_PER_PTE` pages, as it does not cross page boundaries.

Important: This means that the range that is faulted ‘around’ truly is around the address. For instance, if 16 pages (the default for a 4 KiB page size) are faulted in, it will be the 16 pages *containing* the faulting address, not those that come immediately afterwards.

E.g. if the faulting address was `0xe000`, then addresses in the range `0` to `0xf000` inclusive would be faulted in, *not* the 16 pages at or beyond `0xe000` (reading *ahead* is what readahead does, and is distinct, discussed in section 9.7 in the page cache chapter).

Examining `should_fault_around()` as shown in Listing 6-27.

```

4496 /* Return true if we should do read fault-around, false otherwise */
4497 static inline bool should_fault_around(struct vm_fault *vmf)
4498 {
4499     /* No ->map_pages? No way to fault around... */
4500     if (!vmf->vma->vm_ops->map_pages)
4501         return false;
4502
4503     if (uffd_disable_fault_around(vmf->vma))
4504         return false;
4505
4506     return fault_around_bytes >> PAGE_SHIFT > 1;
4507 }

```

Listing 6-27: `mm/memory.c: should_fault_around()`

Other than a `userfaultfd`-specific check (out of scope for the book), this simply checks to see that the `map_pages` handler is available and that `fault_around_bytes` specifies at least 1 page.

The actual fault-around is performed in `do_fault_around()`. If it returns anything other than `0` then this indicates that either an error has occurred or

the read faulting address has already been faulted in (which will be reported as `VM_FAULT_NOPAGE`), and we should not proceed with the fault.

Examining this function as shown in Listing 6-28.

```

4443  /*
4444   * do_fault_around() tries to map few pages around the fault address. The hope
4445   * is that the pages will be needed soon and this will lower the number of
4446   * faults to handle.
4447   *
4448   * It uses vm_ops->map_pages() to map the pages, which skips the page if it's
4449   * not ready to be mapped: not up-to-date, locked, etc.
4450   *
4451   * This function doesn't cross the VMA boundaries, in order to call map_pages
4452   *   ()
4453   * only once.
4454   *
4455   * fault_around_bytes defines how many bytes we'll try to map.
4456   * do_fault_around() expects it to be set to a power of two less than or equal
4457   * to PTRS_PER_PTE.
4458   *
4459   * The virtual address of the area that we map is naturally aligned to
4460   * fault_around_bytes rounded down to the machine page size
4461   * (and therefore to page order). This way it's easier to guarantee
4462   * that we don't cross page table boundaries.
4463   */
4464  static vm_fault_t do_fault_around(struct vm_fault *vmf)
4465  {
4466      unsigned long address = vmf->address, nr_pages, mask;
4467      pgoff_t start_pgoff = vmf->pgoff;
4468      pgoff_t end_pgoff;
4469      int off;
4470
4471      nr_pages = READ_ONCE(fault_around_bytes) >> PAGE_SHIFT;
4472      mask = ~(nr_pages * PAGE_SIZE - 1) & PAGE_MASK;
4473
4474      address = max(address & mask, vmf->vma->vm_start);
4475      off = ((vmf->address - address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1);
4476      start_pgoff -= off;
4477
4478      /*
4479       * end_pgoff is either the end of the page table, the end of
4480       * the vma or nr_pages from start_pgoff, depending what is nearest.
4481       */
4482      end_pgoff = start_pgoff -
4483                  ((address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1)) +
4484                  PTRS_PER_PTE - 1;
4485      end_pgoff = min3(end_pgoff, vma_pages(vmf->vma) + vmf->vma->vm_pgoff -
4486                      1,

```

```

4485         start_pgoff + nr_pages - 1);
4486
4487     if (pmd_none(*vmf->pmd)) {
4488         vmf->prealloc_pte = pte_alloc_one(vmf->vma->vm_mm);
4489         if (!vmf->prealloc_pte)
4490             return VM_FAULT_OOM;
4491     }
4492
4493     return vmf->vma->vm_ops->map_pages(vmf, start_pgoff, end_pgoff);
4494 }

```

Listing 6-28: mm/memory.c: do_fault_around()

This delegates the operation to the struct `vm_area_struct` (VMA)'s object struct `vm_operations_struct` `vm_ops->map_pages` handler is passed the struct `vm_fault` object along with a range of offsets to map in `start_pgoff` and `end_pgoff`. Prior to calling this, if the PMD entry is not set, we 'preallocate' a new PTE table and place it in the `prealloc_pte` field for the handler to use.

We perform this pre-allocation to avoid issues with any invocation of the allocator that may occur from atomic context within the `map_pages` handler (introduced in 7267ec008b5c: mm: postpone page table allocation until we have page to map and refined in f9ce0be71d1f: mm: Cleanup faultaround and finish_fault() codepaths).

The remaining code calculates what `start_pgoff` and `end_pgoff` should be. This is quite tricky so we'll go over it carefully:

- We determine the number of pages specified by `fault_around_bytes` and store this in `nr_pages`.
- We obtain a mask at the granularity of the fault-around block size, i.e. for 64 KiB this will mask out blocks of memory in 64 KiB chunks*.
- We want to keep the range over which we span aligned to the fault-around value, which is guaranteed to be a power-of-2, after which we fault in only this many pages – this means that we are guaranteed not to cross a page table boundary†. Why?

Virtual addresses are, by the structure of page tables, aligned to the address range they span, e.g. you cannot possibly cross a PTE page table boundary unless a virtual address passes through a value aligned to 2 MiB (the range a PTE table can map).

Since we explicitly restrict `fault_around_bytes` to being at last equal to the size of a base page, less than 2 MiB and a power-of-2 which we align the start and end range of the faulted in addresses to, it's simply not pos-

*. This uses a very common bit-wise trick – if you subtract 1 from a power of 2 you obtain a mask for its lower bits (e.g. `0b01000` yields `0b00111`) and then if you complement that you mask out the higher bits, including that bit value (e.g. `0b01000` yields `0b111...11000`).

†. Page table boundaries are point at which an increment in a virtual address causes the mapping to be specified by at least one different page table, e.g. moving from one PTE to another or one PMD or another and so on (if the page table entries mapping an address sit in the last entry of more than one page tables, then it's possible for many to be crossed at once).

sible to step outside of the bounds of the page table (note that we also clamp this range to that of the containing VMA).

For example, a range starting at 0x200000 of 0x200000 bytes will terminate at 0x3fffff, not crossing the boundary.

- Returning to the code, we apply the previously obtained mask to our address, which aligns it to `fault_around_bytes`. We must take into account this might result in a value less than the starting address, e.g. consider a `vm_start` of 0x51000 and an address of 0x5f000, if we were to align to 0x10000 this would result in an address of 0x50000, which precedes the VMA.

We resolve this issue by ‘clamping’ to the start of the VMA should it exceed our aligned value.

- Earlier, we set `start_pgoff` to the struct `vm_fault->pgoff` field (set by `linear_page_index()`) which will have been offset by the address’s page offset into the VMA. We must take into account that the start address might now be offset from this, so subtract the appropriate value from `start_pgoff`.
- Next we must obtain the *inclusive* end page offset to map, `end_pgoff`. The fact that we must ‘clamp’ the start of the range to fault around to the start of the VMA presents a problem – this value might not be aligned to `nr_pages`, so if we simply add `nr_pages` we might pass a page boundary regardless. We might also exceed the end of the VMA `vm_end`.

To resolve this issue we must choose the minimum of three values – the end of the PTE, `vm_end` and simply offsetting by `nr_pages - 1`.

We must keep in mind that our `start_pgoff` value not only encodes the offset of the address within the VMA but also but also the offset of the VMA itself in the underlying page cache object (typically a file), thus this necessitates something of a dance here.

To obtain the offset which will be mapped to the final PTE entry, we must subtract from `start_pgoff` the portion of the target start address which is not aligned with the PTE, i.e. by masking against the lower bits of `PTRS_PER_PTE`, then offset by the maximum index into the page table (since `end_pgoff` is an *inclusive* bound), i.e. adding `PTRS_PER_PTE - 1`. We place this in `end_pgoff`.

The other two values are obtained simply – we can figure out `vm_end` relative to the page offset by adding the number of pages spanned by it (via `vma_pages()`) to the VMA’s `vm_pgoff` and subtracting 1.

Obtaining the end of the fault-around block is very simply – we just offset `start_pgoff` by `nr_pages - 1`.

- We now possess both `start_pgoff` and `end_pgoff` and are ready to invoke the `map_pages` handler.

We discuss the specifics of the default `map_pages` handler in the page cache chapter in section 9.8.

6.6 Non-anonymous MAP_PRIVATE Copy on Write page fault

When a mapping is obtained via `mmap()` for a file but with `MAP_PRIVATE` specified, this necessitates special handling – if a read-fault occurs, then this will result in a Copy on Write mapping that, other than being mapped read-only is equivalent in hardware to that of a shared mapping.

However, once a write fault occurs, the folio is copied into an anonymous page. This therefore creates the edge case of a non-anonymous mapping that can experience a Copy-on-Write fault.

Importantly, if the file mapping is truncated such that a mapped page is no longer valid (e.g. a file is opened with the `O_TRUNC` flag for overwrite or `ftruncate()` is invoked with a size at or below that of the mapped page), then the mapping is invalidated, losing all of the data written into any such CoW'd anonymous pages.

This is what makes this type of mapping hazardous – the underlying file must remain intact for them to remain valid. This is true of shared mappings too, except the data written will be written back to the underlying file so is not lost.

This edge case is handled via `do_cow_fault()` (eliding out of scope cgroup handling) as shown in Listing 6-29.

```

4535 static vm_fault_t do_cow_fault(struct vm_fault *vmf)
4536 {
4537     struct vm_area_struct *vma = vmf->vma;
4538     vm_fault_t ret;
4539
4540     if (unlikely(anon_vma_prepare(vma)))
4541         return VM_FAULT_OOM;
4542
4543     vmf->cow_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma, vmf->address
4544 );
4545     if (!vmf->cow_page)
4546         return VM_FAULT_OOM;
4547     :
4548     ret = __do_fault(vmf);
4549     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)
4550 ))
4551         goto uncharge_out;
4552     if (ret & VM_FAULT_DONE_COW)
4553         return ret;
4554
4555     copy_user_highpage(vmf->cow_page, vmf->page, vmf->address, vma);
4556     __SetPageUptodate(vmf->cow_page);
4557
4558     ret |= finish_fault(vmf);
4559     unlock_page(vmf->page);
4560     put_page(vmf->page);

```

```

4566         if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)
4567             ))
4568             goto uncharge_out;
4569         return ret;
4570 uncharge_out:
4571         put_page(vmf->cow_page);
4572         return ret;
4573     }

```

Listing 6-29: mm/memory.c: do_cow_fault()

This somewhat mirrors `do_anonymous_page()` (see listing 6-19), except that rather than allocating a page to copy into via `alloc_zeroed_user_highpage_movable()`, it allocates using `alloc_page_vma()` directly specifying `GFP_HIGHUSER_MOVABLE`. Since we'll be directly copying into this page, there is no need to zero it.

Additionally this allocated page is placed into the `struct vm_fault->cow_page` field rather than the `page` field, and if no such allocation is possible, returns `VM_FAULT_OOM` to indicate an out of memory condition.

Prior to this, we set up the reverse mapping via `anon_vma_prepare()` (again, see Chapter 7 for a deeper explanation) as this mapping has now become anonymous.

However, this differs from an anonymous page fault in that we must invoke `__do_fault()` to populate the `struct vm_fault->page` field in order for us to copy it to the `cow_page`.

If an error arises, we drop our reference to the `cow_page` and exit. Otherwise, if `VM_FAULT_DONE_COW` is returned, this indicates the Copy on Write was performed for us and we can simply return.

If neither are the case, we will now have a pointer to the page cache entry folio in `page` and a pointer to the CoW anonymous page folio is `cow_page`. We must then perform the copy via `copy_user_highpage()` which, on a 64-bit system, is equivalent to invoking `copy_user_page()`, obtaining kernel virtual addresses for the specified folios via `page_to_virt()`, which amounts to a `memcpy()`.

We assign the folio flag `PG_uptodate` to `cow_page` as with an anonymous page fault, before deferring the completion of the fault (including establishing the PTE entry) to `finish_fault()`.

The page at this stage will be locked (having been so by `__do_fault()`), so we unlock it, then remove our reference to it via `put_page()`. Finally, we check against error conditions or ones which indicate that `cow_page` was not used (in which case dropping a reference from it) before returning.

6.7 Non-anonymous shared write fault

When a non-anonymous page faults which is mapped shared (i.e. `mmap()`'d using `MAP_SHARED`), translating into the VMA possessing a `VM_SHARED` flag, having

not been read faulted before, it undergoes a shared write fault, handled by `do_shared_fault()` as shown in Listing 6-30.

```

4574 static vm_fault_t do_shared_fault(struct vm_fault *vmf)
4575 {
4576     struct vm_area_struct *vma = vmf->vma;
4577     vm_fault_t ret, tmp;
4578
4579     ret = __do_fault(vmf);
4580     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY)
4581         ))
4582         return ret;
4583
4584     /*
4585      * Check if the backing address space wants to know that the page is
4586      * about to become writable
4587      */
4588     if (vma->vm_ops->page_mkwrite) {
4589         unlock_page(vmf->page);
4590         tmp = do_page_mkwrite(vmf);
4591         if (unlikely(!tmp ||
4592             (tmp & (VM_FAULT_ERROR | VM_FAULT_NOPAGE)))) {
4593             put_page(vmf->page);
4594             return tmp;
4595         }
4596     }
4597
4598     ret |= finish_fault(vmf);
4599     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE |
4600         VM_FAULT_RETRY))) {
4601         unlock_page(vmf->page);
4602         put_page(vmf->page);
4603         return ret;
4604     }
4605
4606     ret |= fault_dirty_shared_page(vmf);
4607     return ret;
4608 }

```

Listing 6-30: `mm/memory.c: do_shared_fault()`

As with the other non-anonymous fault handlers, this performs the actual faulting in via `__do_fault()` performs error checks and establishes the new mapping via `finish_fault()`, but varies in two respects – if the struct `vm_area_struct` (VMA)’s object struct `vm_operations_struct` `vm_ops->page_mkwrite` handler is specified, then this suggests that the underlying filesystem or equivalent wishes to be notified when a shared folio becomes writable and secondly, dirty page state is updated via `fault_dirty_shared_page()`.

Examining `do_page_mkwrite()` (eliding out of scope swap logic) as shown in Listing 6-31.

```

2953  /*
2954   * Notify the address space that the page is about to become writable so that
2955   * it can prohibit this or wait for the page to get into an appropriate state.
2956   *
2957   * We do this without the lock held, so that it can sleep if it needs to.
2958   */
2959  static vm_fault_t do_page_mkwrite(struct vm_fault *vmf)
2960  {
2961      vm_fault_t ret;
2962      struct page *page = vmf->page;
2963      unsigned int old_flags = vmf->flags;
2964
2965      vmf->flags = FAULT_FLAG_WRITE|FAULT_FLAG_MKWRITE;
2966      :
2971      ret = vmf->vma->vm_ops->page_mkwrite(vmf);
2972      /* Restore original flags so that caller is not surprised */
2973      vmf->flags = old_flags;
2974      if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE)))
2975          return ret;
2976      if (unlikely(!(ret & VM_FAULT_LOCKED))) {
2977          lock_page(page);
2978          if (!page->mapping) {
2979              unlock_page(page);
2980              return 0; /* retry */
2981          }
2982          ret |= VM_FAULT_LOCKED;
2983      } else
2984          VM_BUG_ON_PAGE(!PageLocked(page), page);
2985      return ret;
2986  }

```

Listing 6-31: `mm/memory.c: do_page_mkwrite()`

This allows an underlying filesystem to be able to deny write access to a folio or perform operations required to permit this when either a read-only mapping becomes writable (invoked from `wp_page_shared()`, discussed in section 6.9) or here when a shared mapping is first written to without an intervening read.

This ultimately invokes the `struct vm_area_struct` (VMA)'s object `struct vm_operations_struct vm_ops->page_mkwrite` handler with the flags `FAULT_FLAG_WRITE` and `FAULT_FLAG_MKWRITE` set.

After this we exit if an error is raised or it is indicated the fault has otherwise been handled. We relock the folio if necessary and, should the address space no longer be valid (indicative of a file truncation or other form of invalidation of the folio in respect of its associated `struct address_space`), the

function returns 0 which indicates to `do_shared_fault()` that the fault is complete (and will refault).

The other difference between other fault handlers is the invocation of `fault_dirty_shared_page()`, which is used to handle the fact that either a read-only non-anonymous mapping is becoming writable (see section 6.9 for discussion of this case), or, as here, a newly write faulted in shared mapping.

This is invoked with the folio locked, meaning it can rely on the folio remaining stable throughout the operation, which it releases when it is complete. We defer discussion of the details of this function to the page cache chapter.

6.8 Shared non-anonymous fault logic

The non-anonymous fault handlers all share logic – `__do_fault()` which performs the actual faulting in of the folio, locking it as it does so and `finish_fault()` which performs the actual mapping of the faulted-in folio (and presumes that it is locked as it does so).

Let's examine `__do_fault()` to begin with, eliding the out of scope page poisoning logic as shown in Listing 6-32.

```

4142  /*
4143   * The mmap_lock must have been held on entry, and may have been
4144   * released depending on flags and vma->vm_ops->fault() return value.
4145   * See filemap_fault() and __lock_page_retry().
4146   */
4147  static vm_fault_t __do_fault(struct vm_fault *vmf)
4148  {
4149      struct vm_area_struct *vma = vmf->vma;
4150      vm_fault_t ret;
4151
4152      /*
4153       * Preallocate pte before we take page_lock because this might lead to
4154       * deadlocks for memcg reclaim which waits for pages under writeback:
4155       *                                     lock_page(A)
4156       *                                     SetPageWriteback(A)
4157       *                                     unlock_page(A)
4158       * lock_page(B)
4159       *                                     lock_page(B)
4160       * pte_alloc_one
4161       *   shrink_page_list
4162       *     wait_on_page_writeback(A)
4163       *                                     SetPageWriteback(B)
4164       *                                     unlock_page(B)
4165       *                                     # flush A, B to clear the writeback
4166       */
4167      if (pmd_none(*vmf->pmd) && !vmf->prealloc_pte) {
4168          vmf->prealloc_pte = pte_alloc_one(vma->vm_mm);
4169          if (!vmf->prealloc_pte)

```



```

4170         return VM_FAULT_OOM;
4171     }
4172
4173     ret = vma->vm_ops->fault(vmf);
4174     if (unlikely(ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE | VM_FAULT_RETRY
4175         |
4176             VM_FAULT_DONE_COW)))
4177         return ret;
4178     :
4195     if (unlikely(!(ret & VM_FAULT_LOCKED)))
4196         lock_page(vmf->page);
4197     else
4198         VM_BUG_ON_PAGE(!PageLocked(vmf->page), vmf->page);
4199
4200     return ret;
4201 }

```

Listing 6-32: mm/memory.c: `__do_fault()`

This logic begins with some careful handling of the pre-allocation of the PTE if no PTE is pointed at by the PMD entry, should it not already be pre-allocated. This is to avoid a deadlock under writeback (a topic we discuss in the page cache chapter) were the faulting logic to have to perform this allocation. This was added and is well described in commit 63f3655f9501: mm, memcg: fix reclaim deadlock with writeback.

Next we simply defer the fault handling to the underlying struct `vm_area_struct` (VMA)'s object struct `vm_operations_struct` `vm_ops->fault` handler, before checking for errors or cases which require us to exit with no further action taken in which case we exit early.

Finally, we check whether the fault handler did indeed lock the folio (via the `VM_FAULT_LOCKED` flag) and do so ourselves if not, as the caller expects the folio to be locked after this operation so we can safely map it without it disappearing from beneath us.

The intricacies of what is actually implemented in this handler, which is often deferred to `filemap_fault()` are discussed in detail in the page cache chapter.

Once the folio has been faulted in, we must establish it in its mapping and perform final setup while the folio is locked. This is handled by `finish_fault()` (eliding out of scope huge page and devmap handling) as shown in Listing 6-33.

```

4330 /**
4331  * finish_fault - finish page fault once we have prepared the page to fault
4332  *
4333  * @vmf: structure describing the fault
4334  *
4335  * This function handles all that is needed to finish a page fault once the
4336  * page to fault in is prepared. It handles locking of PTEs, inserts PTE for
4337  * given page, adds reverse page mapping, handles memcg charges and LRU

```

```

4338  * addition.
4339  *
4340  * The function expects the page to be locked and on success it consumes a
4341  * reference of a page being mapped (for the PTE which maps it).
4342  *
4343  * Return: %0 on success, %VM_FAULT_ code in case of error.
4344  */
4345  vm_fault_t finish_fault(struct vm_fault *vmf)
4346  {
4347      struct vm_area_struct *vma = vmf->vma;
4348      struct page *page;
4349      vm_fault_t ret;
4350
4351      /* Did we COW the page? */
4352      if ((vmf->flags & FAULT_FLAG_WRITE) && !(vma->vm_flags & VM_SHARED))
4353          page = vmf->cow_page;
4354      else
4355          page = vmf->page;
4356
4357      /*
4358       * check even for read faults because we might have lost our CoW'ed
4359       * page
4360       */
4361      if (!(vma->vm_flags & VM_SHARED)) {
4362          ret = check_stable_address_space(vma->vm_mm);
4363          if (ret)
4364              return ret;
4365      }
4366
4367      if (pmd_none(*vmf->pmd)) {
4368          :
4369          if (vmf->prealloc_pte)
4370              pmd_install(vma->vm_mm, vmf->pmd, &vmf->prealloc_pte);
4371          else if (unlikely(pte_alloc(vma->vm_mm, vmf->pmd)))
4372              return VM_FAULT_OOM;
4373      }
4374      :
4375      vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd,
4376                                     vmf->address, &vmf->ptl);
4377
4378      /* Re-check under ptl */
4379      if (likely(!vmf_pte_changed(vmf))) {
4380          do_set_pte(vmf, page, vmf->address);
4381
4382          /* no need to invalidate: a not-present page won't be cached
4383           */
4384
4385          update_mmu_cache(vma, vmf->address, vmf->pte);

```

```

4396
4397         ret = 0;
4398     } else {
4399         update_mmu_tlb(vma, vmf->address, vmf->pte);
4400         ret = VM_FAULT_NOPAGE;
4401     }
4402
4403     pte_unmap_unlock(vmf->pte, vmf->ptl);
4404     return ret;
4405 }

```

Listing 6-33: mm/memory.c: finish_fault()

We start by determining which field of the struct `vm_fault` object actually contains the faulted-in page – if this was a `MAP_PRIVATE` mapping and a write fault, then `cow_page`, otherwise it will be located in `page`.

We then perform a careful check of the state of the address space via `check_stable_address_space()` in the `MAP_PRIVATE` case, as if the Out Of Memory (OOM) killer has been activated, our CoW page may disappear beneath us (see the Out Of Memory killer chapter for more on this).

If the PTE has not yet been allocated, we allocate either from the pre-allocated PTE or directly, before acquiring a PTE lock via `pte_offset_map_lock()` (again note for 64-bit architectures no ‘mapping’ is required).

With the PTE lock acquired, we check to see if the PTE has changed via `vmf_pte_changed()` – if so, we return `VM_FAULT_NOPAGE` to indicate that it has changed beneath us and thus the allocated page is not required. Examining this function as shown in Listing 6-34.

```

4322 static bool vmf_pte_changed(struct vm_fault *vmf)
4323 {
4324     if (vmf->flags & FAULT_FLAG_ORIG_PTE_VALID)
4325         return !pte_same(*vmf->pte, vmf->orig_pte);
4326
4327     return !pte_none(*vmf->pte);
4328 }

```

Listing 6-34: mm/memory.c: vmf_pte_changed()

If an original PTE entry was stored (i.e. a PTE *table* was assigned to the appropriate PMD at the time and thus the original PTE entry could be obtained, even if empty) then we check to see if it has changed prior to us acquiring the PTE table lock by comparing it to the original.

If not, then the PTE table will not have been allocated at the time and thus one will have been allocated with each PTE entry being empty and we can check to see if anything has changed by checking to see if it still is via `pte_none()`.

If nothing has changed beneath us, we then set the PTE to map the newly faulted-in page and perform various housekeeping tasks via `do_set_pte()` as shown in Listing 6-35.

```

4290 void do_set_pte(struct vm_fault *vmf, struct page *page, unsigned long addr)
4291 {
4292     struct vm_area_struct *vma = vmf->vma;
4293     bool uffd_wp = pte_marker_uffd_wp(vmf->orig_pte);
4294     bool write = vmf->flags & FAULT_FLAG_WRITE;
4295     bool prefault = vmf->address != addr;
4296     pte_t entry;
4297
4298     flush_icache_page(vma, page);
4299     entry = mk_pte(page, vma->vm_page_prot);
4300
4301     if (prefault && arch_wants_old_prefaulted_pte())
4302         entry = pte_mkold(entry);
4303     else
4304         entry = pte_sw_mkyoung(entry);
4305
4306     if (write)
4307         entry = maybe_mkwwrite(pte_mkdirty(entry), vma);
4308     if (unlikely(uffd_wp))
4309         entry = pte_mkuffd_wp(pte_wrprotect(entry));
4310     /* copy-on-write page */
4311     if (write && !(vma->vm_flags & VM_SHARED)) {
4312         inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
4313         page_add_new_anon_rmap(page, vma, addr);
4314         lru_cache_add_inactive_or_unevictable(page, vma);
4315     } else {
4316         inc_mm_counter_fast(vma->vm_mm, mm_counter_file(page));
4317         page_add_file_rmap(page, vma, false);
4318     }
4319     set_pte_at(vma->vm_mm, addr, vmf->pte, entry);
4320 }

```

Listing 6-35: mm/memory.c: do_set_pte()

For architectures that require it, the instruction cache is manually flushed, however this is not the case for x86-64.

Then `arch_wants_old_prefaulted_pte()` (relevant to arm64) and `pte_sw_mkyoung()` (relevant to MIPS) handle some edge case page table update logic. Neither of these are relevant to x86-64.

Similar to `do_anonymous_page()`, the PTE entry is marked writable and dirty if the fault was a write fault. This is checked via `maybe_mkwwrite()` which simply marks the mapping writable if its containing VMA has `VM_WRITE` specified. GUP can trigger write faults for non-writable VMAs hence the need for this check.

Finally we differentiate between a `MAP_PRIVATE` Copy on Write page and a shared one – in both cases the relevant statistics are updated and in the former case the newly faulted page is added to the anonymous reverse mapping

via `page_add_new_anon_rmap()` (see listing 7-20 and section 7.0.12 for a detailed examination of this).

In the latter case the folio is added to the file reverse mapping via `page_add_file_rmap()` (for a broader discussion of the reverse mapping see Chapter 7).

6.9 Write-protected page fault

When a mapping exists on a write fault but the existing mapping is read-only (though with `VM_WRITE` set), or the Get User Pages (GUP) logic requests that an *anonymous* memory mapping is unshared via `FAULT_FLAG_UNSHARE` we need to perform a Copy on Write operation. This case is handled by `do_wp_page()` as shown in Figure 6-4.

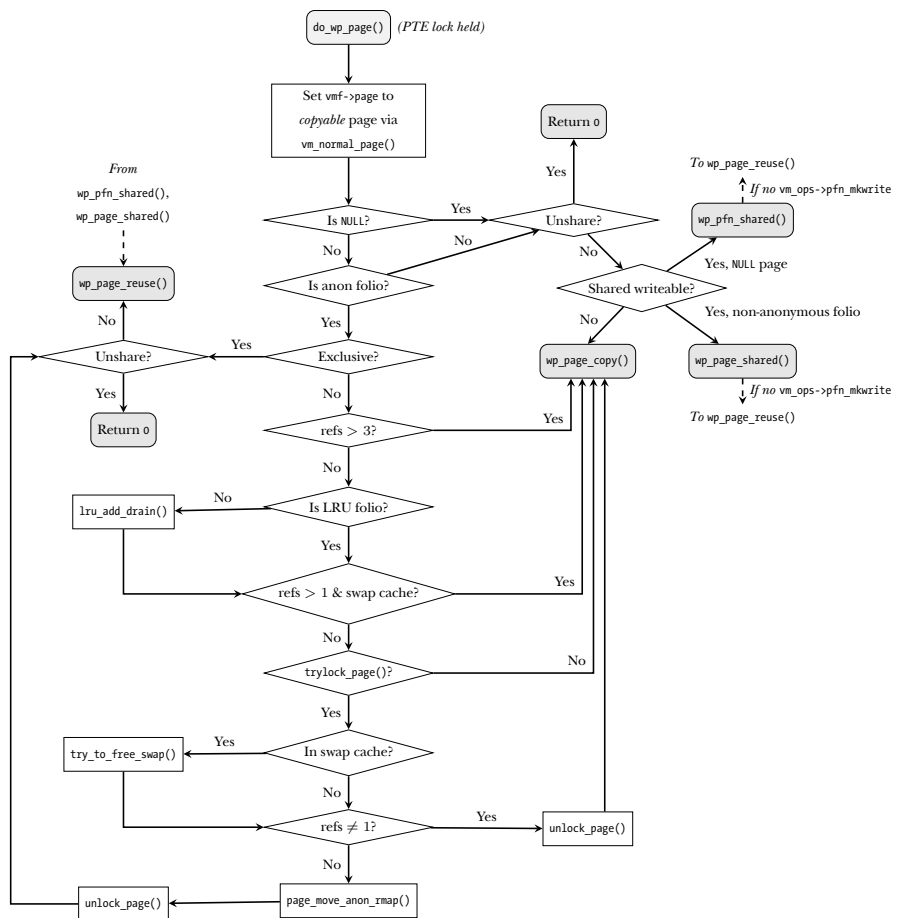


Figure 6-4: Simplified overview of write-protect fault handler `do_wp_page()`

Note that the case of an attempted write access to a VMA area which does not have `VM_WRITE` set this will have already been handled by `access_error()`.

This function applies to both anonymous and non-anonymous mappings, as anonymous mappings can become Copy on Write when a process is forked, and non-anonymous mappings, if mapped via `MAP_PRIVATE`, also require Copy on Write semantics.

`do_wp_page()` is invoked with the PTE table containing the entry in question locked. Examining this, eliding out of scope `userfaultfd` and `KSM` logic as shown in Listing 6-36.

```

3338 /*
3339  * This routine handles present pages, when
3340  * * users try to write to a shared page (FAULT_FLAG_WRITE)
3341  * * GUP wants to take a R/O pin on a possibly shared anonymous page
3342  *   (FAULT_FLAG_UNSHARE)
3343  *
3344  * It is done by copying the page to a new address and decrementing the
3345  * shared-page counter for the old page.
3346  *
3347  * Note that this routine assumes that the protection checks have been
3348  * done by the caller (the low-level page fault routine in most cases).
3349  * Thus, with FAULT_FLAG_WRITE, we can safely just mark it writable once we've
3350  * done any necessary COW.
3351  *
3352  * In case of FAULT_FLAG_WRITE, we also mark the page dirty at this point even
3353  * though the page will change only once the write actually happens. This
3354  * avoids a few races, and potentially makes it more efficient.
3355  *
3356  * We enter with non-exclusive mmap_lock (to exclude vma changes,
3357  * but allow concurrent faults), with pte both mapped and locked.
3358  * We return with mmap_lock still held, but pte unmapped and unlocked.
3359  */
3360 static vm_fault_t do_wp_page(struct vm_fault *vmf)
3361     __releases(vmf->ptl)
3362 {
3363     const bool unshare = vmf->flags & FAULT_FLAG_UNSHARE;
3364     struct vm_area_struct *vma = vmf->vma;
3365
3366     VM_BUG_ON(unshare && (vmf->flags & FAULT_FLAG_WRITE));
3367     VM_BUG_ON(!unshare && !(vmf->flags & FAULT_FLAG_WRITE));
3368     :
3384     vmf->page = vm_normal_page(vma, vmf->address, vmf->orig_pte);
3385     if (!vmf->page) {
3386         if (unlikely(unshare)) {
3387             /* No anonymous page -> nothing to do. */
3388             pte_unmap_unlock(vmf->pte, vmf->ptl);
3389             return 0;

```

```

3390         }
3391
3392     /*
3393     * VM_MIXEDMAP !pfn_valid() case, or VM_SOFTDIRTY clear on a
3394     * VM_PFNMAP VMA.
3395     *
3396     * We should not cow pages in a shared writeable mapping.
3397     * Just mark the pages writable and/or call ops->pfn_mkwrite.
3398     */
3399     if ((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
3400         (VM_WRITE|VM_SHARED))
3401         return wp_pfn_shared(vmf);
3402
3403     pte_unmap_unlock(vmf->pte, vmf->ptl);
3404     return wp_page_copy(vmf);
3405 }
3406
3407 /*
3408  * Take out anonymous pages first, anonymous shared vmas are
3409  * not dirty accountable.
3410  */
3411 if (PageAnon(vmf->page)) {
3412     struct page *page = vmf->page;
3413
3414     /*
3415     * If the page is exclusive to this process we must reuse the
3416     * page without further checks.
3417     */
3418     if (PageAnonExclusive(page))
3419         goto reuse;
3420
3421     /*
3422     * We have to verify under page lock: these early checks are
3423     * just an optimization to avoid locking the page and freeing
3424     * the swapcache if there is little hope that we can reuse.
3425     *
3426     * PageKsm() doesn't necessarily raise the page refcount.
3427     */
3428     if (PageKsm(page) || page_count(page) > 3)
3429         goto copy;
3430     if (!PageLRU(page))
3431         /*
3432         * Note: We cannot easily detect+handle references
3433         * remote LRU pagevecs or references to PageLRU()
3434         pages.
3435         */

```

```

3435         lru_add_drain();
3436         if (page_count(page) > 1 + PageSwapCache(page))
3437             goto copy;
3438         if (!trylock_page(page))
3439             goto copy;
3440         if (PageSwapCache(page))
3441             try_to_free_swap(page);
3442         if (PageKsm(page) || page_count(page) != 1) {
3443             unlock_page(page);
3444             goto copy;
3445         }
3446         /*
3447          * Ok, we've got the only page reference from our mapping
3448          * and the page is locked, it's dark out, and we're wearing
3449          * sunglasses. Hit it.
3450          */
3451         page_move_anon_rmap(page, vma);
3452         unlock_page(page);
3453 reuse:
3454         if (unlikely(unshare)) {
3455             pte_unmap_unlock(vmf->pte, vmf->ptl);
3456             return 0;
3457         }
3458         wp_page_reuse(vmf);
3459         return VM_FAULT_WRITE;
3460     } else if (unshare) {
3461         /* No anonymous page -> nothing to do. */
3462         pte_unmap_unlock(vmf->pte, vmf->ptl);
3463         return 0;
3464     } else if (unlikely((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
3465                        (VM_WRITE|VM_SHARED))) {
3466         return wp_page_shared(vmf);
3467     }
3468 copy:
3469     /*
3470      * Ok, we need to copy. Oh, well..
3471      */
3472     get_page(vmf->page);
3473
3474     pte_unmap_unlock(vmf->pte, vmf->ptl);
3475     :
3476     return wp_page_copy(vmf);
3480 }

```

Listing 6-36: mm/memory.c: do_wp_page()

This starts by asserting that, if the `FAULT_FLAG_UNSHARE` flag has been specified, that the `FAULT_FLAG_WRITE` cannot be – i.e. they are mutually exclusive.

Equally, if `FAULT_FLAG_UNSHARE` has *not* been specified, then `FAULT_FLAG_WRITE` *must* be.

Next, we obtain the struct page object associated with the PTE via `vm_normal_page()` and place it in the struct `vm_fault->page` field. See section 6.12 for more on this.

We then deal with the case in which no *copyable* page object is associated with the PTE (i.e. a special case handled by `vm_normal_page()` – if we are unsharing then there is simply nothing to do and we exit.

However, if there is no copyable page mapped and it is shared writable – we delegate the operation to `wp_pfn_shared()` – this is specific to a `VM_MIXEDMAP` or `VM_PFNMAP` mapping with `VM_SHARED` set. We will discuss this in more detail below when we examine `wp_pfn_shared()`.

If there is no *copyable* page mapped and neither of two previously described cases apply, which can occur for example if the mapping points at the zero page, then we simply go ahead and copy it via `wp_page_copy()` with the PTE map released.

Next we have separate handling for anonymous and non-anonymous mappings – dealing with non-anonymous first, as this is far simpler.

6.9.1 *Non-anonymous folios*

if we are unsharing then there is simply nothing to do as this operation is not supported for non-anonymous pages.

Otherwise we check to see if the non-anonymous struct `vm_area_struct` (VMA) is writable and shared, if so we handle this fault with the `wp_page_shared()` handler – this is the case where the PTE mapping is read-only.

It might seem odd that a writable *shared* non-anonymous mapping would have a readonly PTE (which is what would have to have happened to reach this point in the code), however this is explicitly the case when `vma_set_page_prot()` on memory map, or `mprotect_fixup()` on modifying a memory region's protection on `mprotect()`, determines that a non-anonymous VMA requires notification when a mapping becomes writable, as determined by `vma_wants_writenotify()`.

6.9.2 *Anonymous folios*

Anonymous mappings (with a copyable source folio) comprise the bulk of the complexity of the function. The logic consists of trying our hardest to reuse folios if we can. We perform the quickest checks first, before going through a process of removing unnecessary references and seeing whether doing so renders reuse possible (via `wp_page_reuse()`).

Examining this (note we elide out of scope *Kernel Samepage Merging* (KSM) logic):

Anonymous exclusive folio – The `PG_anon_exclusive` folio flag is set, meaning that there is one and only one reference to the folio which can simply be reused via `wp_page_reuse()`.

Excessive folio references – Since the folio is not yet locked, we try to avoid doing so if we don't stand any chance of reusing the folio.

Folio batches concerned with LRU operations (such as `folio_add_lru()`) obtain a reference on a folio before adding them to the folio batch (see section 11.7 for details on folio batches) and in addition there might be a swap cache reference (see swap chapter for details of the swap cache). Therefore, if the number of references to the folio (as determined by `page_count()` (which defers to `folio_ref_count()` and ultimately atomically reads the folio's `_refcount` field) exceeds these 3 potential sources of ephemeral references, then it's really not worth attempting to reuse a folio and we shortcut to copying the page via `wp_page_copy()`.

Non-LRU folio – If the folio in question is non-LRU, i.e. is on a folio batch in the process of being added to an LRU list or otherwise moved around, we hold an excess reference.

In order to try to cheaply clear such a reference, we drain folios from the local CPU's LRU folio batches via `lru_add_drain()`. This and the related check above were added in commit `d4c470970d45` – mm: optimize `do_wp_page()` for fresh pages in local LRU pagevecs.

Excessive folio references again – Now we have drained the LRU, we explicitly check to see if we have additional references in excess of the swap cache one – if so, then we must copy.

Free swap cache – We start by trying to lock the folio via `trylock_page()`, this is a non-sleeping lock – if we fail then we simply copy the folio.

Now locked, we can reliably check whether the folio is in the swap cache, and attempting to quickly free it from there using `try_to_free_swap()`. See the swap chapter for more on the swap cache.

Final excessive folio references check – Now, if we have more than 1 reference, we have no more tricks left to reduce them. We are then definitely in a situation where we have to copy the folio.

Folio reuse – We are finally at a point where we definitely have only one reference and can go ahead and try to reuse the folio rather than copy it.

We start by invoking `page_move_anon_rmap()` which sets the folio's mapping field to the struct `anon_vma` object the struct `vm_area_struct` (VMA) containing it points at. E.g., if a process had faulted in a folio, forks and then the parent unmaps it and the child accesses the folio triggering a write fault, this logic would point to the child's `anon_vma` instead (see Chapter 7 for more on this).

This function additionally sets the `PG_anon_exclusive` to indicate that the folio is now mapped exclusively.

Afterwards, the folio is unlocked via `unlock_page()` and check to see if we are GUP unsharing – if we are, then the fault is complete and the unshare has been successful.

Finally, for all other cases we reuse the folio via `wp_page_reuse()`.

6.9.3 Folio/PFN sharing

When a non-anonymous or non-folio backed (see section 6.12 for more on these) shared writable mapping write faults, we resolve this via `wp_page_shared()` for the former case and `wp_pfn_shared()` for the latter.

The primary reason that this kind of fault occurs is because the mapping has been intentionally marked readonly so the underlying filesystem can be notified when the mapping is made writable, as determined by the struct `vm_area_struct` (VMA)'s object struct `vm_operations_struct` `vm_ops->page_mkwrite` or `vm_ops->pfn_mkwrite` handlers respectively.

Each ultimately invokes `wp_page_reuse()` to mark the PTE writable.

Both functions are similar to one another in that they ultimately invoke the `mkwrite` handler, only `wp_page_shared()` has additional handling for the underlying folio. Examining the PFN case first as shown in Listing 6-37.

```

3282 /*
3283  * Handle write page faults for VM_MIXEDMAP or VM_PFNMAP for a VM_SHARED
3284  * mapping
3285  */
3286 static vm_fault_t wp_pfn_shared(struct vm_fault *vmf)
3287 {
3288     struct vm_area_struct *vma = vmf->vma;
3289
3290     if (vma->vm_ops && vma->vm_ops->pfn_mkwrite) {
3291         vm_fault_t ret;
3292
3293         pte_unmap_unlock(vmf->pte, vmf->ptl);
3294         vmf->flags |= FAULT_FLAG_MKWRITE;
3295         ret = vma->vm_ops->pfn_mkwrite(vmf);
3296         if (ret & (VM_FAULT_ERROR | VM_FAULT_NOPAGE))
3297             return ret;
3298         return finish_mkwrite_fault(vmf);
3299     }
3300     wp_page_reuse(vmf);
3301     return VM_FAULT_WRITE;
3302 }

```

Listing 6-37: `mm/memory.c: wp_pfn_shared()`

If there is no `pfn_mkwrite` handler then this simply forwards the fault handling to `wp_page_reuse()`. Otherwise, the PTE lock is released so the `mkwrite` handler can sleep if necessary without lock contention, `FAULT_FLAG_MKWRITE` fault flag is set to indicate that this faulting behaviour is desired, and the handler itself is invoked.

If an error or `VM_FAULT_NOPAGE` condition arises we simply exit with the error, otherwise we finalise the operation in `finish_mkwrite_fault()` which reacquires the PTE lock (which `wp_page_reuse()` expects), checks to see if the PTE changed from underneath us before invoking `wp_page_reuse()` as shown in Listing 6-38.

```

3248 /**
3249  * finish_mkwrite_fault - finish page fault for a shared mapping, making PTE
3250  *                        writeable once the page is prepared
3251  *
3252  * @vmf: structure describing the fault
3253  *
3254  * This function handles all that is needed to finish a write page fault in a
3255  * shared mapping due to PTE being read-only once the mapped page is prepared.
3256  * It handles locking of PTE and modifying it.
3257  *
3258  * The function expects the page to be locked or other protection against
3259  * concurrent faults / writeback (such as DAX radix tree locks).
3260  *
3261  * Return: %0 on success, %VM_FAULT_NOPAGE when PTE got changed before
3262  * we acquired PTE lock.
3263  */
3264 vm_fault_t finish_mkwrite_fault(struct vm_fault *vmf)
3265 {
3266     WARN_ON_ONCE(!(vmf->vma->vm_flags & VM_SHARED));
3267     vmf->pte = pte_offset_map_lock(vmf->vma->vm_mm, vmf->pmd, vmf->address
3268     ,
3269                                     &vmf->ptl);
3270     /*
3271      * We might have raced with another page fault while we released the
3272      * pte_offset_map_lock.
3273      */
3274     if (!pte_same(*vmf->pte, vmf->orig_pte)) {
3275         update_mmu_tlb(vmf->vma, vmf->address, vmf->pte);
3276         pte_unmap_unlock(vmf->pte, vmf->ptl);
3277         return VM_FAULT_NOPAGE;
3278     }
3279     wp_page_reuse(vmf);
3280     return 0;
3281 }

```

Listing 6-38: mm/memory.c: *finish_mkwrite_fault()*

For more details on the VM_MIXEDMAP and VM_PFNMAP mappings which necessitate this machinery see section 6.12.

In the more common case of a shared, writeable *non-anonymous* mapping which has a read-only mapping (most likely because the VMA fulfils `vma_wants_writenotify()`) the write fault is handled by `wp_page_shared()` as shown in Listing 6-39.

```

3304 static vm_fault_t wp_page_shared(struct vm_fault *vmf)
3305     __releases(vmf->ptl)
3306 {
3307     struct vm_area_struct *vma = vmf->vma;

```

```

3308         vm_fault_t ret = VM_FAULT_WRITE;
3309
3310         get_page(vmf->page);
3311
3312         if (vma->vm_ops && vma->vm_ops->page_mkwrite) {
3313             vm_fault_t tmp;
3314
3315             pte_unmap_unlock(vmf->pte, vmf->ptl);
3316             tmp = do_page_mkwrite(vmf);
3317             if (unlikely(!tmp || (tmp &
3318                                     (VM_FAULT_ERROR | VM_FAULT_NOPAGE)))) {
3319                 put_page(vmf->page);
3320                 return tmp;
3321             }
3322             tmp = finish_mkwrite_fault(vmf);
3323             if (unlikely(tmp & (VM_FAULT_ERROR | VM_FAULT_NOPAGE))) {
3324                 unlock_page(vmf->page);
3325                 put_page(vmf->page);
3326                 return tmp;
3327             }
3328         } else {
3329             wp_page_reuse(vmf);
3330             lock_page(vmf->page);
3331         }
3332         ret |= fault_dirty_shared_page(vmf);
3333         put_page(vmf->page);
3334
3335         return ret;
3336     }

```

Listing 6-39: mm/memory.c: wp_page_shared()

This maintains the same structure as `wp_pfn_shared()`, only with machinery to deal with the fact that we have an underlying folio to contend with. The first difference being that we increment the reference counter for the folio via `get_page()` over the operation and decrement it on exit via `put_page()`.

If a `page_mkwrite` handler is in place, we invoke it via `do_page_mkwrite()` (as described in section 6.7, listing 6-31, when `do_shared_fault()` invokes it) which acquires a folio lock if successful. This operation is performed with the PTE table unlocked to permit sleeping in the handler.

Finally this handling invokes `finish_mkwrite_fault()` as described above to reacquire the PTE lock and invoke `wp_page_reuse()` to reuse the underlying folio.

If the VMA does not require notifying of a shared mapping being made writable (i.e. it does not possess a `page_mkwrite` handler), then the function simply invokes `wp_page_reuse()` and locks the folio.

Handling the fact that this mapping has now been marked dirty is performed via `fault_dirty_shared_page()` which expects the underlying folio to be locked. After this is complete, the folio reference is dropped as we no longer have to be concerned about it disappearing beneath us during this operation. Examining this function as shown in Listing 6-40.

```

2988  /*
2989   * Handle dirtying of a page in shared file mapping on a write fault.
2990   *
2991   * The function expects the page to be locked and unlocks it.
2992   */
2993  static vm_fault_t fault_dirty_shared_page(struct vm_fault *vmf)
2994  {
2995      struct vm_area_struct *vma = vmf->vma;
2996      struct address_space *mapping;
2997      struct page *page = vmf->page;
2998      bool dirtied;
2999      bool page_mkwrite = vma->vm_ops && vma->vm_ops->page_mkwrite;
3000
3001      dirtied = set_page_dirty(page);
3002      VM_BUG_ON_PAGE(PageAnon(page), page);
3003      /*
3004       * Take a local copy of the address_space - page.mapping may be zeroed
3005       * by truncate after unlock_page(). The address_space itself remains
3006       * pinned by vma->vm_file's reference. We rely on unlock_page()'s
3007       * release semantics to prevent the compiler from undoing this copying
3008       *
3009       */
3009      mapping = page_rmapping(page);
3010      unlock_page(page);
3011
3012      if (!page_mkwrite)
3013          file_update_time(vma->vm_file);
3014
3015      /*
3016       * Throttle page dirtying rate down to writeback speed.
3017       *
3018       * mapping may be NULL here because some device drivers do not
3019       * set page.mapping but still dirty their pages
3020       *
3021       * Drop the mmap_lock before waiting on IO, if we can. The file
3022       * is pinning the mapping, as per above.
3023       */
3024      if ((dirtied || page_mkwrite) && mapping) {
3025          struct file *fpin;
3026
3027          fpin = maybe_unlock_mmap_for_io(vmf, NULL);
3028          balance_dirty_pages_ratelimited(mapping);

```

```

3029         if (fpin) {
3030             fput(fpin);
3031             return VM_FAULT_COMPLETED;
3032         }
3033     }
3034
3035     return 0;
3036 }

```

Listing 6-40: mm/memory.c: `fault_dirty_shared_page()`

This somewhat overlaps with the page cache logic which, being such a large and important subject, I defer to the page cache chapter, however we will briefly touch on it here. See this chapter for a far more thorough examination.

The folio itself is set dirty and any struct `address_mapping` handling for this operation is performed in `set_page_dirty()` (which is a legacy wrapper around `folio_mark_dirty()`).

With this operation complete, we obtain the `address_mapping` object of the folio via `page_rmapping()` and unlock the folio. This dance is required to handle the fact that truncation may result in the mapping being zeroed when not under lock.

If there was no `page_mkwrite` handler, the file associated with the mapping has its `ctime` and `mtime` fields updated via `file_update_time()`.

Finally, if the folio was successfully dirtied or a `page_mkwrite` handler exists (and was invoked) and an `address_space` mapping exists for it, we immediately release the struct `mm_structmmap_lock` lock if possible as we are about to wait on I/O and want to reduce contention.

We determine whether we can do so via `maybe_unlock_mmap_for_io()` as shown in Listing 6-41.

```

607 static inline struct file *maybe_unlock_mmap_for_io(struct vm_fault *vmf,
608                                                     struct file *fpin)
609 {
610     int flags = vmf->flags;
611
612     if (fpin)
613         return fpin;
614
615     /*
616      * FAULT_FLAG_RETRY_NOWAIT means we don't want to wait on page locks
617      or
618      * anything, so we only pin the file and drop the mmap_lock if only
619      * FAULT_FLAG_ALLOW_RETRY is set, while this is the first attempt.
620      */
621     if (fault_flag_allow_retry_first(flags) &&
622         !(flags & FAULT_FLAG_RETRY_NOWAIT)) {
623         fpin = get_file(vmf->vma->vm_file);
624         mmap_read_unlock(vmf->vma->vm_mm);

```

```

624     }
625     return fpin;
626 }

```

Listing 6-41: mm/internal.h: maybe_unlock_mmap_for_io()

We only drop the lock if retry is still permitted as determined by `fault_flag_allow_retry_first()` (i.e. the fault flag `FAULT_FLAG_ALLOW_RETRY` is set and `FAULT_FLAG_TRIED` is not) as long as `FAULT_FLAG_RETRY_NOWAIT` is not specified (this explicitly indicates that we do *not* want to wait for I/O).

The mapping is pinned via its struct `file` `vm_file` field if the `mmap_lock` was released. Dirty page are then potentially written back depending on the dirty page tunables such as `vm.dirty_ratio` via `balance_dirty_pages_ratelimited()`. See the page cache chapter for a detailed analysis of this.

If a file pin was acquired, this is released and `VM_FAULT_COMPLETED` is returned to indicate that the fault has been handled and lock released, otherwise we return 0 to indicate that no additional fault flags need be set.

6.9.4 Page reuse

When reusing a shared writeable non-anonymous PFN or folio (via `wp_pfn_shared()` or `wp_page_shared()`), or when an anonymous page has been determined to possess only a single reference reuse of the underlying folio is handled by `wp_page_reuse()` as shown in Listing 6-42

```

3038 /*
3039  * Handle write page faults for pages that can be reused in the current vma
3040  *
3041  * This can happen either due to the mapping being with the VM_SHARED flag,
3042  * or due to us being the last reference standing to the page. In either
3043  * case, all we need to do here is to mark the page as writable and update
3044  * any related book-keeping.
3045  */
3046 static inline void wp_page_reuse(struct vm_fault *vmf)
3047     __releases(vmf->ptl)
3048 {
3049     struct vm_area_struct *vma = vmf->vma;
3050     struct page *page = vmf->page;
3051     pte_t entry;
3052
3053     VM_BUG_ON(!(vmf->flags & FAULT_FLAG_WRITE));
3054     VM_BUG_ON(page && PageAnon(page) && !PageAnonExclusive(page));
3055
3056     /*
3057      * Clear the pages cpupid information as the existing
3058      * information potentially belongs to a now completely
3059      * unrelated process.
3060      */

```



```

3061         if (page)
3062             page_cpupid_xchg_last(page, (1 << LAST_CPUPID_SHIFT) - 1);
3063
3064         flush_cache_page(vma, vmf->address, pte_pfn(vmf->orig_pte));
3065         entry = pte_mkyoung(vmf->orig_pte);
3066         entry = maybe_mkwrites(pte_mkdirty(entry), vma);
3067         if (ptep_set_access_flags(vma, vmf->address, vmf->pte, entry, 1))
3068             update_mmu_cache(vma, vmf->address, vmf->pte);
3069         pte_unmap_unlock(vmf->pte, vmf->ptl);
3070         count_vm_event(PGREUSE);
3071     }

```

Listing 6-42: mm/memory.c: wp_page_reuse()

The *NUMA balancing* logic tracks the last CPU which accessed a folio via a cpupid field typically encoded in the struct folio->flags field (or if the architecture and kernel configuration means there is insufficient space here, in the _last_cpupid field). We clear this field via page_cpupid_xchg_last() as this information is now invalidated (the folio may last have been accessed by an unrelated sharing process).

The flush_cache_page() and update_mmu_cache() functions are invoked for architectures which need manual memory cache maintenance, however our focused upon architecture x86-64 does not require this.

A new PTE entry is derived from the original PTE entry struct vm_fault->orig_pte only with its accessed flag set by pte_mkyoung(), dirtied by pte_mkdirty() and set writable by maybe_mkwrites() if the VMA has its VM_WRITE flag set.

The flags are updated by ptep_set_access_flags() before the PTE page table lock is released and statistics are updated.

6.9.5 Folio copying

If all else has failed and we simply need to Copy-on-Write or unshare because there are multiple references to the share read-only folio then we must go ahead and copy it. This is handled by wp_page_copy() (eliding out of scope mmu notifier, userfaultfd, delay accounting and cgroup logic) as shown in Listing 6-43.

```

3073  /*
3074   * Handle the case of a page which we actually need to copy to a new page,
3075   * either due to COW or unsharing.
3076   *
3077   * Called with mmap_lock locked and the old page referenced, but
3078   * without the ptl held.
3079   *
3080   * High level logic flow:
3081   *
3082   * - Allocate a page, copy the content of the old page to the new one.
3083   * - Handle book keeping and accounting - cgroups, mmu-notifiers, etc.

```

```

3084  * - Take the PTL. If the pte changed, bail out and release the allocated page
3085  * - If the pte is still the way we remember it, update the page table and all
3086  *   relevant references. This includes dropping the reference the page-table
3087  *   held to the old page, as well as updating the rmap.
3088  * - In any case, unlock the PTL and drop the reference we took to the old
      page.
3089  */
3090  static vm_fault_t wp_page_copy(struct vm_fault *vmf)
3091  {
3092      const bool unshare = vmf->flags & FAULT_FLAG_UNSHARE;
3093      struct vm_area_struct *vma = vmf->vma;
3094      struct mm_struct *mm = vma->vm_mm;
3095      struct page *old_page = vmf->page;
3096      struct page *new_page = NULL;
3097      pte_t entry;
3098      int page_copied = 0;
3099      :
3103      if (unlikely(anon_vma_prepare(vma)))
3104          goto oom;
3105
3106      if (is_zero_pfn(pte_pfn(vmf->orig_pte))) {
3107          new_page = alloc_zeroed_user_highpage_movable(vma,
3108                                                         vmf->address);
3109          if (!new_page)
3110              goto oom;
3111      } else {
3112          new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma,
3113                                   vmf->address);
3114          if (!new_page)
3115              goto oom;
3116
3117          if (!__wp_page_copy_user(new_page, old_page, vmf)) {
3118              /*
3119               * COW failed, if the fault was solved by other,
3120               * it's fine. If not, userspace would re-fault on
3121               * the same address and we will handle the fault
3122               * from the second attempt.
3123               */
3124              put_page(new_page);
3125              if (old_page)
3126                  put_page(old_page);
3127              :
3129              return 0;
3130          }
3131      }
3132      :
3137      __SetPageUptodate(new_page);

```

```

:
3144      /*
3145      * Re-check the pte - we dropped the lock
3146      */
3147      vmf->pte = pte_offset_map_lock(mm, vmf->pmd, vmf->address, &vmf->ptl);

```

Listing 6-43: mm/memory.c: wp_page_copy() folio copying

The function is called with a reference on the input folio (if it is backed by one, i.e. not a pure PFN mapping) so it won't disappear beneath us.

At this stage, the folio being copied to will be anonymous – either by simply being an anonymous mapping, or if this is a MAP_PRIVATE-mapped file-backed mapping, then on Copy on Write the struct `vm_area_struct` (VMA) will exist in both the struct `anon_vma` and struct `address_mapping->i_mmap` reverse mapping trees.

As a result, we must allocate and prepare the `anon_vma` associated with this mapping if this has not already been done, via `anon_vma_prepare()`. See Chapter 7 for more details on the reverse mapping.

Next, we perform the actual copy of the folio itself – if it is a zero page mapping, then no copying is required and we simply acquire a zeroed page via `alloc_zeroed_user_highpage_movable()`. If we cannot allocate we defer to the out of memory logic which returns `VM_FAULT_OOM` and ultimately invokes the Out Of Memory (OOM) killer (see the chapter on this for more details).

Otherwise, we allocate a new folio (note, not zeroed, as we are about to copy to it) via `alloc_page_vma()` and copy the page via `__wp_page_copy_user()`. If this fails, we decrement reference counts for the source and newly allocated folios and return zero – if the fault was resolved by a racing fault handler then we can simply carry on, if not then we will refault and can try again.

The `__wp_page_copy_user()` function is trivial for the folio-backed case, but more involved for special PFN mappings, so we defer examining it to section 6.12.

After the copying has completed, we mark the folio up to date via the `PG_uptodate` flag using the generated `__SetPageUptodate` helper function.

Next we handle updating the PTE and perform the remaining housekeeping tasks. We start by acquiring the PTE table lock via `pte_offset_map_lock()` which is held in preparation for updating the PTE entry.

Examining the remaining logic as shown in Listing 6-44.

```

3148      if (likely(pte_same(*vmf->pte, vmf->orig_pte))) {
3149          if (old_page) {
3150              if (!PageAnon(old_page)) {
3151                  dec_mm_counter_fast(mm,
3152                                     mm_counter_file(old_page));
3153                  inc_mm_counter_fast(mm, MM_ANONPAGES);
3154              }
3155          } else {
3156              inc_mm_counter_fast(mm, MM_ANONPAGES);
3157          }

```

```

3158         flush_cache_page(vma, vmf->address, pte_pfn(vmf->orig_pte));
3159         entry = mk_pte(new_page, vma->vm_page_prot);
3160         entry = pte_sw_mkyoung(entry);
3161         if (unlikely(unshare)) {
3162             if (pte_soft_dirty(vmf->orig_pte))
3163                 entry = pte_mksoft_dirty(entry);
3164             :
3165
3166         } else {
3167             entry = maybe_mkwrite(pte_mkdirty(entry), vma);
3168         }
3169
3170         /*
3171          * Clear the pte entry and flush it first, before updating the
3172          * pte with the new entry, to keep TLBs on different CPUs in
3173          * sync. This code used to set the new PTE then flush TLBs,
3174
3175          but
3176
3177          * that left a window where the new PTE could be loaded into
3178          * some TLBs while the old PTE remains in others.
3179          */
3180         ptep_clear_flush_notify(vma, vmf->address, vmf->pte);
3181         page_add_new_anon_rmap(new_page, vma, vmf->address);
3182         lru_cache_add_inactive_or_unevictable(new_page, vma);
3183
3184         /*
3185          * We call the notify macro here because, when using secondary
3186          * mmu page tables (such as kvm shadow page tables), we want
3187
3188          the
3189
3190          * new page to be mapped directly into the secondary page
3191
3192          table.
3193
3194          */
3195         BUG_ON(unshare && pte_write(entry));
3196         set_pte_at_notify(mm, vmf->address, vmf->pte, entry);
3197         update_mmu_cache(vma, vmf->address, vmf->pte);
3198         if (old_page) {
3199             /*
3200              * Only after switching the pte to the new page may
3201              * we remove the mapcount here. Otherwise another
3202              * process may come and find the rmap count
3203
3204          decremented
3205
3206              * before the pte is switched to the new page, and
3207              * "reuse" the old page writing into it while our pte
3208              * here still points into it and can be read by other
3209              * threads.
3210              *
3211              * The critical issue is to order this
3212              * page_remove_rmap with the ptp_clear_flush above.
3213              * Those stores are ordered by (if nothing else,)
3214              * the barrier present in the atomic_add_negative

```

```

3202         * in page_remove_rmap.
3203         *
3204         * Then the TLB flush in ptep_clear_flush ensures that
3205         * no process can access the old page before the
3206         * decremented mapcount is visible. And the old page
3207         * cannot be reused until after the decremented
3208         * mapcount is visible. So transitively, TLBs to
3209         * old page will be flushed before it can be reused.
3210         */
3211         page_remove_rmap(old_page, vma, false);
3212     }
3213
3214     /* Free the old page.. */
3215     new_page = old_page;
3216     page_copied = 1;
3217 } else {
3218     update_mmu_tlb(vma, vmf->address, vmf->pte);
3219 }
3220
3221 if (new_page)
3222     put_page(new_page);
3223
3224 pte_unmap_unlock(vmf->pte, vmf->ptl);
3225 :
3226
3227 if (old_page) {
3228     if (page_copied)
3229         free_swap_cache(old_page);
3230     put_page(old_page);
3231 }
3232 :
3233
3234 return (page_copied && !unshare) ? VM_FAULT_WRITE : 0;
3235 oom_free_new:
3236     put_page(new_page);
3237 oom:
3238     if (old_page)
3239         put_page(old_page);
3240 :
3241
3242 return VM_FAULT_OOM;
3243 }

```

Listing 6-44: mm/memory.c: wp_page_copy() PTE update, housekeeping

Given that we previously dropped the PTE page table lock and have only just reacquired, we need to check that the PTE has not changed from beneath us. If not, then we proceed to set up the PTE, starting by updating statistics.

On x86-64 we do not need to flush the cache manually so `flush_cache_page()` and related functions are no-ops which we needn't consider, equally `pte_mksoft_dirty()` is not relevant.

If we aren't simply unsharing a mapping, we mark it dirty via `pte_mkdirty()` and set writable by `maybe_mkwritable()` if the VMA has its `VM_WRITE` flag set as in other fault handlers.

Next we reach more nuanced logic. We are in the awkward situation where we are updating which physical address an existing virtual mapping points to. This means that CPUs might each individually possess a cached mapping between the virtual address and old physical address in their TLB cache, so we have to be *very* careful to ensure that the change in mapping is updated coherently across the system.

We start by invoking `ptep_clear_flush_notify()` – since MMU notification is out of scope for the book, we consider only `ptep_clear_flush()` which is called in turn by `ptep_clear_flush_notify()` as shown in Listing 6-45.

```

91 pte_t ptep_clear_flush(struct vm_area_struct *vma, unsigned long address,
92                       pte_t *ptep)
93 {
94     struct mm_struct *mm = (vma)->vm_mm;
95     pte_t pte;
96     pte = ptep_get_and_clear(mm, address, ptep);
97     if (pte_accessible(mm, pte))
98         flush_tlb_page(vma, address);
99     return pte;
100 }
```

Listing 6-45: mm/pgtable-generic.c: `ptep_clear_flush()`

This clears the PTE via `ptep_get_and_clear()` which retrieves the previous PTE entry before zeroing the entry and returns what was previously there.

We then determine whether the PTE was accessible in the first place (i.e. might be present in a TLB at all) via `pte_accessible()` as shown in Listing 6-46.

```

747 static inline bool pte_accessible(struct mm_struct *mm, pte_t a)
748 {
749     if (pte_flags(a) & _PAGE_PRESENT)
750         return true;
751
752     if ((pte_flags(a) & _PAGE_PROTNONE) &&
753         atomic_read(&mm->tlb_flush_pending))
754         return true;
755
756     return false;
757 }
```

Listing 6-46: arch/x86/include/asm/pgtable.h: `pte_accessible()`

If the PTE entry was marked present, or if it had the `_PAGE_PROTNONE` page table flag set and a flush was pending as determined by `struct mm_struct->tlb_flush_pending` (i.e. a mapping undergoing NUMA re-balancing requiring a flush, see the NUMA chapter for more on that), then it is deemed accessible.

Finally, if we do need to flush the TLB cache for this entry, we do this via `flush_tlb_page()`, the mechanism of which we discuss in detail in section 7.1.

Once this operation is complete any other CPU which attempts to access this memory will trigger a page fault and end up racing with us, rather than accessing the inappropriate physical address.

This means that we can now simply set the PTE without concern for TLB state, anything that races with us on this will either page fault as described above or receive the correct mapping, nothing is stale.

We proceed by adding the new anonymous folio to the reverse mapping via `page_add_new_anon_rmap()` (see section 7.0.12 for more on this) and add the folio to the relevant LRU list via `lru_cache_add_inactive_or_unevictable()` in order that it can be reclaimed (see section 11.7 for more on how the LRU lists function).

We set the PTE entry via `set_pte_at_notify()`, again as MMU notification is out of scope for the book we won't examine this aspect of the logic, but note that this ultimately calls `set_pte_at()` and `set_pte()` in turn.

Afterwards we have some careful logic around decrementing the copied-from folio's map count via `page_remove_rmap()` – the reverse mapping logic described in Chapter 7 might result in an inappropriate reuse of a visible folio. However this being placed after our clearing of the PTE entry avoids this issue.

Finally the `new_page` local variable is pointed at the old folio in order that it has a reference decremented rather than the newly assigned page.

Once this is complete, we ultimately reduce the reference count to the old folio by two both for the newly increment reference count and to account for the removal of the underlying reference from this now-CoW'd mapping.

Otherwise, for example if the PTE has been changed from beneath us (e.g. another page fault has raced with us and 'won'), we decrement the reference to the possibly newly allocated folio and the reference incremented source folio as necessary.

We additionally unlock the PTE page table lock, try free the swap cache if the folio was indeed copied via `free_swap_cache()`, which checks whether this lacks a map count and tries to acquire a lock to do so (see the swap chapter for more on this).

Finally, if a page was copied and it was not an unshare operation, we indicate so by returning `VM_FAULT_WRITE`.

6.10 Stack expansion

In `do_user_addr_fault()`, if a `struct vm_area_struct` (VMA) cannot be found which contains the faulting address, but one *was* found that sits above it

which possesses the `VM_GROWSDOWN` flag, then we invoke `expand_stack()` to expand the stack.

In Linux, stacks automatically expand up to the maximum specified by the stack size `rlimit` and this is the functionality which permits this. The check is performed by `expand_stack()`. We examine the case where the stack grows downwards, which is true of most modern architectures and certain of x86-64.

This in turn invokes `expand_downwards()` which is parameterised by the faulting address and the VMA which sits above it (eliding out of scope huge page, performance event tracking and debug logic) as shown in Listing 6-47.

```

2438 /*
2439  * vma is the first one with address < vma->vm_start. Have to extend vma.
2440  */
2441 int expand_downwards(struct vm_area_struct *vma,
2442                     unsigned long address)
2443 {
2444     struct mm_struct *mm = vma->vm_mm;
2445     struct vm_area_struct *prev;
2446     int error = 0;
2447
2448     address &= PAGE_MASK;
2449     if (address < mmap_min_addr)
2450         return -EPERM;
2451
2452     /* Enforce stack_guard_gap */
2453     prev = vma->vm_prev;
2454     /* Check that both stack segments have the same anon_vma? */
2455     if (prev && !(prev->vm_flags & VM_GROWSDOWN) &&
2456         vma_is_accessible(prev)) {
2457         if (address - prev->vm_end < stack_guard_gap)
2458             return -ENOMEM;
2459     }
2460
2461     /* We must make sure the anon_vma is allocated. */
2462     if (unlikely(anon_vma_prepare(vma)))
2463         return -ENOMEM;
2464
2465     /*
2466      * vma->vm_start/vm_end cannot change under us because the caller
2467      * is required to hold the mmap_lock in read mode. We need the
2468      * anon_vma lock to serialize against concurrent expand_stacks.
2469      */
2470     anon_vma_lock_write(vma->anon_vma);
2471
2472     /* Somebody else might have raced and expanded it already */
2473     if (address < vma->vm_start) {
2474         unsigned long size, grow;

```



```

2475
2476         size = vma->vm_end - address;
2477         grow = (vma->vm_start - address) >> PAGE_SHIFT;
2478
2479         error = -ENOMEM;
2480         if (grow <= vma->vm_pgoff) {
2481             error = acct_stack_growth(vma, size, grow);
2482             if (!error) {
2483                 /*
2484                  * vma_gap_update() doesn't support concurrent
2485                  * updates, but we only hold a shared
2486
2487                 mmap_lock
2488
2489                 * lock here, so we need to protect against
2490                 * concurrent vma expansions.
2491                 * anon_vma_lock_write() doesn't help here, as
2492                 * we don't guarantee that all growable vmas
2493                 * in a mm share the same root anon vma.
2494                 * So, we reuse mm->page_table_lock to guard
2495                 * against concurrent vma expansions.
2496                 */
2497                 spin_lock(&mm->page_table_lock);
2498                 if (vma->vm_flags & VM_LOCKED)
2499                     mm->locked_vm += grow;
2500                 vm_stat_account(mm, vma->vm_flags, grow);
2501                 anon_vma_interval_tree_pre_update_vma(vma);
2502                 vma->vm_start = address;
2503                 vma->vm_pgoff -= grow;
2504                 anon_vma_interval_tree_post_update_vma(vma);
2505                 vma_gap_update(vma);
2506                 spin_unlock(&mm->page_table_lock);
2507
2508             }
2509         }
2510         anon_vma_unlock_write(vma->anon_vma);
2511
2512         return error;
2513     }

```

Listing 6-47: mm/mmap.c: `expand_downwards()`

We start by examining the nearest VMA beneath the stack VMA and, if accessible (as determined by `vma_is_accessible()`, which simply checks whether either `VM_READ`, `VM_WRITE`, or `VM_EXEC` are set), checks to ensure that the end of that VMA sits at least `stack_guard_gap` bytes below this one (this defaults to 256 base pages, which for an architecture with 4 KiB pages is 1 MiB below, but can be adjusted via the kernel command line parameter `stack_guard_gap`).

We then ensure that a struct `anon_vma` object is assigned to the VMA.

We then must perform some intricate lock negotiation in order to prevent concurrent updates – write-locking the `anon_vma` to prevent concurrent stack expansion before re-checking that the VMA has not already been updated.

If it has not, then we first check to see if the required growth of the stack is less than or equal to the virtual page offset of the VMA `vm_pgoff` – if it was larger then the start address would fall below zero.

Next, we perform checks to ensure that the stack expansion is permitted, via `acct_stack_growth()`, which we will examine shortly.

If this is permitted, we then perform another intricate bit of lock wizardry by bizarrely acquiring the struct `mm_struct->page_table_lock`. This is required because we update the VMA gap (as discussed in section 5.1) which cannot be permitted to occur concurrently with other racing faults which also hold the `read` lock on the `mm_struct->mmap_lock` semaphore. See section 4.3.4.1 for more on process address space locking.

If the VMA has the `VM_LOCKED` flag set, indicating that all future allocations should be automatically `mlocked`, i.e. pinned in memory (specified via `mlockall()` with the `MCL_FUTURE` flag set or via `MAP_LOCKED` on `mmap()`), then we update the `mm_struct->locked_vm` statistic appropriately.

We update the `mm_struct->stack_vm` and `mm_struct->total_vm` statistics via `vm_stat_account()` before performing reverse mapping housekeeping (see Chapter 7) – removing references in the struct `anon_vma`-rooted interval trees referencing the VMA via the VMA's struct `anon_vma_chain` glue objects using `anon_vma_interval_tree_pre_update_vma()`.

This is needed because the entries in this tree are keyed by the VMA's `vm_pgoff` field which we are about to update.

We then adjust the `vm_start` and `vm_pgoff` fields to place the faulting address within the realm of the VMA, before reestablishing the reverse mapping via `anon_vma_interval_tree_post_update_vma()`.

The VMA has its subtree gap calculation updated via `vma_gap_update()` after which the `page_table_lock` is released as it is no longer required. Finally the `anon_vma` lock is released and the operation is complete.

Returning to `acct_stack_growth()` (eliding huge page logic) as shown in Listing 6-48.

```

2307 /*
2308  * Verify that the stack growth is acceptable and
2309  * update accounting. This is shared with both the
2310  * grow-up and grow-down cases.
2311  */
2312 static int acct_stack_growth(struct vm_area_struct *vma,
2313                             unsigned long size, unsigned long grow)
2314 {
2315     struct mm_struct *mm = vma->vm_mm;
2316     unsigned long new_start;
2317
2318     /* address space limit tests */

```

```

2319         if (!may_expand_vm(mm, vma->vm_flags, grow))
2320             return -ENOMEM;
2321
2322         /* Stack limit test */
2323         if (size > rlimit(RLIMIT_STACK))
2324             return -ENOMEM;
2325
2326         /* mlock limit tests */
2327         if (mlock_future_check(mm, vma->vm_flags, grow << PAGE_SHIFT))
2328             return -ENOMEM;
2329         :
2330
2331         /*
2332          * Overcommit.. This must be the final test, as it will
2333          * update security statistics.
2334          */
2335         if (security_vm_enough_memory_mm(mm, grow))
2336             return -ENOMEM;
2337
2338         return 0;
2339     }

```

Listing 6-48: mm/mmap.c: acct_stack_growth()

The first check is whether the address space limit `RLIMIT_AS` and data segment limit `RLIMIT_DATA` are being observed via `may_expand_vm()`.

Net, we explicitly ensure that the stack is not growing beyond the maximum permissible stack size as specified by `RLIMIT_STACK`, before checking if `RLIMIT_MEMLOCK` is exceeded, if applicable via `mlock_future_check()` (relevant when `mlockall()` has been called for the process with the `MCL_FUTURE` flag set).

Finally, `security_vm_enough_memory_mm()` is called to ensure that we do not exceed available memory depending on overcommit mode (see section 4.1) as well as performing any existing security hook checks.

6.11 Userland bad area handling

All accesses to incorrect areas of the address space in userland end up invoking `__bad_area_nosemaphore()`, whether invoked directly or from `bad_area_nosemaphore()`, `__bad_area()` or `bad_area()`.

Examining this function, eliding kernel handling as shown in Listing 6-49.

```

799 static void
800 __bad_area_nosemaphore(struct pt_regs *regs, unsigned long error_code,
801                       unsigned long address, u32 pkey, int si_code)
802 {
803     struct task_struct *tsk = current;
804     :
805
806     if (!(error_code & X86_PF_USER)) {
807         /* Implicit user access to kernel memory -- just oops */

```

```

813         page_fault_oops(regs, error_code, address);
814         return;
815     }
816
817     /*
818      * User mode accesses just cause a SIGSEGV.
819      * It's possible to have interrupts off here:
820      */
821     local_irq_enable();
822
823     /*
824      * Valid to do another page fault here because this one came
825      * from user space:
826      */
827     if (is_prefetch(regs, error_code, address))
828         return;
829
830     if (is_errata100(regs, address))
831         return;
832
833     sanitize_error_code(address, &error_code);
834
835     if (fixup_vdso_exception(regs, X86_TRAP_PF, error_code, address))
836         return;
837
838     if (likely(show_unhandled_signals))
839         show_signal_msg(regs, error_code, address, tsk);
840
841     set_signal_archinfo(address, error_code);
842
843     if (si_code == SEGV_PKUERR)
844         force_sig_pkuerr((void __user *)address, pkey);
845     else
846         force_sig_fault(SIGSEGV, si_code, (void __user *)address);
847
848     local_irq_disable();
849 }

```

Listing 6-49: arch/x86/mm/fault.c: x86-64 __bad_area_nosemaphore() userland handling

We start by considering an edge case – a kernel-mode fault but from a process with userland registers and at a userland address – this is specifically to handle a new (at the time of writing) feature from intel – CET – which provides security mitigations using a shadow userland stack (see x86/fault: #PF improvements, mostly related to USER bit for more on this). In this case we just trigger an oops via `page_fault_oops()`.

We make sure for the rest of the handling to enable local IRQ handling as there is now no reason for this these to be suppressed. We then check

two very specific cases – was this a fault that arose from the CPU prefetching memory? This is checked in `is_prefetch()` and is this an ancient AMD K8 processor with a very specific errata as checked by `is_errata100()`? In either case we simply exit and can refault if necessary.

The reported error is adjusted to avoid leaking information about kernel page tables in case of user mode access to kernel addresses via `sanitize_error_code()` as shown in Listing 6-50.

```

605 static void sanitize_error_code(unsigned long address,
606                               unsigned long *error_code)
607 {
608     /*
609      * To avoid leaking information about the kernel page
610      * table layout, pretend that user-mode accesses to
611      * kernel addresses are always protection faults.
612      *
613      * NB: This means that failed vsyscalls with vsyscall=none
614      * will have the PROT bit. This doesn't leak any
615      * information and does not appear to cause any problems.
616      */
617     if (address >= TASK_SIZE_MAX)
618         *error_code |= X86_PF_PROT;
619 }
```

Listing 6-50: `arch/x86/mm/fault.c: sanitize_error_code()`

We check whether we need to perform a fixup for VDSO via `fixup_vdso_exception()`, a mechanism by which certain system calls can use memory shared between the kernel and userland to improve performance. Discussion of this is out of scope for the book.

If unhandled segfaults are set to be displayed in the kernel log via `show_unhandled_signals`, then this is output via `show_signal_msg()`, an output many programmers will be familiar with as shown in Listing 6-51.

```

763 /*
764  * Print out info about fatal segfaults, if the show_unhandled_signals
765  * sysctl is set:
766  */
767 static inline void
768 show_signal_msg(struct pt_regs *regs, unsigned long error_code,
769               unsigned long address, struct task_struct *tsk)
770 {
771     const char *loglvl = task_pid_nr(tsk) > 1 ? KERN_INFO : KERN_EMERG;
772
773     if (!unhandled_signal(tsk, SIGSEGV))
774         return;
775
776     if (!printk_ratelimit())
777         return;
```

```

778
779     printk("%s[%d]: segfault at %lx ip %px sp %px error %lx",
780           loglvl, tsk->comm, task_pid_nr(tsk), address,
781           (void *)regs->ip, (void *)regs->sp, error_code);
782
783     print_vma_addr(KERN_CONT " in ", regs->ip);
784
785     printk(KERN_CONT "\n");
786
787     show_opcodes(regs, loglvl);
788 }

```

Listing 6-51: arch/x86/mm/fault.c: show_signal_msg()

The architecture-specific struct `thread_struct` object contained in the process's struct `task_struct->thread` field is updated by `set_signal_archinfo()`, storing the error code and faulting address.

The signal is then transmitted to the userland process via either `force_sig_pkuerr()` if the error arose due to protection keys (a topic which is out of scope for the book) or `force_sig_fault()` for a standard SIGSEGV fault.

Finally, local IRQs are disabled as the fault handling expects them to be when this function is completed.

6.12 Special mappings

Some userland mappings are deemed 'special', defined by there being no struct `page` (and thus no struct `folio` object) to describe the underlying physical memory range. These might actually exist, but it might not be desired that the usual reference count mechanisms be applied to them (e.g. kernel memory mapped into userland).

For modern architectures a user-definable page table flag is used to designate a mapping as being special – for x86-64 this flag is `_PAGE_SPECIAL`. This makes checking for this simple in `pte_special()`.

There are broadly two forms of special mappings, characterised by their mutually exclusive struct `vm_area_struct` (VMA) flags – `VM_PFNMAP`, which represents a *linear* mapping between virtual addresses and PFNs, and `VM_MIXEDMAP`, which represents a range that can mix mappings that are PFN-only and those with describing folios.

The `vm_normal_page()` function is used to obtain a page object for at a specific address within a specific VMA, examining this (eliding out of scope `devmap` logic and logic for architectures that do not support a special mapping page table flag) as shown in Listing 6-52.

```

570 /*
571  * vm_normal_page -- This function gets the "struct page" associated with a
572  *                    pte.
573  *
574  * "Special" mappings do not wish to be associated with a "struct page" (
575  *                    either

```

```

574 * it doesn't exist, or it exists but they don't want to touch it). In this
575 * case, NULL is returned here. "Normal" mappings do have a struct page.
576 *
577 * There are 2 broad cases. Firstly, an architecture may define a pte_special
578 *   ()
579 * pte bit, in which case this function is trivial. Secondly, an architecture
580 * may not have a spare pte bit, which requires a more complicated scheme,
581 * described below.
582 *
583 * A raw VM_PFNMAP mapping (ie. one that is not COWed) is always considered a
584 * special mapping (even if there are underlying and valid "struct pages").
585 * COWed pages of a VM_PFNMAP are always normal.
586 *
587 * The way we recognize COWed pages within VM_PFNMAP mappings is through the
588 * rules set up by "remap_pfn_range()": the vma will have the VM_PFNMAP bit
589 * set, and the vm_pgoff will point to the first PFN mapped: thus every
590 * special
591 * mapping will always honor the rule
592 *
593 *     pfn_of_page == vma->vm_pgoff + ((addr - vma->vm_start) >> PAGE_SHIFT)
594 *
595 * And for normal mappings this is false.
596 *
597 * This restricts such mappings to be a linear translation from virtual
598 * address
599 * to pfn. To get around this restriction, we allow arbitrary mappings so long
600 * as the vma is not a COW mapping; in that case, we know that all ptes are
601 * special (because none can have been COWed).
602 *
603 *
604 * In order to support COW of arbitrary special mappings, we have VM_MIXEDMAP.
605 *
606 * VM_MIXEDMAP mappings can likewise contain memory with or without "struct
607 * page" backing, however the difference is that _all_ pages with a struct
608 * page (that is, those where pfn_valid is true) are refcounted and considered
609 * normal pages by the VM. The disadvantage is that pages are refcounted
610 * (which can be slower and simply not an option for some PFNMAP users). The
611 * advantage is that we don't have to follow the strict linearity rule of
612 * PFNMAP mappings in order to support COWable mappings.
613 *
614 */
615
616 struct page *vm_normal_page(struct vm_area_struct *vma, unsigned long addr,
617                             pte_t pte)
618 {
619     unsigned long pfn = pte_pfn(pte);
620
621     if (IS_ENABLED(CONFIG_ARCH_HAS_PTE_SPECIAL)) {

```

```

618         if (likely(!pte_special(pte)))
619             goto check_pfn;
620         if (vma->vm_ops && vma->vm_ops->find_special_page)
621             return vma->vm_ops->find_special_page(vma, addr);
622         if (vma->vm_flags & (VM_PFNMAP | VM_MIXEDMAP))
623             return NULL;
624         if (is_zero_pfn(pfn))
625             return NULL;
626         :
627         print_bad_pte(vma, addr, pte, NULL);
628         return NULL;
629     }
630     :
631 check_pfn:
632     if (unlikely(pfn > highest_memmap_pfn)) {
633         print_bad_pte(vma, addr, pte, NULL);
634         return NULL;
635     }
636
637     /*
638      * NOTE! We still have PageReserved() pages in the page tables.
639      * eg. VDSO mappings can cause them to exist.
640      */
641 out:
642     return pfn_to_page(pfn);
643 }

```

Listing 6-52: mm/memory.c: vm_normal_page()

This will return NULL in the case of the zero page or special mappings except when a VMA's struct `vm_operations_struct` `vm_ops->find_special_page` handler is provided. This is only used in one place (within the xen driver, which is a virtualisation framework), so this is unlikely to be the case.

The `VM_MIXEDMAP` variant of this permits insertion of individual pages into an existing VMA via `vm_insert_pages()` for multiple pages and `vm_insert_page()` for individual ones. A simpler interface for this is `vm_map_pages()`. Non-page entries can be mapped as part of a page fault handler via `vmf_insert_mixed()`.

We will not examine `VM_MIXEDMAP` mappings in detail, as it is out of scope for the book, however it is important to be aware such mappings exist and are used for mapping kernel or device memory to userland.

The `VM_PFNMAP` approach is the simpler of the two primary forms of special mappings, typically used to simply map ranges of kernel memory into userland, typically via `remap_pfn_range()`, which ultimately invokes `remap_pfn_range_notrack()` to do the heavy lifting as shown in Listing 6-53.

```

2471 /*
2472  * Variant of remap_pfn_range that does not call track_pfn_remap. The caller
2473  * must have pre-validated the caching bits of the pgprot_t.
2474  */

```



```

2475 int remap_pfn_range_notrack(struct vm_area_struct *vma, unsigned long addr,
2476                             unsigned long pfn, unsigned long size, pgprot_t prot)
2477 {
2478     pgd_t *pgd;
2479     unsigned long next;
2480     unsigned long end = addr + PAGE_ALIGN(size);
2481     struct mm_struct *mm = vma->vm_mm;
2482     int err;
2483
2484     if (WARN_ON_ONCE(!PAGE_ALIGNED(addr)))
2485         return -EINVAL;
2486
2487     /*
2488      * Physically remapped pages are special. Tell the
2489      * rest of the world about it:
2490      *   VM_IO tells people not to look at these pages
2491      *   (accesses can have side effects).
2492      *   VM_PFNMAP tells the core MM that the base pages are just
2493      *   raw PFN mappings, and do not have a "struct page" associated
2494      *   with them.
2495      *   VM_DONTEXPAND
2496      *   Disable vma merging and expanding with mremap().
2497      *   VM_DONTDUMP
2498      *   Omit vma from core dump, even when VM_IO turned off.
2499      *
2500      * There's a horrible special case to handle copy-on-write
2501      * behaviour that some programs depend on. We mark the "original"
2502      * un-COW'ed pages by matching them up with "vma->vm_pgoff".
2503      * See vm_normal_page() for details.
2504      */
2505     if (is_cow_mapping(vma->vm_flags)) {
2506         if (addr != vma->vm_start || end != vma->vm_end)
2507             return -EINVAL;
2508         vma->vm_pgoff = pfn;
2509     }
2510
2511     vma->vm_flags |= VM_IO | VM_PFNMAP | VM_DONTEXPAND | VM_DONTDUMP;
2512
2513     BUG_ON(addr >= end);
2514     pfn -= addr >> PAGE_SHIFT;
2515     pgd = pgd_offset(mm, addr);
2516     flush_cache_range(vma, addr, end);
2517     do {
2518         next = pgd_addr_end(addr, end);
2519         err = remap_p4d_range(mm, pgd, addr, next,
2520                               pfn + (addr >> PAGE_SHIFT), prot);
2521         if (err)

```

```

2522         return err;
2523     } while (pgd++, addr = next, addr != end);
2524
2525     return 0;
2526 }

```

Listing 6-53: mm/memory.c: remap_pfn_range_notrack()

This walks the page tables of the userland mappings, allocating where necessary, placing the specified kernel PFN in the PTE in `remap_pte_range()`. Note that the `vm_pgoff` is set to the PFN of the beginning of the range if this is a Copy On Write mapping. This is required for architectures that do not have a special PTE flag so have to differentiate using this to identify the original mapping.

When a Copy on Write fault occurs for a Copy on Write mapping (i.e. not shared), this operation is handled by `__wp_page_copy_user()` (eliding logic for architectures that don't have hardware accessed bits) as shown in Listing 6-54.

```

2844 static inline bool __wp_page_copy_user(struct page *dst, struct page *src,
2845                                       struct vm_fault *vmf)
2846 {
2847     bool ret;
2848     void *kaddr;
2849     void __user *uaddr;
2850     bool locked = false;
2851     struct vm_area_struct *vma = vmf->vma;
2852     struct mm_struct *mm = vma->vm_mm;
2853     unsigned long addr = vmf->address;
2854
2855     if (likely(src)) {
2856         copy_user_highpage(dst, src, addr, vma);
2857         return true;
2858     }
2859
2860     /*
2861      * If the source page was a PFN mapping, we don't have
2862      * a "struct page" for it. We do a best-effort copy by
2863      * just copying from the original user address. If that
2864      * fails, we just zero-fill it. Live with it.
2865      */
2866     kaddr = kmap_atomic(dst);
2867     uaddr = (void __user *) (addr & PAGE_MASK);
2868     :
2893     /*
2894      * This really shouldn't fail, because the page is there
2895      * in the page tables. But it might just be unreadable,
2896      * in which case we just give up and fill the result with
2897      * zeroes.

```

```

2898     */
2899     if (__copy_from_user_inatomic(kaddr, uaddr, PAGE_SIZE)) {
2900         if (locked)
2901             goto warn;
2902
2903         /* Re-validate under PTL if the page is still mapped */
2904         vmf->pte = pte_offset_map_lock(mm, vmf->pmd, addr, &vmf->ptl);
2905         locked = true;
2906         if (!likely(pte_same(*vmf->pte, vmf->orig_pte))) {
2907             /* The PTE changed under us, update local tlb */
2908             update_mmu_tlb(vma, addr, vmf->pte);
2909             ret = false;
2910             goto pte_unlock;
2911         }
2912
2913         /*
2914          * The same page can be mapped back since last copy attempt.
2915          * Try to copy again under PTL.
2916          */
2917         if (__copy_from_user_inatomic(kaddr, uaddr, PAGE_SIZE)) {
2918             /*
2919              * Give a warn in case there can be some obscure
2920              * use-case
2921              */
2922             warn:
2923                 WARN_ON_ONCE(1);
2924                 clear_page(kaddr);
2925             }
2926         }
2927
2928         ret = true;
2929
2930     pte_unlock:
2931         if (locked)
2932             pte_unmap_unlock(vmf->pte, vmf->ptl);
2933         kunmap_atomic(kaddr);
2934         flush_dcache_page(dst);
2935
2936         return ret;
2937     }

```

Listing 6-54: mm/memory.c: __wp_page_copy_user()

This is also used for non-special mapping copying, however this is a trivial case handled by `copy_user_highpage()`, with the rest of the logic dedicated to special mappings.

In order to perform the Copy On Write, the kernel simply copies from the source page using its userland mapping (which will have the appropriate

PFN set) into the newly allocated page via `__copy_from_user_inatomic()`, which is an unsafe version of the usual `copy_from_user()` function (see section 8.1.1 for more on this).

If the copy fails, we try again until PTE page table lock, if this fails we simply clear the target page.

We have moved quickly through this topic as, while it's useful to be aware of these types of mappings and how they're handled on page fault, the intricate details of how these are used sit outside the scope of the book.

7

REVERSE MAPPINGS

When we try to reclaim or migrate memory, we operate at the folio level. We need the ability to efficiently look up VMAs which map that memory in order that we can ensure the mapping correctly accounts for the change in the physical pages which back it. The means by which we map in reverse from the folio to VMAs is termed the 'reverse mapping', which we explore in this chapter.

A key operation within the kernel is looking up the VMAs which map a given folio. Consider the motivating example of reclaim trying to unmap a freed page from the VMAs which reference it (via `try_to_unmap()`) – in reclaim we only have the folio, so we need to be able to discover which VMAs it is mapped into (remembering that forking means a single anonymous page could be mapped into many processes as Copy-on-Write mappings). This is termed the folio's *reverse mapping* (the 'forward' mapping being the page table mappings from virtual address to physical).

VMAs are ephemeral by nature – they can be split, merged and forked, and thus have many-to-many relationships spanning process address spaces. In this section we will focus on how we trace back folios to their VMAs during VMA lifetime.

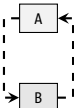
7.0.1 Anonymous reverse mappings

This is a confusing topic so let's examine things from first principles – consider what would happen if we simply mapped each folio directly to a single VMA – immediately this would be unworkable, as on fork a folio is referenced by two different VMAs in two different processes, each set to Copy-on-Write.

How about each folio maintaining a list of related VMAs? This would be unworkable as there would both be *substantial* duplication and it would be prohibitively inefficient to perform splitting, merging and forking operations.

We must therefore maintain an *intermediary* object which folios can point to. We access this via the struct `folio->mapping` field. If the folio is part of the page cache, this field points to a struct `address_space` object which describes a page cache object (typically a file), otherwise it points to a struct `anon_vma` object which describes a set of anonymous folios.

Firstly let's examine how a newly initialised struct `anon_vma` object and its related struct `anon_vma_chain` (AVC) and struct `vm_area_struct` (VMA) objects interact, noting that:-

-  Denotes that B is a member of a list declared in A.
-  Denotes that B is a node in a red/black tree rooted in A.

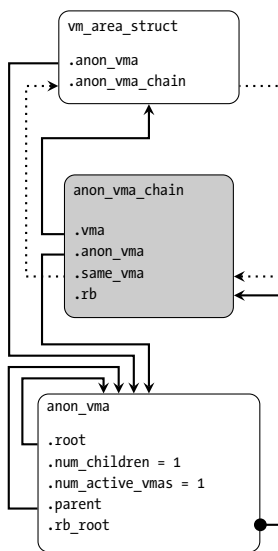


Figure 7-1: Initial struct `anon_vma` state

It's important to note that a newly initialised VMA object will not have a `struct anon_vma` object allocated or attached to it until anonymous folios are faulted in. We will examine this more closely when we examine the related code in detail.

The `struct anon_vma_chain` (AVC) object sits between `struct vm_area_struct` (VMA) and `anon_vma` objects and allows each VMA to maintain a list of `anon_vma` objects containing folios mapped into it. Its usefulness will become apparent as we work through more examples.

We maintain an *interval tree*^{*} between each `anon_vma` and `anon_vma_chain` object, rooted in the `anon_vma.rb_root` field, with each node kept in the `anon_vma_chain.rb` field. This is keyed by the `folio->index` field which is set to the virtual page offset[†] of the address at which it is mapped (via `__page_set_anon_rmap()`) and determines each folio's offset within the mapping.

This is critical, as after a VMA split different folios can map to different VMAs depending on their offset within the mapping, so when trying to determine the VMA with which a folio is associated we need to be able to do so efficiently. In addition we try to merge adjacent `anon_vma` objects where possible, so the ability for a single object to span many mappings is very useful.

Each `anon_vma` maintains pointers to parent and root `anon_vma` objects which are updated on process fork and define the fork hierarchy. Locking is performed using the root `anon_vma`'s `rwsem` semaphore for all objects in the hierarchy.

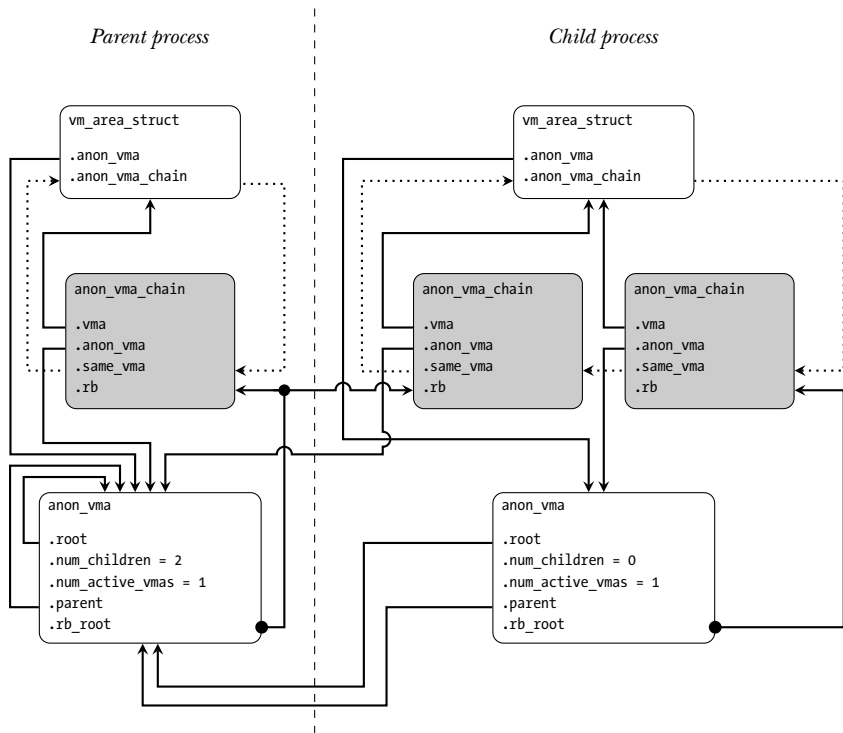
The number of `anon_vma` objects whose parent field points at this `anon_vma` is tracked in the `num_children` field, which is primarily used to permit reuse of empty `anon_vma` objects.

Each VMA backed by anonymous memory points to an `anon_vma` object which newly faulted-in memory will be mapped to. The number of VMAs that point to an `anon_vma` is tracked in the `num_active_vmas` field, which can also be utilised to permit reuse of empty `anon_vma` objects.

The root and parent fields are initialised to point to the `struct anon_vma` itself as we will need to use the only lock available to us and on unlinking this root object, decrementing the parent's `num_children` count indeed should decrement this object's count. The initial case is trivial, things get more interesting once we fork a process:-

*. An interval tree is a red/black search tree which makes it easy to traverse ordered interval ranges.

†. The virtual page offset of a folio is its mapped virtual address divided by the page size. If you imagine all base pages of virtual memory to be an array, this would be an entry's index. Note that if a VMA is moved via `mremap()`, the *original* virtual page offset is retained.

Figure 7-2: Forked struct `anon_vma`

After the process forks we have two sets of folios we need to track – those belonging to the parent process which are mapped into the child process as Copy-on-Write, and folios that been written to and therefore copied.

In order to manage these two sets of folios, we must maintain two separate struct `anon_vma` objects – the existing one associated with the parent process one which represents the CoW folios and a new one which represents folios which have been copied and is linked to the child process.

Since the child process's VMA can reference the parent's CoW folios, we need to connect the parent's `anon_vma` to the child's VMA so CoW folios are able to reference back to *both* VMAs, as well as providing a link in the other direction so VMA operations can be performed for both VMAs correctly.

However, once a folio is copied to the child process (and this is no longer CoW), it can only be associated with the child VMA, so we need to maintain a link between the new `anon_vma` and *only* the child VMA.

The above demonstrates that we *must* maintain separate interval trees of VMAs for each `anon_vma` and separate lists of `anon_vmas` for each VMA since CoW implies that parent and child process address spaces have asymmetric dependencies upon each other's anonymous folios.

Since no node can exist in more than one tree at any one time, nor can one list node exist in more than one list at the same time*, we must maintain entirely separate trees for each `anon_vma` and entirely separate lists for each VMA.

In order to achieve this we introduce the struct `anon_vma_chain` (AVC) type – which simply act as nodes connecting VMAs and `anon_vma` objects, each representing a connection between a specific VMA and `anon_vma`. Each VMA owns its own AVCs, storing them on their `anon_vma_chain` list, each of which can reference any `anon_vma` object.

Note that the interval tree entries for the parent and child VMAs of the shared `anon_vma` will have the same keys, meaning that looking up folios in the address range will return *both* VMAs (via `anon_vma_interval_tree_foreach()`).

Another means by which reverse mapping manipulations can occur is VMA splitting and merging (see section 5.1 for more details), for which these mechanisms remain equally useful.

Consider a series of folios, each of which are associated with a VMA, which shortly afterwards has its middle portion unmapped. We now split the VMA into two, but it would be inefficient to adjust each folio to point at different `anon_vma` objects.

This is where the interval tree structure comes into its own – we can simply remove all AVCs associated with the VMA from the `anon_vma`'s interval tree, create a new VMA and associated AVC, then link to the appropriate AVC based on the struct `folio->index` field:-

*. To see why, consider what make up these data structures – list nodes can only reference previous and subsequent nodes, so there is simply no way to link two lists together without them becoming one aggregate list, equally binary tree nodes must reference up to two nodes below them, traversing from one tree to another is not possible.

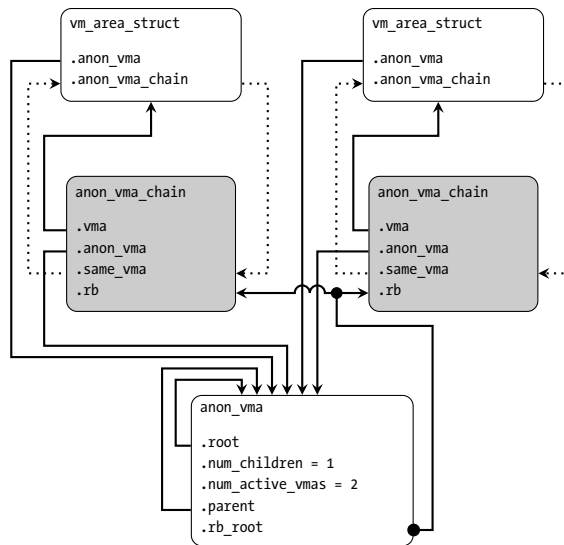


Figure 7-3: *struct anon_vma* state after VMA split

Here we simply create a new `anon_vma_chain` object for the newly split VMA, link both VMAs to the struct `anon_vma` object (as both will place new folios in this block), and link the `anon_vma` object to both AVC objects in the interval tree, mapping each part of the split range to their respective VMAs.

Again, the AVC object represents a connection between a VMA and an `anon_vma` object, acting as the ‘glue’ between these objects. It’s clear here that this structure allows for arbitrary VMA manipulation and forking.

Something worth noting here is that in each instance where we have needed to create a new AVC object, we have made at least one *copy* of the connection to a preexisting `anon_vma` object.

A VMA of a forked child process needs to reference CoW pages and a split VMA needs to reference the shared `anon_vma`. Note that if the VMA being copied from had more than 1 AVC object, it follows that we would want to copy *all* of them (as each of them reference things the copying VMA must also reference).

We will return to this shortly, as *cloning* an `anon_vma` is precisely how this functionality is implemented.

Let’s examine what happens after figure 7-3 is merged back together:-

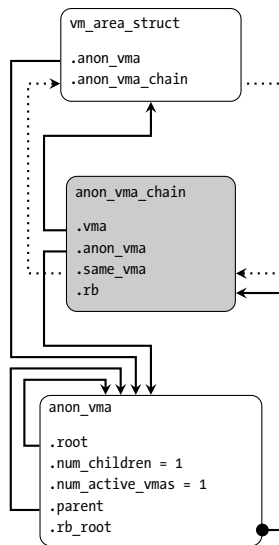


Figure 7-4: `struct anon_vma` state after split VMAs merged

As you can see, this is identical to figure 7-1, i.e. the original state is restored. The kernel endeavours to ensure that each struct `anon_vma_chain` object denotes a connection between distinct struct `vm_area_struct` (VMA) and struct `anon_vma` objects.

What about the merge of two separate VMAs with independent anon_vma objects? This is prohibited and contiguous VMAs remain separate. This is because a VMA can designate one and only one anon_vma object to map newly faulted folios to and it would be prohibitively expensive to remap folios to a single anon_vma. This is checked by `is_mergeable_anon_vma()`, which we will examine in detail shortly:-

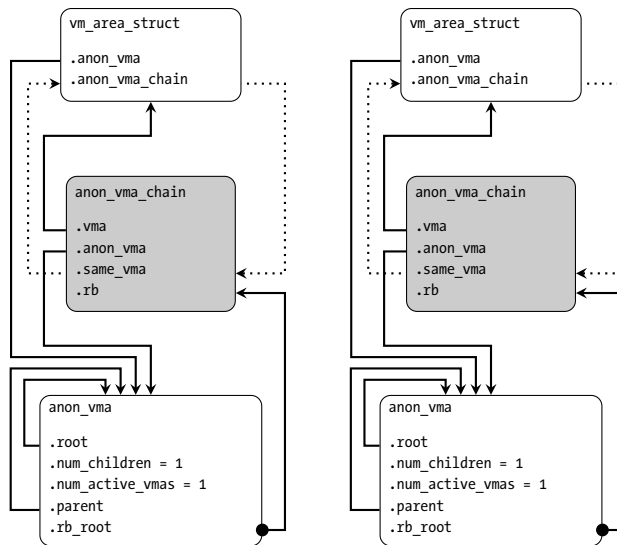


Figure 7-5: VMAs with separate struct anon_vmas after attempted merge

7.0.2 Key points

Since this is such a complicated and confusing topic, let's highlight some useful key points which can help us reason about anonymous reverse mappings:-

- The purpose of a reverse mapping is simply to be able to locate all the VMAs which map a given folio.
- Each struct anon_vma object simply represents a collection of folios which are used for anonymous mappings. They exist because VMAs are subject to a lot of change and it would be slow to look up folios each time this happened, so instead we map folios to this object.
- A struct anon_vma_chain (AVC) object simply represents a connection between a VMA and an anon_vma.
- There exists one and only one AVC object connecting any distinct pair of VMA and anon_vma objects.
- Each AVC connects in both directions – VMAs are able to determine which anon_vma objects they potentially map using the list headed by their anon_vma_chain field, and anon_vma objects can determine VMAs via their interval tree rooted in their rb_root field.
- Each anon_vma object can reference multiple different VMAs using the AVC objects as connectors using its interval tree rooted in rb_root.
- We determine which VMA belongs to which folio by using the folio's index field, which for anonymous mappings specifies the *virtual* page offset of the folio. This is then indexed into the interval tree rooted in the

`anon_vma`'s `rb_root` field to obtain an AVC from which we can determine the VMA.

- The whole reason for this complexity is that VMAs can be split and merged as memory is mapped, unmapped and modified by things such as `mprotect()` and most vexingly of all – forking of processes.
- If a VMA has more than one entry in its `anon_vma_chain` list this means the memory it maps is forked from a parent process, thus you can determine whether a VMA is a 'child VMA' by checking the length of this list.
- The fact that forking exists means that an `anon_vma` can reference VMAs from multiple different processes.
- The fact that VMAs can be split and we try to reuse adjacent VMA `anon_vma` objects, an `anon_vma` object can also reference multiple VMAs in the same process.
- `anon_vma` objects themselves exist in a tree (technically it's not a tree as the root of the tree connects to itself, but within the kernel it's referred to as a tree). This is used to keep a track of child node count for the purposes of potential `anon_vma` reuse and to link each `anon_vma` to its root in order to both share an `anon_vma` lock across the entire fork hierarchy and to enable us to keep track of un-CoW'd, shared folios.
- Looking up VMAs from the reverse mapping when a fork has occurred can result in VMAs that *may* contain the folio being sought – as it may have since been CoW'd to a child process. It is up to the code 'walking' the reverse mapping (see section 9.9.3) to determine whether the folio actually is present in that VMA. Mappings can change at any time, so this must in general be checked. See section 7.0.16 for more on this.

7.0.3 File reverse mappings

We touch on the page cache in Chapter ?? and cover it in substantial detail in the page cache chapter, however the reverse mapping logic is relatively straightforward, so let's examine this here.

Life is substantially easier for *shared* file-backed reverse mappings as splitting/merging of VMAs and forking of processes doesn't change the fact that, fundamentally, the same underlying object is referenced and thus the complex machinery required for anonymous reverse mappings can be avoided.

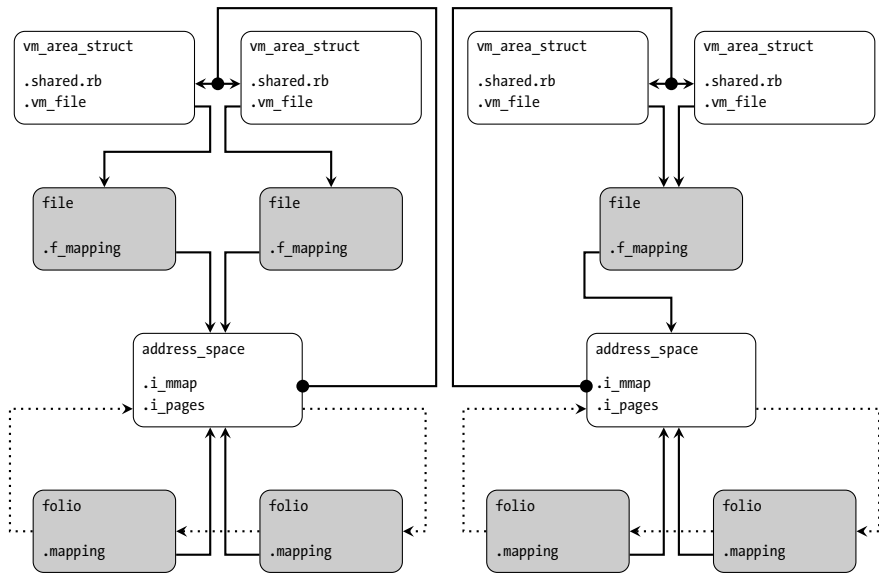
The page cache equivalent of the `anon_vma` object described above is the `struct address_space` object. This represents an object in the page cache, usually a file and a VMA references this in its `vm_file` field via a `struct file` object (which represents a process's open file).

Similar to anonymous reverse mappings a page cache folio maps to its `address_space` via the folio's `mapping` field, with its offset into the mapping specified via its `index` field.

Obtaining VMAs that may map a cache entry is straightforward – the `i_mmap` field is a red/black interval tree containing VMA nodes contained

within each VMA's `shared.rb` union field (the lack of Copy-on-Write obviates the need for an intermediate object equivalent to `anon_vma_chain`).

Figure 7-6: Page cache-backed reverse mapping



Each page cache-backed folio provides a reverse mapping via the mapping field pointing at the struct `address_space` object representing the page cache entry.

Each `address_space` has an `i_mmap` field which is the root of a red/black interval tree of struct `vm_area_struct` (VMA) objects which are keyed on each struct `folio`'s index field which indicates the page offset of this folio within the page cache entry (as opposed to anonymous page folios which put virtual page offset in this field). We can do so as these folios will only ever span a single page cache entry.

File-backed VMAs can split and merge just like their anonymous counterparts, differing in that the same `address_space` is always referenced. On the other hand, each struct `file` references an open file with a specific offset and are, by their nature, ephemeral.

Separate VMAs within the same process may reference the same file object (e.g. if a file-backed VMA is split by having a page in the middle unmapped), or entirely distinct file objects (e.g. mapped from different file descriptors).

Forked *shared* file mappings continue to reference the same `address_space` underneath, so we don't need an additional data structure sat between this object and VMAs to account for this and can therefore simply reference VMAs directly from the `i_mmap` interval tree.

An interesting thing to note is that `address_space` objects actually do reference their folios in `i_pages`. This field is an *eXtensible array* which is a clever means of storing an (extensible of course) array of pointers to, in this case, folios.

This permits the efficient lookup of folios associated with a page cache entry (e.g. `filemap_get_folios()`), an operation which is not possible (or necessary) for anonymous mappings.

Being able to reference folios from a struct `address_space` is *critical* because files can have changes made to them which require underlying folios to be evicted, events which occur at a file granularity (rather than at a folio granularity like anonymous memory being evicted). An example of this is `filemap_read()` which permits reading from the page cache even in the absence of a mapping.

An awkward edge case is a *privately* mapped file-backed mapping (e.g. called with a file descriptor to map a file but with the `MAP_PRIVATE` flag set). This is described in detail in section 6.6, however in brief – VMAs mapped like this can appear in both `address_space i_mmap` trees and `anon_vma rb_root` trees (via struct `anon_vma_chain` objects) *at the same time*.

These are strange beasts – Copy-on-Write mappings to the relevant page cache entries which, on write, become entirely anonymous mappings. However belonging to both file-backed and anonymous reverse mapping trees means that they do not behave entirely like an anonymous mapping even after the Copy-on-Write has occurred.

When file-mapped pages are truncated (e.g. overwritten or reduced in size), `i_mmap` is used to look up VMAs and clear down page table mappings (see section 6.6 for more on this edge case) and see the chapter on page cache for a detailed exploration of file-mapped pages function.

This means that `MAP_PRIVATE` file mappings, which still exist in the relevant `i_mmap` tree, have this done to them *even if they have performed Copy-on-Write*, and all changes to the anonymous backing folios is lost.

For more details on the `address_space` type and a broader discussion of the page cache, see the chapter dedicated to this topic.

7.0.4 Anonymous reverse mapping types

7.0.4.1 Relevant VMA fields

After that whirlwind tour through reverse mapping, let's take a step back and examine each type in detail, starting with the fields relevant to `anon_vma` objects present in the struct `vm_area_struct` data type:-

```

403 struct vm_area_struct {
404     :
405     /*
406      * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
407      * list, after a COW of one of the file pages. A MAP_SHARED vma
408      * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
409      * or brk vma (with NULL file) can only be in an anon_vma list.
410      */
411     struct list_head anon_vma_chain; /* Serialized by mmap_lock &
                                     * page_table_lock */

```



```

462         struct anon_vma *anon_vma;      /* Serialized by page_table_lock */
463         :
483     } __randomize_layout;

```

Listing 7-1: include/linux/mm_types.h: struct vm_area_struct anon_vma fields

Examining each field:-

- anon_vma_chain – This heads the list containing AVC objects related to this VMA, linked by the same_vma field.
- anon_vma – This points to the anon_vma object which newly faulted-in anonymous folios will be mapped to, which is referred to as its *active* VMA. Importantly – if no folios have yet been faulted in, this field will be NULL.

7.0.4.2 struct anon_vma

And now examining struct anon_vma, the core anonymous reverse mapping data structure:-

```

17  /*
18   * The anon_vma heads a list of private "related" vmas, to scan if
19   * an anonymous page pointing to this anon_vma needs to be unmapped:
20   * the vmas on the list will be related by forking, or by splitting.
21   *
22   * Since vmas come and go as they are split and merged (particularly
23   * in mprotect), the mapping field of an anonymous page cannot point
24   * directly to a vma: instead it points to an anon_vma, on whose list
25   * the related vmas can be easily linked or unlinked.
26   *
27   * After unlinking the last vma on the list, we must garbage collect
28   * the anon_vma object itself: we're guaranteed no page can be
29   * pointing to this anon_vma once its vma list is empty.
30   */
31  struct anon_vma {
32      struct anon_vma *root;          /* Root of this anon_vma tree */
33      struct rw_semaphore rwsem;      /* W: modification, R: walking the
34      list */
35      /*
36       * The refcount is taken on an anon_vma when there is no
37       * guarantee that the vma of page tables will exist for
38       * the duration of the operation. A caller that takes
39       * the reference is responsible for clearing up the
40       * anon_vma if they are the last user on release
41       */
42      atomic_t refcount;
43      /*
44       * Count of child anon_vmas. Equals to the count of all anon_vmas that

```

```

45      * have ->parent pointing to this one, including itself.
46      *
47      * This counter is used for making decision about reusing anon_vma
48      * instead of forking new one. See comments in function anon_vma_clone
49      *
50      unsigned long num_children;
51      /* Count of VMAs whose ->anon_vma pointer points to this object. */
52      unsigned long num_active_vmas;
53
54      struct anon_vma *parent;          /* Parent of this anon_vma */
55
56      /*
57       * NOTE: the LSB of the rb_root.rb_node is set by
58       * mm_take_all_locks() _after_ taking the above lock. So the
59       * rb_root must only be read/written after taking the above lock
60       * to be sure to see a valid next pointer. The LSB bit itself
61       * is serialized by a system wide lock only visible to
62       * mm_take_all_locks() (mm_all_locks_mutex).
63       */
64
65      /* Interval tree of private "related" vmas */
66      struct rb_root_cached rb_root;
67  };

```

Listing 7-2: include/linux/rmap.h: struct anon_vma

Examining each field:-

- **root** – This points at the struct anon_vma object which sits at the top of all forked processes above the one that owns this anon_vma. This is primarily used to share the root object's rwsem lock between all forked objects.
- **rwsem** – A read/write semaphore used to synchronise access to all anon_vma objects and the rb nodes contained within struct anon_vma_chain objects.

Note that we do not lock at an anon_vma granularity, rather we lock the *entire* anon_vma tree by locking only the *root* anon_vma object (i.e. the ultimate parent of all the forks under which an anon_vma sits, as referenced by the root field).

Being a read/write semaphore, read locks are non-exclusive with respect to other read locks but write locks are exclusive. This lock is typically acquired via anon_vma_lock_read() and anon_vma_lock_write() and unlocked via anon_vma_unlock_read() and anon_vma_unlock_write(). There is also a try variant of the read – anon_vma_trylock_read().

- **refcount** – An atomic reference count representing the number of times this object is referenced. Incremented by get_anon_vma() and decremented by put_anon_vma() (which performs heavy lifting in

`__put_anon_vma()`). This is also manipulated elsewhere directly within internal `rmap` functions.

- `num_children` – Counts the number of `anon_vma` objects whose `parent` field points at this `anon_vma`. Used along with `num_active_vmas` to determine whether `anon_vma` objects can be reused in `anon_vma_clone()`.
- `num_active_vmas` – Counts the number of VMAs whose `anon_vma` field points at this object. Used in conjunction with `num_children` to permit `anon_vma` object reuse.
- `parent` – Points at the `anon_vma` object of the parent process which forked this one. Used to maintain correct `num_children` counts, incrementing in `anon_vma_fork()` and decrementing in `unlink_anon_vmas()`.
- `rb_root` – The root of a red/black interval tree with `anon_vma_chain` nodes connecting the `anon_vma` object to VMAs which map folios referencing it. The interval tree is keyed on the folio's `index` field which, for anonymous memory, is set to the virtual page offset of the address at which it was *originally* mapped* (via `__page_set_anon_rmap()`). Each VMA specifies its starting virtual page offset in their `vm_pgoff` field. There can be multiple VMAs present within the same interval range (e.g. forked process VMAs), different VMAs can be mapped to different virtual page offsets (e.g. split VMAs) or a combination of both.

7.0.4.3 struct anon_vma_chain

Examining the `struct anon_vma_chain` object:-

```

69  /*
70   * The copy-on-write semantics of fork mean that an anon_vma
71   * can become associated with multiple processes. Furthermore,
72   * each child process will have its own anon_vma, where new
73   * pages for that process are instantiated.
74   *
75   * This structure allows us to find the anon_vmas associated
76   * with a VMA, or the VMAs associated with an anon_vma.
77   * The "same_vma" list contains the anon_vma_chains linking
78   * all the anon_vmas associated with this VMA.
79   * The "rb" field indexes on an interval tree the anon_vma_chains
80   * which link all the VMAs associated with this anon_vma.
81   */
82  struct anon_vma_chain {
83      struct vm_area_struct *vma;
84      struct anon_vma *anon_vma;
85      struct list_head same_vma;    /* locked by mmap_lock & page_table_lock
                                     */

```

*. `mremap()` retains the *original* virtual page offsets if it moves a VMA. See section 8.2.3 for a detailed discussion of this.

```

86      struct rb_node rb;                                /* locked by anon_vma->rwsem
      */
87      unsigned long rb_subtree_last;
88 #ifdef CONFIG_DEBUG_VM_RB
89      unsigned long cached_vma_start, cached_vma_last;
90 #endif
91 };

```

Listing 7-3: include/linux/rmap.h: struct anon_vma_chain

Examining each field:-

- `vma` – This is the `struct vm_area_struct` (VMA) object which this object is linking to an `anon_vma` object.
- `anon_vma` – The `struct anon_vma` object being linked to a VMA.
- `same_vma` – The `struct list_head` object which is the node connecting this to the `anon_vma_chain` field in the owning VMA object and other `struct anon_vma_chain` objects in the list.
- `rb` – The node of the red/black interval tree, rooted in a related `struct anon_vma`'s `rb_root` field.
- `rb_subtree_last` – Stores the largest exclusive end bound of the interval of elements within this subtree. A field required for the interval tree implementation (see `include/linux/interval_tree_generic.h` and `mm/interval_tree.c` for implementation details).
- `cached_vma_start` – Used by the VM red/black tree debug functionality (set by the `CONFIG_DEBUG_VM_RB` flag) to ensure that virtual page offsets of the *start* of the referenced VMA is correct.
- `cached_vma_last` – Used by the same debug functionality to ensure that virtual page offsets of the *end* of the referenced VMA is correct.

7.0.5 Anonymous reverse mapping initialisation

Now we have examined how reverse mappings function and the types which underlie anonymous reverse mappings, let's consider how they are initialised in the first place.

Each `struct vm_area_struct` (VMA) references the `struct anon_vma` object which newly allocated anonymous folios will be mapped to in its `anon_vma` field. Importantly, this field is set to `NULL` by default and is only populated when anonymous memory is faulted in and an `anon_vma` is required.

The initial setup of the `anon_vma` objects is performed in `anon_vma_prepare()` which is invoked by functions faulting in anonymous pages, i.e. `do_anonymous_page()`, `wp_page_copy()`, `do_cow_fault()` and the stack expansion functions `expand_downwards()` and `expand_upwards()` (it is also invoked by huge page and migration functions but these are out of scope here) as shown in Listing 7-4.

```

154 static inline int anon_vma_prepare(struct vm_area_struct *vma)

```

```

155 {
156     if (likely(vma->anon_vma))
157         return 0;
158
159     return __anon_vma_prepare(vma);
160 }

```

Listing 7-4: include/linux/rmap.h: anon_vma_prepare()

This is invoked unconditionally and only actually performs the allocation and initialisation of an anon_vma object if none already exists, which is done in __anon_vma_prepare(), as shown in Listing 7-5.

```

159 /**
160  * __anon_vma_prepare - attach an anon_vma to a memory region
161  * @vma: the memory region in question
162  *
163  * This makes sure the memory mapping described by 'vma' has
164  * an 'anon_vma' attached to it, so that we can associate the
165  * anonymous pages mapped into it with that anon_vma.
166  *
167  * The common case will be that we already have one, which
168  * is handled inline by anon_vma_prepare(). But if
169  * not we either need to find an adjacent mapping that we
170  * can re-use the anon_vma from (very common when the only
171  * reason for splitting a vma has been mprotect()), or we
172  * allocate a new one.
173  *
174  * Anon-vma allocations are very subtle, because we may have
175  * optimistically looked up an anon_vma in folio_lock_anon_vma_read()
176  * and that may actually touch the rwsem even in the newly
177  * allocated vma (it depends on RCU to make sure that the
178  * anon_vma isn't actually destroyed).
179  *
180  * As a result, we need to do proper anon_vma locking even
181  * for the new allocation. At the same time, we do not want
182  * to do any locking for the common case of already having
183  * an anon_vma.
184  *
185  * This must be called with the mmap_lock held for reading.
186  */
187 int __anon_vma_prepare(struct vm_area_struct *vma)
188 {
189     struct mm_struct *mm = vma->vm_mm;
190     struct anon_vma *anon_vma, *allocated;
191     struct anon_vma_chain *avc;
192
193     might_sleep();
194

```

```

195     avc = anon_vma_chain_alloc(GFP_KERNEL);
196     if (!avc)
197         goto out_enomem;
198
199     anon_vma = find_mergeable_anon_vma(vma);
200     allocated = NULL;
201     if (!anon_vma) {
202         anon_vma = anon_vma_alloc();
203         if (unlikely(!anon_vma))
204             goto out_enomem_free_avc;
205         anon_vma->num_children++; /* self-parent link for new root */
206         allocated = anon_vma;
207     }
208
209     anon_vma_lock_write(anon_vma);
210     /* page table_lock to protect against threads */
211     spin_lock(&mm->page_table_lock);
212     if (likely(!vma->anon_vma)) {
213         vma->anon_vma = anon_vma;
214         anon_vma_chain_link(vma, avc, anon_vma);
215         anon_vma->num_active_vmas++;
216         allocated = NULL;
217         avc = NULL;
218     }
219     spin_unlock(&mm->page_table_lock);
220     anon_vma_unlock_write(anon_vma);
221
222     if (unlikely(allocated))
223         put_anon_vma(allocated);
224     if (unlikely(avc))
225         anon_vma_chain_free(avc);
226
227     return 0;
228
229 out_enomem_free_avc:
230     anon_vma_chain_free(avc);
231 out_enomem:
232     return -ENOMEM;
233 }

```

Listing 7-5: mm/rmap.c: __anon_vma_prepare()

There is a lot to unpack here, so let's start by examining how we actually perform the allocations of the struct `anon_vma` and struct `anon_vma_chain` objects.

We start by allocating the struct `anon_vma_chain` object associated with the candidate `anon_vma`. Note that we are only performing this action because the

VMA does not have a `anon_vma` to point at, so we might not *have* to allocate a new `anon_vma` but instead might be able to reuse an existing one.

However in any case we absolutely *do* need to allocate a new `anon_vma_chain`, as this represents the connection between a VMA and an `anon_vma`. We do this in `anon_vma_chain_alloc()` as shown in Listing 7-6.

```

139 static inline struct anon_vma_chain *anon_vma_chain_alloc(gfp_t gfp)
140 {
141     return kmem_cache_alloc(anon_vma_chain_cachep, gfp);
142 }
```

Listing 7-6: mm/rmap.c: `anon_vma_chain_alloc()`

The actual allocation is performed by the slab function `kmem_cache_alloc()`. We go into detail on this in the slab chapter, however fundamentally this function allows for the efficient allocation of objects of a specific type.

In this case, we template objects using the `anon_vma_chain_cachep` object which is initialised early in `anon_vma_init()`.

We free `anon_vma_chain` objects via `anon_vma_chain_free()` as shown in Listing 7-7.

```

144 static void anon_vma_chain_free(struct anon_vma_chain *anon_vma_chain)
145 {
146     kmem_cache_free(anon_vma_chain_cachep, anon_vma_chain);
147 }
```

Listing 7-7: mm/rmap.c: `anon_vma_chain_free()`

Which simply invokes the standard slab free function `kmem_cache_free()`.

Turning now to the allocation of `anon_vma` objects via `anon_vma_alloc()` as shown in Listing 7-8.

```

89 static inline struct anon_vma *anon_vma_alloc(void)
90 {
91     struct anon_vma *anon_vma;
92
93     anon_vma = kmem_cache_alloc(anon_vma_cachep, GFP_KERNEL);
94     if (anon_vma) {
95         atomic_set(&anon_vma->refcount, 1);
96         anon_vma->num_children = 0;
97         anon_vma->num_active_vmas = 0;
98         anon_vma->parent = anon_vma;
99         /*
100          * Initialise the anon_vma root to point to itself. If called
101          * from fork, the root will be reset to the parents anon_vma.
102          */
103         anon_vma->root = anon_vma;
104     }
105 }
```

```

106         return anon_vma;
107     }

```

Listing 7-8: mm/rmap.c: anon_vma_alloc()

Again, we use `kmem_cache_alloc()` to allocate the actual object, this time using the `anon_vma_cachep` slab cache object. A difference from `anon_vma_chain` allocations here is that this is set up to invoke a constructor on allocation, `anon_vma_ctor()` as shown in Listing 7-9.

```

450 static void anon_vma_ctor(void *data)
451 {
452     struct anon_vma *anon_vma = data;
453
454     init_rwsem(&anon_vma->rwsem);
455     atomic_set(&anon_vma->refcount, 0);
456     anon_vma->rb_root = RB_ROOT_CACHED;
457 }

```

Listing 7-9: mm/rmap.c: anon_vma_ctor()

So taking into account this and `anon_vma_alloc()`, on initialisation a `struct anon_vma` object is initialised such that its fields are initialised such that:-

- `root` – Set to point to itself. This is required as all locking performed on the `rwsem` is performed at the root of the `anon_vma` tree, and thus if the tree consists of only one `anon_vma`, as is the case when first initialised, we must lock ourself. Additionally, it is used to assign all non-exclusive mapped folios to the root of the `anon_vma` tree when swapped back in or migrated (see section 7.0.12 for more on this).
- `rwsem` – Initialised via `init_rwsem()` to an unlocked state.
- `refcount` – When allocated within the cache this is atomically initialised to zero, however when this object is obtained from the cache in `anon_vma_alloc()` it is atomically set to 1 as by definition, allocating it means that we maintain a reference to it.
- `num_children` – We initially set this to zero, but as part of `__anon_vma_prepare()` we increment this to 1. We do this because on preparation we don't know whether we can merge an existing `anon_vma` and thus it is more generic to initially set this to zero and to simply increment on preparation.
- `num_active_vmas` – Equally set to zero on allocation but incremented in `__anon_vma_prepare()` when a VMA is pointed at it.
- `parent` – Similar to `root`, this is set to point at the `anon_vma` object itself. This field is used to update the child count of a forked `anon_vma`'s parent which, since we root a singular `anon_vma` is itself, is logically also itself.
- `rb_root` – This is initialised as any red/black tree in the kernel by setting it to the default NULL state defined by `RB_ROOT_CACHED`.

7.0.6 Reusing adjacent VMA's anon_vma objects

Returning to `__anon_vma_prepare()` (in listing 7-5), we see that we try to merge the use of an existing struct `anon_vma` object with this VMA – as we index by anonymously mapped folios's originally mapped *virtual page offset* (as opposed to file page offset for file mappings), which is in turn used to lookup VMAs in the `rb_root`-rooted interval tree, an `anon_vma` can reference multiple VMAs with ease and thus merging across *adjacent* virtual mappings with the same attributes is highly feasible.

We could in theory merge other compatible `anon_vma` objects (we discuss what constitutes compatible `anon_vma` objects below) due to the virtual page offset, however doing so would be time-consuming, and the fact that `mremap()` retains the *original* virtual page offset if it moves pages makes this more complicated (see section 8.2.3 for an in-depth discussion of `mremap()`).

We therefore do attempt to do so in order to minimise kernel overhead, which is performed in `find_mergeable_anon_vma()` as shown in Listing 7-10.

```

1268 /*
1269  * find_mergeable_anon_vma is used by anon_vma_prepare, to check
1270  * neighbouring vmAs for a suitable anon_vma, before it goes off
1271  * to allocate a new anon_vma. It checks because a repetitive
1272  * sequence of mprotects and faults may otherwise lead to distinct
1273  * anon_vmas being allocated, preventing vma merge in subsequent
1274  * mprotect.
1275  */
1276 struct anon_vma *find_mergeable_anon_vma(struct vm_area_struct *vma)
1277 {
1278     struct anon_vma *anon_vma = NULL;
1279
1280     /* Try next first. */
1281     if (vma->vm_next) {
1282         anon_vma = reusable_anon_vma(vma->vm_next, vma, vma->vm_next);
1283         if (anon_vma)
1284             return anon_vma;
1285     }
1286
1287     /* Try prev next. */
1288     if (vma->vm_prev)
1289         anon_vma = reusable_anon_vma(vma->vm_prev, vma->vm_prev, vma);
1290
1291     /*
1292      * We might reach here with anon_vma == NULL if we can't find
1293      * any reusable anon_vma.
1294      * There's no absolute need to look only at touching neighbours:
1295      * we could search further afield for "compatible" anon_vmas.
1296      * But it would probably just be a waste of time searching,
1297      * or lead to too many vmAs hanging off the same anon_vma.
1298      * We're trying to allow mprotect remerging later on,

```

```

1299         * not trying to minimize memory used for anon_vmas.
1300         */
1301         return anon_vma;
1302     }

```

Listing 7-10: mm/mmap.c: *find_mergeable_anon_vma()*

This simply attempts to determine whether either the previous or next VMA has a compatible VMA. If so, it returns its struct `anon_vma` object to be used instead of a newly allocated one.

We test what constitutes an `anon_vma`-mergeable VMA via `reusable_anon_vma()` as shown in Listing 7-11.

```

1235  /*
1236   * Do some basic sanity checking to see if we can re-use the anon_vma
1237   * from 'old'. The 'a'/'b' vma's are in VM order - one of them will be
1238   * the same as 'old', the other will be the new one that is trying
1239   * to share the anon_vma.
1240   *
1241   * NOTE! This runs with mmap_lock held for reading, so it is possible that
1242   * the anon_vma of 'old' is concurrently in the process of being set up
1243   * by another page fault trying to merge _that_. But that's ok: if it
1244   * is being set up, that automatically means that it will be a singleton
1245   * acceptable for merging, so we can do all of this optimistically. But
1246   * we do that READ_ONCE() to make sure that we never re-load the pointer.
1247   *
1248   * IOW: that the "list_is_singular()" test on the anon_vma_chain only
1249   * matters for the 'stable anon_vma' case (ie the thing we want to avoid
1250   * is to return an anon_vma that is "complex" due to having gone through
1251   * a fork).
1252   *
1253   * We also make sure that the two vma's are compatible (adjacent,
1254   * and with the same memory policies). That's all stable, even with just
1255   * a read lock on the mmap_lock.
1256   */
1257  static struct anon_vma *reusable_anon_vma(struct vm_area_struct *old, struct
        vm_area_struct *a, struct vm_area_struct *b)
1258  {
1259      if (anon_vma_compatible(a, b)) {
1260          struct anon_vma *anon_vma = READ_ONCE(old->anon_vma);
1261
1262          if (anon_vma && list_is_singular(&old->anon_vma_chain))
1263              return anon_vma;
1264      }
1265      return NULL;
1266  }

```

Listing 7-11: mm/mmap.c: *reusable_anon_vma()*

The fundamental check is performed via `anon_vma_compatible()` which we examine below, however note that *also* require the candidate `anon_vma` to possess a single entry in its struct `vm_area_struct->anon_vma_chain` list, i.e. to not be a VMA which is derived from a fork. For simplicity, we will aggregate this condition with those imposed by `anon_vma_compatible()` as shown in Listing 7-12.

```

1213 /*
1214  * Rough compatibility check to quickly see if it's even worth looking
1215  * at sharing an anon_vma.
1216  *
1217  * They need to have the same vm_file, and the flags can only differ
1218  * in things that mprotect may change.
1219  *
1220  * NOTE! The fact that we share an anon_vma doesn't _have_ to mean that
1221  * we can merge the two vma's. For example, we refuse to merge a vma if
1222  * there is a vm_ops->close() function, because that indicates that the
1223  * driver is doing some kind of reference counting. But that doesn't
1224  * really matter for the anon_vma sharing case.
1225  */
1226 static int anon_vma_compatible(struct vm_area_struct *a, struct vm_area_struct
                                *b)
1227 {
1228     return a->vm_end == b->vm_start &&
1229            mpol_equal(vma_policy(a), vma_policy(b)) &&
1230            a->vm_file == b->vm_file &&
1231            !((a->vm_flags ^ b->vm_flags) & ~(VM_ACCESS_FLAGS |
VM_SOFTDIRTY)) &&
1232            b->vm_pgoff == a->vm_pgoff + ((b->vm_start - a->vm_start) >>
PAGE_SHIFT);
1233 }

```

Listing 7-12: `mm/mmap.c: anon_vma_compatible()`

In aggregate, `reusable_anon_vma()` and `anon_vma_compatible()` impose the following conditions in order for two VMAs to be considered struct `anon_vma`-mergeable are as follows (designating the preceding VMA A and the following VMA B:-

1. The end of A's virtual mapping must be exactly equal to the beginning of B's, i.e. the VMAs must be adjacent.
2. The NUMA memory policy of both VMAs must be equivalent. This is checked by `mpol_equal()`. This is out of scope for this chapter, but see the NUMA chapter for more on memory policies.
3. Both A and B must have equivalent `vm_file` – either both are anonymous, or both are `MAP_PRIVATE` mappings mapping the same file (a simple file mapping would not be applicable here as it would not possess an `anon_vma` object).

4. Both A and B's VMA flags must be identical, except for the basic access flags `VM_ACCESS_FLAGS` which comprise `VM_READ`, `VM_WRITE`, and `VM_EXEC`. From the point of view of anonymous folio reverse mapping, the access rights established by the page their table mappings and the fact that distinct access rights require distinct VMAs are irrelevant. Additionally the `VM_SOFTDIRTY` flag is ignored as it is a purely informative flag and does not denote an actual difference in VMA behaviour.
5. The `vm_pgoff` of the latter VMA must be equal to the `vm_pgoff` of the preceding VMA offset by the difference in their `vm_start` fields, in other words, if a file is mapped, the offset of the latter VMA within it must be precisely where the former VMA left off. If the VMAs are anonymous then this is equivalent to them being immediately adjacent.
6. The adjacent VMA which already maps to the `anon_vma` must not have an `anon_vma_chain` list larger than one, i.e. it must not be part of a forked child process which has inherited this mapping from its parent.

Looking more closely at this final requirement – this was introduced in commit

`doe9fe1758f2`: Simplify and comment on `anon_vma` re-use for `anon_vma_prepare` by Linus to simplify VMA merging when `struct anon_vma` objects are mapped.

This establishes the invariant that if two VMAs specify the same `anon_vma` object in their `anon_vma` fields, they will have identical `struct anon_vma_chains` in the sense that they will have a *single* AVC that links the same `anon_vma` to themselves.

Since the kernel already establishes the criteria that VMAs with associated `anon_vma` objects may only be merged if each VMA points to same one (via `is_mergeable_anon_vma()`), adding the additional criteria that we cannot reuse an `anon_vma` unless the VMA that points to it has no mappings to parent `anon_vma` objects makes merging easy – we simply unlink the VMA being removed from its `anon_vma` and we're done (in `anon_vma_merge()`).

If we didn't have this criteria, we would have to carefully check and move any mappings to parent `anon_vma` objects and be very careful about locking throughout.

Note that there is one other place from which we can reuse `anon_vma` objects – on cloning an existing VMA's `anon_vma` structure in `anon_vma_clone()`. We discuss this below.

7.0.7 Connecting `anon_vma` objects

We connect a `struct anon_vma` object to a `struct vm_area_struct` (VMA) object via `anon_vma_chain_link()`:-

```

149 static void anon_vma_chain_link(struct vm_area_struct *vma,
150                                struct anon_vma_chain *avc,
151                                struct anon_vma *anon_vma)
152 {
153     avc->vma = vma;
```

```

154         avc->anon_vma = anon_vma;
155         list_add(&avc->same_vma, &vma->anon_vma_chain);
156         anon_vma_interval_tree_insert(avc, &anon_vma->rb_root);
157     }

```

Listing 7-13: mm/rmap.c: anon_vma_chain_link()

This is a convenience function which assigns the AVC's `vma` and `anon_vma` fields to the `anon_vma` and VMA it links, adding the AVC to the front of the VMA's `anon_vma_chain` list via `list_add()` and the `anon_vma`'s interval tree rooted in `rb_root` via `anon_vma_interval_tree_insert()`.

Note that `__anon_vma_prepare()` invokes this to connect the newly allocated `anon_vma` and `struct anon_vma_chain`, in addition to a number of other callers.

7.0.8 Cloning anon_vma objects

It might seem surprising, but the fundamental function for splitting and forking `struct anon_vma` structures is *cloning* an existing VMA's connections to `anon_vma` objects.

However, the more you think about it, the more it makes sense – if there is more than one connection to `anon_vma` objects, that implies the VMA has been forked, and thus all existing such connections must be maintained no matter what.

On splitting a VMA, we share the existing `anon_vma` so that must be maintained for the newly split VMA (see figure 7-3), and on fork we of course import the connections to the parent to our child VMA before adding our own new `anon_vma` for CoW'd pages.

This clone operation is performed in `anon_vma_clone()` as shown in Listing 7-14.

```

261  /*
262   * Attach the anon_vmas from src to dst.
263   * Returns 0 on success, -ENOMEM on failure.
264   *
265   * anon_vma_clone() is called by __vma_adjust(), __split_vma(), copy_vma() and
266   * anon_vma_fork(). The first three want an exact copy of src, while the last
267   * one, anon_vma_fork(), may try to reuse an existing anon_vma to prevent
268   * endless growth of anon_vma. Since dst->anon_vma is set to NULL before call,
269   * we can identify this case by checking (!dst->anon_vma && src->anon_vma).
270   *
271   * If (!dst->anon_vma && src->anon_vma) is true, this function tries to find
272   * and reuse existing anon_vma which has no vmas and only one child anon_vma.
273   * This prevents degradation of anon_vma hierarchy to endless linear chain in
274   * case of constantly forking task. On the other hand, an anon_vma with more
275   * than one child isn't reused even if there was no alive vma, thus rmap
276   * walker has a good chance of avoiding scanning the whole hierarchy when it
277   * searches where page is mapped.
278   */
279  int anon_vma_clone(struct vm_area_struct *dst, struct vm_area_struct *src)

```

```

280 {
281     struct anon_vma_chain *avc, *pavc;
282     struct anon_vma *root = NULL;
283
284     list_for_each_entry_reverse(pavc, &src->anon_vma_chain, same_vma) {
285         struct anon_vma *anon_vma;
286
287         avc = anon_vma_chain_alloc(GFP_NOWAIT | __GFP_NOWARN);
288         if (unlikely(!avc)) {
289             unlock_anon_vma_root(root);
290             root = NULL;
291             avc = anon_vma_chain_alloc(GFP_KERNEL);
292             if (!avc)
293                 goto enomem_failure;
294         }
295         anon_vma = pavc->anon_vma;
296         root = lock_anon_vma_root(root, anon_vma);
297         anon_vma_chain_link(dst, avc, anon_vma);
298
299         /*
300          * Reuse existing anon_vma if it has no vma and only one
301          * anon_vma child.
302          *
303          * Root anon_vma is never reused:
304          * it has self-parent reference and at least one child.
305          */
306         if (!dst->anon_vma && src->anon_vma &&
307             anon_vma->num_children < 2 &&
308             anon_vma->num_active_vmas == 0)
309             dst->anon_vma = anon_vma;
310     }
311     if (dst->anon_vma)
312         dst->anon_vma->num_active_vmas++;
313     unlock_anon_vma_root(root);
314     return 0;
315
316 enomem_failure:
317     /*
318      * dst->anon_vma is dropped here otherwise its degree can be
319      * decremented in unlink_anon_vmas().
320      * We can safely do this because callers of anon_vma_clone() don't
321      * care
322      * about dst->anon_vma if anon_vma_clone() failed.
323      */
324     dst->anon_vma = NULL;
325     unlink_anon_vmas(dst);

```

```

325         return -ENOMEM;
326     }

```

Listing 7-14: mm/rmap.c: *anon_vma_clone()*

This takes an existing struct *vm_area_struct* (VMA)’s links to struct *anon_vma* objects makes a copy of them. It achieves this by iterating in reverse order through the source VMA’s *anon_vma_chain* list*, allocating a new AVC, linking them to each of the source VMA’s linked *anon_vma* objects and the target VMA via *anon_vma_chain_link()*.

Over the course of the iteration we acquire the appropriate root struct *anon_vma* lock via *lock_anon_vma_root()*, a convenience function which unlocks and relocks if the linked *anon_vma* sits in a different tree.

If this function is called when forking, then the destination *anon_vma*’s *anon_vma* field will be NULL, which is used to perform fork-specific and non-fork specific logic.

If the clone operation is *not* caused by a fault (i.e. the destination VMA already has an struct *anon_vma* object assigned), its *num_active_vmas* field is incremented as cloning an existing VMA’s *anon_vma* structure indicates that we are assigning a new VMA to the destination’s *anon_vma*.

anon_vma_clone() is used by *__vma_adjust()* – a function invoked when adjusting VMAs including merging them, *__split_vma()* – a function invoked when splitting VMAs, *copy_vma()* – a function invoked when performing a *mremap()* operation and finally *anon_vma_fork()* when a VMA is in the process of being forked.

7.0.9 Reusing anon_vma objects on fork

Similar to *__anon_vma_prepare()* discussed in section 7.0.6, *anon_vma_clone()* attempts to reuse existing *anon_vma* objects where possible when a fork is performed.

We first determine that this is a fork operation by checking that the destination VMA has no *anon_vma* set, then checks the following criteria:-

1. A source *anon_vma* is present.
2. It has one or no children (i.e. it is not the root of an *anon_vma* tree which has forked already)
3. Vitally – it has *num_active_vmas* set to zero – i.e. it is a zombie object with no VMAs pointing at it whatsoever.

Under what circumstances does this occur? If a process repeatedly forks with each child process quickly exiting (and thus each invoking *unlink_anon_vmas()*), they leave behind empty *anon_vma* objects simply pointing to child AVC objects resulting in an otherwise unnecessary growth in *anon_vma* objects.

*. In reverse order because newer struct *anon_vma_chain* objects are appended to the *front* of this list rather than the rear, and when we add these to the target VMA’s list we are appending to the front so in order to maintain the same order we must iterate backwards.

7.0.10 Forking anon_vma objects

Forking is one of the key operations in any unix system, where a process is cloned from a ‘parent’ process into a ‘child’ one which maps the parent’s memories only utilising Copy-on-Write semantics to do so efficiently.

When forking a process, `dup_mmap()` is invoked to perform the duplication of all existing memory mappings. This in turn invokes `anon_vma_fork()` to perform the forking with respect to `anon_vma` objects as shown in Listing 7-15.

```

328 /*
329  * Attach vma to its own anon_vma, as well as to the anon_vmas that
330  * the corresponding VMA in the parent process is attached to.
331  * Returns 0 on success, non-zero on failure.
332  */
333 int anon_vma_fork(struct vm_area_struct *vma, struct vm_area_struct *pvma)
334 {
335     struct anon_vma_chain *avc;
336     struct anon_vma *anon_vma;
337     int error;
338
339     /* Don't bother if the parent process has no anon_vma here. */
340     if (!pvma->anon_vma)
341         return 0;
342
343     /* Drop inherited anon_vma, we'll reuse existing or allocate new. */
344     vma->anon_vma = NULL;
345
346     /*
347      * First, attach the new VMA to the parent VMA's anon_vmas,
348      * so rmap can find non-COWed pages in child processes.
349      */
350     error = anon_vma_clone(vma, pvma);
351     if (error)
352         return error;
353
354     /* An existing anon_vma has been reused, all done then. */
355     if (vma->anon_vma)
356         return 0;
357
358     /* Then add our own anon_vma. */
359     anon_vma = anon_vma_alloc();
360     if (!anon_vma)
361         goto out_error;
362     anon_vma->num_active_vmas++;
363     avc = anon_vma_chain_alloc(GFP_KERNEL);
364     if (!avc)
365         goto out_error_free_anon_vma;
366

```



```

367      /*
368       * The root anon_vma's rwsem is the lock actually used when we
369       * lock any of the anon_vmas in this anon_vma tree.
370       */
371      anon_vma->root = pvma->anon_vma->root;
372      anon_vma->parent = pvma->anon_vma;
373      /*
374       * With refcounts, an anon_vma can stay around longer than the
375       * process it belongs to. The root anon_vma needs to be pinned until
376       * this anon_vma is freed, because the lock lives in the root.
377       */
378      get_anon_vma(anon_vma->root);
379      /* Mark this anon_vma as the one where our new (COWed) pages go. */
380      vma->anon_vma = anon_vma;
381      anon_vma_lock_write(anon_vma);
382      anon_vma_chain_link(vma, avc, anon_vma);
383      anon_vma->parent->num_children++;
384      anon_vma_unlock_write(anon_vma);
385
386      return 0;
387
388  out_error_free_anon_vma:
389      put_anon_vma(anon_vma);
390  out_error:
391      unlink_anon_vmas(vma);
392      return -ENOMEM;
393  }

```

Listing 7-15: mm/rmap.c: *anon_vma_fork()*

This firstly checks whether the existing VMA is even pointed at a struct `anon_vma` object – if not then nothing need be done.

Otherwise, the function:-

1. Invokes the above described `anon_vma_clone()` function to clone links to its parent's `anon_vma` objects.
2. Checks whether it reused an `anon_vma` – if so, the function returns.
3. If not, allocates a new one via `anon_vma_alloc()` and a struct `anon_vma_chain` object via `anon_vma_chain_alloc()`.
4. Determines the new `anon_vma`'s root from the parent's `anon_vma`'s root and sets its parent, then acquires an additional reference on the root `anon_vma`.
5. Points this VMA at the newly allocated `anon_vma` as the new CoW target and links everything together via `anon_vma_chain_link()`.

When a page is un-CoW'd in `do_wp_page()` it either creates a new `anon_vma` mapping via `page_add_new_anon_rmap()` or, if it is now exclusively owned by a single process, it moves the page from the parent `anon_vma` via

`page_move_anon_rmap()`. We examine this and other folio reverse mapping manipulation in section 7.0.12 below.

Note that any existing shared folios which are either swapped back in or migrated via `page_add_anon_rmap()` will be pointed at the *root* `anon_vma`. Again see section 7.0.12 below for more.

Important – The fact that an `anon_vma` object maps to a forked child’s VMA does not necessarily mean that VMA maps it. The child may have since CoW’d the folio, and in reality at any time a mapping might disappear from under us.

The purpose of the reverse mapping is only to find *candidate* VMAs where a mapping might exist, we must check the mapping is actually present when walking the reverse mapping – we discuss precisely how we do this in section 7.0.16.

7.0.11 VMA split and merge

There are two functions in which an `anon_vma` clone occurs – `__vma_adjust()` which is the core function invoked for modifying VMAs, and `__split_vma()` which is invoked when VMAs are split (and which subsequently calls `__vma_adjust()` to account for changes in VMA structure).

The former clones only if, for example, `mprotect()` changes VMA boundaries due to changing protection attributes of portions of a VMA (e.g. one region read/write, another read-only) and one portion needs to import the `anon_vma` objects of the other.

The latter clones in all cases as of course it is creating a new `struct vm_area_struct` (VMA) which must import the existing `anon_vma` hierarchy.

However, what needs to be clarified is – when such changes are made, what precisely updates the `rb_root` interval tree entries to point to the correct AVC objects after the change is made?

The answer is that this is performed in in `__vma_adjust()` which, prior to performing adjustments to the VMA’s `vm_start`, `vm_end` and `vm_pgoff` fields (all of which are used by the interval tree to determine which VMA to map to), invokes `anon_vma_interval_tree_pre_update_vma()` as shown in Listing 7-16.

```

456 /*
457  * vma has some anon_vma assigned, and is already inserted on that
458  * anon_vma's interval trees.
459  *
460  * Before updating the vma's vm_start / vm_end / vm_pgoff fields, the
461  * vma must be removed from the anon_vma's interval trees using
462  * anon_vma_interval_tree_pre_update_vma().
463  *
464  * After the update, the vma will be reinserted using
465  * anon_vma_interval_tree_post_update_vma().
466  *
467  * The entire update must be protected by exclusive mmap_lock and by
468  * the root anon_vma's mutex.
```

```

469 */
470 static inline void
471 anon_vma_interval_tree_pre_update_vma(struct vm_area_struct *vma)
472 {
473     struct anon_vma_chain *avc;
474
475     list_for_each_entry(avc, &vma->anon_vma_chain, same_vma)
476         anon_vma_interval_tree_remove(avc, &avc->anon_vma->rb_root);
477 }

```

Listing 7-16: mm/mmap.c: anon_vma_interval_tree_pre_update_vma()

This simply rips through all struct anon_vma objects linked to the VMA and removes all references to the VMA via anon_vma_interval_tree_remove(). Once the adjustment is made, anon_vma_interval_tree_post_update_vma() is called to restore these to the now corrected VMA intervals as shown in Listing 7-17.

```

479 static inline void
480 anon_vma_interval_tree_post_update_vma(struct vm_area_struct *vma)
481 {
482     struct anon_vma_chain *avc;
483
484     list_for_each_entry(avc, &vma->anon_vma_chain, same_vma)
485         anon_vma_interval_tree_insert(avc, &avc->anon_vma->rb_root);
486 }

```

Listing 7-17: mm/mmap.c: anon_vma_interval_tree_post_update_vma()

Another task performed in __vma_adjust(), when merging two VMAs which are permitted to be merged (more on this shortly), the actual operation is performed via anon_vma_merge() which, as described in section 7.0.6, is able to perform this action by simply removing the absorbed VMA's AVC state as shown in Listing 7-18.

```

162 static inline void anon_vma_merge(struct vm_area_struct *vma,
163                                 struct vm_area_struct *next)
164 {
165     VM_BUG_ON_VMA(vma->anon_vma != next->anon_vma, vma);
166     unlink_anon_vmas(next);
167 }

```

Listing 7-18: include/linux/rmap.h: anon_vma_merge()

The removal of the absorbed VMA's struct anon_vma objects is performed by unlink_anon_vmas() which we discuss below.

The actual merge itself is performed by vma_merge() which ultimately determines whether anon_vma objects can be merged via is_mergeable_anon_vma() as shown in Listing 7-19.

```

1015 static inline int is_mergeable_anon_vma(struct anon_vma *anon_vma1,

```

```

1016                                     struct anon_vma *anon_vma2,
1017                                     struct vm_area_struct *vma)
1018 {
1019     /*
1020      * The list_is_singular() test is to avoid merging VMA cloned from
1021      * parents. This can improve scalability caused by anon_vma lock.
1022      */
1023     if ((!anon_vma1 || !anon_vma2) && (!vma ||
1024         list_is_singular(&vma->anon_vma_chain)))
1025         return 1;
1026     return anon_vma1 == anon_vma2;
1027 }

```

Listing 7-19: mm/mmap.c: is_mergeable_anon_vma()

If both VMAs possess anon_vma objects then the test is simple – they must both assign newly allocated folios to the same anon_vma (i.e. each VMA’s anon_vma field must point to the same anon_vma).

If either lacks an anon_vma, then the test reduces to ensuring that the VMA is not the product of a fork, as pointing a non-forked VMA at a forked anon_vma would be misleading.

Note that the check on whether vma is NULL is used specifically by vma_merge() when checking whether preceding and succeeding VMAs are anon_vma-mergeable in which case this check is not relevant.

7.0.12 Folio anon_vma operations

The principle means by which a folio is linked to a struct anon_vma is via page_add_new_anon_rmap(). This is invoked when the folio is newly allocated and thus less care need be taken with existing folio state:-

```

1249 /**
1250  * page_add_new_anon_rmap - add mapping to a new anonymous page
1251  * @page:         the page to add the mapping to
1252  * @vma:          the vm area in which the mapping is added
1253  * @address:      the user virtual address mapped
1254  *
1255  * If it's a compound page, it is accounted as a compound page. As the page
1256  * is new, it's assume to get mapped exclusively by a single process.
1257  *
1258  * Same as page_add_anon_rmap but must only be called on *new* pages.
1259  * This means the inc-and-test can be bypassed.
1260  * Page does not have to be locked.
1261  */
1262 void page_add_new_anon_rmap(struct page *page,
1263     struct vm_area_struct *vma, unsigned long address)
1264 {
1265     const bool compound = PageCompound(page);

```

```

1266         int nr = compound ? thp_nr_pages(page) : 1;
1267
1268         VM_BUG_ON_VMA(address < vma->vm_start || address >= vma->vm_end, vma);
1269         __SetPageSwapBacked(page);
1270         if (compound) {
1271             VM_BUG_ON_PAGE(!PageTransHuge(page), page);
1272             /* increment count (starts at -1) */
1273             atomic_set(compound_mapcount_ptr(page), 0);
1274             atomic_set(compound_pincount_ptr(page), 0);
1275
1276             __mod_lruvec_page_state(page, NR_ANON_THPS, nr);
1277         } else {
1278             /* increment count (starts at -1) */
1279             atomic_set(&page->_mapcount, 0);
1280         }
1281         __mod_lruvec_page_state(page, NR_ANON_MAPPED, nr);
1282         __page_set_anon_rmap(page, vma, address, 1);
1283     }

```

Listing 7-20: mm/rmap.c: *page_add_new_anon_rmap()*

This sets the `PG_swapbacked` folio flag as this folio is now established as anonymous, initialises the critical `_mapcount` field (this is initialised to -1 on allocation, and thus setting to zero is an increment, indicating that the folio is now mapped), updates statistics and handles the Transparent Huge Page (THP) case of a compound folio (see the huge page chapter for more on this).

Finally the actual assignment to the struct `anon_vma` is performed in `__page_set_anon_rmap()` which we shall examine shortly.

This function is invoked by all the usual fault handling logic – `do_anonymous_page()` when new anonymous pages are faulted in, `do_swap_page()` when new swap pages are faulted in, `wp_page_copy()` when a page has to be copied for a CoW mapping, `copy_present_page()` after a fork (and after `anon_vma_fork()` has been performed) and `do_set_pte()` after a `MAP_PRIVATE` file-backed mapping has been CoW'd.

The function is also reference dbt migrate, huge page, uprobe, swap file and `userfaultfd()` logic but discussion of this is outside the scope of this section.

If we are adding a new mapping to a folio that already possesses mapping, we have to be a little more careful as shown in Listing 7-21.

```

1188 /**
1189  * page_add_anon_rmap - add pte mapping to an anonymous page
1190  * @page:         the page to add the mapping to
1191  * @vma:          the vm area in which the mapping is added
1192  * @address:       the user virtual address mapped
1193  * @flags:         the rmap flags
1194  *
1195  * The caller needs to hold the pte lock, and the page must be locked in

```

```

1196  * the anon_vma case: to serialize mapping, index checking after setting,
1197  * and to ensure that PageAnon is not being upgraded racily to PageKsm
1198  * (but PageKsm is never downgraded to PageAnon).
1199  */
1200  void page_add_anon_rmap(struct page *page,
1201                        struct vm_area_struct *vma, unsigned long address, rmap_t flags)
1202  {
1203      bool compound = flags & RMAP_COMPOUND;
1204      bool first;
1205
1206      if (unlikely(PageKsm(page)))
1207          lock_page_memcg(page);
1208      else
1209          VM_BUG_ON_PAGE(!PageLocked(page), page);
1210
1211      if (compound) {
1212          atomic_t *mapcount;
1213          VM_BUG_ON_PAGE(!PageLocked(page), page);
1214          VM_BUG_ON_PAGE(!PageTransHuge(page), page);
1215          mapcount = compound_mapcount_ptr(page);
1216          first = atomic_inc_and_test(mapcount);
1217      } else {
1218          first = atomic_inc_and_test(&page->_mapcount);
1219      }
1220      VM_BUG_ON_PAGE(!first && (flags & RMAP_EXCLUSIVE), page);
1221      VM_BUG_ON_PAGE(!first && PageAnonExclusive(page), page);
1222
1223      if (first) {
1224          int nr = compound ? thp_nr_pages(page) : 1;
1225          /*
1226           * We use the irq-unsafe __{inc/mod}_zone_page_stat because
1227           * these counters are not modified in interrupt context, and
1228           * pte lock(a spinlock) is held, which implies preemption
1229           * disabled.
1230           */
1231          if (compound)
1232              __mod_lruvec_page_state(page, NR_ANON_THPS, nr);
1233          __mod_lruvec_page_state(page, NR_ANON_MAPPED, nr);
1234      }
1235
1236      if (unlikely(PageKsm(page)))
1237          unlock_page_memcg(page);
1238
1239      /* address might be in next vma when migration races vma_adjust */
1240      else if (first)
1241          __page_set_anon_rmap(page, vma, address,
1242                              !(flags & RMAP_EXCLUSIVE));

```

```

1243     else
1244         __page_check_anon_rmap(page, vma, address);
1245
1246     mlock_vma_page(page, vma, compound);
1247 }

```

Listing 7-21: mm/rmap.c: page_add_anon_rmap()

This is broadly similar to `page_add_new_anon_rmap()` with some additional checks, Kernel Same-page Merging (KSM) handling (out of scope for the book), incrementing of the map count rather than setting it outright and performing statistical updates only if this is the first mapping (as determined by `atomic_inc_and_test()`, which returns true if the incremented value is zero).

An additional difference is the use of the `rmap_t` for passing flags specifying characteristics of the mapping:-

- `RMAP_NONE` – No special treatment, if the struct page pointer `page` is in fact a subpage of a compound folio, then it is mapped at the PTE level rather than a huge page.
- `RMAP_EXCLUSIVE` – Indicates that this is an exclusive mapping of the folio to a single process.
- `RMAP_COMPOUND` – Indicates that the folio is mapped as a huge page.

We also invoke `mlock_vma_page()` which sets the folios as `mlock()`'d if the `VM_LOCKED` flag is set in the VMA (typically via a user specifying the `MCL_FUTURE` flag). See section 8.2.1 for more on this.

`page_add_anon_rmap()` is invoked by KSM and huge memory logic (the former out of scope for the book, the latter discussed exclusively in the huge page chapter), as well as in `restore_exclusive_pte()` (with `RMAP_NONE` specified) on restoring an exclusive PTE entry when forking, on swapping in via `do_swap_page()`, on migration in `remove_migration_pte()` and when making PTEs unused when swapped out in `unuse_pte()`.

For both `page_add_anon_rmap()` and `page_add_new_anon_rmap()`, the actual mapping of the folio is performed in `__page_set_anon_rmap()` as shown in Listing 7-22.

```

1122 /**
1123  * __page_set_anon_rmap - set up new anonymous rmap
1124  * @page:      Page or Hugepage to add to rmap
1125  * @vma:       VM area to add page to.
1126  * @address:    User virtual address of the mapping
1127  * @exclusive:  the page is exclusively owned by the current process
1128  */
1129 static void __page_set_anon_rmap(struct page *page,
1130     struct vm_area_struct *vma, unsigned long address, int exclusive)
1131 {
1132     struct anon_vma *anon_vma = vma->anon_vma;
1133
1134     BUG_ON(!anon_vma);

```

```

1135
1136         if (PageAnon(page))
1137             goto out;
1138
1139         /*
1140          * If the page isn't exclusively mapped into this vma,
1141          * we must use the _oldest_ possible anon_vma for the
1142          * page mapping!
1143          */
1144         if (!exclusive)
1145             anon_vma = anon_vma->root;
1146
1147         /*
1148          * page_idle does a lockless/optimistic rmap scan on page->mapping.
1149          * Make sure the compiler doesn't split the stores of anon_vma and
1150          * the PAGE_MAPPING_ANON type identifier, otherwise the rmap code
1151          * could mistake the mapping for a struct address_space and crash.
1152          */
1153         anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;
1154         WRITE_ONCE(page->mapping, (struct address_space *) anon_vma);
1155         page->index = linear_page_index(vma, address);
1156     out:
1157         if (exclusive)
1158             SetPageAnonExclusive(page);
1159     }

```

Listing 7-22: mm/rmap.c: `__page_set_anon_rmap()`

The logic of this function is as follows:-

1. Check whether the folio has already been marked anonymous via `PageAnon()` which is a simple wrapper around `folio_test_anon()` which simply checks whether the mapping field has `PAGE_MAPPING_ANON` set (more on this below).
2. *Important* – If the page is not exclusively mapped via `RMAP_EXCLUSIVE`, it will be mapped to the *root* struct `anon_vma` rather than the one the struct `vm_area_struct` (VMA) points at. This makes sense for forked processes (the ones for which *root* will point at anything other than themselves) as these will be Copy-on-Write folios only moved to a dedicated `anon_vma` once the CoW occurs. This is typically used when swapping in via `do_swap_page()` and on migration in `remove_migration_pte()`.
3. The folio's mapping field is set to the `anon_vma` object offset by `PAGE_MAPPING_ANON` which is used to indicate that the mapping is not to a struct `address_space` but rather an `anon_vma` object.
4. The index of the folio is set to the virtual page offset of the folio within the VMA via `linear_page_index()`.
5. If the mapping is exclusive, the `PG_anon_exclusive` folio flag is set.

Examining `linear_page_index()` as shown in Listing 7-23.

```

845 static inline pgoff_t linear_page_index(struct vm_area_struct *vma,
846                                         unsigned long address)
847 {
848     pgoff_t pgoff;
849     if (unlikely(is_vm_hugetlb_page(vma)))
850         return linear_hugepage_index(vma, address);
851     pgoff = (address - vma->vm_start) >> PAGE_SHIFT;
852     pgoff += vma->vm_pgoff;
853     return pgoff;
854 }
```

Listing 7-23: `include/linux/pagemap.h: linear_page_index()`

Putting aside the out of scope huge page edge case, this determines the page offset of the specified address within the VMA and adds this to the VMA's `vm_pgoff` field. For anonymous mappings this field is set to the virtual page offset of the mapping.

Note that there is a special case handled very carefully here – `mremap()` can move VMAs around, but to avoid having to update the index field of the associated folios, the `vm_pgoff` field is kept the same. This explains why we find the page offset of the address relative to the start of the VMA then offset by `vm_pgoff` rather than simply shifting the address by the page size. See section 8.2.3 for an in depth description of `mremap()`.

If a folio is CoW'd but is the sole remaining instance, instead of adding it to an `anon_vma` we move it via `page_move_anon_rmap()` as shown in Listing 7-24.

```

1092 /**
1093  * page_move_anon_rmap - move a page to our anon_vma
1094  * @page:         the page to move to our anon_vma
1095  * @vma:          the vma the page belongs to
1096  *
1097  * When a page belongs exclusively to one process after a COW event,
1098  * that page can be moved into the anon_vma that belongs to just that
1099  * process, so the rmap code will not search the parent or sibling
1100  * processes.
1101  */
1102 void page_move_anon_rmap(struct page *page, struct vm_area_struct *vma)
1103 {
1104     struct anon_vma *anon_vma = vma->anon_vma;
1105     struct page *subpage = page;
1106
1107     page = compound_head(page);
1108
1109     VM_BUG_ON_PAGE(!PageLocked(page), page);
1110     VM_BUG_ON_VMA(!anon_vma, vma);
1111
1112     anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;
```

```

1113      /*
1114       * Ensure that anon_vma and the PAGE_MAPPING_ANON bit are written
1115       * simultaneously, so a concurrent reader (eg folio_referenced())'s
1116       * folio_test_anon()) will not see one without the other.
1117       */
1118      WRITE_ONCE(page->mapping, (struct address_space *) anon_vma);
1119      SetPageAnonExclusive(subpage);
1120  }

```

Listing 7-24: mm/rmap.c: `page_move_anon_rmap()`

This simply sets the mapping and the PG_anon_exclusive folio flag.

Removing the anon_vma mapping from a folio is performed by `page_remove_rmap()` as shown in Listing 7-73.

```

1421  /**
1422   * page_remove_rmap - take down pte mapping from a page
1423   * @page:           page to remove mapping from
1424   * @vma:            the vm area from which the mapping is removed
1425   * @compound:       uncharge the page as compound or small page
1426   *
1427   * The caller needs to hold the pte lock.
1428   */
1429  void page_remove_rmap(struct page *page,
1430                       struct vm_area_struct *vma, bool compound)
1431  {
1432      lock_page_memcg(page);
1433
1434      if (!PageAnon(page)) {
1435          page_remove_file_rmap(page, compound);
1436          goto out;
1437      }
1438
1439      if (compound) {
1440          page_remove_anon_compound_rmap(page);
1441          goto out;
1442      }
1443
1444      /* page still mapped by someone else? */
1445      if (!atomic_add_negative(-1, &page->mapcount))
1446          goto out;
1447
1448      /*
1449       * We use the irq-unsafe __{inc/mod}_zone_page_stat because
1450       * these counters are not modified in interrupt context, and
1451       * pte lock(a spinlock) is held, which implies preemption disabled.
1452       */
1453      __dec_lruvec_page_state(page, NR_ANON_MAPPED);
1454

```

```

1455         if (PageTransCompound(page))
1456             deferred_split_huge_page(compound_head(page));
1457
1458         /*
1459          * It would be tidy to reset the PageAnon mapping here,
1460          * but that might overwrite a racing page_add_anon_rmap
1461          * which increments mapcount after us but sets mapping
1462          * before us: so leave the reset to free_unref_page,
1463          * and remember that it's only reliable while mapped.
1464          * Leaving it set also helps swapoff to reinstate ptes
1465          * faster for those pages still in swapcache.
1466          */
1467     out:
1468         unlock_page_memcg(page);
1469
1470         munlock_vma_page(page, vma, compound);
1471     }

```

Listing 7-25: mm/rmap.c: `page_remove_rmap()`

This either defers to `page_remove_file_rmap()` if a file mapping, `page_remove_anon_compound_rmap()` (for Transparent Huge Pages, out of scope for this section) if it's a compound mapping or simply decrements the folio's `_mapcount` field and if it has reached -1, decrement stats on the number of anonymous mapped pages.

After some THP and cgroup handling (out of scope for this section) `munlock_vma_page()` is called to remove any existing `mlock()` condition of the folios if the `VM_LOCKED` flag of the VMA is set (see section 8.2.1 for more details on this).

This removal is invoked by `zap_pte_range()`, `wp_page_copy()` when an old page is freed, the reverse mapping walk function `try_to_unmap_one()` when unmapping memory (discussed in more detail below in section 9.9.3), and the reverse mapping walk functions `try_to_migrate_one()` and `page_make_device_exclusive_one()` in addition to some out of scope KSM and huge page functions.

Note the importance of the folio's `_mapcount` field – this is incremented when the folio is mapped and decremented where, as here, it is unmapped. It is used here and in other places to determine whether a folio is mapped at all, exclusively mapped or part of a shared mapping.

7.0.13 File-backed reverse mapping folio operations

File-backed folios also count the number of mapping VMAs via `_mapcount`, adding the mapping to the folio reverse mapping via `page_add_file_rmap()`:-

```

1285  /**
1286   * page_add_file_rmap - add pte mapping to a file page
1287   * @page:           the page to add the mapping to

```

```

1288 * @vma:      the vm area in which the mapping is added
1289 * @compound: charge the page as compound or small page
1290 *
1291 * The caller needs to hold the pte lock.
1292 */
1293 void page_add_file_rmap(struct page *page,
1294                        struct vm_area_struct *vma, bool compound)
1295 {
1296     int i, nr = 0;
1297
1298     VM_BUG_ON_PAGE(compound && !PageTransHuge(page), page);
1299     lock_page_memcg(page);
1300     :
1301     if (atomic_inc_and_test(&page->_mapcount))
1302         nr++;
1303     :
1304     if (nr)
1305         __mod_lruvec_page_state(page, NR_FILE_MAPPED, nr);
1306     unlock_page_memcg(page);
1307
1308     mlock_vma_page(page, vma, compound);
1309 }

```

Listing 7-26: mm/rmap.c: page_add_file_rmap()

We elide out of scope huge page handling. Here we simply increment the `_mapcount` and also invoke `mlock_vma_page()` to handle `mlock'd` VMAs.

Removal of file-backed folio mappings is performed via `page_remove_rmap()` which in turn invokes `page_remove_file_rmap()` as shown in Listing 7-74.

```

1342 static void page_remove_file_rmap(struct page *page, bool compound)
1343 {
1344     int i, nr = 0;
1345
1346     VM_BUG_ON_PAGE(compound && !PageHead(page), page);
1347     :
1348     if (atomic_add_negative(-1, &page->_mapcount))
1349         nr++;
1350     :
1351     if (nr)
1352         __mod_lruvec_page_state(page, NR_FILE_MAPPED, -nr);
1353 }

```

Listing 7-27: mm/rmap.c: page_remove_file_rmap()

Eliding the out of scope huge page considerations, this simply deducts the folio's `mapcount`.

7.0.14 Unlinking anon_vma objects

When a struct `vm_area_struct` (VMA) needs to unlink all attached struct `anon_vma` mappings, the code doing so invokes `unlink_anon_vmas()`:-

```

395 void unlink_anon_vmas(struct vm_area_struct *vma)
396 {
397     struct anon_vma_chain *avc, *next;
398     struct anon_vma *root = NULL;
399
400     /*
401      * Unlink each anon_vma chained to the VMA. This list is ordered
402      * from newest to oldest, ensuring the root anon_vma gets freed last.
403      */
404     list_for_each_entry_safe(avc, next, &vma->anon_vma_chain, same_vma) {
405         struct anon_vma *anon_vma = avc->anon_vma;
406
407         root = lock_anon_vma_root(root, anon_vma);
408         anon_vma_interval_tree_remove(avc, &anon_vma->rb_root);
409
410         /*
411          * Leave empty anon_vmas on the list - we'll need
412          * to free them outside the lock.
413          */
414         if (RB_EMPTY_ROOT(&anon_vma->rb_root.rb_root)) {
415             anon_vma->parent->num_children--;
416             continue;
417         }
418
419         list_del(&avc->same_vma);
420         anon_vma_chain_free(avc);
421     }
422     if (vma->anon_vma) {
423         vma->anon_vma->num_active_vmas--;
424
425         /*
426          * vma would still be needed after unlink, and anon_vma will
427          * be prepared
428          * when handle fault.
429          */
430         vma->anon_vma = NULL;
431     }
432     unlock_anon_vma_root(root);
433
434     /*
435      * Iterate the list once more, it now only contains empty and unlinked

```

```

435      * anon_vmas, destroy them. Could not do before due to __put_anon_vma
      ()
436      * needing to write-acquire the anon_vma->root->rwsem.
437      */
438      list_for_each_entry_safe(avc, next, &vma->anon_vma_chain, same_vma) {
439          struct anon_vma *anon_vma = avc->anon_vma;
440
441          VM_WARN_ON(anon_vma->num_children);
442          VM_WARN_ON(anon_vma->num_active_vmas);
443          put_anon_vma(anon_vma);
444
445          list_del(&avc->same_vma);
446          anon_vma_chain_free(avc);
447      }
448 }

```

Listing 7-28: mm/rmap.c: `unlink_anon_vmas()`

This iterates through a struct `vm_area_struct` (VMA)'s `anon_vma_chain` list, looking up each struct `anon_vma` object it is linked too via each struct `anon_vma_chain` object, and removes the VMA from the underlying `anon_vma`.

This is performed under the lock of the root `anon_vma` via `lock_anon_vma_root()`, a convenience function which acquires a write lock on the read/write semaphore, releasing the previous and acquiring the new one if we encounter an AVC pointing to a different `anon_vma` tree.

If any are therefore rendered empty, its `num_children` count is subtracted and it *remains* on the `anon_vma_chain` for later processing. This is because we need to call `put_anon_vma()` to decrement the `anon_vma`'s reference count and free it if it reaches zero, which itself requires a write lock on the root read/write semaphore so we must process the list and release this lock before we can process these.

If the entry is not empty then we simply delete the entry from the list and free the AVC via `anon_vma_chain_free()`. Note this means that `anon_vma` objects can stick around and even become 'zombies' if they form part of an `anon_vma` tree, however we have mechanisms for reusing these, as discussed in section 7.0.9.

After this process is complete, we decrement the VMA's target `anon_vma->num_active_vmas` field and set its target `anon_vma` to `NULL`.

Once this is done, the lock is released and we can then finally clear down the empty `anon_vma`'s. This is done via `put_anon_vma()` before removing the AVC from the `anon_vma_chain` VMA list and freeing the AVC via `anon_vma_chain_free()` as above.

Examining `put_anon_vma()` as shown in Listing 7-29.

```

113 static inline void put_anon_vma(struct anon_vma *anon_vma)
114 {
115     if (atomic_dec_and_test(&anon_vma->refcount))
116         __put_anon_vma(anon_vma);

```

117 }

Listing 7-29: include/linux/rmap.h: put_anon_vma()

The refcount is decremented and if it reaches zero, `__put_anon_vma()` as shown in Listing 7-30.

```

2343 void __put_anon_vma(struct anon_vma *anon_vma)
2344 {
2345     struct anon_vma *root = anon_vma->root;
2346
2347     anon_vma_free(anon_vma);
2348     if (root != anon_vma && atomic_dec_and_test(&root->refcount))
2349         anon_vma_free(root);
2350 }
```

Listing 7-30: mm/rmap.c: __put_anon_vma()

This frees the `anon_vma` and, if the roots reference count is 1, frees that too.

7.0.15 Walking the reverse mapping

We have examined how the reverse mapping is structured and maintained in detail – however we have yet to examine it in *action*, i.e. being used to perform an operation on a folio's VMAs.

The key function which perform this action is `rmap_walk()` as shown in Listing 7-31.

```

2494 void rmap_walk(struct folio *folio, struct rmap_walk_control *rwc)
2495 {
2496     if (unlikely(folio_test_ksm(folio)))
2497         rmap_walk_ksm(folio, rwc);
2498     else if (folio_test_anon(folio))
2499         rmap_walk_anon(folio, rwc, false);
2500     else
2501         rmap_walk_file(folio, rwc, false);
2502 }
```

Listing 7-31: mm/rmap.c: rmap_walk()

Alternatively the same thing can be achieved via `rmap_walk_locked()` which assumes the relevant reverse mapping locks are held (the root struct `anon_vma->rwsem` lock if anonymous, and the struct `address_space->i_mmap_rwsem`) lock if file-backed as shown in Listing 7-32.

```

2504 /* Like rmap_walk, but caller holds relevant rmap lock */
2505 void rmap_walk_locked(struct folio *folio, struct rmap_walk_control *rwc)
2506 {
2507     /* no ksm support for now */
```

```

2508     VM_BUG_ON_FOLIO(folio_test_ksm(folio), folio);
2509     if (folio_test_anon(folio))
2510         rmap_walk_anon(folio, rwc, true);
2511     else
2512         rmap_walk_file(folio, rwc, true);
2513 }

```

Listing 7-32: mm/rmap.c: rmap_walk_locked()

The Kernel Same-page Merging (KSM) handling is out of scope, so we will elide this discussion. Before we dive into the individual walk functions, let's examine struct rmap_walk_control as shown in Listing 7-33.

```

379  /*
380   * rmap_walk_control: To control rmap traversing for specific needs
381   *
382   * arg: passed to rmap_one() and invalid_vma()
383   * try_lock: bail out if the rmap lock is contended
384   * contended: indicate the rmap traversal bailed out due to lock contention
385   * rmap_one: executed on each vma where page is mapped
386   * done: for checking traversing termination condition
387   * anon_lock: for getting anon_lock by optimized way rather than default
388   * invalid_vma: for skipping uninterested vma
389   */
390  struct rmap_walk_control {
391      void *arg;
392      bool try_lock;
393      bool contended;
394      /*
395       * Return false if page table scanning in rmap_walk should be stopped.
396       * Otherwise, return true.
397       */
398      bool (*rmap_one)(struct folio *folio, struct vm_area_struct *vma,
399                      unsigned long addr, void *arg);
400      int (*done)(struct folio *folio);
401      struct anon_vma *(*anon_lock)(struct folio *folio,
402                                   struct rmap_walk_control *rwc);
403      bool (*invalid_vma)(struct vm_area_struct *vma, void *arg);
404  };

```

Listing 7-33: include/linux/rmap.h: struct rmap_walk_control

This allows a caller to specify parameters for the walk and a series of functions that will be invoked at specific points during the walk:-

- **arg** – This is an arbitrary user-defined value which will be passed to the rmap_one() and invalid_vma() on invocation during the walk (and available to anon_lock()) via the rwc object). This can be useful if the walk has to thread some kind of state through it.

- `try_lock` – Indicates whether to abort the walk if the root struct `anon_vma` read/write semaphore becomes contended. If it does, the `contended` field will be set true. This is used when the `anon_vma_trylock_read()` variant of the lock should be preferred (and thus we don't try to wait to acquire a lock) over `anon_vma_lock_read()`.
- `contended` – *See above.*
- `rmap_one` – A function that is called for each of the folio's located struct `vm_area_struct` (VMA)'s object. If the function returns false, then the walk is aborted.
- `done` – Checked after each VMA is examined, parameterised by a folio and used to determine whether the operation is complete. If this is non-NULL and return true then the walk is terminated.
- `anon_lock` – An optional function which is used instead of the default locking mechanism, `rmap_walk_anon_lock()` if a more efficient means can be determined by the caller.
- `invalid_vma` – Used to determine whether a VMA is of interest or not – if it returns true then this VMA is skipped.

Examining the `rmap_walk_anon()` function as shown in Listing 7-34.

```

2384 /*
2385  * rmap_walk_anon - do something to anonymous page using the object-based
2386  * rmap method
2387  * @page: the page to be handled
2388  * @rwc: control variable according to each walk type
2389  *
2390  * Find all the mappings of a page using the mapping pointer and the vma
2391  * chains
2392  * contained in the anon_vma struct it points to.
2393  */
2394 static void rmap_walk_anon(struct folio *folio,
2395                          struct rmap_walk_control *rwc, bool locked)
2396 {
2397     struct anon_vma *anon_vma;
2398     pgoff_t pgoff_start, pgoff_end;
2399     struct anon_vma_chain *avc;
2400
2401     if (locked) {
2402         anon_vma = folio_anon_vma(folio);
2403         /* anon_vma disappear under us? */
2404         VM_BUG_ON_FOLIO(!anon_vma, folio);
2405     } else {
2406         anon_vma = rmap_walk_anon_lock(folio, rwc);
2407     }
2408     if (!anon_vma)
2409         return;

```

```

2409
2410     pgoff_start = folio_pgoff(folio);
2411     pgoff_end = pgoff_start + folio_nr_pages(folio) - 1;
2412     anon_vma_interval_tree_foreach(avc, &anon_vma->rb_root,
2413                                     pgoff_start, pgoff_end) {
2414         struct vm_area_struct *vma = avc->vma;
2415         unsigned long address = vma_address(&folio->page, vma);
2416
2417         VM_BUG_ON_VMA(address == -EFAULT, vma);
2418         cond_resched();
2419
2420         if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
2421             continue;
2422
2423         if (!rwc->rmap_one(folio, vma, address, rwc->arg))
2424             break;
2425         if (rwc->done && rwc->done(folio))
2426             break;
2427     }
2428
2429     if (!locked)
2430         anon_vma_unlock_read(anon_vma);
2431 }

```

Listing 7-34: mm/rmap.c: rmap_walk_anon()

If a lock is held elsewhere (i.e. `rmap_walk_locked()` was invoked), then the struct `anon_vma` object is obtained via the `folio_anon_vma()` helper function as shown in Listing 7-35.

```

778 struct anon_vma *folio_anon_vma(struct folio *folio)
779 {
780     unsigned long mapping = (unsigned long)folio->mapping;
781
782     if ((mapping & PAGE_MAPPING_FLAGS) != PAGE_MAPPING_ANON)
783         return NULL;
784     return (void *) (mapping - PAGE_MAPPING_ANON);
785 }

```

Listing 7-35: mm/util.c: folio_anon_vma()

This function extracts the `anon_vma` mapping from a `folio`. Since the `folio`'s mapping field can be overloaded with either `PAGE_MAPPING_ANON` (indicating it is an anonymous mapping), `PAGE_MAPPING_MOVABLE` or `PAGE_MAPPING_KSM` (a combination of the previous two), we must compare the `PAGE_MAPPING_FLAGS`-masked bits precisely with `PAGE_MAPPING_ANON` to determine that this is anonymous.

This function then simply extracts the struct `anon_vma` from this field and returns it.

Examining `rmap_walk_anon_lock()` as shown in Listing 7-36.

```

2352 static struct anon_vma *rmap_walk_anon_lock(struct folio *folio,
2353                                           struct rmap_walk_control *rwc)
2354 {
2355     struct anon_vma *anon_vma;
2356
2357     if (rwc->anon_lock)
2358         return rwc->anon_lock(folio, rwc);
2359
2360     /*
2361      * Note: remove_migration_ptes() cannot use folio_lock_anon_vma_read()
2362      * because that depends on page_mapped(); but not all its usages
2363      * are holding mmap_lock. Users without mmap_lock are required to
2364      * take a reference count to prevent the anon_vma disappearing
2365      */
2366     anon_vma = folio_anon_vma(folio);
2367     if (!anon_vma)
2368         return NULL;
2369
2370     if (anon_vma_trylock_read(anon_vma))
2371         goto out;
2372
2373     if (rwc->try_lock) {
2374         anon_vma = NULL;
2375         rwc->contended = true;
2376         goto out;
2377     }
2378
2379     anon_vma_lock_read(anon_vma);
2380 out:
2381     return anon_vma;
2382 }

```

Listing 7-36: mm/rmap.c: *rmap_walk_anon_lock()*

If a custom `anon_lock()` function is supplied, then this is used. Otherwise, we determine the `anon_vma` object from the `folio` via `folio_anon_vma()` as described previously.

We then try to acquire a read lock on the `anon_vma` tree's root object via `anon_vma_trylock_read()`. If this does not succeed, and the `try_lock` field is specified, we mark the lock contended and abort. Otherwise we wait for the lock to be available via `anon_vma_lock_read()`.

Returning to `rmap_walk_anon_lock()`, we next determine the range of `struct anon_vma_chain` objects mapping our `anon_vma` object to VMAs via `folio_pgoff()` which, for non-huge mapping just uses the `folio`'s index field.

We determine which AVCs span this range via the macro defined in `anon_vma_interval_tree_foreach()` as shown in Listing 7-37.

```

2572 #define anon_vma_interval_tree_foreach(avc, root, start, last) \

```

```

2573         for (avc = anon_vma_interval_tree_iter_first(root, start, last); \
2574             avc; avc = anon_vma_interval_tree_iter_next(avc, start, last))

```

Listing 7-37: include/linux/mm.h: anon_vma_interval_tree_foreach()

This iterates through all of the struct `vm_area_struct` (VMA)s connected to this folio's struct `anon_vma` field via struct `anon_vma_chain` connecting objects.

In each of the iterations of this loop executed by `rmap_walk_anon()`, we determine the address of the folio within the VMA via `vma_address()` (and in turn `vma_pgoff_address()`), then run it through the provided `invalid_vma()`, `rmap_one()` and `done()` functions with semantics as described above.

Examining the equivalent walk function for file-backed folios, `rmap_walk_file()` as shown in Listing 7-38.

```

2433  /*
2434   * rmap_walk_file - do something to file page using the object-based rmap
2435   *                   method
2436   * @page: the page to be handled
2437   * @rwc: control variable according to each walk type
2438   * Find all the mappings of a page using the mapping pointer and the vma
2439   *       chains
2440   * contained in the address_space struct it points to.
2441   */
2442  static void rmap_walk_file(struct folio *folio,
2443                           struct rmap_walk_control *rwc, bool locked)
2444  {
2445      struct address_space *mapping = folio_mapping(folio);
2446      pgoff_t pgoff_start, pgoff_end;
2447      struct vm_area_struct *vma;
2448
2449      /*
2450       * The page lock not only makes sure that page->mapping cannot
2451       * suddenly be NULLified by truncation, it makes sure that the
2452       * structure at mapping cannot be freed and reused yet,
2453       * so we can safely take mapping->i_mmap_rwsem.
2454       */
2455      VM_BUG_ON_FOLIO(!folio_test_locked(folio), folio);
2456
2457      if (!mapping)
2458          return;
2459
2460      pgoff_start = folio_pgoff(folio);
2461      pgoff_end = pgoff_start + folio_nr_pages(folio) - 1;
2462      if (!locked) {
2463          if (i_mmap_trylock_read(mapping))
2464              goto lookup;

```

```

2465         if (rwc->try_lock) {
2466             rwc->contended = true;
2467             return;
2468         }
2469
2470         i_mmap_lock_read(mapping);
2471     }
2472 lookup:
2473     vma_interval_tree_foreach(vma, &mapping->i_mmap,
2474                               pgoff_start, pgoff_end) {
2475         unsigned long address = vma_address(&folio->page, vma);
2476
2477         VM_BUG_ON_VMA(address == -EFAULT, vma);
2478         cond_resched();
2479
2480         if (rwc->invalid_vma && rwc->invalid_vma(vma, rwc->arg))
2481             continue;
2482
2483         if (!rwc->rmap_one(folio, vma, address, rwc->arg))
2484             goto done;
2485         if (rwc->done && rwc->done(folio))
2486             goto done;
2487     }
2488
2489 done:
2490     if (!locked)
2491         i_mmap_unlock_read(mapping);
2492 }

```

Listing 7-38: mm/rmap.c: *rmap_walk_file()*

This mirrors *rmap_walk_anon()*, only with the equivalent operations for file-backed mappings.

In both cases, we determine a folio's virtual address via *vma_address()* as shown in Listing 7-39.

```

573 /*
574  * Return the start of user virtual address of a page within a vma.
575  * Returns -EFAULT if all of the page is outside the range of vma.
576  * If page is a compound head, the entire compound page is considered.
577  */
578 static inline unsigned long
579 vma_address(struct page *page, struct vm_area_struct *vma)
580 {
581     VM_BUG_ON_PAGE(PageKsm(page), page); /* KSM page->index unusable */
582     return vma_pgoff_address(page_to_pgoff(page), compound_nr(page), vma);
583 }

```

Listing 7-39: mm/internal.h: *vma_address()*

This functions by determining the page offset of the folio within the mapping via `page_to_pgoff()` (for anything other than huge pages this will simply be the struct `folio`'s `index` field), which is passed to `vma_pgoff_address()` to do the heavy lifting, as shown in Listing 7-40.

```

548 /*
549  * Return the start of user virtual address at the specific offset within
550  * a vma.
551  */
552 static inline unsigned long
553 vma_pgoff_address(pgoff_t pgoff, unsigned long nr_pages,
554                  struct vm_area_struct *vma)
555 {
556     unsigned long address;
557
558     if (pgoff >= vma->vm_pgoff) {
559         address = vma->vm_start +
560                 ((pgoff - vma->vm_pgoff) << PAGE_SHIFT);
561         /* Check for address beyond vma (or wrapped through 0?) */
562         if (address < vma->vm_start || address >= vma->vm_end)
563             address = -EFAULT;
564     } else if (pgoff + nr_pages - 1 >= vma->vm_pgoff) {
565         /* Test above avoids possibility of wrap to 0 on 32-bit */
566         address = vma->vm_start;
567     } else {
568         address = -EFAULT;
569     }
570     return address;
571 }

```

Listing 7-40: `mm/internal.h: vma_pgoff_address()`

For non-compound (i.e. in this scenario, non-huge) pages, this is simply the provided page offset, less the struct `vm_area_struct` (VMA)'s `vm_pgoff` field, multiplied by base page size and added to the `vm_start` field – in other words, the virtual address of the folio within the mapping, as required.

Note that for anonymous backings the page offsets are virtual page offsets and for file backings the page offsets are 0-indexed offsets into the page cache object.

The additional logic here accounts for a compound case wrapping around on 32-bit systems and of course, if the folio does not reside within the VMA, it returns an error.

The equivalent to `folio_anon_vma()` is `folio_mapping()`, as shown in Listing 7-41.

```

787 /**
788  * folio_mapping - Find the mapping where this folio is stored.
789  * @folio: The folio.
790  *

```

```

791  * For folios which are in the page cache, return the mapping that this
792  * page belongs to. Folios in the swap cache return the swap mapping
793  * this page is stored in (which is different from the mapping for the
794  * swap file or swap device where the data is stored).
795  *
796  * You can call this for folios which aren't in the swap cache or page
797  * cache and it will return NULL.
798  */
799  struct address_space *folio_mapping(struct folio *folio)
800  {
801      struct address_space *mapping;
802
803      /* This happens if someone calls flush_dcache_page on slab page */
804      if (unlikely(folio_test_slab(folio)))
805          return NULL;
806
807      if (unlikely(folio_test_swapcache(folio)))
808          return swap_address_space(folio_swap_entry(folio));
809
810      mapping = folio->mapping;
811      if ((unsigned long)mapping & PAGE_MAPPING_FLAGS)
812          return NULL;
813
814      return mapping;
815  }

```

Listing 7-41: mm/util.c: folio_mapping()

This performs a check against a race with the kernel slab mechanism (see the slab chapter for more on this), then a swap cache check (see the swap chapter for a discussion of the kernel swap mechanism, this is out of scope here), before finally checking whether any PAGE_MAPPING_FLAGS have been set - if they have, the mapping is invalid and it returns NULL, otherwise it returns a pointer to the relevant struct address_space object.

rmap_walk_file() then performs a check to ensure that the folio is not locked (more on folio locking in the page cache chapter), and aborts if no address_space object could be found.

Next the offsets to be searched for are determined in precisely the same fashion as in rmap_walk_anon(), then the try lock/lock dance is performed, only this time against the i_mmap_rwsem lock, using i_mmap_trylock_read() and i_mmap_lock_read().

We loop over the mappings using vma_interval_tree_foreach() as shown in Listing 7-42.

```

2555 #define vma_interval_tree_foreach(vma, root, start, last)      \
2556     for (vma = vma_interval_tree_iter_first(root, start, last); \
2557          vma; vma = vma_interval_tree_iter_next(vma, start, last))

```

Listing 7-42: include/linux/mm.h: vma_interval_tree_foreach()

Which iterates through `i_mmap` returning all of the related `struct vm_area_struct` (VMA)s. The operations performed on them mirror those performed by `rmap_walk_anon()`.

Finally, the `i_mmap_rwsem` read lock is released via `i_mmap_unlock_read()` if this is not invoked via `rmap_walk_locked()`.

There are a number of users of this mechanism, each of which make intricate use of this functionality. For brevity, we won't examine these closely but instead list some core examples:-

- `folio_referenced()` – An absolutely critical function used in reclaim to determine the number of mappings which reference the specified folio.
- `try_to_unmap()` – Another absolutely critical function, and indeed the motivating case we discussed at the start of this section – this attempts to unmap the specified folio from all VMAs which map it. This can call either the unlocked or the locked version of the walk function depending on passed flags.
- `folio_mkclean()` – Clears the dirty flag for the specified folio and marks the mapping write-protected as part of writeback via `folio_clear_dirty_for_io()`.
- `try_to_migrate()` – Used as part of the migration process (see the chapter on this for more details on this topic as a whole), replacing mappings with swap cache entries (for details on the swap cache, see the swap chapter). This can call either the unlocked or the locked version of the walk function depending on passed flags.
- `folio_make_device_exclusive()` – Used to mark a folio as being exclusively owned by a specific device, as invoked by `make_device_exclusive_range()` as part of the functionality provided by `CONFIG_DEVICE_PRIVATE`.
- `remove_migration_ptes()` – Removes migration entries and sets actual page mappings as part of the migration process. This can call either the unlocked or the locked version of the walk function depending on its locked parameter.

7.0.16 Walking the VMA

Recall that the reverse mapping might reference VMAs which do not, in fact, map the folio being walked – the folio may have since been CoW'd to a child process, but the parent's `struct anon_vma` will still map to the child's VMA.

This is something that has to be checked once a VMA is obtained from the reverse mapping. In addition a mapping can disappear from beneath us at any time, without a lock on the PTE we simply cannot be sure that the mapping still exists, so this is absolutely necessary.

We therefore must walk the page tables to check that the folio is indeed mapped by the VMA at a specific address, and acquire its PTE lock while we do so. This is performed by `page_vma_mapped_walk()` and mediated by the `struct page_vma_mapped_walk` object, which we shall examine first as shown in Listing 7-43.

```

316 struct page_vma_mapped_walk {
317     unsigned long pfn;
318     unsigned long nr_pages;
319     pgoff_t pgoff;
320     struct vm_area_struct *vma;
321     unsigned long address;
322     pmd_t *pmd;
323     pte_t *pte;
324     spinlock_t *ptl;
325     unsigned int flags;
326 };

```

Listing 7-43: include/linux/rmap.h: struct page_vma_mapped_walk

Examining each field:-

- *pfn – required* – The Page Frame Number (PFN) of the folio we are checking the mappings for. This is the physical address of the folio divided by the base page size and can be obtained for a folio via `folio_pfn()`.
- *nr_pages – required* – The number of base pages to iterate over starting from the PFN. Typically this will be equal to 1 except for huge pages or for operations which are iterating over a range of folios like `pfn_mkclean_range()`.
- *pgoff – required* – The page offset of the folio. For anything other than huge pages this will be equal to the `struct folio->index` field.
- *vma – required* – The `struct vm_area_struct` (VMA) whose page table mappings we are walking.
- *address – required* – The virtual address we are checking exists within the VMA. In the usual case where this walk is being called from within a reverse-mapping walk this will have been determined via `vma_address()`.
- *pmd – output* – A pointer to the PMD entry associated with this mapping.
- *pte – output* – A pointer to the PTE entry associated with this mapping.
- *ptl – output* – The page table lock which will be acquired should a mapping be discovered.
- *flags – required* (can be zero) – Flags which modify the behaviour of the walk – can either be zero if there are no special requirements, `PVMW_SYNC` if checks which might result in race conditions are to be avoided and `PVMW_MIGRATION` if, instead of PTEs, we are looking for migration entries (see the migration chapter for more on this).

Often these parameters are set by macro, specifically `DEFINE_FOLIO_VMA_WALK` and the struct page equivalent `DEFINE_PAGE_VMA_WALK`. Examining the former as shown in Listing 7-44.

```

338 #define DEFINE_FOLIO_VMA_WALK(name, _folio, _vma, _address, _flags)    \
339     struct page_vma_mapped_walk name = {                               \

```

```

340         .pfn = folio_pfn(_folio),
341         .nr_pages = folio_nr_pages(_folio),
342         .pgoff = folio_pgoff(_folio),
343         .vma = _vma,
344         .address = _address,
345         .flags = _flags,
346     }

```

Listing 7-44: include/linux/rmap.h: DEFINE_FOLIO_VMA_WALK

This determines the PFN via `folio_pfn()`, the page count via `folio_nr_pages()` and the page offset via `folio_pgoff()`.

The `page_vma_mapped_walk()` function itself accepts a single parameter of a struct `page_vma_mapped_walk` object and returns true if a valid mapping is received, with `pmd`, `pte` and `ptl` set and false when none remain, with the `ptl` lock released.

This function is therefore typically called in a while loop in order to handle huge page cases. If this process needs to be exited early, `page_vma_mapped_walk_done()` can be called, which unlocks the page table lock, thus aborting the process as shown in Listing 7-45.

```

348 static inline void page_vma_mapped_walk_done(struct page_vma_mapped_walk *pvmw
349 )
350 {
351     /* HugeTLB pte is set to the relevant page table entry without
352     pte_mapped. */
353     if (pvmw->pte && !is_vm_hugetlb_page(pvmw->vma))
354         pte_unmap(pvmw->pte);
355     if (pvmw->ptl)
356         spin_unlock(pvmw->ptl);
357 }

```

Listing 7-45: include/linux/rmap.h: page_vma_mapped_walk_done()

For anything resembling a modern system, `pte_unmap()` is a no-op, so this simply frees the PTE page table lock `ptl`.

Examining `page_vma_mapped_walk()` (eliding out of scope huge page and `devmap` logic) as shown in Listing 7-46.

```

127 /**
128  * page_vma_mapped_walk - check if @pvmw->pfn is mapped in @pvmw->vma at
129  * @pvmw->address
130  * @pvmw: pointer to struct page_vma_mapped_walk. page, vma, address and flags
131  * must be set. pmd, pte and ptl must be NULL.
132  *
133  * Returns true if the page is mapped in the vma. @pvmw->pmd and @pvmw->pte
134  * point
135  * to relevant page table entries. @pvmw->ptl is locked. @pvmw->address is
136  * adjusted if needed (for PTE-mapped THPs).
137  */

```

```

137  * If @pvmw->pmd is set but @pvmw->pte is not, you have found PMD-mapped page
138  * (usually THP). For PTE-mapped THP, you should run page_vma_mapped_walk() in
139  * a loop to find all PTEs that map the THP.
140  *
141  * For HugeTLB pages, @pvmw->pte is set to the relevant page table entry
142  * regardless of which page table level the page is mapped at. @pvmw->pmd is
143  * NULL.
144  *
145  * Returns false if there are no more page table entries for the page in
146  * the vma. @pvmw->ptl is unlocked and @pvmw->pte is unmapped.
147  *
148  * If you need to stop the walk before page_vma_mapped_walk() returned false,
149  * use page_vma_mapped_walk_done(). It will do the housekeeping.
150  */
151  bool page_vma_mapped_walk(struct page_vma_mapped_walk *pvmw)
152  {
153      struct vm_area_struct *vma = pvmw->vma;
154      struct mm_struct *mm = vma->vm_mm;
155      unsigned long end;
156      pgd_t *pgd;
157      p4d_t *p4d;
158      pud_t *pud;
159      pmd_t pmd;
160
161      /* The only possible pmd mapping has been handled on last iteration */
162      if (pvmw->pmd && !pvmw->pte)
163          return not_found(pvmw);
164
165      if (unlikely(is_vm_hugetlb_page(vma))) {
166          :
167      }
168
169      end = vma_address_end(pvmw);
170      if (pvmw->pte)
171          goto next_pte;
172
173  restart:
174      do {
175          pgd = pgd_offset(mm, pvmw->address);
176          if (!pgd_present(*pgd)) {
177              step_forward(pvmw, PGDIR_SIZE);
178              continue;
179          }
180          p4d = p4d_offset(pgd, pvmw->address);
181          if (!p4d_present(*p4d)) {
182              step_forward(pvmw, P4D_SIZE);
183              continue;
184          }
185      }

```

```

198         pud = pud_offset(p4d, pvmw->address);
199         if (!pud_present(*pud)) {
200             step_forward(pvmw, PUD_SIZE);
201             continue;
202         }
203
204         pvmw->pmd = pmd_offset(pud, pvmw->address);
205         /*
206          * Make sure the pmd value isn't cached in a register by the
207          * compiler and used as a stale value after we've observed a
208          * subsequent update.
209          */
210         pmde = READ_ONCE(*pvmw->pmd);
211
212         if (pmd_trans_huge(pmde) || is_pmd_migration_entry(pmde) ||
213             (pmd_present(pmde) && pmd_devmap(pmde))) {
214             :
215
216             } else if (!pmd_present(pmde)) {
217                 :
218
219                 step_forward(pvmw, PMD_SIZE);
220                 continue;
221             }
222         if (!map_pte(pvmw))
223             goto next_pte;
224
225         this_pte:
226         if (check_pte(pvmw))
227             return true;
228
229         next_pte:
230         do {
231             pvmw->address += PAGE_SIZE;
232             if (pvmw->address >= end)
233                 return not_found(pvmw);
234             /* Did we cross page table boundary? */
235             if ((pvmw->address & (PMD_SIZE - PAGE_SIZE)) == 0) {
236                 if (pvmw->ptl) {
237                     spin_unlock(pvmw->ptl);
238                     pvmw->ptl = NULL;
239                 }
240                 pte_unmap(pvmw->pte);
241                 pvmw->pte = NULL;
242                 goto restart;
243             }
244             pvmw->pte++;
245             if ((pvmw->flags & PVMW_SYNC) && !pvmw->ptl) {
246                 pvmw->ptl = pte_lockptr(mm, pvmw->pmd);
247                 spin_lock(pvmw->ptl);
248             }
249         }

```

```

279         } while (pte_none(*pvmw->pte));
280
281         if (!pvmw->ptl) {
282             pvmw->ptl = pte_lockptr(mm, pvmw->pmd);
283             spin_lock(pvmw->ptl);
284         }
285         goto this_pte;
286     } while (pvmw->address < end);
287
288     return false;
289 }

```

Listing 7-46: mm/page_vma_mapped.c: page_vma_mapped_walk()

This uses helper functions to adjust the state of the struct page_vma_mapped_walk object – not_found() and step_forward(). The former simply invokes page_vma_mapped_walk_done() and returns false, therefore aborting the operation, let’s examine the latter as shown in Listing 7-47.

```

120 static void step_forward(struct page_vma_mapped_walk *pvmw, unsigned long size
121 )
122 {
123     pvmw->address = (pvmw->address + size) & ~(size - 1);
124     if (!pvmw->address)
125         pvmw->address = ULONG_MAX;
126 }

```

Listing 7-47: mm/page_vma_mapped.c: step_forward()

This advances the address field by the specified size, clearing the lower bits as it does so. If the value wraps around, ULONG_MAX is set, i.e. placing the address at the maximum possible value and therefore aborting the operation on the next iteration.

The vma_address_end() helper function is critical here – it determines the first address after that spanned by the folio (i.e. an exclusive bound). Examining it as shown in Listing 7-48.

```

585 /*
586  * Then at what user virtual address will none of the range be found in vma?
587  * Assumes that vma_address() already returned a good starting address.
588  */
589 static inline unsigned long vma_address_end(struct page_vma_mapped_walk *pvmw)
590 {
591     struct vm_area_struct *vma = pvmw->vma;
592     pgoff_t pgoff;
593     unsigned long address;
594
595     /* Common case, plus ->pgoff is invalid for KSM */
596     if (pvmw->nr_pages == 1)

```

```

597         return pvmw->address + PAGE_SIZE;
598
599         pgoff = pvmw->pgoff + pvmw->nr_pages;
600         address = vma->vm_start + ((pgoff - vma->vm_pgoff) << PAGE_SHIFT);
601         /* Check for address beyond vma (or wrapped through 0?) */
602         if (address < vma->vm_start || address > vma->vm_end)
603             address = vma->vm_end;
604         return address;
605     }

```

Listing 7-48: mm/internal.h: vma_address_end()

For single-page folios (i.e. non-huge), this simply returns the address field incremented by base page size. Otherwise, this limits the return value to the final address of the struct `vm_area_struct` (VMA).

PTEs are mapped via `map_pte()` which, eliding out of scope swap and migration cases, is rather simple as shown in Listing 7-49.

```

16 static bool map_pte(struct page_vma_mapped_walk *pvmw)
17 {
18     pvmw->pte = pte_offset_map(pvmw->pmd, pvmw->address);
19     if (!pvmw->flags & PVMW_SYNC) {
20         if (pvmw->flags & PVMW_MIGRATION) {
21             :
47             } else if (!pte_present(*pvmw->pte))
48                 return false;
49         }
50     }
51     pvmw->ptl = pte_lockptr(pvmw->vma->vm_mm, pvmw->pmd);
52     spin_lock(pvmw->ptl);
53     return true;
54 }

```

Listing 7-49: mm/page_vma_mapped.c: map_pte()

The PTE entry is obtained via `pte_offset_map()` from the previously obtained `pmd` entry, the PTE lock is obtained via `pte_lockptr()` and placed in the `ptl` field before being locked.

Before a candidate PTE is returned, the `check_pte()` function is called. With migration and swap cases elided as out of scope for this section, the function is relatively simple as shown in Listing 7-50.

```

56 /**
57  * check_pte - check if @pvmw->page is mapped at the @pvmw->pte
58  * @pvmw: page_vma_mapped_walk struct, includes a pair pte and page for
59  *         checking
60  * page_vma_mapped_walk() found a place where @pvmw->page is *potentially*
61  * mapped. check_pte() has to validate this.
62  */

```

```

63  * pvmw->pte may point to empty PTE, swap PTE or PTE pointing to
64  * arbitrary page.
65  *
66  * If PVMW_MIGRATION flag is set, returns true if @pvmw->pte contains
    migration
67  * entry that points to @pvmw->page or any subpage in case of THP.
68  *
69  * If PVMW_MIGRATION flag is not set, returns true if pvmw->pte points to
70  * pvmw->page or any subpage in case of THP.
71  *
72  * Otherwise, return false.
73  *
74  */
75 static bool check_pte(struct page_vma_mapped_walk *pvmw)
76 {
77     unsigned long pfn;
78
79     if (pvmw->flags & PVMW_MIGRATION) {
80         :
81     } else if (is_swap_pte(*pvmw->pte)) {
82         :
83     } else {
84         if (!pte_present(*pvmw->pte))
85             return false;
86
87         pfn = pte_pfn(*pvmw->pte);
88     }
89
90     return (pfn - pvmw->pfn) < pvmw->nr_pages;
91 }

```

Listing 7-50: mm/page_vma_mapped.c: *check_pte()*

This simply checks whether the PTE entry has its present bit set via `pte_present()`, obtains the PFN specified by the PTE entry via `pte_pfn()` and checks that this is within the `nr_pages` specified.

Now we have examined the helper functions, we can return to the logic of `page_vma_mapped_walk()`:-

1. Determine the PGD entry for the target address via `pgd_offset()` and confirm it has its present bit set via `pgd_present()`, if not we advance to the next PGD entry.
2. Determine the P4D entry for the target address via `p4d_offset()` and confirm it has its present bit set via `p4d_present()`, if not we advance to the next P4D entry.
3. Determine the PUD entry for the target address via `pud_offset()` and confirm it has its present bit set via `pud_present()`, if not we advance to the next PUD entry.

4. Determine the PMD entry for the target address via `pmd_offset()` and place it in the `pvmw` object's `pmd` field. Then check it has its present bit set via `pmd_present()`, if not we advance to the next PMD entry.
5. If we reach this stage with all the prior entries, we attempt to map the PTE referenced by the PMD entry via `map_pte()` as described above, if we cannot we advance to the `next_pte` logic.
6. If we were able to map, we then invoke `check_pte()` as described above – if the check passes, we return true to indicate that we have found a valid PTE. Note that the PTE lock is held at this point so the caller can operate on this mapping without fear of it disappearing.
7. If the map or check failed, we enter the `next_pte` logic – this attempts to find the first non-empty PTE in the folio range, which is then subject to the same checks and the process repeated throughout the range. This would only be required for huge pages, as ordinary mappings would span only one and exit quickly by invoking `not_found()`.

7.0.17 Summary

The reverse mapping functionality provides a critical link between folios and overlying VMAs, without which core kernel functionality could not be implemented.

The equivalence between the anonymous and file-backed versions of the reverse mapping are critical, as the kernel can then therefore lookup mappings indifferent to what backs a specific folio.

The subject is complicated and confusing, but once you break it down into the motivating reasons for maintaining the mapping and examining the more involved intricacies of the mechanism – changes in VMAs and forking – it becomes clear that the complexity serves an important service.

It's important to emphasise that the reverse mapping find VMAs which *might* map the folio being sought, but examining of the 'forward' mappings is still required via `page_vma_mapped_walk()`.

7.1 Freeing userland memory and the TLB

Freeing pages mapped by userland is more complicated than it might seem as a result of modern memory management hardware, specifically the Translation Lookaside Buffer (TLB) which maintains a cache between virtual addresses and physical addresses on each CPU core.

We briefly touched on this topic in the virtual memory chapter, however now we have a good grounding in how userland memory functions in the kernel, we will look at how we interact with this hardware in practice.

Kernel memory by its nature requires significantly less maintenance – there is only one kernel mapping, memory allocated from the slab or page allocators are mapped through the direct mapping which requires very little

TLB maintenance* and cases where memory is actually mapped and unmapped like `vmalloc` are infrequent enough that they require little special handling.

Userland on the other hand is eminently complicated – there are multiple address spaces each of which could be running simultaneously on multiple CPUs and actively utilising each core’s TLB cache at the point of memory being freed.

We therefore must be especially careful about how we interact with the TLB and only free userland pages once the TLB is definitely invalidated across all cores sharing the same address space.

The process of *gathering* TLB operations to perform is termed *MMU-gather* and described in detail in the generic assembly header `include/asm-generic/tlb.h`.

The order in which operations are performed is:-

1. Remove page table mappings (but TLB entries may remain in multiple CPU caches)
2. Invalidate TLB caches (now nothing should be accessing the pages)
3. (*If freeing memory*) Finally, free the underlying pages

7.1.1 Unmapping memory mapped regions

See Chapter 5 for more on memory mapping as a whole.

When examining how the TLB functionality works, it’s instructive to examine a motivating example. One of the key users of the MMU gather functionality is the code which unmaps `mmap()`’d memory ranges. This is ultimately performed by `unmap_region()` as shown in Listing 7-51.

```

2607 /*
2608  * Get rid of page table information in the indicated region.
2609  *
2610  * Called with the mm semaphore held.
2611  */
2612 static void unmap_region(struct mm_struct *mm,
2613                         struct vm_area_struct *vma, struct vm_area_struct *prev,
2614                         unsigned long start, unsigned long end)
2615 {
2616     struct vm_area_struct *next = vma_next(mm, prev);
2617     struct mmu_gather tlb;
2618
2619     lru_add_drain();
2620     tlb_gather_mmu(&tlb, mm);
2621     update_hiwater_rss(mm);

```

*. Mappings which exist there will always be mapped to the same physical memory under most circumstances (though a few exist where the direct mapping might get updated like memory hotplug).

```

2622         unmap_vmas(&tlb, vma, start, end);
2623         free_pgtables(&tlb, vma, prev ? prev->vm_end : FIRST_USER_ADDRESS,
2624                     next ? next->vm_start : USER_PGTABLES_CEILING
                );
2625         tlb_finish_mmu(&tlb);
2626     }

```

Listing 7-51: mm/mmap.c: unmap_region()

We start by invoking `tlb_gather_mmu()` which initialises a struct `mmu_gather` object which will be used throughout the operation. The heavy lifting of performing cache invalidations is handled by `unmap_vmas()`, page tables are freed via `free_pgtables()` before the TLB operation is executed (invalidating the cache and freeing memory) via `tlb_finish_mmu()`.

It's important to note that prior to this call being made, `vma` will either not be attached to any subsequent VMAs in the process address space, or will have been detached from subsequent VMAs.

This is important as later we rely on this fact to iterate through affected VMAs (described in section 7.1.5).

7.1.2 MMU gather initialisation

The struct `mmu_gather` data structure maintains state throughout the TLB operation:-

```

263  /*
264   * struct mmu_gather is an opaque type used by the mm code for passing around
265   * any data needed by arch specific code for tlb_remove_page.
266   */
267  struct mmu_gather {
268      struct mm_struct      *mm;
269
270  #ifdef CONFIG_MMU_GATHER_TABLE_FREE
271      struct mmu_table_batch *batch;
272  #endif
273
274      unsigned long        start;
275      unsigned long        end;
276      /*
277       * we are in the middle of an operation to clear
278       * a full mm and can make some optimizations
279       */
280      unsigned int          fullmm : 1;
281
282      /*
283       * we have performed an operation which
284       * requires a complete flush of the tlb
285       */

```

```

286         unsigned int          need_flush_all : 1;
287
288         /*
289          * we have removed page directories
290          */
291         unsigned int          freed_tables : 1;
292
293         /*
294          * at which levels have we cleared entries?
295          */
296         unsigned int          cleared_ptes : 1;
297         unsigned int          cleared_pmds : 1;
298         unsigned int          cleared_puds : 1;
299         unsigned int          cleared_p4ds : 1;
300
301         /*
302          * tracks VM_EXEC | VM_HUGETLB in tlb_start_vma
303          */
304         unsigned int          vma_exec : 1;
305         unsigned int          vma_huge : 1;
306         unsigned int          vma_pfn : 1;
307
308         unsigned int          batch_count;
309
310 #ifndef CONFIG_MMU_GATHER_NO_GATHER
311         struct mmu_gather_batch *active;
312         struct mmu_gather_batch local;
313         struct page             *__pages[MMU_GATHER_BUNDLE];
314
315 #ifdef CONFIG_MMU_GATHER_PAGE_SIZE
316         unsigned int page_size;
317 #endif
318 #endif
319 };

```

Listing 7-52: include/asm-generic/tlb.h: struct mmu_gather

This stores `mm`, a pointer to the struct `mm_struct` which describes the process address space whose TLB state we are modifying, the start and (exclusive) end of the range being operated upon in `start` and `end`.

MMU operations are divided into *batches* which are described by the struct `mmu_gather_batch` type which we will examine shortly. The total batch count is tracked by the `batch_count` field, with a pointer to the current batch stored in `active`.

The `local` and `__pages` fields overlap one another as the struct `mmu_gather_batch` object has a variable array as its final field and this forms the initial batch (further batches are allocated).

Finally there are a number of well-documented flags and a `struct mmu_table_batch` object referenced in the `batch` field used for freeing page tables. Page table size (to accommodate huge pages) is stored in the `page_size` field.

Whether we batch operations at all is determined by `CONFIG_MMU_GATHER_NO_GATHER`, whether we store variable page size is determined by `CONFIG_MMU_GATHER_PAGE_SIZE` and whether we batch page table TLB operations is determined by `CONFIG_MMU_GATHER_TABLE_FREE`. Modern architectures will typically specify all of these, and x86-64 certainly does so we will proceed on the assumption they are all set.

Examining `struct mmu_gather_batch` as shown in Listing 7-53.

```

241 struct mmu_gather_batch {
242     struct mmu_gather_batch *next;
243     unsigned int          nr;
244     unsigned int          max;
245     struct page           *pages[];
246 };

```

Listing 7-53: include/asm-generic/tlb.h: struct mmu_gather_batch

The batch consists of pages, an array of `struct page` objects which are undergoing TLB cache invalidation, which is specified as a variable length array placed immediately after the other fields. There are `nr` pages up to a maximum of `max` with the next batch, if any, pointed at by `next`.

For the initial local copy of this object (which will typically be on the stack) `max` is set to `MMU_GATHER_BUNDLE` (hardcoded to 8). Afterwards a base page is allocated for each new batch with these fields adjusted accordingly.

We examine `tlb_gather_mmu()` in Listing 7-54.

```

289 /**
290  * tlb_gather_mmu - initialize an mmu_gather structure for page-table tear-
291  *                 down
292  * @tlb: the mmu_gather structure to initialize
293  * @mm: the mm_struct of the target address space
294  *
295  * Called to initialize an (on-stack) mmu_gather structure for page-table
296  * tear-down from @mm.
297  */
298 void tlb_gather_mmu(struct mmu_gather *tlb, struct mm_struct *mm)
299 {
300     __tlb_gather_mmu(tlb, mm, false);

```

Listing 7-54: mm/mmu_gather.c: tlb_gather_mmu()

It's worth noting that there is an alternative to this function which performs this operation across the entirety of the process address space, `tlb_gather_mmu_fullmm()` as shown in Listing 7-55.

```

302 /**
303  * tlb_gather_mmu_fullmm - initialize an mmu_gather structure for page-table
    tear-down
304  * @tlb: the mmu_gather structure to initialize
305  * @mm: the mm_struct of the target address space
306  *
307  * In this case, @mm is without users and we're going to destroy the
308  * full address space (exit/execve).
309  *
310  * Called to initialize an (on-stack) mmu_gather structure for page-table
311  * tear-down from @mm.
312  */
313 void tlb_gather_mmu_fullmm(struct mmu_gather *tlb, struct mm_struct *mm)
314 {
315     __tlb_gather_mmu(tlb, mm, true);
316 }

```

Listing 7-55: mm/mmu_gather.c: tlb_gather_mmu_fullmm

These each differ only in whether they specify the fullmm boolean argument to `__tlb_gather_mmu()` which does the heavy lifting as shown in Listing 7-56.

```

265 static void __tlb_gather_mmu(struct mmu_gather *tlb, struct mm_struct *mm,
266                             bool fullmm)
267 {
268     tlb->mm = mm;
269     tlb->fullmm = fullmm;
270
271 #ifndef CONFIG_MMU_GATHER_NO_GATHER
272     tlb->need_flush_all = 0;
273     tlb->local.next = NULL;
274     tlb->local.nr = 0;
275     tlb->local.max = ARRAY_SIZE(tlb->__pages);
276     tlb->active = &tlb->local;
277     tlb->batch_count = 0;
278 #endif
279
280     tlb_table_init(tlb);
281 #ifdef CONFIG_MMU_GATHER_PAGE_SIZE
282     tlb->page_size = 0;
283 #endif
284
285     __tlb_reset_range(tlb);
286     inc_tlb_flush_pending(tlb->mm);
287 }

```

Listing 7-56: mm/mmu_gather.c: __tlb_gather_mmu()

This initialises all fields in the struct `mmu_gather` object, setting `active` to point to the local stack object before initialising it.

`tlb_table_init()` simply sets `batch` to `NULL`, `__tlb_reset_range()` resets flags and the start and end values to preset initial values as shown in Listing 7-57.

```

331 static inline void __tlb_reset_range(struct mmu_gather *tlb)
332 {
333     if (tlb->fullmm) {
334         tlb->start = tlb->end = ~0;
335     } else {
336         tlb->start = TASK_SIZE;
337         tlb->end = 0;
338     }
339     tlb->freed_tables = 0;
340     tlb->cleared_ptes = 0;
341     tlb->cleared_pmds = 0;
342     tlb->cleared_puds = 0;
343     tlb->cleared_p4ds = 0;
344     /*
345      * Do not reset mmu_gather::vma_* fields here, we do not
346      * call into tlb_start_vma() again to set them if there is an
347      * intermediate flush.
348      */
349 }
```

Listing 7-57: `include/asm-generic/tlb.h: __tlb_reset_range()`

Finally, we invoke the function `inc_tlb_flush_pending()` which increments the struct `mm_struct->tlb_flush_pending` atomic counter to indicate that a TLB flush is pending*. This value will be decremented via `dec_tlb_flush_pending()` once the flush operation is complete and is checked via `mm_tlb_flush_pending()` and `mm_tlb_flush_nested()` for the nested case.

7.1.3 VMA unmapping

Once we establish the struct `mmu_gather` object, ready to perform TLB operations upon it we do so via `unmap_vmas()`:-

```

1698 /**
1699  * unmap_vmas - unmap a range of memory covered by a list of vma's
1700  * @tlb: address of the caller's struct mmu_gather
1701  * @vma: the starting vma
1702  * @start_addr: virtual address at which to start unmapping
```

*. One of the primary place where this value is utilised is in the x86-64 specific `pte_accessible()` when determining whether an unmapped entry used in `ptep_clear_flush()`. This was introduced to avoid a data race with a concurrent NUMA balance, added in commit 20841405940e7: mm: fix TLB flush race between migration, and change_protection_range.

```

1703  * @end_addr: virtual address at which to end unmapping
1704  *
1705  * Unmap all pages in the vma list.
1706  *
1707  * Only addresses between `start' and `end' will be unmapped.
1708  *
1709  * The VMA list must be sorted in ascending virtual address order.
1710  *
1711  * unmap_vmas() assumes that the caller will flush the whole unmapped address
1712  * range after unmap_vmas() returns. So the only responsibility here is to
1713  * ensure that any thus-far unmapped pages are flushed before unmap_vmas()
1714  * drops the lock and schedules.
1715  */
1716 void unmap_vmas(struct mmu_gather *tlb,
1717                struct vm_area_struct *vma, unsigned long start_addr,
1718                unsigned long end_addr)
1719 {
1720     struct mmu_notifier_range range;
1721     struct zap_details details = {
1722         .zap_flags = ZAP_FLAG_DROP_MARKER,
1723         /* Careful - we need to zap private pages too! */
1724         .even_cows = true,
1725     };
1726
1727     mmu_notifier_range_init(&range, MMU_NOTIFY_UNMAP, 0, vma, vma->vm_mm,
1728                             start_addr, end_addr);
1729     mmu_notifier_invalidate_range_start(&range);
1730     for ( ; vma && vma->vm_start < end_addr; vma = vma->vm_next)
1731         unmap_single_vma(tlb, vma, start_addr, end_addr, &details);
1732     mmu_notifier_invalidate_range_end(&range);
1733 }

```

Listing 7-58: mm/memory.c: `unmap_vmas()`

The *MMU notifier* logic is out of scope, however briefly this provides a means by which interested parties can be notified of key memory management events used by, for example, virtualisation in order to manage memory for guests.

Other than this, we establish a struct `zap_details` state to be used when *zapping* the address range. Zapping pages in an address range removes their page table entries, decrements the underlying struct `folio->_mapcount` and performs related cleanup.

The `ZAP_FLAG_DROP_MARKER` flag is userfaultfd-specific and thus out of scope for the book. We additionally set the `even_cows` option as we are performing an unqualified unmapping in this instance.

Note that the input `vma` parameter is simply the *first* VMA in the range which we iterate through, passing each to `unmap_single_vma()` (eliding out of scope huge page and uprobe handling) as shown in Listing 7-59.

```

1652 static void unmap_single_vma(struct mmu_gather *tlb,
1653                             struct vm_area_struct *vma, unsigned long start_addr,
1654                             unsigned long end_addr,
1655                             struct zap_details *details)
1656 {
1657     unsigned long start = max(vma->vm_start, start_addr);
1658     unsigned long end;
1659
1660     if (start >= vma->vm_end)
1661         return;
1662     end = min(vma->vm_end, end_addr);
1663     if (end <= vma->vm_start)
1664         return;
1665     :
1669     if (unlikely(vma->vm_flags & VM_PFNMAP))
1670         untrack_pfn(vma, 0, 0);
1671
1672     if (start != end) {
1673         :
1694         unmap_page_range(tlb, vma, start, end, details);
1695     }
1696 }

```

Listing 7-59: mm/memory.c: unmap_single_vma()

The `untrack_pfn()` handling for PFN mappings is architecture-specific and out of scope for the book. Otherwise, after clamping start and end to be within the current VMA the per-VMA unmapping is forwarded to `unmap_page_range()`, which *zaps* the page range in the specified vma.

7.1.4 Zapping memory ranges

Examining `unmap_page_range()`:-

```

1631 void unmap_page_range(struct mmu_gather *tlb,
1632                      struct vm_area_struct *vma,
1633                      unsigned long addr, unsigned long end,
1634                      struct zap_details *details)
1635 {
1636     pgd_t *pgd;
1637     unsigned long next;
1638
1639     BUG_ON(addr >= end);
1640     tlb_start_vma(tlb, vma);
1641     pgd = pgd_offset(vma->vm_mm, addr);
1642     do {
1643         next = pgd_addr_end(addr, end);

```

```

1644         if (pgd_none_or_clear_bad(pgd))
1645             continue;
1646         next = zap_p4d_range(tlb, vma, pgd, addr, next, details);
1647     } while (pgd++, addr = next, addr != end);
1648     tlb_end_vma(tlb, vma);
1649 }

```

Listing 7-60: mm/memory.c: `unmap_page_range()`

This follows the typical page table walking pattern – obtaining a pointer to the PGD entry mapping `addr` via `pgd_offset()`, before finding the next PGD entry via `pgd_addr_end()`.

It differs a little in that it uses `pgd_none_or_clear_bad()` which checks whether the PGD entry is empty via `pgd_none()`, whether it is not as expected via `pgd_bad()` and in this instance clearing the entry. This makes sense in the instance of clearing entries, as given the entry is corrupted and we intend to clear at least part of the range it represents the best means of fixing it is to at least make it consistent in the portion whose desired state we know.

The next level in the page table hierarchy is handled by `zap_p4d_range()`, which we shall return to shortly.

In relation to the MMU gather logic, we can see here in practice two key functions – `tlb_start_vma()` and `tlb_end_vma()`.

These indicate that a TLB flush operation is about to begin and about to finish, respectively. Examining `tlb_start_vma()` as shown in Listing 7-61.

```

484  /*
485   * In the case of tlb vma handling, we can optimise these away in the
486   * case where we're doing a full MM flush. When we're doing a munmap,
487   * the vmas are adjusted to only cover the region to be torn down.
488   */
489  static inline void tlb_start_vma(struct mmu_gather *tlb, struct vm_area_struct
                                *vma)
490  {
491      if (tlb->fullmm)
492          return;
493
494      tlb_update_vma_flags(tlb, vma);
495  #ifndef CONFIG_MMU_GATHER_NO_FLUSH_CACHE
496      flush_cache_range(vma, vma->vm_start, vma->vm_end);
497  #endif
498  }

```

Listing 7-61: include/asm-generic/tlb.h: `tlb_start_vma()`

In the case that the `fullmm` flag has been specified, we are in any case performing this operation over the entire process address space and thus individual VMAs need not be considered and so in that case the function simply exits. Of course, in the context of the `unmap_region()` operation, this will *not* be set.

The `flush_cache_range()` function is called to ensure the TLB cache for the VMA range is consistent for those architectures which require manual cache maintenance operations, the focused upon architecture x86-64 does not require this.

In any case, `tlb_update_vma_flags()` is called which updates state in the struct `mmu_gather` object according to reflect that of the VMA as shown in Listing 7-62.

```

397 static inline void
398 tlb_update_vma_flags(struct mmu_gather *tlb, struct vm_area_struct *vma)
399 {
400     /*
401      * flush_tlb_range() implementations that look at VM_HUGETLB (tile,
402      * mips-4k) flush only large pages.
403      *
404      * flush_tlb_range() implementations that flush I-TLB also flush D-TLB
405      * (tile, xtensa, arm), so it's ok to just add VM_EXEC to an existing
406      * range.
407      *
408      * We rely on tlb_end_vma() to issue a flush, such that when we reset
409      * these values the batch is empty.
410      */
411     tlb->vma_huge = is_vm_hugetlb_page(vma);
412     tlb->vma_exec = !(vma->vm_flags & VM_EXEC);
413     tlb->vma_pfn = !(vma->vm_flags & (VM_PFNMAP|VM_MIXEDMAP));
414 }
```

Listing 7-62: `include/asm-generic/tlb.h: tlb_update_vma_flags()`

In order to keep the order of this analysis in line with the order in which steps actually occur in the kernel, we shall return to `tlb_end_vma()` at the point at which it would be invoked in a zap operation.

Returning to our traversal of the page tables as shown in Listing 7-63.

```

1612 static inline unsigned long zap_p4d_range(struct mmu_gather *tlb,
1613                                         struct vm_area_struct *vma, pgd_t *pgd,
1614                                         unsigned long addr, unsigned long end,
1615                                         struct zap_details *details)
1616 {
1617     p4d_t *p4d;
1618     unsigned long next;
1619
1620     p4d = p4d_offset(pgd, addr);
1621     do {
1622         next = p4d_addr_end(addr, end);
1623         if (p4d_none_or_clear_bad(p4d))
1624             continue;
1625         next = zap_pud_range(tlb, vma, p4d, addr, next, details);
1626     } while (p4d++, addr = next, addr != end);
```

```

1627
1628     return addr;
1629 }

```

Listing 7-63: mm/memory.c: zap_p4d_range()

This mirrors `unmap_page_range()` (see listing 7-60) except referencing the P4D equivalent functions – `p4d_offset()`, `p4d_addr_end()` and `p4d_none_or_clear_bad()`.

Examining the PUD equivalent (eliding out of scope huge page and device mapping handling) as shown in Listing 7-64.

```

1583 static inline unsigned long zap_pud_range(struct mmu_gather *tlb,
1584                                         struct vm_area_struct *vma, p4d_t *p4d,
1585                                         unsigned long addr, unsigned long end,
1586                                         struct zap_details *details)
1587 {
1588     pud_t *pud;
1589     unsigned long next;
1590
1591     pud = pud_offset(p4d, addr);
1592     do {
1593         next = pud_addr_end(addr, end);
1594         :
1602         if (pud_none_or_clear_bad(pud))
1603             continue;
1604         next = zap_pmd_range(tlb, vma, pud, addr, next, details);
1605         :
1607     } while (pud++, addr = next, addr != end);
1608
1609     return addr;
1610 }

```

Listing 7-64: mm/memory.c: zap_pud_range()

This resembles the two levels above, except referencing the PUD equivalent functions – `pud_offset()`, `pud_addr_end()` and `pud_none_or_clear_bad()`.

Examining the PMD equivalent (eliding out of scope swap, huge page and device mapping handling) as shown in Listing 7-65.

```

1537 static inline unsigned long zap_pmd_range(struct mmu_gather *tlb,
1538                                         struct vm_area_struct *vma, pud_t *pud,
1539                                         unsigned long addr, unsigned long end,
1540                                         struct zap_details *details)
1541 {
1542     pmd_t *pmd;
1543     unsigned long next;
1544
1545     pmd = pmd_offset(pud, addr);
1546     do {

```

```

1547         next = pmd_addr_end(addr, end);
1548         :
1549
1550         /*
1551          * Here there can be other concurrent MADV_DONTNEED or
1552          * trans huge page faults running, and if the pmd is
1553          * none or trans huge it can change under us. This is
1554          * because MADV_DONTNEED holds the mmap_lock in read
1555          * mode.
1556          */
1557         if (pmd_none_or_trans_huge_or_clear_bad(pmd))
1558             goto next;
1559         next = zap_pte_range(tlb, vma, pmd, addr, next, details);
1560 next:
1561         :
1562     } while (pmd++, addr = next, addr != end);
1563
1564     return addr;
1565 }

```

Listing 7-65: mm/memory.c: zap_pmd_range()

This maintains much of the logic as seen previously, differing in that `pmd_none_or_trans_huge_or_clear_bad()` incorporates huge page logic, out of scope for this section. Otherwise the `pmd_offset()` and `pmd_addr_end()` functions follow the established pattern.

Finally we reach the key function which performs the operation, `zap_pte_range()` (eliding out of scope swap, migration, userfaultfd and hardware poisoning logic) as shown in Listing 7-66.

```

1402 static unsigned long zap_pte_range(struct mmu_gather *tlb,
1403                                   struct vm_area_struct *vma, pmd_t *pmd,
1404                                   unsigned long addr, unsigned long end,
1405                                   struct zap_details *details)
1406 {
1407     struct mm_struct *mm = tlb->mm;
1408     int force_flush = 0;
1409     int rss[NR_MM_COUNTERS];
1410     spinlock_t *ptl;
1411     pte_t *start_pte;
1412     pte_t *pte;
1413     swp_entry_t entry;
1414
1415     tlb_change_page_size(tlb, PAGE_SIZE);
1416 again:
1417     init_rss_vec(rss);
1418     start_pte = pte_offset_map_lock(mm, pmd, addr, &ptl);
1419     pte = start_pte;
1420     flush_tlb_batched_pending(mm);
1421     arch_enter_lazy_mmu_mode();

```

Listing 7-66: mm/memory.c: zap_pte_range() preface

We start by setting the page size to PAGE_SIZE via `tlb_change_page_size()`, which is relevant only to huge page handling.

The rss array is zeroed via `init_rss_vec()` and the PTE entry associated with the beginning address `addr` is assigned to `start_pte` via `pte_offset_map_lock()` while also acquiring the PTE lock. This is also assigned to `pte`.

Finally two interesting functions are called here – `arch_enter_lazy_mmu_mode()` which is relevant only to x86 paravirtualisation and the powerpc and sparc architectures (and thus out of scope), and `flush_tlb_batched_pending()` as shown in Listing 7-67.

```

706 /*
707  * Reclaim unmmaps pages under the PTL but do not flush the TLB prior to
708  * releasing the PTL if TLB flushes are batched. It's possible for a parallel
709  * operation such as mprotect or munmap to race between reclaim unmapping
710  * the page and flushing the page. If this race occurs, it potentially allows
711  * access to data via a stale TLB entry. Tracking all mm's that have TLB
712  * batching in flight would be expensive during reclaim so instead track
713  * whether TLB batching occurred in the past and if so then do a flush here
714  * if required. This will cost one additional flush per reclaim cycle paid
715  * by the first operation at risk such as mprotect and mmap.
716  *
717  * This must be called under the PTL so that an access to tlb_flush_batched
718  * that is potentially a "reclaim vs mprotect/munmap/etc" race will
719  *   synchronise
720  *   via the PTL.
721  */
722 void flush_tlb_batched_pending(struct mm_struct *mm)
723 {
724     int batch = atomic_read(&mm->tlb_flush_batched);
725     int pending = batch & TLB_FLUSH_BATCH_PENDING_MASK;
726     int flushed = batch >> TLB_FLUSH_BATCH_FLUSHED_SHIFT;
727
728     if (pending != flushed) {
729         flush_tlb_mm(mm);
730         /*
731          * If the new TLB flushing is pending during flushing, leave
732          * mm->tlb_flush_batched as is, to avoid losing flushing.
733          */
734         atomic_cmpxchg(&mm->tlb_flush_batched, batch,
735                        pending | (pending <<
736                                TLB_FLUSH_BATCH_FLUSHED_SHIFT));
737     }
738 }
```

Listing 7-67: mm/rmap.c: flush_tlb_batched_pending()

This function, as its name implies, flushes any pending batched TLB operations. This is tracked via the struct `mm_struct->tlb_flush_batched` field.

This field is used to track pending and flushed TLB operations used during reclaim via the reverse mapping in `try_to_unmap()`. We shall explore this in detail in the reclaim chapter. The actual flush is performed via `flush_tlb_mm()` which performs a TLB flush across the entire process address space. This ultimately defers the operation to the architecture-specific `flush_tlb_mm_range()` which we will examine shortly.

Returning to `zap_pte_range()` we enter its main loop (eliding out of scope `userfaultfd`, `swap`, `migration` and `hardware poison` logic) as shown in Listing ??.

```

1422         do {
1423             pte_t ptent = *pte;
1424             struct page *page;
1425
1426             if (pte_none(ptent))
1427                 continue;
1428
1429             if (need_resched())
1430                 break;
1431
1432             if (pte_present(ptent)) {
1433                 page = vm_normal_page(vma, addr, ptent);
1434                 if (unlikely(!should_zap_page(details, page)))
1435                     continue;
1436                 ptent = ptep_get_and_clear_full(mm, addr, pte,
1437                                                 tlb->fullmm);
1438                 tlb_remove_tlb_entry(tlb, pte, addr);
1439                 :
1441                 if (unlikely(!page))
1442                     continue;
1443
1444                 if (!PageAnon(page)) {
1445                     if (pte_dirty(ptent)) {
1446                         force_flush = 1;
1447                         set_page_dirty(page);
1448                     }
1449                     if (pte_young(ptent) &&
1450                         likely(!(vma->vm_flags & VM_SEQ_READ)))
1451                         mark_page_accessed(page);
1452                 }
1453                 rss[mm_counter(page)]--;
1454                 page_remove_rmap(page, vma, false);
1455                 if (unlikely(page_mapcount(page) < 0))

```

```

1456             print_bad_pte(vma, addr, pte, page);
1457             if (unlikely(__tlb_remove_page(tlb, page))) {
1458                 force_flush = 1;
1459                 addr += PAGE_SIZE;
1460                 break;
1461             }
1462             continue;
1463         }
1464         :
1508     } while (pte++, addr += PAGE_SIZE, addr != end);

```

Listing 7-68: mm/memory.c: zap_pte_range() main loop

The top-level logic here checks whether the PTE entry is empty via `pte_none()` or non-present via `pte_present()` – if either of these are the case then no action need be taken. This logic additionally calls `need_resched()` to determine whether rescheduling would be appropriate (breaking out of the loop to do so) as the unmapping can be a lengthily operation.

The page associated with the mapping is obtained via `vm_normal_page()` (see section 6.12 for details on this and special mappings for which will return `NULL`).

We next call `should_zap_page()` to determine whether the struct page object can be zapped as shown in Listing 7-69.

```

1364 /* Decides whether we should zap this page with the page pointer specified */
1365 static inline bool should_zap_page(struct zap_details *details, struct page *
1366     page)
1367 {
1368     /* If we can make a decision without *page.. */
1369     if (should_zap_cows(details))
1370         return true;
1371
1372     /* E.g. the caller passes NULL for the case of a zero page */
1373     if (!page)
1374         return true;
1375
1376     /* Otherwise we should only zap non-anon pages */
1377     return !PageAnon(page);

```

Listing 7-69: mm/memory.c: should_zap_page()

This primarily determines whether this is a Copy On Write (i.e. anonymous) page which, unless the struct `zap_details->even_cows` flag is set (checked by `should_zap_cows()` – note that if no details are provided this defaults to doing so), is not performed.

The next step is subtle – `ptep_get_and_clear_full()` re-retrieves the PTE and clears the PTE entry. This is important as, prior to the TLB operation, we must ensure that page table entries do not point to the page whose entry

we are clearing, otherwise the TLB could immediately be repopulated with the stale entry.

The TLB entry is marked removed via `tlb_remove_tlb_entry()` as shown in Listing 7-70.

```

556 /**
557  * tlb_remove_tlb_entry - remember a pte unmapping for later tlb invalidation.
558  *
559  * Record the fact that pte's were really unmapped by updating the range,
560  * so we can later optimise away the tlb invalidate. This helps when
561  * userspace is unmapping already-unmapped pages, which happens quite a lot.
562  */
563 #define tlb_remove_tlb_entry(tlb, ptep, address)          \
564     do {                                                  \
565         tlb_flush_pte_range(tlb, address, PAGE_SIZE);    \
566         __tlb_remove_tlb_entry(tlb, ptep, address);      \
567     } while (0)

```

Listing 7-70: `include/asm-generic/tlb.h: tlb_remove_tlb_entry()`

The `__tlb_remove_tlb_entry()` function is used for architectures which require manual TLB cache maintenance to indicate that an entry is to be removed, however this is a no-op for x86-64.

Examining `tlb_flush_pte_range()` as shown in Listing 7-71.

```

520 /*
521  * tlb_flush_{pte/pmd/pud/p4d}_range() adjust the tlb->start and tlb->end,
522  * and set corresponding cleared_*.
523  */
524 static inline void tlb_flush_pte_range(struct mmu_gather *tlb,
525                                       unsigned long address, unsigned long size
526                                       )
527 {
528     __tlb_adjust_range(tlb, address, size);
529     tlb->cleared_ptes = 1;
530 }

```

Listing 7-71: `include/asm-generic/tlb.h: tlb_flush_pte_range()`

This marks that PTEs have been cleared and updates the `struct mmu_gather` range parameters via `__tlb_adjust_range()` as shown in Listing 7-72.

```

323 static inline void __tlb_adjust_range(struct mmu_gather *tlb,
324                                     unsigned long address,
325                                     unsigned int range_size)
326 {
327     tlb->start = min(tlb->start, address);
328     tlb->end = max(tlb->end, address + range_size);
329 }

```

Listing 7-72: include/asm-generic/tlb.h: `__tlb_adjust_range()`

As we walk through each PTE entry we adjust the start and end values in order that at the end of the walk these are equal to a range that spans all mappings to be removed (and perhaps some gaps).

At this point we check to see whether we are dealing with a special mapping – i.e. one for which a struct `page` does not exist (again see section 6.12 for more details on this), if so we have no further action to perform. Otherwise we process the page object:-

- If the page is not anonymous (i.e. not `PageAnon()`):-
 - Take special care to ensure the underlying folio is correctly marked dirty so dirty pages are guaranteed to be written back correctly, via `set_page_dirty()`, ultimately invoking `folio_mark_dirty()` (see the page cache chapter for more details on this).
The `force_flush` flag is set to ensure that a TLB flush occurs *immediately* under the PTE lock rather than being batched up, as otherwise a racing write back could try to flush the TLB, find an empty PTE entry and not correctly wait for stale TLB entries to be invalidated before starting write back.
 - Equally, if the page has been recently accessed as indicated by the `_PAGE_ACCESSED` page table flag as checked by `pte_young()`, and the `VM_SEQ_READ` flag has not been set on the VMA, indicating that this page may indeed be accessed again soon, albeit not from this mapping, then the underlying folio is updated via `mark_page_accessed()` (which ultimately calls `folio_mark_accessed()`).

After this the `rss` field containing statistics for various different types of pages is updated, using `mm_counter()` to identify which statistic to decrement.

The page is marked for removal from the reverse mapping via `page_remove_rmap()`, which is very important for tearing down the link between the virtual mapping and thus PTE now being unmapped and the underlying page (eliding out of scope cgroup and huge page logic) as shown in Listing 7-73.

```

1421 /**
1422  * page_remove_rmap - take down pte mapping from a page
1423  * @page:           page to remove mapping from
1424  * @vma:            the vm area from which the mapping is removed
1425  * @compound:       uncharge the page as compound or small page
1426  *
1427  * The caller needs to hold the pte lock.
1428  */
1429 void page_remove_rmap(struct page *page,
1430                      struct vm_area_struct *vma, bool compound)
1431 {
1432     :
1433     :
1434     if (!PageAnon(page)) {
```

```

1435         page_remove_file_rmap(page, compound);
1436         goto out;
1437     }
1438     :
1444     /* page still mapped by someone else? */
1445     if (!atomic_add_negative(-1, &page->_mapcount))
1446         goto out;
1447
1448     /*
1449     * We use the irq-unsafe __{inc/mod}_zone_page_stat because
1450     * these counters are not modified in interrupt context, and
1451     * pte lock(a spinlock) is held, which implies preemption disabled.
1452     */
1453     __dec_lruvec_page_state(page, NR_ANON_MAPPED);
1454     :
1458     /*
1459     * It would be tidy to reset the PageAnon mapping here,
1460     * but that might overwrite a racing page_add_anon_rmap
1461     * which increments mapcount after us but sets mapping
1462     * before us: so leave the reset to free_unref_page,
1463     * and remember that it's only reliable while mapped.
1464     * Leaving it set also helps swapoff to reinstate ptes
1465     * faster for those pages still in swapcache.
1466     */
1467 out:
1468     :
1470     munlock_vma_page(page, vma, compound);
1471 }

```

Listing 7-73: mm/rmap.c: `page_remove_rmap()`

It's important to note that in actual fact this does *not* clear the mapped field at this stage. Note that the *reverse* mapping by its nature does not provide a data structure connecting VMAs to pages (this must be determined by walking page tables), but rather folios are linked to either page cache entries or a struct anon_vma object via their mapping fields.

The key thing here is that the struct folio->_mapcount field is decremented. Importantly, this field is initialised to -1 when first allocated, such that 0 indicates it is mapped by a single mapping, 1 by 2, etc., so this atomic_add_negative() invocation, which subtracts 1 and returns true if the number is now negative. If not, then the page is mapped by somebody else.

If this is not the case, we update statistics and unlock the page via munlock_vma_page() if it was previously locked via mlock().

If the page was not in fact an anonymous one, page_remove_file_rmap() is instead invoked (eliding out of scope huge page logic) as shown in Listing 7-74.

```

1342 static void page_remove_file_rmap(struct page *page, bool compound)
1343 {

```

```

1344         int i, nr = 0;
1345         :
1372         if (atomic_add_negative(-1, &page->_mapcount))
1373             nr++;
1374         :
1376         if (nr)
1377             __mod_lruvec_page_state(page, NR_FILE_MAPPED, -nr);
1378     }

```

Listing 7-74: mm/rmap.c: page_remove_file_rmap()

Which effectively does the same thing.

Returning to zap_pte_range() (listing 7-68), after checking that the mapcount is valid (note that page_mapcount() returns the _mapcount field incremented by 1), __tlb_remove_page() is invoked to action the removal of the page. If this fails then a full flush is forced and the loop is aborted.

The __tlb_remove_page() function invokes __tlb_remove_page_size() in turn, passing PAGE_SIZE as the page_size argument.

This adds the page to the active batch struct mmu_gather_batch object as shown in Listing 7-75.

```

79 bool __tlb_remove_page_size(struct mmu_gather *tlb, struct page *page, int
    page_size)
80 {
81     struct mmu_gather_batch *batch;
82
83     VM_BUG_ON(!tlb->end);
84
85 #ifdef CONFIG_MMU_GATHER_PAGE_SIZE
86     VM_WARN_ON(tlb->page_size != page_size);
87 #endif
88
89     batch = tlb->active;
90     /*
91      * Add the page and check if we are full. If so
92      * force a flush.
93      */
94     batch->pages[batch->nr++] = page;
95     if (batch->nr == batch->max) {
96         if (!tlb_next_batch(tlb))
97             return true;
98         batch = tlb->active;
99     }
100     VM_BUG_ON_PAGE(batch->nr > batch->max, page);
101
102     return false;
103 }

```

Listing 7-75: mm/mmu_gather.c: __tlb_remove_page_size()

We simply add the page to the active batch's pages array, then checks to see whether the batch is full, if so `tlb_next_batch()` is invoked to obtain the next one, if it can, indicating whether it succeeded or not.

Examining `tlb_next_batch()` as shown in Listing 7-76.

```

17 static bool tlb_next_batch(struct mmu_gather *tlb)
18 {
19     struct mmu_gather_batch *batch;
20
21     batch = tlb->active;
22     if (batch->next) {
23         tlb->active = batch->next;
24         return true;
25     }
26
27     if (tlb->batch_count == MAX_GATHER_BATCH_COUNT)
28         return false;
29
30     batch = (void *)__get_free_pages(GFP_NOWAIT | __GFP_NOWARN, 0);
31     if (!batch)
32         return false;
33
34     tlb->batch_count++;
35     batch->next = NULL;
36     batch->nr = 0;
37     batch->max = MAX_GATHER_BATCH;
38
39     tlb->active->next = batch;
40     tlb->active = batch;
41
42     return true;
43 }

```

Listing 7-76: `mm/mmu_gather.c: tlb_next_batch()`

This checks whether the batch already has a next batch object available, activating this, whether the `struct mmu_gather` object's `batch_count` exceeds `MAX_GATHER_BATCH_COUNT` which is the number of batches containing 10,000 pages (19 for x86-64).

The maximum number of pages in each batch is determined by the size remaining in a page allocated to store a batch and the array of pages, `MAX_GATHER_BATCH`, which on x86-64 is equal to 509.

Returning to `zap_pte_range()` (listing 7-68) for the last time, where the `zap_pte_range()` operation is finalised as shown in Listing 7-77.

```

1510     add_mm_rss_vec(mm, rss);
1511     arch_leave_lazy_mmu_mode();
1512
1513     /* Do the actual TLB flush before dropping ptl */

```

```

1514     if (force_flush)
1515         tlb_flush_mmu_tlbonly(tlb);
1516     pte_unmap_unlock(start_pte, ptl);
1517
1518     /*
1519      * If we forced a TLB flush (either due to running out of
1520      * batch buffers or because we needed to flush dirty TLB
1521      * entries before releasing the ptl), free the batched
1522      * memory too. Restart if we didn't do everything.
1523      */
1524     if (force_flush) {
1525         force_flush = 0;
1526         tlb_flush_mmu(tlb);
1527     }
1528
1529     if (addr != end) {
1530         cond_resched();
1531         goto again;
1532     }
1533
1534     return addr;
1535 }

```

Listing 7-77: mm/memory.c: zap_pte_range() finalisation

Firstly, this registers RSS statistics changes via `add_mm_rss_vec()`, exits architecture-specific lazy MMU mode via `arch_leave_lazy_mmu_mode()` (only relevant for SPARC).

If we must force the flush, `tlb_flush_mmu_tlbonly()` is called (we return to this in section 7.1.6 below) is called prior to releasing the PTE lock via `pte_unmap_unlock()` to avoid a race with a write back (discussed above). Note this *only* flushes the range, it does not free underlying pages.

Finally, after the lock is released, a flush is performed *and* pages are freed via `tlb_flush_mmu()`.

If we had to break out of the loop, we conditionally reschedule in case any more urgent task need be performed via `cond_resched()` and loop to the `again` label.

Returning to `unmap_page_range()`, this invokes `tlb_end_vma()` after each VMA has been flushed as shown in Listing 7-78.

```

500 static inline void tlb_end_vma(struct mmu_gather *tlb, struct vm_area_struct *
      vma)
501 {
502     if (tlb->fullmm)
503         return;
504
505     /*
506      * VM_PFNMAP is more fragile because the core mm will not track the

```

```

507      * page mapcount -- there might not be page-frames for these PFNs
      after
508      * all. Force flush TLBs for such ranges to avoid munmap() vs
509      * unmap_mapping_range() races.
510      */
511      if (tlb->vma_pfn || !IS_ENABLED(CONFIG_MMU_GATHER_MERGE_VMAS)) {
512          /*
513          * Do a TLB flush and reset the range at VMA boundaries; this
      avoids
514          * the ranges growing with the unused space between
      consecutive VMAs.
515          */
516          tlb_flush_mmu_tlbonly(tlb);
517      }
518 }

```

Listing 7-78: include/asm-generic/tlb.h: `tlb_end_vma()`

This takes special care to avoid races for PFN only mappings (see section 6.12 for more on this).

Finally, `unmap_region()` invokes `free_pgtables()` to clear page tables (see section 7.1.5 below) and then `tlb_finish_mmu()` to finish the batched TLB operation as shown in Listing 7-79.

```

318 /**
319  * tlb_finish_mmu - finish an mmu_gather structure
320  * @tlb: the mmu_gather structure to finish
321  *
322  * Called at the end of the shutdown operation to free up any resources that
323  * were required.
324  */
325 void tlb_finish_mmu(struct mmu_gather *tlb)
326 {
327     /*
328     * If there are parallel threads are doing PTE changes on same range
329     * under non-exclusive lock (e.g., mmap_lock read-side) but defer TLB
330     * flush by batching, one thread may end up seeing inconsistent PTEs
331     * and result in having stale TLB entries. So flush TLB forcefully
332     * if we detect parallel PTE batching threads.
333     *
334     * However, some syscalls, e.g. munmap(), may free page tables, this
335     * needs force flush everything in the given range. Otherwise this
336     * may result in having stale TLB entries for some architectures,
337     * e.g. aarch64, that could specify flush what level TLB.
338     */
339     if (mm_tlb_flush_nested(tlb->mm)) {
340         /*
341         * The aarch64 yields better performance with fullmm by
342         * avoiding multiple CPUs spamming TLBI messages at the

```

```

343         * same time.
344         *
345         * On x86 non-fullmm doesn't yield significant difference
346         * against fullmm.
347         */
348         tlb->fullmm = 1;
349         __tlb_reset_range(tlb);
350         tlb->freed_tables = 1;
351     }
352
353     tlb_flush_mmu(tlb);
354
355 #ifndef CONFIG_MMU_GATHER_NO_GATHER
356     tlb_batch_list_free(tlb);
357 #endif
358     dec_tlb_flush_pending(tlb->mm);
359 }

```

Listing 7-79: mm/mmu_gather.c: tlb_finish_mmu()

This checks whether another TLB flush has been initiated by calling the predicate function `mm_tlb_flush_nested()` (which simply checks whether `struct mm_struct->tlb_flush_pending` is greater than 1) in which case it switches to a flush of the entire process address space.

The actual flush and freeing of memory is performed via `tlb_flush_mmu()` which we discuss in section 7.1.6.

After this is done, the batch list is freed via `tlb_batch_list_free()`, and finally the number of concurrent TLB flushes is decremented via `dec_tlb_flush_pending()`.

7.1.5 Freeing page tables

Page tables are freed via `free_pgtables()` (eliding out of scope huge page logic):-

```

405 void free_pgtables(struct mmu_gather *tlb, struct vm_area_struct *vma,
406                  unsigned long floor, unsigned long ceiling)
407 {
408     while (vma) {
409         struct vm_area_struct *next = vma->vm_next;
410         unsigned long addr = vma->vm_start;
411
412         /*
413          * Hide vma from rmap and truncate_pagecache before freeing
414          * pgtables
415          */
416         unlink_anon_vmas(vma);
417         unlink_file_vma(vma);

```

```

:
423      /*
424      * Optimization: gather nearby vmAs into one call down
425      */
426      while (next && next->vm_start <= vma->vm_end +
            PMD_SIZE
427              && !is_vm_hugetlb_page(next)) {
428          vma = next;
429          next = vma->vm_next;
430          unlink_anon_vmas(vma);
431          unlink_file_vma(vma);
432      }
433      free_pgd_range(tlb, addr, vma->vm_end,
434                    floor, next ? next->vm_start : ceiling);
:
436      vma = next;
437  }
438 }

```

Listing 7-80: mm/memory.c: free_pgtables()

This both removes page tables and unlinks reverse mapping via `unlink_anon_vmas()` (see section 7.0.14 for a description of this) and file mapping objects via `unlink_file_vma()` as shown in Listing 7-81.

```

118 /*
119  * Unlink a file-based vm structure from its interval tree, to hide
120  * vma from rmap and vmtruncate before freeing its page tables.
121  */
122 void unlink_file_vma(struct vm_area_struct *vma)
123 {
124     struct file *file = vma->vm_file;
125
126     if (file) {
127         struct address_space *mapping = file->f_mapping;
128         i_mmap_lock_write(mapping);
129         __remove_shared_vm_struct(vma, file, mapping);
130         i_mmap_unlock_write(mapping);
131     }
132 }

```

Listing 7-81: mm/mmap.c: unlink_file_vma()

This removes the VMA object from the struct `address_space`'s interval tree via `__remove_shared_vm_struct()` (eliding out of scope manual cache maintenance logic irrelevant to x86-64) as shown in Listing 7-82.

```

104 /*
105  * Requires inode->i_mapping->i_mmap_rwsem
106  */

```



```

107 static void __remove_shared_vm_struct(struct vm_area_struct *vma,
108                                     struct file *file, struct address_space *mapping)
109 {
110     if (vma->vm_flags & VM_SHARED)
111         mapping_unmap_writable(mapping);
112     :
114     vma_interval_tree_remove(vma, &mapping->i_mmap);
115     :
116 }

```

Listing 7-82: mm/mmap.c: __remove_shared_vm_struct()

This atomically decrements the struct address_space->i_mmap_writable counter via mapping_unmap_writable() if the mapping is shared, to indicate that a potentially writable page cache object has been unmapped. In any case it removes the VMA from the page cache object's interval tree.

It's important to take a step back and to consider what we are doing here – we are removing these reverse mapping links from *all* the VMAs at and beyond the one provided to the function (which is the first in the range of VMAS to unmap).

This seems as if it would affect unrelated VMAs, however the call assumes implicitly that the VMAs being processed here, with vma being the first, have already been split and all VMAs in the range span only the range being freed and are not attached to the other VMAs in the process address space.

For an unmapping, unmap_region() is called from __do_munmap() which performs this detachment via detach_vmas_to_be_unmapped() (see Chapter 5 for more on memory mapping in general).

As a result, we can optimise the operation and batch the page table removal and reverse mapping unlinking for each VMA.

Note that, since we have already cleared the PTE entry for each mapping, we need not concern ourselves with this task. The granularity at which we free page tables is always at the PMD level, as each PMD entry spans an entire PTE directory page, e.g. on x86-64, 2 MiB of mappings. Only when we know for sure we can free such a range do we free the underlying PTE directory.

We pass the PMD-level range of addresses to free_pgd_range() (eliding huge page logic) as shown in Listing 7-83.

```

340 /*
341  * This function frees user-level page tables of a process.
342  */
343 void free_pgd_range(struct mmu_gather *tlb,
344                   unsigned long addr, unsigned long end,
345                   unsigned long floor, unsigned long ceiling)
346 {
347     pgd_t *pgd;
348     unsigned long next;
349

```

```

350      /*
351       * The next few lines have given us lots of grief...
352       *
353       * Why are we testing PMD* at this top level? Because often
354       * there will be no work to do at all, and we'd prefer not to
355       * go all the way down to the bottom just to discover that.
356       *
357       * Why all these "- 1"s? Because 0 represents both the bottom
358       * of the address space and the top of it (using -1 for the
359       * top wouldn't help much: the masks would do the wrong thing).
360       * The rule is that addr 0 and floor 0 refer to the bottom of
361       * the address space, but end 0 and ceiling 0 refer to the top
362       * Comparisons need to use "end - 1" and "ceiling - 1" (though
363       * that end 0 case should be mythical).
364       *
365       * Wherever addr is brought up or ceiling brought down, we must
366       * be careful to reject "the opposite 0" before it confuses the
367       * subsequent tests. But what about where end is brought down
368       * by PMD_SIZE below? no, end can't go down to 0 there.
369       *
370       * Whereas we round start (addr) and ceiling down, by different
371       * masks at different levels, in order to test whether a table
372       * now has no other vmas using it, so can be freed, we don't
373       * bother to round floor or end up - the tests don't need that.
374       */
375
376      addr &= PMD_MASK;
377      if (addr < floor) {
378          addr += PMD_SIZE;
379          if (!addr)
380              return;
381      }
382      if (ceiling) {
383          ceiling &= PMD_MASK;
384          if (!ceiling)
385              return;
386      }
387      if (end - 1 > ceiling - 1)
388          end -= PMD_SIZE;
389      if (addr > end - 1)
390          return;
391      :
396      pgd = pgd_offset(tlb->mm, addr);
397      do {
398          next = pgd_addr_end(addr, end);
399          if (pgd_none_or_clear_bad(pgd))
400              continue;

```

```

401         free_p4d_range(tlb, pgd, addr, next, floor, ceiling);
402     } while (pgd++, addr = next, addr != end);
403 }

```

Listing 7-83: mm/memory.c: free_pgd_range()

The initial operations are quite intricate as the very large comment suggests:-

- We perform a test to ensure the range is valid at a PMD granularity (which notably is the granularity at which the page table unmap operations are being performed in any case) here rather than later to quickly exit no-op cases.
- We denote the bottom *and* top of the address range by zero, therefore in order to perform correct comparisons, we must subtract 1 from the upper bounds, therefore all such comparisons to end and ceiling must be decremented by one.
- When adding PMD_SIZE to addr in the addr < floor branch, we explicitly check for overflow and return if it occurs. This would suggest addr is less than the floor but the PMD-aligned floor is at the end of the address space.
- If the ceiling, aligned to PMD size (and not 0 indicating the top of the address space), would be equal to zero, then we do not span at least a PTE directory's worth of mappings, and thus can't be sure it is safe to free at the PMD level, so must abort.
- If the now PMD-aligned ceiling would exceed the end value, this means we must round end down to keep it below ceiling. If this would result in addr no longer being in range, we exit.

After this, we perform a fairly typical page table walk, deferring the next level of the operation to free_p4d_range(). Note that the entry of the level above is cleared at the level below, i.e. it is the P4D freeing function that will clear the PGD entry, the PUD freeing function that will clear the P4D entry etc.

```

307 static inline void free_p4d_range(struct mmu_gather *tlb, pgd_t *pgd,
308                                unsigned long addr, unsigned long end,
309                                unsigned long floor, unsigned long ceiling)
310 {
311     p4d_t *p4d;
312     unsigned long next;
313     unsigned long start;
314
315     start = addr;
316     p4d = p4d_offset(pgd, addr);
317     do {
318         next = p4d_addr_end(addr, end);
319         if (p4d_none_or_clear_bad(p4d))

```

```

320             continue;
321             free_pud_range(tlb, p4d, addr, next, floor, ceiling);
322     } while (p4d++, addr = next, addr != end);
323
324     start &= PGDIR_MASK;
325     if (start < floor)
326         return;
327     if (ceiling) {
328         ceiling &= PGDIR_MASK;
329         if (!ceiling)
330             return;
331     }
332     if (end - 1 > ceiling - 1)
333         return;
334
335     p4d = p4d_offset(pgd, start);
336     pgd_clear(pgd);
337     p4d_free_tlb(tlb, p4d, start);
338 }

```

Listing 7-84: mm/memory.c: *free_p4d_range()*

This begins by performing page table freeing at the PUD level, *before* clearing the containing PGD entry. This is achieved by obtaining the P4D entry via `p4d_offset()`, clearing the PGD entry via `pgd_clear()` before performing a TLB state update via `p4d_free_tlb()`. Note this only occurs if the input range spans a P4D.

This sets the struct `mmu_gather->freed_tables` flag, adjusts the range to encompass the entire P4D entry as well as performing architecture-specific operations via `__p4d_free_tlb()` (which for x86-64 simply informs any para-virtualised host that this has occurred).

Examining the PUD level as shown in Listing 7-85.

```

273 static inline void free_pud_range(struct mmu_gather *tlb, p4d_t *p4d,
274                                 unsigned long addr, unsigned long end,
275                                 unsigned long floor, unsigned long ceiling)
276 {
277     pud_t *pud;
278     unsigned long next;
279     unsigned long start;
280
281     start = addr;
282     pud = pud_offset(p4d, addr);
283     do {
284         next = pud_addr_end(addr, end);
285         if (pud_none_or_clear_bad(pud))
286             continue;
287         free_pmd_range(tlb, pud, addr, next, floor, ceiling);
288     } while (pud++, addr = next, addr != end);

```

```

289
290     start &= P4D_MASK;
291     if (start < floor)
292         return;
293     if (ceiling) {
294         ceiling &= P4D_MASK;
295         if (!ceiling)
296             return;
297     }
298     if (end - 1 > ceiling - 1)
299         return;
300
301     pud = pud_offset(p4d, start);
302     p4d_clear(p4d);
303     pud_free_tlb(tlb, pud, start);
304     mm_dec_nr_puds(tlb->mm);
305 }

```

Listing 7-85: mm/memory.c: free_pud_range()

This is almost entirely identical to the P4D level, only differing in that it invokes `mm_dec_nr_puds()` which decrements the `struct mm_struct->pgtables_bytes` field accordingly.

At the PMD level as shown in Listing 7-86.

```

239 static inline void free_pmd_range(struct mmu_gather *tlb, pud_t *pud,
240                                 unsigned long addr, unsigned long end,
241                                 unsigned long floor, unsigned long ceiling)
242 {
243     pmd_t *pmd;
244     unsigned long next;
245     unsigned long start;
246
247     start = addr;
248     pmd = pmd_offset(pud, addr);
249     do {
250         next = pmd_addr_end(addr, end);
251         if (pmd_none_or_clear_bad(pmd))
252             continue;
253         free_pte_range(tlb, pmd, addr);
254     } while (pmd++, addr = next, addr != end);
255
256     start &= PUD_MASK;
257     if (start < floor)
258         return;
259     if (ceiling) {
260         ceiling &= PUD_MASK;
261         if (!ceiling)
262             return;

```

```

263     }
264     if (end - 1 > ceiling - 1)
265         return;
266
267     pmd = pmd_offset(pud, start);
268     pud_clear(pud);
269     pmd_free_tlb(tlb, pmd, start);
270     mm_dec_nr_pmds(tlb->mm);
271 }

```

Listing 7-86: mm/memory.c: free_pmd_range()

This again follows the same pattern, invoking `mm_dec_nr_pmds()` to update `struct mm_struct->pgtables_bytes`.

The `__pmd_free_tlb()` function differs here in that, in addition to paravirtualisation calls, it also ultimately invokes `__pmd_free_tlb()` which handles some architecture-specific logic as well as invoking the PMD page table destructor `pgtable_pmd_page_dtor()` which frees locks updates statistics and removes the `PG_table` flag from the underlying folio.

Finally the PTE function as shown in Listing 7-87.

```

226 /*
227  * Note: this doesn't free the actual pages themselves. That
228  * has been handled earlier when unmapping all the memory regions.
229  */
230 static void free_pte_range(struct mmu_gather *tlb, pmd_t *pmd,
231                          unsigned long addr)
232 {
233     pgtable_t token = pmd_pgtable(*pmd);
234     pmd_clear(pmd);
235     pte_free_tlb(tlb, token, addr);
236     mm_dec_nr_ptes(tlb->mm);
237 }

```

Listing 7-87: mm/memory.c: free_pte_range()

This is significantly less involved, as we have already established that we clear at the PMD granularity so no checks are required for this. `pte_free_tlb()` marks `struct mmu_gather->freed_tables` and adjusting the `tlb` range as with the other levels.

It also in turn invokes `__pte_free_tlb()` which invokes `__pte_free_tlb()` in turn. Other than paravirtualised calls, this invokes the PTE destructor via `pgtable_pte_page_dtor()` which, as at the PMD level, frees the page table lock, clears the folio flag and adjusts statistics.

7.1.6 Flushing the TLB

The TLB mechanisms already described provide us with two sets of data – a *range* of virtual addresses which require their TLB cache to be invalidated* and an array of struct page objects that need to have their reference counter decremented and may need to be freed.

Note that the flushing logic is architecture-dependent, however for brevity as usual we simply go ahead and examine the x86-64 case through-out.

The key function for flushing the TLB and freeing pages is `tlb_flush_mmu()` as shown in Listing 7-88.

```

259 void tlb_flush_mmu(struct mmu_gather *tlb)
260 {
261     tlb_flush_mmu_tlbonly(tlb);
262     tlb_flush_mmu_free(tlb);
263 }
```

Listing 7-88: mm/mmu_gather.c: tlb_flush_mmu()

With the `tlb_flush_mmu_tlbonly()` function performing the TLB flushing and `tlb_flush_mmu_free()` which potentially frees pages. Note that these functions are also called individually where it is sensible to do so.

Examining `tlb_flush_mmu_tlbonly()` (eliding an out of scope MMU notifier call) as shown in Listing 7-89.

```

416 static inline void tlb_flush_mmu_tlbonly(struct mmu_gather *tlb)
417 {
418     /*
419      * Anything calling __tlb_adjust_range() also sets at least one of
420      * these bits.
421      */
422     if (!(tlb->freed_tables || tlb->cleared_ptes || tlb->cleared_pmds ||
423         tlb->cleared_puds || tlb->cleared_p4ds))
424         return;
425
426     tlb_flush(tlb);
427     :
428     __tlb_reset_range(tlb);
429 }
```

Listing 7-89: include/asm-generic/tlb.h: tlb_flush_mmu_tlbonly()

Not that the fields indicating that individual page table levels have been cleared like `cleared_ptes` are huge table-specific and out of scope.

*. We use ‘invalidate’ and ‘flush’ interchangeably here. Typically the difference between the two is that the former simply marks the cache invalid while the latter also write what is cached to the backing store, however in the case of a TLB this doesn’t make sense so the two can be considered equivalent.

If something has changed in the range, then `tlb_flush()` is invoked and the range is reset by `__tlb_reset_range()` to mark that the flush has occurred. Examining the flush function as shown in Listing 7-90.

```

10 static inline void tlb_flush(struct mmu_gather *tlb)
11 {
12     unsigned long start = 0UL, end = TLB_FLUSH_ALL;
13     unsigned int stride_shift = tlb_get_unmap_shift(tlb);
14
15     if (!tlb->fullmm && !tlb->need_flush_all) {
16         start = tlb->start;
17         end = tlb->end;
18     }
19
20     flush_tlb_mm_range(tlb->mm, start, end, stride_shift, tlb->
21         freed_tables);
22 }
```

Listing 7-90: arch/x86/include/asm/tlb.h: `tlb_flush()`

Note that `stride_shift` will be set to `PAGE_SHIFT` for non-huge pages.

The start and end values will either span the range specified by the `struct mmu_gather` object or the entire range to `TLB_FLUSH_ALL` (this is hard-coded to the maximum unsigned long value possible) if either `fullmm` or `need_flush_all` is specified.

The heavy lifting is performed by `flush_tlb_mm_range()` as shown in Listing 7-91.

```

981 void flush_tlb_mm_range(struct mm_struct *mm, unsigned long start,
982                        unsigned long end, unsigned int stride_shift,
983                        bool freed_tables)
984 {
985     struct flush_tlb_info *info;
986     u64 new_tlb_gen;
987     int cpu;
988
989     cpu = get_cpu();
990
991     /* Should we flush just the requested range? */
992     if ((end == TLB_FLUSH_ALL) ||
993         ((end - start) >> stride_shift) > tlb_single_page_flush_ceiling) {
994         start = 0;
995         end = TLB_FLUSH_ALL;
996     }
997
998     /* This is also a barrier that synchronizes with switch_mm(). */
999     new_tlb_gen = inc_mm_tlb_gen(mm);
1000
1001     info = get_flush_tlb_info(mm, start, end, stride_shift, freed_tables,
```



```

1002                 new_tlb_gen);
1003
1004     /*
1005     * flush_tlb_multi() is not optimized for the common case in which
1006     * only
1007     * a local TLB flush is needed. Optimize this use-case by calling
1008     * flush_tlb_func_local() directly in this case.
1009     */
1010     if (cpumask_any_but(mm_cpumask(mm), cpu) < nr_cpu_ids) {
1011         flush_tlb_multi(mm_cpumask(mm), info);
1012     } else if (mm == this_cpu_read(cpu_tlbstate.loaded_mm)) {
1013         lockdep_assert_irqs_enabled();
1014         local_irq_disable();
1015         flush_tlb_func(info);
1016         local_irq_enable();
1017     }
1018
1019     put_flush_tlb_info();
1020     put_cpu();
1021 }

```

Listing 7-91: arch/x86/mm/tlb.c: *flush_tlb_mm_range()*

We start by disabling preemption and obtaining the processor CPU via `get_cpu()`.

We then determine whether we should flush the entire range – this can be activated by the caller specifying so or if the number of pages to flush exceeds `tlb_single_page_flush_ceiling` which is hardcoded to 33 pages.

This exists as an optimisation where the range-based flush operations may end up less efficient than a simple global TLB flush.

`inc_mm_tlb_gen()` is called which is used to track which *generation* of TLB operation is being performed, used to synchronise these operations with context switches.

The generation of a TLB operation is a simple atomically accessed, monotonically incrementing, counter which keeps a track of which TLB state a CPU currently possesses.

The `get_flush_tlb_info()` function establishes a new struct `flush_tlb_info` object which is threaded through remaining logic as shown in Listing 7-92.

```

174 /*
175  * TLB flushing:
176  *
177  * - flush_tlb_all() flushes all processes TLBs
178  * - flush_tlb_mm(mm) flushes the specified mm context TLB's
179  * - flush_tlb_page(vma, vmaddr) flushes one page
180  * - flush_tlb_range(vma, start, end) flushes a range of pages
181  * - flush_tlb_kernel_range(start, end) flushes a range of kernel pages
182  * - flush_tlb_multi(cpumask, info) flushes TLBs on multiple cpus
183  *

```

```

184 * ..but the i386 has somewhat limited tlb flushing capabilities,
185 * and page-granular flushes are available only on i486 and up.
186 */
187 struct flush_tlb_info {
188     /*
189      * We support several kinds of flushes.
190      *
191      * - Fully flush a single mm. .mm will be set, .end will be
192      *   TLB_FLUSH_ALL, and .new_tlb_gen will be the tlb_gen to
193      *   which the IPI sender is trying to catch us up.
194      *
195      * - Partially flush a single mm. .mm will be set, .start and
196      *   .end will indicate the range, and .new_tlb_gen will be set
197      *   such that the changes between generation .new_tlb_gen-1 and
198      *   .new_tlb_gen are entirely contained in the indicated range.
199      *
200      * - Fully flush all mms whose tlb_gens have been updated. .mm
201      *   will be NULL, .end will be TLB_FLUSH_ALL, and .new_tlb_gen
202      *   will be zero.
203      */
204     struct mm_struct      *mm;
205     unsigned long         start;
206     unsigned long         end;
207     u64                   new_tlb_gen;
208     unsigned int          initiating_cpu;
209     u8                    stride_shift;
210     u8                    freed_tables;
211 };

```

Listing 7-92: arch/x86/include/asm/tlbflush.h: struct flush_tlb_info

We then reach an important part of the logic and an absolutely *key* part of the TLB flush mechanism. If the memory in question is accessed by more than one CPU, we need to perform a *TLB remote shutdown* operation.

A TLB remote shutdown occurs when we initiate an inter-processor-interrupt (IPI) to indicate to other CPUs that they must flush their TLBs. This is an expensive operation and best avoided, so if we need only flush the current CPU we do so.

The fundamental function which performs the actual hardware TLB flushing is `flush_tlb_func()`, to which we shall return. This is called directly if we do not need to perform a remote shutdown, otherwise it is called by each affected CPU.

How do we determine which CPUs have access to the memory described by the struct `mm_struct` object? This is determined via `mm_cpumask()` which looks up the `cpu_bitmap` field which tracks this information.

Finally when the operation is complete, the `put_cpu()` function is called which re-enables preemption (note that the info object does not require any ‘put’ operation other than in debug modes).

The function which is invoked when a remote shutdown is required is `flush_tlb_multi()`, which in turn invokes `__flush_tlb_multi()` and `native_flush_tlb_multi()` (eliding out of scope tracing logic) as shown in Listing 7-93.

```

889 STATIC_NOPV void native_flush_tlb_multi(const struct cpumask *cpumask,
890                                         const struct flush_tlb_info *info)
891 {
892     :
904     /*
905      * If no page tables were freed, we can skip sending IPIs to
906      * CPUs in lazy TLB mode. They will flush the CPU themselves
907      * at the next context switch.
908      *
909      * However, if page tables are getting freed, we need to send the
910      * IPI everywhere, to prevent CPUs in lazy TLB mode from tripping
911      * up on the new contents of what used to be page tables, while
912      * doing a speculative memory access.
913      */
914     if (info->freed_tables)
915         on_each_cpu_mask(cpumask, flush_tlb_func, (void *)info, true);
916     else
917         on_each_cpu_cond_mask(tlb_is_not_lazy, flush_tlb_func,
918                               (void *)info, 1, cpumask);
919 }
```

Listing 7-93: `arch/x86/mm/tlb.c: native_flush_tlb_multi()`

The heavy lifting of the IPI is performed via `on_each_cpu_mask()` and `on_each_cpu_cond_mask()`, however discussion of this is out of scope for the book.

If no page tables are freed, then we skip those processes which are currently marked as being in lazy TLB mode (see section 7.1.8 for more details on this).

Both the remote and local shutdown variants ultimately invoke `flush_tlb_func()` where the actual flushing takes place (eliding tracing, statistics and ASID logic) as shown in Listing 7-94.

```

716 /*
717  * flush_tlb_func()'s memory ordering requirement is that any
718  * TLB fills that happen after we flush the TLB are ordered after we
719  * read active_mm's tlb_gen. We don't need any explicit barriers
720  * because all x86 flush operations are serializing and the
721  * atomic64_read operation won't be reordered by the compiler.
722  */
723 static void flush_tlb_func(void *info)
724 {
725     /*
726      * We have three different tlb_gen values in here. They are:
```

```

727      *
728      * - mm_tlb_gen:      the latest generation.
729      * - local_tlb_gen:  the generation that this CPU has already caught
730      *                    up to.
731      * - f->new_tlb_gen:  the generation that the requester of the flush
732      *                    wants us to catch up to.
733      */
734      const struct flush_tlb_info *f = info;
735      struct mm_struct *loaded_mm = this_cpu_read(cpu_tlbstate.loaded_mm);
736      u32 loaded_mm_asid = this_cpu_read(cpu_tlbstate.loaded_mm_asid);
737      u64 local_tlb_gen = this_cpu_read(cpu_tlbstate.ctxs[loaded_mm_asid].
tlb_gen);
738      bool local = smp_processor_id() == f->initiating_cpu;
739      unsigned long nr_invalidate = 0;
740      u64 mm_tlb_gen;
741
742      /* This code cannot presently handle being reentered. */
743      VM_WARN_ON(!irqs_disabled());
744
745      if (!local) {
746          :
749              /* Can only happen on remote CPUs */
750              if (f->mm && f->mm != loaded_mm)
751                  return;
752      }
753
754      if (unlikely(loaded_mm == &init_mm))
755          return;
756      :
760      if (this_cpu_read(cpu_tlbstate_shared.is_lazy)) {
761          /*
762           * We're in lazy mode. We need to at least flush our
763           * paging-structure cache to avoid speculatively reading
764           * garbage into our TLB. Since switching to init_mm is barely
765           * slower than a minimal flush, just switch to init_mm.
766           *
767           * This should be rare, with native_flush_tlb_multi() skipping
768           * IPIs to lazy TLB mode CPUs.
769           */
770          switch_mm_irqs_off(NULL, &init_mm, NULL);
771          return;
772      }
773
774      if (unlikely(f->new_tlb_gen != TLB_GENERATION_INVALID &&
775          f->new_tlb_gen <= local_tlb_gen)) {
776          /*
777           * The TLB is already up to date in respect to f->new_tlb_gen.

```

```

778             * While the core might be still behind mm_tlb_gen, checking
779             * mm_tlb_gen unnecessarily would have negative caching
           effects
780             * so avoid it.
781             */
782             return;
783     }
784
785     /*
786     * Defer mm_tlb_gen reading as long as possible to avoid cache
787     * contention.
788     */
789     mm_tlb_gen = atomic64_read(&loaded_mm->context.tlb_gen);
790
791     if (unlikely(local_tlb_gen == mm_tlb_gen)) {
792         /*
793         * There's nothing to do: we're already up to date. This can
794         * happen if two concurrent flushes happen -- the first flush
           to
795         * be handled can catch us all the way up, leaving no work for
796         * the second flush.
797         */
798         goto done;
799     }
800
801     WARN_ON_ONCE(local_tlb_gen > mm_tlb_gen);
802     WARN_ON_ONCE(f->new_tlb_gen > mm_tlb_gen);

```

Listing 7-94: arch/x86/mm/tlb.c: flush_tlb_func() initialisation and guard clauses

We start by retrieving per-CPU state from `cpu_tlbstate` indicating the currently loaded `struct mm_struct` object in `loaded_mm`, the local TLB generation in `local_tlb_gen` and whether the operation is local in `local`.

We assert that IRQs are disabled, abort the operation if this the TLB batch we are parameterised by does not specify the same process address space as this CPU and ensure that we are not running in the context of the zygote kernel space `init_mm`.

If the current CPU is in lazy TLB mode (see section 7.1.8 for more on this), i.e. we are a kernel thread which has ‘borrowed’ the page table mappings of the previously running process, it is simpler at this point to just switch to the zygote process which we do via `switch_mm_irqs_off()`.

If the generation number is not valid, then we abort, otherwise we finally read the current generation number of the currently loaded process address space and place it in `mm_tlb_gen`. If this matches the current CPU’s generation (stored in `local_tlb_gen`) then we have nothing to do.

Finally, we perform some sanity checks on the TLB generation, before moving ahead with the TLB flush itself as shown in Listing 7-95.

```

804      /*
805       * If we get to this point, we know that our TLB is out of date.
806       * This does not strictly imply that we need to flush (it's
807       * possible that f->new_tlb_gen <= local_tlb_gen), but we're
808       * going to need to flush in the very near future, so we might
809       * as well get it over with.
810       *
811       * The only question is whether to do a full or partial flush.
812       *
813       * We do a partial flush if requested and two extra conditions
814       * are met:
815       *
816       * 1. f->new_tlb_gen == local_tlb_gen + 1. We have an invariant that
817       *    we've always done all needed flushes to catch up to
818       *    local_tlb_gen. If, for example, local_tlb_gen == 2 and
819       *    f->new_tlb_gen == 3, then we know that the flush needed to bring
820       *    us up to date for tlb_gen 3 is the partial flush we're
821       *    processing.
822       *
823       * As an example of why this check is needed, suppose that there
824       * are two concurrent flushes. The first is a full flush that
825       * changes context.tlb_gen from 1 to 2. The second is a partial
826       * flush that changes context.tlb_gen from 2 to 3. If they get
827       * processed on this CPU in reverse order, we'll see
828       *    local_tlb_gen == 1, mm_tlb_gen == 3, and end != TLB_FLUSH_ALL.
829       *    If we were to use __flush_tlb_one_user() and set local_tlb_gen
830       *    to
831       *    3, we'd break the invariant: we'd update local_tlb_gen above
832       *    1 without the full flush that's needed for tlb_gen 2.
833       *
834       * 2. f->new_tlb_gen == mm_tlb_gen. This is purely an optimization.
835       *    Partial TLB flushes are not all that much cheaper than full TLB
836       *    flushes, so it seems unlikely that it would be a performance win
837       *    to do a partial flush if that won't bring our TLB fully up to
838       *    date. By doing a full flush instead, we can increase
839       *    local_tlb_gen all the way to mm_tlb_gen and we can probably
840       *    avoid another flush in the very near future.
841       */
842      if (f->end != TLB_FLUSH_ALL &&
843          f->new_tlb_gen == local_tlb_gen + 1 &&
844          f->new_tlb_gen == mm_tlb_gen) {
845          /* Partial flush */
846          unsigned long addr = f->start;
847
848          /* Partial flush cannot have invalid generations */
849          VM_WARN_ON(f->new_tlb_gen == TLB_GENERATION_INVALID);

```

```

850         /* Partial flush must have valid mm */
851         VM_WARN_ON(f->mm == NULL);
852
853         nr_invalidate = (f->end - f->start) >> f->stride_shift;
854
855         while (addr < f->end) {
856             flush_tlb_one_user(addr);
857             addr += 1UL << f->stride_shift;
858         }
859         :
861     } else {
862         /* Full flush. */
863         nr_invalidate = TLB_FLUSH_ALL;
864
865         flush_tlb_local();
866         :
868     }
869
870     /* Both paths above update our state to mm_tlb_gen. */
871     this_cpu_write(cpu_tlbstate.ctxs[loaded_mm_asid].tlb_gen, mm_tlb_gen);
872
873     /* Tracing is done in a unified manner to reduce the code size */
874 done:
875     :
879 }

```

Listing 7-95: arch/x86/mm/tlb.c: flush_tlb_func() flushing

At this point the flush is either performed per each stride, via `flush_tlb_one_user()` (for range-based flushing) or globally via `flush_tlb_local()`.

The range-based approach ultimately invokes `native_flush_tlb_one_user()` which performs the flush via the x86-64 `invlpg` instruction.

The global flush ultimately invokes `native_flush_tlb_local()` which performs the flush by replacing the contents of the x86-64 `cr3` control register.

After the flush is complete, the CPU's TLB generation is updated.

7.1.7 Freeing pages

struct page objects are freed via `tlb_flush_mmu_free()` which invokes `tlb_batch_pages_flush()` in turn:-

```

45 static void tlb_batch_pages_flush(struct mmu_gather *tlb)
46 {
47     struct mmu_gather_batch *batch;
48
49     for (batch = &tlb->local; batch && batch->nr; batch = batch->next) {
50         struct page **pages = batch->pages;

```

```

51
52         do {
53             /*
54              * limit free batch count when PAGE_SIZE > 4K
55              */
56             unsigned int nr = min(512U, batch->nr);
57
58             free_pages_and_swap_cache(pages, nr);
59             pages += nr;
60             batch->nr -= nr;
61
62             cond_resched();
63         } while (batch->nr);
64     }
65     tlb->active = &tlb->local;
66 }

```

Listing 7-96: mm/mmu_gather.c: tlb_batch_pages_flush()

This walks all of the struct `mmu_gather_batch` objects, freeing via `free_pages_and_swap_cache()` as shown in Listing 7-97.

```

299 /*
300  * Passed an array of pages, drop them all from swapcache and then release
301  * them. They are removed from the LRU and freed if this is their last use.
302  */
303 void free_pages_and_swap_cache(struct page **pages, int nr)
304 {
305     struct page **pagep = pages;
306     int i;
307
308     lru_add_drain();
309     for (i = 0; i < nr; i++)
310         free_swap_cache(pagep[i]);
311     release_pages(pagep, nr);
312 }

```

Listing 7-97: mm/swap_state.c: free_pages_and_swap_cache()

This drains folio batches to ensure that there are no odd non-LRU cases to deal with when freeing these pages via `lru_add_drain()` (see section 11.7 for more on folio batches), before freeing related swap caches (see the swap chapter for more on that) via `free_swap_cache()`.

The actual decrement of the page reference count and possible freeing is achieved via `release_pages()` (eliding out of scope huge page, zone device, cgroup and statistics logic) as shown in Listing 7-98.

```

926 /**
927  * release_pages - batched put_page()
928  * @pages: array of pages to release

```



```

929  * @nr: number of pages
930  *
931  * Decrement the reference count on all the pages in @pages. If it
932  * fell to zero, remove the page from the LRU and free it.
933  */
934  void release_pages(struct page **pages, int nr)
935  {
936      int i;
937      LIST_HEAD(pages_to_free);
938      struct lruvec *lruvec = NULL;
939      unsigned long flags = 0;
940      unsigned int lock_batch;
941
942      for (i = 0; i < nr; i++) {
943          struct folio *folio = page_folio(pages[i]);
944
945          /*
946           * Make sure the IRQ-safe lock-holding time does not get
947           * excessive with a continuous string of pages from the
948           * same lruvec. The lock is held only if lruvec != NULL.
949           */
950          if (lruvec && ++lock_batch == SWAP_CLUSTER_MAX) {
951              unlock_page_lruvec_irqrestore(lruvec, flags);
952              lruvec = NULL;
953          }
954          :
955          if (!folio_put_testzero(folio))
956              continue;
957          :
958          if (folio_test_lru(folio)) {
959              struct lruvec *prev_lruvec = lruvec;
960
961              lruvec = folio_lruvec_relock_irqsave(folio, lruvec,
962                                                    &flags);
963          };
964
965          if (prev_lruvec != lruvec)
966              lock_batch = 0;
967
968          lruvec_del_folio(lruvec, folio);
969          __folio_clear_lru_flags(folio);
970      }
971
972      /*
973       * In rare cases, when truncation or holepunching raced with
974       * munlock after VM_LOCKED was cleared, Mlocked may still be
975       * found set here. This does not indicate a problem, unless
976       * "unevictable_pgs_cleared" appears worryingly large.

```

```

999             */
1000             if (unlikely(folio_test_mlocked(folio))) {
1001                 __folio_clear_mlocked(folio);
1002             }
1003         }
1004     }
1005     list_add(&folio->lru, &pages_to_free);
1006 }
1007 if (lruvec)
1008     unlock_page_lruvec_irqrestore(lruvec, flags);
1009 }
1010 free_unref_page_list(&pages_to_free);
1011 }

```

Listing 7-98: mm/swap.c: release_pages()

We won't go into extensive detail here as this is the purview of the physical memory chapter, however the key thing is the `folio_put_testzero()` call which decrements the folio's reference count and if it is non-zero we move to the next folio.

The actual physical allocation is ultimately performed by `free_unref_page_list()`.

Note that we do not perform a check on the folio map count. This is because every mapping will also result in the folio's reference counter being incremented – this may exceed the map count (the kernel may need to take a reference) but will never be below it.

7.1.8 Lazy TLB mode

Lazy TLB mode is a feature within the kernel whereby kernel threads are able to simply 'borrow' the address space of the previous running process rather than performing a costly context switch when context switching from a userland process to a kernel thread.

The fact that a process has entered lazy TLB mode is set by `enter_lazy_tlb()` as shown in Listing 7-99.

```

650  /*
651   * Please ignore the name of this function. It should be called
652   * switch_to_kernel_thread().
653   *
654   * enter_lazy_tlb() is a hint from the scheduler that we are entering a
655   * kernel thread or other context without an mm. Acceptable implementations
656   * include doing nothing whatsoever, switching to init_mm, or various clever
657   * lazy tricks to try to minimize TLB flushes.
658   *
659   * The scheduler reserves the right to call enter_lazy_tlb() several times
660   * in a row. It will notify us that we're going back to a real mm by
661   * calling switch_mm_irqs_off().
662   */

```

```

663 void enter_lazy_tlb(struct mm_struct *mm, struct task_struct *tsk)
664 {
665     if (this_cpu_read(cpu_tlbstate.loaded_mm) == &init_mm)
666         return;
667
668     this_cpu_write(cpu_tlbstate_shared.is_lazy, true);
669 }

```

Listing 7-99: arch/x86/mm/tlb.c: `enter_lazy_tlb()`

This stores the fact that a process is lazy so the TLB logic can know that there is not an issue as a result. In addition, the fact that we do this can result in corrupted TLB in kernel mode, which is an expected side effect – the `spurious_kernel_fault()` function explicitly handles this by (see section 6.1 on hardware page faulting).

When a process is switched and it is no longer appropriate to maintain lazy TLB mode, this is achieved in `switch_mm_irqs_off()`.

8

MANIPULATING USERLAND MEMORY

There are a number of situations where both kernel and userland need to manipulation userland memory. This chapter firstly examines the means by which the kernel can do so, via direct user memory access (uaccess) and the Get User Pages (GUP) mechanisms, before examining a number of userland APIs for manipulating the attributes of userland memory: `mremap()`, `madvise()`, `mlock()` and `mprotect()`.

8.1 Accessing User Memory from the Kernel

Sometimes the kernel needs to access userland memory. This is significantly less straightforward than accessing kernel memory, which can simply be addressed using the direct mapping (or kernel text mapping) via `__pa()`, `__va()`, `virt_to_folio()` and friends and for which only kernel synchronisation need be considered.

Userland mappings, on the other hand, are not at all so easily accessed – they each have their own independent page tables which must be walked if we need to determine the folios associated with them, may be accessed

and manipulated at any time by userland and can disappear from under you altogether as the userland process maps and unmaps memory.

There are two separate parts of the kernel that enable the kernel to access user memory – *user access* (or *uaccess* for short) and the *Get User Pages* (*GUP*) interface. We will examine both.

User access functions are used by the kernel mode of the *current* process to simply do a best effort to transfer data between kernel and user memory, with no attempt to perform locks or adjust reference counts of any kind.

GUP on the other hand allow folios to be *pinned* (i.e. incrementing the underlying folio's reference count such that they will not be freed until unpinned), optionally returns folio and VMA objects and allows fine-grained control over how the whole operation is performed.

8.1.1 User Access

Many kernel system calls read data from or write data to userland mappings. These occur in kernel mode from the context of the current process, so if the specified userland virtual address is valid, either its userland mappings will be in place or it can be faulted in as required (i.e. the address will be marked valid in a VMA).

The *uaccess* functionality takes advantage of this by performing a *very* carefully orchestrated and tailored version of a *memcpy()* from kernel mode.

The fundamental issue with doing this, and the reason why you must *never* access user memory directly in the kernel, is that *unhandled* page faults originating from kernel mode ordinarily have no means of recovery and thus result in a kernel oops.

Note that when userland memory is accessed from the kernel and a page fault occurs which is correctly handled (i.e. faulted in), then everything functions as if that memory had been faulted in by userland. See section 6 for details on page fault logic.

However, some kernel functions are given a special designation whereby they are permitted to fault. These use what is known as an 'exception table' to mark regions of kernel virtual memory where access is permitted as well as information on how to recover from the error.

The key user access function, *copy_from_user()* and *copy_to_user()* use precisely this mechanism in order to perform these actions. Examining both:-

```

148 static __always_inline unsigned long __must_check
149 copy_from_user(void *to, const void __user *from, unsigned long n)
150 {
151     if (check_copy_size(to, n, false))
152         n = _copy_from_user(to, from, n);
153     return n;
154 }
```

Listing 8-1: include/linux/uaccess.h: copy_from_user()

```

156 static __always_inline unsigned long __must_check
```

```

157 copy_to_user(void __user *to, const void *from, unsigned long n)
158 {
159     if (check_copy_size(from, n, true))
160         n = _copy_to_user(to, from, n);
161     return n;
162 }

```

Listing 8-2: include/linux/uaccess.h: copy_to_user()

Note that `__must_check` is a compiler directive that generates a warning if the returned value is not used and `__user` is a static analysis tag used to assert that the pointers passed are userland ones.

Each are parameterised by `from` and `to` indicating source and target buffers respectively, `n` specifies how many bytes to copy and the return value indicates the number of bytes that could *not* be copied. This being non-zero typically implies a page fault has occurred, so this can be used as an error indicator – if the result is zero then the operation succeeded, otherwise it failed. Trailing bytes are cleared on failure.

The actual implementation of these functions are provided by `_copy_from_user()` and `_copy_to_user()` which can be inlined depending on architecture, but on x86-64, which we focus on, it is not. Examining these:-

```

10 unsigned long _copy_from_user(void *to, const void __user *from, unsigned long
    n)
11 {
12     unsigned long res = n;
13     might_fault();
14     if (!should_fail_usercopy() && likely(access_ok(from, n))) {
15         instrument_copy_from_user(to, from, n);
16         res = raw_copy_from_user(to, from, n);
17     }
18     if (unlikely(res))
19         memset(to + (n - res), 0, res);
20     return res;
21 }

```

Listing 8-3: lib/usercopy.c: _copy_from_user()

```

26 unsigned long _copy_to_user(void __user *to, const void *from, unsigned long n
    )
27 {
28     might_fault();
29     if (should_fail_usercopy())
30         return n;
31     if (likely(access_ok(to, n))) {
32         instrument_copy_to_user(to, from, n);
33         n = raw_copy_to_user(to, from, n);
34     }
35     return n;

```

36 }

Listing 8-4: lib/usercopy.c: `_copy_to_user()`

Note that:-

- `might_fault()` is used by kernel lock/atomic sleep debugging and only meaningful when either `CONFIG_PROVE_LOCKING` or `CONFIG_DEBUG_ATOMIC_SLEEP` are configured.
- `should_fail_usercopy()` is part of the fault injection logic (only meaningful if `CONFIG_FAULT_INJECTION_USERCOPY` is configured) and out of scope for the book.
- `instrument_copy_from_user()` and `instrument_copy_to_user()` are used by KASAN and KCSAN both of which are out of scope of the book.

The `access_ok()` macro determines whether the specified address is valid for userland access, which in x86-64 uses the generic `__access_ok()` handler which simply checks that the address is below the maximum permitted userland virtual address, `TASK_SIZE_MAX`.

The heavy lifting is left to the raw versions of the functions. Examining these:-

```

49 static __always_inline __must_check unsigned long
50 raw_copy_from_user(void *dst, const void __user *src, unsigned long size)
51 {
52     return copy_user_generic(dst, (__force void *)src, size);
53 }
```

Listing 8-5: arch/x86/include/asm/uaccess_64.h: `raw_copy_from_user()`

```

55 static __always_inline __must_check unsigned long
56 raw_copy_to_user(void __user *dst, const void *src, unsigned long size)
57 {
58     return copy_user_generic((__force void *)dst, src, size);
59 }
```

Listing 8-6: arch/x86/include/asm/uaccess_64.h: `raw_copy_to_user()`

Note that `__force` is a static analysis hint that has no effect for an ordinary build. It indicates that the pointer should not be checked for address space consistency.

These both invoke `copy_user_generic()` which performs the actual copy operation:-

```

27 static __always_inline __must_check unsigned long
28 copy_user_generic(void *to, const void *from, unsigned len)
29 {
30     unsigned ret;
31
32     /*
```

```

33      * If CPU has ERMS feature, use copy_user_enhanced_fast_string.
34      * Otherwise, if CPU has rep_good feature, use
      copy_user_generic_string.
35      * Otherwise, use copy_user_generic_unrolled.
36      */
37      alternative_call_2(copy_user_generic_unrolled,
38                        copy_user_generic_string,
39                        X86_FEATURE_REP_GOOD,
40                        copy_user_enhanced_fast_string,
41                        X86_FEATURE_ERMS,
42                        ASM_OUTPUT2("=a" (ret), "=D" (to), "=S" (from),
43                                   "=d" (len)),
44                                   "1" (to), "2" (from), "3" (len)
45                                   : "memory", "rcx", "r8", "r9", "r10", "r11");
46      return ret;
47  }

```

Listing 8-7: arch/x86/include/asm/uaccess_64.h: `copy_user_generic()`

This x86-64-specific routine invokes different assembly routines to perform the actual copy depending on available CPU features. For the sake of brevity we will examine only one, `copy_user_enhanced_fast_string()`:-

```

149  /*
150   * Some CPUs are adding enhanced REP MOVSB/STOSB instructions.
151   * It's recommended to use enhanced REP MOVSB/STOSB if it's enabled.
152   *
153   * Input:
154   * rdi destination
155   * rsi source
156   * rdx count
157   *
158   * Output:
159   * eax uncopied bytes or 0 if successful.
160   */
161  SYM_FUNC_START(copy_user_enhanced_fast_string)
162      ASM_STAC
163      /* CPUs without FSRM should avoid rep movsb for short copies */
164      ALTERNATIVE "cmpl $64, %edx; jb copy_user_short_string", "",
      X86_FEATURE_FSRM
165      movl %edx,%ecx
166  1:    rep movsb
167      xorl %eax,%eax
168      ASM_CLAC
169      RET
170
171  12:   movl %ecx,%edx      /* ecx is zero/rest also */
172      jmp .lcopy_user_handle_tail
173

```



```

174     _ASM_EXTABLE_COPY(1b, 12b)
175     SYM_FUNC_END(copy_user_enhanced_fast_string)

```

Listing 8-8: arch/x86/lib/copy_user_64.S: copy_user_enhanced_fast_string()

A description of how the actual copy is performed here is out of scope, other than to observe that we operate directly on the supplied addresses with no further abstraction. The key thing to note here is the invocation of the assembler macro `_ASM_EXTABLE_COPY()`. This is used at compile time to specify that the exception table should contain the address of the instruction at label 1 (the one performing the actual copy) and, if an *unhandled* page fault occurs while executing it, to resume execution at label 12. This is performed in `_ASM_EXTABLE_TYPE()`:-

```

135 # define _ASM_EXTABLE_TYPE(from, to, type)          \
136     .pushsection "__ex_table","a" ;                 \
137     .balign 4 ;                                     \
138     .long (from) - . ;                             \
139     .long (to) - . ;                               \
140     .long type ;                                    \
141     .popsection

```

Listing 8-9: arch/x86/include/asm/asm.h: _ASM_EXTABLE_TYPE()

This sets up an entry in the exception table of type `struct exception_table_entry`, storing the permitted exception address (here denoted `from`) and fixup address (here denoted `to`) relative to the exception table entry to save on space.

If a page fault within the memory range being read from/to occurs and is handled successfully, then the operation proceeds as if it were performed in userland. However, if an invalid address is accessed or another error occurs on page fault, then it receives special handling.

The function which determines whether an invalid page fault in kernel mode can be handled or ought to result in a kernel oops (and thus potentially a system lockup) in x86-64 is `kernelmode_fixup_or_oops()`. Stripping out irrelevant details:-

```

711 static ninline void
712 kernelmode_fixup_or_oops(struct pt_regs *regs, unsigned long error_code,
713                          unsigned long address, int signal, int si_code,
714                          u32 pkey)
715 {
716     WARN_ON_ONCE(user_mode(regs));
717
718     /* Are we prepared to handle this kernel fault? */
719     if (fixup_exception(regs, X86_TRAP_PF, error_code, address)) {
720         :
721     }
722     return;
723 }

```

```

760         page_fault_oops(regs, error_code, address);
761     }

```

Listing 8-10: arch/x86/mm/fault.c: `kernelmode_fixup_or_oops()`

This invokes `fixup_exception()` to check whether the faulting address is referenced in an exception table (and perform the fixup if so). If it isn't then `page_fault_oops()` is called.

Examining this function:-

```

205 int fixup_exception(struct pt_regs *regs, int trapnr, unsigned long error_code
    ,
206                     unsigned long fault_addr)
207 {
208     const struct exception_table_entry *e;
209     int type, reg, imm;
210     :
225     e = search_exception_tables(regs->ip);
226     if (!e)
227         return 0;
228
229     type = FIELD_GET(EX_DATA_TYPE_MASK, e->data);
230     reg = FIELD_GET(EX_DATA_REG_MASK, e->data);
231     imm = FIELD_GET(EX_DATA_IMM_MASK, e->data);
232
233     switch (type) {
234     :
242     case EX_TYPE_COPY:
243         return ex_handler_copy(e, regs, trapnr);
244     :
275     }
276     BUG();
277 }

```

Listing 8-11: arch/x86/mm/extable.c: `fixup_exception()`

The key function here is `search_exception_tables()` which searches each of the kernel, module and BPF exception tables for the faulting address. The key of these for the purposes of uaccess being the kernel table searched by `search_kernel_exception_table()`. This invokes `search_extable()` to do the heavy lifting.

This performs a binary search over the exception table for the address, using `ex_to_insn()` to obtain the faulting instruction from the stored relative address:-

```

18 static inline unsigned long ex_to_insn(const struct exception_table_entry *x)
19 {
20     return (unsigned long)&x->insn + x->insn;
21 }

```

Listing 8-12: lib/extable.c: `ex_to_insn()`

Which offsets from the location of the exception table entry to obtain the correct address with which to compare the faulting one.

Once an entry is located, `ex_handler_copy()` is invoked to perform the fixup. This in turn invokes `ex_handler_fault()` which calls `ex_handler_default()` which fixes up the faulting instruction pointer using `ex_fixup_addr()`:-

```

26 static inline unsigned long
27 ex_fixup_addr(const struct exception_table_entry *x)
28 {
29     return (unsigned long)&x->fixup + x->fixup;
30 }
```

Listing 8-13: arch/x86/mm/extable.c: `ex_fixup_addr()`

This again uses the relative offset to point the instruction pointer at the specified fixup address allowing the page fault to be cleared. This will result in a non-zero return value as not all bytes were copied and thus the caller can deduce an error has occurred.

When memory is copied between kernel and userspace but we have a *kernel* virtual address for the source/target userland buffer, the functions `copy_from_user_page()` and `copy_to_user_page()` are used.

These perform careful cache operations to ensure that, on architectures where accessing the memory through the direct mapping might result in inconsistency in userland, the appropriate cache lines are invalidated. This is not applicable to x86-64.

8.1.2 Get User Pages (GUP)

The Get User Pages (GUP) interface consists of a number of functions used for walking userland mappings from the kernel and, in most cases, *pinning* their underlying folios (incrementing their reference count so they cannot be freed). Most of these are exported for drivers.

When pinning memory, the GUP functions retrieve a list of struct `page` objects (and in some cases, optionally, a list of struct `vm_area_struct` objects) associated with the specified range of user memory.

Note that we exclude discussions of huge pages (deferring all such discussions to the dedicated chapter on the topic), as well as very specific features like DAX.

GUP can be used for operations other than pinning pages, for example faulting in pages, marking them dirty or performing various checks on them. These are often exported by specific helper functions, which we examine briefly in section 8.1.9.

What GUP does with the input memory range is determined via *GUP flags*, each of which we will examine shortly. Firstly, let's take a look at the core GUP API and supporting functions:-

Table 8-1: Core GUP functions

GUP Function	Must hold mmap_lock?	Sets FOLL_GET?	Sets FOLL_PIN?*	Sets FOLL_TOUCH?	Must output pages?	Can output vmas?
get_user_pages()	•	If pages set		•	If FOLL_GET	•
pin_user_pages()	•		•		•	•
get_user_pages_unlocked()		If pages set		•	If FOLL_GET	
pin_user_pages_unlocked()			•		•	
get_user_pages_fast()		•				
get_user_pages_fast_only()		•				
pin_user_pages_fast()			•		•	
pin_user_pages_fast_only()			•		•	
get_user_pages_remote()	•	If pages set		•	If FOLL_GET	•
pin_user_pages_remote()	•		•	•	•	•

* *and allows* the use of FOLL_LONGTERM.

We'll get into what FOLL_GET, FOLL_PIN and FOLL_LONGTERM mean shortly. Only the `_remote()` variants permit access to the `mm_struct` of processes other than the currently running one, and only some functions, notably the ones that require the `mmap_lock` read semaphore to be held, return an array of VMAs.

These core functions share the majority of their parameters, with some additional ones that are specific to certain functions only. Examining them all:-

- `start` – (*all functions*) – An initial page-aligned virtual address.
- `nr_pages` – (*all functions*) – The number of base pages to be processed.
- `gup_flags` – (*all functions*) – GUP flags modifying how the memory is walked, what actions are to be performed and other modifiers.
- `pages` – (*all functions*) – A pointer to an optional *output* array of struct `page` objects in which to place these objects relating to mapping pages (must have at least `nr_pages` entries). All functions which use FOLL_GET or FOLL_PIN should specify these in order to be able to have something to unpin later. This is not, however enforced for `get_user_pages_fast()` or `get_user_pages_fast_only()`.
- `vmas` – `get_user_pages[_remote]()`, `pin_user_pages[_remote]()` – A pointer to an optional *output* array of struct `vm_area_struct` objects (must have at least `nr_pages` entries). The GUP functions which support this require the caller to hold the struct `mm_struct->mmap_lock` read semaphore and will be invalidated as soon as this is dropped. The `_remote` variants therefore must *not* specify the `locked` parameter if they require VMAs to be returned (see below description of this parameter).
- `mm` – `get_user_pages_remote()`, `pin_user_pages_remote()` – An object of struct `mm_struct` type specifying the process address space in which to perform the GUP operation.
- `locked` – `get_user_pages_remote()`, `pin_user_pages_remote()` – A pointer to an optional *output* boolean indicating whether the `mmap_lock` read

semaphore has been dropped due to `VM_FAULT_RETRY`. If specified, then the `vmas` parameter cannot be used as VMAs are no longer valid once the lock is dropped.

All of the functions return the number of base pages that the actions have been applied to, or a negative error code if something has gone wrong.

8.1.3 Pinning Folios

Pinning folios of userland memory simply means incrementing the underlying folio's reference count in order that they will not be freed until the folio is unpinning. This can be done with both anonymous and file-backed memory.

If a folio is pinned there is no guarantee that the userland mapping still exists on return, or is even meaningful in any way from userland's perspective. Therefore the strongest guarantee we can give is that these folios were valid *at some point*. See the *pin user pages* documentation for more on the background of this.

There are two means of pinning folios – `FOLL_GET` which can be only be performed via the `get_*()` family of functions, and `FOLL_PIN` which is performed only (and unconditionally) via the `pin_*()` functions. Both operations require that the `pages` output parameter be specified to return `struct pages`.

The former simply increments the reference count of the underlying folio to ensure it is not freed and expects the user to call `folio_put()` or `put_page()` to decrement the reference count. The latter uses a different means of pinning termed *GUP pinning* (described below) to achieve the same ends and must be unpinning via `unpin_user_page()`, `unpin_user_pages()`, `unpin_user_pages_dirty_lock()` or `unpin_user_page_range_dirty_lock()`.

When deciding which interface to use the criteria are as follows*:-

- Use `get_*_user_pages*()` when the user memory access is short-term, i.e. all actions are performed pinned under the `mmap_lock` (or no pinning is performed). Either data is manipulated (e.g. `__access_remote_vm()`) or only `struct page/struct folio` or `struct vm_area_struct` (VMA) metadata is. Note that this memory will be subject to reclaim, migration and becoming a CoW mapping on fork once the lock is released.
- Use `pin_*_user_pages*()` when memory needs to be pinned longer than the `mmap_lock` is held. Memory can either be kept briefly during which an operation occurs (e.g. DMA writeback) or for the long term, at which point the `FOLL_LONGTERM` flag should be specified to prevent memory fragmentation (this causes the memory to be migrated to an unmovable page block, more on this below).

In either case, the memory cannot be reclaimed or become a CoW mapping on fork and any existing CoW mapping is immediately unshared even if read-only.

*. See the *pin user pages* documentation for a lot more detail on this

The key users of the `pin_*`() functions are DMA* routines utilising userland memory for DMA operations, though other users which ultimately touches folio data, such as direct I/O, also use it.

This is checked permit other parts of the kernel to determine whether these folios are likely to be used for DMA or otherwise have their data manipulated. This is checked by `folio_maybe_dma_pinned()` and `page_maybe_dma_pinned()`. The places where this check are performed are:-

Reclaim – Skips memory reclaim of folios which are marked this way in `shrink_page_list()`.

procsfs – The `/proc/$pid/clear_refs` logic, when used to clear the *soft-dirty* bit, uses this in `pte_is_pinned()`, called from `clear_soft_dirty()`. This causes GUP pinned folios to be skipped for this operation (also based on other criteria).

Reverse mapping , clear anonymous exclusive – Used in the function `page_try_share_anon_rmap()`, which is used to strip the anonymous exclusive folio flag when a folio is potentially about to be shared. If the folio is GUP pinned, then this operation is not performed.

Fork , avoid CoW on GUP pinned – The function `page_needs_cow_for_dma()` is used by `page_try_dup_anon_rmap()`. This returns an error code if a folio is GUP pinned. This is most notably used by the fork logic ultimately invoked via `copy_page_range()` (called by `dup_mmap()`) to force a copy. It is additionally used by huge page logic in the same fashion.

When `FOLL_PIN` is used, the struct `mm_struct` flag `MMF_HAS_PINNED` flag is set in `mm_set_has_pinned_flag()` for the lifetime of the `mm_struct` and used as an early exit condition for `page_needs_cow_for_dma()` described above, for efficiency.

The pinning and unpinning of GUP pinned folios is tracked by `/proc/vmstat` in the `nr_foll_pin_acquired` and `nr_foll_pin_released` statistics.

The actual pinning of *non-compound* folios is performed by either `try_grab_page()` or `try_grab_folio()`. In each case `FOLL_GET` results in a reference count being incremented by one for each folio pinned, and `FOLL_PIN` increments instead by `GUP_PIN_COUNTING_BIAS` for each folio pinned. Unpinning `FOLL_PIN` folios ultimately invokes `gup_put_folio()` which performs the reverse action.

In this case, reference count increments/decrements are multiplied by `GUP_PIN_COUNTING_BIAS`, specified as 2^{10} , for `FOLL_PIN` as the value is both easy to ‘eyeball’ when debugging and unlikely to skew the existing reference count so this can easily be ascertained.

However for compound folios, we have sufficient space in the 2nd subpage to store a precise GUP-pin count, `compound_pincount`, accessed via `compound_pincount_ptr()` or `folio_pincount_ptr()`. You can see this in action in `folio_maybe_dma_pinned()`:-

*. DMA refers to Direct Memory Access – which refers to devices’s ability to read or write memory directly rather than have memory accesses mediated by the CPU.

```

1473 /**
1474  * folio_maybe_dma_pinned - Report if a folio may be pinned for DMA.
1475  * @folio: The folio.
1476  *
1477  * This function checks if a folio has been pinned via a call to
1478  * a function in the pin_user_pages() family.
1479  *
1480  * For small folios, the return value is partially fuzzy: false is not fuzzy,
1481  * because it means "definitely not pinned for DMA", but true means "probably
1482  * pinned for DMA, but possibly a false positive due to having at least
1483  * GUP_PIN_COUNTING_BIAS worth of normal folio references".
1484  *
1485  * False positives are OK, because: a) it's unlikely for a folio to
1486  * get that many refcounts, and b) all the callers of this routine are
1487  * expected to be able to deal gracefully with a false positive.
1488  *
1489  * For large folios, the result will be exactly correct. That's because
1490  * we have more tracking data available: the compound_pincount is used
1491  * instead of the GUP_PIN_COUNTING_BIAS scheme.
1492  *
1493  * For more information, please see Documentation/core-api/pin_user_pages.rst.
1494  *
1495  * Return: True, if it is likely that the page has been "dma-pinned".
1496  * False, if the page is definitely not dma-pinned.
1497  */
1498 static inline bool folio_maybe_dma_pinned(struct folio *folio)
1499 {
1500     if (folio_test_large(folio))
1501         return atomic_read(folio_pincount_ptr(folio)) > 0;
1502
1503     /*
1504      * folio_ref_count() is signed. If that refcount overflows, then
1505      * folio_ref_count() returns a negative value, and callers will avoid
1506      * further incrementing the refcount.
1507      *
1508      * Here, for that overflow case, use the sign bit to count a little
1509      * bit higher via unsigned math, and thus still get an accurate result
1510      */
1511     return ((unsigned int)folio_ref_count(folio)) >=
1512         GUP_PIN_COUNTING_BIAS;
1513 }

```

Listing 8-14: include/linux/mm.h: `folio_maybe_dma_pinned()`

Note the careful handling of reference count overflow.

A flag that is closely related to `FOLL_PIN` (and requires it to be set to be available) is `FOLL_LONGTERM`. This is a hint that the mapping is indefinite and likely to have its data manipulated while in use.

When `FOLL_LONGTERM` is used, each folio is checked to determine whether it is long-term pinnable (i.e. is not in `ZONE_MOVABLE` or otherwise unavailable long-term) via `is_longterm_pinnable_page()`, and unpins them if not. This is checked either in `try_grab_folio()` for the fast path or `check_and_migrate_movable_pages()` otherwise.

This also invokes some additional checks (for example, disallowing this mode for VMAs which are marked for DAX, i.e. direct access, bypassing the page cache, a topic that is out of scope for the book) to take into account this intended usage of the GUP-pinned memory.

8.1.4 Follow Flags

What we actually *do* with these mappings is determined by the `gup_flags` field via ‘follow’ flags prefixed with `FOLL_` and declared in the `include/linux/mm.h` header. Examining each:-

- `FOLL_GET` – One of the two core pinning flags, *see the above section on pinning for a description*.
- `FOLL_PIN` – One of the two core pinning flags, *see the above section on pinning for a description*.
- `FOLL_LONGTERM` – A hint that folios being GUP pinned will have a long life time, *see the above section on pinning for a description*.
- `FOLL_TOUCH` – This sets the `PG_accessed` folio flag in `follow_page_pte()` and `follow_pfn_pte()` via `pte_mkyoung()`, and if `FOLL_WRITE` is set, also marks it dirty (see description of this flag below). It is implicitly set in `get_user_pages()`, `__get_user_pages_remote()` (therefore also in `get_user_pages_remote()` and `pin_user_pages_remote()`), and `get_user_pages_unlocked()`.
- `FOLL_HWPOISON` – Walk the memory range and, if memory is discovered to be hardware poisoned (i.e. marked by hardware as unavailable) when faulted in, explicitly return an `-EHWPOISON` error instead of `-EFAULT` error in `vm_fault_to_errno()`. Used exclusively by the KVM virtualisation logic.
- `FOLL_DUMP` – As the name suggests, used by the core dump functionality in `get_dump_page()`, though also used elsewhere. This flag checks whether folios that are unavailable would ultimately result in references to the zero page and prevents them from being faulted in. This is checked in `no_page_table()` and `follow_page_pte()`.
- `FOLL_FORCE` – Overrides VMA flags `VM_READ` and `VM_WRITE` and permits access to any VMA. Checked in `can_follow_write_pte()` and `check_vma_flags()`. A very powerful and dangerous flag that should be used carefully and sparingly. Typically used by `ptrace` to access otherwise inaccessible memory.

- **FOLL_NOWAIT** – Indicates that the `mmap_lock` should *not* be dropped, and we should not wait when retrying a page fault. This maps on to the `FAULT_FLAG_RETRY_NOWAIT` flag. If this flag is specified, then it also implies `FAULT_FLAG_ALLOW_RETRY`. This flag is checked in `faultin_page()`. See section 6 for details on how this flag impacts page faulting.
- **FOLL_NOFAULT** – Indicates that memory should not be faulted in, and if a fault were required, the `-EFAULT` error should instead be returned. Checked in `faultin_page()`.
- **FOLL_MIGRATION** – Causes migration logic to be invoked if a page table mapping is not valid, rather than immediately trying to fault folios in (or, if `FOLL_NOFAULT` is specified, return an error), try to wait for migration folios to be populated. This used by the *Kernel Samepage Merging (KSM)* functionality (out of scope for the book).
- **FOLL_REMOTE** – Indicates that the GUP actions are to be performed on a remote `mm_struct`. Set by `get_user_pages_remote()` and `pin_user_pages_remote()`. When this is set, arch-specific access checks invoked by `check_vma_flags()` faults have remote access enabled and any page faults are invoked with the `FAULT_FLAG_REMOTE` set.
- **FOLL_ANON** – Enforces that what is being pinned consists *only* of anonymous folios. This is checked in `check_vma_flags()` and returns a `-EFAULT` error if the VMA is not anonymous (as checked by `vma_is_anonymous()`).
- **FOLL_SPLIT_PMD** – Used as part of the huge page functionality, causing huge pages at the PMD level to be split into base pages. This is used by `make_device_exclusive_range()` and calls `split_huge_pmd()` from `follow_pmd_mask()`. We intentionally do not cover the huge page logic contained within GUP here in order to maintain brevity, see the chapter on huge page support for more details on page splitting.
- **FOLL_WRITE** – Check to ensure that the memory range is writeable (unless `FOLL_FORCE` overrides this, in which case we proceed regardless) and fault in. If `FOLL_TOUCH` is also specified set the page table dirty flag (in either `follow_page_pte()` or `follow_pfn_pte()`).
- **FOLL_NUMA** – If a page mapping is indicative of NUMA balancing being performed on the memory mapped, treat it as if the page is not mapped in order to trigger a page fault (see the NUMA chapter for more on NUMA balancing). Checked in `follow_page_pte()` and `follow_pmd_mask()`. This is set in `__get_user_pages()` for all cases other than when `FOLL_FORCE` is specified.
- **FOLL_TRIED** – Maps on to the fault flag `FAULT_FLAG_TRIED` and set when the `mmap_lock` has been released on faulting in pages (indicating that a `VM_FAULT_RETRY` or `VM_FAULT_COMPLETED` fault has arisen), and faulting in will be retried in `__get_user_pages_locked()`.
- **FOLL_FAST_ONLY** – Set by the functions `get_user_pages_fast_only()` and `pin_user_pages_fast_only()`. This prevents these functions from falling back to non-fast versions if a page fault is needed.

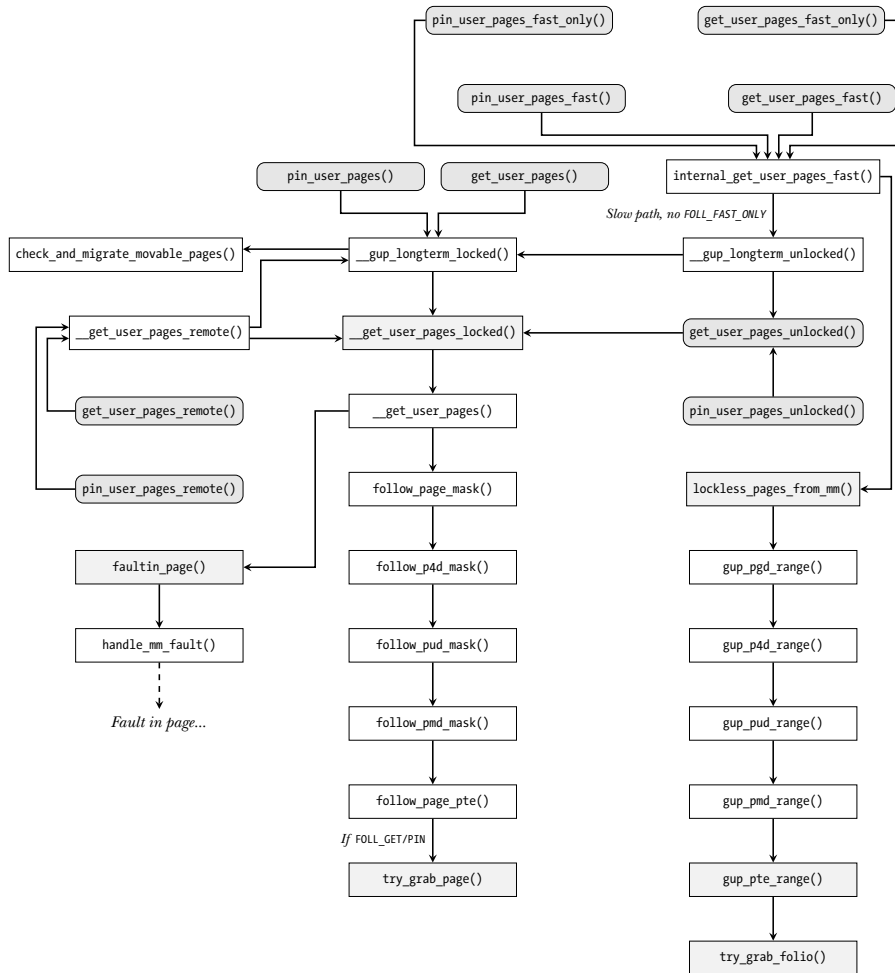


Figure 8-1: GUP functions

Note that:-

- Denotes non-static GUP functions.
- Denotes key functions.

8.1.5 GUP Functions

We will examine all of the code paths from figure 8-1 one-by-one to examine how the Get User Pages functionality logic actually operates.

There are two key code paths which all of the GUP functions take – for the `_fast()` functions, passing through `lockless_pages_from_mm()` and for the non-fast functions, passing through `_get_user_pages_locked()`. To start with, let's examine the non-fast GUP functions:-

```

2230 * get_user_pages() - pin user pages in memory
2231 * @start:      starting user address
2232 * @nr_pages:   number of pages from start to pin
2233 * @gup_flags:  flags modifying lookup behaviour
2234 * @pages:      array that receives pointers to the pages pinned.
2235 *              Should be at least nr_pages long. Or NULL, if caller
2236 *              only intends to ensure the pages are faulted in.
2237 * @vmas:       array of pointers to vmas corresponding to each page.
2238 *              Or NULL if the caller does not require them.
2239 *
2240 * This is the same as get_user_pages_remote(), just with a less-flexible
2241 * calling convention where we assume that the mm being operated on belongs to
2242 * the current task, and doesn't allow passing of a locked parameter. We also
2243 * obviously don't pass FOLL_REMOTE in here.
2244 */
2245 long get_user_pages(unsigned long start, unsigned long nr_pages,
2246                    unsigned int gup_flags, struct page **pages,
2247                    struct vm_area_struct **vmas)
2248 {
2249     if (!is_valid_gup_flags(gup_flags))
2250         return -EINVAL;
2251
2252     return __gup_longterm_locked(current->mm, start, nr_pages,
2253                                 pages, vmas, gup_flags | FOLL_TOUCH);
2254 }

```

Listing 8-15: mm/gup.c: `get_user_pages()`

This requires the `mmap_lock` read semaphore to be held on entry. It checks flags for the `get_*`() core functions before invoking `__gup_longterm_locked()` to do the heavy lifting, with `FOLL_TOUCH` set. Examining the check function, `is_valid_gup_flags()`:-

```

2089 static bool is_valid_gup_flags(unsigned int gup_flags)
2090 {
2091     /*
2092      * FOLL_PIN must only be set internally by the pin_user_pages*() APIs,
2093      * never directly by the caller, so enforce that with an assertion:
2094      */
2095     if (WARN_ON_ONCE(gup_flags & FOLL_PIN))
2096         return false;
2097     /*
2098      * FOLL_PIN is a prerequisite to FOLL_LONGTERM. Another way of saying
2099      * that is, FOLL_LONGTERM is a specific case, more restrictive case of
2100      * FOLL_PIN.
2101      */
2102     if (WARN_ON_ONCE(gup_flags & FOLL_LONGTERM))
2103         return false;
2104 }

```

```

2105     return true;
2106 }

```

Listing 8-16: mm/gup.c: is_valid_gup_flags()

This enforces the rule that FOLL_PIN and FOLL_LONGTERM flags can only be set by the pin_*() functions.

Examining the pin variant, pin_user_pages():-

```

3204 /**
3205  * pin_user_pages() - pin user pages in memory for use by other devices
3206  *
3207  * @start:      starting user address
3208  * @nr_pages:   number of pages from start to pin
3209  * @gup_flags:  flags modifying lookup behaviour
3210  * @pages:      array that receives pointers to the pages pinned.
3211  *              Should be at least nr_pages long.
3212  * @vmas:       array of pointers to vmas corresponding to each page.
3213  *              Or NULL if the caller does not require them.
3214  *
3215  * Nearly the same as get_user_pages(), except that FOLL_TOUCH is not set, and
3216  * FOLL_PIN is set.
3217  *
3218  * FOLL_PIN means that the pages must be released via unpin_user_page().
3219  * Please
3220  * see Documentation/core-api/pin_user_pages.rst for details.
3221  */
3222 long pin_user_pages(unsigned long start, unsigned long nr_pages,
3223                    unsigned int gup_flags, struct page **pages,
3224                    struct vm_area_struct **vmas)
3225 {
3226     /* FOLL_GET and FOLL_PIN are mutually exclusive. */
3227     if (WARN_ON_ONCE(gup_flags & FOLL_GET))
3228         return -EINVAL;
3229
3230     if (WARN_ON_ONCE(!pages))
3231         return -EINVAL;
3232
3233     gup_flags |= FOLL_PIN;
3234     return __gup_longterm_locked(current->mm, start, nr_pages,
3235                                pages, vmas, gup_flags);
3236 }

```

Listing 8-17: mm/gup.c: pin_user_pages()

This performs a check to ensure that FOLL_GET is not set, setting FOLL_PIN before again invoking the shared function __gup_longterm_locked() to do the heavy lifting.

Examining the remote variant, get_user_pages_remote():-

```

2138 /**
2139  * get_user_pages_remote() - pin user pages in memory
2140  * @mm:          mm_struct of target mm
2141  * @start:       starting user address
2142  * @nr_pages:    number of pages from start to pin
2143  * @gup_flags:   flags modifying lookup behaviour
2144  * @pages:       array that receives pointers to the pages pinned.
2145  *              Should be at least nr_pages long. Or NULL, if caller
2146  *              only intends to ensure the pages are faulted in.
2147  * @vmas:        array of pointers to vmAs corresponding to each page.
2148  *              Or NULL if the caller does not require them.
2149  * @locked:      pointer to lock flag indicating whether lock is held and
2150  *              subsequently whether VM_FAULT_RETRY functionality can be
2151  *              utilised. Lock must initially be held.
2152  *
2153  * Returns either number of pages pinned (which may be less than the
2154  * number requested), or an error. Details about the return value:
2155  *
2156  * -- If nr_pages is 0, returns 0.
2157  * -- If nr_pages is >0, but no pages were pinned, returns -errno.
2158  * -- If nr_pages is >0, and some pages were pinned, returns the number of
2159  *    pages pinned. Again, this may be less than nr_pages.
2160  *
2161  * The caller is responsible for releasing returned @pages, via put_page().
2162  *
2163  * @vmas are valid only as long as mmap_lock is held.
2164  *
2165  * Must be called with mmap_lock held for read or write.
2166  *
2167  * get_user_pages_remote walks a process's page tables and takes a reference
2168  * to each struct page that each user address corresponds to at a given
2169  * instant. That is, it takes the page that would be accessed if a user
2170  * thread accesses the given user virtual address at that instant.
2171  *
2172  * This does not guarantee that the page exists in the user mappings when
2173  * get_user_pages_remote returns, and there may even be a completely different
2174  * page there in some cases (eg. if mmapped pagecache has been invalidated
2175  * and subsequently re faulted). However it does guarantee that the page
2176  * won't be freed completely. And mostly callers simply care that the page
2177  * contains data that was valid *at some point in time*. Typically, an IO
2178  * or similar operation cannot guarantee anything stronger anyway because
2179  * locks can't be held over the syscall boundary.
2180  *
2181  * If gup_flags & FOLL_WRITE == 0, the page must not be written to. If the
2182  *    page
2183  * is written to, set_page_dirty (or set_page_dirty_lock, as appropriate) must
2184  * be called after the page is finished with, and before put_page is called.

```

```

2184 *
2185 * get_user_pages_remote is typically used for fewer-copy IO operations,
2186 * to get a handle on the memory by some means other than accesses
2187 * via the user virtual addresses. The pages may be submitted for
2188 * DMA to devices or accessed via their kernel linear mapping (via the
2189 * kmap APIs). Care should be taken to use the correct cache flushing APIs.
2190 *
2191 * See also get_user_pages_fast, for performance critical applications.
2192 *
2193 * get_user_pages_remote should be phased out in favor of
2194 * get_user_pages_locked/unlocked or get_user_pages_fast. Nothing
2195 * should use get_user_pages_remote because it cannot pass
2196 * FAULT_FLAG_ALLOW_RETRY to handle mm_fault.
2197 */
2198 long get_user_pages_remote(struct mm_struct *mm,
2199                          unsigned long start, unsigned long nr_pages,
2200                          unsigned int gup_flags, struct page **pages,
2201                          struct vm_area_struct **vmas, int *locked)
2202 {
2203     if (!is_valid_gup_flags(gup_flags))
2204         return -EINVAL;
2205
2206     return __get_user_pages_remote(mm, start, nr_pages, gup_flags,
2207                                   pages, vmas, locked);
2208 }

```

Listing 8-18: mm/gup.c: get_user_pages_remote()

This again checks the flags are valid for the `get_*`() set of GUP functions via `is_valid_gup_flags()` before invoking the shared remote function `__get_user_pages_remote()`.

Examining its pin equivalent, `pin_user_pages_remote()`:-

```

3164 /**
3165  * pin_user_pages_remote() - pin pages of a remote process
3166  *
3167  * @mm:      mm_struct of target mm
3168  * @start:    starting user address
3169  * @nr_pages: number of pages from start to pin
3170  * @gup_flags: flags modifying lookup behaviour
3171  * @pages:    array that receives pointers to the pages pinned.
3172  *            Should be at least nr_pages long.
3173  * @vmas:     array of pointers to vmas corresponding to each page.
3174  *            Or NULL if the caller does not require them.
3175  * @locked:    pointer to lock flag indicating whether lock is held and
3176  *            subsequently whether VM_FAULT_RETRY functionality can be
3177  *            utilised. Lock must initially be held.
3178  *

```

```

3179  * Nearly the same as get_user_pages_remote(), except that FOLL_PIN is set.
      See
3180  * get_user_pages_remote() for documentation on the function arguments,
      because
3181  * the arguments here are identical.
3182  *
3183  * FOLL_PIN means that the pages must be released via unpin_user_page().
      Please
3184  * see Documentation/core-api/pin_user_pages.rst for details.
3185  */
3186  long pin_user_pages_remote(struct mm_struct *mm,
3187                          unsigned long start, unsigned long nr_pages,
3188                          unsigned int gup_flags, struct page **pages,
3189                          struct vm_area_struct **vmas, int *locked)
3190  {
3191      /* FOLL_GET and FOLL_PIN are mutually exclusive. */
3192      if (WARN_ON_ONCE(gup_flags & FOLL_GET))
3193          return -EINVAL;
3194
3195      if (WARN_ON_ONCE(!pages))
3196          return -EINVAL;
3197
3198      gup_flags |= FOLL_PIN;
3199      return __get_user_pages_remote(mm, start, nr_pages, gup_flags,
3200                                  pages, vmas, locked);
3201  }

```

Listing 8-19: mm/gup.c: pin_user_pages_remote()

This checks to ensure that the invalid FOLL_GET is not set, that pages are specified, and then sets FOLL_PIN before invoking the shared remote function `__get_user_pages_remote()`.

Let's examine it now:-

```

2109  static long __get_user_pages_remote(struct mm_struct *mm,
2110                                  unsigned long start, unsigned long
2111                                  nr_pages,
2112                                  unsigned int gup_flags, struct page **
2113                                  pages,
2114                                  struct vm_area_struct **vmas, int *locked)
2115  {
2116      /*
2117       * Parts of FOLL_LONGTERM behavior are incompatible with
2118       * FAULT_FLAG_ALLOW_RETRY because of the FS DAX check requirement on
2119       * vmas. However, this only comes up if locked is set, and there are
2120       * callers that do request FOLL_LONGTERM, but do not set locked. So,
2121       * allow what we can.
2122       */
2123      if (gup_flags & FOLL_LONGTERM) {

```

```

2122         if (WARN_ON_ONCE(locked))
2123             return -EINVAL;
2124         /*
2125          * This will check the vmas (even if our vmas arg is NULL)
2126          * and return -ENOTSUPP if DAX isn't allowed in this case:
2127          */
2128         return __gup_longterm_locked(mm, start, nr_pages, pages,
2129                                     vmas, gup_flags | FOLL_TOUCH |
2130                                     FOLL_REMOTE);
2131     }
2132
2133     return __get_user_pages_locked(mm, start, nr_pages, pages, vmas,
2134                                   locked,
2135                                   gup_flags | FOLL_TOUCH | FOLL_REMOTE);
2136 }

```

Listing 8-20: mm/gup.c: `__get_user_pages_remote()`

If the `FOLL_LONGTERM` flag is set, this invokes `__gup_longterm_locked()`, otherwise it invokes `__get_user_pages_locked()` and in both cases `FOLL_TOUCH` and `FOLL_REMOTE` are set to indicate that folios should be touched and that a remote `mm_struct` is being accessed.

Since this function, as well as the others previously examined, reference `__gup_longterm_locked()`, let's examine this now (we will examine `__get_user_pages_locked()` shortly):-

```

2059 /*
2060  * __gup_longterm_locked() is a wrapper for __get_user_pages_locked which
2061  * allows us to process the FOLL_LONGTERM flag.
2062  */
2063 static long __gup_longterm_locked(struct mm_struct *mm,
2064                                  unsigned long start,
2065                                  unsigned long nr_pages,
2066                                  struct page **pages,
2067                                  struct vm_area_struct **vmas,
2068                                  unsigned int gup_flags)
2069 {
2070     unsigned int flags;
2071     long rc;
2072
2073     if (!(gup_flags & FOLL_LONGTERM))
2074         return __get_user_pages_locked(mm, start, nr_pages, pages,
2075                                       vmas,
2076                                       NULL, gup_flags);
2077     flags = memalloc_pin_save();
2078     do {
2079         rc = __get_user_pages_locked(mm, start, nr_pages, pages, vmas,
2080                                     NULL, gup_flags);
2081         if (rc <= 0)

```



```

2081             break;
2082             rc = check_and_migrate_movable_pages(rc, pages, gup_flags);
2083         } while (!rc);
2084         memalloc_pin_restore(flags);
2085
2086         return rc;
2087     }

```

Listing 8-21: mm/gup.c: __gup_longterm_locked()

Note that all invocations of `__get_user_pages_locked()` pass `NULL` as the `locked` parameter, which means that, if folios need to be faulted in, they cannot be retried with the `mmap_lock` being dropped.

If the `FOLL_LONGTERM` flag is not set, we simply invoke `__get_user_pages_locked()` otherwise, if this flag is set, we firstly invoke `memalloc_pin_save()` to set the process allocation flag `PF_MEMALLOC_PIN`.

This is used by `current_gfp_context()` to mask out the `__GFP_MOVABLE` allocation flag for all physical allocations for the process, meaning faulted in folios will be from a page block of non-movable memory. This flag is unset again after the loop completes via `memalloc_pin_restore()`.

In each loop iterate, if we either fail to pin any folios (no forward progress), or an error arises, we simply exit with that result. Otherwise, we invoke the function `check_and_migrate_movable_pages()` (eliding huge page and device coherent page code as out of scope):-

```

1929  /*
1930   * Check whether all pages are pinnable, if so return number of pages. If
1931   * some
1932   * pages are not pinnable, migrate them, and unpin all pages. Return zero if
1933   * pages were migrated, or if some pages were not successfully isolated.
1934   * Return negative error if migration fails.
1935   */
1936  static long check_and_migrate_movable_pages(unsigned long nr_pages,
1937                                              struct page **pages,
1938                                              unsigned int gup_flags)
1939  {
1940      unsigned long isolation_error_count = 0, i;
1941      struct folio *prev_folio = NULL;
1942      LIST_HEAD(movable_page_list);
1943      bool drain_allow = true, coherent_pages = false;
1944      int ret = 0;
1945
1946      for (i = 0; i < nr_pages; i++) {
1947          struct folio *folio = page_folio(pages[i]);
1948
1949          if (folio == prev_folio)
1950              continue;
1951          prev_folio = folio;
1952      }

```

```

1982         if (folio_is_longterm_pinnable(folio))
1983             continue;
1984         :
1994         if (!folio_test_lru(folio) && drain_allow) {
1995             lru_add_drain_all();
1996             drain_allow = false;
1997         }
1998
1999         if (folio_isolate_lru(folio)) {
2000             isolation_error_count++;
2001             continue;
2002         }
2003         list_add_tail(&folio->lru, &movable_page_list);
2004         node_stat_mod_folio(folio,
2005                             NR_ISOLATED_ANON + folio_is_file_lru(folio
2006                             ),
2007                             folio_nr_pages(folio));
2008     }
2009     if (!list_empty(&movable_page_list) || isolation_error_count ||
2010         coherent_pages)
2011         goto unpin_pages;
2012
2013     /*
2014      * If list is empty, and no isolation errors, means that all pages are
2015      * in the correct zone.
2016      */
2017     return nr_pages;

```

Listing 8-22: mm/gup.c: *check_and_migrate_movable_pages()* get movable pages

This is used to determine how many folios are actually long-term pinnable and, if any are not, *unpin* them all, before trying to migrate all non-movable folios.

This function has 3 different possible return values – `nr_pages`, the number of folios that were passed to it (i.e. the number that were able to be pinned by `__get_user_pages_locked()`), 0 if migration succeeded and a negative number if an error arose.

Therefore, if we succeed at migrating folios we try pinning them again in the next iteration of the loop.

The first part of this function tries to determine how many folios need migration, storing them in `movable_page_list` and performing the steps (examining distinct folios only):-

1. Check whether the folio is long-term pinnable via the function `folio_is_longterm_pinnable()`. If so, then examine the next folio.
2. If the folio is non-LRU, i.e. in a folio batch awaiting LRU designation, trigger an LRU drain (see section 11.7.12), but only once during the iteration in order to rate-limit this expensive operation.

3. Try to isolate the folio (remove it from all existing LRU lists) via `folio_isolate_lru()`*, if the isolation attempt fails (i.e. the folio has become non-LRU since the drain), the `isolation_error_count` is incremented.
4. Append the folio to the `movable_page_list` and update statistics.

If, in this process no folios are found that fail the `folio_is_longterm_pinnable()` test, we simply return the `nr_pages` to indicate that no action was taken. Otherwise we proceed with the unpinning.

Examining `folio_is_longterm_pinnable()`:-

```

1565 static inline bool folio_is_longterm_pinnable(struct folio *folio)
1566 {
1567     return is_longterm_pinnable_page(&folio->page);
1568 }

```

Listing 8-23: include/linux/mm.h: `folio_is_longterm_pinnable()`

Which invokes `is_longterm_pinnable_page()` (eliding out of scope CMA handling):-

```

1539 static inline bool is_longterm_pinnable_page(struct page *page)
1540 {
1541     :
1542     /* The zero page may always be pinned */
1543     if (is_zero_pfn(page_to_pfn(page)))
1544         return true;
1545
1546     /* Coherent device memory must always allow eviction. */
1547     if (is_device_coherent_page(page))
1548         return false;
1549
1550     /* Otherwise, non-movable zone pages can be pinned. */
1551     return !is_zone_movable_page(page);
1552 }

```

Listing 8-24: include/linux/mm.h: `is_longterm_pinnable_page()`

By default, the zero page is pinnable as by its nature it never moves. Device coherent memory (out of scope for the book) is absolutely not pinnable. Finally we are left with `is_zone_movable_page()`:-

```

811 static inline bool is_zone_movable_page(const struct page *page)
812 {
813     return page_zonenum(page) == ZONE_MOVABLE;
814 }

```

*. See the reclaim chapter for more details on `folio_isolate_lru()`, however broadly this pulls a page out of its current LRU list ready for it to be transferred into another. See section 11.2 for more on LRUs as a concept.

Listing 8-25: include/linux/mmzone.h: is_zone_movable_page()

For the majority of examined pages the need to migrate to an unmovable page block will be determined by whether they are located in ZONE_MOVABLE or not.

Once the list of movable pages is established in movable_page_list, we move on to attempting to migrate them:-

```

2019 unpin_pages:
2020     /*
2021      * pages[i] might be NULL if any device coherent pages were found.
2022      */
2023     for (i = 0; i < nr_pages; i++) {
2024         if (!pages[i])
2025             continue;
2026
2027         if (gup_flags & FOLL_PIN)
2028             unpin_user_page(pages[i]);
2029         else
2030             put_page(pages[i]);
2031     }
2032
2033     if (!list_empty(&movable_page_list)) {
2034         struct migration_target_control mtc = {
2035             .nid = NUMA_NO_NODE,
2036             .gfp_mask = GFP_USER | __GFP_NOWARN,
2037         };
2038
2039         ret = migrate_pages(&movable_page_list, alloc_migration_target
2040             ,
2041                             NULL, (unsigned long)&mtc, MIGRATE_SYNC,
2042                             MR_LONGTERM_PIN, NULL);
2043         if (ret > 0) /* number of pages not migrated */
2044             ret = -ENOMEM;
2045     }
2046
2047     if (ret && !list_empty(&movable_page_list))
2048         putback_movable_pages(&movable_page_list);
2049     return ret;
2050 }
```

Listing 8-26: mm/gup.c: check_and_migrate_movable_pages() unpin pages

This starts by relinquishing the reference count on each page acquired when invoking __get_user_pages_locked(), or if the page was GUP-pinned unpinned via unpin_user_page(). We are about to migrate these which requires them to be unpinned.

If we have at least some pages to move we then invoke `migrate_pages()` to do so (see the chapter on migration for details as to how this functions). Finally, if an error occurred on migration we put the pages back where they were via `putback_movable_pages()`.

8.1.5.1 Unlocked GUP Functions

We have now examined the key non-fast interfaces to the GUP functionality, however there are a few functions which are entered without the `struct mm_struct->mmap_lock` held, first of which is `__gup_longterm_unlocked()` which is invoked by `internal_get_user_pages_fast()` when the fast path cannot be used:-

```

2892 static int __gup_longterm_unlocked(unsigned long start, int nr_pages,
2893                                   unsigned int gup_flags, struct page **pages
2894                                   )
2895 {
2896     int ret;
2897
2898     /*
2899      * FIXME: FOLL_LONGTERM does not work with
2900      * get_user_pages_unlocked() (see comments in that function)
2901      */
2902     if (gup_flags & FOLL_LONGTERM) {
2903         mmap_read_lock(current->mm);
2904         ret = __gup_longterm_locked(current->mm,
2905                                     start, nr_pages,
2906                                     pages, NULL, gup_flags);
2907         mmap_read_unlock(current->mm);
2908     } else {
2909         ret = get_user_pages_unlocked(start, nr_pages,
2910                                       pages, gup_flags);
2911     }
2912     return ret;
2913 }

```

Listing 8-27: `mm/gup.c: __gup_longterm_unlocked()`

This simply invokes `__gup_longterm_locked()` when `FOLL_LONGTERM` is set (see listing 8-21) with a newly acquired read lock on the `mmap_lock` semaphore, otherwise it invokes the more general exported `get_user_pages_unlocked()` function:-

```

2257 /*
2258  * get_user_pages_unlocked() is suitable to replace the form:
2259  *
2260  *     mmap_read_lock(mm);
2261  *     get_user_pages(mm, ..., pages, NULL);

```

```

2262 *      mmap_read_unlock(mm);
2263 *
2264 * with:
2265 *
2266 *      get_user_pages_unlocked(mm, ..., pages);
2267 *
2268 * It is functionally equivalent to get_user_pages_fast so
2269 * get_user_pages_fast should be used instead if specific gup_flags
2270 * (e.g. FOLL_FORCE) are not required.
2271 */
2272 long get_user_pages_unlocked(unsigned long start, unsigned long nr_pages,
2273                             struct page **pages, unsigned int gup_flags)
2274 {
2275     struct mm_struct *mm = current->mm;
2276     int locked = 1;
2277     long ret;
2278
2279     /*
2280      * FIXME: Current FOLL_LONGTERM behavior is incompatible with
2281      * FAULT_FLAG_ALLOW_RETRY because of the FS DAX check requirement on
2282      * vmas. As there are no users of this flag in this call we simply
2283      * disallow this option for now.
2284      */
2285     if (WARN_ON_ONCE(gup_flags & FOLL_LONGTERM))
2286         return -EINVAL;
2287
2288     mmap_read_lock(mm);
2289     ret = __get_user_pages_locked(mm, start, nr_pages, pages, NULL,
2290                                  &locked, gup_flags | FOLL_TOUCH);
2291     if (locked)
2292         mmap_read_unlock(mm);
2293     return ret;
2294 }

```

Listing 8-28: mm/gup.c: `get_user_pages_unlocked()`

Again, this defers the heavy lifting to `__get_user_pages_locked()` after checking that `FOLL_LONGTERM` is not set (as this should be handled by `__gup_longterm_unlocked()` instead). Note that fault handling can result in the `mmap_lock` being released so that is tracked and we only attempt to unlock if the lock is still held (see section 6 for more on page faulting).

Finally, the GUP-pinned variant of this function, `pin_user_pages_unlocked()` mirrors its locked version `pin_user_pages()` in that it asserts that pages must be specified, that `FOLL_GET` cannot be and sets `FOLL_PIN` if not already specified before invoking `get_user_pages_unlocked()` to do the heavy lifting:-

```

3238 /*
3239  * pin_user_pages_unlocked() is the FOLL_PIN variant of
3240  * get_user_pages_unlocked(). Behavior is the same, except that this one sets

```

```

3241  * FOLL_PIN and rejects FOLL_GET.
3242  */
3243  long pin_user_pages_unlocked(unsigned long start, unsigned long nr_pages,
3244                               struct page **pages, unsigned int gup_flags)
3245  {
3246      /* FOLL_GET and FOLL_PIN are mutually exclusive. */
3247      if (WARN_ON_ONCE(gup_flags & FOLL_GET))
3248          return -EINVAL;
3249
3250      if (WARN_ON_ONCE(!pages))
3251          return -EINVAL;
3252
3253      gup_flags |= FOLL_PIN;
3254      return get_user_pages_unlocked(start, nr_pages, pages, gup_flags);
3255  }

```

Listing 8-29: mm/gup.c: pin_user_pages_unlocked()

8.1.5.2 Core Non-Fast GUP Functions

All non-fast GUP functions eventually invoke `__get_user_pages_locked()`. The bulk of this function is dedicated to handling fault retrying logic, which occurs where the lock parameter is non-NULL. Examining the start of the function, which is the entirety of the logic should the caller not wish to retry:-

```

1383  /*
1384   * Please note that this function, unlike __get_user_pages will not
1385   * return 0 for nr_pages > 0 without FOLL_NOWAIT
1386   */
1387  static __always_inline long __get_user_pages_locked(struct mm_struct *mm,
1388                                                       unsigned long start,
1389                                                       unsigned long nr_pages,
1390                                                       struct page **pages,
1391                                                       struct vm_area_struct **vmas,
1392                                                       int *locked,
1393                                                       unsigned int flags)
1394  {
1395      long ret, pages_done;
1396      bool lock_dropped;
1397
1398      if (locked) {
1399          /* if VM_FAULT_RETRY can be returned, vmas become invalid */
1400          BUG_ON(vmas);
1401          /* check caller initialized locked */
1402          BUG_ON(*locked != 1);
1403      }
1404
1405      if (flags & FOLL_PIN)

```

```

1406         mm_set_has_pinned_flag(&mm->flags);
1407
1408     /*
1409      * FOLL_PIN and FOLL_GET are mutually exclusive. Traditional behavior
1410      * is to set FOLL_GET if the caller wants pages[] filled in (but has
1411      * carelessly failed to specify FOLL_GET), so keep doing that, but
1412      * only
1413      * for FOLL_GET, not for the newer FOLL_PIN.
1414      *
1415      * FOLL_PIN always expects pages to be non-null, but no need to assert
1416      * that here, as any failures will be obvious enough.
1417      */
1418     if (pages && !(flags & FOLL_PIN))
1419         flags |= FOLL_GET;
1420
1421     pages_done = 0;
1422     lock_dropped = false;
1423     for (;;) {
1424         ret = __get_user_pages(mm, start, nr_pages, flags, pages,
1425                                vmas, locked);
1426         if (!locked)
1427             /* VM_FAULT_RETRY couldn't trigger, bypass */
1428             return ret;

```

Listing 8-30: mm/gup.c: `__get_user_pages_locked()` non-retry logic

We perform some sanity checks, set `FOLL_GET` if pages are specified but this flag was not before performing the heavy lifting in `__get_user_pages()`. We also set the `MMF_HAS_PINNED` flag for the struct `mm_struct` object as described previously if the page is to be pinned.

The lock parameter determines whether the page faulting mechanism returning `VM_FAULT_RETRY` or `VM_FAULT_COMPLETED` can be handled. These are returned when the `FAULT_FLAG_ALLOW_RETRY` flag is specified to the fault-handler (see section 6 for more on page faulting and the page cache chapter for more on this specific behaviour) when a file-backed folio is locked typically for write-back.

When this flag is set and the folio lock can't be obtained, the struct `mm_struct->mmap_lock` lock is dropped in order to not have to block on the writeback operation and to make forward progress without having to wait for the lock to be released, in order to reduce contention on the `mmap_lock`.

Only functions which can handle the lock being dropped are able to do this, specifically `__get_user_pages_remote()` and `get_user_pages_unlocked()` and their callers— see figure 8-1 to see which functions these are, others simply pass `NULL` as the lock parameter and thus this logic is skipped in these cases.

Examining the remainder of the function (reiterating the start of the loop for clarity) which handles this case:-

```

1422     for (;;) {
1423         ret = __get_user_pages(mm, start, nr_pages, flags, pages,

```



```

1424                                     vmas, locked);
1425                                     :
1426                                     /* VM_FAULT_RETRY or VM_FAULT_COMPLETED cannot return errors
1427                                     */
1428
1429                                     if (!*locked) {
1430                                         BUG_ON(ret < 0);
1431                                         BUG_ON(ret >= nr_pages);
1432                                     }
1433
1434                                     if (ret > 0) {
1435                                         nr_pages -= ret;
1436                                         pages_done += ret;
1437                                         if (!nr_pages)
1438                                             break;
1439                                     }
1440                                     if (*locked) {
1441                                         /*
1442                                         * VM_FAULT_RETRY didn't trigger or it was a
1443                                         * FOLL_NOWAIT.
1444                                         */
1445                                         if (!pages_done)
1446                                             pages_done = ret;
1447                                         break;
1448                                     }
1449                                     /*
1450                                     * VM_FAULT_RETRY triggered, so seek to the faulting offset.
1451                                     * For the prefault case (!pages) we only update counts.
1452                                     */
1453                                     if (likely(pages))
1454                                         pages += ret;
1455                                     start += ret << PAGE_SHIFT;
1456                                     lock_dropped = true;
1457
1458                                     retry:
1459                                     /*
1460                                     * Repeat on the address that fired VM_FAULT_RETRY
1461                                     * with both FAULT_FLAG_ALLOW_RETRY and
1462                                     * FAULT_FLAG_TRIED. Note that GUP can be interrupted
1463                                     * by fatal signals, so we need to check it before we
1464                                     * start trying again otherwise it can loop forever.
1465                                     */
1466
1467                                     if (fatal_signal_pending(current)) {
1468                                         if (!pages_done)
1469                                             pages_done = -EINTR;
1470                                         break;
1471                                     }
1472

```

```

1473
1474         ret = mmap_read_lock_killable(mm);
1475         if (ret) {
1476             BUG_ON(ret > 0);
1477             if (!pages_done)
1478                 pages_done = ret;
1479             break;
1480         }
1481
1482         *locked = 1;
1483         ret = __get_user_pages(mm, start, 1, flags | FOLL_TRIED,
1484                               pages, NULL, locked);
1485         if (!*locked) {
1486             /* Continue to retry until we succeeded */
1487             BUG_ON(ret != 0);
1488             goto retry;
1489         }
1490         if (ret != 1) {
1491             BUG_ON(ret > 1);
1492             if (!pages_done)
1493                 pages_done = ret;
1494             break;
1495         }
1496         nr_pages--;
1497         pages_done++;
1498         if (!nr_pages)
1499             break;
1500         if (likely(pages))
1501             pages++;
1502         start += PAGE_SIZE;
1503     }
1504     if (lock_dropped && *locked) {
1505         /*
1506          * We must let the caller know we temporarily dropped the lock
1507          * and so the critical section protected by it was lost.
1508          */
1509         mmap_read_unlock(mm);
1510         *locked = 0;
1511     }
1512     return pages_done;
1513 }

```

Listing 8-31: mm/gup.c: `__get_user_pages_locked()` retry logic

The logic is as follows:-

1. Attempt to process `nr_pages` pages started at address `start` via `__get_user_pages()`.

2. Assert that neither an error occurred nor do we appear to have processed more pages than expected (`__get_user_pages()` returns either a positive number of pages processed or a negative error).
3. Update `nr_pages` and `pages_done` counts and if we have processed sufficient pages, exit the loop.
4. We are now in the position where pages remain to be processed – if the lock has not been dropped then we are not in the retry logic and thus must exit.
5. If the pages array of struct page pointers was provided, then we update the variable to point to the offset position based on how many pages have been processed.
Whether this was specified or not, offset start and set `lock_dropped` so we know to drop the lock before returning so the caller is aware the lock was dropped at least once even if it is later reacquired.
6. Check to see if a fatal signal is pending via `fatal_signal_pending()`. If so, then continuing this operation is not useful and we should exit the loop.
7. Attempt to reacquire a read lock on the struct `mm_struct->mmap_lock` semaphore via `mmap_read_lock_killable()`. If this fails, we return the error code.
8. Try to process (and thus fault in) one page via `__get_user_pages()` with `FOLL_TRIED` set to indicate that we have retried once. If we have to retry again (i.e. the lock has been dropped again), we keep retrying this operation from step 6 until we succeed or an error occurs.
9. On success, update the counts and offsets accordingly and if we are done, exit the loop, otherwise repeat the whole operation from step 1 again until we are done or an error occurs.

Now we will examine the core non-fast handler, `__get_user_pages()`, starting with the rather complete comment describing it and initial checks and initialisation:-

```

1080 /**
1081  * __get_user_pages() - pin user pages in memory
1082  * @mm:           mm_struct of target mm
1083  * @start:        starting user address
1084  * @nr_pages:     number of pages from start to pin
1085  * @gup_flags:    flags modifying pin behaviour
1086  * @pages:        array that receives pointers to the pages pinned.
1087  *                Should be at least nr_pages long. Or NULL, if caller
1088  *                only intends to ensure the pages are faulted in.
1089  * @vmas:         array of pointers to vmas corresponding to each page.
1090  *                Or NULL if the caller does not require them.
1091  * @locked:       whether we're still with the mmap_lock held
1092  *
1093  * Returns either number of pages pinned (which may be less than the
1094  * number requested), or an error. Details about the return value:

```

```

1095 *
1096 * -- If nr_pages is 0, returns 0.
1097 * -- If nr_pages is >0, but no pages were pinned, returns -errno.
1098 * -- If nr_pages is >0, and some pages were pinned, returns the number of
1099 *    pages pinned. Again, this may be less than nr_pages.
1100 * -- 0 return value is possible when the fault would need to be retried.
1101 *
1102 * The caller is responsible for releasing returned @pages, via put_page().
1103 *
1104 * @vmass are valid only as long as mmap_lock is held.
1105 *
1106 * Must be called with mmap_lock held. It may be released. See below.
1107 *
1108 * __get_user_pages walks a process's page tables and takes a reference to
1109 * each struct page that each user address corresponds to at a given
1110 * instant. That is, it takes the page that would be accessed if a user
1111 * thread accesses the given user virtual address at that instant.
1112 *
1113 * This does not guarantee that the page exists in the user mappings when
1114 * __get_user_pages returns, and there may even be a completely different
1115 * page there in some cases (eg. if mmapped pagecache has been invalidated
1116 * and subsequently re faulted). However it does guarantee that the page
1117 * won't be freed completely. And mostly callers simply care that the page
1118 * contains data that was valid *at some point in time*. Typically, an IO
1119 * or similar operation cannot guarantee anything stronger anyway because
1120 * locks can't be held over the syscall boundary.
1121 *
1122 * If @gup_flags & FOLL_WRITE == 0, the page must not be written to. If
1123 * the page is written to, set_page_dirty (or set_page_dirty_lock, as
1124 * appropriate) must be called after the page is finished with, and
1125 * before put_page is called.
1126 *
1127 * If @locked != NULL, *@locked will be set to 0 when mmap_lock is
1128 * released by an up_read(). That can happen if @gup_flags does not
1129 * have FOLL_NOWAIT.
1130 *
1131 * A caller using such a combination of @locked and @gup_flags
1132 * must therefore hold the mmap_lock for reading only, and recognize
1133 * when it's been released. Otherwise, it must be held for either
1134 * reading or writing and will not be released.
1135 *
1136 * In most cases, get_user_pages or get_user_pages_fast should be used
1137 * instead of __get_user_pages. __get_user_pages should be used only if
1138 * you need some special @gup_flags.
1139 */
1140 static long __get_user_pages(struct mm_struct *mm,
1141                             unsigned long start, unsigned long nr_pages,

```

```

1142         unsigned int gup_flags, struct page **pages,
1143         struct vm_area_struct **vmas, int *locked)
1144     {
1145         long ret = 0, i = 0;
1146         struct vm_area_struct *vma = NULL;
1147         struct follow_page_context ctx = { NULL };
1148
1149         if (!nr_pages)
1150             return 0;
1151
1152         start = untagged_addr(start);
1153
1154         VM_BUG_ON(!pages != !(gup_flags & (FOLL_GET | FOLL_PIN)));
1155
1156         /*
1157          * If FOLL_FORCE is set then do not force a full fault as the hinting
1158          * fault information is unrelated to the reference behaviour of a task
1159          * using the address space
1160          */
1161         if (!(gup_flags & FOLL_FORCE))
1162             gup_flags |= FOLL_NUMA;

```

Listing 8-32: mm/gup.c: `__get_user_pages()` initialisation

This starts by initialising local variables and handling the trivial case where `nr_pages` is zero. The invocation of `untagged_addr()` is only relevant to architectures which tag addresses (x86-64, the architecture we examine in this book does not).

The key points here are that – if pages are specified then either `FOLL_GET` or `FOLL_PIN` must be specified, equally if either of these flags are specified, then pages must be too (you cannot unpin pages without having a reference to them).

Finally, `FOLL_NUMA` is specified for all cases except `FOLL_FORCE` which causes the GUP code to trigger a full fault when NUMA balancing is indicated*. Examining the remainder of the function which comprises its main loop (eliding out of scope gate area, huge page, device page map and manual cache maintenance logic):-

```

1164         do {
1165             struct page *page;
1166             unsigned int foll_flags = gup_flags;
1167             unsigned int page_increm;
1168
1169             /* first iteration or cross vma bound */
1170             if (!vma || start >= vma->vm_end) {

```

*. NUMA balancing is a process by which the kernel periodically unmaps memory and remaps it on a local node if it was not already.

```

1171         vma = find_extend_vma(mm, start);
1172         :
1182         if (!vma) {
1183             ret = -EFAULT;
1184             goto out;
1185         }
1186         ret = check_vma_flags(vma, gup_flags);
1187         if (ret)
1188             goto out;
1189         :
1205     }
1206 retry:
1207     /*
1208      * If we have a pending SIGKILL, don't keep faulting pages and
1209      * potentially allocating memory.
1210      */
1211     if (fatal_signal_pending(current)) {
1212         ret = -EINTR;
1213         goto out;
1214     }
1215     cond_resched();
1216
1217     page = follow_page_mask(vma, start, foll_flags, &ctx);
1218     if (!page || PTR_ERR(page) == -EMLINK) {
1219         ret = faultin_page(vma, start, &foll_flags,
1220                           PTR_ERR(page) == -EMLINK, locked);
1221         switch (ret) {
1222             case 0:
1223                 goto retry;
1224             case -EBUSY:
1225             case -EAGAIN:
1226                 ret = 0;
1227                 fallthrough;
1228             case -EFAULT:
1229             case -ENOMEM:
1230             case -EHWPOISON:
1231                 goto out;
1232         }
1233         BUG();
1234     } else if (PTR_ERR(page) == -EEXIST) {
1235         /*
1236             * Proper page table entry exists, but no
1237             * corresponding
1238             * struct page. If the caller expects **pages to be
1239             * filled in, bail out now, because that can't be done
1240             * for this page.
1241         */

```

```

1241         if (pages) {
1242             ret = PTR_ERR(page);
1243             goto out;
1244         }
1245
1246         goto next_page;
1247     } else if (IS_ERR(page)) {
1248         ret = PTR_ERR(page);
1249         goto out;
1250     }
1251     if (pages) {
1252         pages[i] = page;
1253         :
1254     }
1255 next_page:
1256     if (vmass) {
1257         vmass[i] = vma;
1258         :
1259     }
1260     page_increm = 1 + (~(start >> PAGE_SHIFT) & ctx.page_mask);
1261     :
1262     i += page_increm;
1263     start += page_increm * PAGE_SIZE;
1264     nr_pages -= page_increm;
1265 } while (nr_pages);
1266 out:
1267 :
1268 return i ? i : ret;
1269

```

Listing 8-33: mm/gup.c: __get_user_pages() main loop

The logic is as follows:-

1. If necessary, retrieve the current VMA containing start via `find_extend_vma()`. This either locates the VMA containing the address, or extends the stack if the range is within the stack. If so, we check that the VMA flags are consistent via `check_vma_flags()` which we will examine shortly below. If we cannot find a VMA or the check fails we return an error.
2. If a fatal signal is pending we exit early with an `EINTR` error code. We check this via `fatal_signal_pending()`. We also indicate that the scheduler can deschedule us in favour of a higher priority task via `cond_resched()` as at this stage we're well placed to do so (pertinent if the kernel is not running with full preemption).
3. We walk the page tables for the specified address via `follow_page_mask()`, this either returns a pointer to its struct page object, NULL if the page is not faulted in, or an error if one occurs. We discuss the details of this function in section 8.1.6 below.

4. If either the page was not faulted in or an EMLINK error occurred, indicating that a shared mapping must be unshared (more on this in section 8.1.6), then we try to fault in the page via `faultin_page()`, discussed below in section 8.1.7.
5. If the faulting in succeeds, we loop to step 2 and repeat the process again. Note that a race could occur between pages being retrieved from the swap or migration or NUMA balancing or any other such operation and this GUP function so we could loop multiple times here.
6. If the faulting in fails with either `EBUSY` (indicates `VM_FAULT_RETRY` was returned by the faulting logic) or `EAGAIN` (indicates `VM_FAULT_COMPLETED` was returned by the faulting logic) then we simply exit so the process can be tried again by the caller, otherwise the return value is set to the error code.
7. If `follow_page_mask()` returned the `EEXIST` error this indicates that the entry is valid but there is no `struct page` associated with the entry. If the user expects pages to be returned, something we now simply cannot do, then we error out, otherwise we simply move on to the next page (see section 6.12 for more details on this kind of mapping.)
8. If the follow operation simply resulted in an error then set the return value to that error code and exit.
9. If we succeeded in retrieving a page, place it in the pages array if it exists.
10. If the user requires the VMA to be returned, place a pointer to the VMA in the supplied array. Note that these will only be valid as long as a lock on the `struct mm_struct->mmap_lock` semaphore is held.
11. We increment `i`, which counts the number of base pages we have processed, by one. Note that here the logic seems more complicated even with parts of the code elided – this is to handle huge page logic which we exclude from this discussion for the sake of brevity. In non-huge page cases `page_increm` will always be 1. We additionally increment `start` by a base page size and decrement `nr_pages` by 1.
12. Finally, if any pages were processed, we return the count, otherwise we return an error if any occurred or zero if we could simply not progress. We prioritise indicating how many pages were pinned or had actions performed upon them so the caller can retry or otherwise respond to the fact that others were not processed as they intend.

Returning to `check_vma_flags()` (eliding out of scope DAX and secret memory logic):-

```

1027 static int check_vma_flags(struct vm_area_struct *vma, unsigned long gup_flags
1028                             )
1029 {
1030     vm_flags_t vm_flags = vma->vm_flags;
1031     int write = (gup_flags & FOLL_WRITE);
1032     int foreign = (gup_flags & FOLL_REMOTE);

```



```

1032
1033     if (vm_flags & (VM_IO | VM_PFNMAP))
1034         return -EFAULT;
1035
1036     if (gup_flags & FOLL_ANON && !vma_is_anonymous(vma))
1037         return -EFAULT;
1038     :
1045     if (write) {
1046         if (!(vm_flags & VM_WRITE)) {
1047             if (!(gup_flags & FOLL_FORCE))
1048                 return -EFAULT;
1049             /*
1050              * We used to let the write,force case do COW in a
1051              * VM_MAYWRITE VM_SHARED !VM_WRITE vma, so ptrace
could
1052                  * set a breakpoint in a read-only mapping of an
1053                  * executable, without corrupting the file (yet only
1054                  * when that file had been opened for writing!).
1055                  * Anon pages in shared mappings are surprising: now
1056                  * just reject it.
1057                  */
1058             if (!is_cow_mapping(vm_flags))
1059                 return -EFAULT;
1060         }
1061     } else if (!(vm_flags & VM_READ)) {
1062         if (!(gup_flags & FOLL_FORCE))
1063             return -EFAULT;
1064         /*
1065          * Is there actually any vma we can reach here which does not
1066          * have VM_MAYREAD set?
1067          */
1068         if (!(vm_flags & VM_MAYREAD))
1069             return -EFAULT;
1070     }
1071     /*
1072      * gups are always data accesses, not instruction
1073      * fetches, so execute=false here
1074      */
1075     if (!arch_vma_access_permitted(vma, write, false, foreign))
1076         return -EFAULT;
1077     return 0;
1078 }

```

Listing 8-34: mm/gup.c: `check_vma_flags()`

This performs sanity checks on the struct `vm_area_struct` (VMA)’s flags given the specified GUP flags, including permissions checks.

An error is returned if:-

- Either `VM_IO` or `VM_PFNMAP` are specified, indicating that this is a raw mapping of device memory or a mapping from kernel memory respectively. These are not regions of memory that can be sanely interacted with via GUP.
- The `FOLL_ANON` GUP flag has been specified but the VMA is not anonymous.
- The VMA is read-only (i.e. does not have the `VM_WRITE` flag set), but `FOLL_WRITE` is specified (unless `FOLL_FORCE` overrides this). If this is a potentially-writable shared mapping (as determined by `is_cow_mapping()`) then we ignore `FOLL_FORCE` and error out anyway as indicated by the comment.
- The VMA does not permit reading (i.e. the `VM_READ` flag is cleared) and `FOLL_FORCE` is not specified. If `FOLL_FORCE` is specified but the mapping can never become readable (i.e. does not have `VM_MAYREAD` set) then we error out regardless.
- The architecture-specific check `arch_vma_access_permitted()` fails.

8.1.6 Walking Page Tables

In order to obtain the underlying physical pages below a userland page table range, we must walk the page tables at each level manually. This process is performed by a series of functions which operate at each level, beginning at the PGD level with `follow_page_mask()` (eliding out of scope huge page and `ZONE_DEVICE` memory):-

```

820 /**
821  * follow_page_mask - look up a page descriptor from a user-virtual address
822  * @vma: vm_area_struct mapping @address
823  * @address: virtual address to look up
824  * @flags: flags modifying lookup behaviour
825  * :
826  * @flags can have FOLL_ flags set, defined in <linux/mm.h>
827  * :
828  * When getting an anonymous page and the caller has to trigger unsharing
829  * of a shared anonymous page first, -EMLINK is returned. The caller should
830  * trigger a fault with FAULT_FLAG_UNSHARE set. Note that unsharing is only
831  * relevant with FOLL_PIN and !FOLL_WRITE.
832  * :
833  * Return: the mapped (struct page *), %NULL if no mapping exists, or
834  * an error pointer if there is a mapping to something not represented
835  * by a page descriptor (see also vm_normal_page()).
836  */
837
838 static struct page *follow_page_mask(struct vm_area_struct *vma,
839                                     unsigned long address, unsigned int flags,
840                                     struct follow_page_context *ctx)

```

```

847 {
848     pgd_t *pgd;
849     struct page *page;
850     struct mm_struct *mm = vma->vm_mm;
851     :
861     pgd = pgd_offset(mm, address);
862
863     if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
864         return no_page_table(vma, flags);
865     :
881     return follow_p4d_mask(vma, address, pgd, flags, ctx);
882 }

```

Listing 8-35: mm/gup.c: `follow_page_mask()`

On each occasion when a page table entry can't be found, `no_page_table()` is invoked. This handles an edge case around core dumping:-

```

444 static struct page *no_page_table(struct vm_area_struct *vma,
445                                   unsigned int flags)
446 {
447     /*
448      * When core dumping an enormous anonymous area that nobody
449      * has touched so far, we don't want to allocate unnecessary pages or
450      * page tables. Return error instead of NULL to skip handle_mm_fault,
451      * then get_dump_page() will return NULL to leave a hole in the dump.
452      * But we can only make this optimization where a hole would surely
453      * be zero-filled if handle_mm_fault() actually did handle it.
454      */
455     if ((flags & FOLL_DUMP) &&
456         (vma_is_anonymous(vma) || !vma->vm_ops->fault))
457         return ERR_PTR(-EFAULT);
458     return NULL;
459 }

```

Listing 8-36: mm/gup.c: `no_page_table()`

At each of the page table levels this is invoked when an entry cannot be found (indicating that the memory needs to be faulted in).

Examining `follow_p4d_mask()` (eliding huge page handling):-

```

794 static struct page *follow_p4d_mask(struct vm_area_struct *vma,
795                                     unsigned long address, pgd_t *pgdp,
796                                     unsigned int flags,
797                                     struct follow_page_context *ctx)
798 {
799     p4d_t *p4d;
800     struct page *page;
801
802     p4d = p4d_offset(pgdp, address);

```

```

803         if (p4d_none(*p4d))
804             return no_page_table(vma, flags);
805         :
806         if (unlikely(p4d_bad(*p4d)))
807             return no_page_table(vma, flags);
808         :
817         return follow_pud_mask(vma, address, p4d, flags, ctx);
818     }

```

Listing 8-37: mm/gup.c: follow_p4d_mask()

This, as with all of the other page table walking logic, simply uses the `p*d_offset()` functions to access each page table level (see the virtual memory chapter for more on this). Similarly, `follow_pud_mask()` (eliding out of scope huge page and device map logic):-

```

754 static struct page *follow_pud_mask(struct vm_area_struct *vma,
755                                     unsigned long address, pud_t *p4dp,
756                                     unsigned int flags,
757                                     struct follow_page_context *ctx)
758 {
759     pud_t *pud;
760     :
761     struct page *page;
762     struct mm_struct *mm = vma->vm_mm;
763
764     pud = pud_offset(p4dp, address);
765     if (pud_none(*pud))
766         return no_page_table(vma, flags);
767     :
768     if (unlikely(pud_bad(*pud)))
769         return no_page_table(vma, flags);
770
771     return follow_pmd_mask(vma, address, pud, flags, ctx);
772 }

```

Listing 8-38: mm/gup.c: follow_pud_mask()

Examining `follow_pmd_mask()` (eliding out of scope huge page logic, huge page migration logic, device map logic):-

```

646 static struct page *follow_pmd_mask(struct vm_area_struct *vma,
647                                     unsigned long address, pud_t *pudp,
648                                     unsigned int flags,
649                                     struct follow_page_context *ctx)
650 {
651     pmd_t *pmd, pmdval;
652     spinlock_t *ptl;
653     struct page *page;
654     struct mm_struct *mm = vma->vm_mm;

```

```

655
656     pmd = pmd_offset(pudp, address);
657     :
661     pmdval = READ_ONCE(*pmd);
662     if (pmd_none(pmdval))
663         return no_page_table(vma, flags);
664     :
707     if (likely(!pmd_trans_huge(pmdval)))
708         return follow_page_pte(vma, address, pmd, flags, &ctx->pgmap);
709     :
752 }

```

Listing 8-39: mm/gup.c: follow_pmd_mask()

Note that the `pmd_bad()` check is deferred to `follow_page_pte()`, which we examine now, eliding out of scope Kernel Same page Merging (KSM) and device mapping logic:-

```

519 static struct page *follow_page_pte(struct vm_area_struct *vma,
520                                     unsigned long address, pmd_t *pmd, unsigned int flags,
521                                     struct dev_pagemap **pgmap)
522 {
523     struct mm_struct *mm = vma->vm_mm;
524     struct page *page;
525     spinlock_t *ptl;
526     pte_t *ptep, pte;
527     int ret;
528
529     /* FOLL_GET and FOLL_PIN are mutually exclusive. */
530     if (WARN_ON_ONCE((flags & (FOLL_PIN | FOLL_GET)) ==
531                      (FOLL_PIN | FOLL_GET)))
532         return ERR_PTR(-EINVAL);
533     :
534     if (unlikely(pmd_bad(*pmd)))
535         return no_page_table(vma, flags);
536
537     ptep = pte_offset_map_lock(mm, pmd, address, &ptl);
538     pte = *ptep;
539     if (!pte_present(pte)) {
540         :
546         if (likely(!(flags & FOLL_MIGRATION)))
547             goto no_page;
548         if (pte_none(pte))
549             goto no_page;
550         :
556     }
557     if ((flags & FOLL_NUMA) && pte_protnone(pte))
558         goto no_page;
559

```

```
560         page = vm_normal_page(vma, address, pte);
```

Listing 8-40: mm/gup.c: follow_page_pte() page lookup

We start by asserting that we do not simultaneously specify both `FOLL_PIN` and `FOLL_GET`, before performing the deferred `pmd_bad()` check, acquiring the PTE lock via `pte_offset_map_lock()` (see the virtual memory chapter for more on this) before checking whether the PTE entry is present and exiting if not (eliding out of scope migration logic).

Finally we consider the NUMA hinting case, checked by `pte_protnone()`, in this instance no GUP should be performed and it should be treated as if the page could not be found.

Finally, we obtain the underlying page via `vm_normal_page()` (see section 6.12 for details).

Next, we perform a series of checks on the page:-

```
562         /*
563          * We only care about anon pages in can_follow_write_pte() and don't
564          * have to worry about pte_devmap() because they are never anon.
565          */
566         if ((flags & FOLL_WRITE) &&
567             !can_follow_write_pte(pte, page, vma, flags)) {
568             page = NULL;
569             goto out;
570         }
571         :
583     } else if (unlikely(!page)) {
584         if (flags & FOLL_DUMP) {
585             /* Avoid special (like zero) pages in core dumps */
586             page = ERR_PTR(-EFAULT);
587             goto out;
588         }
589
590         if (is_zero_pfn(pte_pfn(pte))) {
591             page = pte_page(pte);
592         } else {
593             ret = follow_pfn_pte(vma, address, ptep, flags);
594             page = ERR_PTR(ret);
595             goto out;
596         }
597     }
598
599     if (!pte_write(pte) && gup_must_unshare(flags, page)) {
600         page = ERR_PTR(-EMLINK);
601         goto out;
602     }
603
604     VM_BUG_ON_PAGE((flags & FOLL_PIN) && PageAnon(page) &&
605                    !PageAnonExclusive(page), page);
```

Listing 8-41: mm/gup.c: follow_page_pte() page checks

Firstly, we determine whether we are permitted to write fault the memory if the GUP write flag `FOLL_WRITE` is set. This is done via `can_follow_write_pte()`:-

```

481 /* FOLL_FORCE can write to even unwritable PTEs in COW mappings. */
482 static inline bool can_follow_write_pte(pte_t pte, struct page *page,
483                                         struct vm_area_struct *vma,
484                                         unsigned int flags)
485 {
486     /* If the pte is writable, we can write to the page. */
487     if (pte_write(pte))
488         return true;
489
490     /* Maybe FOLL_FORCE is set to override it? */
491     if (!(flags & FOLL_FORCE))
492         return false;
493
494     /* But FOLL_FORCE has no effect on shared mappings */
495     if (vma->vm_flags & (VM_MAYSHARE | VM_SHARED))
496         return false;
497
498     /* ... or read-only private ones */
499     if (!(vma->vm_flags & VM_MAYWRITE))
500         return false;
501
502     /* ... or already writable ones that just need to take a write fault */
503     if (vma->vm_flags & VM_WRITE)
504         return false;
505
506     /*
507      * See can_change_pte_writable(): we broke COW and could map the page
508      * writable if we have an exclusive anonymous page ...
509      */
510     if (!page || !PageAnon(page) || !PageAnonExclusive(page))
511         return false;
512
513     /* ... and a write-fault isn't required for other reasons. */
514     if (vma_soft_dirty_enabled(vma) && !pte_soft_dirty(pte))
515         return false;
516     return !userfaultfd_pte_wp(vma, pte);
517 }

```

Listing 8-42: mm/gup.c: can_follow_write_pte()

This starts by checking whether the page tables already map this read/write – if so then no write fault will occur and thus we do not need to indicate that this page ought to be faulted in.

All other cases apply *only* if `FOLL_FORCE` is specified – the page must be private, potentially writable (as determined by the `VM_MAYWRITE` flag), not already writable (which a write fault would solve, i.e. not specify `VM_WRITE`), normal (not a PFN map or similar), not exclusively anonymous and not in need of a write fault to satisfy soft-dirty state.

This is a corner case designed to handle the situation where an anonymous folio was previously shared in multiple places due to forking, but is now an exclusive mapping but still a Copy-on-Write mapping (where a Copy-on-Write operation would simply map this exclusively). This is used for example when performing `MAP_POPULATE` over a memory range mapped `PROT_READ` only with a read/write file descriptor where the pages need to be touched.

Finally we consider the mapping is a special mapping, i.e. `vm_normal_page()` returns `NULL` (see section 6.12 for details). In the case that we are core dumping, i.e. `FOLL_DUMP` is specified, then we error out as these should not appear, otherwise we either simply return to page in the case of the zero page or otherwise defer to `follow_pfn_pte()`:-

```

461 static int follow_pfn_pte(struct vm_area_struct *vma, unsigned long address,
462                          pte_t *pte, unsigned int flags)
463 {
464     if (flags & FOLL_TOUCH) {
465         pte_t entry = *pte;
466
467         if (flags & FOLL_WRITE)
468             entry = pte_mkdirty(entry);
469         entry = pte_mkyoung(entry);
470
471         if (!pte_same(*pte, entry)) {
472             set_pte_at(vma->vm_mm, address, pte, entry);
473             update_mmu_cache(vma, address, pte);
474         }
475     }
476
477     /* Proper page table entry exists, but no corresponding struct page */
478     return -EEXIST;
479 }

```

Listing 8-43: `mm/gup.c: follow_pfn_pte()`

This marks the PTE entry dirty if both `FOLL_TOUCH` and `FOLL_WRITE` were set before returning `EEXIST`. This indicates to `__get_user_pages()` that no struct `page` could be associated with the address, if the caller requires an array of pages then an error is returned otherwise this is not an issue.

Returning to the `follow_page_pte()` page checks, we check to see whether the page mapping is currently marked read-only but needs to be *unshared* in order to address a thorny corner case – this is checked in `gup_must_unshare()`:-

```

2968 /*
2969  * Indicates for which pages that are write-protected in the page table,
2970  * whether GUP has to trigger unsharing via FAULT_FLAG_UNSHARE such that the
2971  * GUP pin will remain consistent with the pages mapped into the page tables
2972  * of the MM.
2973  *
2974  * Temporary unmapping of PageAnonExclusive() pages or clearing of
2975  * PageAnonExclusive() has to protect against concurrent GUP:
2976  * * Ordinary GUP: Using the PT lock
2977  * * GUP-fast and fork(): mm->write_protect_seq
2978  * * GUP-fast and KSM or temporary unmapping (swap, migration):
2979  *   clear/invalidate+flush of the page table entry
2980  *
2981  * Must be called with the (sub)page that's actually referenced via the
2982  * page table entry, which might not necessarily be the head page for a
2983  * PTE-mapped THP.
2984  */
2985 static inline bool gup_must_unshare(unsigned int flags, struct page *page)
2986 {
2987     /*
2988      * FOLL_WRITE is implicitly handled correctly as the page table entry
2989      * has to be writable -- and if it references (part of) an anonymous
2990      * folio, that part is required to be marked exclusive.
2991      */
2992     if ((flags & (FOLL_WRITE | FOLL_PIN)) != FOLL_PIN)
2993         return false;
2994     /*
2995      * Note: PageAnon(page) is stable until the page is actually getting
2996      * freed.
2997      */
2998     if (!PageAnon(page))
2999         return false;
3000     /*
3001      * Note that PageKsm() pages cannot be exclusive, and consequently,
3002      * cannot get pinned.
3003      */
3004     return !PageAnonExclusive(page);
3005 }

```

Listing 8-44: include/linux/mm.h: `gup_must_unshare()`

`GUP sharing*` deals with the scenario where a forked Copy-on-Write page is read-only pinned, but mapped by more than one process. In this scenario, the pinned page will no longer be what's mapped by the process in question once a fork takes place, which is likely not what the caller expects.

*. Added in commit a7f226604170: mm/gup: trigger FAULT_FLAG_UNSHARE when R/O-pinning.

This issue is resolved by 'unsharing', i.e. triggering a CoW using the `FAULT_FLAG_UNSHARE` flag (see the page fault section 6 for more details on this) – the criteria are a read-only GUP access with `FOLL_PIN` set on a non-exclusive anonymous page as shown above.

Finally, once this is complete, we assert that no GUP unsharable memory escaped this check. We then move on to the actual pinning of pages and cleanup tasks:-

```

607      /* try_grab_page() does nothing unless FOLL_GET or FOLL_PIN is set. */
608      if (unlikely(!try_grab_page(page, flags))) {
609          page = ERR_PTR(-ENOMEM);
610          goto out;
611      }
612      /*
613       * We need to make the page accessible if and only if we are going
614       * to access its content (the FOLL_PIN case). Please see
615       * Documentation/core-api/pin_user_pages.rst for details.
616       */
617      if (flags & FOLL_PIN) {
618          ret = arch_make_page_accessible(page);
619          if (ret) {
620              unpin_user_page(page);
621              page = ERR_PTR(ret);
622              goto out;
623          }
624      }
625      if (flags & FOLL_TOUCH) {
626          if ((flags & FOLL_WRITE) &&
627              !pte_dirty(pte) && !PageDirty(page))
628              set_page_dirty(page);
629      }
630      /*
631       * pte_mkyoung() would be more correct here, but atomic care
632       * is needed to avoid losing the dirty bit: it is easier to
633       use
634       * mark_page_accessed().
635       */
636      mark_page_accessed(page);
637  }
638  out:
639      pte_unmap_unlock(pte, ptl);
640      return page;
641  no_page:
642      pte_unmap_unlock(pte, ptl);
643      if (!pte_none(pte))
644          return NULL;
645      return no_page_table(vma, flags);
646  }

```

Listing 8-45: mm/gup.c: follow_page_pte() pinning and cleanup

The actual pinning is performed by `try_grab_page()` which we will examine shortly. If the `FOLL_PIN` flag was set, then we make it accessible via `arch_make_page_accessible()` which is only relevant for the s390 architecture and, if `FOLL_TOUCH` flag is specified we mark the page accessed via `mark_page_accessed()` and, if `FOLL_DIRTY` is also set mark it dirty via `set_page_dirty()`.

Finally, the PTE lock is released via `pte_unmap_unlock()` and the pointer to the struct page object is returned.

Examining `try_grab_page()`:-

```

184 /**
185  * try_grab_page() - elevate a page's refcount by a flag-dependent amount
186  * @page:    pointer to page to be grabbed
187  * @flags:   gup flags: these are the FOLL_* flag values.
188  *
189  * This might not do anything at all, depending on the flags argument.
190  *
191  * "grab" names in this file mean, "look at flags to decide whether to use
192  * FOLL_PIN or FOLL_GET behavior, when incrementing the page's refcount.
193  *
194  * Either FOLL_PIN or FOLL_GET (or neither) may be set, but not both at the
195  *   same
196  * time. Cases: please see the try_grab_folio() documentation, with
197  * "refs=1".
198  *
199  * Return: true for success, or if no action was required (if neither FOLL_PIN
200  *   nor FOLL_GET was set, nothing is done). False for failure: FOLL_GET or
201  *   FOLL_PIN was set, but the page could not be grabbed.
202  */
203 bool __must_check try_grab_page(struct page *page, unsigned int flags)
204 {
205     struct folio *folio = page_folio(page);
206
207     WARN_ON_ONCE((flags & (FOLL_GET | FOLL_PIN)) == (FOLL_GET | FOLL_PIN))
208     ;
209     if (WARN_ON_ONCE(folio_ref_count(folio) <= 0))
210         return false;
211
212     if (flags & FOLL_GET)
213         folio_ref_inc(folio);
214     else if (flags & FOLL_PIN) {
215         /*
216          * Similar to try_grab_folio(): be sure to *also*
217          * increment the normal page refcount field at least once,
218          * so that the page really is pinned.

```

```

217         */
218         if (folio_test_large(folio)) {
219             folio_ref_add(folio, 1);
220             atomic_add(1, folio_pincount_ptr(folio));
221         } else {
222             folio_ref_add(folio, GUP_PIN_COUNTING_BIAS);
223         }
224
225         node_stat_mod_folio(folio, NR_FOLL_PIN_ACQUIRED, 1);
226     }
227
228     return true;
229 }

```

Listing 8-46: mm/gup.c: try_grab_page()

This starts by obtaining the struct `folio` associated with the page, that `FOLL_GET` and `FOLL_PIN` are not wrongfully set together and that the folio's reference count is positive.

The `FOLL_GET` case is simple – we simply increment the folio's reference. The `FOLL_PIN` case is more complicated – the first tail page of a huge folio has a `compound_pincount` field which can store the pins.

For order-0 folios, we must resort to something of a hack. We instead increment the reference count by `GUP_PIN_COUNTING_BIAS`, which is set to 2^{10} or 1,024. This is chosen as a kind of mask far higher than you'd expect a reference count to reach, and allows us to identify the pinned pages in `folio_maybe_dma_pinned()` described previously in listing 8-14.

8.1.7 Faulting in Pages

The actual faulting in of a page is performed via `faultin_page()` which is referenced through GUP whenever a faulting in operation has to be performed:-

```

955 /*
956  * mmap_lock must be held on entry. If @locked != NULL and *@flags
957  * does not include FOLL_NOWAIT, the mmap_lock may be released. If it
958  * is, *@locked will be set to 0 and -EBUSY returned.
959  */
960 static int faultin_page(struct vm_area_struct *vma,
961                        unsigned long address, unsigned int *flags, bool unshare,
962                        int *locked)
963 {
964     unsigned int fault_flags = 0;
965     vm_fault_t ret;
966
967     if (*flags & FOLL_NOFAULT)
968         return -EFAULT;
969     if (*flags & FOLL_WRITE)

```

```

970         fault_flags |= FAULT_FLAG_WRITE;
971     if (*flags & FOLL_REMOTE)
972         fault_flags |= FAULT_FLAG_REMOTE;
973     if (locked)
974         fault_flags |= FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;
975     if (*flags & FOLL_NOWAIT)
976         fault_flags |= FAULT_FLAG_ALLOW_RETRY |
FAULT_FLAG_RETRY_NOWAIT;
977     if (*flags & FOLL_TRIED) {
978         /*
979          * Note: FAULT_FLAG_ALLOW_RETRY and FAULT_FLAG_TRIED
980          * can co-exist
981          */
982         fault_flags |= FAULT_FLAG_TRIED;
983     }
984     if (unshare) {
985         fault_flags |= FAULT_FLAG_UNSHARE;
986         /* FAULT_FLAG_WRITE and FAULT_FLAG_UNSHARE are incompatible */
987         VM_BUG_ON(fault_flags & FAULT_FLAG_WRITE);
988     }
989
990     ret = handle_mm_fault(vma, address, fault_flags, NULL);
991
992     if (ret & VM_FAULT_COMPLETED) {
993         /*
994          * With FAULT_FLAG_RETRY_NOWAIT we'll never release the
995          * mmap lock in the page fault handler. Sanity check this.
996          */
997         WARN_ON_ONCE(fault_flags & FAULT_FLAG_RETRY_NOWAIT);
998         if (locked)
999             *locked = 0;
1000         /*
1001          * We should do the same as VM_FAULT_RETRY, but let's not
1002          * return -EBUSY since that's not reflecting the reality of
1003          * what has happened - we've just fully completed a page
1004          * fault, with the mmap lock released. Use -EAGAIN to show
1005          * that we want to take the mmap lock_again_.
1006          */
1007         return -EAGAIN;
1008     }
1009
1010     if (ret & VM_FAULT_ERROR) {
1011         int err = vm_fault_to_errno(ret, *flags);
1012
1013         if (err)
1014             return err;
1015         BUG();

```

```

1016     }
1017
1018     if (ret & VM_FAULT_RETRY) {
1019         if (locked && !(fault_flags & FAULT_FLAG_RETRY_NOWAIT))
1020             *locked = 0;
1021         return -EBUSY;
1022     }
1023
1024     return 0;
1025 }

```

Listing 8-47: mm/gup.c: `faultin_page()`

This starts by checking whether the `FOLL_NOFAULT` flag is set – if so this is a no-op, as clearly no fault should take place, and this is indicated by the function returning the `EFAULT` error code.

The `FAULT_FLAG_WRITE` and `FAULT_FLAG_REMOTE` flags are passed through from the respective `FOLL_WRITE` and `FOLL_REMOTE` flags.

If the `locked` parameter is non-NULL then this indicates that the caller will accept the `struct mm_struct->mmap_lock` being dropped for I/O and performing a fault retry, setting the `FAULT_FLAG_ALLOW_RETRY` and `FAULT_FLAG_KILLABLE` flags as a consequence.

if the `FOLL_NOWAIT` flag is specified, then we are permitted to retry for I/O but will *not* drop the lock, via the `FAULT_FLAG_ALLOW_RETRY` and `FAULT_FLAG_RETRY_NOWAIT` flags.

Finally, if the `unshare` parameter was set true to handle the case where a read fault needs to CoW a page, the `FAULT_FLAG_UNSHARE` flag is set. We also check to ensure we are faulting read-only.

With all of this set up, the actual page fault is handled via `handle_mm_fault()`. For a detailed examination of this, see the page fault section 6.

Once the actual faulting in is complete the return value is handled:-

- `VM_FAULT_COMPLETED` – Indicates that the lock has been dropped for I/O and the operation is complete. We update `locked` accordingly and return `EAGAIN` to indicate that this has occurred.
- `VM_FAULT_ERROR` – An error has arisen, we convert it into an error code via `vm_fault_to_errno()` and exit (raising a kernel oops if the fault could not be converted to an error code.)
- `VM_FAULT_RETRY` – The fault needs to be retrieved, indicate via `EBUSY` and update `locked` accordingly.

8.1.8 Fast GUP Functions

The fast GUP path attempts to perform the same operation as the non-fast kind but trying to avoid having to take a lock on the `struct mm_struct->mmap_lock` semaphore.

These differ from their non-fast counterparts in a few respects:-

- There is no means of returning an array of struct `vm_area_struct` (VMA) objects as these are only ever valid with the `mmap_lock` semaphore held and of course the fast path functions are trying to avoid this.
- The `get_*`() variants explicitly set the `FOLL_GET` flag, so the `pages` argument must also be non-NULL.
- Unless the `*_user_pages_fast_only()` variants are used, the functions will fall back to using the non-fast path if the fast one is unable to perform the operation locklessly.
- Only a subset of flags are permitted, see below:-

Table 8-2: GUP flag validity

GUP flag	Non-fast	Fast
FOLL_WRITE	•	•
FOLL_TOUCH	•	
FOLL_GET	•	•
FOLL_DUMP	•	
FOLL_FORCE	•	•
FOLL_NOWAIT	•	
FOLL_NOFAULT	•	•
FOLL_HWPOISON	•	
FOLL_NUMA	•	
FOLL_MIGRATION	•	
FOLL_TRIED	•	
FOLL_REMOTE	•	
FOLL_ANON	•	
FOLL_LONGTERM	•	•
FOLL_SPLIT_PMD	•	
FOLL_PIN	•	•
FOLL_FAST_ONLY		•

Examining `get_user_pages_fast()`:-

```

3061 /**
3062  * get_user_pages_fast() - pin user pages in memory
3063  * @start:      starting user address
3064  * @nr_pages:   number of pages from start to pin
3065  * @gup_flags:  flags modifying pin behaviour
3066  * @pages:      array that receives pointers to the pages pinned.
3067  *              Should be at least nr_pages long.
3068  *
3069  * Attempt to pin user pages in memory without taking mm->mmap_lock.
3070  * If not successful, it will fall back to taking the lock and
3071  * calling get_user_pages().
3072  *
3073  * Returns number of pages pinned. This may be fewer than the number requested
3074  * .
3075  * If nr_pages is 0 or negative, returns 0. If no pages were pinned, returns

```

```

3075  * -errno.
3076  */
3077  int get_user_pages_fast(unsigned long start, int nr_pages,
3078                          unsigned int gup_flags, struct page **pages)
3079  {
3080      if (!is_valid_gup_flags(gup_flags))
3081          return -EINVAL;
3082
3083      /*
3084       * The caller may or may not have explicitly set FOLL_GET; either way
3085       * is
3086       * OK. However, internally (within mm/gup.c), gup fast variants must
3087       * set
3088       * FOLL_GET, because gup fast is always a "pin with a +1 page refcount
3089       * request.
3090       */
3091      gup_flags |= FOLL_GET;
3092      return internal_get_user_pages_fast(start, nr_pages, gup_flags, pages)
3093      ;
3094  }

```

Listing 8-48: mm/gup.c: get_user_pages_fast()

This checks `is_valid_gup_flags()` as do the non-fast `get_user_pages*()` functions (see listing 8-16) before unconditionally setting `FOLL_GET` and invoking the shared `internal_get_user_pages_fast()` function.

Note that this form of the function will fall back to the non-fast path if it cannot pin all of the requested pages. If this behaviour is not desired, then `get_user_pages_fast_only()` will pin pages using the fast method *only*:-

```

3012  /**
3013   * get_user_pages_fast_only() - pin user pages in memory
3014   * @start:      starting user address
3015   * @nr_pages:   number of pages from start to pin
3016   * @gup_flags:  flags modifying pin behaviour
3017   * @pages:      array that receives pointers to the pages pinned.
3018   *              Should be at least nr_pages long.
3019   *
3020   * Like get_user_pages_fast() except it's IRQ-safe in that it won't fall back
3021   * to
3022   * the regular GUP.
3023   * Note a difference with get_user_pages_fast: this always returns the
3024   * number of pages pinned, 0 if no pages were pinned.
3025   *
3026   * If the architecture does not support this function, simply return with no
3027   * pages pinned.
3028   * Careful, careful! COW breaking can go either way, so a non-write

```



```

3029  * access can get ambiguous page results. If you call this function without
3030  * 'write' set, you'd better be sure that you're ok with that ambiguity.
3031  */
3032  int get_user_pages_fast_only(unsigned long start, int nr_pages,
3033                               unsigned int gup_flags, struct page **pages)
3034  {
3035      int nr_pinned;
3036      /*
3037       * Internally (within mm/gup.c), gup fast variants must set FOLL_GET,
3038       * because gup fast is always a "pin with a +1 page refcount" request.
3039       *
3040       * FOLL_FAST_ONLY is required in order to match the API description of
3041       * this routine: no fall back to regular ("slow") GUP.
3042       */
3043      gup_flags |= FOLL_GET | FOLL_FAST_ONLY;
3044
3045      nr_pinned = internal_get_user_pages_fast(start, nr_pages, gup_flags,
3046                                              pages);
3047
3048      /*
3049       * As specified in the API description above, this routine is not
3050       * allowed to return negative values. However, the common core
3051       * routine internal_get_user_pages_fast() can return -errno.
3052       * Therefore, correct for that here:
3053       */
3054      if (nr_pinned < 0)
3055          nr_pinned = 0;
3056
3057      return nr_pinned;
3058  }

```

Listing 8-49: mm/gup.c: get_user_pages_fast_only()

This sets FOLL_FAST_ONLY and returns the number pinned or zero if an error occurs, no fall back to the non-fast path occurs. This is useful in situations where the mmap_lock cannot be acquired.

Similarly, the GUP pin variants are similar to these, only setting FOLL_PIN rather than FOLL_GET:-

```

3094  /**
3095   * pin_user_pages_fast() - pin user pages in memory without taking locks
3096   *
3097   * @start:      starting user address
3098   * @nr_pages:   number of pages from start to pin
3099   * @gup_flags:  flags modifying pin behaviour
3100   * @pages:     array that receives pointers to the pages pinned.
3101   *             Should be at least nr_pages long.
3102   *
3103   * Nearly the same as get_user_pages_fast(), except that FOLL_PIN is set. See

```

```

3104  * get_user_pages_fast() for documentation on the function arguments, because
3105  * the arguments here are identical.
3106  *
3107  * FOLL_PIN means that the pages must be released via unpin_user_page().
      Please
3108  * see Documentation/core-api/pin_user_pages.rst for further details.
3109  */
3110  int pin_user_pages_fast(unsigned long start, int nr_pages,
3111                          unsigned int gup_flags, struct page **pages)
3112  {
3113      /* FOLL_GET and FOLL_PIN are mutually exclusive. */
3114      if (WARN_ON_ONCE(gup_flags & FOLL_GET))
3115          return -EINVAL;
3116
3117      if (WARN_ON_ONCE(!pages))
3118          return -EINVAL;
3119
3120      gup_flags |= FOLL_PIN;
3121      return internal_get_user_pages_fast(start, nr_pages, gup_flags, pages)
      ;
3122  }

```

Listing 8-50: mm/gup.c: `pin_user_pages_fast()`

And the fast path only equivalent:-

```

3125  /*
3126  * This is the FOLL_PIN equivalent of get_user_pages_fast_only(). Behavior
3127  * is the same, except that this one sets FOLL_PIN instead of FOLL_GET.
3128  *
3129  * The API rules are the same, too: no negative values may be returned.
3130  */
3131  int pin_user_pages_fast_only(unsigned long start, int nr_pages,
3132                              unsigned int gup_flags, struct page **pages)
3133  {
3134      int nr_pinned;
3135
3136      /*
3137       * FOLL_GET and FOLL_PIN are mutually exclusive. Note that the API
3138       * rules require returning 0, rather than -errno:
3139       */
3140      if (WARN_ON_ONCE(gup_flags & FOLL_GET))
3141          return 0;
3142
3143      if (WARN_ON_ONCE(!pages))
3144          return 0;
3145
3146      /*
3147       * FOLL_FAST_ONLY is required in order to match the API description of
3148       * this routine: no fall back to regular ("slow") GUP.

```

```

3148     */
3149     gup_flags |= (FOLL_PIN | FOLL_FAST_ONLY);
3150     nr_pinned = internal_get_user_pages_fast(start, nr_pages, gup_flags,
3151                                             pages);
3152     /*
3153      * This routine is not allowed to return negative values. However,
3154      * internal_get_user_pages_fast() can return -errno. Therefore,
3155      * correct for that here:
3156      */
3157     if (nr_pinned < 0)
3158         nr_pinned = 0;
3159
3160     return nr_pinned;
3161 }

```

Listing 8-51: mm/gup.c: pin_user_pages_fast_only()

Each of these ultimately invoke `internal_get_user_pages_fast()`. This is somewhat equivalent to the non-fast `__get_user_pages()` only with some additional handling for the fast path:-

```

2964 static int internal_get_user_pages_fast(unsigned long start,
2965                                         unsigned long nr_pages,
2966                                         unsigned int gup_flags,
2967                                         struct page **pages)
2968 {
2969     unsigned long len, end;
2970     unsigned long nr_pinned;
2971     int ret;
2972
2973     if (WARN_ON_ONCE(gup_flags & ~(FOLL_WRITE | FOLL_LONGTERM |
2974                                   FOLL_FORCE | FOLL_PIN | FOLL_GET |
2975                                   FOLL_FAST_ONLY | FOLL_NOFAULT)))
2976         return -EINVAL;
2977
2978     if (gup_flags & FOLL_PIN)
2979         mm_set_has_pinned_flag(&current->mm->flags);
2980
2981     if (!(gup_flags & FOLL_FAST_ONLY))
2982         might_lock_read(&current->mm->mmap_lock);
2983
2984     start = untagged_addr(start) & PAGE_MASK;
2985     len = nr_pages << PAGE_SHIFT;
2986     if (check_add_overflow(start, len, &end))
2987         return 0;
2988     if (unlikely(!access_ok((void __user *)start, len)))
2989         return -EFAULT;
2990
2991     nr_pinned = lockless_pages_from_mm(start, end, gup_flags, pages);

```

```

2992     if (nr_pinned == nr_pages || gup_flags & FOLL_FAST_ONLY)
2993         return nr_pinned;
2994
2995     /* Slow path: try to get the remaining pages with get_user_pages */
2996     start += nr_pinned << PAGE_SHIFT;
2997     pages += nr_pinned;
2998     ret = __gup_longterm_unlocked(start, nr_pages - nr_pinned, gup_flags,
2999                                   pages);
3000
3001     if (ret < 0) {
3002         /*
3003          * The caller has to unpin the pages we already pinned so
3004          * returning -errno is not an option
3005          */
3006         if (nr_pinned)
3007             return nr_pinned;
3008         return ret;
3009     }
3010     return ret + nr_pinned;
3011 }

```

Listing 8-52: mm/gup.c: internal_get_user_pages_fast()

This asserts that the flags are permitted as described in figure 8-2, before setting the struct `mm_struct` `MMF_HAS_PINNED` flag via `mm_set_has_pinned_flag()` if pinning, as described previously.

Then we indicate that a lock can be acquired here if the `FOLL_FAST_ONLY` flag is not set (since we can degrade to using the non-fast path), check for overflow via `check_add_overflow()`, check accessibility to the memory range via `access_ok()`, before deferring the heavy lifting to `lockless_pages_from_mm()`.

If fewer than expected pages are returned and `FOLL_FAST_ONLY` is specified then we simply return this count, otherwise we fall back to the non-fast path for the remaining pages after offsetting to account for this invoking `__gup_longterm_locked()` to do so.

If we do fall back to this code path we, similar to `__get_user_pages()`, only return an error if no pages have been pinned, if some have then we simply indicate how many were.

Examining `lockless_pages_from_mm()`:-

```

2915 static unsigned long lockless_pages_from_mm(unsigned long start,
2916                                             unsigned long end,
2917                                             unsigned int gup_flags,
2918                                             struct page **pages)
2919 {
2920     unsigned long flags;
2921     int nr_pinned = 0;
2922     unsigned seq;
2923
2924     if (!IS_ENABLED(CONFIG_HAVE_FAST_GUP) ||
2925         !gup_fast_permitted(start, end))

```

```

2926         return 0;
2927
2928     if (gup_flags & FOLL_PIN) {
2929         seq = raw_read_seqcount(&current->mm->write_protect_seq);
2930         if (seq & 1)
2931             return 0;
2932     }
2933
2934     /*
2935      * Disable interrupts. The nested form is used, in order to allow full
2936      * general purpose use of this routine.
2937      *
2938      * With interrupts disabled, we block page table pages from being
2939      * freed
2940      * from under us. See struct mmu_table_batch comments in
2941      * include/asm-generic/tlb.h for more details.
2942      *
2943      * We do not adopt an rcu_read_lock() here as we also want to block
2944      * IPIs
2945      * that come from THPs splitting.
2946      */
2947     local_irq_save(flags);
2948     gup_pgdrange(start, end, gup_flags, pages, &nr_pinned);
2949     local_irq_restore(flags);
2950
2951     /*
2952      * When pinning pages for DMA there could be a concurrent write
2953      * protect
2954      * from fork() via copy_page_range(), in this case always fail fast
2955      * GUP.
2956      */
2957     if (gup_flags & FOLL_PIN) {
2958         if (read_seqcount_retry(&current->mm->write_protect_seq, seq))
2959             {
2960                 unpin_user_pages_lockless(pages, nr_pinned);
2961                 return 0;
2962             } else {
2963                 sanity_check_pinned_pages(pages, nr_pinned);
2964             }
2965     }
2966     return nr_pinned;
2967 }

```

Listing 8-53: mm/gup.c: lockless_pages_from_mm()

The whole idea here is to try to increment a reference count on pages in the range optimistically on the assumption that the pages have not been

freed beneath us, this is ultimately done after traversing page tables as with the non-fast path starting from `gup_pgdp_range()`.

Given we do not possess the luxury of the `struct mm_struct->mmap_lock` which we are trying to avoid taking to reduce contention, we have to be careful of races.

We check the state of the `mm_struct->write_protect_seq` lock to avoid a race between forking and pinning (due to the fork logic branching on `page_maybe_dma_pinned()` on write-protecting newly CoW'd mappings*).

See the *sequence counter locking* documentation for details, but essentially it is an atomic sequence number which is set odd when the critical section is being updated and even when not. Therefore, if the value is odd, we abort the operation right away, otherwise we store the sequence number, retrieving the value via `raw_read_seqcount()`.

Once `gup_pgdp_range()` is complete, we check again via `read_seqcount_retry()` which checks to see whether the sequence number has changed since the operation started, which would indicate a race may have occurred, erroring out if so. Note the invocation of `sanity_check_pinned_pages()` when the race does not occur is only relevant if `CONFIG_DEBUG_VM` is specified.

This race condition aside, we invoke `gup_pgdp_range()` with interrupts disabled in order to prevent page tables disappearing from beneath us. This function updates `nr_pinned` which we return.

Prior to this, we invoke the architecture-specific `gup_fast_permitted()` to determine whether the operation is permitted at all – for x86-64 this simply checks to ensure the end of the range is a valid virtual address.

Similar to the non-fast path, we must traverse through the page table levels, starting from PGD. Again we elide out-of-scope huge page handling here:-

```

2849 static void gup_pgdp_range(unsigned long addr, unsigned long end,
2850                          unsigned int flags, struct page **pages, int *nr)
2851 {
2852     unsigned long next;
2853     pgdp_t *pgdp;
2854
2855     pgdp = pgdp_offset(current->mm, addr);
2856     do {
2857         pgd_t pgd = READ_ONCE(*pgdp);
2858
2859         next = pgd_addr_end(addr, end);
2860         if (pgd_none(pgd))
2861             return;
2862         :
2870     } else if (!gup_p4d_range(pgdp, pgd, addr, next, flags, pages,
2871                             nr))
2872         return;

```

*. See commit 57efa1fe5957: `mm/gup: prevent gup_fast from racing with COW during fork.`

```

2872         } while (pgdp++, addr = next, addr != end);
2873     }

```

Listing 8-54: mm/gup.c: gup_pgd_range()

This iterates through the input range, crossing PGD entry boundaries as necessary, deferring to the next level via gup_p4d_range() (whose return value indicates if the operation has failed at any stage):-

```

2824 static int gup_p4d_range(pgd_t *pgdp, pgd_t pgd, unsigned long addr, unsigned
        long end,
2825                        unsigned int flags, struct page **pages, int *nr)
2826 {
2827     unsigned long next;
2828     p4d_t *p4dp;
2829
2830     p4dp = p4d_offset_lockless(pgdp, pgd, addr);
2831     do {
2832         p4d_t p4d = READ_ONCE(*p4dp);
2833
2834         next = p4d_addr_end(addr, end);
2835         if (p4d_none(p4d))
2836             return 0;
2837         :
2842     } else if (!gup_pud_range(p4dp, p4d, addr, next, flags, pages,
        nr))
2843         return 0;
2844     } while (p4dp++, addr = next, addr != end);
2845
2846     return 1;
2847 }

```

Listing 8-55: mm/gup.c: gup_p4d_range()

One difference here is the use of the p4d_offset_lockless() variant of the page table offset function. This is exactly equivalent to the non-lockless one for all architectures except s390. See the virtual memory chapter for more on these functions in general.

Examining the same function for the next level, gup_pud_range():-

```

2796 static int gup_pud_range(p4d_t *p4dp, p4d_t p4d, unsigned long addr, unsigned
        long end,
2797                        unsigned int flags, struct page **pages, int *nr)
2798 {
2799     unsigned long next;
2800     pud_t *pudp;
2801
2802     pudp = pud_offset_lockless(p4dp, p4d, addr);
2803     do {
2804         pud_t pud = READ_ONCE(*pudp);

```

```

2805
2806         next = pud_addr_end(addr, end);
2807         if (unlikely(!pud_present(pud)))
2808             return 0;
2809         :
2817     } else if (!gup_pmd_range(pudp, pud, addr, next, flags, pages,
nr))
2818         return 0;
2819     } while (pudp++, addr = next, addr != end);
2820
2821     return 1;
2822 }

```

Listing 8-56: mm/gup.c: gup_pud_range()

And for PMD:-

```

2753 static int gup_pmd_range(pud_t *pudp, pud_t pud, unsigned long addr, unsigned
long end,
2754                         unsigned int flags, struct page **pages, int *nr)
2755 {
2756     unsigned long next;
2757     pmd_t *pmdp;
2758
2759     pmdp = pmd_offset_lockless(pudp, pud, addr);
2760     do {
2761         pmd_t pmd = READ_ONCE(*pmdp);
2762
2763         next = pmd_addr_end(addr, end);
2764         if (!pmd_present(pmd))
2765             return 0;
2766         :
2789     } else if (!gup_pte_range(pmd, pmdp, addr, next, flags, pages,
nr))
2790         return 0;
2791     } while (pmdp++, addr = next, addr != end);
2792
2793     return 1;
2794 }

```

Listing 8-57: mm/gup.c: gup_pmd_range()

And finally we arrive at gup_pte_range() (eliding out of scope huge page, device map, secret memory logic and s390-specific page accessibility logic):-

```

2348 /*
2349  * Fast-gup relies on pte change detection to avoid concurrent pgtable
2350  * operations.
2351  *
2352  * To pin the page, fast-gup needs to do below in order:

```



```

2353  * (1) pin the page (by prefetching pte), then (2) check pte not changed.
2354  *
2355  * For the rest of pgtable operations where pgtable updates can be racy
2356  * with fast-gup, we need to do (1) clear pte, then (2) check whether page
2357  * is pinned.
2358  *
2359  * Above will work for all pte-level operations, including THP split.
2360  *
2361  * For THP collapse, it's a bit more complicated because fast-gup may be
2362  * walking a pgtable page that is being freed (pte is still valid but pmd
2363  * can be cleared already). To avoid race in such condition, we need to
2364  * also check pmd here to make sure pmd doesn't change (corresponds to
2365  * pmdp_collapse_flush() in the THP collapse code path).
2366  */
2367  static int gup_pte_range(pmd_t pmd, pmd_t *pmdp, unsigned long addr,
2368                          unsigned long end, unsigned int flags,
2369                          struct page **pages, int *nr)
2370  {
2371      :
2372      int nr_start = *nr, ret = 0;
2373      pte_t *ptep, *ptem;
2374
2375      ptem = ptep = pte_offset_map(&pmd, addr);
2376      do {
2377          pte_t pte = ptep_get_lockless(ptep);
2378          struct page *page;
2379          struct folio *folio;
2380
2381          /*
2382           * Similar to the PMD case below, NUMA hinting must take slow
2383           * path using the pte_protnone check.
2384           */
2385          if (pte_protnone(pte))
2386              goto pte_unmap;
2387
2388          if (!pte_access_permitted(pte, flags & FOLL_WRITE))
2389              goto pte_unmap;
2390          :
2391      } else if (pte_special(pte))
2392          goto pte_unmap;
2393
2394      VM_BUG_ON(!pfn_valid(pte_pfn(pte)));
2395      page = pte_page(pte);
2396
2397      folio = try_grab_folio(page, 1, flags);
2398      if (!folio)
2399          goto pte_unmap;

```

```

:
2415         if (unlikely(pmd_val(pmd) != pmd_val(*pmdp)) ||
2416             unlikely(pte_val(pte) != pte_val(*ptep))) {
2417             gup_put_folio(folio, 1, flags);
2418             goto pte_unmap;
2419         }
2420
2421         if (!pte_write(pte) && gup_must_unshare(flags, page)) {
2422             gup_put_folio(folio, 1, flags);
2423             goto pte_unmap;
2424         }
2425         :
2439         folio_set_referenced(folio);
2440         pages[*nr] = page;
2441         (*nr)++;
2442     } while (ptep++, addr += PAGE_SIZE, addr != end);
2443
2444     ret = 1;
2445
2446 pte_unmap:
2447     :
2449     pte_unmap(ptem);
2450     return ret;
2451 }

```

Listing 8-58: mm/gup.c: gup_pte_range()

This follows the typical pattern of iterating through the page table as with the page table levels above, however a key part of the lockless algorithm is applied here – the value of and the pointer to the PMD entry containing the reference to this PTE is passed in and prior to processing it the PTE value is retrieved (the PMD check being more relevant to the out of scope case of transparent huge page coalescing).

Note, again, that mapping and unmapping the PTE is a no-op on x86-64 and other modern 64-bit architectures, so these can be duly ignored for the most part as 32-bit architectures are out of scope for the book.

In all cases where we cannot proceed with the lockless pinning of the underlying page, we simply exit. Since we’re updating `nr_Pages` as we pin pages, the progress of which pages we were able to pin will be propagated back to the caller.

The function will exit in this fashion if the PTE entry indicates NUMA hinting is taking place checked via `pte_protnone()` or if the mapping is special checked via `pte_special()` (see section 6.12 for more details).

Since we cannot fault pages in (this is the fast path after all), we explicitly check that the page table flags match expectations via the architecture-specific `pte_access_permitted()`.

The most pertinent check here is whether the entry is writable should `FOLL_WRITE` be specified, if not we simply exit early and (unless `FOLL_FAST_ONLY` is specified) we fallback to the slow path to fault the page in.

We assert that the PFN referenced by the PTE is valid via `pfn_valid()` as by this point the checks prior to this stage should assure us of this fact.

We then are able to obtain a pointer to the underlying struct `page` via `pte_page()`.

We then perform the actual pinning via `try_grab_folio()`, which we will examine in detail shortly. If this fails we abort.

After we pinned this page, we perform our consistency check – did the PTE (or PMD in huge case) change beneath us while we were pinning? We do this by comparing PMD and PTE values before and after the operation – if they do differ we release the pin via `gup_put_folio()` and abort.

After this, we check to see whether the page actually required unsharing in which case we again undo our optimistic pinning via `gup_put_folio()`.

Finally, we set the folio referenced flag `PG_referenced` as the GUP operation clearly implies that a reference has been made and we add it to the pages array.

Coming back to `try_grab_folio()`:-

```

98 /**
99  * try_grab_folio() - Attempt to get or pin a folio.
100  * @page: pointer to page to be grabbed
101  * @refs: the value to (effectively) add to the folio's refcount
102  * @flags: gup flags: these are the FOLL_* flag values.
103  *
104  * "grab" names in this file mean, "look at flags to decide whether to use
105  * FOLL_PIN or FOLL_GET behavior, when incrementing the folio's refcount.
106  *
107  * Either FOLL_PIN or FOLL_GET (or neither) must be set, but not both at the
108  * same time. (That's true throughout the get_user_pages*() and
109  * pin_user_pages*() APIs.) Cases:
110  *
111  *   FOLL_GET: folio's refcount will be incremented by @refs.
112  *
113  *   FOLL_PIN on large folios: folio's refcount will be incremented by
114  *   @refs, and its compound_pincount will be incremented by @refs.
115  *
116  *   FOLL_PIN on single-page folios: folio's refcount will be incremented by
117  *   @refs * GUP_PIN_COUNTING_BIAS.
118  *
119  * Return: The folio containing @page (with refcount appropriately
120  * incremented) for success, or NULL upon failure. If neither FOLL_GET
121  * nor FOLL_PIN was set, that's considered failure, and furthermore,
122  * a likely bug in the caller, so a warning is also emitted.
123  */
124 struct folio *try_grab_folio(struct page *page, int refs, unsigned int flags)
125 {

```

```

126         if (flags & FOLL_GET)
127             return try_get_folio(page, refs);
128         else if (flags & FOLL_PIN) {
129             struct folio *folio;
130
131             /*
132              * Can't do FOLL_LONGTERM + FOLL_PIN gup fast path if not in a
133              * right zone, so fail and let the caller fall back to the
134              * path.
135              */
136             if (unlikely((flags & FOLL_LONGTERM) &&
137                          !is_longterm_pinnable_page(page)))
138                 return NULL;
139
140             /*
141              * CAUTION: Don't use compound_head() on the page before this
142              * point, the result won't be stable.
143              */
144             folio = try_get_folio(page, refs);
145             if (!folio)
146                 return NULL;
147
148             /*
149              * When pinning a large folio, use an exact count to track it.
150              *
151              * However, be sure to *also* increment the normal folio
152              * refcount field at least once, so that the folio really
153              * is pinned. That's why the refcount from the earlier
154              * try_get_folio() is left intact.
155              */
156             if (folio_test_large(folio))
157                 atomic_add(refs, folio_pincount_ptr(folio));
158             else
159                 folio_ref_add(folio,
160                             refs * (GUP_PIN_COUNTING_BIAS - 1));
161             node_stat_mod_folio(folio, NR_FOLL_PIN_ACQUIRED, refs);
162
163             return folio;
164         }
165
166         WARN_ON_ONCE(1);
167         return NULL;
168     }

```

Listing 8-59: mm/gup.c: try_grab_folio()

The `FOLL_GET` case is easy – the actual increment of the folio reference count is achieved via `try_get_folio()` which is simply deferred to (we will examine this shortly).

In the case of a `FOLL_PIN`, we start by checking whether `FOLL_LONGTERM` has been specified. If so, the fast path, instead of migrating long term pinned pages to an unmovable page block, simply checks whether they already in this state via `is_longterm_pinnable_page()` (see listing 8-24), aborting if it is not.

As with `FOLL_GET` we try to increment the reference count via `try_get_folio()`, aborting if we cannot, in order to keep the page around for the next step.

Finally, similar, to `try_grab_page()` we apply the `GUP_PIN_COUNTING_BIAS` (offset by one to account for the reference count increment we did to keep the page around) or in the case of a huge page increment its `compound_pincount`.

Examining `try_get_folio()` (eliding out of scope device mapping logic):-

```

65  /*
66   * Return the folio with ref appropriately incremented,
67   * or NULL if that failed.
68   */
69  static inline struct folio *try_get_folio(struct page *page, int refs)
70  {
71      struct folio *folio;
72
73  retry:
74      folio = page_folio(page);
75      if (WARN_ON_ONCE(folio_ref_count(folio) < 0))
76          return NULL;
77      if (unlikely(!folio_ref_try_add_rcu(folio, refs)))
78          return NULL;
79
80      /*
81       * At this point we have a stable reference to the folio; but it
82       * could be that between calling page_folio() and the refcount
83       * increment, the folio was split, in which case we'd end up
84       * holding a reference on a folio that has nothing to do with the page
85       * we were given anymore.
86       * So now that the folio is stable, recheck that the page still
87       * belongs to this folio.
88       */
89      if (unlikely(page_folio(page) != folio)) {
90          :
91
92          goto retry;
93      }
94
95      return folio;
96  }
```

Listing 8-60: `mm/gup.c: try_get_folio()`

The key operation here is `folio_ref_try_add_rcu()` which ultimately invokes `folio_ref_add_unless()` which will increment the folio's reference number unless it is currently zero.

It's out of scope here, but we will briefly touch upon transparent huge pages functionality – this occasionally splits a higher order folio into order-0 folios. In this case, the folio associated with the input page may no longer belong to the folio whose reference count we incremented, in which case we must retry.

8.1.9 GUP Helper Functions

There are a number of parts of the kernel which invoke GUP logic through means other than the fundamental GUP functions discussed here. These hook into various parts of GUP, most notably the faulting logic.

The header file `include/linux/mm.h` lists the exported functions like this, however we shall examine one case in particular – `__mm_populate()`. This is important because it is the function used by the mapping logic when faulting in pages when the `MAP_POPULATE` function is invoked:-

```

1645  /*
1646   * __mm_populate - populate and/or mlock pages within a range of address space
1647   *
1648   * This is used to implement mlock() and the MAP_POPULATE / MAP_LOCKED mmap
1649   * flags. VMAs must be already marked with the desired vm_flags, and
1650   * mmap_lock must not be held.
1651   */
1652  int __mm_populate(unsigned long start, unsigned long len, int ignore_errors)
1653  {
1654      struct mm_struct *mm = current->mm;
1655      unsigned long end, nstart, nend;
1656      struct vm_area_struct *vma = NULL;
1657      int locked = 0;
1658      long ret = 0;
1659
1660      end = start + len;
1661
1662      for (nstart = start; nstart < end; nstart = nend) {
1663          /*
1664           * We want to fault in pages for [nstart; end) address range.
1665           * Find first corresponding VMA.
1666           */
1667          if (!locked) {
1668              locked = 1;
1669              mmap_read_lock(mm);
1670              vma = find_vma(mm, nstart);
1671          } else if (nstart >= vma->vm_end)
1672              vma = vma->vm_next;

```

```

1673         if (!vma || vma->vm_start >= end)
1674             break;
1675         /*
1676          * Set [nstart; nend) to intersection of desired address
1677          * range with the first VMA. Also, skip undesirable VMA types.
1678          */
1679         nend = min(end, vma->vm_end);
1680         if (vma->vm_flags & (VM_IO | VM_PFNMAP))
1681             continue;
1682         if (nstart < vma->vm_start)
1683             nstart = vma->vm_start;
1684         /*
1685          * Now fault in a range of pages. populate_vma_page_range()
1686          * double checks the vma flags, so that it won't mlock pages
1687          * if the vma was already munlocked.
1688          */
1689         ret = populate_vma_page_range(vma, nstart, nend, &locked);
1690         if (ret < 0) {
1691             if (ignore_errors) {
1692                 ret = 0;
1693                 continue;        /* continue at next VMA */
1694             }
1695             break;
1696         }
1697         nend = nstart + ret * PAGE_SIZE;
1698         ret = 0;
1699     }
1700     if (locked)
1701         mmap_read_unlock(mm);
1702     return ret;    /* 0 or negative error code */
1703 }

```

Listing 8-61: mm/gup.c: `__mm_populate()`

This acquires a read lock on the struct `mm_struct->mmap_lock` semaphore, iterating through each VMA and performing the actual faulting in of pages in each VMA via `populate_vma_page_range()`, taking care to write fault in Copy on Write (CoW) mappings, deferring the actual work of doing so to `__get_user_pages()`:-

```

1515 /**
1516  * populate_vma_page_range() - populate a range of pages in the vma.
1517  * @vma:    target vma
1518  * @start:  start address
1519  * @end:    end address
1520  * @locked: whether the mmap_lock is still held
1521  *
1522  * This takes care of mlocking the pages too if VM_LOCKED is set.
1523  *

```

```

1524  * Return either number of pages pinned in the vma, or a negative error
1525  * code on error.
1526  *
1527  * vma->vm_mm->mmap_lock must be held.
1528  *
1529  * If @locked is NULL, it may be held for read or write and will
1530  * be unperturbed.
1531  *
1532  * If @locked is non-NULL, it must held for read only and may be
1533  * released. If it's released, *@locked will be set to 0.
1534  */
1535  long populate_vma_page_range(struct vm_area_struct *vma,
1536                             unsigned long start, unsigned long end, int *locked)
1537  {
1538      struct mm_struct *mm = vma->vm_mm;
1539      unsigned long nr_pages = (end - start) / PAGE_SIZE;
1540      int gup_flags;
1541      long ret;
1542
1543      VM_BUG_ON(!PAGE_ALIGNED(start));
1544      VM_BUG_ON(!PAGE_ALIGNED(end));
1545      VM_BUG_ON_VMA(start < vma->vm_start, vma);
1546      VM_BUG_ON_VMA(end > vma->vm_end, vma);
1547      mmap_assert_locked(mm);
1548
1549      /*
1550       * Rightly or wrongly, the VM_LOCKONFAULT case has never used
1551       * faultin_page() to break COW, so it has no work to do here.
1552       */
1553      if (vma->vm_flags & VM_LOCKONFAULT)
1554          return nr_pages;
1555
1556      gup_flags = FOLL_TOUCH;
1557      /*
1558       * We want to touch writable mappings with a write fault in order
1559       * to break COW, except for shared mappings because these don't COW
1560       * and we would not want to dirty them for nothing.
1561       */
1562      if ((vma->vm_flags & (VM_WRITE | VM_SHARED)) == VM_WRITE)
1563          gup_flags |= FOLL_WRITE;
1564
1565      /*
1566       * We want mlock to succeed for regions that have any permissions
1567       * other than PROT_NONE.
1568       */
1569      if (vma_is_accessible(vma))
1570          gup_flags |= FOLL_FORCE;

```



```

1571
1572     /*
1573      * We made sure addr is within a VMA, so the following will
1574      * not result in a stack expansion that recurses back here.
1575      */
1576     ret = __get_user_pages(mm, start, nr_pages, gup_flags,
1577                           NULL, NULL, locked);
1578     lru_add_drain();
1579     return ret;
1580 }

```

Listing 8-62: mm/gup.c: populate_vma_page_range()

We won't examine this too closely, but the key points here are that customised logic is applied – here we always touch the pages, we make sure to write fault CoW mappings and use `FOLL_FORCE` to ensure that the process always succeeds. We eventually drain folio batches via `lru_add_drain()` to ensure folios are present in LRU lists immediately afterwards (see section 11.7 for more on folio batches).

8.2 Userland Memory Manipulation APIs

8.2.1 `mlock()`

The `mlock()` system call allows the user to specify that ranges of memory must not be subject to reclaim, but rather must be pinned into memory. This is parameterised by an address and a length over which this should be applied.

The practical upshot of this is that folios in the specified range are not placed on any LRU list (synthetically they are placed on the `LRU_UNEVICTABLE` list, but this does not exist in reality and is simply a designation indicating for the folio not to be placed on a list at all). See Section 11.2 for more details.

The `mlock2()` system call provides the same functionality, only providing a flags parameter allowing for the `MLOCK_ONFAULT` flag to be specified, indicating that the range should be locked and all future mappings should be locked as soon as they are faulted into memory.

The `munlock()` system call reverses this operation.

In addition to these, there are `mlockall()` and `munlockall()` system calls which locks and unlocks all memory in the system.

The `mlockall()` system call can specify flags, one or more of which must be specified—`MCL_CURRENT`, which indicates that all current mappings should be locked, `MCL_FUTURE`, which indicates all future mappings should be locked, and finally `MCL_ONFAULT`, which is the `mlockall()` equivalent of `MLOCK_ONFAULT` which causes locking to occur on fault for mappings not already resident in memory.

By default, locking memory in this fashion causes non-resident memory to be populated and made resident before being locked. The `MLOCK_ONFAULT`

and `MCL_ONFAULT` flags alter this behaviour, locking all resident memory and deferring the locking to the time at which they fault in.

The locking functionality is implemented using the `VM_LOCKED` flag at the struct `vm_area_struct` (VMA) level, and utilising the folio flags `PG_mlocked` and `PG_unevictable` to indicate that the memory should not be permitted to be reclaimed.

The `mlock()` system call is implemented in `mlock()` as shown in Listing 8-63.

```

615 SYSCALL_DEFINE2(mlock, unsigned long, start, size_t, len)
616 {
617     return do_mlock(start, len, VM_LOCKED);
618 }
```

Listing 8-63: mm/mlock.c: `mlock()` System Call

The `mlock()` system call defers the actual locking mechanism implemented in `do_mlock()` (shown in Listing 8-66), specifying that we want `VM_LOCKED` to be applied to the memory range, and as a consequence, to cause all of the memory in the range to have the `PG_mlocked` and `PG_unevictable` flags applied and thus to remove them from LRU lists and prevent reclaim.

The `mlock2()` system call is similar and implemented in `mlock2()` as shown in Listing 8-64.

```

620 SYSCALL_DEFINE3(mlock2, unsigned long, start, size_t, len, int, flags)
621 {
622     vm_flags_t vm_flags = VM_LOCKED;
623
624     if (flags & ~MLOCK_ONFAULT)
625         return -EINVAL;
626
627     if (flags & MLOCK_ONFAULT)
628         vm_flags |= VM_LOCKONFAULT;
629
630     return do_mlock(start, len, vm_flags);
631 }
```

Listing 8-64: mm/mlock.c: `mlock2()` System Call

The `mlock2()` system call behaves similarly to the `mlock()` system call, only it additionally checks for the `MLOCK_ONFAULT` flag (also raising an error if flags other than this were set), and if it is provided, specifies that both the `VM_LOCKED` and the `VM_LOCKONFAULT` flags should be set to `do_mlock()` (see Listing 8-66).

If no flags are specified (i.e. the flags argument is equal to zero), then only `VM_LOCKED` is provided to `do_mlock()`, and the system call behaves identically to the `mlock()` system call.

The `munlock()` system call is implemented in `munlock()` as shown in Listing 8-65.

```

633 SYSCALL_DEFINE2(munlock, unsigned long, start, size_t, len)
634 {
635     int ret;
636
637     start = untagged_addr(start);
638
639     len = PAGE_ALIGN(len + (offset_in_page(start)));
640     start &= PAGE_MASK;
641
642     if (mmap_write_lock_killable(current->mm))
643         return -EINTR;
644     ret = apply_vma_lock_flags(start, len, 0);
645     mmap_write_unlock(current->mm);
646
647     return ret;
648 }

```

Listing 8-65: mm/mlock.c: munlock() System Call

The `munlock()` system call first obtains an untagged address equivalent of the provided start address (this is relevant only to architectures which are out of scope for the book).

The entire range is page-aligned, so all base pages touched by the range will have the unlock operation applied to them.

A write lock on the struct `mm_struct->mmap_lock` semaphore is acquired then released, and within the critical section the `apply_vma_lock_flags()` function (see Listing 8-68) is used to effect the unlocking operation, specifying zero in the flags parameter to indicate that any locking VMA flags should be cleared (and with them, the underlying folio flags updated to clear locking flags and place the memory back on reclaim LRU lists).

For the sake of brevity, we won't examine the `mlockall()` and `munlockall()` system call implementations as they are a variation on the same theme.

```

568 static __must_check int do_mlock(unsigned long start, size_t len, vm_flags_t
    flags)
569 {
570     unsigned long locked;
571     unsigned long lock_limit;
572     int error = -ENOMEM;
573
574     start = untagged_addr(start);
575
576     if (!can_do_mlock())
577         return -EPERM;
578
579     len = PAGE_ALIGN(len + (offset_in_page(start)));
580     start &= PAGE_MASK;
581

```

```

582     lock_limit = rlimit(RLIMIT_MEMLOCK);
583     lock_limit >>= PAGE_SHIFT;
584     locked = len >> PAGE_SHIFT;
585
586     if (mmap_write_lock_killable(current->mm))
587         return -EINTR;
588
589     locked += current->mm->locked_vm;
590     if ((locked > lock_limit) && (!capable(CAP_IPC_LOCK))) {
591         /*
592          * It is possible that the regions requested intersect with
593          * previously mlocked areas, that part area in "mm->locked_vm"
594          * should not be counted to new mlock increment count. So
595
596         check
597          * and adjust locked count if necessary.
598          */
599         locked -= count_mm_mlocked_page_nr(current->mm,
600             start, len);
601     }
602
603     /* check against resource limits */
604     if ((locked <= lock_limit) || capable(CAP_IPC_LOCK))
605         error = apply_vma_lock_flags(start, len, flags);
606
607     mmap_write_unlock(current->mm);
608     if (error)
609         return error;
610
611     error = __mm_populate(start, len, 0);
612     if (error)
613         return __mlock_posix_error_return(error);
614     return 0;
615 }

```

Listing 8-66: mm/mlock.c: do_mlock()

In `do_mlock()` we start by acquiring an untagged address (not relevant to architectures in scope in the book), before performing a simple sanity check via `can_do_mlock()` which we examine in Listing 8-67.

```

40 bool can_do_mlock(void)
41 {
42     if (rlimit(RLIMIT_MEMLOCK) != 0)
43         return true;
44     if (capable(CAP_IPC_LOCK))
45         return true;
46     return false;
47 }

```

Listing 8-67: mm/mlock.c: can_do_mlock()

In `can_do_mlock()` we check to ensure that the `RLIMIT_MEMLOCK` resource limit has not been set to indicate that no locking at all can be specified, if this is not the case we can proceed.

However if this is the case, we also check to see whether the process has the `CAP_IPC_LOCK` capability, which overrides the limit and provides the process permission to alter memory locking regardless.

Returning to `do_mlock()` in Listing 8-66 we return with an error if these basic checks fail, then we adjust the length and start of the specified range to make them page-aligned and therefore to ensure we span all pages containing the specified range.

We attempt to acquire a write lock on the struct `mm_struct->mmap_lock` semaphore, and carefully check the `RLIMIT_MEMLOCK` limit, overriding it if the process has the `CAP_IPC_LOCK` capability.

With all checks passing, the `apply_vma_lock_flags()` function is invoked (see Listing 8-68) to apply the locking flags.

With the lock relinquished, the entire range is populated, i.e. faulted into memory via `__mm_populate()` (see Listing 8-61 and Section 8.1.9).

Importantly, `populate_vma_page_range()` (see Listing 8-62), invoked by `__mm_populate()`, will skip VMAs with the `VM_LOCKONFAULT` flag set, so this will have no impact on ranges locked via `mlock2()` with `MLOCK_ONFAULT` set.

We examine the invoked `apply_vma_lock_flags()` in Listing 8-68, eliding debug asserts.

```

468 tatic int apply_vma_lock_flags(unsigned long start, size_t len,
469                               vm_flags_t flags)
470 {
471     unsigned long nstart, end, tmp;
472     struct vm_area_struct *vma, *prev;
473     int error;
474     :
475
476     end = start + len;
477     if (end < start)
478         return -EINVAL;
479     if (end == start)
480         return 0;
481     vma = find_vma(current->mm, start);
482     if (!vma || vma->vm_start > start)
483         return -ENOMEM;
484
485     prev = vma->vm_prev;
486     if (start > vma->vm_start)
487         prev = vma;
488
489     for (nstart = start ; ; ) {
490         vm_flags_t newflags = vma->vm_flags & VM_LOCKED_CLEAR_MASK;
491

```

```

492
493         newflags |= flags;
494
495         /* Here we know that vma->vm_start <= nstart < vma->vm_end.
496
497         */
496         tmp = vma->vm_end;
497         if (tmp > end)
498             tmp = end;
499         error = mlock_fixup(vma, &prev, nstart, tmp, newflags);
500         if (error)
501             break;
502         nstart = tmp;
503         if (nstart < prev->vm_end)
504             nstart = prev->vm_end;
505         if (nstart >= end)
506             break;
507
508         vma = prev->vm_next;
509         if (!vma || vma->vm_start != nstart) {
510             error = -ENOMEM;
511             break;
512         }
513     }
514     return error;
515 }

```

Listing 8-68: mm/mlock.c: `apply_vma_lock_flags()`

The `apply_vma_lock_flags()` function starts by checking for overflow and empty ranges, before trying to find the first VMA that ends after the start of the specified range via `find_vma()` (see Listing 4-24 in Chapter 4), returning an error should either no VMA exist after the specified start or if it located VMA starts after this (both indicating that the range is invalid).

We then iterate through each of the VMAs in the range specified, clearing locking flags specified by `VM_LOCKED_CLEAR_MASK` (defined as the bitwise complement of `VM_LOCKED` and `VM_LOCKONFAULT`), before applying the specified flags.

Doing it this way allows for the same function to be used for both locking and unlocking memory ranges.

We specify `tmp` as either the end of the current VMA or the entire range being modified, whichever one comes first before invoking `mlock_fixup()` to do the heavy lifting.

After this, we exit if an error arose, before correctly accounting for the fact that `prev` can be modified by `mlock_fixup()` during a merge, before resetting everything ready for the next iteration, or if the iteration is complete, exiting, and handling the case where the next VMA in the range is invalid.

The `mlock_fixup()` function handles any splitting/merging of VMAs as a result of the application of the new VMA locking flag(s) which are duly set, before setting folio-level flags via `mlock_vma_pages_range()`.

For brevity, we will not examine these functions in detail.

8.2.2 *mprotect()*

The `mprotect()` system call allows users to alter the protection flags describing a range of virtual memory, e.g. adjusting whether the memory range permits read, write or execute access (as well as some architecture/stack-specific flags).

The `mprotect()` system call ultimately invokes `do_mprotect_pkey()` to perform the modification, which follows the typical pattern of checks to ensure the specified ranges are valid before splitting/merging VMAs accordingly via `mprotect_fixup()`.

As the behaviour of `mprotect()` is relatively straightforward, but lengthily and similar to other user memory manipulation functions explored here, we won't examine the kernel implementation in detail.

8.2.3 *mremap()*

The `mremap()` system call allows for the expansion, shrinking and/or relocation of an existing memory mapping in a process's virtual address space (i.e. that of a VMA).

By default, this operation will fail if there is insufficient free space to perform the operation (e.g. expansion of a VMA when another VMA already exists in the upper part of the range where the VMA would need to be expanded to).

However, there is a flags field which allows the specification of `MREMAP_MAYMOVE` which permits the VMA to be moved, at which point pointers to the previously mapped memory range become invalidated.

If, however, the user requires the original mapping to remain, then the `MREMAP_DONTUNMAP` flag can be used (and if so, must be used in conjunction with the `MREMAP_MAYMOVE` flag). This is only valid for private anonymous mappings, and results in the original mapping being maintained but zeroed, i.e. now pointing to the zero page, thus providing zeroes on read, and resulting in a fresh page fault on write.

Finally, the `MREMAP_FIXED` flag permits an additional parameter to be passed in which a page-aligned fixed address must be specified to indicate where the memory mapping should be relocated. If this flag is specified, then `MREMAP_MAYMOVE` must also be provided as well.

The kernel implementation of the `mremap()` system call in `mremap()` is very delicate as it must consider a great many different cases, so we will not examine it in detail for sake of brevity.

8.2.4 *madvise()*

The `madvise()` system call is used to modify attributes (provide “advice”) for a memory range. It is considered advisory, as what can be manipulated by this API are things which impact performance broadly speaking.

As a result, it is not usually guaranteed that the requested operation will be performed.

The reason the kernel exposes the ability to do so is because the user might well have more knowledge about how the memory range is intended to be used than the kernel possesses.

We examine the userland interface in Listing 8-69.

```
int madvise(void *addr, size_t length, int advice);
```

Listing 8-69: `madvise()` User API

As with all kernel memory interfaces this operates at a page granularity and thus address must be page-aligned (otherwise a `-EINVAL` error will arise) and the length will be rounded up to the page size.

What the function does depends on which flag is specified in the `advice` field. This specifies a single flag which determines what `madvise()` will do:

MADV_RANDOM Indicates that the memory region is going to be accessed randomly and thus a minimal amount of read-ahead and read-behind should be performed. Sets the `VM_RAND_READ` VMA flag. Mutually exclusive of the `MADV_NORMAL` and `MADV_SEQUENTIAL` flags.

MADV_SEQUENTIAL Indicates that the memory region is going to be accessed sequentially therefore aggressive read-ahead is appropriate. Sets the `VM_RAND_READ` VMA flag. Mutually exclusive of the `MADV_NORMAL` and `MADV_RANDOM` flags.

MADV_NORMAL Undoes any previously applied `MADV_RANDOM` or `MADV_SEQUENTIAL` in a memory region.

MADV_WILLNEED If the memory range is anonymous, swap in all memory in range and add an LRU drain so any folios are present in a non-LRU batch get placed into their appropriate LRU vectors. If the range is a `shmem` mapping, also swap the memory in. If the memory is file-backed, trigger a read-ahead of the entire range. See Chapter 9 and Section 9.7 for more details on readahead. This is equivalent to calling `posix_fadvise()` on a file specifying `POSIX_FADV_WILLNEED`.

MADV_DONTNEED Unmap the entire range of memory, i.e. remove all page table mappings and drop the reverse mapping (and thus a `_mapcount`) from the underlying folio. This is a destructive action – if the memory is anonymous, then it is *not* swapped out and the data is lost. If it is file-backed and dirty, it will not be written back to disk immediately. This operation will fail for `mlock()`ed memory.

The underlying VMA is not adjusted so if this memory is accessed again the backing store will be faulted back in (though for anonymous pages

it will be the zero page). This operation is useful for userland freeing up memory it *may* need in the future but knows it does not need now.

MADV_DONTNEED_LOCKED The same as **MADV_DONTNEED**, however it will also remove mappings for `mlock()`ed memory.

MADV_FREE (*Anonymous only*) Mark a memory range as lazy-free. This defers clearing of the memory until reclaim, but prioritises it for this. If the memory is swapped out, then the mapping is simply removed. If it is in the swap cache (see Chapter 12 for details) then the cache is freed.

MADV_REMOVE (*File-backed only*) Removes folios from a file-mapping, essentially ‘punching a hole’ in the file mapping over the specified range, which will be mapped to the zero page. This requires the mapping to be shared and writeable, the memory cannot be locked and the filesystem must support the `fallocate()` mode `FALLOC_FL_PUNCH_HOLE`.

MADV_DONTFORK Indicates that the memory range must not be copied to a child when a process forks. Sets the `VM_DONTCOPY` VMA flag.

MADV_DOFORK Reverses the effect of the **MADV_DONTFORK** operation and clears the `VM_DONTCOPY` VMA flag. Fails if the `VM_IO` VMA flag is set.

MADV_WIPEONFORK (*Anonymous only*) Marks that memory should be cleared on fork. The memory must be private, i.e. not shared. Set the `VM_WIPEONFORK` VMA flag.

MADV_KEEPONFORK (*Anonymous only*) Reverses the **MADV_WIPEONFORK** action.

MADV_HWPOISON Forms part of the memory failure functionality in the kernel. Marks the memory as if it were corrupted by hardware as a form of fault-injection. Out of scope for the book.

MADV_SOFT_OFFLINE Forms part of the memory failure functionality in the kernel. Removes memory from general usage through software means as if it were being offlined. Out of scope for the book.

MADV_MERGEABLE Forms part of the Kernel Same page Merging (KSM) functionality, indicating that the memory range is a good target for KSM. Out of scope for the book.

MADV_UNMERGEABLE Reverses the **MADV_MERGEABLE** action.

MADV_HUGEPAGE Marks the memory range as being suitable for Transparent Huge Page (THP) merging. Typically this is not required unless the `/sys/kernel/mm/transparent_hugepage/enabled` tuneable is set to `madvise`. Huge pages are out of scope for the book.

MADV_NOHUGEPAGE Marks a memory range as not being suitable to being merged into THP. Huge pages are out of scope for the book.

MADV_DONTDUMP Marks memory such that it will not appear in core dumps. Sets the `VM_DONTDUMP` VMA flag.

MADV_DODUMP Reverses the **MADV_DONTDUMP** action, though it is invalid to do this with non-hugetlb VMAs possessing flags in the `VM_SPECIAL` bitmap.

MADV_COLD Deactivates pages in the memory range, causing them to be more likely to be reclaimed soon under memory pressure. This is used to mark memory that is not likely to be used soon.

MADV_PAGEOUT A stronger version of **MADV_COLD** – direct reclaim the memory in the range immediately.

MADV_POPULATE_READ Fault-in the memory range immediately via the GUP function `faultin_vma_page_range()` by touching the memory but not writing to it, i.e. CoW will not be activated.

MADV_POPULATE_WRITE Similar to **MADV_POPULATE_READ**, only triggering write faults, i.e. CoW events where necessary.

Note that in addition to the `madvise()` system call, there is also a `process_madvise()` system call which allows you to provide memory advise for a remote process.

The `madvise()` system call is implemented in `madvise()` and the `process_madvise()` system call is implemented in `process_madvise()`. Both ultimately call `do_madvise()` which we examine in Listing 8-70 (eliding out of scope block device hotplug and memory failure logic and removal of address tagging which is not relevant to x86-64).

```

1371 int do_madvise(struct mm_struct *mm, unsigned long start, size_t len_in, int
      behavior)
1372 {
1373     unsigned long end;
1374     int error;
1375     int write;
1376     size_t len;
1377     :
1381     if (!madvise_behavior_valid(behavior))
1382         return -EINVAL;
1383
1384     if (!PAGE_ALIGNED(start))
1385         return -EINVAL;
1386     len = PAGE_ALIGN(len_in);
1387
1388     /* Check to see whether len was rounded up from small -ve to zero */
1389     if (len_in && !len)
1390         return -EINVAL;
1391
1392     end = start + len;
1393     if (end < start)
1394         return -EINVAL;
1395
1396     if (end == start)
1397         return 0;
1398     :
1404     write = madvise_need_mmap_write(behavior);

```

```

1405     if (write) {
1406         if (mmap_write_lock_killable(mm))
1407             return -EINTR;
1408     } else {
1409         mmap_read_lock(mm);
1410     }
1411     :
1412
1413     error = madvise_walk_vmas(mm, start, end, behavior,
1414                             madvise_vma_behavior);
1415     :
1416     if (write)
1417         mmap_write_unlock(mm);
1418     else
1419         mmap_read_unlock(mm);
1420
1421     return error;
1422 }

```

Listing 8-70: mm/madvise.c: do_madvise()

We can see that in `do_madvise()` (Listing 8-70) no matter what the requested action is, we both ensure that the behaviour is valid via `madvise_behavior_valid()` (which we examine in Listing 8-71) and that the start address is page-aligned, returning the `EINVAL` error code if either of these checks fail.

In addition, we perform some checks to ensure the range values specified are not so large as to overflow, and if the range turns out to be empty, we trivially indicate that the operation succeeded.

```

1131 static bool
1132 madvise_behavior_valid(int behavior)
1133 {
1134     switch (behavior) {
1135     case MADV_DOFORK:
1136     case MADV_DONTFORK:
1137     case MADV_NORMAL:
1138     case MADV_SEQUENTIAL:
1139     case MADV_RANDOM:
1140     case MADV_REMOVE:
1141     case MADV_WILLNEED:
1142     case MADV_DONTNEED:
1143     case MADV_DONTNEED_LOCKED:
1144     case MADV_FREE:
1145     case MADV_COLD:
1146     case MADV_PAGEOUT:
1147     case MADV_POPULATE_READ:
1148     case MADV_POPULATE_WRITE:
1149 #ifdef CONFIG_KSM
1150     case MADV_MERGEABLE:

```

```

1151         case MADV_UNMERGEABLE:
1152     #endif
1153     #ifdef CONFIG_TRANSPARENT_HUGEPAGE
1154         case MADV_HUGEPAGE:
1155         case MADV_NOHUGEPAGE:
1156     #endif
1157         case MADV_DONTDUMP:
1158         case MADV_DODUMP:
1159         case MADV_WIPEONFORK:
1160         case MADV_KEEPONFORK:
1161     #ifdef CONFIG_MEMORY_FAILURE
1162         case MADV_SOFT_OFFLINE:
1163         case MADV_HWPOISON:
1164     #endif
1165         return true;
1166
1167     default:
1168         return false;
1169     }
1170 }

```

Listing 8-71: mm/madvise.c: `madvise_behavior_valid()`

This ensures that the specified `madvise` flag is valid (i.e. a known one) and also that, if a feature-specific flag is specified, then that feature has to be enabled in the kernel configuration.

We then invoke `madvise_need_mmap_write()` to determine whether the action being performed needs to acquire the struct `mm_struct->mmap_lock` read-/write semaphore for writing. The remainder simply need to acquire it for reading in order that traversed VMAs remain stable throughout (see Section 4.3.4 in Chapter 4 for more details on this critical lock).

We examine this function in Listing 8-72.

```

45  /*
46   * Any behaviour which results in changes to the vma->vm_flags needs to
47   * take mmap_lock for writing. Others, which simply traverse vmAs, need
48   * to only take it for reading.
49   */
50  static int madvise_need_mmap_write(int behavior)
51  {
52      switch (behavior) {
53          case MADV_REMOVE:
54          case MADV_WILLNEED:
55          case MADV_DONTNEED:
56          case MADV_DONTNEED_LOCKED:
57          case MADV_COLD:
58          case MADV_PAGEOUT:
59          case MADV_FREE:
60          case MADV_POPULATE_READ:

```

```

61         case MADV_POPULATE_WRITE:
62             return 0;
63         default:
64             /* be safe, default to 1. list exceptions explicitly */
65             return 1;
66     }
67 }

```

Listing 8-72: mm/madvise.c: `madvice_need_mmap_write()`

In order to be conservative and to avoid performing writes without adequate locking, this function assumes locking unless a read lock is explicitly whitelisted as with the commands listed here.

Once we've determined whether a lock is required, we then acquire that lock, and perform the required behaviour using `madvice_walk_vmas()` (Which we examine in Listing 8-73) which in turn uses `madvice_vma_behavior()` to visit each VMA and perform the required action.

```

1193 static
1194 int madvice_walk_vmas(struct mm_struct *mm, unsigned long start,
1195                     unsigned long end, unsigned long arg,
1196                     int (*visit)(struct vm_area_struct *vma,
1197                                struct vm_area_struct **prev, unsigned long
1198                                start,
1199                                unsigned long end, unsigned long arg))
1200 {
1201     struct vm_area_struct *vma;
1202     struct vm_area_struct *prev;
1203     unsigned long tmp;
1204     int unmapped_error = 0;
1205
1206     /*
1207      * If the interval [start,end) covers some unmapped address
1208      * ranges, just ignore them, but return -ENOMEM at the end.
1209      * - different from the way of handling in mlock etc.
1210      */
1211     vma = find_vma_prev(mm, start, &prev);
1212     if (vma && start > vma->vm_start)
1213         prev = vma;
1214
1215     for (;;) {
1216         int error;
1217
1218         /* Still start < end. */
1219         if (!vma)
1220             return -ENOMEM;
1221
1222         /* Here start < (end/vma->vm_end). */
1223         if (start < vma->vm_start) {

```

```

1223             unmapped_error = -ENOMEM;
1224             start = vma->vm_start;
1225             if (start >= end)
1226                 break;
1227         }
1228
1229         /* Here vma->vm_start <= start < (end/vma->vm_end) */
1230         tmp = vma->vm_end;
1231         if (end < tmp)
1232             tmp = end;
1233
1234         /* Here vma->vm_start <= start < tmp <= (end/vma->vm_end). */
1235         error = visit(vma, &prev, start, tmp, arg);
1236         if (error)
1237             return error;
1238         start = tmp;
1239         if (prev && start < prev->vm_end)
1240             start = prev->vm_end;
1241         if (start >= end)
1242             break;
1243         if (prev)
1244             vma = prev->vm_next;
1245         else /* madvise_remove dropped mmap_lock */
1246             vma = find_vma(mm, start);
1247     }
1248
1249     return unmapped_error;
1250 }

```

Listing 8-73: mm/madvise.c: *madvise_walk_vmas()*

The `madvise_walk_vmas()` function iterates through all VMAs in the range specified, executing the `visit()` function for each. If at any stage a gap is located, the returned error value is set to `ENOMEM`, however the operation is still performed on all valid VMAs within the range.

We start by using `find_vma_prev()` to locate the first VMA whose `struct vm_area_struct->vm_end` exceeds `start`, and locates the prior VMA, if one exists.

If the `start` address exceeds the beginning of the located VMA, we set `prev` to the same VMA, as `prev` is intended to track the first VMA prior to the start of the range.

We then enter a loop, where `start` is updated as we traverse VMAs in the specified range.

If no VMA can be found, this indicates that `start` exceeds the range of mapped VMAs and thus we simply return `ENOMEM` and exit, as there can be no further VMAs to examine.

Since `find_vma_prev()` (and equally `find_vma()`, which it calls, which we examine in Listing 4-24 in Chapter 4) finds the first VMA whose end exceeds

the specified address, this can result in returning a VMA which begins after the address.

`madvise_walk_vmas()` addresses this by checking for this condition, marking the error condition as `ENOMEM` (as we span a gap), resetting `start` to the start of the VMA. If this exceeds the end of the specified range, then the range is invalid, there are no further VMAs to examine and we exit.

We then set `tmp` to the end of the current VMA (or the end of the range, whichever is smaller), therefore obtaining the range within the VMA which we need to apply the `visit()` function to, and call it. If an error arises from this operation, we return that error value.

Note that we pass a pointer to the `prev` value which, if the caller uses this value, will be updated.

At this point we reset `start` to the `tmp` value, which is the exclusive bound on the currently examined VMA, and thus will be one past the end of the VMA we just examined.

Since the `prev` value might be updated by the caller, set to the VMA we just examined, and a VMA merge might have taken place which could have merged this VMA with one after it, we must consider the case where the start value is located before the end of `prev`, and thus adjust it to the exclusive end value of this VMA.

We check whether we have completed the traversal (as indicated by `start` equalling or exceeding `end`), exiting the loop if so.

Finally, we either use `prev` (if it exists) to locate the next VMA to examine, or if this is not available, use `find_vma()` (see Listing 4-24) to locate the next VMA.

For more on VMA traversal, see Section 4.4.5 in Chapter 4.

When making use of the `madvise()` system call, the `madvise_walk_vmas()` function will be called with `visit` set to `madvise_vma_behavior()`, which we examine in Listing 8-74.

The only other function which might be provided to `madvise_walk_vmas()` is `madvise_vma_anon_name()`, however this is invoked by `prctl()` with `PR_SET_VMA_ANON_NAME` set, so is out of scope here.

```

987 /*
988  * Apply an madvise behavior to a region of a vma.  madvise_update_vma
989  * will handle splitting a vm area into separate areas, each area with its own
990  * behavior.
991  */
992 static int madvise_vma_behavior(struct vm_area_struct *vma,
993                                struct vm_area_struct **prev,
994                                unsigned long start, unsigned long end,
995                                unsigned long behavior)
996 {
997     int error;
998     struct anon_vma_name *anon_name;
999     unsigned long new_flags = vma->vm_flags;
1000
1001     switch (behavior) {
```

```

1002     case MADV_REMOVE:
1003         return madvise_remove(vma, prev, start, end);
1004     case MADV_WILLNEED:
1005         return madvise_willneed(vma, prev, start, end);
1006     case MADV_COLD:
1007         return madvise_cold(vma, prev, start, end);
1008     case MADV_PAGEOUT:
1009         return madvise_pageout(vma, prev, start, end);
1010     case MADV_FREE:
1011     case MADV_DONTNEED:
1012     case MADV_DONTNEED_LOCKED:
1013         return madvise_dontneed_free(vma, prev, start, end, behavior);
1014     case MADV_POPULATE_READ:
1015     case MADV_POPULATE_WRITE:
1016         return madvise_populate(vma, prev, start, end, behavior);
1017     case MADV_NORMAL:
1018         new_flags = new_flags & ~VM_RAND_READ & ~VM_SEQ_READ;
1019         break;
1020     case MADV_SEQUENTIAL:
1021         new_flags = (new_flags & ~VM_RAND_READ) | VM_SEQ_READ;
1022         break;
1023     case MADV_RANDOM:
1024         new_flags = (new_flags & ~VM_SEQ_READ) | VM_RAND_READ;
1025         break;
1026     case MADV_DONTFORK:
1027         new_flags |= VM_DONTCOPY;
1028         break;
1029     case MADV_DOFORK:
1030         if (vma->vm_flags & VM_IO)
1031             return -EINVAL;
1032         new_flags &= ~VM_DONTCOPY;
1033         break;
1034     case MADV_WIPEONFORK:
1035         /* MADV_WIPEONFORK is only supported on anonymous memory. */
1036         if (vma->vm_file || vma->vm_flags & VM_SHARED)
1037             return -EINVAL;
1038         new_flags |= VM_WIPEONFORK;
1039         break;
1040     case MADV_KEEONFORK:
1041         new_flags &= ~VM_WIPEONFORK;
1042         break;
1043     case MADV_DONTDUMP:
1044         new_flags |= VM_DONTDUMP;
1045         break;
1046     case MADV_DODUMP:
1047         if (!is_vm_hugetlb_page(vma) && new_flags & VM_SPECIAL)
1048             return -EINVAL;

```



```

1049         new_flags &= ~VM_DONTDUMP;
1050         break;
1051     case MADV_MERGEABLE:
1052     case MADV_UNMERGEABLE:
1053         error = ksm_madvise(vma, start, end, behavior, &new_flags);
1054         if (error)
1055             goto out;
1056         break;
1057     case MADV_HUGEPAGE:
1058     case MADV_NOHUGEPAGE:
1059         error = hugepage_madvise(vma, &new_flags, behavior);
1060         if (error)
1061             goto out;
1062         break;
1063     }
1064
1065     anon_name = anon_vma_name(vma);
1066     anon_vma_name_get(anon_name);
1067     error = madvise_update_vma(vma, prev, start, end, new_flags,
1068                               anon_name);
1069     anon_vma_name_put(anon_name);
1070
1071 out:
1072     /*
1073      * madvise() returns EAGAIN if kernel resources, such as
1074      * slab, are temporarily unavailable.
1075      */
1076     if (error == -ENOMEM)
1077         error = -EAGAIN;
1078     return error;
1079 }

```

Listing 8-74: mm/madvise.c: *madvise_vma_behavior()*

We won't examine the details of each behaviour supported by `madvise_vma_behavior()`, but we note that if the VMA is modified in cases which do not defer to a helper function to entirely perform the requested action, we invoke `madvise_update_vma()` to apply changes in the VMA, which we examine in Listing 8-75.

```

133 /*
134  * Update the vm_flags on region of a vma, splitting it or merging it as
135  * necessary. Must be called with mmap_sem held for writing;
136  * Caller should ensure anon_name stability by raising its refcount even when
137  * anon_name belongs to a valid vma because this function might free that vma.
138  */
139 static int madvise_update_vma(struct vm_area_struct *vma,
140                               struct vm_area_struct **prev, unsigned long
141                               start,

```

```

141             unsigned long end, unsigned long new_flags,
142             struct anon_vma_name *anon_name)
143 {
144     struct mm_struct *mm = vma->vm_mm;
145     int error;
146     pgoff_t pgoff;
147
148     if (new_flags == vma->vm_flags && anon_vma_name_eq(anon_vma_name(vma),
149 anon_name)) {
150         *prev = vma;
151         return 0;
152     }
153
154     pgoff = vma->vm_pgoff + ((start - vma->vm_start) >> PAGE_SHIFT);
155     *prev = vma_merge(mm, *prev, start, end, new_flags, vma->anon_vma,
156                     vma->vm_file, pgoff, vma_policy(vma),
157                     vma->vm_userfaultfd_ctx, anon_name);
158
159     if (*prev) {
160         vma = *prev;
161         goto success;
162     }
163
164     *prev = vma;
165
166     if (start != vma->vm_start) {
167         if (unlikely(mm->map_count >= sysctl_max_map_count))
168             return -ENOMEM;
169         error = __split_vma(mm, vma, start, 1);
170         if (error)
171             return error;
172     }
173
174     if (end != vma->vm_end) {
175         if (unlikely(mm->map_count >= sysctl_max_map_count))
176             return -ENOMEM;
177         error = __split_vma(mm, vma, end, 0);
178         if (error)
179             return error;
180     }
181
182 success:
183     /*
184     * vm_flags is protected by the mmap_lock held in write mode.
185     */
186     vma->vm_flags = new_flags;
187     if (!vma->vm_file) {
188         error = replace_anon_vma_name(vma, anon_name);

```

```

187             if (error)
188                 return error;
189         }
190
191     return 0;
192 }

```

Listing 8-75: mm/madvise.c: `madvise_update_vma()`

The `madvise_update_vma()` function assumes that only the flags or anonymous VMA name might change, so starts by checking whether either of these have changed—if not, we have nothing to do and exit.

Otherwise, we need to determine whether we can merge the VMA with ones immediately previous to or succeeding it. This is performed by VMA merging, and if this fails, splitting the VMA if the range being altered does not align with the entire length of the VMA (remembering that VMAs are defined by having the same characteristics, so if flags or the name of the VMA are altered this dictates that a new VMA is required).

The merging of the VMA is performed by `vma_merge()`, which we explore starting in Listing ?? in Section 5.1.1 and Chapter 5. VMA splitting is handled by `__split_vma()` which we explore in Listing 5-47 and Section 5.1.4 in the same chapter.

At this stage we have explored the general structure of the `madvise()` system call implementation, leaving the detailed implementation of the various operations unexplored for brevity. However exploring the functions referenced by `madvise_vma_behavior()` and shown in Listing 8-74 will indicate how each of these are implemented.

9

THE PAGE CACHE

By default all file accesses in Linux are mediated by a transparent cache that sits between userspace and the underlying disk known as the *page cache*. This significantly boosts performance when reading from/writing to files as accessing memory is far faster than accessing a disk. In this chapter we will examine how this functions in detail.

Accessing data stored on permanent storage is significantly slower than accessing it in memory (except, perhaps, in specific memory-like storage systems or others this book has yet to anticipate), ranging from greater than 4 times slower when accessing data on an SSD to greater than 80 times slower when accessing data on a traditional hard drive.

It is therefore sensible to cache this data and hold on to it as long as possible. This is known as the ‘page cache’, and all file access other than that specifically intended to bypass it (for instance direct I/O or direct access (DAX), however both are out of the scope of this book) retrieves and places data into this cache rather than accessing it directly.

This also simplifies the kernel’s handling of files—any action that would read data from or write data to disk needs to use a buffer in which to maintain this data—the page cache provides this buffer.

Reading from a file therefore becomes a matter of looking it up in the page cache and, if not present, performing the required I/O to put it there while the requesting process sleeps.

See Sections 9.3 and 9.4 below for details of how we read data from the page cache via the `read()` and memory-mapped page faults respectively.

Writing to files is rather more complicated, however hardware support in the Memory Management Unit (MMU) makes life easier—this allows us to map pages read-only, generating a page fault when written to (see Section 6.9), which we can use to notify the file system that a file is about to be written in a process termed *dirty tracking*.

We refer to this process as ‘dirty tracking’ as a page which has had data modified is termed *dirty* (see Section 10.1 for more on dirty tracking in general and Section 10.12 for more on page fault dirty tracking).

This ‘dirty’ data is synchronised to disk in a process known as *writeback* which can be triggered directly (e.g. via `fsync()`, `msync()` or `munmap()`) or indirectly as a kernel background task which periodically writes data back to disk according to system configuration (we will examine how this is performed later). We will examine the details of this in the dedicated writeback chapter.

Once the data is synchronised to disk, it is mapped read-only once again and is therefore termed *clean* as this point. This is performed by utilising the reverse mapping (see Chapter 7) to look up what maps the memory pages and then updating all of the mappings.

NOTE

RAM-backed file systems such as `tmpfs` or abstract file systems like `hugetlbfs`, whose entries in the page cache constitute the entirety of their state, of course do not require dirty tracking. Theoretically, due to the flexibility of the VFS a file system which actually writes to a permanent medium could eschew dirty tracking as a mechanism for writeback tracking, however it wouldn’t be very sensible to do so.

We have briefly touched upon writeback here, however as it is such a large topic, we defer discussion of it to the dedicated chapter on the subject.

There are a number of questions that immediately spring to mind regarding the page cache – where is it stored? How is it accounted for? How is it accessed and updated?

We will find answers for each of these, however to provide meaningful answers, we must first step back and examine the Virtual File System (VFS) which provides the generalised kernel file abstraction.

9.1 The Virtual File System (VFS)

The kernel supports a huge variety of different file systems by abstracting them through an interface termed the *Virtual File System* (VFS). Understanding how this operates is key to understanding how the page cache operates.

Discussion of the VFS and file systems as a whole is a broad topic, so we will provide a brief overview of the fundamentals before focusing upon the specifics of the page cache.

The key entity associated with any file in a Linux file system is an *inode*. This is an object containing the metadata associated with a file (noting that

directories are considered special kinds of files in Linux) and is described by the `struct inode` kernel data type.

NOTE

There is debate as to the meaning of the ‘i’, but it is generally accepted to be an abbreviation of ‘index’.

Every file and directory is described by an inode, each indexed by an *inode number* (you can examine these by running `ls -li` in a shell) maintained in `struct inode->i_ino`.

Userland processes typically don’t interact with inodes directly, but rather with a *file descriptor* (fd) obtained by `open()`ing a file. Each of a process’s open files is tracked by the kernel in a `struct file` object, which references the opened inode via their `f_inode` field, the current offset within the file via `f_pos`, the mode in which the file is opened via `f_mode`, file path and other metadata.

NOTE

Of course the `open()` process might be abstracted by something like `fopen()` or another library function, underlying it however is the same open system call.

As a result, multiple `struct file` objects may exist referencing the same underlying file, each representing a mapping of that file.

Each object within the page cache is uniquely described by a `struct address_space` object, to which each file-backed `struct folio` (and equally `struct page`) refers in their mapping field, as well as `struct inode` objects via their `i_mapping` field and `struct file` objects via their `f_mapping` fields.

NOTE

From the perspective of the memory management subsystem, the `struct address_space` object is the most important metadata for a file or other cachable entity – it represents an entry in the page cache itself.

Each instance of a mounted filesystem has its core metadata described by a *super block* object, `struct super_block`, under which all inodes are maintained in the `s_inodes` linked list.

Each of these objects have customised functions associated with them which permits a file system to provide its own custom functionality:

- `struct inode->i_op` specifies the functions the file system provides for inode operations in an `struct inode_operations` object. This is typically set when a new inode is first initialised.
- `struct file->f_op` specifies the functions the file system provides for file operations in an `struct file_operations` object. This is obtained from each inode’s `struct inode->i_fop` field on file open in `do_dentry_open()`. The `i_fop` field is typically set on inode or super block initialisation.
- `struct address_space->a_ops` specifies the functions the file system provides for address space operations in a `struct address_space_operations` object. This, like `struct inode->i_op`, is typically set on inode initialisation.

- `struct super_block->s_op` specifies the functions the file system provides for super block operations in a `struct super_operations` object. This is set on super block initialisation.
- `struct vm_area_struct->vm_ops` specifies the functions the file system provides for VMA operations when the region described by the VMA is memory-mapped. These callbacks are wrapped in a `struct vm_operations_struct` type. This is set when the `struct file_operations->mmap` callback is invoked when the VMA is first mapped.

We will examine the operations relevant to the page cache in detail. We examine the relationships between VFS objects as a whole in Figure 9-1.

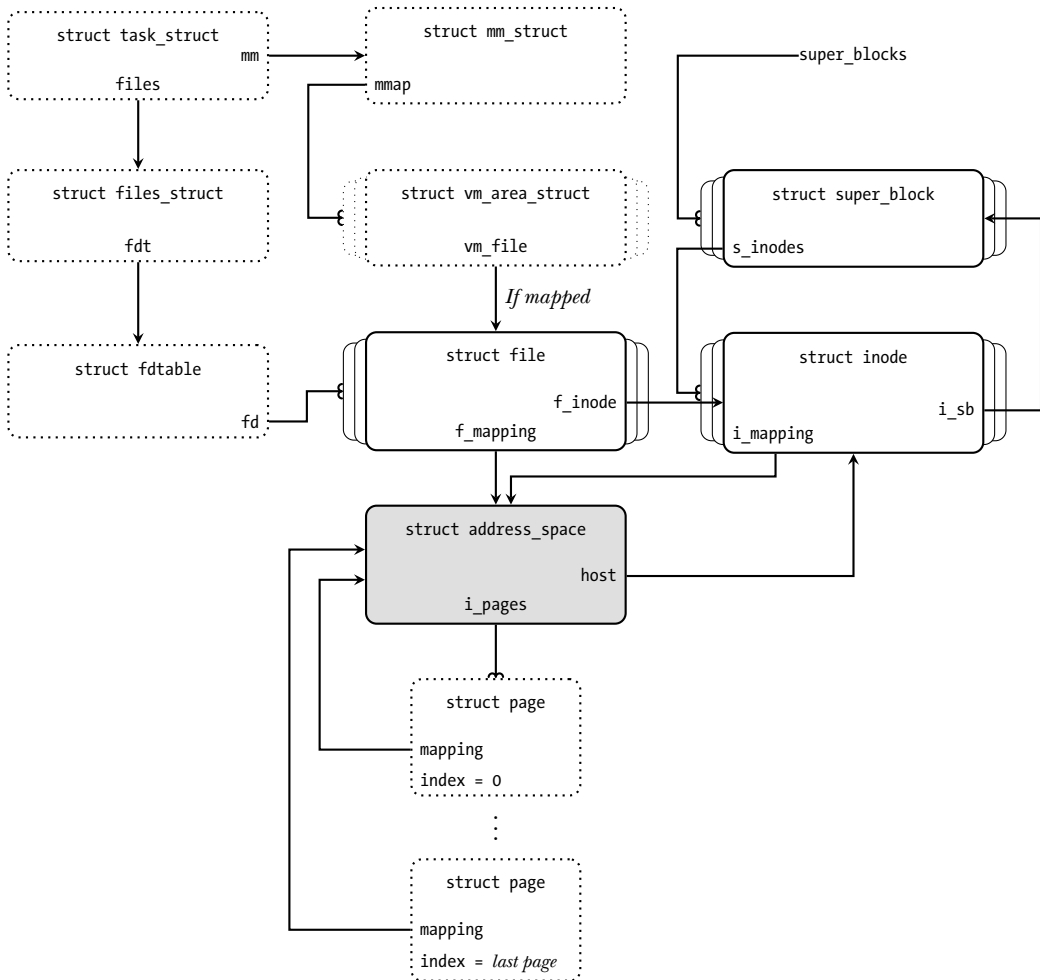


Figure 9-1: Core Virtual File System (VFS) Objects

Underlying it all is the core struct `address_space` object, which defines a *mappable page cache entry*, i.e. an object which can exist within the page cache, typically a file.

Where does the struct `address_space` ‘mapping’ originate from? It points to the struct `inode->i_data` field, set in `inode_init_always()` once initialisation of the inode and mapping object are complete, invoked via `alloc_inode()`, which is called by file systems when establishing inodes.

9.2 A Brief Digression: eXtensible Arrays (xarrays)

A fundamental data structure used within the page cache is the *xarray* (eXtensible array), and as such is worth examining in detail. The intent of the xarray is to implement a data structure which possesses the following properties:

- Maps system word size indices to pointer or numerical values while remaining space and cache efficient when traversing adjacent entries. Accessing memory immediately before or after an address is vastly easier on a modern CPU’s caches than accessing arbitrary pointers such as in a doubly-linked list.
- Expands and contracts efficiently when new elements are added or removed.
- Perform lockless *reads* via the *RCU* mechanism.
- Automatically performs locking on write operations which require it.
- Is able to map a *range* of keys to a single value.
- Allows elements to be assigned a ‘mark’ which can then be separately and efficiently iterated over to enable fast categorisation of elements.

NOTE

Read-Create-Update (RCU) is a locking mechanism which permits users to read data without having to acquire a lock or perform a memory fence/atomic operation. It does this by deferring writes to a ‘grace period’ during which there can be no reads. Using the RCU can significantly improve performance for read-mostly data structures.

Considering alternatives – A dynamic array is not at all space efficient for arbitrary keys, a hash is but has poor cache performance as does a red/black tree, a B-tree needs to store key/value pairs so is not space efficient and the existing kernel radix tree implementation had poor worst-case space and time efficiency (an unfortunate index could result in significantly inflated tree depth).

An xarray can contain two kinds of data – pointers (which must be 4 byte-aligned, we will examine why shortly) and *signed, positive* numeric values which can fit inside a pointer (i.e. numbers from zero to `LONG_MAX` inclusive).

An xarray is represented by a lightweight struct `xarray` object (the fundamental user of this type in the page cache is struct `address_space->i_pages`), shown in Listing 9-1.

```

280 /**
281  * struct xarray - The anchor of the XArray.
282  * @xa_lock: Lock that protects the contents of the XArray.
283  *
284  * To use the xarray, define it statically or embed it in your data structure.
285  * It is a very small data structure, so it does not usually make sense to
286  * allocate it separately and keep a pointer to it in your data structure.
287  *
288  * You may use the xa_lock to protect your own data structures as well.
289  */
290 /*
291  * If all of the entries in the array are NULL, @xa_head is a NULL pointer.
292  * If the only non-NULL entry in the array is at index 0, @xa_head is that
293  * entry. If any other entry in the array is non-NULL, @xa_head points
294  * to an @xa_node.
295  */
296 struct xarray {
297     spinlock_t      xa_lock;
298     /* private: The rest of the data structure is not to be used directly. */
299     gfp_t           xa_flags;
300     void __rcu *     xa_head;
301 };

```

Listing 9-1: include/linux/xarray.h: struct xarray

This consists of a spin lock (xa_lock) for operations which cannot be performed under *RCU* alone, flags (xa_flags) where xarray flags can be specified and the head of the xarray tree xa_head. The latter two fields being private and not intended for use outside of the xarray code.

We can see what a newly allocated xarray object looks like by examining XARRAY_INIT(), shown in Listing 9-2.

```

303 #define XARRAY_INIT(name, flags) {                                \
304     .xa_lock = __SPIN_LOCK_UNLOCKED(name.xa_lock),              \
305     .xa_flags = flags,                                           \
306     .xa_head = NULL,                                             \
307 }

```

Listing 9-2: include/linux/xarray.h: XARRAY_INIT()

As per the comment in listing 9-1, a NULL entry indicates that the array is entirely empty and if the array consists of only one entry at index 0, it will be contained in xa_head.

What values can be elements in an xarray? This is described in the comment for BITS_PER_XA_VALUE, shown in Listing 9-3.

```

23 /*
24  * The bottom two bits of the entry determine how the XArray interprets
25  * the contents:

```

```

26  *
27  * 00: Pointer entry
28  * 10: Internal entry
29  * x1: Value entry or tagged pointer
30  *
31  * Attempting to store internal entries in the XArray is a bug.
32  *
33  * Most internal entries are pointers to the next node in the tree.
34  * The following internal entries have a special meaning:
35  *
36  * 0-62: Sibling entries
37  * 256: Retry entry
38  * 257: Zero entry
39  *
40  * Errors are also represented as internal entries, but use the negative
41  * space (-4094 to -2). They're never stored in the slots array; only
42  * returned by the normal API.
43  */
44
45 #define BITS_PER_XA_VALUE      (BITS_PER_LONG - 1)

```

Listing 9-3: include/linux/xarray.h: `BITS_PER_XA_VALUE()`

This explains why pointers referenced by an xarray must be 4-byte aligned – to accommodate for these identifiers in the lower two bits for pointer types.

This also explains why integer values stored within an xarray must be a *positive* value of `LONG_MAX` or less – we give up a bit for these value to mark them as such.

A user of the xarray API should not encounter internal entries unless a bug arises, so need only concern themselves with whether the entry is a value or not via `xa_is_value()`, shown in Listing 9-4.

```

72 /**
73  * xa_is_value() - Determine if an entry is a value.
74  * @entry: XArray entry.
75  *
76  * Context: Any context.
77  * Return: True if the entry is a value, false if it is a pointer.
78  */
79 static inline bool xa_is_value(const void *entry)
80 {
81     return (unsigned long)entry & 1;
82 }

```

Listing 9-4: include/linux/xarray.h: `xa_is_value()`

We can then extract a value entry via `xa_to_value()`, shown in Listing 9-5.

```

60 /**

```

```

61  * xa_to_value() - Get value stored in an XArray entry.
62  * @entry: XArray entry.
63  *
64  * Context: Any context.
65  * Return: The value stored in the XArray entry.
66  */
67  static inline unsigned long xa_to_value(const void *entry)
68  {
69      return (unsigned long)entry >> 1;
70  }

```

Listing 9-5: include/linux/xarray.h: `xa_to_value()`

And generate it in the first place with `xa_mk_value()`, shown in Listing 9-6.

```

47  /**
48  * xa_mk_value() - Create an XArray entry from an integer.
49  * @v: Value to store in XArray.
50  *
51  * Context: Any context.
52  * Return: An entry suitable for storing in the XArray.
53  */
54  static inline void *xa_mk_value(unsigned long v)
55  {
56      WARN_ON((long)v < 0);
57      return (void *)((v << 1) | 1);
58  }

```

Listing 9-6: include/linux/xarray.h: `xa_mk_value()`

However, of course the internal xarray implementation must reference internal elements. We can determine whether an entry is internal via `xa_is_internal()`, shown in Listing 9-7.

```

162 /*
163  * xa_is_internal() - Is the entry an internal entry?
164  * @entry: XArray entry.
165  *
166  * Context: Any context.
167  * Return: %true if the entry is an internal entry.
168  */
169  static inline bool xa_is_internal(const void *entry)
170  {
171      return ((unsigned long)entry & 3) == 2;
172  }

```

Listing 9-7: include/linux/xarray.h: `xa_is_internal()`

And extract it via `xa_to_internal()`, shown in Listing 9-8.

```

150 /*

```

```

151 * xa_to_internal() - Extract the value from an internal entry.
152 * @entry: XArray entry.
153 *
154 * Context: Any context.
155 * Return: The value which was stored in the internal entry.
156 */
157 static inline unsigned long xa_to_internal(const void *entry)
158 {
159     return (unsigned long)entry >> 2;
160 }

```

Listing 9-8: include/linux/xarray.h: `xa_to_internal()`

Finally, these values can be created via `xa_mk_internal()`, shown in Listing 9-9.

```

132 /*
133 * xa_mk_internal() - Create an internal entry.
134 * @v: Value to turn into an internal entry.
135 *
136 * Internal entries are used for a number of purposes. Entries 0-255 are
137 * used for sibling entries (only 0-62 are used by the current code). 256
138 * is used for the retry entry. 257 is used for the reserved / zero entry.
139 * Negative internal entries are used to represent errnos. Node pointers
140 * are also tagged as internal entries in some situations.
141 *
142 * Context: Any context.
143 * Return: An XArray internal entry corresponding to this value.
144 */
145 static inline void *xa_mk_internal(unsigned long v)
146 {
147     return (void *)((v << 2) | 2);
148 }

```

Listing 9-9: include/linux/xarray.h: `xa_mk_internal()`

As kernel pointers won't sensibly be placed in the first or last 4 KiB of virtual address space, we can use these to store special values.

We utilise the space in the uppermost portion of the virtual address space to be able to store errors – these are encoded as negative integer values, up to a maximum of -4095 (i.e. `-MAX_ERRNO`).

This way, errors can be passed back to callers by encoding a negative error value is an internal entry. Users can check this via `xa_is_err()`, shown in Listing 9-10.

```

190 /**
191 * xa_is_err() - Report whether an XArray operation returned an error
192 * @entry: Result from calling an XArray function
193 *
194 * If an XArray operation cannot complete an operation, it will return

```

```

195  * a special value indicating an error. This function tells you
196  * whether an error occurred; xa_err() tells you which error occurred.
197  *
198  * Context: Any context.
199  * Return: %true if the entry indicates an error.
200  */
201  static inline bool xa_is_err(const void *entry)
202  {
203      return unlikely(xa_is_internal(entry) &&
204                     entry >= xa_mk_internal(-MAX_ERRNO));
205  }

```

Listing 9-10: include/linux/xarray.h: `xa_is_err()`

These error values can be extracted via `xa_err()`, shown in Listing 9-11.

```

207  /**
208   * xa_err() - Turn an XArray result into an errno.
209   * @entry: Result from calling an XArray function.
210   *
211   * If an XArray operation cannot complete an operation, it will return
212   * a special pointer value which encodes an errno. This function extracts
213   * the errno from the pointer value, or returns 0 if the pointer does not
214   * represent an errno.
215   *
216   * Context: Any context.
217   * Return: A negative errno or 0.
218   */
219  static inline int xa_err(void *entry)
220  {
221      /* xa_to_internal() would not do sign extension. */
222      if (xa_is_err(entry))
223          return (long)entry >> 2;
224      return 0;
225  }

```

Listing 9-11: include/linux/xarray.h: `xa_err()`

Internal entries *typically* provide a pointer to a struct `xa_node` object which forms part of a tree. An xarray node is described by the struct `xa_node` type and check for by `xa_is_node()`, shown in Listing 9-12.

```

1239  /* Private */
1240  static inline bool xa_is_node(const void *entry)
1241  {
1242      return xa_is_internal(entry) && (unsigned long)entry > 4096;
1243  }

```

Listing 9-12: include/linux/xarray.h: `xa_is_node()`

Converted to a struct `xa_node` via `xa_to_node()`, shown in Listing 9-13.

```

1233 /* Private */
1234 static inline struct xa_node *xa_to_node(const void *entry)
1235 {
1236     return (struct xa_node *)((unsigned long)entry - 2);
1237 }

```

Listing 9-13: include/linux/xarray.h: `xa_to_node()`

And constructed via `xa_mk_node()`, shown in Listing 9-14.

```

1227 /* Private */
1228 static inline void *xa_mk_node(const struct xa_node *node)
1229 {
1230     return (void *)((unsigned long)node | 2);
1231 }

```

Listing 9-14: include/linux/xarray.h: `xa_mk_node()`

When an internal entry does not refer to a node or an error, it utilises the lower part of the virtual address space range up to be one of the following special cases (as described in the comment in listing 9-3):

Sibling entry 0 - 62 – Used only in the *advanced* xarray API. This indicates that multiple indices span a single entry in the xarray, with the value indicating the offset *this* entry is from the spanned one.

Retry entry 256 – XA_RETRY_ENTRY – Used only in the *advanced* xarray API. Indicates that another thread holds the `xa_lock` and is midway through updating the entry. To avoid obtaining a reference that might be dropped momentarily, restart the operation.

Zero entry 257 – XA_ZERO_ENTRY – Indicates that the entry is non-empty, but represents the NULL value. In the normal API (see Section 9.2.1 below for a discussion on xarray APIs) these are simply expressed as NULL values, however the advanced API can encounter these values directly.

That this is a *node* gives a hint to the internal structure of an xarray – it is fundamentally a radix tree, though optimised to handle arbitrary address ranges as its key, improving upon the prior kernel radix tree implementation.

We examine the `struct xa_node` data structure in Listing 9-15.

```

1133 /*
1134  * @count is the count of every non-NULL element in the ->slots array
1135  * whether that is a value entry, a retry entry, a user pointer,
1136  * a sibling entry or a pointer to the next level of the tree.
1137  * @nr_values is the count of every element in ->slots which is
1138  * either a value entry or a sibling of a value entry.
1139  */
1140 struct xa_node {
1141     unsigned char  shift;           /* Bits remaining in each slot */
1142     unsigned char  offset;         /* Slot offset in parent */

```

```

1143     unsigned char   count;           /* Total entry count */
1144     unsigned char   nr_values;       /* Value entry count */
1145     struct xa_node __rcu *parent;    /* NULL at top of tree */
1146     struct xarray   *array;         /* The array we belong to */
1147     union {
1148         struct list_head private_list; /* For tree user */
1149         struct rcu_head rcu_head;      /* Used when freeing node */
1150     };
1151     void __rcu      *slots[XA_CHUNK_SIZE];
1152     union {
1153         unsigned long tags[XA_MAX_MARKS][XA_MARK_LONGS];
1154         unsigned long marks[XA_MAX_MARKS][XA_MARK_LONGS];
1155     };
1156 };

```

Listing 9-15: include/linux/xarray.h: struct xa_node

Each individual node contains XA_CHUNK_SIZE ‘slots’, containing pointers. This, for anything other than a tiny embedded system will be 2^6 , or 64 pointers (512 bytes for x86-64). Each of these slots consist of entries, some of which might be further nodes.

Rather, similar to a dynamic array, the xarray doubles in capacity when it runs out of space. Unlike a dynamic array, an xarray achieves this by installing new struct xa_nodes (the first expansion installing one in struct xarray->xa_head with a node entry pointing at the previous head placed in the first slot).

This parent node is inserted at the head with its shift increased by XA_CHUNK_SIZE, a value which indicates the range of indices over which each node entry spans, e.g. a shift of 6 indicates that the node in slot entry 1 is offset by 2^6 i.e. 64 indices, entry 2 by 128, etc.

Within an xarray tree, each index is simply determined to be the *i*th leaf node (accounting for entries which span multiple indices which are indexed over their whole range) so by expanding the tree in the fashion described above we only increment the maximum used index within the array.

Let’s examine the case of a *single* value in an xarray, to which we add one more value (all diagrams below use * to indicate value and N to indicate a pointer to a node), shown in Figure 9-2.

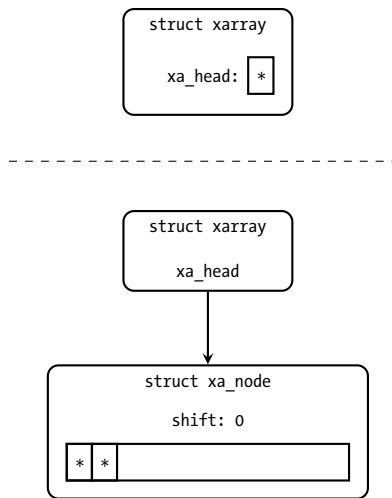


Figure 9-2: Expanding an xarray From One Element to Two

In the case of an xarray containing one or no elements, no additional memory need be allocated and instead the value can simply be placed in the `struct xarray->xa_head` field.

Now let's examine how things look once we populate an entire node's `XA_CHUNK_SIZE` slot entries, then expand that by a single value, as shown in Figure 9-3.

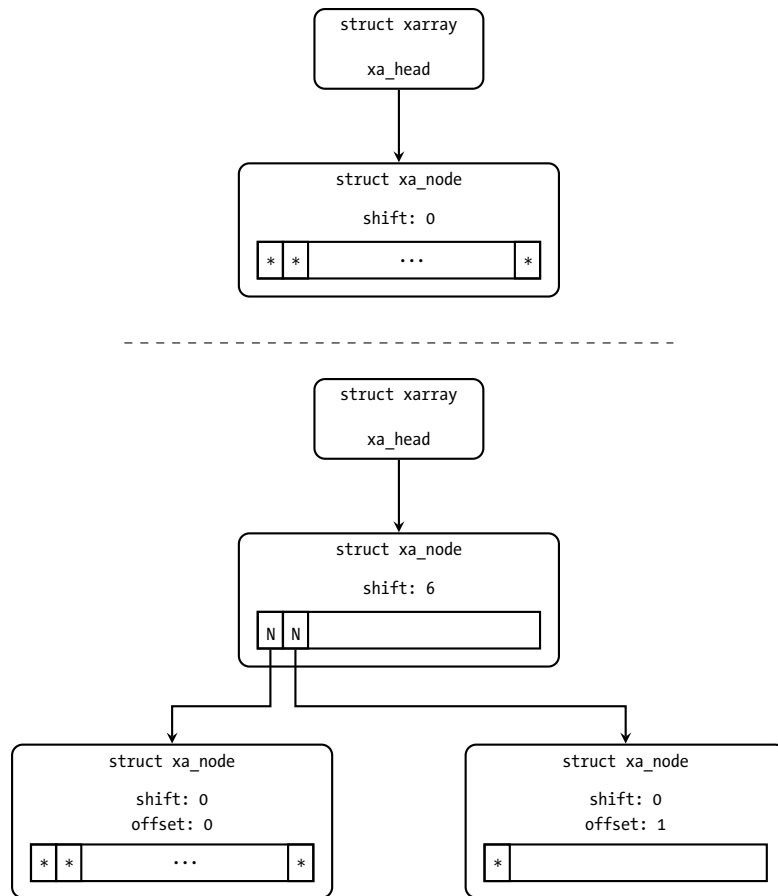


Figure 9-3: Expanding an xarray of `XA_CHUNK_SIZE` Elements By 1

We perform the minimum amount of work required to expand the xarray by preserving the existing `struct xa_node` and referencing it in an entry in a new `struct xa_node` which we assign `XA_CHUNK_SHIFT`, which here we take to be 6 (indicating 2^6 `XA_CHUNK_SIZE` i.e. 64).

This therefore indicates that all entries in the topmost `struct xa_node` are offset by `XA_CHUNK_SIZE` multiplied by the index within the array. We encapsulate this in the new `struct xa_node` object we allocate and assign to the second entry in the topmost `struct xa_node`'s array, which adds the 65th entry. This node has offset of 1, and a combination of the parent's shift and leaf's offset tells us the location of each value within the array.

The next expansion will occur when the topmost `struct xa_node` becomes full. Since each node contains `XA_CHUNK_SIZE` entries and the top-level node stores `XA_CHUNK_SIZE` references to leaf nodes, each containing `XA_CHUNK_SIZE` values, we can therefore store XA_CHUNK_SIZE^2 entries, or 4,096 with `XA_CHUNK_SIZE` of 64, meaning that we must expand the xarray after we reach this point as shown in Figure 9-4.

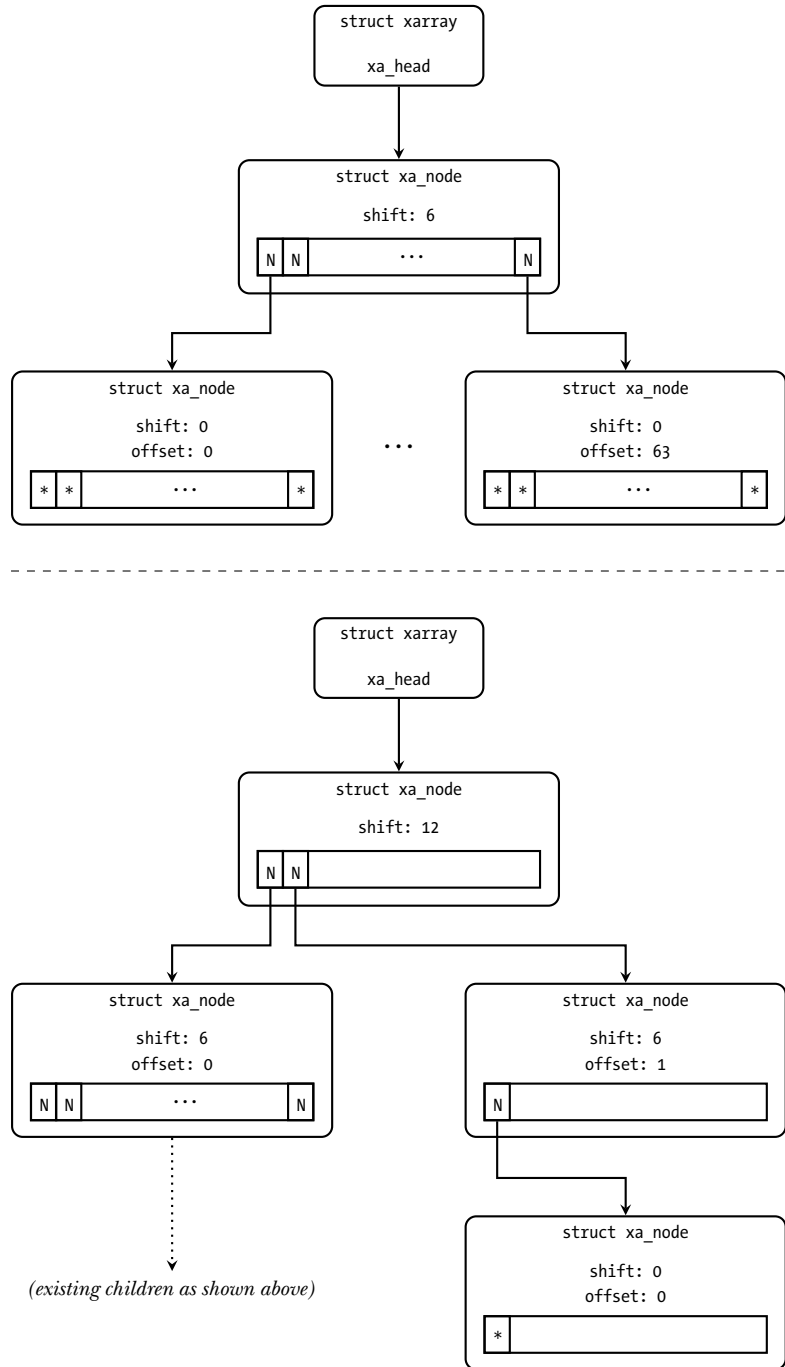


Figure 9-4: Expanding an xarray of $\text{XA\_CHUNK\_SIZE}^2 + 1$ Elements By 1

Once again the procedure at a top-level here is quite simple—we allocate a new `struct xa_node` which we place immediately below the `struct xarray` in the head position, and we assign the previous node as the first element in the new array.

The expansion is slightly more involved, as we allocate *two* nodes rather than one in order to add a new value entry at the correct shift level.

NOTE

If entries are added that skip indices (e.g. xarray of size 1, insert an entry at index 4097), then empty entries are added to pad out the xarray to incorporate the newly inserted entry.

So far we've examined value entries that exist only in leaf nodes, i.e. at shift 0. It is in fact possible to maintain entries at higher shifts that span multiple indices, e.g. representing a folio of higher order (an order-1 folio spans 2 subpages, an order-2 folio spans 4, etc., see Sections ?? and ?? for more details).

In order to accommodate such a higher-order value entry for shift levels other than those which naturally arise on xarray expansion, the process for storing these values is to find the highest shift level less than or equal to the number of spanned indices, and to use 'sibling' entries immediately adjacent to it to extend the occupied range.

For instance, if I wish to span 2⁹ indices, The entry for this can be placed in a shift 6 node (where each entry spans 64 indices) followed by 7 sibling entries (therefore spanning 64×8 i.e. 512 indices as required).

We have mentioned `XA_CHUNK_SIZE` above, so let's examine the definition in Listing 9-16.

```

1115 /*
1116  * The xarray is constructed out of a set of 'chunks' of pointers. Choosing
1117  * the best chunk size requires some tradeoffs. A power of two recommends
1118  * itself so that we can walk the tree based purely on shifts and masks.
1119  * Generally, the larger the better; as the number of slots per level of the
1120  * tree increases, the less tall the tree needs to be. But that needs to be
1121  * balanced against the memory consumption of each node. On a 64-bit system,
1122  * xa_node is currently 576 bytes, and we get 7 of them per 4kB page. If we
1123  * doubled the number of slots per node, we'd get only 3 nodes per 4kB page.
1124  */
1125 #ifndef XA_CHUNK_SHIFT
1126 #define XA_CHUNK_SHIFT      (CONFIG_BASE_SMALL ? 4 : 6)
1127 #endif
1128 #define XA_CHUNK_SIZE      (1UL << XA_CHUNK_SHIFT)
1129 #define XA_CHUNK_MASK      (XA_CHUNK_SIZE - 1)
1130 #define XA_MAX_MARKS      3
1131 #define XA_MARK_LONGS      DIV_ROUND_UP(XA_CHUNK_SIZE, BITS_PER_LONG)

```

Listing 9-16: `include/linux/xarray.h`: `XA_CHUNK_SIZE`

It's very useful to associate metadata with xarray entries, so we do so by using *tags* and *marks* associated with each node up to `XA_MAX_MARKS` (3), with

each bit of the tags and marks fields used to tag entries in the chunk of entries contained in this node.

These are stored in a union so are mutually exclusive – the difference between the two are that marks are propagated up through the tree, making it easier to efficiently *search* for marked entries, while tags are not.

Having this facility is very useful, as when traversing a large xarray, it's useful to differentiate between different elements and to, for instance, search only specific kinds of entries which we can indicate by marking or tagging them.

This is used extensively by the page cache mechanism to mark dirty/writeback folios as you will observe later in the chapter.

9.2.1 API

There are two separate APIs provided in the *xarray* implementation – the ‘normal’ API and the ‘advanced’ API. The former handles the locking automatically and provides a relatively straightforward interface, with the latter offering more flexibility but requiring the user to correct perform locking as required.

We won't examine the code deeply at all, but as a taster let's examine a basic function in the normal API, `xa_store()`, shown in Listing 9-17.

```

1560 /**
1561  * xa_store() - Store this entry in the XArray.
1562  * @xa: XArray.
1563  * @index: Index into array.
1564  * @entry: New entry.
1565  * @gfp: Memory allocation flags.
1566  *
1567  * After this function returns, loads from this index will return @entry.
1568  * Storing into an existing multi-index entry updates the entry of every index
1569  * .
1570  * The marks associated with @index are unaffected unless @entry is %NULL.
1571  *
1572  * Context: Any context. Takes and releases the xa_lock.
1573  * May sleep if the @gfp flags permit.
1574  * Return: The old entry at this index on success, xa_err(-EINVAL) if @entry
1575  * cannot be stored in an XArray, or xa_err(-ENOMEM) if memory allocation
1576  * failed.
1577  */
1578 void *xa_store(struct xarray *xa, unsigned long index, void *entry, gfp_t gfp)
1579 {
1580     void *curr;
1581
1582     xa_lock(xa);
1583     curr = __xa_store(xa, index, entry, gfp);
1584     xa_unlock(xa);

```

```

1585         return curr;
1586     }

```

Listing 9-17: lib/xarray.c: xa_store()

This stores the entry entry into the struct xarray xa at index index. As you can see, the struct xarray->xa_lock is acquired here on the behalf of the user.

The heavy lifting is performed by __xa_store() (eliding out of scope entry free tracking logic), shown in Listing 9-18.

```

1525  /**
1526   * __xa_store() - Store this entry in the XArray.
1527   * @xa: XArray.
1528   * @index: Index into array.
1529   * @entry: New entry.
1530   * @gfp: Memory allocation flags.
1531   *
1532   * You must already be holding the xa_lock when calling this function.
1533   * It will drop the lock if needed to allocate memory, and then reacquire
1534   * it afterwards.
1535   *
1536   * Context: Any context. Expects xa_lock to be held on entry. May
1537   * release and reacquire xa_lock if @gfp flags permit.
1538   * Return: The old entry at this index or xa_err() if an error happened.
1539   */
1540  void *__xa_store(struct xarray *xa, unsigned long index, void *entry, gfp_t
    gfp)
1541  {
1542      XA_STATE(xas, xa, index);
1543      void *curr;
1544
1545      if (WARN_ON_ONCE(xa_is_advanced(entry)))
1546          return XA_ERROR(-EINVAL);
1547      :
1550      do {
1551          curr = xas_store(&xas, entry);
1552          :
1554      } while (__xas_nomem(&xas, gfp));
1555
1556      return xas_result(&xas, curr);
1557  }

```

Listing 9-18: lib/xarray.c: __xa_store()

The function asserts that the entry does not comprise any advanced API entity via xa_is_advanced(), such as XA_RETRY_ENTRY or a sibling entry, as the basic API calls are not suited to such operations.

The rest of the logic relies on an advanced API feature, which is used internally here. The advanced API features tend to use a state iterator variable

of type `struct xa_state`, which is threaded through function calls. We examine this object in Listing 9-19.

```

1309  /*
1310  * The xa_state is opaque to its users. It contains various different pieces
1311  * of state involved in the current operation on the XArray. It should be
1312  * declared on the stack and passed between the various internal routines.
1313  * The various elements in it should not be accessed directly, but only
1314  * through the provided accessor functions. The below documentation is for
1315  * the benefit of those working on the code, not for users of the XArray.
1316  *
1317  * @xa_node usually points to the xa_node containing the slot we're operating
1318  * on (and @xa_offset is the offset in the slots array). If there is a
1319  * single entry in the array at index 0, there are no allocated xa_nodes to
1320  * point to, and so we store %NULL in @xa_node. @xa_node is set to
1321  * the value %XAS_RESTART if the xa_state is not walked to the correct
1322  * position in the tree of nodes for this operation. If an error occurs
1323  * during an operation, it is set to an %XAS_ERROR value. If we run off the
1324  * end of the allocated nodes, it is set to %XAS_BOUNDS.
1325  */
1326  struct xa_state {
1327      struct xarray *xa;
1328      unsigned long xa_index;
1329      unsigned char xa_shift;
1330      unsigned char xa_sibs;
1331      unsigned char xa_offset;
1332      unsigned char xa_pad;           /* Helps gcc generate better code */
1333      struct xa_node *xa_node;
1334      struct xa_node *xa_alloc;
1335      xa_update_node_t xa_update;
1336      struct list_lru *xa_lru;
1337  };

```

Listing 9-19: `include/linux/xarray.h: struct xa_state`

This object is used for traversal as well as updating of xarray elements. It maintains state about an entry we are interacting with, for example searching for or newly inserting.

Examining each field, eliding the padding and out of scope LRU fields:

- `xa` – The `struct xarray` object we are interacting with.
- `xa_index` – The index within the xarray we are interested in. Entries in the xarray start at zero and an entry's index is determined by it being the *i*th leaf node, offsetting accordingly for any entry which spans a range of indices.
- `xa_shift` – Indicates the level at which this entry will be placed. This is only relevant to entries which span multiple indices, ordinary single-index entries will have `xa_shift` of zero.

- `xa_sibs` – The number of sibling entries this entry requires.
- `xa_offset` – The offset of this entry within a given node.
- `xa_node` – The node within which the entry is located.
- `xa_alloc` – A pointer to newly allocated memory to be used for performing this operation.
- `xa_update` – An optional `xa_update_node_t` callback function provided by the user through the advanced API which is invoked every time either `count` or `nr_values` is updated in the struct `xa_node` node containing this entry.

The struct `xa_state` object is often initialised via the `XA_STATE()` macro, shown in Listing 9-20.

```

1360 /**
1361  * XA_STATE() - Declare an XArray operation state.
1362  * @name: Name of this operation state (usually xas).
1363  * @array: Array to operate on.
1364  * @index: Initial index of interest.
1365  *
1366  * Declare and initialise an xa_state on the stack.
1367  */
1368 #define XA_STATE(name, array, index)
1369         struct xa_state name = __XA_STATE(array, index, 0, 0)

```

Listing 9-20: `include/linux/xarray.h: XA_STATE()`

Which, in turn, invokes the `__XA_STATE()` macro to actually assign field values, shown in Listing 9-21.

```

1347 #define __XA_STATE(array, index, shift, sibs) {
1348         .xa = array,
1349         .xa_index = index,
1350         .xa_shift = shift,
1351         .xa_sibs = sibs,
1352         .xa_offset = 0,
1353         .xa_pad = 0,
1354         .xa_node = XAS_RESTART,
1355         .xa_alloc = NULL,
1356         .xa_update = NULL,
1357         .xa_lru = NULL,
1358 }

```

Listing 9-21: `include/linux/xarray.h: __XA_STATE()`

The `__XA_STATE()` macro can be used directly for cases of entries which span multiple indices, otherwise `XA_STATE()` is sufficient.

Additionally, a user wishing to interact with an entry spanning multiple indices, a convenient macro to configure a struct `xa_state` object is `XA_STATE_ORDER()`, shown in Listing 9-22.

```

1371 /**
1372  * XA_STATE_ORDER() - Declare an XArray operation state.
1373  * @name: Name of this operation state (usually xas).
1374  * @array: Array to operate on.
1375  * @index: Initial index of interest.
1376  * @order: Order of entry.
1377  *
1378  * Declare and initialise an xa_state on the stack. This variant of
1379  * XA_STATE() allows you to specify the 'order' of the element you
1380  * want to operate on.
1381  */
1382 #define XA_STATE_ORDER(name, array, index, order)          \
1383     struct xa_state name = __XA_STATE(array,              \
1384                                     (index >> order) << order, \
1385                                     order - (order % XA_CHUNK_SHIFT), \
1386                                     (1U << (order % XA_CHUNK_SHIFT)) - 1)

```

Listing 9-22: include/linux/xarray.h: XA_STATE_ORDER()

This is actually rather useful in understanding entries spanning a range of indices – order indicates the power of two number of indices to span (since the primary use case again is the page cache which can comprise higher order folios, use of this terminology is helpful).

This aligns the index of the entry, determines the next lowest shift value which can accommodate it (subtracting the remainder after dividing the order by chunk size) and calculating the number of siblings to be one less than this offset in bytes.

Note that there are certain `xa_node` values which have particular meanings:

- `XA_ERROR()` – Indicates that an error has occurred identically to an error being reported for an entry that can be checked via `xa_is_err()` (as described in listing 9-10).
- `XAS_BOUNDS` – Indicates that the current iteration has run past the end of the currently available indices in any node.
- `XAS_RESTART` – Indicates that the iterator is not in a correct state for traversal and must be restarted. This is the initial default value.

9.2.2 The Rest

We have touched on `xa_store()` as an example of how the xarray API is implemented, and examined core data structures used through an xarray traversal.

If we were to dive deep into the API this would require an entire chapter's worth of exploration. Naturally, we will encounter xarray operations as we continue to explore the page cache, as these are used extensively, and from this we can gain insights into how the API is used, especially the advanced API.

The *xarray documentation* is, additionally, extensive.

Broadly speaking, simple operations from the basic API are `xa_load()` which loads an entry from the xarray, `xa_store()` which stores an entry into the xarray, `xa_erase()` which erases an entry from the xarray and `xa_find()` which finds an entry in the xarray.

9.3 Reading From a File

We examine an example of an unbuffered read from a file into a buffer in C in Listing 9-23.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(void)
{
    int fd = open("test.txt", O_RDONLY);
    ssize_t num_bytes = 0;
    int err = EXIT_FAILURE;
    struct stat statbuf;
    char *buf;
    size_t size;

    if (fd == -1) {
        perror("open");
        goto exit;
    }

    if (fstat(fd, &statbuf)) {
        perror("fstat");
        goto exit_close;
    }

    size = statbuf.st_size;
    if (size == 0) {
        err = EXIT_SUCCESS;
        goto exit_close;
    }

    buf = malloc(size + 1);
    if (!buf) {
        perror("malloc");
        goto exit_close;
    }
}
```

```

do {
    ssize_t ret = read(fd, &buf[num_bytes], size - num_bytes);

    if (ret == -1) {
        perror("read");
        goto exit_free;
    }

    num_bytes += ret;
} while (num_bytes < size);

buf[size] = '\0';
printf("%s", buf);

err = EXIT_SUCCESS;

exit_free:
    free(buf);
exit_close:
    close(fd);
exit:
    return err;
}

```

Listing 9-23: Example file read operation

This uses `read()` to read data into a buffer from a file descriptor (the commonly used alternative `fread()` does the same, only internally buffered and with other semantic differences).

This invokes the `read()` system call, which calls the `ksys_read()` function in turn. This obtains a struct `fd` object for the open file descriptor (containing flags and a reference to the struct `file` object describing this open file), incrementing the reference count if necessary via `fdget_pos()`, invoking `vfs_read()` to perform the read before decrementing the reference count if necessary via `fdput_pos()`.

NOTE

The struct `file` object keeps track of the current position within the file. Given multiple threads could be accessing this file at the same time, it raises the question of a possible conflict between multiple reads/writes overlapping one another.

Thankfully, since Linux 3.14, this question is answered with a lock – the `FMODE_ATOMIC_POS` mode is automatically set for regular files and directories if the file is opened by more than one thread.

In this case, the struct `file->f_pos_lock` is automatically acquired on `fdget_pos()` and releasing on `fdput_pos()`.

This is codified in [POSIX.1-2008/SUSv4 Section XSI 2.9.7](#) (“Thread Interactions with Regular File Operations”), see the ‘bugs’ section of the `read()` manual page entry for details.

This is where the flexible implementation of the VFS comes into play – `vfs_read()` invokes either the `struct file->f_op->read()` function (a file system-specified field in the `struct file_operations f_op` object) if one is provided, or if `read_iter()` is specified instead, invoking `new_sync_read()` to call this instead.

The `read_iter` file operation performs the same task, only making use of a `struct iov_iter` iterator object to perform the task. This approach abstracts I/O operations across a number of different types of I/O entities (pipes, `iovec` objects and userland buffers, among others). We will be focusing solely on the user buffer case.

We examine `new_sync_read()` in Listing 9-24.

```

379 static ssize_t new_sync_read(struct file *filp, char __user *buf, size_t len,
    loff_t *ppos)
380 {
381     struct kiocb kiocb;
382     struct iov_iter iter;
383     ssize_t ret;
384
385     init_sync_kiocb(&kiocb, filp);
386     kiocb.ki_pos = (ppos ? *ppos : 0);
387     iov_iter_ubuf(&iter, READ, buf, len);
388
389     ret = call_read_iter(filp, &kiocb, &iter);
390     BUG_ON(ret == -EIOCBQUEUED);
391     if (ppos)
392         *ppos = kiocb.ki_pos;
393     return ret;
394 }
```

Listing 9-24: fs/read_write.c: `new_sync_read()`

This starts by initialising the `struct kiocb` kernel I/O control block object via `init_sync_kiocb()`, setting its `ki_filp` field to the `struct file` object `filp`, its `ki_flags` field to `struct file->f_iocb_flags` and the `ki_ioprio` field to the current task's I/O priority (out of scope for the book) and its `ki_pos` field to the current file position. All other fields are zeroed.

The 'I/O control block' object `kiocb` acts as state threaded through the I/O operation, maintained on the stack as the operation is synchronous. The `struct file->f_iocb_flags` field having been determined via the `iocb_flags()` helper function on file open in `do_dentry_open()`, shown in Listing 9-25.

```

3384 static inline int iocb_flags(struct file *file)
3385 {
3386     int res = 0;
3387     if (file->f_flags & O_APPEND)
3388         res |= IOCB_APPEND;
3389     if (file->f_flags & O_DIRECT)
3390         res |= IOCB_DIRECT;
```

```

3391     if (file->f_flags & O_DSYNC)
3392         res |= IOCB_DSYNC;
3393     if (file->f_flags & __O_SYNC)
3394         res |= IOCB_SYNC;
3395     return res;
3396 }

```

Listing 9-25: include/linux/fs.h: iocb_flags()

See the manual page for `open` for more details on each of these flags, noting that `__O_SYNC` is not used directly by the user but rather implies that `O_SYNC` has been specified.

After `kiocb` is initialised, the `struct iov_iter` object is set up for `READ` of a user buffer via `iov_iter_ubuf()`, before calling `call_read_iter()` to perform the actual read prior to a bug condition check and resetting the `ppos` file position if relevant.

We examine `call_read_iter()` in Listing 9-26.

```

2178 static inline ssize_t call_read_iter(struct file *file, struct kiocb *kio,
2179                                     struct iov_iter *iter)
2180 {
2181     return file->f_op->read_iter(kio, iter);
2182 }

```

Listing 9-26: include/linux/fs.h: call_read_iter()

This simply invokes the `struct file->f_op->read_iter()` function (a file system-specified field in the `struct file_operations f_op` field).

Each file system specifies its own function here, though most will end up invoking internal kernel functions for the core implementation. Let's look at a concrete example, the `ext4` file system's implementation `ext4_file_read_iter()`, specified in `ext4_file_operations` (eliding out of scope direct I/O and DAX logic), shown in Listing 9-27.

```

115 static ssize_t ext4_file_read_iter(struct kiocb *iocb, struct iov_iter *to)
116 {
117     struct inode *inode = file_inode(iocb->ki_filp);
118
119     if (unlikely(ext4_forced_shutdown(EXT4_SB(inode->i_sb))))
120         return -EIO;
121
122     if (!iov_iter_count(to))
123         return 0; /* skip atime */
124     :
132     return generic_file_read_iter(iocb, to);
133 }

```

Listing 9-27: fs/ext4/file.c: ext4_file_read_iter()

Other than some ext4-specific handling and the no-op case, this simply forwards the operation to `generic_file_read_iter()`, implemented in the `mm/filemap.c` file.

When examining file system-specific logic, you'll quickly notice that all roads tend to lead back to functions in `mm/filemap.c` – which provides what are in essence the kernel's *library* functions for page cache operations.

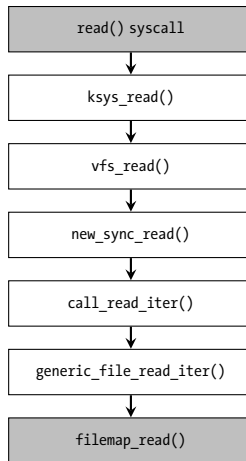
We examine `generic_file_read_iter()` (eliding out of scope direct I/O logic) in Listing 9-28.

```

2734 /**
2735  * generic_file_read_iter - generic filesystem read routine
2736  * @iocb:      kernel I/O control block
2737  * @iter:      destination for the data read
2738  *
2739  * This is the "read_iter()" routine for all filesystems
2740  * that can use the page cache directly.
2741  *
2742  * The IOCB_NOWAIT flag in iocb->ki_flags indicates that -EAGAIN shall
2743  * be returned when no data can be read without waiting for I/O requests
2744  * to complete; it doesn't prevent readahead.
2745  *
2746  * The IOCB_NOIO flag in iocb->ki_flags indicates that no new I/O
2747  * requests shall be made for the read or for readahead. When no data
2748  * can be read, -EAGAIN shall be returned. When readahead would be
2749  * triggered, a partial, possibly empty read shall be returned.
2750  *
2751  * Return:
2752  * * number of bytes copied, even for partial reads
2753  * * negative error code (or 0 if IOCB_NOIO) if nothing was read
2754  */
2755 ssize_t
2756 generic_file_read_iter(struct kiocb *iocb, struct iov_iter *iter)
2757 {
2758     size_t count = iov_iter_count(iter);
2759     ssize_t retval = 0;
2760
2761     if (!count)
2762         return 0; /* skip atime */
2763     :
2806     return filemap_read(iocb, iter, retval);
2807 }
```

Listing 9-28: `mm/filemap.c: generic_file_read_iter()`

This performs the empty iterator no-op check once again, before simply delegating the heavy lifting to `filemap_read()`. To summarise, a typical route from a `read()` call to the kernel actually reading data from the page cache is shown in Listing 9-29.

Figure 9-5: `filemap_read()` Page Cache Retrieval

We examine `filemap_read()` (eliding out of scope `io_uring`-specific and `dcache` flush logic) in Listing 9-29.

```

2613 /**
2614  * filemap_read - Read data from the page cache.
2615  * @iocb: The iocb to read.
2616  * @iter: Destination for the data.
2617  * @already_read: Number of bytes already read by the caller.
2618  *
2619  * Copies data from the page cache. If the data is not currently present,
2620  * uses the readahead and read_folio address_space operations to fetch it.
2621  *
2622  * Return: Total number of bytes copied, including those already read by
2623  * the caller. If an error happens before any bytes are copied, returns
2624  * a negative error number.
2625  */
2626 ssize_t filemap_read(struct kiocb *iocb, struct iov_iter *iter,
2627                     ssize_t already_read)
2628 {
2629     struct file *filp = iocb->ki_filp;
2630     struct file_ra_state *ra = &filp->f_ra;
2631     struct address_space *mapping = filp->f_mapping;
2632     struct inode *inode = mapping->host;
2633     struct folio_batch fbatch;
2634     int i, error = 0;
2635     :
2636     loff_t isize, end_offset;
2637
2638     if (unlikely(iocb->ki_pos >= inode->i_sb->s_maxbytes))
2639         return 0;
2640     if (unlikely(!iov_iter_count(iter)))

```

```

2641         return 0;
2642
2643         iov_iter_truncate(iter, inode->i_sb->s_maxbytes);
2644         folio_batch_init(&fbatch);

```

Listing 9-29: mm/filemap.c: filemap_read() Initialisation

We start by performing a couple of sanity checks—the `struct kiocb->ki_pos` field indicates the current offset within the file, we check to ensure this does not exceed the `struct inode->i_sb->s_maxbytes` (i.e. `struct super_block->s_maxbytes`) value which indicates the maximum file size permitted on the mounted file system, and again check to ensure the iterator isn't empty.

We then limit the size of the target buffer being written to this maximum file size via `iov_iter_truncate()` (this is often a very large value, for instance my ext4 system has a limit of 16 TiB).

We process folios in the page cache in `struct folio_batch` batches, using the `fbatch` variable, which we initialise via `folio_batch_init()`.

Next, we begin a loop where we synchronously work to retrieve a batch of folios of `PAGEVEC_SIZE` at a time (the standard folio batch size, see Section 11.7 for more on folio batches in general), shown in Listing 9-30.

```

2646         do {
2647             cond_resched();
2648             :
2657             if (unlikely(iocb->ki_pos >= i_size_read(inode)))
2658                 break;
2659
2660             error = filemap_get_pages(iocb, iter, &fbatch);
2661             if (error < 0)
2662                 break;

```

Listing 9-30: mm/filemap.c: filemap_read() page retrieval (Start of Loop)

We start by conditionally permitting rescheduling of the process on each loop, relevant only for kernels which lack full preemption via `cond_resched()`.

If the position within the `struct kiocb` object specified by the `ki_pos` field is equal to or exceeds the inode-indicated file size as determined by `i_size_read()` (for 64-bit systems this is precisely equal to `struct inode->i_size`), then we are done and exit the loop.

Next, we retrieve the folios from the page cache via `filemap_get_pages()` which passes the `struct kiocb` and `struct iov_iter` pointers to indicate what page cache folios to place in the `struct folio_batch fbatch` object.

We examine `filemap_get_pages()` in detail in section 9.5 below. Invocation of this function is core to `filemap_read()` as it is what actually performs the page cache lookup.

Importantly, the reference count of each folio thus returned will be incremented, pinning them in memory for the duration of the operation.

After folios have been retrieved, we perform some post-processing before we copy the data to the target buffer before looping around again to retrieve the next batch of data, shown in Listing 9-31.

```

2664      /*
2665      * i_size must be checked after we know the pages are Uptodate
2666      *
2667      * Checking i_size after the check allows us to calculate
2668      * the correct value for "nr", which means the zero-filled
2669      * part of the page is not copied back to userspace (unless
2670      * another truncate extends the file - this is desired though)
2671      */
2672      isize = i_size_read(inode);
2673      if (unlikely(iocb->ki_pos >= isize))
2674          goto put_folios;
2675      end_offset = min_t(loff_t, isize, iocb->ki_pos + iter->count);
2676      :
2683      /*
2684      * When a read accesses the same folio several times, only
2685      * mark it as accessed the first time.
2686      */
2687      if (!pos_same_folio(iocb->ki_pos, ra->prev_pos - 1,
2688                          fbatch.folios[0]))
2689          folio_mark_accessed(fbatch.folios[0]);

```

Listing 9-31: mm/filemap.c: `filemap_read()` Post-Processing

We check to see whether the newly updated file position stored in struct `kiocb->ki_pos` is equal to or exceeds the *current* file size (possibly extended by another thread), only after the folios have been retrieved, to ensure that what we copy is actually within the file that we are reading from (as per the comment). If not, we jump to the `put_folios` label, and immediately exit the loop.

We determine the terminating offset of the operation as the minimum of the size of the file as specified by the inode (`isize`) and the sum of the offset into the file (struct `kiocb->ki_pos`) and the *remaining* bytes to process (struct `iov_iter->count`), which indicates the exclusive upper bound of this read.

Next, we determine whether to mark the first folio as accessed via `folio_mark_accessed()` (see Section 11.2 for more details on this) – we mark the folio as such in order to indicate to reclaim that this folio has recently been used and thus should be a lower priority for reclaim.

Why do we sometimes *not* do this to the first folio in the batch? This is because a sequential read may start and stop within the same folio – remember that the offset into the file is specified at *byte* granularity, however folios exist at minimum at base page granularity.

As a result, we might in theory read from bytes 0 to 100, then bytes 101 to 200, then bytes 201 to 300 and so on in the same folio, and as such the folio is really only in demand once, so repeatedly marking the folio accessed would be inaccurate.

As we read through the file, we keep track of the exclusive bound of the last requested block of data in the struct `file_ra_state` object in its `prev_pos` field. This object (stored in variable `ra`) stores *readahead* state of the file, and is sourced from the struct `file->f_ra` field. We go into detail on readahead in section 9.7, but in this instance it suffices to note that this is where the `prev_pos` field is stored.

We determine whether the current offset into the file (struct `kiocb->ki_pos`) and the known last position point to the same first folio via `pos_same_folio()`, shown in Listing 9-32.

```

2606 static inline bool pos_same_folio(loff_t pos1, loff_t pos2, struct folio *
      folio)
2607 {
2608     unsigned int shift = folio_shift(folio);
2609
2610     return (pos1 >> shift == pos2 >> shift);
2611 }

```

Listing 9-32: mm/filemap.c: pos_same_folio()

This simply determines the shift (i.e. the number of bits) comprising the folio size via `folio_shift()` and offsets each byte position accordingly before comparing them.

Once this check has been completed, we can go ahead and start copying page cache folios into the iterator, shown in Listing 9-33.

```

2691         for (i = 0; i < folio_batch_count(&fbatch); i++) {
2692             struct folio *folio = fbatch.folios[i];
2693             size_t fsize = folio_size(folio);
2694             size_t offset = iocb->ki_pos & (fsize - 1);
2695             size_t bytes = min_t(loff_t, end_offset - iocb->ki_pos
2696
2697             ,
2698                                     fsize - offset);
2699             size_t copied;
2700
2701             if (end_offset < folio_pos(folio))
2702                 break;
2703             if (i > 0)
2704                 folio_mark_accessed(folio);
2705             :
2706
2707             copied = copy_folio_to_iter(folio, offset, bytes, iter
2708
2709             );
2710
2711             already_read += copied;
2712             iocb->ki_pos += copied;

```

```

2715         ra->prev_pos = iocb->ki_pos;
2716
2717         if (copied < bytes) {
2718             error = -EFAULT;
2719             break;
2720         }
2721     }

```

Listing 9-33: mm/filemap.c: filemap_read() Folio Copying

We iterate through each of the page cache folios obtained via `filemap_get_pages()` (counting these using `folio_batch_count()`), processing each individually.

We determine the size of the folio, `fsize` using `folio_size()`, the offset within the folio, `offset` (by masking against the folio size), the number of bytes to be read from the folio, `bytes`, equal to either the folio size minus offset or, if the read range stops short of the end of the folio, the number of bytes until the final offset.

If, for some reason, the starting offset of this folio within the file (determined by `folio_pos()`) exceeds that of the exclusive end bound of the range being read, `end_offset` we abort the operation.

For all but the first folio, which has special treatment as described above, the folio is marked accessed via `folio_mark_accessed()`. This is important for the purposes of reclaim – by reading from the file its associated page cache folios have been accessed and thus should be deprioritised for reclaim. See Section 11.2 for more details on how this functions.

The actual copy operation is performed via `copy_folio_to_iter()`. If this fails to copy all of the bytes, which implies that an unrecoverable page fault occurred (at least part of the buffer is unmapped or read-only for instance), we store the `EFAULT` error code in `error` and we exit out of the copying loop.

Interestingly, on return, if some bytes have been copied, we ignore the error message and simply return the count of bytes that were copied, meaning that if a user for instance `mmap()`'s a region but tries to copy more bytes into that region then mapped (and no mapping immediately follows it), no error will be returned.

Prior to checking for this error state, we update the struct `kiocb->ki_pos` field to update the position, and importantly update the struct `file_ra_state->prev_pos` field to indicate how much data has been successfully read.

```

2722 put_folios:
2723     for (i = 0; i < folio_batch_count(&fbatch); i++)
2724         folio_put(fbatch.folios[i]);
2725     folio_batch_init(&fbatch);
2726     } while (iov_iter_count(iter) && iocb->ki_pos < isize && !error);
2727
2728     file_accessed(filp);
2729
2730     return already_read ? already_read : error;

```

2731 }

Listing 9-34: mm/filemap.c: filemap_read() folio cleanup (End of Loop)

Finally within the loop, we decrement the reference count for each returned folio which was incremented by `filemap_get_pages()`, and reset the folio batch `fbatch` via `folio_batch_init()` ready for the next iteration.

The do/while loop is terminated either by the iteration being complete (i.e. `iov_iter_count()` returning zero), the struct `kiocb->ki_pos` offset counter being equal to or exceeding the file size, or an error having occurred.

Prior to returning, the function invokes `file_accessed()` which updates the file's *access* time (i.e. `atime`), via the `touch_atime()` function.

NOTE

Typically modern file systems are mounted with the `relatime` option, to prevent the file metadata from being updated on each access which would create egregiously unnecessary I/O.

In this mode, the access time is updated if either the file or its metadata was updated since last update, or if the last access was over a day ago (`touch_atime()` calls `atime_needs_update()` which calls `relatime_need_update()` in turn to perform this check).

The key bit of code we are missing here is the implementation of `filemap_get_pages()`, however we will defer discussion of this to section 9.5 below, and instead look at the other most common means of accessing page cache entries, namely file-backed page faults.

9.4 File-Backed Read Faults

If a file is mapped into memory using `mmap()`, then looking up its contents from the page cache occurs at the point of page fault. This is another means by which user space applications are able to access file data (in this instance, on demand at page fault time), so we examine it in detail here.

We explored how the page fault logic functions in this instance in section 6.2 (see Figure 6-3 especially and a specific discussion of file-backed faulting in section 6.8), noting that the heavy lifting is performed in `__do_fault()` (see Listing 6-32).

We won't duplicate this listing, instead as this is an operation that occurs at a VMA level, we observe that the fault handling is delegated to the file system-defined struct `vm_area_struct->vm_ops->fault()` function (i.e. struct `vm_operations_struct->fault()`).

Again, using `ext4` as our example file system, we see that the `ext4_file_vm_ops->fault` function forwards the operation to `filemap_fault()`. This is the key 'library function' (in essence) provided by the kernel for file-backed faults, and as a result many file systems simply defer the operation to it entirely.

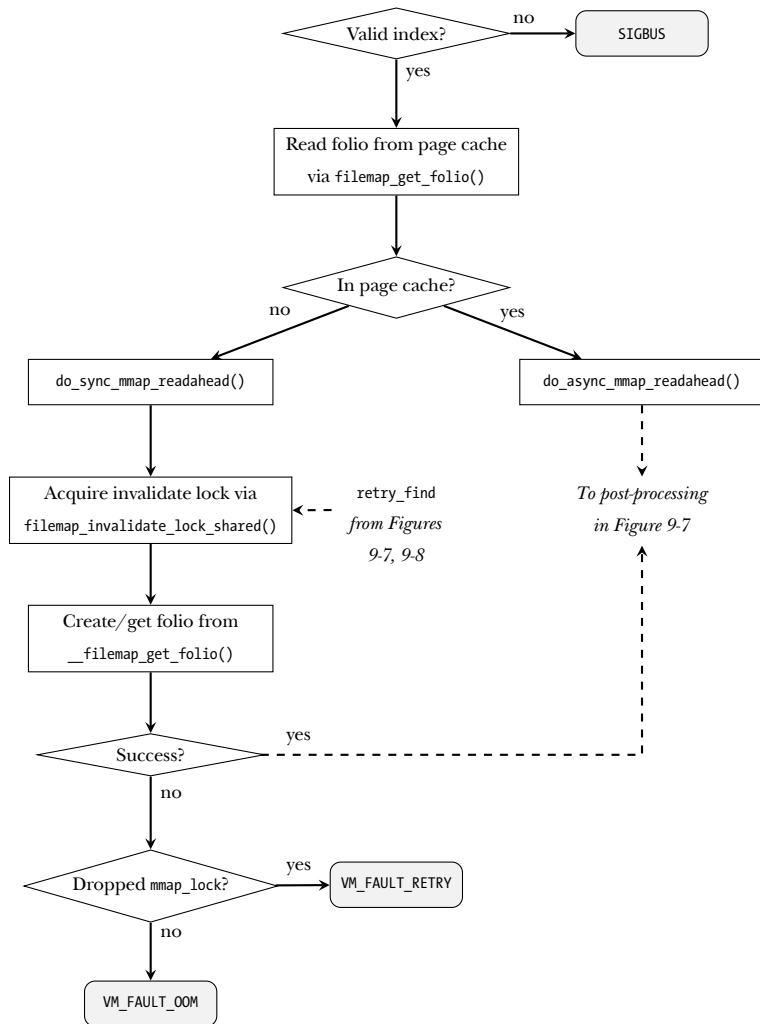
A key difference between this operation and the one performed on read discussed in section 9.3 above is that the fault occurs on a *single* folio basis rather than a batch. In some respects this simplifies things, however we also

need to take into account the properties of the VMA into which the folio is being faulted.

Nuances of the handling of page faults (folio at a time, must sit in the page fault logic and interact with the page fault API such as interacting with the `struct vm_fault` state object, VMA granularity rather than file granularity among other things) means that logic that is shared between a read and a page fault must necessarily get somewhat duplicated.

As a result, we will tread similar ground here and in the below discussion of accessing page cache entries (section 9.5) which perform similar steps as the `read()` approach (in section 9.3 above), exploring however in detail the nuanced details in which each differ.

As this function is rather complicated, let's step back and examine the function diagrammatically in Figure 9-6, before diving into the function in detail.

Figure 9-6: `filemap_fault()` Page Cache Retrieval

We start by checking to ensure that the index into the file is valid, if not we simply send the process a SIGBUS signal.

NOTE

The difference between a SIGBUS and a SIGSEGV signal isn't always entirely clear. The lines between the two are rather blurry (both imply an invalid memory access), however broadly speaking:

- A SIGSEGV arises when an invalid access to a valid address occurs, e.g. accessing unmapped memory, or memory to which you do not have permission to access, (e.g. writing to read-only mappings).
- A SIGBUS arises when an invalid address is accessed (whether the access is valid or not), e.g. accessing unaligned addresses on an architecture which doesn't support unaligned access, or, pertinently, access to portions of a file mapping that are not backed by the file.

This can be trivially triggered by `mmap()`’ing a file at a size greater than the file size and accessing a page past the end of the file (a page containing a part of the file will return zeroes for the portion which are not mapped).

The out of scope accesses can be observed in Figures 9-6 and 9-7 (these arise as the struct `folio->index` being deemed invalid).

In addition, an error occurring on attempted read of the file (for instance, access to a file on a network filesystem which has encountered connectivity issues) will cause a `SIGBUS` to arise, as shown in Figure 9-8.

If the index is valid, we invoke `filemap_get_folio()` to see whether the folio already exists in the page cache. If so, this is a *minor* fault (more on these later) and we simply kick off any required asynchronous readahead via `do_async_mmap_readahead()` (see Section 9.7 below for a detailed description of readahead), and move on to the post-processing phase.

Otherwise, this is a *major* fault, so we must attempt *blocking* readahead via `do_sync_mmap_readahead()`, before seeing whether we can read back the folio we just read (or if readahead failed allocate a folio to populate with disk data) via `__filemap_get_folio()`.

Note that we acquire the *invalidate lock* in the meantime via `filemap_invalidate_lock_shared()` to protect against racing folio *truncation* and *invalidation*, more on this in the detailed examination of the code.

If we fail at this point we are out of memory, so depending on whether we dropped the struct `mm_struct->mmap_lock`, we either resolve the fault with `VM_FAULT_RETRY` if we did (in order to simply reacquire it and try again, if haven’t retried once already) or with `VM_FAULT_OOM` to indicate the out of memory condition.

If we succeed in this portion of the function, we move on to the post-processing phase, as shown in Figure 9-7.

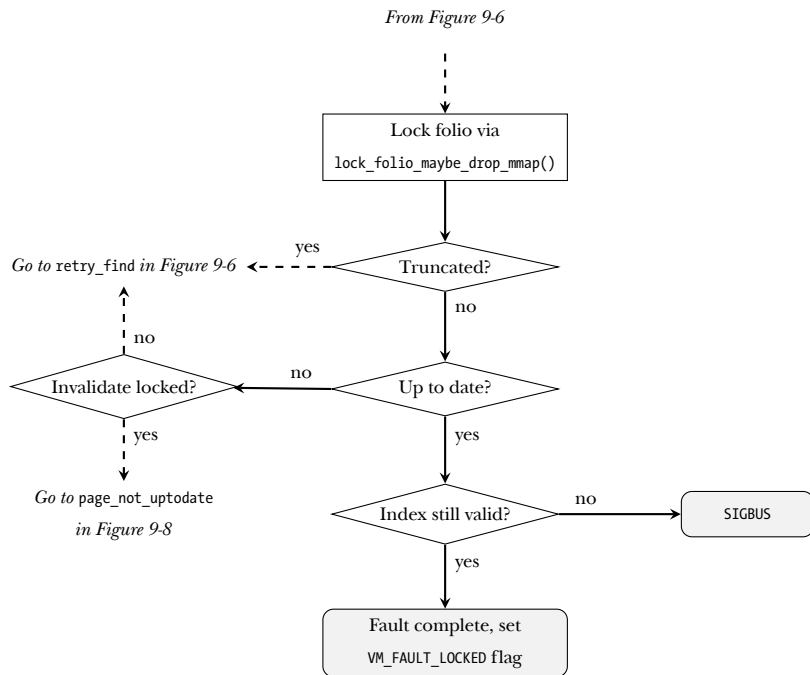


Figure 9-7: *filemap_fault()* Post-Processing

Here we must lock the folio to ensure it is stable and not undergoing update or truncation. Since these operations can take a while to complete, we will drop the struct `mm_struct->mmap_lock` lock if it is contended – this process is performed by `lock_folio_maybe_drop_mmap()`.

After we have acquired the lock, we must now to see if anything changed underneath us in the mean time – if it has, we retry folio lookup via the `retry_find` label (and acquiring the invalidation lock if not already held).

There is one specific edge case – where the folio is not ‘up to date’, i.e. has not been fully retrieved from disk. This can either be due to the read-ahead failing to retrieve data, or an error having occurred. We resolve this at `page_not_uptodate`, described in Figure 9-8 below.

If the index of this folio in the file is not valid (i.e. truncation has occurred and we have taken it into account, but simply discover our index falls off the end of the file), we send a `SIGBUS` signal to the process.

Finally if all is good, we complete the operation, indicating the folio is locked via `VM_FAULT_LOCKED`.

We examine the case in which the folio is not up to date, from label `page_not_uptodate` within `filemap_fault()` in Figure 9-8.

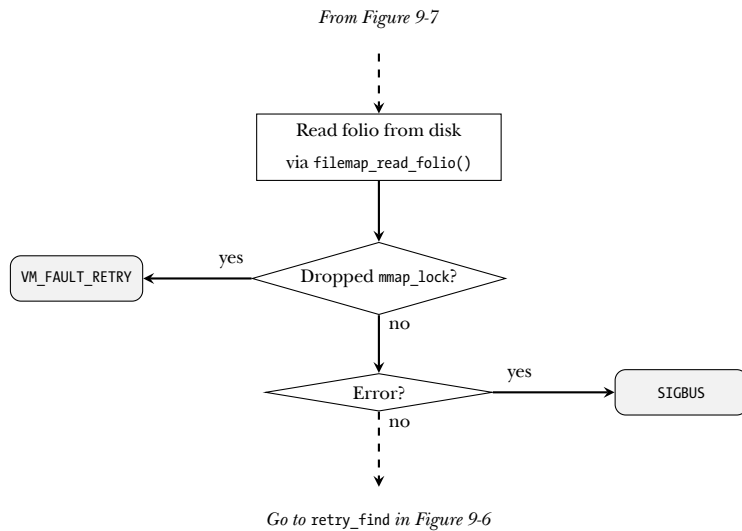


Figure 9-8: *filemap_fault()* Page Not Up To Date Handling

This is relatively straightforward – we try to *explicitly* read the folio via `filemap_read_folio()`. Regardless of whether an error occurs, if we dropped the struct `mm_struct->mmap_lock` we indicate that the fault should be tried via `VM_FAULT_RETRY` (dropping this lock might mean this mapping is simply not even valid now).

If we did not drop the lock, we check whether an error arose – if so, the `SIGBUS` signal is sent to the process, otherwise if all is good and the folio was retrieved, we jump to the `retry_find` label described in Figure 9-6.

Now we have examining the function in brief, let's examine `filemap_fault()` in detail line-by-line, the initial part of which is shown in Listing 9-35.

```

3061 /**
3062  * filemap_fault - read in file data for page fault handling
3063  * @vmf:          struct vm_fault containing details of the fault
3064  *
3065  * filemap_fault() is invoked via the vma operations vector for a
3066  * mapped memory region to read in file data during a page fault.
3067  *
3068  * The goto's are kind of ugly, but this streamlines the normal case of having
3069  * it in the page cache, and handles the special cases reasonably without
3070  * having a lot of duplicated code.
3071  *
3072  * vma->vm_mm->mmap_lock must be held on entry.
3073  *
3074  * If our return value has VM_FAULT_RETRY set, it's because the mmap_lock
3075  * may be dropped before doing I/O or by lock_folio_maybe_drop_mmap().

```



```

3076  *
3077  * If our return value does not have VM_FAULT_RETRY set, the mmap_lock
3078  * has not been released.
3079  *
3080  * We never return with VM_FAULT_RETRY and a bit from VM_FAULT_ERROR set.
3081  *
3082  * Return: bitwise-OR of %VM_FAULT_ codes.
3083  */
3084  vm_fault_t filemap_fault(struct vm_fault *vmf)
3085  {
3086      int error;
3087      struct file *file = vmf->vma->vm_file;
3088      struct file *fpin = NULL;
3089      struct address_space *mapping = file->f_mapping;
3090      struct inode *inode = mapping->host;
3091      pgoff_t max_idx, index = vmf->pgoff;
3092      struct folio *folio;
3093      vm_fault_t ret = 0;
3094      bool mapping_locked = false;
3095
3096      max_idx = DIV_ROUND_UP(i_size_read(inode), PAGE_SIZE);
3097      if (unlikely(index >= max_idx))
3098          return VM_FAULT_SIGBUS;

```

Listing 9-35: mm/filemap.c: `filemap_fault()` Initialisation

We begin by initialising variables, recalling from section 6.2 that an `struct vm_fault` object threads fault state through the fault process, and we return a `vm_fault_t` value indicating the outcome of the fault handling.

We determine the maximum *exclusive* index that can reference within the file-backed mapping, `max_idx` by rounding up the size of the file (as indicated by its `struct inode`) obtained from `i_size_read()` to the system `PAGE_SIZE`.

If the page offset into the file (`struct vm_fault->pgoff`) equals or exceeds this the fault extends past the end of the file and we must return a `SIGBUS` error, indicated by returning `VM_FAULT_SIGBUS`.

Next, we attempt to retrieve the folio from the page cache, as shown in Listing 9-36.

```

3100      /*
3101      * Do we have something in the page cache already?
3102      */
3103      folio = filemap_get_folio(mapping, index);
3104      if (likely(folio)) {
3105          /*
3106          * We found the page, so try async readahead before waiting
3107          * the lock.
3108          */
3109          if (!(vmf->flags & FAULT_FLAG_TRIED))

```

```

3110         fpin = do_async_mmap_readahead(vmf, folio);
3111         if (unlikely(!folio_test_uptodate(folio))) {
3112             filemap_invalidate_lock_shared(mapping);
3113             mapping_locked = true;
3114         }
3115     } else {
3116         /* No page in the page cache at all */
3117         count_vm_event(PGMAJFAULT);
3118         count_memcg_event_mm(vmf->vma->vm_mm, PGMAJFAULT);
3119         ret = VM_FAULT_MAJOR;
3120         fpin = do_sync_mmap_readahead(vmf);
3121     retry_find:
3122         /*
3123          * See comment in filemap_create_folio() why we need
3124          * invalidate_lock
3125          */
3126         if (!mapping_locked) {
3127             filemap_invalidate_lock_shared(mapping);
3128             mapping_locked = true;
3129         }
3130         folio = __filemap_get_folio(mapping, index,
3131                                     FGP_CREAT|FGP_FOR_MMAP,
3132                                     vmf->gfp_mask);
3133         if (!folio) {
3134             if (fpin)
3135                 goto out_retry;
3136             filemap_invalidate_unlock_shared(mapping);
3137             return VM_FAULT_OOM;
3138         }
3139     }

```

Listing 9-36: mm/filemap.c: filemap_fault() Page Cache Retrieval

We try to retrieve the folio from the page cache using `filemap_get_folio()` (described in section 9.5 below), and determine what to do based on whether we successfully retrieve a folio from the page cache or not.

A ‘major’ fault occurs when a process accesses memory mapped to data which is not currently in memory *at all*, i.e. file-backed memory that does not exist in the page cache currently, or anonymous memory which needs to be swapped back in. In both cases, the major fault entails I/O.

A ‘minor’ fault on the other hand is a page fault which occurs when a process accesses memory that exists in RAM but is simply not yet mapped yet (i.e. new anonymous or shmem page private folios, or existing page cache folios).

We can see this in action here, as should a folio not be present in the page cache, we denote this by incrementing statistics for the PGMAJFAULT event type (minor faults can be determined by taking the count of PGFAULT events and subtracting it by the count of PGMAJFAULT events).

NOTE

Total page faults across the system can be determined via `/proc/vmstat` in the `pgfault` and `pgmajfault` entries which output the statistics for `PGFAULT` and `PGMAJFAULT` respectively.

Note that these memory management level major fault counters are not the only ones measuring major and minor faults, for instance each `struct task_struct` maintains `min_flt` and `maj_flt` statistics, accessible via `/proc/$pid/stat` (or more conveniently the `ps` command), `getrusage()`, among others. In addition, statistics are accounted for the `perf` utility.

These values won't perfectly align with the system-wide statistics as these are accounted by `mm_account_fault()`, for instance a minor fault that had to be retried as indicated by the `FAULT_FLAG_TRIED` flag is considered a major fault.

See Section 6.2 for more details on page fault handling as a whole.

The simpler case is when the folio is present in the page cache – we do not have to do anything further in order to back the memory. We do however have to taken into account a couple of considerations.

Firstly, if this fault hasn't just been retried (as indicated by the `FAULT_FLAG_TRIED` flag), we try to perform asynchronous readahead via `do_async_mmap_readahead()`.

We test this flag because `FAULT_FLAG_TRIED` is set when a previous attempt at servicing this page fault relinquished the `struct mm_struct->mmap_lock`. If it had done, it would already have performed this step so we'd be duplicating work.

The asynchronous readahead performed in `do_async_mmap_readahead()` will try to read data we expect the user to access soon into the page cache, asynchronously, naturally. We go into detail on how this functions in section 9.7.

Next we test to see if the folio is not up to date, i.e. does not have the `PG_uptodate` flag set. This flag is set when data from disk has been read and the folio populated with it. However, since this folio already exists in the page cache, the only reason this could be the case is if an *invalidation* (e.g. the cache page no longer represents the file contents and must be updated) is occurring, as we have not yet locked the folio.

In this case, we acquire the `struct address_space->invalidate_lock` via `filemap_invalidate_lock_shared()` to synchronise against other invalidations and operations that might race with an invalidation. We set `mapping_locked` to true to know to release the lock later.

In the instance where no folio could be found in the page cache, we update statistics to account for this fact (and set the return value to `VM_FAULT_MAJOR` accordingly), before simply going ahead and performing *synchronous* readahead in order to populate the page cache, via `do_sync_mmap_readahead()` (see Section 9.7 for details on what happens here).

NOTE

Note that this readahead still may not actually retrieve the folio from the disk. For instance, if a user specifies that a region of memory is random-access memory via `madvise()` with the `MADV_RANDOM` flag set, this will cause readahead to abort.

We will see below how we retrieve data in this instance.

This function, as well as the asynchronous readahead discussed above returns a struct file object, `fpin`. What does this value represent?

Performing I/O is a lengthy operation, so we prefer not to hold the struct `mm_struct->mmap_lock` semaphore throughout as this can lead to contention when another thread needs to acquire a *write* lock (this lock is a read-write semaphore, so reads happen simultaneously, but writes are exclusive).

If we do so, we must maintain the file's reference count in order to ensure that it does not disappear from beneath us while the `mmap_lock` is dropped. Therefore, each function which might drop this lock both accepts and returns an `fpin` parameter.

The input `fpin` is checked to see if the file was already pinned (implying lock released), if not then `get_file()` is called, which increments the struct `file->f_count` reference count before returning the file object, which is assigned to the local `fpin` variable.

Once the synchronous readahead is complete, we acquire the struct `address_space->invalidate_lock` via `filemap_invalidate_lock_shared()` if we had not already done so, and again mark `mapping_locked` to indicate we have.

We do so, because we are about to invoke `__filemap_get_folio()` with the `FGP_CREAT` flag set, indicating that we will create a new folio to be populated from the disk, and must ensure that no race exists between the creation and invalidation of the folio.

NOTE

We will get into this logic in more detail in section 9.5 below, but note that if `__filemap_get_folio()` allocates a new folio via `filemap_alloc_folio()` and adds it to the page cache via `filemap_add_folio()`, this will cause there to be a single reference count on the folio.

Therefore, if subsequent logic determines that this folio cannot satisfy the fault (e.g. if file truncation raced with the fault), it will invoke `folio_put()` as part of cleanup which result in the folio being freed.

We will have, most likely, retrieved the folio from disk in the synchronous readahead above. However in cases where either an error occurred or readahead will otherwise not proceed, we may not have.

This can be due to an error or, for instance, the memory having been marked random-access via `madvise()` with the `MADV_RANDOM` flag set (leading to the struct `vm_area_struct` (VMA) flag possessing the `VM_RAND_READ` flag). In this case readahead is simply never performed.

Regardless, we must be able to retrieve the folio, whether it is located from the readahead operation or if it is a freshly allocated one which is not yet marked up to date (we will consider how to actually retrieve this data in the discussion around listing 9-37 below).

If we cannot retrieve a folio as needed in `__filemap_get_folio()`, even though `FGP_CREAT` dictates that one should be created if we cannot find an existing folio, then an out of memory condition has occurred.

If we have taken an `fpin`, indicating the lock is dropped, we can have the fault be retried, by which point more memory might be available by includ-

ing the `VM_FAULT_RETRY` flag in the return value (this is handled at the `out_retry` label, described below).

Otherwise we simply return `VM_FAULT_OOM` indicating an out of memory condition has occurred and the fault cannot proceed.

After each of these cases are considered, we have post-processing to perform, shown in Listing 9-37.

```

3141         if (!lock_folio_maybe_drop_mmap(vmf, folio, &fpin))
3142             goto out_retry;
3143
3144         /* Did it get truncated? */
3145         if (unlikely(folio->mapping != mapping)) {
3146             folio_unlock(folio);
3147             folio_put(folio);
3148             goto retry_find;
3149         }
3150         VM_BUG_ON_FOLIO(!folio_contains(folio, index), folio);
3151
3152         /*
3153          * We have a locked page in the page cache, now we need to check
3154          * that it's up-to-date. If not, it is going to be due to an error.
3155          */
3156         if (unlikely(!folio_test_uptodate(folio))) {
3157             /*
3158              * The page was in cache and uptodate and now it is not.
3159              * Strange but possible since we didn't hold the page lock all
3160              * the time. Let's drop everything get the invalidate lock and
3161              * try again.
3162              */
3163             if (!mapping_locked) {
3164                 folio_unlock(folio);
3165                 folio_put(folio);
3166                 goto retry_find;
3167             }
3168             goto page_not_uptodate;
3169         }
3170
3171         /*
3172          * We've made it this far and we had to drop our mmap_lock, now is the
3173          * time to return to the upper layer and have it re-find the vma and
3174          * redo the fault.
3175          */
3176         if (fpin) {
3177             folio_unlock(folio);
3178             goto out_retry;
3179         }
3180         if (mapping_locked)
3181             filemap_invalidate_unlock_shared(mapping);

```

```

3182
3183     /*
3184     * Found the page and have a reference on it.
3185     * We must recheck i_size under page lock.
3186     */
3187     max_idx = DIV_ROUND_UP(i_size_read(inode), PAGE_SIZE);
3188     if (unlikely(index >= max_idx)) {
3189         folio_unlock(folio);
3190         folio_put(folio);
3191         return VM_FAULT_SIGBUS;
3192     }
3193
3194     vmf->page = folio_file_page(folio, index);
3195     return ret | VM_FAULT_LOCKED;

```

Listing 9-37: mm/filemap.c: filemap_fault() Post-Processing

Firstly, we must lock the folio before we can return it to the generic fault handler logic. we do this via `lock_folio_maybe_drop_mmap()`. If the folio is already locked due to I/O currently occurring on the underlying folio, this function attempts to release the struct `mm_struct->mmap_lock` via `maybe_unlock_mmap_for_io()`. Either way it tries to acquire the lock (waiting if necessary).

If it succeeds we continue, otherwise we attempt to retry the fault, returning `VM_FAULT_RETRY` (see listing 9-39 below).

We examine this function in detail as well as folio locking as a hole in section 9.11 below.

Now we have the folio locked, we can check for certain to determine whether the folio is in a good, stable state. We start by determining whether it was truncated, i.e. the file size having reduced, rendering this folio no longer part of this file mapping.

By convention, this will be indicated by the struct `folio->mapping` field being set to `NULL` (for instance in `page_cache_delete()`).

Any change to this field has to be performed under the folio lock, so by now possessing it we can be certain that it will remain stable.

Since it may also in theory have had its mapping reassigned since being truncated, we simply check to see if the mapping remains the same as before. If not – we unlock and reduce the folio’s reference count and try finding it/creating it again at `retry_find` in listing 9-36 above, this time certainly establishing the invalidation lock.

We perform a `CONFIG_DEBUG_VM` sanity check to ensure the folio does indeed sit at the expected index via `folio_contains()`,

Next, we determine if this folio is actually up to date with the disk (i.e. possessing the folio flag `PG_uptodate`). Despite the rather misleading comment suggesting only an ‘error’ might have caused this, other conditions can cause, as determined in `do_sync_mmap_readahead()` – for instance the user having specified the region as being random access using `madvise()` with the `MADV_RANDOM` flag, or repeated cache misses.

If the folio is indeed not up to date, we first check to see whether we held the invalidation lock (`struct address_space->invalidate_lock`). If not, then we retry the operation with this held to account for races by unlocking and reducing the reference count on the folio and jumping to `retry_find` (see listing 9-37 above).

Otherwise, if we have the lock held, we force the issue and proactively attempt to read the folio from disk at label `page_not_uptodate`, discussed below listing 9-38. This will retry the operation when done.

At this stage we have the correct folio. If we dropped the `struct mm_struct->mmap_lock` to reach this point, it is no longer for certain that the virtual mapping still exists, so we have to inform the generic fault handler that we must retry the fault by returning `VM_FAULT_RETRY`. We do this by unlocking the folio and jumping to `out_retry`, described in listing 9-39 described below.

Now we are sure the folio is as we expect, we drop the invalidate lock via `filemap_invalidate_unlock_shared()`.

We must perform one final check before we can resolve the page fault – check that the index of the faulting folio still resides within the size of the file, in case it had been truncated prior to the folio being locked.

If this is the case, we unlock the folio, decrement its reference count and return `VM_FAULT_SIGBUS`, indicating that a SIGBUS signal should be sent to the process.

Finally, we have the correct folio and can resolve the page fault. We look up the sub-page within the folio relating to its index via `folio_file_page()` and set this as the `struct page` to resolve the fault, returning with the `VM_FAULT_LOCKED` flag set to indicate the folio is locked.

Next we examine the edge case handling within the code, shown in Listing 9-38.

```

3197 page_not_uptodate:
3198     /*
3199         * Umm, take care of errors if the page isn't up-to-date.
3200         * Try to re-read it _once_. We do this synchronously,
3201         * because there really aren't any performance issues here
3202         * and we need to check for errors.
3203     */
3204     fpin = maybe_unlock_mmap_for_io(vmf, fpin);
3205     error = filemap_read_folio(file, mapping->a_ops->read_folio, folio);
3206     if (fpin)
3207         goto out_retry;
3208     folio_put(folio);
3209
3210     if (!error || error == AOP_TRUNCATED_PAGE)
3211         goto retry_find;
3212     filemap_invalidate_unlock_shared(mapping);
3213
3214     return VM_FAULT_SIGBUS;

```

Listing 9-38: mm/filemap.c: filemap_fault() Page Not Up to Date Case

This case is only reached if, in listing 9-37 described above, the folio is not up to date but the invalidate lock, `struct address_space->invalidate_lock`, is held. This indicates that no readahead was performed (for instance the memory is marked random-access or an error arose), and therefore the folio must be read directly.

Before attempting I/O again, we try to unlock `struct mm_struct->mmap_lock` in order to prevent the virtual address space becoming functionally read-only throughout the operation via `maybe_unlock_mmap_for_io()` (see listing 6-41).

After this is done, we directly attempt to read the data into the folio via `filemap_read_folio()` (see Listing 9-56), which wraps the file system-specified `struct address_space->a_ops->read_folio()` function (part of the `struct address_space_operations` set of callbacks), clearing errors and checking that the folio is up to date after the retry. It does this synchronously, waiting for the folio to become unlocked by the read operation via `folio_wait_locked_killable()`.

See Section 9.6 for a detailed exploration of this function and reading folios from disk into the page cache in general. See Section 9.11 for details of how the waiting logic functions.

In the case that we dropped the `struct mm_struct->mmap_lock` as indicated by `fpin` being set, we jump to the `out_retry` label (discussed after listing 9-39 below) and indicate to the generic fault handler that we wish to retry the fault, having now read the required folio into the page cache or observed an error we can retry avoiding (again, we must do this in either case as the mapping itself may no longer exist).

If we maintained the lock, we unpin the folio via `folio_put()`, then check to determine whether an error arose – if none did, or it was an `AOP_TRUNCATED_PAGE` error indicating the folio may have simply been truncated, we jump to the `retry_find` label shown in listing 9-36 discussed above.

Finally we are at a point where an error *has* occurred that we cannot recover from, so we drop the invalidate lock and indicate the calling process should receive the `SIGBUS` signal by returning `VM_FAULT_SIGBUS` since this is an invalid access.

The final edge case is shown in Listing 9-39.

```

3216 out_retry:
3217     /*
3218      * We dropped the mmap_lock, we need to return to the fault handler to
3219      * re-find the vma and come back and find our hopefully still
3220      * populated
3221      * page.
3222      */
3223     if (folio)
3224         folio_put(folio);
3225     if (mapping_locked)

```



```

3225         filemap_invalidate_unlock_shared(mapping);
3226     if (fpin)
3227         fput(fpin);
3228     return ret | VM_FAULT_RETRY;
3229 }

```

Listing 9-39: mm/filemap.c: filemap_fault() Retry Case

To reach this point, we will have dropped the struct `mm_struct->mmap_lock`. As a result, the faulting memory may have changed beneath us and therefore the fault cannot continue and must be retried.

Firstly, we unwind our pinned folio by reducing its reference count via `folio_put()`, drop the invalidate lock if held via `filemap_invalidate_unlock_shared()`, and remove the pin on the struct `file` via `fput()`.

We then return with the `VM_FAULT_RETRY` flag set to indicate that the fault must be retried.

9.5 Reading Page Cache Entries

There are a number of routes by which we read folios from the page cache. Each differs in whether we attempt only to read *from* the page cache, i.e. indicating that an entry was not found if it was not already present there, or *into* the page cache where, if not already present, we read a folio from disk into the page cache and return this.

In addition we may also choose to read from the page cache, creating a page cache entry which we mark as not 'uptodate' (i.e. not yet read from disk) but not proceeding with the read just yet. Examining each case:

Reading a batch of folios from the page cache In instances where we do not want to read into the page cache but rather only from it (indicating missing entries if not present), we use functions such as `filemap_get_folios()`, `find_get_entries()` and `find_lock_entries()`. These functions simply do not return non-present entries. We examine this in Section 9.5.1 below.

Reading a batch of folios into the page cache In instances where we need to read a *batch* of folios *from* the page cache if already present or from disk *into* the page cache if not, we utilise the `filemap_get_pages()` function, used by `filemap_read()` described in Section 9.3 above, reads folios in a specified range from the page cache and if not present, reads them from disk, performing readahead as necessary. We describe this in Section 9.5.2 below.

Reading a single folio from the page cache The `filemap_get_folio()` function simply checks to see if the folio is present in the cache, retrieving it if so, returning an error if not. This wraps `__filemap_get_folio()`, passing no flags, a function which is also invoked directly elsewhere. In instances where the `FGP_CREAT` flag is passed to `__filemap_get_folio()`, a folio is allocated in the instance none exist in the page cache and this newly

allocated folio is added to the page cache. It will not be marked up to date, so still needs to be read from disk to be usable. This is examined in Section 9.5.3 below.

Reading a single folio into the page cache A number of filesystem, driver and other kernel components need to read individual folios from the page cache, ultimately invoking `do_read_cache_folio()` to do so, which we examine in Section 9.5.4.

Finally, we examine the actual mechanism of adding a folio to the page cache in Section 12.2.1.

The functions which actually perform the read from disk are examined in Section 9.6, and readahead is described in Section 9.7.

9.5.1 Reading a Batch of Folios From the Page Cache

Often we simply need to retrieve folios from the page cache for *present* entries, i.e. entries that already exist within the page cache having already been read from disk. This is what `find_get_entries()` accomplishes, shown in Listing 9-40.

```

2036 /**
2037  * find_get_entries - gang pagecache lookup
2038  * @mapping:      The address_space to search
2039  * @start:        The starting page cache index
2040  * @end:          The final page index (inclusive).
2041  * @fbatch:       Where the resulting entries are placed.
2042  * @indices:      The cache indices corresponding to the entries in @entries
2043  *
2044  * find_get_entries() will search for and return a batch of entries in
2045  * the mapping. The entries are placed in @fbatch. find_get_entries()
2046  * takes a reference on any actual folios it returns.
2047  *
2048  * The entries have ascending indexes. The indices may not be consecutive
2049  * due to not-present entries or large folios.
2050  *
2051  * Any shadow entries of evicted folios, or swap entries from
2052  * shmem/tmpfs, are included in the returned array.
2053  *
2054  * Return: The number of entries which were found.
2055  */
2056 unsigned find_get_entries(struct address_space *mapping, pgoff_t start,
2057                          pgoff_t end, struct folio_batch *fbatch, pgoff_t *indices)
2058 {
2059     XA_STATE(xas, &mapping->i_pages, start);
2060     struct folio *folio;
2061
2062     rcu_read_lock();
2063     while ((folio = find_get_entry(&xas, end, XA_PRESENT)) != NULL) {

```

```

2064         indices[fbatch->nr] = xas.xa_index;
2065         if (!folio_batch_add(fbatch, folio))
2066             break;
2067     }
2068     rcu_read_unlock();
2069
2070     return folio_batch_count(fbatch);
2071 }

```

Listing 9-40: mm/filemap.c: find_get_entries()

This function walks through the page cache entry's xarray (as located in struct `address_space->i_pages`), locating all folios in the page range start to end inclusive, i.e. all present folios containing the pages within the specified range in the page cache entry. It places these folios into the struct `folio_batch` in `fbatch` up to the maximum folio batch size (`PAGEVEC_SIZE`, i.e. 15 folios), containing up to this many folios.

Since we may end up processing non-present folios in the range, we also need a means of indicating which page offsets each of the batch entries refer to, so a pointer to an array of pointer offsets of size `PAGEVEC_SIZE` is passed in `indices` for this purpose.

Each entry is located using `find_get_entry()`, with the state of the xarray walk threaded through via the struct `xa_state` value `xas`, initialised via `XA_STATE()`.

The function also returns 'exceptional' entries, these are value rather than pointer entries which indicate that a folio has since been removed, which can be tested for via `xa_is_value()` which the caller is expected to handle. The whole operation is performed under an *RCU* lock which protects against other threads manipulating the folio in the interim.

Every folio that it does find has its reference count incremented before being returned.

We examine `find_get_entry()` in Listing 9-41.

```

2001 static inline struct folio *find_get_entry(struct xa_state *xas, pgoff_t max,
2002                                           xa_mark_t mark)
2003 {
2004     struct folio *folio;
2005
2006     retry:
2007     if (mark == XA_PRESENT)
2008         folio = xas_find(xas, max);
2009     else
2010         folio = xas_find_marked(xas, max, mark);
2011
2012     if (xas_retry(xas, folio))
2013         goto retry;
2014     /*
2015      * A shadow entry of a recently evicted page, a swap
2016      * entry from shmem/tmpfs or a DAX entry. Return it

```

```

2017      * without attempting to raise page count.
2018      */
2019      if (!folio || xa_is_value(folio))
2020          return folio;
2021
2022      if (!folio_try_get_rcu(folio))
2023          goto reset;
2024
2025      if (unlikely(folio != xas_reload(xas))) {
2026          folio_put(folio);
2027          goto reset;
2028      }
2029
2030      return folio;
2031 reset:
2032      xas_reset(xas);
2033      goto retry;
2034 }

```

Listing 9-41: mm/filemap.c: find_get_entry()

Though this function can be used to look up entries with specific xarray marks via `xas_find_marked()`, we exclusively consider the case where mark is set to `XA_PRESENT`, i.e. looking up present folios.

The remainder of the logic resembles that of `filemap_get_read_batch()` shown in Listing 9-47 above, though most closely matches that of `mapping_get_entry()`. In order to avoid repeating ourselves, we defer our description of this logic to the discussion around Listing 9-52 below, and the associated discussion around Listing 9-51 describing the lockless page cache protocol as a whole.

In the instance where we wish to retrieve folios in this fashion but also lock them, further effort is required, which is provided in `find_lock_entries()`, shown in Listing 9-42 below. This is utilised by the cache dropping logic discussed in Section 9.9.4, in the function `invalidate_mapping_pagevec()` shown in Listing 9-116.

```

2073 /**
2074  * find_lock_entries - Find a batch of pagecache entries.
2075  * @mapping:      The address_space to search.
2076  * @start:        The starting page cache index.
2077  * @end:          The final page index (inclusive).
2078  * @fbatch:       Where the resulting entries are placed.
2079  * @indices:      The cache indices of the entries in @fbatch.
2080  *
2081  * find_lock_entries() will return a batch of entries from @mapping.
2082  * Swap, shadow and DAX entries are included. Folios are returned
2083  * locked and with an incremented refcount. Folios which are locked
2084  * by somebody else or under writeback are skipped. Folios which are
2085  * partially outside the range are not returned.

```

```

2086  *
2087  * The entries have ascending indexes. The indices may not be consecutive
2088  * due to not-present entries, large folios, folios which could not be
2089  * locked or folios under writeback.
2090  *
2091  * Return: The number of entries which were found.
2092  */
2093  unsigned find_lock_entries(struct address_space *mapping, pgoff_t start,
2094                          pgoff_t end, struct folio_batch *fbatch, pgoff_t *indices)
2095  {
2096      XA_STATE(xas, &mapping->i_pages, start);
2097      struct folio *folio;
2098
2099      rcu_read_lock();
2100      while ((folio = find_get_entry(&xas, end, XA_PRESENT))) {
2101          if (!xa_is_value(folio)) {
2102              if (folio->index < start)
2103                  goto put;
2104              if (folio->index + folio_nr_pages(folio) - 1 > end)
2105                  goto put;
2106              if (!folio_trylock(folio))
2107                  goto put;
2108              if (folio->mapping != mapping ||
2109                  folio_test_writeback(folio))
2110                  goto unlock;
2111              VM_BUG_ON_FOLIO(!folio_contains(folio, xas.xa_index),
2112                             folio);
2113          }
2114          indices[fbatch->nr] = xas.xa_index;
2115          if (!folio_batch_add(fbatch, folio))
2116              break;
2117          continue;
2118      unlock:
2119          folio_unlock(folio);
2120      put:
2121          folio_put(folio);
2122      }
2123      rcu_read_unlock();
2124
2125      return folio_batch_count(fbatch);
2126  }

```

Listing 9-42: mm/filemap.c: find_lock_entries()

This function behaves similarly to `find_get_entries()`, only additionally locking folios it finds (but not waiting on the lock, merely optimistically attempting to acquire the lock for each folio). As with that function, we must return page offset indices as we may skip over non-present folios, or en-

counter compound folios which span multiple pages. We of course increment the reference count of all folios we do find.

`find_lock_entries()` is invoked from callers where the folio is compound, so we must be careful to ensure that each folio we retrieve is actually within the specified range (from the page offset start to end inclusive) by testing that each struct `folio->index` is within the required range and does not span beyond it.

Importantly, we do not wait for the folio to be locked, only speculatively attempt to acquire it via `folio_trylock()`. If the folio is already locked, we skip it.

Once safely locked, we must ensure that the folio has not been truncated or since reallocated and attached to another page cache entry by comparing struct `folio->mapping` to `mapping`. In addition, we do not return folios under writeback (see the writeback chapter for details)

Finally, before adding folios to the batch, we perform a `CONFIG_DEBUG_VM` sanity check to ensure the folio has the correct index.

In `filemap_get_folios()` we retrieve folios in a similar fashion to `find_get_entry()`, only we do not return indices, skip exceptional values and update the start parameter (which is a pointer in this instance) to the start of the next folio to be examined once the batch is full.

Unlike `find_lock_entries()`, we permit folios to overhang the start and beginning of the input range. This function is shown in Listing 9-43 (eliding out of scope `hugetlb` logic).

```

2128 /**
2129  * filemap_get_folios - Get a batch of folios
2130  * @mapping:      The address_space to search
2131  * @start:        The starting page index
2132  * @end:          The final page index (inclusive)
2133  * @fbatch:       The batch to fill.
2134  *
2135  * Search for and return a batch of folios in the mapping starting at
2136  * index @start and up to index @end (inclusive). The folios are returned
2137  * in @fbatch with an elevated reference count.
2138  *
2139  * The first folio may start before @start; if it does, it will contain
2140  * @start. The final folio may extend beyond @end; if it does, it will
2141  * contain @end. The folios have ascending indices. There may be gaps
2142  * between the folios if there are indices which have no folio in the
2143  * page cache. If folios are added to or removed from the page cache
2144  * while this is running, they may or may not be found by this call.
2145  *
2146  * Return: The number of folios which were found.
2147  * We also update @start to index the next folio for the traversal.
2148  */
2149 unsigned filemap_get_folios(struct address_space *mapping, pgoff_t *start,
2150                             pgoff_t end, struct folio_batch *fbatch)
2151 {

```

```

2152     XA_STATE(xas, &mapping->i_pages, *start);
2153     struct folio *folio;
2154
2155     rcu_read_lock();
2156     while ((folio = find_get_entry(&xas, end, XA_PRESENT)) != NULL) {
2157         /* Skip over shadow, swap and DAX entries */
2158         if (xa_is_value(folio))
2159             continue;
2160         if (!folio_batch_add(fbatch, folio)) {
2161             unsigned long nr = folio_nr_pages(folio);
2162             :
2165             *start = folio->index + nr;
2166             goto out;
2167         }
2168     }
2169
2170     /*
2171     * We come here when there is no page beyond @end. We take care to not
2172     * overflow the index @start as it confuses some of the callers. This
2173     * breaks the iteration when there is a page at index -1 but that is
2174     * already broken anyway.
2175     */
2176     if (end == (pgoff_t)-1)
2177         *start = (pgoff_t)-1;
2178     else
2179         *start = end + 1;
2180 out:
2181     rcu_read_unlock();
2182
2183     return folio_batch_count(fbatch);
2184 }

```

Listing 9-43: mm/filemap.c: `filemap_get_folios()`

This function resembles those that we have examined already so we won't belabour the point by going over the logic once again, though do note the careful handling of the case where no page exists beyond end.

There are other variants on the theme of these lookup functions like `find_get_pages_contig()` and `find_get_pages_range_tag()`, however they broadly resemble what we have already examined so what we have discussed so far should suffice to underline the general mechanisms by which folios are found in the page cache.

9.5.2 Reading a Batch of Folios Into the Page Cache

We start by examining `filemap_get_pages()` in Listing 9-44 (eliding out-of-scope iouring-specific IOCB_WAITQ handling), which is used by `filemap_read()` (see Section 9.3 and Listing 9-30) to retrieve a *batch* of folios from the page cache.

```

2546 static int filemap_get_pages(struct kiocb *iocb, struct iov_iter *iter,
2547                             struct folio_batch *fbatch)
2548 {
2549     struct file *filp = iocb->ki_filp;
2550     struct address_space *mapping = filp->f_mapping;
2551     struct file_ra_state *ra = &filp->f_ra;
2552     pgoff_t index = iocb->ki_pos >> PAGE_SHIFT;
2553     pgoff_t last_index;
2554     struct folio *folio;
2555     int err = 0;
2556
2557     last_index = DIV_ROUND_UP(iocb->ki_pos + iter->count, PAGE_SIZE);
2558     retry:
2559     if (fatal_signal_pending(current))
2560         return -EINTR;
2561
2562     filemap_get_read_batch(mapping, index, last_index, fbatch);
2563     if (!folio_batch_count(fbatch)) {
2564         if (iocb->ki_flags & IOCB_NOIO)
2565             return -EAGAIN;
2566         page_cache_sync_readahead(mapping, ra, filp, index,
2567                                   last_index - index);
2568         filemap_get_read_batch(mapping, index, last_index, fbatch);
2569     }
2570     if (!folio_batch_count(fbatch)) {
2571         if (iocb->ki_flags & (IOCB_NOWAIT | IOCB_WAITQ))
2572             return -EAGAIN;
2573         err = filemap_create_folio(filp, mapping,
2574                                   iocb->ki_pos >> PAGE_SHIFT, fbatch);
2575         if (err == AOP_TRUNCATED_PAGE)
2576             goto retry;
2577         return err;
2578     }
2579
2580     folio = fbatch->folios[folio_batch_count(fbatch) - 1];
2581     if (folio_test_readahead(folio)) {
2582         err = filemap_readahead(iocb, filp, mapping, folio, last_index
2583                                );
2584         if (err)
2585             goto err;
2586     }
2587     if (!folio_test_uptodate(folio)) {
2588         :
2589
2590         err = filemap_update_page(iocb, mapping, iter, folio);
2591         if (err)
2592             goto err;
2593     }

```



```

2594
2595         return 0;
2596     err:
2597         if (err < 0)
2598             folio_put(folio);
2599         if (likely(--fbatch->nr))
2600             return 0;
2601         if (err == AOP_TRUNCATED_PAGE)
2602             goto retry;
2603         return err;
2604     }

```

Listing 9-44: mm/filemap.c: `filemap_get_pages()`

We determine the range of pages we need to read based on the struct `kiocb` parameter `iocb` (see Section 9.3 for more details on this type—it is in effect used to transmit state threaded through the iterator operation).

We use the struct `kiocb->ki_pos` field to determine our position within the file, shifted by `PAGE_SHIFT` to determine the base page index from which we must read, and place this in the `index` variable.

The whole read operation is ultimately controlled by the struct `iov_iter` object, from which we can determine the number of *bytes* remaining to be read from the struct `iov_iter->count` field.

We use this to calculate the *inclusive* upper bound of the page index via `DIV_ROUND_UP()` (noting that if the index is offset into a folio, we will obtain the full folio).

NOTE

A consequence of this is that if the iterator range is page-aligned, we also read the folio after the last requested. So for instance, if we try to read a single page, we will actually read two pages, not one.

At this point we have a starting point within the struct `address_space` object for this page cache entry, `index`, and an inclusive upper bound on the range we wish to read in `last_index` and are ready to retrieve associated folios and place them into the struct `folio_batch` object, `fbatch` (see Section 11.7 for more on folio batches).

Before we proceed with this operation, we check to see whether a fatal signal is pending for the process before we potentially perform any I/O via `readahead` (see Section 9.7). If so we exit early, specifying the `-EINTR` error code.

We perform the actual read of page cache pages via `filemap_get_read_batch()` which we shall examine below in Listing 9-47.

We check to see whether we retrieved any folios via `folio_batch_count()`, if not then we try to perform *synchronous* `readahead` via `page_cache_sync_readahead()`, unless the `IOCB_NODATA` flag has been specified for the iterator, indicating that I/O is not permitted (used by some specific file systems for scenarios where this is appropriate)—in this case we simply return the `EAGAIN` error to indicate I/O was requested so repeat the operation.

This function unconditionally reads ahead at least a single page, even if the read is performed in random-access mode (unlike `do_sync_mmap_readahead()`), so we should obtain at least a single folio when we invoke `filemap_get_read_batch()` again. See Section 9.7 for an exploration of `readahead` in general.

Afterwards we encounter a slightly tricky edge case, where we fail to retrieve folios even after synchronous `readahead`:

Check whether we did in fact retrieve any folios after readahead

`Readahead` may fail due to either the system simply being out of memory (it may even be in the midst of servicing an OOM killer event), or the allocation GFP flags (see Section 2.6 for more on this) did not permit an allocation to occur. In addition, extreme memory pressure even without OOM may have prevented an allocation, for instance if `GFP_NOFS` was specified, but the allocation would require file system access. In these circumstances, there can be significant races between freeing of memory and an attempt at reading it.

If `readahead` failed, non-blocking case We will have to perform I/O so we start by checking whether `IOCB_NOWAIT` (or the out-of-scope `io_uring`-specified `IOCB_WAITQ`) flags are specified, in which case we return the `EAGAIN` error code to indicate that we would block in instances where that's not permitted.

Last-ditch attempt to read data At this point we are under serious memory pressure, so we simply try to *directly* read in a folio, via `filemap_create_folio()` which both allocates a folio to read the data into and goes ahead and reads it from disk.

Check for truncation If `filemap_create_folio()` returns `AOP_TRUNCATED_PAGE` this indicates that the folio might have been truncated. In this case we jump back to the `retry` label and attempt the operation again.

Exit Finally the `readahead` failure handling logic exits indicating whether an error occurred. We have done our best—attempting to retrieve a single folio.

We examine `filemap_create_folio()` in Listing 9-45 below.

```

2489 static int filemap_create_folio(struct file *file,
2490                               struct address_space *mapping, pgoff_t index,
2491                               struct folio_batch *fbatch)
2492 {
2493     struct folio *folio;
2494     int error;
2495
2496     folio = filemap_alloc_folio(mapping_gfp_mask(mapping), 0);
2497     if (!folio)
2498         return -ENOMEM;
2499
2500     /*
```

```

2501      * Protect against truncate / hole punch. Grabbing invalidate_lock
2502      * here assures we cannot instantiate and bring uptodate new
2503      * pagecache folios after evicting page cache during truncate
2504      * and before actually freeing blocks. Note that we could
2505      * release invalidate_lock after inserting the folio into
2506      * the page cache as the locked folio would then be enough to
2507      * synchronize with hole punching. But there are code paths
2508      * such as filemap_update_page() filling in partially uptodate
2509      * pages or ->readahead() that need to hold invalidate_lock
2510      * while mapping blocks for IO so let's hold the lock here as
2511      * well to keep locking rules simple.
2512      */
2513      filemap_invalidate_lock_shared(mapping);
2514      error = filemap_add_folio(mapping, folio, index,
2515                               mapping_gfp_constraint(mapping, GFP_KERNEL));
2516      if (error == -EEXIST)
2517          error = AOP_TRUNCATED_PAGE;
2518      if (error)
2519          goto error;
2520
2521      error = filemap_read_folio(file, mapping->a_ops->read_folio, folio);
2522      if (error)
2523          goto error;
2524
2525      filemap_invalidate_unlock_shared(mapping);
2526      folio_batch_add(fbatch, folio);
2527      return 0;
2528  error:
2529      filemap_invalidate_unlock_shared(mapping);
2530      folio_put(folio);
2531      return error;
2532  }

```

Listing 9-45: mm/filemap.c: filemap_create_folio()

We start by allocating a folio via `filemap_alloc_folio()`, returning an out of memory error if we are unable to do so. After this we perform some careful juggling with locking around invalidation via `filemap_invalidate_lock_shared()`, before adding the newly created folio to the page cache via `filemap_add_folio()` which we examine in Listing 9-54 and Section 12.2.1.

If we are unable to add the folio to the page cache we exit with an error, explicitly indicating whether this was a duplicate entry caused by truncation via `AOP_TRUNCATED_PAGE`.

Finally we perform the actual read from disk via `filemap_read_folio()`, a function we examine in detail in Section 9.6 and Listing 9-56 below.

Returning to `filemap_get_pages()`, once we have handled this edge case, we proceed with the usual case—the batch returned by `filemap_get_read_batch()`

contains specific additional information in the final folio in the batch, as this might either be the last folio before which further readahead is required, or the last folio before which folios are not up to date with the disk, in which case the folios need to be read from disk.

In the first case, `filemap_readahead()` is performed, see Section 9.7 and Listing 9-67 for an exploration of this.

In the second case, `filemap_update_page()` is performed, which we examine in Listing 9-46 below (eliding out of scope `io_uring`-specific logic and distracting no wait handling):

```

2434 static int filemap_update_page(struct kiocb *iocb,
2435                               struct address_space *mapping, struct iov_iter *iter,
2436                               struct folio *folio)
2437 {
2438     int error;
2439     :
2440     :
2441     filemap_invalidate_lock_shared(mapping);
2442     :
2443     if (!folio_trylock(folio)) {
2444         :
2445         :
2446         filemap_invalidate_unlock_shared(mapping);
2447         /*
2448          * This is where we usually end up waiting for a
2449          * previously submitted readahead to finish.
2450          */
2451         folio_put_wait_locked(folio, TASK_KILLABLE);
2452         return AOP_TRUNCATED_PAGE;
2453     }
2454     :
2455     error = AOP_TRUNCATED_PAGE;
2456     if (!folio->mapping)
2457         goto unlock;
2458     :
2459     error = 0;
2460     if (filemap_range_uptodate(mapping, iocb->ki_pos, iter, folio))
2461         goto unlock;
2462     :
2463     error = filemap_read_folio(iocb->ki_filp, mapping->a_ops->read_folio,
2464                               folio);
2465     goto unlock_mapping;
2466 unlock:
2467     folio_unlock(folio);
2468 unlock_mapping:
2469     filemap_invalidate_unlock_shared(mapping);
2470     if (error == AOP_TRUNCATED_PAGE)
2471         folio_put(folio);
2472     return error;

```

2487 }

Listing 9-46: mm/filemap.c: filemap_update_page()

We carefully lock around the invalidation using `filemap_invalidate_lock_shared()`, before trying to obtain a lock on the folio in order to read from disk into it. If there is contention on the lock we release the invalidation lock and combine the decrement of the folio's reference count and waiting on the lock to be released in a single operation via `folio_put_wait_locked()`. See Section 9.11 for more details on folio locking as a whole. We indicate that this occurred, likely due to a truncation, by returning `AOP_TRUNCATED_PAGE`.

if the locking is successful, but the struct `folio->mapping` field is empty, this indicates a racing truncation occurred and thus we return `AOP_TRUNCATED_PAGE`, indicating that this has happened.

We check to see if the folio is uptodate via `filemap_range_uptodate()`, which has special handling for this case where the folio is part of an iterator range. If so, then we can simply exit.

Otherwise, we perform the actual read from disk into the folio via `filemap_read_folio()`, a function we examine in detail in Section 9.6 and Listing 9-56 below.

When done we unlock the folio and return any error that arose. Otherwise the folio is now present and uptodate.

Returning to `filemap_get_pages()`—regardless of whether an update was required or not, if an error occurs, we drop the final folio's reference, and if we processed at least one folio successfully, we treat this as if it were a success. Otherwise, if the error indicated a possible truncation we jump to the retry label, finally returning any error if no folios could be processed.

The heavy lifting of this operation is performed in `filemap_get_read_batch()`, which we explore in Listing 9-47.

```

2337 /*
2338  * filemap_get_read_batch - Get a batch of folios for read
2339  *
2340  * Get a batch of folios which represent a contiguous range of bytes in
2341  * the file. No exceptional entries will be returned. If @index is in
2342  * the middle of a folio, the entire folio will be returned. The last
2343  * folio in the batch may have the readahead flag set or the uptodate flag
2344  * clear so that the caller can take the appropriate action.
2345  */
2346 static void filemap_get_read_batch(struct address_space *mapping,
2347                                   pgoff_t index, pgoff_t max, struct folio_batch *fbatch)
2348 {
2349     XA_STATE(xas, &mapping->i_pages, index);
2350     struct folio *folio;
2351
2352     rcu_read_lock();
2353     for (folio = xas_load(&xas); folio; folio = xas_next(&xas)) {
2354         if (xas_retry(&xas, folio))

```

```

2355         continue;
2356         if (xas.xa_index > max || xa_is_value(folio))
2357             break;
2358         if (xa_is_sibling(folio))
2359             break;
2360         if (!folio_try_get_rcu(folio))
2361             goto retry;
2362
2363         if (unlikely(folio != xas_reload(&xas)))
2364             goto put_folio;
2365
2366         if (!folio_batch_add(fbatch, folio))
2367             break;
2368         if (!folio_test_uptodate(folio))
2369             break;
2370         if (folio_test_readahead(folio))
2371             break;
2372         xas_advance(&xas, folio->index + folio_nr_pages(folio) - 1);
2373         continue;
2374 put_folio:
2375         folio_put(folio);
2376 retry:
2377         xas_reset(&xas);
2378     }
2379     rcu_read_unlock();
2380 }

```

Listing 9-47: mm/filemap.c: filemap_get_read_batch()

We use the xarray data structure obtained from the page cache mapping's struct `address_space` object to look up folios (see Section 9.2 for more on this).

Throughout we thread a struct `xa_state` object, initialised via `XA_STATE()`.

The operation is performed under *RCU* lock, the discussion of which is out of scope here, however we can be certain in this circumstance that, while under this lock the folio cannot be pulled out from beneath us.

We start by checking whether the operation needs to be retried based on the result of `xas_retry()`—these are cases where the xarray walk must be restarted. The function invokes `xas_reset()` to reset the walk so we need only loop around in this circumstance.

Next, we consider edge cases. This list is a great example of just how careful you have to be around races in the memory management subsystem, and the degree to which things are permitted to remain in flight for efficiency:

Has the iteration terminated? If struct `xa_state->xa_index` exceeds the maximum inclusive index `max`, we're done and exit.

Is the entry exceptional? The page cache mechanism sometimes causes 'exceptional' entries to be present in the xarray, i.e. ones that contain

values rather than pointers. If this is the case (checked via `xa_is_value()`), then this indicates the folio has been evicted and thus we must abort the operation (we strictly return contiguous entries).

Did we race with an invalidate/read? If we race with an invalidation followed by another read, this can lead to a higher order folio being loaded, and we may end up observing a tail page. In this instance, signified by `xa_is_sibling()`, we must abort our iteration.

Did the folio get released? We try to get a reference to the folio via `folio_try_get_rcu()`, however it may have been ‘frozen’ (see the reclaim chapter on this) in preparation for being freed, something which the RCU doesn’t protect us again, so we retry in this case.

Has the folio moved? We can finally relax a little as we’ve obtained a reference count to the folio, however it still might have moved (or been removed) in the meantime, so we check by reloading the entry via `xas_reload()`—if the folio is not the same then we must drop the reference and try this dance again.

See also Listing 9-51 below for a discussion around a highly relevant comment contained in the filemap code which outlines the lockless page cache protocol.

Once we have ruled out these edge cases, we have a folio which has had its reference count incremented (i.e. is pinned) and is as advertised, so we add it to the struct `folio_batch` via `folio_batch_add()`.

After this, we then must consider whether the just-added folio both up to date with the disk via `folio_test_uptodate()` and does not have the readahead flag set (which indicates that this is the point where the last readahead stopped and more need be performed), as checked by `folio_test_readahead()`.

If either of these are the case, then we must return to the caller to have them deal with this, so we exit early.

Finally, we advance the struct `xa_state` past all tail pages (if the folio is higher order) using `xas_advance()`.

9.5.3 Reading a Single Folio From the Page Cache

When we perform a page fault, as discussed in Section 9.4, we first need to determine whether that folio is present in the page cache and retrieve it if so. We do this via `filemap_get_folio()` as shown in Listing 9-48.

```

516 /**
517  * filemap_get_folio - Find and get a folio.
518  * @mapping: The address_space to search.
519  * @index: The page index.
520  *
521  * Looks up the page cache entry at @mapping & @index. If a folio is
522  * present, it is returned with an increased refcount.
523  *
524  * Otherwise, %NULL is returned.
```

```

525 */
526 static inline struct folio *filemap_get_folio(struct address_space *mapping,
527                                              pgoff_t index)
528 {
529     return __filemap_get_folio(mapping, index, 0, 0);
530 }

```

Listing 9-48: include/linux/pagemap.h: filemap_get_folio()

The folio is not locked, nor is it read from disk if it is not uptodate nor is a new folio allocated if one was not already present. The heavy lifting is deferred to `__filemap_get_folio()`, which has a great deal of flexibility in its behaviour, examined in Listing 9-49 (eliding comments describing flags which we describe below and out of scope page idle handling).

In addition, `__filemap_get_folio()` can be invoked with a set of flags to enable a great deal of control over how the folio is retrieved and whether or not we allocate a new folio and add it to the page cache should one not be present (as we do on page fault in `filemap_fault()` described in Section 9.4 above).

`__filemap_get_folio()` uses ‘Find Get Page’ (FGP) flags to determine its behaviour:

FGP_ACCESSED The folio will be marked accessed.

FGP_LOCK The folio will be returned locked.

FGP_ENTRY By default, if the folio entry in the xarray is not actually a pointer to a folio but is rather ‘exceptional’, i.e. indicates a freshly reclaimed entry (a.k.a. a ‘shadow’ entry, this is out of scope for the book), or a swap entry (see the chapter on swap) or a DAX entry (out of scope for the book), this function will not return them. This flag alters this behaviour and instead returns the entry as-is in this case.

FGP_CREAT This is a key behavioural flag—if a folio cannot be found for whatever reason and this flag is *not* set, then the function returns `NULL`. However if this flag is set, then a folio will be allocated, added to the page cache and locked, ready to be read in to. Importantly, the folio will *not* be up to date, i.e. will not be read from disk, it is up to the caller to do this if required. The call may sleep in this instance

FGP_FOR_MM Indicates that the folio should be returned unlocked, as the caller setting this flag, i.e. `filemap_fault()`, as it is the only function to do so, expects it.

FGP_WRITE Indicates that the folio will be written to. This ultimately results in the `__GFP_WRITE` flag being used on folio allocation if `FGP_CREATE` is also specified and the underlying block device can write back (as determined by `mapping_can_writeback()`). See Section 2.6 for details on this flag. `FGP_WRITE` also affects idle page handling, however this topic is out of scope for the book.

FGP_NOFS (Only meaningful if `FGP_CREAT` is set) Indicates that the file system should not be invoked on allocation by clearing the `__GFP_FS` flag on

physical page allocation. This is typically used when a filesystem must retrieve a folio but recursive access to the filesystem cannot be permitted (for instance to avoid deadlocks).

FGP_NOWAIT If `FGP_LOCK` is specified, indicates that we should not wait for the folio to become unlocked before we try to lock it. If we cannot, we indicate this by explicitly returning `NULL`. Additionally, if `FGP_CREAT` is specified, we clear the default `GFP_KERNEL` physical allocation flag and instead set the combination of `GFP_NOWAIT` and `__GFP_NOWARN` to indicate we must not perform direct reclaim (which might cause waiting), implies that no file system or I/O activity may be permitted and does not output warning about a failure to allocate to the kernel log. See Section 2.6 for more details.

FGP_STABLE There are situations where it is important that the folio is ‘stable’, i.e. not undergoing writeback at the present time. Setting this flag ensures that writeback is complete by the time the folio is returned, as implemented by the `folio_wait_stable()` function. See the writeback chapter for details on how writeback functions as a whole.

We start by examining the beginning of `__filemap_get_folio()` which performs the initial read from the page cache and handles associated FGP flags in Listing 9-49.

```

1881 /**
1882  * __filemap_get_folio - Find and get a reference to a folio.
1883  * @mapping: The address_space to search.
1884  * @index: The page index.
1885  * @fgp_flags: %FGP flags modify how the folio is returned.
1886  * @gfp: Memory allocation flags to use if %FGP_CREAT is specified.
1887  *
1888  * Looks up the page cache entry at @mapping & @index.
1889  *
1890  * @fgp_flags can be zero or more of these flags:
1891  *
1892  * If %FGP_LOCK or %FGP_CREAT are specified then the function may sleep even
1893  * if the %GFP flags specified for %FGP_CREAT are atomic.
1894  *
1895  * If there is a page cache page, it is returned with an increased refcount.
1896  *
1897  * Return: The found folio or %NULL otherwise.
1898  */
1899 struct folio *__filemap_get_folio(struct address_space *mapping, pgoff_t index
1900 ,
1901                                int fgp_flags, gfp_t gfp)
1902 {
1903     struct folio *folio;
1904
1905     repeat:

```

```

1920         folio = mapping_get_entry(mapping, index);
1921         if (xa_is_value(folio)) {
1922             if (fgp_flags & FGP_ENTRY)
1923                 return folio;
1924             folio = NULL;
1925         }
1926         if (!folio)
1927             goto no_page;
1928
1929         if (fgp_flags & FGP_LOCK) {
1930             if (fgp_flags & FGP_NOWAIT) {
1931                 if (!folio_trylock(folio)) {
1932                     folio_put(folio);
1933                     return NULL;
1934                 }
1935             } else {
1936                 folio_lock(folio);
1937             }
1938
1939             /* Has the page been truncated? */
1940             if (unlikely(folio->mapping != mapping)) {
1941                 folio_unlock(folio);
1942                 folio_put(folio);
1943                 goto repeat;
1944             }
1945             VM_BUG_ON_FOLIO(!folio_contains(folio, index), folio);
1946         }
1947
1948         if (fgp_flags & FGP_ACCESSED)
1949             folio_mark_accessed(folio);
1950         :
1951
1952         if (fgp_flags & FGP_STABLE)
1953             folio_wait_stable(folio);

```

Listing 9-49: mm/filemap.c: `__filemap_get_folio()` Reading From Page Cache

The actual read from the page cache is performed by `mapping_get_entry()`, which is examined below in Listing 9-52. Note that this will increment the reference count of the retrieved folio to pin it in memory, which must be decremented if something goes wrong.

NOTE

It's important to note here that all folios that are read into the page cache whether faulted-in, read from a file or via readahead will have their reference incremented when doing so. This is a product of the page cache 'holding on' to file data as long as possible in order to minimise I/O, relying on reclaim to released file-backed folios under memory pressure.

The reference count is incremented on introduction to the page cache, but also when added to private filesystem metadata, added to an LRU list, mapped into memory

and thus referenced by a page table, as well as any other mean by which the kernel references it.

We start by examining whether this value is ‘exceptional’, i.e. not NULL but also not a pointer to a folio, as determined by `xa_is_value()`. If `FGP_ENTRY` has been specified then we simply return this value as the caller expects it, otherwise we treat it as if the folio is not present (which is in fact accurate, these values always provide information describing a missing entry) setting folio to NULL.

Next, we check whether the folio either not found or exceptional without `FGP_ENTRY` specified, in which case we defer to the missing page handling at label `no_page`, as described below in Listing 9-50.

We then try to lock the folio if requested via `FGP_LOCK`. If this is not specified then the caller either performs locking itself, or whether the folio is in flux is immaterial to it. We will return to this logic shortly.

After the locking logic is handled, we simply set the folio accessed via `folio_mark_accessed()` if `FGP_ACCESSED` is specified (see the chapter on reclaim for more on the impact of this) and invoke `folio_wait_stable()` if `FGP_STABLE` is specified to wait for writeback to complete (see the writeback chapter for more details on writeback).

Returning to the lock logic. This is invoked only if the `FGP_LOCK` flag is set. If `FGP_NOWAIT` is also specified then we don’t wait for the lock but rather use `folio_trylock()` to attempt to acquire a lock if it is not already held. If it is then we release the reference on the folio via `folio_put()` and return NULL. If `FGP_NOWAIT` is not specified we simply use `folio_lock()` to wait for any existing lock to be released (see Section 9.11 below for more details on folio locking).

Finally, once the lock is acquired we check to determine whether the folio has been truncated and possibly since allocated to another mapping, which is indicated by the struct `folio->mapping` field no longer matching the mapping for which we are looking up folios.

In this instance, we unlock the folio via `folio_unlock()`, decrement its reference count via `folio_put()` and retry the operation by jumping to the repeat flag.

Finally the locking logic performs a sanity check (only applicable if `CONFIG_DEBUG_VM` is set) to ensure the index and folio tally to one another.

At this point, if the folio was found, we return it (on line 1998), otherwise we invoke the missing page logic shown in Listing 9-50.

```

1958 no_page:
1959     if (!folio && (fgp_flags & FGP_CREAT)) {
1960         int err;
1961         if ((fgp_flags & FGP_WRITE) && mapping_can_writeback(mapping))
1962             gfp |= __GFP_WRITE;
1963         if (fgp_flags & FGP_NOFS)
1964             gfp &= ~__GFP_FS;
1965         if (fgp_flags & FGP_NOWAIT) {
1966             gfp &= ~GFP_KERNEL;
1967             gfp |= GFP_NOWAIT | __GFP_NOWARN;

```

```

1968         }
1969
1970         folio = filemap_alloc_folio(gfp, 0);
1971         if (!folio)
1972             return NULL;
1973
1974         if (WARN_ON_ONCE(!(fgp_flags & (FGP_LOCK | FGP_FOR_MMAP))))
1975             fgp_flags |= FGP_LOCK;
1976
1977         /* Init accessed so avoid atomic mark_page_accessed later */
1978         if (fgp_flags & FGP_ACCESSED)
1979             __folio_set_referenced(folio);
1980
1981         err = filemap_add_folio(mapping, folio, index, gfp);
1982         if (unlikely(err)) {
1983             folio_put(folio);
1984             folio = NULL;
1985             if (err == -EEXIST)
1986                 goto repeat;
1987         }
1988
1989         /*
1990          * filemap_add_folio locks the page, and for mmap
1991          * we expect an unlocked page.
1992          */
1993         if (folio && (fgp_flags & FGP_FOR_MMAP))
1994             folio_unlock(folio);
1995     }
1996
1997     return folio;
1998 }

```

Listing 9-50: mm/filemap.c: __filemap_get_folio() Missing Page Handling

We only actually do something in the instance of a missing folio if `FGP_CREAT` is specified. In this case, we will allocate a folio and add it into the page cache.

We start by determining the ‘Get Free Pages’ (GFP) flags to be passed into the physical allocator. This is discussed in great detail in the Physical Memory chapter in Section 2.6, and these flags were already discussed in detail above.

Next we perform the actual allocation via `filemap_alloc_folio()`, which performs some NUMA-specific behaviour before deferring the operation to `folio_alloc()` which finally allocates using the physical page allocator via `alloc_pages()`, see Section 2.8 in the Physical Memory chapter for more on this.

If this allocation fails, then we simply return `NULL`. Otherwise we perform a sanity check to ensure that either we don’t require a lock as indicated by

FGP_FOR_MMAP or we do with FGP_LOCK specified. If this edge case occurs then we warn about it and override by setting this flag anyway (this is meaningful because we may repeat the operation).

We then add the newly allocated folio into the page cache via `filemap_add_folio()` (see Listing 9-54 and Section 12.2.1 below). Note that this locks the folio as part of this operation.

If an error occurs, we drop our reference on the folio, setting it to NULL unless the error is that the folio at this position already exists and a race occurred, in which case we repeat the operation to get it.

Finally, if FGP_FOR_MMAP is set we unlock the folio via `folio_unlock()` as the caller expects us to.

The actual lookup of a folio in the page cache is performed by `mapping_get_entry()`.

We examine this function in Listing 9-52. However, prior to this there is a very useful comment contained in the source, shown in Listing 9-51.

```

1818 /*
1819  * Lockless page cache protocol:
1820  * On the lookup side:
1821  * 1. Load the folio from i_pages
1822  * 2. Increment the refcount if it's not zero
1823  * 3. If the folio is not found by xas_reload(), put the refcount and retry
1824  *
1825  * On the removal side:
1826  * A. Freeze the page (by zeroing the refcount if nobody else has a reference)
1827  * B. Remove the page from i_pages
1828  * C. Return the page to the page allocator
1829  *
1830  * This means that any page may have its reference count temporarily
1831  * increased by a speculative page cache (or fast GUP) lookup as it can
1832  * be allocated by another user before the RCU grace period expires.
1833  * Because the refcount temporarily acquired here may end up being the
1834  * last refcount on the page, any page allocation must be freeable by
1835  * folio_put().
1836  */

```

Listing 9-51: mm/filemap.c: Lockless Page Cache Protocol

This protocol details how page cache operations can be performed under RCU lock only (which does not function like a classical lock but rather utilises the fact that we can know when things might be updated to know when it is safe to write to values without anything unexpectedly reading it).

We examine how folio removal happens in the page cache in Section 9.9 below. However broadly the ordering described here, with folios being retrieved/removed from the `struct address_space->i_pages` xarray results in the possibility that the reference count of a folio that has in fact been removed from the xarray has gets temporarily boosted prior to being freed thus preventing its freeing.

To handle this, we make sure to always use `xas_reload()` in all instances where this might happen, for instance in `mapping_get_entry()` described in Listing 9-52 below and in `filemap_get_read_batch()` described in Listing 9-47 described previously.

```

1838  /*
1839  * mapping_get_entry - Get a page cache entry.
1840  * @mapping: the address_space to search
1841  * @index: The page cache index.
1842  *
1843  * Looks up the page cache entry at @mapping & @index. If it is a folio,
1844  * it is returned with an increased refcount. If it is a shadow entry
1845  * of a previously evicted folio, or a swap entry from shmem/tmpfs,
1846  * it is returned without further action.
1847  *
1848  * Return: The folio, swap or shadow entry, %NULL if nothing is found.
1849  */
1850  static void *mapping_get_entry(struct address_space *mapping, pgoff_t index)
1851  {
1852      XA_STATE(xas, &mapping->i_pages, index);
1853      struct folio *folio;
1854
1855      rcu_read_lock();
1856  repeat:
1857      xas_reset(&xas);
1858      folio = xas_load(&xas);
1859      if (xas_retry(&xas, folio))
1860          goto repeat;
1861      /*
1862       * A shadow entry of a recently evicted page, or a swap entry from
1863       * shmem/tmpfs. Return it without attempting to raise page count.
1864       */
1865      if (!folio || xa_is_value(folio))
1866          goto out;
1867
1868      if (!folio_try_get_rcu(folio))
1869          goto repeat;
1870
1871      if (unlikely(folio != xas_reload(&xas))) {
1872          folio_put(folio);
1873          goto repeat;
1874      }
1875  out:
1876      rcu_read_unlock();
1877
1878      return folio;
1879  }

```

Listing 9-52: mm/filemap.c: mapping_get_entry()

This parallels `filemap_get_read_batch()` in its general operation (see Listing 9-47 above), though it retrieves only a single folio.

The operation is performed under an *RCU* lock, as briefly described above in the discussion around 9-51, is inherently racey so must be performed with care. As in `filemap_get_read_batch()`, we load an entry via `xas_load()` and check whether the entry indicates the operation needs to be retried via `xas_retry()`, resetting and repeating the entire operation if so. If no folio is found, we return `NULL`.

Next we check to see if the entry is ‘exceptional’, i.e. not `NULL` but also not a pointer to a folio, this indicates it has recently been freed either as a ‘shadow’ entry left after removal (see Section 9.9 below) or is a swap entry (see the Swap chapter for more). In each case we simply return the value for the caller to deal with.

Next we try to increment the reference count for the folio via `folio_try_get_rcu()`. This may race with folio removal, so if it fails we simply repeat the operation.

Finally, now we have incremented the reference count we, as discussed, may race with the folio being moved (or removed) so we reload the entry and check to ensure that the folio matches. If not we drop the reference count and repeat.

9.5.4 Reading a Single Folio Into the Page Cache

There are a number of file system, driver and internal kernel callers which ultimately need to retrieve a single folio from the page cache, reading from disk into it if it is not already present. We examine these callers in Figure 9-9.

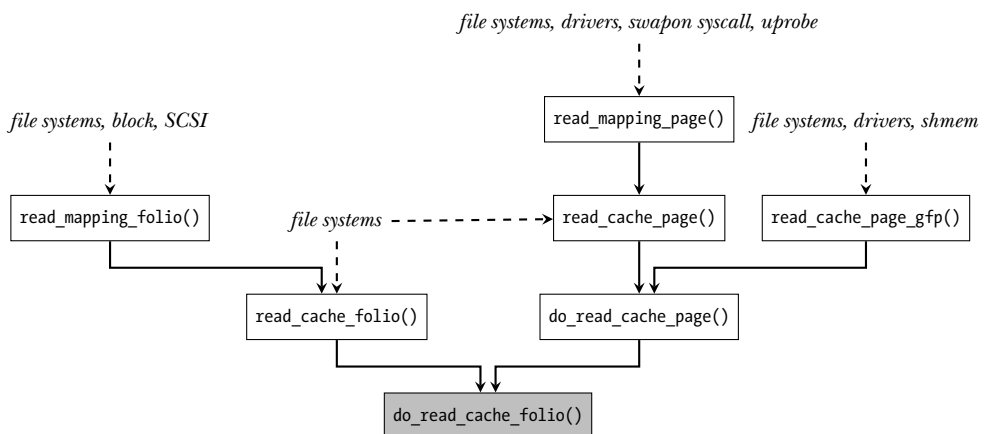


Figure 9-9: Callers Reading a Single Folio Into the Page Cache

Regardless of what invokes the original read, we ultimately end up in `do_read_cache_folio()` which we examine in Listing 9-53.

```

3472 static struct folio *do_read_cache_folio(struct address_space *mapping,
3473                                         pgoff_t index, filler_t filler, struct file *file, gfp_t gfp)
3474 {
3475     struct folio *folio;
3476     int err;
3477
3478     if (!filler)
3479         filler = mapping->a_ops->read_folio;
3480 repeat:
3481     folio = filemap_get_folio(mapping, index);
3482     if (!folio) {
3483         folio = filemap_alloc_folio(gfp, 0);
3484         if (!folio)
3485             return ERR_PTR(-ENOMEM);
3486         err = filemap_add_folio(mapping, folio, index, gfp);
3487         if (unlikely(err)) {
3488             folio_put(folio);
3489             if (err == -EEXIST)
3490                 goto repeat;
3491             /* Presumably ENOMEM for xarray node */
3492             return ERR_PTR(err);
3493         }
3494
3495         goto filler;
3496     }
3497     if (folio_test_uptodate(folio))
3498         goto out;
3499
3500     if (!folio_trylock(folio)) {
3501         folio_put_wait_locked(folio, TASK_UNINTERRUPTIBLE);
3502         goto repeat;
3503     }
3504
3505     /* Folio was truncated from mapping */
3506     if (!folio->mapping) {
3507         folio_unlock(folio);
3508         folio_put(folio);
3509         goto repeat;
3510     }
3511
3512     /* Someone else locked and filled the page in a very small window */
3513     if (folio_test_uptodate(folio)) {
3514         folio_unlock(folio);
3515         goto out;
3516     }
3517
3518 filler:

```



```

3519         err = filemap_read_folio(file, filler, folio);
3520         if (err) {
3521             folio_put(folio);
3522             if (err == AOP_TRUNCATED_PAGE)
3523                 goto repeat;
3524             return ERR_PTR(err);
3525         }
3526
3527     out:
3528         folio_mark_accessed(folio);
3529         return folio;
3530 }

```

Listing 9-53: mm/filemap.c: do_read_cache_folio()

The function is passed an optional `filler_t` callback (see Listing 9-57) via the `filler` parameter, which specifies a function which performs the actual read from disk of a folio if necessary. If not specified, this is defaulted to the struct `address_space_operations`-specified struct `address_space->a_ops->read_folio` filesystem-specific callback. We discuss how these reads function in Section 9.6 below.

We attempt to retrieve the folio from the page cache via `filemap_get_folio()` (see Section 9.5.3 and Listing 9-48). If a page cache entry is not found, we allocate one via `filemap_alloc_folio()`, returning an out of memory error if the allocation fails.

We then add this folio into the page cache via `filemap_add_folio()` which we examine in Listing 9-54 and Section 12.2.1 below.

If we are unable to add the folio to the page cache due to an existing folio being present at this offset, we simply repeat our attempt to retrieve the folio from the page cache in order to locate the already existing one.

otherwise, if another error arose, we return this error. In both instances we drop the reference count on the allocated folio such that it will be reclaimed

If we allocated a folio then it cannot be up-to-date, so in this instance we skip straight ahead to ‘filling’ the folio at label `filler`, i.e. reading it from disk.

If we did indeed locate the folio within the page cache, which at this point will have its reference count incremented, we check to see if it is up-to-date. If so then we have nothing further to do other than housekeeping, jumping to the `out` label, otherwise we must lock the folio and get ready to read into it from disk.

When we try to lock the folio, we combine the reference count decrement and sleep function in an invocation of `folio_put_wait_locked()`. See Section 9.11 for more details on folio locking as a whole.

Before we read from disk, we must perform the typical dance to protect ourselves against truncation. If we find that the struct `folio->mapping` reference to page cache object does not match the page cache object under examination then we unlock the folio, decrement its reference count and repeat

the procedure (perhaps allocating a new folio had the prior one been truncated).

Next we examine the edge case where the folio was updated in the very brief time window available to it. If so we need not read from disk, and so unlock the folio and exit.

Finally we are in the position where the folio is both locked and in need of being read from disk in this instance then we perform the read using `filemap_read_folio()`, a function we examine in detail in Section 9.6 and Listing 9-56 below.

If this fails, we drop the reference count, repeating the whole operation if this was due to truncation (as indicated by an error code of `AOP_TRUNCATED_PAGE`), exiting otherwise.

Finally, all being well, we mark the folio accessed via `folio_mark_accessed` (see Listing 11-11 in the reclaim chapter for a discussion of this function) as this page cache access indicates that the file has actively been accessed. After doing so we return the folio, now present and uptodate in the page cache.

9.5.5 Adding Folios to the Page Cache

Once folios intended for the page cache have been allocated, they need to be added to the page cache and inserted into the correct position within the relevant xarray. This is performed by `filemap_add_folio()` which we examine in Listing 9-54 (eliding out of scope working set logic).

```

926 int filemap_add_folio(struct address_space *mapping, struct folio *folio,
927                      pgoff_t index, gfp_t gfp)
928 {
929     void *shadow = NULL;
930     int ret;
931
932     __folio_set_locked(folio);
933     ret = __filemap_add_folio(mapping, folio, index, gfp, &shadow);
934     if (unlikely(ret))
935         __folio_clear_locked(folio);
936     else {
937         :
938         folio_add_lru(folio);
939     }
940     return ret;
941 }

```

Listing 9-54: `mm/filemap.c: filemap_add_folio()`

Note that we track folios which were recently evicted for the working set logic, which is stored in value fields contained in the xarray which reference ‘shadow’ entries. This is out of scope for the book.

This locks the folio then invokes `__filemap_add_folio()` to do the heavy lifting, which also clears the lock if no error arose, we clear it manually if one did. This therefore returns locked folios on success.

Finally if this succeeded, we add the folio to an LRU via `folio_add_lru()` (see the reclaim chapter in Section ?? and Listing 11-88 for details).

We examine `__filemap_add_folio()` in Listing 9-55 (eliding out of scope working set logic, debug assertions, huge page, tracing, statistics and cgroup logic).

```

838 inline int __filemap_add_folio(struct address_space *mapping,
839                               struct folio *folio, pgoff_t index, gfp_t gfp, void **shadowp)
840 {
841     XA_STATE(xas, &mapping->i_pages, index);
842     :
843     :
844     long nr = 1;
845     :
846     xas_set_order(&xas, index, folio_order(folio));
847     nr = folio_nr_pages(folio);
848     :
849     gfp &= GFP_RECLAIM_MASK;
850     folio_ref_add(folio, nr);
851     folio->mapping = mapping;
852     folio->index = xas.xa_index;
853
854     do {
855         unsigned int order = xa_get_order(xas.xa, xas.xa_index);
856         void *entry, *old = NULL;
857
858         if (order > folio_order(folio))
859             xas_split_alloc(&xas, xa_load(xas.xa, xas.xa_index),
860                             order, gfp);
861
862         xas_lock_irq(&xas);
863         xas_for_each_conflict(&xas, entry) {
864             old = entry;
865             if (!xa_is_value(entry)) {
866                 xas_set_err(&xas, -EEXIST);
867                 goto unlock;
868             }
869         }
870
871         if (old) {
872             :
873             /* entry may have been split before we acquired lock
874              */
875             order = xa_get_order(xas.xa, xas.xa_index);
876             if (order > folio_order(folio)) {
877                 :
878                 xas_split(&xas, old, order);
879                 xas_reset(&xas);
880             }
881         }
882     }

```

```

893
894         xas_store(&xas, folio);
895         if (xas_error(&xas))
896             goto unlock;
897
898         mapping->nripages += nr;
899         :
900 unlock:
901         xas_unlock_irq(&xas);
902     } while (xas_nomem(&xas, gfp));
903
904     if (xas_error(&xas))
905         goto error;
906     :
907     return 0;
908 error:
909     :
910     folio->mapping = NULL;
911     /* Leave page->index set: truncation relies upon it */
912     folio_put_refs(folio, nr);
913     return xas_error(&xas);
914 }

```

Listing 9-55: mm/filemap.c: `__filemap_add_folio()`

NOTE

The majority of the discussion of this function centres around logic relating to working set detection and ‘shadow entries’ used as part of this logic. This is out of scope for the book, however since it comprises such a large part of this function it’d be remiss not to cover it. For simplicity’s sake you might instead focus on `xas_store()` which is the actual function call which stores the entry as well as the edge case out of memory handling.

This establishes a struct `xa_state` object to retain xarray state at the specified index into the struct `address_space` page cache entry.

We take special care to account for larger folio size, incrementing the reference count for the folio being added accordingly and establishing both its index and struct `address_space` mapping.

As part of this, we set up the `xas` object to be utilised to place the folio order in the current position via `xas_set_order()`. This does not mean that any existing entry at this position is of the same order, so we must explicitly check this later on.

We then enter a loop intended to handle a high memory pressure scenarios, it loops only if the `xas_nomem()` indicates that an allocation previously failed due to an out of memory condition (for instance when splitting up an existing xarray entry).

If this is so, `xas_nomem()` attempts to allocate memory, and we loop if it succeeds, otherwise if out of memory we exit indicating this error.

In ordinary circumstances this loop will be executed in only one iteration.

The complexity here largely centres around working set logic and handling larger folio sizes. As touched upon in Section 9.2, xarray entries which span multiple indexes (i.e. larger folio size) comprise a more complicated tree structure with sibling nodes and so on.

This, combined with the fact that existing entries which have been evicted may be left behind with a value entry used for working set calculations mean we have to tread carefully here.

We start by determining the order of any existing xarray node in this position via `xa_get_order()`. if this *exceeds* the order of the folio we intend to add at this position, we must split it such that we obtain an entry of the correct order.

At this point we acquire a lock on the entry, before iterating through any existing entries via `xa_find_conflict()`. Again, we are looking for ‘shadow’ entries, entries which contain values used for working set logic.

If any entry discovered in the range in which we intend to place our folio is not a value entry, then we have a conflict and indicate an `EEXIST` error to indicate as much (we are considering a range only if the folio is of order greater than 0).

If we did, in fact, discover a conflicting entry in the range in which we hope to add our folio, we must split it if it exceeds our required order.

Ultimately the folio is stored in the xarray via `xa_store()`, the lock eliminated and we proceed with error handling if no out of memory condition arose.

Importantly, we update the struct `address_space->npages` count to indicate that we have now added a page table entry (perhaps spanning multiple pages if a folio of order greater than 0).

This is very important as this counter is used explicitly in the kernel to determine how many pages are actually present in the page cache for a given page cache entry.

9.6 Reading Folios From Disk

There are two principle means by which data is actually read from disk into the page cache—readahead, which we explore in section 9.7, and the reading of individual folios from disk (perhaps in a batched operation). We explore the latter in this section.

There are a number of callers which require us to read individual folios from disk. We’ve already examined two core means of doing so in Section 9.5 when exploring how page cache folios are generally accessed by the kernel.

When reading folios via the `read()` system call, we ultimately read folios in a batch via `filemap_get_pages()` as shown in Listing 9-44 and Section 9.5.2. Equally `filemap_fault()` requires us to read from disk if necessary as described in Section 9.4 and Listing 9-35.

In addition to this there are a number of filesystem, driver and other sub-system folio accesses which ultimately invoke `do_read_cache_folio()` (see Figure 9-9 and Listing 9-53) which reads from disk if necessary.

Regardless of where the operation originates from, the folio is ultimately read from disk via `filemap_read_folio()` which we examine in Listing 9-56.

```

2382 static int filemap_read_folio(struct file *file, filler_t filler,
2383                             struct folio *folio)
2384 {
2385     int error;
2386
2387     /*
2388      * A previous I/O error may have been due to temporary failures,
2389      * eg. multipath errors. PG_error will be set again if read_folio
2390      * fails.
2391      */
2392     folio_clear_error(folio);
2393     /* Start the actual read. The read will unlock the page. */
2394     error = filler(file, folio);
2395     if (error)
2396         return error;
2397
2398     error = folio_wait_locked_killable(folio);
2399     if (error)
2400         return error;
2401     if (folio_test_uptodate(folio))
2402         return 0;
2403     if (file)
2404         shrink_readahead_size_eio(&file->f_ra);
2405     return -EIO;
2406 }

```

Listing 9-56: mm/filemap.c: `filemap_read_folio()`

We require the folio passed into this function to already be locked, ready to be written in to.

This function is essentially a wrapper around `filler` which is of type `filler_t` and specifies a callback which performs the actual disk read, defined as shown in Listing 9-57.

```

493 typedef int filler_t(struct file *, struct folio *);

```

Listing 9-57: include/linux/pagemap.h: `filler_t`

This function accepts the folio to read from, the `struct file` object which describes the open file we read from and returns an error code or 0 if the read is successful.

Before we invoke this function, we clear the `PG_error` flag for the folio on the assumption that any such prior error was temporary.

If the filler callback returns an error we simply forward this to the caller. Otherwise we wait for the lock to be released, at which point the folio should be read into and uptodate. If this is so, we return zero to indicate no error arose.

Otherwise, we invoke `shrink_readahead_size_eio()` to scale down readahead as a result of the error, before returning `EIO` to indicate that an error arose.

Ultimately the actual read from disk is file-system specific, so we do not examine this in detail here. However ultimately whatever the implementation, the folio will be unlocked once the data has been read, and marked with the `PG_uptodate` folio flag to indicate the data has been read from disk correctly.

This is why, in this function, we can sleep on the folio being unlocked knowing that the data will be correctly read into the folio by this point (or an error indicated).

Typically, when disk I/O is complete, `page_endio()` is invoked, which we examine in Listing 9-58.

```

1630 /*
1631  * After completing I/O on a page, call this routine to update the page
1632  * flags appropriately
1633  */
1634 void page_endio(struct page *page, bool is_write, int err)
1635 {
1636     if (!is_write) {
1637         if (!err) {
1638             SetPageUptodate(page);
1639         } else {
1640             ClearPageUptodate(page);
1641             SetPageError(page);
1642         }
1643         unlock_page(page);
1644     } else {
1645         if (err) {
1646             struct address_space *mapping;
1647
1648             SetPageError(page);
1649             mapping = page_mapping(page);
1650             if (mapping)
1651                 mapping_set_error(mapping, err);
1652         }
1653         end_page_writeback(page);
1654     }
1655 }
```

Listing 9-58: `mm/filemap.c: page_endio()`

This is divided into two parts—read and write. In both cases we set the `PG_error` folio flag should an error have arisen, on write we addition-

ally set an error in the related struct `address_space` mapping object via `mapping_set_error()`.

For writeback we invoke `end_page_writeback()` which calls `folio_end_writeback()`, which wraps `__folio_end_writeback()` which we explore in the writeback chapter in Listing ??.

Pertinent here is the read case—we use the generated folio flag macro helpers to set/clear folio flags, setting `PG_uptodate` if no error arose, clearing it if one did (and setting `PG_error` if so). In both cases we unlock the folio via `unlock_page()` which wraps `folio_unlock()`. See Section 9.11 for more details on folio locking as a whole.

9.7 Readahead

Readahead is the process by which pages are read into the page cache beyond those required by a read or file-backed page fault. This is done on the basis that it is likely that the user will want to access sequential data so it's simply more efficient to read in this data ahead at the same time as requesting data preceding it.

The readahead mechanism allows for interleaving of reads by marking a folio with the `PG_readahead` flag when readahead occurs on a *major* fault (i.e. one which necessitates the blocking of the calling process until the data can be read back from disk), such that when this folio is read in a *minor* fault (i.e. one where the folio is present in the page cache but simply not yet mapped to the process), readahead is triggered once again.

The former kind of readahead is termed ‘synchronous’ readahead and the latter asynchronous. As we read through data sequentially, each time we come across a `PG_readahead` in minor fault we start the next batch of readahead and mark a subsequent folio with `PG_readahead` to trigger readahead again.

This allows for a continual process of background, asynchronous readahead operations to occur, meaning that for sequential access we can ideally avoid major page faults from occurring at all in the first place, or at least significantly reduce them.

The maximum number of pages that are readahead are, by default, determined by the block device information object (BDI), specified by struct `backing_dev_info->ra_pages`.

This parameter can be changed via a `ioctl()` call on the block device file (e.g. `/dev/sda1`) using the `BLKASET` command (and retrieved using the `BLKRGGET` command). This defaults to `VM_READAHEAD_PAGES`, set in `bdi_alloc()`.

This readahead value can also be read and set in the `sysfs` directory `/sys/devices/virtual/bdi/<maj:min>/read_ahead_kb` (where `maj` and `min` are the major and minor device numbers respectively) or more conveniently via `/sys/block/sda1/bdi/read_ahead_kb` for e.g. `/dev/sda1`. This is specified in kilobytes.

You can also use the `blockdev` CLI command via `blockdev --getra /dev/sda1` to get the readahead expressed in *logical sectors* (hardcoded 512 byte units)

for e.g. `/dev/sda1` and `blockdev --setra 512 /dev/sda1` to set the readahead to e.g. 512 sectors (256 KiB).

Readahead behaviour can further be moderated by the user who can use `madvise()` or `posix_fadvise()` to hint that a file will be accessed sequentially (via `MADV_SEQUENTIAL` or `POSIX_FADV_SEQUENTIAL` respectively), or that it will be accessed randomly and thus that readahead should *not* occur (via `MADV_RANDOM` or `POSIX_FADV_RANDOM` respectively).

This therefore allows for three different readahead ‘modes’ maintained at either the struct `vm_area_struct` (VMA) level for file-backed memory mappings (using VMA flags) or struct `file` level (using file modes), shown in Table 9-1.

Table 9-1: Readahead Modes

Mode	<code>madvise()</code> Flags	<code>posix_fadvise()</code> Flags	VMA Flags	File Mode
Random-Access	<code>MADV_RANDOM</code>	<code>POSIX_FADV_RANDOM</code>	<code>VM_RAND_READ</code>	<code>FMODE_RANDOM</code>
Normal	<code>MADV_NORMAL</code>	<code>POSIX_FADV_NORMAL</code>	-	-
Sequential	<code>MADV_SEQUENTIAL</code>	<code>POSIX_FADV_SEQ</code>	<code>VM_SEQ_READ</code>	-

The readahead itself is instigated either by the usual `read()` or page fault mechanisms for reading pages described in Section 9.5 via `filemap_get_pages()` as shown in Listing 9-44 and Section 9.5.2, or page fault as shown in `filemap_fault()` as shown in Listing 9-35 and Section 9.4, or accessed directly either by custom filesystem code or other kernel mechanism which require it.

There are two kinds of readahead:

Synchronous Readahead Performed when a *major* page fault occurs, i.e. a folio is not present in the page cache at all. At this point, we should try to retrieve the folio and initiate readahead from this point on. We read in some folios, and establish an ‘asynchronous tail’— a number of folios in addition to those we have read immediately which we will read once a user accesses one of the synchronous readahead folios. We mark this folio with the `PG_readahead` flag which asynchronous readahead uses to determine when to perform this task. We explore synchronous readahead in Section 9.7.1.

Asynchronous Readahead Performed when a *minor* page fault occurs, i.e. one that exists within the page cache but is not yet mapped by a process. When this happens, the mapped folio is checked to determine whether it has the `PG_readahead` flag set. If so, asynchronous readahead goes ahead, otherwise no readahead is performed. We explore asynchronous readahead in Section 9.7.2.

We examine the code paths readahead takes through the kernel in Figure 9-10.

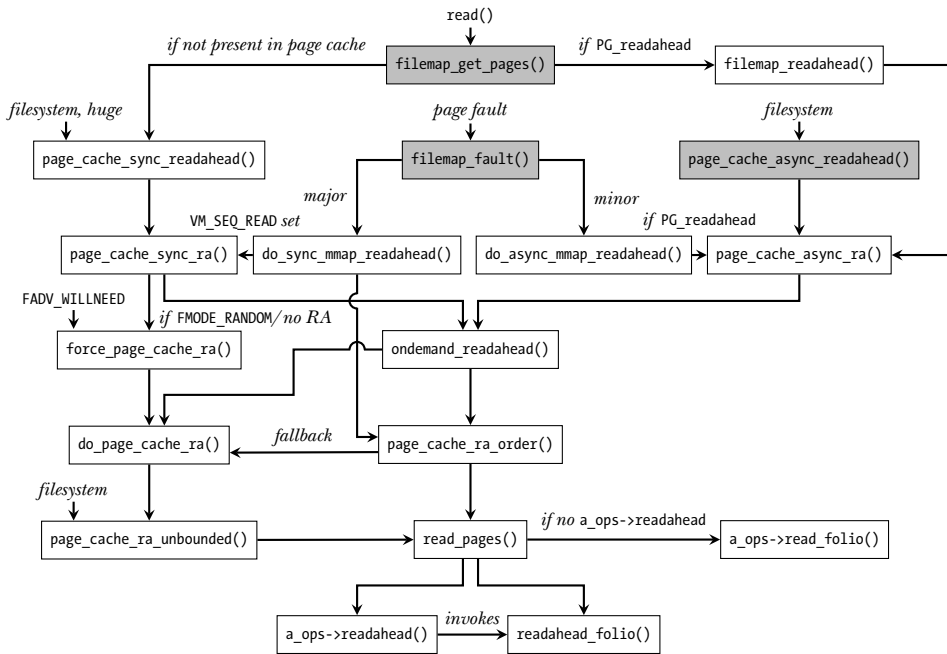


Figure 9-10: Readahead Call Paths

NOTE

There is additionally a `readahead()` system call which invokes `ksys_readahead()`, this is implemented by effectively performing a `posix_fadvise()` invocation specifying `POSIX_FADV_WILLNEED` via `vfs_fadvise()`. As Figure 9-10 shows, this ultimately invokes `force_page_cache_ra()`, called via `force_page_cache_readahead()`, called by `generic_fadvise()`.

Before we examine how readahead is performed, let's examine the means by which readahead state is perpetuated—the struct `readahead_control` object is threaded through readahead calls, as shown in Listing 9-59.

```

1152 /**
1153  * struct readahead_control - Describes a readahead request.
1154  *
1155  * A readahead request is for consecutive pages. Filesystems which
1156  * implement the ->readahead method should call readahead_page() or
1157  * readahead_page_batch() in a loop and attempt to start I/O against
1158  * each page in the request.
1159  *
1160  * Most of the fields in this struct are private and should be accessed
1161  * by the functions below.
1162  *
1163  * @file: The file, used primarily by network filesystems for authentication.
1164  *       May be NULL if invoked internally by the filesystem.
1165  * @mapping: Readahead this filesystem object.
1166  * @ra: File readahead state. May be NULL.
1167  */

```

```

1168 struct readahead_control {
1169     struct file *file;
1170     struct address_space *mapping;
1171     struct file_ra_state *ra;
1172     /* private: use the readahead_* accessors instead */
1173     pgoff_t _index;
1174     unsigned int _nr_pages;
1175     unsigned int _batch_count;
1176 };

```

Listing 9-59: include/linux/pagemap.h: struct readahead_control

Examining each field:

file The struct `file` associated with the readahead operation (if one is indeed associated with it). These objects describe open file descriptors associated with a process (even if closed by a user process after memory-mapping a file, this object will remain as it will be pinned by the `mmap()`.) If this field is present, the `ra` field is typically a pointer to struct `file->f_ra`.

mapping The struct `address_space` page cache object for which we are performing readahead.

ra A `file_ra_state` object which tracks statistics associated with the readahead operation, which we explore below in Listing 9-61. If a file object is associated with the operation, this will typically be a pointer to struct `file->f_ra`.

_index (*internal*) The page index of the first page in this readahead request, relative to the struct `address_space`. Accessible via `readahead_index()` or `readahead_pos()` if a byte offset is required.

_nr_pages (*internal*) The number of pages in this readahead request. Accessible via `readahead_count()` or `readahead_length()` if a byte count is required.

_batch_count (*internal*) The size, in subpages, of the last folio we read back. Accessible via `readahead_batch_count()`.

Typically a struct `readahead_control` object is initialised by the `DEFINE_READAHEAD()` macro, as shown in Listing 9-60.

```

#define DEFINE_READAHEAD(ractl, f, r, m, i)
    struct readahead_control ractl = {
        .file = f,
        .mapping = m,
        .ra = r,
        .index = i,
    }

```

Listing 9-60: include/linux/pagemap.h: DEFINE_READAHEAD()

This initialises the file, mapping, ra and _index fields as specified and zeroes the _nr_pages and _batch_count fields.

We examine file readahead state described by struct file_ra_state in Listing 9-61.

```

908 /**
909  * struct file_ra_state - Track a file's readahead state.
910  * @start: Where the most recent readahead started.
911  * @size: Number of pages read in the most recent readahead.
912  * @async_size: Number of pages that were/are not needed immediately
913  *             and so were/are genuinely "ahead". Start next readahead when
914  *             the first of these pages is accessed.
915  * @ra_pages: Maximum size of a readahead request, copied from the bdi.
916  * @mmap_miss: How many mmap accesses missed in the page cache.
917  * @prev_pos: The last byte in the most recent read request.
918  *
919  * When this structure is passed to ->readahead(), the "most recent"
920  * readahead means the current readahead.
921  */
922 struct file_ra_state {
923     pgoff_t start;
924     unsigned int size;
925     unsigned int async_size;
926     unsigned int ra_pages;
927     unsigned int mmap_miss;
928     loff_t prev_pos;
929 };

```

Listing 9-61: include/linux/fs.h: struct file_ra_state

Examining each field:

start The index within the struct address_space page cache object in which readahead started.

size The number of pages read so far in this readahead.

async_size The number of pages from the end of the range at which we wish to start the next readahead attempt, asynchronously. We keep track of this by tagging this folio with the PG_readahead flag (note—only if the folio was not already present in the page cache), which is checked on minor fault and serviced via the asynchronous readahead code path (see Section 9.7.2).

ra_pages Maximum number of pages to readahead, derived from struct backing_dev_info->ra_pages.

mmap_miss The number of times a page fault for a given file has led to a major fault, indicating a cache miss from readahead. We use this in do_sync_mmap_readahead() to determine whether to give up on readahead altogether (if the miss count exceeds MMAP_LOTSAMISS, hardcoded to 100 cache misses for a file, then we give up).

prev_pos The last byte from the latest read request.

This type is initialised in `file_ra_state_init()`, performed on file open, as shown in Listing 9-62.

```

134 /*
135  * Initialise a struct file's readahead state. Assumes that the caller has
136  * memset *ra to zero.
137  */
138 void
139 file_ra_state_init(struct file_ra_state *ra, struct address_space *mapping)
140 {
141     ra->ra_pages = inode_to_bdi(mapping->host)->ra_pages;
142     ra->prev_pos = -1;
143 }
```

Listing 9-62: `mm/readahead.c: file_ra_state_init()`

This function simply sets `prev_pos` to an invalid value and retrieves the `ra_pages` field from the BDI field `struct backing_dev_info->ra_pages` as previously discussed.

9.7.1 Synchronous Readahead

There is a significant difference between the reads performed by `filemap_get_pages()` originating from a `read()` which spans a buffer's worth of read and therefore multiple folios and those originating from `filemap_fault()` which comprise the faulting in of a single folio.

We start by examining the conceptually simpler of the two— a file-backed page fault handled by `filemap_fault()`. If a major fault occurs here, we perform *synchronous* readahead via `do_sync_mmap_readahead()`, which we examine in Listing 9-63 (eliding out of scope transparent huge page handling).

```

2691 /*
2692  * Synchronous readahead happens when we don't even find a page in the page
2693  * cache at all. We don't want to perform IO under the mmap sem, so if we
2694  *   have
2695  * to drop the mmap sem we return the file that was pinned in order for us to
2696  *   do
2697  * that. If we didn't pin a file then we return NULL. The file that is
2698  * returned needs to be fput()'ed when we're done with it.
2699  */
2700 static struct file *do_sync_mmap_readahead(struct vm_fault *vmf)
2701 {
2702     struct file *file = vmf->vma->vm_file;
2703     struct file_ra_state *ra = &file->f_ra;
2704     struct address_space *mapping = file->f_mapping;
2705     DEFINE_READAHEAD(ractl, file, ra, mapping, vmf->pgoff);
2706     struct file *fpin = NULL;
2707     unsigned long vm_flags = vmf->vma->vm_flags;
```

```

2706     unsigned int mmap_miss;
2707     :
2726     /* If we don't want any read-ahead, don't bother */
2727     if (vm_flags & VM_RAND_READ)
2728         return fpin;
2729     if (!ra->ra_pages)
2730         return fpin;
2731
2732     if (vm_flags & VM_SEQ_READ) {
2733         fpin = maybe_unlock_mmap_for_io(vmf, fpin);
2734         page_cache_sync_ra(&ractl, ra->ra_pages);
2735         return fpin;
2736     }
2737
2738     /* Avoid banging the cache line if not needed */
2739     mmap_miss = READ_ONCE(ra->mmap_miss);
2740     if (mmap_miss < MMAP_LOTSAMISS * 10)
2741         WRITE_ONCE(ra->mmap_miss, ++mmap_miss);
2742
2743     /*
2744      * Do we miss much more than hit in this file? If so,
2745      * stop bothering with read-ahead. It will only hurt.
2746      */
2747     if (mmap_miss > MMAP_LOTSAMISS)
2748         return fpin;
2749
2750     /*
2751      * mmap read-around
2752      */
2753     fpin = maybe_unlock_mmap_for_io(vmf, fpin);
2754     ra->start = max_t(long, 0, vmf->pgoff - ra->ra_pages / 2);
2755     ra->size = ra->ra_pages;
2756     ra->async_size = ra->ra_pages / 4;
2757     ractl._index = ra->start;
2758     page_cache_ra_order(&ractl, ra, 0);
2759     return fpin;
2760 }

```

Listing 9-63: mm/filemap.c: do_sync_mmap_readahead()

The function forms part of the faulting mechanism, so will be performed with the struct `mm_struct->mmap_lock` held. Since this can be significantly contended, we want the ability to be able to drop this lock if I/O in the form of reading data from disk is performed. If so, the file which is being read into must be *pinned*, i.e. have its reference count increased so the object is not released.

As a result, this function returns a pointer to a struct `file` object if the lock was dropped and a file object was pinned, returning the pinned file. See Section 6.2 for more details.

We see the use of the `DEFINE_READAHEAD()` macro as shown in Listing 9-60 above to define the struct `readahead_control` to be threaded through `readahead` calls, specifying that struct `file->f_ra` be used as the struct `file_ra_state` state.

The other fields placed into the struct `readahead_control` object are straightforward — the faulting file, the struct `address_space` page cache object to which the faulting folio belongs and the page offset of the folio within the page cache object specified by struct `vm_fault->pgoff`.

Immediately, we check to see if the struct `vm_area_struct` (VMA) mapping the faulting folio has the `VM_RANDOM_READ` flag set, indicating that the mapping is to be accessed in random fashion, i.e. not sequentially. In this case there is no point in proceeding with `readahead` and we simply abort.

Equally, if the BDI or user indicates that the maximum number of `readahead` pages is zero, then `readahead` cannot proceed and we also abort the operation in this instance.

9.7.1.1 Sequential Page Fault Read

We then determine which of the two modes we will utilise to perform `readahead`—the first of which is to retrieve the maximum possible number of `readahead` pages. This is performed when the struct `vm_area_struct` (VMA) flag `VM_SEQ_READ` is specified, i.e. indicating sequential `readahead` (the user having specified `MADV_SEQUENTIAL` via `madvise()`).

In this instance, we try to drop the struct `mm_struct->mmap_lock` for I/O via `maybe_unlock_mmap_for_io()` (see Listing ?? in the page fault chapter for a detailed explanation of this) before proceeding with the `readahead` operation via `page_cache_sync_ra()` which we explore below in Listing 9-65, specifying the maximum number of pages to `readahead`, struct `file_ra_state->ra_pages`.

Note that we perform `readahead` in this mode *starting at the current offset* within the file. We also do not attempt to track whether page faults are being missed, but unconditionally perform `readahead` in this instance.

When we perform `readahead` in this mode, this results in ‘on demand’ `readahead` ultimately invoking `ondemand_readahead()`, which heuristically determines how to size the `readahead`. We explore this in Listing 9-70 below.

9.7.1.2 mmap Readaround

The alternative mode does not start at the current offset within the file, rather it reads *around* the faulting folio on the assumption that perhaps the user might read backwards as well as forwards. We keep track of how well `readahead` is helping with this file using the struct `file_ra_state->mmap_miss` field.

Each time this function is invoked, it implies a `readahead` cache miss (as otherwise we’d not have encountered a major fault), so we increment the struct `file_ra_state->mmap_miss` count, unless it exceeds the `MMAP_LOTSAMISS` limit multiplied by 10 at which it is capped.

Each time a cache *hit* occurs (i.e. a *minor* fault within the file) this count is decremented. This occurs in `do_async_mmap_readahead()`.

If, at this stage, we exceed the `MMAP_LOTSAMISS` limit (hardcoded to 100 pages), we abort the operation.

Otherwise we specify `readahead` at the maximum `readahead` size, centered on the offset of the page fault within the file, clamping to the start of the file if necessary. As with the sequential case, we try to drop the `struct mm_struct->mmap_lock` if we can via `maybe_unlock_mmap_for_io()`, before proceeding with `readahead` via `page_cache_ra_order()` as described in Listing 9-77 below.

The configuration of the `struct file_ra_state` object is such as to perform `readahead` as shown in Figure 9-11, where each square represents a folio within the file, the black square indicates the faulting folio, and the grey square indicates where the `PG_readahead` flag will be set, i.e. the point at which asynchronous `readahead` will be started next.

The `readahead` is placed within the middle of the range of pages we `readahead`, with the `async` size equal to one quarter of this range (rounded down as performed with integers).

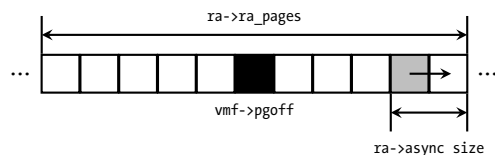


Figure 9-11: `mmap` Readaround Readahead Range

9.7.1.3 `read()` Synchronous Readahead

If `filemap_get_pages()` (see Listing 9-44 and Section 9.5.2), invoked as part of a `read()` operation, cannot retrieve any of a batch of folios as part of the operation, then it proceeds with synchronous `readahead` via `page_cache_sync_readahead()`, specifying the requested folio count of `req_count` equal to the number of folios yet to be read. We examine `page_cache_sync_readahead()` in Listing 9-64.

```

1196 /**
1197  * page_cache_sync_readahead - generic file readahead
1198  * @mapping: address_space which holds the pagecache and I/O vectors
1199  * @ra: file_ra_state which holds the readahead state
1200  * @file: Used by the filesystem for authentication.
1201  * @index: Index of first page to be read.
1202  * @req_count: Total number of pages being read by the caller.
1203  *
1204  * page_cache_sync_readahead() should be called when a cache miss happened:
1205  * it will submit the read. The readahead logic may decide to piggyback more
1206  * pages onto the read request if access patterns suggest it will improve
1207  * performance.
1208  */

```



```

1209 static inline
1210 void page_cache_sync_readahead(struct address_space *mapping,
1211                               struct file_ra_state *ra, struct file *file, pgoff_t index,
1212                               unsigned long req_count)
1213 {
1214     DEFINE_READAHEAD(ractl, file, ra, mapping, index);
1215     page_cache_sync_ra(&ractl, req_count);
1216 }

```

Listing 9-64: include/linux/pagemap.h: page_cache_sync_readahead()

This declares a struct `readahead_control` object to proceed with the readahead via `DEFINE_READAHEAD()` macro as shown in Listing 9-60 above, where the struct `file_ra_state` `ra` parameter has been set by `filemap_get_pages()` to point to struct `file->f_ra`.

This object is then passed to `page_cache_sync_ra()` which we examine in Listing 9-65 below.

9.7.1.4 Shared Synchronous Readahead

Both page fault handling invoking synchronous readahead when `VM_SEQ_READ` is specified, and `filemap_get_pages()` when synchronous readahead is performed in general, ultimately invoke `page_cache_sync_ra()` which we explore in Listing 9-65.

```

675 void page_cache_sync_ra(struct readahead_control *ractl,
676                        unsigned long req_count)
677 {
678     bool do_forced_ra = ractl->file && (ractl->file->f_mode & FMODE_RANDOM
679 );
680
681     /*
682      * Even if readahead is disabled, issue this request as readahead
683      * as we'll need it to satisfy the requested range. The forced
684      * readahead will do the right thing and limit the read to just the
685      * requested range, which we'll set to 1 page for this case.
686      */
687     if (!ractl->ra->ra_pages || blk_cgroup_congested()) {
688         if (!ractl->file)
689             return;
690         req_count = 1;
691         do_forced_ra = true;
692     }
693
694     /* be dumb */
695     if (do_forced_ra) {
696         force_page_cache_ra(ractl, req_count);
697         return;
698     }

```

```

699         ondemand_readahead(ractl, NULL, req_count);
700     }

```

Listing 9-65: mm/readahead.c: page_cache_sync_ra()

This first checks to see whether the `FMODE_RANDOM` flag has been set on the file (i.e. specified by using `posix_fadvise()` specifying `POSIX_FADV_RANDOM`).

Ordinarily you'd expect this to mean the readahead would be aborted, and until Linux 2.6.33 in 2010 at commit `0141450f66c3: readahead: introduce FMODE_RANDOM for POSIX_FADV_RANDOM` this was so.

However, it was determined that, despite the file being labelled random-access, reads performed by `read()` would simply invoke multiple reads when all could have been read in one operation.

This logic is not relevant to page fault on a file-backed mapping with the `VM_RAND_READ` flag set, as this would have been aborted before reaching here (this logic wouldn't be useful in that case, as indeed a file-backed mapping can be accessed in random fashion and isn't read a buffer size at a time, rather a folio at a time by the user).

When this edge case arises, we *force* the readahead via `force_page_cache_ra()` which we examine in Listing 9-66 below.

We also consider the case where readahead is disabled altogether—i.e. where `struct file_ra_state->ra_pages` is set to zero. This may be due to the device being unable to service readahead, an error having occurred which caused readahead to be disabled or the user specifying it should not occur. Note that we do not consider the `blk_cgroup_congested()` check here as this is cgroup-related and thus out of scope for the book.

In the case that readahead is disabled, we limit ourself to a single folio and perform a force readahead so we do at least provide the minimum required folio.

Note that any folio returned will not be read *immediately*, but rather have the I/O scheduled asynchronously. The caller must deal with a folio which is not yet uptodate (which both `read()`-mandated and page fault-mandated readahead do).

In the likely case that readahead is not forced, then we perform 'on demand' readahead via `ondemand_readahead()`, which we explore in Listing 9-70 below.

9.7.1.5 Forced Readahead

When readahead should no proceed normally due to a file having `FMODE_RANDOM` set or if readahead is disabled altogether, we force a simple form of readahead via `force_page_cache_ra()` as shown in Listing 9-66 below.

```

296  /*
297   * Chunk the readahead into 2 megabyte units, so that we don't pin too much
298   * memory at once.
299   */
300  void force_page_cache_ra(struct readahead_control *ractl,
301                          unsigned long nr_to_read)

```

```

302 {
303     struct address_space *mapping = ractl->mapping;
304     struct file_ra_state *ra = ractl->ra;
305     struct backing_dev_info *bdi = inode_to_bdi(mapping->host);
306     unsigned long max_pages, index;
307
308     if (unlikely(!mapping->a_ops->read_folio && !mapping->a_ops->readahead
309 ))
310         return;
311
312     /*
313      * If the request exceeds the readahead window, allow the read to
314      * be up to the optimal hardware IO size
315      */
316     index = readahead_index(ractl);
317     max_pages = max_t(unsigned long, bdi->io_pages, ra->ra_pages);
318     nr_to_read = min_t(unsigned long, nr_to_read, max_pages);
319     while (nr_to_read) {
320         unsigned long this_chunk = (2 * 1024 * 1024) / PAGE_SIZE;
321
322         if (this_chunk > nr_to_read)
323             this_chunk = nr_to_read;
324         ractl->_index = index;
325         do_page_cache_ra(ractl, this_chunk, 0);
326
327         index += this_chunk;
328         nr_to_read -= this_chunk;
329     }
330 }

```

Listing 9-66: mm/readahead.c: force_page_cache_ra()

This aborts if the struct `address_space->a_ops` address space operations of the struct `address_space_operations` page cache object do not specify a `readahead()` or a `read_folio()` callback (the latter is used as a fallback if the first is not specified when performing `readahead`) as `readahead` in that instance simply cannot proceed.

Otherwise, a simple form of `readahead` proceeds until all requested pages have been `readahead`.

The struct `backing_dev_info->io_pages` field determines the maximum number of pages which can be read in a single submitted I/O request and thus sets a ceiling on the maximum number of pages to be read (which is otherwise set to struct `file_ra_state->ra_pages` which defaults to struct `backing_dev_info->ra_pages`, but can be altered elsewhere).

We use this value as a ceiling on `nr_to_read` which specifies how many pages should be force-`readahead`. We then loop, reading up to a maximum of 2 MiB chunks of data at a time, using `do_page_cache_ra()` to do the heavy lifting.

9.7.2 Asynchronous Readahead

Asynchronous readahead is performed when a minor fault occurs, i.e. the page cache entry is found on page fault or read()-mandated file read, and the most recently read folio has the PG_readahead flag set.

This is set by either synchronous or asynchronous readahead logic on a folio to trigger readahead again ahead of a further major fault (thus *inter-leaving* readahead operations). Importantly, note that this flag will *not* be set on any folio that already existed within the page cache at the point at which readahead is occurring.

In the case of a read()-mandated minor fault read operation occurring in `filemap_get_pages()` (see Listing 9-44 above) `filemap_readahead()` is invoked to handle the asynchronous readahead, which we examine in Listing 9-67. This function is only called if the PG_readahead flag was set on the folio.

```

2534 static int filemap_readahead(struct kiocb *iocb, struct file *file,
2535                             struct address_space *mapping, struct folio *folio,
2536                             pgoff_t last_index)
2537 {
2538     DEFINE_READAHEAD(ractl, file, &file->f_ra, mapping, folio->index);
2539
2540     if (iocb->ki_flags & IOCB_NOIO)
2541         return -EAGAIN;
2542     page_cache_async_ra(&ractl, folio, last_index - folio->index);
2543     return 0;
2544 }
```

Listing 9-67: `mm/filemap.c: filemap_readahead()`

This defines the struct `readahead_control` object as you would expect using `DEFINE_READAHEAD()` macro (as shown in Listing 9-60 above), however note that it explicitly sets the `_index` field to `struct folio->index`, i.e. setting the operation to proceed forward at and including the folio marked by PG_readahead.

If the kernel I/O operation struct `kiocb` state variable `iocb` has the `IOCB_NOIO` flag set, we must abort with a `EAGAIN` error as of course, we are about to perform I/O.

We then defer the operation to `page_cache_async_ra()`, specifying that the entire range of data starting from this folio is to be read. We examine this function below in Listing 9-69.

NOTE

As discussed above around `filemap_get_pages()` (see Listing 9-44), if the range of folios being read is page-aligned, and we have read only part of the file, then we will read one folio after the range, i.e. inclusive of `last_index`. We may also mark this folio with PG_readahead, in which case we will immediately trigger readahead, invoking `filemap_readahead()` with `last_index` equal to `struct folio->index`, therefore passing a `req_count` of 0 to `page_cache_async_ra()`.

In the case of a page fault, `filemap_fault()` unconditionally calls `do_async_mmap_readahead()` as long as this is the first time the fault has

been tried. It does this without checking for `PG_readahead`, leaving this to `do_async_mmap_readahead()` to perform. We examine this function in Listing 9-68.

```

3032 /*
3033  * Asynchronous readahead happens when we find the page and PG_readahead,
3034  * so we want to possibly extend the readahead further. We return the file
3035  * that
3036  * was pinned if we have to drop the mmap_lock in order to do IO.
3037  */
3038 static struct file *do_async_mmap_readahead(struct vm_fault *vmf,
3039                                           struct folio *folio)
3040 {
3041     struct file *file = vmf->vma->vm_file;
3042     struct file_ra_state *ra = &file->f_ra;
3043     DEFINE_READAHEAD(ractl, file, ra, file->f_mapping, vmf->pgoff);
3044     struct file *fpin = NULL;
3045     unsigned int mmap_miss;
3046
3047     /* If we don't want any read-ahead, don't bother */
3048     if (vmf->vma->vm_flags & VM_RAND_READ || !ra->ra_pages)
3049         return fpin;
3050
3051     mmap_miss = READ_ONCE(ra->mmap_miss);
3052     if (mmap_miss)
3053         WRITE_ONCE(ra->mmap_miss, --mmap_miss);
3054
3055     if (folio_test_readahead(folio)) {
3056         fpin = maybe_unlock_mmap_for_io(vmf, fpin);
3057         page_cache_async_ra(&ractl, folio, ra->ra_pages);
3058     }
3059     return fpin;
3060 }

```

Listing 9-68: `mm/filemap.c: do_async_mmap_readahead()`

This defines the readahead object as you would expect via `DEFINE_READAHEAD()` macro (as shown in Listing 9-60 above), maintaining the same file pinning logic as described above in Section 9.7.1 for synchronous readahead.

If the `struct vm_area_struct` (VMA) mapping specifies the `VM_RAND_READ` flag or readahead is disabled, there is nothing to do and we simply exit.

Otherwise, we update the `struct file_ra_state->mmap_miss` field, reducing the miss count even if this folio does not specify `PG_readahead`, as the fact a minor fault occurred at all implies a cache hit.

Finally, if `PG_readahead` is specified, we go ahead and invoke asynchronous readahead, first performing the `struct mm_struct->mmap_lock` dropping dance via `maybe_unlock_mmap_for_io()` before performing the readahead via `page_cache_async_ra()` which we examine in Listing 9-69 below.

9.7.2.1 Shared Asynchronous Readahead

Asynchronous readahead for both `read()`-mandated and page fault readahead is ultimately deferred to `page_cache_async_ra()` which we examine in Listing 9-69 below (eliding out of scope `cgroup` logic).

```

703 void page_cache_async_ra(struct readahead_control *ractl,
704                          struct folio *folio, unsigned long req_count)
705 {
706     /* no readahead */
707     if (!ractl->ra->ra_pages)
708         return;
709
710     /*
711      * Same bit is used for PG_readahead and PG_reclaim.
712      */
713     if (folio_test_writeback(folio))
714         return;
715
716     folio_clear_readahead(folio);
717     :
721     ondemand_readahead(ractl, folio, req_count);
722 }
```

Listing 9-69: mm/readahead.c: `page_cache_async_ra()`

If readahead is disabled (indicated by the struct `file_ra_state->ra_pages` field being set to zero), then there is nothing to do here and we abort.

Otherwise we are careful about writeback—`PG_writeback` implies that the `PG_reclaim` flag might be set, which `PG_readahead` overloads and so in this instance, to avoid a bug arising from this being unintentionally set, we abort the operation.

Otherwise, we clear the `PG_readahead` flag as we are now about to process it and we do not want a competing asynchronous readahead operation to begin.

Finally, we defer the actual readahead to on-demand readahead via `ondemand_readahead()`, which we explore in Listing 9-70 below.

9.7.3 On-Demand Readahead

When asynchronous readahead occurs, or either synchronous `read()`-mandated readahead (via `filemap_get_pages()` shown in Listing 9-44 above) or page fault-mandated synchronous readahead in the presence of the `VM_SEQ_READ` flag occurs, this results in the kernel performing ‘on-demand’ readahead.

In this instance, the kernel is permitted to use heuristics to determine how to adjust the readahead as it proceeds rather than it being mandated by a caller.

As this function is rather complicated and relies upon heuristics, we will briefly examine an example of how readahead behaves for a series of se-

quential `read()` operations of 1 page each of a very large file not yet in the page cache, with a maximum readahead count of 32 pages, as shown in Figure 9-12.

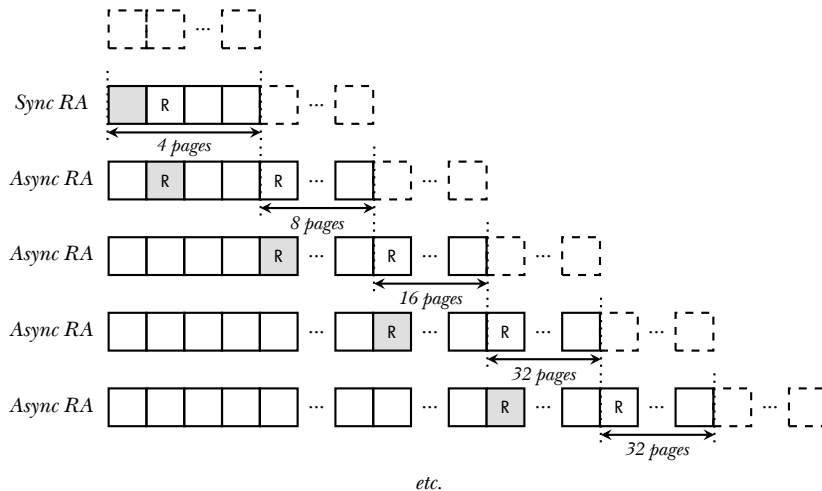


Figure 9-12: Example On-Demand `read()` Readahead for Sequential Single-Page Reads

Note that dashed boxes denote folios not yet in the page cache, the shaded boxes denote the folio currently being read by `filemap_get_pages()` (see Listing 9-44) and R denotes the folio tagged with the `PG_readahead` flag.

As we will examine shortly, the initial synchronous readahead is scaled to 4 pages given a maximum number of readahead pages of 32 pages (which the typical default) and reading a single page at a time. This is determined by `get_init_ra_size()` (see Listing 9-76 below).

The first folio tagged with `PG_readahead` is offset by the request size (in this case 1 page). Since `filemap_get_pages()` reads an additional page past the last if page-aligned (see discussion around Listing 9-44 for details), we will always trigger the first asynchronous readahead immediately after the synchronous one.

This first asynchronous readahead then doubles the readahead size to 8 pages, as mandated by `get_next_ra_size()` (see Listing 9-72).

From here on in the process continues until we reach the maximum readahead size of 32 pages. Note that we enter into a pattern of kicking off asynchronous readahead at the start of processing the minor faults associated with the next readahead block of folios, so are continually avoiding major faults on a sequential read, as intended.

Even if the user closes the file descriptor and opens another reading the same file (and thus we no longer have access to the struct `file_ra_state` object in struct `file->f_ra`), we are able to recover this state on encountering the next `PG_readahead` via `page_cache_next_miss()` (see this in context in Listing 9-71 below).

In fact, even in the case of synchronous readahead that is not the first that has occurred in the file, we attempt to ascertain any evidence of

context-based sequential reads via `try_context_readahead()` (see Listing 9-73 below).

In all cases the on-demand readahead does its best to read in pages if there's any evidence at all of sequential reads, given the significant performance benefits of having this data in memory rather than having to wait for it to be read from the disk.

As we explore in the reclaim chapter, the kernel very quickly reads in page cache data but will equally quickly reclaim it under memory pressure, so it is entirely fine to be profligate here.

Let's examine this in detail in the function which performs on-demand readahead, `ondemand_readahead()`, which we examine starting with Listing 9-70 below.

```

553 /*
554  * A minimal readahead algorithm for trivial sequential/random reads.
555  */
556 static void ondemand_readahead(struct readahead_control *ractl,
557                               struct folio *folio, unsigned long req_size)
558 {
559     struct backing_dev_info *bdi = inode_to_bdi(ractl->mapping->host);
560     struct file_ra_state *ra = ractl->ra;
561     unsigned long max_pages = ra->ra_pages;
562     unsigned long add_pages;
563     pgoff_t index = readahead_index(ractl);
564     pgoff_t expected, prev_index;
565     unsigned int order = folio ? folio_order(folio) : 0;
566
567     /*
568      * If the request exceeds the readahead window, allow the read to
569      * be up to the optimal hardware IO size
570      */
571     if (req_size > max_pages && bdi->io_pages > max_pages)
572         max_pages = min(req_size, bdi->io_pages);

```

Listing 9-70: `mm/readahead.c: ondemand_readahead()` prelude

We start by obtaining the index from which we should proceed from the `struct readahead_control` object and the folio order if a folio is supplied (if a folio is supplied, this implies asynchronous readahead). We also place the maximum pages that can be readahead, derived from `struct file_ra_state->ra_pages` in `max_pages`.

We then check to ensure that the requested readahead size exceeds the maximum pages. If it does this not taken as a hard limit, rather the `struct backing_dev_info->io_pages` field indicating the maximum optimal number of pages to retrieve in a single I/O operation is used as maximum page count.

Once we have established some initial state, we start trying to heuristically determine where we are with the readahead and how to proceed, which we explore in Listing 9-71.


```

574      /*
575       * start of file
576       */
577      if (!index)
578          goto initial_readahead;
579
580      /*
581       * It's the expected callback index, assume sequential access.
582       * Ramp up sizes, and push forward the readahead window.
583       */
584      expected = round_up(ra->start + ra->size - ra->async_size,
585                          1UL << order);
586      if (index == expected || index == (ra->start + ra->size)) {
587          ra->start += ra->size;
588          ra->size = get_next_ra_size(ra, max_pages);
589          ra->async_size = ra->size;
590          goto readit;
591      }
592
593      /*
594       * Hit a marked folio without valid readahead state.
595       * E.g. interleaved reads.
596       * Query the pagecache for async_size, which normally equals to
597       * readahead size. Ramp it up and use it as the new readahead size.
598       */
599      if (folio) {
600          pgoff_t start;
601
602          rcu_read_lock();
603          start = page_cache_next_miss(ractl->mapping, index + 1,
604                                      max_pages);
605          rcu_read_unlock();
606
607          if (!start || start - index > max_pages)
608              return;
609
610          ra->start = start;
611          ra->size = start - index;          /* old async_size */
612          ra->size += req_size;
613          ra->size = get_next_ra_size(ra, max_pages);
614          ra->async_size = ra->size;
615          goto readit;
616      }
617
618      /*
619       * oversize read
620       */

```

```

621     if (req_size > max_pages)
622         goto initial_readahead;
623
624     /*
625      * sequential cache miss
626      * trivial case: (index - prev_index) == 1
627      * unaligned reads: (index - prev_index) == 0
628      */
629     prev_index = (unsigned long long)ra->prev_pos >> PAGE_SHIFT;
630     if (index - prev_index <= 1UL)
631         goto initial_readahead;
632
633     /*
634      * Query the page cache and look for the traces(cached history pages)
635      * that a sequential stream would leave behind.
636      */
637     if (try_context_readahead(ractl->mapping, ra, index, req_size,
638                             max_pages))
639         goto readit;

```

Listing 9-71: mm/readahead.c: `ondemand_readahead()` heuristics

We have special handling for the case where we are at the beginning of the file under readahead, so if the readahead index is 0, we proceed to this immediately.

Next we consider the asynchronous readahead case— we expect asynchronous readahead to kick in precisely struct `file_ra_state->async_size` pages prior to the end of the range (see Figure 9-11 for instance). In the instance we have an asynchronous readahead hit and prepare for the next readahead.

In this instance, we proceed with readahead, offsetting the next readahead to proceed immediately after the folios the last readahead populated, setting the asynchronous size to be equal to the readahead size (i.e. placing it at the start of the next readahead batch) and obtaining the new readahead size via `get_next_ra_size()`, which we examine in Listing 9-72.

```

351  /*
352   * Get the previous window size, ramp it up, and
353   * return it as the new window size.
354   */
355  static unsigned long get_next_ra_size(struct file_ra_state *ra,
356                                       unsigned long max)
357  {
358      unsigned long cur = ra->size;
359
360      if (cur < max / 16)
361          return 4 * cur;
362      if (cur <= max / 2)
363          return 2 * cur;

```

```

364     return max;
365 }

```

Listing 9-72: mm/readahead.c: `get_next_ra_size()`

This is entirely heuristic and scales the readahead by a factor of 4 if the size is less than 1/16th of the maximum possible (meaning at most 2 scale up operations are necessary to surpass this), doubles it if it is less than or equal to half of the maximum readahead size and set to the maximum readahead size otherwise.

Note that, since we will now proceed with this readahead, all readahead pages should be in place when the folio at the beginning of the next block which will have `PG_readahead` set is encountered, we should encounter no major faults.

If we do encounter a major fault (perhaps by accessing a folio outside of the readahead range, perhaps due to an interleaved read), the `struct file->f_ra` will be reset.

In this case, we might encounter a `PG_readahead` flag but this occurring in an unexpected position. In this instance, the `folio` parameter will be set by the calling asynchronous readahead handler so we can explicitly check for it.

Here we attempt to heuristically determine where the next block should be located via `page_cache_next_miss()`, which scans the page cache looking for a non-present page from this point on.

If none is found, or it is located beyond the maximum number of readahead pages we abort.

Otherwise we setup the next readahead using this heuristically obtained information in precisely the same way as if it were discovered where it was expected.

Next we consider the case where the requester requested more than the maximum pages (even taking into account the increase that is applied to `struct backing_dev_info->io_pages`), which we treat the same as if this were initial readahead which we explore below.

The `struct file_ra_state->prev_pos` field is updated in `filemap_read()` on each read to indicate the position of the last read()-mandated read. If this is was 0 or 1 pages ago, we again treat this as if it were initial readahead as this implies a sequential cache miss.

This might be caused by a series of `read()` invocations of a single page which amount to sequential reads and thus imply that readahead on this basis makes sense here.

Finally we become fully heuristic and go to great lengths to determine whether traces exist that imply a sequential read is proceeding, via `try_context_readahead()` which we examine in Listing 9-73 below.

```

424 static int try_context_readahead(struct address_space *mapping,
425                                struct file_ra_state *ra,
426                                pgoff_t index,
427                                unsigned long req_size,
428                                unsigned long max)
429 {

```

```

430     pgoff_t size;
431
432     size = count_history_pages(mapping, index, max);
433
434     /*
435      * not enough history pages:
436      * it could be a random read
437      */
438     if (size <= req_size)
439         return 0;
440
441     /*
442      * starts from beginning of file:
443      * it is a strong indication of long-run stream (or whole-file-read)
444      */
445     if (size >= index)
446         size *= 2;
447
448     ra->start = index;
449     ra->size = min(size + req_size, max);
450     ra->async_size = 1;
451
452     return 1;
453 }

```

Listing 9-73: mm/readahead.c: try_context_readahead()

This utilises `count_history_pages()` (which we examine in Listing 9-74 below) to determine how many pages from `index - 1` up to `index - max` are present in the page cache as an estimate on contiguously cached pages indicating a sequential read pattern.

If the size is less than that requested, we accept that this is not a sequential read pattern.

Otherwise, we accept that it is. If the cached page count in `size` equals or exceeds the index, i.e. offset into the file, this indicates that entries from the start of the file to here have been stored in the page cache, which heavily implies that this is a read of the whole file, at which point the size is doubled to increase the readahead we are about to establish.

We then set struct `file_ra_state` fields such that we start at the current index, reading the combination of the size we have established was previously contiguously read in (possibly doubled) and the requested size (up to a maximum of the maximum readahead size), and establish that we will provoke the next asynchronous readahead at the end of this block.

Examining `count_history_pages()` in Listing 9-74:

```

406  /*
407   * Count contiguously cached pages from @index-1 to @index-@max,
408   * this count is a conservative estimation of
409   *     - length of the sequential read sequence, or

```

```

410  *      - thrashing threshold in memory tight systems
411  */
412  static pgoff_t count_history_pages(struct address_space *mapping,
413                                   pgoff_t index, unsigned long max)
414  {
415      pgoff_t head;
416
417      rcu_read_lock();
418      head = page_cache_prev_miss(mapping, index - 1, max);
419      rcu_read_unlock();
420
421      return index - 1 - head;
422  }

```

Listing 9-74: mm/readahead.c: count_history_pages()

This uses `page_cache_prev_miss()` to scan backwards through the xarray describing the struct `address_space` page cache object, returning the index of the first entry, starting from `index - 1` which is not present in the cache.

Subtracting this from `index - 1` provides the contiguous range.

Finally, once all of these heuristics have been applied, we proceed with actually executing `readahead`, as explored in Listing 9-75 below.

```

641  /*
642  * standalone, small random read
643  * Read as is, and do not pollute the readahead state.
644  */
645  do_page_cache_ra(ractl, req_size, 0);
646  return;
647
648  initial_readahead:
649      ra->start = index;
650      ra->size = get_init_ra_size(req_size, max_pages);
651      ra->async_size = ra->size > req_size ? ra->size - req_size : ra->size;
652
653  readit:
654  /*
655   * Will this read hit the readahead marker made by itself?
656   * If so, trigger the readahead marker hit now, and merge
657   * the resulted next readahead window into the current one.
658   * Take care of maximum IO pages as above.
659   */
660  if (index == ra->start && ra->size == ra->async_size) {
661      add_pages = get_next_ra_size(ra, max_pages);
662      if (ra->size + add_pages <= max_pages) {
663          ra->async_size = add_pages;
664          ra->size += add_pages;
665      } else {
666          ra->size = max_pages;

```

```

667             ra->async_size = max_pages >> 1;
668         }
669     }
670
671     ractl->_index = ra->start;
672     page_cache_ra_order(ractl, ra, order);
673 }

```

Listing 9-75: mm/readahead.c: `ondemand_readahead()` `readahead` invocations

In the instance that no sequential readahead could be detected, then we treat this like a random read, and defer it to `do_page_cache_ra()` to perform the readahead, specifying zero look ahead and not modifying the readahead state at all.

In the case that initial readahead is to be performed, we reset the start to the current index, determine the size to readahead via the heuristic `get_init_ra_size()` which we examine in Listing 9-76 below.

We set the position at which to perform the next asynchronous readahead operation (as mandated by `struct file_ra_state->async_size`) to be equal to either the a number of pages equal to the newly determined size from the end, should this size exceed that requested, or if not, offset by the requested size.

```

331 /*
332  * Set the initial window size, round to next power of 2 and square
333  * for small size, x 4 for medium, and x 2 for large
334  * for 128k (32 page) max ra
335  * 1-2 page = 16k, 3-4 page 32k, 5-8 page = 64k, > 8 page = 128k initial
336  */
337 static unsigned long get_init_ra_size(unsigned long size, unsigned long max)
338 {
339     unsigned long newsize = roundup_pow_of_two(size);
340
341     if (newsize <= max / 32)
342         newsize = newsize * 4;
343     else if (newsize <= max / 4)
344         newsize = newsize * 2;
345     else
346         newsize = max;
347
348     return newsize;
349 }

```

Listing 9-76: mm/readahead.c: `get_init_ra_size()`

This heuristically determines the size to be equal to the requested size rounded up to the nearest power of two and squared, then multiplied by 4 should it be less than or equal to 1/32 of the maximum readahead size, multiplied by 2 should be less than or equal to 1/4 of the maximum readahead size or otherwise simply set to the maximum readahead size.

This then falls through to the `readit` label where sequential readahead is performed, the same logic invoked by other logic above which calls into this.

There is an additional check to see if the specified we can merge the next asynchronous readahead point, doing so if we can (note that we place the next `PG_readahead` either at the start of the next readahead block or halfway through it if we are capped at the maximum readahead pages).

Finally, we set the `struct readahead_control` readahead index and defer the actual readahead itself to the `page_cache_ra_order()` function which we explore in Listing 9-77 below.

9.7.4 Common Readahead Code

There are two functions which are ultimately invoked to perform the readahead into the page cache, `page_cache_ra_order()` which we examine in Listing 9-77 below and `do_page_cache_ra()` which we examine in Listing 9-78.

```

490 void page_cache_ra_order(struct readahead_control *ractl,
491                          struct file_ra_state *ra, unsigned int new_order)
492 {
493     struct address_space *mapping = ractl->mapping;
494     pgoff_t index = readahead_index(ractl);
495     pgoff_t limit = (i_size_read(mapping->host) - 1) >> PAGE_SHIFT;
496     pgoff_t mark = index + ra->size - ra->async_size;
497     int err = 0;
498     gfp_t gfp = readahead_gfp_mask(mapping);
499
500     if (!mapping_large_folio_support(mapping) || ra->size < 4)
501         goto fallback;
502
503     limit = min(limit, index + ra->size - 1);
504
505     if (new_order < MAX_PAGECACHE_ORDER) {
506         new_order += 2;
507         if (new_order > MAX_PAGECACHE_ORDER)
508             new_order = MAX_PAGECACHE_ORDER;
509         while ((1 << new_order) > ra->size)
510             new_order--;
511     }
512
513     filemap_invalidate_lock_shared(mapping);
514     while (index <= limit) {
515         unsigned int order = new_order;
516
517         /* Align with smaller pages if needed */
518         if (index & ((1UL << order) - 1)) {
519             order = __ffs(index);
520             if (order == 1)
521                 order = 0;

```

```

522     }
523     /* Don't allocate pages past EOF */
524     while (index + (1UL << order) - 1 > limit) {
525         if (--order == 1)
526             order = 0;
527     }
528     err = ra_alloc_folio(ractl, index, mark, order, gfp);
529     if (err)
530         break;
531     index += 1UL << order;
532 }
533
534 if (index > limit) {
535     ra->size += index - limit - 1;
536     ra->async_size += index - limit - 1;
537 }
538
539 read_pages(ractl);
540 filemap_invalidate_unlock_shared(mapping);
541
542 /*
543  * If there were already pages in the page cache, then we may have
544  * left some gaps. Let the regular readahead code take care of this
545  * situation.
546  */
547 if (!err)
548     return;
549 fallback:
550     do_page_cache_ra(ractl, ra->size, ra->async_size);
551 }

```

Listing 9-77: mm/readahead.c: page_cache_ra_order()

This function uses `mapping_large_folio_support()` to determine whether to proceed or revert to fallback. This is out of scope for the book and only the case if the file system has enabled large folio support, which only a minority do. In any case, it is only relevant when transparent huge page functionality is available which renders the function out of scope for the book, and it is therefore included here for informational purposes only.

We instead note that the fallback case as well as forced readahead utilise `do_page_cache_ra()` to perform readahead, which we examine in Listing 9-78.

```

269 /*
270  * do_page_cache_ra() actually reads a chunk of disk. It allocates
271  * the pages first, then submits them for I/O. This avoids the very bad
272  * behaviour which would occur if page allocations are causing VM writeback.
273  * We really don't want to intermingle reads and writes like that.
274  */
275 static void do_page_cache_ra(struct readahead_control *ractl,

```



```

276             unsigned long nr_to_read, unsigned long lookahead_size)
277 {
278     struct inode *inode = ractl->mapping->host;
279     unsigned long index = readahead_index(ractl);
280     loff_t isize = i_size_read(inode);
281     pgoff_t end_index;    /* The last page we want to read */
282
283     if (isize == 0)
284         return;
285
286     end_index = (isize - 1) >> PAGE_SHIFT;
287     if (index > end_index)
288         return;
289     /* Don't read past the page containing the last byte of the file */
290     if (nr_to_read > end_index - index)
291         nr_to_read = end_index - index + 1;
292
293     page_cache_ra_unbounded(ractl, nr_to_read, lookahead_size);
294 }

```

Listing 9-78: mm/readahead.c: do_page_cache_ra()

The `lookahead_size` parameter will be set to `struct file_ra_state->async_size` if falling back from `page_cache_ra_order()`, otherwise it will be set to zero either when performing random readaheads in `ondemand_readahead()` (see Listing 9-75), or a forced readahead via `force_page_cache_ra()` (see Listing ??).

This starts by checking whether the file is now truncated, i.e. possessing a zero size. If so, we abort. Equally, if the starting index, set as `struct readahead_control->_index`, exceeds the last page of data in the file given its size, we also abort.

Finally the number of pages to read is capped by the file size, before deferring the actual readahead operation to `page_cache_ra_unbounded()`, which we examine in Listing 9-79 below.

```

186 /**
187  * page_cache_ra_unbounded - Start unchecked readahead.
188  * @ractl: Readahead control.
189  * @nr_to_read: The number of pages to read.
190  * @lookahead_size: Where to start the next readahead.
191  *
192  * This function is for filesystems to call when they want to start
193  * readahead beyond a file's stated i_size. This is almost certainly
194  * not the function you want to call. Use page_cache_async_readahead()
195  * or page_cache_sync_readahead() instead.
196  *
197  * Context: File is referenced by caller. Mutexes may be held by caller.
198  * May sleep, but will not reenter filesystem to reclaim memory.
199  */

```

```

200 void page_cache_ra_unbounded(struct readahead_control *ractl,
201                             unsigned long nr_to_read, unsigned long lookahead_size)
202 {
203     struct address_space *mapping = ractl->mapping;
204     unsigned long index = readahead_index(ractl);
205     gfp_t gfp_mask = readahead_gfp_mask(mapping);
206     unsigned long i;
207
208     /*
209      * Partway through the readahead operation, we will have added
210      * locked pages to the page cache, but will not yet have submitted
211      * them for I/O. Adding another page may need to allocate memory,
212      * which can trigger memory reclaim. Telling the VM we're in
213      * the middle of a filesystem operation will cause it to not
214      * touch file-backed pages, preventing a deadlock. Most (all?)
215      * filesystems already specify __GFP_NOFS in their mapping's
216      * gfp_mask, but let's be explicit here.
217      */
218     unsigned int nofs = memalloc_nofs_save();
219
220     filemap_invalidate_lock_shared(mapping);
221     /*
222      * Preallocate as many pages as we will need.
223      */
224     for (i = 0; i < nr_to_read; i++) {
225         struct folio *folio = xa_load(&mapping->i_pages, index + i);
226
227         if (folio && !xa_is_value(folio)) {
228             /*
229              * Page already present? Kick off the current batch
230              * of contiguous pages before continuing with the
231              * next batch. This page may be the one we would
232              * have intended to mark as Readahead, but we don't
233              * have a stable reference to this page, and it's
234              * not worth getting one just for that.
235              */
236             read_pages(ractl);
237             ractl->_index++;
238             i = ractl->_index + ractl->_nr_pages - index - 1;
239             continue;
240         }
241
242         folio = filemap_alloc_folio(gfp_mask, 0);
243         if (!folio)
244             break;
245         if (filemap_add_folio(mapping, folio, index + i,
246                             gfp_mask) < 0) {

```

```

247         folio_put(folio);
248         read_pages(ractl);
249         ractl->_index++;
250         i = ractl->_index + ractl->_nr_pages - index - 1;
251         continue;
252     }
253     if (i == nr_to_read - lookahead_size)
254         folio_set_readahead(folio);
255     ractl->_nr_pages++;
256 }
257
258 /*
259  * Now start the IO. We ignore I/O errors - if the folio is not
260  * uptodate then the caller will launch read_folio again, and
261  * will then handle the error.
262  */
263 read_pages(ractl);
264 filemap_invalidate_unlock_shared(mapping);
265 memalloc_nofs_restore(nofs);
266 }

```

Listing 9-79: mm/readahead.c: page_cache_ra_unbounded()

This function performs the meat and potatoes of allocating and placing folios into the page cache, then triggering the underlying block device to read data into them.

We establish the GFP flags to use when performing physical memory allocations using `readahead_gfp_mask()`, which uses `mapping_gfp_mask()` to determine those flags appropriate for the struct `address_space` mapping as well as specifying `__GFP_NORETRY` and `__GFP_NOWARN` to prevent allocation retry or warning on failure of allocation here. See Section 2.6 in the Physical Memory chapter for more details.

We also utilise `memalloc_nofs_save()` to implicitly prevent `__GFP_FS` from being set on allocations here to avoid deadlocking. Again, see the Physical Memory chapter for more details on this.

We acquire the invalidation lock over the operation via `filemap_invalidate_lock_shared()` (releasing it via `filemap_invalidate_unlock_shared()` afterwards) to prevent a competing operation performing any kind of possible invalidation interfering.

We then loop through each page in the xarray associated with the page cache struct `address_space` object, first checking to see if somehow we're in a position where the folio is actually already present in the page cache.

If so, we go ahead and perform I/O for these pages via `read_pages()` (see Listing 9-80 below), updating the index count accordingly to account for the pages read before continuing with the loop.

Otherwise, we allocate a folio of order-0 (higher order folios are handled by `do_page_cache_ra()`, whose logic is out of the scope of this book) via `filemap_alloc_folio()`, before adding it to the page cache via

`filemap_add_folio()` (see Listing 9-54 in Section for more details). Note that this function returns locked folios ready to have data written into them.

If adding the folio to the page cache fails, we drop its reference count and attempt to read data into the pages via `read_pages()` in the same fashion we did if the page was already present and continue the loop past whatever pages were read.

Next, we see the code which sets the `PG_readahead` flag, if the index is equal to the size of the readhead to perform less the lookahead size.

NOTE

For completeness we note that this flag is also set for the higher order case in `ra_alloc_folio()` as invoked by `page_cache_ra_order()` –again this is out of scope for the book so we explore this no further.

After this is complete, we kick off the physical readahead using `read_pages()` as shown in Listing 9-80 below.

9.7.5 Physical Readahead

The actual physical act of reading data from disk for the purposes of readahead is performed by `read_pages()`, which we examine in Listing 9-80 (eliding out of scope block device plug logic and an extraneous bug check).

```

146 static void read_pages(struct readahead_control *rac)
147 {
148     const struct address_space_operations *aops = rac->mapping->a_ops;
149     struct folio *folio;
150     :
151
152     if (!readahead_count(rac))
153         return;
154     :
155
156     if (aops->readahead) {
157         aops->readahead(rac);
158         /*
159          * Clean up the remaining folios. The sizes in ->ra
160          * may be used to size the next readahead, so make sure
161          * they accurately reflect what happened.
162          */
163         while ((folio = readahead_folio(rac)) != NULL) {
164             unsigned long nr = folio_nr_pages(folio);
165
166             folio_get(folio);
167             rac->ra->size -= nr;
168             if (rac->ra->async_size >= nr) {
169                 rac->ra->async_size -= nr;
170                 filemap_remove_folio(folio);
171             }
172             folio_unlock(folio);
173             folio_put(folio);
174         }
175     }

```

```

175         }
176     } else {
177         while ((folio = readahead_folio(rac)) != NULL)
178             aops->read_folio(rac->file, folio);
179     }
180     :
181 }

```

Listing 9-80: mm/readahead.c: read_pages()

We start by checking struct `readahead_control->_nr_pages` via `readahead_count()`, if this is zero then there is nothing to do and we abort.

Then we determine how to proceed—if the file system provides struct `address_space->a_ops` address space operations of the struct `address_space_operations` type specifying a `readahead()` handler, then we use this to cause the file system to perform the actual readahead, reading into each folio.

If none is provided, then we fallback to using `readahead_folio()` to retrieve each folio in the range and then invoke individual struct `address_space_operations->read_folio` calls to perform the readahead manually.

If the file system is unable to readahead all of the folios, we manually iterate through those that remain and remove them from the page cache altogether via `filemap_remove_folio()` (see Listing 9-88 in Section 9.9 for more details).

We make sure in this instance to update the struct `file_ra_state->async_size` value to remain valid within the readahead range.

The file system-specified struct `address_space_operations->read_folio` handler typically uses `readahead_folio()` to retrieve the folio in the first instance, as does our fallback handler, so we examine this in Listing 9-81.

```

1277 /**
1278  * readahead_folio - Get the next folio to read.
1279  * @ractl: The current readahead request.
1280  *
1281  * Context: The folio is locked. The caller should unlock the folio once
1282  * all I/O to that folio has completed.
1283  * Return: A pointer to the next folio, or %NULL if we are done.
1284  */
1285 static inline struct folio *readahead_folio(struct readahead_control *ractl)
1286 {
1287     struct folio *folio = __readahead_folio(ractl);
1288
1289     if (folio)
1290         folio_put(folio);
1291     return folio;
1292 }

```

Listing 9-81: include/linux/pagemap.h: readahead_folio()

This simply defers the heavy lifting to `__readahead_folio()`, dropping the folio's reference count to unpin it once done. We examine `__readahead_folio()` in Listing 9-82 (eliding some distracting bug checks).

```

1241 static inline struct folio *__readahead_folio(struct readahead_control *ractl)
1242 {
1243     struct folio *folio;
1244     :
1246     ractl->_nr_pages -= ractl->_batch_count;
1247     ractl->_index += ractl->_batch_count;
1248
1249     if (!ractl->_nr_pages) {
1250         ractl->_batch_count = 0;
1251         return NULL;
1252     }
1253
1254     folio = xa_load(&ractl->mapping->i_pages, ractl->_index);
1255     :
1256     ractl->_batch_count = folio_nr_pages(folio);
1257
1258     return folio;
1259 }
```

Listing 9-82: include/linux/pagemap.h: __readahead_folio()

Everything is performed at a granularity determined by the `struct readahead_control->_batch_count` field, which is set in each instance by the previous folio's size.

This is because the readahead state is set to point to the folio we are now loading and thus on the next iteration we must know the previous folio's size in order to skip past it to the next one.

This simply uses `xa_load()` to retrieve the folio from the page cache at the current readahead index, returning `NULL` if we have exceeded the size of the readahead.

This therefore allows this function to be used in the loops we have seen thus far to iterate through each folio in a readahead range.

9.8 Fault-Around

In order to minimise the amount of time spent faulting memory in, the page faulting mechanism will perform 'fault-around' on *read* fault in `do_read_fault()`.

Importantly, this only performs *minor* faults, i.e. faulting in memory that is already present in the page cache, in a batch, skipping anything that requires a major fault.

We discuss this faulting mechanism in section 6.5 in the page fault chapter, however let's review `do_fault_around()` (previously shown in listing 6-28) repeated here for convenience in Listing 9-83.

```

4443  /*
4444  * do_fault_around() tries to map few pages around the fault address. The hope
4445  * is that the pages will be needed soon and this will lower the number of
4446  * faults to handle.
4447  *
4448  * It uses vm_ops->map_pages() to map the pages, which skips the page if it's
4449  * not ready to be mapped: not up-to-date, locked, etc.
4450  *
4451  * This function doesn't cross the VMA boundaries, in order to call map_pages
4452  *   ()
4453  * only once.
4454  *
4455  * fault_around_bytes defines how many bytes we'll try to map.
4456  * do_fault_around() expects it to be set to a power of two less than or equal
4457  * to PTRS_PER_PTE.
4458  *
4459  * The virtual address of the area that we map is naturally aligned to
4460  * fault_around_bytes rounded down to the machine page size
4461  * (and therefore to page order). This way it's easier to guarantee
4462  * that we don't cross page table boundaries.
4463  */
4464  static vm_fault_t do_fault_around(struct vm_fault *vmf)
4465  {
4466      unsigned long address = vmf->address, nr_pages, mask;
4467      pgoff_t start_pgoff = vmf->pgoff;
4468      pgoff_t end_pgoff;
4469      int off;
4470
4471      nr_pages = READ_ONCE(fault_around_bytes) >> PAGE_SHIFT;
4472      mask = ~(nr_pages * PAGE_SIZE - 1) & PAGE_MASK;
4473
4474      address = max(address & mask, vmf->vma->vm_start);
4475      off = ((vmf->address - address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1);
4476      start_pgoff -= off;
4477
4478      /*
4479       * end_pgoff is either the end of the page table, the end of
4480       * the vma or nr_pages from start_pgoff, depending what is nearest.
4481       */
4482      end_pgoff = start_pgoff -
4483                  ((address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1)) +
4484                  PTRS_PER_PTE - 1;
4485      end_pgoff = min3(end_pgoff, vma_pages(vmf->vma) + vmf->vma->vm_pgoff -
4486                      1,

```

```

4485         start_pgoff + nr_pages - 1);
4486
4487     if (pmd_none(*vmf->pmd)) {
4488         vmf->prealloc_pte = pte_alloc_one(vmf->vma->vm_mm);
4489         if (!vmf->prealloc_pte)
4490             return VM_FAULT_OOM;
4491     }
4492
4493     return vmf->vma->vm_ops->map_pages(vmf, start_pgoff, end_pgoff);
4494 }

```

Listing 9-83: mm/memory.c: do_fault_around()

Broadly speaking this reads up to `fault_around_bytes` (configurable by the `/sys/kernel/debug/fault_around_bytes` sysfs tunable) of data to fault in around the faulting address, if the file system supports it.

This defaults to 64 KiB or 16 pages if they are of 4 KiB size (as in x86-64).

We align the start of the range to the fault around size and then fault in the number of requested pages containing the faulting address (e.g. address 0x1000 will fault in pages in the range 0 to 0xf000 inclusive).

We then clamp this to VMA and page table boundaries, so we avoid any issue with acquiring locks and avoid complexity as a result.

We go into more detail about this in the page fault

See Section 6.5 in the Page Fault chapter for a detailed explanation of this code which we explore in more detail.

Here we will instead focus on the struct `vm_area_struct` (VMA)'s `vm_ops->map_pages()` function which performs the heavy lifting here (part of the struct `vm_operations_struct` callbacks).

As with reading data and handling file-mapped page fault resolution, file systems are free to do what they want but typically use a library function, in this case `filemap_map_pages()`. We examine this function in Listing 9-84 (eliding out of scope huge page and hardware poisoning handling).

```

3322 vm_fault_t filemap_map_pages(struct vm_fault *vmf,
3323                             pgoff_t start_pgoff, pgoff_t end_pgoff)
3324 {
3325     struct vm_area_struct *vma = vmf->vma;
3326     struct file *file = vma->vm_file;
3327     struct address_space *mapping = file->f_mapping;
3328     pgoff_t last_pgoff = start_pgoff;
3329     unsigned long addr;
3330     XA_STATE(xas, &mapping->i_pages, start_pgoff);
3331     struct folio *folio;
3332     struct page *page;
3333     unsigned int mmap_miss = READ_ONCE(file->f_ra.mmap_miss);
3334     vm_fault_t ret = 0;
3335
3336     rcu_read_lock();
3337     folio = first_map_page(mapping, &xas, end_pgoff);

```



```

3338         if (!folio)
3339             goto out;
3340         :
3341     addr = vma->vm_start + ((start_pgoff - vma->vm_pgoff) << PAGE_SHIFT);
3342     vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, addr, &vmf->ptl);
3343     do {
3344         :
3345         page = folio_file_page(folio, xas.xa_index);
3346         :
3347         if (mmap_miss > 0)
3348             mmap_miss--;
3349         :
3350         addr += (xas.xa_index - last_pgoff) << PAGE_SHIFT;
3351         vmf->pte += xas.xa_index - last_pgoff;
3352         last_pgoff = xas.xa_index;
3353         :
3354         /*
3355          * NOTE: If there're PTE markers, we'll leave them to be
3356          * handled in the specific fault path, and it'll prohibit the
3357          * fault-around logic.
3358          */
3359         if (!pte_none(*vmf->pte))
3360             goto unlock;
3361         :
3362         /* We're about to handle the fault */
3363         if (vmf->address == addr)
3364             ret = VM_FAULT_NOPAGE;
3365         :
3366         do_set_pte(vmf, page, addr);
3367         /* no need to invalidate: a not-present page won't be cached
3368          */
3369         :
3370         update_mmu_cache(vma, addr, vmf->pte);
3371         :
3372         folio_unlock(folio);
3373         continue;
3374     unlock:
3375         :
3376         folio_unlock(folio);
3377         folio_put(folio);
3378     } while ((folio = next_map_page(mapping, &xas, end_pgoff)) != NULL);
3379     pte_unmap_unlock(vmf->pte, vmf->ptl);
3380     out:
3381     rcu_read_unlock();
3382     WRITE_ONCE(file->f_ra.mmap_miss, mmap_miss);
3383     return ret;
3384 }

```

Listing 9-84: mm/filemap.c: `filemap_map_pages()`

After some initialisation and establishment of an xarray struct `xa_state` object via `XA_STATE()`, initialising it to point at the first page offset of the sought folio.

We retrieve the first folio in the range using `first_map_page()` which we examine in Listing ??.

```

3306 static inline struct folio *first_map_page(struct address_space *mapping,
3307                                           struct xa_state *xas,
3308                                           pgoff_t end_pgoff)
3309 {
3310     return next_uptodate_page(xas_find(xas, end_pgoff),
3311                               mapping, xas, end_pgoff);
3312 }

```

Listing 9-85: mm/filemap.c: first_map_page()

This calls the `next_uptodate_page()` helper function which we examine shortly in Listing 9-87.

We provide a folio to it using `xas_find()`, specifying that we wish to find the next folio up to and including `end_pgoff`.

Later in the loop we utilise `next_map_page()` which we examine in Listing ??.

```

3314 static inline struct folio *next_map_page(struct address_space *mapping,
3315                                           struct xa_state *xas,
3316                                           pgoff_t end_pgoff)
3317 {
3318     return next_uptodate_page(xas_next_entry(xas, end_pgoff),
3319                               mapping, xas, end_pgoff);
3320 }

```

Listing 9-86: mm/filemap.c: next_map_page()

This is roughly equivalent to `first_map_page()`, only it uses `xas_next_entry()` which is an inline, more efficient version of `xas_find()` for finding the next entry.

Each of these are simply helpers which find the next `PG_uptodate` folio in the page cache within this range, i.e. the next folio which is present and valid there.

We'll return to `next_uptodate_page()` shortly and in the mean time examine the logic which uses it.

If `first_map_page()` fails to find a folio, then this implies there are no suitable folios in the page cache in the specified range and we simply abort.

Otherwise, we determine the address to be equal to the start of the VMA less the difference between `start_pgoff` and the struct `vm_area_struct->vm_pgoff` field. This might seem odd, but takes into account the fact that `mremap()` can actually move a VMA while maintaining the same struct `vm_area_struct->vm_pgoff` even when it is no longer valid, in order to avoid having to update reverse mapping data structures unnecessarily (see Section 8.2.3 for more details).

This mimics the logic found in `linear_page_index()`, see the discussion around this in Listing 7-23 for further discussion.

Once this address is obtained, we acquire a lock around the PTE page table via `pte_offset_map_lock()` (see the Virtual Memory chapter for details of page tables and locking).

With this lock acquired, we're able to manipulate the PTE entry and thus very quickly directly perform a minor fault operation here.

We loop around, invoking `next_map_page()` (see Listing ??) on each iteration to obtain the next valid page.

Note that at this stage, either `first_map_page()` or `next_map_page()` (both invoking `next_uptodate_page()`) will have incremented the reference count and obtained a lock on the folio and also ensured that it is stable and has not been truncated as well as being uptodate.

We may actually encounter the same folio multiple times if it is greater in size than order-0, with the lookup functions simply incrementing the index each time. We therefore start each loop by determining the struct page subpage at the current index via `folio_file_page()`.

We keep track of the struct `file->f_ra` struct `file_ra_state->mmap_miss` readahead counter, which we update after the operation is complete.

We then update offsets to take into account the fact that this might be a subpage within a larger folio, before determining if the PTE in question is empty or not—it is only valid to proceed if it is empty, so we move on to the next page if it is not.

If the page fault state field struct `vm_fault->address` is already set to the target address, this suggests that the file system-specific fault handler has already installed an address so we set the return value to `VM_FAULT_NOPAGE` to indicate this.

At this stage we perform the actual setting of the PTE to point at the located page, i.e. perform a minor fault via `do_set_pte()` (see Listing 6-35 in Section 6.8 for a detailed exploration of this function).

For architectures that need it (x86-64 does not) we indicate that this page table has been updated via `update_mmu_cache()`. No TLB actions need be performed as we require that the PTE entry had to be empty before we proceeded.

After this is done we unlock the struct `folio` and loop to the next page to examine.

Note that we only unlock the PTE after the loop is complete—remember that we ensure we don't cross page table boundaries, and therefore need only acquire this lock once for a single PTE.

Now we have examined the core logic of this function, let's return to `next_uptodate_page()` in Listing 9-87.

```

3265 static struct folio *next_uptodate_page(struct folio *folio,
3266                                         struct address_space *mapping,
3267                                         struct xa_state *xas, pgoff_t end_pgoff
3268                                         )
3269 {
3270     unsigned long max_idx;
```

```

3270
3271     do {
3272         if (!folio)
3273             return NULL;
3274         if (xas_retry(xas, folio))
3275             continue;
3276         if (xa_is_value(folio))
3277             continue;
3278         if (folio_test_locked(folio))
3279             continue;
3280         if (!folio_try_get_rcu(folio))
3281             continue;
3282         /* Has the page moved or been split? */
3283         if (unlikely(folio != xas_reload(xas)))
3284             goto skip;
3285         if (!folio_test_uptodate(folio) || folio_test_readahead(folio)
3286             )
3287             goto skip;
3288         if (!folio_trylock(folio))
3289             goto skip;
3290         if (folio->mapping != mapping)
3291             goto unlock;
3292         if (!folio_test_uptodate(folio))
3293             goto unlock;
3294         max_idx = DIV_ROUND_UP(i_size_read(mapping->host), PAGE_SIZE);
3295         if (xas->xa_index >= max_idx)
3296             goto unlock;
3297         return folio;
3298     unlock:
3299         folio_unlock(folio);
3300     skip:
3301         folio_put(folio);
3302     } while ((folio = xas_next_entry(xas, end_pgoff)) != NULL);
3303     return NULL;
3304 }

```

Listing 9-87: mm/filemap.c: *next_uptodate_page()*

This is a useful helper function that does all of the housekeeping required to check that a folio is uptodate, stable and suitable to be referenced while acquiring a reference count on it and locking it while doing all the required dances and handling all edge cases that might arise.

Note that this function *tries* to acquire a lock on the folio, if there is contention on it the folio is skipped. The fault around logic is intended to be a best-effort optimisation, so we will not wait for a lock to become available.

The function is handed a present folio from the page cache, and we perform a do/while loop in case we discover this folio is not suitable so we can examine the next.

We cover the case where the xarray entry indicates that the lookup must be retried via `xas_retry()`, doing so if this is the case. We then determine if the entry contains a value rather than a pointer (which would indicate that the folio has been evicted) via `xa_is_value()`, and again continue to the next folio if so.

We do a fast check before trying to obtain a reference to see whether the folio is already locked, if not then we try to obtain a folio lock via `folio_try_get_rcu()`. If either fail then we examine the next folio.

We check whether the folio has been moved or split via `xas_reload()`, if so we decrement the reference count and examine the next folio in the range.

Next we perform the key check – determining whether the folio is upto-date via `folio_test_uptodate()`. If not, then it has not yet been read back from the disk and thus cannot be used. Note that we do this *prior* to obtaining a lock for speed, we will recheck this after a lock is acquired.

We then carefully check to see if the folio has been marked with the `PG_readahead` flag. If so, this indicates that upon reading this folio in, further readahead should be performed. Since we are unable to do so here, we decrement the reference count and examine the next folio.

We then attempt to acquire the folio lock via `folio_trylock()`, if this fails then there is lock contention and we decrement the reference count and examine the next folio.

With the lock acquired, we check to ensure that the struct `folio->mapping` field matches the struct `address_space` page cache object we are attempting to access. If not, this indicates truncation, and thus we unlock the folio and examine the next.

Equally, we perform the `PG_uptodate` check once again with the lock acquired to ensure that it has not been acquired prior to us obtaining the lock.

Finally we perform a sanity check to ensure the file size has not changed such as to render the current index within the folio invalid, aborting if so.

If all checks are complete, we return the folio. If the loop iterates through the entire range and finds nothing, then we return `NULL` and ultimately `filemap_map_pages()` handles this scenario by exiting its own loop.

9.9 Removing Page Cache Entries and Truncation

The removal of folios from the page cache involves a lot of moving parts, as what the folios are being used for must be taken into account—importantly, this involves *unmapping* the folios using the reverse mapping as any number of users might have the page cache folios currently mapped. We explore this in Section 9.9.3.

However, we start by exploring the fundamental mechanism of removing a folio from the page cache once this has been taken into account in Section 9.9.1, as this is relatively straight forward in comparison to the machinery that sits above it.

We therefore examine the provided mechanisms for removing folios from the page cache through *truncation*, i.e. the process of adjusting file size or file removal resulting in folios no longer being present in the page cache, in Section 9.9.2.

Next we examine how the mechanism for dropping caches via the `vm.drop_caches` tunable is implemented in Section 9.9.4, before exploring how this evicts folios altogether from the page cache in Section 9.9.5.

9.9.1 Removing Folios from the Page Cache

The most straightforward means of removing a folio from the page cache, once we know that it ought to be removed is via `filemap_remove_folio()` which we examine in Listing ?? (eliding some irrelevant inode reclaim logic).

```

240 /**
241  * filemap_remove_folio - Remove folio from page cache.
242  * @folio: The folio.
243  *
244  * This must be called only on folios that are locked and have been
245  * verified to be in the page cache. It will never put the folio into
246  * the free list because the caller has a reference on the page.
247  */
248 void filemap_remove_folio(struct folio *folio)
249 {
250     struct address_space *mapping = folio->mapping;
251
252     BUG_ON(!folio_test_locked(folio));
253     spin_lock(&mapping->host->i_lock);
254     xa_lock_irq(&mapping->i_pages);
255     __filemap_remove_folio(folio, NULL);
256     xa_unlock_irq(&mapping->i_pages);
257     :
258     spin_unlock(&mapping->host->i_lock);
259
260     filemap_free_folio(mapping, folio);
261 }
262 
```

Listing 9-88: `mm/filemap.c: filemap_remove_folio()`

Note that this function is also wrapped by the legacy `delete_from_page_cache()` function.

This requires the incoming folio to be locked, as do all truncation and folio removal functions. The host inode is locked, and then the xarray which describes it in turn, which we remove via `__filemap_remove_folio()`, which we examine in Listing 9-89 below.

Finally this invokes `filemap_free_folio()` which invokes any custom file system folio freeing logic in `struct address_space->a_ops` specifying a `struct address_space_operations` handler for `free_folio()`.

We examine `__filemap_remove_folio()` in Listing 9-89 (eliding trace hook).

```

212 /*
213  * Delete a page from the page cache and free it. Caller has to make
214  * sure the page is locked and that nobody else uses it - or that usage
215  * is safe. The caller must hold the i_pages lock.
216  */
217 void __filemap_remove_folio(struct folio *folio, void *shadow)
218 {
219     struct address_space *mapping = folio->mapping;
220     :
221     filemap_unaccount_folio(mapping, folio);
222     page_cache_delete(mapping, folio, shadow);
223 }
224 
```

Listing 9-89: mm/filemap.c: `__filemap_remove_folio()`

This updates statistics accordingly via `filemap_unaccount_folio()`, before removing the folio from the associated xarray and updating folio via `page_cache_delete()`, which we examine in Listing 9-90 (eliding out of scope huge page and DAX logic and a debug check).

```

124 static void page_cache_delete(struct address_space *mapping,
125                             struct folio *folio, void *shadow)
126 {
127     XA_STATE(xas, &mapping->i_pages, folio->index);
128     long nr = 1;
129     :
140     xas_store(&xas, shadow);
141     xas_init_marks(&xas);
142
143     folio->mapping = NULL;
144     /* Leave page->index set: truncation lookup relies upon it */
145     mapping->nrpages -= nr;
146 }

```

Listing 9-90: mm/filemap.c: `page_cache_delete()`

This replaces the page cache entry with shadow, which can be used by the working set logic to track eviction and refaulting (which is out of scope for the book), but in this instance we have passed `NULL` in any case. The value is set via `xas_store()`.

We also clear any marks (such as writeback/dirty etc.) as this folio is now removed, via `xas_init_marks()`.

Finally, and very importantly, we mark the folio truncated by setting the struct `folio->mapping` field to `NULL`. This is heavily relied upon in the rest of the kernel to detect when a folio is truncated, and equally eliminates any incorrect reverse mapping traversal from a now-truncated folio.

We also update the struct `address_space->nrpages` count to indicate that this folio has been removed from the xarray describing the page cache entry.

In addition to removing a single folio, folios can be removed from the page cache in batches via `delete_from_page_cache_batch()` which we examine in Listing 9-91 (eliding trace hook and out of scope inode reclaim logic).

```

318 void delete_from_page_cache_batch(struct address_space *mapping,
319                                 struct folio_batch *fbatch)
320 {
321     int i;
322
323     if (!folio_batch_count(fbatch))
324         return;
325
326     spin_lock(&mapping->host->i_lock);
327     xa_lock_irq(&mapping->i_pages);
328     for (i = 0; i < folio_batch_count(fbatch); i++) {
329         struct folio *folio = fbatch->folios[i];
330         :
331
332         filemap_unaccount_folio(mapping, folio);
333     }
334     page_cache_delete_batch(mapping, fbatch);
335     xa_unlock_irq(&mapping->i_pages);
336     :
337
338     spin_unlock(&mapping->host->i_lock);
339
340     for (i = 0; i < folio_batch_count(fbatch); i++)
341         filemap_free_folio(mapping, fbatch->folios[i]);
342 }

```

Listing 9-91: mm/filemap.c: delete_from_page_cache_batch()

This mirrors the `__filemap_remove_folio()` logic, only performed in batches using the `struct folio_batch` type (see Section 11.7 for more on this).

As before, folios have statistics updated via `filemap_unaccount_folio()` and any file system custom freeing logic is invoked via `filemap_free_folio()`. The role played by `page_cache_delete()` in the single folio case is replaced with `page_cache_delete_batch()` which we examine in Listing 9-92 (eliding out of scope DAX handling and debug check).

```

264 /*
265  * page_cache_delete_batch - delete several folios from page cache
266  * @mapping: the mapping to which folios belong
267  * @fbatch: batch of folios to delete
268  *
269  * The function walks over mapping->i_pages and removes folios passed in
270  * @fbatch from the mapping. The function expects @fbatch to be sorted
271  * by page index and is optimised for it to be dense.
272  * It tolerates holes in @fbatch (mapping entries at those indices are not
273  * modified).
274  *

```



```

275  * The function expects the i_pages lock to be held.
276  */
277  static void page_cache_delete_batch(struct address_space *mapping,
278                                     struct folio_batch *fbatch)
279  {
280      XA_STATE(xas, &mapping->i_pages, fbatch->folios[0]->index);
281      long total_pages = 0;
282      int i = 0;
283      struct folio *folio;
284      :
285      xas_for_each(&xas, folio, ULONG_MAX) {
286          if (i >= folio_batch_count(fbatch))
287              break;
288
289          /* A swap/dax/shadow entry got inserted? Skip it. */
290          if (xa_is_value(folio))
291              continue;
292
293          /*
294          * A page got inserted in our range? Skip it. We have our
295          * pages locked so they are protected from being removed.
296          * If we see a page whose index is higher than ours, it
297          * means our page has been removed, which shouldn't be
298          * possible because we're holding the PageLock.
299          */
300          if (folio != fbatch->folios[i]) {
301              :
302              continue;
303          }
304
305          WARN_ON_ONCE(!folio_test_locked(folio));
306
307          folio->mapping = NULL;
308          /* Leave folio->index set: truncation lookup relies on it */
309
310          i++;
311          xas_store(&xas, NULL);
312          total_pages += folio_nr_pages(folio);
313      }
314      mapping->numpages -= total_pages;
315  }

```

Listing 9-92: mm/filemap.c: page_cache_delete_batch()

This iterates over all of the folios in the range starting from the first folio in the batch until a number of folios equal to the number in the batch have been examined. As folio locks will have been taken, we are protected from racing truncation, but new folios may have been inserted in the range, which

we skip, asserting in each instance that folio under examination is equal to the one we expect in the batch.

We then assign NULL to the struct `folio->mapping` field, and keep a running count of folio pages to decrement in struct `address_space->npages`, which we do at the end of the operation.

9.9.2 Folio Truncation

There are a number of ways in which file systems and the kernel can trigger folio truncation in the page cache. We examine the different possible routes in Figure 9-13.

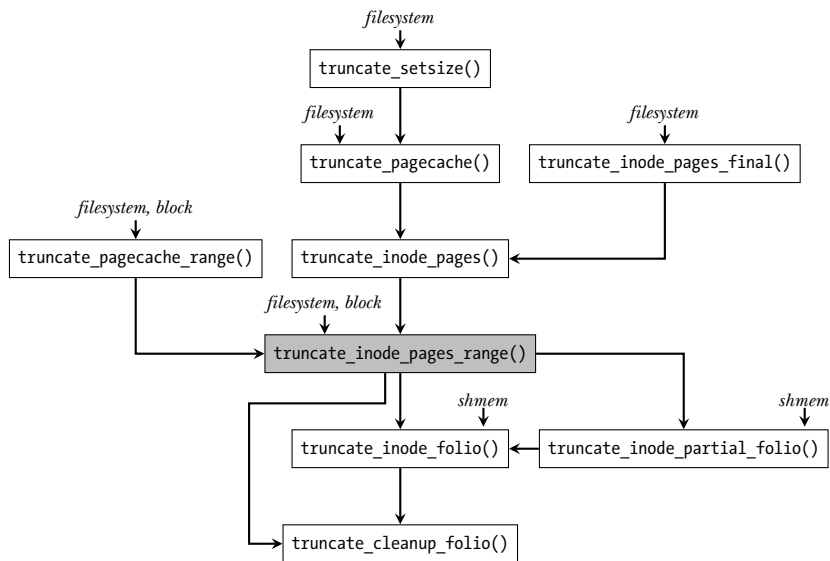


Figure 9-13: Folio Truncation Code Paths

First we examine `truncate_setsize()` in Listing 9-93.

```

758 /**
759  * truncate_setsize - update inode and pagecache for a new file size
760  * @inode: inode
761  * @newsize: new file size
762  *
763  * truncate_setsize updates i_size and performs pagecache truncation (if
764  * necessary) to @newsize. It will be typically be called from the filesystem'
765  * s
766  * setattr function when ATTR_SIZE is passed in.
767  *
768  * Must be called with a lock serializing truncates and writes (generally
769  * i_rwsem but e.g. xfs uses a different lock) and before all filesystem
770  * specific block truncation has been performed.
771  */

```

```

771 void truncate_setsize(struct inode *inode, loff_t newsize)
772 {
773     loff_t oldsize = inode->i_size;
774
775     i_size_write(inode, newsize);
776     if (newsize > oldsize)
777         pagecache_isize_extended(inode, oldsize, newsize);
778     truncate_pagecache(inode, newsize);
779 }

```

Listing 9-93: mm/truncate.c: truncate_setsize()

This updates the file size via `i_size_write()` and, if extended, handles this via `pagecache_isize_extended()`. We are focused on files shrinking so we do not examine these.

The truncation is then deferring to `truncate_pagecache()` which we examine in Listing 9-94.

```

723 /**
724  * truncate_pagecache - unmap and remove pagecache that has been truncated
725  * @inode: inode
726  * @newsize: new file size
727  *
728  * inode's new i_size must already be written before truncate_pagecache
729  * is called.
730  *
731  * This function should typically be called before the filesystem
732  * releases resources associated with the freed range (eg. deallocates
733  * blocks). This way, pagecache will always stay logically coherent
734  * with on-disk format, and the filesystem would not have to deal with
735  * situations such as writepage being called for a page that has already
736  * had its underlying blocks deallocated.
737  */
738 void truncate_pagecache(struct inode *inode, loff_t newsize)
739 {
740     struct address_space *mapping = inode->i_mapping;
741     loff_t holebegin = round_up(newsize, PAGE_SIZE);
742
743     /*
744      * unmap_mapping_range is called twice, first simply for
745      * efficiency so that truncate_inode_pages does fewer
746      * single-page unmmaps. However after this first call, and
747      * before truncate_inode_pages finishes, it is possible for
748      * private pages to be COWed, which remain after
749      * truncate_inode_pages finishes, hence the second
750      * unmap_mapping_range call must be made for correctness.
751      */
752     unmap_mapping_range(mapping, holebegin, 0, 1);
753     truncate_inode_pages(mapping, newsize);

```

```

754         unmap_mapping_range(mapping, holebegin, 0, 1);
755     }

```

Listing 9-94: mm/truncate.c: truncate_pagecache()

This unmaps mappings to the portion of the folio that has shrunk (if indeed it has), using `unmap_mapping_range()` to perform the unmapping, which we examine in Section 9.9.3.

The unmapping is done once before truncation and once after, as the comment points out both to reduce the number of operations the truncation operation need perform as well as handling any late Copy-on-Write faults.

It defers the actual truncation operation to `truncate_inode_pages()` which we examine in Listing 9-96.

Before we examine this, we examine `truncate_inode_pages_final()` in Listing 9-95.

```

456  /**
457   * truncate_inode_pages_final - truncate all pages before inode dies
458   * @mapping: mapping to truncate
459   *
460   * Called under (and serialized by) inode->i_rwsem.
461   *
462   * Filesystems have to use this in the .evict_inode path to inform the
463   * VM that this is the final truncate and the inode is going away.
464   */
465  void truncate_inode_pages_final(struct address_space *mapping)
466  {
467      /*
468       * Page reclaim can not participate in regular inode lifetime
469       * management (can't call iput()) and thus can race with the
470       * inode teardown. Tell it when the address space is exiting,
471       * so that it does not install eviction information after the
472       * final truncate has begun.
473       */
474      mapping_set_exiting(mapping);
475
476      if (!mapping_empty(mapping)) {
477          /*
478           * As truncation uses a lockless tree lookup, cycle
479           * the tree lock to make sure any ongoing tree
480           * modification that does not see AS_EXITING is
481           * completed before starting the final truncate.
482           */
483          xa_lock_irq(&mapping->i_pages);
484          xa_unlock_irq(&mapping->i_pages);
485      }
486
487      truncate_inode_pages(mapping, 0);

```

488 }

Listing 9-95: mm/truncate.c: truncate_inode_pages_final()

This function is invoked just before an inode is destroyed completely, which ultimately defers the truncation operation (with size set to zero as the file is being removed) to truncate_inode_pages() which we examine in Listing 9-96.

```

437 /**
438  * truncate_inode_pages - truncate *all* the pages from an offset
439  * @mapping: mapping to truncate
440  * @lstart: offset from which to truncate
441  *
442  * Called under (and serialised by) inode->i_rwsem and
443  * mapping->invalidate_lock.
444  *
445  * Note: When this function returns, there can be a page in the process of
446  * deletion (inside __filemap_remove_folio()) in the specified range. Thus
447  * mapping->nripages can be non-zero when this function returns even after
448  * truncation of the whole mapping.
449  */
450 void truncate_inode_pages(struct address_space *mapping, loff_t lstart)
451 {
452     truncate_inode_pages_range(mapping, lstart, (loff_t)-1);
453 }
454 EXPORT_SYMBOL(truncate_inode_pages);

```

Listing 9-96: mm/truncate.c: truncate_inode_pages()

This invokes truncate_inode_pages_range() specifying the end offset to be unbounded. We examine this function in Listing 9-98.

The kernel also provides truncate_pagecache_range(), which we examine in Listing 9-97.

```

833 /**
834  * truncate_pagecache_range - unmap and remove pagecache that is hole-punched
835  * @inode: inode
836  * @lstart: offset of beginning of hole
837  * @lend: offset of last byte of hole
838  *
839  * This function should typically be called before the filesystem
840  * releases resources associated with the freed range (eg. deallocates
841  * blocks). This way, pagecache will always stay logically coherent
842  * with on-disk format, and the filesystem would not have to deal with
843  * situations such as writepage being called for a page that has already
844  * had its underlying blocks deallocated.
845  */
846 void truncate_pagecache_range(struct inode *inode, loff_t lstart, loff_t lend)
847 {

```

```

848     struct address_space *mapping = inode->i_mapping;
849     loff_t unmap_start = round_up(lstart, PAGE_SIZE);
850     loff_t unmap_end = round_down(1 + lend, PAGE_SIZE) - 1;
851     /*
852      * This rounding is currently just for example: unmap_mapping_range
853      * expands its hole outwards, whereas we want it to contract the hole
854      * inwards. However, existing callers of truncate_pagecache_range are
855      * doing their own page rounding first. Note that unmap_mapping_range
856      * allows holelen 0 for all, and we allow lend -1 for end of file.
857      */
858
859     /*
860      * Unlike in truncate_pagecache, unmap_mapping_range is called only
861      * once (before truncating pagecache), and without "even_cows" flag:
862      * hole-punching should not remove private COWed pages from the hole.
863      */
864     if ((u64)unmap_end > (u64)unmap_start)
865         unmap_mapping_range(mapping, unmap_start,
866                             1 + unmap_end - unmap_start, 0);
867     truncate_inode_pages_range(mapping, lstart, lend);
868 }

```

Listing 9-97: mm/truncate.c: `truncate_pagecache_range()`

This provides the means for file system to ‘hole punch’ files, that is to delete portions of a file in order that they can be treated as if they were sparse. When we do so, we must unmap the hole-punched region and truncate within this range.

This function achieves that by unmapping in the hole-punched range via `unmap_mapping_range()` (see Listing ??), and truncating the hole-punched range in `truncate_inode_pages_range()`, which we examine starting at Listing 9-98 (eliding out of scope realtime scheduler hints and debug check).

```

306  /**
307   * truncate_inode_pages_range - truncate range of pages specified by start &
308   *                               end byte offsets
309   * @mapping: mapping to truncate
310   * @lstart: offset from which to truncate
311   * @lend: offset to which to truncate (inclusive)
312   *
313   * Truncate the page cache, removing the pages that are between
314   * specified offsets (and zeroing out partial pages
315   * if lstart or lend + 1 is not page aligned).
316   *
317   * Truncate takes two passes - the first pass is nonblocking. It will not
318   * block on page locks and it will not block on writeback. The second pass
319   * will wait. This is to prevent as much IO as possible in the affected
320   * region.

```

```

319  * The first pass will remove most pages, so the search cost of the second
      pass
320  * is low.
321  *
322  * We pass down the cache-hot hint to the page freeing code. Even if the
323  * mapping is large, it is probably the case that the final pages are the most
324  * recently touched, and freeing happens in ascending file offset order.
325  *
326  * Note that since ->invalidate_folio() accepts range to invalidate
327  * truncate_inode_pages_range is able to handle cases where lend + 1 is not
328  * page aligned properly.
329  */
330 void truncate_inode_pages_range(struct address_space *mapping,
331                               loff_t lstart, loff_t lend)
332 {
333     pgoff_t      start;          /* inclusive */
334     pgoff_t      end;            /* exclusive */
335     struct folio_batch fbatch;
336     pgoff_t      indices[PAGEVEC_SIZE];
337     pgoff_t      index;
338     int          i;
339     struct folio  *folio;
340     bool         same_folio;
341
342     if (mapping_empty(mapping))
343         return;
344
345     /*
346      * 'start' and 'end' always covers the range of pages to be fully
347      * truncated. Partial pages are covered with 'partial_start' at the
348      * start of the range and 'partial_end' at the end of the range.
349      * Note that 'end' is exclusive while 'lend' is inclusive.
350      */
351     start = (lstart + PAGE_SIZE - 1) >> PAGE_SHIFT;
352     if (lend == -1)
353         /*
354          * lend == -1 indicates end-of-file so we have to set 'end'
355          * to the highest possible pgoff_t and since the type is
356          * unsigned we're using -1.
357          */
358         end = -1;
359     else
360         end = (lend + 1) >> PAGE_SHIFT;
361
362     folio_batch_init(&fbatch);

```

Listing 9-98: mm/truncate.c: truncate_inode_pages_range() Preface

As the comment suggests, this function performs two passes, each of which we examine separately. It uses a struct `folio_batch` type to perform this operation in batches (see Section 11.7 for more on this data type).

If the mapping is empty we have nothing to do so abort. Otherwise we establish start and end to the page-aligned inclusive and *exclusive* bounds of the range to examine.

We initialise the folio batch to begin the batched operation via `folio_batch_init()` before beginning the first pass which we examine in Listing 9-99.

```

363     index = start;
364     while (index < end && find_lock_entries(mapping, index, end - 1,
365         &fbatch, indices)) {
366         index = indices[folio_batch_count(&fbatch) - 1] + 1;
367         truncate_folio_batch_exceptionals(mapping, &fbatch, indices);
368         for (i = 0; i < folio_batch_count(&fbatch); i++)
369             truncate_cleanup_folio(fbatch.folios[i]);
370         delete_from_page_cache_batch(mapping, &fbatch);
371         for (i = 0; i < folio_batch_count(&fbatch); i++)
372             folio_unlock(fbatch.folios[i]);
373         folio_batch_release(&fbatch);
374     }
375 
```

Listing 9-99: mm/truncate.c: truncate_inode_pages_range() First Pass

We use `find_lock_entries()` (see Listing 9-40 in Section 9.5.1) to find page cache entries in the range, optimistically attempting to lock each, skipping entries that it can't lock or have been meanwhile truncated. This therefore does not block.

We invoke `truncate_folio_batch_exceptionals()` to remove 'exceptional' entries in the batch. These are page cache entries which have been evicted but have left behind data for the working set logic to utilise within the kernel. This is out of scope for the book, but suffice to say these entries are not valid and should be filtered.

For each valid folio, we invoke `truncate_cleanup_folio()` to perform the actual truncation, a function we will return to in Listing 9-103.

Once truncation has completed, we remove the folios from the page cache (note they are each now locked) via `delete_from_page_cache_batch()` (see Listing 9-91).

We then unlock each folio and free the folio batch object via `folio_batch_release()`.

Once this first pass is complete, we proceed with handling the portion of the specified range which is not page-aligned on either side of the range, i.e. partial truncation. Within this range we simply zero values, which we examine in Listing 9-100.

```

363     same_folio = (lstart >> PAGE_SHIFT) == (lend >> PAGE_SHIFT);

```

```

364     folio = __filemap_get_folio(mapping, lstart >> PAGE_SHIFT, FGP_LOCK,
    0);
365     if (folio) {
366         same_folio = lend < folio_pos(folio) + folio_size(folio);
367         if (!truncate_inode_partial_folio(folio, lstart, lend)) {
368             start = folio->index + folio_nr_pages(folio);
369             if (same_folio)
370                 end = folio->index;
371         }
372         folio_unlock(folio);
373         folio_put(folio);
374         folio = NULL;
375     }
376
377     if (!same_folio)
378         folio = __filemap_get_folio(mapping, lend >> PAGE_SHIFT,
379                                     FGP_LOCK, 0);
380     if (folio) {
381         if (!truncate_inode_partial_folio(folio, lstart, lend))
382             end = folio->index;
383         folio_unlock(folio);
384         folio_put(folio);
385     }

```

Listing 9-100: mm/truncate.c: truncate_inode_pages_range() Partial Truncation

We start by setting `same_folio` to indicate whether the start and end of the range exist within the same page. We correct this shortly to account for higher order folios.

We use `FGP_LOCK` to indicate that the folio should be locked, passed to `__filemap_get_folio()` to retrieve the folio. This will wait for the lock to become available if it is currently contended.

Once obtained, we update `same_folio` to account for the fact that the folio may be higher order, before invoking `truncate_inode_partial_folio()` to perform the actual heavy lifting.

In the instance that this was a larger order folio and we were unable to split it we update the start (and possibly the end) of the overall truncation range to skip it (this behaviour is out of scope for the book, however).

After this operation we unlock and drop the reference count for the folio, before performing the same operation for the end of the range.

After this is complete, we continue with the second pass of the operation, which we explore in Listing 9-101.

```

401     index = start;
402     while (index < end) {
403         :
404         if (!find_get_entries(mapping, index, end - 1, &fbatch,
405                             indices)) {
406             /* If all gone from start onwards, we're done */

```

```

407             if (index == start)
408                 break;
409             /* Otherwise restart to make sure all gone */
410             index = start;
411             continue;
412         }
413
414         for (i = 0; i < folio_batch_count(&fbatch); i++) {
415             struct folio *folio = fbatch.folios[i];
416
417             /* We rely upon deletion not changing page->index */
418             index = indices[i];
419
420             if (xa_is_value(folio))
421                 continue;
422
423             folio_lock(folio);
424             :
425             folio_wait_writeback(folio);
426             truncate_inode_folio(mapping, folio);
427             folio_unlock(folio);
428             index = folio_index(folio) + folio_nr_pages(folio) -
1;
429         }
430         truncate_folio_batch_exceptionals(mapping, &fbatch, indices);
431         folio_batch_release(&fbatch);
432         index++;
433     }
434 }

```

Listing 9-101: mm/truncate.c: truncate_inode_pages_range() Second Pass

In this pass we retrieve the raw entries via `find_get_entries()` (see Listing 9-40 and Section 9.5.1).

We then proceed as before. If the `find_get_entries()` fails (perhaps because all entries are eliminated) then if the operation is partially complete we restart to ensure we get everything, otherwise we abort.

Note that we may encounter entries deleted by the previous pass. We rely on the fact that we intentionally leave the `struct folio->index` field in place even when deleted to correctly keep track of indices in the range (which `find_get_entries()` has placed in the `indices` array).

We skip non-folio pointer entries then this time intentionally wait for the folio lock to become available via `folio_lock()` (see Listing 9-137), before waiting for any writeback to complete also via `folio_wait_writeback()` (see Listing 9-146, and Section 9.11 for more details on folio locking in general). This ensures that the folio is stable if it was not competitively truncated, before relying on `truncate_inode_folio()` to perform the actual truncation and deletion of the folio.

Next we examine `truncate_inode_folio()` in Listing 9-102.

```

190 int truncate_inode_folio(struct address_space *mapping, struct folio *folio)
191 {
192     if (folio->mapping != mapping)
193         return -EIO;
194
195     truncate_cleanup_folio(folio);
196     filemap_remove_folio(folio);
197     return 0;
198 }

```

Listing 9-102: mm/truncate.c: truncate_inode_folio()

This assumes the passed folio is locked, then performs the usual truncation dance (we are meant to be truncating, if somebody else has got to it first then there's nothing for us to do and return an error to indicate that this is unexpected).

We then perform the truncation via `truncate_cleanup_folio()`, which we examine in Listing 9-103, removing the folio altogether via `filemap_remove_folio()` (see Listing ??).

```

163 /*
164  * If truncate cannot remove the fs-private metadata from the page, the page
165  * becomes orphaned. It will be left on the LRU and may even be mapped into
166  * user pagetables if we're racing with filemap_fault().
167  *
168  * We need to bail out if page->mapping is no longer equal to the original
169  * mapping. This happens a) when the VM reclaimed the page while we waited on
170  * its lock, b) when a concurrent invalidate_mapping_pages got there first and
171  * c) when tmpfs swizzles a page between a tmpfs inode and swapper_space.
172  */
173 static void truncate_cleanup_folio(struct folio *folio)
174 {
175     if (folio_mapped(folio))
176         unmap_mapping_folio(folio);
177
178     if (folio_has_private(folio))
179         folio_invalidate(folio, 0, folio_size(folio));
180
181     /*
182      * Some filesystems seem to re-dirty the page even after
183      * the VM has canceled the dirty bit (eg ext3 journaling).
184      * Hence dirty accounting check is placed after invalidation.
185      */
186     folio_cancel_dirty(folio);
187     folio_clear_mappedtodisk(folio);
188 }

```

Listing 9-103: mm/truncate.c: truncate_cleanup_folio()

We perform unmapping of the folio if it is currently mapped (as inferred by map count via `folio_mapped()`) via `unmap_mapping_folio()` (see Listing 9-106).

If the folio has private state then this indicates that there may be state the file system needs to invalidate, in this case we invoke `folio_invalidate()` which, if the page cache entry's address space operations specified in `struct address_space->a_ops` of type `struct address_space_operations` possesses a handler for `invalidate_folio()`, invokes this handler.

Finally we have some edge case handling to take care of—revoking any erroneous `PG_dirty` flag via `folio_cancel_dirty()` before removing the flag.

Finally we examine `truncate_inode_partial_folio()` in Listing 9-104 (eliding out of scope huge page handling).

```

200 /*
201  * Handle partial folios. The folio may be entirely within the
202  * range if a split has raced with us. If not, we zero the part of the
203  * folio that's within the [start, end] range, and then split the folio if
204  * it's large. split_page_range() will discard pages which now lie beyond
205  * i_size, and we rely on the caller to discard pages which lie within a
206  * newly created hole.
207  *
208  * Returns false if splitting failed so the caller can avoid
209  * discarding the entire folio which is stubbornly unsplit.
210  */
211 bool truncate_inode_partial_folio(struct folio *folio, loff_t start, loff_t
    end)
212 {
213     loff_t pos = folio_pos(folio);
214     unsigned int offset, length;
215
216     if (pos < start)
217         offset = start - pos;
218     else
219         offset = 0;
220     length = folio_size(folio);
221     if (pos + length <= (u64)end)
222         length = length - offset;
223     else
224         length = end + 1 - pos - offset;
225
226     folio_wait_writeback(folio);
227     if (length == folio_size(folio)) {
228         truncate_inode_folio(folio->mapping, folio);
229         return true;
230     }
231
```

```

232      /*
233       * We may be zeroing pages we're about to discard, but it avoids
234       * doing a complex calculation here, and then doing the zeroing
235       * anyway if the page split fails.
236       */
237      folio_zero_range(folio, offset, length);
238
239      if (folio_has_private(folio))
240          folio_invalidate(folio, offset, length);
241      if (!folio_test_large(folio))
242          return true;
243      :
249  }

```

Listing 9-104: mm/truncate.c: truncate_inode_partial_folio()

This function handles the portion of truncated folios which are not page-aligned. Obviously in this case we cannot actually remove entries from the page cache, so we zero the data in the range instead.

We start by determining the offset of the start within the file, and clamping the length of the operation within the folio, we wait for any ongoing writeback via `folio_wait_writeback()` (see Listing 9-146, and Section 9.11 for more on this) before checking the edge case that we do, in fact, span the entire folio, in which case we truncate it via `truncate_inode_folio()` (see Listing 9-102) and exit.

Otherwise, we zero the portion of the range required via `folio_zero_range()`, before completing the same invalidate check as in `truncate_cleanup_folio()` (see discussion around Listing 9-103 above).

Finally, there is logic pertaining to huge page handling, however this is out of scope for the book so we don't examine this part of the function.

9.9.3 Unmapping Folios

An important part of truncation is dealing with processes which currently have these page cache entries mapped.

We handle this by simply unmapping all truncated folios for all folios which are mapped.

If we want to unmap folios over the span of a byte range, we utilise `unmap_mapping_range()`, which we examine in Listing ??.

```

3575 /**
3576  * unmap_mapping_range - unmap the portion of all mmmaps in the specified
3577  * address_space corresponding to the specified byte range in the underlying
3578  * file.
3579  *
3580  * @mapping: the address space containing mmmaps to be unmapped.
3581  * @holebegin: byte in first page to unmap, relative to the start of
3582  * the underlying file. This will be rounded down to a PAGE_SIZE
3583  * boundary. Note that this is different from truncate_pagecache(), which

```

```

3584  * must keep the partial page. In contrast, we must get rid of
3585  * partial pages.
3586  * @holelen: size of prospective hole in bytes. This will be rounded
3587  * up to a PAGE_SIZE boundary. A holelen of zero truncates to the
3588  * end of the file.
3589  * @even_cows: 1 when truncating a file, unmap even private COWed pages;
3590  * but 0 when invalidating pagecache, don't throw away private data.
3591  */
3592  void unmap_mapping_range(struct address_space *mapping,
3593                          loff_t const holebegin, loff_t const holelen, int even_cows)
3594  {
3595      pgoff_t hba = holebegin >> PAGE_SHIFT;
3596      pgoff_t hlen = (holelen + PAGE_SIZE - 1) >> PAGE_SHIFT;
3597
3598      /* Check for overflow. */
3599      if (sizeof(holelen) > sizeof(hlen)) {
3600          long long holeend =
3601              (holebegin + holelen + PAGE_SIZE - 1) >> PAGE_SHIFT;
3602          if (holeend & ~(long long)ULONG_MAX)
3603              hlen = ULONG_MAX - hba + 1;
3604      }
3605
3606      unmap_mapping_pages(mapping, hba, hlen, even_cows);
3607  }

```

Listing 9-105: mm/memory.c: unmap_mapping_range()

After performing some housekeeping to ensure overflow doesn't occur, the operation is ultimately deferred to `unmap_mapping_pages()`, which we examine below in Listing 9-107.

This is parameterised by `even_cow` to indicate whether Copy-on-Write folios should also be unmapped in this range.

If we wish to unmap a single folio, we can do so via `unmap_mapping_folio()`, which we examine in Listing 9-106 (eliding a debug check).

```

3510  /**
3511  * unmap_mapping_folio() - Unmap single folio from processes.
3512  * @folio: The locked folio to be unmapped.
3513  *
3514  * Unmap this folio from any userspace process which still has it mmaped.
3515  * Typically, for efficiency, the range of nearby pages has already been
3516  * unmapped by unmap_mapping_pages() or unmap_mapping_range(). But once
3517  * truncation or invalidation holds the lock on a folio, it may find that
3518  * the page has been remapped again: and then uses unmap_mapping_folio()
3519  * to unmap it finally.
3520  */
3521  void unmap_mapping_folio(struct folio *folio)
3522  {
3523      struct address_space *mapping = folio->mapping;

```

```

3524     struct zap_details details = { };
3525     pgoff_t first_index;
3526     pgoff_t last_index;
3527     :
3530     first_index = folio->index;
3531     last_index = folio->index + folio_nr_pages(folio) - 1;
3532
3533     details.even_cows = false;
3534     details.single_folio = folio;
3535     details.zap_flags = ZAP_FLAG_DROP_MARKER;
3536
3537     i_mmap_lock_read(mapping);
3538     if (unlikely(!RB_EMPTY_ROOT(&mapping->i_mmap.rb_root)))
3539         unmap_mapping_range_tree(&mapping->i_mmap, first_index,
3540                                 last_index, &details);
3541     i_mmap_unlock_read(mapping);
3542 }

```

Listing 9-106: mm/memory.c: unmap_mapping_folio()

We prepare ourselves for the mapping removal (termed ‘zapping’), specifying the ZAP_FLAG_DROP_MARKER flag to indicate that we are removing the folio entirely.

While we examine the process of zapping in detail in Section 7.1.4, in this section we will briefly explore this topic for the purposes of observing the unmapping of page cache entries utilising this functionality.

if the page cache object’s red/black tree rooted in struct address_space->i_mmap is empty then there is nothing to do, otherwise we defer the operation to unmap_mapping_range_tree(), which we examine in Listing 9-108.

See the reverse mapping chapter for more details on this tree structure.

Next we examine unmap_mapping_pages() which performs the unmapping of a page range. We examine this in Listing 9-107.

```

3544 /**
3545  * unmap_mapping_pages() - Unmap pages from processes.
3546  * @mapping: The address space containing pages to be unmapped.
3547  * @start: Index of first page to be unmapped.
3548  * @nr: Number of pages to be unmapped. 0 to unmap to end of file.
3549  * @even_cows: Whether to unmap even private COWed pages.
3550  *
3551  * Unmap the pages in this address space from any userspace process which
3552  * has them mmaped. Generally, you want to remove COWed pages as well when
3553  * a file is being truncated, but not when invalidating pages from the page
3554  * cache.
3555  */
3556 void unmap_mapping_pages(struct address_space *mapping, pgoff_t start,
3557                         pgoff_t nr, bool even_cows)
3558 {

```

```

3559     struct zap_details details = { };
3560     pgoff_t first_index = start;
3561     pgoff_t last_index = start + nr - 1;
3562
3563     details.even_cows = even_cows;
3564     if (last_index < first_index)
3565         last_index = ULONG_MAX;
3566
3567     i_mmap_lock_read(mapping);
3568     if (unlikely(!RB_EMPTY_ROOT(&mapping->i_mmap.rb_root)))
3569         unmap_mapping_range_tree(&mapping->i_mmap, first_index,
3570                                 last_index, &details);
3571     i_mmap_unlock_read(mapping);
3572 }

```

Listing 9-107: mm/memory.c: unmap_mapping_pages()

This defers the operation to `unmap_mapping_range_tree()`, being careful to handle an overflow scenario (unmapping from the start onwards if one is detected).

We examine `unmap_mapping_range_tree()` in Listing 9-108.

```

3489 static inline void unmap_mapping_range_tree(struct rb_root_cached *root,
3490                                             pgoff_t first_index,
3491                                             pgoff_t last_index,
3492                                             struct zap_details *details)
3493 {
3494     struct vm_area_struct *vma;
3495     pgoff_t vba, vea, zba, zea;
3496
3497     vma_interval_tree_foreach(vma, root, first_index, last_index) {
3498         vba = vma->vm_pgoff;
3499         vea = vba + vma_pages(vma) - 1;
3500         zba = max(first_index, vba);
3501         zea = min(last_index, vea);
3502
3503         unmap_mapping_range_vma(vma,
3504                                 ((zba - vba) << PAGE_SHIFT) + vma->vm_start,
3505                                 ((zea - vba + 1) << PAGE_SHIFT) + vma->vm_start,
3506                                 details);
3507     }
3508 }

```

Listing 9-108: mm/memory.c: unmap_mapping_range_tree()

This iterates through each of the `struct vm_area_struct` (VMA) objects which potentially map this folio via `vma_interval_tree_foreach()` (see Listing 7-42 in Section in the reverse mapping chapter for more details on this).

For each of these, we attempt to perform the actual unmapping via `unmap_mapping_range_vma()` which we examine in Listing 9-109.

```

3482 static void unmap_mapping_range_vma(struct vm_area_struct *vma,
3483                                   unsigned long start_addr, unsigned long end_addr,
3484                                   struct zap_details *details)
3485 {
3486     zap_page_range_single(vma, start_addr, end_addr - start_addr, details)
3487     ;
3488 }

```

Listing 9-109: mm/memory.c: unmap_mapping_range_vma()

This simply wraps `zap_page_range_single()` which we examine in Listing 9-110 (eliding out of scope MMU notifier logic).

```

1761 /**
1762  * zap_page_range_single - remove user pages in a given range
1763  * @vma: vm_area_struct holding the applicable pages
1764  * @address: starting address of pages to zap
1765  * @size: number of bytes to zap
1766  * @details: details of shared cache invalidation
1767  *
1768  * The range must fit into one VMA.
1769  */
1770 static void zap_page_range_single(struct vm_area_struct *vma, unsigned long
1771                                address,
1772                                unsigned long size, struct zap_details *details)
1773 {
1774     :
1775     struct mmu_gather tlb;
1776     lru_add_drain();
1777     :
1778     tlb_gather_mmu(&tlb, vma->vm_mm);
1779     update_hiwater_rss(vma->vm_mm);
1780     :
1781     unmap_single_vma(&tlb, vma, address, range.end, details);
1782     :
1783     tlb_finish_mmu(&tlb);
1784 }
1785 }

```

Listing 9-110: mm/memory.c: zap_page_range_single()

This invokes `unmap_single_vma()` (see Listing 7-59 in Section 7.1.3), performing TLB invalidation utilising `tlb_gather_mmu()` (see Listing 7-54 in Section 7.1.2), and complete the TLB invalidation in `tlb_finish_mmu()` (see Listing 7-79 in Section 7.1.4), using `update_hiwater_rss()` to update the high watermark of RSS memory usage accordingly.

At this point, all processes which map the truncated mapping will now have it unmapped and will thus page fault if accessing a file within the truncated range.

9.9.4 Dropping Caches

The `vm.drop_caches` tunable is a means through which the kernel will ‘drop’ (i.e. free) all *clean* page cache pages (as well as optionally dropping all cached slab entries, however this is out of scope for this chapter).

A typical use of this is measuring the performance of a program which interacts with I/O with a desire to eliminate measurement noise, shown in Listing 9-111.

```
$ echo 1 | sudo tee /proc/sys/vm/drop_caches
$ sudo sysctl vm.drop_caches=1 # alternative
```

Listing 9-111: Example use of `drop_caches` interface

This is not something that a user should ordinarily do in a production system, as caches will be reclaimed when the system comes under memory pressure (see the reclaim chapter), so clearing this caches will only degrade system performance.

Let’s have a look at the kernel implementation – the `sysctl` is configured in the `vm_table` object and interactions with it are set to be handled by `drop_caches_sysctl_handler()`, shown in Listing 9-112.

```
50 int drop_caches_sysctl_handler(struct ctl_table *table, int write,
51                               void *buffer, size_t *length, loff_t *ppos)
52 {
53     int ret;
54
55     ret = proc_dointvec_minmax(table, write, buffer, length, ppos);
56     if (ret)
57         return ret;
58     if (write) {
59         static int stfu;
60
61         if (sysctl_drop_caches & 1) {
62             iterate_supers(drop_pagecache_sb, NULL);
63             count_vm_event(DROP_PAGECACHE);
64         }
65         if (sysctl_drop_caches & 2) {
66             drop_slab();
67             count_vm_event(DROP_SLAB);
68         }
69         if (!stfu) {
70             pr_info("%s (%d): drop_caches: %d\n",
71                   current->comm, task_pid_nr(current),
72                   sysctl_drop_caches);
73         }
74         stfu |= sysctl_drop_caches & 4;
75     }
76     return 0;
77 }
```

Listing 9-112: fs/drop_caches.c: drop_caches_sysctl_handler()

The `proc_dointvec_minmax()` function simply retrieves parameters from the `sysctl` invocation which are used to determine how to proceed.

If the user passes a value which sets bit 0, then the page cache is cleared. If bit 1 is set slab caches are dropped via `drop_slab()`. Finally setting bit 2 limits kernel log output.

The function `iterate_supers()` executes a function over all *active* super blocks, shown in Listing 9-113.

```

702 /**
703  *      iterate_supers - call function for all active superblocks
704  *      @f: function to call
705  *      @arg: argument to pass to it
706  *
707  *      Scans the superblock list and calls given function, passing it
708  *      locked superblock and given argument.
709  */
710 void iterate_supers(void (*f)(struct super_block *, void *), void *arg)
711 {
712     struct super_block *sb, *p = NULL;
713
714     spin_lock(&sb_lock);
715     list_for_each_entry(sb, &super_blocks, s_list) {
716         if (hlist_unhashed(&sb->s_instances))
717             continue;
718         sb->s_count++;
719         spin_unlock(&sb_lock);
720
721         down_read(&sb->s_umount);
722         if (sb->s_root && (sb->s_flags & SB_BORN))
723             f(sb, arg);
724         up_read(&sb->s_umount);
725
726         spin_lock(&sb_lock);
727         if (p)
728             __put_super(p);
729         p = sb;
730     }
731     if (p)
732         __put_super(p);
733     spin_unlock(&sb_lock);
734 }
```

Listing 9-113: fs/super.c: iterate_supers()

The `super_blocks` list is a global variable protected by `sb_lock` containing all super blocks in the system (as shown in Figure 9-1), tied together by the `s_list` field.

The `struct super_block->s_instances` field is the node threaded through super blocks tying them to a `struct file_system_type`. If it is unhashed and thus doesn't belong to any known file system, we skip this entry.

We increment the super block's reference count `s_count` directly, before dropping the super block list lock `sb_lock` in order to perform the callback on the superblock without contending this lock.

We also drop the `s_umount` semaphore to prevent the file system from being unmounted beneath us as we do this.

We only reference a super block if the `SB_BORN` flag is set, indicating that the file system is mounted.

After the operation is done, we drop the reference count of the previously referenced super block via `__put_super()` (this is delayed to avoid the need for a safe list iteration).

The function we iterate over is `drop_pagecache_sb()`, shown in Listing 9-114.

```

18 static void drop_pagecache_sb(struct super_block *sb, void *unused)
19 {
20     struct inode *inode, *toput_inode = NULL;
21
22     spin_lock(&sb->s_inode_list_lock);
23     list_for_each_entry(inode, &sb->s_inodes, i_sb_list) {
24         spin_lock(&inode->i_lock);
25         /*
26          * We must skip inodes in unusual state. We may also skip
27          * inodes without pages but we deliberately won't in case
28          * we need to reschedule to avoid softlockups.
29          */
30         if ((inode->i_state & (I_FREEING|I_WILL_FREE|I_NEW)) ||
31             (mapping_empty(inode->i_mapping) && !need_resched())) {
32             spin_unlock(&inode->i_lock);
33             continue;
34         }
35         __iget(inode);
36         spin_unlock(&inode->i_lock);
37         spin_unlock(&sb->s_inode_list_lock);
38
39         invalidate_mapping_pages(inode->i_mapping, 0, -1);
40         iput(toput_inode);
41         toput_inode = inode;
42
43         cond_resched();
44         spin_lock(&sb->s_inode_list_lock);
45     }
46     spin_unlock(&sb->s_inode_list_lock);

```

```

47         iput(toput_inode);
48     }

```

Listing 9-114: fs/drop_caches.c: drop_pagecache_sb()

The struct `super_block->s_inode_list_lock` lock is held in order to access the super block's `s_inodes` list, after which the inode's individual struct `inode` lock is acquired.

If the inode is in an 'unusual' state, e.g. one of the below cases apply, then it is skipped:

- Its `i_state` has the `I_FREEING` flag set – This indicates that the inode is about to be freed, pending dirty data being written back.
- Its `i_state` has the `I_WILL_FREE` flag set – This indicates that the inode is about to be freed (i.e. set the `I_FREEING` flag).
- Its `i_state` has the `I_NEW` flag set – This indicates that the inode is newly created and may be incorrectly duplicated. The inode is only stable once this flag is cleared and this is used to address races in inode creation.
- The struct `address_space` page cache object associated with the inode does not map any pages (as determined by `mapping_empty()`) and rescheduling is not required – freeing inodes would be redundant here.

Special care is taken to avoid contention on `s_inode_list_lock`, as many such objects may be traversed in some circumstances which can lead to soft lookup, therefore if a reschedule is required, we proceed (thereby dropping this lock). See commit `c27d82f52f75: fs/drop_caches.c: avoid softlockups in drop_pagecache_sb()` for details.

If the inode passes these tests, we increment its reference count via `__iget()` and drop its lock (since now it cannot disappear from beneath us) as well as the struct `super_block->s_inode_list_lock`.

The actual freeing of the cache is accomplished via `invalidate_mapping_pages()`. We take special care to drop the inode reference count for the *previous* inode in order to avoid a lock inversion, as well as allowing the process to be rescheduled with locks dropped via `cond_resched()` (relevant only to kernels without full preemption).

NOTE

A lock (ordering) inversion occurs when the ordering of two or more acquired locks is unintentionally reversed between locking and unlocking, causing a deadlock.

We examine the `invalidate_mapping_pages()` function in detail in Listing 9-115.

```

550  /**
551   * invalidate_mapping_pages - Invalidate all clean, unlocked cache of one
552   *                           inode
553   * @mapping: the address_space which holds the cache to invalidate
554   * @start: the offset 'from' which to invalidate
555   * @end: the offset 'to' which to invalidate (inclusive)

```

```

555  *
556  * This function removes pages that are clean, unmapped and unlocked,
557  * as well as shadow entries. It will not block on IO activity.
558  *
559  * If you want to remove all the pages of one inode, regardless of
560  * their use and writeback state, use truncate_inode_pages().
561  *
562  * Return: the number of the cache entries that were invalidated
563  */
564 unsigned long invalidate_mapping_pages(struct address_space *mapping,
565                                       pgoff_t start, pgoff_t end)
566 {
567     return invalidate_mapping_pagevec(mapping, start, end, NULL);
568 }

```

Listing 9-115: mm/truncate.c: `invalidate_mapping_pages()`

This defers the operation to `invalidate_mapping_pagevec()`, shown in Listing 9-116.

```

491 /**
492  * invalidate_mapping_pagevec - Invalidate all the unlocked pages of one inode
493  * @mapping: the address_space which holds the pages to invalidate
494  * @start: the offset 'from' which to invalidate
495  * @end: the offset 'to' which to invalidate (inclusive)
496  * @nr_pagevec: invalidate failed page number for caller
497  *
498  * This helper is similar to invalidate_mapping_pages(), except that it
499  * accounts
500  * for pages that are likely on a pagevec and counts them in @nr_pagevec,
501  * which
502  * will be used by the caller.
503  */
504 unsigned long invalidate_mapping_pagevec(struct address_space *mapping,
505                                       pgoff_t start, pgoff_t end, unsigned long *nr_pagevec)
506 {
507     pgoff_t indices[PAGEVEC_SIZE];
508     struct folio_batch fbatch;
509     pgoff_t index = start;
510     unsigned long ret;
511     unsigned long count = 0;
512     int i;
513
514     folio_batch_init(&fbatch);
515     while (find_lock_entries(mapping, index, end, &fbatch, indices)) {
516         for (i = 0; i < folio_batch_count(&fbatch); i++) {
517             struct folio *folio = fbatch.folios[i];

```

```

/* We rely upon deletion not changing folio->index */

```

```

518             index = indices[i];
519
520             if (xa_is_value(folio)) {
521                 count += invalidate_exceptional_entry(mapping,
522                                                         index,
523                                                         folio);
524                 continue;
525             }
526             index += folio_nr_pages(folio) - 1;
527
528             ret = mapping_evict_folio(mapping, folio);
529             folio_unlock(folio);
530             /*
531              * Invalidation is a hint that the folio is no longer
532              * of interest and try to speed up its reclaim.
533              */
534             if (!ret) {
535                 deactivate_file_folio(folio);
536                 /* It is likely on the pagevec of a remote CPU
537
538
539                 if (nr_pagevec)
540                     (*nr_pagevec)++;
541             }
542             count += ret;
543         }
544         folio_batch_remove_exceptionals(&fbatch);
545         folio_batch_release(&fbatch);
546         cond_resched();
547         index++;
548     }
549     return count;
550 }

```

Listing 9-116: mm/truncate.c: `invalidate_mapping_pagevec()`

This function retrieves folios a struct `folio_batch` at a time (see Section 11.7 for a broader discussion of these) using `find_lock_entries()`.

We will examine `find_lock_entries()` in Section 9.5.1 and in Listing 9-42, however as the name suggests, it finds folios associated with the mapping (i.e. struct `address_space` object) in the given range, which are locked (i.e. have the `PG_locked` folio flag applied).

We will also examine folio locking in Section 9.11, but broadly speaking this causes any other thread simultaneously performing an operation on the folio to sleep until the folio is unlocked.

We process each of the now locked file folios one at a time in order to evict them from memory. For each folio we:

- Handle *exceptional* entries that might be present in the returned folio batch which contain a *shadow* entry for an already evicted folio. This is

determined by `xa_is_value()` (see Section 9.2), which checks whether the `xarray` entry returned is tagged as a value rather than a pointer.

- Attempt to remove the entry from the page cache via `mapping_evict_folio()` (we will examine this in Listing 9-117 shortly). If the eviction failed, the fact we wanted to remove it in the first instance means it is sensible to *deactivate* the folio so it is reclaimed sooner which we do via `deactivate_file_folio()` (see chapter 11 for more on reclaim and active/inactive folios).

Finally, after we have processed all folios, we eliminate any ‘exceptional’ entries via `folio_batch_remove_exceptionals()` in order to be able to pass the batch to `folio_batch_release()` which ultimately frees all of the underlying physical pages via `release_pages()`.

NOTE

It may seem confusing to reference ‘shadow’ and ‘exceptional’ entries, however for the time being you should simply gloss over these points and note that these are edge cases that we must address here. Exceptional entries are a product of ‘eXtensible Arrays’ (see Section 9.2) and shadow entries part of the working set logic (out of scope for the book).

9.9.5 Folio Eviction

When dropping caches, the `mapping_evict_folio()` function is used to evict folios which have had their caches dropped. We examine this in Listing 9-117.

```

270 static long mapping_evict_folio(struct address_space *mapping,
271                               struct folio *folio)
272 {
273     if (folio_test_dirty(folio) || folio_test_writeback(folio))
274         return 0;
275     /* The refcount will be elevated if any page in the folio is mapped */
276     if (folio_ref_count(folio) >
277         folio_nr_pages(folio) + folio_has_private(folio) + 1)
278         return 0;
279     if (folio_has_private(folio) && !filemap_release_folio(folio, 0))
280         return 0;
281
282     return remove_mapping(mapping, folio);
283 }
```

Listing 9-117: `mm/truncate.c: mapping_evict_folio()`

This function assumes the folio passed in is already locked.

If the folio is dirty or is undergoing writeback, then there is nothing to do and we duly abort the operation. In either case, we would experience data loss if the folio were to be evicted at this stage.

We also check see if the folio is pinned by the kernel, i.e. has an elevated reference count compared to what would be expected it were only referenced by userland.

If the folio has private data associated with it, this implies that custom release logic must be applied on an invalidation like this, which we perform in `filemap_release_folio()`.

This invokes the struct `address_space->a_ops->release_folio()` handler if it exists (from the struct `address_space_operations` object specified by the file system for custom handling).

Finally, the folio is removed from the page cache altogether via `remove_mapping()`, which is part of the reclaim logic used for removing folios under memory pressure, and whose logic we share. We examine this in Listing ?? in the reclaim chapter.

9.10 Buffers and Block I/O

While the Page Cache abstracts file operations from the underlying disk, rubber must ultimately meet road and (if not a RAM-backed mapping) I/O must be performed in order to actually retrieve data from or write data to disk. Typically this will be via the use of a *block device*. That is a storage device that operates at a granularity of *blocks*.

The smallest size a block can be is 512 bytes, and the maximum it can be is the base page size, which is 4 KiB on x86-64. This is therefore referred to as the device's *block size*.

A block device can be envisioned as an array of blocks, with the index of each block within the array referred to as its *block number*. Abstracting devices in this way is termed Logical Block Addressing (LBA).

We thus must be able to translate between page cache entries' folios and the block numbers of the blocks spanned by them.

The fundamental data structure used to map each block within a page is struct `buffer_head`. This has historically been used for both this mapping and as the fundamental unit of block I/O operations.

However in the modern kernel this data structure is used only for mapping purposes, with block device I/O operations encoded in struct `bio` objects which we will come on to in Section 9.10.5.

9.10.1 An Introduction to Buffer Heads

The struct `buffer_head` data structure encodes a mapping between blocks and memory pages.

This data structure is a rather unloved corner of the kernel and file systems do not *have* to use them, but many do, although for far less than they did in the past.

The primary modern purpose of buffer heads is to map blocks (which can be, for instance, of size 512 bytes, less than page size) to positions within pages. There are alternative mechanisms for this such as the `iomap` function-

ality implemented in the `include/linux/iomap.h` header but discussion of this is out of scope for the book.

We examine `struct buffer_head` in Listing 9-118.

```

52  /*
53   * Historically, a buffer_head was used to map a single block
54   * within a page, and of course as the unit of I/O through the
55   * filesystem and block layers. Nowadays the basic I/O unit
56   * is the bio, and buffer_heads are used for extracting block
57   * mappings (via a get_block_t call), for tracking state within
58   * a page (via a page_mapping) and for wrapping bio submission
59   * for backward compatibility reasons (e.g. submit_bh).
60   */
61  struct buffer_head {
62      unsigned long b_state;           /* buffer state bitmap (see above) */
63      struct buffer_head *b_this_page; /* circular list of page's buffers */
64      struct page *b_page;             /* the page this bh is mapped to */
65
66      sector_t b_blocknr;              /* start block number */
67      size_t b_size;                   /* size of mapping */
68      char *b_data;                    /* pointer to data within the page */
69
70      struct block_device *b_bdev;
71      bh_end_io_t *b_end_io;           /* I/O completion */
72      void *b_private;                 /* reserved for b_end_io */
73      struct list_head b_assoc_buffers; /* associated with another mapping
74                                          */
75      struct address_space *b_assoc_map; /* mapping this buffer is
76                                          associated with */
77      atomic_t b_count;                /* users using this buffer_head */
78      spinlock_t b_uptodate_lock;      /* Used by the first bh in a page, to
79                                          * serialise IO completion of other
80                                          * buffers in the page */
81  };

```

Listing 9-118: `include/linux/buffer_head.h`: `struct buffer_head`

The key field `b_state` contains a bit set of enum `bh_state_bits` fields describing the buffer state as shown in Listing 9-119.

```

21  enum bh_state_bits {
22      BH_Uptodate,    /* Contains valid data */
23      BH_Dirty,       /* Is dirty */
24      BH_Lock,        /* Is locked */
25      BH_Req,         /* Has been submitted for I/O */
26
27      BH_Mapped,       /* Has a disk mapping */
28      BH_New,          /* Disk mapping was newly created by get_block */
29      BH_Async_Read,   /* Is under end_buffer_async_read I/O */

```

```

30     BH_Async_Write, /* Is under end_buffer_async_write I/O */
31     BH_Delay,       /* Buffer is not yet allocated on disk */
32     BH_Boundary,    /* Block is followed by a discontinuity */
33     BH_Write_EIO,   /* I/O error on write */
34     BH_Unwritten,   /* Buffer is allocated on disk but not written */
35     BH_Quiet,       /* Buffer Error Prints to be quiet */
36     BH_Meta,        /* Buffer contains metadata */
37     BH_Prio,        /* Buffer should be submitted with REQ_PRIO */
38     BH_Defer_Completion, /* Defer AIO completion to workqueue */
39
40     BH_PrivateStart, /* not a state bit, but the first bit available
41                      * for private allocation by other entities
42                      */
43 };

```

Listing 9-119: include/linux/buffer_head.h: enum bh_state_bits

Considering each field in struct `buffer_head`:

b_state Describes the current state of the *buffer*, i.e. the region of memory mapped to the associated on-disk block, consisting of a bitwise-or combined bitmap of enum `bh_state_bits` flags as shown in Listing 9-119.

b_this_page If the block size is less than that of the page, then multiple buffer heads will be required to describe the blocks containing the page's data. This field is used to tie them together, in a circular singly-linked list.

b_page The struct `page` the blocks map to.

b_blocknr The block number that this mapping starts at.

b_size The size of the mapping, in bytes.

b_data A convenient field containing a pointer to the offset within the page to which this block maps. The kernel typically uses the direct mapping to make this easy via `set_bh_page()`.

b_bdev A reference to the object describing the block device of type struct `block_device`.

b_end_io A callback invoked when I/O is completed of type `bh_end_io_t` with signature `void (bh_end_io_t)(struct buffer_head *bh, int uptodate)`. This is called passing the buffer head and a boolean indicating whether the I/O operation managed to synchronise the block with the disk.

b_private A void `*` user-defined field made available to the `b_end_io` callback to pass any required additional state.

b_assoc_buffers A struct `list_head` node enabling the buffer head to be kept in another linked list.

b_assoc_map The struct `address_space` page cache mapping object within which this buffer and page reside.

b_count An atomic reference count of the number of users of this buffer head. Incremented by `get_bh()` and decremented by `put_bh()`.

b_uptodate_lock A lock, present in the first buffer head within the `b_this_page` linked list, used to ensure serialisation between multiple buffers within a page when I/O completes in order that the underlying page is only marked `PG_uptodate` when all blocks are up to date. Used for instance in `end_buffer_async_read()` and `end_buffer_async_write()`, both of which are used as `b_end_io` callbacks. Some file systems use this for the same purpose.

Buffer heads are allocated from a slab cache in `alloc_buffer_head()` (via `kmem_cache_zalloc()`) and are always attached to a folio in its `struct folio->private` field.

Not every page cache folio needs to have a buffer head attached, though precisely how this operates is up to the file system. However, reads into the page cache *typically* do not cause buffer heads to get attached a folio, but writes do unless the file system uses some other method for tracking writes.

This is because data read into the page cache is only accessible by user-land (signified by the `uptodate` flag) once every block that comprises a folio is read. Since we encapsulate actual I/O operations in `struct bio` objects (see Section 9.10.5 below), which specifies a callback to be invoked when done, we obviate the need for mapping blocks to folios, since when the read is complete this information is no longer required.

Writes are different. These are ongoing, occur after a folio has been made available to userspace and are performed on a block level, so we need some means of tracking the current state of writes per-block. Therefore, we *do* attach buffer heads to folios when writing data if the file system uses buffer heads.

9.10.2 *blockdev*

We've discussed the relationship between disk blocks and file page cache entries, however in order for a file system to function correctly, it must be able to access metadata on disk which does not form part of the data associated with any file. Equally, the kernel makes it possible to access the *raw* data contained on a disk via the device file representing that disk's block device, e.g. `/dev/sda1` as a typical example of such a device.

In both cases we require block-level access to the data contained on the disk, and it would be sensible to keep this data in the page cache. This is achieved via the special *blockdev* file system.

Each block device is described by the `struct block_device` type, which is allocated and initialised by `add_partition()`. Each partition on a disk comprises an individual block device.

The actual allocation of the block device is performed by `bdev_alloc()` which allocates an inode in the *blockdev* file system (represented by the special `blockdev_superblock` super block). This inode, and more importantly its `struct address_space` page cache mapping, represent the *entire* block device and is placed in `struct block_device->bd_inode`.

Unlike a typical file system where the inode number is assigned on creation the blockdev file system, existing as an abstract and ephemeral in-memory entity, assigns the inode number `struct inode->i_ino` to be equal to the *device number* of the block device in `bdev_add()`, as shown in Listing 9-120.

NOTE

A device number, consisting of a major and a minor number pair, describes a device in the system and are represented in the `devtmpfs` file system typically mounted at `/dev` as special files whose major and minor can be viewed via `ls -l`.

```

507 void bdev_add(struct block_device *bdev, dev_t dev)
508 {
509     bdev->bd_dev = dev;
510     bdev->bd_inode->i_rdev = dev;
511     bdev->bd_inode->i_ino = dev;
512     insert_inode_hash(bdev->bd_inode);
513 }
```

Listing 9-120: `block/bdev.c: bdev_add()`

When the inode for a block device file such as `/dev/sda1` is created, `init_special_inode()` is invoked which assigns it the `def_blk_fops` `struct file_operations` callbacks.

This is important, because this inode is *not* the same inode as the one created on block device initialisation and placed in `struct block_device->bd_inode` in `bdev_alloc()`.

In order for the device to access the same page cache entry as the block device's regardless of which device file is used the `def_blk_fops->open` function, `blkdev_open()` sets up the opened `struct file` object accordingly as shown in Listing 9-121 (eliding irrelevant code).

```

465 static int blkdev_open(struct inode *inode, struct file *filp)
466 {
467     struct block_device *bdev;
468     :
485     bdev = blkdev_get_by_dev(inode->i_rdev, filp->f_mode, filp);
486     if (IS_ERR(bdev))
487         return PTR_ERR(bdev);
488
489     filp->private_data = bdev;
490     filp->f_mapping = bdev->bd_inode->i_mapping;
491     filp->f_wb_err = filemap_sample_wb_err(filp->f_mapping);
492     return 0;
493 }
```

Listing 9-121: `block/fops.c: blkdev_open()`

This invokes `blkdev_get_by_dev()` which in turn calls `blkdev_get_no_open()` which uses `ilookup()` to lookup the inode whose number we assigned to device number in `bdev_add()` (see Listing 9-120).

We assign the block device's struct `block_device->bd_inode->i_mapping` struct `address_space` page cache mapping object to struct `file->f_mapping` and set the struct `file->private_data` field to the struct `block_device` object.

This is important, as the device file inode essentially *points* at the block device, and thus each device file is meaningless in itself, only the contents of the block device itself matter.

9.10.2.1 blockdev Buffers

The blockdev filesystem exposes the entirety of the raw data contained within the partition, directly accessible via a flat file in Logical Block Addressing fashion.

This is critical as by simply calculating the number of blocks per page in the file system, we now have the means to directly access the block device at any arbitrary block number.

NOTE

Rather irritatingly, the kernel often denominates filesystem related indexes in terms of 'logical sectors', hardcoded to 512 bytes each, of type `sector_t`, rather than more sensibly subdividing by block or bytes. Adding to this is the fact that the kernel is not always consistent in doing so, nor does the use of the `sector_t` type necessarily indicate that a value is expressed in sectors. Therefore, when examining file system code, take care to ensure any values you observe are expressed in the units you expect.

The page cache entries containing these block device pages exist entirely in parallel, and are duplicates of, any already existing in the page cache for the files which they back.

Therefore it is ill-advised to directly access this data when a filesystem is mounted, especially if writing to the block device directly, a write which might race against those from dirty file pages.

NOTE

However, from a user's perspective, directly accessing block devices when unmounted is very useful. This enables operations such as writing a disk image to a USB stick to be performed very easily.

However, the most useful aspect of having this interface from a file system's perspective is the ability to access those blocks on the underlying disk which contain metadata, which would not otherwise be readily accessible.

The page cache entries containing this raw block data are the *buffers* referred to in `/proc/meminfo` and thus the free command line tool, as determined by `meminfo_proc_show()` and `si_meminfo()` in turn. This value is calculated via `nr_blockdev_pages()` shown in Listing 9-122.

```

515 long nr_blockdev_pages(void)
516 {
517     struct inode *inode;
518     long ret = 0;
519
520     spin_lock(&blockdev_superblock->s_inode_list_lock);
```

```

521     list_for_each_entry(inode, &blockdev_superblock->s_inodes, i_sb_list)
522         ret += inode->i_mapping->npages;
523     spin_unlock(&blockdev_superblock->s_inode_list_lock);
524
525     return ret;
526 }

```

Listing 9-122: block/bdev.c: `nr_blockdev_pages()`

You can observe that this walks all blockdev inodes (which are mapped by device), accumulating struct `address_space->npages` allocated page counts.

9.10.3 Accessing Blocks

There are a number of functions which leverage blockdev in order to directly lookup blocks on the backing block device for a filesystem in order to be able to retrieve filesystem metadata. These operate in terms of the struct `buffer_head` type, which returns data describing the block looked up and are shown in Figure 9-14.

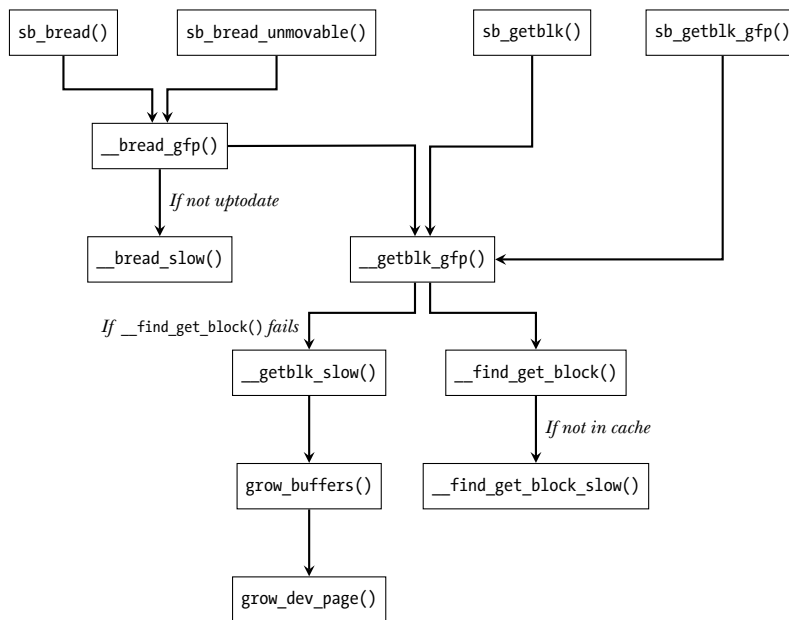


Figure 9-14: Accessing Disk Blocks

NOTE

As with much in the filesystem code, filesystems can implement their own versions of these functions. For instance, `ext4` implements their own `ext4_sb_bread()` function which performs the work of `sb_bread()` with changes made specific to `ext4`. These functions should be interpreted therefore as utility functions for performing block reads rather than a canonical implementation.

Broadly speaking, Figure 9-14 can be divided into those functions which wish to retrieve a struct `buffer_head` describing a block within a buffer, and

expect the buffer to be read into the page cache if the page cache entry is not uptodate—`sb_bread()` (shown in Listing 9-123) and `sb_bread_unmovable()` (shown in Listing 9-124)—and those which simply retrieve a buffer head for the block even if the buffer is not uptodate—`sb_getblk()` (shown in Listing 9-125) and `sb_getblk_gfp()` (shown in Listing 9-126).

In either case, if the buffer page is not present or does not have buffer heads attached, a slow patch is taken to allocate memory for these in `__getblk_slow()` (see Listing ??).

The variants of the initial functions determine how any allocation of buffer heads, if necessary, should be performed, with `sb_bread_unmovable()` used when the allocation must be unmovable and `sb_getblk_gfp()` used when the ability to specify GFP physical allocation flags is required (see Section 2.6 for details on these).

```

336 static inline struct buffer_head *
337 sb_bread(struct super_block *sb, sector_t block)
338 {
339     return __bread_gfp(sb->s_bdev, block, sb->s_blocksize, __GFP_MOVABLE);
340 }
```

Listing 9-123: include/linux/buffer_head.h: `sb_bread()`

This function defers the heavy lifting to `__bread_gfp()` (see Listing 9-127), looking up the struct `block_device` block device upon which the super block resides via the struct `super_block->s_bdev` field, as well as its blocksize, the sought block, and the `__GFP_MOVABLE` physical page flag indicating that the buffer head allocation can be movable.

```

342 static inline struct buffer_head *
343 sb_bread_unmovable(struct super_block *sb, sector_t block)
344 {
345     return __bread_gfp(sb->s_bdev, block, sb->s_blocksize, 0);
346 }
```

Listing 9-124: include/linux/buffer_head.h: `sb_bread_unmovable()`

`sb_bread_unmovable()` does the same thing as `sb_bread()`, only does not specify `__GFP_MOVABLE`, which will result in an unmovable struct `buffer_head` allocation.

```

360 static inline struct buffer_head *
361 sb_getblk(struct super_block *sb, sector_t block)
362 {
363     return __getblk_gfp(sb->s_bdev, block, sb->s_blocksize, __GFP_MOVABLE)
364     ;
365 }
```

Listing 9-125: include/linux/buffer_head.h: `sb_getblk()`

This function defers its heavy lifting to `__getblk_gfp()` (see Listing 9-131), specifying `__GFP_MOVABLE` to indicate that the buffer head physical allocation should be movable.

```

367 static inline struct buffer_head *
368 sb_getblk_gfp(struct super_block *sb, sector_t block, gfp_t gfp)
369 {
370     return __getblk_gfp(sb->s_bdev, block, sb->s_blocksize, gfp);
371 }

```

Listing 9-126: include/linux/buffer_head.h: `sb_getblk_gfp()`

`sb_getblk_gfp()` does the same thing as `sb_getblk()`, only permits the specification of GFP flags via the `gfp` parameter.

Ultimately each of `sb_bread()` and `sb_bread_unmovable()` ultimately invoke `__bread_gfp()`, shown in Listing 9-127.

```

1362 /**
1363  * __bread_gfp() - reads a specified block and returns the bh
1364  * @bdev: the block_device to read from
1365  * @block: number of block
1366  * @size: size (in bytes) to read
1367  * @gfp: page allocation flag
1368  *
1369  * Reads a specified block, and returns buffer head that contains it.
1370  * The page cache can be allocated from non-movable area
1371  * not to prevent page migration if you set gfp to zero.
1372  * It returns NULL if the block was unreadable.
1373  */
1374 struct buffer_head *
1375 __bread_gfp(struct block_device *bdev, sector_t block,
1376             unsigned size, gfp_t gfp)
1377 {
1378     struct buffer_head *bh = __getblk_gfp(bdev, block, size, gfp);
1379
1380     if (likely(bh) && !buffer_uptodate(bh))
1381         bh = __bread_slow(bh);
1382     return bh;
1383 }

```

Listing 9-127: fs/buffer.c: `__bread_gfp()`

This retrieves the `struct buffer_head` describing the block using `__getblk_gfp()` then, if the block of interest is not uptodate, then it is read from disk via `__bread_slow()`, shown in Listing 9-128.

```

1168 static struct buffer_head *__bread_slow(struct buffer_head *bh)
1169 {
1170     lock_buffer(bh);
1171     if (buffer_uptodate(bh)) {

```

```

1172         unlock_buffer(bh);
1173         return bh;
1174     } else {
1175         get_bh(bh);
1176         bh->b_end_io = end_buffer_read_sync;
1177         submit_bh(REQ_OP_READ, bh);
1178         wait_on_buffer(bh);
1179         if (buffer_uptodate(bh))
1180             return bh;
1181     }
1182     brelse(bh);
1183     return NULL;
1184 }

```

Listing 9-128: fs/buffer.c: `__bread_slow()`

Reading a block using `__bread_slow()` is performed using the legacy `submit_bh()` function (see Listing 9-135 in Section 9.10.6) which wraps the operation in a struct `bio` (see Section 9.10.5 for a description of this process).

The function `end_buffer_read_sync()` function is specified as the callback to invoke once the read operation is complete. We explore this in Listing 9-129.

```

153  /*
154   * Default synchronous end-of-IO handler.. Just mark it up-to-date and
155   * unlock the buffer. This is what ll_rw_block uses too.
156   */
157  void end_buffer_read_sync(struct buffer_head *bh, int uptodate)
158  {
159      __end_buffer_read_notouch(bh, uptodate);
160      put_bh(bh);
161  }

```

Listing 9-129: fs/buffer.c: `end_buffer_read_sync()`

This function defers the operation to `__end_buffer_read_notouch()` shown in Listing 9-130 before dropping the buffer head reference count via `put_bh()`.

```

134  /*
135   * End-of-IO handler helper function which does not touch the bh after
136   * unlocking it.
137   * Note: unlock_buffer() sort-of does touch the bh after unlocking it, but
138   * a race there is benign: unlock_buffer() only use the bh's address for
139   * hashing after unlocking the buffer, so it doesn't actually touch the bh
140   * itself.
141   */
142  static void __end_buffer_read_notouch(struct buffer_head *bh, int uptodate)
143  {
144      if (uptodate) {

```

```

145         set_buffer_uptodate(bh);
146     } else {
147         /* This happens, due to failed read-ahead attempts. */
148         clear_buffer_uptodate(bh);
149     }
150     unlock_buffer(bh);
151 }

```

Listing 9-130: fs/buffer.c: __end_buffer_read_notouch()

This simply marks the buffer uptodate if the read succeeded, before unlocking the buffer (clearing the BH_Locked flag in struct buffer_head->b_state).

Returning to __bread_slow() in Listing 9-128, we see that after submitting the buffer head, we *wait* for the operation to complete, via wait_on_buffer() which, if the buffer is locked, invokes __wait_on_buffer(). This defers the waiting to wait_on_bit_io() which is out of scope for the book, but causes the process to wait until the bit is cleared.

Finally, __bread_slow() returns the buffer head if it is uptodate, otherwise if an error occurred, drops the reference to the buffer head via brelse() and returns NULL.

Regardless of the function invoked, all ultimately call __getblk_gfp(), shown in Listing 9-131.

```

1317 /*
1318  * __getblk_gfp() will locate (and, if necessary, create) the buffer_head
1319  * which corresponds to the passed block_device, block and size. The
1320  * returned buffer has its reference count incremented.
1321  *
1322  * __getblk_gfp() will lock up the machine if grow_dev_page's
1323  * try_to_free_buffers() attempt is failing. FIXME, perhaps?
1324  */
1325 struct buffer_head *
1326 __getblk_gfp(struct block_device *bdev, sector_t block,
1327             unsigned size, gfp_t gfp)
1328 {
1329     struct buffer_head *bh = __find_get_block(bdev, block, size);
1330
1331     might_sleep();
1332     if (bh == NULL)
1333         bh = __getblk_slow(bdev, block, size, gfp);
1334     return bh;
1335 }

```

Listing 9-131: fs/buffer.c: __getblk_gfp()

At this point we will not examine the logic any further as we are straying from memory management code into that of the file system.

9.10.4 Block Writes

We will not dwell too long on the block write mechanism as this is rather logic belonging to the file system subsystem rather than memory management, but we will briefly examine some relevant functions, starting with `__block_commit_write()`, as shown in Listing 9-132.

```

2061 static int __block_commit_write(struct inode *inode, struct page *page,
2062                               unsigned from, unsigned to)
2063 {
2064     unsigned block_start, block_end;
2065     int partial = 0;
2066     unsigned blocksize;
2067     struct buffer_head *bh, *head;
2068
2069     bh = head = page_buffers(page);
2070     blocksize = bh->b_size;
2071
2072     block_start = 0;
2073     do {
2074         block_end = block_start + blocksize;
2075         if (block_end <= from || block_start >= to) {
2076             if (!buffer_uptodate(bh))
2077                 partial = 1;
2078         } else {
2079             set_buffer_uptodate(bh);
2080             mark_buffer_dirty(bh);
2081         }
2082         if (buffer_new(bh))
2083             clear_buffer_new(bh);
2084
2085         block_start = block_end;
2086         bh = bh->b_this_page;
2087     } while (bh != head);
2088
2089     /*
2090     * If this is a partial write which happened to make all buffers
2091     * uptodate then we can optimize away a bogus read_folio() for
2092     * the next read(). Here we 'discover' whether the page went
2093     * uptodate as a result of this (potentially partial) write.
2094     */
2095     if (!partial)
2096         SetPageUptodate(page);
2097     return 0;
2098 }

```

Listing 9-132: fs/buffer.c: `__block_commit_write()`

This is invoked when writeback is complete, marking each buffer in the specified range uptodate, and checking if all other buffers in the range are equally marked as such we set the folio PG_uptodate flag.

This is a flag relevant to reading in data from disk, but a write might result in the same goal as, having overwritten data that would otherwise render the folio not uptodate we achieve an uptodate state.

The read equivalent of this is `end_buffer_async_read()`.

We mark a block dirty via `mark_buffer_dirty()` as part of file write dirty tracking, see Section 10.12 in the writeback chapter for more details. We explore this function in Listing 9-133 (eliding irrelevant trace and cgroup logic).

```

1044 /*
1045  * The relationship between dirty buffers and dirty pages:
1046  *
1047  * Whenever a page has any dirty buffers, the page's dirty bit is set, and
1048  * the page is tagged dirty in the page cache.
1049  *
1050  * At all times, the dirtiness of the buffers represents the dirtiness of
1051  * subsections of the page. If the page has buffers, the page dirty bit is
1052  * merely a hint about the true dirty state.
1053  *
1054  * When a page is set dirty in its entirety, all its buffers are marked dirty
1055  * (if the page has buffers).
1056  *
1057  * When a buffer is marked dirty, its page is dirtied, but the page's other
1058  * buffers are not.
1059  *
1060  * Also. When blockdev buffers are explicitly read with bread(), they
1061  * individually become uptodate. But their backing page remains not
1062  * uptodate - even if all of its buffers are uptodate. A subsequent
1063  * block_read_full_folio() against that folio will discover all the uptodate
1064  * buffers, will set the folio uptodate and will perform no I/O.
1065  */
1066
1067 /**
1068  * mark_buffer_dirty - mark a buffer_head as needing writeout
1069  * @bh: the buffer_head to mark dirty
1070  *
1071  * mark_buffer_dirty() will set the dirty bit against the buffer, then set
1072  * its backing page dirty, then tag the page as dirty in the page cache
1073  * and then attach the address_space's inode to its superblock's dirty
1074  * inode list.
1075  *
1076  * mark_buffer_dirty() is atomic. It takes bh->b_page->mapping->private_lock,
1077  * i_pages lock and mapping->host->i_lock.
1078  */
1079 void mark_buffer_dirty(struct buffer_head *bh)

```

```

1080 {
1081     WARN_ON_ONCE(!buffer_uptodate(bh));
1082     :
1083
1084     /*
1085      * Very *carefully* optimize the it-is-already-dirty case.
1086      *
1087      * Don't let the final "is it dirty" escape to before we
1088      * perhaps modified the buffer.
1089      */
1090     if (buffer_dirty(bh)) {
1091         smp_mb();
1092         if (buffer_dirty(bh))
1093             return;
1094     }
1095
1096     if (!test_set_buffer_dirty(bh)) {
1097         struct page *page = bh->b_page;
1098         struct address_space *mapping = NULL;
1099         :
1100         if (!TestSetPageDirty(page)) {
1101             mapping = page_mapping(page);
1102             if (mapping)
1103                 __set_page_dirty(page, mapping, 0);
1104         }
1105         :
1106         if (mapping)
1107             __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
1108     }
1109 }
1110
1111 }

```

Listing 9-133: fs/buffer.c: `mark_buffer_dirty()`

This starts by very cautiously performing a check to see if the buffer was already marked dirty with a memory barrier set to prevent CPU pipelining from confusing matters. We then mark the buffer dirty and, if the page was not already marked dirty, we invoke `__set_page_dirty()`. We also make sure to mark the inode dirty via `__mark_inode_dirty()`.

See the discussion around dirty tracking in the writeback chapter in Sections 10.12 and 10.12 for more details on this as a whole.

9.10.5 Block I/O (BIO) Operations

Again, we are straying from memory management into file system logic here, so we will examine things very briefly.

The fundamental operator of a block I/O operation is the `bio`, as described by the `struct bio` type. It describes an asynchronous I/O operation that is submitted to the relevant block device.

These operations are typically submitted via `submit_bio()` to be performed by the relevant block device asynchronously.

Let's additionally examine the case When files are synchronised, performed by `generic_file_fsync()` (see Listing ?? in the writeback chapter), the function `blkdev_issue_flush()` is called in order to submit an I/O operation to the block subsystem. We examine it in 9-134.

```

452 /**
453  * blkdev_issue_flush - queue a flush
454  * @bdev:      blockdev to issue flush for
455  *
456  * Description:
457  *   Issue a flush for the block device in question.
458  */
459 int blkdev_issue_flush(struct block_device *bdev)
460 {
461     struct bio bio;
462
463     bio_init(&bio, bdev, NULL, 0, REQ_OP_WRITE | REQ_PREFLUSH);
464     return submit_bio_wait(&bio);
465 }
```

Listing 9-134: block/blk-flush.c: `blkdev_issue_flush()`

Here we initialise this object with `bio_init`, specifying a zero size and requesting a flush, before submitting the operation and waiting for it to complete via `submit_bio_wait()`.

9.10.6 Buffer Head I/O Operations

The legacy wrapper function used for submitting buffer heads for read I/O is `submit_bh()` shown in Listing 9-135.

```

2723 int submit_bh(blk_opf_t opf, struct buffer_head *bh)
2724 {
2725     return submit_bh_wbc(opf, bh, NULL);
2726 }
```

Listing 9-135: fs/buffer.c: `submit_bh()`

This function defers the heavy lifting to `submit_bh_wbc()` which also handles writes, shown in Listing 9-136 (eliding out of scope `fscrypt` and `cgroup` logic).

```

2676 static int submit_bh_wbc(blk_opf_t opf, struct buffer_head *bh,
2677                        struct writeback_control *wbc)
2678 {
2679     const enum req_op op = opf & REQ_OP_MASK;
2680     struct bio *bio;
2681
2682     BUG_ON(!buffer_locked(bh));
2683     BUG_ON(!buffer_mapped(bh));
```

```

2684     BUG_ON(!bh->b_end_io);
2685     BUG_ON(buffer_delay(bh));
2686     BUG_ON(buffer_unwritten(bh));
2687
2688     /*
2689      * Only clear out a write error when rewriting
2690      */
2691     if (test_set_buffer_req(bh) && (op == REQ_OP_WRITE))
2692         clear_buffer_write_io_error(bh);
2693
2694     if (buffer_meta(bh))
2695         opf |= REQ_META;
2696     if (buffer_prio(bh))
2697         opf |= REQ_PRIO;
2698
2699     bio = bio_alloc(bh->b_bdev, 1, opf, GFP_NOIO);
2700     :
2703     bio->bi_iter.bi_sector = bh->b_blocknr * (bh->b_size >> 9);
2704
2705     bio_add_page(bio, bh->b_page, bh->b_size, bh_offset(bh));
2706     BUG_ON(bio->bi_iter.bi_size != bh->b_size);
2707
2708     bio->bi_end_io = end_bio_bh_io_sync;
2709     bio->bi_private = bh;
2710
2711     /* Take care of bh's that straddle the end of the device */
2712     guard_bio_eod(bio);
2713
2714     if (wbc) {
2715         wbc_init_bio(wbc, bio);
2716     :
2716     }
2717
2718     submit_bio(bio);
2719     return 0;
2720 }

```

Listing 9-136: fs/buffer.c: submit_bh_wbc()

Again, we do not dwell too long on this function, which we present here for informational purposes, but rather note that this converts the buffer operation into a block I/O one, using `submit_bio()` to submit it to the block subsystem, asynchronously.

9.11 Folio Locking and Waiting

When performing certain operations, we need to indicate that the folio should not be accessed until that operation is complete, functioning like a

mutual exclusion lock. This is implemented via the `PG_locked` folio flag, and if a lock is required unconditionally, applied via `folio_lock()` which we examine in Listing 9-137.

```

913 /**
914  * folio_lock() - Lock this folio.
915  * @folio: The folio to lock.
916  *
917  * The folio lock protects against many things, probably more than it
918  * should. It is primarily held while a folio is being brought uptodate,
919  * either from its backing file or from swap. It is also held while a
920  * folio is being truncated from its address_space, so holding the lock
921  * is sufficient to keep folio->mapping stable.
922  *
923  * The folio lock is also held while write() is modifying the page to
924  * provide POSIX atomicity guarantees (as long as the write does not
925  * cross a page boundary). Other modifications to the data in the folio
926  * do not hold the folio lock and can race with writes, eg DMA and stores
927  * to mapped pages.
928  *
929  * Context: May sleep. If you need to acquire the locks of two or
930  * more folios, they must be in order of ascending index, if they are
931  * in the same address_space. If they are in different address_spaces,
932  * acquire the lock of the folio which belongs to the address_space which
933  * has the lowest address in memory first.
934  */
935 static inline void folio_lock(struct folio *folio)
936 {
937     might_sleep();
938     if (!folio_trylock(folio))
939         __folio_lock(folio);
940 }

```

Listing 9-137: `include/linux/pagemap.h: folio_lock()`

Note that the legacy `lock_page()` function does the equivalent operation for the `struct page` type.

As the comment suggests, there are a number of different purposes for which the folio lock is used, primary among them being bringing the folio uptodate. In addition folios which are being faulted in are locked, as are folios which are being truncated or undergoing `write()`.

The lock is held in these instances such that the kernel does not try to access a folio which does not have valid state when it requires it, and additionally has a means of ‘stabilising’ a folio, especially with respect to truncation, by acquiring the lock.

The page cache separates the state of a folio being allocated and placed in the page cache and that folio having data read back from disk (at which point it is termed ‘uptodate’ as indicated by the `PG_uptodate` folio flag).

A folio can sit in this state awaiting an asynchronous I/O operation for potentially quite some time, which is why the faulting mechanism for instance implements the `maybe_unlock_mmap_for_io()` mechanism for dropping the highly contended `struct mm_struct->mmap_lock` when waiting for this to complete.

It also means that the act of waiting for a lock is quite a bit more involved than a typical mutual exclusion lock—by acquiring a lock using for instance `folio_lock()`, we imply that this thread should wait for any I/O that need be performed before the lock can be relinquished.

This also means that code which begins asynchronous I/O to read in data implicitly waits for that I/O, for instance the page fault mechanism after performing synchronous readahead (see Section ??) invokes `lock_folio_maybe_drop_mmap()` which locks the folio (if the fault state permits), implicitly waiting on the I/O to complete.

This also explains why so much kernel code first attempts the optimistic `folio_trylock()` before engaging the lock, as this implies no waiting on I/O whatsoever. Even `folio_lock()` tries this first as we can see in Listing 9-137.

We examine `folio_trylock()` in Listing 9-138.

```

888 /**
889  * folio_trylock() - Attempt to lock a folio.
890  * @folio: The folio to attempt to lock.
891  *
892  * Sometimes it is undesirable to wait for a folio to be unlocked (eg
893  * when the locks are being taken in the wrong order, or if making
894  * progress through a batch of folios is more important than processing
895  * them in order). Usually folio_lock() is the correct function to call.
896  *
897  * Context: Any context.
898  * Return: Whether the lock was successfully acquired.
899  */
900 static inline bool folio_trylock(struct folio *folio)
901 {
902     return likely(!test_and_set_bit_lock(PG_locked, folio_flags(folio, 0))
903 );
904 }
```

Listing 9-138: `include/linux/pagemap.h: folio_trylock()`

Which simply performs a bitwise operation which tests to see whether `PG_locked` is set, if not setting it. Note that the legacy `trylock_page()` wraps this function.

Since a folio lock implies that a thread can sleep waiting on I/O, we implement the ability to specify that a fatal signal could interrupt this process (by default it is uninterruptible), doing so via `folio_lock_killable()` which we examine in Listing 9-139.

```

963 /**
964  * folio_lock_killable() - Lock this folio, interruptible by a fatal signal.
```

```

965  * @folio: The folio to lock.
966  *
967  * Attempts to lock the folio, like folio_lock(), except that the sleep
968  * to acquire the lock is interruptible by a fatal signal.
969  *
970  * Context: May sleep; see folio_lock().
971  * Return: 0 if the lock was acquired; -EINTR if a fatal signal was received.
972  */
973 static inline int folio_lock_killable(struct folio *folio)
974 {
975     might_sleep();
976     if (!folio_trylock(folio))
977         return __folio_lock_killable(folio);
978     return 0;
979 }

```

Listing 9-139: include/linux/pagemap.h: `folio_lock_killable()`

This differs from the unconditional `folio_lock()` in that it returns an error code indicating whether it succeeded or not. In addition, it invokes `__folio_lock_killable()` (see Listing ??) rather than `__folio_lock()` (shown in Listing 9-140) which specifies that the operation is killable.

Note that this is wrapped by the legacy `lock_page_killable()` function.

```

1658 /**
1659  * __folio_lock - Get a lock on the folio, assuming we need to sleep to get it
1660  *
1661  * @folio: The folio to lock
1662  */
1663 void __folio_lock(struct folio *folio)
1664 {
1665     folio_wait_bit_common(folio, PG_locked, TASK_UNINTERRUPTIBLE,
1666                          EXCLUSIVE);
1667 }

```

Listing 9-140: mm/filemap.c: `__folio_lock()`

This function defers the heavy lifting of causing the process to sleep to `folio_wait_bit_common()`, which we examine in Listing 9-150 below. This specifies `TASK_UNINTERRUPTIBLE` to indicate that the thread mustn't be interrupted during this operation.

It also specifies `EXCLUSIVE` to indicate that the flag should be set when the waiting thread is woken. This is an enumeration value from `enum behavior` which we examine in Listing 9-141.

```

1181 /*
1182  * A choice of three behaviors for folio_wait_bit_common():
1183  */
1184 enum behavior {
1185     EXCLUSIVE,      /* Hold ref to page and take the bit when woken, like

```

```

1186         * __folio_lock() waiting on then setting PG_locked.
1187         */
1188     SHARED,    /* Hold ref to page and check the bit when woken, like
1189                * folio_wait_writeback() waiting on PG_writeback.
1190                */
1191     DROP,      /* Drop ref to page before wait, no check when woken,
1192                * like folio_put_wait_locked() on PG_locked.
1193                */
1194 };

```

Listing 9-141: mm/filemap.c: enum behavior

We examine `__folio_lock_killable()` in Listing ??.

```

1669 int __folio_lock_killable(struct folio *folio)
1670 {
1671     return folio_wait_bit_common(folio, PG_locked, TASK_KILLABLE,
1672                                 EXCLUSIVE);
1673 }

```

Listing 9-142: mm/filemap.c: __folio_lock_killable()

Here the `TASK_KILLABLE` flag is used to indicate that the thread can be killed during the operation.

The fact that we are implementing careful waiting code which has potentially sleeping threads waiting on the change of the flag, we must therefore implement careful handling on unlock in `folio_unlock()` which we examine in Listing 9-143 (eliding distracting bug checks).

```

1517 /**
1518  * folio_unlock - Unlock a locked folio.
1519  * @folio: The folio.
1520  *
1521  * Unlocks the folio and wakes up any thread sleeping on the page lock.
1522  *
1523  * Context: May be called from interrupt or process context. May not be
1524  * called from NMI context.
1525  */
1526 void folio_unlock(struct folio *folio)
1527 {
1528     :
1532     if (clear_bit_unlock_is_negative_byte(PG_locked, folio_flags(folio, 0))
1533         )
1534         folio_wake_bit(folio, PG_locked);
1535 }

```

Listing 9-143: mm/filemap.c: folio_unlock()

This invokes `folio_wake_bit()` to wake any thread waiting on the lock to be cleared. We examine this in Listing 9-144.

```

1127 static void folio_wake_bit(struct folio *folio, int bit_nr)
1128 {
1129     wait_queue_head_t *q = folio_waitqueue(folio);
1130     struct wait_page_key key;
1131     unsigned long flags;
1132     wait_queue_entry_t bookmark;
1133
1134     key.folio = folio;
1135     key.bit_nr = bit_nr;
1136     key.page_match = 0;
1137
1138     bookmark.flags = 0;
1139     bookmark.private = NULL;
1140     bookmark.func = NULL;
1141     INIT_LIST_HEAD(&bookmark.entry);
1142
1143     spin_lock_irqsave(&q->lock, flags);
1144     __wake_up_locked_key_bookmark(q, TASK_NORMAL, &key, &bookmark);
1145
1146     while (bookmark.flags & WQ_FLAG_BOOKMARK) {
1147         /*
1148          * Take a breather from holding the lock,
1149          * allow pages that finish wake up asynchronously
1150          * to acquire the lock and remove themselves
1151          * from wait queue
1152          */
1153         spin_unlock_irqrestore(&q->lock, flags);
1154         cpu_relax();
1155         spin_lock_irqsave(&q->lock, flags);
1156         __wake_up_locked_key_bookmark(q, TASK_NORMAL, &key, &bookmark)
1157     };
1158
1159     /*
1160      * It's possible to miss clearing waiters here, when we woke our page
1161      * waiters, but the hashed waitqueue has waiters for other pages on it
1162      *
1163      * That's okay, it's a rare case. The next waker will clear it.
1164      *
1165      * Note that, depending on the page pool (buddy, hugetlb, ZONE_DEVICE,
1166      * other), the flag may be cleared in the course of freeing the page;
1167      * but that is not required for correctness.
1168      */
1169     if (!waitqueue_active(q) || !key.page_match)
1170         folio_clear_waiters(folio);
1171     spin_unlock_irqrestore(&q->lock, flags);

```

1172 }

Listing 9-144: mm/filemap.c: folio_wake_bit()

We won't get into the gritty details of this function, but we shall note a few things here—there is an underlying mechanism which maintains wait queues for folios to allow for us to queue up threads waiting on a folio flag to change. This wait queue is accessible via `folio_waitqueue()`. Wait queues are a means by which the kernel allows for threads to wait for certain events to arise while sleeping.

We also note here that a folio with waiters will have the `PG_waiters` flag specified. This is made more explicit in `folio_wake()` which explicitly only invokes `folio_wake_bit()` if this is set, as shown in Listing 9-145.

```
1174 static void folio_wake(struct folio *folio, int bit)
1175 {
1176     if (!folio_test_waiters(folio))
1177         return;
1178     folio_wake_bit(folio, bit);
1179 }
```

Listing 9-145: mm/filemap.c: folio_wake()

This is utilised by `folio_end_writeback()` to wake waiters on the `PG_writeback` flag given that writeback at this point will have completed.

As should be clear by now, it is possible to wait on folio flags other than `PG_locked`. this is indeed done in `folio_wait_writeback()` to wait for writeback to complete for a folio for file systems and block devices which require it (see the writeback chapter for details on how this is utilised).

We examine this in Listing 9-146 (eliding an out of scope trace hook).

```
3019 /**
3020  * folio_wait_writeback - Wait for a folio to finish writeback.
3021  * @folio: The folio to wait for.
3022  *
3023  * If the folio is currently being written back to storage, wait for the
3024  * I/O to complete.
3025  *
3026  * Context: Sleeps. Must be called in process context and with
3027  * no spinlocks held. Caller should hold a reference on the folio.
3028  * If the folio is not locked, writeback may start again after writeback
3029  * has finished.
3030  */
3031 void folio_wait_writeback(struct folio *folio)
3032 {
3033     while (folio_test_writeback(folio)) {
3034         :
3035         folio_wait_bit(folio, PG_writeback);
3036     }
3037 }
```

Listing 9-146: mm/page-writeback.c: folio_wait_writeback()

This simply loops while the PG_writeback flag is set, waiting on folio_wait_bit(). This function is simply a wrapper around folio_wait_bit_common() (see Listing ??) specifying TASK_UNINTERRUPTIBLE but setting SHARED to indicate that we are waiting for the bit to be cleared, but not setting it at this point (unlike EXCLUSIVE used by folio_lock() in Listing 9-137 above). We examine this in Listing 9-147.

```

1445 void folio_wait_bit(struct folio *folio, int bit_nr)
1446 {
1447     folio_wait_bit_common(folio, bit_nr, TASK_UNINTERRUPTIBLE, SHARED);
1448 }
```

Listing 9-147: mm/filemap.c: folio_wait_bit()

The folio_put_wait_locked() function, as referenced by do_read_cache_folio(), uses the DROP behaviour to drop a reference and then wait for unlock, as shown in Listing 9-148.

```

1457 /**
1458  * folio_put_wait_locked - Drop a reference and wait for it to be unlocked
1459  * @folio: The folio to wait for.
1460  * @state: The sleep state (TASK_KILLABLE, TASK_UNINTERRUPTIBLE, etc).
1461  *
1462  * The caller should hold a reference on @folio. They expect the page to
1463  * become unlocked relatively soon, but do not wish to hold up migration
1464  * (for example) by holding the reference while waiting for the folio to
1465  * come unlocked. After this function returns, the caller should not
1466  * dereference @folio.
1467  *
1468  * Return: 0 if the folio was unlocked or -EINTR if interrupted by a signal.
1469  */
1470 int folio_put_wait_locked(struct folio *folio, int state)
1471 {
1472     return folio_wait_bit_common(folio, PG_locked, state, DROP);
1473 }
```

Listing 9-148: mm/filemap.c: folio_put_wait_locked()

This, as with other wait functionality, defers the operation to folio_wait_bit_common() which we will examine in Listing ?? shortly.

Sometimes we want to wait on PG_locked without acquiring the lock. We utilise SHARED semantics via folio_wait_bit() (see Listing 9-147 above) to do so in folio_wait_locked() as shown in Listing 9-149.

```

1015 /*
1016  * Wait for a folio to be unlocked.
1017  *
1018  * This must be called with the caller "holding" the folio,
```

```

1019  * ie with increased folio reference count so that the folio won't
1020  * go away during the wait.
1021  */
1022  static inline void folio_wait_locked(struct folio *folio)
1023  {
1024      if (folio_test_locked(folio))
1025          folio_wait_bit(folio, PG_locked);
1026  }

```

Listing 9-149: include/linux/pagemap.h: folio_wait_locked()

Equally `folio_wait_locked_killable()` does the same thing, only permitting the thread to be interrupted by fatal signals.

Finally, let's examine `folio_wait_bit_common()` starting in Listing 9-150 (eliding out of scope PSI and delayed accounting thrashing logic).

```

1216  static inline int folio_wait_bit_common(struct folio *folio, int bit_nr,
1217                                         int state, enum behavior behavior)
1218  {
1219      wait_queue_head_t *q = folio_waitqueue(folio);
1220      int unfairness = sysctl_page_lock_unfairness;
1221      struct wait_page_queue wait_page;
1222      wait_queue_entry_t *wait = &wait_page.wait;
1223      :
1224      init_wait(wait);
1225      wait->func = wake_page_function;
1226      wait_page.folio = folio;
1227      wait_page.bit_nr = bit_nr;
1228
1229  repeat:
1230      wait->flags = 0;
1231      if (behavior == EXCLUSIVE) {
1232          wait->flags = WQ_FLAG_EXCLUSIVE;
1233          if (--unfairness < 0)
1234              wait->flags |= WQ_FLAG_CUSTOM;
1235      }
1236
1237      /*
1238       * Do one last check whether we can get the
1239       * page bit synchronously.
1240       *
1241       * Do the folio_set_waiters() marking before that
1242       * to let any waker we _just_ missed know they
1243       * need to wake us up (otherwise they'll never
1244       * even go to the slow case that looks at the
1245       * page queue), and add ourselves to the wait
1246       * queue if we need to sleep.
1247       *
1248       * This part needs to be done under the queue

```



```

1262      * lock to avoid races.
1263      */
1264      spin_lock_irq(&q->lock);
1265      folio_set_waiters(folio);
1266      if (!folio_trylock_flag(folio, bit_nr, wait))
1267          __add_wait_queue_entry_tail(q, wait);
1268      spin_unlock_irq(&q->lock);
1269
1270      /*
1271       * From now on, all the logic will be based on
1272       * the WQ_FLAG_WOKEN and WQ_FLAG_DONE flag, to
1273       * see whether the page bit testing has already
1274       * been done by the wake function.
1275       *
1276       * We can drop our reference to the folio.
1277       */
1278      if (behavior == DROP)
1279          folio_put(folio);

```

Listing 9-150: mm/filemap.c: folio_wait_bit_common() preamble

We won't examine this function in absolute microscopic detail, but we will examine the key elements of it. A lot of this functionality derives from the kernel's wait queue implementation, which this relies upon to queue up threads waiting for folio bits to be cleared.

The function which is invoked when a bit is changed is `wake_page_function()`, one which we won't examine in order to avoid getting bogged down in write queue implementation details but we will reproduce the comment here which explains how each of the wait queue flags we establish here are utilised in Listing 9-151.

```

1042  /*
1043   * The page wait code treats the "wait->flags" somewhat unusually, because
1044   * we have multiple different kinds of waits, not just the usual "exclusive"
1045   * one.
1046   *
1047   * We have:
1048   *
1049   * (a) no special bits set:
1050   *
1051   *     We're just waiting for the bit to be released, and when a waker
1052   *     calls the wakeup function, we set WQ_FLAG_WOKEN and wake it up,
1053   *     and remove it from the wait queue.
1054   *
1055   *     Simple and straightforward.
1056   *
1057   * (b) WQ_FLAG_EXCLUSIVE:
1058   *
1059   *     The waiter is waiting to get the lock, and only one waiter should

```

```

1060 *      be woken up to avoid any thundering herd behavior. We'll set the
1061 *      WQ_FLAG_WOKEN bit, wake it up, and remove it from the wait queue.
1062 *
1063 *      This is the traditional exclusive wait.
1064 *
1065 *      (c) WQ_FLAG_EXCLUSIVE | WQ_FLAG_CUSTOM:
1066 *
1067 *      The waiter is waiting to get the bit, and additionally wants the
1068 *      lock to be transferred to it for fair lock behavior. If the lock
1069 *      cannot be taken, we stop walking the wait queue without waking
1070 *      the waiter.
1071 *
1072 *      This is the "fair lock handoff" case, and in addition to setting
1073 *      WQ_FLAG_WOKEN, we set WQ_FLAG_DONE to let the waiter easily see
1074 *      that it now has the lock.
1075 */
1076 static int wake_page_function(wait_queue_entry_t *wait, unsigned mode, int
      sync, void *arg)
1077 {
      :
1125 }

```

Listing 9-151: mm/filemap.c: wake_page_function()

Returning to Listing 9-150, we see that we reference the `sysctl_page_lock_unfairness` global variable, which refers to the `vm.page_lock_unfairness` tunable.

This is used to limit how often we will tolerate the lock being stolen from underneath us in EXCLUSIVE mode, i.e. the one in which we intend to take the lock as soon as another thread has relinquished it. This has already occurred once as we are having to wait, so we decrement this value to begin with. If we exceed this limit, we finally set the `WQ_FLAG_CUSTOM` flag to indicate that we now be given this lock regardless, as handled by `wake_page_function()`.

Before we enter into the wait loop, we make one last gasp effort to acquire the flag we desire, while setting `PG_waiters` to indicate there are threads waiting on a flag. If we can't acquire the flag, we add the current thread to the wait queue.

Next, if the DROP mode is specified, we drop a folio reference as the semantics imply.

We examine the core waiting loop in Listing 9-152 below.

```

1281 /*
1282  * Note that until the "finish_wait()", or until
1283  * we see the WQ_FLAG_WOKEN flag, we need to
1284  * be very careful with the 'wait->flags', because
1285  * we may race with a waker that sets them.
1286 */
1287 for (;;) {
1288     unsigned int flags;

```

```

1289
1290         set_current_state(state);
1291
1292         /* Loop until we've been woken or interrupted */
1293         flags = smp_load_acquire(&wait->flags);
1294         if (!(flags & WQ_FLAG_WOKEN)) {
1295             if (signal_pending_state(state, current))
1296                 break;
1297
1298             io_schedule();
1299             continue;
1300         }
1301
1302         /* If we were non-exclusive, we're done */
1303         if (behavior != EXCLUSIVE)
1304             break;
1305
1306         /* If the waker got the lock for us, we're done */
1307         if (flags & WQ_FLAG_DONE)
1308             break;
1309
1310         /*
1311          * Otherwise, if we're getting the lock, we need to
1312          * try to get it ourselves.
1313          *
1314          * And if that fails, we'll have to retry this all.
1315          */
1316         if (unlikely(test_and_set_bit(bit_nr, folio_flags(folio, 0))))
1317             goto repeat;
1318
1319         wait->flags |= WQ_FLAG_DONE;
1320         break;
1321     }

```

Listing 9-152: mm/filemap.c: folio_wait_bit_common() main loop

We start by setting the scheduler mode specified in the state parameter via `set_current_state()`, i.e. whether the task should be uninterruptible or killable.

We then loop, waiting for the work queue to be woken, rescheduling (and thus sleeping this process) via `io_schedule()`.

At this stage, if the mode was not exclusive, we are done and can exit the loop. Equally if the wait queue flag `WQ_FLAG_DONE` is set in `EXCLUSIVE` mode, we have the lock and can exit.

Otherwise we check to see if the lock has been taken from underneath us, looping back to the `repeat` label if so to perform unfairness checks.

Finally, if we have the lock we mark it and exit the loop.

Next, we wrap things up in Listing 9-153.

```

1216      /*
1217       * If a signal happened, this 'finish_wait()' may remove the last
1218       * waiter from the wait-queues, but the folio waiters bit will remain
1219       * set. That's ok. The next wakeup will take care of it, and trying
1220       * to do it here would be difficult and prone to races.
1221       */
1222      finish_wait(q, wait);
1223      :
1237      /*
1238       * NOTE! The wait->flags weren't stable until we've done the
1239       * 'finish_wait()', and we could have exited the loop above due
1240       * to a signal, and had a wakeup event happen after the signal
1241       * test but before the 'finish_wait()'.
1242       *
1243       * So only after the finish_wait() can we reliably determine
1244       * if we got woken up or not, so we can now figure out the final
1245       * return value based on that state without races.
1246       *
1247       * Also note that WQ_FLAG_WOKEN is sufficient for a non-exclusive
1248       * waiter, but an exclusive one requires WQ_FLAG_DONE.
1249       */
1250      if (behavior == EXCLUSIVE)
1251          return wait->flags & WQ_FLAG_DONE ? 0 : -EINTR;
1252
1253      return wait->flags & WQ_FLAG_WOKEN ? 0 : -EINTR;
1254  }

```

Listing 9-153: mm/filemap.c: folio_wait_bit_common() suffix

We wrap up the wait queue operation via `finish_wait()`, before performing some careful checks to see if we were interrupted by a signal, returning the `EINTR` error if we were.

At this point the waiting is complete and the desired behaviour has been achieved.

10

WRITEBACK

Writeback, as the names suggests, is the process by which Linux ultimately writes data back to disk from the page cache. The kernel must maintain a trade-off between caching writes in memory and writing back to disk and handle scenarios under which dirtying exceeds writeback bandwidth. We discuss how all of this is performed within this chapter.

In order to track which page cache folios need writeback, we track which ones have had changes made to them since they were last synchronised to disk. These pages are termed *dirty*. Those which are in sync with the disk (and uptodate, i.e. have already successfully been read from disk) are termed *clean*. We discuss the process of tracking dirty data in Section 10.1.

The actual process of writeback is performed one of two ways—periodically in the background, or when explicit synchronisation is triggered. When folios are dirtied, they automatically become subject to background writeback. In detail:

Writeback in Linux is divided into several different stages:

Dirty Tracking We first need to know that data has changed in the page cache, which occurs when data is written to the page cache. Both the folio written to and the inode describing the file are marked dirty as described in Section 10.1.

Dirty Page Balancing Each time data is marked dirty, the kernel checks to see whether either the amount of dirty data exceeds `vm.dirty_bytes`, or the ratio between this data and *available* memory exceeds `vm.dirty_ratio`. If this is the case, the I/O operation blocks and writeback is performed immediately, as described in Section 10.14.

Background Dirty Page Balancing If direct balancing is not required, the kernel check to see whether the amount of dirty data exceeds `vm.dirty_background_bytes`, or the ratio between this data and *available* memory exceeds `vm.dirty_background_ratio` (these are mutually exclusive, of one tuneable is set, the other is cleared). In either case, the writeback work queue thread is immediately woken to perform background writeback. Again, see Section 10.14 for details on this process.

Synchronisation A user can additionally opt to manually trigger synchronisation to the disk via a number of system calls, e.g. `sync()`. We examine how manual synchronisation to disk is performed in the kernel in Section 10.6.

Background Writeback If not immediately awoken, the writeback kernel thread is otherwise woken up `vm.dirty_writeback_centisecs` (expressed in ‘centiseconds’, or hundredths of a second) after the last time this check was performed.

Writing Back to Disk Either synchronisation is requested or the writeback work queue thread has identified data that need to be written back, i.e. older than `vm.dirty_expire_centisecs`. In either case writeback to disk is begun, as described in in Section 10.10.

Writing Back to the Block Device Ultimately the actual I/O must be submitted to the underlying block device. What we must write back is determined by the levels above, which are then translated into block operations, which we examine in Section 9.10.4.

10.1 Dirty Tracking in the Kernel

In order to keep track of what needs to be written back to disk in the page cache we need to track two separate things—the dirty folios themselves and the inodes which back them.

Recalling from Section 4.5 that the `struct address_space` object is enveloped by its describing `struct inode` object, we can observe how this tracking is performed in Figure 10-1.

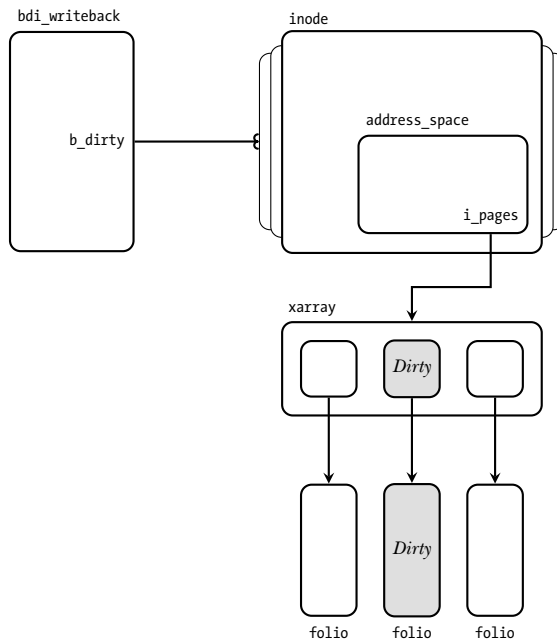


Figure 10-1: Dirty Page and inode Tracking

Each device which ‘backs’ the data stored in the page cache is termed a ‘Backing Device’, and data associated with it is termed ‘Backing Device Information’ or BDI for short. BDI is typically used interchangeably to describe the backing device itself, as will we. These are described by the struct `backing_dev_info` type.

The state of *writeback* on each BDI is tracked using a struct `bdi_writeback` object, and it is here that dirty inodes are tracked in its `b_dirty` list as shown in Figure 10-1.

We tag dirty folios in the struct `address_space->i_pages` xarray using `xas_set_mark()` which uses the xarray *mark* functionality to set the `PAGECACHE_TAG_DIRTY` tag.

This enables us to very quickly iterate through all dirty pages in an struct `address_space` using `xas_for_each_marked()`.

Once we have tagged the dirty pages, we also need to keep track of which inodes possess struct `address_space` objects which have dirty pages attached. These are termed ‘dirty inodes’.

If the kernel configuration option `CONFIG_CGROUP_WRITEBACK` is *not* set, this object is simply accessible via struct `backing_dev_info->wb` field and we simply maintain a single writeback object per BDI.

This makes sense, as writeback is limited by each backing device’s available I/O bandwidth, so the state should be tied to each of these devices.

However, if cgroups are in use, available bandwidth in the BDI may be subdivided between cgroups. Discussion of cgroups is out of scope for the book, but it is useful to note that if `CONFIG_CGROUP_WRITEBACK` is set,

then a pointer to the struct `bdi_writeback` relevant object is stored in the struct `inode->i_wb` field instead.

In either case, we obtain the relevant BDI writeback object via `inode_to_wb()`.

Maintaining this list is how we track which pages inodes are dirty and whose pages need writeback. We examine how this is ultimately performed in Section 10.10.

Placing inodes into the dirty list and marking pages as dirty typically occur either through a write operation such as that accomplished through the `write()` system call, discussed in Section 10.12, or by a page fault occurring on a memory-mapped file-backed page, discussed in Section 10.12.

10.2 Marking the Folio Dirty

Regardless of what makes the pages dirty, we mark the xarray accordingly in `__folio_mark_dirty()`, as shown in Listing 10-1.

```

2594 /*
2595  * Mark the folio dirty, and set it dirty in the page cache, and mark
2596  * the inode dirty.
2597  *
2598  * If warn is true, then emit a warning if the folio is not uptodate and has
2599  * not been truncated.
2600  *
2601  * The caller must hold lock_page_memcg(). Most callers have the folio
2602  * locked. A few have the folio blocked from truncation through other
2603  * means (eg zap_page_range() has it mapped and is holding the page table
2604  * lock). This can also be called from mark_buffer_dirty(), which I
2605  * cannot prove is always protected against truncate.
2606  */
2607 void __folio_mark_dirty(struct folio *folio, struct address_space *mapping,
2608                        int warn)
2609 {
2610     unsigned long flags;
2611
2612     xa_lock_irqsave(&mapping->i_pages, flags);
2613     if (folio->mapping) { /* Race with truncate? */
2614         WARN_ON_ONCE(warn && !folio_test_uptodate(folio));
2615         folio_account_dirtied(folio, mapping);
2616         __xa_set_mark(&mapping->i_pages, folio_index(folio),
2617                      PAGECACHE_TAG_DIRTY);
2618     }
2619     xa_unlock_irqrestore(&mapping->i_pages, flags);
2620 }

```

Listing 10-1: `mm/page-writeback.c: __folio_mark_dirty()`

Here the mark is set to `PAGECACHE_TAG_DIRTY` for the folio under lock, with care to avoid a race with truncation (see Section 9.9 for more on this operation), which naturally clears the `struct folio->mapping` field.

Statistics are updated in `folio_account_dirtied()`, which also assigns the `struct inode->i_wb` field if not already set via `inode_attach_wb()`.

10.3 Marking the inode Dirty

Now we have marked the dirty folios in the xarray, we need to add the surrounding dirty inodes to the BDI dirty list, so we know what to write back when the time comes.

This is performed in `__mark_inode_dirty()`, which again is invoked both when writeback performed via the `write()` system call or when a page fault occurs.

The page cache code must handle the case of both writing back changes to inode metadata itself as well as the underlying folios. We will examine only writing back the underlying data. However note that the inode flags `I_DIRTY_SYNC` and `I_DIRTY_DATASYNC`, combined in the mask `I_DIRTY_INODE`, represent the need to write back this metadata.

In addition, the kernel handles a special case of changes to file time metadata for a filesystem mounted with the `lazytime` option specified via `I_DIRTY_TIME`. Again, this is out of scope for our discussion, but it is worth noting that these cases are handled.

NOTE

Linux provides three timestamps associated with file metadata encapsulated in a file's `inode->atime`, `ctime` and `mtime`. These specify the last time the file was accessed, the last time its metadata was changed and the last time the underlying file data was modified, respectively.

Typically file systems are mounted such that the access time is updated only if the `mtime` or `ctime` was updated more recently (`relatime` mode), or written to disk only when absolutely necessary (`lazytime` mode).

The `I_DIRTY_PAGES` flag specifies that the underlying file data has been dirtied, which is what we focus upon. We therefore examine the logic specifically tied to this case in `__mark_inode_dirty()` in Listing 10-2, eliding all irrelevant code.

```

2363 void __mark_inode_dirty(struct inode *inode, int flags)
2364 {
2365     struct super_block *sb = inode->i_sb;
2366     :
2367     struct bdi_writeback *wb = NULL;
2368     :
2406     spin_lock(&inode->i_lock);
2407     :
2409     if ((inode->i_state & flags) != flags) {
2410         const int was_dirty = inode->i_state & I_DIRTY;
2411 
```

```

2412         inode_attach_wb(inode, NULL);
2413         :
2417         inode->i_state |= flags;
2418
2419         /*
2420         * Grab inode's wb early because it requires dropping i_lock
and we
2421         * need to make sure following checks happen atomically with
dirty
2422         * list handling so that we don't move inodes under flush
worker's
2423         * hands.
2424         */
2425         if (!was_dirty) {
2426             wb = locked_inode_to_wb_and_lock_list(inode);
2427             spin_lock(&inode->i_lock);
2428         }
2429
2430         /*
2431         * If the inode is queued for writeback by flush worker, just
2432         * update its dirty state. Once the flush worker is done with
2433         * the inode it will place it on the appropriate superblock
2434         * list, based upon its state.
2435         */
2436         if (inode->i_state & I_SYNC_QUEUED)
2437             goto out_unlock;
2438
2439         /*
2440         * Only add valid (hashed) inodes to the superblock's
2441         * dirty list. Add blockdev inodes as well.
2442         */
2443         if (!S_ISBLK(inode->i_mode)) {
2444             if (inode_unhashed(inode))
2445                 goto out_unlock;
2446         }
2447         if (inode->i_state & I_FREEING)
2448             goto out_unlock;
2449
2450         /*
2451         * If the inode was already on b_dirty/b_io/b_more_io, don't
2452         * reposition it (that would break b_dirty time-ordering).
2453         */
2454         if (!was_dirty) {
2455             struct list_head *dirty_list;
2456             bool wakeup_bdi = false;
2457
2458             inode->dirtied_when = jiffies;

```

```

:
2462         if (inode->i_state & I_DIRTY)
2463             dirty_list = &wb->b_dirty;
:
2467         wakeup_bdi = inode_io_list_move_locked(inode, wb,
2468                                             dirty_list);
2469
2470         spin_unlock(&wb->list_lock);
2471         spin_unlock(&inode->i_lock);
:
2474         /*
2475          * If this is the first dirty inode for this bdi,
2476          * we have to wake-up the corresponding bdi thread
2477          * to make sure background write-back happens
2478          * later.
2479          */
2480         if (wakeup_bdi &&
2481             (wb->bdi->capabilities & BDI_CAP_WRITEBACK))
2482             wb_wakeup_delayed(wb);
2483         return;
2484     }
2485 }
2486 out_unlock:
2487     if (wb)
2488         spin_unlock(&wb->list_lock);
2489 out_unlock_inode:
2490     spin_unlock(&inode->i_lock);
2491 }

```

Listing 10-2: fs/fs-writeback.c: `__mark_inode_dirty()`

The flags specified for the dirtying operation are provided by the `flags` parameter. As previously discussed, we are only interested in the `I_DIRTY_PAGES` flag, however note that the `I_DIRTY` mask includes this flag.

Interacting with the struct `inode` fields is performed under its `i_lock` spin-lock and we begin by proceeding only if this dirtying represents a change in the inode's state, maintained in its struct `inode->i_state` field—if nothing has changed, there's nothing to do.

Next we determine whether the node was previously dirty in the `was_dirty` predicate. This would only be set true if the flags had changed, but varied whether the inode, pages or lazytime dirty flags were set.

If the inode has transitioned from being clean to dirty, we obtain the relevant struct `bdi_writeback` object early via `locked_inode_to_wb_and_lock_list()`.

This handles the `CONFIG_CGROUP_WRITEBACK` case using `inode_to_wb()` and accounting for races, ensuring that reference counting is performed correctly and acquires the struct `bdi_writeback->list_lock` spin lock.

We do this early, as the kernel thread which ultimately flushes data back to disk (see Section 10.10) may race with us if we do not

acquire this at this stage. We must re-lock `struct inode->i_lock` as `locked_inode_to_wb_and_lock_list()` releases it.

If the inode is already queued for writeback, as indicated by the `I_SYNC_QUEUED` flag being set in `struct inode->i_state`, we abort as no action is required.

Next we check to see whether the inode is *hashed* in the inode cache, a hash table maintained for fast access by the VFS. If it is not, then the inode is not in a state where it can be correctly dirty tracked, so we abort. We make an exception for blockdev inodes (see Section 9.10.2).

Finally, if the inode is in the process of being freed as indicated by the `I_FREEING`, we do not need to proceed with marking it dirty.

We always keep track of *when* the inode was marked dirty in its `struct inode->dirty_when` field, which is vital for making decisions on writeback (see Section 10.10 for more details on how this plays a role), before placing the inode on the dirty list at `struct bdi_writeback->b_dirty` via `inode_io_list_move_locked()`.

An important nuance to take into account here is that, when dirtying pages, the inode passed to the function is the `struct address_space->host` one. In the case of accessing a block device directly via e.g. `/dev/sda1`, it is a special blockdev inode that is present here rather than the inode of the individual device file, which keeps the dirtying time consistent.

See Section 9.10.2 for more details on blockdev and accessing block devices directly via device files.

NOTE

The kernel counts time in jiffies, which is an internal kernel field that is updated a certain number of times a second.

`inode_io_list_move_locked()` proceeds to place the inode on the BDI's dirty list `struct bdi_writeback->b_dirty` using its `struct inode->i_io_list` list node.

Each block device has a writeback flush kernel thread which performs background writeback based on tuneable configuration (see Section 10.11 for details), which is woken up when the first dirty inode is placed on its associated `struct bdi_writeback` dirty list.

We achieve this by first checking whether wakeup needs to be done in `inode_io_list_move_locked()`, which uses the `wb_io_lists_populated()` function to do so, which in turn sets the `WB_has_dirty_io` flag in the `struct bdi_writeback->state` field to track this if this is the case.

We then use the result of this call, if the BDI is capable of writeback, by invoking `wb_wakeup_delayed()` which we explore in Listing 10-32 in Section 10.10.

10.4 Page Fault Dirty Tracking

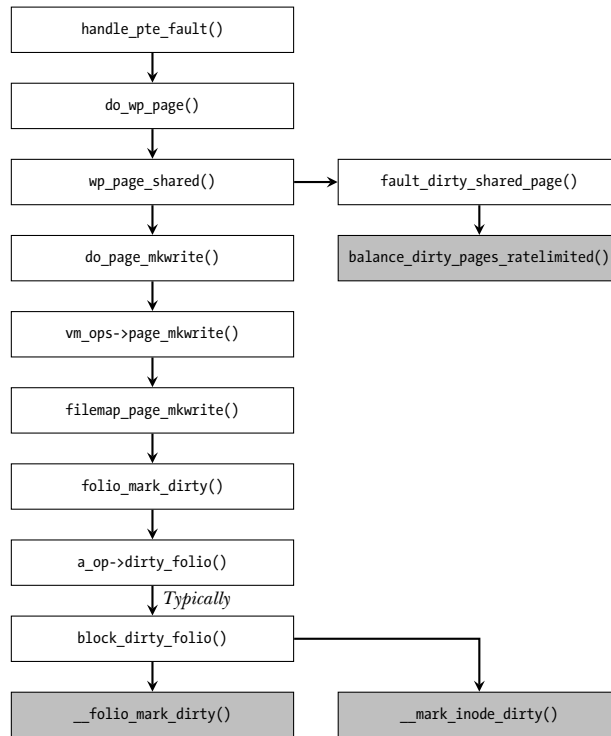


Figure 10-2: Typical Write Fault Dirty Page Tracking

We have already discussed how folios are *read* from disk in Section 9.4 above, but we have not yet explored how data is written back to disk when modified in a memory-mapped file, which we will do so here.

We leverage the hardware memory management to determine when a file-backed mapping is written to by making the mapping read-only when it is clean, and triggering a file system-specific hook when it is written to.

This is handled in `wp_page_shared()` which is discussed in Section 6.9 and Listing 6-39. This ultimately invokes `do_page_mkwrite()` (discussed in Listing 6-31), which calls the struct `vm_operations_struct->page_mkwrite` callback specified in the VMA struct `vm_area_struct->vm_ops` field.

As discussed in Section 9.1, when a VMA is memory-mapped, its struct `file_operations->mmap` is invoked in struct `file->f_op` and it is this which specifies the struct `vm_area_struct->vm_ops` callbacks in which `page_mkwrite` is specified.

The majority of files systems use a generic function when memory mapping, `generic_file_mmap()` as shown in Listing 10-3.

```

3431 /* This is used for a general mmap of a disk file */
3432
3433 int generic_file_mmap(struct file *file, struct vm_area_struct *vma)

```

```

3434 {
3435     struct address_space *mapping = file->f_mapping;
3436
3437     if (!mapping->a_ops->read_folio)
3438         return -ENOEXEC;
3439     file_accessed(file);
3440     vma->vm_ops = &generic_file_vm_ops;
3441     return 0;
3442 }

```

Listing 10-3: mm/filemap.c: generic_file_mmap()

This function sets struct vm_area_struct->vm_ops to the generic_file_vm_ops object, which specifies each of the fault, map_pages and page_mkwrite callbacks, shown in Listing 10-4.

```

3425 const struct vm_operations_struct generic_file_vm_ops = {
3426     .fault          = filemap_fault,
3427     .map_pages      = filemap_map_pages,
3428     .page_mkwrite   = filemap_page_mkwrite,
3429 };

```

Listing 10-4: mm/filemap.c: generic_file_vm_ops

We have already discussed filemap_fault() in Listing 9-35 and filemap_map_pages() in Listing 9-84 in Section 9.4 above.

However, we have yet to examine filemap_page_mkwrite(), which determines how the folio dirtying takes place, as shown in Listing 10-5.

```

3399 vm_fault_t filemap_page_mkwrite(struct vm_fault *vmf)
3400 {
3401     struct address_space *mapping = vmf->vma->vm_file->f_mapping;
3402     struct folio *folio = page_folio(vmf->page);
3403     vm_fault_t ret = VM_FAULT_LOCKED;
3404
3405     sb_start_pagefault(mapping->host->i_sb);
3406     file_update_time(vmf->vma->vm_file);
3407     folio_lock(folio);
3408     if (folio->mapping != mapping) {
3409         folio_unlock(folio);
3410         ret = VM_FAULT_NOPAGE;
3411         goto out;
3412     }
3413     /*
3414      * We mark the folio dirty already here so that when freeze is in
3415      * progress, we are guaranteed that writeback during freezing will
3416      * see the dirty folio and writeprotect it again.
3417      */
3418     folio_mark_dirty(folio);
3419     folio_wait_stable(folio);

```

```

3420 out:
3421     sb_end_pagefault(mapping->host->i_sb);
3422     return ret;
3423 }

```

Listing 10-5: mm/filemap.c: filemap_page_mkwrite()

This starts by invoking `sb_start_pagefault()` which increments a struct `sb_writers` read/write semaphore specific to page faults, which is in place in order to permit the *freezing* of file systems such that they can be kept stable, as shown in Listing 10-6. `sb_end_pagefault()` reverts this lock.

```

1909 /**
1910  * sb_start_pagefault - get write access to a superblock from a page fault
1911  * @sb: the super we write to
1912  *
1913  * When a process starts handling write page fault, it should embed the
1914  * operation into sb_start_pagefault() - sb_end_pagefault() pair to get
1915  * exclusion against file system freezing. This is needed since the page fault
1916  * is going to dirty a page. This function increments number of running page
1917  * faults preventing freezing. If the file system is already frozen, the
1918  * function waits until the file system is thawed.
1919  *
1920  * Since page fault freeze protection behaves as a lock, users have to
1921  * preserve
1922  * ordering of freeze protection and other filesystem locks. It is advised to
1923  * put sb_start_pagefault() close to mmap_lock in lock ordering. Page fault
1924  * handling code implies lock dependency:
1925  *
1926  * mmap_lock
1927  *   -> sb_start_pagefault
1928  */
1928 static inline void sb_start_pagefault(struct super_block *sb)
1929 {
1930     __sb_start_write(sb, SB_FREEZE_PAGEFAULT);
1931 }

```

Listing 10-6: include/linux/fs.h: sb_start_pagefault()

The `mtime` and `ctime` of the file are updated via `file_update_time()`, and then a folio lock is acquired via `folio_lock()`. As is typical in fault handling logic, this returns the folio locked and expects the caller to unlock as required.

Once the lock is acquired, as is traditional we check to ensure the folio has not been truncated by ensuring the mapping is as expected.

Once we have obtained the correct, locked folio, we go ahead and mark the folio dirty in `folio_mark_dirty()`, shown in Listing 10-8.

We will examine this function shortly, but prior to doing so we will examine what remains, specifically `folio_wait_stable()`, which we explore in Listing 10-7.

```

3065 /**
3066  * folio_wait_stable() - wait for writeback to finish, if necessary.
3067  * @folio: The folio to wait on.
3068  *
3069  * This function determines if the given folio is related to a backing
3070  * device that requires folio contents to be held stable during writeback.
3071  * If so, then it will wait for any pending writeback to complete.
3072  *
3073  * Context: Sleeps. Must be called in process context and with
3074  * no spinlocks held. Caller should hold a reference on the folio.
3075  * If the folio is not locked, writeback may start again after writeback
3076  * has finished.
3077  */
3078 void folio_wait_stable(struct folio *folio)
3079 {
3080     if (folio_inode(folio)->i_sb->s_iflags & SB_I_STABLE_WRITES)
3081         folio_wait_writeback(folio);
3082 }

```

Listing 10-7: mm/page-writeback.c: folio_wait_stable()

Once the page fault is complete and the folio is unlocked, it becomes possible for the mapping userland code to write to the folio at will. If the folio is in the midst of being written back to the underlying block device, a user could race with this code and cause a *torn write*. Consider a buffer that contains:

AAAAAA

Imagine that the first three bytes have been written back to the block device, but afterwards a user writes BBBBBB to the buffer, before writeback completes, permitting the final three bytes to be written from this new buffer state. After this has happened, the disk will contain:

AAABBB

Which will be observable data corruption, and the kernel will mark the folio clean and uptodate as if it reflected the contents of the disk although it does not.

Regardless of what the kernel does to mitigate this, when the second write occurs, the folio will be marked dirty and the on-disk representation will *eventually* be written out and be made consistent with the page cache.

For many modern file systems and backing devices, this suffices. The user has to expect that it is feasible that a write in such a system could end up corrupted however unlikely the race is, and thus power loss might result in observable corruption.

In some cases however, we cannot tolerate this possibility. For instance, the NFS file system allows other systems to observe the written data. Also some block devices might not be able to tolerate this kind of race midway through a write, or either a block device or filesystem may rely on checksums remaining consistent.

In these cases, either the file system institutes its own means of maintaining consistency, or the block device or file system sets a flag to indicate that writes must be *stable*, i.e. that we must wait for any parallel writeback to complete before proceeding.

This is achieved via the `SB_I_STABLE_WRITES` super block flag set in `struct super_block->s_iflags`. This is either set by the file system itself, or determined by the block device in `set_bdev_super()` via the `bdev_stable_writes()` predicated that checks for the `QUEUE_FLAG_STABLE_WRITES` flag.

This is required for a RAID 5 array for instance, which ensures parity across disks which cannot be guaranteed if writes are unstable.

If stable writes are required, then we wait for any remaining writeback to complete via `folio_wait_writeback()`. This forms part of the folio wait logic, and thus is discussed in Section 9.11 and Listing 9-146.

See also Section 10.10 and Listing 10-26 for further discussion about this issue.

Returning to folio dirtying, we can observe this function in Listing 10-8.

```

2717 /**
2718  * folio_mark_dirty - Mark a folio as being modified.
2719  * @folio: The folio.
2720  *
2721  * The folio may not be truncated while this function is running.
2722  * Holding the folio lock is sufficient to prevent truncation, but some
2723  * callers cannot acquire a sleeping lock. These callers instead hold
2724  * the page table lock for a page table which contains at least one page
2725  * in this folio. Truncation will block on the page table lock as it
2726  * unmaps pages before removing the folio from its mapping.
2727  *
2728  * Return: True if the folio was newly dirtied, false if it was already dirty.
2729  */
2730 bool folio_mark_dirty(struct folio *folio)
2731 {
2732     struct address_space *mapping = folio_mapping(folio);
2733
2734     if (likely(mapping)) {
2735         /*
2736          * readahead/lru_deactivate_page could remain
2737          * PG_readahead/PG_reclaim due to race with
2738          folio_end_writeback
2739          * About readahead, if the folio is written, the flags would
2740          be
2741          * reset. So no problem.
2742          * About lru_deactivate_page, if the folio is redirtied,
2743          * the flag will be reset. So no problem. but if the
2744          * folio is used by readahead it will confuse readahead
2745          * and make it restart the size rampup process. But it's
2746          * a trivial problem.
2747          */

```

```

2746         if (folio_test_reclaim(folio))
2747             folio_clear_reclaim(folio);
2748         return mapping->a_ops->dirty_folio(mapping, folio);
2749     }
2750
2751     return noop_dirty_folio(mapping, folio);
2752 }

```

Listing 10-8: mm/page-writeback.c: folio_mark_dirty()

We start by handling a race condition, before deferring to the file system-specific struct address_space->a_ops->dirty_folio function in its struct address_space_operations callback object.

Again, this can be customised per-file system, but many file systems will defer this to a generic library function, block_dirty_folio(), shown in Listing 10-9 (eliding out of scope cgroup logic).

```

591  /*
592   * Add a page to the dirty page list.
593   *
594   * It is a sad fact of life that this function is called from several places
595   * deeply under spinlocking. It may not sleep.
596   *
597   * If the page has buffers, the uptodate buffers are set dirty, to preserve
598   * dirty-state coherency between the page and the buffers. If the page does
599   * not have buffers then when they are later attached they will all be set
600   * dirty.
601   *
602   * The buffers are dirtied before the page is dirtied. There's a small race
603   * window in which a writepage caller may see the page cleanness but not the
604   * buffer dirtiness. That's fine. If this code were to set the page dirty
605   * before the buffers, a concurrent writepage caller could clear the page
606   * dirty
607   * bit, see a bunch of clean buffers and we'd end up with dirty buffers/clean
608   * page on the dirty page list.
609   *
610   * We use private_lock to lock against try_to_free_buffers while using the
611   * page's buffer list. Also use this to protect against clean buffers being
612   * added to the page after it was set dirty.
613   *
614   * FIXME: may need to call ->reservepage here as well. That's rather up to
615   * the
616   * address_space though.
617   */
618  bool block_dirty_folio(struct address_space *mapping, struct folio *folio)
619  {
620      struct buffer_head *head;
621      bool newly_dirty;
622

```

```

621     spin_lock(&mapping->private_lock);
622     head = folio_buffers(folio);
623     if (head) {
624         struct buffer_head *bh = head;
625
626         do {
627             set_buffer_dirty(bh);
628             bh = bh->b_this_page;
629         } while (bh != head);
630     }
631     :
636     newly_dirty = !folio_test_set_dirty(folio);
637     spin_unlock(&mapping->private_lock);
638
639     if (newly_dirty)
640         __folio_mark_dirty(folio, mapping, 1);
641     :
639     if (newly_dirty)
640         __mark_inode_dirty(mapping->host, I_DIRTY_PAGES);
641
642     return newly_dirty;
643 }

```

Listing 10-9: fs/buffer.c: block_dirty_folio()

This starts by performing some block logic, setting any attached struct buffer_head objects to be marked dirty. See Section 9.10.1 for more on buffers heads, but in brief this retrieves attached buffer heads mapping disk blocks to the folio via folio_buffers(), iterates through each setting the dirty flag.

This operation is performed with the struct address_space->private_lock spin lock acquired, as any attached buffer heads are placed in the struct folio->private field.

The actual dirtying of the folio is relatively straightforward, invoking __folio_mark_dirty() as discussed around Listing 10-1 above, which marks the folios dirty in their xarray, and __mark_inode_dirty(), shown in Listing 10-2 which marks the actual inode dirty and adds it to the struct bdi_writeback dirty list as discussed above.

These operations are only performed if the folio was not already marked dirty, and the function returns a boolean to indicate whether this was the case for convenience.

After the folio dirtying is complete, the page fault handler performs various house keeping tasks and then in wp_page_shared() (see Section 6.9 and Listing 6-39), calls fault_dirty_shared_page() (see Listing 6-40), which executes the important balance_dirty_pages_ratelimited() function, which we cover in Section 10.14 below and Listing ??.

This balancing is performed whenever folios are dirtied and is a key part of the writeback mechanism. Since this always occurs we defer the description of this to Section 10.14.

10.5 File Write Dirty Tracking

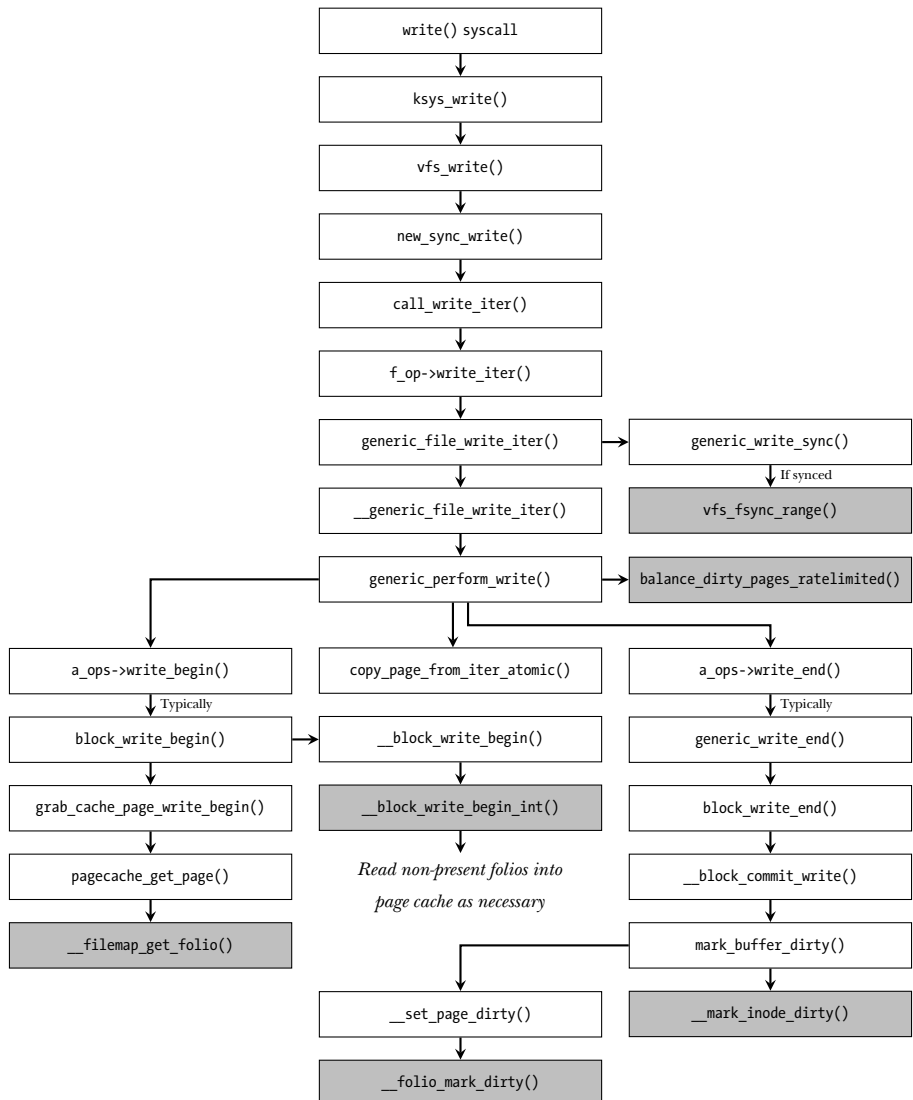


Figure 10-3: Typical File Write Dirty Page Tracking

We have already discussed how files are read from the page cache using the `read()` system call in Section 9.3. On write we perform essentially the same task as a write page fault does (discussed in Section 10.12 above), only span-

ning the whole range of the write and copying data from the input user buffer into the page cache folios.

Assuming a file system exposes the struct `file->f_op->write_iter` callback, then the write system call will call `ksys_write()`, which calls `vfs_write()` and `new_sync_write()` in turn which invokes struct `file->f_op->write_iter` in call `write_iter()`.

Again, the file system is free to do what it wants, however in very many cases file systems use the library function `generic_file_write_iter()`. This in turn invokes `__generic_file_write_iter()` to perform the heavy lifting (with the struct `inode->i_rwsem` read/write semaphore held for write), and `generic_write_sync()` to synchronise the data to disk if the write was configured to do so.

We discuss write synchronisation in Section 10.6, and examine `generic_write_sync()` in Listing 10-14 within that section.

We examine `__generic_file_write_iter()` in Listing 10-10, eliding out of scope direct write logic.

```

3779 /**
3780  * __generic_file_write_iter - write data to a file
3781  * @iocb:      IO state structure (file, offset, etc.)
3782  * @from:      iov_iter with data to write
3783  *
3784  * This function does all the work needed for actually writing data to a
3785  * file. It does all basic checks, removes SUID from the file, updates
3786  * modification times and calls proper subroutines depending on whether we
3787  * do direct IO or a standard buffered write.
3788  *
3789  * It expects i_rwsem to be grabbed unless we work on a block device or
3790  * similar
3791  * object which does not need locking at all.
3792  *
3793  * This function does not take care of syncing data in case of O_SYNC write.
3794  * A caller has to handle it. This is mainly due to the fact that we want to
3795  * avoid syncing under i_rwsem.
3796  *
3797  * Return:
3798  * * number of bytes written, even for truncated writes
3799  * * negative error code if no data has been written at all
3800  */
3801 ssize_t __generic_file_write_iter(struct kiocb *iocb, struct iov_iter *from)
3802 {
3803     struct file *file = iocb->ki_filp;
3804     struct address_space *mapping = file->f_mapping;
3805     struct inode *inode = mapping->host;
3806     ssize_t written = 0;
3807     ssize_t err;
3808     ssize_t status;

```

```

3809      /* We can write back this queue in page reclaim */
3810      current->backing_dev_info = inode_to_bdi(inode);
3811      err = file_remove_privs(file);
3812      if (err)
3813          goto out;
3814
3815      err = file_update_time(file);
3816      if (err)
3817          goto out;
3818      :
3866      written = generic_perform_write(iocb, from);
3867      if (likely(written > 0))
3868          iocb->ki_pos += written;
3869      :
3870 out:
3871      current->backing_dev_info = NULL;
3872      return written ? written : err;
3873 }

```

Listing 10-10: mm/filemap.c: __generic_file_write_iter()

This function first sets the current thread's backing device field to the inode's backing device object obtained from `inode_to_bdi()`, removes any special file privileges that might exist such as `suid` or capabilities via `file_remove_privs()`, updates `mtime` and `ctime` accordingly via `file_update_time()` before deferring to `generic_perform_write()` to execute the write.

After the write is complete, the iterator write position is updated accordingly and the current thread's backing device field is cleared.

The actual 'meat' of the write is performed in `generic_perform_write()` which we examine in Listing 10-11 (eliding out of scope real time kernel logic and x86-64 irrelevant `dcache` flushes).

```

3701 ssize_t generic_perform_write(struct kiocb *iocb, struct iov_iter *i)
3702 {
3703     struct file *file = iocb->ki_filp;
3704     loff_t pos = iocb->ki_pos;
3705     struct address_space *mapping = file->f_mapping;
3706     const struct address_space_operations *a_ops = mapping->a_ops;
3707     long status = 0;
3708     ssize_t written = 0;
3709
3710     do {
3711         struct page *page;
3712         unsigned long offset; /* Offset into pagecache page */
3713         unsigned long bytes; /* Bytes to write to page */
3714         size_t copied; /* Bytes copied from user */
3715         void *fsdata;
3716
3717         offset = (pos & (PAGE_SIZE - 1));

```

```

3718         bytes = min_t(unsigned long, PAGE_SIZE - offset,
3719                        iov_iter_count(i));
3720
3721     again:
3722         /*
3723          * Bring in the user page that we will copy from _first_.
3724          * Otherwise there's a nasty deadlock on copying from the
3725          * same page as we're writing to, without it being marked
3726          * up-to-date.
3727          */
3728         if (unlikely(fault_in_iov_iter_readable(i, bytes) == bytes)) {
3729             status = -EFAULT;
3730             break;
3731         }
3732
3733         if (fatal_signal_pending(current)) {
3734             status = -EINTR;
3735             break;
3736         }
3737
3738         status = a_ops->write_begin(file, mapping, pos, bytes,
3739                                   &page, &fsdata);
3740         if (unlikely(status < 0))
3741             break;
3742         :
3746         copied = copy_page_from_iter_atomic(page, offset, bytes, i);
3747         :
3749         status = a_ops->write_end(file, mapping, pos, bytes, copied,
3750                                  page, fsdata);
3751         if (unlikely(status != copied)) {
3752             iov_iter_revert(i, copied - max(status, 0L));
3753             if (unlikely(status < 0))
3754                 break;
3755         }
3756         :
3758         if (unlikely(status == 0)) {
3759             /*
3760              * A short copy made ->write_end() reject the
3761              * thing entirely. Might be memory poisoning
3762              * halfway through, might be a race with munmap,
3763              * might be severe memory pressure.
3764              */
3765             if (copied)
3766                 bytes = copied;
3767             goto again;
3768         }
3769         pos += status;

```



```

3770             written += status;
3771
3772             balance_dirty_pages_ratelimited(mapping);
3773         } while (iov_iter_count(i));
3774
3775         return written ? written : status;
3776     }

```

Listing 10-11: mm/filemap.c: generic_perform_write()

For brevity we will skip over some of the iterator housekeeping here which we have already examined in detail in Section 9.3 for the read case, and focus on the parts of the function specific to the write operation.

The function is divided into three key areas, as highlighted in Figure 10-3—a ‘write begin’ operation, the copying of the data from the user buffer into the page cache folio via `copy_page_from_iter_atomic()`, and a ‘write end’ operation which ultimately marks dirty folios in the struct `address_space->i_pages` xarray and adds dirty inodes to the struct `bdi_writeback->b_dirty` dirty list.

As usual, the write begin operation, struct `address_space->a_ops->write_begin` (part of the struct `address_space_operations` callback object) is customisable by the file system, but is often deferred to the generic library function `block_write_begin()`, which we examine in Listing 10-12.

```

2100  /*
2101   * block_write_begin takes care of the basic task of block allocation and
2102   * bringing partial write blocks uptodate first.
2103   *
2104   * The filesystem needs to handle block truncation upon failure.
2105   */
2106  int block_write_begin(struct address_space *mapping, loff_t pos, unsigned len,
2107                      struct page **pagep, get_block_t *get_block)
2108  {
2109      pgoff_t index = pos >> PAGE_SHIFT;
2110      struct page *page;
2111      int status;
2112
2113      page = grab_cache_page_write_begin(mapping, index);
2114      if (!page)
2115          return -ENOMEM;
2116
2117      status = __block_write_begin(page, pos, len, get_block);
2118      if (unlikely(status)) {
2119          unlock_page(page);
2120          put_page(page);
2121          page = NULL;
2122      }
2123

```

```

2124     *pagep = page;
2125     return status;
2126 }

```

Listing 10-12: fs/buffer.c: block_write_begin()

This starts by actually pulling the page cache folio from the page cache via `grab_cache_page_write_begin()`, shown in Listing ??.

```

111 struct page *grab_cache_page_write_begin(struct address_space *mapping,
112                                         pgoff_t index)
113 {
114     unsigned fgp_flags = FGP_LOCK | FGP_WRITE | FGP_CREAT | FGP_STABLE;
115
116     return pagecache_get_page(mapping, index, fgp_flags,
117                               mapping_gfp_mask(mapping));
118 }

```

Listing 10-13: mm/folio-compat.c: grab_cache_page_write_begin()

The `pagecache_get_page()` function this defers to ultimately wraps the `__filemap_get_folio()` function we examined in Section 9.5.3 and the Listings starting with ??.

This sets the `FGP_LOCK`, `FGP_WRITE`, `FGP_CREAT` and `FGP_STABLE` flags which cause the returned folio to be locked, to be writable, to be created if it does not exist and for it to have had `folio_wait_stable()` run after obtaining it (see Listing 10-7 for more details on what this stability means).

This means that this function should either return the page cache entry or NULL if we were unable to allocate a new one. Therefore `block_write_begin()` exits with an `ENOMEM` error if this fails.

`block_write_begin()` defers its heavy lifting to `__block_write_begin()` which, in turn, defers its heavy lifting to `__block_write_begin_int()`.

The `__block_write_begin_int()` function sets up `struct buffer_head` objects, described in Section 9.10.1, which map disk blocks to folios and reads any data into the page cache that has not previously been read before proceeding.

This function is very block I/O specific, so discussion of this is out of scope. However see Section 9.10.4 for a brief discussion on block writes.

We now have all page cache folios present in memory and can safely copy from the user buffer into the page cache entries via `copy_page_from_iter_atomic()`. Once this is done, it's time to mark the folios and inodes dirty ready for writeback.

Again, the file system has a great deal of control over this, with the write end callback being specified in `struct address_space->a_ops->write_end`. However again file systems often defer this to the library function `generic_write_end()`.

The `generic_write_end()` function performs general housekeeping but otherwise defers the heavy lifting to `block_write_end()`, which in turn defers the key logic to `__block_commit_write()`.

The `__block_commit_write()` function is rather block I/O-specific, so we defer examination of this to Section 9.10.4 and Listing 9-132.

However the key thing to note is that on first dirtying a folio, this function invokes `mark_buffer_dirty()`. Again we defer discussion of this to Section 9.10.4 below, and in Listing 9-133.

However regardless we observe the key fact that, if newly dirty, the relevant folio is marked dirty by `mark_buffer_dirty()` using `__set_page_dirty()` (which is simply a wrapper around `__folio_mark_dirty()`) as discussed around Listing 10-1 above. This marks folios dirty in their relevant xarray.

`mark_buffer_dirty()` also marks the inode dirty via `__mark_inode_dirty()`, shown in Listing 10-2 above, which marks the actual inode dirty and adds it to the struct `bdi_writeback` dirty list as previously discussed.

This explains clearly how writing a file results in page cache folios being correctly marked dirty. However one more task is performed by `generic_perform_write()`, which is to perform dirty page balancing via `balance_dirty_pages_ratelimited()`, as discussed below in Section 10.14 and Listing 10-79.

10.6 Synchronising to Disk

Sometimes a user needs to ensure that the page cache is synchronised to disk manually which ultimately invokes the core writeback logic described in Section 10.10. This can be achieved through various system calls as shown in Figure 10-4 (we elide the `sync_file_range()` and `sync_file_range2()` system calls here as these are considered dangerous).

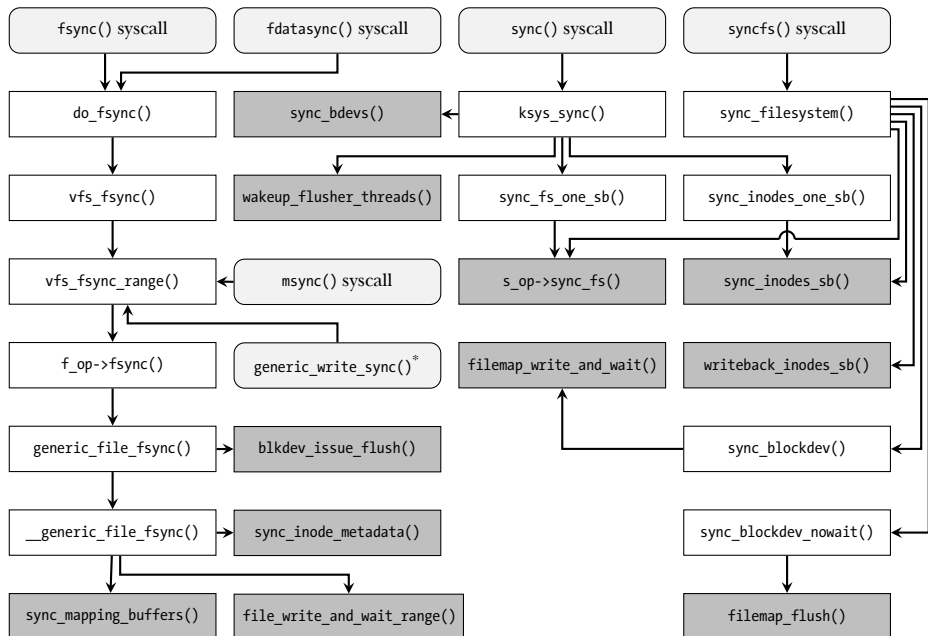


Figure 10-4: Page Cache Synchronisation

Key:

-  Denotes that A is an entry point to the synchronisation process.
-  Denotes that B is a function which forms part of the actual write back procedure described in Section 10.10.

*Note that we also include the `generic_write_sync()` call, which we discussed in Section 10-3. This is invoked by `generic_file_write_iter()` if a file is opened with synchronisation options specified (and the file system opts to use this general library function). We examine this function in Listing 10-14.

```

2861 /*
2862  * Sync the bytes written if this was a synchronous write. Expect ki_pos
2863  * to already be updated for the write, and will return either the amount
2864  * of bytes passed in, or an error if syncing the file failed.
2865  */
2866 static inline ssize_t generic_write_sync(struct kiocb *iocb, ssize_t count)
2867 {
2868     if (iocb_is_dsync(iocb)) {
2869         int ret = vfs_fsync_range(iocb->ki_filp,
2870                                   iocb->ki_pos - count, iocb->ki_pos - 1,
2871                                   (iocb->ki_flags & IOCB_SYNC) ? 0 : 1);
2872         if (ret)
2873             return ret;
2874     }
2875     return count;
2876 }
2877 
```

Listing 10-14: `include/linux/fs.h: generic_write_sync()`

This ultimately defers the operation to `vfs_fsync_range()` which we describe below in Listing 10-19. The operation is performed when the iterator described by `struct kiocb` is configured to synchronise as checked for by `iocb_is_dsync()`.

10.7 sync

The most general means of synchronising the page cache to disk is via the `sync()` syscall, which is familiar to most Linux users via the `sync` CLI tool which, if no parameters are specified, invokes this system call to perform a general file system sync. We examine the `sync()` system call in Listing 10-15.

```

111 SYSCALL_DEFINE0(sync)
112 {
113     ksys_sync();

```

```

114         return 0;
115     }

```

Listing 10-15: fs/sync.c: sync() system call

This defers the core of the operation to `ksys_sync()` which we examine in Listing 10-16.

Note that we keep track of the reason for writeback using the enum `wb_reason` enumeration type. We indicate here that we are performing writeback to perform synchronisation via `WB_REASON_SYNC`.

```

87  /*
88   * Sync everything. We start by waking flusher threads so that most of
89   * writeback runs on all devices in parallel. Then we sync all inodes reliably
90   * which effectively also waits for all flusher threads to finish doing
91   * writeback. At this point all data is on disk so metadata should be stable
92   * and we tell filesystems to sync their metadata via ->sync_fs() calls.
93   * Finally, we writeout all block devices because some filesystems (e.g. ext2)
94   * just write metadata (such as inodes or bitmaps) to block device page cache
95   * and do not sync it on their own in ->sync_fs().
96   */
97  void ksys_sync(void)
98  {
99      int nowait = 0, wait = 1;
100
101      wakeup_flusher_threads(WB_REASON_SYNC);
102      iterate_supers(sync_inodes_one_sb, NULL);
103      iterate_supers(sync_fs_one_sb, &nowait);
104      iterate_supers(sync_fs_one_sb, &wait);
105      sync_bdevs(false);
106      sync_bdevs(true);
107      if (unlikely(laptop_mode))
108          laptop_sync_completion();
109  }

```

Listing 10-16: fs/sync.c: ksys_sync()

The function pipelines the synchronisation by first ensuring I/O is submitted for *all* pending writeback requests, before synchronising *each* I/O operation, waiting for each to complete one-by-one. This proceeds as follows:

Wake up flusher threads We start by waking up the background writeback flushing kernel thread via `wakeup_flusher_threads()`. We examine the behaviour of these threads in Section 10.10 below. This should cause I/O operations to be submitted for all dirty inodes.

Synchronously writeback Iterate through each super block performing synchronisation using `iterate_supers()` to perform the iteration, synchronising data and filesystem metadata via `sync_inodes_one_sb()`. This wraps a call to `sync_inodes_sb()`, which we examine in Section 10.10 and Listing

10-39. This is the key function which actually performs the synchronised writeback.

Synchronise super block metadata Ensure superblock metadata is synchronised as implemented by the super block operation (provided in the callback object of type struct `super_operations`) specified in struct `super_block->s_op->sync_fs`. This is wrapped by `sync_fs_one_sb()`. As this concerns filesystem metadata we don't examine it in detail.

Synchronise blockdev device Synchronise I/O operations caused by filesystem metadata writes (see Section 9.10.2 for more on the raw blockdev file system) via `sync_bdevs()`, which is out of scope for the book.

Laptop mode handling This is a special mode which is out of scope of the book.

10.8 syncfs

The `syncfs()` system call performs the same task as `sync()` only it operates *only* on the file system containing the specified file. The `syncfs()` system call wraps a call to `sync_filesystem()`, described in Listing 10-17.

```

25  /*
26   * Write out and wait upon all dirty data associated with this
27   * superblock. Filesystem data as well as the underlying block
28   * device. Takes the superblock lock.
29   */
30  int sync_filesystem(struct super_block *sb)
31  {
32      int ret = 0;
33
34      /*
35       * We need to be protected against the filesystem going from
36       * r/o to r/w or vice versa.
37       */
38      WARN_ON(!rwsem_is_locked(&sb->s_umount));
39
40      /*
41       * No point in syncing out anything if the filesystem is read-only.
42       */
43      if (sb_rdonly(sb))
44          return 0;
45
46      /*
47       * Do the filesystem syncing work. For simple filesystems
48       * writeback_inodes_sb(sb) just dirties buffers with inodes so we have
49       * to submit I/O for these buffers via sync_blockdev(). This also
50       * speeds up the wait == 1 case since in that case write_inode()
51       * methods call sync_dirty_buffer() and thus effectively write one
52       block

```

```

52     * at a time.
53     */
54     writeback_inodes_sb(sb, WB_REASON_SYNC);
55     if (sb->s_op->sync_fs) {
56         ret = sb->s_op->sync_fs(sb, 0);
57         if (ret)
58             return ret;
59     }
60     ret = sync_blockdev_nowait(sb->s_bdev);
61     if (ret)
62         return ret;
63
64     sync_inodes_sb(sb);
65     if (sb->s_op->sync_fs) {
66         ret = sb->s_op->sync_fs(sb, 1);
67         if (ret)
68             return ret;
69     }
70     return sync_blockdev(sb->s_bdev);
71 }

```

Listing 10-17: fs/sync.c: sync_filesystem()

This proceeds only if the super block is read/write (naturally), checked by `sb_rdonly()`. It follows the pattern that `ksys_sync()` establishes of initiating background writeback before performing blocking writeback.

Initiate folio writeback We begin the process of writing back on this super block via `writeback_inodes_sb()`. This is performed asynchronously. See Listing 10-40 in Section 10.10 for details.

Initiate super block synchronisation As in `ksys_sync()` super block meta-data synchronisation is performed via `s_op->sync_fs`, in this instance without waiting on the process to complete.

Initiate blockdev synchronisation File system metadata and data written raw to the block device is stored in the blockdev file system, described in Section 9.10.2. The `sync_blockdev_nowait()` function wraps a call to `filemap_flush()` on the blockdev inode to trigger asynchronous writeback on this data. See Listing 10-68 in Section 10.10 for details.

Wait on folio writeback We now perform writeback of dirty folios, waiting on each to operation to complete in `sync_inodes_sb()` as shown in Listing 10-39 and Section 10.10.

Wait on super block synchronisation We invoke `s_op->sync_fs()` again, only specifying that this should be done synchronously waiting on each operation. This synchronises super block metadata.

Wait on blockdev synchronisation Finally we wait on writeback for the blockdev file system describing the block device the super block is mounted on via `sync_blockdev()`. This ultimately invokes

`filemap_write_and_wait()` which invokes `filemap_write_and_wait_range()` for the entire address range. We explore this function in Listing 10-71 and Section 10.10.

10.9 fsync

Synchronisation can be performed on a file level as well as on a general or file system level. This is achieved via either the `fsync()` or `fdatasync()` system calls, with kernel implementations in `fsync()` and `fdatasync()` respectively.

Both of these functions ultimately invoke `do_fsync()` as shown in Listing 10-18, with the former setting `datasync` to 0 (i.e. false) and the latter setting it to 1 (i.e. true). This indicates whether filesystem metadata should be written back alongside data folios (the former case) or not (the latter case).

```

206 static int do_fsync(unsigned int fd, int datasync)
207 {
208     struct fd f = fdget(fd);
209     int ret = -EBADF;
210
211     if (f.file) {
212         ret = vfs_fsync(f.file, datasync);
213         fdput(f);
214     }
215     return ret;
216 }
```

Listing 10-18: `fs/sync.c: do_fsync()`

This increments the reference count for the `struct file` associated with the supplied `fd` if valid and deferring the operation to `vfs_fsync()`, before dropping the reference count after this is complete and returning any error that arose.

The `vfs_fsync()` function is a simple wrapper around the `vfs_fsync_range()`, passing an input range specifying that the entirety of the file should be synchronised. We examine this function in Listing 10-19 (eliding out of scope dirty time logic).

```

169 /**
170  * vfs_fsync_range - helper to sync a range of data & metadata to disk
171  * @file:           file to sync
172  * @start:          offset in bytes of the beginning of data range to sync
173  * @end:            offset in bytes of the end of data range (inclusive)
174  * @datasync:       perform only datasync
175  *
176  * Write back data in range @start..@end and metadata for @file to disk. If
177  * @datasync is set only metadata needed to access modified file data is
178  * written.
179  */
180 int vfs_fsync_range(struct file *file, loff_t start, loff_t end, int datasync)
```

```

181 {
182     struct inode *inode = file->f_mapping->host;
183
184     if (!file->f_op->fsync)
185         return -EINVAL;
186     :
187
188     return file->f_op->fsync(file, start, end, datasync);
189 }

```

Listing 10-19: fs/sync.c: `vfs_fsync_range()`

This defers the fsync operation to the struct `file->f_op->fsync` function from its struct `file_operations` callback object.

As always, file systems are able to implement things however they choose, however typically they invoke a library function provided for this operation, `generic_file_fsync()` shown in Listing ??.

```

1140 /**
1141  * generic_file_fsync - generic fsync implementation for simple filesystems
1142  *                      with flush
1143  * @file:               file to synchronize
1144  * @start:              start offset in bytes
1145  * @end:               end offset in bytes (inclusive)
1146  * @datasync:          only synchronize essential metadata if true
1147  *
1148  */
1149
1150 int generic_file_fsync(struct file *file, loff_t start, loff_t end,
1151                       int datasync)
1152 {
1153     struct inode *inode = file->f_mapping->host;
1154     int err;
1155
1156     err = __generic_file_fsync(file, start, end, datasync);
1157     if (err)
1158         return err;
1159     return blkdev_issue_flush(inode->i_sb->s_bdev);
1160 }

```

Listing 10-20: fs/libfs.c: `generic_file_fsync()`

This defers the heavy lifting of the operation to `__generic_file_fsync()`, issuing a block device flush operation on the underlying block device via `blkdev_issue_flush()` afterwards, shown in Listing 9-134 described in Section 9.10.5.

We examine `__generic_file_fsync()` in Listing 10-21.

```

1096 /**
1097  * __generic_file_fsync - generic fsync implementation for simple filesystems
1098  *

```

```

1099 * @file:      file to synchronize
1100 * @start:     start offset in bytes
1101 * @end:       end offset in bytes (inclusive)
1102 * @datasync:  only synchronize essential metadata if true
1103 *
1104 * This is a generic implementation of the fsync method for simple
1105 * filesystems which track all non-inode metadata in the buffers list
1106 * hanging off the address_space structure.
1107 */
1108 int __generic_file_fsync(struct file *file, loff_t start, loff_t end,
1109                        int datasync)
1110 {
1111     struct inode *inode = file->f_mapping->host;
1112     int err;
1113     int ret;
1114
1115     err = file_write_and_wait_range(file, start, end);
1116     if (err)
1117         return err;
1118
1119     inode_lock(inode);
1120     ret = sync_mapping_buffers(inode->i_mapping);
1121     if (!(inode->i_state & I_DIRTY_ALL))
1122         goto out;
1123     if (datasync && !(inode->i_state & I_DIRTY_DATASYNC))
1124         goto out;
1125
1126     err = sync_inode_metadata(inode, 1);
1127     if (ret == 0)
1128         ret = err;
1129
1130 out:
1131     inode_unlock(inode);
1132     /* check and advance again to catch errors after syncing out buffers
1133     */
1134     err = file_check_and_advance_wb_err(file);
1135     if (ret == 0)
1136         ret = err;
1137     return ret;
1138 }

```

Listing 10-21: fs/libfs.c: `__generic_file_fsync()`

This performs the write via `file_write_and_wait_range()` which we examine in Listing 10-74 and Section 10.10.

We acquire the inode lock and submit any outstanding I/O related to the file via `sync_mapping_buffers()`, which is out of scope for the book.

Now we have the inode lock we can then check the current state of the inode to determine whether to write back inode-specific metadata via `sync_inode_metadata()` — if the inode is not dirty at all then there is no need, equally if the user has *not* specified that only data pages should be written back (as specified by the `datasync` parameter) we do write back.

We also cover the edge case where the caller specified that we should only writeback data pages but the inode is still marked dirty for data pages, then it is appropriate to write this metadata back to disk.

Finally, we retrieve any errors that arose during the operation and return them to the caller via `file_check_and_advance_wb_err()`.

10.10 Writing Back to Disk

The code paths which perform the ultimate writeback operation to disk are shown in Figure 10-5.

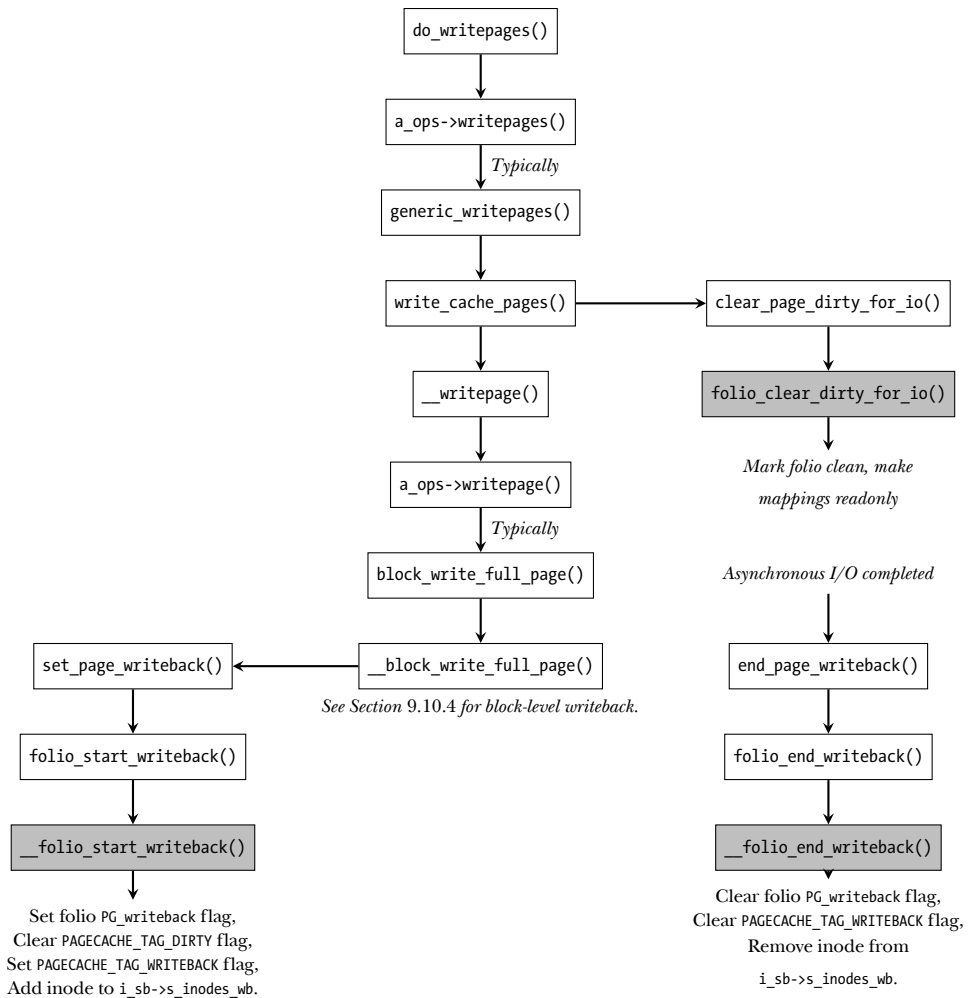


Figure 10-5: Core Writeback Code Paths

There are two principle means through which data is ultimately written back to disk—periodically by background flusher threads, or directly through a flush operation (see Section 10.6 for details on how this can be performed via system call).

However from the kernel’s perspective there is a more important distinction—those operations which span file systems, and are dealt with by the background flusher thread (confusingly spanning both background writeback *and* general synchronisation), which we explore in Section 10.11, and those which are a product of file synchronisation which we explore in Section 10.13.

Both however must interact with block device (BDI) dirty tracking. Recall from figure 10-1 that each BDI has one or more `struct bdi_writeback` objects associated with it, each of which independently represents pending

writeback work which can be *throttled* according to the current I/O bandwidth of the block device.

The only instance in which there will be more struct `bdi_writeback` objects than BDIs is where memory cgroups are utilised for writeback. This is out of scope for the book.

We examine the struct `bdi_writeback` type in Listing 10-22, eliding out of scope cgroup logic and dirty time stamp tracking.

```

83  /*
84   * Each wb (bdi_writeback) can perform writeback operations, is measured
85   * and throttled, independently. Without cgroup writeback, each bdi
86   * (bdi_writeback) is served by its embedded bdi->wb.
87   :
104  */
105  struct bdi_writeback {
106      struct backing_dev_info *bdi; /* our parent bdi */
107
108      unsigned long state; /* Always use atomic bitops on this */
109      unsigned long last_old_flush; /* last old data flush */
110
111      struct list_head b_dirty; /* dirty inodes */
112      struct list_head b_io; /* parked for writeback */
113      struct list_head b_more_io; /* parked for more writeback */
114      :
115      spinlock_t list_lock; /* protects the b_* lists */
116
117      atomic_t writeback_inodes; /* number of inodes under writeback */
118      struct percpu_counter stat[NR_WB_STAT_ITEMS];
119
120      unsigned long bw_time_stamp; /* last time write bw is updated */
121      unsigned long dirtied_stamp;
122      unsigned long written_stamp; /* pages written at bw_time_stamp */
123      unsigned long write_bandwidth; /* the estimated write bandwidth */
124      unsigned long avg_write_bandwidth; /* further smoothed write bw, > 0
125
126      /*
127       * The base dirty throttle rate, re-calculated on every 200ms.
128       * All the bdi tasks' dirty rate will be curbed under it.
129       * @dirty_ratelimit tracks the estimated @balanced_dirty_ratelimit
130       * in small steps and is much more smooth/stable than the latter.
131       */
132      unsigned long dirty_ratelimit;
133      unsigned long balanced_dirty_ratelimit;
134
135      struct fprop_local_percpu completions;
136      int dirty_exceeded;
137      enum wb_reason start_all_reason;

```

```

138
139     spinlock_t work_lock;           /* protects work_list & dwork
        scheduling */
140     struct list_head work_list;
141     struct delayed_work dwork;      /* work item used for writeback */
142     struct delayed_work bw_dwork;   /* work item used for bandwidth
        estimate */
143
144     unsigned long dirty_sleep;      /* last wait */
145
146     struct list_head bdi_node;      /* anchored at bdi->wb_list */
147     :
163 };

```

Listing 10-22: include/linux/backing-dev-defs.h: *struct bdi_writeback*

We already discussed the `b_dirty` list in Section 10.1, which is the *key* list which keeps track of all the dirty inodes requiring writeback.

We also maintain a great many statistics which are used to maintain flusher thread behaviour as described at the start of Section ??, which we will discuss shortly when examining background writeback, and the process of *balancing* dirty writes as a whole in Section 10.14 below.

The key fields for background writeback are `b_io` and `b_more_io` which keep track of inodes which have been scheduled for writeback immediately, and those which will be processed on the next background flush interval. See Section 10.11 for a detailed examination of the background writeback process.

Ultimately writeback is performed against specific inodes, with details as to how the writeback should be performed expressed via a `struct writeback_control` object (see Listing 10-24) in `do_writepages()` (see Listing 10-23).

However, when specifying writeback spanning a subset of dirty inodes with specific behaviour as we do for background writeback (see Section 10.11) and synchronisation (see Section 10.6), we need to specify required writeback behaviour using only a subset of `struct writeback_control` fields.

This is achieved by the `struct wb_writeback_work` type (we examine this in Section 10.11, see Listing 10-38) which is ultimately passed into `wb_writeback()` (see Listing 10-55 in the same section).

When work *in addition* to the background writeback described in Section 10.11 is required to be performed (for instance by synchronisation), these are specified in the `struct bdi_writeback->work_list` field and processed by `wb_do_writeback()` (see Listing 10-50 in Section 10.11) when the background flusher thread is awoken.

No matter what path is taken, the actual process of writeback is performed in `do_writepages()` as shown in Listing 10-23.

```

2457 int do_writepages(struct address_space *mapping, struct writeback_control *wbc
        )
2458 {

```

```

2459     int ret;
2460     struct bdi_writeback *wb;
2461
2462     if (wbc->nr_to_write <= 0)
2463         return 0;
2464     wb = inode_to_wb_wbc(mapping->host, wbc);
2465     wb_bandwidth_estimate_start(wb);
2466     while (1) {
2467         if (mapping->a_ops->writepages)
2468             ret = mapping->a_ops->writepages(mapping, wbc);
2469         else
2470             ret = generic_writepages(mapping, wbc);
2471         if ((ret != -ENOMEM) || (wbc->sync_mode != WB_SYNC_ALL))
2472             break;
2473
2474         /*
2475          * Lacking an allocation context or the locality or writeback
2476          * state of any of the inode's pages, throttle based on
2477          * writeback activity on the local node. It's as good a
2478          * guess as any.
2479          */
2480         reclaim_throttle(NODE_DATA(numa_node_id()),
2481                         VMSCAN_THROTTLE_WRITEBACK);
2482     }
2483     /*
2484      * Usually few pages are written by now from those we've just
2485      submitted
2486      * but if there's constant writeback being submitted, this makes sure
2487      * writeback bandwidth is updated once in a while.
2488      */
2489     if (time_is_before_jiffies(READ_ONCE(wb->bw_time_stamp) +
2490                                BANDWIDTH_INTERVAL))
2491         wb_update_bandwidth(wb);
2492     return ret;
2493 }

```

Listing 10-23: mm/page-writeback.c: `do_writepages()`

We start by obtaining the struct `bdi_writeback` object associated with this struct `address_space` page cache object using `inode_to_wb_wbc()`.

We maintain an estimate on writeback bandwidth using per-CPU statistics for the number of dirtied and written base pages, maintained in struct `bdi_writeback->stat` array.

Before writeback we set the current dirtied and written base pages in the `dirtied_stamp` and `written_stamp` fields and the last updated timestamp field `bw_time_stamp` via `wb_bandwidth_estimate_start()`.

We then enter a loop in which the actual write operation is deferred to the struct `address_space->a_ops->writepages` handler (from the page cache en-

try's struct `address_space_operations` callback object), or `generic_writepages()` if the file system does not specify one.

The loop continues only if the `writepages` operation failed due to lack of memory and the struct `writeback_control`-specified synchronisation mode is set to `WB_SYNC_ALL`, i.e. indicating that the caller expects the operation to be complete on return.

In this instance reclaim behaviour is adjusted to account for this scenario in `reclaim_throttle()` (see the chapter on reclaim for more details on this).

Finally, if we are within the bandwidth interval `BANDWIDTH_INTERVAL`, hard-coded to 200ms, we update the bandwidth statistics via `wb_update_bandwidth()`.

Regardless of whether `generic_writepages()` is used or a file system-supplied callback is used, an ordinary file systems will invoke `write_cache_pages()` shown in Listing 10-25, passing in a callback which writes to each page of function type `writepage_t`, i.e. `writepage(struct page *page, struct writeback_control *wbc, void *data)`.

Before examining `write_cache_pages()` let's examine struct `writeback_control` in Listing 10-24 (eliding out of scope `cgroup` and `swap plug logic`).

```

45  /*
46   * A control structure which tells the writeback code what to do. These are
47   * always on the stack, and hence need no locking. They are always
48   * initialised
49   * in a manner such that unspecified fields are set to zero.
50   */
51  struct writeback_control {
52      long nr_to_write;           /* Write this many pages, and
53                                  decrement
54                                  this for each page written */
55      long pages_skipped;        /* Pages which were not written */
56      /*
57       * For a_ops->writepages(): if start or end are non-zero then this is
58       * a hint that the filesystem need only write out the pages inside
59       * that
60       * byterange. The byte at `end' is included in the writeout request.
61       */
62      loff_t range_start;
63      loff_t range_end;
64
65      enum writeback_sync_modes sync_mode;
66
67      unsigned for_kupdate:1;    /* A kupdate writeback */
68      unsigned for_background:1; /* A background writeback */
69      unsigned tagged_writepages:1; /* tag-and-write to avoid livelock */
70      unsigned for_reclaim:1;    /* Invoked from the page allocator */
71      unsigned range_cyclic:1;   /* range_start is cyclic */
72      unsigned for_sync:1;       /* sync(2) WB_SYNC_ALL writeback */

```



```

71         unsigned unpinned_fscache_wb:1; /* Cleared I_PINNING_FSCACHE_WB */
    :
102 };

```

Listing 10-24: include/linux/writeback.h: struct writeback_control

Examining each field:

nr_to_write The number of pages to write, or `LONG_MAX` if all dirty pages should be written. This is ignored if the synchronisation mode is set to anything other than `WB_SYNC_NONE`, i.e. if synchronisation is required on each writeback.

pages_skipped The number of pages which were not able to be written, accumulated through the writeback operation.

range_start The start of the range to be written, or 0 if all dirty pages in the inode should be written.

range_end The *inclusive* end of the range, or `LLONG_MAX` if all dirty pages in the inode should be written.

sync_mode Indicates whether the kernel should wait on each write. This field is of the type `enum writeback_sync_modes`. If set to `WB_SYNC_NONE` no sync is performed, otherwise if set to `WB_SYNC_ALL` the kernel will wait after each writeback.

for_kupdate Bit flag indicating that this is an ordinary background flush periodically writing back inodes older than `vm.dirty_expire_centisecs`. See Section 10.11.

for_background Bit flag indicating that either the ratio between dirty pages and available memory has exceeded `vm.dirty_background_ratio` or the sum of dirty bytes has exceeded `v.dirty_background_bytes` and therefore writeback is required until this is resolved.

tagged_writepages Bit field used to avoid a livelock situation where more and more pages are marked dirty faster than we can write them back. This causes all entries in the `struct address_space->i_pages` xarray tagged with `PAGECACHE_TAG_DIRTY` to be tagged with `PAGECACHE_TAG_TOWRITE` and for these to be looked up by `write_cache_pages()` rather than dirty-tagged pages.

for_reclaim Bit field that indicates that the writeback has been triggered by reclaim in `pageout()`. See the reclaim chapter for more details.

range_cyclic Bit field that indicates that rather than specifying a range over which to scan in `range_start` and `range_end`, we operate cyclically over the entire range of pages within the `struct address_space`, using its `writeback_index` field to store the page offset of the next page to write back.

for_sync Bit field indicating that a full sync is being performed. When queueing I/O via `queue_io()` as part of the background writeback process, if this flag is set, the expire time is not taken into account. In addition, `__writeback_single_inode()` does not wait on writeout as a calling

function handles this separately. This is set in `sync_inodes_sb()` and indicates that the user has invoked a `sync()` system call.

unpinned_fscache_wb Indicates that an underlying folio is pinned by the file system. Out of scope for the book.

Now we have examined the struct `writeback_control` type, we can examine how it is used on writeback in `write_cache_pages()` in Listing 10-25 (eliding out of scope trace logic).

```

2250 /**
2251  * write_cache_pages - walk the list of dirty pages of the given address space
2252  *                    and write all of them.
2253  * @mapping: address space structure to write
2254  * @wbc: subtract the number of written pages from *@wbc->nr_to_write
2255  * @writepage: function called for each page
2256  * @data: data passed to writepage function
2257  *
2258  * If a page is already under I/O, write_cache_pages() skips it, even
2259  * if it's dirty. This is desirable behaviour for memory-cleaning writeback,
2260  * but it is INCORRECT for data-integrity system calls such as fsync(). fsync
2261  * ()
2262  * and msync() need to guarantee that all the data which was dirty at the time
2263  * the call was made get new I/O started against them. If wbc->sync_mode is
2264  * WB_SYNC_ALL then we were called for data integrity and we must wait for
2265  * existing IO to complete.
2266  *
2267  * To avoid livelocks (when other process dirties new pages), we first tag
2268  * pages which should be written back with TOWRITE tag and only then start
2269  * writing them. For data-integrity sync we have to be careful so that we do
2270  * not miss some pages (e.g., because some other process has cleared TOWRITE
2271  * tag we set). The rule we follow is that TOWRITE tag can be cleared only
2272  * by the process clearing the DIRTY tag (and submitting the page for IO).
2273  *
2274  * To avoid deadlocks between range_cyclic writeback and callers that hold
2275  * pages in PageWriteback to aggregate IO until write_cache_pages() returns,
2276  * we do not loop back to the start of the file. Doing so causes a page
2277  * lock/page writeback access order inversion - we should only ever lock
2278  * multiple pages in ascending page->index order, and looping back to the
2279  * start
2280  * of the file violates that rule and causes deadlocks.
2281  *
2282  * Return: %0 on success, negative error code otherwise
2283  */
2284 int write_cache_pages(struct address_space *mapping,
2285                      struct writeback_control *wbc, writepage_t writepage,
2286                      void *data)
2287 {
2288     int ret = 0;

```

```

2286     int done = 0;
2287     int error;
2288     struct pagevec pvec;
2289     int nr_pages;
2290     pgoff_t index;
2291     pgoff_t end;           /* Inclusive */
2292     pgoff_t done_index;
2293     int range_whole = 0;
2294     xa_mark_t tag;
2295
2296     pagevec_init(&pvec);
2297     if (wbc->range_cyclic) {
2298         index = mapping->writeback_index; /* prev offset */
2299         end = -1;
2300     } else {
2301         index = wbc->range_start >> PAGE_SHIFT;
2302         end = wbc->range_end >> PAGE_SHIFT;
2303         if (wbc->range_start == 0 && wbc->range_end == LLONG_MAX)
2304             range_whole = 1;
2305     }
2306     if (wbc->sync_mode == WB_SYNC_ALL || wbc->tagged_writepages) {
2307         tag_pages_for_writeback(mapping, index, end);
2308         tag = PAGECACHE_TAG_TOWRITE;
2309     } else {
2310         tag = PAGECACHE_TAG_DIRTY;
2311     }
2312     done_index = index;

```

Listing 10-25: mm/page-writeback.c: write_cache_pages() initialisation

Note that the function is passed a void * field in data which contains user-defined data that file systems which perform custom invocations of this function can use to thread a value through to the writepage function. In generic_writepages() the struct address_space mapping object is placed in this field.

The function operations on batches of pages utilising the struct pagevec type. This is a legacy version of the struct folio_batch type (see Section 11.7 for more details on this type), but behaves identically other than storing struct page objects rather than struct folio ones. We initialise this via pagevec_init(), and it contains up to PAGEVEC_SIZE (hardcoded to 15) pages.

We then set the start and end page indexes for the range we will explore in index and end respectively (note that end is an inclusive bound).

If the operation is cyclic, we use struct address_space->writeback_index to retrieve the last successfully written to page index, and set the unsigned long end object to -1, which is converted to the largest value the type can represent, indicating we should examine the whole range.

In non-cyclic mode, we use the range_start and range_end fields of the struct writeback_control argument to determine which page indexes to span,

setting `range_whole` to true if this implies a scan of the whole of the page cache object.

Finally, we determine how to identify dirty pages for writeback—if `struct writeback_control->tagged_writepages` is set or if a full synchronisation is to be performed (which will be slow as we wait on each writeout), we immediately update all currently dirty folios setting the `PAGECACHE_TAG_TOWRITE` xarray tag via `tag_pages_for_writeback()`, and set tag to whichever tag we should look up.

As discussed above, we do this to avoid live locks where a process dirties pages faster than we can write them back.

Next we examine the core loop of this function in Listing 10-26.

```

2313     while (!done && (index <= end)) {
2314         int i;
2315
2316         nr_pages = pagevec_lookup_range_tag(&pvec, mapping, &index,
2317         end,
2318                                     tag);
2319         if (nr_pages == 0)
2320             break;
2321
2322         for (i = 0; i < nr_pages; i++) {
2323             struct page *page = pvec.pages[i];
2324
2325             done_index = page->index;
2326
2327             lock_page(page);
2328
2329             /*
2330              * Page truncated or invalidated. We can freely skip
2331              * it
2332              * then, even for data integrity operations: the page
2333              * has disappeared concurrently, so there could be no
2334              * real expectation of this data integrity operation
2335              * even if there is now a new, dirty page at the same
2336              * pagecache address.
2337              */
2338             if (unlikely(page->mapping != mapping)) {
2339                 continue_unlock:
2340                     unlock_page(page);
2341                     continue;
2342             }
2343
2344             if (!PageDirty(page)) {
2345                 /* someone wrote it for us */
2346                 goto continue_unlock;
2347             }

```

```

2347         if (PageWriteback(page)) {
2348             if (wbc->sync_mode != WB_SYNC_NONE)
2349                 wait_on_page_writeback(page);
2350             else
2351                 goto continue_unlock;
2352         }
2353
2354         BUG_ON(PageWriteback(page));
2355         if (!clear_page_dirty_for_io(page))
2356             goto continue_unlock;
2357         :
2358
2359         error = (*writepage)(page, wbc, data);
2360         if (unlikely(error)) {
2361             /*
2362              * Handle errors according to the type of
2363              * writeback. There's no need to continue for
2364              * background writeback. Just push done_index
2365              * past this page so media errors won't choke
2366              * writeout for the entire file. For integrity
2367              * writeback, we must process the entire dirty
2368              * set regardless of errors because the fs may
2369              * still have state to clear for each page. In
2370              * that case we continue processing and return
2371              * the first error.
2372              */
2373             if (error == AOP_WRITEPAGE_ACTIVATE) {
2374                 unlock_page(page);
2375                 error = 0;
2376             } else if (wbc->sync_mode != WB_SYNC_ALL) {
2377                 ret = error;
2378                 done_index = page->index + 1;
2379                 done = 1;
2380                 break;
2381             }
2382             if (!ret)
2383                 ret = error;
2384         }
2385
2386         /*
2387          * We stop writing back only if we are not doing
2388          * integrity sync. In case of integrity sync we have
2389
2390          to
2391
2392          * keep going until we have written all the pages
2393          * we tagged for writeback prior to entering this loop
2394
2395          .
2396
2397          */
2398         if (--wbc->nr_to_write <= 0 &&

```

```

2393             wbc->sync_mode == WB_SYNC_NONE) {
2394                 done = 1;
2395                 break;
2396             }
2397         }
2398         pagevec_release(&pvec);
2399         cond_resched();
2400     }

```

Listing 10-26: mm/page-writeback.c: write_cache_pages() main loop

This starts by looking up page cache entries in the specified range via `pagevec_lookup_range_tag()` which wraps a call to `find_get_pages_range_tag()`, which retrieves pages with the specified tag via `find_get_entry()`, shown in Listing 9-41 and Section 9.5.1, setting `index` to the lower bound at which the next search should commence.

If no pages were retrieved we exit the loop, otherwise we iterate through each retrieved page, locking them as they go. We perform a typical check as, prior to the lock, the page may have been truncated beneath us (and the struct `page` we are examining may have been used for another purpose) by ensuring struct `page->mapping` is equal to the expected page cache mapping. If not, we unlock the page and move on to the next page.

Note that, since we've updated `index` field, we track the last completed page index in the `done_index` field.

We are about to mark the folio clean for writeback, so if it is currently marked clean, the write has already been performed by somebody else and we can simply skip to the next page.

Equally, if the folio is marked with the writeback flag, which is set by the block I/O logic once the writepage has been submitted, then a parallel writeback is in action.

It is however possible for folios to be re-dirtied midway through a writeback operation, so we must handle this scenario correctly.

If this is a synchronous operation we wait for this to complete via `wait_on_page_writeback()` (see Section 9.11 for more details on waiting on folios) since we need to ensure this operation has successfully completed. Otherwise, we simply move on to the next page, and rely on the newly dirtied page being written back once expired.

After a sanity check to ensure that the page no longer has the writeback flag set, we move on to one of the most critical parts of the operation. In order to maintain dirty tracking as described in Section 10.1, we must mark the page mappings of all mapped pages being written read-only once writeback begins before clearing the folio's dirty flag.

This way, if a file write or file-backed page fault occurs in the meantime, we redirty the folio and correctly mark it for writeback at a late point. This is achieved by `clear_page_dirty_for_io()`, which wraps `folio_clear_dirty_for_io()`, which we will explore shortly in detail in Listing 10-30.

NOTE

When a page fault occurs, most modern file systems and block devices will not wait for a folio undergoing writeback to complete. Rather, they rely on the fact that any

further writes to the folio will cause it to be marked dirty and thus written back again at a later date. This does mean that an unfortunate race can result in torn writes, i.e. observable data corruption. We discussed this in Section 10.12 and Listing 10-7 in reference to a page fault, but this is equally applicable to a write() operation.

Note that on page fault `folio_wait_stable()` (see Listing 10-7) is invoked to check to see whether the file system or block device requires waiting on writeback in `filemap_page_mkwrite()` and also via `grab_cache_page_write_begin()` on a `write()` call (see Listing 10-5).

This renders the correct ‘cleaning’ of the folios to be written to in `folio_clear_dirty_for_io()` absolutely critical. Page faults and `write()` operations will be blocked from interfering with this operation into it is complete as this is protected by the folio lock.

If we are unable to mark the folio clean for I/O (for instance if it was already marked dirty), then we simply move on to the next folio.

Otherwise, we invoke the passed-in `writepage` function to perform the underlying page writeback. This of course varies by file system, and operates at the block level. A commonly used function for this operation is `__mpage_writepage()`, however discussion of this is out of scope for the book. We pass the struct `writeback_control` state to this function so it knows to handle the specified synchronisation requirements.

Note that importantly, the page passed to `writepage()` is locked when passed, but will be unlocked once the operation is complete, except in the specific case of an error being returned of value `AOP_WRITEPAGE_ACTIVATE`. This indicates that the page writeback has completed, and this is not in fact an error but rather indicates the page is locked and must be unlocked by us.

Regardless of the implementation of the `writepage()` callback, the function will ultimately invoke `folio_start_writeback()` when the writeback starts (which in turn wraps `__folio_start_writeback()`) which sets the folio writeback flag if not already set, and marks the relevant entry in the page cache object’s xarray with the tag. See Listing 10-28 for details.

Equally, when the I/O operation is complete, `folio_end_writeback()` is invoked, which in turn invokes `__folio_end_writeback()` (see Listing 10-29) which clears the writeback flag from the folio and the tag from its xarray.

Next we have some rather fragile error handling. As the comment suggests, in cases other where where we must synchronise the write (the ‘integrity writeback’ case), we can simply abort the writeback and return the error.

However in the case of a synchronised write we simply record the first error and carry on.

Finally, if we have reached or exceeded the specified struct `writeback_control->nrr_to_write` limit and are not synchronising, we are done and can abort the operation.

Once the loop is complete, we handle some cyclic range edge cases as shown in Listing 10-27.

```

2402     /*
2403     * If we hit the last page and there is more work to be done: wrap
```

```

2404     * back the index back to the start of the file for the next
2405     * time we are called.
2406     */
2407     if (wbc->range_cyclic && !done)
2408         done_index = 0;
2409     if (wbc->range_cyclic || (range_whole && wbc->nr_to_write > 0))
2410         mapping->writeback_index = done_index;
2411
2412     return ret;
2413 }

```

Listing 10-27: mm/page-writeback.c: write_cache_pages() cyclic edge cases

If the range is cyclic and we were *not* able to process all dirty pages (for instance if struct writeback_control->nr_pages was set to scan the whole file and an error did not occur), we reset the index to start at on the next cycle to the beginning of the page cache entry in struct address_space->writeback_index.

If the range was cyclic and struct writeback_control->nr_pages was specified and that number of pages were processed, we set struct address_space->writeback_index to the next index at which we should resume writeback.

Finally we have the edge case of a non-cyclic writeback, but where the entire range of the page cache entry is required to be written back, but pages remain to be written (or an unlimited number were specified to be written), in which case we also set this offset in order that a cyclical write will pick up where it left off.

By doing this, we avoid situations where folios at the beginning of a page cache entry keep on being dirtied and written back without writing back later folios.

We examine __folio_start_writeback() in Listing 10-28, eliding statistical updates, out of scope cgroup logic and details that distract from the core of this function.

```

2955 bool __folio_start_writeback(struct folio *folio, bool keep_write)
2956 {
2957     long nr = folio_nr_pages(folio);
2958     struct address_space *mapping = folio_mapping(folio);
2959     bool ret;
2960     int access_ret;
2961     :
2962
2964     XA_STATE(xas, &mapping->i_pages, folio_index(folio));
2965     struct inode *inode = mapping->host;
2966     struct backing_dev_info *bdi = inode_to_bdi(inode);
2967     unsigned long flags;
2968
2969     xas_lock_irqsave(&xas, flags);
2970     xas_load(&xas);
2971     ret = folio_test_set_writeback(folio);

```



```

2972         if (!ret) {
2973             bool on_wblist;
2974
2975             on_wblist = mapping_tagged(mapping,
2976                                     PAGECACHE_TAG_WRITEBACK);
2977
2978             xas_set_mark(&xas, PAGECACHE_TAG_WRITEBACK);
2979             :
2992             if (mapping->host && !on_wblist)
2993                 sb_mark_inode_writeback(mapping->host);
2994         }
2995         if (!folio_test_dirty(folio))
2996             xas_clear_mark(&xas, PAGECACHE_TAG_DIRTY);
2997         if (!keep_write)
2998             xas_clear_mark(&xas, PAGECACHE_TAG_TOWRITE);
2999         xas_unlock_irqrestore(&xas, flags);
3000         :
3015     return ret;
3016 }

```

Listing 10-28: mm/page-writeback.c: `__folio_start_writeback()`

This sets the folio's writeback flag if not already set and marks it tagged as `PAGECACHE_TAG_WRITEBACK` in the relevant xarray entry, clearing any existing flag to indicate that I/O has been submitted.

Except in an edge case specific to the ext4 file system the `keep_write` flag will be false, therefore any existing `PAGECACHE_TAG_TOWRITE` will have this mark cleared also (see discussion around `write_cache_pages()` in Listing 10-25 for details on how this is utilised).

It also invokes `sb_mark_inode_writeback()` for the *first* writeback folio in the mapping, which places the inode on the super block's writeback list, `struct super_block->i_wb_list`. This will be reversed when the writeback operation is complete.

When the writeback has completed, `folio_end_writeback()` will be invoked (most likely via `page_endio()`, see Listing 9-58 in the page cache chapter), which performs some reclaim-specific logic out of scope for this chapter, before invoking `__folio_end_writeback()` shown in Listing 10-29 before waking anything waiting on the writeback via `folio_wake()` as shown in Listing 9-145 in Section 9.11. Again we elide irrelevant detail to focus on the core of what this does.

```

2910 bool __folio_end_writeback(struct folio *folio)
2911 {
2912     long nr = folio_nr_pages(folio);
2913     struct address_space *mapping = folio_mapping(folio);
2914     bool ret;
2915     :
2918     struct inode *inode = mapping->host;
2919     struct backing_dev_info *bdi = inode_to_bdi(inode);

```

```

2920         unsigned long flags;
2921
2922         xa_lock_irqsave(&mapping->i_pages, flags);
2923         ret = folio_test_clear_writeback(folio);
2924         if (ret) {
2925             __xa_clear_mark(&mapping->i_pages, folio_index(folio),
2926                             PAGECACHE_TAG_WRITEBACK);
2927             :
2928
2929             if (!mapping_tagged(mapping,
2930                                 PAGECACHE_TAG_WRITEBACK))
2931                 wb_inode_writeback_end(wb);
2932             :
2933         }
2934
2935         if (mapping->host && !mapping_tagged(mapping,
2936                                             PAGECACHE_TAG_WRITEBACK))
2937             sb_clear_inode_writeback(mapping->host);
2938
2939         xa_unlock_irqrestore(&mapping->i_pages, flags);
2940         :
2941         return ret;
2942     }
2943 }

```

Listing 10-29: mm/page-writeback.c: `__folio_end_writeback()`

This is more or less the inverse of `__folio_start_writeback()`, clearing the folio's writeback flag and removing xarray tags. Two important points to note—if this was the final folio to have its tag cleared, then two extra tasks are performed—a worker thread is scheduled to update the bandwidth estimate in the struct `bdi_writeback` object via `wb_inode_writeback_end()` (see section 10.14 for more details on how these values are used) and `sb_clear_inode_writeback()` removes the containing inode from the super block's writeback list block's writeback list, struct `super_block->i_wb_list`.

Finally, we examine the key `folio_clear_dirty_for_io()` function in Listing 10-30, eliding less critical logic around update of statistics and debug checks.

```

2812 /*
2813  * Clear a folio's dirty flag, while caring for dirty memory accounting.
2814  * Returns true if the folio was previously dirty.
2815  *
2816  * This is for preparing to put the folio under writeout. We leave
2817  * the folio tagged as dirty in the xarray so that a concurrent
2818  * write-for-sync can discover it via a PAGECACHE_TAG_DIRTY walk.
2819  * The ->writepage implementation will run either folio_start_writeback()
2820  * or folio_mark_dirty(), at which stage we bring the folio's dirty flag
2821  * and xarray dirty tag back into sync.
2822  *
2823  * This incoherency between the folio's dirty flag and xarray tag is
2824  * unfortunate, but it only exists while the folio is locked.

```

```

2825  */
2826  bool folio_clear_dirty_for_io(struct folio *folio)
2827  {
2828      struct address_space *mapping = folio_mapping(folio);
2829      bool ret = false;
2830      :
2831      if (mapping && mapping_can_writeback(mapping)) {
2832          struct inode *inode = mapping->host;
2833          struct bdi_writeback *wb;
2834          :
2835          /*
2836           * Yes, Virginia, this is indeed insane.
2837           *
2838           * We use this sequence to make sure that
2839           * (a) we account for dirty stats properly
2840           * (b) we tell the low-level filesystem to
2841           *     mark the whole folio dirty if it was
2842           *     dirty in a pagetable. Only to then
2843           * (c) clean the folio again and return 1 to
2844           *     cause the writeback.
2845           *
2846           * This way we avoid all nasty races with the
2847           * dirty bit in multiple places and clearing
2848           * them concurrently from different threads.
2849           *
2850           * Note! Normally the "folio_mark_dirty(folio)"
2851           * has no effect on the actual dirty bit - since
2852           * that will already usually be set. But we
2853           * need the side effects, and it can help us
2854           * avoid races.
2855           *
2856           * We basically use the folio "master dirty bit"
2857           * as a serialization point for all the different
2858           * threads doing their things.
2859           */
2860          if (folio_mkclean(folio))
2861              folio_mark_dirty(folio);
2862          /*
2863           * We carefully synchronise fault handlers against
2864           * installing a dirty pte and marking the folio dirty
2865           * at this point. We do this by having them hold the
2866           * page lock while dirtying the folio, and folios are
2867           * always locked coming in here, so we get the desired
2868           * exclusion.
2869           */
2870          :
2871          if (folio_test_clear_dirty(folio)) {

```

```

:
2879             ret = true;
2880         }
:
2882         return ret;
2883     }
2884     return folio_test_clear_dirty(folio);
2885 }

```

Listing 10-30: mm/page-writeback.c: folio_clear_dirty_for_io()

This starts by checking whether the struct `address_space` page cache object is capable of writing back at all via `mapping_can_writeback()`, which determines this from the underlying BDI.

If not then the operation is simple, we use `folio_test_clear_dirty()` to check if the folio was marked dirty, if so we simply clear the flag and return indicating that we have done this.

As can be determined from the colourful comment, the writeback case is rather more tricky.

We perform the most critical task of this function, which is to use the reverse mapping to mark all page mappings to the folio read-only via `folio_mkclean()` which returns the number of page table mappings that were marked read-only. See Chapter 7 on the reverse mapping for details as to how this is achieved.

At this point, any writes that are performed to memory mappings of this folio will cause a page fault which will re-dirty the folio. This cannot happen while the `folio_clear_dirty_for_io()` function is in-flight, as the folio lock is held at this point, and page faults also acquire this lock, so we serialise on it (see Section 10.12 for more on dirty tracking on page faults).

Writes via `write()` are safe as these always correctly mark the written-to folios dirty (see Section 10.12 for details on dirty tracking over `write()` operations).

This provides a guarantee that any further writes to the folio will cause those writes to be correctly accounted as dirtying the folio and result in later writeback. This is *absolutely critical* to the correct functioning of writeback as it eliminates the possibility of a race on writeback causing writes to be ‘missed’.

As the comment suggest however there is some nuance here in ensuring that all state is correctly serialised and statistics are correctly updated. The `folio_mark_dirty()` function (see Section 10.12 and Listing 10-8) is called *after* the page table mapping cleaning performed by `folio_mkclean()` marks at least one mapping clean.

This serialises any possible race condition between concurrent clearing of the folio dirty flag from multiple threads and ensuring that the folio is in a correct dirty state to be processed by writeback.

Once this is done, we can simply invoke `folio_test_clear_dirty()` to clear the folio dirty flag and begin writeback, if another thread has not already cleared the folio at this point.

As the comment states, the fact we hold a folio lock on page fault prevents a race between a dirty page table entry being installed and flipping the folio to dirty.

10.11 File System and Background Writeback

When folios are dirtied within the file system, we must track which folios and inodes are dirty in order that we can later write them back. This is achieved by storing dirty folios in struct `bdi_writeback` and the struct `address_space->i_pages` xarray as described in Section 10.12.

This is how most writeback within the kernel is performed, both periodic background writeback and file system synchronisation via `sync()` and `syncfs()` system calls as described in Section 10.6.

The exception to this rule is file synchronisation via e.g. `fsync()` which directly invokes per-file writeback. We explore this in Section 10.13.

We examine how the process of file system writeback occurs in Figure 10-6.

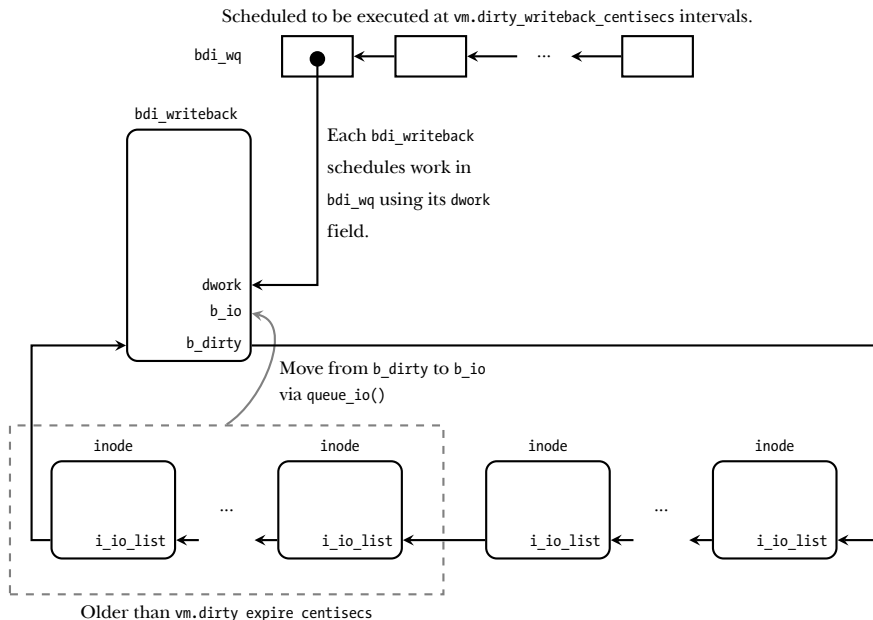


Figure 10-6: File System Writeback

The process of file system level writeback is accomplished via a *work queue*, which is a kernel API which abstracts the ability to perform a series of tasks in a kernel thread.

All struct `bdi_writeback` objects share the same struct `workqueue_struct` work queue object, `bdi_wq`.

NOTE

Work queues are a generic means by which the kernel can execute work asynchronously and in the context of a process, using a kernel-managed thread pool.

The `bdi_wq` work queue is initialised in `default_bdi_init()`, shown in Listing 10-31.

```

234 static int __init default_bdi_init(void)
235 {
236     bdi_wq = alloc_workqueue("writeback", WQ_MEM_RECLAIM | WQ_UNBOUND |
237                               WQ_SYSFS, 0);
238     if (!bdi_wq)
239         return -ENOMEM;
240     return 0;
241 }

```

Listing 10-31: `mm/backing-dev.c: default_bdi_init()`

This function allocates a work queue via `alloc_workqueue()`, notably setting `WQ_UNBOUND` to indicate that the work can be scheduled on any core, and `WQ_MEM_RECLAIM` to indicate that the work queue may be used during the memory reclaim process and thus might operate under very low memory conditions.

Before any folios are dirtied, this queue is empty. However, when a folio is dirtied in `__mark_inode_dirty()` (see Listing 10-2 in Section 10.1), we check to see whether the associated struct `bdi_writeback` has any dirty folios attached to it. If not, then we queue the struct `delayed_work` task in struct `bdi_writeback->dwork` on `bdi_wq` via `wb_wakeup_delayed()`, shown in Listing 10-32.

We keep track of whether a struct `bdi_writeback` has associated dirty folios in `wb_io_lists_populated()`, which checks this condition in `wb_has_dirty_io()`, which checks whether the struct `bdi_writeback->status` field has the `WB_has_dirty_io` bit set. If this bit was not previously present, it is immediately set at this point, and is cleared in `wb_io_lists_depopulated()` at the point no dirty folios remain.

As a result, no writeback work is actually performed when a block device has no dirtied folios, in other words the work queue task is only ‘woken up’ when there is work to do.

```

244 /*
245  * This function is used when the first inode for this wb is marked dirty. It
246  * wakes-up the corresponding bdi thread which should then take care of the
247  * periodic background write-out of dirty inodes. Since the write-out would
248  * start only 'dirty_writeback_interval' centisecs from now anyway, we just
249  * set up a timer which wakes the bdi thread up later.
250  *
251  * Note, we wouldn't bother setting up the timer, but this function is on the
252  * fast-path (used by '__mark_inode_dirty()'), so we save few context switches
253  * by delaying the wake-up.
254  *
255  * We have to be careful not to postpone flush work if it is scheduled for
256  * earlier. Thus we use queue_delayed_work().
257  */

```

```

258 void wb_wakeup_delayed(struct bdi_writeback *wb)
259 {
260     unsigned long timeout;
261
262     timeout = msecs_to_jiffies(dirty_writeback_interval * 10);
263     spin_lock_irq(&wb->work_lock);
264     if (test_bit(WB_registered, &wb->state))
265         queue_delayed_work(bdi_wq, &wb->dwork, timeout);
266     spin_unlock_irq(&wb->work_lock);
267 }

```

Listing 10-32: mm/backing-dev.c: `wb_wakeup_delayed()`

This checks to see whether the object has been correctly initialised and registered via the `WB_registered` flag and then queues the work via the work queue utility function `queue_delayed_work()`.

The time interval is based on the `vm.dirty_writeback_centisecs` tunable which is known to the kernel via the `dirty_writeback_interval` global value.

When each `struct bdi_writeback` object is initialised in `wb_init()`, it has its `dwork` field set to invoke `wb_workfn()` which is the core function which performs file system writeback.

We will examine this in Listing 10-44, however we will first examine in a little more detail how the kernel interacts with these work queue tasks. One thing to note at this stage is that each time a queued delayed work queue item is complete, i.e. invoking `wb_workfn()` in this case, it is completely removed from the queue, and thus we must reschedule, which this function does.

The periodic background writeback is only one scenario in which writeback needs to be performed, another is for instance when file synchronisation is performed at which point the writeback needs to be performed *immediately*.

When synchronisation wakes up flusher threads, or if writeback work other than periodic writeback still needs to be performed after writeback has occurred, this is ultimately performed by `wb_wakeup()`, which we examine in Listing 10-33.

```

135 static void wb_wakeup(struct bdi_writeback *wb)
136 {
137     spin_lock_irq(&wb->work_lock);
138     if (test_bit(WB_registered, &wb->state))
139         mod_delayed_work(bdi_wq, &wb->dwork, 0);
140     spin_unlock_irq(&wb->work_lock);
141 }

```

Listing 10-33: fs/fs-writeback.c: `wb_wakeup()`

This function uses `mod_delayed_work()` to set the delay to zero, effectively causing the task to be executed immediately, i.e. invoking `wb_workfn()` for the specified `struct bdi_writeback` immediately.

When file system super blocks need to be explicitly synchronised, this is done by adding struct `wb_writeback_work` objects (see Listing 10-38) to the struct `bdi_writeback->work_list` and then immediately waking up the struct `bdi_writeback`'s writeback work queue thread via `wb_queue_work()` which we examine in Listing 10-34 (eliding out of scope tracing logic).

```

159 static void wb_queue_work(struct bdi_writeback *wb,
160                          struct wb_writeback_work *work)
161 {
162     :
163     if (work->done)
164         atomic_inc(&work->done->cnt);
165
166     spin_lock_irq(&wb->work_lock);
167
168     if (test_bit(WB_registered, &wb->state)) {
169         list_add_tail(&work->list, &wb->work_list);
170         mod_delayed_work(bdi_wq, &wb->dwork, 0);
171     } else
172         finish_writeback_work(wb, work);
173
174     spin_unlock_irq(&wb->work_lock);
175 }
176 
```

Listing 10-34: fs/fs-writeback.c: `wb_queue_work()`

This starts by incrementing the atomic counter of the struct `wb_completion` field stored in struct `wb_writeback_work->done` if this field is set. This field is used by the caller to track when a specified work task is complete.

If the struct `bdi_writeback` object is registered, i.e. not removed or in the process of removal, then the work task is simply added to the struct `bdi_writeback->work_list` and the same `mod_delayed_work()` trick as used by `wb_wakeup()` is used to wake up the associated writeback wait queue task.

if the struct `bdi_writeback` is not registered, then `finish_writeback_work()` is invoked to perform clean up tasks.

Note that when a BDI device or related cgroup structure is unregistered, the kernel invokes `wb_shutdown()` which also invokes immediate execution of writeback, however for space we will disregard this.

Before diving into the implementation of file system and background writeback, we examine how `wb_wakeup()`, `wb_wakeup_delayed()` and `wb_queue_work()` are invoked in Figure 10-7 (eliding out of scope dirty time, cgroup and laptop mode handling).

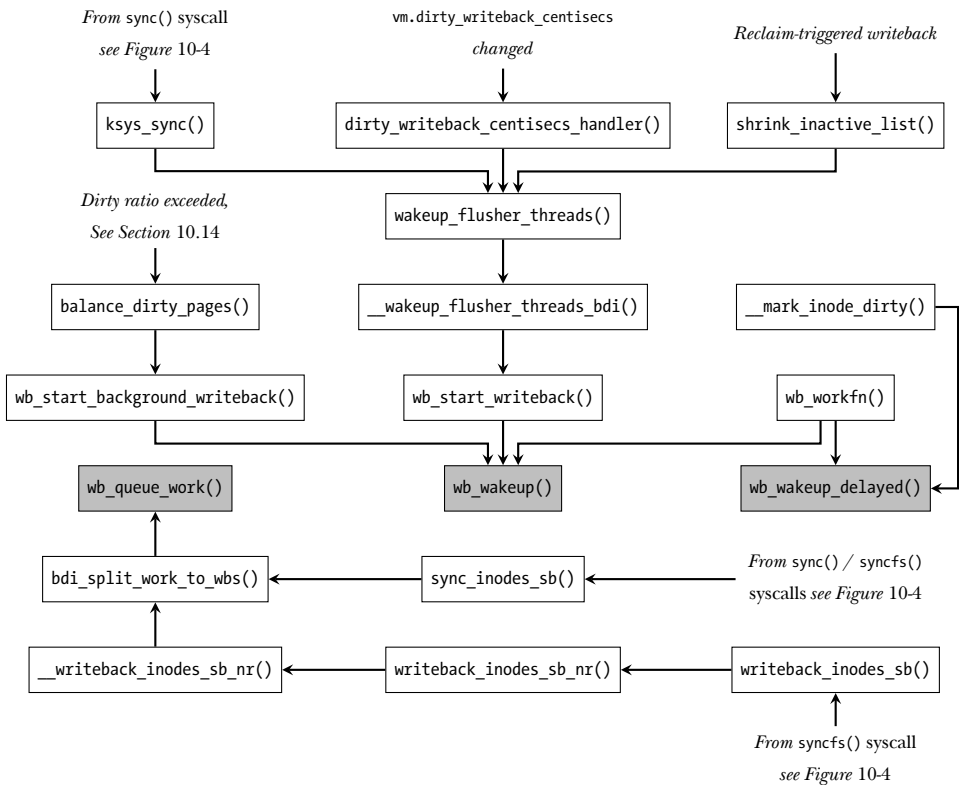


Figure 10-7: File System Writeback Wakeup

This diagram shows all of the means by which struct `bdi_writeback` writeback work queue threads can be woken up to perform the actual writeback, which is another way of saying all of the means *file system*-level writeback can be performed.

We have explored the various means by which file system synchronisation can be performed in Section 10.6, and here we see how the file system-level synchronisation system calls, `sync()` and `syncfs()` ultimately wake up the file system via `wakeup_flusher_threads()` which we examine below in Listing 10-35.

When synchronisation is started, the request to perform writeback of inodes is performed asynchronously by `writeback_inodes_sb()` and synchronously by `sync_inodes_sb()` which we examine in Listings 10-40 and 10-39 below, respectively.

The flusher threads are also awoken by reclaim in `shrink_inactive_list()`, which we explore in detail in the dedicated reclaim chapter. When `vm.dirty_writeback_centisecs` is altered, this also causes flusher threads to be awoken immediately via `dirty_writeback_centisecs_handler()`.

When the `vm.dirty_background_ratio` or `vm.dirty_background_bytes` limits are breached when we dirty a folio, we invoke background writeback, if the `vm.dirty_ratio` or `vm.dirty_bytes` limits are breached then the process doing this waits on the operation to complete. This is all achieved in

`balance_dirty_pages()`, which we discuss in detail in Section 10.14, invoking the background writeback in `wb_start_background_writeback()` which simply wraps `wb_wakeup()`.

We have also previously explored how the first dirty inode associated with a struct `bdi_writeback` can trigger the standard background flushing process (invoked at `vm.dirty_writeback_centisecs` intervals) in Listing 10-2 and Section 10.1 which we observe here also invoking `wb_wakeup_delayed()`.

We note that the actual worker function executed when the work queue is awoken, `wb_workfn()`, must reschedule itself via `wb_wakeup_delayed()`, though also note that if it has immediate work to do, it reinvokes itself immediately via `wb_wakeup()`. We will examine this function in detail in Listing 10-44 below.

We examine the function which wakes up the flusher threads for each struct `bdi_writeback`, `wakeup_flusher_threads()` in Listing 10-35.

```

2266 /*
2267  * Wakeup the flusher threads to start writeback of all currently dirty pages
2268  */
2269 void wakeup_flusher_threads(enum wb_reason reason)
2270 {
2271     struct backing_dev_info *bdi;
2272
2273     /*
2274      * If we are expecting writeback progress we must submit plugged IO.
2275      */
2276     blk_flush_plug(current->plug, true);
2277
2278     rcu_read_lock();
2279     list_for_each_entry_rcu(bdi, &bdi_list, bdi_list)
2280         __wakeup_flusher_threads_bdi(bdi, reason);
2281     rcu_read_unlock();
2282 }
```

Listing 10-35: `fs/fs-writeback.c: wakeup_flusher_threads()`

We disregard the `blk_flush_plug()` function here as this is an out of scope scaling optimisation for the block layer, as well as the *RCU*-specific optimised locking logic as equally out of scope.

The key thing to note here is that we iterate through each block device in the global variable `bdi_list` which contains these, and invokes `__wakeup_flusher_threads_bdi()` on each of them, shown in Listing 10-36.

```

2246 static void __wakeup_flusher_threads_bdi(struct backing_dev_info *bdi,
2247                                         enum wb_reason reason)
2248 {
2249     struct bdi_writeback *wb;
2250
2251     if (!bdi_has_dirty_io(bdi))
2252         return;
```

```

2253
2254         list_for_each_entry_rcu(wb, &bdi->wb_list, bdi_node)
2255             wb_start_writeback(wb, reason);
2256     }

```

Listing 10-36: fs/fs-writeback.c: __wakeup_flusher_threads_bdi()

This function starts by invoking `bdi_has_dirty_io()` to determine if any associated struct `bdi_writeback` object has outstanding dirty I/O to perform, as if not waking the threads is not useful. It then iterates through each of the BDI's struct `bdi_writeback` objects, stored as a list in struct `backing_dev_info->wb_list` and invokes `wb_start_writeback()` on each, which we examine in Listing 10-37.

```

1188 static void wb_start_writeback(struct bdi_writeback *wb, enum wb_reason reason
1189                               )
1189 {
1190     if (!wb_has_dirty_io(wb))
1191         return;
1192
1193     /*
1194      * All callers of this function want to start writeback of all
1195      * dirty pages. Places like vmscan can call this at a very
1196      * high frequency, causing pointless allocations of tons of
1197      * work items and keeping the flusher threads busy retrieving
1198      * that work. Ensure that we only allow one of them pending and
1199      * inflight at the time.
1200      */
1201     if (test_bit(WB_start_all, &wb->state) ||
1202         test_and_set_bit(WB_start_all, &wb->state))
1203         return;
1204
1205     wb->start_all_reason = reason;
1206     wb_wakeup(wb);
1207 }

```

Listing 10-37: fs/fs-writeback.c: wb_start_writeback()

If the struct `bdi_writeback` object does not have dirty I/O associated with it (i.e. entries on one of its `b_dirty`, `b_io` or `b_more_io` lists), then there is nothing to do and we abort. This is checked via `wb_has_dirty_io()` which checks for the `WB_has_dirty_io` flag, conditionally set by `wb_io_lists_populated()` and conditionally cleared by `wb_io_lists_depopulated()`, both invoked when entries may be added or removed from lists respectively.

As the comment suggests, we maintain a specific flag for the case of wanting to writeback *all* dirty folios to prevent duplicate invocations for efficiency, via the `WB_start_all` flag which we both set and clear here.

Finally we invoke `wb_wakeup()` as described in Listing 10-33.

In order to indicate to writeback that we have work to perform other than periodic writeback of folios older than `vm.dirty_expire_centisecs`,

we maintain the struct `bdi_writeback->work_list` which maintains a list of struct `wb_writeback_work` objects, a type which is essentially a subset of the struct `writeback_control`. We examine this in Listing 10-38.

```

39  /*
40   * Passed into wb_writeback(), essentially a subset of writeback_control
41   */
42  struct wb_writeback_work {
43      long nr_pages;
44      struct super_block *sb;
45      enum writeback_sync_modes sync_mode;
46      unsigned int tagged_writepages:1;
47      unsigned int for_kupdate:1;
48      unsigned int range_cyclic:1;
49      unsigned int for_background:1;
50      unsigned int for_sync:1;      /* sync(2) WB_SYNC_ALL writeback */
51      unsigned int auto_free:1;    /* free on completion */
52      enum wb_reason reason;      /* why was writeback initiated? */
53
54      struct list_head list;      /* pending work list */
55      struct wb_completion *done; /* set if the caller waits */
56  };

```

Listing 10-38: fs/fs-writeback.c: struct `wb_writeback_work`

This is a subset of struct `writeback_control` (see Listing 10-24) but includes some additional fields:

- nr_pages** This field indicates the number of pages to write, similar to struct `writeback_control->nr_to_write`, though accounted separately.
- sb** The struct `super_block` which is undergoing writeback, or if `NULL`, all superblocks.
- list** A struct `list_head` node used to place struct `wb_writeback_work` objects in struct `bdi_writeback->work_list`.
- done** A struct `wb_completion` object used to notify waiters that the work has been completed. item
- auto_free** A flag indicating whether the work object should be freed automatically when the writeback is complete by `finish_writeback_work()`. This is needed because `bdi_split_work_to_wbs()` might allocate a struct `wb_writeback_work` object using the slab allocator, which needs to be freed. Other invocations use stack-allocated objects so do not need this so will not set this flag.

Importantly, while the actual underlying writeback as described in Section 10.10 is performed in terms of struct `writeback_control` objects, file system writeback, whether specific writeback invoked by synchronisation functions, or periodic background writeback is performed using struct `wb_writeback_work` objects.

Let's examine how struct `bdi_writeback->work_list` gets populated. This is required by `sync()` and `syncfs()` system calls in order to specify what should be written back immediately rather than awaiting periodic writeback, performed in two stages—synchronous writeback performed by `sync_inodes_sb()` and asynchronous writeback performed by `writeback_inodes_sb()`.

Let's start by examining `sync_inodes_sb()` in Listing 10-39.

```

2660 /**
2661  * sync_inodes_sb          -          sync sb inode pages
2662  * @sb: the superblock
2663  *
2664  * This function writes and waits on any dirty inode belonging to this
2665  * super_block.
2666  */
2667 void sync_inodes_sb(struct super_block *sb)
2668 {
2669     struct backing_dev_info *bdi = sb->s_bdi;
2670     DEFINE_WB_COMPLETION(done, bdi);
2671     struct wb_writeback_work work = {
2672         .sb                = sb,
2673         .sync_mode         = WB_SYNC_ALL,
2674         .nr_pages          = LONG_MAX,
2675         .range_cyclic      = 0,
2676         .done              = &done,
2677         .reason            = WB_REASON_SYNC,
2678         .for_sync          = 1,
2679     };
2680
2681     /*
2682      * Can't skip on !bdi_has_dirty() because we should wait for !dirty
2683      * inodes under writeback and I_DIRTY_TIME inodes ignored by
2684      * bdi_has_dirty() need to be written out too.
2685      */
2686     if (bdi == &noop_backing_dev_info)
2687         return;
2688     WARN_ON(!rwsem_is_locked(&sb->s_umount));
2689
2690     /* protect against inode wb switch, see inode_switch_wbs_work_fn() */
2691     bdi_down_write_wb_switch_rwsem(bdi);
2692     bdi_split_work_to_wbs(bdi, &work, false);
2693     wb_wait_for_completion(&done);
2694     bdi_up_write_wb_switch_rwsem(bdi);
2695
2696     wait_sb_inodes(sb);
2697 }

```

Listing 10-39: `fs/fs-writeback.c: sync_inodes_sb()`

This establishes a struct `wb_writeback_work` object spanning all dirty folios (as indicated by setting `nr_pages` to `LONG_MAX`) for the struct `super_block` object describing the file system being synchronised. The object is marked to indicate that synchronisation should occur.

A struct `wb_completion` object is established via the `DEFINE_WB_COMPLETION()` macro, which is used to wait on the operation completing, which is waited on via `wb_wait_for_completion()`. Discussion of the details of this are omitted for brevity.

If the block device is unspecified we take no action and we provide a kernel warning should an unexpected race occur between this operation and unmounting of the super block.

After acquiring the appropriate lock, the actual queueing of the work is deferred to `bdi_split_work_to_wbs()` which we will examine later in Listing 10-43.

Finally, this invokes `wait_sb_inodes()` in order to synchronise concurrent synchronisation operations. Again, for brevity, we do not examine this in detail.

We examine the asynchronous version of this performed by `writeback_inodes_sb()` in Listing 10-40.

```

2628 /**
2629  * writeback_inodes_sb -      writeback dirty inodes from given super_block
2630  * @sb: the superblock
2631  * @reason: reason why some writeback work was initiated
2632  *
2633  * Start writeback on some inodes on this super_block. No guarantees are made
2634  * on how many (if any) will be written, and this function does not wait
2635  * for IO completion of submitted IO.
2636  */
2637 void writeback_inodes_sb(struct super_block *sb, enum wb_reason reason)
2638 {
2639     return writeback_inodes_sb_nr(sb, get_nr_dirty_pages(), reason);
2640 }
```

Listing 10-40: `fs/fs-writeback.c: writeback_inodes_sb()`

This function obtains an upper bound for the number of dirty pages via `get_nr_dirty_inodes()` and uses this to define the number of pages to write in `writeback_inodes_sb_nr()` to which it defers the operation, shown in Listing 10-41.

```

2610 /**
2611  * writeback_inodes_sb_nr -    writeback dirty inodes from given super_block
2612  * @sb: the superblock
2613  * @nr: the number of pages to write
2614  * @reason: reason why some writeback work initiated
2615  *
2616  * Start writeback on some inodes on this super_block. No guarantees are made
2617  * on how many (if any) will be written, and this function does not wait
```

```

2618  * for IO completion of submitted IO.
2619  */
2620  void writeback_inodes_sb_nr(struct super_block *sb,
2621                             unsigned long nr,
2622                             enum wb_reason reason)
2623  {
2624      __writeback_inodes_sb_nr(sb, nr, reason, false);
2625  }

```

Listing 10-41: fs/fs-writeback.c: writeback_inodes_sb_nr()

This performs writeback without synchronisation in the `__writeback_inodes_sb_nr()` function, shown in Listing 10-42.

```

2588  static void __writeback_inodes_sb_nr(struct super_block *sb, unsigned long nr,
2589                                     enum wb_reason reason, bool skip_if_busy)
2590  {
2591      struct backing_dev_info *bdi = sb->s_bdi;
2592      DEFINE_WB_COMPLETION(done, bdi);
2593      struct wb_writeback_work work = {
2594          .sb                = sb,
2595          .sync_mode         = WB_SYNC_NONE,
2596          .tagged_writepages = 1,
2597          .done              = &done,
2598          .nr_pages          = nr,
2599          .reason            = reason,
2600      };
2601
2602      if (!bdi_has_dirty_io(bdi) || bdi == &noop_backing_dev_info)
2603          return;
2604      WARN_ON(!rwsem_is_locked(&sb->s_umount));
2605
2606      bdi_split_work_to_wbs(sb->s_bdi, &work, skip_if_busy);
2607      wb_wait_for_completion(&done);
2608  }

```

Listing 10-42: fs/fs-writeback.c: __writeback_inodes_sb_nr()

This resembles `sync_inodes_sb()` (see Listing 10-39) except that synchronisation is not specified and the careful locking performed for the synchronous call is not performed here.

In each case, the heavy lifting is performed by `bdi_split_work_to_wbs()` which we examine in Listing 10-43.

```

925  /**
926   * bdi_split_work_to_wbs - split a wb_writeback_work to all wb's of a bdi
927   * @bdi: target backing_dev_info
928   * @base_work: wb_writeback_work to issue
929   * @skip_if_busy: skip wb's which already have writeback in progress
930   *

```

```

931  * Split and issue @base_work to all wb's (bdi_writeback's) of @bdi which
932  * have dirty inodes. If @base_work->nr_page isn't %LONG_MAX, it's
933  * distributed to the busy wbs according to each wb's proportion in the
934  * total active write bandwidth of @bdi.
935  */
936  static void bdi_split_work_to_wbs(struct backing_dev_info *bdi,
937                                   struct wb_writeback_work *base_work,
938                                   bool skip_if_busy)
939  {
940      struct bdi_writeback *last_wb = NULL;
941      struct bdi_writeback *wb = list_entry(&bdi->wb_list,
942                                             struct bdi_writeback, bdi_node);
943
944      might_sleep();
945  restart:
946      rcu_read_lock();
947      list_for_each_entry_continue_rcu(wb, &bdi->wb_list, bdi_node) {
948          DEFINE_WB_COMPLETION(fallback_work_done, bdi);
949          struct wb_writeback_work fallback_work;
950          struct wb_writeback_work *work;
951          long nr_pages;
952
953          if (last_wb) {
954              wb_put(last_wb);
955              last_wb = NULL;
956          }
957
958          /* SYNC_ALL writes out I_DIRTY_TIME too */
959          if (!wb_has_dirty_io(wb) &&
960              (base_work->sync_mode == WB_SYNC_NONE ||
961               list_empty(&wb->b_dirty_time)))
962              continue;
963          if (skip_if_busy && writeback_in_progress(wb))
964              continue;
965
966          nr_pages = wb_split_bdi_pages(wb, base_work->nr_pages);
967
968          work = kmalloc(sizeof(*work), GFP_ATOMIC);
969          if (work) {
970              *work = *base_work;
971              work->nr_pages = nr_pages;
972              work->auto_free = 1;
973              wb_queue_work(wb, work);
974              continue;
975          }
976

```



```

977          /* alloc failed, execute synchronously using on-stack fallback
          */
978          work = &fallback_work;
979          *work = *base_work;
980          work->nr_pages = nr_pages;
981          work->auto_free = 0;
982          work->done = &fallback_work_done;
983
984          wb_queue_work(wb, work);
985
986          /*
987           * Pin @wb so that it stays on @bdi->wb_list. This allows
988           * continuing iteration from @wb after dropping and
989           * reggrabbing rcu read lock.
990           */
991          wb_get(wb);
992          last_wb = wb;
993
994          rcu_read_unlock();
995          wb_wait_for_completion(&fallback_work_done);
996          goto restart;
997      }
998      rcu_read_unlock();
999
1000      if (last_wb)
1001          wb_put(last_wb);
1002  }

```

Listing 10-43: fs/fs-writeback.c: `bdi_split_work_to_wbs()`

This function is meaningful only when cgroup writeback is used, meaning that multiple struct `bdi_writeback` objects can be tied to a single BDI. This topic is therefore somewhat out of scope but as this is a key function, it's important to examine it nonetheless, albeit briefly.

This ultimately invokes `wb_queue_work()` described in Listing 10-34 above, splitting the work across struct `bdi_writeback` objects using `wb_split_bdi_pages()` to do so fairly based on writeback bandwidth.

An interesting point to note here is that this function attempts to perform the operation asynchronously, allocating its struct `wb_writeback_work` objects using a GFP_ATOMIC (since we are in a context where we can't take the locks that an GFP_KERNEL allocation would take see Section 2.6 for more details on GFP flags). If this fails, then it uses a stack-allocated struct `wb_writeback_work` object and does so synchronously.

10.12 Flusher Thread Operation

We have now explored how the flusher thread is ultimately invoked in detail, so it's time to examine what it actually does. The function which the flusher threads invoke is `wb_workfn()`, shown in Listing 10-44 (eliding trace hooks).

```

2201 /*
2202  * Handle writeback of dirty data for the device backed by this bdi. Also
2203  * reschedules periodically and does kupdated style flushing.
2204  */
2205 void wb_workfn(struct work_struct *work)
2206 {
2207     struct bdi_writeback *wb = container_of(to_delayed_work(work),
2208                                             struct bdi_writeback, dwork);
2209     long pages_written;
2210
2211     set_worker_desc("flush-%s", bdi_dev_name(wb->bdi));
2212
2213     if (likely(!current_is_workqueue_rescuer() ||
2214               !test_bit(WB_registered, &wb->state))) {
2215         /*
2216          * The normal path. Keep writing back @wb until its
2217          * work_list is empty. Note that this path is also taken
2218          * if @wb is shutting down even when we're running off the
2219          * rescuer as work_list needs to be drained.
2220          */
2221         do {
2222             pages_written = wb_do_writeback(wb);
2223             :
2224         } while (!list_empty(&wb->work_list));
2225     } else {
2226         /*
2227          * bdi_wq can't get enough workers and we're running off
2228          * the emergency worker. Don't hog it. Hopefully, 1024 is
2229          * enough for efficient IO.
2230          */
2231         pages_written = writeback_inodes_wb(wb, 1024,
2232                                             WB_REASON_FORKER_THREAD);
2233         :
2234     }
2235
2236     if (!list_empty(&wb->work_list))
2237         wb_wakeup(wb);
2238     else if (wb_has_dirty_io(wb) && dirty_writeback_interval)
2239         wb_wakeup_delayed(wb);
2240 }

```

Listing 10-44: `fs/fs-writeback.c: wb_workfn()`

The flusher thread is implemented as a wait queue thread as described at the start of Section 10.11. It is established with the `WQ_MEM_RECLAIM` flag set (see `default_bdi_init()` shown in Listing 10-31), the details of which are out of scope for the book but generally this means that a ‘rescuer’ thread always exists to ensure that, even under extreme memory pressure, this function executes.

If you examine running processes, you’ll observe a `[writeback]` kernel thread. This is running the rescuer thread, as can be observed using `procs` as shown in Listing 10-45.

```
$ sudo cat /proc/$(pidof -sw "writeback")/stack
[<0>] rescuer_thread+0x2b0/0x3b0
[<0>] kthread+0xe5/0x120
[<0>] ret_from_fork+0x31/0x50
[<0>] ret_from_fork_asm+0x1b/0x30
```

Listing 10-45: procs Output of Writeback Rescuer Thread Call Stack

This thread is not used outside of extreme memory pressure, rather the threads that typically execute writeback are *kworker* threads, invoked by the kernel’s wait queue implementation and used by `bdi_wq`. These have rather generic names, something like `kworker/u32:0-writeback`. This indicates that the thread is invoked *unbound* (meaning it can run on any core) on pool 32 and happens to be assigned to CPU 0 to handle writeback events.

These kernel threads are only guaranteed to exist during the actual operation and are, by their natural, ephemeral. However to assist with identification of these threads should they be long lasting, `wb_workfn()` adds an additional description to the running thread using `set_worker_desc()`.

Taking the example of `kworker/u32:0-events_unbound`, this thread will instead be renamed to `kworker/u32:0-flush-259:0`, assuming that the BDI being written back to has major and minor version number of 259 and 0 respectively.

As a result of this rescuer thread mechanism we can observe in Listing 10-44, note that `wb_workfn()` takes special care to handle this case via `current_is_workqueue_rescuer()`.

If it is not, or the struct `bdi_writeback` to which this operation belongs is deregistered so needs clean up on the usual path (i.e. work list drained), then writeback is invoked via the core path in `wb_do_writeback()` which we examine below in Listing 10-50.

We invoke `wb_do_writeback()` at least once, repeating the operation should work remain on the struct `bdi_writeback->work_list` indicating that immediate synchronisation work needs to be performed.

If the rescuer thread is in use, then a special path is taken to prevent hogging of this extremely limited resource. The `writeback_inodes_wb()` function is invoked and writes back 1,024 pages in the background. We examine this function in Listing 10-49.

Finally the function must reschedule itself, as once the delayed work item assigned to `bdi_wq` has been executed, invoking this function, it is also removed the work queue altogether.

If, despite looping on writeback, the struct `bdi_writeback->work_list` is still non-empty, then we immediately re-invoke the function via `wb_wakeup()`, otherwise we schedule the next invocation `vm.dirty_writeback_centisecs` later via `wb_wakeup_delayed()`, only if we have outstanding I/O or dirty folios to process. We can rely on `__mark_inode_dirty()` to wake up the flusher threads again if new dirtying occurs after all inodes have been submitted for writeback.

We examine the code paths invoked by `wb_workfn()` in Figure 10-8.

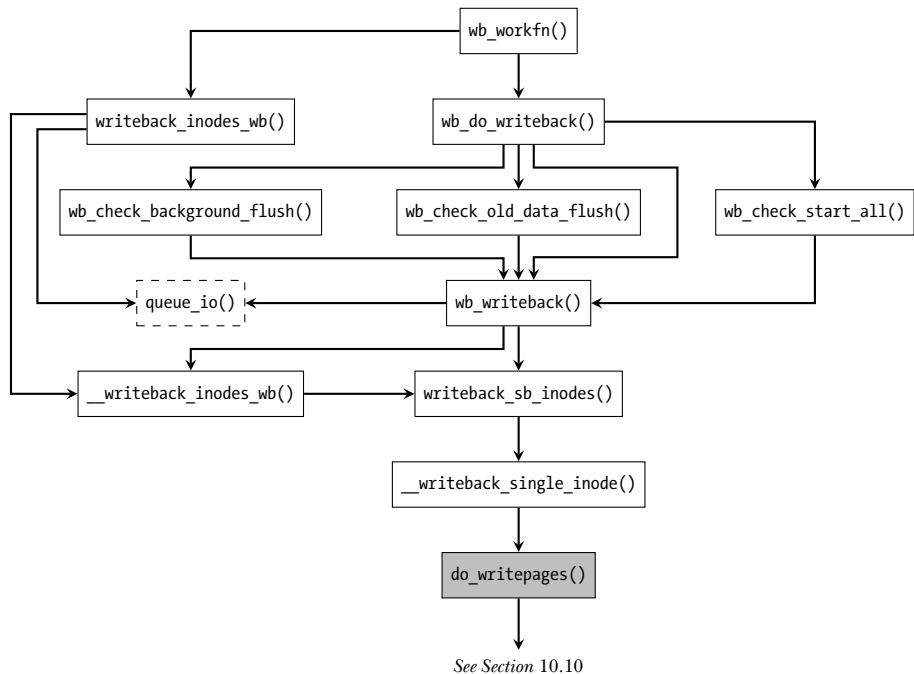


Figure 10-8: `wb_workfn()` Code Paths

NOTE

`queue_io()` is called by both `writeback_inodes_wb()` and `wb_writeback()`, prior to calling `writeback_sb_inodes()`. We denote this with a dashed outline, but since `queue_io()` is so critical a function it's important to highlight it here.

Ultimately the task of the functions shown in Figure 10-8 is to prepare struct `wb_writeback_work` objects to define the actual writeback work to be performed by `writeback_sb_inodes()`, and to move dirtied inodes from struct `bdi_writeback->b_dirty` to struct `bdi_writeback->b_io` (and possibly struct `bdi_writeback->b_more_io` which we will explore shortly) before performing the actual writeback via `do_writepages()` as described in Section 10.10.

Both `__writeback_inodes_wb()` and `wb_writeback()` use `queue_io()`. This is a key function which transfers dirty inodes from the struct `bdi_writeback->b_dirty` list to the `b_io` list, which we examine in Listing 10-46 (eliding out of scope dirty time handling and trace callbacks).

Dirtied inodes transition through three different lists—`b_dirty`, `b_io` and `b_more_io`. The key move is from `b_dirty` to `b_io`, which determines which inodes will be processed by `writeback_sb_inodes()` (see Listing 10-57) and are placed there by `queue_io()` (see Listing 10-46).

We examine how these transition in Figure 10-9 (eliding out of scope cgroup logic and list moves performed by `writeback_single_inode()`).

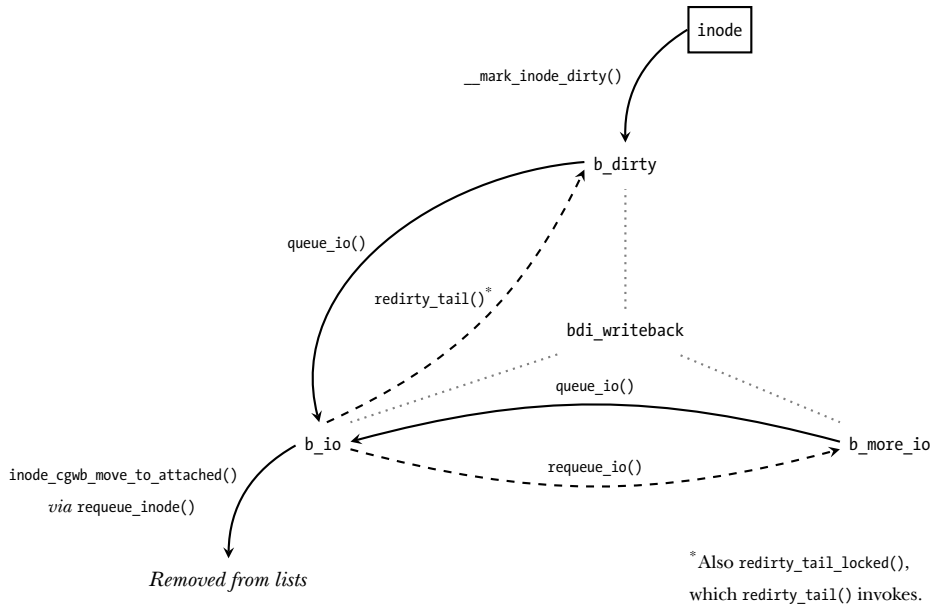


Figure 10-9: *struct bdi_writeback* lifecycle in `writeback_sb_inodes()`

A folio that is written back at a file-system level (i.e. not synchronised by `fsync()` or similar system calls) goes through the following life-cycle (as shown in Figure 10-9):

Dirtying When an inode is first dirtied, it is added to the `struct bdi_writeback->b_dirty` list by `__mark_inode_dirty()` (see Section for dirty tracking via `write()` and Section for dirty tracking via page fault, both of which end up invoking `__mark_inode_dirty()`).

Queueing When the writeback flusher thread for a `struct bdi_writeback` object has been invoked (as shown in Figure 10-8), it ultimately places eligible inodes from the `b_dirty` list into the `b_io` list via `queue_io()` (see Listing 10-46).

In addition, `queue_io()` moves any previously outstanding I/O from the `b_more_io` list back to `b_io` (we will examine this in detail, but briefly, if inodes are unable to be submitted for write back due to bandwidth or other constraints, they get added to `b_more_io` to be requeued at the next invocation).

Submitting As shown in Figure 10-8, once inodes have been queued for I/O, `writeback_sb_inodes()` (see Listing 10-57) is invoked for all file sys-

tem writeback. This function iterates through all inodes belonging to the specified struct `super_block` and struct `bdi_writeback` and attempts to submit them for I/O.

Requeueing If `writeback_sb_inodes()` is unable to submit inodes for writeback to the block level (discussed in Section 9.10.4), then the inodes must be *requeued*, i.e. either placed back into `b_dirty` via `redirty_tail_locked()` or `redirty_tail()`, or placed into `b_more_io` to be processed the next time the flusher thread is invoked via `requeue_io()`. We discuss both cases this when we examine this function in Listing 10-57.

Dequeuing If everything has gone correctly in `writeback_sb_inodes()` and no requeueing was yet necessary, `requeue_inode()` is invoked, which checks each state in which the inode might be in, requeueing if necessary, but if all has gone as planned and the inode has been correctly submitted for I/O, it is then removed from the `b_io` list altogether via `inode_cgwb_move_to_attached()` which has cgroup-specific handling but also deletes the inode from the list.

NOTE

An important point to pay attention to here is that the `b_dirty` list is ordered from youngest to eldest. Therefore, `__mark_inode_dirty()` uses `inode_io_list_move_locked()` to place the newly dirtied folio at the head of struct `bdi_writeback->b_dirty`, however `writeback_sb_inodes()` processes inodes that have been moved from `b_dirty` to `b_io` backwards as it is the inode updated least recently that must be written back first.

In exploring this lifecycle, we will first examine the *queueing* step, performed by `queue_io()` in Listing 10-46.

```

1405 /*
1406  * Queue all expired dirty inodes for io, eldest first.
1407  * Before
1408  *      newly dirtied      b_dirty      b_io      b_more_io
1409  *      =====>         gf          edc      BA
1410  * After
1411  *      newly dirtied      b_dirty      b_io      b_more_io
1412  *      =====>         g           fBAedc
1413  *                               |
1414  *                               +--> dequeue for IO
1415  */
1416 static void queue_io(struct bdi_writeback *wb, struct wb_writeback_work *work,
1417                     unsigned long dirtied_before)
1418 {
1419     int moved;
1420     :
1421     :
1422     assert_spin_locked(&wb->list_lock);
1423     list_splice_init(&wb->b_more_io, &wb->b_io);
1424     moved = move_expired_inodes(&wb->b_dirty, &wb->b_io, dirtied_before);
1425     :
1426     if (moved)

```

```

1430         wb_io_lists_populated(wb);
        :
1432     }

```

Listing 10-46: fs/fs-writeback.c: queue_io()

Note that this function is parameterised by `dirtyed_before`, which specifies the struct `inode->dirtyed_when` at which time (or prior) inodes are considered for writeback.

This varies depending on the caller—`writeback_inodes_wb()` simply writes back inodes based on the current time regardless (see Listing 10-49 below), whereas `wb_writeback()` specifies the current time less `vm.dirty_expire_centisecs` if the struct `wb_writeback_work` object specifies that the writeback is periodic background writeback (we examine this in Listing 10-55).

This function starts by invoking `list_splice_init()`, which takes the list specified in the first parameter (struct `bdi_writeback->b_more_io`), and places it at the *front* of the list specified in the second parameter (struct `bdi_writeback->b_io`), and clears the original list.

Importantly, `queue_io()` is *only* invoked if struct `bdi_writeback->b_io` is empty, so `b_more_io`, if non-empty, is always processed first before any other I/O is queued, and this queueing is not repeated before all entries are processed by `writeback_sb_inodes()`.

Note that `writeback_sb_inodes()` processes inodes in struct `bdi_writeback->b_io` *backwards*, so the existing order of inodes is preserved.

Next, inodes which are older than `dirtyed_before` are moved from struct `bdi_writeback->b_dirty` to `b_io` via `move_expired_inodes()` which we examine in Listing 10-47.

```

1350 /*
1351  * Move expired (dirtyed before dirtyed_before) dirty inodes from
1352  * @delaying_queue to @dispatch_queue.
1353  */
1354 static int move_expired_inodes(struct list_head *delaying_queue,
1355                               struct list_head *dispatch_queue,
1356                               unsigned long dirtyed_before)
1357 {
1358     LIST_HEAD(tmp);
1359     struct list_head *pos, *node;
1360     struct super_block *sb = NULL;
1361     struct inode *inode;
1362     int do_sb_sort = 0;
1363     int moved = 0;
1364
1365     while (!list_empty(delaying_queue)) {
1366         inode = wb_inode(delaying_queue->prev);
1367         if (inode_dirtyed_after(inode, dirtyed_before))
1368             break;

```

```

1369         spin_lock(&inode->i_lock);
1370         list_move(&inode->i_io_list, &tmp);
1371         moved++;
1372         inode->i_state |= I_SYNC_QUEUED;
1373         spin_unlock(&inode->i_lock);
1374         if (sb_is_blkdev_sb(inode->i_sb))
1375             continue;
1376         if (sb && sb != inode->i_sb)
1377             do_sb_sort = 1;
1378         sb = inode->i_sb;
1379     }
1380
1381     /* just one sb in list, splice to dispatch_queue and we're done */
1382     if (!do_sb_sort) {
1383         list_splice(&tmp, dispatch_queue);
1384         goto out;
1385     }
1386
1387     /*
1388      * Although inode's i_io_list is moved from 'tmp' to 'dispatch_queue',
1389      * we don't take inode->i_lock here because it is just a pointless
1390      * Inode is already marked as I_SYNC_QUEUED so writeback list handling
1391      * is
1392      * fully under our control.
1393      */
1394     while (!list_empty(&tmp)) {
1395         sb = wb_inode(tmp.prev)->i_sb;
1396         list_for_each_prev_safe(pos, node, &tmp) {
1397             inode = wb_inode(pos);
1398             if (inode->i_sb == sb)
1399                 list_move(&inode->i_io_list, dispatch_queue);
1400         }
1401     }
1402 out:
1403     return moved;
1404 }

```

Listing 10-47: fs/fs-writeback.c: `move_expired_inodes()`

The `delaying_queue` parameter is set to `struct bdi_writeback->b_dirty` and `dispatch_queue` is set to `struct bdi_writeback->b_io`.

The function loops through dirty inodes *backwards*, obtaining the `struct inode` object associated with each entry via `wb_inode()`, and moves the inode to the *front* of the `tmp` list if it is as old or older than `dirtyed_before` (as checked by `inode_dirtyed_after()`).

This traversal assumes that `b_dirty` is maintained in an ordering of youngest inode to eldest, which is correct, as at each point of being dirtied,

an inode is placed at the *front* of the list. This makes the list a queue, though in reverse order.

Traversing the list in reverse and appending to the front of the accumulating list maintains the order. Consider an input list:

ABCD

After each iteration of this loop, assuming all inodes are old enough to be queued, the list will be accumulated thusly:

D

CD

BCD

ABCD

And therefore the order is maintained as expected.

The inode has the `I_SYNC_QUEUED` flag set which indicates it is queued for I/O, which is cleared once the inode has been requeued or dequeued.

Finally in this loop, we check to determine whether the superblock containing the inode is the special blockdev superblock, if not, the loop updates the `do_sb_sort` variable to indicate whether the inodes span multiple superblocks, in which case they must be sorted by super block.

If we are dealing with inodes in a single super block, then no sorting is required and we simply transfer all inodes directly. Otherwise, `writeback_sb_inodes()` expects inodes to be grouped by superblock.

The final loop iterates backwards through each entry in `b_dirty`, checking for each inode that has the same superblock as the final entry, moving these from the `tmp` list to the output queue accordingly.

Consider a `b_dirty` list consisting of inodes from 3 superblocks `a`, `b` and `c`. These will be present in `b_dirty` ordered from youngest to eldest, but inodes of different superblocks will be interleaved. Denoting the ordering by subscript, consider the following `b_dirty` (and thus `tmp`, assuming all expired) list:

`a1a2b1a3c1c2b2a4`

In the first iteration, the last entry is `a4` so we accumulate entries to `b_io` for superblock `a` as follows:

`a4`

`a3a4`

`a2a3a4`

`a1a2a3a4`

This leaves `tmp` as follows:

`b1c1c2b2`

We next append entries to `b_io` for superblock `b` as follows:

`b2a1a2a3a4`

`b1b2a1a2a3a4`

Finally, we are left with `tmp` as follows:

`c1c2`

Which trivially gets prepended to `b_io` thusly:

`c2b1b2a1a2a3a4`

`c1c2b1b2a1a2a3a4`

When we examine `writeback_sb_inodes()` in Listing 10-57, we will see how this ordering is relied upon to correctly process superblocks as expected.

Finally, the function returns a boolean value indicating whether inodes were moved or not.

Returning to `queue_io()` shown in Listing 10-46, we note that after the move has occurred, we invoke `wb_io_lists_populated()`, which we examine in Listing 10-48.

```

85 static bool wb_io_lists_populated(struct bdi_writeback *wb)
86 {
87     if (wb_has_dirty_io(wb)) {
88         return false;
89     } else {
90         set_bit(WB_has_dirty_io, &wb->state);
91         WARN_ON_ONCE(!wb->avg_write_bandwidth);
92         atomic_long_add(wb->avg_write_bandwidth,
93                        &wb->bdi->tot_write_bandwidth);
94         return true;
95     }
96 }
```

Listing 10-48: fs/fs-writeback.c: `wb_io_lists_populated()`

This function returns a boolean indicating whether this is the first dirty inode added to the `struct bdi_writeback->b_dirty` list, which is used in `inode_io_list_move_locked()` which is used, in turn by `__mark_inode_dirty()` (see Listing 10-2) to wake up flusher threads on first dirty.

However, `queue_io()` ignores the return value and instead uses this to set the `WB_has_dirty_io` flag in `struct bdi_writeback->flags`, and also updates writeback bandwidth statistics (we discuss bandwidth and throttling in Section 10.14).

The sister function of `wb_io_lists_populated()`, `wb_io_lists_depopulated()`, is invoked when inodes are *removed* from I/O lists, which clears this flag if there are *no* inodes on *any* of the b-lists, dirty or I/O.

NOTE

We can observe from these hook functions that the writeback flusher threads are an entirely on-demand facility. The `WB_has_dirty_io` flag serialises any ongoing process of flusher thread writeback and dirtying of folios, ensuring that, should there be dirty folios to flush, the thread is always woken up, but if there aren't, it isn't.

Now we have examined inode queuing, and before diving into the core `wb_do_writeback()` function which dispatches the various forms of writeback (and thusly different forms of queuing I/O) we examine `writeback_inodes_wb()`, which is invoked by `wb_workfn()` when memory is critically low and the ‘rescuer’ thread is used. We examine this function in Listing 10-49, eliding out of scope block device plug logic.

```

1951 static long writeback_inodes_wb(struct bdi_writeback *wb, long nr_pages,
1952                               enum wb_reason reason)
```

```

1953 {
1954     struct wb_writeback_work work = {
1955         .nr_pages      = nr_pages,
1956         .sync_mode     = WB_SYNC_NONE,
1957         .range_cyclic  = 1,
1958         .reason        = reason,
1959     };
1960     :
1961     spin_lock(&wb->list_lock);
1962     if (list_empty(&wb->b_io))
1963         queue_io(wb, &work, jiffies);
1964     __writeback_inodes_wb(wb, &work);
1965     spin_unlock(&wb->list_lock);
1966     :
1967     return nr_pages - work.nr_pages;
1968 }

```

Listing 10-49: fs/fs-writeback.c: writeback_inodes_wb()

This function establishes struct `wb_writeback_work` with synchronisation disabled and with cyclical range enabled (meaning blocks of `nr_pages` pages are written back cycling around the page cache entries for each inode in each superblock).

We acquire struct `bdi_writeback->list_lock` in order to modify the `b_`-lists, then invoke `queue_io()` to transfer inodes with outstanding I/O from `b_more_io` to `b_io` and unconditionally queue all `b_dirty` inodes regardless of whether they have expired or not, as this is an emergency measure for extreme memory pressure, and dirty inodes must be written back in order for reclaim to proceed.

In this instance the actual writeback is performed by `__writeback_inodes_wb()`, which we shall return to in Listing 10-56.

Having examined this emergency writeback mode, we return to the usual procedure and examine `wb_do_writeback()` in Listing 10-50 (eliding out of scope tracing hook).

```

2171 /*
2172  * Retrieve work items and do the writeback they describe
2173  */
2174 static long wb_do_writeback(struct bdi_writeback *wb)
2175 {
2176     struct wb_writeback_work *work;
2177     long wrote = 0;
2178     :
2179     set_bit(WB_writeback_running, &wb->state);
2180     while ((work = get_next_work_item(wb)) != NULL) {
2181         :
2182         wrote += wb_writeback(wb, work);
2183         finish_writeback_work(wb, work);
2184     }

```

```

2185
2186     /*
2187      * Check for a flush-everything request
2188      */
2189     wrote += wb_check_start_all(wb);
2190
2191     /*
2192      * Check for periodic writeback, kupdated() style
2193      */
2194     wrote += wb_check_old_data_flush(wb);
2195     wrote += wb_check_background_flush(wb);
2196     clear_bit(WB_writeback_running, &wb->state);
2197
2198     return wrote;
2199 }

```

Listing 10-50: fs/fs-writeback.c: wb_do_writeback()

During core writeback in `wb_do_writeback()`, the `struct bdi_writeback->state` field has the `WB_writeback_running` flag set, which is used by the `writeback_in_progress()` to determine whether a writeback is in progress, used by `balance_dirty_pages()` and `bdi_split_work_to_wbs()` to avoid unnecessary work.

The function starts by looping through all of the writeback work in addition to the standard periodic writeback, `get_next_work_item()` to pop the next entry off the `struct bdi_writeback->work_list`. We examine this in Listing ??.

```

2078 /*
2079  * Return the next wb_writeback_work struct that hasn't been processed yet.
2080  */
2081 static struct wb_writeback_work *get_next_work_item(struct bdi_writeback *wb)
2082 {
2083     struct wb_writeback_work *work = NULL;
2084
2085     spin_lock_irq(&wb->work_lock);
2086     if (!list_empty(&wb->work_list)) {
2087         work = list_entry(wb->work_list.next,
2088                         struct wb_writeback_work, list);
2089         list_del_init(&work->list);
2090     }
2091     spin_unlock_irq(&wb->work_lock);
2092     return work;
2093 }

```

Listing 10-51: fs/fs-writeback.c: get_next_work_item()

This is the mirror image of `wb_queue_work()` (shown in Listing 10-34), which places `struct wb_writeback_work` entries on the *tail* (i.e. end) of the list, so here we pop from the *head* of the list (i.e. its first entry) each time. This makes this a work *queue*.

Each time an entry is retrieved the `work_lock` is taken and the work entry is deleted from the work list and reinitialised via `list_del_init()`.

Each individual work entry is then passed directly to `wb_writeback()`, the core writeback function, as a priority before any periodic work is considered.

This is vital, as this work will be specified by synchronisation operations and specify writeback work that must be performed immediately.

After each work item is processed, `finish_writeback_work()` is invoked which handles freeing of the work object if `struct wb_writeback_work->auto_free` is specified and calling back any callers which are waiting on `struct wb_writeback_work->done`.

Once these work list items have been processed, we proceed with the usual, periodic cases, which are handled by three different functions—`wb_check_start_all()`, `wb_check_old_data_flush()` and `wb_check_background_flush()`.

The `wb_check_start_all()` function is the function which ultimately performs the writeback that `wb_start_writeback()` scheduled (see Listing 10-37). We examine this in Listing 10-52.

```

2147 static long wb_check_start_all(struct bdi_writeback *wb)
2148 {
2149     long nr_pages;
2150
2151     if (!test_bit(WB_start_all, &wb->state))
2152         return 0;
2153
2154     nr_pages = get_nr_dirty_pages();
2155     if (nr_pages) {
2156         struct wb_writeback_work work = {
2157             .nr_pages      = wb_split_bdi_pages(wb, nr_pages),
2158             .sync_mode     = WB_SYNC_NONE,
2159             .range_cyclic  = 1,
2160             .reason        = wb->start_all_reason,
2161         };
2162
2163         nr_pages = wb_writeback(wb, &work);
2164     }
2165
2166     clear_bit(WB_start_all, &wb->state);
2167     return nr_pages;
2168 }

```

Listing 10-52: fs/fs-writeback.c: `wb_check_start_all()`

This uses `get_nr_dirty_pages()` to obtain an estimate of the number of dirty pages, establishing a `struct wb_writeback_work` object, specifying that the number of pages should be scaled to the write bandwidth of the `struct bdi_writeback` object describing the writeback via `wb_split_bdi_pages()`. This is specifically implemented to handle cgroup-mediated writeback which is out of scope of the book so we will not examine this.

The work is cyclic, and thus we set `struct address_space->writeback_index` of each relevant page cache entry to the next index at which we should resume writeback (see discussion around Listing 10-25 for more details).

The work is only performed if the `WB_start_all` flag is set in `struct bdi_writeback->state`, and cleared once it is complete. We have already ensured only one such action should be in flight at any one time (see the discussion around Listing Listing 10-37 for details).

Ultimately the function, as with each that initiate file system writeback, defers to `wb_writeback()` (see Listing 10-55).

The `wb_check_background_flush()` function checks to see if the background writeback limits established by `vm.dirty_background_bytes` or `vm.dirty_background_ratio` have been exceeded. We examine it in Listing ??.

```

2095 static long wb_check_background_flush(struct bdi_writeback *wb)
2096 {
2097     if (wb_over_bg_thresh(wb)) {
2098
2099         struct wb_writeback_work work = {
2100             .nr_pages      = LONG_MAX,
2101             .sync_mode     = WB_SYNC_NONE,
2102             .for_background = 1,
2103             .range_cyclic  = 1,
2104             .reason        = WB_REASON_BACKGROUND,
2105         };
2106
2107         return wb_writeback(wb, &work);
2108     }
2109
2110     return 0;
2111 }

```

Listing 10-53: fs/fs-writeback.c: `wb_check_background_flush()`

This determines whether the background writeback threshold has been exceeded via `wb_over_bg_thresh()`. We examine this in Section 10.14 and Listing 10-103. If the limits are exceeded, we then perform background writeback of all dirty pages attached to the the `struct bdi_writeback`.

We indicate that this operation is for background writeback by setting the `for_background` flag.

These prior functions are run only if either synchronisation is requested or background writeback is triggered, the true workhorse of writeback is `wb_check_old_data_flush()`, which performs ordinary periodic writeback. We examine it in Listing ??.

```

2113 static long wb_check_old_data_flush(struct bdi_writeback *wb)
2114 {
2115     unsigned long expired;
2116     long nr_pages;
2117

```

```

2118     /*
2119     * When set to zero, disable periodic writeback
2120     */
2121     if (!dirty_writeback_interval)
2122         return 0;
2123
2124     expired = wb->last_old_flush +
2125             msec_to_jiffies(dirty_writeback_interval * 10);
2126     if (time_before(jiffies, expired))
2127         return 0;
2128
2129     wb->last_old_flush = jiffies;
2130     nr_pages = get_nr_dirty_pages();
2131
2132     if (nr_pages) {
2133         struct wb_writeback_work work = {
2134             .nr_pages      = nr_pages,
2135             .sync_mode     = WB_SYNC_NONE,
2136             .for_kupdate   = 1,
2137             .range_cyclic  = 1,
2138             .reason        = WB_REASON_PERIODIC,
2139         };
2140
2141         return wb_writeback(wb, &work);
2142     }
2143
2144     return 0;
2145 }

```

Listing 10-54: fs/fs-writeback.c: wb_check_old_data_flush()

The `vm.dirty_expire_centisecs` tunable is maintained in the `dirty_writeback_interval` global variable. If this is set to zero, this implies that periodic writeback is simply disabled and so the function exists.

Otherwise, before proceeding with periodic writeback, we check to determine whether `struct bdi_writeback->last_old_flush` is younger than `vm.dirty_expire_centisecs`. If so, we abort the operation, which helps us avoid fruitless examination of inodes which will not be old enough to have expired.

We indicate that the writeback is the standard periodic writeback of expired inodes by setting the `for_kupdate` flag. This form of writeback can be referred to as a ‘kupdate’ writeback for historic reasons.

Again we initiate a cyclic writeback for all dirty pages and defer the heavy lifting to `wb_writeback()`, which we examine in Listing 10-55 (eliding out of scope trace and block device plug hooks).

```

1973 /*
1974 * Explicit flushing or periodic writeback of "old" data.
1975 *

```

```

1976  * Define "old": the first time one of an inode's pages is dirtied, we mark
1977  the
1978  * dirtying-time in the inode's address_space. So this periodic writeback
1979  code
1980  * just walks the superblock inode list, writing back any inodes which are
1981  * older than a specific point in time.
1982  *
1983  * Try to run once per dirty_writeback_interval. But if a writeback event
1984  * takes longer than a dirty_writeback_interval interval, then leave a
1985  * one-second gap.
1986  *
1987  * dirtied_before takes precedence over nr_to_write. So we'll only write back
1988  * all dirty pages if they are all attached to "old" mappings.
1989  */
1990  static long wb_writeback(struct bdi_writeback *wb,
1991                          struct wb_writeback_work *work)
1992  {
1993      long nr_pages = work->nr_pages;
1994      unsigned long dirtied_before = jiffies;
1995      struct inode *inode;
1996      long progress;
1997      :
1998      spin_lock(&wb->list_lock);
1999      for (;;) {
2000          /*
2001           * Stop writeback when nr_pages has been consumed
2002           */
2003          if (work->nr_pages <= 0)
2004              break;
2005
2006          /*
2007           * Background writeout and kupdate-style writeback may
2008           * run forever. Stop them if there is other work to do
2009           * so that e.g. sync can proceed. They'll be restarted
2010           * after the other works are all done.
2011           */
2012          if ((work->for_background || work->for_kupdate) &&
2013              !list_empty(&wb->work_list))
2014              break;
2015
2016          /*
2017           * For background writeout, stop when we are below the
2018           * background dirty threshold
2019           */
2020          if (work->for_background && !wb_over_bg_thresh(wb))
2021              break;
2022

```



```

2023      /*
2024      * Kupdate and background works are special and we want to
2025      * include all inodes that need writing. Livelock avoidance is
2026      * handled by these works yielding to any other work so we are
2027      * safe.
2028      */
2029      if (work->for_kupdate) {
2030          dirtied_before = jiffies -
2031                          msecs_to_jiffies(dirty_expire_interval * 10);
2032      } else if (work->for_background)
2033          dirtied_before = jiffies;
2034      :
2035
2036      if (list_empty(&wb->b_io))
2037          queue_io(wb, work, dirtied_before);
2038      if (work->sb)
2039          progress = writeback_sb_inodes(work->sb, wb, work);
2040      else
2041          progress = __writeback_inodes_wb(wb, work);
2042      :
2043
2044      /*
2045      * Did we write something? Try for more
2046      *
2047      * Dirty inodes are moved to b_io for writeback in batches.
2048      * The completion of the current batch does not necessarily
2049      * mean the overall work is done. So we keep looping as long
2050      * as made some progress on cleaning pages or inodes.
2051      */
2052      if (progress)
2053          continue;
2054      /*
2055      * No more inodes for IO, bail
2056      */
2057      if (list_empty(&wb->b_more_io))
2058          break;
2059      /*
2060      * Nothing written. Wait for some inode to
2061      * become available for writeback. Otherwise
2062      * we'll just busyloop.
2063      */
2064      :
2065      inode = wb_inode(wb->b_more_io.prev);
2066      spin_lock(&inode->i_lock);
2067      spin_unlock(&wb->list_lock);
2068      /* This function drops i_lock... */
2069      inode_sleep_on_writeback(inode);
2070      spin_lock(&wb->list_lock);
2071  }

```

```

2072     spin_unlock(&wb->list_lock);
      :
2075     return nr_pages - work->nr_pages;
2076 }

```

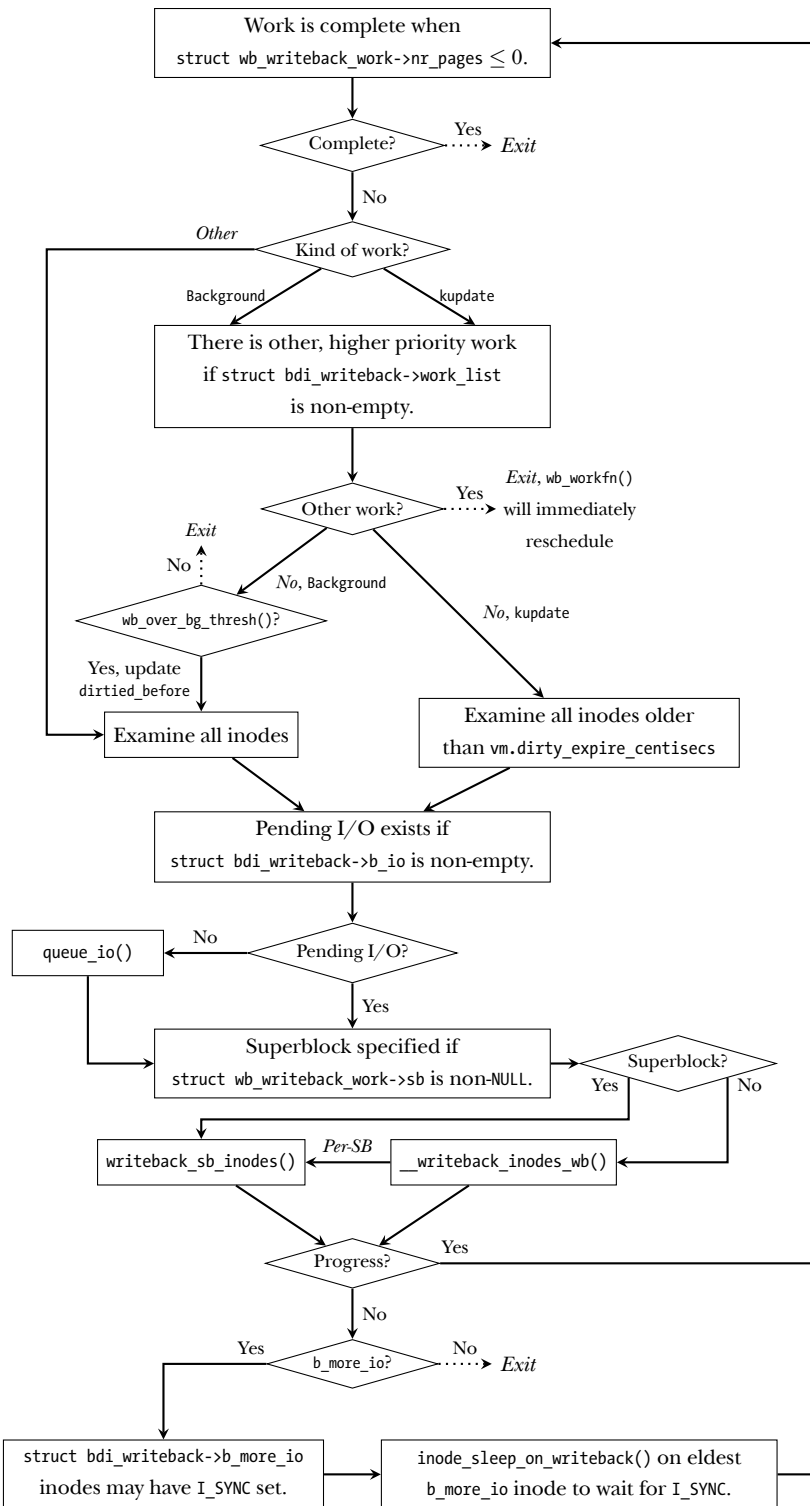
Listing 10-55: fs/fs-writeback.c: wb_writeback()

NOTE

*Moving forward we will refer to the standard periodic background writeback as kupdate to clearly separate this from **inl**obol**d**background writeback executed because the `vm.background_ratio` or `vm.background_bytes` background dirty limits were exceeded.*

This function loops until either—the struct `wb_writeback_work->nr_pages` field is at or below zero, indicating the work is complete, the task is background or kupdate writeback but there is outstanding higher priority work to be performed, the task is performing background writeback but the threshold has been reached, or finally no progress has been made.

We examine the details of the logic of this function in Figure 10-10.

Figure 10-10: `wb_writeback()` Logic

NOTE

An important nuance to note here is that the `dirtyed_before` argument is set prior to the loop starting. Therefore, any inodes dirtied after the loop begins will be processed, avoiding live locks where newly inodes cause the loop to continue indefinitely. This value is updated, however, if this is background or kupdate writeback as described below.

This operates in a loop, processing all pending work specified by functions called by `wb_workfn()` (see Listing 10-44), whose call graph can be observed in Figure 10-8.

Note that the function relies on this when encountering pending work items in `struct bdi_writeback->work_list`, exiting in order that `wb_workfn()` can immediately reinvoke the flusher thread to process the pending work, prioritising it over any background or kupdate writeback.

Equally, `writeback_sb_inodes()` and `__writeback_inodes_wb()` periodically exit in order to check termination conditions such as the background writeback threshold no longer being exceeded for a background writeback task (checked via `wb_over_bg_thresh()`, which we examine in Section 10.14 and Listing 10-103).

If this is a background writeback, the `dirtyed_when` parameter is reset to the current time permitting the writeback of newly dirtied inodes until dirty pages is reduced below the background dirty threshold. Equally a kupdate writeback invocation examines inodes older than the current time less `vm.dirty_expire_centisecs` and updates this on each loop. Other forms of writeback do not reset this time.

It is here where the core invocation of `queue_io()` is performed (see Listing 10-46), before either `writeback_sb_inodes()` (see Listing 10-57) or `__writeback_inodes_wb()` are invoked (see Listing 10-56). The former is the key writeback function for writeback to a specific superblock, which the latter invokes for each inode's super block individually.

We partition inodes by superblock as we require each super block to be locked against unmount during the writeback operation. `__writeback_inodes_wb()` acquires this lock for each inode's super block which we will examine in Listing 10-56.

As discussed around the listing for `move_expired_inodes()` (invoked by `queue_io()`), Listing 10-47, the queued inode list is sorted by super block, so we will already naturally have this ordering when processing inodes.

As further inodes might have been dirtied during this operation, this function will continue to loop as long as progress is made by `writeback_sb_inodes()`.

If no progress is made, then we check the edge case where inodes could not be processed due to various reasons (which we will examine shortly when we explore `writeback_sb_inodes()`), and have thus been appended to `struct bdi_writeback->b_more_io`.

These may be due to a concurrent writeback process having been initiated. When an inode is submitted to `writeback_sb_inodes()`, it has its `i_state` field updated with the `I_SYNC` flag set, which is only cleared once the inode has all dirty folios submitted for writeback to the underlying block device.

This failing to have occurred, resulting in the inode being transferred to `struct bdi_writeback->b_more_io`, means it is sensible for us to wait for the next inode which we plan to process to have its I/O submitted to avoid potentially busy-looping in `wb_writeback()`.

This is performed by `inode_sleep_on_writeback()` which causes the process to sleep until this flag is cleared, if it is indeed currently set.

We next examine the function which performs writeback on all super blocks owned by all inodes scheduled for writeback, `__writeback_inodes_wb()`, which we examine in Listing 10-56.

```

1917 static long __writeback_inodes_wb(struct bdi_writeback *wb,
1918                                 struct wb_writeback_work *work)
1919 {
1920     unsigned long start_time = jiffies;
1921     long wrote = 0;
1922
1923     while (!list_empty(&wb->b_io)) {
1924         struct inode *inode = wb_inode(wb->b_io.prev);
1925         struct super_block *sb = inode->i_sb;
1926
1927         if (!trylock_super(sb)) {
1928             /*
1929              * trylock_super() may fail consistently due to
1930              * s_umount being grabbed by someone else. Don't use
1931              * requeue_io() to avoid busy retrying the inode/sb.
1932              */
1933             redirty_tail(inode, wb);
1934             continue;
1935         }
1936         wrote += writeback_sb_inodes(sb, wb, work);
1937         up_read(&sb->s_umount);
1938
1939         /* refer to the same tests at the end of writeback_sb_inodes
1940          */
1941         if (wrote) {
1942             if (time_is_before_jiffies(start_time + HZ / 10UL))
1943                 break;
1944             if (work->nr_pages <= 0)
1945                 break;
1946         }
1947     }
1948     /* Leave any unwritten inodes on b_io */
1949     return wrote;
1950 }

```

Listing 10-56: fs/fs-writeback.c: `__writeback_inodes_wb()`

This function simply iterates through each of the inodes in `struct bdi_writeback->b_io`, working backwards, therefore working from the oldest inode to youngest.

It then attempts to acquire the inode's `struct super_block->s_umount` lock via `trylock_super()`. It does not wait on this lock, if the lock is not immediately available, the inode is immediately 'redirtied', i.e. placed on the `struct bdi_writeback->b_dirty` list via `redirty_tail()` (we examine this later in Listing 10-59).

Each inode is then subject to writeback via the core `writeback_sb_inodes()` function (see Listing 10-57). This function returns the number of pages which have successfully been scheduled for writeback, which we accumulate and ultimately return to the caller.

This function is rate-limited (in a similar fashion to `writeback_sb_inodes()`) to a hardcoded maximum one tenth of a second interval between iterations, assuming at least 1 page has been written back (i.e. we process all inodes until we make at least 1 page's progress). Additionally we manually check to see if the work task has been completed, at which point we exit early.

This allows for the function to exit to its caller on a regular basis to see if any conditions have arisen which would terminate the writeback.

Now we have examined the functions which ultimately invoke `writeback_sb_inodes()`, it's time to examine this function in detail, which we do in Listing 10-57 (eliding irrelevant tracing hooks and realtime scheduler rescheduling logic).

```

1763 /*
1764  * Write a portion of b_io inodes which belong to @sb.
1765  *
1766  * Return the number of pages and/or inodes written.
1767  *
1768  * NOTE! This is called with wb->list_lock held, and will
1769  * unlock and relock that for each inode it ends up doing
1770  * IO for.
1771  */
1772 static long writeback_sb_inodes(struct super_block *sb,
1773                                struct bdi_writeback *wb,
1774                                struct wb_writeback_work *work)
1775 {
1776     struct writeback_control wbc = {
1777         .sync_mode           = work->sync_mode,
1778         .tagged_writepages   = work->tagged_writepages,
1779         .for_kupdate         = work->for_kupdate,
1780         .for_background      = work->for_background,
1781         .for_sync            = work->for_sync,
1782         .range_cyclic        = work->range_cyclic,
1783         .range_start         = 0,
1784         .range_end           = LLONG_MAX,
1785     };
1786     unsigned long start_time = jiffies;

```

```

1787     long write_chunk;
1788     long total_wrote = 0; /* count both pages and inodes */

```

Listing 10-57: fs/fs-writeback.c: writeback_sb_inodes() preface

The writeback core, as described in Section 10.10 uses the struct `writeback_control` object to manage the process (explored in Listing 10-24), which we must generate based on the struct `wb_writeback_work` argument.

We *always* specify the entire range of the dirty inodes by setting `range_start` and `range_end` appropriately. For writeback spanning only a subset of folios belonging to an inode, file writeback by `fsync()` and related APIs must be used (see Section 10.13).

At this stage we do not specify the `nr_to_write` parameter, which we will specify later on in the function based on an empirically determined ‘chunk size’ which we will discuss below.

We note the start time using the global `jiffies` value before proceeding with the core writeback loop beginning in Listing 10-58.

```

1790     while (!list_empty(&wb->b_io)) {
1791         struct inode *inode = wb_inode(wb->b_io.prev);
1792         struct bdi_writeback *tmp_wb;
1793         long wrote;
1794
1795         if (inode->i_sb != sb) {
1796             if (work->sb) {
1797                 /*
1798                  * We only want to write back data for this
1799                  * superblock, move all inodes not belonging
1800                  * to it back onto the dirty list.
1801                  */
1802                 redirty_tail(inode, wb);
1803                 continue;
1804             }
1805
1806             /*
1807              * The inode belongs to a different superblock.
1808              * Bounce back to the caller to unpin this and
1809              * pin the next superblock.
1810              */
1811             break;
1812         }
1813
1814         /*
1815          * Don't bother with new inodes or inodes being freed, first
1816          * kind does not need periodic writeout yet, and for the
1817          * latter
1818          * kind writeout is handled by the freer.
1819          */

```

```

1819         spin_lock(&inode->i_lock);
1820         if (inode->i_state & (I_NEW | I_FREEING | I_WILL_FREE)) {
1821             redirty_tail_locked(inode, wb);
1822             spin_unlock(&inode->i_lock);
1823             continue;
1824         }
1825         if ((inode->i_state & I_SYNC) && wbc.sync_mode != WB_SYNC_ALL)
1826             {
1827                 /*
1828                  * If this inode is locked for writeback and we are
1829                  not
1830                  * doing writeback-for-data-integrity, move it to
1831                  * b_more_io so that writeback can proceed with the
1832                  * other inodes on s_io.
1833                  *
1834                  * We'll have another go at writing back this inode
1835                  * when we completed a full scan of b_io.
1836                  */
1837                 requeue_io(inode, wb);
1838                 spin_unlock(&inode->i_lock);
1839                 :
1840                 continue;
1841             }
1842         spin_unlock(&wb->list_lock);
1843         /*
1844          * We already requeued the inode if it had I_SYNC set and we
1845          * are doing WB_SYNC_NONE writeback. So this catches only the
1846          * WB_SYNC_ALL case.
1847          */
1848         if (inode->i_state & I_SYNC) {
1849             /* Wait for I_SYNC. This function drops i_lock... */
1850             inode_sleep_on_writeback(inode);
1851             /* Inode may be gone, start again */
1852             spin_lock(&wb->list_lock);
1853             continue;
1854         }

```

Listing 10-58: fs/fs-writeback.c: writeback_sb_inodes() initial checks

We iterate through each struct `inode` in the struct `bdi_writeback->b_io` list, moving *backwards* through the list, therefore acquiring the eldest inode first which is the appropriate ordering as the intent is to minimise how out of date the on-disk representation of the data is in comparison to its state in the page cache.

We then perform a series of initial checks, firstly ensuring that the inode belongs to the super block. This is important, as we enter this function with the super block locked against unmount, and therefore absolutely must limit ourselves to processing inodes which belong to specified superblock.

In order to put this check in context we examine from where `writeback_sb_inodes()` is called. This is either by either `__writeback_inodes_wb()` (see Listing 10-56) or `wb_writeback()` (see Listing 10-55).

In the former case, `__writeback_inodes_wb()` is invoked either by `writeback_inodes_wb()` which specifies a struct `wb_writeback_work` object without `sb` specified and iterates through inodes backwards invoking `writeback_sb_inodes()` specifying a super block of struct `inode->i_sb`, or via `wb_writeback()` again only when no struct `wb_writeback_work->sb` is specified.

When we queue inodes via `move_expired_inodes()` (see Listing 10-47, note this function is invoked by `queue_io()` shown in Listing 10-46) we ensure that inodes are grouped by superblock.

Therefore, in the case of writeback arising via `__writeback_inodes_wb()`, the test for `work->sb` simply does not apply, and this check simply breaks out of the loop (thereby exiting the function) which causes `__writeback_inodes_wb()` to iterate to the next superblock which is locked against unmount.

The remaining case is an invocation from `wb_writeback()` (see Listing 10-55) where struct `wb_writeback_work->sb` is specified.

The struct `wb_writeback_work->sb` parameter is set in two places—`__writeback_inodes_sb_nr()` (see Listing 10-42) and `sync_inodes_sb()` (see Listing 10-39), each of which are precipitated by a `sync()` or `syncfs()` system call, the former iterating through *all* superblocks and the later limiting itself to a single one.

We can observe how these functions queue work and wake up the writeback flusher threads in Figure 10-7.

In each of these instances, when queueing work via `queue_io()`, inodes which belong to another superblock may indeed be queued for writeback, which we simply append to the the struct `bdi_writeback->b_dirty` list.

Since we can rely on inodes being partitioned by superblock, by simply continuing the loop here each time we encounter an irrelevant inode, we will eventually reach the inodes which need writeback.

Since we keep ‘redirtying’ the inodes, these will eventually get picked up again. If they are inodes belonging to a synchronisation-specified struct `wb_writeback_work` task found in struct `bdi_writeback->work_list`, these will simply be scheduled to be processed later by `wb_workfn()` (see Listing 10-44), if they are specific to background or kupdate flushing, these are processed *last* by `wb_workfn()` so will get queued later. We can see how this logic fits into `wb_writeback()` in Figure 10-10.

The ‘redirtying’ process is performed by `redirty_tail()` shown in Listing 10-59.

```

1308 static void redirty_tail(struct inode *inode, struct bdi_writeback *wb)
1309 {
1310     spin_lock(&inode->i_lock);
1311     redirty_tail_locked(inode, wb);
1312     spin_unlock(&inode->i_lock);
1313 }
```

Listing 10-59: fs/fs-writeback.c: redirty_tail()

This locks the inode and defers the actual redirtying to `redirty_tail_locked()` shown in Listing 10-60.

```

1284 /*
1285  * Redirty an inode: set its when-it-was dirtied timestamp and move it to the
1286  * furthest end of its superblock's dirty-inode list.
1287  *
1288  * Before stamping the inode's ->dirtyed_when, we check to see whether it is
1289  * already the most-recently-dirtied inode on the b_dirty list. If that is
1290  * the case then the inode must have been redirtied while it was being written
1291  * out and we don't reset its dirtyed_when.
1292  */
1293 static void redirty_tail_locked(struct inode *inode, struct bdi_writeback *wb)
1294 {
1295     assert_spin_locked(&inode->i_lock);
1296
1297     if (!list_empty(&wb->b_dirty)) {
1298         struct inode *tail;
1299
1300         tail = wb_inode(wb->b_dirty.next);
1301         if (time_before(inode->dirtyed_when, tail->dirtyed_when))
1302             inode->dirtyed_when = jiffies;
1303     }
1304     inode_io_list_move_locked(inode, wb, &wb->b_dirty);
1305     inode->i_state &= ~I_SYNC_QUEUED;
1306 }

```

Listing 10-60: fs/fs-writeback.c: `redirty_tail_locked()`

This ultimately moves the inode to the tail of the `struct bdi_writeback->b_dirty` list using `inode_io_list_move_locked()`, clearing its `I_SYNC_QUEUED` flag to indicate that the inode is no longer queued for writeback (a flag that is set by `move_expired_inodes()`, shown in Listing 10-47).

NOTE

Despite `redirty_tail_locked()`'s name, the `inode_io_list_move_locked()` function ultimately invokes `list_move()` rather than `list_move_tail()`, meaning that the inode is actually placed at the head of `struct bdi_writeback->b_dirty` rather than its tail. However, since we process inodes backwards, the inode is thereby placed at the back of this list, so is conceptually indeed its tail.

We have to be careful about the inode's timestamp here, it is vital to maintain the ordering of inodes on `struct bdi_writeback->b_dirty` in age order, so before placing the inode at the head of this list, we examine the existing head. If it is *older* than this, which was the previously youngest entry in this list, we restamp it to the current time.

Returning to the `writeback_sb_inodes()` initial check logic we were examining in Listing 10-58, we next acquire the `struct inode->i_lock` as we wish to keep the inode stable as we examine it for subsequent state checks.

We check whether the inode is either new, freeing or soon to be freed. As the comment suggests, none of these indicate writeout is currently required. We redirty the inode using `redirty_tail_locked()` directly as we already hold the inode's lock, unlock it and continue on to the next inode.

Finally, we check the case where the inode is already being processed for writeback. This can occur if `writeback_single_inode()` has been triggered to immediately writeback an inode.

We don't examine `writeback_single_inode()` for the sake of brevity, but broadly speaking it mimics `writeback_sb_inodes()` only explicitly writing only a single inode with a user-specified struct `writeback_control` object.

inodes are serialised between these calls using the `I_SYNC` state flag, which is set immediately after this check is successfully completed.

There are two variants of this case we must consider—where synchronisation is required, i.e. where completion of writeback implies that the data is actually written back to disk rather than simply scheduled to be written back, and where this is not required. This is indicated by the struct `wb_writeback_work->sync_mode` being set to the enum `writeback_sync_mode` value `WB_SYNC_ALL`.

In the case where it is *not* required, we can simply requeue the inode for later processing by placing it on the struct `bdi_writeback->b_more_io` list using `requeue_io()` which very simply moves the inode to this list as shown in Listing ??.

```

1315 /*
1316  * requeue inode for re-scanning after bdi->b_io list is exhausted.
1317  */
1318 static void requeue_io(struct inode *inode, struct bdi_writeback *wb)
1319 {
1320     inode_io_list_move_locked(inode, wb, &wb->b_more_io);
1321 }

```

Listing 10-61: fs/fs-writeback.c: requeue_io()

You can observe how the struct `bdi_writeback->b_more_io` list is processed in `wb_writeback()` (see Listing 10-55) in Figure 10-10. Note that all entries in this list are immediately queued on the next invocation of `queue_io()` (see Listing 10-46).

If we do require sync however, we do not move the inode, but rather *wait* for the writeback process to complete via `inode_sleep_on_writeback()`.

Once these checks are complete, we are ready to move on with scheduling the actual writeback in `writeback_sb_inodes()` which we examine in Listing 10-62.

```

1854     inode->i_state |= I_SYNC;
1855     wbc_attach_and_unlock_inode(&wbc, inode);
1856
1857     write_chunk = writeback_chunk_size(wb, work);
1858     wbc.nr_to_write = write_chunk;
1859     wbc.pages_skipped = 0;

```

```

1860
1861      /*
1862       * We use I_SYNC to pin the inode in memory. While it is set
1863       * evict_inode() will wait so the inode cannot be freed.
1864       */
1865      __writeback_single_inode(inode, &wbc);
1866
1867      wbc_detach_inode(&wbc);
1868      work->nr_pages -= write_chunk - wbc.nr_to_write;
1869      wrote = write_chunk - wbc.nr_to_write - wbc.pages_skipped;
1870      wrote = wrote < 0 ? 0 : wrote;
1871      total_wrote += wrote;
1872      :

```

Listing 10-62: fs/fs-writeback.c: writeback_sb_inodes() writeback

We start by marking the inode being processed for writeback via `I_SYNC`, preventing any other part of the kernel from simultaneously attempting writeback (i.e. `writeback_single_inode()`) nor attempt to evict the inode meta-data from memory.

The function invokes `wbc_attach-and_unlock_inode()` which is out of scope for our discussion here as it is specific to cgroup-moderated writeback, however note that it as the function name suggests drops the inode lock.

It's at this point we determine how many pages to write back via `writeback_chunk_size()`. This determines how many pages to attempt to write-back in one operation. Note that for writeback work which is either synchronised (i.e. `struct wb_writeback_work->sync_mode` is set to `WB_SYNC_ALL`) or those tasks which have `struct wb_writeback_work->tagged_writeback` set, this chunking doesn't apply (this applies to `__writeback_inodes_sb_nr()` and `sync_inodes_wb()`, both of which are invoked by `sync()` and/or `syncfs()` system calls).

We defer the description of `writeback_chunk_size()` to Section 10.14 in Listing 10-102 as this relates directly to dirty throttling logic within the kernel.

Again the `wbc_detach_inode()` function relates to cgroup writeback functionality and thus is out of scope for the book. Finally the actual writeback is executed in `__writeback_single_inode()`, which we examine below in Listing 10-67.

After the writeback has been kicked off, statistics are updated accordingly—`struct wb_writeback_work->nr_pages` is decremented by the number of pages written back or skipped and the total actually written pages is added to the `total_wrote` counter which is ultimately returned to the caller.

Finally, we examine the post-writeback handling at the end of the core loop in Listing 10-63.

```

1886      /*
1887       * Requeue @inode if still dirty. Be careful as @inode may
1888       * have been switched to another wb in the meantime.
1889       */
1890      tmp_wb = inode_to_wb_and_lock_list(inode);

```

```

1891         spin_lock(&inode->i_lock);
1892         if (!(inode->i_state & I_DIRTY_ALL))
1893             total_wrote++;
1894         requeue_inode(inode, tmp_wb, &wbc);
1895         inode_sync_complete(inode);
1896         spin_unlock(&inode->i_lock);
1897
1898         if (unlikely(tmp_wb != wb)) {
1899             spin_unlock(&tmp_wb->list_lock);
1900             spin_lock(&wb->list_lock);
1901         }
1902
1903         /*
1904          * bail out to wb_writeback() often enough to check
1905          * background threshold and other termination conditions.
1906          */
1907         if (total_wrote) {
1908             if (time_is_before_jiffies(start_time + HZ / 10UL))
1909                 break;
1910             if (work->nr_pages <= 0)
1911                 break;
1912         }
1913     }
1914     return total_wrote;
1915 }

```

Listing 10-63: fs/fs-writeback.c: writeback_sb_inodes() suffix

We start by looking up the current struct `bdi_writeback` object associated with the inode. This might have changed if cgroup-mediate writeback is in place which we take care to account for (though discussion of this is out of scope for the book). We also increment `total_wrote` if the inode metadata itself was written back.

There are two key functions which are invoked at this point—`requeue_inode()`, which determines whether the inode needs to be requeued or writeback is complete for it, which we examine in Listing 10-64 and `inode_sync_complete()`, whose core task is to clear the inode's `I_SYNC` flag, waking up anything that was waiting on it to be cleared.

After this is done, the function checks to determine if the hard-coded tenth of a second interval period has expired since the previously stored start time of the operation, in which case we break out of the loop and return to `wb_writeback()` (see Listing 10-55) to proceed as shown in Figure 10-10, in case conditions have arisen which would result in termination of the writeback.

The requeue logic in `requeue_inode()` crucially checks whether the inode was completely written out, whether work remains and where to place the inode. We examine it in Listing 10-64 (eliding out of scope dirty time handling).

```

1497 /*
1498  * Find proper writeback list for the inode depending on its current state and
1499  * possibly also change of its state while we were doing writeback. Here we
1500  * handle things such as livelock prevention or fairness of writeback among
1501  * inodes. This function can be called only by flusher thread - noone else
1502  * processes all inodes in writeback lists and requeueing inodes behind
1503  * flusher
1504  * thread's back can have unexpected consequences.
1505  */
1506 static void requeue_inode(struct inode *inode, struct bdi_writeback *wb,
1507                          struct writeback_control *wbc)
1508 {
1509     if (inode->i_state & I_FREEING)
1510         return;
1511
1512     /*
1513      * Sync livelock prevention. Each inode is tagged and synced in one
1514      * shot. If still dirty, it will be redirty_tail()'ed below. Update
1515      * the dirty time to prevent enqueue and sync it again.
1516      */
1517     if ((inode->i_state & I_DIRTY) &&
1518         (wbc->sync_mode == WB_SYNC_ALL || wbc->tagged_writepages))
1519         inode->dirty_when = jiffies;
1520
1521     if (wbc->pages_skipped) {
1522         /*
1523          * writeback is not making progress due to locked
1524          * buffers. Skip this inode for now.
1525          */
1526         redirty_tail_locked(inode, wb);
1527         return;
1528     }
1529
1530     if (mapping_tagged(inode->i_mapping, PAGECACHE_TAG_DIRTY)) {
1531         /*
1532          * We didn't write back all the pages. nfs_writepages()
1533          * sometimes bales out without doing anything.
1534          */
1535         if (wbc->nr_to_write <= 0) {
1536             /* Slice used up. Queue for next turn. */
1537             requeue_io(inode, wb);
1538         } else {
1539             /*
1540              * Writeback blocked by something other than
1541              * congestion. Delay the inode for some time to
1542              * avoid spinning on the CPU (100% iowait)
1543              * retrying writeback of the dirty page/inode

```

```

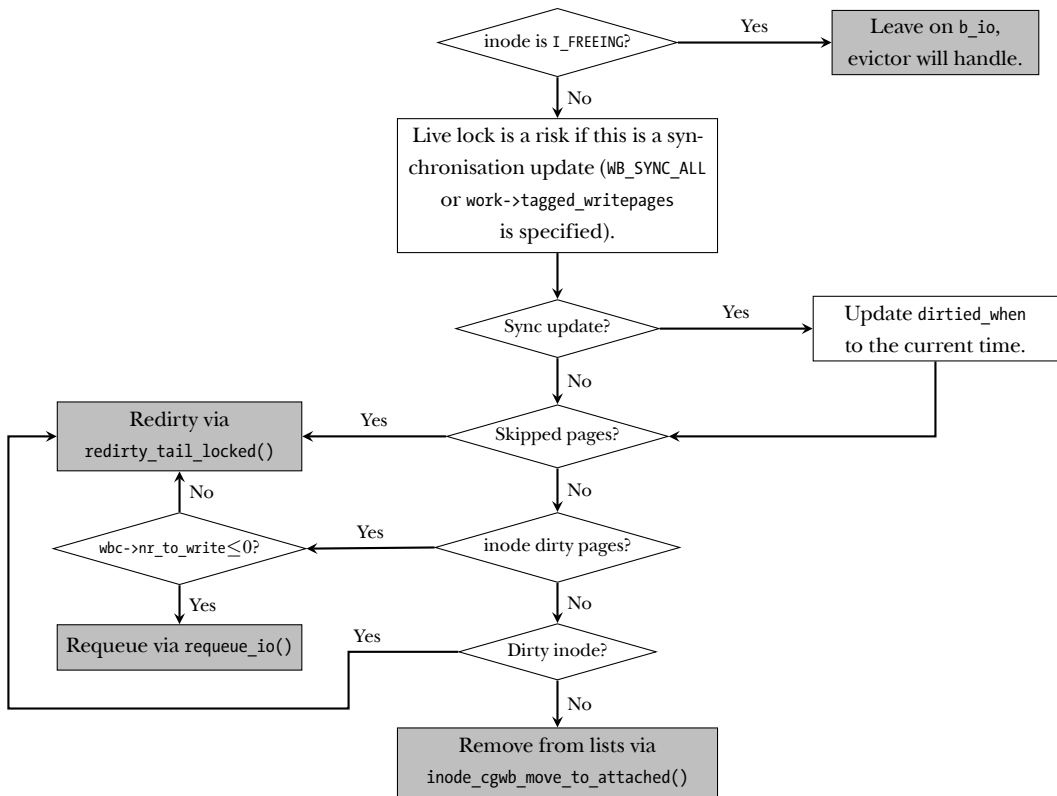
1543             * that cannot be performed immediately.
1544             */
1545             redirty_tail_locked(inode, wb);
1546         }
1547     } else if (inode->i_state & I_DIRTY) {
1548         /*
1549          * Filesystems can dirty the inode during writeback operations
1550          * such as delayed allocation during submission or metadata
1551          * updates after data IO completion.
1552          */
1553         redirty_tail_locked(inode, wb);
1554         :
1555     } else {
1556         /* The inode is clean. Remove from writeback lists. */
1557         inode_cgwb_move_to_attached(inode, wb);
1558     }
1559 }

```

Listing 10-64: fs/fs-writeback.c: `requeue_inode()`

This processes each inode, determining whether to place it back on the struct `bdi_writeback->b_dirty` list via `redirty_tail_locked()` (see Listing 10-60), requeue it onto the struct `bdi_writeback->b_more_io` list via `requeue_io()` (see Listing ??) or remove it from these lists altogether via `inode_cgwb_move_to_attached()` (see Listing 10-65).

We explore the logic of this function in Figure 10-11.

Figure 10-11: `reqqueue_inode()` Logic

We start by checking whether the inode is currently being freed, if so we need do nothing as this will eventually be evicted. In addition logic elsewhere in the writeback code will skip freeing inodes so it has no material impact on writeback.

We then cover a special case where we try to avoid live-lock when performing synchronisation writeback, where inodes may be constantly redirtied and processed again. We avoid this by resetting the struct `inode->dirtied_when` timestamp to now, which causes those inodes to not be queued again on the next loop in `wb_writeback()` (see Listing 10-55), as shown in Figure 10-10.

Next we determine if any pages were skipped during writeback due to the filesystem declining to perform writeback, a count of which is maintained in struct `writeback_control->pages_skipped`. If so, then we cannot make further progress on this inode at this time (perhaps requiring locks to be released before proceeding), and therefore we *redirty* the inode in question via `redirty_tail_locked()` (see Listing 10-60).

We then consider cases where the inode is still dirty—in this instance we *must* requeue the inode as it has not yet completed its writeback I/O.

We start by determining if *any* inode folios are still dirty using the xarray associated with the inode's struct `address_space` via `mapping_tagged()`. If so, then we proceed on the basis that dirty *data* remains.

In this case, where we have simply written at or beyond the current specified chunk size, we requeue the I/O for a subsequent run via `requeue_io()` (see Listing ??).

If it was not the ‘slice’ of data being insufficient to writeback the entire inode, we take the safe route and simply redirty the inode via `redirty_tail_locked()`.

If no folio associated with the inode is dirty but the inode is still marked as such, then this indicates that inode metadata itself requires writeback so in this instance we also redirty via `redirty_tail_locked()`.

Finally, if no requeueing is deemed to be required, we remove the inode from all lists using `inode_cgwb_move_to_attached()`, as shown in Listing 10-65.

```

270 /**
271  * inode_cgwb_move_to_attached - put the inode onto wb->b_attached list
272  * @inode: inode of interest with i_lock held
273  * @wb: target bdi_writeback
274  *
275  * Remove the inode from wb's io lists and if necessarily put onto b_attached
276  * list. Only inodes attached to cgwb's are kept on this list.
277  */
278 static void inode_cgwb_move_to_attached(struct inode *inode,
279                                         struct bdi_writeback *wb)
280 {
281     assert_spin_locked(&wb->list_lock);
282     assert_spin_locked(&inode->i_lock);
283
284     inode->i_state &= ~I_SYNC_QUEUED;
285     if (wb != &wb->bdi->wb)
286         list_move(&inode->i_io_list, &wb->b_attached);
287     else
288         list_del_init(&inode->i_io_list);
289     wb_io_lists_depopulated(wb);
290 }
```

Listing 10-65: fs/fs-writeback.c: `inode_cgwb_move_to_attached()`

This function starts by clearing the inode’s `I_SYNC_QUEUED` flag to indicate that it is no longer on any queue. The move logic is specific to cgroup writeback where the inode’s struct `bdi_writeback` may have changed. This is out of scope for the book, so we rather consider the case where it is simply removed from the list on which it resides and has its list node reset by `list_del_init()`.

This also invokes `wb_io_lists_depopulated()`, which does the inverse of `wb_io_lists_populated()` (see Listing 10-48) and clears the `WB_has_dirty_io` flag in the struct `bdi_writeback->state` field, as shown in Listing 10-66.

```

98 static void wb_io_lists_depopulated(struct bdi_writeback *wb)
99 {
100     if (wb_has_dirty_io(wb) && list_empty(&wb->b_dirty) &&
```

```

101         list_empty(&wb->b_io) && list_empty(&wb->b_more_io)) {
102             clear_bit(WB_has_dirty_io, &wb->state);
103             WARN_ON_ONCE(atomic_long_sub_return(wb->avg_write_bandwidth,
104                                                 &wb->bdi->tot_write_bandwidth) < 0);
105         }
106     }

```

Listing 10-66: fs/fs-writeback.c: wb_io_lists_depopulated()

This resets the WB_has_dirty_io flag if all struct bdi_writeback lists are empty, and updates bandwidth statistics.

Returning to writeback_sb_inodes(), specifically the portion which invokes writeback as shown in Listing 10-62, this invokes __writeback_single_inode() which invokes the actual writeback, which we examine in Listing 10-67 (eliding irrelevant trace hook points and out of scope dirty time handling and fscache pinning logic).

```

1564  /*
1565   * Write out an inode and its dirty pages (or some of its dirty pages,
1566   *   depending
1567   * on @wbc->nr_to_write), and clear the relevant dirty flags from i_state.
1568   * This doesn't remove the inode from the writeback list it is on, except
1569   * potentially to move it from b_dirty_time to b_dirty due to timestamp
1570   * expiration. The caller is otherwise responsible for writeback list
1571   *   handling.
1572   * The caller is also responsible for setting the I_SYNC flag beforehand and
1573   * calling inode_sync_complete() to clear it afterwards.
1574   */
1575  static int
1576  __writeback_single_inode(struct inode *inode, struct writeback_control *wbc)
1577  {
1578      struct address_space *mapping = inode->i_mapping;
1579      long nr_to_write = wbc->nr_to_write;
1580      unsigned dirty;
1581      int ret;
1582
1583      WARN_ON(!(inode->i_state & I_SYNC));
1584      :
1587      ret = do_writepages(mapping, wbc);
1588
1589      /*
1590       * Make sure to wait on the data before writing out the metadata.
1591       * This is important for filesystems that modify metadata on data
1592       * I/O completion. We don't do it for sync(2) writeback because it has
1593       *   a
1594       * separate, external IO completion path and ->sync_fs for
1595       *   guaranteeing

```

```

1594     * inode metadata is written back correctly.
1595     */
1596     if (wbc->sync_mode == WB_SYNC_ALL && !wbc->for_sync) {
1597         int err = filemap_fdatawait(mapping);
1598         if (ret == 0)
1599             ret = err;
1600     }
1601     :
1615     /*
1616     * Get and clear the dirty flags from i_state. This needs to be done
1617     * after calling writepages because some filesystems may redirty the
1618     * inode during writepages due to delalloc. It also needs to be done
1619     * after handling timestamp expiration, as that may dirty the inode
1620     too.
1621     */
1622     spin_lock(&inode->i_lock);
1623     dirty = inode->i_state & I_DIRTY;
1624     inode->i_state &= ~dirty;
1625     /*
1626     * Paired with smp_mb() in __mark_inode_dirty(). This allows
1627     * __mark_inode_dirty() to test i_state without grabbing i_lock -
1628     * either they see the I_DIRTY bits cleared or we see the dirtied
1629     * inode.
1630     *
1631     * I_DIRTY_PAGES is always cleared together above even if @mapping
1632     * still has dirty pages. The flag is reinstated after smp_mb() if
1633     * necessary. This guarantees that either __mark_inode_dirty()
1634     * sees clear I_DIRTY_PAGES or we see PAGECACHE_TAG_DIRTY.
1635     */
1636     smp_mb();
1637
1638     if (mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
1639         inode->i_state |= I_DIRTY_PAGES;
1640     :
1648     spin_unlock(&inode->i_lock);
1649
1650     /* Don't write the inode if only I_DIRTY_PAGES was set */
1651     if (dirty & ~I_DIRTY_PAGES) {
1652         int err = write_inode(inode, wbc);
1653         if (ret == 0)
1654             ret = err;
1655     }
1656     :
1658     return ret;
1659 }

```

Listing 10-67: fs/fs-writeback.c: `__writeback_single_inode()`

This function immediately invokes `do_writepages()` shown in Listing 10-23 and Section 10.10 which performs the actual writeback. The remainder of the function performs housekeeping tasks after completion.

We next cover the case of synchronised writeback which has not originated from a synchronisation system call such as `sync()` or `syncfs()`. In those cases, waiting is handled by the calling functions (`writeback_inodes_sb()`, see Listing 10-40 and `sync_inodes_sb()`, see Listing 10-39), so in these instances `__writeback_single_inode()` need not perform the wait. For other synchronising cases, this is not the case and we must explicitly wait, which is achieved via `filemap_fdatawait()` which we examine in the subsequent file writeback Section 10.13 in Listing 10-72.

In all invocations of this function, the dirty flags of the inode are cleared, regardless of the outcome of the operation, under `struct inode->i_lock`.

If we subsequently discover that at least one page cache folio containing the data of the inode remains dirty, we update the inode state regardless, with a careful locking negotiation, pairing a memory barrier with `__mark_inode_dirty()` (see Listing 10-2) to permit the latter to check `struct inode->i_state` without a lock.

Finally, if the inode metadata is dirty, the inode is written back via `write_inode()`, however this is out of scope for the book.

10.13 File Writeback

We've seen how writeback proceeds at a *file system* granularity, but there remains another means by which writeback can proceed—at a *file* granularity.

Users can request the synchronisation of an open file descriptor to disk via the `fsync()` or `fdatasync()` system calls and the synchronisation of individual pages of a `mmap()` memory-mapped file via `msync()`.

NOTE

Users can also specify the synchronisation of individual pages of an open file descriptor via `sync_file_range()`, however the use of this system call is heavily discouraged as dangerous, so we don't examine it in detail. This system call ultimately invokes the same functions that we examine here in any case.

In Section 10.6 and Figure 10-4 we examine these synchronisation methods, each of which ultimately invoke the kernel functions `filemap_flush()` (see Listing 10-68), `file_write_and_wait_range()` (see Listing 10-74) and `file_write_and_wait_range()` (see Listing 10-74). In addition, we've observed that `__writeback_single_inode()` (see Listing 10-67) invokes `filemap_fdatawait()` (see Listing 10-72).

We will examine each of these functions, but the key point to observe here is that ultimately all of these functions and any others which perform writeback on a file granularity ultimately invoke two core functions—`__filemap_fdatawrite_range()` and `__filemap_fdatawait_range()`.

We observe how each of the aforementioned calls ultimately end up calling this pair functions in Figure 10-12.

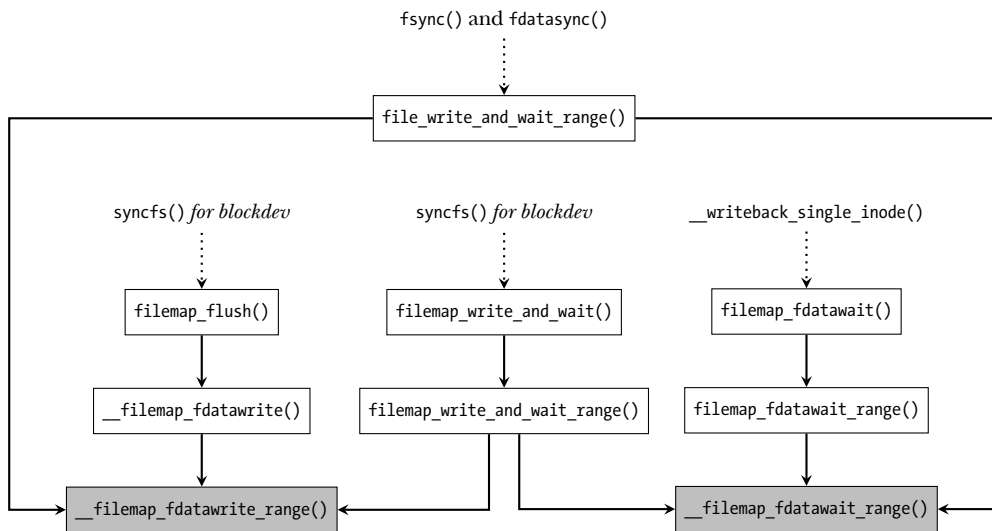


Figure 10-12: File Writeback Call Stack

Let's examine each of the functions referenced here, starting with `filemap_flush()` in Listing 10-68.

```

443 /**
444  * filemap_flush - mostly a non-blocking flush
445  * @mapping:    target address_space
446  *
447  * This is a mostly non-blocking flush. Not suitable for data-integrity
448  * purposes - I/O may not be started against all dirty pages.
449  *
450  * Return: %0 on success, negative error code otherwise.
451  */
452 int filemap_flush(struct address_space *mapping)
453 {
454     return __filemap_fdatawrite(mapping, WB_SYNC_NONE);
455 }
  
```

Listing 10-68: mm/filemap.c: `filemap_flush()`

This function simply defers its operation to `__filemap_fdatawrite()`, specifying `WB_SYNC_NONE` to indicate that no synchronisation should be applied. We examine this in Listing 10-69.

```

424 static inline int __filemap_fdatawrite(struct address_space *mapping,
425     int sync_mode)
426 {
427     return __filemap_fdatawrite_range(mapping, 0, LLONG_MAX, sync_mode);
428 }
  
```

Listing 10-69: mm/filemap.c: `__filemap_fdatawrite()`

This simply defers its operation to the core function `__filemap_fdatawrite`, specifying that the entire file be written back. We examine this in Listing 10-69.

Next we examine `filemap_write_and_wait()` in ??.

```

58 static inline int filemap_write_and_wait(struct address_space *mapping)
59 {
60     return filemap_write_and_wait_range(mapping, 0, LLONG_MAX);
61 }

```

Listing 10-70: /include/linux/pagemap.h: filemap_write_and_wait()

This simply defers the heavy lifting to `filemap_write_and_wait_range()`, specifying the entire range of the Listing 10-71.

```

654 **
655 * filemap_write_and_wait_range - write out & wait on a file range
656 * @mapping:    the address_space for the pages
657 * @lstart:     offset in bytes where the range starts
658 * @lend:       offset in bytes where the range ends (inclusive)
659 *
660 * Write out and wait upon file offsets lstart->lend, inclusive.
661 *
662 * Note that @lend is inclusive (describes the last byte to be written) so
663 * that this function can be used to write to the very end-of-file (end = -1).
664 *
665 * Return: error status of the address space.
666 */
667 int filemap_write_and_wait_range(struct address_space *mapping,
668                                loff_t lstart, loff_t lend)
669 {
670     int err = 0, err2;
671
672     if (mapping_needs_writeback(mapping)) {
673         err = __filemap_fdatawrite_range(mapping, lstart, lend,
674                                         WB_SYNC_ALL);
675         /*
676          * Even if the above returned error, the pages may be
677          * written partially (e.g. -ENOSPC), so we wait for it.
678          * But the -EIO is special case, it may indicate the worst
679          * thing (e.g. bug) happened, so we avoid waiting for it.
680          */
681         if (err != -EIO)
682             __filemap_fdatawait_range(mapping, lstart, lend);
683     }
684     err2 = filemap_check_errors(mapping);
685     if (!err)
686         err = err2;
687     return err;

```

688 }

Listing 10-71: mm/filemap.c: filemap_write_and_wait_range()

This uses `mapping_needs_writeback()` to determine whether to proceed which simply checks whether `struct address_space->npages` is non-zero. This field indicates how many pages are present in the page cache for this page cache entry, if none are present then writeback can not possibly proceed and therefore there is no point in trying to do so.

After this, the actual writing is performed by `__filemap_fdatawrite_range()` (see Listing 10-75) specifying `WB_SYNC_ALL` to indicate that all writes will be synchronised.

Other than in the case where an EIO error arises (indicating an error which would have prevented writeback generally), we then wait for the operation to complete via `__filemap_fdatawait_range()` (shown in Listing 10-77).

This seems redundant, as we have already specified `WB_SYNC_ALL`, however this flag only results in waiting on writeback in the file system writeback logic described above, for file writeback, we must perform the waiting ourselves and this flag simply provides the context to underlying logic that the operation will proceed synchronously.

Finally this checks for errors that occurred during writeback that will have been placed in the `struct address_space->flags` field. This is done via `filemap_check_errors()`, which clears these flags and reports the relevant errors (prioritising ones that were returned directly by `__filemap_fdatawrite_range()`).

Next we examine `filemap_fdatawait()` in Listing 10-72.

```
41 static inline int filemap_fdatawait(struct address_space *mapping)
42 {
43     return filemap_fdatawait_range(mapping, 0, LLONG_MAX);
44 }
```

Listing 10-72: include/linux/pagemap.h: filemap_fdatawait()

This defers its operation to `filemap_fdatawait_range()`, specifying that the writeback for the entire file should be waited upon. This is shown in Listing 10-73.

```
532 /**
533  * filemap_fdatawait_range - wait for writeback to complete
534  * @mapping:          address space structure to wait for
535  * @start_byte:       offset in bytes where the range starts
536  * @end_byte:         offset in bytes where the range ends (inclusive)
537  *
538  * Walk the list of under-writeback pages of the given address space
539  * in the given range and wait for all of them. Check error status of
540  * the address space and return it.
541  *
542  * Since the error status of the address space is cleared by this function,
543  * callers are responsible for checking the return value and handling and/or
```

```

544  * reporting the error.
545  *
546  * Return: error status of the address space.
547  */
548  int filemap_fdatawait_range(struct address_space *mapping, loff_t start_byte,
549                             loff_t end_byte)
550  {
551      __filemap_fdatawait_range(mapping, start_byte, end_byte);
552      return filemap_check_errors(mapping);
553  }

```

Listing 10-73: mm/filemap.c: filemap_fdatawait_range()

This defers the heavy lifting to `__filemap_fdatawait_range()` (shown in Listing 10-77), before running `filemap_check_errors()` to pick up any errors attached to the `struct address_space` object.

Finally, we examine `file_write_and_wait_range()`, shown in Listing 10-74.

```

751  /**
752  * file_write_and_wait_range - write out & wait on a file range
753  * @file:      file pointing to address_space with pages
754  * @lstart:    offset in bytes where the range starts
755  * @lend:      offset in bytes where the range ends (inclusive)
756  *
757  * Write out and wait upon file offsets lstart->lend, inclusive.
758  *
759  * Note that @lend is inclusive (describes the last byte to be written) so
760  * that this function can be used to write to the very end-of-file (end = -1).
761  *
762  * After writing out and waiting on the data, we check and advance the
763  * f_wb_err cursor to the latest value, and return any errors detected there.
764  *
765  * Return: %0 on success, negative error code otherwise.
766  */
767  int file_write_and_wait_range(struct file *file, loff_t lstart, loff_t lend)
768  {
769      int err = 0, err2;
770      struct address_space *mapping = file->f_mapping;
771
772      if (mapping_needs_writeback(mapping)) {
773          err = __filemap_fdatawrite_range(mapping, lstart, lend,
774                                           WB_SYNC_ALL);
775          /* See comment of filemap_write_and_wait() */
776          if (err != -EIO)
777              __filemap_fdatawait_range(mapping, lstart, lend);
778      }
779      err2 = file_check_and_advance_wb_err(file);
780      if (!err)
781          err = err2;

```



```

782     return err;
783 }

```

Listing 10-74: mm/filemap.c: file_write_and_wait_range()

This defers the write and wait to `__filemap_fdatawrite_range()` (see Listing 10-75) and `__filemap_fdatawait_range()` (see Listing 10-77) similar to `filemap_write_and_wait_range()` shown in Listing 10-71 above. However a key difference here is that this function invokes `file_check_and_advance_wb_err()` rather than `filemap_check_errors()`. This keeps track of the last write error at a struct file granularity. For brevity we will omit a close examination of this logic.

Now we have examined the core means by which we reach `__filemap_fdatawrite_range()` and `__filemap_fdatawait_range()`, we will examine these functions in detail.

We begin by examining `__filemap_fdatawrite_range()` in Listing 10-75.

```

394 /**
395  * __filemap_fdatawrite_range - start writeback on mapping dirty pages in
396  *   range
397  * @mapping:   address space structure to write
398  * @start:     offset in bytes where the range starts
399  * @end:       offset in bytes where the range ends (inclusive)
400  * @sync_mode: enable synchronous operation
401  *
402  * Start writeback against all of a mapping's dirty pages that lie
403  * within the byte offsets <start, end> inclusive.
404  *
405  * If sync_mode is WB_SYNC_ALL then this is a "data integrity" operation, as
406  * opposed to a regular memory cleansing writeback. The difference between
407  * these two operations is that if a dirty page/buffer is encountered, it must
408  * be waited upon, and not just skipped over.
409  *
410  * Return: %0 on success, negative error code otherwise.
411  */
412 int __filemap_fdatawrite_range(struct address_space *mapping, loff_t start,
413                               loff_t end, int sync_mode)
414 {
415     struct writeback_control wbc = {
416         .sync_mode = sync_mode,
417         .nr_to_write = LONG_MAX,
418         .range_start = start,
419         .range_end = end,
420     };
421     return filemap_fdatawrite_wbc(mapping, &wbc);
422 }

```

Listing 10-75: mm/filemap.c: __filemap_fdatawrite_range()

This establishes a struct `writeback_control` object used to proceed with writeback, passing this to `filemap_fdatawrite_wbc()` to perform the writeback, as shown in Listing 10-76.

```

368 /**
369  * filemap_fdatawrite_wbc - start writeback on mapping dirty pages in range
370  * @mapping:    address space structure to write
371  * @wbc:        the writeback_control controlling the writeout
372  *
373  * Call writepages on the mapping using the provided wbc to control the
374  * writeout.
375  *
376  * Return: %0 on success, negative error code otherwise.
377  */
378 int filemap_fdatawrite_wbc(struct address_space *mapping,
379                            struct writeback_control *wbc)
380 {
381     int ret;
382
383     if (!mapping_can_writeback(mapping) ||
384         !mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
385         return 0;
386
387     wbc_attach_fdatawrite_inode(wbc, mapping->host);
388     ret = do_writepages(mapping, wbc);
389     wbc_detach_inode(wbc);
390     return ret;
391 }

```

Listing 10-76: mm/filemap.c: `filemap_fdatawrite_wbc()`

This starts by checking whether the mapping has any pages in the page cache via `mapping_can_writeback()` and whether any are tagged dirty, checked via `mapping_tagged()`, either of these not being the case indicates that there is no writeback to perform and if so we simply exit early.

Otherwise we invoke writeback cgroup-specific logic via `wbc_attach_fdatawrite_inode()` and `wbc_detach_inode()` surrounding the core writeback function `do_writepages()` (see Listing 10-23).

The cgroup writeback logic is out of scope for the book, and the core writeback logic is described in detail in Section 10.10.

We will now examine the core `__filemap_fdatawait_range()` function in Listing 10-77.

```

501 static void __filemap_fdatawait_range(struct address_space *mapping,
502                                     loff_t start_byte, loff_t end_byte)
503 {
504     pgoff_t index = start_byte >> PAGE_SHIFT;
505     pgoff_t end = end_byte >> PAGE_SHIFT;
506     struct pagevec pvec;

```

```

507         int nr_pages;
508
509         if (end_byte < start_byte)
510             return;
511
512         pagevec_init(&pvec);
513         while (index <= end) {
514             unsigned i;
515
516             nr_pages = pagevec_lookup_range_tag(&pvec, mapping, &index,
517                                                end, PAGECACHE_TAG_WRITEBACK);
518             if (!nr_pages)
519                 break;
520
521             for (i = 0; i < nr_pages; i++) {
522                 struct page *page = pvec.pages[i];
523
524                 wait_on_page_writeback(page);
525                 ClearPageError(page);
526             }
527             pagevec_release(&pvec);
528             cond_resched();
529         }
530     }

```

Listing 10-77: mm/filemap.c: __filemap_fdatawait_range()

This iterates through folio batches at a time of folios tagged with `PAGECACHE_TAG_WRITEBACK` (it uses the legacy form of folio batches, `struct pagevec`. This behaves the same as folio batches, which are described in Section 11.7).

It uses `pagevec_lookup_range_tag()` (which in turn invokes `find_get_pages_range_tag()`) to retrieve folios tagged with `PAGECACHE_TAG_WRITEBACK`, i.e. ones which are in the midst of being written back, and then uses `wait_on_page_writeback()` to wait until this operation is complete.

The `wait_on_page_writeback()` function wraps `folio_wait_writeback()`, which we examine in Section 9.11 and Listing 9-146. This causes the process to sleep until each folio's writeback flag has been cleared, indicating the folio has been written back.

After each wait period, if the folio had been marked with an error using the `PG_error` flag this is cleared using the `ClearPageError()` flag.

After each batch is processed, it is freed via `pagevec_release()` and real-time kernel scheduling modes are able to reschedule using `cond_resched()` (this is out of scope for the book).

The net result is all folios being written back to disk are waited upon as required.

10.14 Dirty Throttling

In the process of performing writeback, it is absolutely vital to ensure that the system does not get overwhelmed with writeback requests to the point that the system becomes unstable.

In such a circumstance the system might become unstable as critical functions such as swap become bogged down, preventing the computer from being able to function correctly at all.

In addition, reaching a point where the number of dirty pages is such that the page cache is significantly inconsistent with the disk state it is caching dramatically increases the risk of data corruption should writeback not quickly resolve the situation.

Under ordinary circumstances no restrictions need be applied, pages can be dirtied and then written back as normal. However once a certain degree of pages are dirty, the kernel must take steps to mitigate this scenario.

The primary means by which this is mediated is through the `vm.dirty_background_ratio` (or alternatively `vm.dirty_background_bytes`) and `vm.dirty_ratio` (or `vm.dirty_bytes`) tunables.

These specify the maximum degree to which the system accumulates dirty pages, with the ratio values expressed as a proportion of available ‘dirtyable’ memory, i.e. memory that *could* be dirtied, before certain action is taken by the kernel:

Background dirty limits `vm.dirty_background_ratio` (or, alternatively, expressed as `vm.dirty_background_bytes`) dictate when the kernel addresses the issue that page cache state has become significantly inconsistent against disk state by initiating writeback that, as the name suggests, occurs in the background utilising the file system writeback described in Section 10.11. This defaults to using `vm.dirty_background_ratio` equal to 10% of dirtyable memory.

Foreground dirty limits `vm.dirty_ratio` (or alternatively `vm.dirty_bytes`) dictate the hard limit at which processes writing back to a block device *block* when dirtying pages until writeback can ensure that the proportion of dirty pages has dropped below this threshold again. This defaults to using `vm.dirty_ratio` equal to 20% of dirtyable memory.

This does not tell the whole story, however. Between these two limits, it’s important to ensure that we *throttle* dirtying of pages according to the level of bandwidth utilised per block device, taking proactive steps to avoid the worst case scenario of the hard limit being reached globally from occurring.

It’s also important to note that *all* the writeback that actually occurs as a result of dirty throttling is performed via background writeback, we throttle by simply ‘pausing’ processes which require it to keep us at an acceptable dirtying rate.

The overall process of activating background writeback and blocking processes in order to manage writeback is termed *dirty throttling*.

We will examine how this works in detail, but before we do we will examine what the *ratio* versions of the aforementioned tunables are a proportion

of. This is what is referred to as the global (i.e. system-wide) dirtyable pages count.

This is determined by the `global_dirtyable_memory()` function, shown in Listing 10-78 (eliding out of scope 32-bit system high memory logic).

```

338 /**
339  * global_dirtyable_memory - number of globally dirtyable pages
340  *
341  * Return: the global number of pages potentially available for dirty
342  * page cache. This is the base value for the global dirty limits.
343  */
344 static unsigned long global_dirtyable_memory(void)
345 {
346     unsigned long x;
347
348     x = global_zone_page_state(NR_FREE_PAGES);
349     /*
350      * Pages reserved for the kernel should not be considered
351      * dirtyable, to prevent a situation where reclaim has to
352      * clean pages in order to balance the zones.
353      */
354     x -= min(x, totalreserve_pages);
355
356     x += global_node_page_state(NR_INACTIVE_FILE);
357     x += global_node_page_state(NR_ACTIVE_FILE);
358     :
362     return x + 1;    /* Ensure that we never return 0 */
363 }

```

Listing 10-78: mm/page-writeback.c: `global_dirtyable_memory()`

This uses kernel statistics to determine how many free pages are available on the system, as each free page could end up mapped to a page cache entry and become dirty. It sums this with the total number of file pages, subtracting the global `totalreserve_pages` count, which comprises the sum of all zone high water marks and worst-case low memory reserve (see Section 2.4.2 for details on how this is calculated).

The means by which dirty throttling is applied is by the `balance_dirty_pages_ratelimited()` function being invoked each time a page cache object is dirtied, which we examine in Listing 10-79.

This should be invoked by any function which dirties page cache entries, *after* the dirtying has taken place. This is therefore invoked by numerous file system-specific functions, but also generic writeback functionality, specifically in `generic_perform_write()` as part of file write dirty tracking (see Section 10.12 and Listing 10-11) and in `fault_dirty_shared_page()` (see Listing 6-40 in Section 6.9) as part of memory-mapped page fault handling.

```

1937 /**
1938  * balance_dirty_pages_ratelimited - balance dirty memory state.

```

```

1939 * @mapping: address_space which was dirtied.
1940 *
1941 * Processes which are dirtying memory should call in here once for each page
1942 * which was newly dirtied. The function will periodically check the system's
1943 * dirty state and will initiate writeback if needed.
1944 *
1945 * Once we're over the dirty memory limit we decrease the ratelimiting
1946 * by a lot, to prevent individual processes from overshooting the limit
1947 * by (ratelimit_pages) each.
1948 */
1949 void balance_dirty_pages_ratelimited(struct address_space *mapping)
1950 {
1951     balance_dirty_pages_ratelimited_flags(mapping, 0);
1952 }

```

Listing 10-79: mm/page-writeback.c: `balance_dirty_pages_ratelimited()`

This invokes `balance_dirty_pages_ratelimited_flags()` in turn, which we examine below in Listing 10-81. Before we do so, we should examine how the dirty balancing functions conceptually.

The ultimate intent is to prevent the number of dirty pages in the system from exceeding a proportion of all ‘dirtyable’ pages (`vm.dirty_ratio`) or an explicit number of bytes (`vm.dirty_bytes`).

However precisely *how* we do this matters. The least sensible way of doing this would be to wait until the system hits the *global* dirty limit and cause all threads that dirty pages to sleep until writeback brings us back below the limit.

Imagine a single thread writing to a slow medium at a rate exceeding its ability to write back—having the kernel block all writeback to all block devices until we drop below the limit would cause all writing threads to stall until the slow medium had all of its data written out, resulting in an unstable system.

In this scenario it is also difficult to determine how long to wait for and getting this wrong could lock up the system altogether.

So this is certainly not sensible, however nor is throttling threads too soon—in the instance where the number of dirtied pages are below the dirty threshold, it’d simply be wasteful to unnecessarily put any thread to sleep when dirtying pages.

NOTE

To avoid having to repeatedly refer to `vm.dirty_background_ratio` or `vm.dirty_background_bytes`, and `vm.dirty_ratio` or `vm.dirty_bytes`, we will instead say ‘background threshold’ and ‘dirty threshold’ respectively, from this point on.

We must therefore find a solution that performs better. Ultimately the kernel’s weapon for battling too many dirty pages is to put threads to sleep so they can no longer dirty pages. All of the dirty throttling logic attempts to do so in sensible fashion as follows:

Throttle threads earlier Instead of only putting threads to sleep at the dirty threshold, we start doing so *prior* to reaching this limit so we avoid unnecessarily long stalls.

Divide dirtying into scopes Designate two different ‘scopes’ within which dirtying can occur—a *freerun* scope where no dirtying threads are put to sleep at all, and a *global dirty control* scope where threads are put to sleep for an appropriate period of time up to the dirty threshold.

Keep track of writeback bandwidth In order to inform the degree to which threads are paused, we keep track of each struct `bdi_writeback`’s write *bandwidth*, i.e. how many dirty pages per second are written back on average in struct `bdi_writeback->avg_write_bandwidth`.

Scale threads based on dirty rate The degree to which a thread contributes towards the number of dirtied pages is equal to the number of pages dirtied since last paused, therefore we use this metric to scale throttling per-thread.

Keep track of each thread’s dirtied pages In order to scale threads this way it is vital to track how many pages each thread has dirtied. This is done in struct `task_struct->nr_dirtied` which counts how many pages were dirtied since last examined by `balance_dirty_pages()` (which we examine below in Listing 10-97) at struct `task_struct->dirty_paused_when`.

Establish a per-block device, per-thread dirty rate limit Ideally each thread would be limited to a dirtying rate equal to the writeback bandwidth divided by the number of threads writing back. We are unable to determine precisely how many threads are writing back at any one time (and this can vary tremendously), but we can deduce this. The determined dirty rate is stored in struct `bdi_writeback->dirty_ratelimit`.

Establish an ideal setpoint of global dirty pages We want to strike a balance between causing threads to sleep too much and hitting the dirty threshold, so establish the mid point between these two as the *setpoint* of global dirty pages which we aim to maintain during the period when dirtying rate exceeds writeback bandwidth.

Smoothly adjust dirty rate limits to maintain setpoint It is critical to ‘smooth out’ the degree by which we pause threads, which we achieve by smoothing out the rate at which we adjust the struct `bdi_writeback->dirty_ratelimit`. We do so by scaling per-thread dirty rate by a *position ratio*, a coefficient which ranges from 0 to 2, adjusted such as to eventually converge on a value of 1, at which point the number of dirtied pages will be equal to the set point.

Scale thread pauses by distance from dirty rate Once we have establish a smoothed target per-thread per-block device dirty rate, we can use this to scale pauses such that we can pause each thread for either a shorter or a longer period of time such that the dirty rate trends towards the setpoint.

Maintain minimum and maximum thread pause times To avoid a pause so short that it has no material impact on dirty rate, or one so long as to cause a thread to become unresponsive, maintain minimum and maximum pause times using `wb_min_pause()` and `wb_max_pause()` (capped at `MAX_PAUSE`, hardcoded to 200ms).

The point at which we transition from freerun to global dirty control scope is determined by `dirty_freerun_ceiling()` which we examine in Listing 10-80 which is determined to be halfway between background and dirty threshold.

```

701 static unsigned long dirty_freerun_ceiling(unsigned long thresh,
702                                           unsigned long bg_thresh)
703 {
704     return (thresh + bg_thresh) / 2;
705 }

```

Listing 10-80: `mm/page-writeback.c: dirty_freerun_ceiling()`

This implementation results in heavy dirtying proceeding as shown in Figure 10-13.

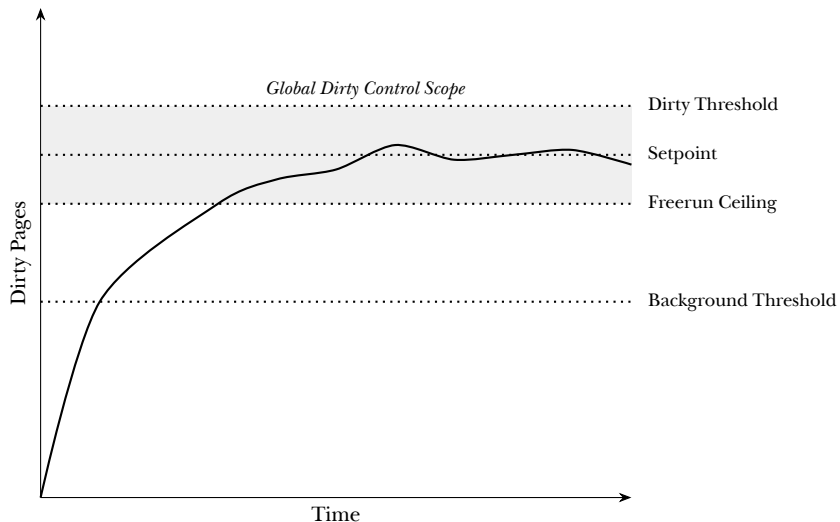


Figure 10-13: Example Dirty Page Throttling

Here dirtying threads cause a dramatic increase in dirtied pages before hitting the background threshold at which point writeback begins slowing things down. However this is insufficient to curtail the rate at which pages are dirtied, so the dirtying exceeds the freerun ceiling before threads start being slowed down, establishing an equilibrium around the setpoint.

10.15 Rate-Limited Dirty Throttling

It is relatively expensive to perform this balancing operation even if it results in no action being taken, so in addition to performing this balancing we also rate limit the operation itself. We can observe this in `balance_dirty_pages_ratelimited_flags()` which we examine in Listing 10-81.

```

1863 /**
1864  * balance_dirty_pages_ratelimited_flags - Balance dirty memory state.
1865  * @mapping: address_space which was dirtied.
1866  * @flags: BDP flags.
1867  *
1868  * Processes which are dirtying memory should call in here once for each page
1869  * which was newly dirtied. The function will periodically check the system's
1870  * dirty state and will initiate writeback if needed.
1871  *
1872  * See balance_dirty_pages_ratelimited() for details.
1873  *
1874  * Return: If @flags contains BDP_ASYNC, it may return -EAGAIN to
1875  * indicate that memory is out of balance and the caller must wait
1876  * for I/O to complete. Otherwise, it will return 0 to indicate
1877  * that either memory was already in balance, or it was able to sleep
1878  * until the amount of dirty memory returned to balance.
1879  */
1880 int balance_dirty_pages_ratelimited_flags(struct address_space *mapping,
1881                                         unsigned int flags)
1882 {
1883     struct inode *inode = mapping->host;
1884     struct backing_dev_info *bdi = inode_to_bdi(inode);
1885     struct bdi_writeback *wb = NULL;
1886     int ratelimit;
1887     int ret = 0;
1888     int *p;
1889
1890     if (!(bdi->capabilities & BDI_CAP_WRITEBACK))
1891         return ret;
1892
1893     if (inode_cgwb_enabled(inode))
1894         wb = wb_get_create_current(bdi, GFP_KERNEL);
1895     if (!wb)
1896         wb = &bdi->wb;
1897
1898     ratelimit = current->nr_dirtied_pause;
1899     if (wb->dirty_exceeded)
1900         ratelimit = min(ratelimit, 32 >> (PAGE_SHIFT - 10));
1901
1902     preempt_disable();
1903     /*

```

```

1904      * This prevents one CPU to accumulate too many dirtied pages without
1905      * calling into balance_dirty_pages(), which can happen when there are
1906      * 1000+ tasks, all of them start dirtying pages at exactly the same
1907      * time, hence all honoured too large initial task->nr_dirtied_pause.
1908      */
1909      p = this_cpu_ptr(&bdp_ratelimits);
1910      if (unlikely(current->nr_dirtied >= ratelimit))
1911          *p = 0;
1912      else if (unlikely(*p >= ratelimit_pages)) {
1913          *p = 0;
1914          ratelimit = 0;
1915      }
1916      /*
1917      * Pick up the dirtied pages by the exited tasks. This avoids lots of
1918      * short-lived tasks (eg. gcc invocations in a kernel build) escaping
1919      * the dirty throttling and livelock other long-run dirtiers.
1920      */
1921      p = this_cpu_ptr(&dirty_throttle_leaks);
1922      if (*p > 0 && current->nr_dirtied < ratelimit) {
1923          unsigned long nr_pages_dirtied;
1924          nr_pages_dirtied = min(*p, ratelimit - current->nr_dirtied);
1925          *p -= nr_pages_dirtied;
1926          current->nr_dirtied += nr_pages_dirtied;
1927      }
1928      preempt_enable();
1929
1930      if (unlikely(current->nr_dirtied >= ratelimit))
1931          ret = balance_dirty_pages(wb, current->nr_dirtied, flags);
1932
1933      wb_put(wb);
1934      return ret;
1935  }

```

Listing 10-81: mm/page-writeback.c: balance_dirty_pages_ratelimited_flags()

We start by checking whether the underlying block device is even capable of writeback. If not, then there is nothing to do. Next we obtain the struct `bdi_writeback` object associated with this mapping. The logic here is writeback cgroup-specific so out of scope.

After this we examine state associated with the last dirty throttle operation performed both by this thread and this CPU.

We query the struct `task_struct->nr_dirtied_pause` field associated with the current process, which indicates how many pages can be dirtied before we invoke the actual dirty throttling logic at `balance_dirty_pages()` logic (see Listing 10-97).

This value is set by `balance_dirty_pages()` to a value obtained from `dirty_poll_interval()` (see Listing 10-91) when the number of dirtied pages is still within the freerun region (which could be relatively large), or the num-

ber of pages that would be dirtied before reaching the *minimum* pause interval (as it'd be pointless trying to balance dirty pages if we are below this) as determined by `wb_min_pause()` (see Listing 10-96), a value which will be relatively small.

We cap this value to a hardcoded value, equal to 32 KiB of data (i.e. 8 pages for 4 KiB page size) if we have previously exceeded the dirty threshold (as determined by `struct bdi_writeback->dirty_exceeded`) to cause earlier dirty throttling to ensure we bring things under control quickly.

After this we perform a careful check to see whether the number of pages dirtied for this CPU (using the per-CPU `bdp_ratelimits` static value) exceeds the rate limit (if the current process's number of dirty pages has not already done so).

This check is performed against the hardcoded `ratelimit_pages` value (set to 32 pages), which sets a maximum number of dirty pages per CPU after which an invocation of `balance_dirty_pages()` must occur in order to perform dirty throttling.

This is done in order to avoid a scenario where a great many processes dirty at the same time in the freerunning regime resulting in a relatively large `struct task_struct->nr_dirtied_pause` value, the combined dirtying of which might push us over the dirty threshold or at least delay us bringing things back under control.

We therefore use this per-CPU check to set a hard upper limit on how many pages can be dirtied before dirty throttling occurs regardless of the number of processes.

If this edge case does arise then the `ratelimit` value is set to zero, meaning that the subsequent comparison between `struct task_struct->nr_dirtied` and `ratelimit` will always cause a balance to occur.

The `bdp_ratelimits` values for each CPU are updated when folios are dirtied in `folio_account_dirtied()`, as are each process's `struct task_struct->nr_dirtied`.

After this we perform another edge case check where we account for processes which exited before dirty throttling was able to take into account the process's `struct task_struct->nr_dirtied`.

On thread exit in `do_exit()`, the per-CPU value `dirty_throttle_leaks` is updated, accumulating the 'left over' `nr_dirtied` value.

This value is then simply added to the current thread's `struct task_struct->nr_dirtied` value to ensure that threads are scaled correctly.

This is specifically implemented to avoid a worst case scenario where a number of processes exit with a `struct task_struct->nr_dirtied` value close but not equal to `struct task_struct->nr_dirtied_pause` which would otherwise potentially significantly delay dirty throttling being performed.

Finally we perform the core check—determining whether the thread's `struct task_struct->nr_dirtied` value equals or exceeds the determined rate limit value. If so, we invoke the `balance_dirty_pages()` (see Listing 10-97) function in order to perform dirty throttling.

We will examine `balance_dirty_pages()` shortly, but before we do so let's examine the various statistics which are used in dirty throttle logic and how they are updated.

The key parameters we need are the number of pages dirtied and written and the *rate* at which pages are dirtied/written. For threads we also want to track the number of pages dirtied since we last paused a thread and when to assess the same thing per-thread.

10.16 Dirty Throttle Statistics

Let's examine each field which we regularly update:

- `struct bdi_writeback->stats[WB_DIRTIED]` – Measures the number of folios dirtied under this `struct bdi_writeback`. This uses a `struct percpu_counter` to track this parameter, with per-CPU values accumulated in batches of `WB_STAT_BATCH` folios by `wb_stat_mod()` in `folio_account_dirtied()` when a folio is dirtied, which is invoked by `__folio_mark_dirty()` (see Listing 10-1 in Section 10.1). It is also reduced in `folio_account_redirty()` if a file system rejects the dirtying of a folio, discussion of this is out of scope for the book however.
- `struct bdi_writeback->stats[WB_WRITTEN]` – Measures the number of folios written back under this `struct bdi_writeback`. This is updated by `__wb_writeout_add()` which is called by `__folio_end_writeback()` when writeback has completed (see Listing 10-29 in Section 10.10). The same batched per-CPU logic applies to `WB_WRITTEN` as with `WB_DIRTIED`.
- `struct bdi_writeback->dirtied_stamp` – A copy of `stats[WB_DIRTIED]` taken at `bw_time_stamp` at the same time `written_stamp` is taken, so the two values can be used in conjunction with one another. This is done in `__wb_update_bandwidth()` (or `wb_bandwidth_estimate_start()` if writeback has become idle). The `__wb_update_bandwidth()` function is invoked periodically at `BANDWIDTH_INTERVAL` (200ms) intervals, as well as invoked by `balance_dirty_pages()` and `do_writepages()` (at points at which the facts on the ground may have changed).
- `struct bdi_writeback->written_stamp` – A copy of `stats[WB_WRITTEN]`, updated alongside `stats[WB_DIRTIED]` in `__wb_update_bandwidth()` or `wb_bandwidth_estimate_start()`.
- `struct bdi_writeback->bw_time_stamp` – The time at which `__wb_update_bandwidth()` or `wb_bandwidth_estimate_start()` was last executed.
- `struct bdi_writeback->write_bandwidth` – The number of pages written per second for this `struct bdi_writeback`. This is updated in `wb_update_write_bandwidth()` by `__wb_update_bandwidth()`.
- `struct bdi_writeback->avg_write_bandwidth` – A smoothed average derived from `write_bandwidth` which evens out short-term variations in the write bandwidth figure.

- `struct bdi_writeback->balanced_dirty_ratelimit` – The *per-thread* limit on the number of pages dirtied per second for this `struct bdi_writeback`, updated in `wb_update_dirty_ratelimit()` invoked by `__wb_update_bandwidth()` but, importantly, only when the `update_ratelimit` parameter is specified, which is *not* the case on periodic bandwidth update but rather invoked only by `balance_dirty_pages()` immediately prior to this rate limit being used to make dirty throttle decisions. This value is scaled using the position ratio coefficient in order to implement sensible dirty throttling, we will discuss the details of this shortly.
- `struct bdi_writeback->dirty_ratelimit` – Logically equivalent to `balanced_dirty_ratelimit`, however updated in small steps to provide an smoothed averaged dirty rate limit with lower variance.
- `struct bdi_writeback->bw_dwork` – A `struct delayed_work` object which provides a work queue for bandwidth update operations. The bandwidth update is ultimately performed in `__wb_update_bandwidth()` as described above, the work queue periodically invoking `wb_update_bandwidth_workfn()` to do so. This work queue and thus the bandwidth updating (importantly *not* updating the dirty rate limit) is scheduled by `wb_inode_writeback_end()` to be run after a delay of `BANDWIDTH_INTERVAL` (200ms), called by `__folio_end_writeback()` when writeback has completed (see Listing 10-29 in Section 10.10). Thus this is only called when writes are actually occurring.
- `struct bdi_writeback->completions` – Maintains a count of the number of written back pages which have successfully completed writeback for this `struct bdi_writeback`. This is stored in such a way as to easily calculate the proportion of this value against overall writeback completion rate. This is updated in `wb_domain_writeout_add()` which is in turn invoked by `__wb_writeout_add()` and `__folio_end_writeback()` when writeback has completed (see Listing 10-29 in Section 10.10).
- `struct wb_domain->completions` – The `struct wb_domain` object is used to describe a domain of writeback. The only domain we consider is the global one (any other is relevant only to cgroup-mediated writeback which is out of scope for the book), which is maintained in the global variable `global_wb_domain`. Its `Completions` field is the system-wide equivalent to `struct bdi_writeback->completions`, and stored in a similar fashion (updated at precisely the same point in `wb_domain_writeout_add()`).
- `struct task_struct->nr_dirtied` – A count of the number of pages dirtied since we last paused the thread due to dirty throttling. This is updated in `folio_account_dirtied()` when a folio is dirtied, which is invoked by `__folio_mark_dirty()` (see Listing 10-1 in Section 10.1). It is also reduced in `folio_account_redirty()` if a file system rejects the dirtying of a folio. It is reset to zero when a thread is paused in `balance_dirty_pages()`.
- `struct task_struct->dirty_paused_when` – A timestamp indicating when last the thread was paused (or permitted to freerun).

- `struct task_struct->nr_dirtied_pause` – The number of pages which must be dirtied by this thread before dirty throttling is applied in `balance_dirty_pages()`. Determined to be either the minimum pause time if the thread was paused, or by `dirty_poll_interval()` (see Listing ??) if the thread is freerunning.

10.17 Bandwidth Updates

Let's examine the functions we've referred to here, starting with bandwidth updates in `__wb_update_bandwidth()`, which we explore in Listing 10-82 (eliding out of scope cgroup writeback logic).

```

1330 static void __wb_update_bandwidth(struct dirty_throttle_control *gdtc,
1331                                 struct dirty_throttle_control *mdtc,
1332                                 bool update_ratelimit)
1333 {
1334     struct bdi_writeback *wb = gdtc->wb;
1335     unsigned long now = jiffies;
1336     unsigned long elapsed;
1337     unsigned long dirtied;
1338     unsigned long written;
1339
1340     spin_lock(&wb->list_lock);
1341
1342     /*
1343      * Lockless checks for elapsed time are racy and delayed update after
1344      * IO completion doesn't do it at all (to make sure written pages are
1345      * accounted reasonably quickly). Make sure elapsed >= 1 to avoid
1346      * division errors.
1347      */
1348     elapsed = max(now - wb->bw_time_stamp, 1UL);
1349     dirtied = percpu_counter_read(&wb->stat[WB_DIRTIED]);
1350     written = percpu_counter_read(&wb->stat[WB_WRITTEN]);
1351
1352     if (update_ratelimit) {
1353         domain_update_dirty_limit(gdtc, now);
1354         wb_update_dirty_ratelimit(gdtc, dirtied, elapsed);
1355         :
1364     }
1365     wb_update_write_bandwidth(wb, elapsed, written);
1366
1367     wb->dirtied_stamp = dirtied;
1368     wb->written_stamp = written;
1369     WRITE_ONCE(wb->bw_time_stamp, now);
1370     spin_unlock(&wb->list_lock);
1371 }

```

Listing 10-82: `mm/page-writeback.c: __wb_update_bandwidth()`

This uses the struct `dirty_throttle_control` object to thread state through the function (as is used elsewhere in dirty throttling). The only version of this we concern ourselves with is the so-called `gdtc` which refers to *global* dirty throttle control (the `mdtc` value refers to out-of-scope cgroup writeback dirty throttle control).

We examine this type in Listing 10-84 shortly. For the time being, we can simply observe that it is passed to called functions and from it we obtain the struct `bdi_writeback` object to which it pertains from its `wb` field.

The function derives the elapsed time since last the bandwidth was updated (avoiding a potential later division by zero by assigning it one if it turns out to be zero).

The `dirtyed` and `written` values are derived from struct `bdi_writeback->stats` as described above.

If called from `balance_dirty_pages()` with `update_ratelimit` set, we update the dirty limit for the struct `bdi_writeback` object via `wb_update_dirty_ratelimit()` (see Listing 10-92). We also update the global struct `wb_domain` dirty limit using `domain_update_dirty_limit()`, however this is really only relevant for cgroup writeback so we do not examine it in detail.

Finally we always update struct `bdi_writeback` write bandwidth statistics via `wb_update_write_bandwidth()` (see Listing 10-85), as well as setting struct `bdi_writeback->dirtyed_stamp` and `written_stamp` values as well as the `bw_time_stamp`.

When this function is periodically invoked at `BANDWIDTH_INTERVAL` (200ms) intervals, this is done via `wb_update_bandwidth()`, which we examine in Listing 10-83.

```

1373 void wb_update_bandwidth(struct bdi_writeback *wb)
1374 {
1375     struct dirty_throttle_control gdtc = { GDTC_INIT(wb) };
1376
1377     __wb_update_bandwidth(&gdtc, NULL, false);
1378 }
```

Listing 10-83: mm/page-writeback.c: `wb_update_bandwidth()`

Importantly, this establishes a local struct `dirty_throttle_control` object with its `wb` field set to the appropriate struct `bdi_writeback` object to be passed into `__wb_update_bandwidth()` using `GDTC_INIT()` to initialise it.

We now return to the struct `dirty_throttle_control` type, examining it in Listing 10-84, eliding out of scope cgroup writeback fields.

```

122 /* consolidated parameters for balance_dirty_pages() and its subroutines */
123 struct dirty_throttle_control {
124     :
128     struct bdi_writeback    *wb;
129     struct fprop_local_percpu *wb_completions;
130
131     unsigned long          avail;           /* dirtyable */
132     unsigned long          dirty;          /* file_dirty + write + nfs */

```

```

133         unsigned long      thresh;        /* dirty threshold */
134         unsigned long      bg_thresh;      /* dirty background threshold
        */
135
136         unsigned long      wb_dirty;       /* per-wb counterparts */
137         unsigned long      wb_thresh;
138         unsigned long      wb_bg_thresh;
139
140         unsigned long      pos_ratio;
141     };

```

Listing 10-84: mm/page-writeback.c: struct dirty_throttle_control

Each time this is used in a context where a struct `bdi_writeback` must be referenced, this is set by the calling functions in the `wb` field, with `wb_completions` set to the struct `bdi_writeback->completions` field.

The `avail` field indicates the total number of dirtyable pages as determined by `global_dirtyable_memory()` (see Listing 10-78) and the `dirty` field set to the sum of global dirty and writeback pages.

The `thresh` and `bg_thresh` value are set to the dirty threshold and background dirty threshold system values in `domain_dirty_limits()`.

The struct `bdi_writeback`-specific `wb_dirty` value is set in `wb_dirty_limits()` (see Listing 10-89), along side `wb_thresh` and `wb_bg_thresh`.

Finally, the `pos_ratio` field, specifying the position ratio coefficient which is carefully calculated as part of the dirty throttling mechanism, is set in `wb_position_ratio()` (see Listing 10-86), which we examine later.

Generally speaking, changes made to an struct `dirty_throttle_control` object are ultimately invoked by the core dirty throttle function `balance_dirty_pages()` (see Listing 10-97).

Returning to `_wb_update_bandwidth()` (see Listing 10-82), we will start by examining the common write bandwidth update logic in `wb_update_write_bandwidth()` in Listing 10-85.

```

1068 static void wb_update_write_bandwidth(struct bdi_writeback *wb,
1069                                     unsigned long elapsed,
1070                                     unsigned long written)
1071 {
1072     const unsigned long period = roundup_pow_of_two(3 * HZ);
1073     unsigned long avg = wb->avg_write_bandwidth;
1074     unsigned long old = wb->write_bandwidth;
1075     u64 bw;
1076
1077     /*
1078      * bw = written * HZ / elapsed
1079      *
1080      *          bw * elapsed + write_bandwidth * (period -
1081      *          elapsed)
1082      * write_bandwidth =
1083      *
1084      *
1085      *
1086      *
1087      *
1088      *
1089      *
1090      *
1091      *
1092      *
1093      *
1094      *
1095      *
1096      *
1097      *
1098      *
1099      *
1100      *
1101      *
1102      *
1103      *
1104      *
1105      *
1106      *
1107      *
1108      *
1109      *
1110      *
1111      *
1112      *
1113      *
1114      *
1115      *
1116      *
1117      *
1118      *
1119      *
1120      *
1121      *
1122      *
1123      *
1124      *
1125      *
1126      *
1127      *
1128      *
1129      *
1130      *
1131      *
1132      *
1133      *
1134      *
1135      *
1136      *
1137      *
1138      *
1139      *
1140      *
1141      *
1142      *
1143      *
1144      *
1145      *
1146      *
1147      *
1148      *
1149      *
1150      *
1151      *
1152      *
1153      *
1154      *
1155      *
1156      *
1157      *
1158      *
1159      *
1160      *
1161      *
1162      *
1163      *
1164      *
1165      *
1166      *
1167      *
1168      *
1169      *
1170      *
1171      *
1172      *
1173      *
1174      *
1175      *
1176      *
1177      *
1178      *
1179      *
1180      *
1181      *
1182      *
1183      *
1184      *
1185      *
1186      *
1187      *
1188      *
1189      *
1190      *
1191      *
1192      *
1193      *
1194      *
1195      *
1196      *
1197      *
1198      *
1199      *
1200      *
1201      *
1202      *
1203      *
1204      *
1205      *
1206      *
1207      *
1208      *
1209      *
1210      *
1211      *
1212      *
1213      *
1214      *
1215      *
1216      *
1217      *
1218      *
1219      *
1220      *
1221      *
1222      *
1223      *
1224      *
1225      *
1226      *
1227      *
1228      *
1229      *
1230      *
1231      *
1232      *
1233      *
1234      *
1235      *
1236      *
1237      *
1238      *
1239      *
1240      *
1241      *
1242      *
1243      *
1244      *
1245      *
1246      *
1247      *
1248      *
1249      *
1250      *
1251      *
1252      *
1253      *
1254      *
1255      *
1256      *
1257      *
1258      *
1259      *
1260      *
1261      *
1262      *
1263      *
1264      *
1265      *
1266      *
1267      *
1268      *
1269      *
1270      *
1271      *
1272      *
1273      *
1274      *
1275      *
1276      *
1277      *
1278      *
1279      *
1280      *
1281      *
1282      *
1283      *
1284      *
1285      *
1286      *
1287      *
1288      *
1289      *
1290      *
1291      *
1292      *
1293      *
1294      *
1295      *
1296      *
1297      *
1298      *
1299      *
1300      *
1301      *
1302      *
1303      *
1304      *
1305      *
1306      *
1307      *
1308      *
1309      *
1310      *
1311      *
1312      *
1313      *
1314      *
1315      *
1316      *
1317      *
1318      *
1319      *
1320      *
1321      *
1322      *
1323      *
1324      *
1325      *
1326      *
1327      *
1328      *
1329      *
1330      *
1331      *
1332      *
1333      *
1334      *
1335      *
1336      *
1337      *
1338      *
1339      *
1340      *
1341      *
1342      *
1343      *
1344      *
1345      *
1346      *
1347      *
1348      *
1349      *
1350      *
1351      *
1352      *
1353      *
1354      *
1355      *
1356      *
1357      *
1358      *
1359      *
1360      *
1361      *
1362      *
1363      *
1364      *
1365      *
1366      *
1367      *
1368      *
1369      *
1370      *
1371      *
1372      *
1373      *
1374      *
1375      *
1376      *
1377      *
1378      *
1379      *
1380      *
1381      *
1382      *
1383      *
1384      *
1385      *
1386      *
1387      *
1388      *
1389      *
1390      *
1391      *
1392      *
1393      *
1394      *
1395      *
1396      *
1397      *
1398      *
1399      *
1400      *
1401      *
1402      *
1403      *
1404      *
1405      *
1406      *
1407      *
1408      *
1409      *
1410      *
1411      *
1412      *
1413      *
1414      *
1415      *
1416      *
1417      *
1418      *
1419      *
1420      *
1421      *
1422      *
1423      *
1424      *
1425      *
1426      *
1427      *
1428      *
1429      *
1430      *
1431      *
1432      *
1433      *
1434      *
1435      *
1436      *
1437      *
1438      *
1439      *
1440      *
1441      *
1442      *
1443      *
1444      *
1445      *
1446      *
1447      *
1448      *
1449      *
1450      *
1451      *
1452      *
1453      *
1454      *
1455      *
1456      *
1457      *
1458      *
1459      *
1460      *
1461      *
1462      *
1463      *
1464      *
1465      *
1466      *
1467      *
1468      *
1469      *
1470      *
1471      *
1472      *
1473      *
1474      *
1475      *
1476      *
1477      *
1478      *
1479      *
1480      *
1481      *
1482      *
1483      *
1484      *
1485      *
1486      *
1487      *
1488      *
1489      *
1490      *
1491      *
1492      *
1493      *
1494      *
1495      *
1496      *
1497      *
1498      *
1499      *
1500      *
1501      *
1502      *
1503      *
1504      *
1505      *
1506      *
1507      *
1508      *
1509      *
1510      *
1511      *
1512      *
1513      *
1514      *
1515      *
1516      *
1517      *
1518      *
1519      *
1520      *
1521      *
1522      *
1523      *
1524      *
1525      *
1526      *
1527      *
1528      *
1529      *
1530      *
1531      *
1532      *
1533      *
1534      *
1535      *
1536      *
1537      *
1538      *
1539      *
1540      *
1541      *
1542      *
1543      *
1544      *
1545      *
1546      *
1547      *
1548      *
1549      *
1550      *
1551      *
1552      *
1553      *
1554      *
1555      *
1556      *
1557      *
1558      *
1559      *
1560      *
1561      *
1562      *
1563      *
1564      *
1565      *
1566      *
1567      *
1568      *
1569      *
1570      *
1571      *
1572      *
1573      *
1574      *
1575      *
1576      *
1577      *
1578      *
1579      *
1580      *
1581      *
1582      *
1583      *
1584      *
1585      *
1586      *
1587      *
1588      *
1589      *
1590      *
1591      *
1592      *
1593      *
1594      *
1595      *
1596      *
1597      *
1598      *
1599      *
1600      *
1601      *
1602      *
1603      *
1604      *
1605      *
1606      *
1607      *
1608      *
1609      *
1610      *
1611      *
1612      *
1613      *
1614      *
1615      *
1616      *
1617      *
1618      *
1619      *
1620      *
1621      *
1622      *
1623      *
1624      *
1625      *
1626      *
1627      *
1628      *
1629      *
1630      *
1631      *
1632      *
1633      *
1634      *
1635      *
1636      *
1637      *
1638      *
1639      *
1640      *
1641      *
1642      *
1643      *
1644      *
1645      *
1646      *
1647      *
1648      *
1649      *
1650      *
1651      *
1652      *
1653      *
1654      *
1655      *
1656      *
1657      *
1658      *
1659      *
1660      *
1661      *
1662      *
1663      *
1664      *
1665      *
1666      *
1667      *
1668      *
1669      *
1670      *
1671      *
1672      *
1673      *
1674      *
1675      *
1676      *
1677      *
1678      *
1679      *
1680      *
1681      *
1682      *
1683      *
1684      *
1685      *
1686      *
1687      *
1688      *
1689      *
1690      *
1691      *
1692      *
1693      *
1694      *
1695      *
1696      *
1697      *
1698      *
1699      *
1700      *
1701      *
1702      *
1703      *
1704      *
1705      *
1706      *
1707      *
1708      *
1709      *
1710      *
1711      *
1712      *
1713      *
1714      *
1715      *
1716      *
1717      *
1718      *
1719      *
1720      *
1721      *
1722      *
1723      *
1724      *
1725      *
1726      *
1727      *
1728      *
1729      *
1730      *
1731      *
1732      *
1733      *
1734      *
1735      *
1736      *
1737      *
1738      *
1739      *
1740      *
1741      *
1742      *
1743      *
1744      *
1745      *
1746      *
1747      *
1748      *
1749      *
1750      *
1751      *
1752      *
1753      *
1754      *
1755      *
1756      *
1757      *
1758      *
1759      *
1760      *
1761      *
1762      *
1763      *
1764      *
1765      *
1766      *
1767      *
1768      *
1769      *
1770      *
1771      *
1772      *
1773      *
1774      *
1775      *
1776      *
1777      *
1778      *
1779      *
1780      *
1781      *
1782      *
1783      *
1784      *
1785      *
1786      *
1787      *
1788      *
1789      *
1790      *
1791      *
1792      *
1793      *
1794      *
1795      *
1796      *
1797      *
1798      *
1799      *
1800      *
1801      *
1802      *
1803      *
1804      *
1805      *
1806      *
1807      *
1808      *
1809      *
1810      *
1811      *
1812      *
1813      *
1814      *
1815      *
1816      *
1817      *
1818      *
1819      *
1820      *
1821      *
1822      *
1823      *
1824      *
1825      *
1826      *
1827      *
1828      *
1829      *
1830      *
1831      *
1832      *
1833      *
1834      *
1835      *
1836      *
1837      *
1838      *
1839      *
1840      *
1841      *
1842      *
1843      *
1844      *
1845      *
1846      *
1847      *
1848      *
1849      *
1850      *
1851      *
1852      *
1853      *
1854      *
1855      *
1856      *
1857      *
1858      *
1859      *
1860      *
1861      *
1862      *
1863      *
1864      *
1865      *
1866      *
1867      *
1868      *
1869      *
1870      *
1871      *
1872      *
1873      *
1874      *
1875      *
1876      *
1877      *
1878      *
1879      *
1880      *
1881      *
1882      *
1883      *
1884      *
1885      *
1886      *
1887      *
1888      *
1889      *
1890      *
1891      *
1892      *
1893      *
1894      *
1895      *
1896      *
1897      *
1898      *
1899      *
1900      *
1901      *
1902      *
1903      *
1904      *
1905      *
1906      *
1907      *
1908      *
1909      *
1910      *
1911      *
1912      *
1913      *
1914      *
1915      *
1916      *
1917      *
1918      *
1919      *
1920      *
1921      *
1922      *
1923      *
1924      *
1925      *
1926      *
1927      *
1928      *
1929      *
1930      *
1931      *
1932      *
1933      *
1934      *
1935      *
1936      *
1937      *
1938      *
1939      *
1940      *
1941      *
1942      *
1943      *
1944      *
1945      *
1946      *
1947      *
1948      *
1949      *
1950      *
1951      *
1952      *
1953      *
1954      *
1955      *
1956      *
1957      *
1958      *
1959      *
1960      *
1961      *
1962      *
1963      *
1964      *
1965      *
1966      *
1967      *
1968      *
1969      *
1970      *
1971      *
1972      *
1973      *
1974      *
1975      *
1976      *
1977      *
1978      *
1979      *
1980      *
1981      *
1982      *
1983      *
1984      *
1985      *
1986      *
1987      *
1988      *
1989      *
1990      *
1991      *
1992      *
1993      *
1994      *
1995      *
1996      *
1997      *
1998      *
1999      *
2000      *
2001      *
2002      *
2003      *
2004      *
2005      *
2006      *
2007      *
2008      *
2009      *
2010      *
2011      *
2012      *
2013      *
2014      *
2015      *
2016      *
2017      *
2018      *
2019      *
2020      *
2021      *
2022      *
2023      *
2024      *
2025      *
2026      *
2027      *
2028      *
2029      *
2030      *
2031      *
2032      *
2033      *
2034      *
2035      *
2036      *
2037      *
2038      *
2039      *
2040      *
2041      *
2042      *
2043      *
2044      *
2045      *
2046      *
2047      *
2048      *
2049      *
2050      *
2051      *
2052      *
2053      *
2054      *
2055      *
2056      *
2057      *
2058      *
2059      *
2060      *
2061      *
2062      *
2063      *
2064      *
2065      *
2066      *
2067      *
2068      *
2069      *
2070      *
2071      *
2072      *
2073      *
2074      *
2075      *
2076      *
2077      *
2078      *
2079      *
2080      *
2081      *
2082      *
2083      *
2084      *
2085      *
2086      *
2087      *
2088      *
2089      *
2090      *
2091      *
2092      *
2093      *
2094      *
2095      *
2096      *
2097      *
2098      *
2099      *
2100      *
2101      *
2102      *
2103      *
2104      *
2105      *
2106      *
2107      *
2108      *
2109      *
2110      *
2111      *
2112      *
2113      *
2114      *
2115      *
2116      *
2117      *
2118      *
2119      *
2120      *
2121      *
2122      *
2123      *
2124      *
2125      *
2126      *
2127      *
2128      *
2129      *
2130      *
2131      *
2132      *
2133      *
2134      *
2135      *
2136      *
2137      *
2138      *
2139      *
2140      *
2141      *
2142      *
2143      *
2144      *
2145      *
2146      *
2147      *
2148      *
2149      *
2150      *
2151      *
2152      *
2153      *
2154      *
2155      *
2156      *
2157      *
2158      *
2159      *
2160      *
2161      *
2162      *
2163      *
2164      *
2165      *
2166      *
2167      *
2168      *
2169      *
2170      *
2171      *
2172      *
2173      *
2174      *
2175      *
2176      *
2177      *
2178      *
2179      *
2180      *
2181      *
2182      *
2183      *
2184      *
2185      *
2186      *
2187      *
2188      *
2189      *
2190      *
2191      *
2192      *
2193      *
2194      *
2195      *
2196      *
2197      *
2198      *
2199      *
2200      *
2201      *
2202      *
2203      *
2204      *
2205      *
2206      *
2207      *
2208      *
2209      *
2210      *
2211      *
2212      *
2213      *
2214      *
2215      *
2216      *
2217      *
2218      *
2219      *
2220      *
2221      *
2222      *
2223      *
2224      *
2225      *
2226      *
2227      *
2228      *
2229      *
2230      *
2231      *
2232      *
2233      *
2234      *
2235      *
2236      *
2237      *
2238      *
2239      *
2240      *
2241      *
2242      *
2243      *
2244      *
2245      *
2246      *
2247      *
2248      *
2249      *
2250      *
2251      *
2252      *
2253      *
2254      *
2255      *
2256      *
2257      *
2258      *
2259      *
2260      *
2261      *
2262      *
2263      *
2264      *
2265      *
2266      *
2267      *
2268      *
2269      *
2270      *
2271      *
2272      *
2273      *
2274      *
2275      *
2276      *
2277      *
2278      *
2279      *
2280      *
2281      *
2282      *
2283      *
2284      *
2285      *
2286      *
2287      *
2288      *
2289      *
2290      *
2291      *
2292      *
2293      *
2294      *
2295      *
2296      *
2297      *
2298      *
2299      *
2300      *
2301      *
2302      *
2303      *
2304      *
2305      *
2306      *
2307      *
2308      *
2309      *
2310      *
2311      *
2312      *
2313      *
2314      *
2315      *
2316      *
2317      *
2318      *
2319      *
2320      *
2321      *
2322      *
2323      *
2324      *
2325      *
2326      *
2327      *
2328      *
2329      *
2330      *
2331      *
2332      *
2333      *
2334      *
2335      *
2336      *
2337      *
2338      *
2339      *
2340      *
2341      *
2342      *
2343      *
2344      *
2345      *
2346      *
2347      *
2348      *
2349      *
2350      *
2351      *
2352      *
2353      *
2354      *
2355      *
2356      *
2357      *
2358      *
2359      *
2360      *
2361      *
2362      *
2363      *
2364      *
2365      *
2366      *
2367      *
2368      *
2369      *
2370      *
2371      *
2372      *
2373      *
2374      *
2375      *
2376      *
2377      *
2378      *
2379      *
2380      *
2381      *
2382      *
2383      *
2384      *
2385      *
2386      *
2387      *
2388      *
2389      *
2390      *
2391      *
2392      *
2393      *
2394      *
2395      *
2396      *
2397      *
2398      *
2399      *
2400      *
2401      *
2402      *
2403      *
2404      *
2405      *
2406      *
2407      *
2408      *
2409      *
2410      *
2411      *
2412      *
2413      *
2414      *
2415      *
2416      *
2417      *
2418      *
2419      *
2420      *
2421      *
2422      *
2423      *
2424      *
2425      *
2426      *
2427      *
2428      *
2429      *
2430      *
2431      *
2432      *
2433      *
2434      *
2435      *
2436      *
2437      *
2438      *
2439      *
2440      *
2441      *
2442      *
2443      *
2444      *
2445      *
2446      *
2447      *
2448      *
2449      *
2450      *
2451      *
2452      *
2453      *
2454      *
2455      *
2456      *
2457      *
2458      *
2459      *
2460      *
2461      *
2462      *
2463      *
2464      *
2465      *
2466      *
2467      *
2468      *
2469      *
2470      *
2471      *
2472      *
2473      *
2474      *
2475      *
2476      *
2477      *
2478      *
2479      *
2480      *
2481      *
2482      *
2483      *
2484      *
2485      *
2486      *
2487      *
2488      *
2489      *
2490      *
2491      *
2492      *
2493      *
2494      *
2495      *
2496      *
2497      *
2498      *
2499      *
2500      *
2501      *
2502      *
2503      *
2504      *
2505      *
2506      *
2507      *
2508      *
2509      *
2510      *
2511      *
2512      *
2513      *
2514      *
2515      *
2516      *
2517      *
2518      *
2519      *
2520      *
2521      *
2522      *
2523      *
2524      *
2525      *
2526      *
2527      *
2528      *
2529      *
2530      *
2531      *
2532      *
2533      *
2534      *
2535      *
2536      *
2537      *
2538      *
2539      *
2540      *
2541      *
2542      *
2543      *
2544      *
2545      *
2546      *
2547      *
2548      *
2549      *
2550      *
2551      *
2552      *
2553      *
2554      *
2555      *
2556      *
2557      *
2558      *
2559      *
2560      *
2561      *
2562      *
2563      *
2564      *
2565      *
2566      *
2567      *
2568      *
2569      *
2570      *
2571      *
2572      *
2573      *
2574      *
2575      *
2576      *
2577      *
2578      *
2579      *
2580      *
2581      *
2582      *
2583      *
2584      *
2585      *
2586      *
2587      *
2588      *
2589      *
2590      *
2591      *
2592      *
2593      *
2594      *
2595      *
2596      *
2597      *
2598      *
2599      *
2600      *
2601      *
2602      *
2603      *
2604      *
2605      *
2606      *
2607      *
2608      *
2609      *
2610      *
2611      *
2612      *
2613      *
2614      *
2615      *
2616      *
2617      *
2618      *
2619      *
2620      *
2621      *
2622      *
2623      *
2624      *
2625      *
2626      *
2627      *
2628      *
2629      *
2630      *
2631      *
2632      *
2633      *
2634      *
2635      *
2636      *
2637      *
2638      *
2639      *
2640      *
2641      *
2642      *
2643      *
2644      *
2645      *
2646      *
2647      *
2648      *
2649      *
2650      *
2651      *
2652      *
2653      *
2654      *
2655      *
2656      *
2657      *
2658      *
2659      *
2660      *
2661      *
2662      *
2663      *
2664      *
2665      *
2666      *
2667      *
2668      *
2669      *
2670      *
2671      *
2672      *
2673      *
2674      *
2675      *
2676      *
2677      *
2678      *
2679      *
2680      *
2681      *
2682      *
2683      *
2684      *
2685      *
2686      *
2687      *
2688      *
2689      *
2690      *
2691      *
2692
```



```

1082         *
1083         *
1084         * @written may have decreased due to folio_account_redirty().
1085         * Avoid underflowing @bw calculation.
1086         */
1087         bw = written - min(written, wb->written_stamp);
1088         bw *= HZ;
1089         if (unlikely(elapsed > period)) {
1090             bw = div64_ul(bw, elapsed);
1091             avg = bw;
1092             goto out;
1093         }
1094         bw += (u64)wb->write_bandwidth * (period - elapsed);
1095         bw >>= ilog2(period);
1096
1097         /*
1098         * one more level of smoothing, for filtering out sudden spikes
1099         */
1100         if (avg > old && old >= (unsigned long)bw)
1101             avg -= (avg - old) >> 3;
1102
1103         if (avg < old && old <= (unsigned long)bw)
1104             avg += (old - avg) >> 3;
1105
1106     out:
1107         /* keep avg > 0 to guarantee that tot > 0 if there are dirty wbs */
1108         avg = max(avg, 1LU);
1109         if (wb_has_dirty_io(wb)) {
1110             long delta = avg - wb->avg_write_bandwidth;
1111             WARN_ON_ONCE(atomic_long_add_return(delta,
1112                                     &wb->bdi->tot_write_bandwidth) <= 0);
1113         }
1114         wb->write_bandwidth = bw;
1115         WRITE_ONCE(wb->avg_write_bandwidth, avg);
1116     }

```

Listing 10-85: mm/page-writeback.c: `wb_update_write_bandwidth()`

NOTE

The dirty throttling logic we explore from this point on involves some rather delicate mathematics. Rather than dwell too long on these, we shall examine the core of the functions under examination, showing the remainder so as to leave further mathematical examination of the details as an exercise for the reader.

Note that in this function and many others, careful choices around how integer mathematical operations are performed occur, such as using power-of-2 values for instance here via `roundup_pow_of_two()`. In service of brevity, we do not examine these too closely but rather proceed referring to what the value ultimately represents.

Note that this function is passed `struct bdi_writeback->stats[WB_WRITTEN]` via the `written` parameter.

We subtract the value that this was at when last we obtained a bandwidth from the `struct bdi_writeback->written_stamp` field (which the calling `_wb_update_bandwidth()` function will update *after* this function is complete), obtaining the number of pages written since we last updated the bandwidth value.

We attempt to obtain a rolling average over a hardcoded period of 3 seconds (the multiplication in this function of values with HZ are to convert them to the internal kernel time metric of *jiffies*).

If the elapsed period happens to exceed this period then we simply take the write bandwidth (and smoothed averaged write bandwidth) to be equal to $\frac{\Delta_{written}}{elapsed}$ and exit the function early.

Otherwise, we calculate the write bandwidth to be equal to $\frac{\Delta_{written}}{period} + old \times \frac{period - elapsed}{period}$, providing a rolling average over the hardcoded 3 second period.

After obtaining this value from `struct bdi_writeback->write_bandwidth`, we perform some careful smoothing against the previous average value, assigning this to `struct bdi_writeback->avg_write_bandwidth`.

10.18 Dirty Position Control Ratio

Before we examine the dirty rate calculation, let's reflect on the *dirty position control ratio*, which we maintain in `struct dirty_throttle_control->pos_ratio`. This value is determined in `wb_position_ratio()` which we examine in Listing 10-86.

As discussed previously, this is a coefficient we use to scale the dirty rate limit such that each individual thread converges on the desired set point under writeback pressure, clamped to a minimum of zero and a maximum of two, allowing for the dirty rate to be at most doubled if the dirtying rate is running to low, or turned into a hard limit if the dirty rate is so high as to overwhelm the block device.

We examine how this value is intended to be maintained in Figure 10-14.

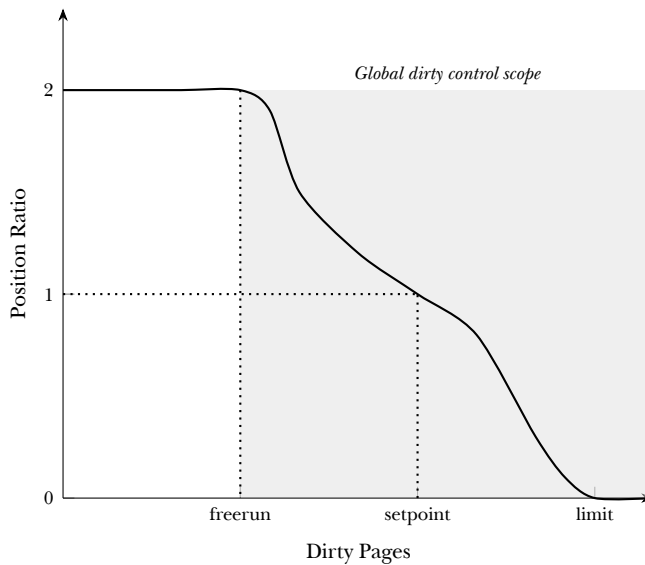


Figure 10-14: Example Dirty Position Control Ratio

Compare this figure to the impact this is intended to have in Figure 10-13. The intent is to maintain a position ratio of 1, and varying this only when thread dirty rate differs from write bandwidth scaled by the number of threads.

The dirty position control ratio is determined in `wb_position_ratio()`, which we examine in Listing 10-86. Note that we elide handling of block device strict limit logic for brevity (as specified by `BDI_CAP_STRICTLIMIT`), and a number of comments that would make the function overlong.

```

889 static void wb_position_ratio(struct dirty_throttle_control *dtc)
890 {
891     struct bdi_writeback *wb = dtc->wb;
892     unsigned long write_bw = READ_ONCE(wb->avg_write_bandwidth);
893     unsigned long freerun = dirty_freerun_ceiling(dtc->thresh, dtc->
      bg_thresh);
894     unsigned long limit = hard_dirty_limit(dtc_dom(dtc), dtc->thresh);
895     unsigned long wb_thresh = dtc->wb_thresh;
896     unsigned long x_intercept;
897     unsigned long setpoint;          /* dirty pages' target balance point
      */
898     unsigned long wb_setpoint;
899     unsigned long span;
900     long long pos_ratio;             /* for scaling up/down the rate limit
      */
901     long x;
902
903     dtc->pos_ratio = 0;
904
```

```

905         if (unlikely(dtc->dirty >= limit))
906             return;
907         :
913         setpoint = (freerun + limit) / 2;
914         pos_ratio = pos_ratio_polynom(setpoint, dtc->dirty, limit);
915         :
987         /*
988          * We have computed basic pos_ratio above based on global situation.
989          * If
990          * the wb is over/under its share of dirty pages, we want to scale
991          * pos_ratio further down/up. That is done by the following mechanism.
992          */
993         :
1018        if (unlikely(wb_thresh > dtc->thresh))
1019            wb_thresh = dtc->thresh;
1020        /*
1021         * It's very possible that wb_thresh is close to 0 not because the
1022         * device is slow, but that it has remained inactive for long time.
1023         * Honour such devices a reasonable good (hopefully IO efficient)
1024         * threshold, so that the occasional writes won't be blocked and
1025         active
1026         * writes can rampup the threshold quickly.
1027         */
1028        wb_thresh = max(wb_thresh, (limit - dtc->dirty) / 8);
1029        /*
1030         * scale global setpoint to wb's:
1031         *      wb_setpoint = setpoint * wb_thresh / thresh
1032         */
1033        x = div_u64((u64)wb_thresh << 16, dtc->thresh | 1);
1034        wb_setpoint = setpoint * (u64)x >> 16;
1035        /*
1036         * Use span=(8*write_bw) in single wb case as indicated by
1037         * (thresh - wb_thresh ~= 0) and transit to wb_thresh in JBOD case.
1038         *
1039         *      wb_thresh                thresh - wb_thresh
1040         * span = ----- * (8 * write_bw) + ----- * wb_thresh
1041         *      thresh                    thresh
1042         */
1043        span = (dtc->thresh - wb_thresh + 8 * write_bw) * (u64)x >> 16;
1044        x_intercept = wb_setpoint + span;
1045
1046        if (dtc->wb_dirty < x_intercept - span / 4) {
1047            pos_ratio = div64_u64(pos_ratio * (x_intercept - dtc->wb_dirty
1048            ),
1049                                   (x_intercept - wb_setpoint) | 1);
1050        } else
1051            pos_ratio /= 4;

```

```

1050
1051      /*
1052      * wb reserve area, safeguard against dirty pool underrun and disk
      idle
1053      * It may push the desired control point of global dirty pages higher
1054      * than setpoint.
1055      */
1056      x_intercept = wb_thresh / 2;
1057      if (dtc->wb_dirty < x_intercept) {
1058          if (dtc->wb_dirty > x_intercept / 8)
1059              pos_ratio = div_u64(pos_ratio * x_intercept,
1060                                  dtc->wb_dirty);
1061          else
1062              pos_ratio *= 8;
1063      }
1064
1065      dtc->pos_ratio = pos_ratio;
1066  }

```

Listing 10-86: mm/page-writeback.c: wb_position_ratio()

We won't delve too deeply into the mathematics here, rather we will make some simple observations around the logic.

If the global number of dirty pages exceeds the dirty threshold (specified in `limit`), then we do not need to calculate the ratio, we set it to zero such that we get the maximum process pause.

We note that the setpoint is determined to be half way between the freerun value determined by `dirty_freerun_ceiling()` (see Listing 10-80) and the dirty threshold determined by `hard_dirty_limit()` (since cgroup writeback is out of scope for the book, we consider only the case where this is equal to the global dirty threshold).

The `dirty_freerun_ceiling()` function determines the freerun zone to start halfway between the background threshold and dirty threshold values, and thus this establishes the setpoint as halfway between this value and the dirty threshold.

By default, `vm.dirty_background_ratio` is set to 10% of dirtyable memory and `vm.dirty_ratio` is set to 20%. This therefore implies that freerun by default (i.e. no attempt to dirty throttle) applies up to 15% of dirtyable memory, with a setpoint of 17.5%.

While we won't examine the mathematics, it's instructive to observe `pos_ratio_polynom()` in Listing 10-87.

```

783  /*
784  *                               setpoint - dirty 3
785  *      f(dirty) := 1.0 + (-----)
786  *                               limit - setpoint
787  *
788  * it's a 3rd order polynomial that subjects to
789  *

```

```

790 * (1)  $f(\text{freerun}) = 2.0 \Rightarrow$  rampup dirty_ratelimit reasonably fast
791 * (2)  $f(\text{setpoint}) = 1.0 \Rightarrow$  the balance point
792 * (3)  $f(\text{limit}) = 0 \Rightarrow$  the hard limit
793 * (4)  $df/dx \leq 0 \Rightarrow$  negative feedback control
794 * (5) the closer to setpoint, the smaller  $|df/dx|$  (and the reverse)
795 *  $\Rightarrow$  fast response on large errors; small oscillation near setpoint
796 */
797 static long long pos_ratio_polynom(unsigned long setpoint,
798                                   unsigned long dirty,
799                                   unsigned long limit)
800 {
801     long long pos_ratio;
802     long x;
803
804     x = div64_s64(((s64)setpoint - (s64)dirty) << RATELIMIT_CALC_SHIFT,
805                  (limit - setpoint) | 1);
806     pos_ratio = x;
807     pos_ratio = pos_ratio * x >> RATELIMIT_CALC_SHIFT;
808     pos_ratio = pos_ratio * x >> RATELIMIT_CALC_SHIFT;
809     pos_ratio += 1 << RATELIMIT_CALC_SHIFT;
810
811     return clamp(pos_ratio, 0LL, 2LL << RATELIMIT_CALC_SHIFT);
812 }

```

Listing 10-87: mm/page-writeback.c: pos_ratio_polynom()

Importantly, note that lengths are taken in order to maintain this calculation using integers (the kernel cannot sensibly perform floating point operations), therefore to work around the fact that this ratio is inevitably a floating point multiplier of a thread's dirty rate, we bit-shift the calculation by `RATELIMIT_CALC_SHIFT` (10 bits), shifting back again once the calculation is complete.

The function calculates the ratio to be $1 + \left(\frac{\text{setpoint} - \text{dirty}}{\text{limit} - \text{setpoint}}\right)^3$, where `dirty` is the total number of dirty pages, and `limit` the dirty threshold.

The result of this function is explicitly clamped between zero and two, matching the required range of the value.

The ratio indicates the divergence between the number of dirty pages and the setpoint, as a proportion of the part of the global dirty control scope range between the set point and the dirty threshold, i.e. scaling higher if this is small relative to if this is large.

If dirty pages are in excess of the setpoint the second part of this equation will be negative, causing the dirty rate to be scaled *down*, otherwise it will be positive, causing the dirty throttle rate to be scaled *up*.

Returning to `wb_position_ratio()` in Listing 10-86 above, we observe that once this *global pos_ratio* value has been obtained, we must take into account the proportion of the struct `bdi_writeback-specific` dirty threshold struct `dirty_throttle_control->wb_thresh` to the global dirty threshold struct `dirty_throttle_control->thresh`.

We therefore obtain a scaled struct `bdi_writeback`-specific setpoint `wb_setpoint` equal to $setpoint \times \frac{wb\_thresh}{thresh}$, which is then used to scale the position ratio accordingly.

Again we do not dwell the mathematical details of this function as they are somewhat out of the scope of the book.

10.19 Dirty Limits

As we’ve examined both global and struct `bdi_writeback`-specific threshold values, let’s look at how these are calculated. The global values are calculated in `domain_dirty_limits()` as shown in Listing 10-88 (eliding out of scope cgroup writeback and tracing logic).

```

365 /**
366  * domain_dirty_limits - calculate thresh and bg_thresh for a wb_domain
367  * @dtc: dirty_throttle_control of interest
368  *
369  * Calculate @dtc->thresh and ->bg_thresh considering
370  * vm_dirty_{bytes|ratio} and dirty_background_{bytes|ratio}. The caller
371  * must ensure that @dtc->avail is set before calling this function. The
372  * dirty limits will be lifted by 1/4 for real-time tasks.
373  */
374 static void domain_dirty_limits(struct dirty_throttle_control *dtc)
375 {
376     const unsigned long available_memory = dtc->avail;
377     :
378     unsigned long bytes = vm_dirty_bytes;
379     unsigned long bg_bytes = dirty_background_bytes;
380     /* convert ratios to per-PAGE_SIZE for higher precision */
381     unsigned long ratio = (vm_dirty_ratio * PAGE_SIZE) / 100;
382     unsigned long bg_ratio = (dirty_background_ratio * PAGE_SIZE) / 100;
383     unsigned long thresh;
384     unsigned long bg_thresh;
385     struct task_struct *tsk;
386     :
407     if (bytes)
408         thresh = DIV_ROUND_UP(bytes, PAGE_SIZE);
409     else
410         thresh = (ratio * available_memory) / PAGE_SIZE;
411
412     if (bg_bytes)
413         bg_thresh = DIV_ROUND_UP(bg_bytes, PAGE_SIZE);
414     else
415         bg_thresh = (bg_ratio * available_memory) / PAGE_SIZE;
416
417     if (bg_thresh >= thresh)
418         bg_thresh = thresh / 2;
419     tsk = current;

```

```

420         if (rt_task(tsk)) {
421             bg_thresh += bg_thresh / 4 + global_wb_domain.dirty_limit /
32;
422             thresh += thresh / 4 + global_wb_domain.dirty_limit / 32;
423         }
424         dtc->thresh = thresh;
425         dtc->bg_thresh = bg_thresh;
426         :
430     }

```

Listing 10-88: mm/page-writeback.c: domain_dirty_limits()

This retrieves the `vm.dirty_bytes` from the `vm_dirty_bytes` global variable and places it in `bytes`, `vm.dirty_background_bytes` from `dirty_background_bytes` and places it in `bg_bytes`, `vm.dirty_ratio` from `vm_dirty_ratio` and places it in `ratio` and `vm.dirty_background_ratio` from `dirty_background_ratio` and places it in `bg_ratio`.

The code then applies these tunables—if `vm.dirty_bytes` was specified then the threshold (expressed in pages) is calculated to simply be this value divided by and aligned to page size. Otherwise if `vm.dirty_ratio` was specified instead, this ratio is taken and multiplied by `struct dirty_throttle_control->avail` (which has previously been set to the number of available, dirtyable bytes as calculated by `global_dirtyable_memory()` shown in Listing 10-78, and divided by page size to obtain threshold page count stored in `thresh`.

Equally the same process is applied to the background threshold value which is stored in `bg_thresh`.

Finally, we consider the case where background threshold is erroneously higher than the dirty threshold, which cannot be permitted. In this instance we set the background threshold to half of dirty threshold as a sensible compromise.

There is some special handling for threads with real-time priority where the thresholds are increased, before assigning the `thresh` and `bg_thresh` fields of `domain_dirty_limits()` to the values calculated as described above.

We determine `struct bdi_writeback`-specific threshold values in `wb_dirty_limits()` which we explore in Listing 10-89.

```

1509 static inline void wb_dirty_limits(struct dirty_throttle_control *dtc)
1510 {
1511     struct bdi_writeback *wb = dtc->wb;
1512     unsigned long wb_reclaimable;
1513
1514     /*
1515      * wb_thresh is not treated as some limiting factor as
1516      * dirty_thresh, due to reasons
1517      * - in JBOD setup, wb_thresh can fluctuate a lot
1518      * - in a system with HDD and USB key, the USB key may somehow
1519      *   go into state (wb_dirty >> wb_thresh) either because
1520      *   wb_dirty starts high, or because wb_thresh drops low.

```



```

1521      * In this case we don't want to hard throttle the USB key
1522      * dirtiers for 100 seconds until wb_dirty drops under
1523      * wb_thresh. Instead the auxiliary wb control line in
1524      * wb_position_ratio() will let the dirtier task progress
1525      * at some rate <= (write_bw / 2) for bringing down wb_dirty.
1526      */
1527      dtc->wb_thresh = __wb_calc_thresh(dtc);
1528      dtc->wb_bg_thresh = dtc->thresh ?
1529          div_u64((u64)dtc->wb_thresh * dtc->bg_thresh, dtc->thresh) :
1530          0;
1531      /*
1532      * In order to avoid the stacked BDI deadlock we need
1533      * to ensure we accurately count the 'dirty' pages when
1534      * the threshold is low.
1535      *
1536      * Otherwise it would be possible to get thresh+n pages
1537      * reported dirty, even though there are thresh-m pages
1538      * actually dirty; with m+n sitting in the percpu
1539      * deltas.
1540      */
1541      if (dtc->wb_thresh < 2 * wb_stat_error()) {
1542          wb_reclaimable = wb_stat_sum(wb, WB_RECLAIMABLE);
1543          dtc->wb_dirty = wb_reclaimable + wb_stat_sum(wb, WB_WRITEBACK)
1544      ;
1545      } else {
1546          wb_reclaimable = wb_stat(wb, WB_RECLAIMABLE);
1547          dtc->wb_dirty = wb_reclaimable + wb_stat(wb, WB_WRITEBACK);
1548      }

```

Listing 10-89: mm/page-writeback.c: *wb_dirty_limits()*

This defers the calculation of the struct `bdi_writeback`-specified dirty threshold to `__wb_calc_thresh()` which we examine shortly in Listing 10-90. After doing so, we simply calculate the background equivalent, struct `dirty_throttle_control->wb_bg_thresh` to be equal to the proportion between global background and dirty threshold multiplied by `wb_thresh`.

After this we determine the count of dirty pages associated with the struct `bdi_writeback` object, using the count of dirtied folios associated with the writeback object via the `WB_RECLAIMABLE` statistic summed with the number of pages undergoing writeback (mirroring the global calculation), using `WB_WRITEBACK`.

We are cautious if the threshold is especially low in which case we do not accept the batch error associated with these statistics being updated in `WB_STAT_BATCH` increments (as indicated by `wb_stat_error()`). If so we use `wb_stat_sum()` rather than `wb_stat()` to force the summing of each per-CPU value rather than taking the typical batch-updated value.

We store this dirty count in struct `dirty_throttle_control->wb_dirty`.

Let's now examine `__wb_calc_thresh()` which performs the actual per-struct `bdi_writeback` dirty threshold calculation in Listing 10-90.

```

728 /**
729  * __wb_calc_thresh - @wb's share of dirty throttling threshold
730  * @dte: dirty_throttle_context of interest
731  *
732  * Note that balance_dirty_pages() will only seriously take it as a hard limit
733  * when sleeping max_pause per page is not enough to keep the dirty pages
734  * under
735  * control. For example, when the device is completely stalled due to some
736  * error
737  * conditions, or when there are 1000 dd tasks writing to a slow 10MB/s USB
738  * key.
739  * In the other normal situations, it acts more gently by throttling the tasks
740  * more (rather than completely block them) when the wb dirty pages go high.
741  *
742  * It allocates high/low dirty limits to fast/slow devices, in order to
743  * prevent
744  * - starving fast devices
745  * - piling up dirty pages (that will take long time to sync) on slow devices
746  *
747  * The wb's share of dirty limit will be adapting to its throughput and
748  * bounded by the bdi->min_ratio and/or bdi->max_ratio parameters, if set.
749  *
750  * Return: @wb's dirty limit in pages. The term "dirty" in the context of
751  * dirty balancing includes all PG_dirty and PG_writeback pages.
752  */
753 static unsigned long __wb_calc_thresh(struct dirty_throttle_control *dte)
754 {
755     struct wb_domain *dom = dte_dom(dte);
756     unsigned long thresh = dte->thresh;
757     u64 wb_thresh;
758     unsigned long numerator, denominator;
759     unsigned long wb_min_ratio, wb_max_ratio;
760
761     /*
762      * Calculate this BDI's share of the thresh ratio.
763      */
764     fprop_fraction_percpu(&dom->completions, dte->wb_completions,
765                          &numerator, &denominator);
766
767     wb_thresh = (thresh * (100 - bdi_min_ratio)) / 100;
768     wb_thresh *= numerator;
769     wb_thresh = div64_ul(wb_thresh, denominator);
770
771     wb_min_max_ratio(dte->wb, &wb_min_ratio, &wb_max_ratio);

```

```

768
769     wb_thresh += (thresh * wb_min_ratio) / 100;
770     if (wb_thresh > (thresh * wb_max_ratio) / 100)
771         wb_thresh = thresh * wb_max_ratio / 100;
772
773     return wb_thresh;
774 }

```

Listing 10-90: mm/page-writeback.c: `__wb_calc_thresh()`

This obtains a fraction representing the proportion of struct `dirty_throttle_control->wb_completions` to struct `wb_domain->completions`, obtaining the fraction's numerator and denominator. Again we disregard the cgroup logic, so here we assume that `dtc_dom()` returns `global_wb_domain` which represents total system statistics.

We fundamentally determine the struct `bdi_writeback`-specific dirty threshold to be equal to $wb\_thresh = thresh \times \frac{wb\_completions}{completions}$, where `thresh` is the system-wide dirty threshold, `wb_completions` the number of pages which completed writeback for this struct `bdi_writeback` and `completions` the number of pages which completed writeback in the system.

This value is additionally modified to account for BDI minimum and maximum ratio sysfs tunables described in Documentation/ABI/testing/sysfs-class-bdi tunables documentation. The minimum and maximum ratios default to 0% and 100% respectively, so unless these are specified to be otherwise, this will have no effect.

10.20 Dirty Poll Interval

Let's briefly examine `dirty_poll_interval()`, which determines the struct `task_struct->nr_dirtied_pause` value for freerunning threads. This is shown in Listing 10-91.

```

1398 /*
1399  * After a task dirtied this many pages, balance_dirty_pages_ratelimited()
1400  * will look to see if it needs to start dirty throttling.
1401  *
1402  * If dirty_poll_interval is too low, big NUMA machines will call the
1403  *   expensive
1404  * global_zone_page_state() too often. So scale it near-sqrt to the safety
1405  *   margin
1406  * (the number of pages we may dirty without exceeding the dirty limits).
1407  */
1408 static unsigned long dirty_poll_interval(unsigned long dirty,
1409                                         unsigned long thresh)
1410 {
1411     if (thresh > dirty)
1412         return 1UL << (ilog2(thresh - dirty) >> 1);
1413 }

```

```

1412         return 1;
1413     }

```

Listing 10-91: mm/page-writeback.c: dirty_poll_interval()

This determines the number of pages to be dirtied prior to $2^{\lg(thresh - dirty)/2}$ which is equivalent to $\sqrt{thresh - dirty}$, though constrained to an integer calculation form of this. If the threshold somehow exceeds the dirty threshold at this point we default the result to 1 to encourage dirty throttling to occur almost immediately.

10.21 Dirty Rate Limit

We have now examined the means by which a great many of the parameters upon which dirty throttling relies are obtained. The last remaining such parameters is are dirty rate limits which are determined by `wb_update_dirty_ratelimit()` upon dirty throttling, which we explore in Listing 10-92 (eliding out of scope BDI strict limit mode and tracing logic and some superfluous comments).

```

1166  /*
1167   * Maintain wb->dirty_ratelimit, the base dirty throttle rate.
1168   *
1169   * Normal wb tasks will be curbed at or below it in long term.
1170   * Obviously it should be around (write_bw / N) when there are N dd tasks.
1171   */
1172  static void wb_update_dirty_ratelimit(struct dirty_throttle_control *dte,
1173                                       unsigned long dirtied,
1174                                       unsigned long elapsed)
1175  {
1176      struct bdi_writeback *wb = dte->wb;
1177      unsigned long dirty = dte->dirty;
1178      unsigned long freerun = dirty_freerun_ceiling(dte->thresh, dte->
1179      bg_thresh);
1180      unsigned long limit = hard_dirty_limit(dte_dom(dte), dte->thresh);
1181      unsigned long setpoint = (freerun + limit) / 2;
1182      unsigned long write_bw = wb->avg_write_bandwidth;
1183      unsigned long dirty_ratelimit = wb->dirty_ratelimit;
1184      unsigned long dirty_rate;
1185      unsigned long task_ratelimit;
1186      unsigned long balanced_dirty_ratelimit;
1187      unsigned long step;
1188      unsigned long x;
1189      unsigned long shift;
1190
1191      /*
1192       * The dirty rate will match the writeout rate in long term, except
1193       * when dirty pages are truncated by userspace or re-dirtied by FS.
1194       */

```

```

1194         dirty_rate = (dirtied - wb->dirtied_stamp) * HZ / elapsed;
1195
1196     /*
1197      * task_ratelimit reflects each dd's dirty rate for the past 200ms.
1198      */
1199     task_ratelimit = (u64)dirty_ratelimit *
1200                     dtc->pos_ratio >> RATELIMIT_CALC_SHIFT
1201     ;
1202
1203     task_ratelimit++; /* it helps rampup dirty_ratelimit from tiny values
1204     */
1205
1206     /*
1207      * A linear estimation of the "balanced" throttle rate. The theory is,
1208      * if there are N dd tasks, each throttled at task_ratelimit, the wb's
1209      * dirty_rate will be measured to be (N * task_ratelimit). So the
1210      below
1211      * formula will yield the balanced rate limit (write_bw / N).
1212      *
1213      * Note that the expanded form is not a pure rate feedback:
1214      *      rate_(i+1) = rate_(i) * (write_bw / dirty_rate)
1215      (1)
1216      * but also takes pos_ratio into account:
1217      *      rate_(i+1) = rate_(i) * (write_bw / dirty_rate) * pos_ratio
1218      (2)
1219      :
1220      * So we end up using (2) to always keep
1221      *      rate_(i+1) ~= (write_bw / N)
1222      (8)
1223      * regardless of the value of pos_ratio. As long as (8) is satisfied,
1224      * pos_ratio is able to drive itself to 1.0, which is not only where
1225      * the dirty count meet the setpoint, but also where the slope of
1226      * pos_ratio is most flat and hence task_ratelimit is least fluctuated
1227      .
1228      */
1229     balanced_dirty_ratelimit = div_u64((u64)task_ratelimit * write_bw,
1230                                         dirty_rate | 1);
1231
1232     /*
1233      * balanced_dirty_ratelimit ~= (write_bw / N) <= write_bw
1234      */
1235     if (unlikely(balanced_dirty_ratelimit > write_bw))
1236         balanced_dirty_ratelimit = write_bw;
1237
1238     /*
1239      * We could safely do this and return immediately:
1240      *
1241      *      wb->dirty_ratelimit = balanced_dirty_ratelimit;
1242      */

```

Listing 10-92: mm/page-writeback.c: `wb_update_dirty_ratelimit()` basic calculation

We start by examining the core dirty rate logic. The function is invoked by `__wb_update_bandwidth()` (see Listing 10-82) which is in turn invoked by `balance_dirty_pages()` with the `update_ratelimit` parameter set, having set the appropriate fields in the struct `dirty_throttle_control` object, including the struct `bdi_writeback` object and associated thresholds.

We start by performing a simple calculation to obtain `dirty_rate`—we calculate $\frac{\text{dirtied} - \text{dirtied\_stamp}}{\text{elapsed}}$, where `dirtied` is equal to the struct `bdi_writeback->stats[WB_DIRTIED]` parameter snapshotted in `__wb_update_bandwidth()`, `dirtied_stamp` equal to struct `bdi_writeback->dirtied_stamp` and therefore the previous `dirtied` timestamp and `elapsed` is the time period since the last time we updated the dirty rate. This therefore gives us a raw, linear dirty rate since last updated.

Note that we also multiply by the global value `HZ`, this converts the elapsed time from the kernel-internal jiffies time metric to seconds such that this rate has units of dirty pages per second.

Note that, importantly, this is the per-struct `bdi_writeback` dirty rate rather than global dirty rate.

NOTE

Here I refer to threads in order specifically to differentiate between processes which may have multiple threads each writing back separately and the actual running threads to which this dirty throttling applies (and equally, to which the global current macro refers). Within the kernel, there is no process abstraction but rather a task one, embodied in a struct `task_struct` type. Each task is at a thread granularity rather than a process one.

The `dirty_ratelimit` variable is set to struct `bdi_writeback->dirty_ratelimit`, i.e. the previous *per-thread* dirty rate limit.

After having obtained this, we multiply this by position ratio, previously calculated in `wb_position_ratio()` (see Listing 10-86) which we place in `task_ratelimit`, incrementing by 1 to prevent truly tiny rate limits.

The position ratio is a modifier used to adjust the dirty rate limit such that it converges on the setpoint, so this `task_ratelimit` value represents the target, per-thread dirty rate.

Note that this is the *previous* dirty rate, now adjusted to converge on the setpoint, we need a means of sensibly obtaining the *current* dirty rate that should be applied per-thread.

This is where a magical inference takes place. Ideally, once at the setpoint, we would use all of the available write bandwidth provided by the block device (which may vary over time).

Thus if we had N processes and `write_bw` average write bandwidth (obtained from struct `bdi_writeback->avg_write_bandwidth`), we'd aim to maintain a per-thread dirty limit of $\frac{\text{write\_bw}}{N}$.

The issue is that we do not know N . Instead we infer it—if we assume that the dirty rate expressed in `dirty_rate` is achieving this aim, then we can infer

that $\text{dirty\_rate} = N \times \text{task\_ratelimit}$, as each thread which is writing back would be operating at this rate limit.

Since this is the `task_ratelimit` field which has the position ratio applied, this assumption will converge to being true as the ratio coefficient adjusts dirtying rate to achieve a steady state set point.

We can therefore determine the new *balanced* (among threads) dirty limit to be:

$$\frac{\text{task\_ratelimit} \times \text{write\_bw}}{\text{dirty\_rate}}$$

Which, since we have inferred that $\text{dirty\_rate} = N \times \text{task\_ratelimit}$ (or we are at least converging towards this) reduces to:

$$\frac{\text{task\_ratelimit} \times \text{write\_bw}}{\text{task\_ratelimit} \times N} = \frac{\text{write\_bw}}{N}$$

Therefore matching the dirty rate limit to the evenly-divided per-thread write bandwidth value. This forms the core of our calculation of the new dirty rate.

However, it is also useful to maintain a smoothed equivalent of this value which does not vary as much as this potentially will to ease calculations which might otherwise contain a great deal of variance, so we take steps to do so as shown in Listing 10-93.

```

1246      * However to get a more stable dirty_ratelimit, the below elaborated
1247      * code makes use of task_ratelimit to filter out singular points and
1248      * limit the step size.
1249      *
1250      * The below code essentially only uses the relative value of
1251      *
1252      *     task_ratelimit - dirty_ratelimit
1253      *     = (pos_ratio - 1) * dirty_ratelimit
1254      *
1255      * which reflects the direction and size of dirty position error.
1256      */
1257
1258     /*
1259      * dirty_ratelimit will follow balanced_dirty_ratelimit iff
1260      * task_ratelimit is on the same side of dirty_ratelimit, too.
1261      * For example, when
1262      * - dirty_ratelimit > balanced_dirty_ratelimit
1263      * - dirty_ratelimit > task_ratelimit (dirty pages are above setpoint)
1264      * lowering dirty_ratelimit will help meet both the position and rate
1265      * control targets. Otherwise, don't update dirty_ratelimit if it will
1266      * only help meet the rate target. After all, what the users
ultimately
1267      * feel and care are stable dirty rate and small position error.
1268      *
1269      * |task_ratelimit - dirty_ratelimit| is used to limit the step size
1270      * and filter out the singular points of balanced_dirty_ratelimit.
Which
1271      * keeps jumping around randomly and can even leap far away at times
1272      * due to the small 200ms estimation period of dirty_rate (we want to

```

```

1273     * keep that period small to reduce time lags).
1274     */
1275     step = 0;
1276     :
1296     if (dirty < setpoint) {
1297         x = min3(wb->balanced_dirty_ratelimit,
1298                 balanced_dirty_ratelimit, task_ratelimit);
1299         if (dirty_ratelimit < x)
1300             step = x - dirty_ratelimit;
1301     } else {
1302         x = max3(wb->balanced_dirty_ratelimit,
1303                 balanced_dirty_ratelimit, task_ratelimit);
1304         if (dirty_ratelimit > x)
1305             step = dirty_ratelimit - x;
1306     }
1307
1308     /*
1309     * Don't pursue 100% rate matching. It's impossible since the balanced
1310     * rate itself is constantly fluctuating. So decrease the track speed
1311     * when it gets close to the target. Helps eliminate pointless tremors
1312     */
1313     shift = dirty_ratelimit / (2 * step + 1);
1314     if (shift < BITS_PER_LONG)
1315         step = DIV_ROUND_UP(step >> shift, 8);
1316     else
1317         step = 0;
1318
1319     if (dirty_ratelimit < balanced_dirty_ratelimit)
1320         dirty_ratelimit += step;
1321     else
1322         dirty_ratelimit -= step;

```

Listing 10-93: mm/page-writeback.c: wb_update_dirty_ratelimit() refinement

As usual we do not delve too deeply into the mathematics for brevity, but we can see the steps taken here to smoothly adjust the dirty_ratelimit value.

Finally, we examine how these rate limit values are outputted in Listing 10-94.

```

1246     WRITE_ONCE(wb->dirty_ratelimit, max(dirty_ratelimit, 1UL));
1247     wb->balanced_dirty_ratelimit = balanced_dirty_ratelimit;
1248     :
1328 }

```

Listing 10-94: mm/page-writeback.c: wb_update_dirty_ratelimit() suffix

Here we store the raw balanced dirty rate balanced_dirty_ratelimit in struct bdi_writeback->balanced_dirty_ratelimit and the smoothed dirty rate dirty_ratelimit in struct bdi_writeback->dirty_ratelimit.

10.22 Maximum and Minimum Pause Time

In order to prevent threads from entirely freezing and to avoid sleeps so long writeback falls off altogether, we specify a maximum task pause period, calculated in `wb_max_pause()`, which we explore in Listing ??.

```

1415 static unsigned long wb_max_pause(struct bdi_writeback *wb,
1416                                 unsigned long wb_dirty)
1417 {
1418     unsigned long bw = READ_ONCE(wb->avg_write_bandwidth);
1419     unsigned long t;
1420
1421     /*
1422      * Limit pause time for small memory systems. If sleeping for too long
1423      * time, a small pool of dirty/writeback pages may go empty and disk
1424      * idle.
1425      *
1426      * 8 serves as the safety ratio.
1427      */
1428     t = wb_dirty / (1 + bw / roundup_pow_of_two(1 + HZ / 8));
1429     t++;
1430
1431     return min_t(unsigned long, t, MAX_PAUSE);
1432 }

```

Listing 10-95: `mm/page-writeback.c: wb_max_pause()`

This is always capped by `MAX_PAUSE`, which is hardcoded to 200ms, so we are guaranteed that the pause time will never exceed this amount of time.

The function is passed the `wb_dirty` parameter which is derived from `struct dirty_throttle_control->wb_dirty` and equal to the number of dirty and writeback pages associated with the `struct bdi_writeback` object.

The `bw` variable is set to the `struct bdi_writeback->avg_write_bandwidth` value, determined in `wb_update_write_bandwidth()` (see Listing 10-85).

Note that the offset-by-1 values are in place to avoid divide-by-zero here. Ignoring these we see that this value ultimately amounts to a time period of 125ms, scaled by $\frac{wb\_dirty}{bw}$, i.e. the fraction of dirty memory memory scaled by write bandwidth for this `struct bdi_writeback`.

This handles machines with very small memory which might otherwise pause so long that block devices go idle.

Again the use of the `HZ` constant is to convert this value into units of kernel-internal jiffies.

This calculation is simple relative to that for the *minimum* pause, which is determined by `wb_min_pause()`, which we examine in Listing 10-96 (eliding some of the longer comments which, while useful, take up rather a lot of space).

```

1434 static long wb_min_pause(struct bdi_writeback *wb,

```

```

1435         long max_pause,
1436         unsigned long task_ratelimit,
1437         unsigned long dirty_ratelimit,
1438         int *nr_dirtied_pause)
1439     {
1440         long hi = ilog2(READ_ONCE(wb->avg_write_bandwidth));
1441         long lo = ilog2(READ_ONCE(wb->dirty_ratelimit));
1442         long t;        /* target pause */
1443         long pause;    /* estimated next pause */
1444         int pages;     /* target nr_dirtied_pause */
1445
1446         /* target for 10ms pause on 1-dd case */
1447         t = max(1, HZ / 100);
1448         :
1449
1450         if (hi > lo)
1451             t += (hi - lo) * (10 * HZ) / 1024;
1452         :
1453
1454         t = min(t, 1 + max_pause / 2);
1455         pages = dirty_ratelimit * t / roundup_pow_of_two(HZ);
1456         :
1457
1458         if (pages < DIRTY_POLL_THRESH) {
1459             t = max_pause;
1460             pages = dirty_ratelimit * t / roundup_pow_of_two(HZ);
1461             if (pages > DIRTY_POLL_THRESH) {
1462                 pages = DIRTY_POLL_THRESH;
1463                 t = HZ * DIRTY_POLL_THRESH / dirty_ratelimit;
1464             }
1465         }
1466
1467         pause = HZ * pages / (task_ratelimit + 1);
1468         if (pause > max_pause) {
1469             t = max_pause;
1470             pages = task_ratelimit * t / roundup_pow_of_two(HZ);
1471         }
1472
1473         *nr_dirtied_pause = pages;
1474         /*
1475          * The minimal pause time will normally be half the target pause time.
1476          */
1477         return pages >= DIRTY_POLL_THRESH ? 1 + t / 2 : t;
1478     }

```

Listing 10-96: mm/page-writeback.c: `wb_min_pause()`

We establish a baseline of 10ms, which we initially scale up using logarithms such that this time is multiplied by M for 2^M concurrent threads, ensuring that each thread receives sufficient time to dirty pages, while scaling

in such a way that we don't slow things down. We clamp this to a maximum of half the previously determined maximum pause time.

Next we determine how many pages would have been dirtied had the thread not been paused using the passed in per-thread dirty rate limit `dirty_ratelimit` (again utilising the `HZ` constant to convert between jiffies and seconds in the rate value) and place this value in pages.

We maintain a minimum dirty threshold of `DIRTY_POLL_THRESH` pages, which is hard-coded to 128 KiB, i.e. 32 pages for systems with 4 KiB page size.

This is to prevent threads which are writing back very slowly from triggering the dirty throttling logic too often. If this threshold is not breached, then we revert to the maximum pause time to cause the thread to dirty more pages before triggering throttling again.

We output the number of pages from the function which is then placed into `struct task_struct->nr_dirtied_pause`, which is used by `balance_dirty_pages_ratelimited()` (see Listing 10-79) to rate limit invocations of `balance_dirty_pages()` which we explore in Listing 10-97 below.

Finally this function will typically only return with *half* of the target pause time if we've exceeded `DIRTY_POLL_THRESH` pages worth of pause time.

This allows processes to operate for a period of time without triggering dirty throttling (as they will be rate limited) and reduces latency.

10.23 Core Dirty Throttling

The machinery discussed so far has all been in service of, and largely invoked by, the core dirty throttle mechanism implemented in `balance_dirty_pages()` which we start examining in Listing 10-97 (eliding out of scope `cgroup`, tracing, strict BDI limit, flag handling logic and `PF_LOCAL_THROTTLE` logic).

This function is invoked by `balance_dirty_pages_ratelimited()` after pages have been dirtied (which we examined above in Listing 10-79) and is passed the relevant `struct bdi_writeback` object, the number of pages just dirtied (equal to the current thread's `struct task_struct->nr_dirtied`) and flags (these flags are only used by an out of scope caller so we can ignore them).

```

1550 /*
1551  * balance_dirty_pages() must be called by processes which are generating
1552    dirty
1553  * data. It looks at the number of dirty pages in the machine and will force
1554  * the caller to wait once crossing the (background_thresh + dirty_thresh) /
1555    2.
1556  * If we're over `background_thresh' then the writeback threads are woken to
1557  * perform some writeout.
1558  */
1559 static int balance_dirty_pages(struct bdi_writeback *wb,
1560                               unsigned long pages_dirtied, unsigned int flags
1561                               )
1562 {

```

```

1560     struct dirty_throttle_control gdtc_stor = { GDTC_INIT(wb) };
1561     :
1562     struct dirty_throttle_control * const gdtc = &gdtc_stor;
1563     :
1564     struct dirty_throttle_control *sdtc;
1565     unsigned long nr_reclaimable;  /* = file_dirty */
1566     long period;
1567     long pause;
1568     long max_pause;
1569     long min_pause;
1570     int nr_dirtied_pause;
1571     bool dirty_exceeded = false;
1572     unsigned long task_ratelimit;
1573     unsigned long dirty_ratelimit;
1574     struct backing_dev_info *bdi = wb->bdi;
1575     :
1576     unsigned long start_time = jiffies;
1577     int ret = 0;

```

Listing 10-97: mm/page-writeback.c: balance_dirty_pages() Prelude

We start by establishing a struct `dirty_throttle_control` object which we use to thread state through the process of dirty throttling, which is initialised by `GDTC_INIT()`.

Note that we also keep a track of the start of the process in the `start_time` parameter.

After this, we start an infinite loop, one which is typically exited without repeating. We explore the beginning of this loop in Listing 10-98.

```

1580     for (;;) {
1581         unsigned long now = jiffies;
1582         unsigned long dirty, thresh, bg_thresh;
1583         :
1584         nr_reclaimable = global_node_page_state(NR_FILE_DIRTY);
1585         gdtc->avail = global_dirtyable_memory();
1586         gdtc->dirty = nr_reclaimable + global_node_page_state(
1587             NR_WRITEBACK);
1588         :
1589         domain_dirty_limits(gdtc);
1590         :
1591         dirty = gdtc->dirty;
1592         thresh = gdtc->thresh;
1593         bg_thresh = gdtc->bg_thresh;
1594         :
1595         /*
1596          * In laptop mode, we wait until hitting the higher threshold
1597          * before starting background writeout, and then write out all
1598          * the way down to the lower threshold. So slow writers cause
1599          * minimal disk activity.

```

```

1636      *
1637      * In normal mode, we start background writeout at the lower
1638      * background_thresh, to keep the amount of dirty memory low.
1639      */
1640      if (!laptop_mode && nr_reclaimable > gdtc->bg_thresh &&
1641          !writeback_in_progress(wb))
1642          wb_start_background_writeback(wb);

```

Listing 10-98: mm/page-writeback.c: balance_dirty_pages() Initial Checks

We set the struct `dirty_throttle_control->avail` field to the total global dirtyable memory pages value obtained by `global_dirtyable_memory()` as shown in Listing 10-78, and the total number of dirty and writeback pages in the `dirty` field.

We then establish the global dirty and background threshold via `domain_dirty_limits()` as described in Listing 10-88.

We set `dirty`, `thresh` and `bg_thresh` parameters to the struct `dirty_throttle_control` fields of the same name just established.

After this we, if not in laptop mode, dirty pages exceed the now established background threshold variable and writeback is *not* in progress, we start background writeback via `wb_start_background_writeback()` as discussed in Section 10.11.

Note that we do not examine laptop mode closely, however as can be observed here this optionally delays background writeback for laptops in order to preserve battery life, and can be enabled via the `vm.laptop_mode` tunable.

Next, we gather the statistics previously discussed in order to be able to perform dirty throttling, which we examine in Listing 10-99.

```

1644      /*
1645      * Throttle it only when the background writeback cannot
1646      * catch-up. This avoids (excessively) small writeouts
1647      * when the wb limits are ramping up in case of !strictlimit.
1648      */
1649      if (dirty <= dirty_freerun_ceiling(thresh, bg_thresh) &&
1650          :
1651          :
1652          :
1653          :
1654          :
1655          :
1656          :
1657          :
1658          :
1659          :
1660          :
1661          :
1662          :
1663          :
1664          :
1665          :
1666          :
1667          :
1668          :
1669          :
1670          :
1671          :
1672          :
1673          :
1674          :
1675          :
1676          :
1677          :
1678          :
1679          :
1680          :
1681          :
1682          :
1683          :
1684          :
1685          :
1686          :
1687          :
1688          :
1689          :
1690          :
1691          :
1692          :
1693          :
1694          :
1695          :
1696          :
1697          :
1698          :
1699          :
1700          :
1701          :
1702          :
1703          :
1704          :
1705          :
1706          :
1707          :
1708          :
1709          :
1710          :
1711          :
1712          :
1713          :
1714          :
1715          :
1716          :
1717          :
1718          :
1719          :
1720          :
1721          :
1722          :
1723          :
1724          :
1725          :
1726          :
1727          :
1728          :
1729          :
1730          :
1731          :
1732          :
1733          :
1734          :
1735          :
1736          :
1737          :
1738          :
1739          :
1740          :
1741          :
1742          :
1743          :
1744          :
1745          :
1746          :
1747          :
1748          :
1749          :
1750          :
1751          :
1752          :
1753          :
1754          :
1755          :
1756          :
1757          :
1758          :
1759          :
1760          :
1761          :
1762          :
1763          :
1764          :
1765          :
1766          :
1767          :
1768          :
1769          :
1770          :
1771          :
1772          :
1773          :
1774          :
1775          :
1776          :
1777          :
1778          :
1779          :
1780          :
1781          :
1782          :
1783          :
1784          :
1785          :
1786          :
1787          :
1788          :
1789          :
1790          :
1791          :
1792          :
1793          :
1794          :
1795          :
1796          :
1797          :
1798          :
1799          :
1800          :
1801          :
1802          :
1803          :
1804          :
1805          :
1806          :
1807          :
1808          :
1809          :
1810          :
1811          :
1812          :
1813          :
1814          :
1815          :
1816          :
1817          :
1818          :
1819          :
1820          :
1821          :
1822          :
1823          :
1824          :
1825          :
1826          :
1827          :
1828          :
1829          :
1830          :
1831          :
1832          :
1833          :
1834          :
1835          :
1836          :
1837          :
1838          :
1839          :
1840          :
1841          :
1842          :
1843          :
1844          :
1845          :
1846          :
1847          :
1848          :
1849          :
1850          :
1851          :
1852          :
1853          :
1854          :
1855          :
1856          :
1857          :
1858          :
1859          :
1860          :
1861          :
1862          :
1863          :
1864          :
1865          :
1866          :
1867          :
1868          :
1869          :
1870          :
1871          :
1872          :
1873          :
1874          :
1875          :
1876          :
1877          :
1878          :
1879          :
1880          :
1881          :
1882          :
1883          :
1884          :
1885          :
1886          :
1887          :
1888          :
1889          :
1890          :
1891          :
1892          :
1893          :
1894          :
1895          :
1896          :
1897          :
1898          :
1899          :
1900          :
1901          :
1902          :
1903          :
1904          :
1905          :
1906          :
1907          :
1908          :
1909          :
1910          :
1911          :
1912          :
1913          :
1914          :
1915          :
1916          :
1917          :
1918          :
1919          :
1920          :
1921          :
1922          :
1923          :
1924          :
1925          :
1926          :
1927          :
1928          :
1929          :
1930          :
1931          :
1932          :
1933          :
1934          :
1935          :
1936          :
1937          :
1938          :
1939          :
1940          :
1941          :
1942          :
1943          :
1944          :
1945          :
1946          :
1947          :
1948          :
1949          :
1950          :
1951          :
1952          :
1953          :
1954          :
1955          :
1956          :
1957          :
1958          :
1959          :
1960          :
1961          :
1962          :
1963          :
1964          :
1965          :
1966          :
1967          :
1968          :
1969          :
1970          :
1971          :
1972          :
1973          :
1974          :
1975          :
1976          :
1977          :
1978          :
1979          :
1980          :
1981          :
1982          :
1983          :
1984          :
1985          :
1986          :
1987          :
1988          :
1989          :
1990          :
1991          :
1992          :
1993          :
1994          :
1995          :
1996          :
1997          :
1998          :
1999          :
2000          :
2001          :
2002          :
2003          :
2004          :
2005          :
2006          :
2007          :
2008          :
2009          :
2010          :
2011          :
2012          :
2013          :
2014          :
2015          :
2016          :
2017          :
2018          :
2019          :
2020          :
2021          :
2022          :
2023          :
2024          :
2025          :
2026          :
2027          :
2028          :
2029          :
2030          :
2031          :
2032          :
2033          :
2034          :
2035          :
2036          :
2037          :
2038          :
2039          :
2040          :
2041          :
2042          :
2043          :
2044          :
2045          :
2046          :
2047          :
2048          :
2049          :
2050          :
2051          :
2052          :
2053          :
2054          :
2055          :
2056          :
2057          :
2058          :
2059          :
2060          :
2061          :
2062          :
2063          :
2064          :
2065          :
2066          :
2067          :
2068          :
2069          :
2070          :
2071          :
2072          :
2073          :
2074          :
2075          :
2076          :
2077          :
2078          :
2079          :
2080          :
2081          :
2082          :
2083          :
2084          :
2085          :
2086          :
2087          :
2088          :
2089          :
2090          :
2091          :
2092          :
2093          :
2094          :
2095          :
2096          :
2097          :
2098          :
2099          :
2100          :
2101          :
2102          :
2103          :
2104          :
2105          :
2106          :
2107          :
2108          :
2109          :
2110          :
2111          :
2112          :
2113          :
2114          :
2115          :
2116          :
2117          :
2118          :
2119          :
2120          :
2121          :
2122          :
2123          :
2124          :
2125          :
2126          :
2127          :
2128          :
2129          :
2130          :
2131          :
2132          :
2133          :
2134          :
2135          :
2136          :
2137          :
2138          :
2139          :
2140          :
2141          :
2142          :
2143          :
2144          :
2145          :
2146          :
2147          :
2148          :
2149          :
2150          :
2151          :
2152          :
2153          :
2154          :
2155          :
2156          :
2157          :
2158          :
2159          :
2160          :
2161          :
2162          :
2163          :
2164          :
2165          :
2166          :
2167          :
2168          :
2169          :
2170          :
2171          :
2172          :
2173          :
2174          :
2175          :
2176          :
2177          :
2178          :
2179          :
2180          :
2181          :
2182          :
2183          :
2184          :
2185          :
2186          :
2187          :
2188          :
2189          :
2190          :
2191          :
2192          :
2193          :
2194          :
2195          :
2196          :
2197          :
2198          :
2199          :
2200          :
2201          :
2202          :
2203          :
2204          :
2205          :
2206          :
2207          :
2208          :
2209          :
2210          :
2211          :
2212          :
2213          :
2214          :
2215          :
2216          :
2217          :
2218          :
2219          :
2220          :
2221          :
2222          :
2223          :
2224          :
2225          :
2226          :
2227          :
2228          :
2229          :
2230          :
2231          :
2232          :
2233          :
2234          :
2235          :
2236          :
2237          :
2238          :
2239          :
2240          :
2241          :
2242          :
2243          :
2244          :
2245          :
2246          :
2247          :
2248          :
2249          :
2250          :
2251          :
2252          :
2253          :
2254          :
2255          :
2256          :
2257          :
2258          :
2259          :
2260          :
2261          :
2262          :
2263          :
2264          :
2265          :
2266          :
2267          :
2268          :
2269          :
2270          :
2271          :
2272          :
2273          :
2274          :
2275          :
2276          :
2277          :
2278          :
2279          :
2280          :
2281          :
2282          :
2283          :
2284          :
2285          :
2286          :
2287          :
2288          :
2289          :
2290          :
2291          :
2292          :
2293          :
2294          :
2295          :
2296          :
2297          :
2298          :
2299          :
2300          :
2301          :
2302          :
2303          :
2304          :
2305          :
2306          :
2307          :
2308          :
2309          :
2310          :
2311          :
2312          :
2313          :
2314          :
2315          :
2316          :
2317          :
2318          :
2319          :
2320          :
2321          :
2322          :
2323          :
2324          :
2325          :
2326          :
2327          :
2328          :
2329          :
2330          :
2331          :
2332          :
2333          :
2334          :
2335          :
2336          :
2337          :
2338          :
2339          :
2340          :
2341          :
2342          :
2343          :
2344          :
2345          :
2346          :
2347          :
2348          :
2349          :
2350          :
2351          :
2352          :
2353          :
2354          :
2355          :
2356          :
2357          :
2358          :
2359          :
2360          :
2361          :
2362          :
2363          :
2364          :
2365          :
2366          :
2367          :
2368          :
2369          :
2370          :
2371          :
2372          :
2373          :
2374          :
2375          :
2376          :
2377          :
2378          :
2379          :
2380          :
2381          :
2382          :
2383          :
2384          :
2385          :
2386          :
2387          :
2388          :
2389          :
2390          :
2391          :
2392          :
2393          :
2394          :
2395          :
2396          :
2397          :
2398          :
2399          :
2400          :
2401          :
2402          :
2403          :
2404          :
2405          :
2406          :
2407          :
2408          :
2409          :
2410          :
2411          :
2412          :
2413          :
2414          :
2415          :
2416          :
2417          :
2418          :
2419          :
2420          :
2421          :
2422          :
2423          :
2424          :
2425          :
2426          :
2427          :
2428          :
2429          :
2430          :
2431          :
2432          :
2433          :
2434          :
2435          :
2436          :
2437          :
2438          :
2439          :
2440          :
2441          :
2442          :
2443          :
2444          :
2445          :
2446          :
2447          :
2448          :
2449          :
2450          :
2451          :
2452          :
2453          :
2454          :
2455          :
2456          :
2457          :
2458          :
2459          :
2460          :
2461          :
2462          :
2463          :
2464          :
2465          :
2466          :
2467          :
2468          :
2469          :
2470          :
2471          :
2472          :
2473          :
2474          :
2475          :
2476          :
2477          :
2478          :
2479          :
2480          :
2481          :
2482          :
2483          :
2484          :
2485          :
2486          :
2487          :
2488          :
2489          :
2490          :
2491          :
2492          :
2493          :
2494          :
2495          :
2496          :
2497          :
2498          :
2499          :
2500          :
2501          :
2502          :
2503          :
2504          :
2505          :
2506          :
2507          :
2508          :
2509          :
2510          :
2511          :
2512          :
2513          :
2514          :
2515          :
2516          :
2517          :
2518          :
2519          :
2520          :
2521          :
2522          :
2523          :
2524          :
2525          :
2526          :
2527          :
2528          :
2529          :
2530          :
2531          :
2532          :
2533          :
2534          :
2535          :
2536          :
2537          :
2538          :
2539          :
2540          :
2541          :
2542          :
2543          :
2544          :
2545          :
2546          :
2547          :
2548          :
2549          :
2550          :
2551          :
2552          :
2553          :
2554          :
2555          :
2556          :
2557          :
2558          :
2559          :
2560          :
2561          :
2562          :
2563          :
2564          :
2565          :
2566          :
2567          :
2568          :
2569          :
2570          :
2571          :
2572          :
2573          :
2574          :
2575          :
2576          :
2577          :
2578          :
2579          :
2580          :
2581          :
2582          :
2583          :
2584          :
2585          :
2586          :
2587          :
2588          :
2589          :
2590          :
2591          :
2592          :
2593          :
2594          :
2595          :
2596          :
2597          :
2598          :
2599          :
2600          :
2601          :
2602          :
2603          :
2604          :
2605          :
2606          :
2607          :
2608          :
2609          :
2610          :
2611          :
2612          :
2613          :
2614          :
2615          :
2616          :
2617          :
2618          :
2619          :
2620          :
2621          :
2622          :
2623          :
2624          :
2625          :
2626          :
2627          :
2628          :
2629          :
2630          :
2631          :
2632          :
2633          :
2634          :
2635          :
2636          :
2637          :
2638          :
2639          :
2640          :
2641          :
2642          :
2643          :
2644          :
2645          :
2646          :
2647          :
2648          :
2649          :
2650          :
2651          :
2652          :
2653          :
2654          :
2655          :
2656          :
2657          :
2658          :
2659          :
2660          :
2661          :
2662          :
2663          :
2664          :
2665          :
2666          :
2667          :
2668          :
2669          :
2670          :
2671          :
2672          :
2673          :
2674          :
2675          :
2676          :
2677          :
2678          :
2679          :
2680          :
2681          :
2682          :
2683          :
2684          :
2685          :
2686          :
2687          :
2688          :
2689          :
2690          :
2691          :
2692          :
2693          :
2694          :
2695          :
2696          :
2697          :
2698          :
2699          :
2700          :
2701          :
2702          :
2703          :
2704          :
2705          :
2706          :
2707          :
2708          :
2709          :
2710          :
2711          :
2712          :
2713          :
2714          :
2715          :
2716          :
2717          :
2718          :
2719          :
2720          :
2721          :
2722          :
2723          :
2724          :
2725          :
2726          :
2727          :
2728          :
2729          :
2730          :
2731          :
2732          :
2733          :
2734          :
2735          :
2736          :
2737          :
2738          :
2739          :
2740          :
2741          :
2742          :
2743          :
2744          :
2745          :
2746          :
2747          :
2748          :
2749          :
2750          :
2751          :
2752          :
2753          :
2754          :
2755          :
2756          :
2757          :
2758          :
2759          :
2760          :
2761          :
2762          :
2763          :
2764          :
2765          :
2766          :
2767          :
2768          :
2769          :
2770          :
2771          :
2772          :
2773          :
2774          :
2775          :
2776          :
2777          :
2778          :
2779          :
2780          :
2781          :
2782          :
2783          :
2784          :
2785          :
2786          :
2787          :
2788          :
2789          :
2790          :
2791          :
2792          :
2793          :
2794          :
2795          :
2796          :
2797          :
2798          :
2799          :
2800          :
2801          :
2802          :
2803          :
2804          :
2805          :
2806          :
2807          :
2808          :
2809          :
2810          :
2811          :
2812          :
2813          :
2814          :
2815          :
2816          :
2817          :
2818          :
2819          :
2820          :
2821          :
2822          :
2823          :
2824          :
2825          :
2826          :
2827          :
2828          :
2829          :
2830          :
2831          :
2832          :
2833          :
2834          :
2835          :
2836          :
2837          :
2838          :
2839          :
2840          :
2841          :
2842          :
2843          :
2844          :
2845          :
2846          :
2847          :
2848          :
2849          :
2850          :
2851          :
2852          :
2853          :
2854          :
2855          :
2856          :
2857          :
2858          :
2859          :
2860          :
2861          :
2862          :
2863          :
2864          :
2865          :
2866          :
2867          :
2868          :
2869          :
2870          :
2871          :
2872          :
2873          :
2874          :
2875          :
2876          :
2877          :
2878          :
2879          :
2880          :
2881          :
2882          :
2883          :
2884          :
2885          :
2886          :
2887          :
2888          :
2889          :
2890          :
2891          :
2892          :
2893          :
2894          :
2895          :
2896          :
2897          :
2898          :
2899          :
2900          :
2901          :
2902          :
2903          :
2904          :
2905          :
2906          :
2907          :
2908          :
2909          :
2910          :
2911          :
2912          :
2913          :
2914          :
2915          :
2916          :
2917          :
2918          :
2919          :
2920          :
2921          :
2922          :
2923          :
2924          :
2925          :
2926          :
2927          :
2928          :
2929          :
2930          :
2931          :
2932          :
2933          :
2934          :
2935          :
2936          :
2937          :
2938          :
2939          :
2940          :
2941          :
2942          :
2943          :
2944          :
2945          :
2946          :
2947          :
2948          :
2949          :
2950          :
2951          :
2952          :
2953          :
2954          :
2955          :
2956          :
2957          :
2958          :
2959          :
2960          :
2961          :
2962          :
2963          :
2964          :
2965          :
2966          :
2967          :
2968          :
2969          :
2970          :
2971          :
2972          :
2973          :
2974          :
2975          :
2976          :
2977          :
2978          :
2979          :
2980          :
2981          :
2982          :
2983          :
2984          :
2985          :
2986          :
2987          :
2988          :
2989          :
2990          :
2991          :
2992          :
2993          :
2994          :
2995          :
2996          :
2997          :
2998          :
2999          :
3000          :
3001          :
3002          :
3003          :
3004          :
3005          :
3006          :
3007          :
3008          :
3009          :
3010          :
3011          :
3012          :
3013          :
3014          :
3015          :
3016          :
3017          :
3018          :
3019          :
3020          :
3021          :
3022          :
3023          :
3024          :
3025          :
3026          :
3027          :
3028          :
3029          :
3030          :
3031          :
3032          :
3033          :
3034          :
3035          :
3036          :
3037          :
3038          :
3039          :
3040          :
3041          :
3042          :
3043          :
3044          :
3045          :
3046          :
3047          :
3048          :
3049          :
3050          :
3051          :
3052          :
3053          :
3054          :
3055          :
3056          :
3057          :
3058          :
3059          :
3060          :
3061          :
3062          :
3063          :
3064          :
3065          :
3066          :
3067          :
3068          :
3069          :
3070          :
3071          :
3072          :
3073          :
3074          :
3075          :
3076          :
3077          :
3078          :
3079          :
3080          :
3081          :
3082          :
3083          :
3084          :
3085          :
3086          :
3087          :
3088          :
3089          :
3090          :
3091          :
3092          :
3093          :
3094          :
3095          :
3096          :
3097          :
3098          :
3099          :
3100          :
3101          :
3102          :
3103          :
3104          :
3105          :
3106          :
3107          :
3108          :
3109          :
3110          :
3111          :
3112          :
3113          :
3114          :
3115          :
3116          :
3117          :
3118          :
3119          :
3120          :
3121          :
3122          :
3123          :
3124          :
3125          :
3126          :
3127          :
3128          :
3129          :
3130          :
3131          :
3132          :
3133          :
3134          :
3135          :
3136          :
3137          :
3138          :
3139          :
3140          :
3141          :
3142          :
3143          :
3144          :
3145          :
3146          :
3147          :
3148          :
3149          :
3150          :
3151          :
3152          :
3153          :
3154          :
3155          :
3156          :
3157          :
3158          :
3159          :
3160          :
3161          :
3162          :
3163          :
3164          :
3165          :
3166          :
3167          :
3168          :
3169          :
3170          :
3171          :
3172          :
3173          :
3174          :
3175          :
3176          :
3177          :
3178          :
3179          :
3180          :
3181          :
3182          :
3183          :
3184          :
3185          :
3186          :
3187          :
3188          :
3189          :
3190          :
3191          :
3192          :
3193          :
3194          :
3195          :
3196          :
3197          :
3198          :
3199          :
3200          :
3201          :
3202          :
3203          :
3204          :
3205          :
3206          :
3207          :
3208          :
3209          :
3210          :
3211          :
3212          :
3213          :
3214          :
3215          :
3216          :
3217          :
3218          :
3219          :
3220          :
3221          :
3222          :
3223          :
3224          :
3225          :
3226          :
3227          :
3228          :
3229          :
3230          :
3231          :
3232          :
3233          :
3234          :
3235          :
3236          :
3237          :
3238          :
3239          :
3240          :
3241          :
3242          :
3243          :
3244          :
3245          :
3246          :
3247          :
3248          :
3249          :
3250          :
3251          :
3252          :
3253          :
3254          :
3255          :
3256          :
3257          :
3258          :
3259          :
3260          :
3261          :
3262          :
3263          :
3264          :
3265          :
3266          :
3267          :
3268          :
3269          :
3270          :
3271          :
3272          :
3273          :
3274          :
3275          :
3276          :
3277          :
3278          :
3279          :
3280          :
3281          :
3282          :
3283          :
3284          :
3285          :
3286          :
3287          :
3288          :
3289          :

```

```

1675         if (unlikely(!writeback_in_progress(wb)))
1676             wb_start_background_writeback(wb);
1677         :
1680         /*
1681          * Calculate global domain's pos_ratio and select the
1682          * global dtc by default.
1683          */
1684         :
1685         wb_dirty_limits(gdtc);
1686         :
1698         dirty_exceeded = (gdtc->wb_dirty > gdtc->wb_thresh) &&
1699             ((gdtc->dirty > gdtc->thresh) || strictlimit);
1700
1701         wb_position_ratio(gdtc);
1702         sdtc = gdtc;
1703         :
1733         if (dirty_exceeded != wb->dirty_exceeded)
1734             wb->dirty_exceeded = dirty_exceeded;
1735
1736         if (time_is_before_jiffies(READ_ONCE(wb->bw_time_stamp) +
1737             BANDWIDTH_INTERVAL))
1738             __wb_update_bandwidth(gdtc, mdtc, true);

```

Listing 10-99: mm/page-writeback.c: balance_dirty_pages() Gather Statistics

We start by determining whether the global number of dirty pages is within the freerun regime (see figure 10-13). In this case we do not pause the thread at all. This is regime in which systems typically operate most of the time.

In this instance we determine the poll interval to rate limit threads to via `dirty_poll_interval()` (see Listing ??). This is then assigned to `struct task_struct->nr_dirtied_pause` to achieve this rate limiting.

We then reset the `struct task_struct->dirty_paused_when` and `struct task_struct->nr_dirtied` fields, to indicate that we considered pausing the thread now and are resetting the number of dirtied pages since last pause.

After this the freerunning case exits the loop and returns.

If we are not in the freerunning regime, then we are in the global dirty control scope and must apply some throttling. We therefore proceed on this basis.

We start by determining `struct bdi_writeback`-specific dirty limits via `wb_dirty_limits()` which we previously explored in Listing 10-89.

After this we determine whether we have exceeded the dirty threshold for both the `struct bdi_writeback` and globally and place this in `dirty_exceeded`, which we also set in `struct bdi_writeback->dirty_exceeded` for purposes of rate-limiting.

We calculate the position ratio for the global `struct dirty_throttle_control` object via `wb_position_ratio()` (see Listing

10-86), and take a copy of the pointer to the global dirty throttle control object in `sdtc`.

Finally, if bandwidth updating is not idle (i.e. the struct `bdi_writeback` is still experiencing sufficiently regular writeback), as determined by checking whether the current time is less than the next point at which the bandwidth update work queue is scheduled to run, struct `bdi_writeback->bw_time_stamp + BANDWIDTH_INTERVAL`.

In this instance then `__wb_update_bandwidth()` is invoked (see Listing 10-82 with `update_ratelimit` set, causing `wb_update_dirty_ratelimit()` to be invoked (see Listing 10-92) updating the write bandwidth and dirty rate limit statistics.

Next we perform the actual pausing of the calling thread, explored in Listing 10-100.

```

1740      /* throttle according to the chosen dtc */
1741      dirty_ratelimit = READ_ONCE(wb->dirty_ratelimit);
1742      task_ratelimit = ((u64)dirty_ratelimit * sdtc->pos_ratio) >>
1743                      RATELIMIT_CALC_SHIFT;
1744      max_pause = wb_max_pause(wb, sdtc->wb_dirty);
1745      min_pause = wb_min_pause(wb, max_pause,
1746                               task_ratelimit, dirty_ratelimit,
1747                               &nr_dirtied_pause);
1748
1749      if (unlikely(task_ratelimit == 0)) {
1750          period = max_pause;
1751          pause = max_pause;
1752          goto pause;
1753      }
1754      period = HZ * pages_dirtied / task_ratelimit;
1755      pause = period;
1756      if (current->dirty_paused_when)
1757          pause -= now - current->dirty_paused_when;
1758      /*
1759       * For less than 1s think time (ext3/4 may block the dirtier
1760       * for up to 800ms from time to time on 1-HDD; so does xfs,
1761       * however at much less frequency), try to compensate it in
1762       * future periods by updating the virtual time; otherwise just
1763       * do a reset, as it may be a light dirtier.
1764       */
1765      if (pause < min_pause) {
1766          :
1778          if (pause < -HZ) {
1779              current->dirty_paused_when = now;
1780              current->nr_dirtied = 0;
1781          } else if (period) {
1782              current->dirty_paused_when += period;
1783              current->nr_dirtied = 0;
1784          } else if (current->nr_dirtied_pause <= pages_dirtied)

```

```

1785             current->nr_dirtied_pause += pages_dirtied;
1786             break;
1787         }
1788         if (unlikely(pause > max_pause)) {
1789             /* for occasional dropped task_ratelimit */
1790             now += min(pause - max_pause, max_pause);
1791             pause = max_pause;
1792         }
1793
1794 pause:
1795     :
1811     __set_current_state(TASK_KILLABLE);
1812     wb->dirty_sleep = now;
1813     io_schedule_timeout(pause);

```

Listing 10-100: mm/page-writeback.c: `balance_dirty_pages()` Thread Pausing

We establish the struct `bdi_writeback` dirty rate limit and place it in `dirty_ratelimit`, and scale it against the previously determined position ratio and place this value in `task_ratelimit`.

We then use these values to determine the maximum and minimum permitted thread pause times using `wb_max_pause()` (see Listing ??) and `wb_min_pause()` (see Listing 10-96).

If we find ourselves in the unlikely scenario where the rate limit is zero, we immediately set the maximum pause and jump immediately into applying it. This is the one scenario in which we actually continue in the loop and pause again right away if necessary if a thread is so out of control as to be dirtying at a rate which requires it.

Otherwise, we set period to $\frac{\text{pages\_dirtied}}{\text{task\_ratelimit}}$. The `pages_dirtied` parameter is set to the current thread's struct `task_struct->nr_dirtied` field, i.e. the number of pages dirtied since it was last paused. We multiply this result by HZ in order to convert from seconds to jiffies.

This value therefore continues (in jiffies) the amount of time it would have taken to dirty these pages at the rate limit.

We then obtain the time to pause by subtracting the current thread's struct `task_struct->dirty_paused_when` timestamp as, if we have dirtied faster than the limit would allow, this number will be positive, otherwise it will be negative.

We then examine the case of the pause being less than the minimum—if we are writing slower than a second's worth of dirtying, we simply reset this value and the struct `task_struct->dirty_paused_when` timestamp to now as the thread is so slow as to warrant being treated as if it were in a freerun state.

Next if the period is non-zero, indicating that the number of dirtied pages is relatively significant in comparison to the dirty limit, we sum it to the struct `task_struct->dirty_paused_when` field and reset the number of dirtied pages, essentially 'skipping' these pages.

Finally if the number of dirtied pages is very small, but is higher than the last determined minimum pause period, we increase this to allow for greater rate limiting.

All of the above (as with much of the kernel) are careful heuristic steps taken to tune the design of the algorithm to suit real workloads. And each cause the loop to exit when done. Note that the current thread's struct `task_struct->nr_dirtied_pause` field is *not* updated except in the last case, even though we have obtained a local copy of what this value should be set to.

Next we consider the case of the determined pause period exceeding the maximum pause value. In that case we offset the `now` variable by the difference in pause time (up to a maximum of the maximum specified pause time) which will be assigned to the struct `bdi_writeback->dirty_sleep` timestamp on sleep. This is a heuristic step for internal purposes and will have an impact for callers of `wb_recent_wait()`, however this is out of scope for the book.

Finally we actually perform the pausing of the process, marking it as killable so the sleep can handle a `SIGKILL` signal correctly, and executing the sleep via `io_schedule_timeout()`.

After this we perform some house keeping and determine whether to persist in the loop, as explored in Listing 10-101.

```

1740         current->dirty_paused_when = now + pause;
1741         current->nr_dirtied = 0;
1742         current->nr_dirtied_pause = nr_dirtied_pause;
1743
1744         /*
1745          * This is typically equal to (dirty < thresh) and can also
1746          * keep "1000+ dd on a slow USB stick" under control.
1747          */
1748         if (task_ratelimit)
1749             break;
1750
1751         /*
1752          * In the case of an unresponsive NFS server and the NFS dirty
1753          * pages exceeds dirty_thresh, give the other good wb's a pipe
1754          * to go through, so that tasks on them still remain
1755          responsive.
1756          *
1757          * In theory 1 page is enough to keep the consumer-producer
1758          * pipe going: the flusher cleans 1 page => the task dirties 1
1759          * more page. However wb_dirty has accounting errors. So use
1760          * the larger and more IO friendly wb_stat_error.
1761          */
1761         if (sdtc->wb_dirty <= wb_stat_error())
1762             break;
1763
1764         if (fatal_signal_pending(current))

```

```

1765             break;
1766         }
1767         return ret;
1768     }

```

Listing 10-101: mm/page-writeback.c: `balance_dirty_pages()` Suffix

Finally we reset the current thread's `struct task_struct->dirty_paused_when` and `nr_dirtied` fields as a pause just occurred, and set the `nr_dirtied_pause` field. To the value determined by `wb_min_pause()` (see Listing 10-96).

Next we examine whether the `task_ratelimit` value is non-zero—if so, as will almost certainly be the case, we exit the loop and the operation is complete.

If it is indeed zero, then we have a thread that is dirtying at a very high rate compared to the setpoint and which has thereby obtained a position ratio value of zero (as determined by `wb_position_ratio()` (see Listing 10-86).

We first check to see if the `struct dirty_throttle_control->wb_dirty` field indicates a very low number of dirtied pages, in which case we exit the loop early. Equally we exit should a fatal signal be pending (rendering the containing process subject to being torn down). In either case we exit.

Otherwise, in cases of extreme dirtying we repeat the procedure until we are able to bring the thread into line.

10.24 Writeback Chunk Size

When writing back, we determine the ‘chunk size’ i.e the number of pages to write back in one batched operation in `writeback_sb_inodes()` (see Listing 10-62) via `writeback_chunk_size()`, which we explore in Listing 10-102.

```

1732 static long writeback_chunk_size(struct bdi_writeback *wb,
1733                                struct wb_writeback_work *work)
1734 {
1735     long pages;
1736
1737     /*
1738      * WB_SYNC_ALL mode does livelock avoidance by syncing dirty
1739      * inodes/pages in one big loop. Setting wbc.nr_to_write=LONG_MAX
1740      * here avoids calling into writeback_inodes_wb() more than once.
1741      *
1742      * The intended call sequence for WB_SYNC_ALL writeback is:
1743      *
1744      *     wb_writeback()
1745      *     writeback_sb_inodes()      <== called only once
1746      *     write_cache_pages()       <== called once for each inode
1747      *                               (quickly) tag currently dirty pages
1748      *                               (maybe slowly) sync all tagged pages
1749      */
1750     if (work->sync_mode == WB_SYNC_ALL || work->tagged_writepages)
1751         pages = LONG_MAX;

```

```

1752     else {
1753         pages = min(wb->avg_write_bandwidth / 2,
1754                     global_wb_domain.dirty_limit / DIRTY_SCOPE);
1755         pages = min(pages, work->nr_pages);
1756         pages = round_down(pages + MIN_WRITEBACK_PAGES,
1757                             MIN_WRITEBACK_PAGES);
1758     }
1759
1760     return pages;
1761 }

```

Listing 10-102: fs/fs-writeback.c: writeback_chunk_size()

If the user specified that the writeback should be synchronised, then we apply no limit and writeback as much as we can. This is done, in part, to avoid live locks as the comment describes.

Otherwise we limit any one operation to half of the struct `bdi_writeback`'s average write bandwidth (i.e. that requiring 1/2 a second of writeback to occur), or if smaller, the dirty threshold scaled by `DIRTY_SCOPE`, i.e. limiting to 1/8 of the available dirtyable pages.

This is further limited to the struct `wb_writeback_work->nr_pages` count of pages to write, before being rounded down to the `MIN_WRITEBACK_PAGES` limit, which is set to 4 MiB worth of data (1,024 pages for systems with a 4 KiB page size).

All of this is designed to heuristically avoid issues with locking and to improve performance.

10.25 Background Writeback Threshold

When checking to determine whether the current struct `bdi_writeback` object has exceeded the background dirty threshold, we use `wb_over_bg_thresh()`, as invoked by `wb_check_background_flush()` and `wb_writeback()`.

We examine this in Listing 10-103, eliding out of scope cgroup handling.

```

1955 /**
1956  * wb_over_bg_thresh - does @wb need to be written back?
1957  * @wb: bdi_writeback of interest
1958  *
1959  * Determines whether background writeback should keep writing @wb or it's
1960  * clean enough.
1961  *
1962  * Return: %true if writeback should continue.
1963  */
1964 bool wb_over_bg_thresh(struct bdi_writeback *wb)
1965 {
1966     struct dirty_throttle_control gdtc_stor = { GDTC_INIT(wb) };
1967     :
1968     struct dirty_throttle_control * const gdtc = &gdtc_stor;
1969     :

```

```

1971     unsigned long reclaimable;
1972     unsigned long thresh;
1973
1974     /*
1975      * Similar to balance_dirty_pages() but ignores pages being written
1976      * as we're trying to decide whether to put more under writeback.
1977      */
1978     gdtc->avail = global_dirtyable_memory();
1979     gdtc->dirty = global_node_page_state(NR_FILE_DIRTY);
1980     domain_dirty_limits(gdtc);
1981
1982     if (gdtc->dirty > gdtc->bg_thresh)
1983         return true;
1984
1985     thresh = wb_calc_thresh(gdtc->wb, gdtc->bg_thresh);
1986     if (thresh < 2 * wb_stat_error())
1987         reclaimable = wb_stat_sum(wb, WB_RECLAIMABLE);
1988     else
1989         reclaimable = wb_stat(wb, WB_RECLAIMABLE);
1990
1991     if (reclaimable > thresh)
1992         return true;
1993     :
2015     return false;
2016 }

```

Listing 10-103: mm/page-writeback.c: `wb_over_bg_thresh()`

This function is something like a mini-`balance_dirty_pages()` (see Listing 10-97 above), only cut down in order to simply determine whether the background threshold has been exceeded.

A global struct `dirty_throttle_control` state object `gdtc` is established, and dirtyable memory assigned to `avail` via `global_dirtyable_memory()` (see Listing 10-78) and the number of dirty pages set to the global count of dirty files. Files being written back are not considered here, as they are in the general dirty throttling logic, as we are attempting to see whether we need to writeback further dirty pages.

We establish thresholds via `domain_dirty_limits()` as described in Listing 10-88, before simply seeing if the struct `dirty_throttle_control->dirty` value exceeds the background threshold just calculated. If so, then we return indicating that the threshold has been breached, as global dirty pages do exceed the global background threshold.

If this is not the case, we perform an additional check at the granularity of the struct `bdi_writeback` object.

We invoke `wb_calc_thresh()` to determine the struct `bdi_writeback`-specified dirty threshold, which we examine in Listing 10-104.

```

776 unsigned long wb_calc_thresh(struct bdi_writeback *wb, unsigned long thresh)
777 {

```

```

778     struct dirty_throttle_control gdtc = { GDTC_INIT(wb),
779                                           .thresh = thresh };
780     return __wb_calc_thresh(&gdtc);
781 }

```

Listing 10-104: mm/page-writeback.c: `wb_calc_thresh()`

This is simply a wrapper around `__wb_calc_thresh()` (see Listing 10-90), using a local stack variable for the struct `dirty_throttle_control` object to pass state.

The `WB_RECLAIMABLE` statistic we retrieve (being careful about low counts to avoid inaccuracy) indicates the number of dirtied pages (though not yet being written back).

If this exceeds the hard dirty threshold of the struct `bdi_writeback` object, we heuristically deem this worthy of background writeback and thus return indicating that the threshold has been breached, otherwise we do not.

11

RECLAIM AND MEMORY PRESSURE

Memory pressure arises when more memory is requested than can be currently provided without dipping into memory reserves. Reclaim is the means by which the kernel relieves memory pressure. This is critical, as Linux is profligate in caching file data in the page cache, permits memory overcommit (allowing users to map more memory than is physically available) and demand paging (actually allocating memory only when the memory is used), so, as a result, memory pressure might arise at any time. This chapter explores the reclaim mechanism in detail.

There are two forms of reclaim—indirect reclaim, which occurs when a zone from which memory is requested drops below the low watermark, and direct reclaim, which occurs when a zone drops below its minimum watermark.

See Section 2.4 in Chapter 2 for details of what nodes, zones and their watermarks are, but briefly they are NUMA concepts with each node containing memory with broadly equal access time to memory contained within it, and zones describe contiguous physical memory ranges which are main-

tained as distinct entities largely in order to account for hardware devices capable only of accessing memory within a particular range.

The relationship between memory usage, watermarks and the performance of reclaim is shown in Figure 2-12 and Chapter 2, however for convenience we reproduce it here in Figure 11-1.

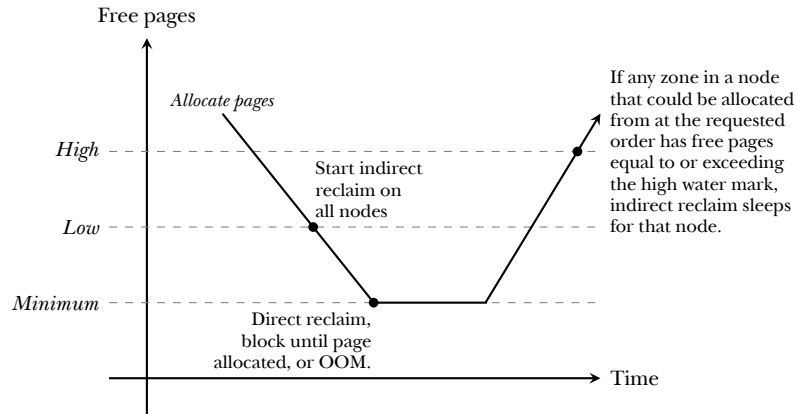


Figure 11-1: Zone Watermarks

We examine indirect and direct reclaim separately in Sections 11.4 and 11.3 respectively, before going on to examine the shared mechanism through which reclaim is progressed within the kernel in Section 11.5.

As we can see in Figure 2-13 in Section 2.8.1 in Chapter 2 on physical memory, regardless of the means by which the kernel invokes the physical allocation of a memory page, all roads lead to `__alloc_pages()` (as shown in Listing 2-52 and Figure 2-14).

`__alloc_pages()` attempts to allocate memory on the “fast path”, i.e. at or above the low watermark in an eligible zone, requiring neither indirect nor direct reclaim. If this fails for all valid zones, we then revert to the “slow path” where indirect or direct reclaim can be invoked as required.

We examine the allocation “slow path” in Section 11.1, as this is the entry point for reclaim in the kernel as a whole.

After this, we examine the LRU mechanism which the kernel uses to keep track of least recently used memory in order to decide what gets reclaimed, both that utilised for the page cache as well as that utilised for anonymous memory, in Section 11.2.

We then examine how the actual reclaim mechanism proceeds for indirect reclaim in Section 11.4 and direct reclaim in Section 11.3.

Finally, we examine folio batches, a means by which folios get stored in fixed arrays before being processed in a batch once each folio batch becomes full. We examine this in Section 11.7.

11.1 Physical Allocation Slow Path

When allocating via the “fast path”, i.e. from either the Per-CPU-Pages (see Section 2.7.3) or from a zone with its watermark at or above the `WMARK_LOW` level in `__alloc_pages()`, the kernel tries to do the least work possible to provide the memory.

However if all nodes and zones from which we could allocate have free pages beneath the `WMARK_LOW` level (accounting for reserves and allocation flags which adjust this as determined in `__zone_watermark_ok()`, as shown in Listing 2-59 and Section 2.8.2), we must take the “slow” path in `__alloc_pages_slowpath()`.

We examine the core of the logic of this function in Figure 11-2.

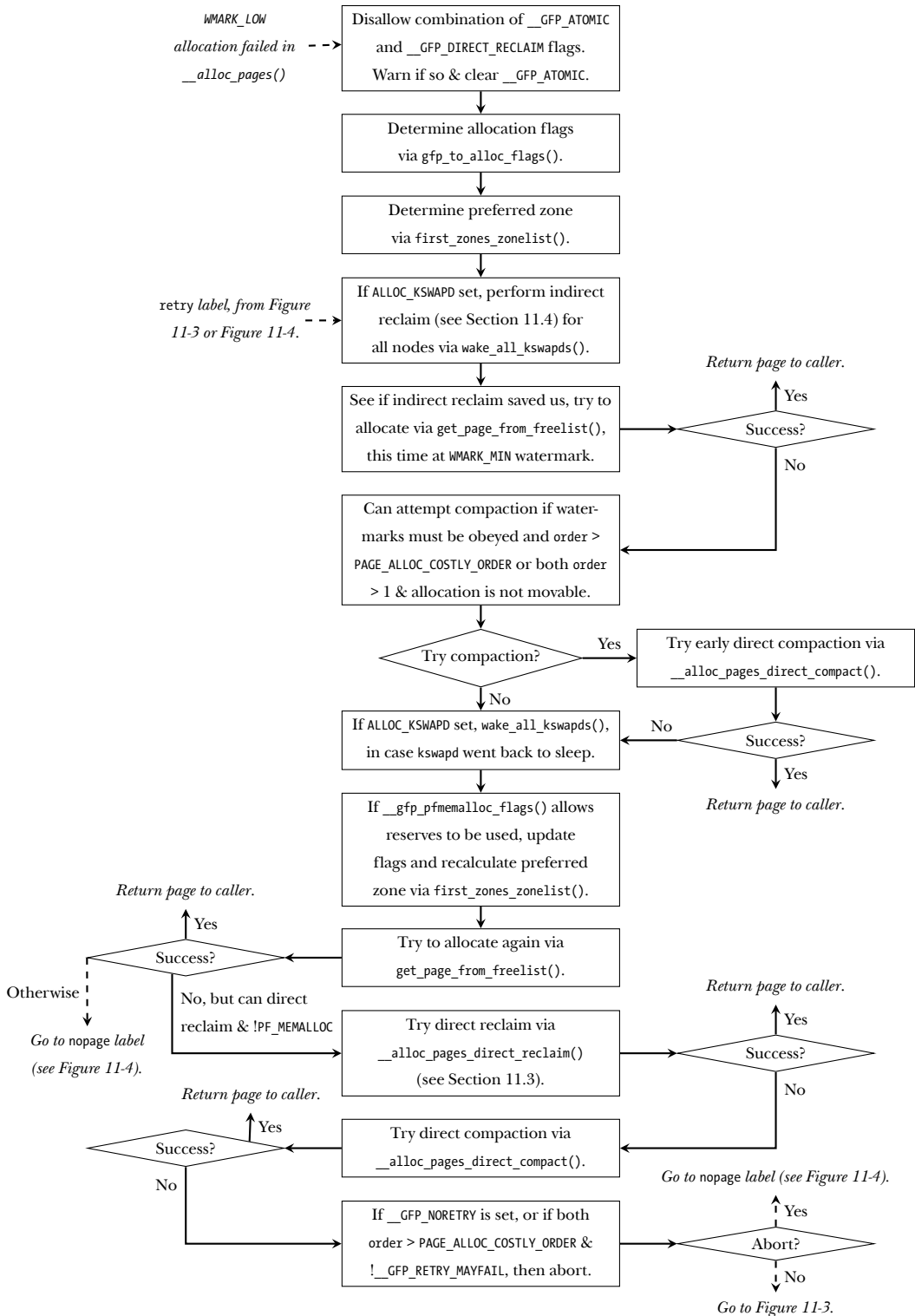


Figure 11-2: __alloc_pages_slowpath() Core Logic

We will examine the code in detail starting at Listing 11-1 below. When the initial attempt by `__alloc_pages_slowpath()` to allocate memory fails, we attempt to retry, possibly invoking the OOM killer. We examine this logic in Figure 11-3.

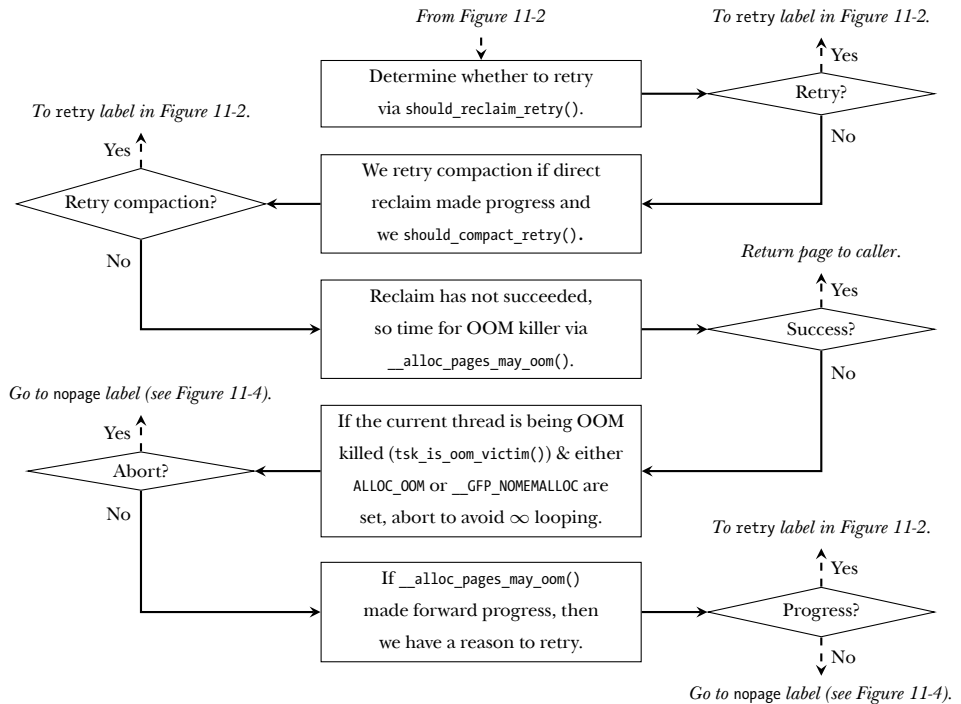


Figure 11-3: `__alloc_pages_slowpath()` Retry/OOM Logic

The logic invoked when no page can be found is shown in Figure 11-4.

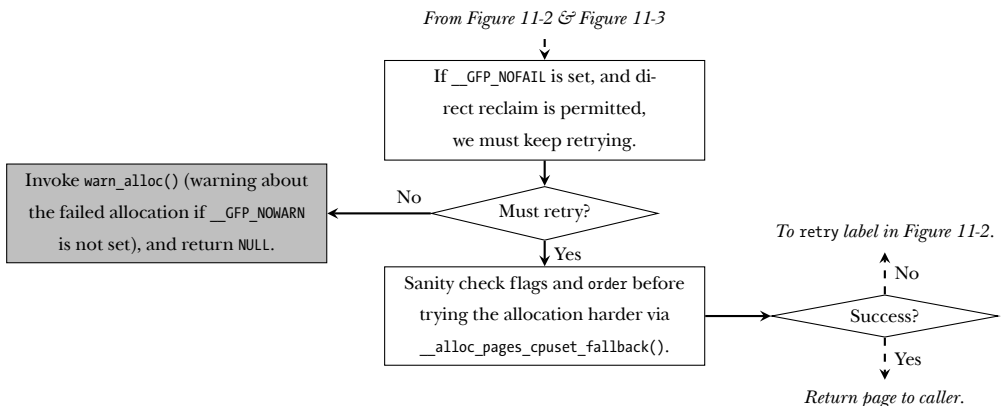


Figure 11-4: `__alloc_pages_slowpath()` No Page Logic

If we reach the slow path allocator function, `__alloc_pages_slowpath()`, then we need to perform reclaim, whether indirect or direct.

We examine `__alloc_pages_slowpath()` starting with Listing 11-1 (eliding some of the out of scope compaction logic, though keeping the general invocation for illustration, also eliding out of scope CMA and the majority of out of scope CPU set logic).

```

5014 static inline struct page *
5015 __alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
5016                       struct alloc_context *ac)
5017 {
5018     bool can_direct_reclaim = gfp_mask & __GFP_DIRECT_RECLAIM;
5019     const bool costly_order = order > PAGE_ALLOC_COSTLY_ORDER;
5020     struct page *page = NULL;
5021     unsigned int alloc_flags;
5022     unsigned long did_some_progress;
5023     enum compact_priority compact_priority;
5024     enum compact_result compact_result;
5025     int compaction_retries;
5026     int no_progress_loops;
5027     unsigned int cpuset_mems_cookie;
5028     unsigned int zonelist_iter_cookie;
5029     int reserve_flags;
5030
5031     /*
5032      * We also sanity check to catch abuse of atomic reserves being used
5033      * by
5034      * callers that are not in atomic context.
5035      */
5036     if (WARN_ON_ONCE((gfp_mask & (__GFP_ATOMIC | __GFP_DIRECT_RECLAIM)) ==
5037                     (__GFP_ATOMIC | __GFP_DIRECT_RECLAIM)))
5038         gfp_mask &= ~__GFP_ATOMIC;

```

Listing 11-1: `mm/page_alloc.c: __alloc_pages_slowpath()` Preface

We record whether we are able to perform direct reclaim in `can_direct_reclaim`, which is determined by the user specifying `__GFP_DIRECT_RECLAIM` (see Section 2.6 for a detailed description of GFP flags as a whole).

We also note whether we have exceeded a “costly” allocation size as stored in `costly_order`. This is determined via the `PAGE_ALLOC_COSTLY_ORDER` value (hard-coded to `order-3`), beyond which allocations are determined to be unlikely to coalesce naturally under ordinary memory pressure on reclaim.

We perform a sanity check to ensure that direct reclaim is not specified when an atomic allocation is requested (an atomic allocation is one performed in a context when sleep cannot be permitted such as interrupt context), as direct reclaim necessarily implies a non-atomic context (for instance, in real-time kernels, it triggers a voluntary context switch).

In this instance the issue is warned about, but the allocation may proceed only with the atomic flag cleared on the assumption that the caller may be trying to abuse memory reserved for atomic allocations.

After we perform this initial logic we proceed with performing indirect reclaim if required, a case we examine in Listing 11-2.

```

5039 restart:
5040     compaction_retries = 0;
5041     no_progress_loops = 0;
5042     compact_priority = DEF_COMPACT_PRIORITY;
5043     :
5044     /*
5045      * The fast path uses conservative alloc_flags to succeed only until
5046      * kswapd needs to be woken up, and to avoid the cost of setting up
5047      * alloc_flags precisely. So we do that now.
5048      */
5049     alloc_flags = gfp_to_alloc_flags(gfp_mask);
5050
5051     /*
5052      * We need to recalculate the starting point for the zonelist iterator
5053      * because we might have used different nodemask in the fast path, or
5054      * there was a cpuset modification and we are retrying - otherwise we
5055      * could end up iterating over non-eligible zones endlessly.
5056      */
5057     ac->preferred_zoneref = first_zones_zonelist(ac->zonelist,
5058                                                ac->highest_zoneidx, ac->nodemask);
5059     if (!ac->preferred_zoneref->zone)
5060         goto nopage;
5061     :
5062     if (alloc_flags & ALLOC_KSWAPD)
5063         wake_all_kswaps(order, gfp_mask, ac);
5064
5065     /*
5066      * The adjusted alloc_flags might result in immediate success, so try
5067      * that first
5068      */
5069     page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
5070     if (page)
5071         goto got_pg;
5072

```

Listing 11-2: mm/page_alloc.c: `__alloc_pages_slowpath()` Indirect Reclaim

We start by determining the correct allocation flags to apply to this allocation via `gfp_to_alloc_flags()`. Previously we set some conservative values, most notably `ALLOC_WMARK_LOW`, indicating that the allocation had to be at or above the low watermark. This time we instead set `ALLOC_WMARK_MIN` and derive the remainder of the flags in `gfp_to_alloc_flags()` as shown in Listing 11-3 (eliding build asserts for brevity and CMA and CPU set logic which is out of scope).

```

415 static inline unsigned int
416 gfp_to_alloc_flags(gfp_t gfp_mask)
417 {
418     unsigned int alloc_flags = ALLOC_WMARK_MIN | ALLOC_CPUSET;
419     :
420     /*
421     * The caller may dip into page reserves a bit more if the caller
422     * cannot run direct reclaim, or if the caller has realtime scheduling
423     * policy or is asking for __GFP_HIGH memory. GFP_ATOMIC requests
424     will
425     * set both ALLOC_HARDER (__GFP_ATOMIC) and ALLOC_HIGH (__GFP_HIGH).
426     */
427     alloc_flags |= (__force int)
428         (gfp_mask & (__GFP_HIGH | __GFP_KSWAPD_RECLAIM));
429
430     if (gfp_mask & __GFP_ATOMIC) {
431         /*
432         * Not worth trying to allocate harder for __GFP_NOMEMALLOC
433         even
434         * if it can't schedule.
435         */
436         if (!(gfp_mask & __GFP_NOMEMALLOC))
437             alloc_flags |= ALLOC_HARDER;
438         :
439     } else if (unlikely(rt_task(current)) && in_task())
440         alloc_flags |= ALLOC_HARDER;
441     :
442     return alloc_flags;
443 }
444

```

Listing 11-3: mm/page_alloc.c: *gfp_to_alloc_flags()*

This sets the `ALLOC_WMARK_MIN` flag, indicating that the zone from which the allocation occurs should be at or above the minimum watermark, i.e. `WMARK_MIN`. It additionally sets `ALLOC_CPUSET` but this is out of scope for the book.

We determine whether to allow access to further reserves depending on allocation flags, firstly if `__GFP_HIGH` is set then we set `ALLOC_HIGH` (the two values are maintained to be the same), which is used explicitly for 32-bit architectures.

Secondly, and more importantly, if `__GFP_KSWAPD_RECLAIM` is set, then so is `ALLOC_KSWAPD` indicating that indirect reclaim can proceed.

If the allocation is atomic, as indicated by the `__GFP_ATOMIC` flag being specified, and emergency reserves are not explicitly forbidden for use (i.e. the `__GFP_NOMEMALLOC` flag not being set), then we default to accessing these reserves by setting the `ALLOC_HARDER` flag.

Finally, if the current thread (represented by the current macro) is a real-time task as checked by `rt_task()` and is in task context as checked by

`in_task()` (i.e. not servicing a non-maskable interrupt or NMI, in a hard interrupt or a soft interrupt handler), then this necessitates the use of emergency reserves and `ALLOC_HARDER` is equally set in this instance.

Returning to the `__alloc_pages_slowpath()` code that invokes this in Listing 11-4, we redetermine the preferred zone in case this changed via the `first_zones_zonelist()` helper function, jumping to the no page handling logic if none can be found (explored in Listing 11-9. For more on zone lists see Section 2.4).

Importantly, next we check the likely now-established `ALLOC_KSWAPD` flag, and if set begin indirect reclaim for all zones on this node via `wake_all_kswapds()`, which we explore in Listing 11-24 and Section 11.4.

With these flags adjusted to permit zone memory to be utilised up to the minimum watermark, there's a good chance that we can now succeed in our attempt to allocate a page which we attempt to do via `get_page_from_freelist()` (see Listing 2-55 in Section 2.8.2 and Chapter 2). If so we return the successfully retrieved page.

If not, then memory pressure is significant—we cannot satisfy the request even at minimum watermark level, so must now perform reclaim directly in order to service it. We examine the direct reclaim logic in Listing 11-4.

```

5039      /*
5040       * For costly allocations, try direct compaction first, as it's likely
5041       * that we have enough base pages and don't need to reclaim. For non-
5042       * movable high-order allocations, do that as well, as compaction will
5043       * try prevent permanent fragmentation by migrating from blocks of the
5044       * same migratetype.
5045       * Don't try this for allocations that are allowed to ignore
5046       * watermarks, as the ALLOC_NO_WATERMARKS attempt didn't yet happen.
5047       */
5048      if (can_direct_reclaim &&
5049          (costly_order ||
5050           (order > 0 && ac->migratetype != MIGRATE_MOVABLE))
5051          && !gfp_pfmemalloc_allowed(gfp_mask)) {
5052          page = __alloc_pages_direct_compact(gfp_mask, order,
5053                                              alloc_flags, ac,
5054                                              INIT_COMPACT_PRIORITY,
5055                                              &compact_result);
5056          if (page)
5057              goto got_pg;
5058          :
5059      }
5060
5061  retry:
5062      /* Ensure kswapd doesn't accidentally go to sleep as long as we loop
5063       */
5064
5065      if (alloc_flags & ALLOC_KSWAPD)
5066          wake_all_kswapds(order, gfp_mask, ac);
5067

```

```

5148     reserve_flags = __gfp_pfmemalloc_flags(gfp_mask);
5149     if (reserve_flags)
5150         alloc_flags = gfp_to_alloc_flags_cma(gfp_mask, reserve_flags);
5151     ;
5152     /*
5153      * Reset the nodemask and zonelist iterators if memory policies can be
5154      * ignored. These allocations are high priority and system rather than
5155      * user oriented.
5156      */
5157     if (!(alloc_flags & ALLOC_CPUSET) || reserve_flags) {
5158         ac->nodemask = NULL;
5159         ac->preferred_zoneref = first_zones_zonelist(ac->zonelist,
5160                                                     ac->highest_zoneidx, ac->nodemask);
5161     }
5162
5163     /* Attempt with potentially adjusted zonelist and alloc_flags */
5164     page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
5165     if (page)
5166         goto got_pg;
5167
5168     /* Caller is not willing to reclaim, we can't balance anything */
5169     if (!can_direct_reclaim)
5170         goto nopage;
5171
5172     /* Avoid recursion of direct reclaim */
5173     if (current->flags & PF_MEMALLOC)
5174         goto nopage;
5175
5176     /* Try direct reclaim and then allocating */
5177     page = __alloc_pages_direct_reclaim(gfp_mask, order, alloc_flags, ac,
5178                                         &did_some_progress);
5179     if (page)
5180         goto got_pg;
5181
5182     /* Try direct compaction and then allocating */
5183     page = __alloc_pages_direct_compact(gfp_mask, order, alloc_flags, ac,
5184                                         compact_priority, &compact_result);
5185     if (page)
5186         goto got_pg;

```

Listing 11-4: mm/page_alloc.c: __alloc_pages_slowpath() Direct Reclaim

While compaction is out of scope for the book, we identify the points at which it would occur during slow-path allocation. Here, we firstly attempt direct compaction, i.e. a blocking form of compaction if and only if direct reclaim is generally permitted and either we have reached a costly order, or the allocation is higher order and the memory being allocated is not mov-

able (this may be the last opportunity to move memory that could be otherwise coalesced before it becomes unmovable).

In these circumstances we invoke `__alloc_pages_direct_compact()`, the details of which are out of scope for the book. If the allocation succeeds under direct compaction, then the page is returned.

NOTE

Compaction is the process by which the kernel migrates movable folios in order to allow the coalescence of buddies into higher order folios. Through this process the kernel is able to avert higher order fragmentation resulting in an inability to allocate memory.

Next, if `ALLOC_KSWAPD` is specified, we reawake `kswapd` for all zones of this node via `wake_all_kswapds()` to ensure that indirect reclaim cannot go to sleep while we are still attempting to allocate.

We then perform some housekeeping before invoking direct reclaim itself. Firstly we check to see if we can use memory reserves via `__gfp_pfmemalloc_flags()` which we examine in Listing 11-5.

Note that we set `alloc_flags` to the result of this function via `gfp_to_alloc_flags_cma()`. CMA is out of scope for the book, but regardless of this, whether enabled or not, it returns the `reserve_flags` parameter.

```

4872 /*
4873  * Distinguish requests which really need access to full memory
4874  * reserves from oom victims which can live with a portion of it
4875  */
4876 static inline int __gfp_pfmemalloc_flags(gfp_t gfp_mask)
4877 {
4878     if (unlikely(gfp_mask & __GFP_NOMEMALLOC))
4879         return 0;
4880     if (gfp_mask & __GFP_MEMALLOC)
4881         return ALLOC_NO_WATERMARKS;
4882     if (in_serving_softirq() && (current->flags & PF_MEMALLOC))
4883         return ALLOC_NO_WATERMARKS;
4884     if (!in_interrupt()) {
4885         if (current->flags & PF_MEMALLOC)
4886             return ALLOC_NO_WATERMARKS;
4887         else if (oom_reserves_allowed(current))
4888             return ALLOC_OOM;
4889     }
4890
4891     return 0;
4892 }
```

Listing 11-5: `mm/page_alloc.c: __gfp_pfmemalloc_flags()`

This determines the circumstances under which emergency memory reserves can be used—broadly `__GFP_MEMALLOC` (which can also be specified across a thread by setting the `PF_MEMALLOC` flag in `struct task_struct->flags`)

implies that watermarks can be ignored altogether and all memory in a zone can be used (as indicated by `ALLOC_NO_WATERMARKS`).

NOTE

Thread-specific allocation flags are often enabled and disabled via `xxx_save()` and `xxx_restore()` helper functions, for instance `PF_MEMALLOC` can be set/cleared via `memalloc_noreclaim_save()` and `memalloc_noreclaim_restore()` respectively.

In non-interrupt context, `oom_reserves_allowed()`, is used to determine if this is an out of memory (OOM) killer victim, in which case additional reserves are permitted as we know this process will soon be freed and will therefore soon free all of its memory at any rate (we denote this with the `ALLOC_OOM` flag).

If `__GFP_NOMEMALLOC` is set, this disallows the use of these reserves altogether.

Returning to Listing 11-4, we then, if reserves can be used, we then eliminate node mask restrictions and redetermine the preferred zone at which to attempt allocation first via `first_zones_zonelist()`.

After this we regardless attempt to allocate a new page to see if changing these flags or indirect reclaim has led to a situation where the allocation would now succeed via `get_page_from_freelist()` (see Listing 2-55 in Section 2.8.2 and Chapter 2). If so, we return the folio.

At this point, we are ready to go ahead and try direct reclaim. If `can_direct_reclaim` is not set, i.e. if the user did not specify `__GFP_DIRECT_RECLAIM`, we enter no page handling (see Listing 11-9). Equally, if `PF_MEMALLOC` is set, indicating that reclaim should not proceed (but emergency reserves should be used), we also enter no page handling.

This is important, because the direct reclaim logic itself sets this flag and we want to avoid reclaim recursing over and over again.

Finally we perform the actual direct reclaim via `__alloc_pages_direct_reclaim()`, which we examine in Listing 11-17 and Section 11.3. If this succeeds we return the folio

If it does not, then finally we try direct compaction via `__alloc_pages_direct_compact()`, again returning the resultant folio if it succeeds.

Once direct reclaim attempts have been exhausted we are at the point where we must determine if a retry makes sense or not, which we examine in Listing 11-6.

```

5188      /* Do not loop if specifically requested */
5189      if (gfp_mask & __GFP_NORETRY)
5190          goto nopage;
5191
5192      /*
5193       * Do not retry costly high order allocations unless they are
5194       * __GFP_RETRY_MAYFAIL
5195       */
5196      if (costly_order && !(gfp_mask & __GFP_RETRY_MAYFAIL))
5197          goto nopage;
```

```

5198
5199         if (should_reclaim_retry(gfp_mask, order, ac, alloc_flags,
5200                                did_some_progress > 0, &no_progress_loops))
5201             goto retry;
5202
5203         /*
5204         * It doesn't make any sense to retry for the compaction if the order
5205         -0
5206         * reclaim is not able to make any progress because the current
5207         * implementation of the compaction depends on the sufficient amount
5208         * of free memory (see __compaction_suitable)
5209         */
5210         if (did_some_progress > 0 &&
5211             should_compact_retry(ac, order, alloc_flags,
5212                                compact_result, &compact_priority,
5213                                &compaction_retries))
5214             goto retry;

```

Listing 11-6: mm/page_alloc.c: __alloc_pages_slowpath() Retry Logic

The purpose of this logic is to assess whether it makes sense to retry, and if so, to do so. Firstly, if the `__GFP_NORETRY` flag is specified, this explicitly hints that no retries should be attempted, so we immediately defer to the no page handling (see Listing 11-9).

Costly allocations are, by their very definition, unlikely to succeed if direct reclaim did not immediately provide a free folio so, unless the user explicitly specifies that they want this behaviour via the `__GFP_RETRY_MAYFAIL` flag, we also do not proceed with retry in this case.

Finally we defer the decision to a helper function, `should_reclaim_retry()` which we examine in Listing 11-7. Note that as one of the parameters, we pass in `did_some_progress`, a field set by `__alloc_pages_direct_reclaim()` (see Listing 11-17 in Section 11.3).

This determines whether forward progress has occurred during direct reclaim, which would make it useful to retry the reclaim operation. This therefore forms part of the criteria of `should_reclaim_retry()`.

Before we examine `should_reclaim_retry()` we see that, if it succeeds we jump to the retry label (Listing 11-4), and if it fails and direct reclaim had made forward progress, we apply the same logic to retrying compaction via `should_compact_retry()`.

Compaction is out of scope for the book so we won't examine that in detail, but rather simply observe that this plays a role in the logic here.

We have therefore only `should_reclaim_retry()` to examine in detail here, as shown in Listing 11-7 (eliding tracing hooks and some work queue congestion logic).

```

4899 /*
4900  * Checks whether it makes sense to retry the reclaim to make a forward
4901  * progress
4902  * for the given allocation request.

```

```

4902  *
4903  * We give up when we either have tried MAX_RECLAIM_RETRIES in a row
4904  * without success, or when we couldn't even meet the watermark if we
4905  * reclaimed all remaining pages on the LRU lists.
4906  *
4907  * Returns true if a retry is viable or false to enter the oom path.
4908  */
4909  static inline bool
4910  should_reclaim_retry(gfp_t gfp_mask, unsigned order,
4911                      struct alloc_context *ac, int alloc_flags,
4912                      bool did_some_progress, int *no_progress_loops)
4913  {
4914      struct zone *zone;
4915      struct zoneref *z;
4916      bool ret = false;
4917
4918      /*
4919       * Costly allocations might have made a progress but this doesn't mean
4920       * their order will become available due to high fragmentation so
4921       * always increment the no progress counter for them
4922       */
4923      if (did_some_progress && order <= PAGE_ALLOC_COSTLY_ORDER)
4924          *no_progress_loops = 0;
4925      else
4926          (*no_progress_loops)++;
4927
4928      /*
4929       * Make sure we converge to OOM if we cannot make any progress
4930       * several times in the row.
4931       */
4932      if (*no_progress_loops > MAX_RECLAIM_RETRIES) {
4933          /* Before OOM, exhaust highatomic_reserve */
4934          return unreserve_highatomic_pageblock(ac, true);
4935      }
4936
4937      /*
4938       * Keep reclaiming pages while there is a chance this will lead
4939       * somewhere. If none of the target zones can satisfy our allocation
4940       * request even if all reclaimable pages are considered then we are
4941       * screwed and have to go OOM.
4942       */
4943      for_each_zone_zonelist_nodemask(zone, z, ac->zonelist,
4944                                      ac->highest_zoneidx, ac->nodemask) {
4945          unsigned long available;
4946          unsigned long reclaimable;
4947          unsigned long min_wmark = min_wmark_pages(zone);
4948          bool wmark;

```

```

4949
4950         available = reclaimable = zone_reclaimable_pages(zone);
4951         available += zone_page_state_snapshot(zone, NR_FREE_PAGES);
4952
4953         /*
4954          * Would the allocation succeed if we reclaimed all
4955          * reclaimable pages?
4956          */
4957         wmark = __zone_watermark_ok(zone, order, min_wmark,
4958                                     ac->highest_zoneidx, alloc_flags, available);
4959         :
4960
4961         if (wmark) {
4962             ret = true;
4963             break;
4964         }
4965     }
4966     :
4967     return ret;
4968 }

```

Listing 11-7: mm/page_alloc.c: should_reclaim_retry()

This function is passed a pointer to the `__alloc_pages_slowpath()`'s `no_progress_loops` counter, which keeps tracks how many loops through the retry logic have occurred without progress, allowing us to eventually abort retries when it makes sense to.

We start by resetting this counter if both some progress was made and this was not a costly order allocation. Some sort of progress being made does not guarantee costly allocations will succeed, so we do not consider that progress has been made in this case. Otherwise, we increment the `no_progress_loops` counter.

We then check to see if this count exceeds `MAX_RECLAIM_RETRIES` (hard-coded to 16 loops of no progress being permitting). If so, we see whether we can utilise reserved high atomic memory via `unreserve_highatomic_pageblock()`.

This reserve is intended for atomic allocations at higher orders (atomic meaning that the thread cannot sleep for the allocation and thus more involved forms of reclaim cannot be performed), however in this instance we permit the use of this reserve to avoid the out of memory killer.

Next we manually iterate through each zone in which the allocation could be performed, and calculate how many folios could be reclaimed and use this to determine if it is at all feasible for a retry to succeed at the minimum water level if indeed all of these folios were reclaimed (this is determined via the `zone_reclaimable_pages()` function).

If absolutely no reclaim is possible, then we abort retries, otherwise we attempt retries up to `MAX_RECLAIM_RETRIES` loops without progress.

This allows for a sensible mechanism which tries its absolute best to service requests even under very extreme memory pressure resorting to the out of memory killer if absolutely necessary.

Finally, if retry is not feasible, we are in a place where we consider an out of memory condition, which we examine in Listing 11-8.

```

5224      /* Reclaim has failed us, start killing things */
5225      page = __alloc_pages_may_oom(gfp_mask, order, ac, &did_some_progress);
5226      if (page)
5227          goto got_pg;
5228
5229      /* Avoid allocations with no watermarks from looping endlessly */
5230      if (tsk_is_oom_victim(current) &&
5231          (alloc_flags & ALLOC_OOM ||
5232           (gfp_mask & __GFP_NOMEMALLOC)))
5233          goto nopage;
5234
5235      /* Retry as long as the OOM killer is making progress */
5236      if (did_some_progress) {
5237          no_progress_loops = 0;
5238          goto retry;
5239      }

```

Listing 11-8: mm/page_alloc.c: `__alloc_pages_slowpath()` OOM

We perform the actual allocation in this instance via `__alloc_pages_may_oom()`, which we explore in Chapter 13, in Section 13.4.1 and Listing 13-2.

If this results in a folio immediately becoming available to fulfil the allocation then we return this, otherwise we examine the edge case of the running thread being the OOM victim (checked via `tsk_is_oom_victim()`) and either using all available reserves (as specified by the `ALLOC_OOM` flag) or by no emergency reserves being available at all because `__GFP_NOMEMALLOC` is set, then we risk looping despite the fact we are about to be killed, so in this instance we simply defer to the no page handling.

Otherwise, if the OOM killer made some progress, we reset the `no_progress_loops` counter and retry.

If we did not, or one of the many other parts of the logic that revert to jumping to the no page handling logic are invoked, we end up at the point of dealing with an allocation failure, a case we examine in Listing 11-9.

```

5039  nopage:
5040      /*
5041       * Deal with possible cpuset update races or zonelist updates to avoid
5042       * a unnecessary OOM kill.
5043       */
5044      if (check_retry_cpuset(cpuset_mems_cookie, ac) ||
5045          check_retry_zonelist(zonelist_iter_cookie))
5046          goto restart;
5047
5048      /*
5049       * Make sure that __GFP_NOFAIL request doesn't leak out and make sure

```

```

5050         * we always retry
5051         */
5052         if (gfp_mask & __GFP_NOFAIL) {
5053             /*
5054              * All existing users of the __GFP_NOFAIL are blockable, so
5055              warn
5056              * of any new users that actually require GFP_NOWAIT
5057              */
5058             if (WARN_ON_ONCE_GFP(!can_direct_reclaim, gfp_mask))
5059                 goto fail;
5060             :
5061             /*
5062              * Help non-failing allocations by giving them access to
5063              memory
5064              * reserves but do not use ALLOC_NO_WATERMARKS because this
5065              * could deplete whole memory reserves which would just make
5066              * the situation worse
5067              */
5068             page = __alloc_pages_cpuset_fallback(gfp_mask, order,
5069             ALLOC_HARDER, ac);
5070             if (page)
5071                 goto got_pg;
5072             :
5073             goto retry;
5074         }
5075 fail:
5076     warn_alloc(gfp_mask, ac->nodemask,
5077               "page allocation failure: order:%u", order);
5078 got_pg:
5079     return page;
5080 }

```

Listing 11-9: mm/page_alloc.c: __alloc_pages_slowpath() Allocation Failure

All of the checks are performed when the `__GFP_NOFAIL` flag is set—in this scenario we are simply not permitted to fail so must keep on retrying regardless. Otherwise, we simply go ahead and fail the request, writing a warning to the kernel circular buffer via `warn_alloc()` (if `__GFP_NOWARN` flag was not set).

Otherwise, we check for the edge case of `__GFP_NOFAIL` being set but `__GFP_DIRECT_RECLAIM` not having been set, in which case we assume the `__GFP_NOFAIL` flag was set in error and abort the allocation.

After this we invoke `__alloc_pages_cpuset_fallback()`, which attempts a last-gasp allocation, setting `ALLOC_HARDER` to access to even more emergency memory reserves.

If this succeeds we return the resultant folio, otherwise we simply unconditionally retry.

11.2 LRU Vectors

In order to most efficiently make use of memory installed in the system the kernel tries to use as much of it as it can to cache data from disk in memory (see the page cache chapter for more details on the precise details of this).

While the kernel is aggressive in caching files in memory, it also needs to be aggressive in freeing them again (termed *reclaim*) when memory pressure arises, i.e. zones reaching their low or minimum watermark level (see the physical memory chapter for more on this). To do this we need to *prioritise* folios in such a way as to ensure we maintain useful folios and eliminate less useful ones.

The kernel *aspires* to implement the Least Recently Used (LRU) cache replacement policy – conceptually this works as its name suggests – always evicting the least recently used folio. In idealised form:-

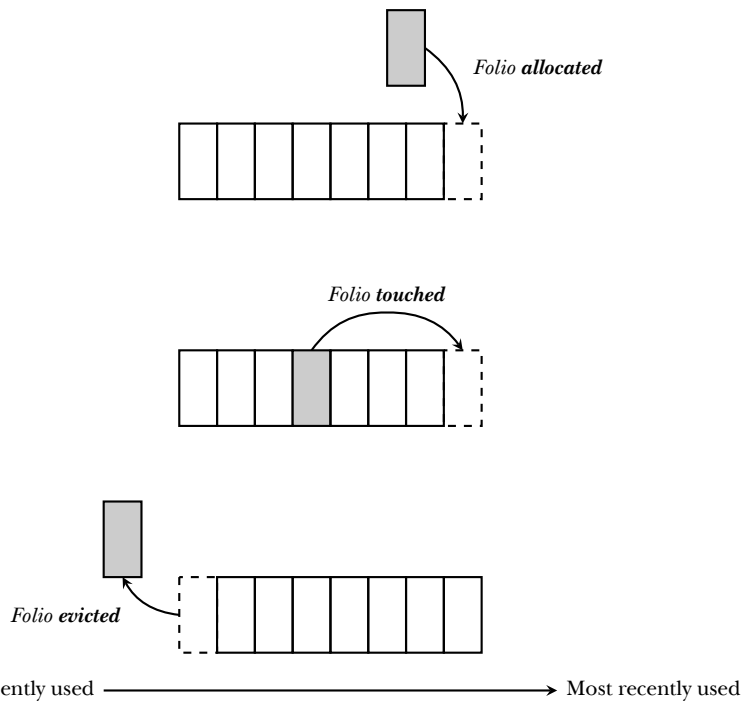


Figure 11-5: Idealised LRU

I say *aspire* to this, as unfortunately this approach is not practical—it would introduce significant latency to capture every single folio access in order to move them around in the LRU list, so the kernel implements an approximation to this approach.

Rather than moving folios around in an LRU on each access the kernel in essence *samples* whether the last folio was accessed on each occasion we contemplate reclaim – this is possible because hardware sets the

`_PAGE_ACCESSED` page table flag when each base page is accessed. This is also referred to as the ‘young’ flag, i.e. the one tested by `pte_young()`.

This is x86-64 specific and precise means of determining page access varies by architecture. In x86-64 this is ultimately performed by `ptep_test_and_clear_young()` which is called by `ptep_clear_flush_young()` and `ptep_clear_flush_young_notify()` in turn.

This flag is ‘sticky’, meaning that it is set by hardware and stays set until the kernel clears it. Therefore when reclaim is performed this flag can be checked to see whether the folio has been active since the last check and cleared for next time. This is done via `folio_referenced()` which performs an rmap walk from the folio to each PTE which maps it.

If we maintain a single LRU list we run into a big problem – we are not moving folios in the LRU each time they are accessed, so whichever happens to be at the rear of the list (and subject to possible reclaim) might be either regularly accessed folios or rarely accessed ones, we don’t know until we check. This can result in a lot of time spent checking folios that won’t be reclaimable and slow things down significantly.

The solution is simple – maintain separate *active* and *inactive* LRU lists. By only trying to reclaim from the inactive list and ‘promoting’ folios that are found to be recently used there to the active one we significantly increase our chances of actually reclaiming memory.

In addition, we can *choose when* to scan the active list in order to determine which folios should be *demoted* to the inactive list at our leisure – under severe memory pressure we can focus on the inactive list and almost certainly free memory, otherwise we can also check the active list to ensure the two are *balanced* appropriately.

‘Eviction’ has an entirely different meaning for file-backed and anonymous folios – the former, when clean (i.e. with no changes pending to be written back to disk), can simply be dropped from the page cache (and re-read if needed again). Anonymous pages will need to be swapped out, a far more expensive operation.

It is therefore sensible to maintain *separate LRUs* for file-backed folios and anonymous ones in order that we can have precise control over how we handle evictions of each and can batch up the appropriate operation in either case.

Finally, pages which cannot be evicted at all (typically memory that has been `mlock()`ed) are not maintained on any of these lists. This leaves us with ostensibly 5 LRU lists overall – however note that unevictable pages do not actually get placed on *any* list in practice.

We explore the actual underlying mechanism in detail in Section 11.5.

LRU lists are kept in a per-memcg (i.e. memory cgroup, though cgroups are out of scope for the book), per-node `struct lruvec` data structure. We visualise this in Figure 11-6.

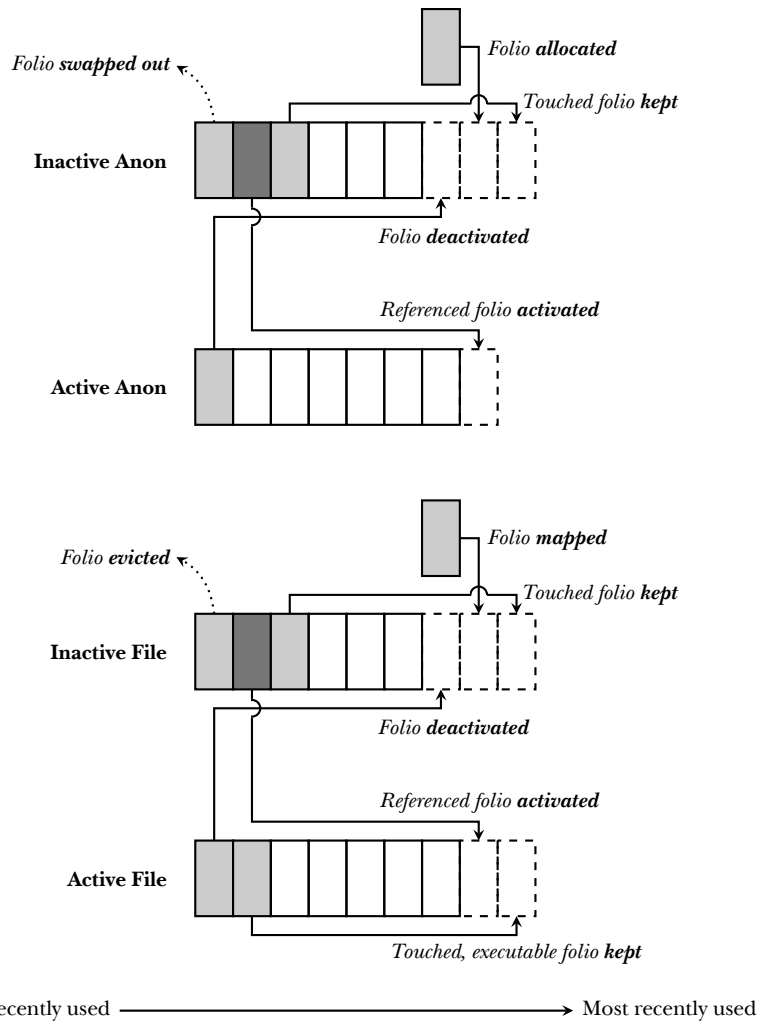


Figure 11-6: LRU lists

This requires some additional explanation – each time a folio reaches near the end of the inactive list (and this is part of the batch of folios under consideration for reclaim), we check whether the page table accessed bit is set for any underlying PTE – it being set is necessary, but insufficient for the folio to be ‘activated’.

Let’s examine a simplified version of the function which performs this check, `folio_check_references()` in Listing 11-10 (note we examine the full thing in Listing 11-70 and Section 11.5.10).

```

1433 static enum page_references folio_check_references(struct folio *folio,
1434                                                  struct scan_control *sc)
1435 {
1436     int referenced_ptes, referenced_folio;

```

```

1437     unsigned long vm_flags;
1438
1439     referenced_ptes = folio_referenced(folio, 1, sc->target_mem_cgroup,
1440                                       &vm_flags);
1441     referenced_folio = folio_test_clear_referenced(folio);
1442     :
1443
1444     if (referenced_ptes) {
1445         /*
1446          * All mapped folios start out with page table
1447          * references from the instantiating fault, so we need
1448          * to look twice if a mapped file/anon folio is used more
1449          * than once.
1450          *
1451          * Mark it and spare it for another trip around the
1452          * inactive list. Another page table reference will
1453          * lead to its activation.
1454          *
1455          * Note: the mark is set for activated folios as well
1456          * so that recently deactivated but used folios are
1457          * quickly recovered.
1458          */
1459         folio_set_referenced(folio);
1460
1461         if (referenced_folio || referenced_ptes > 1)
1462             return PAGEREf_ACTIVATE;
1463         :
1464         return PAGEREf_KEEP;
1465     }
1466     :
1467     return PAGEREf_RECLAIM;
1468 }

```

Listing 11-10: mm/vmscan.c: Simplified folio_check_references()

Note – We elide a number of edge cases here. We will examine the function in full detail in Section 11.5.

This lays out the details of an additional ‘referenced’ check in the inactive LRU list only. When reclaim is performed in this list via `shrink_page_list()`, this determines how to proceed - `PAGEREf_ACTIVATE` indicates that the folio should be placed on the active list, `PAGEREf_KEEP` indicates that it should be placed at the front of the inactive list for another go-around, and `PAGEREf_RECLAIM` indicates that reclaim should occur.

The logic is as follows (again, importantly, we ignore some edge case details here – see Section 11.5 for a *full* analysis):-

1. Determine how many PTEs (or higher page table levels if huge) which map this folio are marked ‘young’ (i.e. whether `_PAGE_ACCESSED` is set for x86-64) via `folio_referenced()`, simultaneously clearing this flag ready for the next check.

2. Check *and clear* the folio's PG_referenced flag.
3. If no PTEs mapping this folio were accessed since the last check, we go ahead and *reclaim*.
4. Otherwise, we set the PG_referenced flag to indicate that the folio is young.
5. If it had the referenced flag set prior to this, or if more than 1 PTE reference the folio, then we *activate* the folio. Note that the folio will retain its PG_referenced flag, and thus when deactivated will get an extra cycle around the inactive list.
6. Otherwise, we *keep* the folio, i.e. place it at the front of the inactive list for another go-around.

For the active LRU the logic is different – for anonymous pages we simply allow folios to become deactivated after they have gone through the list. For file folios we add an edge case for those which back *executable* ranges (i.e. ones which contain code) and, if touched, we maintain them in the active list. See `shrink_active_list()` in Listing 11-56 for more details.

When an event occurs that causes a folio to be accessed such a file being read from or written to or a folio being swapped in then it is manually marked accessed via `folio_mark_accessed()` (more on this function later). Examining how these states vary:-

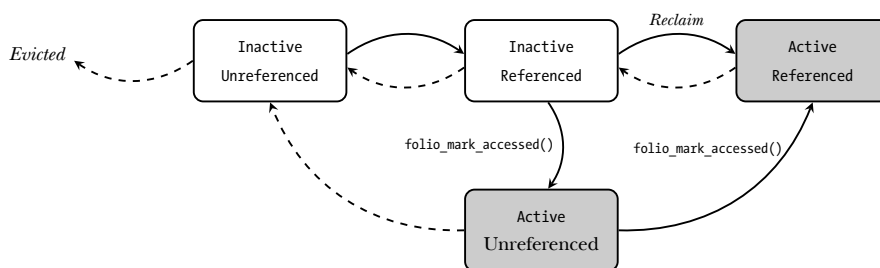


Figure 11-7: Folio active/referenced states

Note that solid lines denote folio promotion, dotted lines denote folio demotion and that `folio_mark_accessed()` will promote from inactive, referenced to active, unreferenced while reclaim will promote to active, referenced, i.e. retaining the PG_referenced folio flag. Additionally, the promotion from active, unreferenced to active, referenced will only be performed by `folio_mark_accessed()`.

Examining `folio_mark_accessed()`:-

```

431 /*
432  * Mark a page as having seen activity.
433  *
434  * inactive,unreferenced    ->    inactive,referenced
435  * inactive,referenced     ->    active,unreferenced
436  * active,unreferenced     ->    active,referenced

```

```

437  *
438  * When a newly allocated page is not yet visible, so safe for non-atomic ops,
439  * __SetPageReferenced(page) may be substituted for mark_page_accessed(page).
440  */
441 void folio_mark_accessed(struct folio *folio)
442 {
443     if (!folio_test_referenced(folio)) {
444         folio_set_referenced(folio);
445     } else if (folio_test_unevictable(folio)) {
446         /*
447          * Unevictable pages are on the "LRU_UNEVICTABLE" list. But,
448          * this list is never rotated or maintained, so marking an
449          * unevictable page accessed has no effect.
450          */
451     } else if (!folio_test_active(folio)) {
452         /*
453          * If the folio is on the LRU, queue it for activation via
454          * cpu_fbatches.activate. Otherwise, assume the folio is in a
455          * folio_batch, mark it active and it'll be moved to the
456          active
457          * LRU on the next drain.
458          */
459         if (folio_test_lru(folio))
460             folio_activate(folio);
461         else
462             __lru_cache_activate_folio(folio);
463         folio_clear_referenced(folio);
464         workingset_activation(folio);
465     }
466     if (folio_test_idle(folio))
467         folio_clear_idle(folio);
468 }

```

Listing 11-11: mm/swap.c: *folio_mark_accessed()*

This follows the logic described above, activating the folio via `folio_activate()`, or if it is on a batch, via `__lru_cache_activate_folio()` which goes through each entry on the appropriate lruvec and tries to find the folio there to activate (both discussed later in section 11.7.6).

Finally, working set activation is noted (though out of scope for the book), and if the folio was marked idle, we clear that flag.

11.2.1 struct lruvec

The data structure which encapsulates all of this is `struct lruvec`. A folio's lruvec can be obtained via `folio_lruvec()`, which obtains a folio's memcg via `folio_memcg()` (ultimately determined from either the `memcg_data` field of the `struct folio` if it sits in a specific memcg, or the `root_mem_cgroup` global if none is set). This ultimately invokes `mem_cgroup_lruvec()` which looks up the node-

specific lruvec (the node having been obtained via `folio_pgdat()` which ultimately invokes `page_to_nid()` extracting the node ID from the folio's flags field).

Examining the data structure:-

```

317 struct lruvec {
318     struct list_head          lists[NR_LRU_LISTS];
319     /* per lruvec lru_lock for memcg */
320     spinlock_t                lru_lock;
321     /*
322      * These track the cost of reclaiming one LRU - file or anon -
323      * over the other. As the observed cost of reclaiming one LRU
324      * increases, the reclaim scan balance tips toward the other.
325      */
326     unsigned long              anon_cost;
327     unsigned long              file_cost;
328     /* Non-resident age, driven by LRU movement */
329     atomic_long_t              nonresident_age;
330     /* Refaults at the time of last reclaim cycle */
331     unsigned long              refaults[ANON_AND_FILE];
332     /* Various lruvec state flags (enum lruvec_flags) */
333     unsigned long              flags;
334 #ifdef CONFIG_MEMCG
335     struct pglist_data *pgdat;
336 #endif
337 };

```

Listing 11-12: include/linux/mmzone.h: struct lruvec

Examining each field individually:-

- `lists[]` – This contains the head of each of the aforementioned LRU lists, with each index in the array defined in `enum lru_list`:
 - `LRU_INACTIVE_ANON` – The inactive anonymous folio LRU list.
 - `LRU_ACTIVE_ANON` – The active anonymous folio LRU list.
 - `LRU_INACTIVE_FILE` – The inactive file folio LRU list.
 - `LRU_ACTIVE_FILE` – The active file folio LRU list.
 - `LRU_UNEVICTABLE` – A placeholder for unevictable folios, however there is no actual list associated with these. The list head is poisoned to avoid inappropriate access in `lruvec_init()`.
- `lru_lock` – Lock which protects fields in the struct `lru_vec` object, typically locked via `folio_lruvec_lock_irqsave()` and unlocked via `unlock_page_lruvec_irqrestore()`.
- `anon_cost` – Used to track the cost of reclaiming the anon LRU over the file LRU. This allows the reclaim scan to maintain balance between the two. Set via `lru_note_cost()` and `lru_note_cost_folio()`. See Section 11.5 for details.

- `file_cost` – Used to track the cost of reclaiming the file LRU over the anon LRU. Again, see Section 11.5 for more details.
- `nonresident_age` – A counter of inactive evictions and activations used by the working set logic, which is out of scope of the book. Incremented by `workingset_age_nonresident()`.
- `refaults[]` – Keeps track of the number of *refaults*, i.e. times folios have been evicted then faulted back in. Used as part of the working set logic.
- `flags` – Contains flags relevant to the `lruvec` as defined in `enum lruvec_flags`. Currently only one flag is specified – `LRUVEC_CONGESTED`. This is used by `reclaim` to note a situation where there are a large number of dirty file folios which are deemed to have become congested (dirty meaning file-back folios which have been modified but not yet written back to disk).
- `pgdat` – Reference to this node's `pg_data_t` metadata object. This is typically retrieved via `lruvec_pgdat()`.

11.2.2 *lruvec Operations*

We now have a picture of what `lruvec` lists are and how folios move around in them, but how are they added to, removed from and moved between LRU lists?

Folios are ultimately added to an `lruvec` by either `lruvec_add_folio()` or `lruvec_add_folio_tail()` and removed by `lruvec_del_folio()` (with the very special exception of `reclaim`'s `isolate_lru_pages()`, discussed later):-

```

97 static __always_inline
98 void lruvec_add_folio(struct lruvec *lruvec, struct folio *folio)
99 {
100     enum lru_list lru = folio_lru_list(folio);
101
102     update_lru_size(lruvec, lru, folio_zonenum(folio),
103                    folio_nr_pages(folio));
104     if (lru != LRU_UNEVICTABLE)
105         list_add(&folio->lru, &lruvec->lists[lru]);
106 }
```

Listing 11-13: include/linux/mm_inline.h: `lruvec_add_folio()`

```

114 static __always_inline
115 void lruvec_add_folio_tail(struct lruvec *lruvec, struct folio *folio)
116 {
117     enum lru_list lru = folio_lru_list(folio);
118
119     update_lru_size(lruvec, lru, folio_zonenum(folio),
120                    folio_nr_pages(folio));
121     /* This is not expected to be used on LRU_UNEVICTABLE */
122     list_add_tail(&folio->lru, &lruvec->lists[lru]);

```

123 }

Listing 11-14: include/linux/mm_inline.h: lruvec_add_folio_tail()

```

131 static __always_inline
132 void lruvec_del_folio(struct lruvec *lruvec, struct folio *folio)
133 {
134     enum lru_list lru = folio_lru_list(folio);
135
136     if (lru != LRU_UNEVICTABLE)
137         list_del(&folio->lru);
138     update_lru_size(lruvec, lru, folio_zonenum(folio),
139                     -folio_nr_pages(folio));
140 }
```

Listing 11-15: include/linux/mm_inline.h: lruvec_del_folio()

Each of these use `folio_lru_list()` to determine which LRU list a particular folio should be assigned to:-

```

74 /**
75  * folio_lru_list - Which LRU list should a folio be on?
76  * @folio: The folio to test.
77  *
78  * Return: The LRU list a folio should be on, as an index
79  * into the array of LRU lists.
80  */
81 static __always_inline enum lru_list folio_lru_list(struct folio *folio)
82 {
83     enum lru_list lru;
84
85     VM_BUG_ON_FOLIO(folio_test_active(folio) && folio_test_unevictable(
86                     folio), folio);
87
88     if (folio_test_unevictable(folio))
89         return LRU_UNEVICTABLE;
90
91     lru = folio_is_file_lru(folio) ? LRU_INACTIVE_FILE : LRU_INACTIVE_ANON
92     ;
93     if (folio_test_active(folio))
94         lru += LRU_ACTIVE;
95
96     return lru;
97 }
```

Listing 11-16: include/linux/mm_inline.h: folio_lru_list()

Everything is predicated on struct folio flags, tested via `folio_test_xxx()` (see chapter 2 on physical memory for details on how these function):-

- `folio_test_active()` – Testing whether the `PG_active` flag is set – When folios ought to be activated (or remain activated) this folio flag is set. If unset then the folio should exist on an inactive list. It is an error if both this and the `PG_unevictable` flag is set.
- `folio_test_unevictable()` – Testing whether the `PG_unevictable` flag is set – If a LRU-inhabiting user-mapped folio that should never be evicted then this folio flag is set, e.g. when `mlocking` folios in `__mlock_page()` or `__mlock_new_page()` or if the folio has the `PG_mlocked` flag set or otherwise has its mapping marked unevictable as tested by `folio_evictable()`.
- `folio_is_file_lru()` – Testing whether the `PG_swapbacked` flag is *not* set – This function determines whether the folio should be placed on a file LRU or not. If the `PG_swapbacked` folio flag is set, indicating that there is a swap backing cache available for the folio (and thus it could be swapped out) then it is considered anonymous, otherwise it is considered a file mapping and should be placed on a file LRU.

Note that the `enum lru_list` type is designed such that you can shift between inactive and active LRUs by simply adding `LRU_ACTIVE` to the value.

When folios are moved between lists they are removed from one and added to another using these fundamental functions, *except* for the special case of `isolate_lru_pages()` which performs the move directly (see below for more).

What invokes these functions? The principal interface is via *folio batches* (otherwise known as *pagevecs*), which batch up folios for placing on or removing from the LRU lists. As this is an important topic in of itself, we will discuss them in detail in the following section.

Meanwhile, let's examine the non-folio batch functions which interact with `lruvec_add_folio()`, `lruvec_add_folio_tail()` and `lruvec_del_folio()`:-

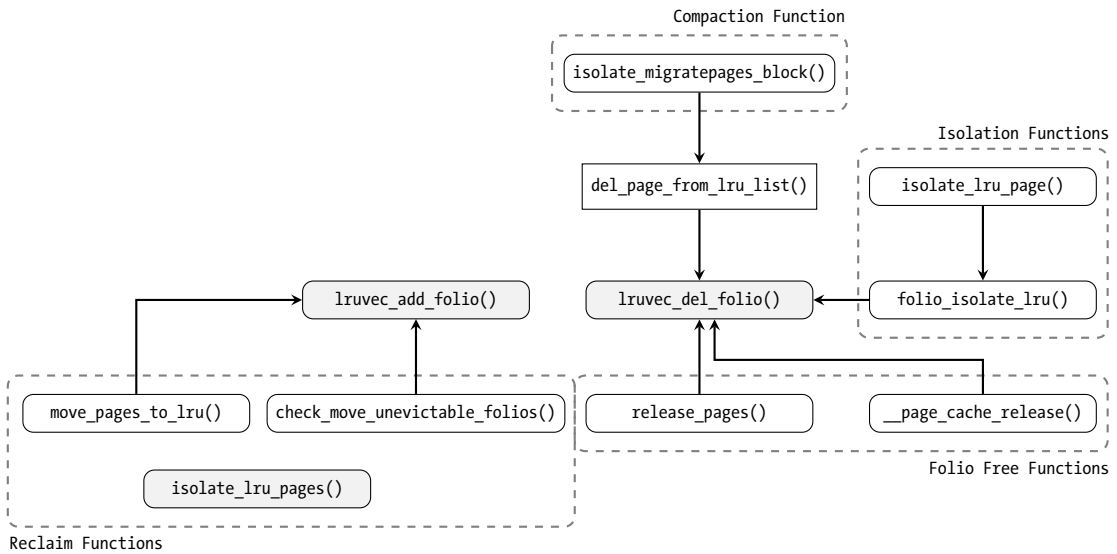


Figure 11-8: Non-batched LRU functions

Note that `lruvec_add_folio_tail()` is not referenced by a non-batch method at all.

There are two legacy non-folio functions `add_page_to_lru_list()` and `del_page_from_lru_list()` which simply wrap the relevant folio functions `lruvec_add_folio()` and `lruvec_del_folio()` respectively. Importantly, both of these assume `lru_lock` is held (only `lruvec_del_folio()` is used for non-batched LRU operations).

Examining each of the different groups of functions by type:-

- **compaction functions** – `isolate_migratepages_block()` works similarly to the isolation functions, however specifically targeting a page block of folios and isolating all of those contained within which are migrate-able. See the below discussion of the isolation function for more on what isolation means in this context, and the compaction chapter for more on compaction generally.
- **reclaim functions** – The key outlier of these functions is `isolate_lru_pages()` because it interacts with `lruvec` lists *directly*. This is special in that it is a key hot path for reclaim and the principal function for moving folios from `lruvecs` to internal reclaim lists, which are then used to perform the reclaim operations, invoked by both `shrink_inactive_list()` and `shrink_active_list()`. It is assumed the `lruvec` is held when it is called. The other reclaim functions which interact with `lruvecs` are `move_pages_to_lru()` and `check_move_unevictable_folios()` which both use the core `lruvec_*_folio()` functions to perform their actions.
- **folio free functions** – When folios are physically released (at the point their reference count reaches zero, tested via `put_page_testzero()`) they need to be removed from the `lruvec` if they are on one. This

is achieved via either `release_pages()` or `__page_cache_release()`. This will either occur on the folio batch/pagevec code path (ultimately by `folio_batch_release()` or `pagevec_release()` which ultimately invoke `release_pages()`, or via the `folio_put()` or `put_page()` code path which ultimately invoke `__page_cache_release()`. See chapter 2 on physical memory and figure 2.24 for more details.

- isolation functions – A fairly typical approach when operating on lruvec folios is to first *isolate* them from the lruvec before performing the required operations. This is achieved via `folio_isolate_lru()` or `isolate_lru_page()`. They ultimately invoke `lruvec_del_folio()` to pull the folio from its lruvec.

Note that the majority of operations discussed above *remove* folios from lruvecs. The reclaim operations when ‘adding’ folios to a list are actually simply updating stats on unevictable folios. Isolation and folio freeing are all about removing folios and at best reclaim might move folios from one list to another, when not removing on reclaim.

The majority of operations where folios are *added* to lruvecs are performed using folio batches, discussed in Section 11.7.

11.3 Direct Reclaim

Direct reclaim arises when the minimum watermark is breached for all zones an allocation could arise in. At this point the process blocks until reclaim frees sufficient pages in order that the minimum watermark of the zone into which we wish to allocate is no longer breached.

As `__alloc_pages_slowpath()` loops while progress is being made throughout direct reclaim, we simply have to try to make *some* reasonable progress on each invocation of `__alloc_pages_direct_reclaim()` (see Listing 11-17).

We examine the direct reclaim call stack in Figure 11-9.

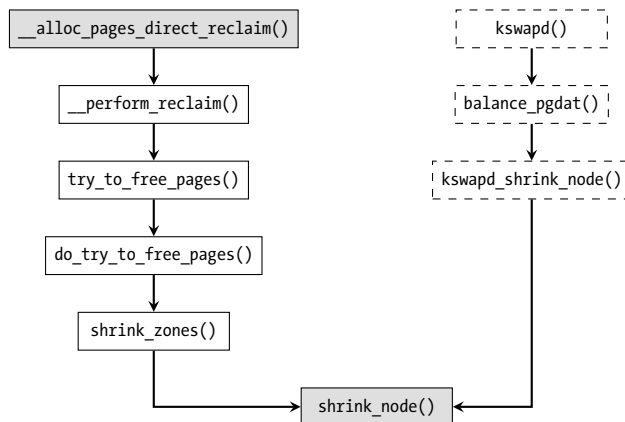


Figure 11-9: Direct Reclaim Code Path

As shown in Figure 11-9, direct reclaim is invoked via the `__alloc_pages_direct_reclaim()`, called from `__alloc_pages_slowpath()` as shown in Listing 11-1 and Section 11.1.

We examine `__alloc_pages_direct_reclaim()` in Listing 11-17 (eliding out of scope PSI memory pressure tracking).

```

4761 /* The really slow allocator path where we enter direct reclaim */
4762 static inline struct page *
4763 __alloc_pages_direct_reclaim(gfp_t gfp_mask, unsigned int order,
4764                             unsigned int alloc_flags, const struct alloc_context *ac,
4765                             unsigned long *did_some_progress)
4766 {
4767     struct page *page = NULL;
4768     :
4769     bool drained = false;
4770     :
4771     *did_some_progress = __perform_reclaim(gfp_mask, order, ac);
4772     if (unlikely(!(*did_some_progress)))
4773         goto out;
4774
4775 retry:
4776     page = get_page_from_freelist(gfp_mask, order, alloc_flags, ac);
4777
4778     /*
4779      * If an allocation failed after direct reclaim, it could be because
4780      * pages are pinned on the per-cpu lists or in high alloc reserves.
4781      * Shrink them and try again
4782      */
4783     if (!page && !drained) {
4784         unreserve_highatomic_pageblock(ac, false);
4785         drain_all_pages(NULL);
4786         drained = true;
4787         goto retry;
4788     }
4789 out:
4790     :
4791     return page;
4792 }

```

Listing 11-17: `mm/page_alloc.c: __alloc_pages_direct_reclaim()`

We start `__alloc_pages_direct_reclaim()` by deferring the actual reclaim to `__perform_reclaim()` (see Listing 11-18), a function which returns a value indicating whether progress was made. If none was, we simply exit returning `NULL`.

Otherwise, we try to complete the allocation—if this succeeds, we return this page via `get_page_from_freelist()` (see Listing 2-55 in Section 2.8.2 and Chapter 2).

If this fails, we try to unreserve the reserved high atomic page block. This is a page block (i.e. a physically contiguous folio of size `pageblock_order` reserved to assist with higher-order allocations in atomic context, which is a context in which a kernel process cannot sleep).

If we need to take this action, then we subsequently invoke `drain_all_pages()` to also ensure that all Per-CPU Pages are drained to free lists (see Section 2.7.3 in Chapter 2 for more details on these).

At this point we have done all we can, so we try this only once before retrying the `get_page_from_freelist()` operation. If this fails, we exit reporting failure.

Having deferred the heavy lifting of the operation to `__perform_reclaim()`, we now examine this in Listing 11-18 (eliding out of scope CPU set, lockdep and realtime scheduler logic).

```

4735 /* Perform direct synchronous page reclaim */
4736 static unsigned long
4737 __perform_reclaim(gfp_t gfp_mask, unsigned int order,
4738                  const struct alloc_context *ac)
4739 {
4740     unsigned int noreclaim_flag;
4741     unsigned long progress;
4742     :
4745     /* We now go into synchronous reclaim */
4746     :
4748     noreclaim_flag = memalloc_noreclaim_save();
4749
4750     progress = try_to_free_pages(ac->zonelist, order, gfp_mask,
4751                                ac->nodemask);
4752
4753     memalloc_noreclaim_restore(noreclaim_flag);
4754     :
4758     return progress;
4759 }
```

Listing 11-18: `mm/page_alloc.c: __perform_reclaim()`

The `__perform_reclaim()` function ultimately wraps `try_to_free_pages()`, but in addition it invokes `memalloc_noreclaim_save()` and `memalloc_noreclaim_restore()` which set (and clear if not previously set) the `PF_MEMALLOC` flag in the current thread's `struct task_struct->flags` field.

See Section 2.6.7 in Chapter 2 for more on `memalloc` flags, but broadly these are per-task (i.e. from userland's perspective, per-thread) flags which affect memory allocation globally until unset.

The `PF_MEMALLOC` flag avoids recursive reclaim occurring when allocating memory as part of performing reclaim (which could otherwise have resulted in races or reclaim becoming stuck), as checked in `__alloc_pages_slowpath()` and `__need_reclaim()`.

It also results in all watermarks being ignored in the process of providing any allocations reclaim needs as checked by `__gfp_pfmemalloc_flags()`.

We wrap the `try_to_free_pages()` function here, which we examine in Listing 11-19 (eliding build bug checks and trace hooks).

```

3801 unsigned long try_to_free_pages(struct zonelist *zonelist, int order,
3802                               gfp_t gfp_mask, nodemask_t *nodemask)
3803 {
3804     unsigned long nr_reclaimed;
3805     struct scan_control sc = {
3806         .nr_to_reclaim = SWAP_CLUSTER_MAX,
3807         .gfp_mask = current_gfp_context(gfp_mask),
3808         .reclaim_idx = gfp_zone(gfp_mask),
3809         .order = order,
3810         .nodemask = nodemask,
3811         .priority = DEF_PRIORITY,
3812         .may_writepage = !laptop_mode,
3813         .may_unmap = 1,
3814         .may_swap = 1,
3815     };
3816     :
3825     /*
3826      * Do not enter reclaim if fatal signal was delivered while throttled.
3827      * 1 is returned so that the page allocator does not OOM kill at this
3828      * point.
3829      */
3830     if (throttle_direct_reclaim(sc.gfp_mask, zonelist, nodemask))
3831         return 1;
3832     :
3833     set_task_reclaim_state(current, &sc.reclaim_state);
3834     :
3836     nr_reclaimed = do_try_to_free_pages(zonelist, &sc);
3837     :
3839     set_task_reclaim_state(current, NULL);
3840     :
3841     return nr_reclaimed;
3842 }

```

Listing 11-19: `mm/vmscan.c: try_to_free_pages()`

We start by establishing a `struct scan_control` object which we thread through the reclaim operation. We examine this in Listing 11-36 and Section 11.5.

Here we set the number of pages to reclaim equal to `SWAP_CLUSTER_MAX`, which is equal to the batch size of pages written to a file-system swap file, and thus forms a natural batch size across reclaim operations (see Section 12.2.3 and Chapter 12 to see how swap write to disk functions).

We indicate that we may unmap memory and swap when reclaiming, and whether or not we may write out dirty folios to disk is dictated by whether “laptop mode” is enabled. In this mode, reclaim attempts to avoid writing to

disk as much as possible in order to avoid power-hungry rotational disk spin up (less relevant in the modern era of SSDs).

Therefore, if laptop mode is enabled, we prefer not to write to disk otherwise we do not try to reduce this.

In addition we set the “priority” of the operation to `DEF_PRIORITY`. This value determines how much of the LRU vectors should be examined (see Section 11.2), expressed as a shift value, and defaulting to 12.

NOTE

The priority value specifies how much the queue length should be right-shifted, so if it is equal to 12 to instance, this implies that the queue will be bit-shifted by 12 bits to the right. This is equivalent to dividing by 4,096, meaning we scan one-4,096th of the LRU vectors for folios to free. As this reduces, we scan more of the lists.

We maintain a priority so reclaim does as little work as possible in order to effect reclaim, only reluctantly scanning more pages if lower priorities (that is, higher priority values resulting in fewer pages scanned) fail to provide us with sufficient reclaim.

We determine if we should throttle direct reclaim via the `throttle_direct_reclaim()` function, which we examine in Listing 11-77. If so, we exit, returning one indicating that forward progress has been made and `__alloc_pages_slowpath()` is permitted to loop around and try the allocation again, engaging in direct reclaim if necessary.

Otherwise, we assign a pointer to a state object describing reclaim state to the current thread via `set_task_reclaim_state()` before performing the operation via `do_try_to_free_pages()` which we examine in Listing 11-20 (eliding out of scope delay accounting, cgroup logic, and working set logic and a statistic update).

```

3567 /*
3568  * This is the main entry point to direct page reclaim.
3569  *
3570  * If a full scan of the inactive list fails to free enough memory then we
3571  * are "out of memory" and something needs to be killed.
3572  *
3573  * If the caller is !__GFP_FS then the probability of a failure is reasonably
3574  * high - the zone may be full of dirty or under-writeback pages, which this
3575  * caller can't do much about. We kick the writeback threads and take
3576  * explicit
3577  * naps in the hope that some of these pages can be written. But if the
3578  * allocating task holds filesystem locks which prevent writeout this might
3579  * not
3580  * work, and the allocation attempt will fail.
3581  *
3582  * returns:    0, if no pages reclaimed
3583  *             else, the number of pages reclaimed
3584  */
3585 static unsigned long do_try_to_free_pages(struct zonelist *zonelist,
3586                                         struct scan_control *sc)

```

```

3585 {
3586     int initial_priority = sc->priority;
3587     :
3588     struct zoneref *z;
3589     struct zone *zone;
3590 retry:
3591     :
3592     do {
3593         :
3600         sc->nr_scanned = 0;
3601         shrink_zones(zonelist, sc);
3602
3603         if (sc->nr_reclaimed >= sc->nr_to_reclaim)
3604             break;
3605
3606         if (sc->compaction_ready)
3607             break;
3608
3609         /*
3610          * If we're getting trouble reclaiming, start doing
3611          * writepage even in laptop mode.
3612          */
3613         if (sc->priority < DEF_PRIORITY - 2)
3614             sc->may_writepage = 1;
3615     } while (--sc->priority >= 0);
3616
3617     last_pgdat = NULL;
3618     for_each_zone_zonelist_nodemask(zone, z, zonelist, sc->reclaim_idx,
3619                                     sc->nodemask) {
3620         if (zone->zone_pgdat == last_pgdat)
3621             continue;
3622         last_pgdat = zone->zone_pgdat;
3623
3624         snapshot_refaults(sc->target_mem_cgroup, zone->zone_pgdat);
3625     }
3626
3627     if (sc->nr_reclaimed)
3628         return sc->nr_reclaimed;
3629
3630     /* Aborted reclaim to try compaction? don't OOM, then */
3631     if (sc->compaction_ready)
3632         return 1;
3633
3634     /*
3635      * We make inactive:active ratio decisions based on the node's
3636      * composition of memory, but a restrictive reclaim_idx or a

```

```

3645      * memory.low cgroup setting can exempt large amounts of
3646      * memory from reclaim. Neither of which are very common, so
3647      * instead of doing costly eligibility calculations of the
3648      * entire cgroup subtree up front, we assume the estimates are
3649      * good, and retry with forcible deactivation if that fails.
3650      */
3651      if (sc->skipped_deactivate) {
3652          sc->priority = initial_priority;
3653          sc->force_deactivate = 1;
3654          sc->skipped_deactivate = 0;
3655          goto retry;
3656      }
3657      :
3669      return 0;
3670 }

```

Listing 11-20: mm/vmscan.c: do_try_to_free_pages()

In `do_try_to_free_pages()` we repeatedly try to free `SWAP_CLUSTER_MAX` pages as we iterate downwards through priority (See the discussion around Listing 11-19 for a definition of reclaim priority), deferring to `shrink_zones()` to perform the actual reclaim operation on each occasion (see Listing 11-21).

If the number of folios reclaimed is equal to or exceeds the number requested to be reclaimed (that is, `SWAP_CLUSTER_MAX`), then we have successfully completed reclaim, and exit the loop.

Otherwise, if the struct `scan_control->compaction_ready` field is set by `shrink_zones()`, this indicates that we ought to abort reclaim to attempt to compact memory.

See Listing 11-36 and Section 11.5 for a detailed examination of the struct `scan_control` type.

Compaction is out of scope for the book, but broadly it obtains more higher order folios by moving folios such that adjacent lower order folios can be coalesced.

If we loop sufficient times that struct `scan_control->priority` drops three or more levels below `DEF_PRIORITY`, then this indicates we are having trouble reclaiming memory, and thus we should permit dirty folio writeback even if laptop mode is enabled.

After the loop is completed, we iterate through each node from which we could allocate, and for each invoke `snapshot_refaults()` (see Listing 11-38) which stores the current statistics for “refaults” for this node. See Section 11.5.2 and the discussion around Listing 11-41 for an explanation as to how this is used.

Then we check to see if any pages were reclaimed, returning if any were—at this point we have tried to reclaim as much as we possibly can so however many pages we have managed has to suffice.

Otherwise, we repeat our compaction check to ensure we don’t incorrectly indicate that reclaim failed when we are simply aborting it to try compaction.

After this, we consider one more case, where we skipped deactivating active folios (that is, moving them from the active LRU vector to the inactive one). If so, given reclaim has yielded nothing, it is worth forcing the matter and trying this regardless and retrying the operation on that basis.

We examine the key `shrink_zones()` function which `do_try_to_free_pages()` invokes in Listing 11-21 (eliding out of scope `cgroup` and 32-bit architecture-specific logic).

```

3455 /*
3456  * This is the direct reclaim path, for page-allocating processes. We only
3457  * try to reclaim pages from zones which will satisfy the caller's allocation
3458  * request.
3459  *
3460  * If a zone is deemed to be full of pinned pages then just give it a light
3461  * scan then give up on it.
3462  */
3463 static void shrink_zones(struct zonelist *zonelist, struct scan_control *sc)
3464 {
3465     struct zoneref *z;
3466     struct zone *zone;
3467     unsigned long nr_soft_reclaimed;
3468     unsigned long nr_soft_scanned;
3469     :
3470     pg_data_t *last_pgdat = NULL;
3471     pg_data_t *first_pgdat = NULL;
3472     :
3473     for_each_zone_zonelist_nodemask(zone, z, zonelist,
3474                                     sc->reclaim_idx, sc->nodemask) {
3475         :
3476         /*
3477          * If we already have plenty of memory free for
3478          * compaction in this zone, don't free any more.
3479          * Even though compaction is invoked for any
3480          * non-zero order, only frequent costly order
3481          * reclamation is disruptive enough to become a
3482          * noticeable problem, like transparent huge
3483          * page allocations.
3484          */
3485         if (IS_ENABLED(CONFIG_COMPACTION) &&
3486             sc->order > PAGE_ALLOC_COSTLY_ORDER &&
3487             compaction_ready(zone, sc)) {
3488             sc->compaction_ready = true;
3489             continue;
3490         }
3491         :
3492         /*
3493          * Shrink each node in the zonelist once. If the

```

```

3513             * zonelist is ordered by zone (not the default) then
                a
3514             * node may be shrunk multiple times but in that case
3515             * the user prefers lower zones being preserved.
3516             */
3517             if (zone->zone_pgdat == last_pgdat)
3518                 continue;
3519             :
3545             if (!first_pgdat)
3546                 first_pgdat = zone->zone_pgdat;
3547             :
3551             last_pgdat = zone->zone_pgdat;
3552             shrink_node(zone->zone_pgdat, sc);
3553         }
3554         :
3555         if (first_pgdat)
3556             consider_reclaim_throttle(first_pgdat, sc);
3557         :
3558     }

```

Listing 11-21: mm/vmscan.c: shrink_zones()

In `shrink_zones()` we iterate through each zone that we could allocate from, checking to see whether we are able to compact in instances where the order of the folio being allocated could benefit from this (i.e. greater than `PAGE_ALLOC_COSTLY_ORDER`), before marking the struct `scan_control->compaction_ready` field to indicate to calling functions that compaction should be attempted in lieu of reclaim, continuing the loop to examine the next zone.

We actually perform the reclaim per-node, so when we find a zone which is eligible for reclaim, we will only instigate reclaim if it is a node different from the previous.

As the comment alludes to, the zone list ordering means this will usually result in each node being considered once, though if the zonelist ordering has changed, this could result in nodes being shrunk multiple times—however if the zonelist has been thus altered, this should align with user requirements (see Section 2.4 for more on nodes, zones and zonelists).

We ultimately invoke `shrink_node()` to perform the reclaim. This is the core reclaim path shared by both direct and indirect reclaim and thus we examine this in Section 11.5 and Listing 11-40.

For the first node we encounter in each iteration, we also determine whether reclaim throttle should be performed via `consider_reclaim_throttle()`, a subject we address in Section 11.6, examining this function in Listing 11-74.

11.4 Indirect Reclaim

Indirect reclaim is reclaim that is performed as a background task and is implemented using a per-node kernel thread which waits to be woken when the need arises (for instance, when an allocation causes a zone to drop below the low water mark in `__alloc_pages_slowpath()`, see the portion of this function described in Listing 11-2).

These indirect reclaim processes are also known as `kswapd`, and are named accordingly, suffixed by the number of the node which they perform reclaim for (for instance if a `kswapd` thread reclaims on node 0, it will appear as `[kswapd0]`).

We can see the code path through which indirect reclaim operates in Figure 11-10, which starts in the `kswapd()` function (see Listing 11-26) executed in a kernel thread as described in Section 11.4.2.

We will examine how this kernel thread is initialised and executed in Section 11.4.1.

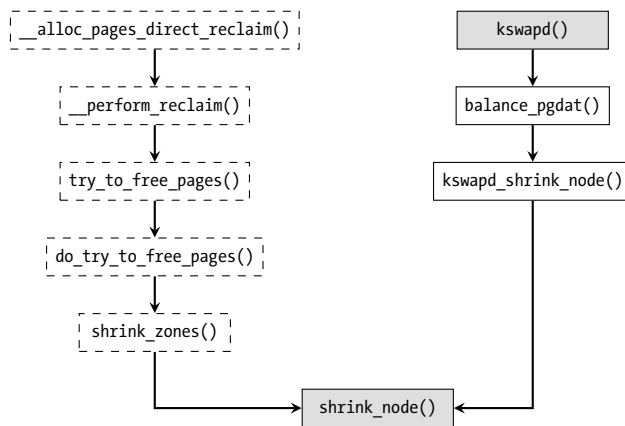


Figure 11-10: Indirect Reclaim Code Path

11.4.1 Initialisation

The kernel threads are initialised in `kswapd_init()`, which we examine in Listing 11-22.

```

4672 static int __init kswapd_init(void)
4673 {
4674     int nid;
4675
4676     swap_setup();
4677     for_each_node_state(nid, N_MEMORY)
4678         kswapd_run(nid);
4679     return 0;
4680 }
  
```

Listing 11-22: `mm/vmscan.c: kswapd_init()`

The `kswapd_init()` function initialises the swap via `swap_setup()` (see Listing 12-46 in Section 12.3.4 and Chapter 12) and then iterates through all nodes which have memory attached, running `kswapd_run()` against each, which we examine in Listing 11-23.

```

4642 void kswapd_run(int nid)
4643 {
4644     pg_data_t *pgdat = NODE_DATA(nid);
4645
4646     if (pgdat->kswapd)
4647         return;
4648
4649     pgdat->kswapd = kthread_run(kswapd, pgdat, "kswapd%d", nid);
4650     if (IS_ERR(pgdat->kswapd)) {
4651         /* failure at boot is fatal */
4652         BUG_ON(system_state < SYSTEM_RUNNING);
4653         pr_err("Failed to start kswapd on node %d\n", nid);
4654         pgdat->kswapd = NULL;
4655     }
4656 }

```

Listing 11-23: mm/vmscan.c: kswapd_run()

The `kswapd_run()` function causes `kswapd()` to be invoked as a kernel thread, placing the kernel thread task of type `struct task_struct` in the node's `struct pglist_data->kswapd` field.

If this fails at boot when this is initialised then we cause a kernel oops, which will lead to the kernel failing to boot altogether. This function might also be invoked during a hot plugging operation, but discussion of this is out of scope for the book.

Typically the `kswapd` process is woken by the `wake_all_kswaps()` function, and this is what `__alloc_pages_slowpath()` (see Listing 11-2) invokes to wake indirect reclaim when a zone drops below the low watermark. We examine this function in Listing 11-24.

```

4796 static void wake_all_kswaps(unsigned int order, gfp_t gfp_mask,
4797                             const struct alloc_context *ac)
4798 {
4799     struct zoneref *z;
4800     struct zone *zone;
4801     pg_data_t *last_pgdat = NULL;
4802     enum zone_type highest_zoneidx = ac->highest_zoneidx;
4803
4804     for_each_zone_zonelist_nodemask(zone, z, ac->zonelist, highest_zoneidx
,
4805                                     ac->nodemask) {
4806         if (!managed_zone(zone))
4807             continue;
4808         if (last_pgdat != zone->zone_pgdat) {

```

```

4809             wakeup_kswapd(zone, gfp_mask, order, highest_zoneidx);
4810             last_pgdat = zone->zone_pgdat;
4811         }
4812     }
4813 }

```

Listing 11-24: mm/page_alloc.c: wake_all_kswapds()

We iterate through each zone which the allocation could make use of, using `managed_zone()` to ensure that each has pages within its ranged managed by the buddy allocator, and calling `wakeup_kswapd()` for each of these node. We examine this function in Listing 11-25 (eliding out of scope cgroup logic and tracing hooks).

```

4555 void wakeup_kswapd(struct zone *zone, gfp_t gfp_flags, int order,
4556                   enum zone_type highest_zoneidx)
4557 {
4558     pg_data_t *pgdat;
4559     enum zone_type curr_idx;
4560
4561     if (!managed_zone(zone))
4562         return;
4563     :
4564     pgdat = zone->zone_pgdat;
4565     curr_idx = READ_ONCE(pgdat->kswapd_highest_zoneidx);
4566
4567     if (curr_idx == MAX_NR_ZONES || curr_idx < highest_zoneidx)
4568         WRITE_ONCE(pgdat->kswapd_highest_zoneidx, highest_zoneidx);
4569
4570     if (READ_ONCE(pgdat->kswapd_order) < order)
4571         WRITE_ONCE(pgdat->kswapd_order, order);
4572
4573     if (!waitqueue_active(&pgdat->kswapd_wait))
4574         return;
4575
4576     /* Hopeless node, leave it to direct reclaim if possible */
4577     if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES ||
4578         (pgdat_balanced(pgdat, order, highest_zoneidx) &&
4579          !pgdat_watermark_boosted(pgdat, highest_zoneidx))) {
4580         /*
4581          * There may be plenty of free memory available, but it's too
4582          * fragmented for high-order allocations. Wake up kcompactd
4583          * and rely on compaction_suitable() to determine if it's
4584          * needed. If it fails, it will defer subsequent attempts to
4585          * ratelimit its work.
4586          */
4587         if (!(gfp_flags & __GFP_DIRECT_RECLAIM))
4588             wakeup_kcompactd(pgdat, order, highest_zoneidx);
4589         return;
4590     }
4591 }

```

```

4593     }
    :
4597     wake_up_interruptible(&pgdat->kswapd_wait);
4598 }

```

Listing 11-25: mm/vmscan.c: wakeup_kswapd()

The `wakeup_kswapd()` function starts by asserting again that the zone spans buddy allocator-managed pages (see Section 2.4 in Chapter 2 for more on nodes and zones) via `managed_zone()`.

It then updates various node statistics in the struct `pglist_data` metadata object describing the node (type aliased as `pg_data_t`).

We store the highest zone in terms of zone index we have submitted to indirect reclaim in struct `pglist_data->kswapd_highest_zoneidx` and the highest order in struct `pglist_data->order`.

We then check to see if the struct `pglist_data->kswapd_wait` wait queue is currently active, i.e. whether `kswapd` is sleeping waiting to be woken up. If not, then we have no capacity to wake the indirect reclaim thread so abort.

Wait queues are a means by which the kernel allows for threads to wait for certain events to occur, in this instance `kswapd` waits to be woken up.

We then consider an edge case—we keep track of the number of times `kswapd` has failed to reclaim memory in struct `pglist_data->kswapd_failures`. Once this reaches or exceeds `MAX_RECLAIM_RETRIES` (hard-coded to 16 attempts), then further attempts at indirect reclaim are likely to be fruitless and we need to leave the node to direct reclaim rather than indirect.

In addition we consider the case where a node is balanced, i.e. has reached or exceeded its high water mark and in addition is not subject to a zone boost (see Section 2.4 in Chapter 2 for more on nodes, zone and zone boost).

In this case, we should not be in a position where `kswapd` was invoked. If we still were, either the wakeup was spurious, or indirect reclaim is unable to resolve the issue, so we must abort the indirect reclaim attempt.

If direct reclaim is not permitted for this allocation, as indicated by the lack of `__GFP_DIRECT_RECLAIM` flag in the specified Get Free Pages (GFP) flag mask (see Section 2.6 in Chapter 2 for more on GFP flags), then we attempt to wake up the compaction process, `kcompactd` to perform compaction.

Compaction is a process by which the kernel moves memory around to allow for the coalescence of lower order folios in to higher order folios in order to provide higher order folios for allocation. This topic is out of scope for the book.

Finally, if nothing is preventing indirect reclaim from commencing, we wake it up.

11.4.2 *kswapd Thread*

The `kswapd()` function (see Listing 11-26) implements the indirect reclaim implementation, this is what the `kswapd` kernel thread runs for each node. We can see in Figure 11-10 how this triggers reclaim, ultimately invoking the core reclaim function `shrink_node()` (see Listing 11-40 in Section 11.5).

We examine the indirect reclaim kernel thread function `kswapd()` in Listing 11-26 (eliding out of scope task freezing logic and trace hooks).

```

4456 /*
4457  * The background pageout daemon, started as a kernel thread
4458  * from the init process.
4459  *
4460  * This basically trickles out pages so that we have _some_
4461  * free memory available even if there is no other activity
4462  * that frees anything up. This is needed for things like routing
4463  * etc, where we otherwise might have all activity going on in
4464  * asynchronous contexts that cannot page things out.
4465  *
4466  * If there are applications that are active memory-allocators
4467  * (most normal use), this basically shouldn't matter.
4468  */
4469 static int kswapd(void *p)
4470 {
4471     unsigned int alloc_order, reclaim_order;
4472     unsigned int highest_zoneidx = MAX_NR_ZONES - 1;
4473     pg_data_t *pgdat = (pg_data_t *)p;
4474     struct task_struct *tsk = current;
4475     const struct cpumask *cpumask = cpumask_of_node(pgdat->node_id);
4476
4477     if (!cpumask_empty(cpumask))
4478         set_cpus_allowed_ptr(tsk, cpumask);
4479
4480     /*
4481      * Tell the memory management that we're a "memory allocator",
4482      * and that if we need more memory we should get access to it
4483      * regardless (see "__alloc_pages()"). "kswapd" should
4484      * never get caught in the normal page freeing logic.
4485      *
4486      * (Kswapd normally doesn't need memory anyway, but sometimes
4487      * you need a small amount of memory in order to be able to
4488      * page out something else, and this flag essentially protects
4489      * us from recursively trying to free more memory as we're
4490      * trying to free the first piece of memory in the first place).
4491      */
4492     tsk->flags |= PF_MEMALLOC | PF_KSWAPD;
4493     :
4495     WRITE_ONCE(pgdat->kswapd_order, 0);
4496     WRITE_ONCE(pgdat->kswapd_highest_zoneidx, MAX_NR_ZONES);
4497     atomic_set(&pgdat->nr_writeback_throttled, 0);
4498     for ( ; ; ) {
4499         bool ret;
4500
4501         alloc_order = reclaim_order = READ_ONCE(pgdat->kswapd_order);

```

```

4502             highest_zoneidx = kswapd_highest_zoneidx(pgdat,
4503                                     highest_zoneidx);
4504
4505 kswapd_try_sleep:
4506             kswapd_try_to_sleep(pgdat, alloc_order, reclaim_order,
4507                                     highest_zoneidx);
4508
4509             /* Read the new order and highest_zoneidx */
4510             alloc_order = READ_ONCE(pgdat->kswapd_order);
4511             highest_zoneidx = kswapd_highest_zoneidx(pgdat,
4512                                                         highest_zoneidx);
4513             WRITE_ONCE(pgdat->kswapd_order, 0);
4514             WRITE_ONCE(pgdat->kswapd_highest_zoneidx, MAX_NR_ZONES);
4515             :
4516             if (kthread_should_stop())
4517                 break;
4518             :
4519             /*
4520              * Reclaim begins at the requested order but if a high-order
4521              * reclaim fails then kswapd falls back to reclaiming for
4522              * order-0. If that happens, kswapd will consider sleeping
4523              * for the order it finished reclaiming at (reclaim_order)
4524              * but kcompactd is woken to compact for the original
4525              * request (alloc_order).
4526              */
4527             :
4528             reclaim_order = balance_pgdat(pgdat, alloc_order,
4529                                             highest_zoneidx);
4530             if (reclaim_order < alloc_order)
4531                 goto kswapd_try_sleep;
4532         }
4533
4534         tsk->flags &= ~(PF_MEMALLOC | PF_KSWAPD);
4535
4536         return 0;
4537     }

```

Listing 11-26: mm/vmscan.c: *kswapd()*

We start by setting various initial local variables and restricting the *kswapd* thread to the same CPUs that the node is restricted to (if any such restrictions apply). Note in particular that we default to the highest possible zone to start allocating from.

We then apply some task-specific memory allocation flags—*PF_MEMALLOC* and *PF_KSWAPD*. The former prevents reclaim being entered and allows use of all memory reserves. The latter flags this process as the *kswapd* process, as checked for by *current_is_kswapd()* in various places.

We then reset various node fields relating to `kswapd`, `struct pglist_data->kswapd_order` which stores the order of the allocation which triggered the current indirect reclaim, `struct pglist_data->kswapd_highest_zoneidx` which stores the highest zone index (we set it to an invalid value here, we will discuss why shortly) and `struct pglist_data->nr_writeback_throttled` which tracks how often writeback throttling has occurred.

We then enter the main loop—this starts by reading the last allocation order which triggered the `kswapd` order, and the highest zone index which we determine via `kswapd_highest_zoneidx()` which we examine in Listing 11-27.

```

4357 /*
4358  * The pgdat->kswapd_highest_zoneidx is used to pass the highest zone index to
4359  * be reclaimed by kswapd from the waker. If the value is MAX_NR_ZONES which
      is
4360  * not a valid index then either kswapd runs for first time or kswapd couldn't
4361  * sleep after previous reclaim attempt (node is still unbalanced). In that
4362  * case return the zone index of the previous kswapd reclaim cycle.
4363  */
4364 static enum zone_type kswapd_highest_zoneidx(pg_data_t *pgdat,
4365                                             enum zone_type prev_highest_zoneidx
      )
4366 {
4367     enum zone_type curr_idx = READ_ONCE(pgdat->kswapd_highest_zoneidx);
4368
4369     return curr_idx == MAX_NR_ZONES ? prev_highest_zoneidx : curr_idx;
4370 }
```

Listing 11-27: `mm/vmscan.c: kswapd_highest_zoneidx()`

The key thing to note about `kswapd_highest_zoneidx()` is that, if the `struct pglist_data->kswapd_highest_zoneidx` field is set to `MAX_NR_ZONES`, then we reset it to the previous highest zone index, which in `kswapd()` for the first invocation is set to the maximum valid zone, i.e. `MAX_NR_ZONES - 1`.

We then invoke `kswapd_try_to_sleep()` which waits for `kswapd` to be woken up for reclaim. We examine this in Listing 11-28 shortly.

We read these fields, and then immediately reset them. We then check whether the kernel thread should terminate via the standard `kthread_should_stop()` function, before performing the actual reclaim via the `balance_pgdat()` function, which we examine in Listing 11-32.

As per the comment, `balance_pgdat()` returns the order that reclaim finished reclaiming at (which might have been reset to a lower value such as `order-0`). If it did, then we do not simply loop around and reset the allocation order, but instead jump directly to invoking `kswapd_try_to_sleep()` which will both cause compaction to be performed at the original allocation order, and, if the reclaim order we last finished reclaiming for is higher than that just set, consider reclaiming at the higher order (see the logic in `kswapd_try_to_sleep()` and Listing 11-28 to see how this is determined).

Finally, if the loop does terminate, we clear the task's PF_MEMALLOC and PF_KSWAPD flags as the thread is no longer being used for kswapd purposes and is about to be terminated.

11.4.3 kswapd Sleeping

A key part of the indirect reclaim operation is sleeping until we have work to do. We wait using `kswapd_try_to_sleep()`, which we examine in Listing 11-28 (eliding tracing hooks).

```

4372 static void kswapd_try_to_sleep(pg_data_t *pgdat, int alloc_order, int
        reclaim_order,
4373                                unsigned int highest_zoneidx)
4374 {
4375     long remaining = 0;
4376     DEFINE_WAIT(wait);
4377
4378     if (freezing(current) || kthread_should_stop())
4379         return;
4380
4381     prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE);
4382
4383     /*
4384      * Try to sleep for a short interval. Note that kcompactd will only be
4385      * woken if it is possible to sleep for a short interval. This is
4386      * deliberate on the assumption that if reclaim cannot keep an
4387      * eligible zone balanced that it's also unlikely that compaction will
4388      * succeed.
4389      */
4390     if (prepare_kswapd_sleep(pgdat, reclaim_order, highest_zoneidx)) {
4391         /*
4392          * Compaction records what page blocks it recently failed to
4393          * isolate pages from and skips them in the future scanning.
4394          * When kswapd is going to sleep, it is reasonable to assume
4395          * that pages and compaction may succeed so reset the cache.
4396          */
4397         reset_isolation_suitable(pgdat);
4398
4399         /*
4400          * We have freed the memory, now we should compact it to make
4401          * allocation of the requested order possible.
4402          */
4403         wakeup_kcompactd(pgdat, alloc_order, highest_zoneidx);
4404
4405         remaining = schedule_timeout(HZ/10);
4406
4407         /*
4408          * If woken prematurely then reset kswapd_highest_zoneidx and

```

```

4409         * order. The values will either be from a wakeup request or
4410         * the previous request that slept prematurely.
4411         */
4412         if (remaining) {
4413             WRITE_ONCE(pgdat->kswapd_highest_zoneidx,
4414                        kswapd_highest_zoneidx(pgdat,
4415                                                highest_zoneidx));
4416
4417             if (READ_ONCE(pgdat->kswapd_order) < reclaim_order)
4418                 WRITE_ONCE(pgdat->kswapd_order, reclaim_order)
4419         }
4420
4421         finish_wait(&pgdat->kswapd_wait, &wait);
4422         prepare_to_wait(&pgdat->kswapd_wait, &wait, TASK_INTERRUPTIBLE
4423     );
4424     }

```

Listing 11-28: mm/vmscan.c: kswapd_try_to_sleep() Short sleep

We start `kswapd_try_to_sleep()` by declaring a wait queue entry via `DEFINE_WAIT()`. This is what we will add to our wait queue struct `pglist_data->kswapd_wait` in order to be woken when there is reclaim to perform.

Discussion of wait queues is out of the scope of the book, but are broadly a means by which threads can wait for an event to occur, sleeping in the meantime.

We then check if the task is being frozen (discussion of which is outside of the scope of the book), or the kernel thread should be terminated, in both cases we must exit early.

We then add the wait queue entry to the struct `pglist_data->kswapd_wait` wait queue via `prepare_to_wait()`, which also sets the current thread's status to interruptible sleep. When the task is scheduled, it will enter into this state until the wait queue wakes it.

We determine whether we need to sleep via `prepare_kswapd_sleep()` which we examine in Listing 11-30 shortly.

If we do need to sleep, then we then proceed by trying to perform a brief sleep first, in order that we can detect whether we were woken up quickly.

If we do need to sleep then firstly, we reset compaction state via `reset_isolation_suitable()` which frees some memory and try waking up the compaction kernel thread, `kcompactd`, in order to potentially free some additional memory.

We sleep via `schedule_timeout()`, specifying that we sleep for 0.1 seconds (specified in units of jiffies, an internal kernel measure of time).

Using this function means we specify a timeout and thus we can detect if this timeout was reached or not.

If so, then this might indicate that we slept prematurely after a failed higher order reclaim attempt, and therefore we must be careful to not pre-

maturely reduce the order of the folios we attempt to reclaim. We therefore reset the highest zone we examine, in case it was reset to MAX_NR_ZONES via `kswapd_highest_zoneidx()` (see Listing 11-27), and choose the maximum of the prior reclaim order and the newly established one.

After we have performed this initial attempt at a sleep, we remove the wait entry from the wait queue via `finish_wait()`, and reset it ready to sleep again via `prepare_to_wait()`.

We then consider a longer sleep, explored in Listing 11-29.

```

4425      /*
4426      * After a short sleep, check if it was a premature sleep. If not,
4427      then
4428      * go fully to sleep until explicitly woken up.
4429      */
4429      if (!remaining &&
4430          prepare_kswapd_sleep(pgdat, reclaim_order, highest_zoneidx)) {
4431          :
4432          /*
4433          * vmstat counters are not perfectly accurate and the
4434          estimated
4435          * value for counters such as NR_FREE_PAGES can deviate from
4436          the
4437          * true value by nr_online_cpus * threshold. To avoid the zone
4438          the
4439          * watermarks being breached while under pressure, we reduce
4440          the
4441          * per-cpu vmstat threshold while kswapd is awake and restore
4442          * them before going back to sleep.
4443          */
4444          set_pgdat_percpu_threshold(pgdat, calculate_normal_threshold);
4445          if (!kthread_should_stop())
4446              schedule();
4447          set_pgdat_percpu_threshold(pgdat, calculate_pressure_threshold
4448          );
4449      } else {
4450          if (remaining)
4451              count_vm_event(KSWAPD_LOW_WMARK_HIT_QUICKLY);
4452          else
4453              count_vm_event(KSWAPD_HIGH_WMARK_HIT_QUICKLY);
4454      }
4455      finish_wait(&pgdat->kswapd_wait, &wait);
4456  }

```

Listing 11-29: `mm/vmscan.c: kswapd_try_to_sleep()` Long Sleep

We examine the portion of `kswapd_try_to_sleep()` which considers performing a long-term sleep in Listing 11-29.

We only perform a longer sleep if the prior attempt at a short sleep did not result in an early wake up, as indicated by remaining being equal to zero, i.e. the previously scheduling attempting having timed out. We also check `prepare_kswapd_sleep()` once again (see Listing 11-30) before immediately sleeping.

Note that, as per the comment, we update the thresholds for the `NR_FREE_PAGES` zone statistic we use to assess reclaim decisions for this node.

This is because, for scaling purposes, we batch up statistics in per-CPU lists and flush them on a regular basis. In this circumstance however, it becomes critical that we make this more accurate to avoid accidentally breaching watermarks while `kswapd` sleeps.

The details of this are out of scope for the book so we don't examine them in detail.

The `schedule()` function causes the scheduler to apply the `TASK_INTERRUPTIBLE` state set by `prepare_to_wait()`, i.e. to enter an interruptible sleep indefinitely until woken from that sleep via the wait queue.

This therefore indicates that `kswapd` enters an indefinite sleep until something wakes it, which is the desired behaviour.

Once woken, the statistical thresholds for the node are restored, and the wait queue is cleared via `finish_wait()`.

At this point, the sleep logic is concluded and any relevant `kswapd` sleeping will have occurred.

We now examine the key `prepare_kswapd_sleep()` function in Listing 11-30.

```

4021 /*
4022  * Prepare kswapd for sleeping. This verifies that there are no processes
4023  * waiting in throttle_direct_reclaim() and that watermarks have been met.
4024  *
4025  * Returns true if kswapd is ready to sleep
4026  */
4027 static bool prepare_kswapd_sleep(pg_data_t *pgdat, int order,
4028                                int highest_zoneidx)
4029 {
4030     /*
4031      * The throttled processes are normally woken up in balance_pgdat() as
4032      * soon as allow_direct_reclaim() is true. But there is a potential
4033      * race between when kswapd checks the watermarks and a process gets
4034      * throttled. There is also a potential race if processes get
4035      * throttled, kswapd wakes, a large process exits thereby balancing
4036      the
4037      * zones, which causes kswapd to exit balance_pgdat() before reaching
4038      * the wake up checks. If kswapd is going to sleep, no process should
4039      * be sleeping on pfmemalloc_wait, so wake them now if necessary. If
4040      * the wake up is premature, processes will wake kswapd and get
4041      * throttled again. The difference from wake ups in balance_pgdat() is
4042      * that here we are under prepare_to_wait().
4043      */
4044     if (waitqueue_active(&pgdat->pfmemalloc_wait))

```

```

4044         wake_up_all(&pgdat->pfnmemalloc_wait);
4045
4046         /* Hopeless node, leave it to direct reclaim */
4047         if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES)
4048             return true;
4049
4050         if (pgdat_balanced(pgdat, order, highest_zoneidx)) {
4051             clear_pgdat_congested(pgdat);
4052             return true;
4053         }
4054
4055         return false;
4056     }

```

Listing 11-30: mm/vmscan.c: prepare_kswapd_sleep()

The `prepare_kswapd_sleep()` function is tasked with determining whether either kswapd has nothing to do, or if it is unable to help with the memory pressure.

It starts by ensuring no direct reclaim is sleeping, waiting for kswapd to improve the situation as, since we are likely about to sleep, we are not going to.

Even if we don't currently sleep, then the worst that this will accomplish is causing direct reclaim to get throttled again and attempt to wake kswapd once more.

See Section 11.6.2 for more on direct reclaim throttling.

After this we check to see whether the nodes count of failed kswapd attempts, `struct pglist_data->kswapd_failures` equals or exceeds the maximum of `MAX_RECLAIM_RETRIES` (hard-coded to 16 attempts).

If so, then kswapd can do nothing to help, so we may as well sleep, leaving the work to direct reclaim.

Otherwise, we determine whether the node is “balanced” via the `pgdat_balanced()` function which we examine in Listing 11-31 in Section 11.4.4.

See the explanation in Section 11.4.4 as to what a balanced node means, but ultimately we take that to indicate that kswapd has no work to do and in that case we ought to sleep, additionally we clear the flag indicating that there is significant congestion in reclaim around this node via `clear_pgdat_congested()`, as it being balanced and thus not requiring reclaim implies that it cannot be congested.

If the node is not balanced, then we indicate that no sleep should occur and thus indirect reclaim should commence.

11.4.4 Node Balancing

The ultimate purpose of indirect reclaim is to balance the node to which it is being performed.

A node being balanced simply means that at least one of the zones that could have been allocated from at the requested order (taking into account

the low memory reserve, see Section 2.4.1 for details of this) equals or exceeds the high watermark.

This is checked in `pgdat_balanced()`, which we examine in Listing 11-31 (eliding out of scope NUMA balancing logic).

```

3972  /*
3973   * Returns true if there is an eligible zone balanced for the request order
3974   * and highest_zoneidx
3975   */
3976  static bool pgdat_balanced(pg_data_t *pgdat, int order, int highest_zoneidx)
3977  {
3978      int i;
3979      unsigned long mark = -1;
3980      struct zone *zone;
3981
3982      /*
3983       * Check watermarks bottom-up as lower zones are more likely to
3984       * meet watermarks.
3985       */
3986      for (i = 0; i <= highest_zoneidx; i++) {
3987          zone = pgdat->node_zones + i;
3988
3989          if (!managed_zone(zone))
3990              continue;
3991          :
3992          mark = high_wmark_pages(zone);
3993          if (zone_watermark_ok_safe(zone, order, mark, highest_zoneidx)
3994              )
3995              return true;
3996      }
3997
3998      /*
3999       * If a node has no managed zone within highest_zoneidx, it does not
4000       * need balancing by definition. This can happen if a zone-restricted
4001       * allocation tries to wake a remote kswapd.
4002       */
4003      if (mark == -1)
4004          return true;
4005
4006      return false;
4007  }
4008
4009  }

```

Listing 11-31: `mm/vmscan.c: pgdat_balanced()`

The `pgdat_balanced()` function iterates through zones in the node, bottom-up, checking that at least one zone has free pages equal to or exceeding the high watermark.

We skip any that don't manage any buddy page allocator controlled pages, as tested by `managed_zone()`, then obtain the high watermark value for the zone via `high_wmark_pages()`.

This is checked against the zone for the specified order via `zone_watermark_ok_safe()`, a function which ultimately invokes `__zone_watermark_ok()` (see Listing 2-59 in Section 2.8.2 and Chapter 2).

If no zone could be checked, then we trivially succeed, otherwise if no zone had sufficient free pages, we indicate failure.

Now we have examined how we determine whether a node is balanced, we can go ahead and examine the next stage in indirect reclaim (as indicated by Figure 11-10), the balancing of a node via `balance_pgdat()`, which we examine starting in Listing 11-32 (eliding out of scope PSI memory pressure hooks, lockdep hooks, cgroup, process freezing, working set and buffer head overrun logic).

```

4133 /*
4134  * For kswapd, balance_pgdat() will reclaim pages across a node from zones
4135  * that are eligible for use by the caller until at least one zone is
4136  * balanced.
4137  *
4138  * Returns the order kswapd finished reclaiming at.
4139  *
4140  * kswapd scans the zones in the highmem->normal->dma direction. It skips
4141  * zones which have free_pages > high_wmark_pages(zone), but once a zone is
4142  * found to have free_pages <= high_wmark_pages(zone), any page in that zone
4143  * or lower is eligible for reclaim until at least one usable zone is
4144  * balanced.
4145  */
4146 static int balance_pgdat(pg_data_t *pgdat, int order, int highest_zoneidx)
4147 {
4148     int i;
4149     unsigned long nr_soft_reclaimed;
4150     :
4151     unsigned long nr_boost_reclaim;
4152     unsigned long zone_boosts[MAX_NR_ZONES] = { 0, };
4153     bool boosted;
4154     struct zone *zone;
4155     struct scan_control sc = {
4156         .gfp_mask = GFP_KERNEL,
4157         .order = order,
4158         .may_unmap = 1,
4159     };
4160     };
4161     set_task_reclaim_state(current, &sc.reclaim_state);
4162     :
4163     count_vm_event(PAGEOUTRUN);
4164     :
4165     /*

```



```

4169      * Account for the reclaim boost. Note that the zone boost is left in
4170      * place so that parallel allocations that are near the watermark will
4171      * stall or direct reclaim until kswapd is finished.
4172      */
4173      nr_boost_reclaim = 0;
4174      for (i = 0; i <= highest_zoneidx; i++) {
4175          zone = pgdat->node_zones + i;
4176          if (!managed_zone(zone))
4177              continue;
4178
4179          nr_boost_reclaim += zone->watermark_boost;
4180          zone_boosts[i] = zone->watermark_boost;
4181      }
4182      boosted = nr_boost_reclaim;

```

Listing 11-32: mm/vmscan.c: balance_pgdat() Zone Boost

We start `balance_pgdat()` by initialising state and setting the struct `task_struct->rs` field to a pointer to the struct `reclaim_state` object newly initialised in the struct `scan_control.reclaim_state` field.

The struct `reclaim_state` field is currently used to track the amount of reclaim performed by slab. Kernel slab allocation is out of scope for the book, but broadly it is the means by which the kernel permits efficient sub-page sized allocations, i.e. a `malloc()` in the kernel.

Objects that are allocated by the slab can also be maintained in caches for efficiency, and those caches can be shrunk during reclaim to free up memory, and that is what is measured here.

Next, we register that indirect reclaim is occurring via the `PAGEOUTRUN` event value.

We then measure zones within the node which have boosted, aggregating the number of boosted pages in `nr_boost_reclaim` and the per-zone boosted in the local `zone_boosts` array.

Zone boosting is performed when migrate type fragmentation occurs, and adds a set factor to each watermark.

We discuss migrate types in Section 2.5 and Chapter 2, however broadly this is an attribute of physical memory pages which determines whether the memory will be moveable or not, such that we can maintain movable memory separate from unmovable (that is, memory used for kernel allocations) so we do not cause fragmentation and thus an inability to free higher order folios by having movable memory interleaved with unmovable.

However, under circumstances where we cannot find memory of the required migrate type we “steal” memory from other migrate type page blocks (again covered in Chapter 2, but broadly page blocks are physically contiguous blocks of memory of order `pageblock_order`, which is order-9 on x86-64).

This is performed in `steal_suitable_fallback()` (see Listing 2-68 and Section 2.8.6 in Chapter 2). If this occurs, we want to ensure that the fragmentation is remedied as soon as possible, and do so by increasing each watermark

by a set number of pages in `boost_watermark()`, up to a maximum of a page block size on each occasion.

When we do so, we proactively wake up `kswapd` for the node on physical page allocation in `rmqueue()` (see Listing 2-60 in Section 2.8.3 and Chapter 2).

We do so in order to prevent further contamination of migrate type page blocks and thus reduce fragmentation as a result.

If the node is balanced and thus reclaim would otherwise not be required, we use the information we gathered throughout this process to determine whether we should perform indirect reclaim to free up memory specifically to avoid further migrate type stealing.

If the node is not balanced, then we ignore zone boosting and reclaim as normal.

With this state established, we proceed with performing reclaim as explored in Listing 11-33.

```

4184 restart:
4185     set_reclaim_active(pgdat, highest_zoneidx);
4186     sc.priority = DEF_PRIORITY;
4187     do {
4188         unsigned long nr_reclaimed = sc.nr_reclaimed;
4189         bool raise_priority = true;
4190         bool balanced;
4191         bool ret;
4192
4193         sc.reclaim_idx = highest_zoneidx;
4194         :
4216         /*
4217         * If the pgdat is imbalanced then ignore boosting and
4218         * preserve
4219         * the watermarks for a later time and restart. Note that the
4220         * zone watermarks will be still reset at the end of balancing
4221         * on the grounds that the normal reclaim should be enough to
4222         * re-evaluate if boosting is required when kswapd next wakes.
4223         */
4224         balanced = pgdat_balanced(pgdat, sc.order, highest_zoneidx);
4225         if (!balanced && nr_boost_reclaim) {
4226             nr_boost_reclaim = 0;
4227             goto restart;
4228         }
4229         /*
4230         * If boosting is not active then only reclaim if there are no
4231         * eligible zones. Note that sc.reclaim_idx is not used as
4232         * buffer_heads_over_limit may have adjusted it.
4233         */
4234         if (!nr_boost_reclaim && balanced)
4235             goto out;
4236

```

```

4237      /* Limit the priority of boosting to avoid reclaim writeback
          */
4238      if (nr_boost_reclaim && sc.priority == DEF_PRIORITY - 2)
4239          raise_priority = false;
4240
4241      /*
4242       * Do not writeback or swap pages for boosted reclaim. The
4243       * intent is to relieve pressure not issue sub-optimal IO
4244       * from reclaim context. If no pages are reclaimed, the
4245       * reclaim will be aborted.
4246      */
4247      sc.may_writepage = !laptop_mode && !nr_boost_reclaim;
4248      sc.may_swap = !nr_boost_reclaim;
4249
4250      /*
4251       * Do some background aging of the anon list, to give
4252       * pages a chance to be referenced before reclaiming. All
4253       * pages are rotated regardless of classzone as this is
4254       * about consistent aging.
4255      */
4256      age_active_anon(pgdat, &sc);
4257
4258      /*
4259       * If we're getting trouble reclaiming, start doing writepage
4260       * even in laptop mode.
4261      */
4262      if (sc.priority < DEF_PRIORITY - 2)
4263          sc.may_writepage = 1;
4264      :
4272      /*
4273       * There should be no need to raise the scanning priority if
4274       * enough pages are already being scanned that that high
4275       * watermark would be met at 100% efficiency.
4276      */
4277      if (kswapd_shrink_node(pgdat, &sc))
4278          raise_priority = false;
4279
4280      /*
4281       * If the low watermark is met there is no need for processes
4282       * to be throttled on pfmemalloc_wait as they should not be
4283       * able to safely make forward progress. Wake them
4284      */
4285      if (waitqueue_active(&pgdat->pfmemalloc_wait) &&
4286          allow_direct_reclaim(pgdat))
4287          wake_up_all(&pgdat->pfmemalloc_wait);
4288
4289      /* Check if kswapd should be suspending */

```

```

:
4293         if (ret || kthread_should_stop())
4294             break;
4295
4296         /*
4297          * Raise priority if scanning rate is too low or there was no
4298          * progress in reclaiming pages
4299          */
4300         nr_reclaimed = sc.nr_reclaimed - nr_reclaimed;
4301         nr_boost_reclaim -= min(nr_boost_reclaim, nr_reclaimed);
4302
4303         /*
4304          * If reclaim made no progress for a boost, stop reclaim as
4305          * IO cannot be queued and it could be an infinite loop in
4306          * extreme circumstances.
4307          */
4308         if (nr_boost_reclaim && !nr_reclaimed)
4309             break;
4310
4311         if (raise_priority || !nr_reclaimed)
4312             sc.priority--;
4313     } while (sc.priority >= 1);

```

Listing 11-33: mm/vmscan.c: balance_pgdat() Core Reclaim

We start by marking all zones up to and including the highest from which we can allocate with the `ZONE_RECLAIM_ACTIVE` flag via `set_reclaim_active()`.

This records for each zone from which we might allocate that indirect reclaim is taking place so that we can track this in other parts of the kernel.

We set the struct `scan_control->priority` field to `DEF_PRIORITY`. This indicates how much of the LRU vectors we should examine when performing reclaim (expressed as a shift value, so `DEF_PRIORITY`, hard-coded to 12, implies we should examine `lru_vec_size >> 12` folios).

We then enter the core reclaim loop. We repeatedly attempt reclaim on the node, gradually increasing priority if either no pages were reclaimed or if `raise_priority` is cleared (this is reset back to true on each loop, and only cleared if reclaim clears pages sufficient that we should soon exit the loop or if zone boosting reclaim is persisting through too many priority levels. We will examine both cases as we come to them.

We start by assessing whether the node is balanced via `pgdat_balanced()` (see Listing 11-31).

As mentioned in the discussion beneath Listing 11-32, we treat zone boosting in effect as a separate indirect reclaim mode—we only perform zone boost-specific logic if the node is already balanced, resetting `nr_boost_reclaim` to zero and restarting the loop if so.

On the other hand, if `pgdat_balanced()` indicates that the node is balanced and there is no zone boost to consider, then we have achieved our aim and exit the loop.

The remainder of the logic in the loop applies whether either the node isn't balanced, or if the zone boost case is being considered.

In the case of us being in zone boost indirect reclaim mode, we ensure we do not proceed below a priority of `DEF_PRIORITY - 2` (keeping in mind that lower priority values implies we consider more folios), so in this instance we disable the raising of priorities (i.e. the decrementing of `struct scan_control->priority` on each loop).

We determine whether we ought to permit the writeback of dirty pages on reclaim, as set in `struct scan_control->may_writepage`, on us both not being in laptop mode (that explicitly avoids spinning up rotational media by avoiding writeback on reclaim) and us not being in zone boosting mode where such actions would only delay us quickly freeing memory in order to avoid page block fragmentation.

In a similar vein, we avoid swapping out to disk if we are in zone boost mode by clearing `struct scan_control->may_swap` in this instance.

We then rotate up to `SWAP_CLUSTER_MAX` anonymous pages (hard-coded to 32) via `age_active_anon()`, where we “age” folios on the active list, moving them to the inactive one if `inactive_is_low()` indicates the inactive anonymous list is too small (see Listing 11-42 and Section 11.5.3).

This aging is ultimately performed by `shrink_active_list()` which we examine in detail in Listing 11-56 and Section ??.

NOTE

We take extra care to perform background aging of the active anonymous LRU vector in order that we ensure we maintain sufficient folios in the list such that anonymous folios get a chance to be referenced before being evicted. Additionally, we are generally less inclined to evict anonymous folios than file-backed (consider cachet rim mode), and additionally thus must ensure consistent aging takes place of anonymous folios.

Next, as in direct reclaim (see Section 11.3), if we are having trouble achieving reclaim, as indicated by us reaching the third priority loop or lower, we should override laptop mode and permit the writing back of dirty pages on reclaim (note this doesn't affect zone boost mode as we already explicitly prevented the priority from being raised beyond this level in this case).

With all of this state put in place, we are ready to actually perform the reclaim, which we do via `kswapd_shrink_node()`, which we examine shortly in Listing 11-35.

The return value of this function indicates whether we at least scanned the number of pages requested—if so then the priority does not need to be raised (and we should simply exit on the next check of the node being balanced), otherwise it ought to be.

Of course allocating processes might result in the node becoming unbalanced immediately after the reclaim, however given we had good results by reclaiming at this priority, it is sensible to simply try again rather than traverse more folios than would otherwise be required.

Next, we consider those threads currently undergoing direct reclaim throttling (see Section 11.6.2). These are waiting on the

struct `pglist_data->pfmemalloc_wait` wait queue, so we check to see if there are waiters, and then whether sufficient memory has been freed to provide enough reserve memory for lagging direct reclaim to be woken, as checked by `allow_direct_reclaim()` (see Listing 11-79 in Section 11.6.2).

If this is so, then we wake all of the sleeping threads so they may resume direct reclaim.

At this point, check whether the kernel thread has been terminated in some fashion via `kthread_should_stop()` and act appropriately.

We then calculate the delta between the number of reclaimed pages, as ultimately set by the reclaim mechanism invoked by `kswpd_shrink_node()` (see Listing 11-35) in struct `scan_control->nr_reclaimed` and the last count of reclaimed pages as stored in `nr_reclaimed` at the start of the loop.

In zone boosting mode, we aim to reclaim at least the number of pages equal to the total number of boost pages, therefore we deduct those pages reclaimed from the `nr_boost_reclaim` value used to determine if this mode should continue.

If this reaches zero and the node is otherwise balanced, we'll exit indirect reclaim on the next iteration.

If we are in zone boosting mode and we reclaim zero pages, we simply abort, as at this stage I/O is probably required, but we are disallowing it, so we may end up looping forever otherwise.

Next, as mentioned at the start of the discussion, we decrement the priority if `raise_priority` was not cleared (it is reset to true at the start of each loop) or if no pages were reclaimed, which indicates no progress is being made so upping the priority makes sense in this case.

Note that we loop while struct `scan_control->priority` is equal to or greater than one, meaning we scan at most one half of the available folios.

After this loop is complete, we proceed with some housekeeping tasks, as explored in Listing 11-34.

```

4315         if (!sc.nr_reclaimed)
4316             pgdat->kswpd_failures++;
4317
4318 out:
4319     clear_reclaim_active(pgdat, highest_zoneidx);
4320
4321     /* If reclaim was boosted, account for the reclaim done in this pass
4322        */
4323     if (boosted) {
4324         unsigned long flags;
4325
4326         for (i = 0; i <= highest_zoneidx; i++) {
4327             if (!zone_boosts[i])
4328                 continue;
4329
4330             /* Increments are under the zone lock */
4331             zone = pgdat->node_zones + i;
4332             spin_lock_irqsave(&zone->lock, flags);

```

```

4332             zone->watermark_boost -= min(zone->watermark_boost,
zone_boosts[i]);
4333             spin_unlock_irqrestore(&zone->lock, flags);
4334         }
4335
4336         /*
4337          * As there is now likely space, wakeup kcompact to defragment
4338          * pageblocks.
4339          */
4340         wakeup_kcompactd(pgdat, pageblock_order, highest_zoneidx);
4341     }
4342     :
4343     snapshot_refaults(NULL, pgdat);
4344     :
4345     set_task_reclaim_state(current, NULL);
4346
4347     /*
4348     * Return the order kswapd stopped reclaiming at as
4349     * prepare_kswapd_sleep() takes it into account. If another caller
4350     * entered the allocator slow path while kswapd was awake, order will
4351     * remain at the higher level.
4352     */
4353     return sc.order;
4354 }
4355

```

Listing 11-34: mm/vmscan.c: balance_pgdat() Post-Reclaim Housekeeping

After the main reclaim loop is complete, we perform some final housekeeping tasks as shown in Listing 11-34.

We start by determining whether any pages were reclaimed as stored in struct scan_control->nr_reclaimed. If none were, then no forward progress has been made and the kswapd invocation was a waste and failed—we therefore increment the struct pglist_data->kswapd_failures counter.

This helps us catch helpless cases where indirect reclaim is simply doing nothing to help the situation.

After we check this we clear the ZONE_RECLAIM_ACTIVE flag from all zones up to and including the highest from which we can allocate via clear_reclaim_active().

This indicates that indirect reclaim is no longer active in these zones.

Next, if a zone boosting pass took place, we clear the watermark boost from each zone that possessed one up to a maximum of its initial boost level (but ensuring we do not reduce it below zero).

This is whether or not we were able to make progress, so the boosting only lasts as long as an indirect reclaim pass.

Equally, we take the time to wake the kcompactd compacting kernel thread to defragment memory through compaction, as this is more likely to now yield results as an effort at this point. Discussion of this is out of scope for the book.

After considering the boosted case, we invoke `snapshot_refaults()` (see Listing 11-38) which stores the current statistics for “refaults” for this node. See Section 11.5.2 and the discussion around Listing 11-41 for an explanation as to how this is used.

After this, we clear the `struct task_struct` reclaim state field and return the order at which `kswapd` stopped reclaim in order that `kswapd()` (see Listing 11-26) can take it into account.

The reclaim is ultimately performed by `kswapd_shrink_node()`, which we explore in Listing 11-35.

```

4058 /*
4059  * kswapd shrinks a node of pages that are at or below the highest usable
4060  * zone that is currently unbalanced.
4061  *
4062  * Returns true if kswapd scanned at least the requested number of pages to
4063  * reclaim or if the lack of progress was due to pages under writeback.
4064  * This is used to determine if the scanning priority needs to be raised.
4065  */
4066 static bool kswapd_shrink_node(pg_data_t *pgdat,
4067                               struct scan_control *sc)
4068 {
4069     struct zone *zone;
4070     int z;
4071
4072     /* Reclaim a number of pages proportional to the number of zones */
4073     sc->nr_to_reclaim = 0;
4074     for (z = 0; z <= sc->reclaim_idx; z++) {
4075         zone = pgdat->node_zones + z;
4076         if (!managed_zone(zone))
4077             continue;
4078
4079         sc->nr_to_reclaim += max(high_wmark_pages(zone),
4080                                SWAP_CLUSTER_MAX);
4081     }
4082     /*
4083      * Historically care was taken to put equal pressure on all zones but
4084      * now pressure is applied based on node LRU order.
4085      */
4086     shrink_node(pgdat, sc);
4087
4088     /*
4089      * Fragmentation may mean that the system cannot be rebalanced for
4090      * high-order allocations. If twice the allocation size has been
4091      * reclaimed then recheck watermarks only at order-0 to prevent
4092      * excessive reclaim. Assume that a process requested a high-order
4093      * can direct reclaim/compact.
4094      */

```



```

4095         if (sc->order && sc->nr_reclaimed >= compact_gap(sc->order))
4096             sc->order = 0;
4097
4098         return sc->nr_scanned >= sc->nr_to_reclaim;
4099     }

```

Listing 11-35: mm/vmscan.c: *kswapd_shrink_node()*

The *kswapd_shrink_node()* function starts with the important task of determining how many pages should be attempted to be reclaimed in aggregate.

It conservatively opts to reclaim pages equal to the high water mark for each zone from which we might be able to allocate, as determined by `struct scan_control->reclaim_idx`.

In each instance, we use `SWAP_CLUSTER_MAX` as a low bar on the number of pages to attempt to reclaim, in keeping with this acting as the minimum granularity for actions taken within reclaim as a whole.

Finally, we perform the actual reclaim via the entrypoint to the core reclaim mechanism, *shrink_node()*, which we explore in Listing 11-40 in Section 11.5.

After this has been performed, we are careful about continuing to attempt reclaim at a higher order, which is more likely to fail than attempts to reclaim at lower order.

We therefore estimate what compaction might be able free via the *compact_gap()* function—this is equal to double the number of pages in the folio order—if we have reclaimed at least this number of pages then we consider it safe to reset to order zero reclaim.

Finally we return a boolean value indicating whether we scanned at least the number of pages we were requested to reclaim in order to determine whether *balance_pgdat()* ought to increase priority (as shown in Listing 11-33).

11.5 The Reclaim Mechanism

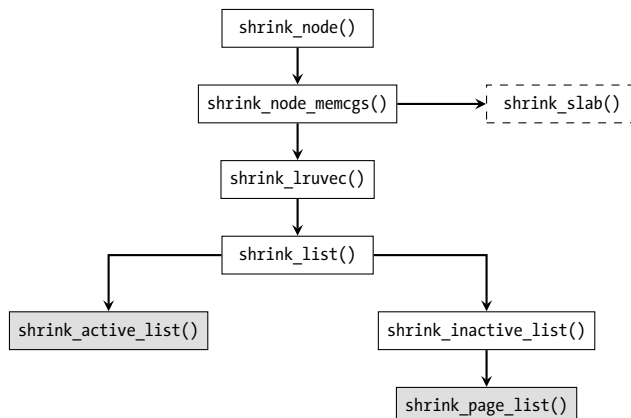


Figure 11-11: Core Reclaim Mechanism Code Path

NOTE

Some of the functions referenced by Figure 11-11 may be invoked by other code paths, however the one shown is the typical means by which reclaim operates.

While the logic surrounding direct reclaim (see Section 11.3) and indirect reclaim (see Section 11.4) determine how reclaim is invoked, how many times and at what priority, the “reclaim mechanism” is the core machinery that actually performs this reclaim.

As with much kernel functionality, it is abstracted into gradually more specific functions. At each stage the reclaim operation is framed as “shrinking” memory, at ever finer granularity.

We start by shrinking an entire node via `shrink_node()` (see Listing 11-40). We examine this in Section 11.5.3.

Next, we shrink the LRU vectors contained within the node (see Section 11.2 for more on LRU vectors), via `shrink_lruvec()` (see Listing 11-51). We examine this in Section 11.5.5.

We shrink each individual LRU vector via `shrink_list()` (see Listing 11-55). We examine this in Section 11.5.6.

When we do so, we may be shrinking the active list (i.e. deactivating folios) via `shrink_active_list()` (see Listing 11-56) or we may be shrinking the inactive list via `shrink_inactive_list()` (see Listing 11-61), which ultimately invokes `shrink_page_list()` (see Listing 11-63), which performs actual folio reclaim.

We explore shrinking of the active LRU vectors (deactivating folios) in Section 11.5.7, and shrinking of inactive LRU vectors in Section 11.5.9, which performs the actual reclaim.

We start by examining the `struct scan_control` (see Listing 11-36) object which is threaded through the reclaim code) in Section 11.5.1.

11.5.1 The Scan Control Object

As mentioned in Sections 11.4 and 11.3, the reclaim mechanism is configured by the `struct scan_control` object.

This data structure has a great many flags within it (defined as bit fields), so for clarity we separate its data and flag fields, starting by examining its data fields in Listing 11-36 (eliding out of scope cgroup fields).

```

66 struct scan_control {
67     /* How many pages shrink_list() should reclaim */
68     unsigned long nr_to_reclaim;
69
70     /*
71      * Nodemask of nodes allowed by the caller. If NULL, all nodes
72      * are scanned.
73      */
74     nodemask_t      *nodemask;
75     :
76     /*
77      * Scan pressure balancing between anon and file LRUs
78      */
79     :
80     :
81     :
82     :
83     :

```

```

84      */
85      unsigned long   anon_cost;
86      unsigned long   file_cost;
87      :
132      /* Allocation order */
133      s8 order;
134
135      /* Scan (total_size >> priority) pages at once */
136      s8 priority;
137
138      /* The highest zone to isolate pages for reclaim from */
139      s8 reclaim_idx;
140
141      /* This context's GFP mask */
142      gfp_t gfp_mask;
143
144      /* Incremented by the number of inactive pages that were scanned */
145      unsigned long nr_scanned;
146
147      /* Number of pages freed so far during a call to shrink_zones() */
148      unsigned long nr_reclaimed;
149
150      struct {
151          unsigned int dirty;
152          unsigned int unqueued_dirty;
153          unsigned int congested;
154          unsigned int writeback;
155          unsigned int immediate;
156          unsigned int file_taken;
157          unsigned int taken;
158      } nr;
159
160      /* for recording the reclaimed slab by now */
161      struct reclaim_state reclaim_state;
162  };

```

Listing 11-36: mm/vmscan.c: struct scan_control Data Fields

Examining each of the fields shown in Listing 11-36:

nr_to_reclaim User-defined—Specifies the number of base pages to attempt to reclaim.

nodemask User-defined—Specifies the nodes which we can consider for reclaim.

anon_cost Set in `shrink_node()` (see Listing 11-40)—Indicates the relative cost of reclaiming anonymous folios as opposed to file-backed folios, as determined by working set logic which examines when folios which have previously been reclaimed are rapidly “refaulted”. A detailed discus-

sion of this is out of scope for the book, but both anonymous and file-backed folio costs are maintained in struct `lruvec` fields, calculated in `lru_note_cost()` and `lru_note_cost_folio()` which calls it, which is triggered by the working set logic in `workingset_refault()`. `shrink_node()` simply sets both `anon_cost` and `file_cost` fields based on the struct `lruvec` values. See Section 11.5.2 for more on this.

- file_cost** Set in `shrink_node()` (see Listing 11-40)—Indicates the relative cost of reclaiming file-backed folios as opposed to anonymous folios. See `anon_cost` entry.
- order** User-defined—Specifies the order of the allocation which caused reclaim to trigger.
- priority** User-defined—Specifies the number of pages we ought to consider in this reclaim pass. Expressed in the form of a right shift value, typically starting at `DEF_PRIORITY` (hard-coded to 12). A shift right of 12 bits for instance implies that we consider 1/4096th of the contents of LRU vectors (see Section 11.2 for more on these).
- reclaim_idx** User-defined—Specifies the index of the maximum zone from which the failing allocation can be performed.
- gfp_mask** User-defined—Specifies the Get Free Pages (GFP) mask of the failing allocation and thus the restrictions placed on the allocation.
- nr_scanned** Set by reclaim—A count of the number of base pages scanned (i.e. examined) by reclaim.
- nr_reclaimed** Set by reclaim—A count of the number of base pages reclaimed and thus freed by reclaim.
- nr.dirty** Set by reclaim—Aggregate of struct `reclaim_state->nr_dirty` counts—The number of scanned folios (expressed in base pages) which are either dirty (have the `PG_dirty` flag set) or undergoing writeback (have the `PG_writeback` flag set).
- nr.unqueued_dirty** Set by reclaim—Aggregate of struct `reclaim_state->nr_unqueued_dirty` counts—The number of scanned folios (expressed in base pages) which are dirty (have the `PG_dirty` flag set) and have not yet been subject to writeback (does not have the `PG_writeback` flag set).
- nr.congested** Set by reclaim—Aggregate of struct `reclaim_state->nr_congested` counts—The number of scanned folios (expressed in base pages) which have both the `PG_writeback` and `PG_reclaim` flags set, indicating this number of pages have cycled through reclaim at least one more time without having completed writeback, indicating writeback congestion of reclaimed folios.
- nr.writeback** Set by reclaim—Aggregate of struct `reclaim_state->nr_writeback` counts—The number of scanned folios (expressed in base pages) which are undergoing writeback (have the `PG_writeback` flag set) at the time of reclaim (and are not marked as immediate, as described in `nr.immediate`).

- nr.immediate** Set by reclaim—Aggregate of struct `reclaim_state->nr_immediate` counts—The number of file-backed folios (expressed in base pages) which are undergoing writeback (have the `PG_writeback` flag set), are being scanned by `kswapd` but are also marked with the `PG_reclaim` flag, indicating they are cycling through reclaim, after which we had determined that excess writeback was happening, as indicated by the node having the `PGDAT_WRITEBACK` flag set (see the discussion around the portion of `shrink_node()` examined in Listing 11-44 for details).
- nr.file_taken** Set by reclaim—The number of file-backed folios (expressed in base pages) which have been isolated from LRUs for reclaim (isolation is performed by `isolate_lru_pages()` which we examine in Listing 11-57).
- nr.taken** Set by reclaim—The number of folios in total (expressed in base pages) which have been isolated from LRUs for reclaim (isolation is performed by `isolate_lru_pages()` which we examine in Listing 11-57).
- reclaim_state** Set by reclaim—Slab-specific state updated after the slab allocator is able to free memory.

We examine the flag-specific fields of struct `scan_control` in Listing 11-37 (eliding out of scope `cgroup`, power management and NUMA demotion flags).

```

66 struct scan_control {
67     :
68     /* Can active pages be deactivated as part of reclaim? */
69 #define DEACTIVATE_ANON 1
70 #define DEACTIVATE_FILE 2
71     unsigned int may_deactivate:2;
72     unsigned int force_deactivate:1;
73     unsigned int skipped_deactivate:1;
74
75     /* Writepage batching in laptop mode; RECLAIM_WRITE */
76     unsigned int may_writepage:1;
77
78     /* Can mapped pages be reclaimed? */
79     unsigned int may_unmap:1;
80
81     /* Can pages be swapped as part of reclaim? */
82     unsigned int may_swap:1;
83     :
84
85     /* One of the zones is ready for compaction */
86     unsigned int compaction_ready:1;
87
88     /* There is easily reclaimable cold cache in the current node */
89     unsigned int cache_trim_mode:1;
90
91     /* The file pages on the current node are dangerously low */
92     unsigned int file_is_tiny:1;

```

```

:
162 };

```

Listing 11-37: mm/vmscan.c: struct scan_control Flag Fields

Each of these use the C bitfield syntax, where the number suffixing the comma delimiter defines the number of bits each field occupies.

Examining each of the fields shown in Listing 11-37:

- may_deactivate** Set in `shrink_node()` (see Listing 11-40)—Determines whether reclaim can be permitted to interact with the active LRU vectors (see Section 11.2 for more on active and inactive LRU vectors), i.e. whether `shrink_list()` (see Listing 11-55) can invoke `shrink_active_list()` (see Listing 11-56). There are separate flags for anonymous and file-backed folios, defined as `DEACTIVATE_ANON` and `DEACTIVATE_FILE`.
- force_deactivate** Set in `do_try_to_free_pages()` (see Listing 11-20) as part of direct reclaim—Forces deactivation of both file-backed and anonymous folios. Set after an instance where deactivation is skipped to ensure these modes of reclaim are performed.
- skipped_deactivate** Set in `shrink_node()` (see Listing 11-40)—Indicates that `force_deactivate` was set so `shrink_list()` (see Listing 11-55) did not invoke `shrink_active_list()` (see Listing 11-56).
- may_writepage** User-defined—Specifies whether dirty folios can be written back to disk as part of reclaim Checked in the core `shrink_page_list()` function.
- may_swap** User-defined—Specifies whether we can swap out folios to disk as part of the reclaim process.
- compaction_ready** Used by direct reclaim—Indicates that compaction is preferred over reclaim. Set in `shrink_zones()` (see Listing 11-21), having been determined to be applicable by `compaction_ready()` and checked for in `do_try_to_free_pages()` (see Listing 11-20) which aborts the reclaim attempt deferring to compaction if this is set.
- cache_trim_mode** Set in `shrink_node()` (see Listing 11-40)—Indicates that only file-backed folios should be considered for reclaim in an instance where both deactivation of file-backed folios is prohibited and a large number of inactive file-backed folios are present in the LRU vector.
- file_is_tiny** Set in `shrink_node()` (see Listing 11-40)—Indicates that a situation exists where there is a large number of anonymous folios subject to reclaim, deactivation of anonymous folios is prohibited and, even if all file-backed folios were freed, this would not fulfill indirect reclaim, that is, the sum of base pages for file-backed folios subject to reclaim and free file-backed folios is less than the aggregated high watermark across zones in a node. In this instance, we limit scanning to anonymous folios only.

The reclaim process which `struct scan_control` governs and which is invoked by both direct and indirect reclaim begins in `shrink_node()` (see Listing 11-40) and proceeds through the code path as shown in Figure 11-11.

11.5.2 A Brief Overview of the Working Set

While a more detailed description of the working set functionality within the kernel is out of scope for the book, the data it provides forms a key part of the decision making in reclaim, so it is important to provide an overview of what it is and how the data it provides is determined.

The working set of a process is the range of memory that is required for the process to function at any given point in time, i.e. the folios that the process is using.

We can expand this definition to the system as the memory in use by all processes at a given point in time, and from the point of view of reclaim the working set matters in so far as—are we “thrashing” folios within the working set?

Thrashing in this context means we are repeatedly using folios but they are being repeatedly reclaimed, i.e. faulting again (termed “refaulting”) shortly after reclaim.

The “refault distance”, equal to the number of folios scanned in the inactive list between a folio being evicted and refaulted, is really critical here. We examine this concept in Figure 11-12

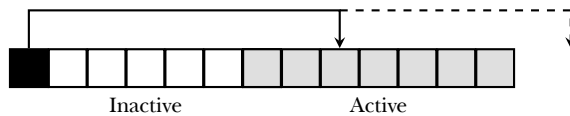


Figure 11-12: Refault Distance Example

In Figure 11-12, we consider the reclaim of the left-most folio (which sits at the tail of the inactive list).

If the refault distance is less than the combined number of folios in the inactive and active lists, then had the folio been activated, it would never have been reclaimed and thus never refaulted. This is illustrated by the solid arrow.

However, if the refault distance is greater than the sum of both inactive and active list, which represents the entire reclaim cache, then nothing could have been done to prevent the thrashing—the user is attempting to use more memory than exists on the system (illustrated by the dashed line).

This is where this information becomes critically useful—if we are thrashing in the former case, the working set logic activates the refaulting folio, so it will get referenced in time for it to avoid being reclaimed.

However this being the case implies that perhaps other folios on the active list are being kept there longer than necessary, and in any case we should demote folios from the active list to make way for the newly promoted one.

If we wrongly demote a folio that goes on to refault, then this mechanism will correct the situation, so this is a safe thing to do.

Overall we can use this information to inform the balance between demoting from the active list and reclaiming from the inactive list—something we examine in the discussion around Listing 11-41.

Examining how this is done in practice, we note that in both direct reclaim in `do_try_to_free_pages()` (see Listing 11-20) and indirect reclaim in `balance_pgdat()` (see Listing 11-34), the `snapshot_refaults()` function is called after the reclaim is complete.

This takes a snapshot of the number of both refaulting anonymous folios and refaulting file-backed folios that have been activated. We can then use this snapshot data to determine whether we have refaulted more folios since the last time we checked and therefore shrink the active list in this case.

The snapshot is performed by `snapshot_refaults()` which we examine in Listing 11-38.

```

3555 static void snapshot_refaults(struct mem_cgroup *target_memcg, pg_data_t *
      pgdat)
3556 {
3557     struct lruvec *target_lruvec;
3558     unsigned long refaults;
3559
3560     target_lruvec = mem_cgroup_lruvec(target_memcg, pgdat);
3561     refaults = lruvec_page_state(target_lruvec, WORKINGSET_ACTIVATE_ANON);
3562     target_lruvec->refaults[0] = refaults;
3563     refaults = lruvec_page_state(target_lruvec, WORKINGSET_ACTIVATE_FILE);
3564     target_lruvec->refaults[1] = refaults;
3565 }
```

Listing 11-38: mm/vmscan.c: snapshot_refaults()

The `snapshot_refaults()` function updates the `struct lruvec` (see Section 11.2 for more on this type) which is associated with the specified node (except under `cgroup`, the discussion of which is out of scope for the book).

This places the current anonymous folio refault and activate count (classified under `WORKINGSET_ACTIVATE_ANON`) in `struct lruvec->refaults[0]`, and the current file-backed folio refault and activate count (classified under `WORKINGSET_ACTIVATE_FILE`) in `struct lruvec->refaults[1]`.

11.5.2.1 Anonymous and File-Backed Refault Costing

It's important to note that refaults can occur both when working sets change, and when known active folios within the working set end up being starved out of the active list and need to be reinstated.

In order to differentiate between the two, the `PG_workingset` flag is set by `shrink_active_list()` (see Listing 11-56) and maintained throughout its lifetime to indicate that it was at least at one time part of the active LRU.

The `workingset_refault()` function, which is invoked when a refault occurs, notes the presence of this flag and uses it to calculate a relative cost

between anonymous and file-backed folios, stored in struct `lruvec->anon_cost` and struct `lruvec->file_cost`.

If we are experiencing a large amount of refaulting, that is thrashing of anonymous folios compared to file-backed ones then we ought to prefer file-backed, and vice-versa.

This is calculated in `get_scan_count()` which we explore in Listing 11-47 and Section 11.5.4.

Therefore, on each occasion a folio which has previously been active is refaulted, we account for its costs ultimately using `lru_note_cost()`, which we explore in Listing 11-39.

In addition, if a dirty folio (whether file-backed or an anonymous folio being swapped out) is successfully paged out to disk, we assume that this implies it was part of the working set and account accordingly.

```

298 void lru_note_cost(struct lruvec *lruvec, bool file, unsigned int nr_pages)
299 {
300     do {
301         unsigned long lrusize;
302
303         /*
304          * Hold lruvec->lru_lock is safe here, since
305          * 1) The pinned lruvec in reclaim, or
306          * 2) From a pre-LRU page during refault (which also holds the
307          *    rcu lock, so would be safe even if the page was on the
          LRU
308          *    and could move simultaneously to a new lruvec).
309          */
310         spin_lock_irq(&lruvec->lru_lock);
311         /* Record cost event */
312         if (file)
313             lruvec->file_cost += nr_pages;
314         else
315             lruvec->anon_cost += nr_pages;
316
317         /*
318          * Decay previous events
319          *
320          * Because workloads change over time (and to avoid
321          * overflow) we keep these statistics as a floating
322          * average, which ends up weighing recent refaults
323          * more than old ones.
324          */
325         lrusize = lruvec_page_state(lruvec, NR_INACTIVE_ANON) +
326                 lruvec_page_state(lruvec, NR_ACTIVE_ANON) +
327                 lruvec_page_state(lruvec, NR_INACTIVE_FILE) +
328                 lruvec_page_state(lruvec, NR_ACTIVE_FILE);
329
330         if (lruvec->file_cost + lruvec->anon_cost > lrusize / 4) {

```

```

331             lruvec->file_cost /= 2;
332             lruvec->anon_cost /= 2;
333         }
334         spin_unlock_irq(&lruvec->lru_lock);
335     } while ((lruvec = parent_lruvec(lruvec)));
336 }

```

Listing 11-39: mm/swap.c: `lru_note_cost()`

Since we do not consider the cgroup case in the book, we will not examine this outer loop in `lru_note_cost()`.

We start by simply accumulating the file and anonymous costs expressed in base pages passed to this function. We then decay the values by half if they, in aggregate, exceed one quarter of the size of the total number of base LRU pages.

This has the effect of both ensuring we do not overflow and making more recent refaults more meaningful than older ones.

This is later set to the struct `scan_control->anon_cost` and struct `scan_control->file_cost` fields in `shrink_node()` as shown in Listing 11-40 and Section 11.5.3.

11.5.3 Shrinking the Node

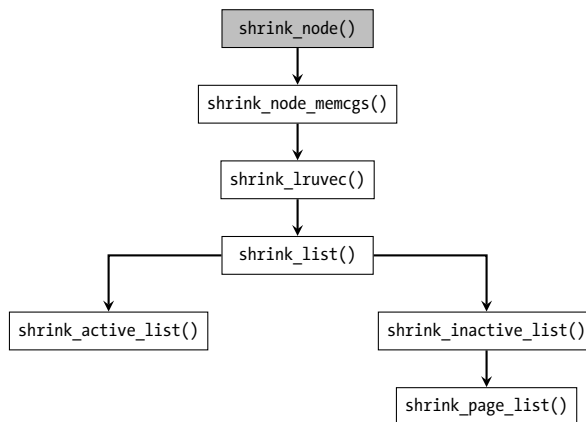
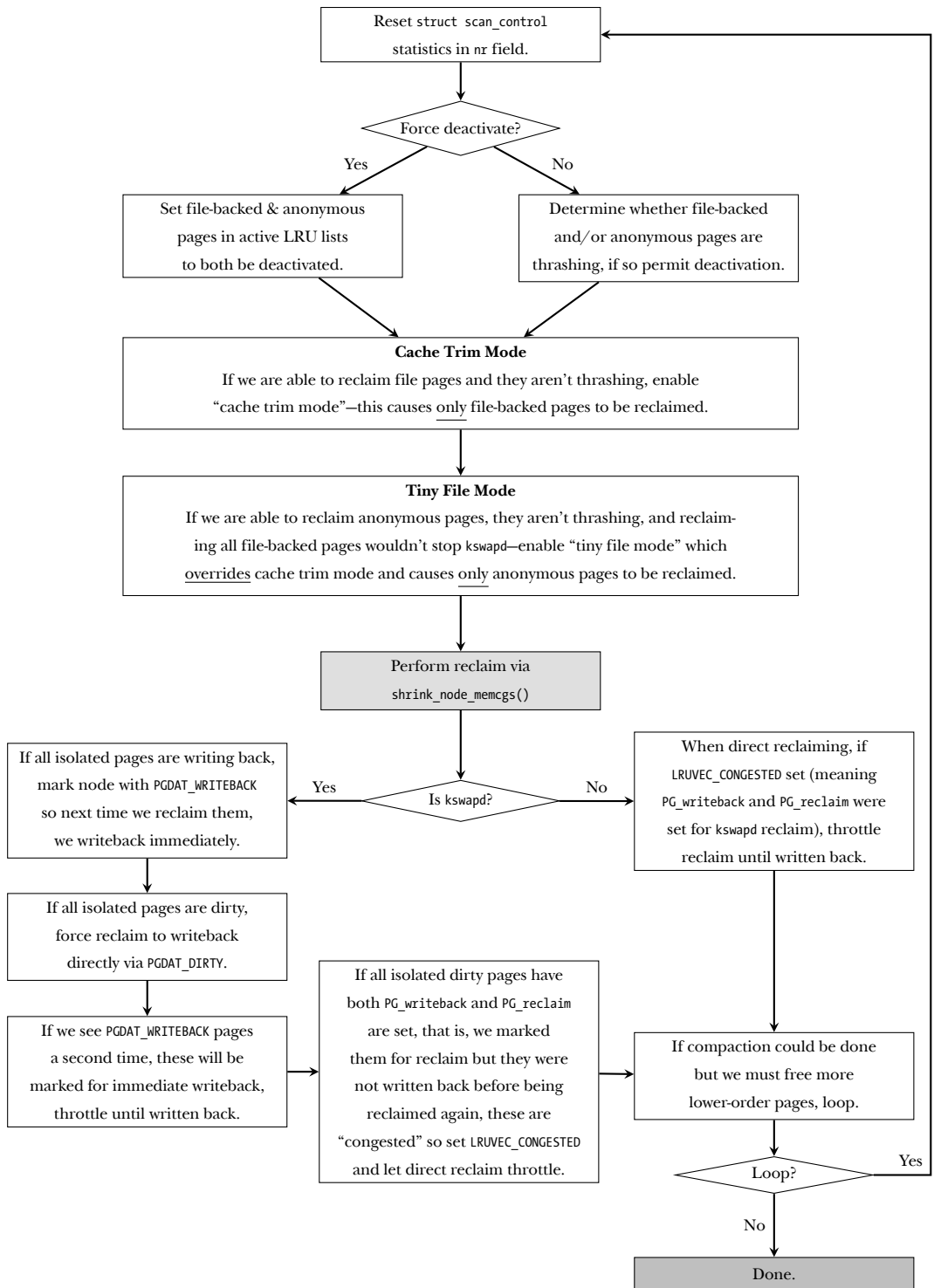


Figure 11-13: Reclaim Mechanism: Shrinking the Node Code Path

The core reclaim logic, shared between all invocations of reclaim begins in `shrink_node()` which we examine starting in Listing 11-40 (eliding any cgroup logic that is entirely out of scope, and power management handling which certainly is).

We examine the logic of this function in Figure 11-14.

Figure 11-14: `shrink_node()` Logic

We start to examine `shrink_node()` in Listing 11-40.

```

3194 static void shrink_node(pg_data_t *pgdat, struct scan_control *sc)
3195 {
3196     struct reclaim_state *reclaim_state = current->reclaim_state;
3197     unsigned long nr_reclaimed, nr_scanned;
3198     struct lruvec *target_lruvec;
3199     bool reclaimable = false;
3200     unsigned long file;
3201
3202     target_lruvec = mem_cgroup_lruvec(sc->target_mem_cgroup, pgdat);
3203
3204     again:
3205     :
3211     memset(&sc->nr, 0, sizeof(sc->nr));
3212
3213     nr_reclaimed = sc->nr_reclaimed;
3214     nr_scanned = sc->nr_scanned;
3215
3216     /*
3217      * Determine the scan balance between anon and file LRUs.
3218      */
3219     spin_lock_irq(&target_lruvec->lru_lock);
3220     sc->anon_cost = target_lruvec->anon_cost;
3221     sc->file_cost = target_lruvec->file_cost;
3222     spin_unlock_irq(&target_lruvec->lru_lock);

```

Listing 11-40: `mm/vmscan.c: shrink_node()` Preface

We start `shrink_node()` by establishing some local state, including determining the `struct lruvec` which pertains to this cgroup or node.

Discussion of cgroups is out of the scope of the book, however this is an instance where the function, while referencing cgroup state, performs double-duty and in the case where cgroups are disabled, the `struct lruvec` (see Section 11.2 for more on this) is associated with the node and the `mem_cgroup_lruvec()` function handles both cases.

We zero the `struct scan_control->nr` statistics object, then keep local copies of the numbers of base pages scanned and reclaimed so far.

Finally, we retrieve the anonymous and file costs associated with anonymous and file folio reclaim (see Section 11.5.2 for details of how these are calculated). We will examine how these values are used in the functions which reference them.

Once this initialisation is complete, we proceed with determining whether deactivation of anonymous and file-backed folios ought to proceed, which we explore in Listing 11-41.

```

3224     /*
3225      * Target desirable inactive:active list ratios for the anon
3226      * and file LRU lists.

```

```

3227      */
3228      if (!sc->force_deactivate) {
3229          unsigned long refaults;
3230
3231          refaults = lruvec_page_state(target_lruvec,
3232                                     WORKINGSET_ACTIVATE_ANON);
3233          if (refaults != target_lruvec->refaults[0] ||
3234              inactive_is_low(target_lruvec, LRU_INACTIVE_ANON))
3235              sc->may_deactivate |= DEACTIVATE_ANON;
3236          else
3237              sc->may_deactivate &= ~DEACTIVATE_ANON;
3238
3239          /*
3240           * When refaults are being observed, it means a new
3241           * workingset is being established. Deactivate to get
3242           * rid of any stale active pages quickly.
3243           */
3244          refaults = lruvec_page_state(target_lruvec,
3245                                     WORKINGSET_ACTIVATE_FILE);
3246          if (refaults != target_lruvec->refaults[1] ||
3247              inactive_is_low(target_lruvec, LRU_INACTIVE_FILE))
3248              sc->may_deactivate |= DEACTIVATE_FILE;
3249          else
3250              sc->may_deactivate &= ~DEACTIVATE_FILE;
3251      } else
3252          sc->may_deactivate = DEACTIVATE_ANON | DEACTIVATE_FILE;

```

Listing 11-41: mm/vmscan.c: shrink_node() Active/Inactive Ratio Logic

The active/inactive ratio logic portion of `shrink_node()` (explored in Listing 11-41) starts by simply checking whether `struct scan_control->force_deactivate` has been set—this is set in `do_try_to_free_pages()` as part of direct reclaim if previously deactivation had been skipped.

Otherwise, we determine whether further refaulting has occurred by checking whether the current number of activated refaulted anonymous and file-backed folios differs from the last snapshotted values. See Section 11.5.2 for a discussion of this logic.

If so, we permit deactivation of folios from the anonymous and/or file-backed LRU lists accordingly, by setting `DEACTIVATE_ANON` and `DEACTIVATE_FILE` flags in `struct scan_control->may_deactivate`.

We also do so if the inactive LRU list is considered to be too small, as checked by `inactive_is_low()`, which we examine in Listing 11-42.

```

2677  /*
2678   * The inactive anon list should be small enough that the VM never has
2679   * to do too much work.
2680   *
2681   * The inactive file list should be small enough to leave most memory

```

```

2682 * to the established workingset on the scan-resistant active list,
2683 * but large enough to avoid thrashing the aggregate readahead window.
2684 *
2685 * Both inactive lists should also be large enough that each inactive
2686 * page has a chance to be referenced again before it is reclaimed.
2687 *
2688 * If that fails and refaulting is observed, the inactive list grows.
2689 *
2690 * The inactive_ratio is the target ratio of ACTIVE to INACTIVE pages
2691 * on this LRU, maintained by the pageout code. An inactive_ratio
2692 * of 3 means 3:1 or 25% of the pages are kept on the inactive list.
2693 *
2694 * total      target      max
2695 * memory     ratio      inactive
2696 * -----
2697 * 10MB        1          5MB
2698 * 100MB       1          50MB
2699 * 1GB         3          250MB
2700 * 10GB        10         0.9GB
2701 * 100GB       31         3GB
2702 * 1TB         101        10GB
2703 * 10TB        320        32GB
2704 */
2705 static bool inactive_is_low(struct lruvec *lruvec, enum lru_list inactive_lru)
2706 {
2707     enum lru_list active_lru = inactive_lru + LRU_ACTIVE;
2708     unsigned long inactive, active;
2709     unsigned long inactive_ratio;
2710     unsigned long gb;
2711
2712     inactive = lruvec_page_state(lruvec, NR_LRU_BASE + inactive_lru);
2713     active = lruvec_page_state(lruvec, NR_LRU_BASE + active_lru);
2714
2715     gb = (inactive + active) >> (30 - PAGE_SHIFT);
2716     if (gb)
2717         inactive_ratio = int_sqrt(10 * gb);
2718     else
2719         inactive_ratio = 1;
2720
2721     return inactive * inactive_ratio < active;
2722 }

```

Listing 11-42: mm/vmscan.c: `inactive_is_low()`

We can see in `inactive_is_low()` that we try to maintain a balance between active and inactive lists which is heavily weighted towards the active list in order that we avoid reclaiming as much as we can, reducing the amount of

work reclaim needs to do and giving folios the maximum chance to avoid reclaim.

The refaulting logic previously described accounts for situations where folios ought to be deactivated so the combination of the two combines a well-tuned heuristic with a mechanism for avoiding thrashing under heavy memory pressure.

With the balance between active and inactive LRU lists accounted for, we proceed with further configuration of the struct `scan_control` object prior to executing reclaim, which we explore in Listing 11-43.

```

3254      /*
3255       * If we have plenty of inactive file pages that aren't
3256       * thrashing, try to reclaim those first before touching
3257       * anonymous pages.
3258       */
3259      file = lruvec_page_state(target_lruvec, NR_INACTIVE_FILE);
3260      if (file >> sc->priority && !(sc->may_deactivate & DEACTIVATE_FILE))
3261          sc->cache_trim_mode = 1;
3262      else
3263          sc->cache_trim_mode = 0;
3264
3265      /*
3266       * Prevent the reclaimer from falling into the cache trap: as
3267       * cache pages start out inactive, every cache fault will tip
3268       * the scan balance towards the file LRU. And as the file LRU
3269       * shrinks, so does the window for rotation from references.
3270       * This means we have a runaway feedback loop where a tiny
3271       * thrashing file LRU becomes infinitely more attractive than
3272       * anon pages. Try to detect this based on file LRU size.
3273       */
3274      :
3275      unsigned long total_high_wmark = 0;
3276      unsigned long free, anon;
3277      int z;
3278
3279      free = sum_zone_node_page_state(pgdat->node_id, NR_FREE_PAGES)
3280      ;
3281      file = node_page_state(pgdat, NR_ACTIVE_FILE) +
3282            node_page_state(pgdat, NR_INACTIVE_FILE);
3283
3284      for (z = 0; z < MAX_NR_ZONES; z++) {
3285          struct zone *zone = &pgdat->node_zones[z];
3286          if (!managed_zone(zone))
3287              continue;
3288
3289          total_high_wmark += high_wmark_pages(zone);
3290      }

```

```

3291         /*
3292         * Consider anon: if that's low too, this isn't a
3293         * runaway file reclaim problem, but rather just
3294         * extreme pressure. Reclaim as per usual then.
3295         */
3296         anon = node_page_state(pgdat, NR_INACTIVE_ANON);
3297
3298         sc->file_is_tiny =
3299             file + free <= total_high_wmark &&
3300             !(sc->may_deactivate & DEACTIVATE_ANON) &&
3301             anon >> sc->priority;

```

Listing 11-43: mm/vmscan.c: shrink_node() Further Reclaim Configuration

Examining the portion of `shrink_node()` which performs additional reclaim configuration as shown in Listing 11-43, we start by determining whether we enable the cache trim mode specified in `struct scan_control->cache_trim_mode`.

In this mode, we prefer to only examine file folios (as determined by `get_scan_count()` which we examine later in Listing 11-47 and Section 11.5.4).

We do so if there are sufficient file folios that they will be scanned at this priority level and there's no indication that they are thrashing (if they were thrashing then `struct scan_control->may_deactivate` will have `DEACTIVATE_FILE` set).

This clearly demonstrates that there is a preference for reclaiming file-backed folios if this is possible, before trying to reclaim anonymous folios, as the latter is likely to be more of an expensive operation than the former.

Having said this, we then must be cautious to avoid a degenerate scenario where we simply never reclaim anonymous folios because we prefer to reclaim file-backed folios, which could cause a small number of file folios to dominate reclaim while ignoring a larger set of anonymous folios.

We detect this by first determining whether the sum of all free base pages and inactive and active file LRU base pages is less than or equal to the sum of zone high water marks—this means freeing file folios alone will not allow `kswapd` to sleep and thus reclaim will continue running potentially indefinitely.

We also make sure that the anonymous LRU is not thrashing (that anonymous folios are not permitted to deactivate) which otherwise would indicate that focusing on the file folios would not be problematic.

Finally we make sure that the number of inactive anonymous folios is sufficiently large that we could perform anonymous folio reclaim at this priority level, if this was not the case then it just indicates that we are under severe memory pressure generally.

If these conditions are met, we set `struct scan_control->file_is_tiny` which overrides `struct scan_control->cache_trim_mode` and causes only anonymous folios to be scanned, as set in `get_scan_count()` (see Listing 11-47 and Section 11.5.4).

With the struct `scan_control` object set up, we are ready to perform the actual reclaim and tasks after reclaim has been executed, we examine this portion of `shrink_node()` in Listing 11-44.

```

3304     shrink_node_memcgs(pgdat, sc);
3305
3306     if (reclaim_state) {
3307         sc->nr_reclaimed += reclaim_state->reclaimed_slab;
3308         reclaim_state->reclaimed_slab = 0;
3309     }
3310     :
3311
3312     if (sc->nr_reclaimed - nr_reclaimed)
3313         reclaimable = true;
3314
3315     if (current_is_kswapd()) {
3316         /*
3317          * If reclaim is isolating dirty pages under writeback,
3318          * it implies that the long-lived page allocation rate
3319          * is exceeding the page laundering rate. Either the
3320          * global limits are not being effective at throttling
3321          * processes due to the page distribution throughout
3322          * zones or there is heavy usage of a slow backing
3323          * device. The only option is to throttle from reclaim
3324          * context which is not ideal as there is no guarantee
3325          * the dirtying process is throttled in the same way
3326          * balance_dirty_pages() manages.
3327          *
3328          * Once a node is flagged PGDAT_WRITEBACK, kswapd will
3329          * count the number of pages under pages flagged for
3330          * immediate reclaim and stall if any are encountered
3331          * in the nr_immediate check below.
3332          */
3333         if (sc->nr.writeback && sc->nr.writeback == sc->nr.taken)
3334             set_bit(PGDAT_WRITEBACK, &pgdat->flags);
3335
3336         /* Allow kswapd to start writing pages during reclaim.*/
3337         if (sc->nr.unqueued_dirty == sc->nr.file_taken)
3338             set_bit(PGDAT_DIRTY, &pgdat->flags);
3339
3340         /*
3341          * If kswapd scans pages marked for immediate
3342          * reclaim and under writeback (nr_immediate), it
3343          * implies that pages are cycling through the LRU
3344          * faster than they are written so forcibly stall
3345          * until some pages complete writeback.
3346          */
3347         if (sc->nr.immediate)
3348             reclaim_throttle(pgdat, VMSCAN_THROTTLE_WRITEBACK);

```

```

3354     }
3355
3356     /*
3357      * Tag a node/memcg as congested if all the dirty pages were marked
3358      * for writeback and immediate reclaim (counted in nr.congested).
3359      :
3360     */
3361     if ((current_is_kswapd() ||
3362          (cgroup_reclaim(sc) && writeback_throttling_sane(sc))) &&
3363         sc->nr.dirty && sc->nr.dirty == sc->nr.congested)
3364         set_bit(LRUVEC_CONGESTED, &target_lruvec->flags);
3365
3366     /*
3367      * Stall direct reclaim for IO completions if the lruvec is
3368      * node is congested. Allow kswapd to continue until it
3369      * starts encountering unqueued dirty pages or cycling through
3370      * the LRU too quickly.
3371     */
3372     if (!current_is_kswapd() && current_may_throttle() &&
3373         :
3374         test_bit(LRUVEC_CONGESTED, &target_lruvec->flags))
3375         reclaim_throttle(pgdat, VMSCAN_THROTTLE_CONGESTED);
3376
3377     if (should_continue_reclaim(pgdat, sc->nr_reclaimed - nr_reclaimed,
3378                                sc))
3379         goto again;
3380
3381     /*
3382      * Kswapd gives up on balancing particular nodes after too
3383      * many failures to reclaim anything from them and goes to
3384      * sleep. On reclaim progress, reset the failure counter. A
3385      * successful direct reclaim run will revive a dormant kswapd.
3386     */
3387     if (reclaimable)
3388         pgdat->kswapd_failures = 0;
3389 }

```

Listing 11-44: mm/vmscan.c: *shrink_node()* Reclaim and Post-processing

We perform the actual reclaim via `shrink_node_memcgs()` which we examine in Listing 11-50 and Section 11.5.5.

After this is done, we first check to see whether we need to update the `struct scan_control->reclaim_state` to store statistics about memory freed by the slab freeing mechanism.

Then we determine if we made any progress, i.e. whether we actually reclaimed any folios, which we store in the boolean `reclaimable` variable.

We then consider indirect reclaim-specific logic, predicated on the `current_is_kswapd()` function.

Firstly, we compare the number of pages which are undergoing writeback (i.e. have the `PG_writeback` flag set) in this scan, (stored in `struct scan_control->nr.writeback`) to the number of pages which we have “isolated” (stored in `struct scan_control->nr.taken`).

Isolated pages are those which have been removed from LRU lists in preparation for being reclaimed, as performed by `isolate_lru_pages()` which we examine in Listing 11-57 later in the chapter.

If the two are equal, then this suggests that we are not throttling dirty pages fast enough to avoid having to write them back (see Section 10.14 in Chapter 10 for details about the dirty page balance throttle mechanism on writeback) or reclaim is not being throttled (see Section 11.6 for details about reclaim throttle).

In this case we set the `PGDAT_WRITEBACK` node flag, which causes the core reclaim function `shrink_page_list()` (see Listing 11-63 in Section 11.5.9), when performing `kswapd` reclaim, to mark these pages for immediate reclaim by ultimately incrementing `struct scan_control->nr.immediate` rather than `struct scan_control->nr.writeback`. This ultimately results in reclaim throttling (we discuss this case momentarily).

Next we examine the number of unqueued dirty pages — these are pages belonging to folios which are dirty but do not yet have the `PG_writeback` flag set. If these are equal to the number of file pages isolated for reclaim this indicates that we have a very large number of dirty pages and thus set the `PGDAT_DIRTY` flag on the node.

The `PGDAT_DIRTY` flag causes `shrink_page_list()` (see Listing 11-63 in Section 11.5.9), for indirect reclaim (that is `kswapd`) to write back file-backed folios directly rather than waiting for the ordinary write back process to complete (see Chapter 10).

Next we check to see if any `PGDAT_WRITEBACK` pages were flagged for immediate reclaim (again in the case that indirect reclaim is being performed), in which case we throttle reclaim via `reclaim_throttle()` setting the `VMSCAN_THROTTLE_WRITEBACK` flag (see Listing 11-75 in Section 11.6 for details).

Next we consider the case, in indirect reclaim (tested via `current_is_kswapd()`), note that the `cgroup` and `sanity` test on line 3364 are out of scope for the book but evaluate to `false` if `cgroup` is disabled) where there are dirty or writeback pages (as stored in `struct scan_control->nr.dirty`) and these are equal to the number of “congested” pages stored in `struct scan_control->nr.congested`.

The congested count is equal to the number of pages which are in writeback and have the `PG_reclaim` flag set, meaning they have looped around the inactive LRU vector again and have yet to be written back.

If this count is equal to the number of writeback pages, then this means we are cycling through reclaim unable to make forward progress for file-backed pages. This implies heavy congestion, and so we set the `LRUVEC_CONGESTED` flag for the LRU vector to indicate this fact.

In the very next predicate we examine whether this is set for the direct reclaim case. We do not do so for indirect reclaim, throttling in this

case only when all of the scanned folios are undergoing writeback as in the `VMSCAN_THROTTLE_CONGESTED` throttle case previously mentioned.

We invoke `current_may_throttle()` to determine if the current thread can throttle, this is only not the case in peculiar edge cases.

In this case we invoke the throttling via `reclaim_throttle()` setting the `VMSCAN_THROTTLE_CONGESTED` flag (see Listing 11-75 in Section 11.6 for details).

Having examined all these throttle cases, we determine whether we should repeat the reclaim attempt via `should_continue_reclaim()`. We examine this in Listing 11-45 shortly.

Otherwise, if we performed at least some reclaim, we reset the struct `pglist_data->kswapd_failures` counter. This is because we've made progress, and we don't want to give up on reclaim as a result.

If we did not do this, then `kswapd` could spin uselessly rather than going to sleep after `MAX_RECLAIM_RETRIES` attempts with no progress (hard-coded to sixteen).

```

3076 /*
3077  * Reclaim/compaction is used for high-order allocation requests. It reclaims
3078  * order-0 pages before compacting the zone. should_continue_reclaim() returns
3079  * true if more pages should be reclaimed such that when the page allocator
3080  * calls try_to_compact_pages() that it will have enough free pages to succeed
3081  *
3082  * It will give up earlier than that if there is difficulty reclaiming pages.
3083  */
3084 static inline bool should_continue_reclaim(struct pglist_data *pgdat,
3085                                           unsigned long nr_reclaimed,
3086                                           struct scan_control *sc)
3087 {
3088     unsigned long pages_for_compaction;
3089     unsigned long inactive_lru_pages;
3090     int z;
3091
3092     /* If not in reclaim/compaction mode, stop */
3093     if (!in_reclaim_compaction(sc))
3094         return false;
3095
3096     /*
3097      * Stop if we failed to reclaim any pages from the last
3098      * SWAP_CLUSTER_MAX
3099      * number of pages that were scanned. This will return to the caller
3100      * with the risk reclaim/compaction and the resulting allocation
3101      * attempt
3102      * fails. In the past we have tried harder for __GFP_RETRY_MAYFAIL
3103      * allocations through requiring that the full LRU list has been
3104      * scanned
3105      * first, by assuming that zero delta of sc->nr_scanned means full LRU
3106      * scan, but that approximation was wrong, and there were corner cases
3107      * where always a non-zero amount of pages were scanned.

```

```

3104     */
3105     if (!nr_reclaimed)
3106         return false;
3107
3108     /* If compaction would go ahead or the allocation would succeed, stop
3109     */
3110     for (z = 0; z <= sc->reclaim_idx; z++) {
3111         struct zone *zone = &pgdat->node_zones[z];
3112         if (!managed_zone(zone))
3113             continue;
3114
3115         switch (compaction_suitable(zone, sc->order, 0, sc->
3116         reclaim_idx)) {
3117             case COMPACT_SUCCESS:
3118             case COMPACT_CONTINUE:
3119                 return false;
3120             default:
3121                 /* check next zone */
3122                 ;
3123         }
3124     }
3125
3126     /*
3127     * If we have not reclaimed enough pages for compaction and the
3128     * inactive lists are large enough, continue reclaiming
3129     */
3130     pages_for_compaction = compact_gap(sc->order);
3131     inactive_lru_pages = node_page_state(pgdat, NR_INACTIVE_FILE);
3132     if (can_reclaim_anon_pages(NULL, pgdat->node_id, sc))
3133         inactive_lru_pages += node_page_state(pgdat, NR_INACTIVE_ANON)
3134     ;
3135
3136     return inactive_lru_pages > pages_for_compaction;
3137 }

```

Listing 11-45: mm/vmscan.c: should_continue_reclaim()

The `should_continue_reclaim()` is only relevant for allocations of higher order folios for which we intend to invoke compaction. As previously mentioned, compaction is out of scope for the book, but is the process through which the kernel migrates movable folios (those with a movable migrate type, see Section 2.5 in Chapter 2 for details), i.e. moves them in physical memory such that adjacent free folios of lower order can be coalesced into a higher order free folio.

Even though this is out of scope, it is a key part of how reclaim performs, so like working set logic, it is important to examine how it interfaces with reclaim, even if we don't go into the details of how compaction itself proceeds.

The `should_continue_reclaim()` function returns true if we need to perform more reclaim to free up sufficient pages for compaction to succeed. Note that we wake up the kernel compaction thread `kcompactd` in `kswapd_try_to_sleep()` as shown in Listing 11-28 and Section 11.4.3.

We start by determining whether this is reclaim subject to compaction via `in_reclaim_compaction()` which we examine in Listing 11-46.

```

3065 /* Use reclaim/compaction for costly allocs or under memory pressure */
3066 static bool in_reclaim_compaction(struct scan_control *sc)
3067 {
3068     if (IS_ENABLED(CONFIG_COMPACTON) && sc->order &&
3069         (sc->order > PAGE_ALLOC_COSTLY_ORDER ||
3070          sc->priority < DEF_PRIORITY - 2))
3071         return true;
3072
3073     return false;
3074 }
```

Listing 11-46: mm/vmscan.c: `in_reclaim_compaction()`

The `in_reclaim_compaction()` function checks that firstly compaction is enabled, that an allocation of an order-4 or greater folio caused the reclaim (since `PAGE_ALLOC_COSTLY_ORDER` is hard-coded to three), or an allocation of an order-1 or greater folio and we on the fourth iteration or more of reclaim due to `struct scan_control->priority` being less than `DEF_PRIORITY` minus two.

Returning to `should_continue_reclaim()` in Listing 11-45, we see that we don't consider continuing to reclaim if `in_reclaim_compaction()` indicates the allocation is not suited to compaction. Equally, we abort if no folios were reclaimed at all.

Then we check to see if any zone from which we could allocate are in a state suitable for compaction via `compaction_suitable()`—if any are then we exit indicating that reclaim should not continue.

Finally, we check to ensure there are sufficient inactive folios (expressed in base pages) such that we could reclaim enough to perform compaction and use this to determine whether reclaim should continue.

We determine the number of pages we ought to have available via `compact_gap()` which is equal to double the number of base pages of the order which failed allocation, and is considered a good heuristic for the number of free pages we require to succeed at compaction.

We then retrieve a count of inactive folios (expressed in base pages), including anonymous folios only if `can_reclaim_anon_pages()`, which first checks that there are at least some swap pages we could swap out to, and if not whether NUMA demotion is available (a topic out of scope for the book).

If the total of these inactive pages exceeds the number of base pages we require for compaction, then continuing reclaim is worthwhile, otherwise it is not.

11.5.4 Determining Scan Balance

As per Figure 11-11, the next substantive layer of reclaim, performed after the struct `scan_control` is configured, is `shrink_lruvec()` (see Listing 11-51 and Section 11.5.5).

A key function which `shrink_lruvec()` invokes is `get_scan_count()` which determines the scanning split between file-backed and anonymous folios. We examine this in Listing 11-47 (eliding out of scope cgroup logic where doing so wouldn't confuse the logic).

```

2731 /*
2732  * Determine how aggressively the anon and file LRU lists should be
2733  * scanned.
2734  *
2735  * nr[0] = anon inactive pages to scan; nr[1] = anon active pages to scan
2736  * nr[2] = file inactive pages to scan; nr[3] = file active pages to scan
2737  */
2738 static void get_scan_count(struct lruvec *lruvec, struct scan_control *sc,
2739                          unsigned long *nr)
2740 {
2741     struct pglist_data *pgdat = lruvec_pgdat(lruvec);
2742     struct mem_cgroup *memcg = lruvec_memcg(lruvec);
2743     unsigned long anon_cost, file_cost, total_cost;
2744     int swappiness = mem_cgroup_swappiness(memcg);
2745     u64 fraction[ANON_AND_FILE];
2746     u64 denominator = 0;    /* gcc */
2747     enum scan_balance scan_balance;
2748     unsigned long ap, fp;
2749     enum lru_list lru;
2750
2751     /* If we have no swap space, do not bother scanning anon pages. */
2752     if (!sc->may_swap || !can_reclaim_anon_pages(memcg, pgdat->node_id, sc
2753 )) {
2754         scan_balance = SCAN_FILE;
2755         goto out;
2756     }
2757     :
2758     /*
2759     * Do not apply any pressure balancing cleverness when the
2760     * system is close to OOM, scan both anon and file equally
2761     * (unless the swappiness setting disagrees with swapping).
2762     */
2763     if (!sc->priority && swappiness) {
2764         scan_balance = SCAN_EQUAL;
2765         goto out;
2766     }
2767
2768     /*
2769
```

```

2780      * If the system is almost out of file pages, force-scan anon.
2781      */
2782      if (sc->file_is_tiny) {
2783          scan_balance = SCAN_ANON;
2784          goto out;
2785      }
2786
2787      /*
2788       * If there is enough inactive page cache, we do not reclaim
2789       * anything from the anonymous working right now.
2790       */
2791      if (sc->cache_trim_mode) {
2792          scan_balance = SCAN_FILE;
2793          goto out;
2794      }

```

Listing 11-47: mm/vmscan.c: get_scan_count() Initialisation & Edge Cases

The `get_scan_count()` function maintains a `enum_scan_balance` value, which indicates how to proceed in the latter part of the function—`SCAN_EQUAL` indicates that each list should be scanned according to the priority-adjusted number of pages in each list, i.e. we should not adjust the number of pages available to scan.

`SCAN_ANON` indicates that only anonymous folios should be scanned and `SCAN_FILE` indicates that only file-backed folios should be scanned.

Finally, `SCAN_FRACT` is the typical mode in which `get_scan_count()` operates, which distributes base pages to scan based on a heuristically determined fraction.

The `nr` output parameter, which points to an unsigned long array of size four, defines how many base pages should be scanned for anonymous inactive, active, file-backed inactive and active LRUs respectively, and is set in all cases.

We start by examining initialisation and the edge cases in Listing 11-47, before examining the `SCAN_FRACT` case in Listing 11-48.

The logic starts with initialisation, most notably invoking `mem_cgroup_swappiness()`. This, in the non-cgroup case, returns the system-wide “swappiness” value which is an integer value indicating the degree to which the system has a preference for swapping out to disk. It is expressed in a range from 0 to 200, with 0 indicating a complete aversion to swapping and 200 indicating a preference for only swapping, with 100 an equal spread between the two.

The global swappiness value is stored in the `vm_swappiness` variable, which defaults to 60.

We then consider a series of edge cases—if the swap is either disallowed or no swap entries exist, we set `SCAN_FILE`.

If we have reached zero priority and are thus scanning everything, and swappiness is non-zero, then at this point where we are close to an Out Of

Memory (OOM) condition, we do not adjust the number of pages to scan in each LRU, setting the mode to `SCAN_EQUAL`.

If the struct `scan_control->file_is_tiny` predicate is set, indicating that we are running low on file-backed folios, then we prefer anonymous folios and set the scan mode to `SCAN_ANON` (see discussion of `shrink_node()` around Listing 11-43 where this is set for details).

Finally, if cache trim mode is enabled, set in struct `scan_control->cache_trim_mode`, then this indicates we have sufficient file-backed folios to free and we are not thrashing them, so we prefer to free file folios, and thus return `SCAN_FILE` (see discussion of `shrink_node()` around Listing 11-43 where this is set for details).

NOTE

Cache trim mode is a key feature of the page cache—the kernel will happily fill up the memory with as much on-disk data as is accessed and keep hold of it, however it'll equally discard it just as quickly. As soon as we reach data that is thrashing or if the inactive file LRU runs low, we attempt to balance reclaim.

We examine the `SCAN_FRACT` case in Listing 11-48.

```

2796     scan_balance = SCAN_FRACT;
2797     /*
2798      * Calculate the pressure balance between anon and file pages.
2799      *
2800      * The amount of pressure we put on each LRU is inversely
2801      * proportional to the cost of reclaiming each list, as
2802      * determined by the share of pages that are refaulting, times
2803      * the relative IO cost of bringing back a swapped out
2804      * anonymous page vs reloading a filesystem page (swappiness).
2805      *
2806      * Although we limit that influence to ensure no list gets
2807      * left behind completely: at least a third of the pressure is
2808      * applied, before swappiness.
2809      *
2810      * With swappiness at 100, anon and file have equal IO cost.
2811      */
2812     total_cost = sc->anon_cost + sc->file_cost;
2813     anon_cost = total_cost + sc->anon_cost;
2814     file_cost = total_cost + sc->file_cost;
2815     total_cost = anon_cost + file_cost;
2816
2817     ap = swappiness * (total_cost + 1);
2818     ap /= anon_cost + 1;
2819
2820     fp = (200 - swappiness) * (total_cost + 1);
2821     fp /= file_cost + 1;
2822
2823     fraction[0] = ap;
2824     fraction[1] = fp;

```

```
denominator = ap + fp;
```

Listing 11-48: mm/vmscan.c: get_scan_count() Fractional Case

In the portion of `get_scan_count()` that calculates the division of pages to scan in the `SCAN_FRACT` mode shown in Listing 11-48 we determine nominator values for anonymous and file-backed pages in the fraction array of size `ANON_AND_FILE` which is hard-coded to two, and the denominator of the fraction.

Here we make use of the refault statistics `struct scan_control->anon_cost` and `struct scan_control->file_cost`, combined with the swappiness statistic (see Section 11.5.2 for a brief explanation of these).

The calculation here is rather fiddly so let's take it step by step. We start by combining these values into a total cost, then defining anonymous page cost and file cost each equal to the sum of the total cost and the respective cost to which the value refers.

We then redefine the total cost to be the sum of these two values.

This is very confusing, so let's denote the anonymous page cost as a , the file-backed page cost as f , the newly-calculated anonymous cost (i.e. `anon_cost`) as A , the newly-calculated file-backed cost (i.e. `file_cost`) as F , the original total cost (`total_cost`) as t and the redefined total cost as T and examine what this looks like:

$$t = a + f$$

$$A = t + a = (a + f) + a = 2a + f$$

$$F = t + f = (a + f) + f = a + 2f$$

$$T = A + F = 2a + f + a + 2f = 3a + 3f$$

From here we use T , A , F and the swappiness parameter (we denote this as S) to calculate anonymous and file-backed fractions.

Note that we add one to values here in order to avoid divide-by zero or zeroed multipliers when performing integer arithmetic, something the kernel has to be very careful of as it is not permitted to perform floating point arithmetic within the kernel.

For the purposes of simplifying this calculation, we will ignore these offsets.

Therefore, denoting the anonymous page numerator as ap as in the code, and the file-backed page denominator as fp :

$$ap = \frac{ST}{A} \quad fp = \frac{(200 - S)T}{F}$$

The denominator (as named in the code) is equal to the some of the two. We denote this D :

$$D = \frac{ST}{A} + \frac{(200 - S)T}{F}$$

We can now calculate the fraction of anonymous pages (denoted as Fr_a) and the fraction of file-backed pages (denoted as Fr_f):

$$Fr_a = \frac{\frac{ST}{A}}{\frac{ST}{A} + \frac{(200 - S)T}{F}}$$

$$Fr_f = \frac{\frac{(200 - S)T}{F}}{\frac{ST}{A} + \frac{(200 - S)T}{F}}$$

Dividing numerator and denominator by T yields:

$$Fr_a = \frac{\frac{S}{A}}{\frac{S}{A} + \frac{200 - S}{F}}$$

$$Fr_f = \frac{\frac{200 - S}{F}}{\frac{S}{A} + \frac{200 - S}{F}}$$

Multiplying numerator and denominator by AF simplifies:

$$Fr_a = \frac{SF}{SF + (200 - S)A}, Fr_f = \frac{(200 - S)A}{SF + (200 - S)A}$$

Since the purpose of the denominator is solely to divide each fraction by the sum of the nominators, i.e. to determine the relative proportion of anonymous and file-backed pages, let's redefine D to the new denominator and express the equations in terms of the struct `scan_control->anon_cost` and struct `scan_control->file_cost` values, a and f respectively:

$$Fr_a = \frac{SF}{D} = \frac{S(a + 2f)}{D}$$

$$Fr_f = \frac{(200 - S)A}{D} = \frac{(200 - S)(2a + f)}{D}$$

NOTE

Having removed the offset-by-1 cases for simplicity, we note that here we can treat there being no refaults as equivalent to there being an equal number of refaults, that is if a and f are equal to zero, consider this equivalent to them both being equal to one.

The fraction of anonymous pages is proportional to swappiness multiplied by the sum of anonymous refaults and double the file-backed refaults, or in other words—up to a maximum of two-thirds file-backed refaults.

The fraction of file-backed pages is proportional to the complement of swappiness (recalling that it ranges from zero to two hundred, spanning a desire to not swap at all to swapping at expense of all file-backed reclaim) multiplied by the sum of double anonymous refaults and file-backed refaults or in other words—up to a maximum of two-thirds anonymous refaults.

Scanning of anonymous pages is therefore proportional to file-backed thrashing up to a maximum of two-thirds of the scan, and scanning of file-backed pages is proportional to anonymous thrashing up to a maximum of two-thirds of the scan.

This renders anonymous scanning inversely proportional to anonymous thrashing, and file-backed scanning equally inversely proportional to file-backed thrashing.

This capping at two-thirds is quite subtle, though the comment hints at it—we do not allow a list to fall completely behind, always providing at least one third of the pressure no matter the thrashing.

The swappiness is always an overriding parameter and anonymous scanning is proportional to this and file-backed scanning is proportional to the complement of this.

For instance, if swappiness is equal to 50 then scanning is performed over 25% of anonymous pages and 75% of file-backed pages, if it's equal to 150 then scanning is performed over 75% of anonymous pages and 25% of file-backed pages.

If we examine reclaim pressure on file-backed folios as anonymous re-faults climb in Figure 11-15, we see that swappiness is always linearly applied, existing file re-faults cause an initial dip in the climbing pressure (as this is proportional to the ratio between the two) and that we're always capped off.

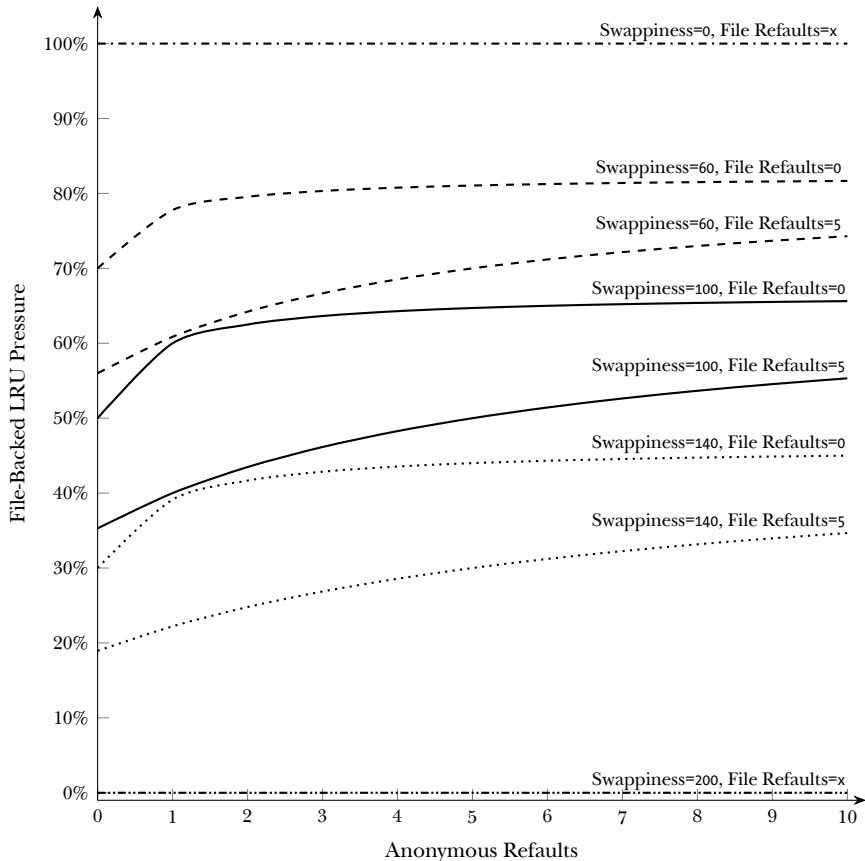


Figure 11-15: Example File-Backed Reclaim Pressure Split

Note that Figure 11-15 we see that in the case of 100 swappiness, we always end up capping at two-thirds of the file-backed pressure.

At other swappiness levels, we need to consider the equations when anonymous refaults significantly outstrip file-backed, i.e. to determine what the file-backed LRU pressure tends to in the limit.

Let's expand out the equation:

$$Fr_f = \frac{(200 - S)(2a + f)}{S(f + 2a) + (200 - S)(2a + f)}$$

If we assume that f is greatly exceeded by a then we can eliminate this term:

$$Max(Fr_f) = \frac{2a(200 - S)}{2aS + 2a(200 - S)}$$

Eliminating a and expanding this out leaves us with:

$$Max(Fr_f) = \frac{200 - S}{200 - S/2}$$

Which we can observe in Figure 11-15.

We consider the equivalent anonymous LRU case in Figure 11-16.

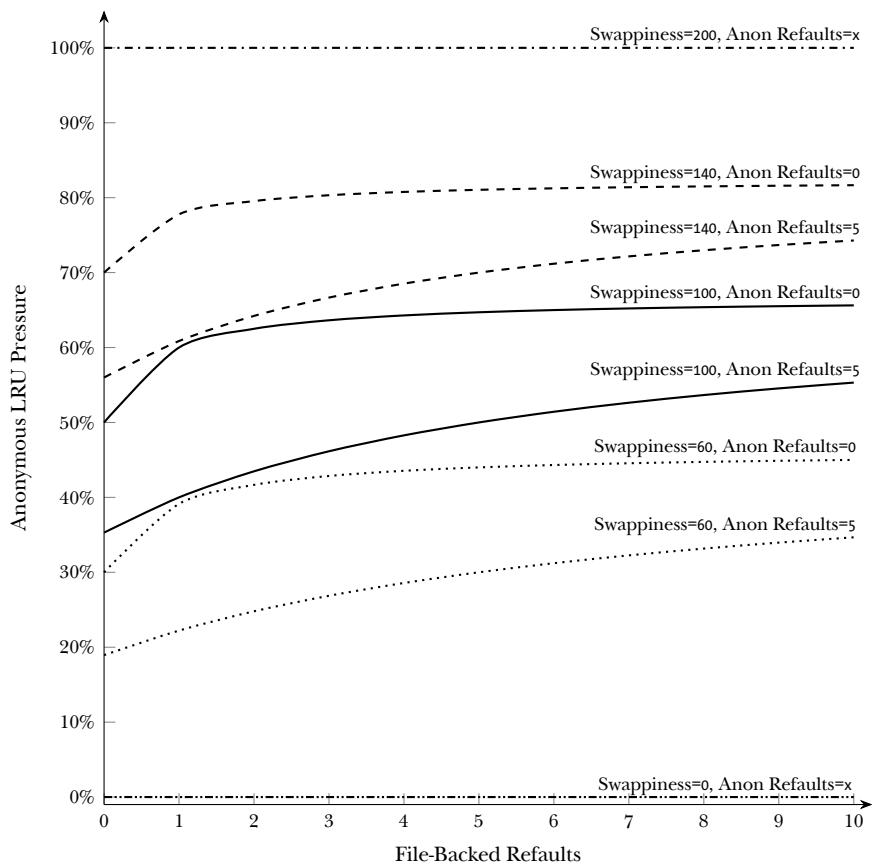


Figure 11-16: Example Anonymous Reclaim Pressure Split

In Figure 11-16 we observe that the same pattern emerges, only with swappiness inverted by subtracting from 200.

Performing the same task as we did with the file-backed case, we observe that in the limit we obtain the maximum anonymous LRU pressure as follows:

$$\text{Max}(Fr_a) = \frac{S(a + 2f)}{S(a + 2f) + (200 - S)(2a + f)}$$

Again, in the limit, the file-backed refaults are large, meaning we can disregard the anonymous refault count and obtain:

$$\text{Max}(Fr_a) = \frac{2fS}{2fS + f(200 - S)}$$

Eliminating f and expanding this out leaves us with:

$$\text{Max}(Fr_a) = \frac{2S}{200 + S}$$

Which we can observe in Figure 11-16.

Now that we've determined how to proceed, we return to `get_scan_count()`, we examine in Listing 11-49 the code that actually applies this to the `nr` output array.

```

2826 out:
2827     for_each_evictable_lru(lru) {
2828         int file = is_file_lru(lru);
2829         unsigned long lruvec_size;
2830         unsigned long low, min;
2831         unsigned long scan;
2832
2833         lruvec_size = lruvec_lru_size(lruvec, lru, sc->reclaim_idx);
2834         :
2891         scan = lruvec_size;
2892         :
2894         scan >>= sc->priority;
2895         :
2903         switch (scan_balance) {
2904             case SCAN_EQUAL:
2905                 /* Scan lists relative to size */
2906                 break;
2907             case SCAN_FRACT:
2908                 /*
2909                  * Scan types proportional to swappiness and
2910                  * their relative recent reclaim efficiency.
2911                  * Make sure we don't miss the last page on
2912                  * the offlined memory cgroups because of a
2913                  * round-off error.
2914                  */
2915                 scan = mem_cgroup_online(memcg) ?
2916                     div64_u64(scan * fraction[file], denominator) :

```

```

2917             DIV64_U64_ROUND_UP(scan * fraction[file],
2918                                 denominator);
2919             break;
2920         case SCAN_FILE:
2921         case SCAN_ANON:
2922             /* Scan one type exclusively */
2923             if ((scan_balance == SCAN_FILE) != file)
2924                 scan = 0;
2925             break;
2926         default:
2927             /* Look ma, no brain */
2928             BUG();
2929     }
2930
2931     nr[lru] = scan;
2932 }
2933 }

```

Listing 11-49: mm/vmscan.c: get_scan_count() Calculating Scan Balance

The portion of `get_scan_count()` that we examine in Listing 11-49 populates the output `nr` array with the count of pages to scan in each LRU.

This uses the `for_each_evictable_lru()` macro to iterate through the inactive anonymous, active anonymous, inactive file-backed and active file-backed LRUs respectively, setting the `lru` variable to each of this of type `enum lru_list`.

We calculate the size each LRU list via `lruvec_lru_size()`, in the non-cgroup case simply setting the base number of pages to scan for each LRU at this count shifted by the priority level.

For each scan mode we see whether we need to adjust the scan count. For `SCAN_EQUAL` we by definition do not need to, as this implies we simply scan each list according to the number of priority-adjusted pages.

In order to determine if we are examining a file-backed LRU, at the start of each loop we define the `file` boolean which uses `is_file_lru()` to determine this. That way we can handle cases which differ between anonymous and file-backed folios correctly.

For `SCAN_FRACT` we apply the previously calculated fraction to the number of pages we need to scan. Note that `mem_cgroup_online()` defaults to true if cgroups are disabled.

The `div64_u64()` permits for safe unsigned 64-bit division even on 32-bit architectures.

Finally for `SCAN_FILE` and `SCAN_ANON` modes we assign all of the pages to scan to either anonymous or file-backed pages.

With this function explored, we are ready to see it put to use in `shrink_lruvec()` which we examine in Listing 11-51 and Section 11.5.5.

11.5.5 Shrinking LRU Vectors

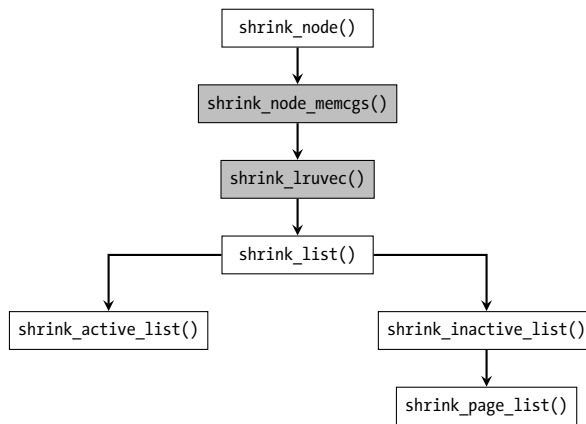


Figure 11-17: Reclaim Mechanism: Shrinking LRU Vectors Code Path

Looking at Figure 11-17, we can see that after `shrink_node()` (see Listing 11-40 and Section 11.5.3 for details) we next invoke `shrink_node_memcgs()`, which we examine in Listing 11-50 (eliding out of scope cgroup and real-time scheduling logic).

```

3136 static void shrink_node_memcgs(pg_data_t *pgdat, struct scan_control *sc)
3137 {
3138     :
3180     shrink_lruvec(lruvec, sc);
3181
3182     shrink_slab(sc->gfp_mask, pgdat->node_id, memcg,
3183                sc->priority);
3184     :
3192 }
  
```

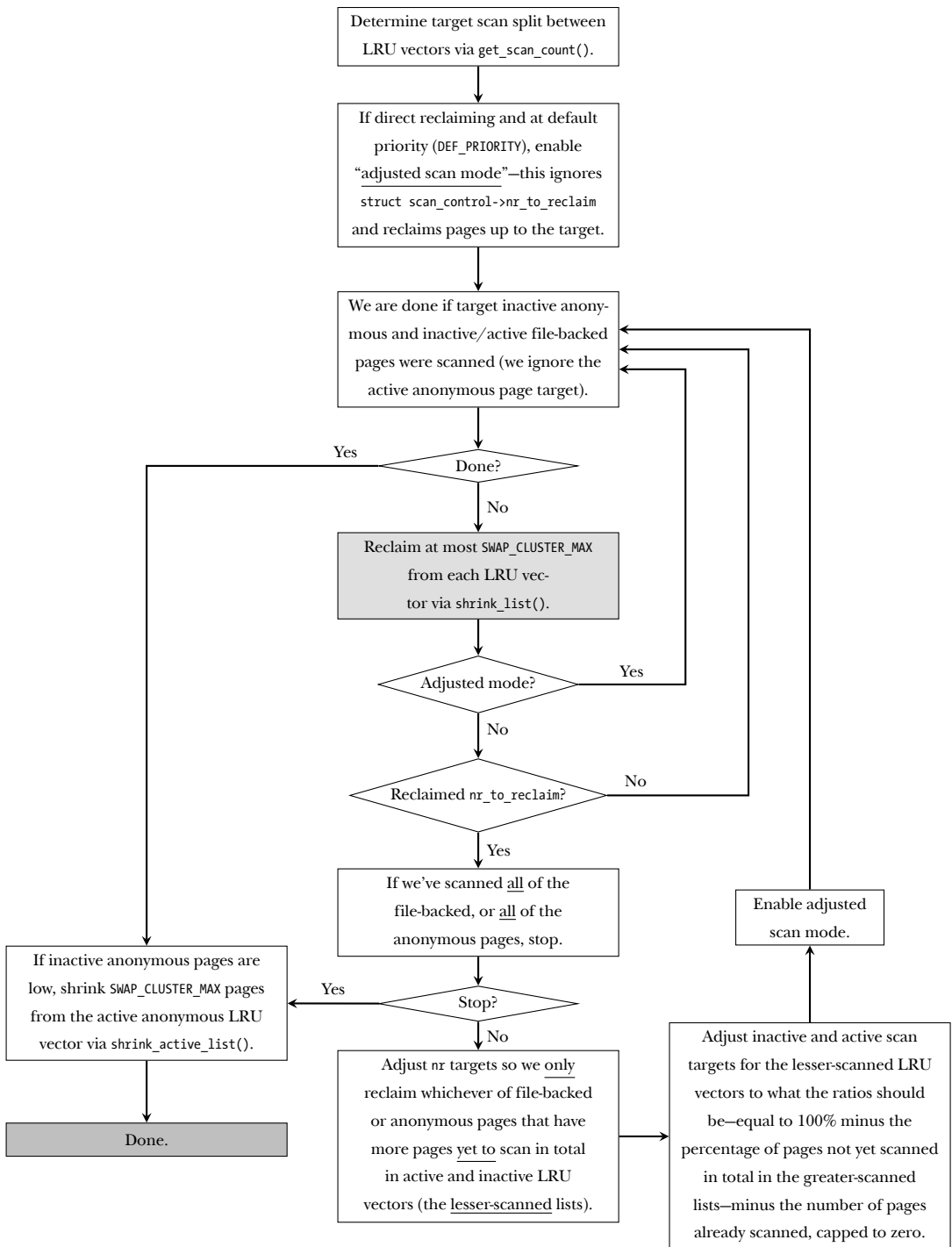
Listing 11-50: mm/vmscan.c: `shrink_node_memcgs()`

The majority of the `shrink_node_memcgs()` function is concerned with cgroup logic which is out of scope for the book, however whether cgroups are enabled or not, `shrink_lruvec()` is invoked to effect the reclaim for the node configured by the specified `struct scan_control` object, which we examine in Listing 11-51.

NOTE

We also invoke `shrink_slab()` which causes slab memory to free up unused cache memory. However, discussion of this is entirely out of the scope of the book.

We examine the logic of `shrink_lruvec()` in Figure 11-18.

Figure 11-18: `shrink_lruvec()` Logic

NOTE

In Figure 11-18 when we refer to greater-scanned and lesser-scanned, we mean as a proportion of the target number of pages to scan, so even if we scanned exactly the same number of pages for file-backed and anonymous LRU vectors, it is the one which has fewer pages to scan that is considered greater-scanned, and the one which has more pages to scan that is considered lesser-scanned.

We examine `shrink_lruvec()` starting in Listing 11-51 (eliding out of scope `cgroup`, real-time scheduling and block plugging logic).

```

2950 static void shrink_lruvec(struct lruvec *lruvec, struct scan_control *sc)
2951 {
2952     unsigned long nr[NR_LRU_LISTS];
2953     unsigned long targets[NR_LRU_LISTS];
2954     unsigned long nr_to_scan;
2955     enum lru_list lru;
2956     unsigned long nr_reclaimed = 0;
2957     unsigned long nr_to_reclaim = sc->nr_to_reclaim;
2958     :
2959     bool scan_adjusted;
2960
2961     get_scan_count(lruvec, sc, nr);
2962
2963     /* Record the original scan target for proportional adjustments later
2964      */
2965     memcpy(targets, nr, sizeof(nr));
2966
2967     /*
2968      * Global reclaiming within direct reclaim at DEF_PRIORITY is a normal
2969      * event that can occur when there is little memory pressure e.g.
2970      * multiple streaming readers/writers. Hence, we do not abort scanning
2971      * when the requested number of pages are reclaimed when scanning at
2972      * DEF_PRIORITY on the assumption that the fact we are direct
2973      * reclaiming implies that kswapd is not keeping up and it is best to
2974      * do a batch of work at once. For memcg reclaim one check is made to
2975      * abort proportional reclaim if either the file or anon lru has
2976      already
2977      * dropped to zero at the first pass.
2978      */
2979     scan_adjusted = (!cgroup_reclaim(sc) && !current_is_kswapd()) &&
2980                    sc->priority == DEF_PRIORITY);

```

Listing 11-51: `mm/vmscan.c: shrink_lruvec()` Preface

The beginning of `shrink_lruvec()` shown in Listing 11-51 starts by invoking `get_scan_count()` (see Listing 11-47 and Section 11.5.4) to determine the number of pages to scan in the file-backed and anonymous inactive and active LRU vectors, all stored in the local `nr` array.

These values are then copied into the target array so we can modify the `nr` array as we proceed but retain the original targets.

We then determine whether we are direct reclaiming and we are at the default priority level, which enables an “adjusted” scan (we do not consider the cgroup case, note that `cgroup_reclaim()` returns false if cgroup is not enabled).

In this mode we ignore the struct `scan_control->nr_to_reclaim` target and instead simply loop around until we have scanned all of the pages specified by `nr`.

The justification for this, as per the comment, is that we are not under significant memory pressure yet (as we are at the default priority), and the fact direct reclaim has been invoked suggests that the target that has otherwise been set is insufficient to keep up with what little memory pressure there is.

The default priority, `DEF_PRIORITY` (hard-coded to 12), results in 1/4096 of the LRU lists being scanned, so this should be relatively safe under ordinary circumstances.

Having established state and determined whether adjusted scan mode is required, we then start the core reclaim loop as shown in Listing 11-52.

```

2981     while (nr[LRU_INACTIVE_ANON] || nr[LRU_ACTIVE_FILE] ||
2982           nr[LRU_INACTIVE_FILE]) {
2983         unsigned long nr_anon, nr_file, percentage;
2984         unsigned long nr_scanned;
2985
2986         for_each_evictable_lru(lru) {
2987             if (nr[lru]) {
2988                 nr_to_scan = min(nr[lru], SWAP_CLUSTER_MAX);
2989                 nr[lru] -= nr_to_scan;
2990
2991                 nr_reclaimed += shrink_list(lru, nr_to_scan,
2992                                           lruvec, sc);
2993             }
2994         }
2995         :
2998         if (nr_reclaimed < nr_to_reclaim || scan_adjusted)
2999             continue;

```

Listing 11-52: mm/vmscan.c: `shrink_lruvec()` Main Reclaim Loop

We loop while inactive anonymous, active file-backed and inactive file-backed pages are still to be scanned (if only active anonymous pages remain, these are handled separately).

We iterate through each LRU list (including active anonymous pages), reclaiming in batches of up to `SWAP_CLUSTER_MAX`, which is decremented from `nr` immediately, whether reclaimed or not. We skip LRU vectors which have zero pages to scan.

The actual reclaim is performed via `shrink_list()`, which we explore later in Listing 11-55, which returns the number of actually reclaimed pages which we accumulate into `nr_reclaimed`.

After this initial pass of reclaim has been performed, we check to see if either we have not scanned all the required pages or if adjusted scan mode is enabled—in either case we repeat the loop until we have scanned all but active anonymous pages counted in `nr`.

With this done we perform post-reclaim logic as explored in Listing 11-53.

```

3001      /*
3002       * For kswapd and memcg, reclaim at least the number of pages
3003       * requested. Ensure that the anon and file LRUs are scanned
3004       * proportionally what was requested by get_scan_count(). We
3005       * stop reclaiming one LRU and reduce the amount scanning
3006       * proportional to the original scan target.
3007       */
3008      nr_file = nr[LRU_INACTIVE_FILE] + nr[LRU_ACTIVE_FILE];
3009      nr_anon = nr[LRU_INACTIVE_ANON] + nr[LRU_ACTIVE_ANON];
3010
3011      /*
3012       * It's just vindictive to attack the larger once the smaller
3013       * has gone to zero. And given the way we stop scanning the
3014       * smaller below, this makes sure that we only make one nudge
3015       * towards proportionality once we've got nr_to_reclaim.
3016       */
3017      if (!nr_file || !nr_anon)
3018          break;
3019
3020      if (nr_file > nr_anon) {
3021          unsigned long scan_target = targets[LRU_INACTIVE_ANON]
+
3022                                     targets[LRU_ACTIVE_ANON] + 1;
3023          lru = LRU_BASE;
3024          percentage = nr_anon * 100 / scan_target;
3025      } else {
3026          unsigned long scan_target = targets[LRU_INACTIVE_FILE]
+
3027                                     targets[LRU_ACTIVE_FILE] + 1;
3028          lru = LRU_FILE;
3029          percentage = nr_file * 100 / scan_target;
3030      }
3031
3032      /* Stop scanning the smaller of the LRU */
3033      nr[lru] = 0;
3034      nr[lru + LRU_ACTIVE] = 0;
3035
3036      /*
3037       * Recalculate the other LRU scan count based on its original
3038       * scan target and the percentage scanning already complete
3039       */

```

```

3040         lru = (lru == LRU_FILE) ? LRU_BASE : LRU_FILE;
3041         nr_scanned = targets[lru] - nr[lru];
3042         nr[lru] = targets[lru] * (100 - percentage) / 100;
3043         nr[lru] -= min(nr[lru], nr_scanned);
3044
3045         lru += LRU_ACTIVE;
3046         nr_scanned = targets[lru] - nr[lru];
3047         nr[lru] = targets[lru] * (100 - percentage) / 100;
3048         nr[lru] -= min(nr[lru], nr_scanned);
3049
3050         scan_adjusted = true;
3051     }

```

Listing 11-53: mm/vmscan.c: shrink_lruvec() Reclaim Loop Post-Processing

The portion of `shrink_lruvec()` we examine in Listing 11-54 is concerned with ensuring that we maintain the proportions between LRU vectors expressed by `get_scan_count()` (see Listing 11-47 and Section 11.5.4).

Since we batch a maximum of `SWAP_CLUSTER_MAX` pages at a time (hard-coded to 32 pages), we may end up in the unfortunate position of reaching or exceeding our reclaim target while not maintaining the proportionality between file-backed and anonymous pages asserted by swappiness at all.

The logic that exists here attempts to correct for this by first halting scanning altogether on the lists which have been scanned most (after all, we can't unscan already scanned pages, so we can only adjust proportionality based on the scanning more, and this is most effectively done on the least-scanned list).

We start by firstly determining how many pages we have yet to scan in aggregate across inactive and active lists for file-backed and anonymous pages separately.

If either of these are zero, we abort, as we might then end up reducing the other lists to zero, causing unnecessary reclaim stalls.

Otherwise, we proceed, determining which of file-backed or anonymous pages have the most number of pages yet to scan (that is, the lesser-scanned lists), as stored in `nr_file` and `nr_anon` respectively.

Whichever has more, we clear the target for the other so the greater-scanned list is the only one we reclaim from, and we store the percentage of pages remaining to be scanned of the greater-scanned lists.

The intent here is to use this percentage in order to ensure that the lesser-scanned lists has a complementary percentage of pages scanned.

The output of `get_scan_count()` (see Listing 11-47) is such that the proportion of file-backed or anonymous pages is always equal to the complement of the other—so for instance, if we are to scan 20% of file-backed pages, we scan 80% of anonymous pages, that is, one has a proportion of pages scanned equal to 100% minus the percentage of pages scanned of the other (having already truncated the scan count by the reclaim priority). See Section 11.5.4 for more details.

To recover proportionality, we apply the same logic here—we know what proportion of target pages were scanned for the greater-scanned lists, so we simply target the complement, or 100% minus this percentage of the lesser-scanned lists.

We additionally subtract the number of pages already scanned so we do not exceed the target, capping this result at zero.

Finally, we set the adjusted scan mode so this logic is only performed once and that we ensure that we do scan pages up to the target amount. We then loop around to perform the adjusted scan as shown in Figure 11-18.

It's easier to understand this with a worked example. Consider a case where the struct `scan_control->nr_to_reclaim` field is set to 128, and `get_scan_count()` (see Listing 11-47) sets anonymous page targets to 128 pages in total split evenly, and file-backed page targets to 256 pages in total, split evenly.

We start with page reclaim targets as shown in Table 11-1.

Table 11-1: Proportionality Example Part 1

LRU	Target Scan	Target%
Anonymous Inactive	64	10%
Anonymous Active	64	10%
File-backed Inactive	256	40%
File-backed Active	256	40%

Note that this implies an 80%/20% split in favour of file-backed pages.

After a single loop through each LRU vector, assuming all pages are reclaimed, we achieve our target struct `scan_control->nr_to_reclaim` value.

However, this has not resulted in a proportional scan at all, in fact we have simply scanned all lists evenly, as shown in Table 11-2.

Table 11-2: Proportionality Example Part 2

LRU	Remaining Scan	Scanned	Scanned%	Scanned/Target%
Anonymous Inactive	32	32	25%	50%
Anonymous Active	32	32	25%	50%
File-backed Inactive	224	32	25%	12.5%
File-backed Active	224	32	25%	12.5%

Table 11-2 perfectly demonstrates the issue. We have hit our reclaim target but the proportions scanned have absolutely nothing to do with what `get_scan_count()` (see Listing 11-47) has specified.

In addition, we can very clearly see that we have little room to manoeuvre from the point of view of anonymous pages—one more batch of `SWAP_CLUSTER_MAX` pages and we will have exhausted the anonymous LRU vectors which would simply be perverse, as our intent is to scan significantly more file-backed pages.

Therefore the wisdom of simply only scanning those lesser-scanned pages, that is, those with the greater number of pages remaining to be scanned, becomes apparent.

We have 448 file-backed pages which we can scan, and only 64 anonymous ones, and we are therefore far better able to correct the situation by scanning more file-backed ones.

In this instance we see the percentage we use in the algorithm shown in Listing 11-53 is equal to 32 divided by a target of 64, that is 50%.

This results in a target of 50% of 256, that is 128 for each of the inactive and active file-backed LRU vectors, less the already scanned 32 pages, leaving us with a target of 96 each or 192 pages in total which to scan file-backed LRU vectors only in adjusted scan mode.

Assuming we successfully scan in these proportions, we are left with the final aggregate results as shown in Table 11-3.

Table 11-3: Proportionality Example Part 3

LRU	Scanned	Scanned%	Scanned/Target%
Anonymous Inactive	32	10%	50%
Anonymous Active	32	10%	50%
File-backed Inactive	128	40%	50%
File-backed Active	128	40%	50%

We can see in Table 11-3 the final scan counts for each LRU vector, the original 32 pages scanned from anonymous page lists, and file-backed vectors with the original 32 plus the added 96 pages.

We will have scanned (and potentially reclaimed) 320 pages, which is two and half times the target number to reclaim, but we will have maintained the required proportions.

NOTE

We hit 50% of the `get_scan_count()` (see Listing 11-47) targets, but in indirect reclaim or direct reclaim below the default priority level, we do not intend to hit these targets, rather at or above the requested number of pages to reclaim.

With this logic complete or on early exit as discussed, we perform post-loop logic as shown in Listing 11-54.

```

3053         sc->nr_reclaimed += nr_reclaimed;
3054
3055     /*
3056      * Even if we did not try to evict anon pages at all, we want to
3057      * rebalance the anon lru active/inactive ratio.
3058      */
3059     if (can_age_anon_pages(lruvec_pgdat(lruvec), sc) &&
3060         inactive_is_low(lruvec, LRU_INACTIVE_ANON))
3061         shrink_active_list(SWAP_CLUSTER_MAX, lruvec,
3062                             sc, LRU_ACTIVE_ANON);
3063 }
```

Listing 11-54: mm/vmscan.c: `shrink_lruvec()` Suffix

At the end of `shrink_lruvec()`, as shown in Listing 11-54, we perform a direct check as to whether we can age anonymous pages, that is whether

`can_age_anon_pages()` returns true. Other than in the case of NUMA demotion, this simply checks to see whether we have at least 1 swap page available.

We then check to see if the inactive anonymous LRU vector is proportionally small, which we check via `inactive_is_low()` (see Listing 11-42 and Section 11.5.3).

If so, then we shrink the active list via `shrink_active_list()` (see Listing 11-56 and Section 11.5.7).

NOTE

Having noted in Section 11.4.4 on indirect reclaim about consistent aging of active anonymous folios, we pay special attention to maintaining a sufficient large inactive anonymous folio LRU list on indirect reclaim, as `get_scan_count()` might very well indicate anonymous active folios should not be reclaimed (for instance in cache trim mode). See Section 11.5.7 for more.

11.5.6 Shrinking an Individual LRU List

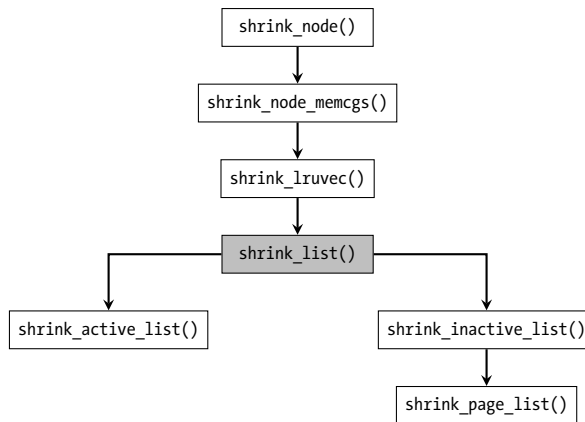


Figure 11-19: Reclaim Mechanism: Shrinking an Individual LRU List Code Path

We are now at the level of granularity of an individual LRU vector, whether file-backed or anonymous, inactive or active, which is shrunk by the `shrink_list()` function, which we examine in Listing 11-55. This is invoked in `shrink_lruvec()` (see Listing 11-51), as discussed in Section 11.5.5.

```

2663 static unsigned long shrink_list(enum lru_list lru, unsigned long nr_to_scan,
2664                                struct lruvec *lruvec, struct scan_control *
2665                                sc)
2666 {
2667     if (is_active_lru(lru)) {
2668         if (sc->may_deactivate & (1 << is_file_lru(lru)))
2669             shrink_active_list(nr_to_scan, lruvec, sc, lru);
2670         else
2671             sc->skipped_deactivate = 1;
2672         return 0;
2673     }

```



```

2673
2674     return shrink_inactive_list(nr_to_scan, lruvec, sc, lru);
2675 }

```

Listing 11-55: mm/vmscan.c: *shrink_list()*

The `shrink_list()` function is rather straightforward—if the LRU is an active one, whether file-backed or anonymous (as checked by `is_active_lru()`) then we check whether we are permitted to perform folio deactivation via the struct `scan_control->may_deactivate` field, offsetting to `DEACTIVATE_ANON` or `DEACTIVATE_FILE` appropriately using `is_file_lru()` to determine if this LRU is file-backed or not.

If so, then we shrink the active list, i.e. deactivate folios within the LRU vector, via `shrink_active_list()` (see Listing 11-56) as discussed in Section 11.5.7.

If we are not permitted to do so, we set the struct `scan_control->skipped_deactivate` field so, if direct reclaim invoked this reclaim, the `do_try_to_free_pages()` function (see Listing 11-20 and Section 11.3) can choose to force deactivation as required.

If this is an inactive LRU, then we perform inactive list shrinking, in other words reclaim via `shrink_inactive_list()` (see Listing 11-61) which we explore in Section 11.5.9.

11.5.7 Shrinking the Active List

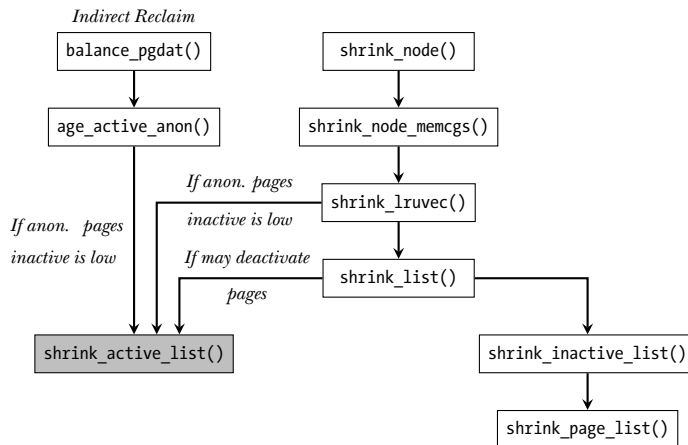


Figure 11-20: Reclaim Mechanism: Shrinking the Active List Code Path

As described in Section 11.2, and in more detail in Section 11.5.9 there is an active LRU list and an inactive LRU list. Folios typically start in the inactive list, then on reclaim they are checked to see if they've been touched since we last checked (via the hardware-implemented but software resettable accessed page table flag). If so, then they are marked referenced and when encountered again, if accessed once more, promoted to the active list.

Demotion from the active list typically occurs when either thrashing is occurring or the inactive list becomes small enough that it makes sense to transfer from the active list to the inactive one.

In general reclaim is more inclined to consistently scan file-backed folios, and thus invoke `shrink_list()` (see Listing 11-55) with `struct scan_control->may_deactivate` having the `DEACTIVATE_FILE` flag set.

However anonymous folios might not be so consistently processed, so we must make extra effort to ensure we consistently deactivate folios, so we proactively move folios from the active anonymous list to inactive, that is deactivate them on indirect reclaim (see Section 11.4.4) and when performing LRU vector shrinking (see Section 11.5.5).

In all cases, this deactivation only occurs if thrashing is occurring or if the inactive list is too small to maintain the desired balance between inactive and active lists as determined by `inactive_is_low()` (see Listing 11-42 and Section 11.5.3).

The function which moves folios from the active list to the inactive one is `shrink_active_list()`, which we explore in Listing 11-56 (eliding out of scope statistical updates, realtime kernel scheduling logic, buffer head handling, cgroup handling, and tracing hooks).

```

2492 /*
2493  * shrink_active_list() moves folios from the active LRU to the inactive LRU.
2494  *
2495  * We move them the other way if the folio is referenced by one or more
2496  * processes.
2497  *
2498  * If the folios are mostly unmapped, the processing is fast and it is
2499  * appropriate to hold lru_lock across the whole operation. But if
2500  * the folios are mapped, the processing is slow (folio_referenced()), so
2501  * we should drop lru_lock around each folio. It's impossible to balance
2502  * this, so instead we remove the folios from the LRU while processing them.
2503  * It is safe to rely on the active flag against the non-LRU folios in here
2504  * because nobody will play with that bit on a non-LRU folio.
2505  *
2506  * The downside is that we have to touch folio->refcount against each folio.
2507  * But we had to alter folio->flags anyway.
2508  */
2509 static void shrink_active_list(unsigned long nr_to_scan,
2510                               struct lruvec *lruvec,
2511                               struct scan_control *sc,
2512                               enum lru_list lru)
2513 {
2514     unsigned long nr_taken;
2515     unsigned long nr_scanned;
2516     unsigned long vm_flags;
2517     LIST_HEAD(l_hold);      /* The folios which were snipped off */
2518     LIST_HEAD(l_active);
2519     LIST_HEAD(l_inactive);

```

```

2520     unsigned nr_deactivate, nr_activate;
2521     unsigned nr_rotated = 0;
2522     int file = is_file_lru(lru);
2523     struct pglist_data *pgdat = lruvec_pgdat(lruvec);
2524
2525     lru_add_drain();
2526
2527     spin_lock_irq(&lruvec->lru_lock);
2528
2529     nr_taken = isolate_lru_pages(nr_to_scan, lruvec, &l_hold,
2530                                &nr_scanned, sc, lru);
2531     :
2532
2533     spin_unlock_irq(&lruvec->lru_lock);
2534
2535     while (!list_empty(&l_hold)) {
2536         struct folio *folio;
2537         :
2538
2539         folio = lru_to_folio(&l_hold);
2540         list_del(&folio->lru);
2541
2542         if (unlikely(!folio_evictable(folio))) {
2543             folio_putback_lru(folio);
2544             continue;
2545         }
2546         :
2547
2548         /* Referenced or rmap lock contention: rotate */
2549         if (folio_referenced(folio, 0, sc->target_mem_cgroup,
2550                             &vm_flags) != 0) {
2551             /*
2552              * Identify referenced, file-backed active folios and
2553              * give them one more trip around the active list. So
2554              * that executable code get better chances to stay in
2555              * memory under moderate memory pressure. Anon folios
2556              * are not likely to be evicted by use-once streaming
2557              * IO, plus JVM can create lots of anon VM_EXEC folios
2558              *
2559              ,
2560              * so we ignore them here.
2561              */
2562             if ((vm_flags & VM_EXEC) && folio_is_file_lru(folio))
2563             {
2564                 nr_rotated += folio_nr_pages(folio);
2565                 list_add(&folio->lru, &l_active);
2566                 continue;
2567             }
2568         }
2569
2570         folio_clear_active(folio); /* we are de-activating */

```

```

2580         folio_set_workingset(folio);
2581         list_add(&folio->lru, &l_inactive);
2582     }
2583
2584     /*
2585      * Move folios back to the lru list.
2586      */
2587     spin_lock_irq(&lruvec->lru_lock);
2588
2589     nr_activate = move_pages_to_lru(lruvec, &l_active);
2590     nr_deactivate = move_pages_to_lru(lruvec, &l_inactive);
2591     /* Keep all free folios in l_active list */
2592     list_splice(&l_inactive, &l_active);
2593     :
2598     spin_unlock_irq(&lruvec->lru_lock);
2599     :
2601     free_unref_page_list(&l_active);
2602     :
2604 }

```

Listing 11-56: mm/vmscan.c: shrink_active_list()

The `shrink_active_list()` function maintains three lists—`l_hold` which contains folios we have isolated from their LRU list, `l_active` which contains folios we intend to put back on to the active list and `l_inactive` which contains folios we intend to deactivate by placing at the end of the inactive list.

We start by draining any batched folios which have been cached to be placed into LRU vectors when each batch becomes full (see Section 11.7 for more on folio batches) via `lru_add_drain()` (See Listing 11-110 and Section 11.7.12).

After this we acquire the lock around struct `lruvec`. This data structure is, when cgroups aren't in effect, a per-node one which contains each individual LRU vector. See Section 11.2.1 for more on this data structure.

NOTE

The struct `lruvec->lru_lock` lock is very heavily contended, so the kernel goes to great lengths to try to minimise the time over which this lock is held.

With the LRU lock acquired, we isolated LRU folios from the list and place them into the `l_hold` list, attempting to scan `nr_to_scan` pages, placing the actual number scanned into `nr_scanned`.

We do this via `isolate_lru_pages()` which we examine in Listing 11-57 and Section 11.5.8. This also increments each isolated folio's reference count, pinning it in memory.

The function returns the number of base pages contained within the isolated folios, which we store in `nr_taken`.

With the folios isolated we very quickly release the lock and traverse the `l_hold` list.

We iterate through the list backwards as does reclaim itself, by looking at the tail of the list and obtaining that folio via `lru_to_folio()`.

Each folio that was added to the isolated folio list will have its `struct folio->lru` list node field reference its position within `l_hold`.

We therefore remove the folio from this list via `list_del()`, ready to place it into whichever list we deem appropriate.

We then take care to check that the folio is not `mlock()`ed or otherwise marked as unevictable. If so, we abort the operation and place the folio back on the appropriate LRU list via `folio_putback_lru()` (this ultimately invokes `folio_add_lru()`, see Listing 11-88 and Section 11.7.5, it also drops the folio's raised reference count).

We consider an edge case—if a folio has been accessed since we last checked in any mapping, as determined by the hardware page table accessed flag (hardware set, software cleared), determined by the reverse mapping function `folio_referenced()` (see Chapter 7), and the folio is file-backed and maps executable code, we simply maintain it in the active list indefinitely, add this folio to the `l_active` list.

In all other cases, we clear the `PG_active` flag, set the `PG_workingset` flag to indicate that this folio was once active (see Section 11.5.2 for a brief overview of the working set), and add the folio to the `l_inactive` list.

With the loop complete, we again acquire the `struct lruvec->lru_lock` lock as we modify the LRU vectors, moving each folio to the appropriate list via `move_pages_to_lru()` (see Listing 11-60 and Section 11.5.8 for an exploration of this). We do this separately for both inactive and active folios.

We then handle an odd wrinkle in the way that `move_pages_to_lru()` operates—if, when dropping a folio's reference count that isolation provided, we find we should now free the folio, we batch this up, with `move_pages_to_lru()` placing folios to be freed in the input list that it is provided.

We combine these two lists together via `list_splice()` into `l_active`, which we then submit to `free_unref_page_list()` to free the folios to the physical allocator (see Chapter 2 for details on the physical memory allocator).

NOTE

Importantly, we see that folios are simply taken from the active list into the inactive one without prejudice (aside from the `VM_EXEC` edge case), there is no decision making performed—once the active list is to be cleared down, those folios are placed into the inactive one and subject to potential reclaim.

11.5.8 Isolating LRU Folios

We divide the operation of shrinking an LRU list into isolating folios from the list, taking care to ensure that they are stable and have not changed beneath us as we do so, and actually shrinking them.

The isolation of LRU folios is performed by `isolate_lru_pages()` which we examine starting in Listing 11-57 (eliding out of scope data prefetch logic, trace hooks, and statistic recording).

This function assumes that the `struct lruvec->lru_lock` is held.

```
2117 /*
2118  * Isolating page from the lruvec to fill in @dst list by nr_to_scan times.
```

```

2119 *
2120 * lruvec->lru_lock is heavily contended. Some of the functions that
2121 * shrink the lists perform better by taking out a batch of pages
2122 * and working on them outside the LRU lock.
2123 *
2124 * For pagecache intensive workloads, this function is the hottest
2125 * spot in the kernel (apart from copy_*_user functions).
2126 *
2127 * Lru_lock must be held before calling this function.
2128 *
2129 * @nr_to_scan: The number of eligible pages to look through on the list.
2130 * @lruvec:     The LRU vector to pull pages from.
2131 * @dst:        The temp list to put pages on to.
2132 * @nr_scanned: The number of pages that were scanned.
2133 * @sc:         The scan_control struct for this reclaim session
2134 * @lru:        LRU list id for isolating
2135 *
2136 * returns how many pages were moved onto *@dst.
2137 */
2138 static unsigned long isolate_lru_pages(unsigned long nr_to_scan,
2139                                       struct lruvec *lruvec, struct list_head *dst,
2140                                       unsigned long *nr_scanned, struct scan_control *sc,
2141                                       enum lru_list lru)
2142 {
2143     struct list_head *src = &lruvec->lists[lru];
2144     unsigned long nr_taken = 0;
2145     :
2146     unsigned long scan, total_scan, nr_pages;
2147     LIST_HEAD(folios_skipped);
2148
2149     total_scan = 0;
2150     scan = 0;

```

Listing 11-57: mm/vmscan.c: isolate_lru_pages() Preface

We start `isolate_lru_pages()` by establishing local variables, including the `folios_skipped` list which stores folios which we choose to skip for various reasons.

We set the `src` list to the `lru` vector from which we are isolating as determined by the input `enum lru_list` which specifies whether this is the `struct lruvec`'s inactive anonymous, active anonymous, inactive file-backed or active file-backed LRU list.

With this done, we loop over folios contained in this list, as explored in Listing 11-58.

```

2153     while (scan < nr_to_scan && !list_empty(src)) {
2154         struct list_head *move_to = src;
2155         struct folio *folio;
2156

```

```

2157         folio = lru_to_folio(src);
2158         :
2160         nr_pages = folio_nr_pages(folio);
2161         total_scan += nr_pages;
2162
2163         if (folio_zonenum(folio) > sc->reclaim_idx) {
2164             :
2165             move_to = &folios_skipped;
2166             goto move;
2167         }
2168
2169         /*
2170          * Do not count skipped folios because that makes the function
2171          * return with no isolated folios if the LRU mostly contains
2172          * ineligible folios. This causes the VM to not reclaim any
2173          * folios, triggering a premature OOM.
2174          * Account all pages in a folio.
2175          */
2176         scan += nr_pages;
2177
2178         if (!folio_test_lru(folio))
2179             goto move;
2180         if (!sc->may_unmap && folio_mapped(folio))
2181             goto move;
2182
2183         /*
2184          * Be careful not to clear the lru flag until after we're
2185          * sure the folio is not being freed elsewhere -- the
2186          * folio release code relies on it.
2187          */
2188         if (unlikely(!folio_try_get(folio)))
2189             goto move;
2190
2191         if (!folio_test_clear_lru(folio)) {
2192             /* Another thread is already isolating this folio */
2193             folio_put(folio);
2194             goto move;
2195         }
2196
2197         nr_taken += nr_pages;
2198         :
2199         move_to = dst;
2200 move:
2201         list_move(&folio->lru, move_to);
2202     }

```

Listing 11-58: mm/vmscan.c: *isolate_lru_pages()* Main Loop

In the portion of `isolate_lru_pages()` explored in Listing 11-58 we iterate over folios in the input LRU vector, extracting the tail item in the list via `lru_to_folio()`.

NOTE

As a reminder—folios are added to LRU linked lists at the head of the list, and read (by reclaim) from the tail. Therefore, when we read the folio from the tail of the LRU list as we isolate this is in line with how these lists are expected to function.

We have not yet extracted the folio from the LRU list, and go about this carefully. Firstly we determine the number of base pages contained within the folio via `folio_nr_pages()`, which we add to the running total of scanned folios, `total_scan`.

Note that we maintain `scan` as a count of non-skipped folios, and it is this we compare to `nr_to_scan` within the loop, so that we do not end up scanning ineligible folios only to isolate nothing.

Where each folio ends up depends on the `move_to` variable, which we initialise to `src`, i.e. simply placing the folio back onto its original list.

A folio is skipped if the folio exists in a greater than the maximum zone from which the failing allocation which triggered reclaim can allocate from (See Section 2.4 and Chapter 2 for a detailed discussion of nodes and zones), as stored in `struct scan_control->reclaim_idx`.

Other instances of us scanning but failing to isolate a folio do not constitute that folio being skipped, so in these instances we account these folios as having been scanned but not isolated.

If we find the folio has for whatever reason already been taken from an LRU, i.e. the `PG_lru` flag is not set, we move the folio back to the head of the input list via `list_move()`.

NOTE

These list operations are safe as it is assumed within this function that we hold the `struct lruvec->lru_lock`.

Equally if we are unable to unmap the folio, as determined by `struct scan_control->may_unmap`, and the folio has an elevated map count as determined by `folio_mapped()`, then we equally move it back to the source list.

With these edge cases considered, we try to increment the folio's reference count via `folio_try_get()`. If there is contention on the folio lock we do not wait for it, instead we do not isolate it.

Finally, if after the folio's reference count has been incremented we find it is no longer marked as being on an LRU (via the `PG_lru` flag) as we attempt to clear it, then we assume another thread is isolating this folio, drop the reference count and abort, placing the folio back on its original LRU list.

With these cases ruled out, we increment `nr_taken` which maintains a count of isolated folios for us to return to caller, and sets `move_to` to the caller-specified destination list.

We examine what we do after this loop in Listing 11-59.

```

2204      /*
2205      * Splice any skipped folios to the start of the LRU list. Note that
2206      * this disrupts the LRU order when reclaiming for lower zones but

```



```

2207      * we cannot splice to the tail. If we did then the SWAP_CLUSTER_MAX
2208      * scanning would soon rescan the same folios to skip and waste lots
2209      * of cpu cycles.
2210      */
2211      if (!list_empty(&folios_skipped)) {
2212          :
2213          :
2214          list_splice(&folios_skipped, src);
2215          :
2216      }
2217      *nr_scanned = total_scan;
2218      :
2219      return nr_taken;
2220 }

```

Listing 11-59: mm/vmscan.c: isolate_lru_pages() Suffix

The portion of `isolate_lru_pages()` examined in Listing 11-59 first places the `folios_skipped` folios back onto the source list, putting them to the back of the queue (by placing them at the head of the list), in order that we don't repeatedly scan the same skipped folios while iterating through `SWAP_CLUSTER_MAX` batches of folios.

Finally we output the total number of pages scanned (including those skipped) to the output parameter `nr_scanned` and return the number of isolated pages.

When isolating folios, these are maintained in a local list, and later need to be put back to whichever list they ought to be on. This is performed by `move_pages_to_lru()` which we examine in Listing 11-60 (eliding debug checks and out of scope working set hooks and huge page handling).

This function also assumes that `struct lruvec->lru_lock` is held.

```

2317 /*
2318  * move_pages_to_lru() moves folios from private @list to appropriate LRU list
2319  *
2320  * On return, @list is reused as a list of folios to be freed by the caller.
2321  *
2322  * Returns the number of pages moved to the given lruvec.
2323  */
2324 static unsigned int move_pages_to_lru(struct lruvec *lruvec,
2325                                     struct list_head *list)
2326 {
2327     int nr_pages, nr_moved = 0;
2328     LIST_HEAD(folios_to_free);
2329
2330     while (!list_empty(list)) {
2331         struct folio *folio = lru_to_folio(list);
2332         :
2333         list_del(&folio->lru);
2334         if (unlikely(!folio_evictable(folio))) {
2335             spin_unlock_irq(&lruvec->lru_lock);

```

```

2336         folio_putback_lru(folio);
2337         spin_lock_irq(&lruvec->lru_lock);
2338         continue;
2339     }
2340
2341     /*
2342      * The folio_set_lru needs to be kept here for list integrity.
2343      * Otherwise:
2344      * #0 move_pages_to_lru          #1 release_pages
2345      * if (!folio_put_testzero())
2346      *                                if (folio_put_testzero())
2347      *                                !lru //skip lru_lock
2348      * folio_set_lru()
2349      * list_add(&folio->lru,)
2350      *                                list_add(&folio->lru
2351      ,)
2352      */
2353     folio_set_lru(folio);
2354
2355     if (unlikely(folio_put_testzero(folio))) {
2356         __folio_clear_lru_flags(folio);
2357         :
2358         list_add(&folio->lru, &folios_to_free);
2359         :
2360         continue;
2361     }
2362     :
2363     lruvec_add_folio(lruvec, folio);
2364     nr_pages = folio_nr_pages(folio);
2365     nr_moved += nr_pages;
2366     :
2367 }
2368
2369 /*
2370  * To save our caller's stack, now use input list for pages to free.
2371  */
2372 list_splice(&folios_to_free, list);
2373
2374 return nr_moved;
2375 }

```

Listing 11-60: mm/vmscan.c: `move_pages_to_lru()`

The `move_pages_to_lru()` function iterates through the list from the tail, extracting the folio in each instance via `lru_to_folio()` and deleting the folio from the input list via `list_del()`.

As in `shrink_active_list()`, we check to ensure the folio is not `mlock()`ed or otherwise marked as unevictable via `folio_evictable()`, if it is, we place the folio back on the appropriate LRU list via `folio_putback_lru()` (this ultimately invokes `folio_add_lru()`, see Listing 11-88 and Section 11.7.5, it also drops the folio's raised reference count).

With this check performed, we set the `PG_lru` flag (note the nuanced timing in how we do this as described in the comment), before dropping the isolated folio's reference count.

We now account for the case where the elevated reference count from the folio being isolated was the last remaining reference count. We want to try to batch cases like this so we can delete them all in one go rather than one at a time, and thus add these folios to the `folios_to_free` list, having cleared its LRU-specific `PG_lru`, `PG_active` and `PG_unevictable` folio flags before skipping to the next folio.

Otherwise, we add the folio to the designated LRU list via `lruvec_add_folio()` (see Listing 11-13 and Section 11.2.2).

We then count the base pages we have moved via `folio_nr_pages()` which we accumulate in `nr_moved`.

Finally, we place all of the folios to be freed on the input list for the caller to free (done this way for efficiency), before returning the count of base pages of folios moved.

11.5.9 Shrinking the Inactive List

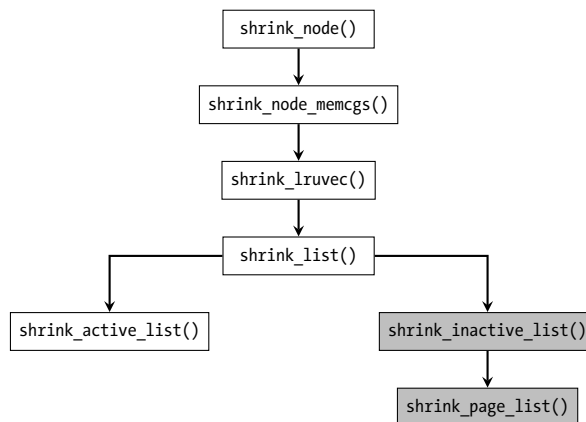


Figure 11-21: Reclaim Mechanism: Shrinking the Inactive List

Shrinking the inactive LRU list is where actual reclaim is performed within the kernel, and all of the remainder of the reclaim logic is simply designed to control how this proceeds.

This occurs in `shrink_inactive_list()`, which we examine in Listing 11-61 (eliding out of scope cgroup logic, statistical updates and trace hooks).

```

2401 static unsigned long
2402 shrink_inactive_list(unsigned long nr_to_scan, struct lruvec *lruvec,
```

```

2403         struct scan_control *sc, enum lru_list lru)
2404     {
2405         LIST_HEAD(page_list);
2406         unsigned long nr_scanned;
2407         unsigned int nr_reclaimed = 0;
2408         unsigned long nr_taken;
2409         struct reclaim_stat stat;
2410         bool file = is_file_lru(lru);
2411         enum vm_event_item item;
2412         struct pglist_data *pgdat = lruvec_pgdat(lruvec);
2413         bool stalled = false;
2414
2415         while (unlikely(too_many_isolated(pgdat, file, sc))) {
2416             if (stalled)
2417                 return 0;
2418
2419             /* wait a bit for the reclaimer. */
2420             stalled = true;
2421             reclaim_throttle(pgdat, VMSCAN_THROTTLE_ISOLATED);
2422
2423             /* We are about to die and free our memory. Return now. */
2424             if (fatal_signal_pending(current))
2425                 return SWAP_CLUSTER_MAX;
2426         }
2427
2428         lru_add_drain();
2429
2430         spin_lock_irq(&lruvec->lru_lock);
2431
2432         nr_taken = isolate_lru_pages(nr_to_scan, lruvec, &page_list,
2433                                     &nr_scanned, sc, lru);
2434         :
2442         spin_unlock_irq(&lruvec->lru_lock);
2443
2444         if (nr_taken == 0)
2445             return 0;
2446
2447         nr_reclaimed = shrink_page_list(&page_list, pgdat, sc, &stat, false);
2448
2449         spin_lock_irq(&lruvec->lru_lock);
2450         move_pages_to_lru(lruvec, &page_list);
2451         :
2458         spin_unlock_irq(&lruvec->lru_lock);
2459
2460         lru_note_cost(lruvec, file, stat.nr_pageout);
2461         :
2461         free_unref_page_list(&page_list);

```

```

2462
2463     /*
2464      * If dirty pages are scanned that are not queued for IO, it
2465      * implies that flushers are not doing their job. This can
2466      * happen when memory pressure pushes dirty pages to the end of
2467      * the LRU before the dirty limits are breached and the dirty
2468      * data has expired. It can also happen when the proportion of
2469      * dirty pages grows not through writes but through memory
2470      * pressure reclaiming all the clean cache. And in some cases,
2471      * the flushers simply cannot keep up with the allocation
2472      * rate. Nudge the flusher threads in case they are asleep.
2473      */
2474     if (stat.nr_unqueued_dirty == nr_taken)
2475         wakeup_flusher_threads(WB_REASON_VMSCAN);
2476
2477     sc->nr.dirty += stat.nr_dirty;
2478     sc->nr.congested += stat.nr_congested;
2479     sc->nr.unqueued_dirty += stat.nr_unqueued_dirty;
2480     sc->nr.writeback += stat.nr_writeback;
2481     sc->nr.immediate += stat.nr_immediate;
2482     sc->nr.taken += nr_taken;
2483     if (file)
2484         sc->nr.file_taken += nr_taken;
2485     :
2486     return nr_reclaimed;
2487 }
2490

```

Listing 11-61: mm/vmscan.c: *shrink_inactive_list()*

We start *shrink_inactive_list()* by initialising a number of local variables including *page_list* which we use to store isolated folios we wish to consider for reclaim.

We then check to see whether we need to throttle reclaim—if we are in a state where a large number of tasks are trying to allocate under heavy memory pressure, each isolating a great many folios to consider for reclaim, which could result in the LRU being shrunk to too small a size to be effectively scanned.

In this instance, we perform reclaim throttling (see Section 11.6). We determine whether this is necessary via *too_many_isolated()* which we examine in Listing 11-62 (eliding out of scope *cgroup* logic).

```

2273     /*
2274      * A direct reclaimer may isolate SWAP_CLUSTER_MAX pages from the LRU list and
2275      * then get rescheduled. When there are massive number of tasks doing page
2276      * allocation, such sleeping direct reclaimers may keep piling up on each CPU,
2277      * the LRU list will go small and be scanned faster than necessary, leading to
2278      * unnecessary swapping, thrashing and OOM.
2279      */
2280     static int too_many_isolated(struct pglist_data *pgdat, int file,

```

```

2281             struct scan_control *sc)
2282 {
2283     unsigned long inactive, isolated;
2284     bool too_many;
2285
2286     if (current_is_kswapd())
2287         return 0;
2288     :
2292     if (file) {
2293         inactive = node_page_state(pgdat, NR_INACTIVE_FILE);
2294         isolated = node_page_state(pgdat, NR_ISOLATED_FILE);
2295     } else {
2296         inactive = node_page_state(pgdat, NR_INACTIVE_ANON);
2297         isolated = node_page_state(pgdat, NR_ISOLATED_ANON);
2298     }
2299
2300     /*
2301     * GFP_NOIO/GFP_NOFS callers are allowed to isolate more pages, so
2302     they
2303     * won't get blocked by normal direct-reclaimers, forming a circular
2304     * deadlock.
2305     */
2306     if ((sc->gfp_mask & (__GFP_IO | __GFP_FS)) == (__GFP_IO | __GFP_FS))
2307         inactive >>= 3;
2308
2309     too_many = isolated > inactive;
2310
2311     /* Wake up tasks throttled due to too_many_isolated. */
2312     if (!too_many)
2313         wake_throttle_isolated(pgdat);
2314
2315     return too_many;
2316 }

```

Listing 11-62: mm/vmscan.c: `too_many_isolated()`

The `too_many_isolated()` function only considers direct reclaim cases, as background reclaim can't scale in a pathological way like direct reclaim can.

We count how many inactive pages exist in LRUs and compare this to the number of total isolated pages. If isolated pages exceeds one-eighth of the number of inactive pages, then we indicate that there are too many isolated pages.

NOTE

In the case of GFP_NOIO or GFP_NOFS allocations where I/O or filesystem access are not permitted in the failing allocation which triggered reclaim, we allow isolated pages to grow all the way to the size of the number of inactive ones to avoid deadlock.

If we determine that we do not have too many isolated pages we wake any threads that were throttled on this via `wake_throttle_isolated()`.

Returning to `shrink_inactive_list()` and Listing 11-61, we see that in the case of `too_many_isolated()` indicating we do have too many isolated folios, we engage in reclaim throttle via `reclaim_throttle()` setting the `VMSCAN_THROTTLE_ISOLATED` flag (see Listing 11-75 in Section 11.6).

We exit early if a fatal signal emerged in the time we spent sleeping, otherwise we check again, exiting the function indicating failure if this recurs.

With this checked, we ensure that any folio batches containing folios that ought to be on an LRU but are not flushed to them yet are immediately drained to the LRUs via `lru_add_drain()` (See Listing 11-110 and Section 11.7.12).

We acquire the struct `lruvec->lru_lock` and isolate LRU folios for us to examine via `isolate_lru_pages()` (see Listing 11-57 and Section 11.5.8), releasing the lock afterwards, exiting indicating failure if this fails to isolate any folios.

With folios isolated and knowing we are able to proceed with reclaim, we do so by invoking `shrink_page_list()`, which is the heart of the reclaim mechanism, and which we examine in Listing 11-63 shortly.

This function returns the number of base pages which were reclaimed storing this in `nr_reclaimed`.

Any folios which were not reclaimed will remain on `page_list`, so we then need to move these to the appropriate LRU lists, which we do, under struct `lruvec->lru_lock`, via `move_pages_to_lru()` (see Listing 11-60 and Section 11.5.8).

After this is done we invoke `lru_note_cost()` to account for any dirty folios successfully paged out to disk. See Listing 11-39 and Section 11.5.2.1 for more details on this.

With this complete, we free any folios that `move_pages_to_lru()` indicated should be freed (again see Listing 11-60 and Section 11.5.8 for details) via `free_unref_page_list()`.

With this complete, we wake up writeback flusher threads if it seems too many dirty folios are not being subject to writeback via `wakeup_flusher_threads()` (see Listing 10-35 and Section 10.11 for details on this in Chapter 10), and update struct `scan_control` statistics before returning the number of reclaimed base pages.

We examine the logic of the key `shrink_page_list()` function in Figure 11-22.

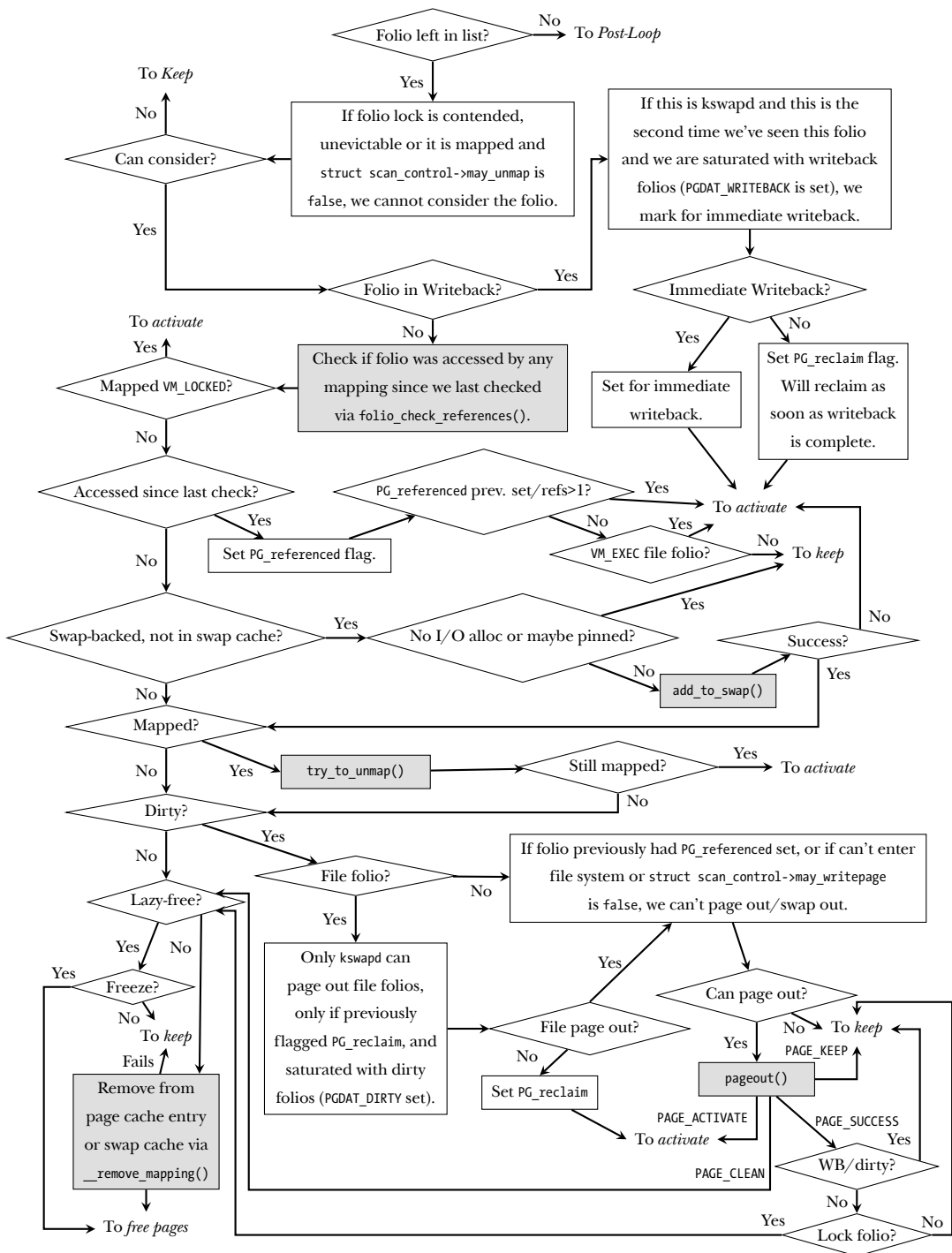


Figure 11-22: *shrink_page_list()* Main Loop Logic

We examine `shrink_page_list()` in Listing 11-63 (eliding out of scope real-time scheduling hooks, cgroup logic, debug asserts, huge page logic, NUMA demotion, statistical updates, buffer head freeing and hot plugging support).

```

1589 static unsigned int shrink_page_list(struct list_head *page_list,
1590                                     struct pglist_data *pgdat,
1591                                     struct scan_control *sc,
1592                                     struct reclaim_stat *stat,
1593                                     bool ignore_references)
1594 {
1595     LIST_HEAD(ret_pages);
1596     LIST_HEAD(free_pages);
1597     :
1598     unsigned int nr_reclaimed = 0;
1599     :
1603     memset(stat, 0, sizeof(*stat));
1604     :
1607 retry:
1608     while (!list_empty(page_list)) {
1609         struct address_space *mapping;
1610         struct folio *folio;
1611         enum page_references references = PAGEREF_RECLAIM;
1612         bool dirty, writeback;
1613         unsigned int nr_pages;
1614         :
1617         folio = lru_to_folio(page_list);
1618         list_del(&folio->lru);
1619
1620         if (!folio_trylock(folio))
1621             goto keep;
1622         :
1625         nr_pages = folio_nr_pages(folio);
1626
1627         /* Account the number of base pages */
1628         sc->nr_scanned += nr_pages;
1629
1630         if (unlikely(!folio_evictable(folio)))
1631             goto activate_locked;
1632
1633         if (!sc->may_unmap && folio_mapped(folio))
1634             goto keep_locked;

```

Listing 11-63: mm/vmscan.c: shrink_page_list() Preface & Loop Start

We start by initialising local state, including the lists `ret_pages` which are the folios we will return in `page_list` for the caller to place into their appropriate LRU list should we not reclaim them and `free_pages` for those folios we

were able to reclaim and thus are ready to be freed, which we bulk free at the end of the operation.

Within the loop we read a folio from the tail of the input list of isolated folios via `lru_to_folio()` and delete the folio from the input list via `list_del()`.

NOTE

We later append (that is splice) the empty input list at the end of the iteration with folios that were not reclaimed that need to be returned to the caller to put back on the appropriate LRUs.

On each iteration we optimistically try to lock the folio—we don’t want to block when we have other folios we can consider for reclaim. If we can’t lock we keep the folio rotating it to the back of the list (we examine each of these labels in Listing 11-69).

If the folio can’t be evicted (`mlock()`ed or otherwise marked as unevictable as checked by `folio_evictable()`), then we activate it so it gets out of the way of folios that can be reclaimed.

Finally if the folio is mapped by userland but `struct scan_control->may_unmap` is false then we are not permitted to unmap, and so cannot reclaim it and thus keep it rotating it to the back of the list.

After ensuring that we can proceed, we examine writeback folios, which we examine in Listing 11-64.

```

1636      /*
1637       * The number of dirty pages determines if a node is marked
1638       * reclaim_congested. kswapd will stall and start writing
1639       * folios if the tail of the LRU is all dirty unqueued folios.
1640       */
1641      folio_check_dirty_writeback(folio, &dirty, &writeback);
1642      if (dirty || writeback)
1643          stat->nr_dirty += nr_pages;
1644
1645      if (dirty && !writeback)
1646          stat->nr_unqueued_dirty += nr_pages;
1647
1648      /*
1649       * Treat this folio as congested if folios are cycling
1650       * through the LRU so quickly that the folios marked
1651       * for immediate reclaim are making it to the end of
1652       * the LRU a second time.
1653       */
1654      if (writeback && folio_test_reclaim(folio))
1655          stat->nr_congested += nr_pages;
1656      :
1670      if (folio_test_writeback(folio)) {
1671          :
1672          if (current_is_kswapd() &&
1673              folio_test_reclaim(folio) &&

```

```

1705         test_bit(PGDAT_WRITEBACK, &pgdat->flags)) {
1706             stat->nr_immediate += nr_pages;
1707             goto activate_locked;
1708         :
1710     } else if (writeback_throttling_sane(sc) ||
1711         :
1712     /*
1713         * This is slightly racy -
1714         * folio_end_writeback() might have
1715         * just cleared the reclaim flag, then
1716         * setting the reclaim flag here ends up
1717         * interpreted as the readahead flag - but
1718         * that does not matter enough to care.
1719         * What we do want is for this folio to
1720         * have the reclaim flag set next time
1721         * memcg reclaim reaches the tests above,
1722         * so it will then wait for writeback to
1723         * avoid OOM; and it's also appropriate
1724         * in global reclaim.
1725         */
1726         folio_set_reclaim(folio);
1727         stat->nr_writeback += nr_pages;
1728         goto activate_locked;
1729     :
1738     }
1739 }

```

Listing 11-64: mm/vmscan.c: shrink_page_list() Writeback Handling

We start the portion of shrink_page_list() explored in Listing 11-64 by determining whether the folio is file-backed and either dirty or in a state of writeback via folio_check_dirty_writeback().

If the file-system supports it, folio_check_dirty_writeback() exposes a hook to allow for custom handling of this. We then populate the output in dirty and writeback.

When a folio is under writeback and we want to reclaim it, we typically set the PG_reclaim, this is seen in the branch starting at line 1710.

NOTE

The source is unfortunately cropped rather awkwardly here—writeback_throttling_sane() is a cgroup-specific check, which if cgroups are not enabled returns true. Therefore from the purpose of what is in scope here in the book, view this as an else branch.

This is what we do in the majority of cases, as when writeback completes, folio_end_writeback() is invoked, which tests for PG_reclaim, and if set, clears this flag and invokes folio_rotate_reclaimable() (see Listing 11-92 and Section 11.7.7)).

This function places the folio at the tail of the inactive LRU, so it will be considered for reclaim as soon as there is memory pressure, and is therefore reclaimed right away.

However, if this is indirect reclaim (as checked for by `current_is_kswapd()`), the `PG_reclaim` has already been set but the folio has not been written back yet and we previously determined that all of the isolated folios were in writeback at once (checked in `shrink_node()`, see Listing 11-40 and Section 11.5.3), having set the node flag `PGDAT_WRITEBACK`, then we increment the `nr_immediate` field in the struct `reclaim_state` field.

When `shrink_node()` (see Listing 11-40) sees that struct `reclaim_state->nr_immediate` is non-zero, then it causes reclaim throttling to allow for the writeback to complete.

In this instance we also activate the folio to get it out of the way, which allows `folio_rotate_reclaimable()` (see Listing 11-92) to move it to the front of the inactive LRU.

When `folio_rotate_reclaimable()` performs rotation it places the folio into a folio batch (see Section 11.7 for more on folio batches) so we can move folios in batches at a time for efficiency (note that reclaim drains these non-LRU batches so nothing gets missed).

The `lru_move_tail_fn()` function is eventually invoked which clears the `PG_active` flag before moving the folio to the tail of the inactive LRU vector.

With writeback folios handled, we examine how we deal with detecting whether folios have been utilised, that is referenced by user space, which we examine in Listing 11-65.

```

1741         if (!ignore_references)
1742             references = folio_check_references(folio, sc);
1743
1744         switch (references) {
1745             case PAGeref_ACTIVATE:
1746                 goto activate_locked;
1747             case PAGeref_KEEP:
1748                 stat->nr_ref_keep += nr_pages;
1749                 goto keep_locked;
1750             case PAGeref_RECLAIM:
1751             case PAGeref_RECLAIM_CLEAN:
1752                 ; /* try to reclaim the folio below */
1753         }

```

Listing 11-65: mm/vmscan.c: shrink_page_list() Reference Handling

The portion of `shrink_page_list()` explored in Listing 11-65 we note that we entirely abstract folio reference handling in `folio_check_references()` which we examine in detail in Listing 11-70 and Section 11.5.10.

We only do so if `ignore_references` is not set—this is not set on ordinary reclaim so we can disregard this. But if it was set, note that `references` defaults to `PAGeref_RECLAIM`.

This function returns an enum `page_references` value indicating how to proceed, with `PAGeref_RECLAIM` indicating we can proceed with reclaim,

PAGEREF_RECLAIM_CLEAN indicating that we can proceed with reclaim but not if the folio is dirty, PAGEREF_KEEP indicating the the folio should be rotated to the end of the list it is currently on and PAGEREF_ACTIVATE indicating the folio should be added to the active LRU vector, i.e. activated.

We handle each of the cases in the switch statement invoked after the invocation of `folio_check_references()`.

With this complete, we handle swap and mapping logic shown in Listing 11-66.

Again, see Section 11.5.10 for a detailed exploration of what folio references entail and how they are determined.

```

1766      /*
1767       * Anonymous process memory has backing store?
1768       * Try to allocate it some swap space here.
1769       * Lazyfree folio could be freed directly
1770       */
1771      if (folio_test_anon(folio) && folio_test_swapbacked(folio)) {
1772          if (!folio_test_swapcache(folio)) {
1773              if (!(sc->gfp_mask & __GFP_IO))
1774                  goto keep_locked;
1775              if (folio_maybe_dma_pinned(folio))
1776                  goto keep_locked;
1777              :
1791              if (!add_to_swap(folio)) {
1792                  :
1793                  goto activate_locked_split;
1794              :
1803              }
1804          }
1805      :
1810      }
1811      :
1822      /*
1823       * The folio is mapped into the page tables of one or more
1824       * processes. Try to unmap it here.
1825       */
1826      if (folio_mapped(folio)) {
1827          enum ttu_flags flags = TTU_BATCH_FLUSH;
1828          bool was_swapbacked = folio_test_swapbacked(folio);
1829          :
1833          try_to_unmap(folio, flags);
1834          if (folio_mapped(folio)) {
1835              stat->nr_unmap_fail += nr_pages;
1836              if (!was_swapbacked &&
1837                  folio_test_swapbacked(folio))
1838                  stat->nr_lazyfree_fail += nr_pages;
1839              goto activate_locked;
1840          }

```

Listing 11-66: mm/vmscan.c: shrink_page_list() Add to Swap & Mappings

Importantly, we start the portion of `shrink_page_list()` explored in Listing 11-66 by checking to see whether we need to perform swap-out.

We explore this case in detail in the dedicated chapter on Swap (Chapter 12) in Section 12.2, however we'll reiterate the details here.

A folio is considered for swap out if it is both anonymous and has the `PG_swapbacked` flag set. If a folio is anonymous without the `PG_swapbacked` flag, this means it is a lazy-free folio, i.e. memory freed using `madvise()` with the `MADV_FREE` flag, a case we examine in Listing 11-68.

If the folio is not already in the swap cache, as indicated by the `PG_swapcache` flag, we consider it for adding to the swap cache.

We do not consider a folio for this if the fault which caused reclaim is unable to perform I/O, in which case we simply rotate the folio, equally so if the folio is pinned by the kernel via the Get User Pages (GUP) mechanism (see Section 8.1.2 and Chapter 8 for more on GUP).

We check for the GUP case via `folio_maybe_dma_pinned()` (see Listing 8-14 and Section 8.1.3 in Chapter 8), if this is so we simply rotate the folio.

Finally if can proceed with placing the folio in the swap cache, we do so via `add_to_swap()` (see Listing 12-20 and Section 12-20).

If this fails (for instance if there is no swap space left) we activate the folio, as we need to get it out of the way of folios we can reclaim, otherwise we are good to proceed forward. Note that ultimately a folio designated for swap out is marked dirty in order that it can be written back to disk. We examine dirty folio handling in Listing ??.

Next, if the folio is mapped in any userland mapping, then we need to unmap it entirely before performing reclaim.

We test for this via `folio_mapped()`, and utilise the reverse mapping function `try_to_unmap()` (see Listing 12-23 and Section ?? in Chapter 12) to do the leg work of iterating through all `struct vm_area_struct` (VMA) objects which map the folio (see Chapter 7 for details on the reverse mapping), unmapping it from them.

A nuance of this is with swap cache entries—the underlying function invoked by the reverse mapping iteration, `try_to_unmap_one()`, has special handling for swap cache entries, replacing them with swap entries rather than clearing them altogether.

See Listing 12-24 and Section ?? in Chapter 12 for details on how this is performed.

For file-backed folios, we simply clear the mapping—if the folio is referenced again, a fault will occur and the file-backed folio will be read back into memory from disk.

If, after invoking `try_to_unmap()`, the folio is still mapped, this mean the effort failed, and we update statistics accordingly and activate the folio to get it out of the way of reclaim.

With this complete, we examine dirty folio handling in Listing 11-67.

```

1844         if (folio_test_dirty(folio)) {
1845             /*
1846              * Only kswapd can writeback filesystem folios
1847              * to avoid risk of stack overflow. But avoid
1848              * injecting inefficient single-folio I/O into
1849              * flusher writeback as much as possible: only
1850              * write folios when we've encountered many
1851              * dirty folios, and when we've already scanned
1852              * the rest of the LRU for clean folios and see
1853              * the same dirty folios again (with the reclaim
1854              * flag set).
1855              */
1856             if (folio_is_file_lru(folio) &&
1857                 (!current_is_kswapd() ||
1858                  !folio_test_reclaim(folio) ||
1859                  !test_bit(PGDAT_DIRTY, &pgdat->flags))) {
1860                 :
1861                 folio_set_reclaim(folio);
1862
1863                 goto activate_locked;
1864             }
1865
1866             if (references == PAGEREF_RECLAIM_CLEAN)
1867                 goto keep_locked;
1868             if (!may_enter_fs(folio, sc->gfp_mask))
1869                 goto keep_locked;
1870             if (!sc->may_writepage)
1871                 goto keep_locked;
1872
1873             /*
1874              * Folio is dirty. Flush the TLB if a writable entry
1875              * potentially exists to avoid CPU writes after I/O
1876              * starts and then write it out here.
1877              */
1878             try_to_unmap_flush_dirty();
1879             switch (pageout(folio, mapping, &plug)) {
1880             case PAGE_KEEP:
1881                 goto keep_locked;
1882             case PAGE_ACTIVATE:
1883                 goto activate_locked;
1884             case PAGE_SUCCESS:
1885                 stat->nr_pageout += nr_pages;
1886
1887                 if (folio_test_writeback(folio))
1888                     goto keep;
1889                 if (folio_test_dirty(folio))
1890                     goto keep;
1891             }

```

```

1898
1899                                     /*
1900                                     * A synchronous write - probably a ramdisk.
1901
1902                                     * ahead and try to reclaim the folio.
1903                                     */
1904                                     if (!folio_trylock(folio))
1905                                         goto keep;
1906                                     if (folio_test_dirty(folio) ||
1907                                         folio_test_writeback(folio))
1908                                         goto keep_locked;
1909                                     mapping = folio_mapping(folio);
1910                                     fallthrough;
1911                                     case PAGE_CLEAN:
1912                                         ; /* try to free the folio below */
1913                                     }

```

Listing 11-67: mm/vmscan.c: shrink_page_list() Dirty Folio Handling

The portion of `shrink_page_list()` examined in Listing 11-67 handles dirty folios undergoing reclaim.

Only indirect reclaim handles dirty file-backed folios, to avoid kernel stacks getting too deep resulting in stack overflow, which in a kernel context is a fatal error and results in a kernel panic and the system halting.

This is because we cannot be sure the size of the kernel stack when direct reclaiming as a result of a page allocation, whereas the `kswapd` kernel process has well defined structure.

In addition, we only writeback dirty file-backed folios if it has not already been marked with `PG_reclaim` (we did not previously skip the folio before) and the `PGDAT_DIRTY` node flag has been set, indicating that all the folios examined were dirty and none had begun writeback.

Otherwise a file-backed folio has `PG_reclaim` set, ensuring that upon writeback it will be immediately scheduled for reclaim via `folio_rotate_reclaimable()` (see Listing 11-92 and Section 11.7.7)) which is invoked on writeback termination in `folio_end_writeback()` (see Chapter 10 for more on writeback as a whole).

This folio is then activated, in order to give it a chance to be written back before being considered for reclaim once again.

Next we check to see whether `folio_check_references()` (see Listing 11-70 and Section 11.5.10) indicated that the reclaim should be only if the folio is clean—that is, it was previously referenced. In this instance we keep the folio, rotating it to the back of the LRU vector.

If we can't enter the file system due to the Get Free Pages (GFP) flags associated with the allocation which triggered reclaim (see Section 2.6 in Chapter 2 for more on GFP flags), as checked by `may_enter_fs()`, or if the `struct scan_control->may_writepage` flag is cleared, then clearly writeback cannot proceed so we also rotate the folio in these cases.

We invoke `try_to_unmap_flush_dirty()` to ensure the TLB is flushed as necessary before we start trying to write I/O which might unnecessarily re-dirty the folio.

See Section 7.1 in Chapter 7 for more details on how folios are freed and the TLB is maintained within the kernel.

Next we go ahead and try to start writeback on the dirty folio. If it is an anonymous, swap-backed folio then this begins the process of writing the folio's data into the swap (see Section 12.2.3 in Chapter 12 for more on this).

We perform this paging out via `pageout()`, which we examine in Listing 11-71 and Section 11.5.11 where we explore this function.

This function returns a `pageout_t` enumeration value indicating how to proceed—`PAGE_KEEP` indicates that the folio should be rotated to the back of the current LRU vector as the writeback is not complete yet (it remains locked), `PAGE_ACTIVATE` indicates that it should be moved to the active LRU vector, that is activated (it remains locked), `PAGE_SUCCESS` indicates that the folio has been immediately written to disk and is now unlocked and finally `PAGE_CLEAN` indicates the folio was found to be clean and didn't require page out (it remains locked).

In the case of `PAGE_KEEP` we move to rotate the folio immediately, otherwise if `PAGE_ACTIVATE` is returned we move to activate it. `PAGE_CLEAN` indicates that the folio was in fact not dirty, and so we immediately move directly to the next part of `shrink_page_list()` in Listing 11-68.

In the case of `PAGE_SUCCESS` (indicating that the folio has been submitted for writeback, most likely asynchronously) there is a little more to do. We check again that `PG_writeback` and `PG_dirty` are cleared, before attempting to lock and checking again under the lock. If the folio is dirty, under writeback or we cannot lock it, then we rotate the folio.

We typically expect the folio here to be under writeback as it will be asynchronously proceeding through writeback, so likely the folio will be rotated here waiting for writeback to complete.

With this done, we proceed to the next part of `shrink_page_list()`, concerned with handling the lazy-free case and removal of the folio from its owning mapping. We explore this in Listing 11-68.

```

1959         if (folio_test_anon(folio) && !folio_test_swapbacked(folio)) {
1960             /* follow __remove_mapping for reference */
1961             if (!folio_ref_freeze(folio, 1))
1962                 goto keep_locked;
1963             /*
1964              * The folio has only one reference left, which is
1965              * from the isolation. After the caller puts the
1966              * folio back on the lru and drops the reference, the
1967              * folio will be freed anyway. It doesn't matter
1968              * which lru it goes on. So we don't bother checking
1969              * the dirty flag here.
1970              */
1971             :
1972         } else if (!mapping || !__remove_mapping(mapping, folio, true,
```

```

1974                                     sc->target_mem_cgroup
                                     ))
1975                                     goto keep_locked;

```

Listing 11-68: mm/vmscan.c: shrink_page_list() Lazy Free and Mapping Removal

In the portion of `shrink_page_list()` explored in Listing 11-68, we first consider the lazy-free case—this is an instance where a user has freed a range of memory using `madvise()` with the `MADV_FREE` flag. These folios are marked as anonymous but without the `PG_swapbacked` flag being set.

In this instance, we just immediately freeze the folio, that is atomically swap its expected reference count of one with zero, so that when we move to free the folio later (in Listing 11-69) it is immediately freed without any further action being required.

Otherwise we check whether the folio has a mapping (rather poorly named unfortunately), that is, a page cache or swap cache entry that refers to it. If so, we remove the folio from this mapping via `__remove_mapping()` (see Listing 11-73), as described in detail in Section 11.5.12.

If this fails, or the lazy-free freezing fails, we simply keep and rotate the folio.

Next we examine the final part of `shrink_page_list()` in Listing 11-69.

```

1978         folio_unlock(folio);
1979         :
1980         /*
1981          * Folio may get swapped out as a whole, need to account
1982          * all pages in it.
1983          */
1984         nr_reclaimed += nr_pages;
1985         :
1992         list_add(&folio->lru, &free_pages);
1993         continue;
1994
1995 activate_locked_split:
1996         :
2004 activate_locked:
2005         /* Not a candidate for swapping, so reclaim swap space. */
2006         if (folio_test_swapcache(folio) &&
2007             (mem_cgroup_swap_full(&folio->page) ||
2008              folio_test_mlocked(folio)))
2009             try_to_free_swap(&folio->page);
2010         :
2011         if (!folio_test_mlocked(folio)) {
2012             int type = folio_is_file_lru(folio);
2013             folio_set_active(folio);
2014             stat->nr_activate[type] += nr_pages;
2015             :
2016         }
2017 keep_locked:

```

```

2018             folio_unlock(folio);
2019 keep:
2020             list_add(&folio->lru, &ret_pages);
2021             :
2022             }
2023             /* 'page_list' is always empty here */
2024             :
2025             try_to_unmap_flush();
2026             free_unref_page_list(&free_pages);
2027             :
2028             list_splice(&ret_pages, page_list);
2029             :
2030             return nr_reclaimed;
2031 }

```

Listing 11-69: mm/vmscan.c: shrink_page_list() Suffix

The portion of `shrink_page_list()` explored in Listing 11-69 firstly handles the successful case where the folio has reached the point of being ready to be freed due to reclaim. It is unlocked, we increment the `nr_reclaimed` count we return to the caller, and we add the folio to the `free_list` for bulk freeing, before looping around to the next folio.

We then consider each of the labels we have been jumping to for activation and keeping (that is rotation) of folios.

On activation we check to see whether the folio is in the swap cache and the swap cache is full (checked via `mem_cgroup_swap_full()`, a function which correctly identifies whether available swap resource is full even in the non-cgroup case) or if the folio is `mlock()`ed and thus not eligible for swapping out, in which case we try to free up the relevant swap cache space via `try_to_free_swap()` (see Listing 12-31 and Section 12.2.4 in Chapter 12 for details).

We check again to ensure the folio is not `mlock()`ed (for instance it became so during a period it was not locked), in which case activation is not especially meaningful.

If not, then we set the `PG_active` flag, which, when the folio is placed back into an LRU will cause it to be placed into the active LRU whether it is anonymous or file-backed.

Whether we are keeping or activating a locked folio, we unlock it before we proceed, and whether activating or keeping a folio in general, we add it to the `ret_pages` list.

At this stage we reach the end of the loop, at which point the input `page_list` will be empty.

We invoke `try_to_unmap_flush` to perform a batched TLB flush before we try to free the folios, especially as we wish to avoid any writes occurring on file-backed folios that might cause them to be unnecessarily re-dirtied.

See Section 7.1 in Chapter 7 for more details on how folios are freed and the TLB is maintained within the kernel.

We then go ahead and free successfully reclaimed folios on the `free_list` via `free_unref_page_list()` to the physical allocator (see Chapter 2 for details on the physical memory allocator).

With this done, we place any folios that are to be kept (that is rotated) or activated which we stored in `ret_pages` to the input `page_list` so the caller can place these back into the appropriate LRU vectors.

Finally, we return the number of base pages we reclaimed during the operation.

11.5.10 Reclaim Folio Reference Checking

A key part of reclaim and the Last Recently Used (LRU) algorithm is being able to determine whether data has been accessed or not. As described in Section 11.2, we maintain active and inactive LRU lists in order to practically approximate a true LRU algorithm in an efficient manner.

We do this by examining folios in the inactive LRU vector when memory pressure arises and only then checking whether the folio has been accessed since we last checked.

We achieve this through a useful feature provided by hardware which is the accessed or “young” page table flag—for instance in x86-64 this is checked via `pte_young()` which checks for the `_PAGE_ACCESSED` flag.

The useful quality of this flag is that it can be cleared by software but is set by hardware when that mapping is accessed.

The function which performs this check and applies the LRU vector logic around folio references is `folio_check_references()`, which we examine in Listing 11-70.

```

1433 static enum page_references folio_check_references(struct folio *folio,
1434                                                  struct scan_control *sc)
1435 {
1436     int referenced_ptes, referenced_folio;
1437     unsigned long vm_flags;
1438
1439     referenced_ptes = folio_referenced(folio, 1, sc->target_mem_cgroup,
1440                                       &vm_flags);
1441     referenced_folio = folio_test_clear_referenced(folio);
1442
1443     /*
1444      * The supposedly reclaimable folio was found to be in a VM_LOCKED vma
1445      *
1446      * Let the folio, now marked Mlocked, be moved to the unevictable list
1447      *
1448      */
1449     if (vm_flags & VM_LOCKED)
1450         return PAGEREf_ACTIVATE;
1451
1452     /* rmap lock contention: rotate */
1453     if (referenced_ptes == -1)

```

```

1452         return PAGEREF_KEEP;
1453
1454     if (referenced_ptes) {
1455         /*
1456          * All mapped folios start out with page table
1457          * references from the instantiating fault, so we need
1458          * to look twice if a mapped file/anon folio is used more
1459          * than once.
1460          *
1461          * Mark it and spare it for another trip around the
1462          * inactive list. Another page table reference will
1463          * lead to its activation.
1464          *
1465          * Note: the mark is set for activated folios as well
1466          * so that recently deactivated but used folios are
1467          * quickly recovered.
1468          */
1469         folio_set_referenced(folio);
1470
1471
1472
1473         if (referenced_folio || referenced_ptes > 1)
1474             return PAGEREF_ACTIVATE;
1475
1476         /*
1477          * Activate file-backed executable folios after first usage.
1478          */
1479         if ((vm_flags & VM_EXEC) && folio_is_file_lru(folio))
1480             return PAGEREF_ACTIVATE;
1481
1482         return PAGEREF_KEEP;
1483     }
1484
1485     /* Reclaim if clean, defer dirty folios to writeback */
1486     if (referenced_folio && folio_is_file_lru(folio))
1487         return PAGEREF_RECLAIM_CLEAN;
1488
1489     return PAGEREF_RECLAIM;
1490 }

```

Listing 11-70: mm/vmscan.c: *folio_check_references()*

The `folio_check_references()` function first uses `folio_referenced()` to walk the reverse-mapping of the folio and first detect all struct `vm_area_struct` (VMA) objects which map the folio, before walking each of their page tables until we reach the PTE level at which point we check for the accessed page table flag, and atomically clear it at the same time.

Each VMA is examined by this reverse mapping functionality using the `folio_referenced_one()` function which performs this operation.

The `folio_referenced()` function also retrieves a set of all the flags of VMAs which reference the folio which gets returned to the caller. This function returns the number of references found.

See Chapter 3 for more on page tables, Chapter 4 for more on VMAs and process memory and Chapter 7 for more on the reverse mapping.

After retrieving this information, we test for, then clear the folio's `PG_referenced` flag.

This flag is used to indicate that a folio was found to be referenced the last time we checked. We clear it each time so it will only be set if the folio was referenced again since we last checked it.

Before performing the reference handling however, we check for two edge cases—if the folio is from a virtual range that is `mlock()`ed, as indicated by the `VM_LOCKED` flag, then it can't be reclaimed, so we activate it to get it out the way of reclaim.

Equally if we had some lock contention when trying to determine folio references, indicated by `folio_referenced()` returning -1, we simply rotate the folio.

Finally we perform the core check—if any mappings did indeed reference this folio since we last checked, we set the `PG_referenced` flag.

We then determine whether this happened on the last check two, or if there was a reference from more than one mapping—in either case this indicates that this folio is likely part of the working set, most certainly in use at this time, and thus should be activated.

Otherwise we consider a special case—if the folio was referenced but did not previously have the `PG_referenced` flag set, but is file-backed and maps executable code (as indicated by the `VM_EXEC` VMA flag), we skip the double reference check and simply indicate that we should activate it right away as indicated by `PAGEREF_ACTIVATE`.

NOTE

The kernel makes a special effort to quickly activate executable code mappings, as reclaiming these can be costly in terms of latency of executing code.

If the folio was referenced but not previously, we rotate it to the back of the current list indicated by `PAGEREF_KEEP`.

If the folio was not referenced since last we checked it, this indicates it is eligible for reclaim. If the folio is file-backed we return `PAGEREF_RECLAIM_CLEAN` indicating that we should reclaim only if clean, leaving writeback to handle the dirty data.

Otherwise, we return `PAGEREF_RECLAIM` indicating the folio is simply available for reclaim.

11.5.11 Reclaim Page Out

When we need to write back dirty folios to disk, whether swap-backed folios in the swap cache being written out to the swap on disk or dirty file-backed

folios, we do this via `pageout()`, which we examine in Listing 11-71 (eliding out of scope buffer handling, trace hooks and statistical updates).

```

1211  /*
1212   * pageout is called by shrink_page_list() for each dirty page.
1213   * Calls ->writepage().
1214   */
1215  static pageout_t pageout(struct folio *folio, struct address_space *mapping,
1216                          struct swap_iocb **plug)
1217  {
1218      /*
1219       * If the folio is dirty, only perform writeback if that write
1220       * will be non-blocking. To prevent this allocation from being
1221       * stalled by pagecache activity. But note that there may be
1222       * stalls if we need to run get_block(). We could test
1223       * PagePrivate for that.
1224       *
1225       * If this process is currently in __generic_file_write_iter() against
1226       * this folio's queue, we can perform writeback even if that
1227       * will block.
1228       *
1229       * If the folio is swapcache, write it back even if that would
1230       * block, for some throttling. This happens by accident, because
1231       * swap_backing_dev_info is bust: it doesn't reflect the
1232       * congestion state of the swapdevs. Easy to fix, if needed.
1233       */
1234      if (!is_page_cache_freeable(folio))
1235          return PAGE_KEEP;
1236      if (!mapping) {
1237          :
1248          return PAGE_KEEP;
1249      }
1250      if (mapping->a_ops->writepage == NULL)
1251          return PAGE_ACTIVATE;
1252
1253      if (folio_clear_dirty_for_io(folio)) {
1254          int res;
1255          struct writeback_control wbc = {
1256              .sync_mode = WB_SYNC_NONE,
1257              .nr_to_write = SWAP_CLUSTER_MAX,
1258              .range_start = 0,
1259              .range_end = LLONG_MAX,
1260              .for_reclaim = 1,
1261              .swap_plug = plug,
1262          };
1263
1264          folio_set_reclaim(folio);
1265          res = mapping->a_ops->writepage(&folio->page, &wbc);

```

```

1266         if (res < 0)
1267             handle_write_error(mapping, folio, res);
1268         if (res == AOP_WRITEPAGE_ACTIVATE) {
1269             folio_clear_reclaim(folio);
1270             return PAGE_ACTIVATE;
1271         }
1272
1273         if (!folio_test_writeback(folio)) {
1274             /* synchronous write or broken a_ops? */
1275             folio_clear_reclaim(folio);
1276         }
1277         :
1279         return PAGE_SUCCESS;
1280     }
1281
1282     return PAGE_CLEAN;
1283 }

```

Listing 11-71: mm/vmscan.c: pageout()

We start `pageout()` by performing the checking whether the associated page cache (or swap) entry is freeable, via `is_page_cache_freeable` which we examine in Listing 11-72.

```

1039 static inline int is_page_cache_freeable(struct folio *folio)
1040 {
1041     /*
1042      * A freeable page cache page is referenced only by the caller
1043      * that isolated the page, the page cache and optional buffer
1044      * heads at page->private.
1045      */
1046     return folio_ref_count(folio) - folio_test_private(folio) ==
1047            1 + folio_nr_pages(folio);
1048 }

```

Listing 11-72: mm/vmscan.c: is_page_cache_freeable

`is_page_cache_freeable` importantly ensures that we do not attempt to reclaim a folio that has been pinned beyond the references that we expect.

Returning to `pageout()` in Listing 11-71, we see that we indicate that the folio should be rotated if it is not currently freeable, equally if no mapping was specified (indicating an anonymous folio without swap cache for instance), then we do the same.

We then check to see whether we have a virtual file system hook to be able to write the individual folio out via a struct `address_space_operations` hook in struct `address_space->a_ops->writepage`.

If not we cannot write the page out, but since it's dirty, we should get it out of the way of reclaim and thus indicate to activate it.

Finally we are ready to proceed with writeback. We start by invoking `folio_clear_dirty_for_io()`, which checks to ensure that we can writeback

and resets the dirty flag, marking the folio clean (and associated mappings, though in this instance mappings should be cleared).

See Listing 10-30 in Section 10.10 and Chapter 10 for a detailed exploration of this.

If we are able to writeback, we write back `SWAP_CLUSTER_MAX` pages, setting the `PG_reclaim` flag so when writeback ends the folio is immediately put to the front of the queue for reclaim via `folio_rotate_reclaimable()` (see Listing 11-92 and Section 11.7.7)). This function is invoked when writeback is complete in `folio_end_writeback()` (see Chapter 10 for more on writeback as a whole).

We handle any write errors, and the file system indicating that the folio should instead be activated, also we check to see if synchronous I/O occurred, i.e. whether the folio's `PG_writeback` flag has been cleared. If so we clear `PG_reclaim` as we know for a fact writeback is now complete.

In this case we return indicating that we succeeded in paging out, otherwise if `folio_clear_dirty_for_io()` failed, this indicates that the folio was not in fact dirty at the point we attempted to invoke this and thus we return indicating the folio is clean.

See Section 12.2.4 in Chapter 12 for a detailed swap-specific analysis of this logic.

11.5.12 Reclaim Mapping Removal

When a folio undergoes reclaim and is about to be freed, we must remove it from the “mapping” which contains it, that is whatever is returned by the `folio_mapping()` function.

In the case of a file-backed folio, this will be the struct `address_space` object describing the page cache entry (see Chapter 9 for more on the page cache), for an anonymous folio this will be the struct `address_space` object describing the swapper address space describing the swap cache which contains the folio, or `NULL` if there is no such entry (that is, the folio is not in the swap cache).

NOTE

Mapping is a poor name for what this entity represents, that is the aggregate abstraction which the folio finds itself a part of. It is easily confused with a folio's references (the number of times it has been accessed since we last checked) or whether it's mapped (determined by folio map counts).

For those instances where there is a mapping, reclaim invokes `_remove_mapping()`, which we examine in Listing 11-73 (eliding bug checks, some larger comments, out of scope working set and cgroup logic, and mapping shrinking logic).

```

1289 static int _remove_mapping(struct address_space *mapping, struct folio *folio
    ,
1290                               bool reclaimed, struct mem_cgroup *target_memcg)
1291 {
1292     int refcount;
    :

```

```

1298     if (!folio_test_swapcache(folio))
1299         spin_lock(&mapping->host->i_lock);
1300     xa_lock_irq(&mapping->i_pages);
1301     :
1326     refcount = 1 + folio_nr_pages(folio);
1327     if (!folio_ref_freeze(folio, refcount))
1328         goto cannot_free;
1329     /* note: atomic_cmpxchg in page_ref_freeze provides the smp_rmb */
1330     if (unlikely(folio_test_dirty(folio))) {
1331         folio_ref_unfreeze(folio, refcount);
1332         goto cannot_free;
1333     }
1334
1335     if (folio_test_swapcache(folio)) {
1336         swp_entry_t swap = folio_swap_entry(folio);
1337         :
1340         __delete_from_swap_cache(folio, swap, shadow);
1341         xa_unlock_irq(&mapping->i_pages);
1342         put_swap_page(&folio->page, swap);
1343     } else {
1344         void (*free_folio)(struct folio *);
1345
1346         free_folio = mapping->a_ops->free_folio;
1347         :
1366         __filemap_remove_folio(folio, shadow);
1367         xa_unlock_irq(&mapping->i_pages);
1368         :
1370         spin_unlock(&mapping->host->i_lock);
1371
1372         if (free_folio)
1373             free_folio(folio);
1374     }
1375
1376     return 1;
1377
1378 cannot_free:
1379     xa_unlock_irq(&mapping->i_pages);
1380     if (!folio_test_swapcache(folio))
1381         spin_unlock(&mapping->host->i_lock);
1382     return 0;
1383 }

```

Listing 11-73: mm/vmscan.c: `__remove_mapping()`

The `__remove_mapping()` function starts by acquiring the appropriate locks to be able to remove the folio from its mapping.

After this we perform a vital reference count check—we must avoid reclaiming folios which are pinned by kernel tasks, and at this stage we expect

the page cache or swap cache to hold a reference in addition to the isolation we have applied to the folio pages.

We invoke `folio_ref_freeze()` which attempts to atomically swap the existing reference count with zero. If this fails we simply exit, as we cannot remove the folio from its mapping and thus cannot go on to free it.

We then double-check the folio is not dirty, unfreezing the folio and aborting if so.

Next we consider the swap case—in this instance we delete the folio from the swap cache. See Section 12.2.4 in Chapter 12 for a detailed analysis.

Otherwise, we remove the folio from the xarray containing it within its containing struct `address_space` object via `__filemap_remove_folio()`, which we examine in detail in Listing 9-89 and Section 9.9.1 in Chapter 9.

If the file system provides a `free_folio()` hook in its struct `address_space_operations` hook in struct `address_space->a_ops->free_folio` then we invoke it.

We then return indicating the operation succeeded.

11.6 Reclaim Throttling

11.6.1 General Reclaim Throttling

There are a number of circumstances in which we must suspend reclaim for a period of time as reclaim is not making progress for one reason or another. This is performed in `reclaim_throttle()`, which we examine shortly in Listing 11-75.

We examine the callers of this function in Figure 11-23.

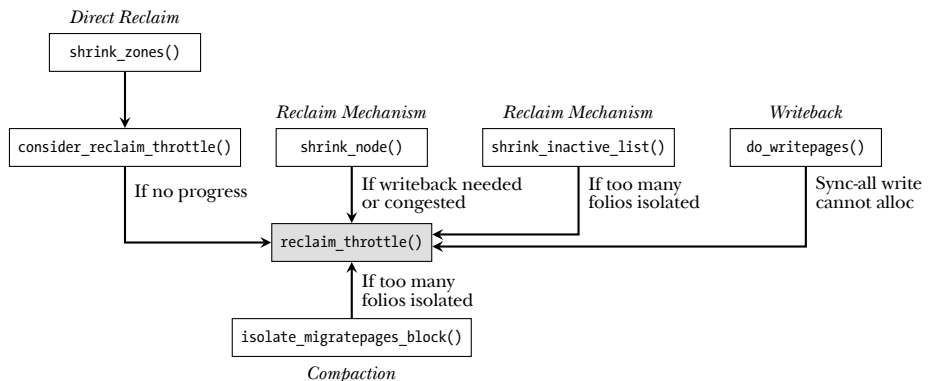


Figure 11-23: Callers of `reclaim_throttle()`

In the case of direct reclaim, we have specific direct reclaim throttling designed for instances where something like a very slow backing store causes writeback to be very slow. This specific case is handled in `throttle_direct_reclaim()` (see Listing 11-77), as discussed in Section 11.6.2.

Additionally, we consider the case where direct reclaim is stuck making no progress, where it would be appropriate to simply sleep for a while. This

is invoked by `shrink_zones()` (see Listing 11-21 and Section 11.3), which in turn calls `consider_reclaim_throttle()` which we examine in Listing 11-74.

```

3425 static void consider_reclaim_throttle(pg_data_t *pgdat, struct scan_control *
      sc)
3426 {
3427     /*
3428      * If reclaim is making progress greater than 12% efficiency then
3429      * wake all the NOPROGRESS throttled tasks.
3430      */
3431     if (sc->nr_reclaimed > (sc->nr_scanned >> 3)) {
3432         wait_queue_head_t *wqh;
3433
3434         wqh = &pgdat->reclaim_wait[VMSCAN_THROTTLE_NOPROGRESS];
3435         if (waitqueue_active(wqh))
3436             wake_up(wqh);
3437
3438         return;
3439     }
3440
3441     /*
3442      * Do not throttle kswapd or cgroup reclaim on NOPROGRESS as it will
3443      * throttle on VMSCAN_THROTTLE_WRITEBACK if there are too many pages
3444      * under writeback and marked for immediate reclaim at the tail of the
3445      * LRU.
3446      */
3447     if (current_is_kswapd() || cgroup_reclaim(sc))
3448         return;
3449
3450     /* Throttle if making no progress at high priorities. */
3451     if (sc->priority == 1 && !sc->nr_reclaimed)
3452         reclaim_throttle(pgdat, VMSCAN_THROTTLE_NOPROGRESS);
3453 }

```

Listing 11-74: `mm/vmscan.c: consider_reclaim_throttle()`

The reclaim throttle mechanism operates using the `struct pglist_data->reclaim_wait` array which contains wait queues for each category of reclaim throttling.

NOTE

As mentioned elsewhere in the book wait queues are a means within the kernel for threads to wait on a certain event to occur, sleeping as they do so. In the instance of reclaim throttling, we establish a wait queue entry when the throttling occurs, then sleep, however we expose the wait queues so that the threads can be woken up.

The `consider_reclaim_throttle()` function shown in Listing 11-74 is invoked by `shrink_zones()` as part of direct reclaim for the first node from which an allocation could be made.

In this function, we are concerned with the determining whether direct reclaim is making sufficient progress—we in essence have a high water mark, at which point we wake all throttled processes, that is if the number of reclaimed pages exceeds 2^3 or $1/8$ that is 12.5% of scanned pages.

Otherwise, if no reclaim has taken place at all and we are at the lowest priority level (one, that is the case in which we scan the entire inactive and active lists), this suggest we are failing to make progress at all, and therefore we should throttle this process via `reclaim_throttle()` (see Listing 11-75).

In this instance we specify this case via the enum `vmscan_throttle_state` value `VMSCAN_THROTTLE_NOPROGRESS`.

We examine the `reclaim_throttle()` function in Listing 11-75 (eliding real-time kernel scheduling logic and trace hooks).

```

1104 void reclaim_throttle(pg_data_t *pgdat, enum vmscan_throttle_state reason)
1105 {
1106     wait_queue_head_t *wqh = &pgdat->reclaim_wait[reason];
1107     long timeout, ret;
1108     DEFINE_WAIT(wait);
1109
1110     /*
1111      * Do not throttle IO workers, kthreads other than kswapd or
1112      * workqueues. They may be required for reclaim to make
1113      * forward progress (e.g. journalling workqueues or kthreads).
1114      */
1115     if (!current_is_kswapd() &&
1116         current->flags & (PF_IO_WORKER|PF_KTHREAD)) {
1117         :
1118         return;
1119     }
1120
1121     /*
1122      * These figures are pulled out of thin air.
1123      * VMSCAN_THROTTLE_ISOLATED is a transient condition based on too many
1124      * parallel reclaimers which is a short-lived event so the timeout is
1125      * short. Failing to make progress or waiting on writeback are
1126      * potentially long-lived events so use a longer timeout. This is
1127      shaky
1128      * logic as a failure to make progress could be due to anything from
1129      * writeback to a slow device to excessive references pages at the
1130      tail
1131      * of the inactive LRU.
1132      */
1133     switch(reason) {
1134     case VMSCAN_THROTTLE_WRITEBACK:
1135         timeout = HZ/10;
1136
1137         if (atomic_inc_return(&pgdat->nr_writeback_throttled) == 1) {
1138             WRITE_ONCE(pgdat->nr_reclaim_start,

```

```

1137             node_page_state(pgdat, NR_THROTTLED_WRITTEN));
1138         }
1139
1140         break;
1141     case VMSCAN_THROTTLE_CONGESTED:
1142         fallthrough;
1143     case VMSCAN_THROTTLE_NOPROGRESS:
1144         if (skip_throttle_noprogress(pgdat)) {
1145             :
1146             return;
1147         }
1148
1149         timeout = 1;
1150
1151         break;
1152     case VMSCAN_THROTTLE_ISOLATED:
1153         timeout = HZ/50;
1154         break;
1155     default:
1156         WARN_ON_ONCE(1);
1157         timeout = HZ;
1158         break;
1159 }
1160
1161     prepare_to_wait(wqh, &wait, TASK_UNINTERRUPTIBLE);
1162     ret = schedule_timeout(timeout);
1163     finish_wait(wqh, &wait);
1164
1165     if (reason == VMSCAN_THROTTLE_WRITEBACK)
1166         atomic_dec(&pgdat->nr_writeback_throttled);
1167     :
1171 }

```

Listing 11-75: mm/vmscan.c: reclaim_throttle()

In `reclaim_throttle()` we handle four different cases:

VMSCAN_THROTTLE_WRITEBACK Folios under writeback saturated reclaim and still have not been written back to disk, so we must throttle reclaim to give them a chance to do so. See Section 11.5.3 for more on how this arises.

VMSCAN_THROTTLE_ISOLATED A situation can arise where many direct reclaimers are sleeping having isolated batches of folios (see Section 11.5.8 for more on folio isolation), resulting in the number of isolated folios growing too far and causing the LRU lists to grow smaller than they should be. In this instance we throttle. This is invoked by both reclaim logic and compaction both of which isolate folios.

VMSCAN_THROTTLE_NOPROGRESS Invoked by `consider_reclaim_throttle()` (see Listing 11-74) when direct reclaim has made no progress.

VMSCAN_THROTTLE_CONGESTED Folios marked for writeback and immediate reclaim have cycled through the list without being written back (and thus reclaimed) yet, so we throttle reclaim in order to give these folios a chance to be written back. See Section 11.5.3 for more on how this arises.

The `reclaim_throttle()` function shown in Listing 11-75 starts by skipping throttling for non-`kswapd` kernel threads or I/O workers in order to avoid stopping forward progress required for reclaim to succeed.

We then determine a series of arbitrary timeouts based on the throttle modes, in the case of `VMSCAN_THROTTLE_WRITEBACK` keeping track of writeback throttling occurring in struct `pglist_data->nr_writeback_throttled`, also setting the struct `pglist_data->nr_reclaim_start` field each time this is first incremented, i.e. when writeback throttling first starts, tracking the number of base pages written when throttled.

We throttle 1/10th of a second in the `VMSCAN_THROTTLE_WRITEBACK` case, in the case of `VMSCAN_THROTTLE_CONGESTED` and `VMSCAN_THROTTLE_NOPROGRESS` we wait the minimum time possible, unless `skip_throttle_noprogress()` indicates we should not (we explore this in Listing 11-76 shortly).

Finally in `VMSCAN_THROTTLE_ISOLATED` mode we wait 1/50th of a second.

We then perform the sleep via `prepare_to_wait()`, `schedule_timeout()` and `finish_wait()`. See the discussion around Listing 11-29 for a discussion of these.

Note that we can be woken up early during this process.

Finally, in the `VMSCAN_THROTTLE_WRITEBACK` case we decrement struct `pglist_data->nr_writeback_throttled` since the throttling is completed.

We examine `skip_throttle_noprogress()` in Listing 11-76.

```

1071 static bool skip_throttle_noprogress(pg_data_t *pgdat)
1072 {
1073     int reclaimable = 0, write_pending = 0;
1074     int i;
1075
1076     /*
1077      * If kswapd is disabled, reschedule if necessary but do not
1078      * throttle as the system is likely near OOM.
1079      */
1080     if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES)
1081         return true;
1082
1083     /*
1084      * If there are a lot of dirty/writeback pages then do not
1085      * throttle as throttling will occur when the pages cycle
1086      * towards the end of the LRU if still under writeback.
1087      */
1088     for (i = 0; i < MAX_NR_ZONES; i++) {
1089         struct zone *zone = pgdat->node_zones + i;
1090
```

```

1091         if (!managed_zone(zone))
1092             continue;
1093
1094         reclaimable += zone_reclaimable_pages(zone);
1095         write_pending += zone_page_state_snapshot(zone,
1096                                                     NR_ZONE_WRITE_PENDING);
1097     }
1098     if (2 * write_pending <= reclaimable)
1099         return true;
1100
1101     return false;
1102 }

```

Listing 11-76: mm/vmscan.c: skip_throttle_noprogress()

In `skip_throttle_noprogress()` we start by indicating that we should skip throttling if indirect reclaim has failed (we have failed to reclaim folios `MAX_RECLAIM_RETRIES` times (hard-coded to sixteen) and this node's `kswapd` kernel thread is now halted.

Then we determine if the write pending page count (a sum of dirty, writeback, or otherwise unstable pages) is less than or equal to half of all reclaimable pages (the sum of the sizes of all of the LRU vectors), if so we skip throttling.

If it is more, then we do not skip throttling. Therefore we must see a lot of dirty/writeback pages before throttling occurs.

All of these kind of consideration are heuristics based on real workloads and tweak reclaim behaviour accordingly.

11.6.2 Direct Reclaim Throttling

When performing direct reclaim, the `PF_MEMALLOC` flag is set in the current thread's `struct task_struct->flags` field in `__perform_reclaim()` (see Listing 11-18).

This causes `__gfp_pfmemalloc_flags()` (see Listing 11-5 in Section 11.1) to permit watermarks to be ignored (via `ALLOC_NO_WATERMARKS`).

This is something of a risk if, for some reason, reclaim performs slowly, then we may end up reaching dangerously low levels of memory reserve. To avoid this, we check whether reserves are running low and throttle direct reclaim if so.

This is performed by `throttle_direct_reclaim()`, as shown in Listing 11-77.

```

3712 /*
3713  * Throttle direct reclaimers if backing storage is backed by the network
3714  * and the PFMEMALLOC reserve for the preferred node is getting dangerously
3715  * depleted. kswapd will continue to make progress and wake the processes
3716  * when the low watermark is reached.
3717  *
3718  * Returns true if a fatal signal was delivered during throttling. If this
3719  * happens, the page allocator should not consider triggering the OOM killer.

```



```

3720 */
3721 static bool throttle_direct_reclaim(gfp_t gfp_mask, struct zonelist *zonelist,
3722                                     nodemask_t *nodemask)
3723 {
3724     struct zoneref *z;
3725     struct zone *zone;
3726     pg_data_t *pgdat = NULL;
3727
3728     /*
3729      * Kernel threads should not be throttled as they may be indirectly
3730      * responsible for cleaning pages necessary for reclaim to make
3731      * progress. kjournald for example may enter direct reclaim while
3732      * committing a transaction where throttling it could forcing other
3733      * processes to block on log_wait_commit().
3734      */
3735     if (current->flags & PF_KTHREAD)
3736         goto out;
3737
3738     /*
3739      * If a fatal signal is pending, this process should not throttle.
3740      * It should return quickly so it can exit and free its memory
3741      */
3742     if (fatal_signal_pending(current))
3743         goto out;
3744
3745     /*
3746      * Check if the pfmemalloc reserves are ok by finding the first node
3747      * with a usable ZONE_NORMAL or lower zone. The expectation is that
3748      * GFP_KERNEL will be required for allocating network buffers when
3749      * swapping over the network so ZONE_HIGHMEM is unusable.
3750      *
3751      * Throttling is based on the first usable node and throttled
3752      * processes
3753      * wait on a queue until kswapd makes progress and wakes them. There
3754      * is an affinity then between processes waking up and where reclaim
3755      * progress has been made assuming the process wakes on the same node.
3756      * More importantly, processes running on remote nodes will not
3757      * compete
3758      * for remote pfmemalloc reserves and processes on different nodes
3759      * should make reasonable progress.
3760      */
3761     for_each_zone_zonelist_nodemask(zone, z, zonelist,
3762                                     gfp_zone(gfp_mask), nodemask) {
3763         if (zone_idx(zone) > ZONE_NORMAL)
3764             continue;
3765     }

```

```

3764             /* Throttle based on the first usable node */
3765             pgdat = zone->zone_pgdat;
3766             if (allow_direct_reclaim(pgdat))
3767                 goto out;
3768             break;
3769         }
3770
3771         /* If no zone was usable by the allocation flags then do not throttle
3772         */
3773         if (!pgdat)
3774             goto out;

```

Listing 11-77: mm/vmscan.c: throttle_direct_reclaim() Initial Checks

The purpose of `throttle_direct_reclaim()` is to consider a scenario where the direct reclaim ought to be throttled due to memory reserves getting low due to reclaim being performed slowly (due to, for instance, the backing store for swap or the page cache existing on a network mount).

As with much of the kernel, this is rather heuristic and considers a number of different edge cases. Note that if the function returns true, then direct reclaim does not occur and the calling `try_to_free_pages()` function will return ultimately to `__alloc_pages_slowpath()` indicating that progress has been made (despite none having actually been made), meaning we will not reach for the OOM killer but instead will loop around and try again.

We only do so in one particular case after having throttled which we will examine shortly, generally this function returns false (which is also the behaviour when code jumps to the out label). When this is returned, direct reclaim proceeds as normal.

We start by excluding kernel threads from consideration, as throttling reclaim for those might cause other things which depend on them to stall.

Next, if a fatal signal is pending, we ought not to throttle as doing so would only delay the tear down of the process, and we need to make sure that we free sufficient memory to do so before then immediately releasing all process memory.

Next we loop through each zone in which we might be able to allocate (See Section 2.4 in Chapter 2 for more on nodes, zones, zonelists and related).

We consider only ordinary zones (those of `ZONE_NORMAL` or below it numerically, so in practice this excludes special zones like `ZONE_MOVABLE` and `ZONE_DEVICE`), and simply check whether each node a zone resides in is permissible for direct reclaim, i.e. do not exhibit signs of depleted memory reserves.

We check whether a node is eligible for direct reclaim via `allow_direct_reclaim()`, which we examine in Listing 11-79 shortly.

Finally, if we find no zone at `ZONE_NORMAL` or below, we simply choose not to throttle at all as are unable to assess these zones.

We examine the throttling logic in Listing 11-78.

```

3775             /* Account for the throttling */

```

```

3776         count_vm_event(PGSCAN_DIRECT_THROTTLE);
3777
3778         /*
3779          * If the caller cannot enter the filesystem, it's possible that it
3780          * is due to the caller holding an FS lock or performing a journal
3781          * transaction in the case of a filesystem like ext[3/4]. In this case
3782          * it is not safe to block on pfmemalloc_wait as kswapd could be
3783          * blocked waiting on the same lock. Instead, throttle for up to a
3784          * second before continuing.
3785          */
3786         if (!(gfp_mask & __GFP_FS))
3787             wait_event_interruptible_timeout(pgdat->pfmemalloc_wait,
3788                 allow_direct_reclaim(pgdat), HZ);
3789         else
3790             /* Throttle until kswapd wakes the process */
3791             wait_event_killable(zone->zone_pgdat->pfmemalloc_wait,
3792                 allow_direct_reclaim(pgdat));
3793
3794         if (fatal_signal_pending(current))
3795             return true;
3796
3797     out:
3798         return false;
3799 }

```

Listing 11-78: mm/vmscan.c: throttle_direct_reclaim() Throttling

The portion of `throttle_direct_reclaim()` which we examine in Listing 11-78 performs the actual throttling, suspending the process until `kswapd`, i.e. the kernel thread which performs indirect reclaim frees enough memory to permit direct reclaim to proceed (noting that by this stage we will have triggered indirect reclaim before trying the direct form of it).

If `__GFP_FS` has not been set in the Get Free Pages (GFP) flag mask, this indicates that access to the filesystem is prohibited (for more on GFP flags see Section 2.6 in Chapter 2).

In this instance, we introduce a timeout to avoid a deadlock between direct and indirect reclaim waiting on the same thing.

If this is not the case, we simply wait as long as it takes for indirect reclaim to establish sufficient memory reserves that we can proceed and for it to notify us that this is the case.

Finally, after throttling, we check again whether a fatal signal is pending, if so in this case we actually do abort direct reclaim as we will now be in a position where indirect reclaim would have done all that is necessary for us to simply exit the reclaim mechanism and handle the fatal signal correctly.

Otherwise, we simply continue with direct reclaim.

We are waiting on `struct pglist_data->pfmemalloc_wait`, a wait queue specifically used to control throttling of direct reclaim.

This wait queue is woken up when indirect reclaim has made progress in `balance_pgdat()` (see Listing 11-32 in Section 11.4.4), where `allow_direct_reclaim()` (see Listing 11-79) is tested again and if it returns true, `struct pglist_data->pfmemalloc_wait` is woken.

It is also woken in `prepare_kswapd_sleep()` (see Listing 11-30 in Section 11.4.3) right before the indirect reclaim process is put to sleep, in order to prevent races that might leave direct reclaim stalled.

We examine `allow_direct_reclaim()` in Listing 11-79.

```

3672 static bool allow_direct_reclaim(pg_data_t *pgdat)
3673 {
3674     struct zone *zone;
3675     unsigned long pfmemalloc_reserve = 0;
3676     unsigned long free_pages = 0;
3677     int i;
3678     bool wmark_ok;
3679
3680     if (pgdat->kswapd_failures >= MAX_RECLAIM_RETRIES)
3681         return true;
3682
3683     for (i = 0; i <= ZONE_NORMAL; i++) {
3684         zone = &pgdat->node_zones[i];
3685         if (!managed_zone(zone))
3686             continue;
3687
3688         if (!zone_reclaimable_pages(zone))
3689             continue;
3690
3691         pfmemalloc_reserve += min_wmark_pages(zone);
3692         free_pages += zone_page_state(zone, NR_FREE_PAGES);
3693     }
3694
3695     /* If there are no reserves (unexpected config) then do not throttle
3696     */
3697     if (!pfmemalloc_reserve)
3698         return true;
3699
3700     wmark_ok = free_pages > pfmemalloc_reserve / 2;
3701
3702     /* kswapd must be awake if processes are being throttled */
3703     if (!wmark_ok && waitqueue_active(&pgdat->kswapd_wait)) {
3704         if (READ_ONCE(pgdat->kswapd_highest_zoneidx) > ZONE_NORMAL)
3705             WRITE_ONCE(pgdat->kswapd_highest_zoneidx, ZONE_NORMAL)
3706     }
3707
3708     wake_up_interruptible(&pgdat->kswapd_wait);
3709 }

```

```

3709         return wmark_ok;
3710     }

```

Listing 11-79: mm/vmscan.c: `allow_direct_reclaim()`

We start `allow_direct_reclaim()` by determining whether the node has undergone more than the permitted number of indirect reclaim attempts (as defined in `MAX_RECLAIM_RETRIES`, hard-coded to 16). If so, then indirect reclaim is of no use to us, and so we have only direct reclaim to work with and so it should be permitted.

Otherwise, we loop through all the zones of the specified node, skipping zones if they are not managed (as determined by `managed_zone()`), i.e. ones where the buddy allocator does not manage a single page there (see Chapter 2 for more on this).

We also skip zones if they simply do not contain any pages which we can reclaim, as determined by `zone_reclaimable_pages()`.

Finally we accumulate all of the reserves that are put in place to allow for allocations using the `PF_MEMALLOC` task flag, i.e. the minimum water mark for each zone, and additionally accumulate the number of free pages for each zone.

If we find ourselves in the unusual situation of there simply being no reserves set at all, we exit and permit direct reclaim, as we can not violate memory reserves that don't exist. This would be an odd configuration but one we nevertheless must account for.

Finally we determine if direct reclaim can proceed by checking to ensure that accumulated free pages across zones exceed at least half of accumulated memory reserves across zones.

This indicates that reclaim is functioning sensibly enough that direct reclaim can continue without throttling.

Finally, if we are about to indicate that throttling should take place, we ensure that indirect swap is woken and we also reset it to ensure that it reclaims from `ZONE_NORMAL` or below to ensure we obtain pages that contribute to memory reserves across allocatable zones.

11.7 Folio Batches

As each `lruvec` possesses a heavily-contended `lru_lock`, it is often more efficient to perform `lruvec` operations in *batches* over which the lock is held rather than acquiring/releasing the lock one folio at a time. Note that we do *not* batch compound folios.

These batches are represented by struct `folio_batch` objects (byte-for-byte identical to the legacy struct `pagevec` objects):-

```

74  /**
75   * struct folio_batch - A collection of folios.
76   *
77   * The folio_batch is used to amortise the cost of retrieving and
78   * operating on a set of folios. The order of folios in the batch may be
79   * significant (eg delete_from_page_cache_batch()). Some users of the

```

```

80  * folio_batch store "exceptional" entries in it which can be removed
81  * by calling folio_batch_remove_exceptionals().
82  */
83  struct folio_batch {
84      unsigned char nr;
85      bool percpu_pvec_drained;
86      struct folio *folios[PAGEVEC_SIZE];
87  };

```

Listing 11-80: include/linux/pagevec.h: struct folio_batch

There are a maximum of `PAGEVEC_SIZE` entries in each batch (this is hard-coded to 15 which, given struct alignment rules, mean that we end up with each batch equal to 16 system words, 128 bytes for 64-bit systems or 2 cache lines per batch).

Examining each field:-

- `nr` – Indicates the number of folios present in the batch.
- `percpu_pvec_drained` – This is a flag indicating whether a *drain* has been initiated – this is when folios in the batch are placed back, or drained, back to the appropriate lruvec lists.
- `folios` – An array of `PAGEVEC_SIZE` folios.

While folio batches can be used independently, *core* ones are maintained in a series of *per-CPU* global objects, separated by the operation which is to be performed on folios contained in each batch:-

11.7.1 CPU Folio Batches

Each of the remaining folio batches are declared in `struct cpu_fbatches` and instantiated in `cpu_fbatches`:-

```

58  /*
59  * The following folio batches are grouped together because they are protected
60  * by disabling preemption (and interrupts remain enabled).
61  */
62  struct cpu_fbatches {
63      local_lock_t lock;
64      struct folio_batch lru_add;
65      struct folio_batch lru_deactivate_file;
66      struct folio_batch lru_deactivate;
67      struct folio_batch lru_lazyfree;
68  #ifdef CONFIG_SMP
69      struct folio_batch activate;
70  #endif
71  };
72  static DEFINE_PER_CPU(struct cpu_fbatches, cpu_fbatches) = {
73      .lock = INIT_LOCAL_LOCK(lock),

```

74 };

Listing 11-81: mm/swap.c: struct cpu_fbatches and cpu_fbatches

These maintain a local lock while being access which disables preemption while the lock is held.

11.7.2 LRU Rotation

A folio batch is maintained for *LRU rotation*. This operation places folios at the *tail* of the inactive list in order that they get reclaimed next (we will discuss this operation in detail shortly below). This is used by `folio_rotate_reclaimable()` at the end of *writeback* (when a dirty file folio's data is written back to disk, see Chapter 10 for more on writeback). This requires IRQs to be disabled so is therefore kept separate from other batches in a local `struct lru_rotate` object and instantiated as `lru_rotate`:-

```

49 /* Protecting only lru_rotate.fbatch which requires disabling interrupts */
50 struct lru_rotate {
51     local_lock_t lock;
52     struct folio_batch fbatch;
53 };
54 static DEFINE_PER_CPU(struct lru_rotate, lru_rotate) = {
55     .lock = INIT_LOCAL_LOCK(lock),
56 };

```

Listing 11-82: mm/swap.c: struct lru_rotate and lru_rotate

These maintain a *local lock* while being accessed, which will disable preemption and mask interrupts while the lock is held.

11.7.3 mlock

Finally, there is a separate `mlock()` folio list for folios which have been mlocked, i.e. made unevictable:-

```

31 struct mlock_pvec {
32     local_lock_t lock;
33     struct pagevec vec;
34 };
35
36 static DEFINE_PER_CPU(struct mlock_pvec, mlock_pvec) = {
37     .lock = INIT_LOCAL_LOCK(lock),
38 };

```

Listing 11-83: mm/mlock.c: struct mlock_pvec and mlock_pvec

This maintains a local lock while being access which disables preemption while the lock is held.

Note that this folio batch is maintained separately from the rest – while the folios on it are batched, they do not use the same call back mechanisms described below.

11.7.4 Folio Operations

Previously we excluded the operations that use folio batches while looking at lruvec operations. These are implemented as callback functions which perform the actual operation required. Let's examine them now:-

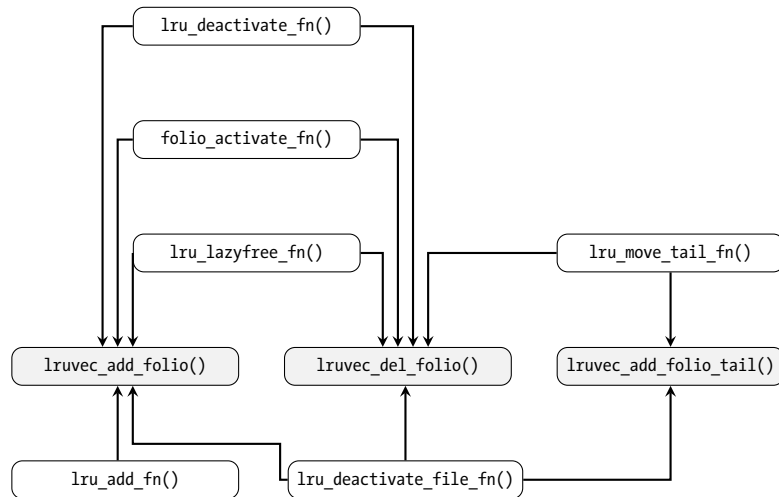


Figure 11-24: *lruvec folio batch drain callbacks*

These are the functions which *actually* add and remove folios from lruvecs, invoked as a `move_fn_t` callback when each folio batch is drained. This occurs both when the folios reach their maximum size (`PAGEVEC_SIZE`) and also when a drain is manually invoked.

The `folio_batch_move_lru()` function performs the actual drain of each batch:-

```

232 static void folio_batch_move_lru(struct folio_batch *fbatch, move_fn_t move_fn
    )
233 {
234     int i;
235     struct lruvec *lruvec = NULL;
236     unsigned long flags = 0;
237
238     for (i = 0; i < folio_batch_count(fbatch); i++) {
239         struct folio *folio = fbatch->folios[i];
240
241         /* block memcg migration while the folio moves between lru */
242         if (move_fn != lru_add_fn && !folio_test_clear_lru(folio))
243             continue;

```



```

244
245         lruvec = folio_lruvec_relock_irqsave(folio, lruvec, &flags);
246         move_fn(lruvec, folio);
247
248         folio_set_lru(folio);
249     }
250
251     if (lruvec)
252         unlock_page_lruvec_irqrestore(lruvec, flags);
253     folios_put(fbatches->folios, folio_batch_count(fbatches));
254     folio_batch_init(fbatches);
255 }

```

Listing 11-84: mm/swap.c: folio_batch_move_lru()

The logic is as follows:-

1. For each entry in the folio batch (the count taken from the `nr` field, accessed via `folio_batch_count()`):-
 - (a) Clear the folio LRU flag, if it is set, and determine if it was previously set, via `folio_test_clear_lru()`.
 - (b) If the specified callback is *not* `lru_add_fn()` and the folio is non-LRU, i.e. not currently present on an LRU, then we skip it – the only situation where we expect non-LRU folios to be operated on here is when they are added via `lru_add_fn()`.
 - (c) Using `folio_lruvec_relock_irqsave()`, determine the `lruvec` the folio is present on (via `folio_lruvec()`), compare it against the last `lruvec` we locked – if they are the same, we perform no action, otherwise we release the previous lock via `unlock_page_lruvec_irqrestore()` and acquire a lock on the new `lruvec` via `folio_lruvec_lock_irqsave()`. We always disable interrupts while holding the `lruvec` lock.
 - (d) Invoke the specified callback, `move_fn`. Note that each callback increments the folio's reference count before performing its operation via `folio_get()` to ensure it is not freed underneath us.
 - (e) Set the folio LRU flag via `folio_set_lru()`.
2. Unlock the `lruvec` via `unlock_page_lruvec_irqrestore()`.
3. Decrement each folio's reference count, which was incremented during the folio operation, using `folios_put()`.
4. Finally, reset the folio via `folio_batch_init()` – this resets the `nr` field to 0 and also sets `percpu_pvec_drained` to false. Note that a local lock will have been acquired before invoking this so it is safe to modify the batch.

`folio_batch_move_lru()` is only directly invoked by functions which perform drain. Otherwise, `folio_batch_add_and_move()` is used which adds to a batch and if either the batch is full, or an edge case condition occurs, it drains the batch via `folio_batch_move_lru()`:-

```

257 static void folio_batch_add_and_move(struct folio_batch *fbatches,
258                                     struct folio *folio, move_fn_t move_fn)

```

```

259 {
260     if (folio_batch_add(fbatch, folio) && !folio_test_large(folio) &&
261         !lru_cache_disabled())
262         return;
263     folio_batch_move_lru(fbatch, move_fn);
264 }

```

Listing 11-85: mm/swap.c: folio_batch_add_and_move()

The edge cases in which we *always* perform drain are – pages being compound (checked via `folio_test_large()`) or the LRU cache being disabled altogether (checked via `lru_cache_disabled()`). In the former case, it makes less sense to batch operations when underlying base pages are already large.

Folios are added to each batch via `folio_batch_add()` which returns the number of still-available slots in the batch. This is, of course, invoked with any local locks applicable to the batch held:-

```

116 /**
117  * folio_batch_add() - Add a folio to a batch.
118  * @fbatch: The folio batch.
119  * @folio: The folio to add.
120  *
121  * The folio is added to the end of the batch.
122  * The batch must have previously been initialised using folio_batch_init().
123  *
124  * Return: The number of slots still available.
125  */
126 static inline unsigned folio_batch_add(struct folio_batch *fbatch,
127                                       struct folio *folio)
128 {
129     fbatch->folios[fbatch->nr++] = folio;
130     return fbatch_space(fbatch);
131 }

```

Listing 11-86: include/linux/pagevec.h: folio_batch_add()

This determines how much folio batch space is available via `fbatch_space()`:-

```

111 static inline unsigned int fbatch_space(struct folio_batch *fbatch)
112 {
113     return PAGEVEC_SIZE - fbatch->nr;
114 }

```

Listing 11-87: include/linux/pagevec.h: fbatch_space()

This simply subtracts the current folio count in the batch from the folio batch size, `PAGEVEC_SIZE`. This operation is safe as the batches are per-CPU and preemption will be disabled due to the local lock.

Examining each function which invokes `folio_batch_add_and_move()`, which are all the functions which manipulate folios in batches other than draining

them, i.e. the usual means by which *all* user space folios are manipulated in lruvecs:-

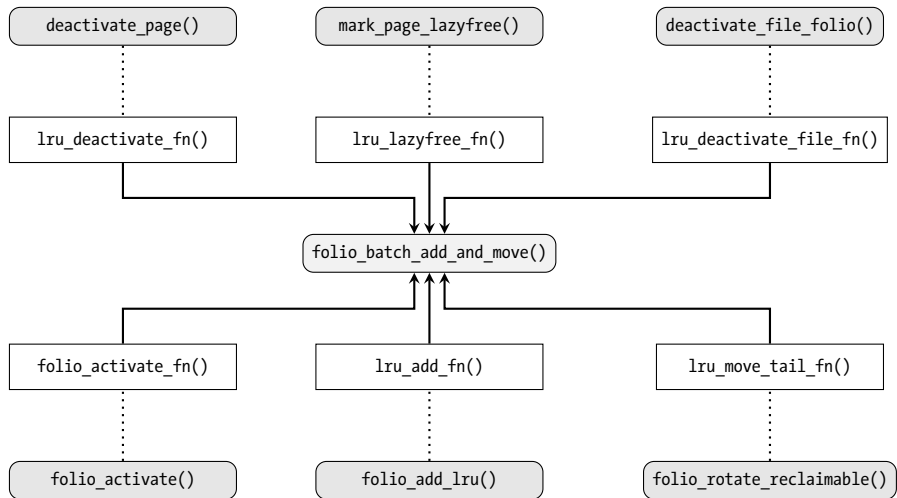


Figure 11-25: Non-drain folio batch operations

All of these other than `folio_add_lru()` move folios between lists (`mark_page_lazyfree()` simply moves a folio to the inactive list), whereas `folio_add_lru()` is uniquely the means by which folios are *first added* to an LRU. Let's examine this first:-

11.7.5 Adding Folios to a Batch

We add folios to a batch using `folio_add_lru()`, which we examine in Listing 11-88.

```

470 /**
471  * folio_add_lru - Add a folio to an LRU list.
472  * @folio: The folio to be added to the LRU.
473  *
474  * Queue the folio for addition to the LRU. The decision on whether
475  * to add the page to the [in]active [file|anon] list is deferred until the
476  * folio_batch is drained. This gives a chance for the caller of folio_add_lru
477  *   ()
478  * have the folio added to the active list using folio_mark_accessed().
479  */
479 void folio_add_lru(struct folio *folio)
480 {
481     struct folio_batch *fbatch;
482
483     VM_BUG_ON_FOLIO(folio_test_active(folio) &&
484                     folio_test_unevictable(folio), folio);
485     VM_BUG_ON_FOLIO(folio_test_lru(folio), folio);

```

```

486
487     folio_get(folio);
488     local_lock(&cpu_fbatches.lock);
489     fbatch = this_cpu_ptr(&cpu_fbatches.lru_add);
490     folio_batch_add_and_move(fbatch, folio, lru_add_fn);
491     local_unlock(&cpu_fbatches.lock);
492 }

```

Listing 11-88: mm/swap.c: folio_add_lru()

The logic here is straightforward – after making some sanity assertions, we increment the reference count for the folio so it isn’t freed beneath us, acquire the local lock from the `cpu_fbatches` object (disabling preemption), obtain the batch for this CPU from the `cpu_fbatches.lru_add` batch, then invoke `folio_batch_add_and_move()` to perform the batch add before releasing the local lock.

`folio_add_lru()`, as the means by which folios are added to an LRU, sits at the heart of *all* evictable user memory folio allocations (as, in order to be evictable, the folios must be placed on an LRU).

It is informative then to examine what calls this function as this gives us a broader picture of the interactions between LRUs and the rest of the kernel. We examine anonymous folio and file-backed folio invocations of `folio_add_lru()` overleaf.

For anonymous invocations we note which non-core subsystem callers invoke certain functions which have also been excluded for brevity. The intent has been to highlight the *core* callers only.

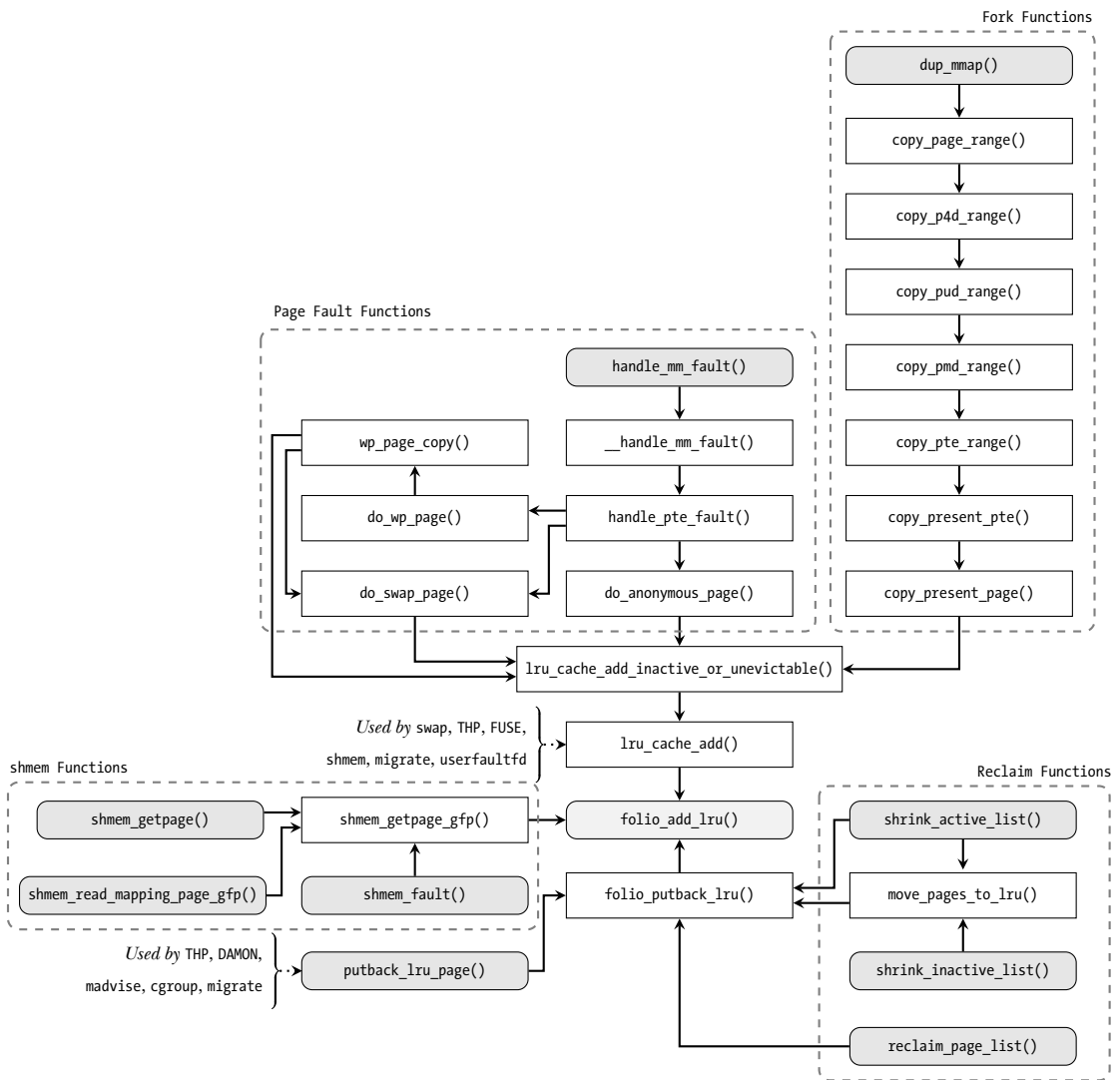
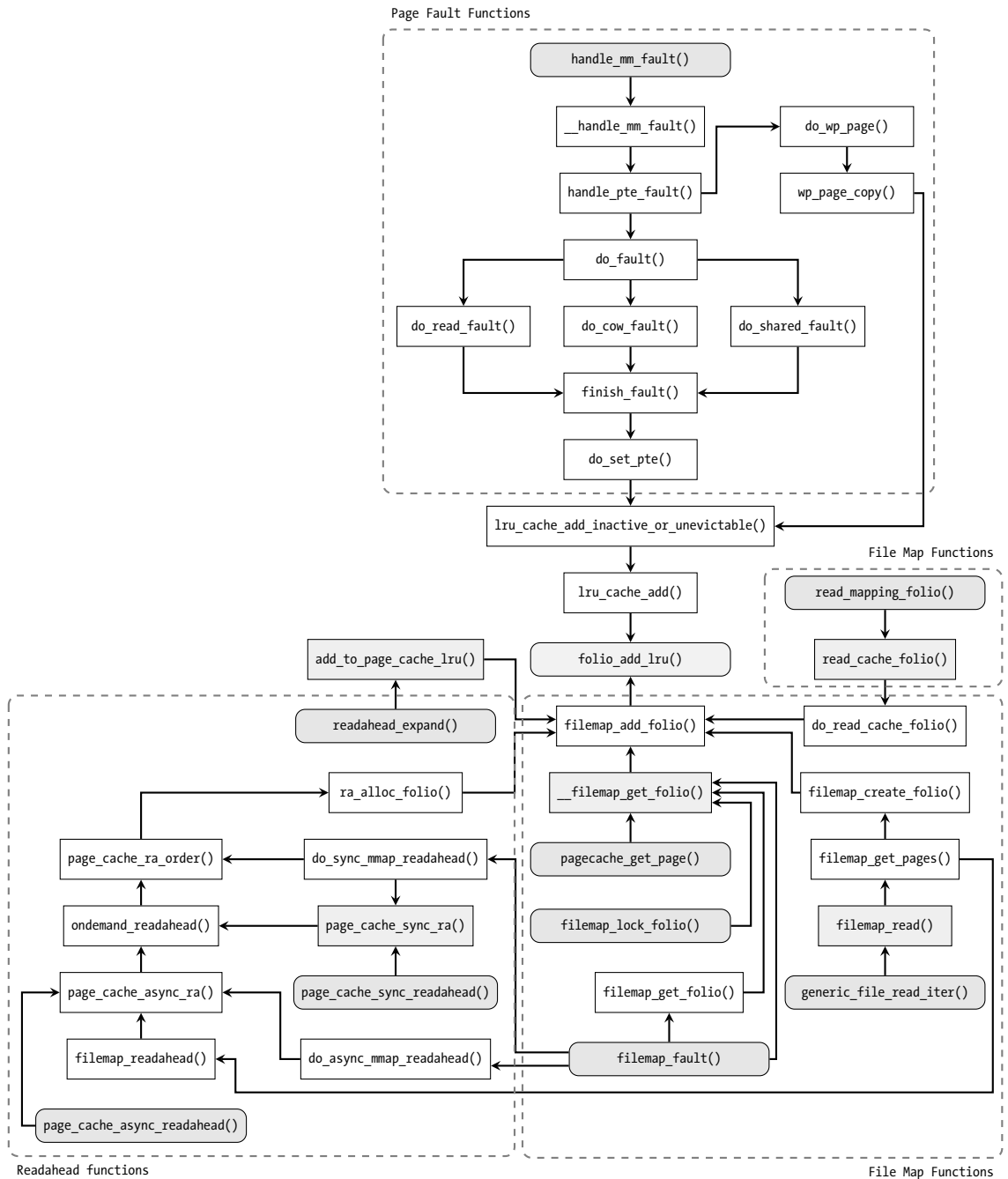


Figure 11-26: Anonymous Folio Invocations of `folio_add_lru()`

Key:

- Denotes the key functions which ultimately trigger a folio being added to an LRU.
- Denotes `folio_add_lru()`.
- Denotes a function that has specific file system and in some cases shmem or secretmem callers which we elide for brevity.

Figure 11-27: File-Backed Folio Invocations of `folio_add_lru()`Examining `lru_add_fn()`:-

```

192 static void lru_add_fn(struct lruvec *lruvec, struct folio *folio)

```

```

193 {
194     int was_unevictable = folio_test_clear_unevictable(folio);
195     long nr_pages = folio_nr_pages(folio);
196
197     VM_BUG_ON_FOLIO(folio_test_lru(folio), folio);
198
199     /*
200      * Is an smp_mb_after_atomic() still required here, before
201      * folio_evictable() tests the mlocked flag, to rule out the
202      * possibility
203      * of stranding an evictable folio on an unevictable LRU? I think
204      * not, because __munlock_page() only clears the mlocked flag
205      * while the LRU lock is held.
206      * (That is not true of __page_cache_release(), and not necessarily
207      * true of release_pages(): but those only clear the mlocked flag
208      * after
209      */
210     if (folio_evictable(folio)) {
211         if (was_unevictable)
212             __count_vm_events(UNEVICTABLE_PGRESCUED, nr_pages);
213     } else {
214         folio_clear_active(folio);
215         folio_set_unevictable(folio);
216         /*
217          * folio->mlock_count = !!folio_test_mlocked(folio)?
218          * But that leaves __mlock_page() in doubt whether another
219          * actor has already counted the mlock or not. Err on the
220          * safe side, underestimate, let page reclaim fix it, rather
221          * than leaving a page on the unevictable LRU indefinitely.
222          */
223         folio->mlock_count = 0;
224         if (!was_unevictable)
225             __count_vm_events(UNEVICTABLE_PGCULLED, nr_pages);
226     }
227
228     lruvec_add_folio(lruvec, folio);
229     trace_mm_lru_insertion(folio);
230 }

```

Listing 11-89: mm/swap.c: `lru_add_fn()`

The logic is as follows:-

1. Test and clear the PG_unevictable folio flag via `folio_test_clear_unevictable()`. This is done as the folio may have been unevictable before but has become evictable (its PG_mlocked flag has been cleared and/or its mapping has had its unevictable status cleared).

2. Asserts that the folio is not on an LRU already via `folio_test_lru()`, as if this were the case we will have encountered a serious kernel bug.
3. Determines whether the folio is evictable via `folio_evictable()`, which ensures the folio does *not* have its mapping field marked unevictable nor has been marked `mlocked`.
4. If the folio is not evictable, we clear the active folio flag, set the `PG_unevictable` folio flag reset the `mlock_count` and update statistics.
5. Finally, we perform the actual task of adding the folio to an LRU list via `lruvec_add_folio()`.

11.7.6 Folio Activation

Examining `folio_activate()`:-

```

369 static void folio_activate(struct folio *folio)
370 {
371     if (folio_test_lru(folio) && !folio_test_active(folio) &&
372         !folio_test_unevictable(folio)) {
373         struct folio_batch *fbatch;
374
375         folio_get(folio);
376         local_lock(&cpu_fbatches.lock);
377         fbatch = this_cpu_ptr(&cpu_fbatches.activate);
378         folio_batch_add_and_move(fbatch, folio, folio_activate_fn);
379         local_unlock(&cpu_fbatches.lock);
380     }
381 }
```

Listing 11-90: mm/swap.c: folio_activate()

This function is called when a folio needs to be proactively activated, i.e. placed on the active LRU list in this folio's `lruvec`. The logic is straightforward – if the folio is marked LRU, is not already activated and is not unevictable, then increment its reference number (to be decremented on drain by `folio_batch_move_lru()`), and invoke `folio_batch_add_and_move()` to add the folio to the activation batch under a local lock.

Examining `folio_activate_fn()`, the function invoked when the batch is drained:-

```

344 static void folio_activate_fn(struct lruvec *lruvec, struct folio *folio)
345 {
346     if (!folio_test_active(folio) && !folio_test_unevictable(folio)) {
347         long nr_pages = folio_nr_pages(folio);
348
349         lruvec_del_folio(lruvec, folio);
350         folio_set_active(folio);
351         lruvec_add_folio(lruvec, folio);

```



```

352         trace_mm_lru_activate(folio);
353
354         __count_vm_events(PGACTIVATE, nr_pages);
355         __count_memcg_events(lruvec_memcg(lruvec), PGACTIVATE,
356                             nr_pages);
357     }
358 }

```

Listing 11-91: mm/swap.c: folio_activate_fn()

We start by checking again whether the folio is active and evictable, this is run under a local lock thus with preemption disabled and so we should not encounter this changing underneath us.

We then simply remove the folio from its current LRU list via `lruvec_del_folio()`, mark the folio's active flag via `folio_set_active()`, before adding it back to the active LRU list via `lruvec_add_folio()`.

We additionally mark the event in **tracefs** via `trace_mm_lru_activate()` and update statistics.

`folio_activate()` is invoked only by `folio_mark_accessed()`, which is a key function used by the kernel to manually cycle folios through the inactive, inactive/referenced, active/unreferenced states (see figure 11-7 and listing 11-11).

11.7.7 Folio Rotation

We examine `folio_rotate_reclaimable()` in Listing 11-92.

```

276 /*
277  * Writeback is about to end against a folio which has been marked for
278  * immediate reclaim. If it still appears to be reclaimable, move it
279  * to the tail of the inactive list.
280  *
281  * folio_rotate_reclaimable() must disable IRQs, to prevent nasty races.
282  */
283 void folio_rotate_reclaimable(struct folio *folio)
284 {
285     if (!folio_test_locked(folio) && !folio_test_dirty(folio) &&
286         !folio_test_unevictable(folio) && folio_test_lru(folio)) {
287         struct folio_batch *fbatch;
288         unsigned long flags;
289
290         folio_get(folio);
291         local_lock_irqsave(&lru_rotate.lock, flags);
292         fbatch = this_cpu_ptr(&lru_rotate.fbatch);
293         folio_batch_add_and_move(fbatch, folio, lru_move_tail_fn);
294         local_unlock_irqrestore(&lru_rotate.lock, flags);
295     }
296 }

```

Listing 11-92: mm/swap.c: folio_rotate_reclaimable()

This function is called in `folio_end_writeback()` when *writeback* is about to end, i.e. when data in a dirty folio (one with data which has changed relative to the file backing it on-disk) has been written back, if and only if the folio has also been marked with the *reclaim* flag, i.e. it is currently subject to reclaim and thus about to be evicted:-

```

1599 void folio_end_writeback(struct folio *folio)
1600 {
1601     /*
1602      * folio_test_clear_reclaim() could be used here but it is an
1603      * atomic operation and overkill in this particular case. Failing
1604      * to shuffle a folio marked for immediate reclaim is too mild
1605      * a gain to justify taking an atomic operation penalty at the
1606      * end of every folio writeback.
1607      */
1608     if (folio_test_reclaim(folio)) {
1609         folio_clear_reclaim(folio);
1610         folio_rotate_reclaimable(folio);
1611     }
1612     :
1627 }
```

Listing 11-93: mm/filemap.c: Excerpt of folio_end_writeback()

The folio may not currently be present at the end of the inactive list (i.e. liable for reclaim) as reclaim may have been blocked on writeback, so this function ensures that, once writeback has been completed, the folio is made immediately available for reclaim.

Note that `folio_rotate_reclaimable()` asserts that the folio must be unlocked, clean (i.e. no further writeback is to be performed), evictable and currently present on an LRU. We invoke `folio_batch_add_and_move()` with both preemption disabled *and* interrupts disabled to avoid unfortunate race conditions.

It, as usual, increments a reference count in the folio (to be decremented on drain by `folio_batch_move_lru()`), and invoke `folio_batch_add_and_move()` to add the folio to the activation batch under a local lock.

Examining `lru_move_tail_fn()`:-

```

266 static void lru_move_tail_fn(struct lruvec *lruvec, struct folio *folio)
267 {
268     if (!folio_test_unevictable(folio)) {
269         lruvec_del_folio(lruvec, folio);
270         folio_clear_active(folio);
271         lruvec_add_folio_tail(lruvec, folio);
272         __count_vm_events(PGROTATED, folio_nr_pages(folio));
273     }
```

274 }

Listing 11-94: mm/swap.c: lru_move_tail_fn()

We reassert the unevictable test to ensure that the folio hasn't since been made unevictable, remove the folio from its current LRU, deactivate it and add it to the tail of the inactive LRU list as shown in listing 11-14.

11.7.8 Folio Deactivation

Examining deactivate_page():-

```

679 /*
680  * deactivate_page - deactivate a page
681  * @page: page to deactivate
682  *
683  * deactivate_page() moves @page to the inactive list if @page was on the
        active
684  * list and was not an unevictable page. This is done to accelerate the
        reclaim
685  * of @page.
686  */
687 void deactivate_page(struct page *page)
688 {
689     struct folio *folio = page_folio(page);
690
691     if (folio_test_lru(folio) && folio_test_active(folio) &&
692         !folio_test_unevictable(folio)) {
693         struct folio_batch *fbatch;
694
695         folio_get(folio);
696         local_lock(&cpu_fbatches.lock);
697         fbatch = this_cpu_ptr(&cpu_fbatches.lru_deactivate);
698         folio_batch_add_and_move(fbatch, folio, lru_deactivate_fn);
699         local_unlock(&cpu_fbatches.lock);
700     }
701 }
```

Listing 11-95: mm/swap.c: deactivate_page()

This asserts that the folio is on an LRU, active and evictable, then performs the usual reference count increment, local lock acquisition (preemption disabled only in this case), before invoking folio_batch_add_and_move() to perform batch population (and draining with lru_deactivate_fn()).

Examining lru_deactivate_fn():-

```

576 static void lru_deactivate_fn(struct lruvec *lruvec, struct folio *folio)
577 {
578     if (folio_test_active(folio) && !folio_test_unevictable(folio)) {
```

```

579         long nr_pages = folio_nr_pages(folio);
580
581         lruvec_del_folio(lruvec, folio);
582         folio_clear_active(folio);
583         folio_clear_referenced(folio);
584         lruvec_add_folio(lruvec, folio);
585
586         __count_vm_events(PGDEACTIVATE, nr_pages);
587         __count_memcg_events(lruvec_memcg(lruvec), PGDEACTIVATE,
588                             nr_pages);
589     }
590 }

```

Listing 11-96: mm/swap.c: lru_deactivate_fn()

This asserts that the folio is still active, and is not unevictable. The LRU then has its active and referenced flags cleared before being added to the inactive list.

This is used by the DAMON monitoring tool (see later chapter on this mechanism) invoked in `damon_pa_deactivate_pages()`, or in the `madvise()` function `madvise_cold_or_pageout_pte_range()` invoked when the `MADV_COLD` or `MADV_PAGEOUT` `madvise` options are used.

11.7.9 File Folio Deactivation

Examining `deactivate_file_folio()`:-

```

654 /**
655  * deactivate_file_folio() - Deactivate a file folio.
656  * @folio: Folio to deactivate.
657  *
658  * This function hints to the VM that @folio is a good reclaim candidate,
659  * for example if its invalidation fails due to the folio being dirty
660  * or under writeback.
661  *
662  * Context: Caller holds a reference on the folio.
663  */
664 void deactivate_file_folio(struct folio *folio)
665 {
666     struct folio_batch *fbatch;
667
668     /* Deactivating an unevictable folio will not accelerate reclaim */
669     if (folio_test_unevictable(folio))
670         return;
671
672     folio_get(folio);
673     local_lock(&cpu_fbatches.lock);
674     fbatch = this_cpu_ptr(&cpu_fbatches.lru_deactivate_file);

```

```

675         folio_batch_add_and_move(fbatch, folio, lru_deactivate_file_fn);
676         local_unlock(&cpu_fbatches.lock);
677     }

```

Listing 11-97: mm/swap.c: deactivate_file_folio()

Note that we do not make assertions about the state of the folio here other than ensuring it is evictable, unlike other LRU operations, we merely disable preemption via a local lock and invoke `folio_batch_add_and_move()` as usual.

Examining `lru_deactivate_file_fn()`:-

```

514  /*
515   * If the folio cannot be invalidated, it is moved to the
516   * inactive list to speed up its reclaim. It is moved to the
517   * head of the list, rather than the tail, to give the flusher
518   * threads some time to write it out, as this is much more
519   * effective than the single-page writeout from reclaim.
520   *
521   * If the folio isn't mapped and dirty/writeback, the folio
522   * could be reclaimed asap using the reclaim flag.
523   *
524   * 1. active, mapped folio -> none
525   * 2. active, dirty/writeback folio -> inactive, head, reclaim
526   * 3. inactive, mapped folio -> none
527   * 4. inactive, dirty/writeback folio -> inactive, head, reclaim
528   * 5. inactive, clean -> inactive, tail
529   * 6. Others -> none
530   *
531   * In 4, it moves to the head of the inactive list so the folio is
532   * written out by flusher threads as this is much more efficient
533   * than the single-page writeout from reclaim.
534   */
535 static void lru_deactivate_file_fn(struct lruvec *lruvec, struct folio *folio)
536 {
537     bool active = folio_test_active(folio);
538     long nr_pages = folio_nr_pages(folio);
539
540     if (folio_test_unevictable(folio))
541         return;
542
543     /* Some processes are using the folio */
544     if (folio_mapped(folio))
545         return;
546
547     lruvec_del_folio(lruvec, folio);
548     folio_clear_active(folio);
549     folio_clear_referenced(folio);
550 }

```

```

551         if (folio_test_writeback(folio) || folio_test_dirty(folio)) {
552             /*
553              * Setting the reclaim flag could race with
554              * folio_end_writeback() and confuse readahead. But the
555              * race window is _really_ small and it's not a critical
556              * problem.
557              */
558             lruvec_add_folio(lruvec, folio);
559             folio_set_reclaim(folio);
560         } else {
561             /*
562              * The folio's writeback ended while it was in the batch.
563              * We move that folio to the tail of the inactive list.
564              */
565             lruvec_add_folio_tail(lruvec, folio);
566             __count_vm_events(PGROTATED, nr_pages);
567         }
568
569         if (active) {
570             __count_vm_events(PGDEACTIVATE, nr_pages);
571             __count_memcg_events(lruvec_memcg(lruvec), PGDEACTIVATE,
572                                nr_pages);
573         }
574     }

```

Listing 11-98: mm/swap.c: lru_deactivate_file_fn()

This function is fairly meaty, taking into account the fact that file-backed folios involve more logic to account for file system operations. We begin by ensuring the folio is neither unevictable (via `folio_test_unevictable()`) nor mapped (via the useful function `folio_mapped()` which determines whether a folio is mapped into userspace). we cannot touch unevictable folios nor should we try to deactivate a folio which is currently mapped into userspace.

We then remove the folio from the folio batch LRU list, clear its active and referenced flags and either mark it for reclaim via `folio_set_reclaim()` if the folio is undergoing writeback (i.e. it was dirtied and is being written back to disk) or is dirty (i.e. has changes to it not yet written back to disk), or if neither of these are the case put immediately to the tail end of the inactive LRU list indicating that the folio will be reclaimed next.

The only function which invokes `deactivate_file_folio()` is `invalidate_mapping_pagevec()` which is called when an inode needs to have all its mappings invalidated including those on folio batches.

11.7.10 Folio Lazy Free

Examining `mark_page_lazyfree()`:-

703 /**

```

704 * mark_page_lazyfree - make an anon page lazyfree
705 * @page: page to deactivate
706 *
707 * mark_page_lazyfree() moves @page to the inactive file list.
708 * This is done to accelerate the reclaim of @page.
709 */
710 void mark_page_lazyfree(struct page *page)
711 {
712     struct folio *folio = page_folio(page);
713
714     if (folio_test_lru(folio) && folio_test_anon(folio) &&
715         folio_test_swapbacked(folio) && !folio_test_swapcache(folio) &&
716         !folio_test_unevictable(folio)) {
717         struct folio_batch *fbatch;
718
719         folio_get(folio);
720         local_lock(&cpu_fbatches.lock);
721         fbatch = this_cpu_ptr(&cpu_fbatches.lru_lazyfree);
722         folio_batch_add_and_move(fbatch, folio, lru_lazyfree_fn);
723         local_unlock(&cpu_fbatches.lock);
724     }
725 }

```

Listing 11-99: mm/swap.c: mark_page_lazyfree()

This performs a great number of tests, ensuring that the folio in question is on an LRU, anonymous, swap-backed (anonymous folios are *always* swap-backed except when being lazy freed), not on the swap cache (more on what the swap cache is in the swap chapter) and not unevictable. It then performs the usual folio reference count increment and disabling of preemption by applying a local lock.

This function is used, as the name implies, to lazily free folios. It is used by the `madvise()` system call when memory is flagged `MADV_DONTNEED` via `madvise_dontneed_free()` and `madvise_free_single_vma()` and `madvise_free_pte_range()` in turn (via `walk_page_range()` using the `madvise_free_walk_ops` callbacks). It is also used for huge page cases in `madvise_free_huge_pmd()`.

Examining `lru_lazyfree_fn()`:-

```

592 static void lru_lazyfree_fn(struct lruvec *lruvec, struct folio *folio)
593 {
594     if (folio_test_anon(folio) && folio_test_swapbacked(folio) &&
595         !folio_test_swapcache(folio) && !folio_test_unevictable(folio)) {
596         long nr_pages = folio_nr_pages(folio);
597
598         lruvec_del_folio(lruvec, folio);
599         folio_clear_active(folio);
600         folio_clear_referenced(folio);
601         /*

```

```

602             * Lazyfree folios are clean anonymous folios. They have
603             * the swapbacked flag cleared, to distinguish them from
        normal
604             * anonymous folios
605             */
606             folio_clear_swapbacked(folio);
607             lruvec_add_folio(lruvec, folio);
608
609             __count_vm_events(PGLAZYFREE, nr_pages);
610             __count_memcg_events(lruvec_memcg(lruvec), PGLAZYFREE,
611                                nr_pages);
612         }
613     }

```

Listing 11-100: mm/swap.c: *lru_lazyfree_fn()*

This reasserts, under lruvec lock, that the folio is anonymous, swap-backed, not swap cache and evictable.

The folio is then removed from its current LRU list, deactivated, dereferenced, then uniquely they have their swapbacked status cleared before being added back to an LRU list, which in this case will be the inactive *file* LRU list as *folio_lru_list()* will identify them as such when being added via *lruvec_add_folio()*.

The lazy free folios are checked explicitly in reclaim in *shrink_page_list()* (see Listing 11-63).

11.7.11 mlock Folio Batch

The *mlock()* logic maintains a separate folio batch used for moving folios to the unevictable LRU list (in fact no such LRU list exists, but the folios will be marked unevictable at any rate and removed from all LRU lists).

Note that this logic uses the legacy naming scheme for folio batches – page vectors or *pagevecs*. These are exactly equivalent to folio batches, so substitute ‘folio batch’ for ‘pagevec’ whenever you see it.

The functions *__mlock_page()*, *__mlock_new_page()* and *__munlock_page()* all interact with LRU lists directly via *add_page_to_lru_list()* and *del_page_from_lru_list()* (each of which ultimately invoke *lruvec_add_folio()* and *lruvec_del_folio()* respectively).

These perform the *actual* lruvec operations and are invoked when the folio batch is drained, either when the batch is full or a drain is triggered directly:-

```

61 static struct lruvec *__mlock_page(struct page *page, struct lruvec *lruvec)
62 {
63     /* There is nothing more we can do while it's off LRU */
64     if (!TestClearPageLRU(page))
65         return lruvec;
66
67     lruvec = folio_lruvec_relock_irq(page_folio(page), lruvec);

```



```

68
69     if (unlikely(page_evictable(page))) {
70         /*
71          * This is a little surprising, but quite possible:
72          * PageMlocked must have got cleared already by another CPU.
73          * Could this page be on the Unevictable LRU? I'm not sure,
74          * but move it now if so.
75          */
76         if (PageUnevictable(page)) {
77             del_page_from_lru_list(page, lruvec);
78             ClearPageUnevictable(page);
79             add_page_to_lru_list(page, lruvec);
80             __count_vm_events(UNEVICTABLE_PGRESCUED,
81                             thp_nr_pages(page));
82         }
83         goto out;
84     }
85
86     if (PageUnevictable(page)) {
87         if (PageMlocked(page))
88             page->mlock_count++;
89         goto out;
90     }
91
92     del_page_from_lru_list(page, lruvec);
93     ClearPageActive(page);
94     SetPageUnevictable(page);
95     page->mlock_count = !!PageMlocked(page);
96     add_page_to_lru_list(page, lruvec);
97     __count_vm_events(UNEVICTABLE_PGCULLED, thp_nr_pages(page));
98 out:
99     SetPageLRU(page);
100    return lruvec;
101 }

```

Listing 11-101: mm/mlock.c: `__mlock_page()`

The logic is:-

1. Assert that the folio is on an LRU and atomically mark it non-LRU, abort if it was already non-LRU.
2. Obtain the lruvec for the folio if we do not already have it, and lock it if not already locked, via `folio_lruvec_relock_irq()`
3. Check whether the page is evictable via `page_evictable()` which checks both the `PG_mlocked` folio lock as well as the `unevictable` flag in the folio mapping field (see the section on `mlock()` for more details on this). At this stage, the folio should at least be flagged as `mlocked`, so we do not expect this to be the case. We then exit early.

- (a) We handle an edge case here – if the `PG_mlocked` flag has been cleared by another core racing this operation, we ‘rescue’ the folio by clearing the `unevictable` folio flag (remember that `lruvec_add_folio()` uses `folio_lru_list()` to determine which LRU list to place the folio on – with the `unevictable` folio flag set, it will not place on an active/inactive LRU list, with it cleared it will).
4. If the page was already marked `unevictable` and `mlocked`, we increment the folio’s `mlock_count` field and exit early.
5. We hit the normal case – we extract the folio from its current LRU list, clear its active folio flag and mark it `unevictable`, setting the `mlock_count` depending on whether the folio flag is currently set, and invoke `add_page_to_lru_list()` which will simply update statistics.
6. Finally, we set the LRU folio flag to mark that the folio is now on the virtual `unevictable` LRU list.

`__mlock_new_page()` is invoked for newly allocated folios and thus is significantly simpler:-

```

103 static struct lruvec *__mlock_new_page(struct page *page, struct lruvec *
    lruvec)
104 {
105     VM_BUG_ON_PAGE(PageLRU(page), page);
106
107     lruvec = folio_lruvec_relock_irq(page_folio(page), lruvec);
108
109     /* As above, this is a little surprising, but possible */
110     if (unlikely(page_evictable(page)))
111         goto out;
112
113     SetPageUnevictable(page);
114     page->mlock_count = !!PageMlocked(page);
115     __count_vm_events(UNEVICTABLE_PGCULLED, thp_nr_pages(page));
116 out:
117     add_page_to_lru_list(page, lruvec);
118     SetPageLRU(page);
119     return lruvec;
120 }
```

Listing 11-102: `mm/mlock.c: __mlock_new_page()`

Finally, `__munlock_page()` performs the same operations as `__mlock_page()` in reverse.

These functions all interact with the `mlock_pvec` folio batch dedicated to `mlock()` operations. The central drain function invokes the specific drain function is `mlock_pagevec()`:-

```

179 /*
180  * mlock_pagevec() is derived from pagevec_lru_move_fn():
```

```

181  * perhaps that can make use of such page pointer flags in future,
182  * but for now just keep it for mlock. We could use three separate
183  * pagevecs instead, but one feels better (munlocking a full pagevec
184  * does not need to drain mlocking pagevecs first).
185  */
186  static void mlock_pagevec(struct pagevec *pvec)
187  {
188      struct lruvec *lruvec = NULL;
189      unsigned long mlock;
190      struct page *page;
191      int i;
192
193      for (i = 0; i < pagevec_count(pvec); i++) {
194          page = pvec->pages[i];
195          mlock = (unsigned long)page & (LRU_PAGE | NEW_PAGE);
196          page = (struct page *)((unsigned long)page - mlock);
197          pvec->pages[i] = page;
198
199          if (mlock & LRU_PAGE)
200              lruvec = __mlock_page(page, lruvec);
201          else if (mlock & NEW_PAGE)
202              lruvec = __mlock_new_page(page, lruvec);
203          else
204              lruvec = __munlock_page(page, lruvec);
205      }
206
207      if (lruvec)
208          unlock_page_lruvec_irq(lruvec);
209      release_pages(pvec->pages, pvec->nr);
210      pagevec_reinit(pvec);
211  }

```

Listing 11-103: mm/mlock.c: mlock_pagevec()

The lower two bits of each folio in the folio batch are used to indicate whether the folio is intended to invoke `__mlock_page()` (`LRU_PAGE` is set), `__mlock_new_page()` (`NEW_PAGE` is set), or `__munlock_page()` (no lower bit flags are set).

This function invokes the individual drain functions as necessary, releases any lruvec lock that is held via `unlock_page_lruvec_irq()`, decrements the folio reference count via `release_pages()`, and clears the folio batch ready for use via `pagevec_reinit()`.

The functions which ultimately invoke `mlock_pagevec()` when the folio batch is full are as follows:-

```

239  /**
240   * mlock_folio - mlock a folio already on (or temporarily off) LRU
241   * @folio: folio to be mlocked.
242   */

```

```

243 void mlock_folio(struct folio *folio)
244 {
245     struct pagevec *pvec;
246
247     local_lock(&mlock_pvec.lock);
248     pvec = this_cpu_ptr(&mlock_pvec.vec);
249
250     if (!folio_test_set_mlocked(folio)) {
251         int nr_pages = folio_nr_pages(folio);
252
253         zone_stat_mod_folio(folio, NR_MLOCK, nr_pages);
254         __count_vm_events(UNEVICTABLE_PGMLOCKED, nr_pages);
255     }
256
257     folio_get(folio);
258     if (!pagevec_add(pvec, mlock_lru(&folio->page)) ||
259         folio_test_large(folio) || lru_cache_disabled())
260         mlock_pagevec(pvec);
261     local_unlock(&mlock_pvec.lock);
262 }

```

Listing 11-104: mm/mlock.c: mlock_folio()

```

264 /**
265  * mlock_new_page - mlock a newly allocated page not yet on LRU
266  * @page: page to be mlocked, either a normal page or a THP head.
267  */
268 void mlock_new_page(struct page *page)
269 {
270     struct pagevec *pvec;
271     int nr_pages = thp_nr_pages(page);
272
273     local_lock(&mlock_pvec.lock);
274     pvec = this_cpu_ptr(&mlock_pvec.vec);
275     SetPageMlocked(page);
276     mod_zone_page_state(page_zone(page), NR_MLOCK, nr_pages);
277     __count_vm_events(UNEVICTABLE_PGMLOCKED, nr_pages);
278
279     get_page(page);
280     if (!pagevec_add(pvec, mlock_new(page)) ||
281         PageHead(page) || lru_cache_disabled())
282         mlock_pagevec(pvec);
283     local_unlock(&mlock_pvec.lock);
284 }

```

Listing 11-105: mm/mlock.c: mlock_new_page()

```

286 /**

```

```

287  * munlock_page - munlock a page
288  * @page: page to be munlocked, either a normal page or a THP head.
289  */
290  void munlock_page(struct page *page)
291  {
292      struct pagevec *pvec;
293
294      local_lock(&mlock_pvec.lock);
295      pvec = this_cpu_ptr(&mlock_pvec.vec);
296      /*
297       * TestClearPageMlocked(page) must be left to __munlock_page(),
298       * which will check whether the page is multiply mlocked.
299       */
300
301      get_page(page);
302      if (!pagevec_add(pvec, page) ||
303          PageHead(page) || lru_cache_disabled())
304          mlock_pagevec(pvec);
305      local_unlock(&mlock_pvec.lock);
306  }

```

Listing 11-106: mm/mlock.c: munlock_page()

Each of these functions perform similar operations – they acquire the folio batch local lock to disable preemption, set or clear the PG_mlocked folio flag, update statistics, increment the folio reference count via folio_get() or the legacy equivalent get_page(), before trying to add the folio to the per-CPU lruvec via pagevec_add() (exactly equivalent to folio_batch_add()), foregoing the batch if the folio is compound (tested via folio_test_large()) or the LRU cache is disabled altogether (discussed below). Finally the local lock is released.

One thing that differs here is setting a flag to indicate which operation each folio in the folio batch should have applied to it – mlock_lru() to assign LRU_PAGE and mlock_new() to assign LRU_NEW.

11.7.12 Folio Batch Drain

When folio batches are ‘drained’ they have their drain callback functions invoked and the operation they are batched up for is performed.

Drains can occur individually when a batch is full via folio_batch_add_and_move(), invoking folio_batch_move_lru() on the appropriate callback, as well as when a per-CPU global drain is performed under certain circumstances.

The key function for performing per-CPU global drain is lru_add_drain_cpu() (it assumes preemption has been disabled, typically via a local lock being applied):-

```

615  /*
616  * Drain pages out of the cpu's folio_batch.

```

```

617  * Either "cpu" is the current CPU, and preemption has already been
618  * disabled; or "cpu" is being hot-unplugged, and is already dead.
619  */
620 void lru_add_drain_cpu(int cpu)
621 {
622     struct cpu_fbatches *fbatches = &per_cpu(cpu_fbatches, cpu);
623     struct folio_batch *fbatch = &fbatches->lru_add;
624
625     if (folio_batch_count(fbatch))
626         folio_batch_move_lru(fbatch, lru_add_fn);
627
628     fbatch = &per_cpu(lru_rotate.fbatch, cpu);
629     /* Disabling interrupts below acts as a compiler barrier. */
630     if (data_race(folio_batch_count(fbatch))) {
631         unsigned long flags;
632
633         /* No harm done if a racing interrupt already did this */
634         local_lock_irqsave(&lru_rotate.lock, flags);
635         folio_batch_move_lru(fbatch, lru_move_tail_fn);
636         local_unlock_irqrestore(&lru_rotate.lock, flags);
637     }
638
639     fbatch = &fbatches->lru_deactivate_file;
640     if (folio_batch_count(fbatch))
641         folio_batch_move_lru(fbatch, lru_deactivate_file_fn);
642
643     fbatch = &fbatches->lru_deactivate;
644     if (folio_batch_count(fbatch))
645         folio_batch_move_lru(fbatch, lru_deactivate_fn);
646
647     fbatch = &fbatches->lru_lazyfree;
648     if (folio_batch_count(fbatch))
649         folio_batch_move_lru(fbatch, lru_lazyfree_fn);
650
651     folio_activate_drain(cpu);
652 }

```

Listing 11-107: mm/swap.c: `lru_add_drain_cpu()`

Note that `data_race()` is used to denote that a data race is expected and can be forgiven, explicitly disabling KCSAN checks for this (a topic out of scope for this section).

This invokes `folio_batch_move_lru()` for each callback in turn:-

Folio addition via `lru_add_fn()`.

Folio reclaimable rotation via `lru_move_tail_fn()`.

Folio file deactivation via `lru_deactivate_file_fn()`.

Folio deactivation via `lru_deactivate_fn()`.

Folio lazy free via `lru_lazyfree_fn()`.

Folio activation via `folio_activate_drain()` and `folio_activate_fn()`.

`mlock()` folios are drained separately in `mlock_page_drain_local()` and `mlock_page_drain_remote()` which perform a similar role, only explicitly taking the `mlock_pvec` local lock in the local case, and explicitly checking that the CPU is offline and thus no lock required in the remote one:-

```

213 void mlock_page_drain_local(void)
214 {
215     struct pagevec *pvec;
216
217     local_lock(&mlock_pvec.lock);
218     pvec = this_cpu_ptr(&mlock_pvec.vec);
219     if (pagevec_count(pvec))
220         mlock_pagevec(pvec);
221     local_unlock(&mlock_pvec.lock);
222 }
```

Listing 11-108: mm/mlock.c: `mlock_page_drain_local()`

```

224 void mlock_page_drain_remote(int cpu)
225 {
226     struct pagevec *pvec;
227
228     WARN_ON_ONCE(cpu_online(cpu));
229     pvec = &per_cpu(mlock_pvec.vec, cpu);
230     if (pagevec_count(pvec))
231         mlock_pagevec(pvec);
232 }
```

Listing 11-109: mm/mlock.c: `mlock_page_drain_remote()`

Both of these invoke `mlock_pagevec()` as described in section 11.7.11.

Both the core folio and the `mlock` batch drain functions are invoked from a number of call sites:-

```

727 void lru_add_drain(void)
728 {
729     local_lock(&cpu_fbatches.lock);
730     lru_add_drain_cpu(smp_processor_id());
731     local_unlock(&cpu_fbatches.lock);
732     mlock_page_drain_local();
733 }
```

Listing 11-110: mm/swap.c: `lru_add_drain()`

This is the *key* drain function. It applies a local lock before invoking `lru_add_drain_cpu()` then invoking `mlock_page_drain_local()` to take care of `mlock` folios.

The function `lru_add_and_bh_lrus_drain()` does the same thing only it also invokes invalidation of buffer head LRU lists via `invalidate_bh_lrus_cpu()`. This is out of scope for this section:-

```

735 /*
736  * It's called from per-cpu workqueue context in SMP case so
737  * lru_add_drain_cpu and invalidate_bh_lrus_cpu should run on
738  * the same cpu. It shouldn't be a problem in !SMP case since
739  * the core is only one and the locks will disable preemption.
740  */
741 static void lru_add_and_bh_lrus_drain(void)
742 {
743     local_lock(&cpu_fbatches.lock);
744     lru_add_drain_cpu(smp_processor_id());
745     local_unlock(&cpu_fbatches.lock);
746     invalidate_bh_lrus_cpu();
747     mlock_page_drain_local();
748 }
```

Listing 11-111: `mm/swap.c: lru_add_and_bh_lrus_drain()`

This function is invoked by `lru_add_drain_per_cpu()`, which is invoked in turn by `__lru_add_drain_all()` (see below for more details on this).

Finally, there is `lru_add_drain_cpu_zone()` which is invoked by compaction in `compact_zone()` (see the compaction chapter for more details on this):-

```

750 void lru_add_drain_cpu_zone(struct zone *zone)
751 {
752     local_lock(&cpu_fbatches.lock);
753     lru_add_drain_cpu(smp_processor_id());
754     drain_local_pages(zone);
755     local_unlock(&cpu_fbatches.lock);
756     mlock_page_drain_local();
757 }
```

Listing 11-112: `mm/swap.c: lru_add_drain_cpu_zone()`

The key difference here is the invocation of `drain_local_pages()` which drains all Per-CPU-Pages (PCPs) to free lists.

So far we've covered draining of folios on a specific CPU, but not draining *all* CPUs. This is handled by `lru_add_drain_all()` which in turn invokes `__lru_add_drain_all()`.

Examining this function, stripping out the careful synchronisation machinery for clarity:-

```

783 /*
784  * Doesn't need any cpu hotplug locking because we do rely on per-cpu
785  * kworkers being shut down before our page_alloc_cpu_dead callback is
786  * executed on the offlined cpu.
787  * Calling this function with cpu hotplug locks held can actually lead
```



```

788  * to obscure indirect dependencies via WQ context.
789  */
790  static inline void __lru_add_drain_all(bool force_all_cpus)
791  {
792      :
863      cpumask_clear(&has_work);
864      for_each_online_cpu(cpu) {
865          struct work_struct *work = &per_cpu(lru_add_drain_work, cpu);
866
867          if (cpu_needs_drain(cpu)) {
868              INIT_WORK(work, lru_add_drain_per_cpu);
869              queue_work_on(cpu, mm_percpu_wq, work);
870              __cpumask_set_cpu(cpu, &has_work);
871          }
872      }
873
874      for_each_cpu(cpu, &has_work)
875          flush_work(&per_cpu(lru_add_drain_work, cpu));
876      :
879  }

```

Listing 11-113: mm/swap.c: `__lru_add_drain_all()`

This uses the `lru_add_drain_work` work queue to perform the work asynchronously*. These each in turn invoke `lru_add_drain_per_cpu()` which simply invokes, in turn, `lru_add_and_bh_lrus_drain()` described above.

This function determines whether to perform a drain via `cpu_needs_drain()`:-

```

768  static bool cpu_needs_drain(unsigned int cpu)
769  {
770      struct cpu_fbatches *fbatches = &per_cpu(cpu_fbatches, cpu);
771
772      /* Check these in order of likelihood that they're not zero */
773      return folio_batch_count(&fbatches->lru_add) ||
774             data_race(folio_batch_count(&per_cpu(lru_rotate.fbatch, cpu)))
775             ||
776             folio_batch_count(&fbatches->lru_deactivate_file) ||
777             folio_batch_count(&fbatches->lru_deactivate) ||
778             folio_batch_count(&fbatches->lru_lazyfree) ||
779             folio_batch_count(&fbatches->activate) ||
780             need_mlock_page_drain(cpu) ||
781             has_bh_in_lru(cpu, NULL);
782  }

```

Listing 11-114: mm/swap.c: `cpu_needs_drain()`

*. Work queues are out of scope for this book, but are a highly documented part of the kernel.

This checks to see if any batch has available folios to flush, as well as checking the equivalent for mlock folios via `need_mlock_page_drain()` and the buffer head equivalent in `has_bh_in_lru()`.

`__lru_add_drain_all()` is invoked from a number of different places, typically via `lru_add_drain_all()`, but is invoked directly by `lru_cache_disable()`.

This function is used to disable folio batching altogether and atomically increments the `lru_disable_count` value, which is used by the `lru_cache_disabled()` function to determine if folio batches are enabled which in turn, if set, makes `folio_batch_add_and_move()` simply immediately invoke `folio_batch_move_lru()`, i.e. effectively pass-through.

`lru_cache_disable()` is paired with `lru_cache_enable()` which simply atomically increments `lru_disable_count`. Use of this pair allows for the systematic disabling/re-enabling of folio batches. Examining `lru_cache_disable()`:-

```

894 /*
895  * lru_cache_disable() needs to be called before we start compiling
896  * a list of pages to be migrated using isolate_lru_page().
897  * It drains pages on LRU cache and then disable on all cpus until
898  * lru_cache_enable is called.
899  *
900  * Must be paired with a call to lru_cache_enable().
901  */
902 void lru_cache_disable(void)
903 {
904     atomic_inc(&lru_disable_count);
905     /*
906      * Readers of lru_disable_count are protected by either disabling
907      * preemption or rcu_read_lock:
908      *
909      * preempt_disable, local_irq_disable [bh_lru_lock()]
910      * rcu_read_lock                      [rt_spin_lock CONFIG_PREEMPT_RT]
911      ]
912      * preempt_disable                    [local_lock !CONFIG_PREEMPT_RT]
913      *
914      * Since v5.1 kernel, synchronize_rcu() is guaranteed to wait on
915      * preempt_disable() regions of code. So any CPU which sees
916      * lru_disable_count = 0 will have exited the critical
917      * section when synchronize_rcu() returns.
918      */
919     synchronize_rcu_expedited();
920 #ifdef CONFIG_SMP
921     __lru_add_drain_all(true);
922 #else
923     lru_add_and_bh_lrus_drain();
924 #endif
925 }
```

Listing 11-115: mm/swap.c: `lru_cache_disable()`

Putting the synchronisation work to one side, we can see that the key operations here are incrementing `lru_disable_count` and invoking `__lru_add_drain_all()`.

The LRU cache is disabled on memory hotplug (in `offline_pages()`), on folio migration (in `do_migrate_pages()`, `do_pages_move()` and `__alloc_contig_migrate_range()`) and on a memory policy being applied via `mbind()` (in `do_mbind()`).

12

SWAP MEMORY

Overcommit and demand paging mean that Linux permits more memory to be mapped by processes than the RAM installed on the system. This means that when a lot of memory is used, the kernel must reclaim memory in order to satisfy future memory requests. Swap provides a means by which to reclaim anonymous memory by writing it out to disk. We will explore the means by which this is achieved in this chapter.

When memory needs to be reclaimed from the page cache, the kernel can simply drop clean pages from it, freeing up memory quickly at the cost of the next access to that page resulting in a major fault.

This isn't the case for anonymous memory, however, which cannot simply be dropped. Instead it is “swapped out” to disk to be “swapped in” later if accessed.

A complexity that arises with this is that the swap out operation takes time to complete, and meanwhile it could be swapped back in at any time.

This means that the kernel must provide some special means of marking memory as being swapped out for some processes which maps it (via “swap

entries”) and designating such memory as being not long for this world (occupying the “swap cache”). We explore both in detail.

The swap is often misunderstood as a means to add memory capacity to a system or act as a memory reserve, however it is neither—its entire purpose in life is to provide the means for anonymous memory to undergo reclaim, thereby providing a mean of balancing memory pressure between anonymous memory and the page cache.

This is important, as under heavy memory pressure reclaiming either page cache folios or anonymous memory alone is likely to cause “thrashing”, and having the ability to reclaim anonymous memory helps address this.

NOTE

Thrashing describes the pathological condition in which heavy memory pressure results in the system being in a perpetual state of faulting data in from disk (whether swapped in or read into the page cache from disk) and writing it back out again (either through swapping it out or writing back dirty file data to disk). This results in the system becoming slow and unstable, and ultimately unusable if the situation is not resolved.

As a result swap is inescapably linked to both reclaim (see Chapter 11) and page faulting (see Chapter 6). We will examine how, under memory pressure, reclaim triggers swap-out, and how, when swapped out memory is accessed, a page fault triggers swap-in.

12.1 The Swap Cache

The page cache acts as a bridge between data that exists on disk and the processes that interact with it, abstracting reading from files (see Chapter ??) and writing back to them when necessary (see Chapter 10).

This simplifies accesses to storage and provides a means of resolving races between processes reading from and writing to files which might undergo reclaim midway through.

The swap cache performs a similar role for anonymous memory, although unlike file pages in the page cache, anonymous memory is not placed in the swap cache until it has been swapped out in the first instance.

The “swap cache” therefore is a repository of folios which reclaim has decided ought to be swapped out, which may or may not yet be swapped out, or may have just been swapped back in. It is the swap cache that resolves these races.

As is customary for this book we will build understanding of the swap cache by examining how it functions “bottom-up”, i.e. examine the data structures and algorithms which implement it, and using this to understand how this fits together with the rest of the swap implementation.

The swap cache is defined by the `swapper_spaces` array, which is an array of `MAX_SWAPFILES` pointers each of which point at an array of `nr_swapper_spaces` `struct address_space` objects, as shown in Listing ??.

```
40 struct address_space *swapper_spaces[MAX_SWAPFILES] __read_mostly;
```

```
41 static unsigned int nr_swapper_spaces[MAX_SWAPFILES] __read_mostly;
```

Listing 12-1: mm/swap_state.c: swapper_spaces and nr_swapper_spaces

In effect, `swapper_spaces` is a 2 dimensional array, with the first dimension being fixed and the second dynamically allocated, whose length is equal to `nr_swapper_spaces[i]` (for the *i*th swap file).

Each dynamic array contained within `swapper_spaces` represents the space available in a specific swap file or partition (for the sake of the swap implementation we refer to all swap backing sources as swap files).

Each struct `address_space` object is used as a “fake” page cache object describing up to a maximum of `SWAP_ADDRESS_SPACE_PAGES` pages each, equal to 16,384 pages or (assuming 4 KiB pages) 64 MiB of data.

Therefore, if we know a specific page offset within a swap file, we need only shift it by `SWAP_ADDRESS_SPACE_SHIFT` bits to determine the index of the struct `address_space` to which it belongs.

If we know the index of the swap file (which is referred to, rather confusingly, as its “type”), and the offset of the page we wish to retrieve from the swap cache within it, we are able to locate the page in `swapper_spaces`.

The combination of these two bits of information is referred to as a “swap entry” and described by the `swp_entry_t` type alias, which simply wraps an unsigned long value, as shown in Listing 12-2. This value is architecture-independent.

```
814 /*
815  * A swap entry has to fit into a "unsigned long", as the entry is hidden
816  * in the "index" field of the swapper address space.
817  */
818 typedef struct {
819     unsigned long val;
820 } swp_entry_t;
```

Listing 12-2: include/linux/mm_types.h: swp_entry_t

Swap entries are encoded via `swp_entry()` as shown in Listing 12-3.

```
38 /*
39  * Store a type+offset into a swp_entry_t in an arch-independent format
40  */
41 static inline swp_entry_t swp_entry(unsigned long type, pgoff_t offset)
42 {
43     swp_entry_t ret;
44
45     ret.val = (type << SWP_TYPE_SHIFT) | (offset & SWP_OFFSET_MASK);
46     return ret;
47 }
```

Listing 12-3: include/linux/swapops.h: swp_entry()

The “type” referred to here, though oddly named, refers to the swap file index. The type values are shifted up to the high bits by `SWP_TYPE_SHIFT`, and

the offset has the `SWP_OFFSET_MASK` applied, limiting its value to that of the unsigned long `swp_entry_t` type less the high bits used for the type value.

We can access swapper struct `address_space` objects via the `swap_address_space()` macro, as shown in Listing 12-4.

```

30 #define swap_address_space(entry) \
31     (&swapper_spaces[swp_type(entry)][swp_offset(entry) \
32         >> SWAP_ADDRESS_SPACE_SHIFT])

```

Listing 12-4: mm/swap.h: `swap_address_space()`

As discussed previously, each struct `address_space` in swapper space manages `SWAP_ADDRESS_SPACE_PAGES` pages, meaning that shifting by `SWAP_ADDRESS_SPACE_SHIFT` obtains the correct index within the dynamic array.

We determine the index of the swap file (or its “type”) via `swp_type()` which we examine in Listing 12-5.

```

49 /*
50  * Extract the `type' field from a swp_entry_t. The swp_entry_t is in
51  * arch-independent format
52  */
53 static inline unsigned swp_type(swp_entry_t entry)
54 {
55     return (entry.val >> SWP_TYPE_SHIFT);
56 }

```

Listing 12-5: include/linux/swapops.h: `swp_type()`

This simply shifts the `swp_entry_t` to extract the type value. We extract the page offset from the entry via `swp_offset()` which we examine in Listing 12-6.

```

58 /*
59  * Extract the `offset' field from a swp_entry_t. The swp_entry_t is in
60  * arch-independent format
61  */
62 static inline pgoff_t swp_offset(swp_entry_t entry)
63 {
64     return entry.val & SWP_OFFSET_MASK;
65 }

```

Listing 12-6: include/linux/swapops.h: `swp_offset()`

Both of these functions simply invert what was performed in `swp_entry()` (see Listing 12-3) to obtain the component type and offset fields contained within the entry.

12.1.1 Swapper Initialisation

A swap file is made available to the kernel via the `swapon()` system call, which in turn invokes `init_swap_address_space()`, which we examine in Listing 12-7.

```

661 int init_swap_address_space(unsigned int type, unsigned long nr_pages)

```

```

662 {
663     struct address_space *spaces, *space;
664     unsigned int i, nr;
665
666     nr = DIV_ROUND_UP(nr_pages, SWAP_ADDRESS_SPACE_PAGES);
667     spaces = kvcalloc(nr, sizeof(struct address_space), GFP_KERNEL);
668     if (!spaces)
669         return -ENOMEM;
670     for (i = 0; i < nr; i++) {
671         space = spaces + i;
672         xa_init_flags(&space->i_pages, XA_FLAGS_LOCK_IRQ);
673         atomic_set(&space->i_mmap_writable, 0);
674         space->a_ops = &swap_aops;
675         /* swap cache doesn't use writeback related tags */
676         mapping_set_no_writeback_tags(space);
677     }
678     nr_swapper_spaces[type] = nr;
679     swapper_spaces[type] = spaces;
680
681     return 0;
682 }

```

Listing 12-7: mm/swap_state.c: init_swap_address_space()

This function is passed the index of the swapper file (it’s “type”), and the number of pages the swap file is able to store, which is used to calculate the number of struct address_space objects that are required to describe these available pages.

We allocate the space required to store these objects via kvcalloc() which allocates using the kernel slab allocator (the kernel equivalent of malloc()), equal to the number of pages divided by SWAP_ADDRESS_SPACE_PAGES, rounding up.

Each of the “fake” struct address_space objects are initialised, disabling writeback functionality as this is not utilised, and importantly setting the struct address_space_operations to swap_aops, which importantly sets the writepage callback to swap_writepage() which we examine later in Listing 12-27.

NOTE

The struct address_space associated with each entry in the swapper_spaces arrays are independent of those describing the underlying swap file or partition, they are entirely an abstraction.

We then set the nr_swapper_spaces field to the page count, and then assign our newly allocated struct address_space array to the appropriate entry in swapper_spaces.

12.1.2 Assigning Folios to the Swap Cache

Once we have placed a folio into the swap cache, we need to be able to look up the swap cache entry it belongs to. We do so by setting the `struct page->private` field for each subpage of the folio (see Section 2.2 for a detailed explanation of folios and how they relate to their subpages) to a `swp_entry_t` value which in turn allows us to look up the swap cache entry.

NOTE

All of the subpages of a folio will be placed in the same struct address_space as consecutive pages. Therefore the swap entries for the folio will be contiguous from the from struct folio->private up to the number of pages the folio spans.

This is done in `add_to_swap_cache()` which we examine in Listing 12-22 below in Section 12.2. This function simply invokes `set_page_private()` to do so.

This field can be used by file mappings, but since only anonymous memory can be swapped out, it is safe for us to use it. Memory that is present in the swap cache also has the folio flag `PG_swapcache` set, so we know if this is set that we ought to look at `struct folio->private` rather than `struct folio->mapping`.

This allows us to cleverly share some of the file handling code and apply it to swap cache entries, most importantly within reclaim, where the dirty state of swapper space `struct address_space` objects determines whether a swap cache entry needs to be written to disk (see Section 12.2 for details).

The utility function `folio_mapping()` (see Listing 7-41) abstracts this swap lookup. For convenience we show the relevant part of this function here in Listing 12-8.

```

799 struct address_space *folio_mapping(struct folio *folio)
800 {
801     :
807     if (unlikely(folio_test_swapcache(folio)))
808         return swap_address_space(folio_swap_entry(folio));
809     :
815 }
```

Listing 12-8: `mm/util.c: folio_mapping()` Swap Cache Handling

This invokes `swap_address_space()` (as shown in Listing 12-4) to look up the entry, which in turn obtains the `swp_entry_t` entry via `folio_swap_entry()` which we examine in Listing 12-9.

```

348 static inline swp_entry_t folio_swap_entry(struct folio *folio)
349 {
350     swp_entry_t entry = { .val = page_private(&folio->page) };
351     return entry;
352 }
```

Listing 12-9: `include/linux/swap.h: folio_swap_entry()`

Which simply looks up the swap entry from `struct folio->private` via `page_private()`.

12.1.3 Page Table Mappings

When memory is swapped out it doesn't get unmapped, but rather the page table mappings are updated to reference the swap cache by encoding a `swp_entry_t` at the PTE level (see Chapter 3 for details on page table levels), or equivalent for huge pages.

Since the `swp_entry_t` spans the size of an unsigned long type and a page table entry is typically equally as long, we must sacrifice some bits of the offset portion of the swap entry to permit each architecture to set hardware-specific page table bits accordingly.

We identify swap PTEs using `is_swap_pte()`, which we examine in Listing 12-10.

```

67 /* check whether a pte points to a swap entry */
68 static inline int is_swap_pte(pte_t pte)
69 {
70     return !pte_none(pte) && !pte_present(pte);
71 }

```

Listing 12-10: include/linux/swapops.h: `is_swap_pte()`

The defining characteristic of a PTE swap entry is that the present bit has not been set, but the entry is not otherwise clear (see Section 3.1.2 for more details on page tables and their flags).

We determine whether the present bit is set via `pte_present()` and whether it is clear via `pte_none()`.

The `swp_entry_to_pte()` function allows the conversion of the architecture-independent `swp_entry_t` value to a PTE entry suitable for the architecture and ready to be inserted in the page table. We examine it in Listing 12-11.

```

86 /*
87  * Convert the arch-independent representation of a swp_entry_t into the
88  * arch-dependent pte representation.
89  */
90 static inline pte_t swp_entry_to_pte(swp_entry_t entry)
91 {
92     swp_entry_t arch_entry;
93
94     arch_entry = __swp_entry(swp_type(entry), swp_offset(entry));
95     return __swp_entry_to_pte(arch_entry);
96 }

```

Listing 12-11: include/linux/swapops.h: `swp_entry_to_pte()`

This extracts the type (i.e. swap file index) and offset of the swap entry via `swp_type()` (see Listing 12-5) and `swp_offset()` (see Listing 12-6) respectively.

This is then passed to the architecture-specific `__swp_entry()` function which generates an entry that can be placed within the PTE. We examine this function in Listing 12-12.

Finally, the function invokes the architecture-specific `__swp_entry_to_pte()` which converts the generated PTE value contained in a `swp_entry_t` value to a PTE-specific type, `pte_t`. For x86-64, this is simply pass-through.

```

229 /*
230  * Shift the offset up "too far" by TYPE bits, then down again
231  * The offset is inverted by a binary not operation to make the high
232  * physical bits set.
233  */
234 #define __swp_entry(type, offset) ((swp_entry_t) { \
235     ~(unsigned long)(offset) << SWP_OFFSET_SHIFT >> SWP_TYPE_BITS) \
236     | ((unsigned long)(type) << (64-SWP_TYPE_BITS)) })

```

Listing 12-12: `arch/x86/include/asm/pgtable_64.h: __swp_entry()`

The x86-64-specific `__swp_entry()` function first shifts the bitwise-inverse of the offset value by `SWP_OFFSET_SHIFT`, this is done in order to truncate higher bits above that for which x86-64 provides space. We examine this value in Listing 12-13.

NOTE

We invert the offset value in order to mitigate the L1TF side-channel vulnerability, discussion of which is outside the scope of the book. However the intent is that this inversion results in an invalid physical address being specified (even once shifted back into place) and thus prevents speculation, mitigating the issue.

```

218 /* We always extract/encode the offset by shifting it all the way up, and then
    down again */
219 #define SWP_OFFSET_SHIFT      (SWP_OFFSET_FIRST_BIT+SWP_TYPE_BITS)

```

Listing 12-13: `arch/x86/include/asm/pgtable_64.h: SWP_OFFSET_SHIFT`

The `SWP_OFFSET_FIRST_BIT` value is equal to `_PAGE_BIT_PROTNONE` plus one, i.e. indicating that this bit and those below it are preserved in the PTE and subtracted from available space for the offset. This occupies 9 bits. The `SWP_TYPE_BITS` value is equal to 5, so that means that the preserved bits and the type value occupy 14 bits, leaving 50 bits for the offset.

NOTE

This operation therefore zeroes all of the lower bits. The meanings of any existing bits are therefore null and void. The defining characteristic of the swap entry is that it is both non-present (typically least significant bit is clear) and non-empty.

Returning to `__swp_entry()` in Listing 12-12, we therefore see that unavailable upper bits for the swap offset are truncated, after which we shift right by `SWP_TYPE_BITS`, making room for the type (i.e. swap file index) at the most significant bits before placing it there.

The inverse of `swp_entry_to_pte()` is `pte_to_swp_entry()` which converts a swap PTE to a `swp_entry_t`. We examine this in Listing 12-11.

```

73 /*
74  * Convert the arch-dependent pte representation of a swp_entry_t into an
75  * arch-independent swp_entry_t.
76  */
77 static inline swp_entry_t pte_to_swp_entry(pte_t pte)
78 {
79     swp_entry_t arch_entry;
80
81     pte = pte_swp_clear_flags(pte);
82     arch_entry = __pte_to_swp_entry(pte);
83     return swp_entry(__swp_type(arch_entry), __swp_offset(arch_entry));
84 }

```

Listing 12-14: include/linux/swapops.h: pte_to_swp_entry()

This starts with a new aspect to swap PTEs—clearing any existing “swap flags” that were established for the mapping via `pte_swp_clear_flags()`, which we examine in Listing 12-15.

We define a number of flags which “borrow” existing bits in the portion of the PTE which we have reserved for flags, i.e. for x86-64, at bit `SWP_OFFSET_FIRST_BIT` or below.

This preserves information that we wish to maintain at the PTE level and retain even while the mapping is swapped out.

```

26 /* Clear all flags but only keep swp_entry_t related information */
27 static inline pte_t pte_swp_clear_flags(pte_t pte)
28 {
29     if (pte_swp_exclusive(pte))
30         pte = pte_swp_clear_exclusive(pte);
31     if (pte_swp_soft_dirty(pte))
32         pte = pte_swp_clear_soft_dirty(pte);
33     if (pte_swp_uffd_wp(pte))
34         pte = pte_swp_clear_uffd_wp(pte);
35     return pte;
36 }

```

Listing 12-15: include/linux/swapops.h: pte_swp_clear_flags()

This indicates three possible flags—a “swap exclusive” flag, `_PAGE_SWP_EXCLUSIVE`, which determines if the swap entry is mapped exclusively to this PTE, a soft-dirty flag which propagates the software-defined dirty bit (see the *soft-dirty* documentation for details), `_PAGE_SWP_SOFT_DIRTY` and an out-of-scope userfaultfd flag used to track user write protect fault handling, `_PAGE_SWP_UFFD_WP`.

In each instance, we check for the flag, and if set, clear it. We won’t look into the functions which do this directly as they are simple wrappers around the aforementioned flag values.

Returning to `pte_to_swp_entry()` (see Listing 12-14), after we clear the flags we now have a value of type `pte_t` containing the raw PTE with lower

bits cleared. We then use the architecture-specific `__pte_to_swp_entry()` to convert it to a `swp_entry_t` (for x86-64 this is pass-through).

Finally we extract the architecture-specific type (i.e. swap file index) and offset using `__swp_type()` and `__swp_offset()` respectively which simply invert the operations performed by `__swp_entry()` (see Listing 12-12).

These are fed into the architecture-independent `swp_entry()` function, as shown in Listing 12-3, which simply combines the two values into a `swp_entry_t`.

12.2 Swapping Out

Swapping out occurs as part of reclaim (see Chapter 11). The key function of which is `shrink_page_list()` (see the discussion around Listing 11-63 for a fuller discussion of this function).

We examine how swapping out is performed in Figure 12-1.

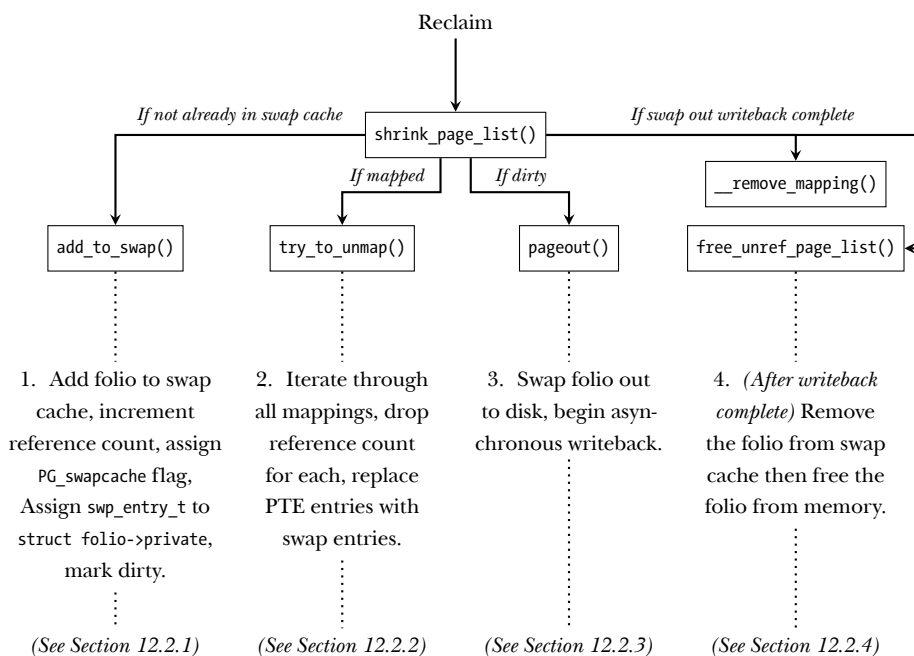


Figure 12-1: Swap Out Overview

NOTE

When paged out, the folio will be moved to the head of the appropriate LRU list (reclaim is performed from the tail). This delays any attempt to actually free the memory until the folio is next considered for reclaim. However, a folio which has just completed writeback and has the `PG_reclaim` flag set (as it will be on swap out) is immediately placed on the inactive tail to be reclaimed next via `folio_rotate_reclaimable()`, called from `folio_end_writeback()`. See Chapter 11 on reclaim for more details.

While we have examined `shrink_page_list()` in detail already, for the purposes of examining how it performs swapping out we will examine only the parts of it which perform tasks related to this, starting in Listing 12-16.

```

1589 static unsigned int shrink_page_list(struct list_head *page_list,
1590                                     struct pglist_data *pgdat,
1591                                     struct scan_control *sc,
1592                                     struct reclaim_stat *stat,
1593                                     bool ignore_references)
1594 {
1595     :
1608     while (!list_empty(page_list)) {
1609         :
1766         /*
1767          * Anonymous process memory has backing store?
1768          * Try to allocate it some swap space here.
1769          * Lazyfree folio could be freed directly
1770          */
1771         if (folio_test_anon(folio) && folio_test_swapbacked(folio)) {
1772             if (!folio_test_swapcache(folio)) {
1773                 :
1791                 if (!add_to_swap(folio)) {
1792                     :
1793                     goto activate_locked_split;
1794                 :
1803             }
1804         }
1805     }
1806 }
1807 :

```

Listing 12-16: `mm/vmscan.c: shrink_page_list()` Adding to Swap Cache

This determines whether the folio is eligible to be swapped out—whether it is both anonymous and swap-backed, the former checked by `folio_test_anon()` and the latter by checking for the `PG_swapbacked` folio flag.

NOTE

The `PG_swapbacked` flag is set in `page_add_new_anon_rmap()` when anonymous memory is allocated (see Chapter 7), and when folios are swapped in (via `do_swap_page()` and `__read_swap_cache_async()`). Anonymous memory without `PG_swapbacked` set is lazyfree (i.e. memory marked by user with the `MADV_FREE` flag via `madvise()` to eventually be freed on reclaim) see Chapter 8). Therefore the `PG_swapbacked` flag is a defining characteristic of anonymous memory.

If it is indeed permitted to be swapped out, then we invoke `add_to_swap()` to add it to the swap cache, a function we explore in Listing 12-20 in Section 12.2.1.

Next, we consider how swap entries are inserted into the page table on swap out, which we examine in Listing 12-17.

```

1822         /*
1823          * The folio is mapped into the page tables of one or more
1824          * processes. Try to unmap it here.
1825          */
1826         if (folio_mapped(folio)) {
1827             enum ttu_flags flags = TTU_BATCH_FLUSH;
1828             :
1833             try_to_unmap(folio, flags);
1834             if (folio_mapped(folio)) {
1835                 :
1839                 goto activate_locked;
1840             }
1841         }

```

Listing 12-17: mm/vmscan.c: *shrink_page_list()* Setting Up Swap Entries

Note that we only perform an action here if the folio is mapped, as determined by the `folio_mapped()` helper function. This function looks up the folio’s map count, as explored in Chapter 2, which is used to track the number of PTEs (or if a huge page, PMDs or PUDs) the folio is mapped into (for more on PUD, PMD and PTE page tables, see Chapter 3).

This makes sense, as obviously attempting to unmap a folio which is not currently mapped would be an exercise in pointlessness. Since the folio is locked at this stage we can rely on the state of the mapping not changing until reclaim has been performed upon it.

The function which ultimately performs the unmapping is `try_to_unmap()`. In the case of swapping out, this instead introduces swap entries into the relevant PTE entries which indicates to the kernel that, if these mappings are faulted upon, a swap in operation has to take place.

We examine the parts of `try_to_unmap()` pertinent to adding swap entries in Listing 12-23 in Section 12.2.2.

If, after the attempted unmapping, the folio remains mapped, this indicates that the unmapping failed. In this instance we “activate” the folio, i.e. if it was not already on the active LRU, putting it there so it can be examined next time (the folio will be added to the end of this list). See Chapter 11 for more details on active/inactive LRUs.

Next, we consider how we actually perform the swapping out to disk, which we examine in Listing 12-18.

```

1843         mapping = folio_mapping(folio);
1844         if (folio_test_dirty(folio)) {
1845             :
1886             switch (pageout(folio, mapping, &plug)) {
1887                 case PAGE_KEEP:
1888                     goto keep_locked;
1889                 case PAGE_ACTIVATE:
1890                     goto activate_locked;
1891                 case PAGE_SUCCESS:

```

```

:
1896         if (folio_test_dirty(folio))
1897             goto keep;
:
1903         if (!folio_trylock(folio))
1904             goto keep;
1905         if (folio_test_dirty(folio) ||
1906             folio_test_writeback(folio))
1907             goto keep_locked;
1908         mapping = folio_mapping(folio);
1909         fallthrough;
1910     case PAGE_CLEAN:
1911         ; /* try to free the folio below */
1912     }
1913 }
:

```

Listing 12-18: mm/vmscan.c: shrink_page_list() Swapping Out to Disk

At this stage, the folio should be marked dirty, either by virtue of the anonymous folio having been written to (and thus not being a Copy-on-Write mapping), thus resulting in the dirty bit being set in the PTE and the folio’s dirty flag having been set in `try_to_unmap_one()` as invoked by `try_to_unmap()` as discussed in Listing 12-23 and Section 12.2.2, or as a result of `add_to_swap()` marking it dirty as shown in Listing 12-20 and Section 12.2.1.

This means that the test for the folio’s dirty flag shown in Listing 12-18 will succeed, and thus `pageout()` will be invoked to perform the actual write out to disk, a function whose swap-specific behaviour we examine in Listing 12-26 and Section 12.2.3.

After the `pageout()` call is complete, we determine its outcome—if the asynchronous I/O was initiated to swap the folio out to disk, then `PAGE_SUCCESS` is returned and the folio is unlocked.

At this point it’s a case of checking whether the folio was somehow redirtied (i.e. could not be written out), cannot be immediately relocked (e.g. a synchronous write back occurred such as for a RAM-backed file system), or if writeback is currently underway (a likely outcome, see Chapter 10 for details on how writeback functions).

In each of these instances we “keep” the folio (perhaps activating it), as we cannot immediately attempt to free it. This means that freeing the folio will be deferred to a later stage (again, see Chapter 11 for a detailed analysis of the reclaim mechanism).

In the usual course of events, the folio will be subject to `pageout()` which swaps out the folio to disk asynchronously, then placed back on its LRU list i.e. “kept”, then when next subject to reclaim, finally freed.

The reclaim logic which causes the freeing of the folio is shown in Listing 12-19.

```

1843         if (folio_test_anon(folio) && !folio_test_swapbacked(folio)) {

```



```

:
1973         } else if (!mapping || !__remove_mapping(mapping, folio, true,
1974                                                     sc->target_mem_cgroup
:
:         ))
:
1975             goto keep_locked;
1976
1977         folio_unlock(folio);
:
:         list_add(&folio->lru, &free_pages);
1992         continue;
1993
1994
1995 activate_locked_split:
:
:
2004 activate_locked:
2005         /* Not a candidate for swapping, so reclaim swap space. */
2006         if (folio_test_swapcache(folio) &&
:
:         folio_test_mlocked(folio)))
2008             try_to_free_swap(&folio->page);
2009
:
:
2017 keep_locked:
2018         folio_unlock(folio);
2019 keep:
2020         list_add(&folio->lru, &ret_pages);
:
:         }
2023
:
:         free_unref_page_list(&free_pages);
2040
2041
2042         list_splice(&ret_pages, page_list);
:
:
2048     }

```

Listing 12-19: mm/vmscan.c: shrink_page_list() Freeing Swap

NOTE

Whatever pages are left in the `page_list` after this function is complete are eventually placed back on the appropriate LRU via `move_pages_to_lru()`.

Note that the mapping value here will be set to the result of `folio_mapping()`, not to be confused with `folio_mapped()` which indicates whether the folio is mapped by any userland process.

So for an entry in the swap cache not currently undergoing writeback, mapping will always be non-NULL, meaning that should we reach this point in `shrink_page_list()`, we will always invoke `__remove_mapping()`.

We explore the swap-relevant parts of `__remove_mapping()` in Listing 12-29 and Section 12.2.4. This function checks that nothing unexpectedly references the folio, then drops the reference count “freezing” it, before removing it from the swap cache and freeing up relevant resources.

Once this has been done, the folio is unlocked and added to the `free_pages` list which is ultimately freed via `free_unref_page_list()` (see Section 2.9 for more details on the physical page allocator’s freeing mechanism).

The final swap case handled by `shrink_page_list()` is one where a swap cache folio has become ineligible for being swapped out, by being `mlocked` (there is also a `cgroup`-specific case not shown in the listing but this is out of scope for the book).

In this instance we simply wish to free up the swap cache entry associated with the folio, which we do via `try_to_free_swap()`, a function we examine in Listing 12-31 and Section 12.2.4.

We note that all instances of activation or keeping the folio we place the folio on to the `ret_pages` list which are ultimately placed back into the appropriate LRU list (see Section 11.2 for more details on LRU lists in general).

Let’s examine each of this stages of swapping out one-by-one.

12.2.1 Adding a Folio to the Swap Cache

When we designate folios to be swapped out, we must first place them into the swap cache, which we perform by invoking `add_to_swap()` during reclaim in `shrink_page_list()` as described in Section 12.2. We examine this function in Listing 12-20.

```

164 /**
165  * add_to_swap - allocate swap space for a folio
166  * @folio: folio we want to move to swap
167  *
168  * Allocate swap space for the folio and add the folio to the
169  * swap cache.
170  *
171  * Context: Caller needs to hold the folio lock.
172  * Return: Whether the folio was added to the swap cache.
173  */
174 bool add_to_swap(struct folio *folio)
175 {
176     swp_entry_t entry;
177     int err;
178     :
179
180     entry = folio_alloc_swap(folio);
181     if (!entry.val)
182         return false;
183
184     /*
185     * XArray node allocations from PF_MEMALLOC contexts could
186     * completely exhaust the page allocator. __GFP_NOMEMALLOC
187     * stops emergency reserves from being allocated.
188     *
189     * TODO: this could cause a theoretical memory reclaim
190     * deadlock in the swap out path.

```

```

193     */
194     /*
195     * Add it to the swap cache.
196     */
197     err = add_to_swap_cache(&folio->page, entry,
198                           __GFP_HIGH|__GFP_NOMEMALLOC|__GFP_NOWARN, NULL);
199     if (err)
200         /*
201         * add_to_swap_cache() doesn't return -EEXIST, so we can
safely
202         * clear SWAP_HAS_CACHE flag.
203         */
204         goto fail;
205     /*
206     * Normally the folio will be dirtied in unmap because its
207     * pte should be dirty. A special case is MADV_FREE page. The
208     * page's pte could have dirty bit cleared but the folio's
209     * SwapBacked flag is still set because clearing the dirty bit
210     * and SwapBacked flag has no lock protected. For such folio,
211     * unmap will not set dirty bit for it, so folio reclaim will
212     * not write the folio out. This can cause data corruption when
213     * the folio is swapped in later. Always setting the dirty flag
214     * for the folio solves the problem.
215     */
216     folio_mark_dirty(folio);
217
218     return true;
219
220 fail:
221     put_swap_page(&folio->page, entry);
222     return false;
223 }

```

Listing 12-20: mm/swap_state.c: `add_to_swap()`

We start by allocating a swap entry using `folio_alloc_swap()`, which we examine in Listing 12-21 (eliding out of scope huge page and cgroup handling).

This uses a global per-CPU cache of swap slots and attempts to assign from here rather than resorting to actually allocating a swap entry from the underlying swap medium (file or partition).

If allocated, then the returned `swp_entry_t` value encodes the specific swap file and offset within it that contains this entry. If there insufficient space available to allocate swap, then we return false.

If we successfully allocate a swap entry, we then proceed to add it to the pertinent struct `address_space` swapper space via `add_to_swap_cache()`.

At this point it is absolutely critical that the underlying folio be marked dirty, which we anticipate will be the case upon setting page table swap en-

tries (see Section 12.2.2), however to ensure that this is always the case, we mark the folio dirty via `folio_mark_dirty()`.

If any error occurs after the swap entry is created, we drop its reference count within the swap file via `put_swap_page()`.

Note that the swap file handling logic which determines how to interact with the swap file itself, as implemented in `mm/swapfile.c`, is complicated, and the detail of this is out of scope for the book. We rather focus on the higher level abstractions which form part of the swap implementation.

```

302 swp_entry_t folio_alloc_swap(struct folio *folio)
303 {
304     swp_entry_t entry;
305     struct swap_slots_cache *cache;
306
307     entry.val = 0;
308     :
309
310     /*
311      * Preemption is allowed here, because we may sleep
312      * in refill_swap_slots_cache(). But it is safe, because
313      * accesses to the per-CPU data structure are protected by the
314      * mutex cache->alloc_lock.
315      */
316     cache = raw_cpu_ptr(&swp_slots);
317
318     if (likely(check_cache_active() && cache->slots)) {
319         mutex_lock(&cache->alloc_lock);
320         if (cache->slots) {
321             repeat:
322                 if (cache->nr) {
323                     entry = cache->slots[cache->cur];
324                     cache->slots[cache->cur++].val = 0;
325                     cache->nr--;
326                 } else if (refill_swap_slots_cache(cache)) {
327                     goto repeat;
328                 }
329             }
330             mutex_unlock(&cache->alloc_lock);
331             if (entry.val)
332                 goto out;
333         }
334
335         get_swap_pages(1, &entry, 1);
336     out:
337     :
338     return entry;

```

350 }

Listing 12-21: mm/swap_slots.c: folio_alloc_swap()

The `folio_alloc_swap()` function starts by checking a swap slot cache, maintained in the `swp_slots` static value. This maintains `SWAP_SLOTS_CACHE_SIZE` number of swap slot values (hard-coded to be equal to `SWAP_BATCH` or 64 slots).

For brevity, we won't examine this implementation in detail, but in either the case where the cache needs refilling or the cache is either disabled or needs refilling, the `get_swap_pages()` is called to obtain swap entries from available swap files.

NOTE

As previously discussed, we will not examine the swap file implementation in detail, as it rapidly becomes a file system discussion rather than a memory management one.

Returning to `add_to_swap()` (as shown in Listing 12-20), we note that the key logic for adding the newly allocated swap entry (i.e. now tied to a specific offset in a swap file on disk) to the swap cache is performed by `add_to_swap_cache()`, as shown in Listing 12-22 (eliding out of scope working set shadow entry tracking and debug logic).

```

84  /*
85   * add_to_swap_cache resembles filemap_add_folio on swapper space,
86   * but sets SwapCache flag and private instead of mapping and index.
87   */
88  int add_to_swap_cache(struct page *page, swp_entry_t entry,
89                      gfp_t gfp, void **shadowp)
90  {
91      struct address_space *address_space = swap_address_space(entry);
92      pgoff_t idx = swp_offset(entry);
93      XA_STATE_ORDER(xas, &address_space->i_pages, idx, compound_order(page)
94  );
95      unsigned long i, nr = thp_nr_pages(page);
96      :
101     page_ref_add(page, nr);
102     SetPageSwapCache(page);
103
104     do {
105         xas_lock_irq(&xas);
106         xas_create_range(&xas);
107         if (xas_error(&xas))
108             goto unlock;
109         for (i = 0; i < nr; i++) {
110             :
116             set_page_private(page + i, entry.val + i);
117             xas_store(&xas, page);
118             xas_next(&xas);
119         }

```

```

120         address_space->numpages += nr;
121         __mod_node_page_state(page_pgdat(page), NR_FILE_PAGES, nr);
122         __mod_lruvec_page_state(page, NR_SWAPCACHE, nr);
123     unlock:
124         xas_unlock_irq(&xas);
125     } while (xas_nomem(&xas, gfp));
126
127     if (!xas_error(&xas))
128         return 0;
129
130     ClearPageSwapCache(page);
131     page_ref_sub(page, nr);
132     return xas_error(&xas);
133 }

```

Listing 12-22: mm/swap_state.c: `add_to_swap_cache()`

In `add_to_swap_cache()` we can observe that the struct `address_space` object which we obtain via `swap_address_space()` (as shown in Listing 12-4) makes use of the struct `address_space->i_pages` xarray in order to store swap cache pages (see Section 9.2 in Chapter 9 for more on xarray).

We won't dwell too deeply on the xarray primitives used here (again, see Section 9.2 for a detailed analysis), but rather observe how we update both the folio and the swapper address space to mark that this folio is now in the swap cache.

We obtain the offset within the swap address via `swp_offset()` (see Listing 12-6), and then construct xarray state to be able to iterate through the relevant sub-pages contained within the folio.

NOTE

In this instance, the only occasion when the folio would be larger than order-0 is if it were a Transparent Huge Page. This is out of scope for the book, but we can see that we obtain the number of sub-pages via `thp_nr_pages()` which simply looks up the compound number of pages, see Chapter 2 for more details on compound folios.

We pin the folio by incrementing its reference via `page_ref_add()`, and mark the folio with the `PG_swapcache` flag via the `SetPageSwapCache()` macro-generated helper function.

For each individual sub-page of the folio, we set its struct `page->private` field to the `swp_entry_t` describing its location in this swap address space, through use of the `set_page_private()` function. Note that we offset each entry by sub-page index if a compound folio to account for it being offset in the xarray.

We discuss how this field is accessed in Section 12.1.2. This field is ultimately accessed via `folio_mapping()`, whose swap-relevant parts we examine in Listing 12-8.

We update the struct `address_space->numpages` field accordingly to count the number of entries in the swapper address space and also update statistics accordingly.

Finally, if no error arose we simply exit, otherwise we clear the folio's PG_swapcache flag, unpin it via `page_ref_sub()` and return the error.

12.2.2 Setting Page Table Swap Entries

Returning to Figure 12-1, we observe that after adding an entry to the swap cache reclaim then proceeds to remove page table mappings via `try_to_unmap()`, which we examine in Listing 12-23.

```

1801 /**
1802  * try_to_unmap - Try to remove all page table mappings to a folio.
1803  * @folio: The folio to unmap.
1804  * @flags: action and flags
1805  *
1806  * Tries to remove all the page table entries which are mapping this
1807  * folio. It is the caller's responsibility to check if the folio is
1808  * still mapped if needed (use TTU_SYNC to prevent accounting races).
1809  *
1810  * Context: Caller must hold the folio lock.
1811  */
1812 void try_to_unmap(struct folio *folio, enum ttu_flags flags)
1813 {
1814     struct rmap_walk_control rwc = {
1815         .rmap_one = try_to_unmap_one,
1816         .arg = (void *)flags,
1817         .done = page_not_mapped,
1818         .anon_lock = folio_lock_anon_vma_read,
1819     };
1820
1821     if (flags & TTU_RMAP_LOCKED)
1822         rmap_walk_locked(folio, &rwc);
1823     else
1824         rmap_walk(folio, &rwc);
1825 }
```

Listing 12-23: `mm/rmap.c: try_to_unmap()`

The `try_to_unmap()` function utilises the reverse mapping functionality (see Chapter 7 for details on this) to find all `struct vm_area_struct` (VMA) mappings which might reference the specified folio, and calls the specified callback function, which is `try_to_unmap_one()`, whose swap-specific code we examine in Listing 12-24.

We stop the traversal if the specified `page_not_mapped()` function returns true, which in turn checks whether `folio_mapped()` returns false—this function simply (atomically) checks whether the `struct folio->mapcount` field is greater than or equal to zero (this is equal to the number of mappings of the folio minus one, so a value of zero implies that it is mapped once).

We pass `enum ttu_flags` to the function which specify various options for modifying the operation, we will not examine these in detail.

We examine the swap-specific logic in `try_to_unmap_one()` in Listing 12-24 (eliding logic irrelevant to swap handling, TTU flags not set by `shrink_page_list()`, out of scope MMU notifier logic, huge page handling, `userfaultfd` handling, hardware poisoning, architecture-specific virtualisation, lazy free (i.e. `MADV_FREE` via `madvise()`), high watermark RSS handling, architecture-specific missing PTE flag handling and debug checks).

```

1476 static bool try_to_unmap_one(struct folio *folio, struct vm_area_struct *vma,
1477                             unsigned long address, void *arg)
1478 {
1479     struct mm_struct *mm = vma->vm_mm;
1480     DEFINE_FOLIO_VMA_WALK(pvmw, folio, vma, address, 0);
1481     pte_t pteval;
1482     struct page *subpage;
1483     bool anon_exclusive, ret = true;
1484     :
1485     enum ttu_flags flags = (enum ttu_flags)(long)arg;
1486     :
1487     while (page_vma_mapped_walk(&pvmw)) {
1488         :
1489         /*
1490          * If the folio is in an mlock()d vma, we must not swap it out
1491          *
1492          */
1493         if (!(flags & TTU_IGNORE_MLOCK) &&
1494             (vma->vm_flags & VM_LOCKED)) {
1495             /* Restore the mlock which got missed */
1496             mlock_vma_folio(folio, vma, false);
1497             page_vma_mapped_walk_done(&pvmw);
1498             ret = false;
1499             break;
1500         }
1501
1502         subpage = folio_page(folio,
1503                             pte_pfn(*pvmw.pte) - folio_pfn(folio))
1504     ;
1505
1506     address = pvmw.address;
1507     anon_exclusive = folio_test_anon(folio) &&
1508                     PageAnonExclusive(subpage);
1509
1510     if (folio_test_hugetlb(folio)) {
1511         :
1512     } else {
1513         :
1514         /*
1515          * Nuke the page table entry. When having to clear
1516          * PageAnonExclusive(), we always have to flush.
1517          */
1518

```



```

1589             if (should_defer_flush(mm, flags) && !anon_exclusive)
1600             {
1601                 :
1598                 pteval = ptep_get_and_clear(mm, address, pvmw.
1602                 pte);
1603                 :
1601             } else {
1602                 pteval = ptep_clear_flush(vma, address, pvmw.
1603                 pte);
1604             }
1605             :
1613             /* Set the dirty flag on the folio now the pte is gone. */
1614             if (pte_dirty(pteval))
1615                 folio_mark_dirty(folio);
1616             :
1620             if (PageHWPoison(subpage) && !(flags & TTU_IGNORE_HWPoison)) {
1621                 :
1645             } else if (folio_test_anon(folio)) {
1646                 swp_entry_t entry = { .val = page_private(subpage) };
1647                 pte_t swp_pte;
1648                 :
1707                 if (swap_duplicate(entry) < 0) {
1708                     set_pte_at(mm, address, pvmw.pte, pteval);
1709                     ret = false;
1710                     page_vma_mapped_walk_done(&pvmw);
1711                     break;
1712                 }
1713                 :
1720                 if (anon_exclusive &&
1721                     page_try_share_anon_rmap(subpage)) {
1722                     swap_free(entry);
1723                     set_pte_at(mm, address, pvmw.pte, pteval);
1724                     ret = false;
1725                     page_vma_mapped_walk_done(&pvmw);
1726                     break;
1727                 }
1728                 :
1745                 dec_mm_counter(mm, MM_ANONPAGES);
1746                 inc_mm_counter(mm, MM_SWAPENTS);
1747                 swp_pte = swp_entry_to_pte(entry);
1748                 if (anon_exclusive)
1749                     swp_pte = pte_swp_mkexclusive(swp_pte);
1750                 if (pte_soft_dirty(pteval))
1751                     swp_pte = pte_swp_mksoft_dirty(swp_pte);
1752                 :
1754                 set_pte_at(mm, address, pvmw.pte, swp_pte);

```

```

:
1758             } else {
:
1771             }
:
1780             page_remove_rmap(subpage, vma, folio_test_hugetlb(folio));
:
1783             folio_put(folio);
1784         }
:
1788         return ret;
1789     }

```

Listing 12-24: mm/rmap.c: try_to_unmap_one() Swap Logic

The `try_to_unmap_one()` function starts by declaring a `struct page_vma_mapped_walk` type (see Listing 12-25) used to thread state while iterating through the page tables mapping the folio, initialised by `DEFINE_FOLIO_VMA_WALK()` and iterated through via `page_vma_mapped_walk()`.

```

316 struct page_vma_mapped_walk {
317     unsigned long pfn;
318     unsigned long nr_pages;
319     pgoff_t pgoff;
320     struct vm_area_struct *vma;
321     unsigned long address;
322     pmd_t *pmd;
323     pte_t *pte;
324     spinlock_t *ptl;
325     unsigned int flags;
326 };

```

Listing 12-25: include/linux/rmap.h: struct page_vma_mapped_walk

The `struct page_vma_mapped_walk` type encodes state about the currently examined PTE as well as the PTE lock it acquired to allow us to interact with the PTE, which we must release once we're done with it.

The `page_vma_mapped_walk()` function walks the page tables and acquires the PTE lock for us (placed in `struct page_vma_mapped_walk->ptl`), handling all edge cases and detecting whether the mapping is still valid, returning true if the iteration can continue, otherwise returning false.

NOTE

At this stage, we can safely assume that the `struct mm_struct->mmap_lock` is held.

We can therefore safely assume that `struct page_vma_mapped_walk->pte` can be updated throughout the operation.

In the case of swap out, `shrink_page_list()` calls with the `TTU_BATCH_FLUSH` flag set (and optionally a huge PMD flag but this is out of scope for the book), therefore `TTU_IGNORE_MLOCK` will not be set and so we must consider the `mlock()` edge case.

The edge case arises when the struct `vm_area_struct` (VMA) flag `VM_LOCKED` has been set, indicating that all memory described by the VMA should be locked by `mlock()`, however it has somehow not been marked by `PG_mlocked` or `PG_unevictable`, or has otherwise somehow ended up in reclaim.

We resolve this edge case by invoking `mlock_vma_folio()` to resolve the situation, before terminating the iteration via `page_vma_mapped_walk_done()` and exiting the loop. See Section 8.2.1 in Chapter 8 for more on `mlock()`.

With this edge case accounted for, we determine the subpage via `folio_page()`, obtaining the offset by calculating the difference between the PFN of the folio (i.e. the head page) and that of the PTE.

This way, if we happen to be examining a compound folio spread over multiple PTEs (possible in some huge page scenarios), we will examine each of tail sub-pages individually (see Chapter 2 for more details on PTEs, folios and compound pages).

We determine whether this is an anonymous page which is mapped exclusively by only one process, as this changes how the kernel handles such mappings and is a trait we encode in swapped out folios. We retain this state in `anon_exclusive`.

NOTE

When updating PTEs, it is very important to ensure that the update proceeds in a predictable order. We therefore always clear the PTE entry (causing accesses to that address that hit the page tables to fault) before setting it to a new value. We must also then ensure that the Transaction Lookaside Buffer (TLB) that caches virtual to physical addresses is appropriately cleared.

The `should_defer_flush()` function indicates whether both `TTU_BATCH_FLUSH` is set (true for reclaim) and whether the struct `mm_struct` is in use by any other CPUs. If both are true, then we are permitted and it is worthwhile for us to batch flushing the TLB rather than flushing at each point of clearing the PTE entry.

In addition, if the mapping is anonymous-exclusive, then we know for certain that no other CPU could be accessing the mapping and thus batching is not useful.

In the case that we defer flushing, we invoke `ptep_get_and_clear()` to clear and retrieve the PTE value atomically, without any update to the TLB. Otherwise, we invoke `ptep_clear_flush()` (see Listing 6-45 in Chapter 6), which both calls `ptep_get_and_clear()` and clears the TLB flush.

See Section 7.1 in Chapter 7 for more on freeing memory and TLB maintenance.

As we have cleared the PTE, it now makes sense to mark the folio dirty via `folio_mark_dirty()`, something that is absolutely critical for the swap write out to occur (see Section 12.2.3) as we make use of the fact that the swap cache folios are dirty to start this write-out.

Having previously set the struct `page->private` field to the `swp_entry_t` describing its assigned position in a swap file (see Section 12.2.1), we retrieve this entry ready to place it in the PTE.

We use the oddly-named `swap_duplicate()` to increment the swap entry's reference count, if this fails we restore the original PTE and abort the walk.

Next, if the folio is anonymous-exclusive, we invoke `page_try_share_anon_rmap()` to strip it of the `PG_anon_exclusive` flag in preparation for swap out. If this fails, we again restore the PTE and abort the walk, freeing the swap file entry via `swap_free()`.

With edge cases handled, we update statistics to account for the swap out, convert the `swp_entry_t` value to a PTE value via `swp_entry_to_pte()` (see Listing 12-11), and proceed by setting swap-specific PTE flags for things we wish to retain and restore on page in—The anonymous-exclusive trait via `pte_swp_mkexclusive()` and the soft-dirty flag via `pte_swp_mksoft_dirty()`.

Having obtained a swap PTE entry and assigned flags, we then set the PTE itself via `set_pte_at()`. At this stage, the swap entry is set.

Finally, we decrement the page's map count, update statistics and handle related edge cases via `page_remove_rmap()`, and decrement the folio's reference count via `folio_put()`, as a mapping of a folio takes a reference on it and the mapping has now been removed.

12.2.3 *Swapping Out to Disk*

We observe in Figure 12-1 that `shrink_page_list()` invokes `pageout()` on dirty folios that need to be reclaimed.

As demonstrated previously, by this stage the swap out procedure will have guaranteed all folios to be swapped out will indeed be dirty, therefore `pageout()` is invoked on these folios, which we explore in Listing 12-26 (eliding out of scope cgroup and non-swap specific logic).

```

1211 /*
1212  * pageout is called by shrink_page_list() for each dirty page.
1213  * Calls ->writepage().
1214  */
1215 static pageout_t pageout(struct folio *folio, struct address_space *mapping,
1216                          struct swap_iocb **plug)
1217 {
1218     :
1219     if (folio_clear_dirty_for_io(folio)) {
1220         int res;
1221         struct writeback_control wbc = {
1222             .sync_mode = WB_SYNC_NONE,
1223             .nr_to_write = SWAP_CLUSTER_MAX,
1224             .range_start = 0,
1225             .range_end = LLONG_MAX,
1226             .for_reclaim = 1,
1227             .swap_plug = plug,
1228         };
1229
1230         folio_set_reclaim(folio);
1231         res = mapping->a_ops->writepage(&folio->page, &wbc);
1232         if (res < 0)
1233             handle_write_error(mapping, folio, res);

```

```

1268         if (res == AOP_WRITEPAGE_ACTIVATE) {
1269             folio_clear_reclaim(folio);
1270             return PAGE_ACTIVATE;
1271         }
1272
1273         if (!folio_test_writeback(folio)) {
1274             /* synchronous write or broken a_ops? */
1275             folio_clear_reclaim(folio);
1276         }
1277         :
1279         return PAGE_SUCCESS;
1280     }
1281
1282     return PAGE_CLEAN;
1283 }

```

Listing 12-26: mm/vmscan.c: pageout() Recap

The `pageout()` function marks the struct `folio` with the `PG_reclaim` flag during this operation, which will trigger writeback and in turn set the `PG_writeback` flag and start asynchronous I/O to perform the actual write to disk.

Prior to doing this, `folio_clear_dirty_for_io()` is invoked, which clears the folio's dirty flag, and marks all mappings to the folio read-only such that we can track whether anything else dirties the folio as they would generate a page fault if they did so (see Listing 10-30 in Chapter 10 for more details on this dance).

We call the struct `address_space->a_ops` callback (the callbacks being specified as a struct `address_space_operations` object) for writing a single page, `writepage()` to perform the actual writeback.

NOTE

We specify that up to `SWAP_CLUSTER_MAX` (hardcoded to 32) base pages should be written at one time—when writing to non-block device (file system-mediated) back ends, page writes are batched up and submitted when the batch size reaches `SWAP_CLUSTER_MAX`. We don't examine the file system swap code in detail but this batch size is important and dictates batch sizing in reclaim generally (see Chapter 11).

This is where the swap logic comes into play, as the address space provided to `pageout()` by `shrink_page_list()` is derived from `folio_mapping()` (see Listing 12-8), which explicitly checks to see if the folio is in the swap cache and if so returns the swapper address space.

The address space operations for the swapper address space are set by `init_swap_address_space()` on initialisation to `swap_aops`, which sets the `writepage` callback to `swap_writepage()`, which is what is invoked here, and we examine in Listing 12-27 (eliding out of scope architecture-specific handling and front-swap logic).

177 /*

```

178  * We may have stale swap cache pages in memory: notice
179  * them here and get rid of the unnecessary final write.
180  */
181  int swap_writepage(struct page *page, struct writeback_control *wbc)
182  {
183      int ret = 0;
184
185      if (try_to_free_swap(page)) {
186          unlock_page(page);
187          goto out;
188      }
189      :
205      ret = __swap_writepage(page, wbc, end_swap_bio_write);
206  out:
207      return ret;
208  }

```

Listing 12-27: mm/page_io.c: *swap_writepage()*

The `swap_writepage()` function starts by trying to free swap cache if there are no swapped out references to this folio as checked by `try_to_free_swap()` (which we examine in Listing 12-31 and Section 12.2.4). If so we exit without having to kick off a write. Otherwise, we invoke `__swap_writepage()`, which we examine in Listing 12-28.

Note that we pass `__swap_writepage()` a pointer to `end_swap_bio_write()` to be called when the asynchronous write is complete, and handle errors.

We examine `__swap_writepage()` in Listing 12-28, eliding out of scope debug checks and an irrelevant comment.

```

335  int __swap_writepage(struct page *page, struct writeback_control *wbc,
336                      bio_end_io_t end_write_func)
337  {
338      struct bio *bio;
339      int ret;
340      struct swap_info_struct *sis = page_swap_info(page);
341      :
348      if (data_race(sis->flags & SWP_FS_OPS))
349          return swap_writepage_fs(page, wbc);
350
351      ret = bdev_write_page(sis->bdev, swap_page_sector(page), page, wbc);
352      if (!ret) {
353          count_swpoout_vm_event(page);
354          return 0;
355      }
356
357      bio = bio_alloc(sis->bdev, 1,
358                     REQ_OP_WRITE | REQ_SWAP | wbc_to_write_flags(wbc),
359                     GFP_NOIO);
360      bio->bi_iter.bi_sector = swap_page_sector(page);

```

```

361         bio->bi_end_io = end_write_func;
362         bio_add_page(bio, page, thp_size(page), 0);
363
364         bio_associate_blkg_from_page(bio, page);
365         count_swapout_vm_event(page);
366         set_page_writeback(page);
367         unlock_page(page);
368         submit_bio(bio);
369
370         return 0;
371     }

```

Listing 12-28: mm/page_io.c: __swap_writepage()

The logic in `__swap_writepage()` makes reference to the swap file implementation, which we do not dive into in detail as this strays too far into file system territory, however we shall examine it from a broad top level.

We retrieve metadata about the swap entry, and determine whether we ought to invoke the containing file system to perform the writeback, in which case we invoke `swap_writepage_fs()`. We don't examine this function in detail as it again pertains very much to file system logic.

If not, then we allocate a struct `bio` object (see Section 9.10.5 in Chapter 9 for more details on this) to describe the I/O operation that is to be performed asynchronously, before setting the `PG_writeback` flag on the struct `page` to indicate writeback is underway and unlocking the page before submitting the block I/O operation via `submit_bio()`.

The `end_swap_bio_write()` function called when asynchronous I/O completes invokes `end_page_writeback()` which calls `folio_end_writeback()` in turn (see Listing ?? in Chapter 10), which importantly terminates writeback and calls `folio_rotate_reclaimable()` which puts it next in line for reclaim — ensuring the logic contained in Section 12.2.4 immediately upon writeback.

12.2.4 Freeing a Swapped Out Folio

Finally, after writeback has been kicked off on swap out and completes, `folio_rotate_reclaimable()` is called which causes the folio to be subject to immediate reclaim, at which point we can observe in Figure 12-1 that `shrink_page_list()` invokes `__remove_mapping()` to attempt to clear the mapping from the swap cache, and `free_unref_page_list()` to free the folio itself.

We don't examine the logic surrounding `free_unref_page_list()` as this is a general physical folio freeing mechanism and one we cover in Section 2.9 and Chapter 2.

However, we do wish to examine in detail the logic surrounding removal of a folio from the swap cache.

We therefore examine core and swap-specific logic in `__remove_mapping()` in Listing 12-29 (eliding out of scope working set, cgroup and non-swap specific logic).

1285 /*

```

1286  * Same as remove_mapping, but if the page is removed from the mapping, it
1287  * gets returned with a refcount of 0.
1288  */
1289  static int __remove_mapping(struct address_space *mapping, struct folio *folio
1290
1291  ,
1292                                bool reclaimed, struct mem_cgroup *target_memcg)
1293  {
1294      int refcount;
1295      :
1296
1297      xa_lock_irq(&mapping->i_pages);
1298      /*
1299       * The non racy check for a busy page.
1300       *
1301       * Must be careful with the order of the tests. When someone has
1302       * a ref to the page, it may be possible that they dirty it then
1303       * drop the reference. So if PageDirty is tested before page_count
1304       * here, then the following race may occur:
1305       *
1306       * get_user_pages(&page);
1307       * [user mapping goes away]
1308       * write_to(page);
1309       *                                     !PageDirty(page)    [good]
1310       * SetPageDirty(page);
1311       * put_page(page);
1312       *                                     !page_count(page)  [good, discard it]
1313       *
1314       * [oops, our write_to data is lost]
1315       *
1316       * Reversing the order of the tests ensures such a situation cannot
1317       * escape unnoticed. The smp_rmb is needed to ensure the page->flags
1318       * load is not satisfied before that of page->_refcount.
1319       *
1320       * Note that if SetPageDirty is always performed via set_page_dirty,
1321       * and thus under the i_pages lock, then this ordering is not required
1322
1323       */
1324
1325      refcount = 1 + folio_nr_pages(folio);
1326      if (!folio_ref_freeze(folio, refcount))
1327          goto cannot_free;
1328      /* note: atomic_cmpxchg in page_ref_freeze provides the smp_rmb */
1329      if (unlikely(folio_test_dirty(folio))) {
1330          folio_ref_unfreeze(folio, refcount);
1331          goto cannot_free;
1332      }
1333
1334      if (folio_test_swapcache(folio)) {
1335          swp_entry_t swap = folio_swap_entry(folio);
1336

```



```

:
1340         __delete_from_swap_cache(folio, swap, shadow);
1341         xa_unlock_irq(&mapping->i_pages);
1342         put_swap_page(&folio->page, swap);
1343     } else {
:
1374     }
1375
1376     return 1;
1377
1378 cannot_free:
1379     xa_unlock_irq(&mapping->i_pages);
1380     if (!folio_test_swapcache(folio))
1381         spin_unlock(&mapping->host->i_lock);
1382     return 0;
1383 }

```

Listing 12-29: mm/vmscan.c: `__remove_mapping()` Swap Logic

The `__remove_mapping()` function starts by freezing the reference count of the folio via `folio_ref_freeze()`, which cannot be elevated by either being mapped in userland or in-use by the kernel, or the freezing fails.

We have already determined by this point that the folio is not dirty or currently in a state of writeback, and thus will have been swapped out to disk.

NOTE

We explore folio freezing further in Chapter 11 on reclaim, however broadly it atomically compares and exchanges an expecting reference count with zero, ensuring that we do not race with anything that might pin the folio in place. Unfreezing reverses this operation.

We explicitly check whether the folio is dirty (and thus not in a state where it can be freed) and unfreeze via `folio_ref_unfreeze()` and abort if so.

Note the comment describing the careful ordering of updating the reference count and checking for the dirty flag.

In the case of a swap cache entry we delete the folio from the swapper struct `address_space` via `__delete_from_swap_cache()`, and clear its `PG_swapcache` flag, which we examine in Listing 12-30 (eliding out of scope debug checks), before invoking `put_swap_page()` to decrement the reference on the swap entry.

```

135 /*
136  * This must be called only on folios that have
137  * been verified to be in the swap cache.
138  */
139 void __delete_from_swap_cache(struct folio *folio,
140                             swp_entry_t entry, void *shadow)
141 {
142     struct address_space *address_space = swap_address_space(entry);

```

```

143     int i;
144     long nr = folio_nr_pages(folio);
145     pgoff_t idx = swp_offset(entry);
146     XA_STATE(xas, &address_space->i_pages, idx);
147     :
152     for (i = 0; i < nr; i++) {
153         void *entry = xas_store(&xas, shadow);
154         :
155         set_page_private(folio_page(folio, i), 0);
156         xas_next(&xas);
157     }
158     folio_clear_swapcache(folio);
159     address_space->nrxpages -= nr;
160     __node_stat_mod_folio(folio, NR_FILE_PAGES, -nr);
161     __lruvec_stat_mod_folio(folio, NR_SWAPCACHE, -nr);
162 }

```

Listing 12-30: mm/swap_state.c: __delete_from_swap_cache()

The `__delete_from_swap_cache()` looks up the appropriate swapper struct `address_space` via `swap_address_space()` (see Listing 12-4) and its index from `swp_offset()` (see Listing 12-6), before performing an xarray traversal, setting the entry to the shadow value.

The shadow value is in reference to workout set tracing logic which is out of scope for the book, so for the sake of argument we can consider this to be NULL (which very often it will be at any rate).

The struct `page->private` field is cleared for every subpage so it no longer references this swap entry and the `PG_swapcache` flag is cleared for the folio, before updating statistics accordingly.

Note that this does not free the folio, the `shrink_page_list()` logic does this separately by batching up folios to delete and passing them to `free_unref_page_list()`.

Another means by which swap entries can potentially be removed from the swap cache is `try_to_free_swap()`, which we examine in Listing 12-31 9eliding debug checks and power management logic.

```

1586 /*
1587  * If swap is getting full, or if there are no more mappings of this page,
1588  * then try_to_free_swap is called to free its swap space.
1589  */
1590 int try_to_free_swap(struct page *page)
1591 {
1592     struct folio *folio = page_folio(page);
1593     :
1595     if (!folio_test_swapcache(folio))
1596         return 0;
1597     if (folio_test_writeback(folio))
1598         return 0;
1599     if (folio_swapped(folio))

```

```

1600             return 0;
1601         :
1620         delete_from_swap_cache(folio);
1621         folio_set_dirty(folio);
1622         return 1;
1623     }

```

Listing 12-31: mm/swapfile.c: try_to_free_swap()

The `try_to_free_swap()` function performs a series of sanity checks to ensure that the operation is proceeding on a folio that is in the swap cache, not in the process of being written out and has at least one reference to the swapped out folio (as checked by `folio_swapped()`), before invoking `delete_from_swap_cache()` which we examine in Listing 12-32.

```

225 /*
226  * This must be called only on folios that have
227  * been verified to be in the swap cache and locked.
228  * It will never put the folio into the free list,
229  * the caller has a reference on the folio.
230  */
231 void delete_from_swap_cache(struct folio *folio)
232 {
233     swp_entry_t entry = folio_swap_entry(folio);
234     struct address_space *address_space = swap_address_space(entry);
235
236     xa_lock_irq(&address_space->i_pages);
237     __delete_from_swap_cache(folio, entry, NULL);
238     xa_unlock_irq(&address_space->i_pages);
239
240     put_swap_page(&folio->page, entry);
241     folio_ref_sub(folio, folio_nr_pages(folio));
242 }

```

Listing 12-32: mm/swap_state.c: delete_from_swap_cache()

The `delete_from_swap_cache()` function obtains the `swp_entry_t` associated with the struct `folio` via `folio_swap_entry()` (see Listing 12-9), before obtaining the swapper struct `address_space` via `swap_address_space()` (see Listing 12-4).

A lock is acquired over the struct `address_space->i_pages` xarray, before invoking `__delete_from_swap_cache()` (see Listing 12-30) to perform the actual removal from the swap cache, before invoking `put_swap_page()` to decrement the reference on the swap entry and decrementing the folio's reference count via `folio_ref_sub()`.

12.3 Swapping In

We have examined swapping out, and observed in Section 12.2.2 that we replace the PTE page table entry for the swapped-out mapping with an

architecture-specific encoding of the `swp_entry_t` describing in which swap file and offset within it the faulted page is located, along with a number of swap flags.

As a result, there is a very clear point at which swap in occurs—when a page fault occurs on a swapped out page. And from the PTE we are able to ascertain both where in the swap file the page might be swapped out to or where in the swap cache the page is located.

NOTE

The reason for a swap cache is precisely because page faults might occur at any time, and writing back to disk takes time to complete, so we must both indicate to the faulting mechanism that the swapped out page should be looked up in the swap while also storing the folio in the swap cache in the meantime.

12.3.1 Page Fault on a Swapped Out Page

We examine the page fault mechanism in great detail in Chapter 6, however we note that the point at which we determine the nature of the fault occurs in `handle_pte_fault()` which determines that the fault is caused by memory being swapped out when the PTE is non-empty, but is marked non-present (see Chapter 3 on Virtual Memory for details on the various page table flags including the present flag).

This then invokes `do_swap_page()` to handle the fault, which we examine starting in Listing 12-33 (eliding out of scope non-swap entry logic, cgroup logic, working set logic, hardware page poisoning logic, Kernel Same page Merging (KSM) logic, bug checks, userfaultfd handling, and architecture-specific cache flushing logic).

```

3710 /*
3711  * We enter with non-exclusive mmap_lock (to exclude vma changes,
3712  * but allow concurrent faults), and pte mapped but not yet locked.
3713  * We return with pte unmapped and unlocked.
3714  *
3715  * We return with the mmap_lock locked or unlocked in the same cases
3716  * as does filemap_fault().
3717  */
3718 vm_fault_t do_swap_page(struct vm_fault *vmf)
3719 {
3720     struct vm_area_struct *vma = vmf->vma;
3721     struct page *page = NULL, *swpcache;
3722     struct swap_info_struct *si = NULL;
3723     rmap_t rmap_flags = RMAP_NONE;
3724     bool exclusive = false;
3725     swp_entry_t entry;
3726     pte_t pte;
3727     int locked;
3728     vm_fault_t ret = 0;
3729     :
3730     entry = pte_to_swp_entry(vmf->orig_pte);

```

```

:
3758      /* Prevent swappoff from happening to us. */
3759      si = get_swap_device(entry);
3760      if (unlikely(!si))
3761          goto out;
3762
3763      page = lookup_swap_cache(entry, vma, vmf->address);
3764      swapcache = page;

```

Listing 12-33: mm/memory.c: do_swap_page() Preface

We start `do_swap_page()` by initialising local variables and obtaining the `swp_entry_t` from the PTE using `pte_to_swp_entry()` (see Listing 12-14 and Section 12.1.3 for details).

We then obtain metadata regarding the swap via `get_swap_device()`, incrementing a reference count on the swap file to ensure that a `swappoff()` cannot be performed beneath us.

NOTE

We do not delve too deeply into mm/swapfile.c implementation details as these rapidly become the remit of file system logic rather than memory management.

We then check the swap cache to see whether the page is still located in the swap cache via `lookup_swap_cache()`, which we examine in detail in Listing 12-39 and Section 12.3.2.

If we can locate the page in the swap cache then we do not need to read from the disk and can simply proceed with mapping the page directly from the swap cache, after considering edge cases as shown in Listing 12-35.

Otherwise, we must read from disk, the logic for which we examine in Listing 12-34.

```

3766      if (!page) {
3767          if (data_race(si->flags & SWP_SYNCHRONOUS_IO) &&
3768              __swap_count(entry) == 1) {
3769              /* skip swapcache */
3770              page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma,
3771                                  vmf->address);
3772              if (page) {
3773                  __SetPageLocked(page);
3774                  __SetPageSwapBacked(page);
3775                  :
3776
3777                  lru_cache_add(page);
3778
3779
3780                  /* To provide entry to swap_readpage() */
3781                  set_page_private(page, entry.val);
3782                  swap_readpage(page, true, NULL);
3783                  set_page_private(page, 0);
3784              }
3785          } else {
3786              page = swapin_readahead(entry, GFP_HIGHUSER_MOVABLE,

```

```

3797                                     vmf);
3798                                     swapcache = page;
3799                                 }
3800
3801                                 if (!page) {
3802                                     /*
3803                                      * Back out if somebody else faulted in this pte
3804                                      * while we released the pte lock.
3805                                      */
3806                                     vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd,
3807                                                             vmf->address, &vmf->ptl);
3808                                     if (likely(pte_same(*vmf->pte, vmf->orig_pte)))
3809                                         ret = VM_FAULT_OOM;
3810                                     goto unlock;
3811                                 }
3812
3813                                 /* Had to read the page from swap area: Major fault */
3814                                 ret = VM_FAULT_MAJOR;
3815                                 count_vm_event(PGMAJFAULT);
3816                                 :
3824                                 }
3825
3826                                 locked = lock_page_or_retry(page, vma->vm_mm, vmf->flags);
3827
3828                                 if (!locked) {
3829                                     ret |= VM_FAULT_RETRY;
3830                                     goto out_release;
3831                                 }

```

Listing 12-34: mm/memory.c: `do_swap_page()` Reading From Disk

If the swap device indicates that it is more efficient to perform synchronous I/O as indicated by the `SWP_SYNCHRONOUS_IO` flag being specified in the swap device’s metadata, and we are the only ones referencing this swap entry, then we take a direct path, bypass the swap cache altogether read the page from the swap directly.

We do so by allocating a user page via `alloc_page_vma()`, setting `locked` and swap-backed flags adding it to the appropriate LRU (see Section 11.2 in Chapter 11 for more on LRU lists) via `lru_cache_add()` (which ultimately invokes `folio_add_lru()`, see Listing 11-88 also in Chapter 11).

We then invoke `swap_readpage()` specifying that the read should be performed synchronously, having set the struct `page->private` to the `swp_entry_t` value describing this swap entry prior to doing so via `set_page_private()` before clearing it afterwards.

We examine `swap_readpage()` in Listing 12-40 in Section 12.3.3.

in the usual case, we do not perform synchronous readahead, but rather perform “swap readahead”, similar to page cache readahead, i.e. reading

additional pages beyond that requested in anticipation of the user needing them in the near future.

This is performed via `swpin_readahead()` which we examine in Listing 12-41 and Section 12.3.3.

In this instance we assign `swapcache` to the retrieved page in order to record the fact that the `swpin_readahead()` function call will have placed this page in the swap cache.

We then consider the situation in which either we had insufficient memory in order to allocate the page, or a race occurred in `swpin_readahead()`.

We determine which of the cases have occurred by locking the PTE and checking to see whether the PTE matches what we faulted on—if not we have been raced with, otherwise we have run out of memory and indicate as much by update the return value to indicate so with `VM_FAULT_OOM` (see Chapter 6 for details on the page fault mechanism as a whole).

In both cases in which we were not able to obtain a page, we abort the fault and exit.

Finally, if the page was successfully retrieved, we count this as a major fault, i.e. one which resulted in I/O.

NOTE

In the instance that asynchronous I/O has been started in order to retrieve data from the swap, the struct page we retrieve will be locked and not `PG_uptodate`—that is, allocated but not yet read from the swap.

The last few lines in Listing 12-35 are of critical importance—we try to acquire a lock on the page we have just obtained via `lock_page_or_retry()`, which checks to determine whether we can simply try retry the fault and drop the struct `mm_struct->mmmap_lock` to avoid unnecessary contention on it (see Section 6.2 and Chapter 6 for more details).

In the instance where we can simply try again later, `lock_page_or_retry()` will return false, otherwise it will wait until the lock is available (see Section 9.11 for details on how this mechanism functions).

Why is waiting for the lock here so critical? In the instance of asynchronous I/O being performed (the far more likely case), the page will be locked, and only unlocked when the data is read into memory (at which point the `PG_uptodate` flag will be set on the folio).

See Section 9.6 in Chapter 9 for more details on how data is read from disk into folios.

Therefore this lock acts as a barrier—prior to it, it is likely that the page does not contain swapped out data, afterwards either it does, a retry is indicated or an error arose.

In the case of a retry being required, we abort setting `VM_FAULT_RETRY` in the return value to indicate to the page fault mechanism that the fault should be retried (again see Chapter 6 for more details on the page fault mechanism).

With the swapped out read from disk, we consider edge cases as shown in Listing 12-35.

3833

```
if (swapcache) {
```

```

3834      /*
3835      * Make sure try_to_free_swap or swapoff did not release the
3836      * swapcache from under us. The page pin, and pte_same test
3837      * below, are not enough to exclude that. Even if it is still
3838      * swapcache, we need to check that the page's swap has not
3839      * changed.
3840      */
3841      if (unlikely(!PageSwapCache(page) ||
3842                  page_private(page) != entry.val))
3843          goto out_page;
3844      :
3858      /*
3859      * If we want to map a page that's in the swapcache writable,
we
3860      * have to detect via the refcount if we're really the
exclusive
3861      * owner. Try removing the extra reference from the local LRU
3862      * pagevecs if required.
3863      */
3864      if ((vmf->flags & FAULT_FLAG_WRITE) && page == swapcache &&
3865          !PageKsm(page) && !PageLRU(page))
3866          lru_add_drain();
3867      }
3868      :
3871      /*
3872      * Back out if somebody else already faulted in this pte.
3873      */
3874      vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, vmf->address,
3875                                     &vmf->ptl);
3876      if (unlikely(!pte_same(*vmf->pte, vmf->orig_pte)))
3877          goto out_nomap;
3878
3879      if (unlikely(!PageUptodate(page))) {
3880          ret = VM_FAULT_SIGBUS;
3881          goto out_nomap;
3882      }
3883      :
3895      /*
3896      * Check under PT lock (to protect against concurrent fork() sharing
3897      * the swap entry concurrently) for certainly exclusive pages.
3898      */
3899      if (!PageKsm(page)) {
3900          :
3904          exclusive = pte_swp_exclusive(vmf->orig_pte);
3905          if (page != swapcache) {
3906              /*
3907              * We have a fresh page that is not exposed to the

```



```

3908         * swapcache -> certainly exclusive.
3909         */
3910         exclusive = true;
3911     } else if (exclusive && PageWriteback(page) &&
3912             data_race(si->flags & SWP_STABLE_WRITES)) {
3913         /*
3914          * This is tricky: not all swap backends support
3915          * concurrent page modifications while under writeback
3916          *
3917          * So if we stumble over such a page in the swapcache
3918          * we must not set the page exclusive, otherwise we
3919          * can
3920          * map it writable without further checks and modify
3921          * it
3922          * while still under writeback.
3923          *
3924          * For these problematic swap backends, simply drop
3925          * the
3926          * exclusive marker: this is perfectly fine as we
3927          * start
3928          * writeback only if we fully unmapped the page and
3929          * there are no unexpected references on the page
3930          * after
3931          * unmapping succeeded. After fully unmapped, no
3932          * further GUP references (FOLL_GET and FOLL_PIN) can
3933          * appear, so dropping the exclusive marker and
3934          * mapping
3935          * it only R/O is fine.
3936          */
3937         exclusive = false;
3938     }
3939 }

```

Listing 12-35: mm/memory.c: do_swap_page() Edge Case Handling

Now we have a lock on the page, we first consider cases where the swap cache has been set, checking to ensure that page has not been freed from the swap cache under us, or if it is still a swap cache page that it refers to the same swap entry that we faulted on, aborting if either of these are not the case.

We then carefully ensure that we do not have an elevated reference count on the folio, as in the instance of a write fault we explicitly need to know whether we exclusively map the entry.

If something is in a folio batch (previously known as lruvec) pending flush to the appropriate LRU, we flush it to the appropriate LRU list (see Sections 11.7.12 on folio batch drain and 11.2 in Chapter 11 for more on this mechanism)—the upshot of this is that drop the reference count as a re-

sult in order to ensure we avoid false negatives when determining whether a folio is exclusively referenced later.

Next, we acquire a lock on the PTE, stabilising it and then immediately check to ensure it has not changed from beneath us due to a race. If it has, we abort. Also, if at this point the page is not marked `PG_uptodate`, we must have encountered an error when reading swap data from the disk, and thus abort, indicating that a terminal `SIGBUS` signal should be sent to the faulting process.

Finally, we carefully determine whether the page is indeed mapped exclusively. We encode this in a swap flag based on whether the swapped out page, which we retrieve via `pte_swp_exclusive()`.

If the swap cache does not match the page we are adding a mapping for, then we know this is a newly allocated page and thus must by definition be exclusively mapped.

Otherwise, we consider the case where `pte_swp_exclusive()` indicates the page is indeed mapped exclusively, but is also in the process of write-back in a scenario where stable writes are required, as indicated by the `SWP_STABLE_WRITES` flag.

In this case, we must avoid indicating that the page is mapped exclusive, even if it is otherwise indicated, otherwise we will simply make the mapping writable and the swap backend, which cannot handle having the swap modified when writing back to disk, might encounter this scenario. We avoid it by simply not considering the folio mapped exclusive under these conditions.

After handling these edge cases, we move on to mapping the swapped in page and freeing up the swap and swap cache as appropriate. We examine this in Listing 12-36.

```

3935      /*
3936       * Remove the swap entry and conditionally try to free up the
3937       * swapcache.
3938       * We're already holding a reference on the page but haven't mapped it
3939       * yet.
3940       */
3941      swap_free(entry);
3942      if (should_try_to_free_swap(page, vma, vmf->flags))
3943          try_to_free_swap(page);
3944
3945      inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
3946      dec_mm_counter_fast(vma->vm_mm, MM_SWAPENTS);
3947      pte = mk_pte(page, vma->vm_page_prot);
3948
3949      /*
3950       * Same logic as in do_wp_page(); however, optimize for pages that are
3951       * certainly not shared either because we just allocated them without
3952       * exposing them to the swapcache or because the swap entry indicates
3953       * exclusivity.
3954       */
3955      if (!PageKsm(page) && (exclusive || page_count(page) == 1)) {

```

```

3955         if (vmf->flags & FAULT_FLAG_WRITE) {
3956             pte = maybe_mkwrite(pte_mkdirty(pte), vma);
3957             vmf->flags &= ~FAULT_FLAG_WRITE;
3958             ret |= VM_FAULT_WRITE;
3959         }
3960         rmap_flags |= RMAP_EXCLUSIVE;
3961     }
3962     :
3963     if (pte_swp_soft_dirty(vmf->orig_pte))
3964         pte = pte_mksoft_dirty(pte);
3965     :
3966     vmf->orig_pte = pte;
3967     :
3968     page_add_anon_rmap(page, vma, vmf->address, rmap_flags);
3969     :
3970     set_pte_at(vma->vm_mm, vmf->address, vmf->pte, pte);
3971     :
3972     unlock_page(page);
3973     if (page != swapcache && swapcache) {
3974         /*
3975          * Hold the lock to avoid the swap entry to be reused
3976          * until we take the PT lock for the pte_same() check
3977          * (to avoid false positives from pte_same). For
3978          * further safety release the lock after the swap_free
3979          * so that the swap count won't change under a
3980          * parallel locked swapcache.
3981          */
3982         unlock_page(swapcache);
3983         put_page(swapcache);
3984     }
3985     :
3986     if (vmf->flags & FAULT_FLAG_WRITE) {
3987         ret |= do_wp_page(vmf);
3988         if (ret & VM_FAULT_ERROR)
3989             ret &= VM_FAULT_ERROR;
3990         goto out;
3991     }
3992 }

```

Listing 12-36: mm/memory.c: `do_swap_page()` Mapping Page and Freeing Swap

We start by freeing the swap entry associated with the underlying swap file via `swap_free()` (we don't examine this in detail as with the rest of the underlying swap file logic due to leaning rather towards file system code), and then attempt to free the swap cache if it makes sense to via `try_to_free_swap()` (see Listing 12-31 in Section 12.2.4).

We determine whether we should via `should_try_to_free_swap()` which we examine in Listing 12-37.

```

3638 static inline bool should_try_to_free_swap(struct page *page,

```

```

3639                                     struct vm_area_struct *vma,
3640                                     unsigned int fault_flags)
3641 {
3642     if (!PageSwapCache(page))
3643         return false;
3644     if (mem_cgroup_swap_full(page) || (vma->vm_flags & VM_LOCKED) ||
3645         PageMlocked(page))
3646         return true;
3647     /*
3648      * If we want to map a page that's in the swapcache writable, we
3649      * have to detect via the refcount if we're really the exclusive
3650      * user. Try freeing the swapcache to get rid of the swapcache
3651      * reference only in case it's likely that we'll be the exclusive user.
3652      */
3653     return (fault_flags & FAULT_FLAG_WRITE) && !PageKsm(page) &&
3654         page_count(page) == 2;
3655 }

```

Listing 12-37: mm/memory.c: should_try_to_free_swap()

We start `should_try_to_free_swap()` by checking whether the page is marked swap cache—of course, if it is not, we cannot free it from the swap cache and so indicate false.

Then, if the page has been marked by `mlock()` to be kept resident in memory, it simply cannot be swapped out, so maintaining it in the swap cache makes absolutely no sense and so in this instance we indicate that it should be freed.

Finally, we consider the case of a write fault in which the swap cache is the only other entity holding a reference on this page, if so then it makes sense to free the swap cache entry to drop this in order that the page has an exclusive mapping.

NOTE

We are so careful about exclusive mapping in the instance of a write fault, as we later invoke `do_wp_page()` if the mapping is not exclusive, which might result in the page being copied if this is not so.

Returning to the portion of `do_swap_page()` explored in Listing 12-36, after we potentially free the swap cache entry, we update statistics accordingly to account for the swap in and establish a new PTE entry that points to the read in page.

Next, all of the hard work to ensure we accurately determine whether we are exclusively mapped pays off, as check whether we have determined explicitly that this is an exclusive mapping (via the exclusive boolean variable) or if the reference count of the folio is equal to one, as checked by `page_count()`.

If this is the case, and this is a write fault, we simply make the PTE write-only via `maybe_mkwrite()` (the “maybe” here refers to the fact that we double check that the containing struct `vm_area_struct` (VMA) has the `VM_WRITE` flag set, indicating that it can be written to), clear the `FAULT_FLAG_WRITE` fault flag

and set `VM_FAULT_WRITE` in the return value to simply cause the mapping to be made writable.

We also set the `RMAP_EXCLUSIVE` flag when adding the folio to the reverse mapping later to indicate to this logic that this is an exclusive mapping.

Next, we reinstate any *soft-dirty* flag that was previously set on the mapping as indicated by the swap flag that stored this.

We add this page to the reverse mapping via `page_add_anon_rmap()` (see Listing 7-21 in Section 7.0.12 and Chapter 7 for a detailed treatment of this functionality), before finally setting the PTE via `set_pte_at()` (see Chapter 3 for more on page tables).

With the page read from the cache, added to the reverse mapping and the PTE set up, we can now unlock the page. We also check to see if we hold a reference to a now redundant swapcache page, which we free if we do so.

Finally, if the `FAULT_FLAG_WRITE` flag is set on the fault, i.e. this was a write fault but non-exclusive, then we must handle this as a Copy-on-Write page fault, as handled by `do_wp_page()` (see Listing 6-36 in Section 6.9 and Chapter 6 for a detailed exploration of this).

With the swap page now read in from disk or the swap cache, and the PTE pointing to it installed in place, we have only to examine cleanup and error cases, which we explore in Listing 12-38.

```

3935 unlock:
3936     pte_unmap_unlock(vmf->pte, vmf->ptl);
3937 out:
3938     if (si)
3939         put_swap_device(si);
3940     return ret;
3941 out_nomap:
3942     pte_unmap_unlock(vmf->pte, vmf->ptl);
3943 out_page:
3944     unlock_page(page);
3945 out_release:
3946     put_page(page);
3947     if (page != swapcache && swapcache) {
3948         unlock_page(swapcache);
3949         put_page(swapcache);
3950     }
3951     if (si)
3952         put_swap_device(si);
3953     return ret;
3954 }
```

Listing 12-38: `mm/memory.c: do_swap_page()` Suffix

If the swap in operation works correctly, or we exited early, we revoke the PTE lock, drop any reference to the swap device and return the fault handling value that we have calculated throughout the operation.

In the case of an error arising where the mapping has failed, we release the PTE lock, unlock the page if necessary and drop a reference to the page

if necessary, before doing the same for the swap cache page if it differs and dropping the reference to the swap device if necessary, before returning the specified fault handling return value.

This simply mirrors cleanup on the successful path through the function, accounting for the fact that the error paths will not have had a chance to do this themselves yet.

This concludes the analysis of the swap fault mechanism, which is the means by which memory is swapped back in after being swapped out, or simply retrieved from the swap cache if not yet successfully swapped out.

NOTE

In the case of a page being retrieved from the swap cache before the write out to disk is complete, we will have marked the swap entry clear and available for reuse by anything else swapping out via `swap_free()`, meaning that we do not occupy swap space unnecessarily.

12.3.2 Looking Up a Folio in the Swap Cache

In `do_swap_page()`, we attempt to look up the page in the swap cache via `lookup_swap_cache()`, which we examine in Listing 12-39 (eliding out of scope huge page logic).

```

319
320 /*
321  * Lookup a swap entry in the swap cache. A found page will be returned
322  * unlocked and with its refcount incremented - we rely on the kernel
323  * lock getting page table operations atomic even if we drop the page
324  * lock before returning.
325  */
326 struct page *lookup_swap_cache(swp_entry_t entry, struct vm_area_struct *vma,
327                               unsigned long addr)
328 {
329     struct page *page;
330     struct swap_info_struct *si;
331
332     si = get_swap_device(entry);
333     if (!si)
334         return NULL;
335     page = find_get_page(swap_address_space(entry), swp_offset(entry));
336     put_swap_device(si);
337
338     if (page) {
339         bool vma_ra = swap_use_vma_readahead();
340         bool readahead;
341         :
342         readahead = TestClearPageReadahead(page);
343         if (vma && vma_ra) {
344             unsigned long ra_val;
345             int win, hits;

```

```

353
354         ra_val = GET_SWAP_RA_VAL(vma);
355         win = SWAP_RA_WIN(ra_val);
356         hits = SWAP_RA_HITS(ra_val);
357         if (readahead)
358             hits = min_t(int, hits + 1, SWAP_RA_HITS_MAX);
359         atomic_long_set(&vma->swap_readahead_info,
360                        SWAP_RA_VAL(addr, win, hits));
361     }
362
363     if (readahead) {
364         count_vm_event(SWAP_RA_HIT);
365         if (!vma || !vma_ra)
366             atomic_inc(&swapin_readahead_hits);
367     }
368 }
369
370 return page;
371 }

```

Listing 12-39: mm/swap_state.c: lookup_swap_cache()

We start by first pinning the swap device via `get_swap_device()`, and if this yields `NULL` indicating we were unable to, we also return `NULL` indicating we could not find a swap cache entry.

Next we leverage the fact we are using a struct `address_space` object for the swapper by utilising the page cache's `find_get_page()` function, which ultimately invokes `__filemap_get_folio()` (see Listing 9-49 in Section 9.5.3 and Chapter 9 for more details) to look up the page in the struct `address_space`'s xarray.

With that done, we drop the reference to the swap device via `put_swap_device()`.

The remainder of the function concerns swap readahead, a topic we will explore in more detail in Section 12.3.3, but this is similar to the page cache's readahead logic, in that we read a batch of folios into the swap cache, marking individual pages with the `PG_readahead` flag, such that when that one is retrieved from the swap cache, it indicates that we should trigger a read of further folios.

In the case of swap readahead, we do not do this right away, but rather simply record the fact that we ought to read in more pages on the next occasion on which we read from the swap.

The method by which the swap readahead does this is to increment the number of folios to readahead on each occasion a `PG_readahead` flag is encountered.

There are two types of swap readahead algorithms available—cluster readahead and VMA readahead. The former reads sequential swap slots, and the latter reads sequential virtual pages.

Cluster readahead is less efficient than VMA readahead, as users read data sequentially in the virtual address space, , however it is more efficient in terms of disk accesses for slow rotational devices such as hard drives (as opposed to SSDs for whom bursts of small random accesses are less problematic).

The function `swap_use_vma_readahead()` determine which to use, and is predicated on a sysfs-tunable flag which allows a user to completely disable swap readahead (`enable_vma_readahead`) and a count of the number of swap files which are based on rotational media (`nr_rotate_swap`)—if this is greater than zero, VMA readahead is disabled.

The VMA readahead state is stored in `struct vm_area_struct->swap_readahead_info` which can be retrieved using the macro `GET_SWAP_RA_VAL()` (and generated via `SWAP_RA_VAL()`).

This state value encodes three integers—a window size (which can be extracted from the readahead value via `SWAP_RA_WIN()`), a count of the number of times folios with `PG_readahead` set have been hit (which can be obtained from the readahead value via `SWAP_RA_HITS()`), and a base page-aligned address which can be obtained from the readahead value via `SWAP_RA_ADDR()`.

Note that by default, `GET_SWAP_RA_VAL()` encodes a hit count of four, and all else zeroed in order to provide an initial number of folios to retrieve.

We will explore how this functions in more detail in Section 12.3.3, but returning to `lookup_swap_cache()` in Listing 12-39, we observe that, if VMA readahead is enabled, we simply increment the hit count (up to a maximum of `SWAP_RA_HITS_MAX`) alongside the existing window size and the address which we looked up, if indeed the `PG_readahead` flag was set on the looked up folio.

Otherwise, if the `PG_readahead` flag was not set, we simply update the address value to the one we just looked up.

If VMA readahead is not enabled (or if no VMA was specified), we update the `swpin_readahead_hits` atomic static value, which is utilised by the cluster readahead algorithm.

Finally, we return the retrieved page, if we located one.

12.3.3 Reading Swapped Out Folios From Disk

When reading a swapped out page directly from the disk, we do so with `swap_readpage()`, which we examine in Listing 12-40 (eliding out of scope working set, delayed account, front swap logic and debug asserts).

```

448 int swap_readpage(struct page *page, bool synchronous,
449                  struct swap_iocb **plug)
450 {
451     struct bio *bio;
452     int ret = 0;
453     struct swap_info_struct *sis = page_swap_info(page);
454     :
455     if (data_race(sis->flags & SWP_FS_OPS)) {
456         swap_readpage_fs(page, plug);
457         goto out;
458     }

```



```

479     }
480
481     if (sis->flags & SWP_SYNCHRONOUS_IO) {
482         ret = bdev_read_page(sis->bdev, swap_page_sector(page), page);
483         if (!ret) {
484             count_vm_event(PSWPIN);
485             goto out;
486         }
487     }
488
489     ret = 0;
490     bio = bio_alloc(sis->bdev, 1, REQ_OP_READ, GFP_KERNEL);
491     bio->bi_iter.bi_sector = swap_page_sector(page);
492     bio->bi_end_io = end_swap_bio_read;
493     bio_add_page(bio, page, thp_size(page), 0);
494     /*
495      * Keep this task valid during swap readpage because the oom killer
may
496      * attempt to access it in the page fault retry time check.
497      */
498     if (synchronous) {
499         get_task_struct(current);
500         bio->bi_private = current;
501     }
502     count_vm_event(PSWPIN);
503     bio_get(bio);
504     submit_bio(bio);
505     while (synchronous) {
506         set_current_state(TASK_UNINTERRUPTIBLE);
507         if (!READ_ONCE(bio->bi_private))
508             break;
509
510         blk_io_schedule();
511     }
512     __set_current_state(TASK_RUNNING);
513     bio_put(bio);
514
515 out:
516     :
517     return ret;
518 }
519
520 }

```

Listing 12-40: mm/page_io.c: swap_readpage()

Similar to `__swap_writepage()`, we won't delve too deeply into `swap_readpage()`'s implementation as this leans into file I/O logic rather than memory management somewhat.

We start by checking if `SWP_FS_OPS` is set on the swap device, which indicates that swap operations should pass through the file system. In this instance, we defer the operation to `swap_readpage_fs()`. For brevity we will not examine this in detail.

If the swap device indicates it is more efficient to access data sequentially via the `SWP_SYNCHRONOUS_IO` flag, then we go ahead and read the page directly via `bdev_read_page()`.

We then ultimately generate a struct bio object (see Section 9.10.5 in Chapter 9 for more details on this) which we submit to the block device via `submit_bio()`.

If the synchronous parameter is specified, then we place the task in an un-interruptible sleep while we wait for the IO to complete via `blk_io_schedule()`.

Overall this function ultimately invokes the block I/O layer to read the data from disk, asynchronously (unless synchronous is set).

When performing readahead as instigated by `do_swap_page()`, the `swpin_readahead()` function is invoked. we explore this in Listing 12-41.

```

835  /**
836   * swpin_readahead - swap in pages in hope we need them soon
837   * @entry: swap entry of this memory
838   * @gfp_mask: memory allocation flags
839   * @vmf: fault information
840   *
841   * Returns the struct page for entry and addr, after queueing swpin.
842   *
843   * It's a main entry function for swap readahead. By the configuration,
844   * it will read ahead blocks by cluster-based(ie, physical disk based)
845   * or vma-based(ie, virtual address based on faulty address) readahead.
846   */
847  struct page *swpin_readahead(swp_entry_t entry, gfp_t gfp_mask,
848                               struct vm_fault *vmf)
849  {
850      return swap_use_vma_readahead() ?
851          swap_vma_readahead(entry, gfp_mask, vmf) :
852          swap_cluster_readahead(entry, gfp_mask, vmf);
853  }

```

Listing 12-41: mm/swap_state.c: `swpin_readahead()`

The `swpin_readahead()` function uses `swap_use_vma_readahead()` (as discussed in Section 12.3.2) to determine whether to perform cluster readahead (explored in Section 12.3.4) via `swap_cluster_readhead()` (as shown in Listing ??), or VMA readahead (explored in Section 12.3.5) via `swap_vma_readahead()` (as shown in Listing 12-48).

The `read_swap_cache_async()` helper function is used to allocate a page, read it into the swap cache and then read swapped out data from disk into the page, optionally synchronously. We examine it in Listing 12-42.

```

507  /*

```

```

508  * Locate a page of swap in physical memory, reserving swap cache space
509  * and reading the disk if it is not already cached.
510  * A failure return means that either the page allocation failed or that
511  * the swap entry is no longer in use.
512  */
513  struct page *read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
514                                   struct vm_area_struct *vma,
515                                   unsigned long addr, bool do_poll,
516                                   struct swap_iocb **plug)
517  {
518      bool page_was_allocated;
519      struct page *retpage = __read_swap_cache_async(entry, gfp_mask,
520                                                    vma, addr, &page_was_allocated);
521
522      if (page_was_allocated)
523          swap_readpage(retpage, do_poll, plug);
524
525      return retpage;
526  }

```

Listing 12-42: mm/swap-state.c: `read_swap_cache_async()`

Ultimately the `read_swap_cache_async()` function wraps `__read_swap_cache_async()`, which we examine in Listing 12-43.

This allocates a page and adds it to the swap cache. If this succeeds, then we read data from disk into the page via `swap_readpage()`, which we examine in Listing 12-40.

We examine `__read_swap_cache_async()` in Listing 12-43 (eliding out of scope working set and cgroup logic).

```

409  struct page *__read_swap_cache_async(swp_entry_t entry, gfp_t gfp_mask,
410                                     struct vm_area_struct *vma, unsigned long addr,
411                                     bool *new_page_allocated)
412  {
413      struct swap_info_struct *si;
414      struct page *page;
415      :
416
417      *new_page_allocated = false;
418
419      for (;;) {
420          int err;
421          /*
422           * First check the swap cache. Since this is normally
423           * called after lookup_swap_cache() failed, re-calling
424           * that would confuse statistics.
425           */
426          si = get_swap_device(entry);
427          if (!si)
428              return NULL;

```

```

429         page = find_get_page(swap_address_space(entry),
430                               swp_offset(entry));
431         put_swap_device(si);
432         if (page)
433             return page;
434
435         /*
436          * Just skip read ahead for unused swap slot.
437          * During swap_off when swap_slot_cache is disabled,
438          * we have to handle the race between putting
439          * swap entry in swap cache and marking swap slot
440          * as SWAP_HAS_CACHE. That's done in later part of code or
441          * else swap_off will be aborted if we return NULL.
442          */
443         if (!__swp_swapcount(entry) && swap_slot_cache_enabled)
444             return NULL;
445
446         /*
447          * Get a new page to read into from swap. Allocate it now,
448          * before marking swap_map SWAP_HAS_CACHE, when -EEXIST will
449          * cause any racers to loop around until we add it to cache.
450          */
451         page = alloc_page_vma(gfp_mask, vma, addr);
452         if (!page)
453             return NULL;
454
455         /*
456          * Swap entry may have been freed since our caller observed it
457          */
458         err = swapcache_prepare(entry);
459         if (!err)
460             break;
461
462         put_page(page);
463         if (err != -EEXIST)
464             return NULL;
465
466         /*
467          * We might race against __delete_from_swap_cache(), and
468          * stumble across a swap_map entry whose SWAP_HAS_CACHE
469          * has not yet been cleared. Or race against another
470          * __read_swap_cache_async(), which has set SWAP_HAS_CACHE
471          * in swap_map, but not yet added its page to swap cache.
472          */
473         schedule_timeout_uninterruptible(1);
474     }

```

```

475
476     /*
477      * The swap entry is ours to swap in. Prepare the new page.
478      */
479
480     __SetPageLocked(page);
481     __SetPageSwapBacked(page);
482     :
483
484     /* May fail (-ENOMEM) if XArray node allocation failed. */
485     if (add_to_swap_cache(page, entry, gfp_mask & GFP_RECLAIM_MASK, &
486         shadow))
487         goto fail_unlock;
488     :
489
490     /* Caller will initiate read into locked page */
491     lru_cache_add(page);
492     *new_page_allocated = true;
493     return page;
494
495 fail_unlock:
496     put_swap_page(page, entry);
497     unlock_page(page);
498     put_page(page);
499     return NULL;
500 }

```

Listing 12-43: mm/swap-state.c: `__read_swap_cache_async()`

In `__read_swap_cache_async()` we immediately enter into a loop, which is used to account for the fact that we may race against other swap operations and want to retry our operation if this occurs.

We start by looking into the swap cache to see if the page is already present, via `find_get_page()` (this ultimately invokes `__filemap_get_folio()` (see Listing 9-49 in Section 9.5.3 and Chapter 9 for more details), again taking advantage of the fact that we maintain the swap cache using a `struct address_space` object and therefore can reuse this page cache functionality here.

Throughout this lookup we ensure that the swap device is pinned via `get_swap_device()`, exiting if the device has been made unavailable.

If the page is present in the swap cache, we return it. Otherwise, we must add a page to the swap cache at the location specified by the `swp_entry_t` value encoded into `entry` we are instructed to read.

If we see that nothing is swapped out at this location, we skip the read-ahead if the swap cache is enabled. There is a race condition that means we cannot do this if it is disabled, as alluded to by the preceding comment.

We then allocate a page to add to the swap via `alloc_page_vma()` (see Chapter 2 for more on physical page allocation), returning `NULL` if we are unable to do so.

We invoke the swap file-specific logic in preparation for this entry to be added to the swap cache via `swapcache_prepare()`. We won't explore this in detail as the specifics of the swap file implementation are out of scope (as rather closer to file system code than memory management), but it's important to note this can cause an error, especially if the swap entry was freed underneath us.

If no error arose, then we break out of loop. If one did indeed occur, then we free up the page by decrementing its reference count, before determining whether the error was indeed one of the entry being freed in the meantime (indicated by `EEXIST`), if not we exit returning `NULL`, otherwise we wait a while, loop around and try again.

After the loop has exited, we are now ready to prepare the file and add it to the swap cache. We mark it locked and swap-backed (it will remain locked until data from disk is read into it elsewhere), and then add it to the swap cache via `add_to_swap_cache()` (see Listing 12-22).

If this fails we free up state and return the error, otherwise we add it to the appropriate LRU list (see Section 11.2 in Chapter 11 for more on LRU lists) via `lru_cache_add()` (which ultimately invokes `folio_add_lru()`, see Listing 11-88 also in Chapter 11).

12.3.4 Swap Cluster Readahead

Swap cluster readahead is a simplistic approach to readahead that uses the `swapin_readahead_hits` atomic static value to track the number of occasions that a `PG_readahead` flag has been encountered on a page when looking up from the swap cache (set by the readahead logic after reading in a number of pages at the point at which the logic determines it would be efficient to load another batch of pages).

This value is then used to determine how many more pages to readahead in each batch.

Cluster readahead is implemented in `swap_cluster_readahead()`, which we examine in Listing ?? (eliding out of scope block plug logic).

```

589 /**
590  * swap_cluster_readahead - swap in pages in hope we need them soon
591  * @entry: swap entry of this memory
592  * @gfp_mask: memory allocation flags
593  * @vmf: fault information
594  *
595  * Returns the struct page for entry and addr, after queueing swapin.
596  *
597  * Primitive swap readahead code. We simply read an aligned block of
598  * (1 << page_cluster) entries in the swap area. This method is chosen
599  * because it doesn't cost us any seek time. We also make sure to queue
600  * the 'original' request together with the readahead ones...
601  *
602  * This has been extended to use the NUMA policies from the mm triggering
603  * the readahead.
```

```

604  *
605  * Caller must hold read mmap_lock if vmf->vma is not NULL.
606  */
607  struct page *swap_cluster_readahead(swp_entry_t entry, gfp_t gfp_mask,
608                                     struct vm_fault *vmf)
609  {
610      struct page *page;
611      unsigned long entry_offset = swp_offset(entry);
612      unsigned long offset = entry_offset;
613      unsigned long start_offset, end_offset;
614      unsigned long mask;
615      struct swap_info_struct *si = swp_swap_info(entry);
616      :
617      bool do_poll = true, page_allocated;
618      struct vm_area_struct *vma = vmf->vma;
619      unsigned long addr = vmf->address;
620
621      mask = swapin_nr_pages(offset) - 1;
622      if (!mask)
623          goto skip;
624
625      do_poll = false;
626      /* Read a page_cluster sized and aligned cluster around offset. */
627      start_offset = offset & ~mask;
628      end_offset = offset | mask;
629      if (!start_offset) /* First page is swap header. */
630          start_offset++;
631      if (end_offset >= si->max)
632          end_offset = si->max - 1;
633      :
634      for (offset = start_offset; offset <= end_offset ; offset++) {
635          /* Ok, do the async read-ahead now */
636          page = __read_swap_cache_async(
637              swp_entry(swp_type(entry), offset),
638              gfp_mask, vma, addr, &page_allocated);
639          if (!page)
640              continue;
641          if (page_allocated) {
642              swap_readpage(page, false, &splug);
643              if (offset != entry_offset) {
644                  SetPageReadahead(page);
645                  count_vm_event(SWAP_RA);
646              }
647          }
648          put_page(page);
649      }
650  }
651  :

```

```

655         lru_add_drain();           /* Push any new pages onto the LRU now */
656     skip:
657         :
658         return read_swap_cache_async(entry, gfp_mask, vma, addr, do_poll, NULL
659     );
660 }

```

Listing 12-44: mm/swap_state.c: swap_cluster_readhead()

The approach in `swap_cluster_readhead()` is simple, we use `swpin_readahead_hits` to determine the number of pages to readahead in from consecutive swap slots and then read those in.

We determine how many pages to read via `swpin_nr_pages()` which we examine in Listing 12-45 shortly.

The returned value will be a power-of-2, so by subtracting 1 we obtain a bit mask we can use to calculate an start and end offset describing a range which is aligned to the cluster size and equal to its size.

We account for the fact that the first page stored in the swap file, i.e. at index 0, is a metadata header by offsetting if the start offset was masked to zero, and also cap the end offset according to the size of the swap file.

We then iterate through each of the entries, allocating a new page to read from disk and adding it to the swap cache via `__read_swap_cache_async()` (see Listing 12-43), and if this allocated successfully, initiating a read from disk into the that memory via `swap_readpage()` (see Listing 12-40).

For each page that is not the first, we set the `PG_readahead` flag in order that this causes `lookup_swap_cache()` (see Listing 12-39) to increment `swpin_readahead_hits`.

After each loop, we drop the increment page reference count, then force any batched operations to place pages into an LRU list (see Section 11.2 in Chapter 11 for more details on LRU lists) via `lru_add_drain()` (see Listing 11-110 in Section 11.7.12 and Chapter 11).

Finally, we make sure that page that triggered the readahead is read in via `read_swap_cache_async()`, which we examine in Listing 12-42 in Section 12.3.3.

We examine `swpin_nr_pages()` in Listing 12-45.

```

568 static unsigned long swpin_nr_pages(unsigned long offset)
569 {
570     static unsigned long prev_offset;
571     unsigned int hits, pages, max_pages;
572     static atomic_t last_readahead_pages;
573
574     max_pages = 1 << READ_ONCE(page_cluster);
575     if (max_pages <= 1)
576         return 1;
577
578     hits = atomic_xchg(&swpin_readahead_hits, 0);
579     pages = __swpin_nr_pages(READ_ONCE(prev_offset), offset, hits,
580                             max_pages,
581                             atomic_read(&last_readahead_pages));

```



```

582         if (!hits)
583             WRITE_ONCE(prev_offset, offset);
584         atomic_set(&last_readahead_pages, pages);
585
586         return pages;
587     }

```

Listing 12-45: mm/swap_state.c: `swapin_nr_pages()`

The `swapin_nr_pages()` function is parameterised on a number of static variables. The first is `page_cluster`, which determines the maximum power-of-two number of pages to readahead in cluster readahead mode.

This is set by `swap_setup()` early in the kernel lifetime, which we examine in Listing 12-46.

```

1068 /*
1069  * Perform any setup for the swap system
1070  */
1071 void __init swap_setup(void)
1072 {
1073     unsigned long megs = totalram_pages() >> (20 - PAGE_SHIFT);
1074
1075     /* Use a smaller cluster for small-memory machines */
1076     if (megs < 16)
1077         page_cluster = 2;
1078     else
1079         page_cluster = 3;
1080
1081     /*
1082      * Right now other parts of the system means that we
1083      * _really_ don't want to cluster much more
1084      */
1085 }

```

Listing 12-46: mm/swap.c: `swap_setup()`

The `swap_setup()` function sets the power-of-two number of pages to readahead in `page_cluster` to two if there are 16 megabytes or less available in the system, or three if there is more, in order to conserve memory in low memory devices.

This equates to a maximum of 4 pages for very low memory systems or 8 for all others.

Note that this can also be updated via the `sysctl` tunable `vm.page-cluster` for custom control of cluster readahead.

NOTE

We also use this value in VMA readahead to determine the maximum number of pages to readahead, see `swap_ra_info()` in Section 12.3.5 to see how this is used.

Returning to `swapin_nr_pages()` and Listing 12-45, we see that if the user has set `page_cluster` to zero, then we return one, indicating that in effect no readahead should take place.

After considering this edge case, we then atomically exchange the `swapin_readahead_hits` with zero, resetting this value and therefore retrieving the number of hits on pages with `PG_readahead` set (i.e. for swap cluster readahead, that is all of the pages that were readahead previously).

The `swapin_readahead_hits` value defaults to four if not previously set, to ensure the first cluster readahead results in a reasonable batch of pages being read in at once.

We retain a function-scope static variable `prev_offset` to keep track of the last offset within a swap file we encountered when there were no hits, which should equate to the beginning of a readahead range.

Equally, we track the last number of pages we calculated should be readahead in the function-scope static variable `last_readahead_pages`.

With these values in hand, we pass the last known of these and the offset, number of hits and maximum pages to readahead to `__swapin_nr_pages()` (see Listing 12-47), which does the heavy lifting of determining the number of pages to readahead.

```

528 static unsigned int __swapin_nr_pages(unsigned long prev_offset,
529                                     unsigned long offset,
530                                     int hits,
531                                     int max_pages,
532                                     int prev_win)
533 {
534     unsigned int pages, last_ra;
535
536     /*
537      * This heuristic has been found to work well on both sequential and
538      * random loads, swapping to hard disk or to SSD: please don't ask
539      * what the "+ 2" means, it just happens to work well, that's all.
540      */
541     pages = hits + 2;
542     if (pages == 2) {
543         /*
544          * We can have no readahead hits to judge by: but must not get
545          * stuck here forever, so check for an adjacent offset instead
546          * (and don't even bother to check whether swap type is same).
547          */
548         if (offset != prev_offset + 1 && offset != prev_offset - 1)
549             pages = 1;
550     } else {
551         unsigned int roundup = 4;
552         while (roundup < pages)
553             roundup <= 1;
554         pages = roundup;
555     }
556
557     if (pages > max_pages)
558         pages = max_pages;

```

```

559
560     /* Don't shrink readahead too fast */
561     last_ra = prev_win / 2;
562     if (pages < last_ra)
563         pages = last_ra;
564
565     return pages;
566 }

```

Listing 12-47: mm/swap_state.c: `__swpin_nr_pages()`

NOTE

Swap VMA readahead also uses this function to determine the number of pages to readahead, see `swap_ra_info()` in Listing 12-50 and Section ?? to see how this is used.

The `__swpin_nr_pages()` function, like much of the kernel, is highly heuristic and tuned to real-world workloads.

We start by determining a baseline number, equal to the number of readahead hits we've encountered (on the basis that if we have read a number of pages in a previously readahead range then reading ahead more is likely to be worthwhile) plus a minimum two pages.

If we are at two pages only, this indicates no hits were encountered, and we need to determine whether readahead is appropriate at all—we do so by checking to see if the offset is immediately adjacent to the previous offset at which no hits were registered.

If this is the case, then it suggests we are reading sequentially, and thus more readahead is appropriate, otherwise we reset pages to one.

If we have encountered hits, then we round up to a minimum of four pages, increasing in powers-of-two. If this exceeds the maximum allowed pages, we cap this number to the maximum.

Finally, having been passed in the `last_readahead_pages` value from `swpin_nr_pages()` (see Listing 12-45) which indicates the last number of pages that were determined should be readahead via cluster readahead as the `prev_win` parameter.

We make sure that we reduce the readahead window by at most half on each occasion it drops so we do not scale down readahead too quickly.

NOTE

We do not consider which swap file (otherwise known as swap type, i.e. the index of the swap file) these indexes pertain to, so inevitably there is a chance that we will miscalculate here. As this is all intended to be a heuristic in order to improve swap read performance, the decision has been made that keeping this simple trumps such miscalculations.

12.3.5 Swap VMA Readahead

When `swpin_readahead()` (see Listing 12-41) invokes VMA swap readahead, as determined by `swap_use_vma_readahead()` (see Section 12.3.2 for an exploration

of this), the `swap_vma_readahead()` function is invoked, which we explore in Listing 12-48 (eliding out of scope block plugging logic).

```

771 /**
772  * swap_vma_readahead - swap in pages in hope we need them soon
773  * @fentry: swap entry of this memory
774  * @gfp_mask: memory allocation flags
775  * @vmf: fault information
776  *
777  * Returns the struct page for entry and addr, after queueing swapin.
778  *
779  * Primitive swap readahead code. We simply read in a few pages whose
780  * virtual addresses are around the fault address in the same vma.
781  *
782  * Caller must hold read mmap_lock if vmf->vma is not NULL.
783  *
784  */
785 static struct page *swap_vma_readahead(swp_entry_t fentry, gfp_t gfp_mask,
786                                       struct vm_fault *vmf)
787 {
788     :
789     struct vm_area_struct *vma = vmf->vma;
790     struct page *page;
791     pte_t *pte, pentry;
792     swp_entry_t entry;
793     unsigned int i;
794     bool page_allocated;
795     struct vma_swap_readahead ra_info = {
796         .win = 1,
797     };
798 };
799
800     swap_ra_info(vmf, &ra_info);
801     if (ra_info.win == 1)
802         goto skip;
803     :
804     for (i = 0, pte = ra_info.ptes; i < ra_info.nr_pte;
805          i++, pte++) {
806         pentry = *pte;
807         if (!is_swap_pte(pentry))
808             continue;
809         entry = pte_to_swp_entry(pentry);
810         if (unlikely(non_swap_entry(entry)))
811             continue;
812         page = __read_swap_cache_async(entry, gfp_mask, vma,
813                                       vmf->address, &page_allocated);
814         if (!page)
815             continue;
816         if (page_allocated) {

```

```

818             swap_readpage(page, false, &splug);
819             if (i != ra_info.offset) {
820                 SetPageReadahead(page);
821                 count_vm_event(SWAP_RA);
822             }
823         }
824         put_page(page);
825     }
826     :
827     lru_add_drain();
828 skip:
829     :
830     return read_swap_cache_async(fentry, gfp_mask, vma, vmf->address,
831                                ra_info.win == 1, NULL);
832 }
833 }

```

Listing 12-48: mm/swap_state.c: swap_vma_readahead()

We start by establishing a struct `vma_swap_readahead` state object which we thread state through the operation. We examine this in Listing 12-49 (assuming a 64-bit architecture).

```

337 struct vma_swap_readahead {
338     unsigned short win;
339     unsigned short offset;
340     unsigned short nr_pte;
341     :
342     pte_t *ptes;
343     :
344 };
345
346 };

```

Listing 12-49: include/linux/swap.h: struct vma_swap_readahead

This keeps track of the window of pages over which reclaim should occur, the offset of the swap entry we are reading in the first instance and a count of the number of PTEs whose swap entries we intend to readahead as well as a pointer to the first PTE entry in the PTE page table containing the PTEs we are reading ahead (which can in effect be treated like an array).

We set up this state in `swap_ra_info()` which we examine in Listing 12-50, this contains core readahead logic which decides how readahead should proceed.

With this state established, we first check whether the window size is only a single entry, if so then we have no readahead to do and skip to the end of the function.

Otherwise, we iterate through PTEs—skipping those which are not swap PTEs or are non-swap PTE entries (out of scope for the book, but these are things like migration entries which share the characteristics of swap entries but don't refer to swap).

NOTE

We copy the PTE into pentry as we do not hold the lock on the PTEs and they may be unmapped or otherwise modified during the course of the loop.

We then read the page for the entry via `__read_swap_cache_async()` (see Listing 12-43), continuing the loop to the next entry if this failed.

If it succeeds, then we read from disk into the swap entry via `swap_readpage()` (see Listing 12-40).

If the offset doesn't refer to that of the swap entry requested to be read, i.e. refers to swap entries being read ahead, we mark these with `PG_readahead` so `lookup_swap_cache()` updates VMA swap readahead state when these are later read from (see Listing 12-39).

We drop the page reference, and after the loop is complete, we force any batched operations to place pages into an LRU list (see Section 11.2 in Chapter 11 for more details on LRU lists) via `lru_add_drain()` (see Listing 11-110 in Section 11.7.12 and Chapter 11).

Finally, and also if the readahead operation was skipped, we use the `read_swap_cache_async()` function to ensure the originally requested swap entry is read (see Listing 12-42 in Section 12.3.3).

We run this function synchronously in the case that we skipped readahead.

The `swap_ra_info()` function populates the struct `vma_swap_readahead` state object but also in doing so implements the logic of swap VMA readahead. We examine this in Listing 12-50 (assuming x86-64 architecture).

```

709 static void swap_ra_info(struct vm_fault *vmf,
710                          struct vma_swap_readahead *ra_info)
711 {
712     struct vm_area_struct *vma = vmf->vma;
713     unsigned long ra_val;
714     unsigned long faddr, pfn, fPFN;
715     unsigned long start, end;
716     pte_t *pte, *orig_pte;
717     unsigned int max_win, hits, prev_win, win, left;
718     :
719
720     max_win = 1 << min_t(unsigned int, READ_ONCE(page_cluster),
721                          SWAP_RA_ORDER_CEILING);
722
723     if (max_win == 1) {
724         ra_info->win = 1;
725         return;
726     }
727
728
729     faddr = vmf->address;
730     orig_pte = pte = pte_offset_map(vmf->pmd, faddr);
731
732     fPFN = PFN_DOWN(faddr);
733     ra_val = GET_SWAP_RA_VAL(vma);
734     pfn = PFN_DOWN(SWAP_RA_ADDR(ra_val));
735     prev_win = SWAP_RA_WIN(ra_val);

```

```

736     hits = SWAP_RA_HITS(ra_val);
737     ra_info->win = win = __swapin_nr_pages(pfn, fpfn, hits,
738                                         max_win, prev_win);
739     atomic_long_set(&vma->swap_readahead_info,
740                    SWAP_RA_VAL(faddr, win, 0));
741
742     if (win == 1) {
743         :
744         return;
745     }
746     :
747
748     if (fpfn == pfn + 1)
749         swap_ra_clamp_pfn(vma, faddr, fpfn, fpfn + win, &start, &end);
750     else if (pfn == fpfn + 1)
751         swap_ra_clamp_pfn(vma, faddr, fpfn - win + 1, fpfn + 1,
752                         &start, &end);
753     else {
754         left = (win - 1) / 2;
755         swap_ra_clamp_pfn(vma, faddr, fpfn - left, fpfn + win - left,
756                         &start, &end);
757     }
758     ra_info->nr_pte = end - start;
759     ra_info->offset = fpfn - start;
760     pte -= ra_info->offset;
761     :
762     ra_info->ptes = pte;
763     :
764 }

```

Listing 12-50: mm/swap_state.c: swap_ra_info()

In `swap_ra_info()` we make use of the `page_cluster` static variable to determine the maximum number of pages to readahead, as also used for swap cluster readahead (see Section 12.3.4).

NOTE

In swap VMA readahead we limit the window size to a maximum of the power-of-two value `SWAP_RA_ORDER_CEILING`, which is hard-coded to five, meaning 32 pages.

We term this the maximum window over which readahead might occur. If this is equal to one, then no readahead will happen and we can simply exit indicating so by setting struct `vma_swap_readahead->win` to one.

At this point, since we are examining a series of contiguous virtual addresses, we need to read the PTE of each mapping in order to retrieve the `swp_entry_t` value stored within these entries.

In order to be able to efficiently do this, we take advantage of the fact we are invoked on the fault path and therefore have access to the struct `vm_fault` object describing the fault (see Chapter 6 for more details on fault handling in general), from which we can obtain the PTE of the faulting address.

We set `faddr` to the faulting address and both `orig_pte` and `pte` to the faulting PTE.

In order to avoid any expensive page table walks, we constrain ourselves to the remainder of the PTE page table starting at the faulting PTE. The logic for this lies later in the code, firstly we retrieve the virtual PFN (see Chapter 2 for more on physical addresses and Page Frame Numbers or PFNs – essentially the address divided by page size, that is if you consider memory to be in effect an array of base pages, the index into that array) containing the page in which the faulting address resides via `PFN_DOWN()`, and store it in `fpfn`.

We say “virtual” PFN here as typically PFNs refer to a page offset for physical addresses, however in this case we are using the same concept only for the virtual fault address.

We retrieve the struct `vm_area_struct` (VMA) encoded swap VMA readahead state via `GET_SWAP_RA_VAL()` (stored in struct `vm_area_struct->swap_readahead_info`), from which we derive state about the last swap lookup. The number of readahead hits is written by `lookup_swap_cache()` (see Listing 12-39) on swap lookup, and the last faulting address and window size are written by this function later in the code.

We use this to retrieve the virtual PFN of the last retrieved address which we store in `pfn`, the previous window size (i.e. the previous number of pages we determined to readahead) as stored in `prev_win` and the number of hits stored in `hits`.

We reuse the `__swapon_nr_pages()` function (see Listing 12-47) as also used in the swap cluster readahead logic.

One key difference here is rather than using offsets into the swap, we use virtual PFN offsets in order to measure distance between pages being read from swap, hence we pass `pfn` as `prev_offset` and `fpfn` as `offset`.

After determining the number of pages to readahead, we update the struct `vma_swap_readahead->win` field and the local `win` value accordingly.

With these determined, we reset the hit count to zero and update the VMA swap readahead state in struct `vm_area_struct->swap_readahead_info` to contain the faulting address and determined window size to be the previous values for these when next we readahead. We generate this value via `SWAP_RA_VAL()`.

Again if we determine that the window size is equal one at this stage, this indicates that no readahead should occur, so we exit in this case.

Otherwise, we must be careful to ensure that we examine only PTEs in this page table and not beyond. We do so via the `swap_ra_clamp_pfn()` function, which we examine in Listing 12-51.

This function accepts parameters for the VMA, the faulting address, the left-most PFN and the right-most PFN over which we propose to readahead, as well as output parameters for the determined start and end of the readahead range.

We consider three different scenarios:

Forward Sequential Access The swap is being faulted in moving forwards through sequential pages, so readahead from the page `fpgn` to the (exclusive) end of the window `fpgn + win`.

Backwards Sequential Access The swap is being faulted in moving backwards through sequential pages, so readahead up to and including the faulting page between `fpgn - win + 1` and `fpgn + 1`.

Random Access The swap is being faulted in using some other access pattern, so split the difference and readahead pages half of which lie prior to the faulting address, and half of which lie after it.

Once we have determined start and end virtual PFN values, we can use these to determine the number of PFNs for `swap_vma_readahead()` to read through, and is simply determined to be `end - start` which we place in `struct vma_swap_readahead->nr_pte`.

Equally the faulting PFN is offset from the start of the range by the number of pages between `fpgn` and `start` so we populate `struct vma_swap_readahead->offset` with this value.

We offset `pte` by this offset value, and then assign it to `struct vma_swap_readahead->pte` to provide the first PTE entry in the PTE page table to be examined by `swap_vma_readahead()`.

We examine the key `swap_ra_clamp_pfn()` function in Listing 12-51.

```

696 static inline void swap_ra_clamp_pfn(struct vm_area_struct *vma,
697                                     unsigned long faddr,
698                                     unsigned long lpgn,
699                                     unsigned long rpgn,
700                                     unsigned long *start,
701                                     unsigned long *end)
702 {
703     *start = max3(lpgn, PFN_DOWN(vma->vm_start),
704                  PFN_DOWN(faddr & PMD_MASK));
705     *end = min3(rpgn, PFN_DOWN(vma->vm_end),
706                PFN_DOWN((faddr & PMD_MASK) + PMD_SIZE));
707 }
```

Listing 12-51: mm/swap_state.c: swap_ra_clamp_pfn()

The `swap_ra_clamp_pfn()` function determines start and end virtual PFN values which are equal to `lpgn` and `rpgn` respectively, only clamped to the start and end of the VMA and PTE page tables containing the PTE entries mapping the pages containing `lpgn` and `rpgn`.

We obtain `start` by taking the maximum of `lpgn`, the virtual PFN of the `struct vm_area_struct` (VMA)'s start address, and the lowest virtual PFN contained within the PTE which also contains the PTE entry mapping the `lpgn` page.

We determine the latter of these values by applying the `PMD_MASK` to the faulting address, which masks the high bits of the address which identify

which PMD page table it refers to, and thus constraining the address to a single PMD entry, which is the PTE page table.

NOTE

This might be slightly hard to follow, as visualising how page tables are laid out can get confusing fast, review Chapter 3 on virtual memory layout to observe how the page tables interact with one another.

We obtain end by taking the minimum of rpf_n, the virtual PFN of the struct `vm_area_struct` (VMA)'s (exclusive) end address, and the virtual PFN of the next PTE page table after the one containing the faulting address.

The end value is always an exclusive value, i.e. the last virtual PFN in the readahead range plus one.

As usual the complicated part of determining end is the constraint to the end of the page table. We essentially apply the same technique we did for start by applying the `PMD_MASK` mask, only we also offset by `PMD_SIZE` to ensure we get the page table after the one containing the faulting address.

Since we obtain the minimum of these three values, we never fall off the end of either the VMA or the PTE page table.

13

THE OUT OF MEMORY (OOM) KILLER

When we are unable to allocate memory due to zones dipping below their minimum watermark, memory reserves are exhausted, and direct reclaim has failed we must take drastic action to avoid the system becoming unusable. At this stage the Out of Memory (OOM) Killer steps in, finds the biggest memory hog and kills the process and all associated threads. In this chapter we explore how this is implemented and the criteria the OOM killer uses to determine which process to eliminate.

One of the central tenets of the Linux memory management subsystem is the disconnect between virtual memory *mappings* and the physical memory which *backs* them.

Ordinarily, a mapping performed by `malloc()` or `mmap()` will result in absolutely no physical memory being allocated. Instead, the `mm_struct` object describing a process's address space object will have a new `struct vm_area_struct` (Virtual Memory Area or “VMA”) object assigned to it, describing the mapped range (see chapters 3 and 4 for an in-depth discussion).

The actual allocation of memory is therefore only performed when a process attempts to access it, at which point a CPU hardware fault arises which is trapped by the kernel and processed as a “page fault” (see Chapter 6).

On page fault, the faulting address is checked to determine whether it is address is within a valid VMA range. If so, memory is allocated as necessary, if not a segfault signal is sent to the process.

As a result, it becomes entirely possible for a process to access a valid mapping that causes the kernel to attempt to allocate physical memory, only to find it is unable to do so.

In this situation the kernel must do *something* in order to maintain system stability, as the mapping of the memory is a long-distant memory and there is no mechanism by which to indicate to the process which mapped it that the allocation could not be performed.

NOTE

It's also important to note that, even if the mapping process could somehow be informed, it would most likely result in that process simply terminating, as it is rarely the case that an arbitrary allocation can simply be allowed to fail. Given that the process(es) which are causing the memory pressure might well be distinct from the one happening to fault, this wouldn't even necessarily help relieve the underlying memory pressure anyway.

That something is the Out of Memory (OOM) killer, which resolves the situation by determining which process is using the most memory (subject to manual user adjustment via the `/proc/oom_score_adj` interface see Section 13.2) and kills it, thus (hopefully) freeing sufficient memory to complete the failing allocation.

The selection of a process to kill (the “victim”) and freeing it (“reaping” it) are two separate events, occurring at different times. The victim is selected as soon as the OOM killer event occurs, and then a kernel thread assigned to reaping victim threads is woken to perform the actual freeing of memory.

A failed allocation will be retried as long as forward progress is made in freeing up memory, and the selection of an OOM victim constitutes progress, thus the actual result of the killing will tell after it has been initiated. See Section 13.4 below for a detailed examination of how this occurs and Chapter 11 for a detailed examination of how forward progress is utilised as a concept on reclaim.

A manual invocation of the OOM killer can also be performed via the `sysrq-f` operation, see Section 13.1.2 for details.

13.1 Causes of Out of Memory Conditions

13.1.1 Memory Allocation

As described in Chapter 2, in order to ensure that the system never reaches a point where memory is so critically low that the kernel might not be able to function, we must maintain minimum reserves and establish *watermarks* below which we start to attempt to free up memory.

These watermarks are maintained *per-zone*, per-NUMA *node* (if NUMA is enabled) and determine the point at which *reclaim* occurs. Reclaim is the means by which the kernel frees up memory without killing any processes (see chapter 11 for a detailed exploration of reclaim).

While this is covered in chapter 2, but briefly—a zone is simply a named physical memory range. For instance, the first 16 MiB of x86-64 machines is designated `ZONE_DMA`, as this is the only range of physical memory very old DMA devices could access. Each NUMA node further subdivides the physical memory range, associating CPUs with the memory they are electronically local to. Nodes can span multiple zones.

NOTE

NUMA is an abbreviation for Non-Uniform Memory Access, and describes the mechanism by which the kernel handles architectures which have memory which is not equally fast to access from all CPUs. Beyond the fundamentals around nodes and zones, this is out of scope for the book.

For file-backed memory, reclaim can be as simple as dropping memory from the page cache (see chapter 9), for anonymous i.e. non-file backed memory this involves swapping memory out to disk. Kernel memory allocated using the slab allocator might also reduce cache sizes via *shrinkers* (the slab allocator is out of scope for the book).

When physical memory is allocated, it ultimately originates from a specific zone (see chapter 2 for the details as to how this is designated). When doing so, the zone watermarks are checked. There are three core watermarks – `WMARK_MIN`, `WMARK_LOW` and `WMARK_HIGH` (there’s also `WMARK_PROMO`, but this is used by NUMA for very specific purposes which are out of scope here).

When the low watermark is reached, *indirect* reclaim is started in the form of the `kswapd` kernel thread (there is one process per NUMA node). This performs reclaim in the background until at least one zone which could serve the allocation reaches the high watermark, at which point it sleeps (again, see chapter 11 for more on reclaim).

If the memory pressure is extreme, this might not be enough – in this instance, a zone might reach the *minimum* water mark at which point *direct* reclaim can occur – this is reclaim that the process must wait until before proceeding.

Finally, if the direct reclaim cannot free up sufficient memory, we are left in a difficult position where we are simply unable to service a page fault, which renders the process unable to continue executing.

The simplest solution to this would be to kill the faulting process causing the allocation, however this would be unfair – the disconnect between mapping and backing of memory in linux means that a faulting process may have absolutely no relation to one which is using excessive memory.

NOTE

It’s actually possible to cause the kernel to kill the allocating process via the `vm.oom_kill_allocating_task` tunable. See Section 13.3 for details.

Instead, we invoke the Out Of Memory killer to elect the most appropriate victim to kill whose memory we can immediately free. This maintains system stability even in the face of extreme memory pressure.

It's worth reflecting on how we might end up in this situation in the first place – how can a process allocate more memory than the system has available? The answer is *overcommit* – by default, the kernel does not constrain memory mapping such that it must fit in available physical memory, rather it simply disallows any one mapping which exceeds available physical and swap memory (see Chapter 4 for more details on this).

13.1.2 *sysrq-f*

The *sysrq* interface is a means by which users can perform actions using a special key combination (typically alt + the sysrq/print screen key + a letter to indicate the task to perform).

This may not even be enabled by your kernel (you can determine this by examining the `kernel.sysrq` tunable which masks out various available sysrq operations). Even if unavailable from the command line, it's possible to trigger sysrq events by writing to the `/proc/sysrq-trigger` procfs interface.

An OOM killer-relevant operation exposed by this interface is `sysrq-f`, which causes the out of memory killer to be unconditionally invoked and to kill the process using the most resources as if an allocation had failed.

This will not result in a kernel panic no matter the outcome of the OOM killer invocation, it will simply perform the kernel's process selection logic and terminate the selected process.

See Section 13.4 for a detailed analysis as to how this actually implemented within the kernel.

13.2 OOM Killer Score Adjustment

It is possible to adjust the likelihood of a process being selected for termination via the OOM killer score adjustment interface as exposed per-process via procfs in `/proc/$pid/oom_score_adj` (there is also an `/proc/$pid/oom_adj` interface, however this is deprecated and simply used to calculate a value to set `oom_score_adj` to so we disregard it).

The score is standardised to a range of -1000 to 1000, and defaults to 0. At 0, the process is as likely as any other (unadjusted) process to be selected by the OOM killer. At -1000 the process will never be killed by the OOM killer, and at 1000 the process will almost certainly be selected (unless an alternative process which also has a very high adjustment score assigned uses more memory).

When the OOM killer considers a process for termination, it does so by determining how much resident memory it is utilising and compares it to all the other processes in the system, selecting the one that is using the most resident memory.

NOTE

It's possible to "constrain" the OOM killer to specific cgroups, cpusets or NUMA arrangements, however this is out of the scope of the book so we consider only the "global" OOM killer scenario where we examine all running processes and examine their total resident memory usage as a proportion of all available memory.

This calculation is performed in terms of the number of base pages each process is using, a value which we can adjust per-process using the per-process `oom_score_adj` value.

The adjustment score is expressed in units of one tenth of a percent of total available memory (combining RAM and total swap). This is used to calculate the equivalent in base pages and this is simply added to the number of base pages of RAM and swap a candidate process is using, thus adjusting the likelihood of that process being selected.

This score can therefore be seen as either permitting a process to utilise a certain percentage of system memory before being considered for termination (if negative) or to be treated as if it already uses a certain percentage of memory before examining its actual usage (if positive).

For instance, a process with an `oom_score_adj` of -400 can utilise 40% of memory and swap before it will be considered for termination (unless all other processes have negative adjustment scores), and a process with an `oom_score_adj` of 20 will be treated as if it is already using 2% more of memory and swap than it actually is.

NOTE

A value of -1000, i.e. the minimum possible OOM score adjustment, is given special treatment and explicitly excludes a process from being considered by the OOM killer.

See Section 13.4 for a detailed analysis of how the kernel actually performs the selection of the "bad" process to be terminated.

13.2.1 OOM Score

It's possible to read the current "score" of a process via `/proc/$pid/oom_score`, as implemented by `proc_oom_score()`.

This determines the number of base pages of memory a process occupies including any adjustment by its `/proc/$pid/oom_score_adj`, converting this into integer units expressed in tenth of a percent of total RAM and swap and then "normalising" it to a range of 0 to 2000.

This normalisation is performed by offsetting the tenth of a percent score value by 1000 (thus mapping the minimum possible value to 0 rather than -1000), and scaling this sum by two thirds.

This scaling by two thirds accounts for the absolute maximum a process could (theoretically, but not practically) occupy, i.e. 100% of memory and the maximum possible `oom_score_adj`, which summed to the offset value would equal 3000, scaling by two thirds constrains this to a maximum value of 2000 instead.

This dance is performed as the value had always been constrained to this range in the past, and userland processes might be reliant on this remaining the case.

13.3 OOM Killer Alternative Behaviours

It's possible to adjust the standard behaviour of the OOM killer further via two specific tunables—`vm.panic_on_oom` and `vm.oom_kill_allocating_task`.

13.3.1 `vm.panic_on_oom`

The `vm.panic_on_oom` tunable causes the OOM killer to unconditionally cause a kernel panic in the event of an Out of Memory condition, except if it was triggered by `sysrq-f` (see Section 13.1.2).

This tunable can be tailored to only cause the panic on global OOM killer invocations rather than ones constrained to cgroup, cpusets or NUMA arrangements, however these types of OOM killer events are out of scope of the book so we disregard this mode of operation.

This might be useful in situations where it is simply unacceptable for the OOM killer to be invoked and it being so is indicative of the system failing to perform as required.

13.3.2 `vm.oom_kill_allocating_task`

The `vm.oom_kill_allocating_task` tunable, if set, causes the allocating process to be the one which is terminated regardless of whether that process utilises a large amount of memory or not.

NOTE

If the process's `oom_score_adj` is set to the minimum value, i.e. `-1000`, then even if this tunable is set it will not be terminated and instead the default OOM killer behaviour will proceed.

This is unlikely to be the kind of behaviour that a user would actually desire in practice, especially as the faulting process might itself be utilising very little memory, and the process which is actually causing the problem might cause a great deal of collateral damage of other faulting processes before it is finally reaped.

13.4 Kernel Interface

We examine an overview of the kernel interface in Figure 13-1.

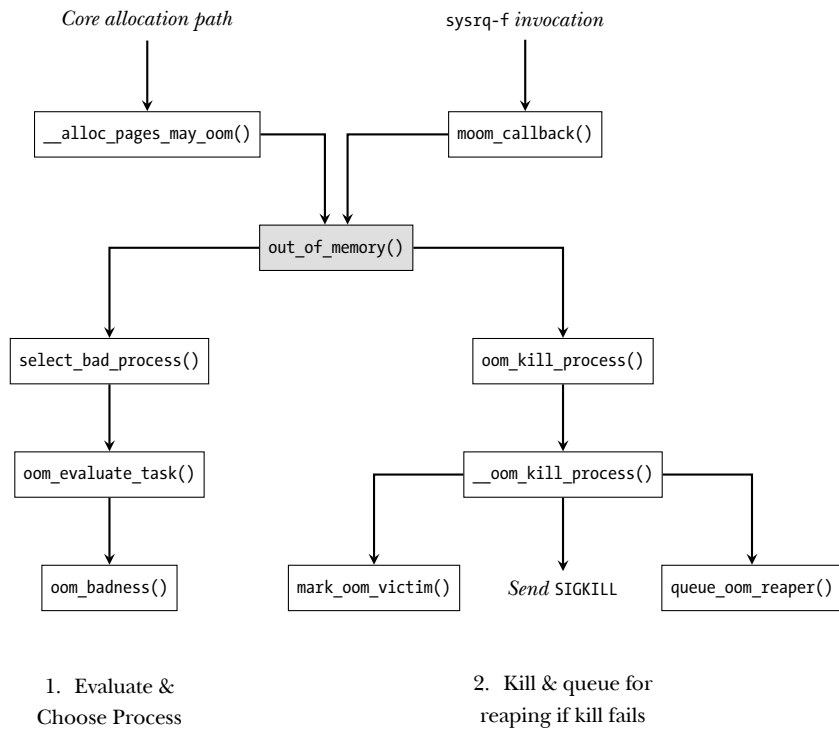


Figure 13-1: OOM Killer Kernel Interface

The OOM killer implementation is conveniently entirely implemented in the `mm/oom_kill.c` file. It is invoked via the `out_of_memory()` function and parameterised by a struct `oom_control` object, as shown in Listing 13-1.

```

25 /*
26  * Details of the page allocation that triggered the oom killer that are used
27  * to
28  * determine what should be killed.
29  */
29 struct oom_control {
30     /* Used to determine cpuset */
31     struct zonelist *zonelist;
32
33     /* Used to determine mempolicy */
34     nodemask_t *nodemask;
35
36     /* Memory cgroup in which oom is invoked, or NULL for global oom */
37     struct mem_cgroup *memcg;
38
39     /* Used to determine cpuset and node locality requirement */
40     const gfp_t gfp_mask;
41
42     /*

```

```

43      * order == -1 means the oom kill is required by sysrq, otherwise only
44      * for display purposes.
45      */
46      const int order;
47
48      /* Used by oom implementation, do not set */
49      unsigned long totalpages;
50      struct task_struct *chosen;
51      long chosen_points;
52
53      /* Used to print the constraint info. */
54      enum oom_constraint constraint;
55 };

```

Listing 13-1: include/linux/oom.h: struct oom_control

Before we investigate the OOM killer's code, let's examine the fields of this object which controls the process.

zonelist A struct zonelist (See Chapter 2 for more details, but broadly this is an array of struct zoneref objects each containing a link to a struct zone object (which describes the portion of a zone belonging to a specific node) and an enum zone_type index value which indicates which *type* of zone this is, e.g. ZONE_DMA, ZONE_DMA32, ZONE_NORMAL, etc.) object containing a list of zones spanned by this NUMA node if the __GFP_THISNODE flag is set, otherwise zones spanned by all nodes, starting with this one in order of the expense of accessing memory on each node. From the point of view of the OOM killer, this is referenced in constrained_alloc() to determine whether the cgroup cpuset functionality constrains the allocation (see the cgroup v2 admin guide on cpuset settings). cgroups are out of scope for the book.

nodemask If a NUMA memory policy is in force which restricts which nodes can be utilised for the allocation, then this parameter will be non-NULL and indicate on which nodes processes we consider for killing may be located. A detailed discussion of NUMA memory policies is out of scope for the book.

memcg Out of memory conditions can occur within a memory cgroup controller or memcg. If so, the condition will be triggered by mem_cgroup_out_of_memory(), which will specify the memcg which has entered an out of memory condition.

Out of memory conditions for memcgs are handled somewhat differently than the global kind and follow different code paths, however discussion of this topic is out of scope for the book.

gfp_mask The Get Free Pages (GFP) flags specified for the physical allocation which caused the out of memory failure. These parameterise kernel allocation functions, modifying how these allocations are performed. This is useful information in a few respects – for the NUMA case, if

`__GFP_THISNODE` is specified, then we know not to further consider node constraints.

In addition this is informational and allows us to report it back to the user.

order Indicates the order of the allocation being performed for informational purposes only. Set to -1 if `sysrq-f` was used to trigger the OOM killer manually.

totalpages *internal* The total number of base pages of memory available in the system as a whole (comprising both physical memory and swap).

chosen *internal* Tracks the process chosen to be terminated.

chosen_points *internal* Tracks the “badness points” assigned to the chosen process.

constraints *internal* Indicates memory policy, cgroup, etc. constraints applied to the out of memory killer invocation. The discussion of these are out of scope for the book.

13.4.1 Allocating Memory at Risk of OOM

As explored in Chapter 2, we attempt to allocate memory using Per-CPU-Page (PCP) lists, failing this we try to allocate such that no zone has its memory reduced below the low water mark. Should this fail, we enter into the allocation “slow path” and invoke `__alloc_pages_slowpath()` (see Section 11.1 and the Listings starting with 11-1 in Chapter 11 for more details).

Reclaim will, under this circumstance, attempt to allocate from a zone at the minimum watermark. Should this fail, and the allocation flags permit it, direct reclaim will be attempted to synchronously make available memory to fulfil the request.

If, finally, this fails to free up sufficient memory to satisfy the allocation, then `__alloc_pages_may_oom()` is invoked to make a last gasp attempt at allocating the memory, invoking the OOM killer and freeing up memory if necessary to do so.

We explore this function in Listing 13-2.

```

4380 static inline struct page *
4381 __alloc_pages_may_oom(gfp_t gfp_mask, unsigned int order,
4382                      const struct alloc_context *ac, unsigned long *did_some_progress)
4383 {
4384     struct oom_control oc = {
4385         .zonelist = ac->zonelist,
4386         .nodemask = ac->nodemask,
4387         .memcg = NULL,
4388         .gfp_mask = gfp_mask,
4389         .order = order,
4390     };
4391     struct page *page;
4392 
```

```

4393     *did_some_progress = 0;
4394
4395     /*
4396      * Acquire the oom lock. If that fails, somebody else is
4397      * making progress for us.
4398      */
4399     if (!mutex_trylock(&oom_lock)) {
4400         *did_some_progress = 1;
4401         schedule_timeout_uninterruptible(1);
4402         return NULL;
4403     }
4404
4405     /*
4406      * Go through the zonelist yet one more time, keep very high watermark
4407      * here, this is only to catch a parallel oom killing, we must fail if
4408      * we're still under heavy pressure. But make sure that this reclaim
4409      * attempt shall not depend on __GFP_DIRECT_RECLAIM && !_GFP_NORETRY
4410      * allocation which will never fail due to oom_lock already held.
4411      */
4412     page = get_page_from_freelist((gfp_mask | __GFP_HARDWALL) &
4413                                   ~__GFP_DIRECT_RECLAIM, order,
4414                                   ALLOC_WMARK_HIGH|ALLOC_CPUSET, ac);
4415     if (page)
4416         goto out;
4417
4418     /* Coredumps can quickly deplete all memory reserves */
4419     if (current->flags & PF_DUMPCORE)
4420         goto out;
4421     /* The OOM killer will not help higher order allocs */
4422     if (order > PAGE_ALLOC_COSTLY_ORDER)
4423         goto out;
4424     /*
4425      * We have already exhausted all our reclaim opportunities without any
4426      * success so it is time to admit defeat. We will skip the OOM killer
4427      * because it is very likely that the caller has a more reasonable
4428      * fallback than shooting a random task.
4429      *
4430      * The OOM killer may not free memory on a specific node.
4431      */
4432     if (gfp_mask & (__GFP_RETRY_MAYFAIL | __GFP_THISNODE))
4433         goto out;
4434     /* The OOM killer does not needlessly kill tasks for lowmem */
4435     if (ac->highest_zoneidx < ZONE_NORMAL)
4436         goto out;
4437     if (pm_suspended_storage())
4438         goto out;
4439     /*

```

```

4440      * XXX: GFP_NOFS allocations should rather fail than rely on
4441      * other request to make a forward progress.
4442      * We are in an unfortunate situation where out_of_memory cannot
4443      * do much for this context but let's try it to at least get
4444      * access to memory reserved if the current task is killed (see
4445      * out_of_memory). Once filesystems are ready to handle allocation
4446      * failures more gracefully we should just bail out here.
4447      */
4448
4449      /* Exhausted what can be done so it's blame time */
4450      if (out_of_memory(&oc) ||
4451          WARN_ON_ONCE_GFP(gfp_mask & __GFP_NOFAIL, gfp_mask)) {
4452          *did_some_progress = 1;
4453
4454          /*
4455           * Help non-failing allocations by giving them access to
4456           * reserves
4457           */
4458          if (gfp_mask & __GFP_NOFAIL)
4459              page = __alloc_pages_cpuset_fallback(gfp_mask, order,
4460                                                  ALLOC_NO_WATERMARKS, ac);
4461      }
4462      out:
4463          mutex_unlock(&oom_lock);
4464          return page;
4465      }

```

Listing 13-2: mm/page_alloc.c: `__alloc_pages_may_oom()`

The function starts by attempting to acquire the global out of memory lock, `oom_lock`. It treats the lock being contended as an indicator that another process is performing out of memory handling, and treats this as indicating that progress is being made.

NOTE

The `__alloc_pages_may_oom()` function operates in a best effort fashion—if progress of any kind has been made, the calling `__alloc_pages_slowpath()` function will loop around and try the allocation again.

We attempt to allocate memory again via `get_page_from_freelist()`, only setting a very high watermark and disallowing direct reclaim—this is a last gasp attempt to see whether we can avoid killing a process, as if a process is being torn down in parallel this may succeed in which case it's safe to abort the operation.

We then handle some edge cases—if the current process is performing a core dump then all memory reserves might quickly be depleted if we permit the allocation to succeed.

Equally higher order folios which reclaim struggles to provide (as determined by the order exceeding `PAGE_ALLOC_COSTLY_ORDER`, hardcoded to order-3)

are unlikely to be helped by the freeing of folios from a memory hog, so in this case we also fail the allocation.

Equally if the `__GFP_RETRY_MAYFAIL` flag is set then it is fine for us to fail and we should not resort to invoking the OOM killer. Also if `__GFP_THISNODE` is set, we cannot help as we are unable to restrict the operation to this node alone.

If the maximum zone that can be allocated is less than `ZONE_NORMAL`, we again cannot be certain that the killing of a process will free memory in the required zone, so we abort in this case also.

Finally, if the global `gfp_allowed_mask` value has `__GFP_IO` or `__GFP_FS` cleared, as checked by `pm_suspended_storage()`, then this indicates that the freeing of memory via the OOM killer will not succeed correctly and thus we must abort the allocation.

Finally, we are ready to actually invoke the OOM killer, which is done via `out_of_memory()`. If this succeeds, or if the `__GFP_NOFAIL` flag was set, indicating that we cannot tolerate an allocation failure, we indicating progress was indeed made, and if the `__GFP_NOFAIL` flag is set we attempt an allocation with no watermark limitations performed in order to access all available memory reserves.

See Chapter 2 for more details on GFP flags and physical allocation, and Chapter 11 for more on the calling `__alloc_pages_slowpath()` function.

We examine `out_of_memory()` in Listing 13-5 and Section 13.4.3.

13.4.2 Manual OOM Killer Invocation Via `sysrq-f`

When a user invokes the `sysrq` key stroke with the `f` key (typically Control + Sysrq/PrintScrn + F), and `sysrq` is enabled at a level that permits this (or the user manually passes `f` to `/proc/sysrq-trigger`), this invokes a manual run of the OOM killer via `sysrq_handle_moom()` which ultimately invokes `moom_callback()` which we examine in Listing 13-3.

```

385 static void moom_callback(struct work_struct *ignored)
386 {
387     const gfp_t gfp_mask = GFP_KERNEL;
388     struct oom_control oc = {
389         .zonelist = node_zonelist(first_memory_node, gfp_mask),
390         .nodemask = NULL,
391         .memcg = NULL,
392         .gfp_mask = gfp_mask,
393         .order = -1,
394     };
395
396     mutex_lock(&oom_lock);
397     if (!out_of_memory(&oc))
398         pr_info("OOM request ignored. No task eligible\n");
399     mutex_unlock(&oom_lock);
400 }
```

Listing 13-3: `drivers/tty/sysrq.c: moom_callback()`

We configure an `struct oom_control` object and pass it to `out_of_memory()` in the same way we invoke the OOM killer on allocation, however importantly we set the order to -1 to indicate that this is a manual invocation.

If the attempt to kill a memory hog fails, then we simply report that this effort failed here.

Within the OOM killer implementation we use the fact that the order of a manual `sysrq`-invoked OOM killer invocation sets order to -1 to identify this and avoid certain behaviours that are pertinent only to low memory scenarios such as `panic-on-oom` and `panicking` should no process be found to terminate. This check is performed by `is_sysrq_oom()` which we examine in Listing 13-4.

```

152 /*
153  * order == -1 means the oom kill is required by sysrq, otherwise only
154  * for display purposes.
155  */
156 static inline bool is_sysrq_oom(struct oom_control *oc)
157 {
158     return oc->order == -1;
159 }
```

Listing 13-4: `mm/oom_kill.c: is_sysrq_oom()`

13.4.3 The Out of Memory Killer

The key function which performs the out of memory handling is `out_of_memory()`, which we explore in Listing 13-5 (eliding OOM killer notifier and `cgroup/NUMA` constraint logic).

```

1098 /**
1099  * out_of_memory - kill the "best" process when we run out of memory
1100  * @oc: pointer to struct oom_control
1101  *
1102  * If we run out of memory, we have the choice between either
1103  * killing a random task (bad), letting the system crash (worse)
1104  * OR try to be smart about which process to kill. Note that we
1105  * don't have to be perfect here, we just have to be good.
1106  */
1107 bool out_of_memory(struct oom_control *oc)
1108 {
1109     unsigned long freed = 0;
1110
1111     if (oom_killer_disabled)
1112         return false;
1113     :
1121     /*
1122      * If current has a pending SIGKILL or is exiting, then automatically
1123      * select it. The goal is to allow it to allocate so that it may
```



```

1124     * quickly exit and free its memory.
1125     */
1126     if (task_will_free_mem(current)) {
1127         mark_oom_victim(current);
1128         queue_oom_reaper(current);
1129         return true;
1130     }
1131
1132     /*
1133     * The OOM killer does not compensate for IO-less reclaim.
1134     * pagefault_out_of_memory lost its gfp context so we have to
1135     * make sure exclude 0 mask - all other users should have at least
1136     * __GFP_DIRECT_RECLAIM to get here. But mem_cgroup_oom() has to
1137     * invoke the OOM killer even if it is a GFP_NOFS allocation.
1138     */
1139     if (oc->gfp_mask && !(oc->gfp_mask & __GFP_FS) && !is_memcg_oom(oc))
1140         return true;
1141     :
1142
1143     check_panic_on_oom(oc);
1144
1145     if (!is_memcg_oom(oc) && sysctl_oom_kill_allocating_task &&
1146         current->mm && !oom_unkillable_task(current) &&
1147         :
1148         current->signal->oom_score_adj != OOM_SCORE_ADJ_MIN) {
1149         get_task_struct(current);
1150         oc->chosen = current;
1151         oom_kill_process(oc, "Out of memory (oom_kill_allocating_task)
1152     ");
1153
1154         return true;
1155     }
1156
1157     select_bad_process(oc);
1158     /* Found nothing?!?! */
1159     if (!oc->chosen) {
1160         dump_header(oc, NULL);
1161         pr_warn("Out of memory and no killable processes...\n");
1162         /*
1163          * If we got here due to an actual allocation at the
1164          * system level, we cannot survive this and will enter
1165          * an endless loop in the allocator. Bail out now.
1166          */
1167         if (!is_sysrq_oom(oc) && !is_memcg_oom(oc))
1168             panic("System is deadlocked on memory\n");
1169     }
1170
1171     if (oc->chosen && oc->chosen != (void *)-1UL)
1172         oom_kill_process(oc, !is_memcg_oom(oc) ? "Out of memory" :
1173             "Memory cgroup out of memory");

```

```

1177         return !!oc->chosen;
1178     }

```

Listing 13-5: mm/oom_kill.c: out_of_memory()

We start by checking whether the OOM killer is completely disabled via the `oom_killer_disabled` global. This is disabled by `oom_killer_disable()` on power events or `kexec` (a means by which a new kernel can be loaded into a running kernel) in `freeze_processes()`. The OOM is re-enabled via `oom_killer_enable()`.

We then invoke the `task_will_free_mem()` on the current process (we examine this in Listing 13-7), which determines whether a process is in the process of freeing memory (i.e. by exiting and tearing down its memory), in which case we should mark it as the target OOM victim and queue it to be reaped immediately, exiting indicating the OOM killing has been initiated.

We mark a process as an OOM victim via `mark_oom_victim()` which we examine in Listing 13-18, and queue it for reaping via `queue_oom_reaper()` which we examine in Listing 13-21.

If a GFP mask has been specified by the failed allocation, and this has `__GFP_FS` cleared, this indicates that the filesystem cannot be accessed (for instance the allocation was made by a filesystem and this flag has been cleared to prevent an infinite regression), but this is not a condition we can satisfy, as killing a process can't provide memory in this context.

By convention, under these circumstances, we return `true`, indicating that we should loop around and attempt allocations again.

We then check to see whether we ought to panic on OOMs as specified by the `vm.panic_on_oom` tunable via `check_panic_on_oom()` which we examine in Listing 13-6 (eliding out of scope constrained NUMA/cgroup allocation checks).

```

1063
1064 static void check_panic_on_oom(struct oom_control *oc)
1065 {
1066     if (likely(!sysctl_panic_on_oom))
1067         return;
1068     :
1077     /* Do not panic for oom kills triggered by sysrq */
1078     if (is_sysrq_oom(oc))
1079         return;
1080     dump_header(oc, NULL);
1081     panic("Out of memory: %s panic_on_oom is enabled\n",
1082          sysctl_panic_on_oom == 2 ? "compulsory" : "system-wide");
1083 }

```

Listing 13-6: mm/oom_kill.c: check_panic_on_oom()

If the `sysctl_panic_on_oom` global variable has been set as a result of the `vm.panic_on_oom` tunable having been set, then we only fail to panic if the OOM killer was invoked via `sysrq-f`.

Returning to `out_of_memory()` in Listing 13-5, we then check for the `vm.oom_kill_allocating_task` tunable being set, checking firstly that this is a userland thread.

We determine whether it is a userland thread by asserting that the current thread's `struct task_struct->mm` value is non-NULL. This field points at the `struct mm_struct` describing the userland process's virtual address space), so it being NULL indicates that this process is a kernel thread (note that kernel threads may sometimes have this value set to something non-NULL, so this is sufficient, but not necessary).

We then check to see whether the current process is “unkillable” as determined by `oom_unkillable_task()` (which we examine in Listing ??), if not and the current process's `oom_score_adj` is not set to the minimum value `OOM_SCORE_ADJ_MIN`, then we go ahead and select the current process, setting `struct oom_control->chosen` to the current process (the current thread's describing `struct task_struct` object being pinned first via `get_task_struct()`).

Ultimately `oom_kill_process()` is invoked to perform the killing (we examine this later in Listing 13-15). Note that this is very different from the `task_will_free_mem()` case we examined above—this is because a lot more housekeeping needs to be done if the process (and thus related threads) are not currently in the process of exiting.

Finally in this instance we exit indicating the OOM killer has succeeded.

After this we have now eliminated all of the edge cases and reach the core logic—deferring the selection of the bad process to `select_bad_process()`, which we examine shortly in Listing 13-10.

If we cannot select a bad process this way, then we are in a state of crisis and report as such (unless this is a `sysrq-f` invocation of the OOM killer), panicking the kernel.

Otherwise, if a process was chosen we again rely on `oom_kill_process()` (which we examine in Listing 13-15) to perform the actual killing of the process.

We return to `task_will_free_mem()` examining it in Listing 13-7.

```

863  /*
864   * Checks whether the given task is dying or exiting and likely to
865   * release its address space. This means that all threads and processes
866   * sharing the same mm have to be killed or exiting.
867   * Caller has to make sure that task->mm is stable (hold task_lock or
868   * it operates on the current).
869   */
870  static bool task_will_free_mem(struct task_struct *task)
871  {
872      struct mm_struct *mm = task->mm;
873      struct task_struct *p;
874      bool ret = true;
875
876      /*
877       * Skip tasks without mm because it might have passed its exit_mm and
878       * exit_oom_victim. oom_reaper could have rescued that but do not rely

```

```

879      * on that for now. We can consider find_lock_task_mm in future.
880      */
881      if (!mm)
882          return false;
883
884      if (!__task_will_free_mem(task))
885          return false;
886
887      /*
888       * This task has already been drained by the oom reaper so there are
889       * only small chances it will free some more
890       */
891      if (test_bit(MMF_OOM_SKIP, &mm->flags))
892          return false;
893
894      if (atomic_read(&mm->mm_users) <= 1)
895          return true;
896
897      /*
898       * Make sure that all tasks which share the mm with the given tasks
899       * are dying as well to make sure that a) nobody pins its mm and
900       * b) the task is also reappable by the oom reaper.
901       */
902      rcu_read_lock();
903      for_each_process(p) {
904          if (!process_shares_mm(p, mm))
905              continue;
906          if (same_thread_group(task, p))
907              continue;
908          ret = __task_will_free_mem(p);
909          if (!ret)
910              break;
911      }
912      rcu_read_unlock();
913
914      return ret;
915 }

```

Listing 13-7: mm/oom_kill.c: `task_will_free_mem()`

NOTE

Processes in Linux can be confusing—each struct `task_struct` object represents a task, which is referred to in the kernel as a process but maps to a thread. `for_each_process()` iterates over each task, i.e. each thread.

This function is parameterised by struct `task_struct` for the task we want to determine whether killing its owning process (or more strictly, thread group leader) and thus itself alongside it is about to free up memory (and

thus selection of the process as the ostensible victim avoids unnecessarily killing still-running processes).

This wraps the `__task_will_free_mem()` function which determines whether killing a process will free memory. We examine this later in Listing 13-9.

To start with, we exclude the case where its `struct mm_struct` virtual address space descriptor is not present, as we cannot be sure that we're not past the point where memory would already have been freed.

We then invoke `__task_will_free_mem()` to determine whether the task will indeed free memory. If not, then we trivially return false.

If so, then we have some edge cases to consider—we examine the `struct mm_struct->flags` field to see if the `MMF_OOM_SKIP` flag has been set. If it has, this indicates that memory has already been drained by the OOM reaper, so we are not going to see any more memory freed.

Finally if there is nothing pinning the virtual address space i.e. `struct mm_struct->mm_users` is not increased (`mm_users` equal to one indicates that userland alone makes use of the virtual mapping), then we have no reason to contradict the result of `__task_will_free_mem()` and return true.

If the virtual address space is being used by the kernel in some fashion, we need to consider a very specific edge case—consider tasks that were spawned in such a way as to share the virtual memory address space, but not sharing signals with other threads sharing the same address space.

This can be achieved via `clone()` with `CLONE_VM` set (indicating that the virtual address space should be shared), but without `CLONE_SIGHAND` set, indicating that it does process signals sent to the thread group (meaning that other threads utilising the same virtual address space will die, but it won't).

Also, a kernel thread can make use of a userland virtual address space and essentially adapt it to be its own via `kthread_use_mm()`, and equally this task will not exit alongside the userland thread group which established the virtual address space in the first instance.

To assess this, we iterate through all tasks via `for_each_process()`, ignore ones which do not share the `struct mm_struct`, checked via `process_shares_mm()` (see Listing 13-8), or ones which share the same signal handler, which we know will certainly exit alongside the task being queried (checked via).

Finally if we find any that do not fit into this category, then we have found an outlier as described above. We then check this task manually against `__task_will_free_mem()` (in Listing 13-9). If this fails, then we exit the loop and return false, otherwise if all checks pass, we return true.

```

484 /*
485  * task->mm can be NULL if the task is the exited group leader. So to
486  * determine whether the task is using a particular mm, we examine all the
487  * task's threads: if one of those is using this mm then this task was also
488  * using it.
489  */
490 bool process_shares_mm(struct task_struct *p, struct mm_struct *mm)
491 {
492     struct task_struct *t;
493 
```

```

494     for_each_thread(p, t) {
495         struct mm_struct *t_mm = READ_ONCE(t->mm);
496         if (t_mm)
497             return t_mm == mm;
498     }
499     return false;
500 }

```

Listing 13-8: mm/oom_kill.c: process_shares_mm()

The `process_shares_mm()` function iterates through each of the tasks associated with each thread belonging to the same thread group (i.e. each thread of the associated userland process), checking to see if any match the queried `struct mm_struct` virtual address space object.

This is necessary as each task might be exiting, and thus may not be referencing a `struct mm_struct`. If one exists then the process shares the virtual address space, even if it is partially exited.

```

842 static inline bool __task_will_free_mem(struct task_struct *task)
843 {
844     struct signal_struct *sig = task->signal;
845
846     /*
847      * A coredumping process may sleep for an extended period in
848      * coredump_task_exit(), so the oom killer cannot assume that
849      * the process will promptly exit and release memory.
850      */
851     if (sig->core_state)
852         return false;
853
854     if (sig->flags & SIGNAL_GROUP_EXIT)
855         return true;
856
857     if (thread_group_empty(task) && (task->flags & PF_EXITING))
858         return true;
859
860     return false;
861 }

```

Listing 13-9: mm/oom_kill.c: __task_will_free_mem()

The `__task_will_free_mem()` function starts by handling the edge case scenario of a process which is core dumping—this is a time-consuming process so even though the process is exiting, we are not going to see memory freed in time to be useful, so we consider this process to not be freeing memory for the purposes of the function.

We then consider the two cases which indicate that the process will exit—the `SIGNAL_GROUP_EXIT` flag has been set, indicating that a signal indicating process exit has been issued, or the `struct task_struct->flags` field has the `PF_EXITING` flag set indicating that the task is in process of being shut down

(since there is no signal, we must also check the thread group is empty, i.e. this is the only task, done via `thread_group_empty()`).

13.4.4 Victim Selection

We examine the function which selects the process `select_bad_process()` in Listing 13-10.

```

360 /*
361  * Simple selection loop. We choose the process with the highest number of
362  * 'points'. In case scan was aborted, oc->chosen is set to -1.
363  */
364 static void select_bad_process(struct oom_control *oc)
365 {
366     oc->chosen_points = LONG_MIN;
367     :
371     struct task_struct *p;
372
373     rcu_read_lock();
374     for_each_process(p)
375         if (oom_evaluate_task(p, oc))
376             break;
377     rcu_read_unlock();
378     :
379 }
```

Listing 13-10: `mm/oom_kill.c: select_bad_process()`

The `select_bad_process()` function accumulates “points” assigned to each process which map to the amount of resident memory used by each process (expressed as a signed long integer), starting at the minimum value possible.

The points assigned to the process with the maximum number of points is assigned to `struct oom_control->chosen_points`, along with the associated process in `struct oom_control->chosen`.

We simply iterate through each process, evaluating each via `oom_evaluate_task()` (shown in Listing 13-11) which calculates each process’s score, and updates the `struct oom_control->chosen_points` and `struct oom_control->chosen` fields.

If at any stage the `oom_evaluate_task()` function returns false, this indicates that the `struct oom_control->chosen` has been set to an invalid value indicating that the operation failed, and thus the loop should be exited early (note that this nonetheless indicates success, as the abort implies that the OOM killer need not select a new victim).

```

308 static int oom_evaluate_task(struct task_struct *task, void *arg)
309 {
310     struct oom_control *oc = arg;
311     long points;
312
```

```

313         if (oom_unkillable_task(task))
314             goto next;
315         :
320         /*
321          * This task already has access to memory reserves and is being killed
322          *
323          * Don't allow any other task to have access to the reserves unless
324          * the task has MMF_OOM_SKIP because chances that it would release
325          * any memory is quite low.
326          */
327         if (!is_sysrq_oom(oc) && tsk_is_oom_victim(task)) {
328             if (test_bit(MMF_OOM_SKIP, &task->signal->oom_mm->flags))
329                 goto next;
330             goto abort;
331         }
332         /*
333          * If task is allocating a lot of memory and has been marked to be
334          * killed first if it triggers an oom, then select it.
335          */
336         if (oom_task_origin(task)) {
337             points = LONG_MAX;
338             goto select;
339         }
340
341         points = oom_badness(task, oc->totalpages);
342         if (points == LONG_MIN || points < oc->chosen_points)
343             goto next;
344
345 select:
346         if (oc->chosen)
347             put_task_struct(oc->chosen);
348         get_task_struct(task);
349         oc->chosen = task;
350         oc->chosen_points = points;
351 next:
352         return 0;
353 abort:
354         if (oc->chosen)
355             put_task_struct(oc->chosen);
356         oc->chosen = (void *)-1UL;
357         return 1;
358     }

```

Listing 13-11: mm/oom_kill.c: `oom_evaluate_task()`

We start by evaluating whether the task under examination is unkillable, which we do via `oom_unkillable_task()` which we examine in Listing ??.

```

161 /* return true if the task is not adequate as candidate victim task. */
162 static bool oom_unkillable_task(struct task_struct *p)
163 {
164     if (is_global_init(p))
165         return true;
166     if (p->flags & PF_KTHREAD)
167         return true;
168     return false;
169 }

```

Listing 13-12: mm/oom_kill.c: oom_unkillable_task()

In `oom_unkillable_task()` we assess whether a task simply cannot be killed. There are two instances—the global init process, as tested by `is_global_init()` (this is the very first task run in userland) and kernel threads—we test this by determining if the task has the `PF_KTHREAD` flag set identifying it as such.

If neither of these are the case, then the task is not considered unkillable.

Returning to `oom_evaluate_task()`, after asserting that the task is in fact not unkillable, we then check, should this not be a `sysrq-f` invocation of the OOM killer (as this test is relevant only under severe memory pressure), whether the task is already marked as an OOM victim via `tsk_is_oom_victim()`.

In this instance, if the struct `mm_struct` which is currently being killed by the OOM killer does not have the `MMF_OOM_SKIP` flag set, this indicates that we should abort as a task in the process of being killed and thus any memory reserves may not be accessible at this time (but will become accessible as soon as the killing is complete).

If `MMF_OOM_SKIP` is set, this indicates that the virtual address space described by the associated struct `mm_struct` is of no interest to OOM and thus we can proceed with examining the next task.

When a process is known to be about to allocate a great deal of memory, it makes sense to ensure that, if the OOM killer is invoked, that it is the process selected as victim.

In these instances, `set_current_oom_origin()` is used to mark the current process as such, later cleared via `clear_current_oom_origin()` and tested for in `oom_evaluate_task()` via `oom_task_origin()`.

This is set by the ring buffer when allocating memory on behalf of a userland thread, by Kernel Shared Memory (KSM) under certain circumstances (out of scope for the book) and by the swap file mechanism when the user invokes `swapon` to disable a swap file.

In `oom_evaluate_task()` we check for this condition, and if it is present, mark this task as if it had the maximum possible points so it almost certainly gets selected.

Finally, once all edge cases have been considered, we take the usual route applicable to most processes wherein we evaluate the “badness” of the task via `oom_badness()` which we examine in Listing 13-13.

```

192 /**

```

```

193  * oom_badness - heuristic function to determine which candidate task to kill
194  * @p: task struct of which task we should calculate
195  * @totalpages: total present RAM allowed for page allocation
196  *
197  * The heuristic for determining which task to kill is made to be as simple
    and
198  * predictable as possible. The goal is to return the highest value for the
199  * task consuming the most memory to avoid subsequent oom failures.
200  */
201  long oom_badness(struct task_struct *p, unsigned long totalpages)
202  {
203      long points;
204      long adj;
205
206      if (oom_unkillable_task(p))
207          return LONG_MIN;
208
209      p = find_lock_task_mm(p);
210      if (!p)
211          return LONG_MIN;
212
213      /*
214       * Do not even consider tasks which are explicitly marked oom
215       * unkillable or have been already oom reaped or the are in
216       * the middle of vfork
217       */
218      adj = (long)p->signal->oom_score_adj;
219      if (adj == OOM_SCORE_ADJ_MIN ||
220          test_bit(MMF_OOM_SKIP, &p->mm->flags) ||
221          in_vfork(p)) {
222          task_unlock(p);
223          return LONG_MIN;
224      }
225
226      /*
227       * The baseline for the badness score is the proportion of RAM that
    each
228       * task's rss, pagetable and swap space use.
229       */
230      points = get_mm_rss(p->mm) + get_mm_counter(p->mm, MM_SWAPENTS) +
231              mm_pgtables_bytes(p->mm) / PAGE_SIZE;
232      task_unlock(p);
233
234      /* Normalize to oom_score_adj units */
235      adj *= totalpages / 1000;
236      points += adj;
237

```

```

238         return points;
239     }

```

Listing 13-13: mm/oom_kill.c: oom_badness()

Here we again assert that the task is not unkillable via `oom_unkillable_task()` (see Listing ??), using `LONG_MIN`, i.e. the minimum possible signed long integer value to provide effectively the minimum possible score thus to preclude this task from being chosen.

Next, we acquire a lock around the task to prevent it disappearing below us via `find_lock_task_mm()`, which we examine in Listing 13-14.

```

127  /*
128   * The process p may have detached its own ->mm while exiting or through
129   * kthread_use_mm(), but one or more of its subthreads may still have a valid
130   * pointer. Return p, or any of its subthreads with a valid ->mm, with
131   * task_lock() held.
132   */
133  struct task_struct *find_lock_task_mm(struct task_struct *p)
134  {
135      struct task_struct *t;
136
137      rcu_read_lock();
138
139      for_each_thread(p, t) {
140          task_lock(t);
141          if (likely(t->mm))
142              goto found;
143          task_unlock(t);
144      }
145      t = NULL;
146  found:
147      rcu_read_unlock();
148
149      return t;
150  }

```

Listing 13-14: mm/oom_kill.c: find_lock_task_mm()

In `find_lock_task_mm()` we iterate through each task representing a thread within the same process as the task passed in, determining if any maintain a reference to the struct `mm_struct` describing the virtual address space shared between them.

This is done as the process may be in the process of being destroyed but at least one thread might still maintain a reference to the virtual address space and that suffices for our needs.

Once we have established a lock on the task we can be certain it can't be removed, so it is therefore safe to return. Should we not find such a thread, we return `NULL`.

Returning to `oom_badness()` in Listing 13-13, we note that again should `find_lock_task_mm()` return `NULL`, we set the score to the minimum possible to indicate that this process should not be selected.

We then consider three final edge cases—if the `/proc/$pid/oom_score_adj` for the process is set to the minimum possible score (`OOM_SCORE_ADJ_MIN`), or if the process has had the `MMF_OOM_SKIP` flag set (indicating that the process is unlikely to yield any significant memory if freed), or if we are in the midst of a `vfork()` operation, then we indicate the task should not be selected by returning `LONG_MIN` (and of course, releasing the lock on the task before doing so).

The first two of these cases are self-explanatory—both are explicitly intended to indicate that the process is not to be considered for OOM killing, the third requires a little explanation, however.

A `vfork()` operation (which can also be achieved via the `clone()` system call specifying `CLONE_VFORK`), is similar to an ordinary `fork()`, only no page tables are copied and the parent process is suspended until the child exits or calls `execve()`.

In `oom_badness()` we check to see if the task is a vforked child of a parent via `in_vfork()`. In which case, it has no memory to free, and we consider its parent separately.

With all edge cases considered, we calculate the “badness” score for a process, i.e. an index value proportional to its resident memory process, taking into account any `/proc/$pid/oom_score_adj` modifiers.

We calculate the “badness” score by obtaining the sum of the resident base pages, swap entries and page tables used by the process, adding the `oom_score_adj` modifier, expressed in base pages (the adjustment score is expressed in thousandths of the total number of base pages available in the system).

After retrieving the pertinent data from the task, we relinquish its lock before adding the adjustment value and returning the same.

13.4.5 Victim Killing

Returning to `out_of_memory()` (as shown in Listing 13-5), we see that, once a process has been selected for termination, the actual killing of the process is deferred to `oom_kill_process()`, which we examine in Listing 13-15 (eliding out of scope `cgroup` handling).

```

1014 static void oom_kill_process(struct oom_control *oc, const char *message)
1015 {
1016     struct task_struct *victim = oc->chosen;
1017     :
1018     static DEFINE_RATELIMIT_STATE(oom_rs, DEFAULT_RATELIMIT_INTERVAL,
1019                                   DEFAULT_RATELIMIT_BURST);
1020
1021     /*
1022      * If the task is already exiting, don't alarm the sysadmin or kill
1023      * its children or threads, just give it access to memory reserves

```

```

1024     * so it can die quickly
1025     */
1026     task_lock(victim);
1027     if (task_will_free_mem(victim)) {
1028         mark_oom_victim(victim);
1029         queue_oom_reaper(victim);
1030         task_unlock(victim);
1031         put_task_struct(victim);
1032         return;
1033     }
1034     task_unlock(victim);
1035
1036     if (__ratelimit(&oom_rs))
1037         dump_header(oc, victim);
1038     :
1046     __oom_kill_process(victim, message);
1047     :
1058 }

```

Listing 13-15: mm/oom_kill.c: oom_kill_process()

We establish a ratelimit variable `oom_rs` in order to limit the repetition of OOM killer output. Discussion of the details of this are out of scope for the book.

We acquire a lock on the task, then again evaluate whether the process is being killed and thus will trivially free memory, via `task_will_free_mem()` as shown in Listing 13-7.

If so, then the task is marked as OOM victim (which in turn gives the process access to memory reserves in service of it freeing memory) via `mark_oom_victim()` (see Listing 13-18), and queue it for reaping via `queue_oom_reaper()` (see Listing 13-21), before unlocking it and reducing its reference count.

NOTE

In `oom_evaluate_task()` (as shown in Listing 13-11) we increment the chosen task's reference count via `get_task_struct()`, so we can be confident that the task will not have disappeared beneath us at this point.

We then output details about the chosen process (in a rate-limited fashion to avoid spamming the kernel ring buffer) via `dump_header()`, which we examine in Listing 13-16 (eliding out of scope cgroup logic).

```

452 static void dump_header(struct oom_control *oc, struct task_struct *p)
453 {
454     pr_warn("%s invoked oom-killer: gfp_mask=%#x(%pGg), order=%d,
455             oom_score_adj=%hd\n",
456             current->comm, oc->gfp_mask, &oc->gfp_mask, oc->order,
457             current->signal->oom_score_adj);
458     if (!IS_ENABLED(CONFIG_COMPACTION) && oc->order)
459         pr_warn("COMPACTION is disabled!!!\n");

```

```

459
460     dump_stack();
461     :
464         show_mem(SHOW_MEM_FILTER_NODES, oc->nodemask);
465         if (should_dump_unreclaim_slab())
466             dump_unreclaimable_slab();
467         :
468     if (sysctl_oom_dump_tasks)
469         dump_tasks(oc);
470     if (p)
471         dump_oom_summary(oc, p);
472 }

```

Listing 13-16: mm/oom_kill.c: dump_header()

The `dump_header()` function outputs the familiar OOM killer header, indicating the current state of free memory via `show_mem()` alongside slab information and if the `vm.oom_dump_tasks` tunable is set, output a list of tasks alongside their memory usage and `/proc/$pid/oom_score_adj` values.

See Section 14.4 in Chapter 14 for an examination of an example output in detail to see how this looks in practice.

Returning to `oom_kill_process()` and Listing 13-15, we see that we defer the heavy lifting of marking the process to be killed to `__oom_kill_process()`, which we examine starting in Listing 13-17 (eliding out of scope cgroup and event logic).

```

917 static void __oom_kill_process(struct task_struct *victim, const char *message
918     )
919 {
920     struct task_struct *p;
921     struct mm_struct *mm;
922     bool can_oom_reap = true;
923
924     p = find_lock_task_mm(victim);
925     if (!p) {
926         pr_info("%s: OOM victim %d (%s) is already exiting. Skip
927             killing the task\n",
928                 message, task_pid_nr(victim), victim->comm);
929         put_task_struct(victim);
930         return;
931     } else if (victim != p) {
932         get_task_struct(p);
933         put_task_struct(victim);
934         victim = p;
935     }
936
937     /* Get a reference to safely compare mm after task_unlock(victim) */
938     mm = victim->mm;
939     mmgrab(mm);

```

```

:
943      /*
944       * We should send SIGKILL before granting access to memory reserves
945       * in order to prevent the OOM victim from depleting the memory
946       * reserves from the user space under its control.
947       */
948      do_send_sig_info(SIGKILL, SEND_SIG_PRIV, victim, PIDTYPE_TGID);
949      mark_oom_victim(victim);
950      pr_err("%s: Killed process %d (%s) total-vm:%lukB, anon-rss:%lukB,
file-rss:%lukB, shmem-rss:%lukB, UID:%u pgtables:%lukB oom_score_adj:%hd\
n",
951            message, task_pid_nr(victim), victim->comm, K(mm->total_vm),
952            K(get_mm_counter(mm, MM_ANONPAGES)),
953            K(get_mm_counter(mm, MM_FILEPAGES)),
954            K(get_mm_counter(mm, MM_SHMEMPAGES)),
955            from_kuid(&init_user_ns, task_uid(victim)),
956            mm_pgtables_bytes(mm) >> 10, victim->signal->oom_score_adj);
957      task_unlock(victim);

```

Listing 13-17: mm/oom_kill.c: `__oom_kill_process()` Header

In `__oom_kill_process()` we start by invoking `find_lock_task_mm()` (as shown in Listing 13-14) to find and lock a task that belongs to the same process as the target victim task and also references a valid `mm_struct` object describing the process's virtual address space.

If we are unable to do so, we note that the process is already exiting, reduce the reference count on exit, otherwise, if the task we are obtained differs from the target victim one, we increment its reference count and decrement the victim's and swap them.

We then use `mmapgrab()` to increment the `struct mm_struct->mm_count` associated with the virtual address space, which protects the object from being freed even if the owning task exits.

Prior to marking the process as being a victim, we send a `SIGKILL` signal to unconditionally start the exit of the process and to activate all the various checks in the memory management code around a pending fatal signal to prevent depletion of memory reserves which are made available once the victim status is marked.

The process is marked as a victim via `mark_oom_victim()` which we examine in Listing 13-18. After doing so we output information about this, before releasing the lock on the task.

The actual killing of the process is performed by the OOM reaper, which we discuss in Section 13.4.6 (eliding out of scope task freezing and tracing logic).

```

747  /**
748   * mark_oom_victim - mark the given task as OOM victim
749   * @tsk: task to mark
750   *
751   * Has to be called with oom_lock held and never after

```

```

752  * oom has been disabled already.
753  *
754  * tsk->mm has to be non NULL and caller has to guarantee it is stable (either
755  * under task_lock or operate on the current).
756  */
757  static void mark_oom_victim(struct task_struct *tsk)
758  {
759      struct mm_struct *mm = tsk->mm;
760
761      WARN_ON(oom_killer_disabled);
762      :
766      /* oom_mm is bound to the signal struct life time. */
767      if (!cmpxchg(&tsk->signal->oom_mm, NULL, mm)) {
768          mmgrab(tsk->signal->oom_mm);
769          set_bit(MMF_OOM_VICTIM, &mm->flags);
770      }
771      :
779      atomic_inc(&oom_victims);
780      :
781  }

```

Listing 13-18: mm/oom_kill.c: mark_oom_victim()

The `mark_oom_victim()` function references the `struct task_struct->signal's` `struct signal_struct->oom_mm` field which is of `struct mm_struct` type and keeps track of the virtual address space subject to OOM reaping.

We atomically compare and exchange this field against that of the victim task, and if there is none already in place (which would indicate that the process is already subject to OOM killing) then we invoke `mmgrab()` to increment the virtual address space's reference count and mark the address space as being a victim using the `MMF_OOM_VICTIM` flag, before incrementing the atomic count of current OOM victims, `oom_victims`.

With the victim marked, we now proceed with ensuring we cover all tasks which share the `struct mm_struct` but do not exist in the same thread group, before performing final cleanup tasks in Listing 13-19.

```

917      /*
918      * Kill all user processes sharing victim->mm in other thread groups,
919      if
920      * any. They don't get access to memory reserves, though, to avoid
921      * depletion of all memory. This prevents mm->mmap_lock livelock when
922      an
923      * oom killed thread cannot exit because it requires the semaphore and
924      * its contended by another thread trying to allocate memory itself.
925      * That thread will now get access to memory reserves since it has a
926      * pending fatal signal.
927      */
928      rcu_read_lock();
929      for_each_process(p) {

```



```

928         if (!process_shares_mm(p, mm))
929             continue;
930         if (same_thread_group(p, victim))
931             continue;
932         if (is_global_init(p)) {
933             can_oom_reap = false;
934             set_bit(MMF_OOM_SKIP, &mm->flags);
935             pr_info("oom killer %d (%s) has mm pinned by %d (%s)\n",
936                    task_pid_nr(victim), victim->comm,
937                    task_pid_nr(p), p->comm);
938             continue;
939         }
940         /*
941          * No kthread_use_mm() user needs to read from the userspace
942          so
943          * we are ok to reap it.
944          */
945         if (unlikely(p->flags & PF_KTHREAD))
946             continue;
947         do_send_sig_info(SIGKILL, SEND_SIG_PRIV, p, PIDTYPE_TGID);
948         rcu_read_unlock();
949
950         if (can_oom_reap)
951             queue_oom_reaper(victim);
952
953         mmdrop(mm);
954         put_task_struct(victim);
955     }
956 #undef K

```

Listing 13-19: mm/oom_kill.c: __oom_kill_process() Footer

In the latter part of `__oom_kill_process()` shown in Listing 13-19 we both handle the edge case of tasks which share the struct `mm_struct` but do not exist in the same thread group, which can occur when `clone()` is used with `CLONE_VM` set but without `CLONE_SIGHAND`, or if kernel threads “adopt” the virtual address space via `kthread_use_mm()`.

We iterate through each process via `for_each_process()`, ignoring ones which do not share the struct `mm_struct`, checked via `process_shares_mm()`, or those which share the same signal handler, checked via `.`

We then consider the unusual case of the global initialisation process pinning the struct `mm_struct`, a process which is unkillable, checked via `is_global_init()`—if so we disallow the reaping and mark the struct `mm_struct` as one to be skipped in further evaluation via the `MMF_OOM_SKIP` flag.

If the task is a kernel thread, we skip forwards, as we are safe to reap the memory, as any attempt to read memory from a reaped process's virtual address space will be handled appropriately.

Finally, ordinary instances of non-thread group shared struct `mm_struct` are set the `SIGKILL` signal. This is done to prevent this tasks from causing issues by allocating memory when we are trying to reap the same virtual address space.

After considering these edge case tasks, we then proceed with queueing the process for reaping via `queue_oom_reaper()` which we examine in Listing 13-21.

We then drop the struct `mm_struct` and struct `task_struct` reference counts we had elevated in order to pin these objects in memory throughout the operation.

13.4.6 OOM Reaper

The actual killing of processes is performed by the OOM “reaper” which runs in a dedicated kernel thread represented by the struct `task_struct` in the static variable `oom_reaper_th`.

The reaper uses the `oom_reaper_wait` wait queue in order to trigger the thread to awaken and perform reaping, storing the head of the list of tasks to reap in `oom_reaper_list` (a linked list of tasks to reap is maintained using the struct `task_struct->oom_reaper_list` field to tie the tasks together) and operating on these fields (and reaping in general) is protected by the `oom_reaper_lock` spinlock.

NOTE

A wait queue is a list of tasks waiting for an event to occur and is a key synchronisation mechanism within the kernel. Waiting tasks suspend their operation until the specified condition resolves to true. A deeper discussion of this primitive is out of scope for the book.

The OOM reaper is initialised in `oom_init()`, which we examine in Listing 13-20.

```

732 static int __init oom_init(void)
733 {
734     oom_reaper_th = kthread_run(oom_reaper, NULL, "oom_reaper");
735 #ifdef CONFIG_SYSCTL
736     register_sysctl_init("vm", vm_oom_kill_table);
737 #endif
738     return 0;
739 }
```

Listing 13-20: `mm/oom_kill.c: oom_init()`

This starts the `oom_reaper` kernel thread which runs `oom_reaper()` (see Listing 13-24) whose struct `task_struct` is stored in `oom_reaper_th`, as well as registering `sysctl` handlers.

The kernel thread runs the `oom_reaper()` function, which we examine in Listing 13-24. Before examining this, we examine `queue_oom_reaper()` in Listing 13-21, the function which queues a task for reaping.

```

682 /*
683  * Give the OOM victim time to exit naturally before invoking the oom_reaping.
684  * The timers timeout is arbitrary... the longer it is, the longer the worst
685  * case scenario for the OOM can take. If it is too small, the oom_reaper can
686  * get in the way and release resources needed by the process exit path.
687  * e.g. The futex robust list can sit in Anon/Private memory that gets reaped
688  * before the exit path is able to wake the futex waiters.
689  */
690 #define OOM_REAPER_DELAY (2*HZ)
691 static void queue_oom_reaper(struct task_struct *tsk)
692 {
693     /* mm is already queued? */
694     if (test_and_set_bit(MMF_OOM_REAP_QUEUED, &tsk->signal->oom_mm->flags)
695         )
696         return;
697     get_task_struct(tsk);
698     timer_setup(&tsk->oom_reaper_timer, wake_oom_reaper, 0);
699     tsk->oom_reaper_timer.expires = jiffies + OOM_REAPER_DELAY;
700     add_timer(&tsk->oom_reaper_timer);
701 }

```

Listing 13-21: `mm/oom_kill.c: queue_oom_reaper()`

Importantly, before being queued for reaping, the struct `mm_struct` describing the process's virtual address space will have been placed into `struct task_struct->signal->oom_mm` by `mark_oom_victim()`.

The struct `signal_struct` field `struct task_struct->signal` represents the shared signal handler for tasks in a thread group (i.e. threads in a process) and thus represents the thread group or process as a whole, thus this is why we store the struct `mm_struct` to be reaped here.

The `MMF_OOM_REAP_QUEUED` struct `mm_struct` flag marks the process as queued for OOM reaping, so `queue_oom_reaper()` checks whether this is already set, exiting if so, otherwise setting it.

Note that by the point we are queueing a process for reaping, we will already have sent a `SIGKILL` signal to the process in `__oom_kill_process()` (as shown in Listing 13-17). We therefore delay the start of the reaping in hopes that exiting the process this way does the reaping for us.

This therefore renders the reaping a last gasp effort to hollow out unresponsive processes freeing as much memory as possible. The delay before waking the reaper is specified in `OOM_REAPER_DELAY` and is hardcoded to 2 seconds (as the comment indicates this is a heuristically determined value).

NOTE

The reference count of both the struct `task_struct` and the struct `mm_struct` objects are carefully maintained—the task's being incremented by `oom_evaluate_task()` (List-

ing 13-11) upon selection and `queue_oom_reaper()` (Listing 13-21) upon queuing and decremented when each is done with it. The `struct mm_struct`'s reference count is incremented in `__oom_kill_process()` and `mark_oom_victim()` and released when used.

When the timer expires, the `wake_oom_reaper()` function is invoked to cause the reaper thread to wake up. We examine this in Listing 13-22 (eliding out of scope tracing hooks).

```

661 static void wake_oom_reaper(struct timer_list *timer)
662 {
663     struct task_struct *tsk = container_of(timer, struct task_struct,
664                                             oom_reaper_timer);
665     struct mm_struct *mm = tsk->signal->oom_mm;
666     unsigned long flags;
667
668     /* The victim managed to terminate on its own - see exit_mmap */
669     if (test_bit(MMF_OOM_SKIP, &mm->flags)) {
670         put_task_struct(tsk);
671         return;
672     }
673
674     spin_lock_irqsave(&oom_reaper_lock, flags);
675     tsk->oom_reaper_list = oom_reaper_list;
676     oom_reaper_list = tsk;
677     spin_unlock_irqrestore(&oom_reaper_lock, flags);
678     :
679     wake_up(&oom_reaper_wait);
680 }
```

Listing 13-22: `mm/oom_kill.c: wake_oom_reaper()`

The `struct mm_struct` describing the victim process's virtual address space is checked to see if it has the `MMF_OOM_SKIP` flag set. If so the reference count for the task is dropped and we exit.

This flag is set when a process should not be considered for OOM killing and set when we know for sure this is the case. One important case of setting this flag is in `exit_mmap()`, invoked when releasing all memory mappings in the process. We examine the pertinent part in Listing 13-23.

```

3074 /* Release all mmaps. */
3075 void exit_mmap(struct mm_struct *mm)
3076 {
3077     :
3084     if (unlikely(mm_is_oom_victim(mm))) {
3085         /*
3086          * Manually reap the mm to free as much memory as possible.
3087          * Then, as the oom reaper does, set MMF_OOM_SKIP to disregard
```

```

3088         * this mm from further consideration. Taking mm->mmap_lock
        for
3089         * write after setting MMF_OOM_SKIP will guarantee that the
        oom
3090         * reaper will not run on this mm again after mmap_lock is
3091         * dropped.
3092         *
3093         * Nothing can be holding mm->mmap_lock here and the above
        call
3094         * to mmu_notifier_release(mm) ensures mmu notifier callbacks
        in
3095         * __oom_reap_task_mm() will not block.
3096         */
3097         (void)__oom_reap_task_mm(mm);
3098         set_bit(MMF_OOM_SKIP, &mm->flags);
3099     }
3100     :
3130 }

```

Listing 13-23: mm/mmap.c: `exit_mmap()` OOM killer logic

This means that we definitely know that the `MMF_OOM_SKIP` flag will be set for an exiting process, and thus our delay in executing the reaper will have given the process, having received a `SIGKILL` signal, a chance to exit and thus for us to skip it here. It also performs a manual reaping via `__oom_reap_task_mm()` (see Listing 13-27) to ensure we reap as much memory as possible.

Returning to `wake_oom_reaper()` in Listing 13-22, once we have checked for `MMF_OOM_SKIP`, we then acquire the `oom_reaper_lock` before placing this task at the top of the `oom_reaper_list` and waking up the reaper thread using the `oom_reaper_wait` wait queue.

This causes `oom_reaper()` to be woken up in the oom reaper kernel thread (listed as `[oom_reaper]` to userland), which we examine in Listing 13-24.

```

639 static int oom_reaper(void *unused)
640 {
641     set_freezable();
642
643     while (true) {
644         struct task_struct *tsk = NULL;
645
646         wait_event_freezable(oom_reaper_wait, oom_reaper_list != NULL)
        ;
647         spin_lock_irq(&oom_reaper_lock);
648         if (oom_reaper_list != NULL) {
649             tsk = oom_reaper_list;
650             oom_reaper_list = tsk->oom_reaper_list;
651         }
652         spin_unlock_irq(&oom_reaper_lock);

```

```

653
654         if (tsk)
655             oom_reap_task(tsk);
656     }
657
658     return 0;
659 }

```

Listing 13-24: mm/oom_kill.c: oom_reaper()

This starts by marking the process as “freezable” via `set_freezable()`, indicating that it can be frozen during suspend power events. Discussion of this is out of scope for the book.

Afterwards, we enter an infinite loop, where we wait on the `oom_reaper_wait` wait queue, proceeding only if it is both notified and the condition that `oom_reaper_list` is not empty is fulfilled.

If so, we acquire the `oom_reaper_lock` which protects `oom_reaper_list`, then pop the top of the list, checking to ensure that the list is indeed non-empty.

We then defer the actual reaping to `oom_reap_task()` which we examine in Listing 13-25. After this is done we loop around and check the list again.

```

608 static void oom_reap_task(struct task_struct *tsk)
609 {
610     int attempts = 0;
611     struct mm_struct *mm = tsk->signal->oom_mm;
612
613     /* Retry the mmap_read_trylock(mm) a few times */
614     while (attempts++ < MAX_OOM_REAP_RETRIES && !oom_reap_task_mm(tsk, mm)
615         )
616
617         schedule_timeout_idle(HZ/10);
618
619         if (attempts <= MAX_OOM_REAP_RETRIES ||
620             test_bit(MMF_OOM_SKIP, &mm->flags))
621             goto done;
622
623     pr_info("oom_reaper: unable to reap pid:%d (%s)\n",
624             task_pid_nr(tsk), tsk->comm);
625     sched_show_task(tsk);
626     debug_show_all_locks();
627
628 done:
629     tsk->oom_reaper_list = NULL;
630
631     /*
632     * Hide this mm from OOM killer because it has been either reaped or
633     * somebody can't call mmap_write_unlock(mm).
634     */
635     set_bit(MMF_OOM_SKIP, &mm->flags);
636
637 }

```

```

635      /* Drop a reference taken by queue_oom_reaper */
636      put_task_struct(tsk);
637  }

```

Listing 13-25: mm/oom_kill.c: oom_reap_task()

We attempt to reap up to MAX_OOM_REAP_RETRIES times before giving up, invoking oom_reap_task_mm() (see Listing 13-26) to do so. Whether we succeed or not, we set the MMF_OOM_SKIP flag in the process's struct mm_struct.

As previously noted, this is applicable to processes which did not exit within OOM_REAPER_DELAY i.e. 2 seconds, so this will typically be those which are in an unresponsive state.

Each time we attempt to reap a process we suspend the process for 1/10th of a second. The reason that the reaper might fail is typically due to not being able to acquire a read lock on struct mm_struct->mmap_lock (it attempts to acquire this optimistically but does not wait on it), so waiting may allow any active writers to complete their work.

This ensures that reaping doesn't become stalled (we constrain the amount time we wait) and we do not contend on this lock, preventing a process which might be in the process of freeing memory from proceeding.

if we succeed in reaping the process or the process has become marked with MMF_OOM_SKIP, then we are done. Otherwise we output debug data to the kernel ring buffer.

In either case we clear the next task in the reaper list (i.e. struct task_struct->oom_reaper_list), which we will have already put at the head of oom_reaper_list, and mark the process MMF_OOM_SKIP.

We examine oom_reap_task_mm() in Listing 13-26, eliding tracing hooks.

```

561  /*
562   * Reaps the address space of the give task.
563   *
564   * Returns true on success and false if none or part of the address space
565   * has been reclaimed and the caller should retry later.
566   */
567  static bool oom_reap_task_mm(struct task_struct *tsk, struct mm_struct *mm)
568  {
569      bool ret = true;
570
571      if (!mmap_read_trylock(mm)) {
572          :
573          return false;
574      }
575
576      /*
577       * MMF_OOM_SKIP is set by exit_mmap when the OOM reaper can't
578       * work on the mm anymore. The check for MMF_OOM_SKIP must run
579       * under mmap_lock for reading because it serializes against the
580       * mmap_write_lock();mmap_write_unlock() cycle in exit_mmap().
581       */

```

```

582         if (test_bit(MMF_OOM_SKIP, &mm->flags)) {
583             :
574             goto out_unlock;
575         }
586         :
589         /* failed to reap part of the address space. Try again later */
590         ret = __oom_reap_task_mm(mm);
591         if (!ret)
592             goto out_finish;
593
594         pr_info("oom_reaper: reaped process %d (%s), now anon-rss:%lukB, file-
595             rss:%lukB, shmem-rss:%lukB\n",
596                 task_pid_nr(tsk), tsk->comm,
597                 K(get_mm_counter(mm, MM_ANONPAGES)),
598                 K(get_mm_counter(mm, MM_FILEPAGES)),
599                 K(get_mm_counter(mm, MM_SHMEMPAGES)));
600 out_finish:
601         :
602 out_unlock:
603         mmap_read_unlock(mm);
604
605         return ret;
606     }

```

Listing 13-26: mm/oom_kill.c: `oom_reap_task_mm()`

The `oom_reap_task_mm()` function is a wrapper around `__oom_reap_task_mm()` (see Listing 13-27), optimistically attempting to acquire a read lock on the struct `mm_struct->mmap_lock` semaphore via `mmap_read_trylock()`. If this fails, it exits indicating failure.

Once we have acquired this lock, we repeat the check for `MMF_OOM_SKIP`, as this may be set by `exit_mmap()` when the process is exiting, (as shown in Listing 13-23).

We repeat this check here after we have acquired the lock, as it will have acquired a write lock on the struct `mm_struct->mmap_lock` semaphore, meaning that our read lock guarantees we are serialised against this.

This ensures that we do not waste time trying to reap the process when it has already been stripped of memory.

If it should not be skipped, then we invoke `__oom_reap_task_mm()`. If this indicates success, we output details of the reaping, in either case we relinquish the read lock on struct `mm_struct->mmap_lock`.

We examine `__oom_reap_task_mm()` in Listing 13-27, eliding out of scope MMU notifier logic.

```

512 bool __oom_reap_task_mm(struct mm_struct *mm)
513 {
514     struct vm_area_struct *vma;
515     bool ret = true;
516

```



```

517      /*
518       * Tell all users of get_user/copy_from_user etc... that the content
519       * is no longer stable. No barriers really needed because unmapping
520       * should imply barriers already and the reader would hit a page fault
521       * if it stumbled over a reaped memory.
522       */
523      set_bit(MMF_UNSTABLE, &mm->flags);
524
525      for (vma = mm->mmap ; vma; vma = vma->vm_next) {
526          if (vma->vm_flags & (VM_HUGETLB|VM_PFNMAP))
527              continue;
528
529          /*
530           * Only anonymous pages have a good chance to be dropped
531           * without additional steps which we cannot afford as we
532           * are OOM already.
533           *
534           * We do not even care about fs backed pages because all
535           * which are reclaimable have already been reclaimed and
536           * we do not want to block exit_mmap by keeping mm ref
537           * count elevated without a good reason.
538           */
539          if (vma_is_anonymous(vma) || !(vma->vm_flags & VM_SHARED)) {
540              :
541              struct mmu_gather tlb;
542              :
543              tlb_gather_mmu(&tlb, mm);
544              :
545              unmap_page_range(&tlb, vma, range.start, range.end,
546                  NULL);
547              :
548              tlb_finish_mmu(&tlb);
549              }
550          }
551
552      return ret;
553 }

```

Listing 13-27: mm/oom_kill.c: __oom_reap_task_mm()

The function starts by setting the MMF_UNSTABLE flag in the struct mm_struct describing the virtual address space of the process.

This flag is checked only by the helper function `check_stable_address_space()` which indicates that the SIGBUS signal should be sent to the process, indicated by the fault code VM_FAULT_SIGBUS (see Chapter 6 for more details on the page fault mechanism).

This is used in `do_anonymous_page()` and `finish_fault()` during the faulting process, which will then indicate the SIGBUS rather than resolve the fault.

With this flag set, we iterate through each of the process address space's `struct vm_area_struct` (VMA) objects (see Section 4.4 for details on VMAs), skipping those which describe hugetlb huge page mappings (as indicated by the VMA flag `VM_HUGETLB`) or those which map memory untracked by the kernel (e.g. directly exposing hardware memory mapping or kernel-mapped memory), as indicated by `VM_PFNMAP`. We skip these as we are in no position to free hugetlb memory in the first instance, and cannot free the memory in the latter instance.

After this we check to see whether the memory is either mapped anonymous, checked by `vma_is_anonymous()` or does not describe shared memory (e.g. a `MAP_PRIVATE` mapping) as indicated by the `VM_SHARED` flag.

As the comment indicates, we consider only these cases as these are the only ones we can be confident in freeing immediately. File-backed memory always has an additional reference from the page cache and thus must be processed through reclaim in order to ensure that any dirty data is written back to disk so this is not as conducive to freeing memory quickly.

For each VMA we choose to free, we establish an `struct mmu_gather` object in order to aggregate the ranges of memory which we intend to free for the purposes of invalidating the Translation Lookaside Buffer (TLB) which is a cache mapping virtual address to physical ones.

See Section 7.1 for a detailed explanation of the process of invalidating the TLB, unmapping and freeing memory.

We set up the `struct mmu_gather` using `tlb_gather_mmu()` (see Listing 7-54), unmap the page range, updating the TLB state in this object via `unmap_page_range()` (see Listing 7-60), before finalising the operation in `tlb_finish_mmu()` (see Listing 7-79).

When complete, we will have stripped all the low-hanging fruit and the process will now be considered reaped. As a `SIGKILL` will have been issued by this point this is the state it will remain in should it be entirely unresponsive, otherwise it will eventually be killed.

14

PRACTICAL MEMORY MANAGEMENT

We've explored the detailed internals of numerous features of the kernel memory manager, however under ordinary circumstances these are details that are transparent to users.

What users interact with are the *interfaces* exposed by the kernel. These consist of:-

Userland libc memory allocation functions – `mmap()` (a wrapper around a syscall) (alongside the related `munmap()` and `msync()`), `brk()` (alongside `sbrk()` a wrapper around a syscall) and `malloc()` (which maintains an internal userland cache of available memory, using the former two functions to obtain memory from the kernel in the first instance). *We have examined these in detail in chapters 5 (memory mapping) and 8 (memory manipulation).*

Reading `procfs` and `sysfs` interfaces – There are a number of means of interrogating memory in the kernel exposed in the `procfs` and `sysfs` file systems.

Tunables – There are a number of tunable flags available in both `procfs` and `sysfs` which allow a user to perform actions as well as to customise the memory manager's behaviour.

Crash reports and sysrq – When the Out Of Memory (OOM) killer frees up memory, it provides a report on currently available memory as well as details of the circumstances around the out of memory condition. This can also be obtained by writing `m` to `/proc/sysrq-trigger`.

Memory manipulation via syscalls – `madvise()` allows for ‘hints’ to be given to the kernel for a range of virtual memory. `mprotect()` permits the manipulation of access protections for a range of virtual memory (principally read/write/execute). Memory mapped via `mmap()` can be moved around using `mremap()`. Memory ranges can be locked to prevent them from being reclaimed via `mlock()` (and unlocked via `munlock()`). See chapter 8 on memory manipulation for more details.

Tracking memory usage – Interfaces such as that provided for tracking *soft-dirty* bits, *idle page tracking*, *soft-dirty* and the DAMON Data Access MONitor – allow for tracking of memory behaviour within processes.

Examining page table metadata – Page tables and flags can be examined via the *page map* interface via `/proc/$pid/pagemap`, `/proc/$pid/kpagecount` and `/proc/$pid/kpageflags` files.

Raw access to process memory – A process’s memory can be accessed directly via `/proc/$pid/mem` with sufficient permission.

Copying memory – Memory can be manually transferred between processes via `process_vm_readv()` and `process_vm_writev()`.

Sharing memory – Memory can be shared between processes by numerous means, for instance mapping anonymous memory via `mmap()` using the `MAP_SHARED` flag to share memory between a parent and forked child processes, children, or by multiple processes mapping a shared file, whether RAM-backed or not.

Sharing memory by file descriptor can be achieved by use of system V shared memory system calls, the POSIX `shm` interface, using `memfd_create()` to create a RAM-backed *anonymous* file (i.e. one that does not appear in the file system), or simply by mapping a file.

userfaultfd – `userfaultfd` is a means by which userland can handle page faults and manually decide how to back them.

`ptrace()` – The `ptrace` functionality permits one process to access the memory of another for the purposes of debugging (it does this utilising the Get User Pages or GUP functionality, see section 8.1.2 for more details on this).

`mincore()` – The `mincore` interface allows an application to determine which pages are resident in memory and which will result in a page fault.

Covering each of these in depth would constitute a book of its own, so instead we will examine a useful subset of these interfaces in detail.

14.1 Measuring free memory

When trying to determine how much free memory is available in the system, a user will almost certainly reach for the `free` utility. For instance:-

```
[~]$ free
```

	total	used	free	shared	buff/cache
Mem:	65779944	3973116	59395864	232428	3344276
	61806828				
Swap:	10485756	0	10485756		

Listing 14-1: free example

By default the output is expressed in KiB, as a result it's often useful to pass the `-h` flag to obtain 'human readable' output:-

```
[~]$ free -h
```

	total	used	free	shared	buff/cache
Mem:	62Gi	3.8Gi	56Gi	212Mi	3.2Gi
	Gi				58
Swap:	9Gi	0B	9Gi		

Listing 14-2: free -h example

There are a number of other options that can be passed to this command, but typically this is sufficient for most users.

In order to determine precisely where these numbers come from, we should examine where `free` obtains its data from in the first place – `/proc/meminfo` (eliding fields not relevant to the output of `free`):-

```
[~]$ cat /proc/meminfo
```

```
MemTotal:      65779944 kB
MemFree:       59849892 kB
MemAvailable:  62306792 kB
Buffers:       395824 kB
Cached:        2767732 kB
SwapCached:    0 kB
...
SwapTotal:     10485756 kB
SwapFree:      10485756 kB
...
Shmem:         209176 kB
Slab:          338064 kB
SReclaimable:  207356 kB
SUnreclaim:    130708 kB
...
```

Listing 14-3: Example output of `/proc/meminfo`

It is this interface that `free` retrieves its data from, mapped as follows:-

- `total` = `MemTotal` – Total *usable* physical memory in the system. This may be less than installed physical memory if running in a virtual machine, or if some memory is otherwise unavailable.
- `used` = `MemTotal` - `MemAvailable` – Perhaps surprisingly, this is an estimate of currently in-use memory which cannot be easily dropped on memory pressure, equal to total physical memory available to the system, less heuristically-determined *available* memory (see below for details).
- `free` = `MemFree` – All memory that is not currently in use for any purpose.
- `shared` = `Shmem` – The total amount of `shmem` memory, i.e. memory used for RAM-backed file systems, most notably `tmpfs`.
- `buff/cache` = `Buffers` + `Cached` + `SReclaimable` – The amount of memory utilised by kernel buffers and the page cache (see the page cache chapter for more details on both) or *reclaimable* kernel slab allocator pages (see the slab chapter for details on what this means specifically).
`Buffers` is equal to the total number of block device pages, `Cached` is equal to total file-backed pages minus swap cache (see the swap chapter for details of how swap cache functions) and block device pages.
 Therefore `buff/cache` sums to the total number of file-backed and slab-reclaimable pages minus swap cache pages.
- `available` = `MemAvailable` – The estimated amount of available memory without causing swapping, thrashing (a cycle of evicting and refaulting page cache entries which causes significant system stress) or triggering the Out Of Memory killer. See below for details on how this is calculated.

NOTE

See <https://gitlab.com/procps-ng/procps/-/blob/master/src/free.c> and <https://gitlab.com/procps-ng/procps/-/blob/master/library/meminfo.c> for *free* implementation details (as of the time of writing).

The most important metric for determining available memory is, naturally, `MemAvailable`. In most cases, a user determines how much free memory is available in the system by looking at the `available` value in `free`, so it's worth investigating the source of this value in detail.

The value is heuristic by nature—in order to avoid thrashing, we must retain a *certain proportion* of the page cache. In order to avoid reclaim being triggered risking the OOM killer, we must maintain pages above both the high water mark (see section 2.4 for more on nodes, zones, and watermarks) and the worst case low memory reserve (see sections 2.4.2 and 2.4.1 respectively for details of these).

The algorithm for determining available pages is determined empirically as follows:

- Determine the sum of the low water marks for each zone in the system. This will be used to determine the minimum amount of memory reserved for the page cache and reclaimable kernel (that is slab) memory. In each case, we cap this reserve at half of the size of reclaimable memory.
- Initialise available memory to total free memory minus `totalreserve_pages`. The `totalreserve_pages` value comprises the sum of all zone high water marks and worst-case low memory reserve (see section 2.4.2 for details on how this is calculated).
- Calculate the total size of the page cache (not including `shmem`), and add it to available memory on assumption it can be freed, less the sum of low water marks for each zone (capped at half of the non-`shmem` page cache size—that is, we reserve at maximum half of the non-`shmem` page cache).
- Calculate the sum of reclaimable slab memory (equal to `SReclaimable` in `/proc/meminfo`), and add it to available memory on assumption it can be freed, less the sum of low water marks for each zone (capped at half of reclaimable slab memory—that is, we reserve at maximum half of slab reclaimable memory).

NOTE

An added complication to page cache considerations is `shmem`—folios belonging to these reside in the page cache but for the purposes of LRU (see Chapter 11 for more on the LRU and reclaim) are treated as anonymous and thus are swapped out under memory pressure. As `shmem` components can't be easily dropped under memory pressure (they need to be swapped out), we do not consider them as part of this calculation.

So in summary – `MemAvailable` is equal to the sum of free and quickly reclaimable non-reserved memory.

14.1.1 Kernel implementation of `MemAvailable`

The function which generates the data for `/proc/meminfo` is `meminfo_proc_show()`. This in turn invokes `si_mem_available()` to obtain the `MemAvailable` value:-

```

5919 long si_mem_available(void)
5920 {
5921     long available;
5922     unsigned long pagecache;
5923     unsigned long wmark_low = 0;
5924     unsigned long pages[NR_LRU_LISTS];
5925     unsigned long reclaimable;
5926     struct zone *zone;
5927     int lru;
5928
```



```

5929         for (lru = LRU_BASE; lru < NR_LRU_LISTS; lru++)
5930             pages[lru] = global_node_page_state(NR_LRU_BASE + lru);
5931
5932         for_each_zone(zone)
5933             wmark_low += low_wmark_pages(zone);
5934
5935         /*
5936          * Estimate the amount of memory available for userspace allocations,
5937          * without causing swapping or OOM.
5938          */
5939         available = global_zone_page_state(NR_FREE_PAGES) - totalreserve_pages
;
5940
5941         /*
5942          * Not all the page cache can be freed, otherwise the system will
5943          * start swapping or thrashing. Assume at least half of the page
5944          * cache, or the low watermark worth of cache, needs to stay.
5945          */
5946         pagecache = pages[LRU_ACTIVE_FILE] + pages[LRU_INACTIVE_FILE];
5947         pagecache -= min(pagecache / 2, wmark_low);
5948         available += pagecache;
5949
5950         /*
5951          * Part of the reclaimable slab and other kernel memory consists of
5952          * items that are in use, and cannot be freed. Cap this estimate at
the
5953          * low watermark.
5954          */
5955         reclaimable = global_node_page_state_pages(NR_SLAB_RECLAIMABLE_B) +
5956             global_node_page_state(NR_KERNEL_MISC_RECLAIMABLE);
5957         available += reclaimable - min(reclaimable / 2, wmark_low);
5958
5959         if (available < 0)
5960             available = 0;
5961         return available;
5962     }

```

Listing 14-4: mm/page_alloc.c: *si_mem_available()*

This function utilises `global_node_page_state()` to obtain aggregate node statistical counters indicating the number of pages on each LRU list (see 11.2 for more details on LRUs), by examining the range of available LRU statistics offset from `LRU_BASE + NR_LRU_BASE` consisting of `NR_LRU_LISTS` elements.

The `global_node_page_state()` function is additionally used to obtain the number of slab-reclaimable pages via `NR_SLAB_RECLAIMABLE_B` and the number of miscellaneous yet reclaimable kernel buffer pages via `NR_KERNEL_MISC_RECLAIMABLE`.

The `global_zone_page_state()` function is used to obtain total free pages via `NR_FREE_PAGES`, minus `totalreserve_pages` (again, see section 2.4.2 for details on how this value is obtained).

14.1.2 Higher order folios

In most cases all a user need know is the total available memory. However, non-vmalloc kernel memory allocations such as those performed in drivers will often require *higher order* allocations. These may arise for instance from file system operations.

The concept of page orders arise from the nature of the underlying physical memory allocator within the kernel, the buddy allocator (see section 2.7 for a detailed description).

Folios of higher order are of size 2^{order} base pages, e.g. an order-5 page is of 2^5 or 32 pages, i.e. on x86-64, 128 KiB in size.

The buddy allocator aggregates physical memory in folios of up to order `MAX_ORDER`–1. When memory of lower orders are allocated and insufficient folios at that size exist, larger ones are split in half repeatedly until folios of the appropriate size are made available.

Order-0 pages are therefore the easiest to come by – either you have some available, or any other higher order folio can be split until you do.

For higher order folios, fragmentation might mean sufficient memory is available in the system but not physically contiguous memory, i.e. at the required order. In this instance, memory allocations will fail leading potentially to the Out Of Memory (OOM) killer being invoked.

In addition, NUMA systems further complicate things, as you may require memory on a specific node. (see section 2.4 for more on nodes and zones).

NOTE

It may also be the case that for instance a driver requires memory from a specific zone, e.g. `ZONE_DMA32`, however low memory reserve protects against exhaustion of this resource (see section 2.4.1), so it is unlikely to cause significant fragmentation issues.

It is therefore useful to get a sense of memory availability for each node, zone and folio order. This information is available in `/proc/buddyinfo`, for example:-

```
$ cat /proc/buddyinfo
Node 0, zone DMA      0      0      0      0      0      0      0      0      1      1
Node 0, zone DMA32    5      6      6      4      5      3      6      6      5      4
Node 0, zone Normal 3310  6163  9144  6336  2200  547   106   53    27   14
```

Listing 14-5: Example of output of `/proc/buddyinfo`

Each column relates to a folio order, so for instance this indicates that zones DMA, DMA32 and normal have order-10 folio counts of 2, 221 and 14,288 respectively.

This interface can be useful for determining if fragmentation has occurred or is likely to soon occur within a system – if there is a scarcity of higher order folios, then higher order folio fragmentation becomes significantly more likely.

14.1.3 Summary

Use free or MemAvailable from `/proc/meminfo` to determine available free memory.

If higher order folio fragmentation is a concern, check `/proc/buddyinfo` to determine the current availability of higher order folios.

14.2 Measuring the memory usage of a process

Often a user will want to assess the impact a particular *process* has on memory usage in the system.

The most direct means of gathering the details of a process's memory usage is via `/proc/$pid/status` or the more succinct `/proc/$pid/stat`.

NOTE

The `/proc/$pid/statm` interface is frankly rather obsolete and doesn't provide overly useful information.

Here we will focus on determining the how much *resident* memory a process is using via `/proc/$pid/status`, referred to as the Resident Set Size or RSS. This is the amount of memory that has been faulted in to this process specifically.

The first port of call is `/proc/$pid/status`. For instance, examining the relevant fields in some example output:-

```
$ cat /proc/$pid/status
...
VmHWM:    97384 kB
VmRSS:    97276 kB
RssAnon:           48080 kB
RssFile:           49196 kB
RssShmem:           0 kB
...
```

Listing 14-6: Example output of `/proc/$pid/status`

Examining each:-

- **VmHWM** – This indicates the *peak* resident memory usage of the process, i.e. the maximum amount of resident memory the process has utilised during its execution.

- **VmRSS** – This indicates the amount of resident memory *currently* in use, equal to the sum of **RssAnon**, **RssFile** and **RssShmem**.
- **RssAnon** – The amount of resident memory currently in use for *anonymous*, i.e. non-file backed mappings.
- **RssFile** – The amount of resident memory currently in use for file-backed mappings (excluding **shmem** memory).
- **RssShmem** – The amount of memory used for **shmem** files, i.e. ones stored in **tmpfs**, ‘anonymous’ shared memory, system V shared memory and the like (see section ?? for more details).

These value are derived from the memory management statistical counters maintained in the kernel to track memory events, the updating of which is throttled in order to avoid performance impact.

As a result, these values will not quite be accurate. They may be offset up to **TASK_RSS_EVENTS_THRESH** (64) pages from reality (i.e. 256 KiB on an x86-64 system) *per thread*. See section 4.6 for a detailed description of RSS counters.

It’s also worth noting that an edge case exists in **vm_insert_pages()**, which does not synchronise RSS statistics at all. This is used by the zero-copy TCP receive mechanism within the kernel so if you are utilising this then data retrieved this way will have its RSS statistics accounted only after a subsequent page fault.

If a user wants to determine *precise* resident memory usage of a process, the **/proc/smmaps** or far more convenient **/proc/smmaps_rollup** interface provides this information by explicitly walking a process’s VMAs and determining memory usage, e.g.:-

```
$ cat /proc/$pid/smmaps_rollup
564c5c06e000-7fffc2bf1000 ---p 00000000 00:00 0 [
rollup]
Rss:                101592 kB
Pss:                71426 kB
Pss_Dirty:          52380 kB
Pss_Anon:           52368 kB
Pss_File:           19046 kB
Pss_Shmem:           12 kB
Shared_Clean:        35892 kB
Shared_Dirty:         28 kB
Private_Clean:       13304 kB
Private_Dirty:       52368 kB
Referenced:          101592 kB
Anonymous:           52368 kB
LazyFree:             0 kB
AnonHugePages:       38912 kB
ShmemPmdMapped:       0 kB
FilePmdMapped:       2048 kB
Shared_Hugetlb:       0 kB
Private_Hugetlb:      0 kB
```

Swap:	0 kB
SwapPss:	0 kB
Locked:	0 kB

Listing 14-7: Example `/proc/$pid/smmaps_rollup` output

This walks the process address space's VMAs while holding `struct mm_struct->mmap_lock`, via `walk_page_vma()`, walking each individual page's page tables in order to reach the underlying `struct page`, from which page table flags are used to determine what pages are *actually* present in each virtual address range (in `smmaps_account()`).

14.2.1 Proportional Set Size

If multiple processes map a file, each one will have its Resident Set Size (RSS) updated upon faulting in the mapping. Therefore, if for instance 100 processes map a single file, summing the RSS of these processes will result in there seeming to be 100 times the amount of memory used than is used in actuality.

Therefore it is not quite right to say that memory indicates how much physical memory a process consumes, but rather how much physical memory it *maps*.

In listing 14-7 we can see entries pertaining to PSS or Proportional Set Size. This is a means of addressing this issue with overcounting RSS – rather than considering each process which maps a file (for instance, a shared library) to account for the *entire* size of that file, each page of resident memory is divided by the number times it is mapped (whether in the same process or by other processes), i.e. each `struct page`'s map count as determined by `page_mapcount()`.

See below in section 14.5.2 for further discussion of proportional set size in the context of a process's `/proc/$pid/smmaps` output.

14.2.2 Summary

Generally, it's sufficient to use `/proc/$pid/status` and examine the `Rss` and `VmHWM` fields to determine current and peak memory usage respectively.

However, if you require absolute precision, your process spawns a great many threads, or you want proportional values then use `/proc/$pid/smmaps` or `/proc/$pid/smmaps_rollup`.

14.3 Memory mapping using `mmap()`

The `mmap()` function is the most direct means of establishing memory mappings within a process. In most cases, its use is unnecessary, as `malloc()` (or abstractions which wrap it) will utilise `sbrk()` and `mmap()` to map memory from the kernel as necessary for you, with the added benefit of maintaining efficient sub-page size free lists.

However there are circumstances where it makes sense to do so, for instance when mapping files and taking advantage of the fact that you can write directly to and read directly from the page cache.

See section 5.0.2 for an in-depth analysis of how this logic is implemented within the kernel.

The `mmap()` command is comprised of six parameters –
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t off)`
 – and returns a pointer to the mapped region, or the `MAP_FAILED` constant value, setting `errno` to indicate why it failed.

The `addr` parameter specifies where we would prefer the mapping to be placed, however in typical usage this is usually set to `NULL`, indicating that we are happy for the kernel to decide (we will examine cases where we *do* specify this shortly).

The `length` parameter specifies the length of the mapping, which does not need to be page-aligned, but if it isn't it will implicitly be rounded up to a page boundary. `prot` and `flags` specify the protection bits (i.e. read-/write/execute) for the mapping and `flags` specify flags specifying how the mapping should be performed.

Finally, `fd` specifies a file descriptor of the file if mapping one, or `-1` if not and `off` specifies the offset into that file.

Again, see section 5.0.2 for a detailed examination of the command.

Note that we explore use of `mmap()` for shared memory applications below in section 14.7.

14.3.1 Mapping anonymous memory

Therefore, a typical invocation of `mmap()` is as follows:-

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/mman.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     const long page_size = sysconf(_SC_PAGESIZE);
9     char *ptr = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
10                     MAP_ANON | MAP_PRIVATE, -1, 0);
11
12     if (ptr == MAP_FAILED) {
13         perror("mmap");
14         return EXIT_FAILURE;
15     }
16
17     /* ...do things with the mapping... */
18
19     if (munmap(ptr, page_size)) {
```

```

20         perror("munmap");
21         return EXIT_FAILURE;
22     }
23
24     return EXIT_SUCCESS;
25 }

```

Listing 14-8: Example of a basic anonymous mmap() mapping

We specify `PROT_READ` and `PROT_WRITE` as we intend the mapping to be read-/write, `MAP_ANON` as shorthand for `MAP_ANONYMOUS` and `MAP_PRIVATE` as we have no intention of sharing memory.

We must always check whether `mmap()` returns `MAP_FAILED`, then eventually `unmap` the memory via `munmap()`.

14.3.2 Mapping a file

Mapping a file is rather more useful, and follows a similar pattern, only we open a file via `open()` and pass the associated file descriptor alongside an offset into the file:-

```

1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <sys/stat.h>
6  #include <unistd.h>
7
8  int main(void)
9  {
10     int fd;
11     char *ptr;
12     struct stat st;
13
14     fd = open("test.txt", O_RDWR);
15     if (fd == -1) {
16         perror("open");
17         return EXIT_FAILURE;
18     }
19
20     if (fstat(fd, &st)) {
21         perror("fstat");
22         return EXIT_FAILURE;
23     }
24
25     ptr = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE,
26               MAP_SHARED_VALIDATE, fd, 0);
27     if (ptr == MAP_FAILED) {

```

```

28         perror("mmap");
29         return EXIT_FAILURE;
30     }
31
32     close(fd);
33
34     /* ...do things with the mapping... */
35
36     if (msync(ptr, st.st_size, MS_SYNC)) {
37         perror("msync");
38         return EXIT_FAILURE;
39     }
40
41     if (munmap(ptr, st.st_size)) {
42         perror("munmap");
43         return EXIT_FAILURE;
44     }
45
46     return EXIT_SUCCESS;
47 }

```

Listing 14-9: Example of a basic file `mmap()` mapping

Here we map the file at offset 0, indicating that we are mapping it shared (we use `MAP_SHARED_VALIDATE` which is identical to `MAP_SHARED` but disallows invalid map flag combinations).

Note that we do not need to keep the file open any longer after we have mapped it, the kernel will have looked up the `struct file` object and the page cache `struct address_space` object and will have incremented the reference count on this object, obviating the need for the file to be kept open.

We synchronise our changes to disk via `msync()`, specifying the `MS_SYNC` flag. This is the only flag that actually does any work (other than checks), as `MS_ASYNC` is handled implicitly by writeback (see section ??) and `MS_INVALIDATE` only checks whether the mapping is `mlock()`'d or not.

This synchronously writes our changes to the file to disk. This is not done implicitly by `munmap()`, as again the kernel will perform writeback in due course anyway so it is not necessarily.

Importantly – like anything mapped by `mmap()`, the length is rounded up to the page boundary. Therefore it becomes possible to write, in the example code shown above, to entries past the end of the file size but within the same page containing valid data.

In this instance, those writes will do absolutely nothing. When mapping files via `mmap()`, that mapping will remain static regardless of whether the size of the file changes.

However, what if we map a range larger than the size of the file or it is truncated to a size less than the page in which we write? In this instance, a `SIGBUS` error will arise, which is very important to account for when memory-mapping files.

It is also important to note that if an error occurs while accessing the file (for instance, a network drive mount becomes disconnected), a SIGBUS will occur in this instance also.

Note that a file being removed, or more accurately `unlink()`'d has no impact – this simply means the file is no longer hard linked from the opened location, but still exists as an inode on disk (until we are finished with it) and thus an entry in the page cache.

14.3.3 Private file mapping

There are instances where the semantics required are that a file is mapped, but the user requires writes to that file to not persist to it, but to cause the modified file data to be copied to anonymous memory.

This can be achieved by setting the `MAP_PRIVATE` flag in `mmap()` when mapping the file. This means of mapping a file this way is quite efficient – it is achieved by establishing a copy-on-write read-only mapping to the page cache (see section 5.0.2 for a detailed analysis of this works under the covers).

However, it is important to remain mindful of the (perhaps surprising) semantics this implies – if mapped pages in the file are truncated (which can result from a file being overwritten or shrunk for instance), any data written by the user will be lost and the buffer will revert to the newly truncated file (see section 6.6 for more details).

Examining an example:-

```

1 #include <fcntl.h>
2 #include <stdbool.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/mman.h>
7 #include <sys/stat.h>
8 #include <unistd.h>
9
10 int main(void)
11 {
12     int fd;
13     char *ptr;
14     struct stat st;
15
16     fd = open("test.txt", O_RDWR);
17     if (fd == -1) {
18         perror("open");
19         return EXIT_FAILURE;
20     }
21
22     if (fstat(fd, &st)) {
```

```

23         perror("fstat");
24         return EXIT_FAILURE;
25     }
26
27     ptr = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE,
28               MAP_PRIVATE, fd, 0);
29     if (ptr == MAP_FAILED) {
30         perror("mmap");
31         return EXIT_FAILURE;
32     }
33
34     close(fd);
35
36     while (true) {
37         char chr;
38
39         if (strncmp(ptr, "exit", 4) == 0)
40             break;
41
42         chr = ptr[0];
43         if (chr < 'a' || chr == 'z') {
44             ptr[0] = 'a';
45         } else {
46             ptr[0]++;
47         }
48
49         printf("%s", ptr);
50         sleep(1);
51     }
52
53     if (munmap(ptr, st.st_size)) {
54         perror("munmap");
55         return EXIT_FAILURE;
56     }
57
58     return EXIT_SUCCESS;
59 }

```

Listing 14-10: Example of a MAP_PRIVATE file mmap() mapping

This program loops updating the contents of the first character of the file which causes it to be copied to an anonymous page and remain there, outputting the contents of the buffer as it goes.

However, if another process were to write to the file, you could observe the data that has been written so far is *immediately* lost as the mapping is ‘re-set’ to match that of the file.

14.3.4 Fixed mappings

It can be useful to establish large, empty virtual ranges, within which we map at *known* virtual addresses, for instance when implementing a custom allocator.

We can achieve this by specifying the `addr` parameter to `mmap()`, alongside `MAP_FIXED`. Without `MAP_FIXED`, the `addr` parameter is merely advisory, and `mmap()` is free to ignore it.

It is important, however, to have established a mapping at the address you are mapping to the first instance. This is confusing, as it seems odd to attempt to overwrite an existing mapping – intuitive would suggest this would result in an error, however this is not the case – `mmap()` with the `MAP_FIXED` flag set will simply overwrite any mapping it spans, replacing it.

If you wish to insist on a specified address but do not want this overwriting behaviour to occur, you can specify `MAP_FIXED_NO_REPLACE` (in which case an existing, overlapping mapping will cause an error to be raised).

However, if you wish is not so much to have *specific* fixed addresses, but rather to have *known* ones, such as the example of a custom allocator, then you can work around this by first establishing an empty mapping using `PROT_NONE`, then established fixed mappings within this space.

A `PROT_NONE` mapping can be as large as you like, it is not accounted for in more restrictive overcommit modes (see section 4.1 for more on overcommit) and will cause page faults if accessed, so this forms the perfect backdrop for such an implementation.

Considering an example:-

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/mman.h>
4 #include <unistd.h>
5
6 #define MAX_ALLOCS (100)
7 #define ALLOC_SIZE (1 << 20) /* 1 MiB */
8
9 int main(void)
10 {
11     int i;
12     void *span;
13     const unsigned long page_size = (unsigned long)sysconf(_SC_PAGESIZE);
14     /* 4 TiB should be sufficient. */
15     const unsigned long size = page_size << 30;
16
17     span = mmap(NULL, size, PROT_NONE, MAP_ANON | MAP_PRIVATE, -1, 0);
18     if (span == MAP_FAILED) {
19         perror("mmap");
20         return EXIT_FAILURE;
21     }

```

```

22
23     for (i = 0; i < MAX_ALLOCS; i++) {
24         void *ptr = mmap(span + i * ALLOC_SIZE, ALLOC_SIZE,
25                         PROT_READ | PROT_WRITE,
26                         MAP_FIXED | MAP_ANON | MAP_PRIVATE, -1, 0);
27
28         if (ptr == MAP_FAILED) {
29             perror("internal mmap");
30             return EXIT_FAILURE;
31         }
32     }
33
34     /* Do something with mappings... */
35
36     if (munmap(span, size)) {
37         perror("munmap");
38         return EXIT_FAILURE;
39     }
40
41     return EXIT_SUCCESS;
42 }

```

Listing 14-11: Example of a fixed `mmap()` mapping

This maps a giant 4 TiB span of memory as `PROT_NONE`, before mapping 100 individual 1 MiB read/write mappings within it. One thing to note here is that `munmap()` is highly flexible – all mappings within the specified range are freed, no matter what is contained within it (see section 5.0.6 for a detailed discussion of how this is implemented).

14.4 Interpreting Out Of Memory reports

When the Out Of Memory (OOM) killer is invoked (see the OOM chapter for more details on the OOM killer), it outputs details about what caused the out of memory condition. Let's examine an example of this report, which appears in the kernel log (accessible via `dmesg`):-

```

[ 5.168012] oom invoked oom-killer: gfp_mask=0x140dca(GFP_HIGHUSER_MOVABLE|__GFP_COMP|
__GFP_ZERO), order=0, oom_score_adj=0
[ 5.168019] CPU: 0 PID: 246 Comm: oom Not tainted 6.0.0+ #27
[ 5.168021] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux
1.16.2-1-1 04/01/2014
[ 5.168022] Call Trace:
[ 5.168026] <TASK>
[ 5.168029] ? __dump_stack+0x1b/0x21
[ 5.168034] ? dump_stack_lvl+0x27/0x3a
[ 5.168036] ? dump_stack+0xc/0x11
[ 5.168038] ? dump_header+0x43/0x97
[ 5.168040] ? oom_kill_process+0xd0/0xd5

```

```

[ 5.168042] ? out_of_memory+0xbf/0x225
[ 5.168044] ? __alloc_pages_may_oom+0xea/0x14f
[ 5.168046] ? __alloc_pages_slowpath+0x2b3/0x4e6
[ 5.168048] ? __alloc_pages+0x157/0x181
[ 5.168050] ? __folio_alloc+0xf/0x36
[ 5.168052] ? vma_alloc_folio+0x79/0x1d9
[ 5.168054] ? do_anonymous_page+0x86/0x31f
[ 5.168056] ? follow_page_pte+0xe3/0x2a2
[ 5.168059] ? handle_pte_fault+0x144/0x168
[ 5.168060] ? __handle_mm_fault+0x26c/0x28d
[ 5.168063] ? handle_mm_fault+0x84/0xf6
[ 5.168064] ? faultin_page+0x52/0xc3
[ 5.168066] ? __get_user_pages+0x25c/0x311
[ 5.168068] ? populate_vma_page_range+0x46/0x67
[ 5.168070] ? __mm_populate+0x6b/0x106
[ 5.168071] ? vm_mmap_pgoff+0xdd/0xeb
[ 5.168074] ? ksys_mmap_pgoff+0x4c/0x154
[ 5.168076] ? __do_sys_mmap+0x12/0x23
[ 5.168078] ? __se_sys_mmap+0x5/0xa
[ 5.168079] ? __x64_sys_mmap+0x20/0x25
[ 5.168081] ? do_syscall_64+0x5a/0x86
[ 5.168083] ? entry_SYSCALL_64_after_hwframe+0x63/0xcd
[ 5.168086] </TASK>
[ 5.168087] Mem-Info:
[ 5.168087] active_anon:45 inactive_anon:1997113 isolated_anon:0
      active_file:301 inactive_file:300 isolated_file:0
      unevictable:0 dirty:56 writeback:0
      slab_reclaimable:2419 slab_unreclaimable:4196
      mapped:21 shmem:80 pagetables:4493 bounce:0
      kernel_misc_reclaimable:0
      free:25619 free_pcp:62 free_cma:0
[ 5.168090] Node 0 active_anon:180kB inactive_anon:7988452kB active_file:1204kB
      inactive_file:1200kB unevictable:0kB isolated(anon):0kB isolated(file):0kB mapped:84kB
      dirty:224kB writeback:0kB shmem:320kB shmem_thp: 0kB shmem_pmdmapped: 0kB anon_thp:
      7897088kB writeback_tmp:0kB kernel_stack:1792kB pagetables:17972kB all_unreclaimable? yes
[ 5.168093] Node 0 DMA free:15360kB boost:0kB min:124kB low:152kB high:180kB
      reserved_highatomic:0kB active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:0
      kB unevictable:0kB writepending:0kB present:15992kB managed:15360kB mlocked:0kB bounce:0kB
      free_pcp:0kB local_pcp:0kB free_cma:0kB
[ 5.168097] lowmem_reserve[]: 0 2959 7938 7938
[ 5.168099] Node 0 DMA32 free:44940kB boost:0kB min:25148kB low:31432kB high:37716kB
      reserved_highatomic:0kB active_anon:0kB inactive_anon:2984160kB active_file:0kB
      inactive_file:0kB unevictable:0kB writepending:0kB present:3129204kB managed:3035452kB
      mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:0kB
[ 5.168102] lowmem_reserve[]: 0 0 4978 4978
[ 5.168104] Node 0 Normal free:42176kB boost:0kB min:42304kB low:52880kB high:63456kB
      reserved_highatomic:0kB active_anon:180kB inactive_anon:5003724kB active_file:1604kB

```

```

inactive_file:800kB unevictable:0kB writepending:224kB present:5242880kB managed:5097828kB
mlocked:0kB bounce:0kB free_pcp:248kB local_pcp:248kB free_cma:0kB
[ 5.168107] lowmem_reserve[:]: 0 0 0 0
[ 5.168109] Node 0 DMA: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB 0*512kB 1*1024kB (U
) 1*2048kB (M) 3*4096kB (M) = 15360kB
[ 5.168117] Node 0 DMA32: 1*4kB (M) 4*8kB (UM) 4*16kB (UM) 3*32kB (U) 3*64kB (UM) 0*128kB
1*256kB (M) 1*512kB (M) 3*1024kB (M) 2*2048kB (UM) 9*4096kB (M) = 45188kB
[ 5.168127] Node 0 Normal: 340*4kB (UME) 164*8kB (UE) 113*16kB (UME) 76*32kB (UME) 53*64kB (
UE) 25*128kB (UE) 6*256kB (UME) 3*512kB (UME) 1*1024kB (E) 0*2048kB 6*4096kB (UM) = 42176
kB
[ 5.168137] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0 hugepages_size=2048kB
[ 5.168138] 706 total pagecache pages
[ 5.168138] 0 pages in swap cache
[ 5.168139] Free swap = 0kB
[ 5.168139] Total swap = 0kB
[ 5.168140] 2097019 pages RAM
[ 5.168141] 0 pages HighMem/MovableOnly
[ 5.168141] 59859 pages reserved
[ 5.168142] Tasks state (memory values in pages):
[ 5.168142] [ pid ] uid tgid total_vm rss pgtables_bytes swapents oom_score_adj
name
[ 5.168145] [ 115] 0 115 12408 270 90112 0 -250 systemd-
journal
[ 5.168147] [ 130] 0 130 7476 429 77824 0 -1000 systemd-
udevd
[ 5.168149] [ 135] 0 135 7312 317 81920 0 0 (udev-
worker)
[ 5.168150] [ 136] 0 136 7312 317 86016 0 0 (udev-
worker)
[ 5.168152] [ 137] 0 137 7312 328 77824 0 0 (udev-
worker)
[ 5.168153] [ 138] 0 138 7312 325 86016 0 0 (udev-
worker)
[ 5.168154] [ 139] 0 139 7312 328 81920 0 0 (udev-
worker)
[ 5.168155] [ 140] 0 140 7312 318 65536 0 0 (udev-
worker)
[ 5.168157] [ 141] 0 141 7312 319 81920 0 0 (udev-
worker)
[ 5.168158] [ 142] 0 142 7312 329 81920 0 0 (udev-
worker)
[ 5.168159] [ 143] 0 143 7312 327 77824 0 0 (udev-
worker)
[ 5.168160] [ 144] 0 144 7312 322 65536 0 0 (udev-
worker)
[ 5.168161] [ 145] 0 145 7312 321 86016 0 0 (udev-
worker)

```

```

[ 5.168163] [ 146] 0 146 7481 377 90112 0 0 (udev-
worker)
[ 5.168164] [ 147] 0 147 7312 323 81920 0 0 (udev-
worker)
[ 5.168165] [ 148] 0 148 7483 355 90112 0 0 (udev-
worker)
[ 5.168166] [ 149] 0 149 7312 326 81920 0 0 (udev-
worker)
[ 5.168167] [ 150] 0 150 7312 325 77824 0 0 (udev-
worker)
[ 5.168169] [ 151] 0 151 7312 326 86016 0 0 (udev-
worker)
[ 5.168170] [ 152] 0 152 7312 327 77824 0 0 (udev-
worker)
[ 5.168171] [ 153] 0 153 7312 327 77824 0 0 (udev-
worker)
[ 5.168172] [ 154] 0 154 7312 328 81920 0 0 (udev-
worker)
[ 5.168173] [ 155] 0 155 7345 341 81920 0 0 (udev-
worker)
[ 5.168174] [ 156] 0 156 7345 329 77824 0 0 (udev-
worker)
[ 5.168175] [ 157] 0 157 7312 330 77824 0 0 (udev-
worker)
[ 5.168176] [ 158] 0 158 7312 331 77824 0 0 (udev-
worker)
[ 5.168178] [ 163] 81 163 2192 131 53248 0 -900 dbus-
daemon
[ 5.168179] [ 166] 974 166 770 79 40960 0 0 dhcpcd
[ 5.168181] [ 167] 0 167 4315 242 81920 0 0 systemd-
logind
[ 5.168182] [ 169] 0 169 873 110 40960 0 0 dhcpcd
[ 5.168183] [ 170] 974 170 722 75 40960 0 0 dhcpcd
[ 5.168184] [ 171] 974 171 722 75 40960 0 0 dhcpcd
[ 5.168186] [ 183] 0 183 757 68 40960 0 0 agetty
[ 5.168187] [ 184] 0 184 2125 200 57344 0 0 login
[ 5.168188] [ 189] 1000 189 4894 377 77824 0 100 systemd
[ 5.168189] [ 190] 974 190 767 101 40960 0 0 dhcpcd
[ 5.168191] [ 191] 974 191 767 101 40960 0 0 dhcpcd
[ 5.168192] [ 195] 1000 195 6020 697 77824 0 100 (sd-pam)
[ 5.168193] [ 203] 1000 203 1846 434 49152 0 0 zsh
[ 5.168194] [ 246] 1000 246 33555023 1991355 16007168 0 0 oom
[ 5.168195] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=/,mems_allowed=0,
global_oom,task_memcg=/user.slice/user-1000.slice/session-c1.scope,task=oom,pid=246,uid
=1000
[ 5.168206] Out of memory: Killed process 246 (oom) total-vm:134220092kB, anon-rss:7965416kB
, file-rss:4kB, shmem-rss:0kB, UID:1000 pgtables:15632kB oom_score_adj:0

```

Listing 14-12: Example OOM report

This is rather intimidating and feels, at a glance feels, perhaps, impenetrable. Let's therefore break this down bit-by-bit and analyse the output, starting with the first line:-

14.4.1 Failed allocation statistics

The first line in the output outputs statistics related to the allocation which failed:-

```
[ 5.168012] oom invoked oom-killer: gfp_mask=0x140dca(GFP_HIGHUSER_MOVABLE|
    __GFP_COMP|__GFP_ZERO), order=0, oom_score_adj=0
```

Listing 14-13: Example OOM failed allocation statistics

This identifies the process which requested memory leading to the OOM condition. Importantly, this is not necessarily the process which is hogging memory, demand paging and overcommit (see section 4.1) dictate that any process at any time might invoke an OOM condition. It is simply the process which happened to trip the condition.

This line, generated by `dump_header()`, consists of four pieces of information:-

Invoking process name – The name of the process that caused the OOM condition. This process may very well be innocent, and simply happened to request memory under severe memory pressure.

In the example above this is a process called 'oom'.

GFP mask – The GFP flags (see section 2.6) used for the physical memory allocation in question. This is important to note, as it will indicate whether a specific zone was specified (e.g. `__GFP_DMA32`) or whether restrictive conditions were placed upon the allocation (e.g. `__GFP_ATOMIC`) which might have led to the allocation being infeasible.

In the example above however, this is a relatively straightforward user allocation – `GFP_HIGHUSER_MOVABLE` has been specified, indicating it was a user allocation which triggered the condition, which must be zeroed (`__GFP_ZERO`) and if it were compound to be initialised as a compound folio (`__GFP_COMP`).

Requested folio order – This is very important information – if the requested memory was of order 0, i.e. at the base page size (4 KiB for x86-64) then higher order fragmentation is certainly not the source of the out of memory condition. If it is, then this may well be a contributing factor.

In the example above, the triggering allocation is of order-0, so higher order folio fragmentation is not the cause.

oom_score_adj – Indicates the user-defined adjustment made to the out of memory 'score' used to apply a weight to processes which are being

considered for OOM killing. See the OOM chapter for more details on how this functions.

NOTE

The out of memory killer will not be invoked for higher-order allocations which exceed `PAGE_ALLOC_COSTLY_ORDER`, i.e. for any allocation at order-4 or above, as it is unlikely killing processes will do much to help free up such higher order memory under heavy fragmentation (see section 11.3 for more details on direct reclaim logic).

14.4.2 Stack trace

Immediately following this is a call stack indicating how the allocation occurred:-

```
[ 5.168019] CPU: 0 PID: 246 Comm: oom Not tainted 6.0.0+ #27
[ 5.168021] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS
Arch Linux 1.16.2-1-1 04/01/2014
[ 5.168022] Call Trace:
[ 5.168026] <TASK>
[ 5.168029] ? __dump_stack+0x1b/0x21
[ 5.168034] ? dump_stack_lvl+0x27/0x3a
[ 5.168036] ? dump_stack+0xc/0x11
[ 5.168038] ? dump_header+0x43/0x97
[ 5.168040] ? oom_kill_process+0xd0/0xd5
[ 5.168042] ? out_of_memory+0xbf/0x225
[ 5.168044] ? __alloc_pages_may_oom+0xea/0x14f
[ 5.168046] ? __alloc_pages_slowpath+0x2b3/0x4e6
[ 5.168048] ? __alloc_pages+0x157/0x181
[ 5.168050] ? __folio_alloc+0xf/0x36
[ 5.168052] ? vma_alloc_folio+0x79/0x1d9
[ 5.168054] ? do_anonymous_page+0x86/0x31f
[ 5.168056] ? follow_page_pte+0xe3/0x2a2
[ 5.168059] ? handle_pte_fault+0x144/0x168
[ 5.168060] ? __handle_mm_fault+0x26c/0x28d
[ 5.168063] ? handle_mm_fault+0x84/0xf6
[ 5.168064] ? faultin_page+0x52/0xc3
[ 5.168066] ? __get_user_pages+0x25c/0x311
[ 5.168068] ? populate_vma_page_range+0x46/0x67
[ 5.168070] ? __mm_populate+0x6b/0x106
[ 5.168071] ? vm_mmap_pgoff+0xdd/0xeb
[ 5.168074] ? ksys_mmap_pgoff+0x4c/0x154
[ 5.168076] ? __do_sys_mmap+0x12/0x23
[ 5.168078] ? __se_sys_mmap+0x5/0xa
[ 5.168079] ? __x64_sys_mmap+0x20/0x25
[ 5.168081] ? do_syscall_64+0x5a/0x86
[ 5.168083] ? entry_SYSCALL_64_after_hwframe+0x63/0xcd
[ 5.168086] </TASK>
```

Listing 14-14: Example OOM stack trace

Importantly, the first line indicates the *CPU* on which the allocation failed, the PID of the failing process (in this example process 'oom' possessing PID 246), the version of the kernel (6.0.0+) and its build ID (in this instance #27).

The header additionally indicates whether the kernel has been *tainted*, i.e. modified in some way that stability may be compromised compared to a vanilla kernel (for instance by loading an out-of-tree kernel module). In the example given, it has not been.

Finally, we observe the kernel stack trace that lead to the out of memory killers invocation, with a brief description of the hardware upon which the fault occurred.

We can observe that the page allocation slow path has been invoked (i.e. direct reclaim was attempted), in `__alloc_pages_slowpath()` (see section 11.1 for a detailed examination of this code), which lead to the OOM killer being invoked via `out_of_memory()`.

14.4.3 Global free page statistics

Following this information about the current state of memory in the system is reported, starting with global page counts:-

```
[ 5.168087] Mem-Info:
[ 5.168087] active_anon:45 inactive_anon:1997113 isolated_anon:0
           active_file:301 inactive_file:300 isolated_file:0
           unevictable:0 dirty:56 writeback:0
           slab_reclaimable:2419 slab_unreclaimable:4196
           mapped:21 shmem:80 pagetables:4493 bounce:0
           kernel_misc_reclaimable:0
           free:25619 free_pcp:62 free_cma:0
```

Listing 14-15: Example OOM memory info – Global free page statistics

This provides a summary of memory status data, invoked from `show_mem()` starting with global statistics (determined in `show_free_areas()`).

All values in this section are expressed in base pages, i.e. the minimum memory page size. For x86-64 pages are 4 KiB in size each. Global file-backed counts include both mapped and unmapped page cache entries.

Each of these other than `free_pcp` utilise either `global_node_page_state()` or `global_zone_page_state()` passing an `enum node_stat_item` or `enum zone_stat_item` parameter respectively to specify the desired counters:-

- `active_anon` – `node` – `NR_ACTIVE_ANON` (counter for `LRU_ACTIVE_ANON`) – Total number of *anonymous* pages (i.e. those which are not file-backed) which sit on the active LRU (i.e. which have been recently used and are unlikely to undergo reclaim in the near future, see section 11.2 for details on LRU lists in general).
- `inactive_anon` – `node` – `NR_INACTIVE_ANON` (counter for `LRU_INACTIVE_ANON`) – Total number of *anonymous* pages which sit on the inactive LRU (i.e.

which have not been recently used and are thus subject to reclaim in the near future should memory pressure arise).

- `isolated_anon - node - NR_ISOLATED_ANON` – Total number of *anonymous* pages which have been (ostensibly temporarily) isolated from an LRU list in order to have an operation performed on them such as migration, `madvise()`-invoked `MADV_PAGEOUT` request or hotplug via `folio_isolate_lru()` or `isolate_lru_page()`.

This and the below `isolated_file` can be a useful metric to determine whether actions which isolate pages from LRU lists (in turn protecting them from reclaim) might have led to an out of memory condition.

- `active_file - node - NR_ACTIVE_FILE` (counter for `LRU_ACTIVE_FILE`) – Total number of *file-backed* pages which sit on the active LRU. This excludes all `shmem` pages (i.e. RAM-backed files located in filesystems such as `tmpfs`), which are swap-backed, and therefore treated as anonymous memory.
- `inactive_file - node - NR_INACTIVE_FILE` (counter for `LRU_INACTIVE_FILE`) – Total number of *file-backed* pages which sit on the inactive LRU, excluding `shmem` pages.
- `isolated_file - node - NR_ISOLATED_FILE` – Total number of *file-backed* pages excluding `shmem` which have been ostensibly temporarily isolated for the same purposes as described in the description of `isolated_anon` above, only in this instance are non-`shmem` file-backed pages.
- `unevictable - node - NR_UNEVICTABLE` (counter for `LRU_UNEVICTABLE`) – Total number of pages placed on an *unevictable* LRU list (in actual fact the memory is not maintained on a list but rather simply designated unevictable).

These are pages that have both been marked locked via `mlock()` and have subsequently been isolated from either anonymous or file LRUs, which removes them from consideration for reclaim until unlocked, effectively pinning this user memory into RAM.

Excessive locking can result in memory pressure so this is a useful metric to determine whether a scenario such as this might have occurred.

- `dirty - node - NR_FILE_DIRTY` – Total number of file-backed pages which have been marked dirty, i.e. files which have been modified but are awaiting being written back to disk.

The number of dirty pages is rate-limited according to the dirty page tunables, for instance `vm.dirty_ratio` and `vm.dirty_background_ratio` (the ratio being that of dirty pages and total pages of *dirtyable* memory (see ?? for more details on the algorithm) expressed as a percentage (see section 14.6 below).

Once the foreground dirty limit has been reached, processes will block on I/O until sufficient writeback has occurred to bring the system below this limit.

When a file page undergoes writeback, it is not considered to be dirty in respect of this statistic, so these statistics are non-overlapping.

- `writeback - node - NR_WRITEBACK` – Total number of file-backed pages which are undergoing writeback (see section ?? for details of the writeback mechanism), i.e. which are in the process of being written back to disk. As stated in the description of `dirty` above, this is mutually exclusive with the `dirty` statistic.
- `slab_reclaimable - node - NR_SLAB_RECLAIMABLE_B` – Total number of kernel-allocated slab pages which can be subject to reclaim by invoking slab ‘shrinkers’ which reduce the size of slab caches to reclaim more memory (see section ?? for more details).
- `slab_unreclaimable - node - NR_SLAB_UNRECLAIMABLE_B` – Total number of kernel-allocated slab pages which cannot be subject to reclaim and are thus unreclaimable.
A large value in this field might indicate a kernel issue, for instance a driver which is allocating far too much memory and not providing a means to reclaim it.
- `mapped - node - NR_FILE_MAPPED` – Total number of file-backed (non-shmem) pages which are mapped as such into at least one set of process page tables (i.e. the same page cache entry mapped by multiple processes will only increment this statistic once).
- `shmem - node - NR_SHMEM` – Total number of shmem pages of memory mapped. This overlaps with anonymous page statistics as these are in effect anonymous (or rather, swap-backed) pages.
- `pagetables - node - NR_PAGETABLE` – Total number of pages used to populate page tables which map pages into userland memory (see section 3.1 for more on page tables in general).
- `bounce - zone - NR_BOUNCE` – Total number of block device ‘bounce’ buffer pages that have been allocated. These are used for devices which cannot access the entire range of memory the CPU is able to, which is enabled via `CONFIG_BOUNCE` which is turned on by default when `CONFIG_HIGHMEM` is set, i.e. the system is 32-bit.
Since we focus on 64 bit architectures only in this book (and specifically when examining architecture-dependent features, x86-64), this is out of scope.
- `kernel_misc_reclaimable - node - NR_KERNEL_MISC_RECLAIMABLE` – Total number of non-slab kernel pages allocated which are otherwise reclaimable. This does not appear to currently be used.
- `free - zone - NR_FREE_PAGES` – Total number of entirely free and allocatable pages in the system.
- `free_pcp` – This value is not derived from either global node or zone statistics, but rather each CPU’s per-cpu-page (PCP) data, from which free pages are accumulated.
The PCP interface provides a fast per-CPU cache for allocating and freeing of a small number of pages which improves page allocation performance (see section 2.7.3 for a complete description).

This value therefore is the total number of free PCP pages across all CPUs.

- `free_cma - zone - NR_FREE_CMA_PAGES` – The total number of free CMA pages, a topic which is outside the scope of this book.

14.4.4 Per-node statistics

Next, statistics are displayed for each online node:-

```
[ 5.168090] Node 0 active_anon:180kB inactive_anon:7988452kB active_file
:1204kB inactive_file:1200kB unevictable:0kB isolated(anon):0kB isolated(
file):0kB mapped:84kB dirty:224kB writeback:0kB shmem:320kB shmem_thp: 0
kB shmem_pmdmapped: 0kB anon_thp: 7897088kB writeback_tmp:0kB
kernel_stack:1792kB pagetables:17972kB all_unreclaimable? yes
```

Listing 14-16: Example OOM memory info – Per-node statistics

These largely provide the same data as above (although expressed in KiB), noting that in the example shown there is only a single node. Examining the additional fields, all of which are derived from `global_node_page_state()`, again invoked from `show_free_areas()`:-

- `shmem_thp - NR_SHMEM_THPS` – Kilobytes of memory used for shmem (mapping files in RAM-backed filesystem such as tmpfs) which are allocated with sufficient physical contiguity that they can be mapped as huge pages by the Transparent Huge Page mechanism (see the chapter on huge pages for more on THP).
- `shmem_pmdmapped - NR_SHMEM_PMDMAPPED` – Kilobytes of memory used for shmem which are currently mapped at the PMD page table level (i.e. which are actually being used as huge pages).
- `anon_thp - NR_ANON_THPS` – Kilobytes of *anonymous* (i.e. non-file backed) memory that is currently mapped as huge pages by the Transparent Huge Page system.
- `writeback_tmp - NR_WRITEBACK_TEMP` – Kilobytes of *file-backed* memory currently undergoing writeback from temporary buffers. This is currently only used by the FUSE userland file system interface.
- `kernel_stack - NR_KERNEL_STACK_KB` – Kilobytes of memory utilised for the stack associated with the kernel portion of processes. Updated in `account_kernel_stack()` in the kernel forking code (in `exit_task_stack_account()` and `dup_task_struct()`).
- `all_unreclaimable?` – If indirect reclaim (via the kswapd kernel thread, see section 11.4 for more details) has failed to make progress freeing *any* pages on a node several times in a row, then direct reclaim will be attempted which will try to make more drastic steps to free memory (see section 11.3 for details on how this functions).

At this stage, the node is considered unreclaimable, and thus in that instance this field will be set to yes, otherwise it is set to no.

The value is determined by `pg_data_t->kswapd_failures` (incremented each time `balance_pgdat()` fails to make progress reclaiming memory) – if it equal to or greater than `MAX_RECLAIM_RETRIES` (hardcoded to 16) then the node is considered unreclaimable.

14.4.5 Zone-specific statistics

And then zone-specific statistics similar to what you would observe in `/proc/zoneinfo` are displayed for each zone in each online node:-

```
[ 5.168093] Node 0 DMA free:15360kB boost:0kB min:124kB low:152kB high:180
kB reserved_highatomic:0kB active_anon:0kB inactive_anon:0kB active_file
:0kB inactive_file:0kB unevictable:0kB writepending:0kB present:15992kB
managed:15360kB mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB
free_cma:0kB
[ 5.168097] lowmem_reserve[]: 0 2959 7938 7938
[ 5.168099] Node 0 DMA32 free:44940kB boost:0kB min:25148kB low:31432kB
high:37716kB reserved_highatomic:0kB active_anon:0kB inactive_anon
:2984160kB active_file:0kB inactive_file:0kB unevictable:0kB writepending
:0kB present:3129204kB managed:3035452kB mlocked:0kB bounce:0kB free_pcp
:0kB local_pcp:0kB free_cma:0kB
[ 5.168102] lowmem_reserve[]: 0 0 4978 4978
[ 5.168104] Node 0 Normal free:42176kB boost:0kB min:42304kB low:52880kB
high:63456kB reserved_highatomic:0kB active_anon:180kB inactive_anon
:5003724kB active_file:1604kB inactive_file:800kB unevictable:0kB
writepending:224kB present:5242880kB managed:5097828kB mlocked:0kB bounce
:0kB free_pcp:248kB local_pcp:248kB free_cma:0kB
[ 5.168107] lowmem_reserve[]: 0 0 0 0
```

Listing 14-17: Example OOM memory info – Per-zone statistics

Importantly, the watermarks and free pages are shown here for each zone which gives a clear indication of the condition that lead to the out of memory condition (for instance, if not a higher order page allocation caused by fragmentation, then likely the minimum watermark has been violated).

See section 2.4 for a detailed discussion of watermarks, but generally speaking for each zone, once free memory dips below its low water mark, indirect reclaim (see section 11.4) is started via a background kernel thread (and stopped once the high watermark is reached).

When free memory dips below its *minimum* water mark, direct reclaim (more drastic, blocking efforts to free memory) is tried (see section 11.3). If progress cannot be made, the out of memory killer is invoked.

Examining the example watermarks:-

Table 14-1: Example OOM watermark status (KiB)

Zone	Minimum	Low	High	Free
DMA	124	152	180	15,360
DMA32	45,060	51,344	37,57,628	44,940
Normal	42,304	52,880	63,456	42,176

In the output shown in listing 14-17, each zone's low memory reserve statistics are shown, labelled as `lowmem_reserve[]`. These are equivalent to the protection values shown in `/proc/zoneinfo`, and described in detail in section 2.4.1 in detail.

In the example above, each entry contains 4 values, each of which refers to the 4 zones contained in the node (note that the final zone, `movable`, does not have statistics displayed separately for it as it is empty in this example).

Each of the low reserve statistics indicate how many additional pages must be reserved in order to consider allocations from zones above this zone, for instance DMA requires there to be 2,959 pages or 11,836 KiB reserved memory if allocating from DMA32, and 7,938 pages or 31,752 KiB reserved memory if allocating from Normal, etc.

If we take into account low memory reserve values, and assume that failing allocation is taken from the normal zone (this is the case unless a GFP flag specifying a different zone is shown in the output, not the case in the example), we end up with the following values:-

Table 14-2: Example OOM watermark status with low memory reserve (KiB)

Zone	Minimum	Low	High	Free
DMA	31,876	31,904	31,932	15,360
DMA32	25,148	31432	37,716	44,940
Normal	42,304	52,880	63,456	42,176

Factoring in the low memory reserve, we can observe immediately that no zone can service the memory request, hence why the out of memory condition has arisen as free pages in each case represent less than the minimum watermark and all reclaim has already been attempted.

Note that, if any zone has had a 'boost' applied, indicating that page blocks are especially fragmented, then this value should be added to each of the watermarks. In the example above there is no boost (see section 2.5 for more details).

Each zone within each node is listed, stating the node number, the zone name and a number of statistics, as determined by `show_free_areas()` (all values expressed in base pages):-

- `free` – The number of free pages within the zone.
- `boost` – The amount of watermark boost which is applied. When page blocks become fragmented, zone watermarks are increased in order to reduce the chances of this causing an out of memory condition. This value is set in `boost_watermark()` which is triggered from `steal_suitable_fallback()`. See section 2.4 for more details.

- `min` – The minimum water mark for this zone.
- `low` – The low water mark for this zone.
- `high` – The high water mark for this zone.
- `reserved_highatomic` – `struct zone->nr_reserved_highatomic` – A certain number of pages can be reserved exclusively for higher order atomic context allocations, as obtaining these otherwise might not be feasible. This field contains the number of pages reserved for this purpose. See section 2.4 for more details.
- `active_anon` – The zone-specific count of of anonymous base pages within the zone which are on the active LRU, i.e. those which are unlikely to be reclaimed soon. Determined from `zone_page_state()`, using field `NR_ZONE_ACTIVE_ANON`.
- `inactive_anon` – The zone-specific count of anonymous base pages within the zone which are on the inactive LRU, i.e. those which are more likely to be reclaimed soon. Determined from `zone_page_state()`, using field `NR_ZONE_INACTIVE_ANON`.
- `active_file` – The zone-specific count of file-backed base pages within the zone that are on the active LRU, i.e. those which are more likely to be reclaimed soon. Determined from `zone_page_state()`, using field `NR_ZONE_ACTIVE_FILE`.
- `inactive_file` – The zone-specific count of file-backed base pages within the zone that are on the inactive LRU, i.e. those which are less likely to be reclaimed soon. Determined from `zone_page_state()`, using field `NR_ZONE_INACTIVE_FILE`.
- `unevictable` – The number of pages in the zone which have been `mlock()`'d, and are thus not on any LRU and not subject to reclaim at all. If this number is high, then this will add memory pressure due to the inability to reclaim it. Determined from `zone_page_state()`, using field `NR_ZONE_UNEVICTABLE`.
- `writepending` – The number of pages in the zone which are either dirty pending writeback to a file system which performs it, currently under writeback or awaiting being cleaned. Determined from `zone_page_state()`, using field `NR_ZONE_WRITE_PENDING`. Updated in `folio_account_dirtied()` when dirtied, `folio_account_cleaned()`, `folio_clear_dirty_for_io()` when cleaned, and `__folio_start_writeback()` and `__folio_end_writeback()` when starting/ending writeback (see section ?? for more on writeback).
- `present` – `struct zone->present_pages` – The number of *present* pages within the zone, i.e. the raw range over which it spans which are backed by physical memory.
- `managed` – `struct zone->managed_pages` – The number of *managed* pages within the zone, i.e. present pages which are available to the buddy allocator (whether allocated or not).

- `mlocked` – The number of resident pages which have been marked as `mlock()`'d. This may differ from the `unevictable` value above as the locking of a page and the placing of it on the `unevictable` list do not occur at precisely the same time. Determined from `zone_page_state()`, using field `NR_MLOCK`.
- `bounce` – The number of bounce buffers within the zone. This is highly unlikely to be relevant to a 64-bit system, see the description of this in section 14.2 above for more details.
- `free_pcp` – The number of free Per-CPU-Pages (PCPs) on all CPUs for this zone. See section 2.7.3 for more on PCPs. This is calculated by summing the per-CPU `struct zone->per_cpu_pageset` values for each online CPU in `show_free_areas()`.
- `local_pcp` – The number of free Per-CPU-Pages (PCPs) on the local CPU for this zone. See section 2.7.3 for more on PCPs.
- `free_cma` – Related to linux CMA functionality, which is out of scope for the book.

14.4.6 Zone buddy page statistics

After this each node and zone outputs its buddy page statistics, giving details similar to `/proc/buddyinfo`:-

```
[ 5.168109] Node 0 DMA: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB
0*512kB 1*1024kB (U) 1*2048kB (M) 3*4096kB (M) = 15360kB
[ 5.168117] Node 0 DMA32: 1*4kB (M) 4*8kB (UM) 4*16kB (UM) 3*32kB (U) 3*64
kB (UM) 0*128kB 1*256kB (M) 1*512kB (M) 3*1024kB (M) 2*2048kB (UM) 9*4096
kB (M) = 45188kB
[ 5.168127] Node 0 Normal: 340*4kB (UME) 164*8kB (UE) 113*16kB (UME) 76*32
kB (UME) 53*64kB (UE) 25*128kB (UE) 6*256kB (UME) 3*512kB (UME) 1*1024kB
(E) 0*2048kB 6*4096kB (UM) = 42176kB
[ 5.168137] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0
hugepages_size=2048kB
```

Listing 14-18: Example OOM memory info – Zone buddy page statistics

These list available buddy allocator pages (the underlying physical allocator in the kernel, see 2.7 for details on this) at different physically contiguous block sizes, each of which are equivalent to a folio in increasing order (the order of a folio is the power-of-2 number of adjacent physical base pages it spans, e.g. an order-1 folio is $2^1 = 2$ pages in size, order-2 $2^2 = 4$ pages in size, order-10 $2^{10} = 1024$ pages in size, etc.

At the end of the report, per-node huge page data is also reported.

In the example given (from an x86-64 system), the base page size is 4 KiB, in a system with a single node and 3 zones (DMA, DMA32 and Normal). Each of the folio orders has its size explicitly shown and then a series

of letters in parentheses indicating which *migrate type* page blocks are available at each order.

Migrate types determine what blocks of memory pages should be preferred to be used for. Memory is subdivided like this in order to avoid mixing of memory that is *movable* (e.g. user memory which the kernel can move around as it likes), and that which is not (e.g. kernel memory which cannot be moved).

When memory is freed back to the buddy allocator, it *coalesces* adjacent blocks of memory into higher order folios. If some memory that would otherwise be coalesced is currently allocated, if movable, the kernel could move it elsewhere to enable the operation to continue and to expand physically contiguous memory (see the chapter on migration and compaction for more on this).

However, if a single page of memory is unmovable in the middle of this block, it is fragmented until that is freed. Thus subdividing into chunks, known as ‘pageblocks’ (of order `pageblock_order`, this is typically order-9 for x86-64) of the same migrate type is a heuristic means of reducing fragmentation.

See section 2.5 for a detailed analysis of migrate types.

The mapping from migrate type to abbreviation seen here is performed in `show_migration_types()` and is as follows:-

- U – `MIGRATE_UNMOVABLE` – *Unmovable* – Kernel allocations which can neither be reclaimed in any fashion nor moved.
- M – `MIGRATE_MOVABLE` – *Movable* – Userland allocations which can be moved.
- E – `MIGRATE_RECLAIMABLE` – *Movable* – Kernel slab allocations that can be shrunk and thus reclaimed, rendering them in effect movable.
- H – `MIGRATE_HIGHATOMIC` – *High atomic* – Memory that has been specifically reserved for high-order atomic memory allocations which, operating in atomic mode, cannot wait to receive their memory.
- C – `MIGRATE_CMA` – *CMA* – CMA memory, out of scope for the book.
- I – `MIGRATE_ISOLATE` – *Isolated* – Isolated (non-allocatable) memory as defined by the `CONFIG_MEMORY_ISOLATION` feature which is out of scope for the book.

If memory of one migrate type is required but the zone lacks memory of this, it can be taken from another kind. The mechanism for doing so is rather complicated and heuristic (again, see section 2.5 for more details on this).

In the worst case, fragmentation will be permitted by splitting up page blocks, however this is avoided as much as possible (as doing so defeats the purpose of maintaining this heuristic in the first place), and rather entire page blocks are preferred to be ‘stolen’ at once, i.e. converted to the required migrate type all at once (the migrate types section 2.5 goes into copious details on this logic).

Some useful rules of thumb:-

- Migrate types of H, C and I cannot be ‘stolen’ or taken from at all. They truly are a finite resource.
- Migrate types at pageblock order or higher are more or less irrelevant as they can immediately be stolen, so if free memory exists at (for instance on x86-64) order-9 or above then migrate type fragmentation is unlikely to be an issue.
- Equally, migrate types at half-pageblock order (e.g. with pageblock size order-9, this will be equal to order-4) or higher may also be stolen from.
- Allocations of unmovable or reclaimable memory always go ahead and steal movable pageblocks regardless.

Generally speaking, these values are only really worth evaluating if there is a significant lack of higher order buddy pages, which might indicate fragmentation. By examining the distribution of migrate types at each order, it might be possible to discern what kind of allocations might be leading to this fragmentation.

Expanding the buddy statistics from our example report to a table:-

Table 14-3: Example OOM free buddy pages

Zone	Order/Migrate types											Total Free
	0	1	2	3	4	5	6	7	8	9	10	
DMA	0	0	0	0	0	0	0	0	1 U	1 M	3 M	15,360 KiB
DMA32	1 M	4 UM	4 UM	3 U	3 UM	0	1 M	1 M	3 M	2 UM	9 M	45,188 KiB
Normal	340 UME	164 UE	113 UME	76 UME	53 UE	25 UE	6 UME	3 UME	1 E	0	6 UM	42,176 KiB

14.4.7 Global statistics

The ‘Show memory’ portion of the output is wrapped up with some global statistics:-

```
[ 5.168138] 706 total pagecache pages
[ 5.168138] 0 pages in swap cache
[ 5.168139] Free swap = 0kB
[ 5.168139] Total swap = 0kB
[ 5.168140] 2097019 pages RAM
[ 5.168141] 0 pages HighMem/MovableOnly
[ 5.168141] 59859 pages reserved
```

Listing 14-19: Example OOM global statistics

Each statistic here is relatively self-explanatory, though it’s important to note that the ‘reserved’ value is physical memory reserved by the system and thus unavailable for allocation, rather than any other definition of this word within memory management, and equal to sum of the differences of each zone’s present and managed pages (see section 2.4 for more details on this).

Note that all of the above ‘Show memory’ information can also be obtained via the sysrq-trigger interface specifying the `m` option, for instance:-

```
$ echo m | sudo tee /proc/sysrq-trigger >/dev/null
$ dmesg
... details as above ...
```

Listing 14-20: Example of obtaining memory info from /proc/sysrq-trigger

14.4.8 Out of memory killer report

Finally, the decision process of the OOM killer is shown with all running processes listed:-

[	5.168142]	[	pid	]	uid	tgid	total_vm		rss	pgtables_bytes	swapents	oom_score_adj
	name											
[	5.168145]	[	115]		0	115	12408		270	90112	0	-250 systemd-journal
[	5.168147]	[	130]		0	130	7476		429	77824	0	-1000 systemd-udevd
[	5.168149]	[	135]		0	135	7312		317	81920	0	0 (udev-worker)
[	5.168150]	[	136]		0	136	7312		317	86016	0	0 (udev-worker)
[	5.168152]	[	137]		0	137	7312		328	77824	0	0 (udev-worker)
[	5.168153]	[	138]		0	138	7312		325	86016	0	0 (udev-worker)
[	5.168154]	[	139]		0	139	7312		328	81920	0	0 (udev-worker)
[	5.168155]	[	140]		0	140	7312		318	65536	0	0 (udev-worker)
[	5.168157]	[	141]		0	141	7312		319	81920	0	0 (udev-worker)
[	5.168158]	[	142]		0	142	7312		329	81920	0	0 (udev-worker)
[	5.168159]	[	143]		0	143	7312		327	77824	0	0 (udev-worker)
[	5.168160]	[	144]		0	144	7312		322	65536	0	0 (udev-worker)
[	5.168161]	[	145]		0	145	7312		321	86016	0	0 (udev-worker)
[	5.168163]	[	146]		0	146	7481		377	90112	0	0 (udev-worker)
[	5.168164]	[	147]		0	147	7312		323	81920	0	0 (udev-worker)
[	5.168165]	[	148]		0	148	7483		355	90112	0	0 (udev-worker)
[	5.168166]	[	149]		0	149	7312		326	81920	0	0 (udev-worker)

```

[  5.168167] [  150]    0  150   7312   325   77824    0    0 (udev-
worker)
[  5.168169] [  151]    0  151   7312   326   86016    0    0 (udev-
worker)
[  5.168170] [  152]    0  152   7312   327   77824    0    0 (udev-
worker)
[  5.168171] [  153]    0  153   7312   327   77824    0    0 (udev-
worker)
[  5.168172] [  154]    0  154   7312   328   81920    0    0 (udev-
worker)
[  5.168173] [  155]    0  155   7345   341   81920    0    0 (udev-
worker)
[  5.168174] [  156]    0  156   7345   329   77824    0    0 (udev-
worker)
[  5.168175] [  157]    0  157   7312   330   77824    0    0 (udev-
worker)
[  5.168176] [  158]    0  158   7312   331   77824    0    0 (udev-
worker)
[  5.168178] [  163]   81  163   2192   131   53248    0   -900 dbus-
daemon
[  5.168179] [  166]  974  166    770    79   40960    0    0 dhcpcd
[  5.168181] [  167]    0  167   4315   242   81920    0    0 systemd-
logind
[  5.168182] [  169]    0  169    873   110   40960    0    0 dhcpcd
[  5.168183] [  170]  974  170    722    75   40960    0    0 dhcpcd
[  5.168184] [  171]  974  171    722    75   40960    0    0 dhcpcd
[  5.168186] [  183]    0  183    757    68   40960    0    0 agetty
[  5.168187] [  184]    0  184   2125   200   57344    0    0 login
[  5.168188] [  189] 1000  189   4894   377   77824    0   100 systemd
[  5.168189] [  190]  974  190    767   101   40960    0    0 dhcpcd
[  5.168191] [  191]  974  191    767   101   40960    0    0 dhcpcd
[  5.168192] [  195] 1000  195   6020   697   77824    0   100 (sd-pam)
[  5.168193] [  203] 1000  203   1846   434   49152    0    0 zsh
[  5.168194] [  246] 1000  246 33555023 1991355 16007168    0    0 oom
[  5.168195] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=/,mems_allowed=0,
global_oom,task_memcg=/user.slice/user-1000.slice/session-c1.scope,task=oom,pid=246,uid
=1000
[  5.168206] Out of memory: Killed process 246 (oom) total-vm:134220092kB, anon-rss:7965416kB
, file-rss:4kB, shmem-rss:0kB, UID:1000 pgtables:15632kB oom_score_adj:0

```

Listing 14-21: Example OOM kill report

This lists all running processes within the system, their OOM score adjustment (see section ?? for more on this), which is used to make the OOM killer more or less likely to kill a process and a number of statistics including each process's PID (from the kernel's perspective) as well as its TGID (Thread Group ID).

This output is generated from `dump_tasks()`, which calls `dump_task()` for each thread. It's important to note that, in Linux, each individual thread is implemented as a separate process, only with a shared virtual address space with all other threads spawned from the original process and threads it originally spawned.

Therefore, when userland encounters a 'PID' it is in fact a TGID value. Therefore, in order to recognise a PID as understood within userland, the TGID column should be referred to.

The total virtual address space occupied by each thread, its Resident Set Size (RSS), memory used to store page table data and swap space are shown. The determining factor of which process is killed is a combination of the latter three of these values, most notably RSS, moderated by the `oom_score_adj` value.

The OOM killer displays any constraints applied to the allocation which might impact the OOM decision (this is relevant to memory cgroup and NUMA memory policies, both of which are out of scope for the book).

Finally, a summary is shown detailing the killed process, its statistics (separating anonymous, non-shmem file-backed and shmem resident memory) and page table and `oom_score_adj` values.

See the out of memory killer chapter for a more detailed analysis of how the kernel handles out of memory conditions.

14.5 procfs memory interfaces

The `procfs` file system provides a great deal of memory management-specific information which can be used to gain insight into the memory status of processes and the system as a whole.

There are a number of different memory interfaces available within the kernel:-

- `/proc/vmstat` – A wrapper around *all* global memory statistical counters, shown in raw form. This lists *everything* that is counted in the kernel in units of base pages. The kernel implementation is in `vmstat_show()`.
- `/proc/meminfo` – While `vmstat` is comprehensive in terms of *counters*, it lacks basic information such as total memory, the heuristically determined available memory among other useful calculations. `meminfo` therefore is the more useful primary source of global memory information, expressed in kilobytes for convenience. The kernel implementation is in `meminfo_proc_show()`.
- `/proc/zoneinfo` – Shows per-node statistics, followed by per-zone statistics, page counts, low memory reserve values and watermarks. In addition, per-zone pagesets (or Per-CPU-Pages/PCPs) (see section 2.7.3) are shown. The kernel implementation is in `zoneinfo_show()`.
- `/proc/buddyinfo` – Provides a count of free pages for each node, zone and folio order. The kernel implementation is in `frag_show()`.

- `/proc/pagetypeinfo` – (*requires elevated permissions*) – Outputs the page back order (see the chapter on physical memory for details on the concept of a page block).

This shows the implied pages per-block, a table showing the count of free pages for each node, zone and migrate type and a table showing how many page blocks are assigned to each migrate type per node and zone. The kernel implementation is in `pagetypeinfo_show()`.

If the `CONFIG_PAGE_OWNER` configuration option is specified, then this shows additional data, however discussion of this feature is out of scope for the book.

- `/proc/pressure/memory` – An interface which forms part of the *PSI* (Pressure Stall Information) framework, which attempts to give an idea of memory pressure.

This outputs rolling-average values indicating the proportion of process time *some* processes are stalled on memory. The kernel implementation is at `psi_memory_show()`.

- `/proc/slabinfo` – (*requires elevated permissions*) – Provides output listing the number of allocated and free slab cache objects of each type (see the slab chapter for more details). The kernel implementation is in `slab_show()`.

- `/proc/vmallocinfo` – (*requires elevated permissions*) – A list of allocated `vmalloc` kernel ranges. If available the function which allocated the memory will be listed, along with range size.

For security reasons, the pointers here will be obfuscated (unless the kernel command line option `no_hash_pointers` has been passed). The kernel implementation is in `s_show()`.

- `/proc/kcore` – Maps the physical memory of the system in the form of an ELF core file. If a tool has symbols available it can seek into this and debug the kernel live accessing memory as required. This is implemented in `fs/proc/kcore.c`. This is out of scope for the book, but is used to enable debuggers like `crash` and `drgn` to gain debug information about a live running kernel.

- `/proc/kpagecount`, `/proc/kpageflags` – (*requires elevated permissions*) – These interfaces, part of the *pagemap* implementation, allow a user to obtain the number of times a particular physical page of memory is mapped into userspace (i.e. its `mapcount`) and page table flags, respectively. These values can be obtained by treating these files as arrays of 64-bit integers and seeking to the Page Frame Number index (PFN, or index of a physical page, equal to the physical address divided by system page size).

The kernel implementation is implemented in `kpageflags_read()` and `kpagecount_read()`.

- `/proc/$pid/status`, `/proc/$pid/stat`, `/proc/$pid/statm` – Shows process status, including memory statistics, most notably `VmRSS` and `VmHWM` indicating

the current and high watermark Resident Set Size usage for the process. The kernel implementation is in `task_mem()`.

- `/proc/$pid/maps`, `/proc/$pid/smmaps`, `/proc/$pid/smmaps_rollup` – A very useful and important set of interfaces which walk a process’s page tables and accumulates data about each mapping within that process, listing virtual address ranges, permissions, offset, path and major/minor device number for the file if mapping a file and name of anonymous mapping (if any) if it is not.

The `maps` interface provides one line per-mapping data, `smmaps` provides accumulated statistics about underlying physical pages in addition to this and `smmaps_rollup` outputs the *total* virtual address range, aggregating these statistics across the whole process.

The kernel implementation are in `show_map()`, `show_smap()` and `show_smmaps_rollup()` respectively.

- `/proc/$pid/map_files` – A handy list of mapped files implemented as a set of subdirectories with names in the format `from-to` (where `from` and `to` are the hexadecimal page-aligned addresses spanning the range), each of which is symlinked to the file they map. The kernel implementation is in `proc_map_files_readdir()`.
- `/proc/$pid/numa_map` – Shows a list of mappings similar to the `maps` interface, alongside NUMA details, with `Nx=y` indicating that node `x` has `y` pages mapped, `mapmax` which shows the maximum number of processes mapping a single page in this range encountered during the scan alongside other statistics.

The kernel implementation of this is in `show_numa_map()`.

- `/proc/$pid/oom_score`, `/proc/$pid/oom_adj`, `/proc/$pid/oom_score_adj` – `oom_score` reads the current Out Of Memory (OOM) killer “score” for a process (indicating how much memory it is occupying, with any adjustment applied) and `oom_score_adj` allowing this score to be manually adjusted per-process (`oom_adj` is deprecated).

The `oom_score_adj` is expressed in units of one tenth of a percent of total RAM and swap, and can range from -1000 (-100%) to 1000 (+100%).

`oom_score` is expressed in the same way, only it is normalised to a range of 0 to 2000.

See Chapter 13 and Sections 13.2.1 and 13.2 for more details on this.

The kernel implementation for `oom_score` is in `proc_oom_score()`. For `oom_score_adj`, `oom_score_adj_read()` implements reading from this value and `oom_score_adj_write()` implements writing to it.

- `/proc/$pid/pagemap` – A series of 64-bit values, one for each page in the virtual address space. Values from these can be obtained by seeking to the virtual page number multiplied by `sizeof(uint64_t)` (i.e. 8). This is documented in *pagemap*.

Each encodes a number of fields indicating whether the page is present in RAM/swapped out/not present at all, the Page Frame Number (PFN) of the underlying physical page if present, alongside other values.

The kernel implementation is in `pagemap_read()`.

- `/proc/$pid/mem` – Provides raw access to the entire virtual address space of a process. This is only accessible by the process itself and processes with sufficient permissions to attach to the process for the purposes of tracing.

This is used by debuggers for instance and can be read from and written to. The offset into the file specifies the virtual address to read from or write to (adjusted by seeking).

This functionality is implemented in the kernel via `mem_rw()`, which utilises the Get User Pages or GUP functionality (see section 8.1.2 for more details on this) in `access_remote_vm()`.

These interfaces are largely self-explanatory well documented in the `procfs` man pages for the most part, so we won't go into enormous depth here on each, however let's take a moment to examine some in detail to explore the kind of information we can obtain:-

14.5.1 A quick tour: Physical memory

Let's examine the state of *physical memory* using `procfs` interfaces. The first port of call is `/proc/meminfo` which provides details on available free memory as discussed in section 14.1 above:-

```
MemTotal:      4000696 kB
MemFree:       302552 kB
MemAvailable:  3112644 kB
```

Listing 14-22: Example `/proc/meminfo` free memory fields

These fields are calculated in `meminfo_proc_show()` and provide the foundations of understanding available free memory – the total physical, usable, memory installed on the system, the amount of free memory and the amount of available memory, if reclaimable memory were freed.

Next, examining `/proc/zoneinfo` which provides per-node, per-zone details of memory state in the system (focusing on the most important values):-

```
Node 0, zone      DMA
per-node stats
  nr_inactive_anon 4014
  nr_active_anon   43
  nr_inactive_file 7977
  nr_active_file   9454
  nr_unevictable   0
  nr_slab_reclaimable 3308
  nr_slab_unreclaimable 3963
  ...
  nr_anon_pages    3977
  nr_mapped        6017
  nr_file_pages    17511
  nr_dirty         30
```

```

nr_writeback 0
nr_writeback_temp 0
nr_shmem      80
...
nr_vmscan_write 0
nr_vmscan_immediate_reclaim 0
nr_dirtied    2947
nr_written    2917
nr_throttled_written 0
nr_kernel_misc_reclaimable 0
nr_foll_pin_acquired 0
nr_foll_pin_released 0
nr_kernel_stack 1376
nr_page_table_pages 196
nr_swapcached 0

```

Listing 14-23: Example /proc/zoneinfo output – per-node statistics

The first zone in each node also includes per-node statistics. These reflect the state of counters found in /proc/vmstat, though in this instance those values specific to the described node.

Following this are per-zone statistics, for instance example output for the normal zone (focusing on core statistics):-

```

Node 0, zone   Normal
pages free    1236891
      boost    0
      min      10576
      low      13220
      high     15864
      spanned  1310720
      present  1310720
      managed  1274457
      ...
      protection: (0, 0, 0, 0)
nr_free_pages 1236891
nr_zone_inactive_anon 4014
nr_zone_active_anon 43
nr_zone_inactive_file 7977
nr_zone_active_file 9454
nr_zone_unevictable 0
nr_zone_write_pending 30
nr_mlock      0
      ...
pagesets
cpu: 0
      count: 1913
      high: 3305
      batch: 63

```

```

vm stats threshold: 42
  cpu: 1
        count: 1561
        high:  3305
        batch: 63
vm stats threshold: 42
  cpu: 2
        count: 815
        high:  3305
        batch: 63
vm stats threshold: 42
  cpu: 3
        count: 1920
        high:  3305
        batch: 63
vm stats threshold: 42
node_unreclaimable: 0
start_pfn:          1048576

```

Listing 14-24: Example `/proc/zoneinfo` output – Normal zone statistics

All values are expressed in base pages. The counters are broadly self-explanatory, so we shall focus on the zone-specific fields:-

- `free` – The number of free pages in the zone.
- `boost` – If pageblocks have become fragmented, an additional ‘boost’ is applied to each watermark to reduce the chances of this causing more fragmentation. If such a boost exists, it will be shown here (see section 2.4 for a detailed explanation).
- `min` – The minimum watermark. If free pages dip below this level, allocations trigger direct reclaim (see section 11.3) and block until this succeeds or if it cannot, triggers the Out Of Memory (OOM) killer.
- `low` – The low watermark. If free pages dip below this level, indirect reclaim is started across all nodes in the permitted nodelist for the allocation (typically this means all nodes).
- `high` – The high watermark. If indirect reclaim is active for a node, it sleeps if all zones are at or above their high watermarks.
- `spanned` – The total number of physical pages spanned over this zone within this node.
- `present` – Of spanned pages, the total number that are not holes (regions of apparent memory addresses that don’t actually address any memory) or otherwise reserved for hardware.
- `managed` – The number of pages available to the buddy allocator. This differs from present due to memory being reserved for various purposes in `free_area_init_core()`, most typically the majority arising from memmap reservation, i.e. the array of struct page objects which describe each physical page (see 2.3 for more details on the memory model as a whole).

- **protection** – Indicates the low memory reserve values for this zone in relation to the zones that sit above it in this node (see section 2.4.1 for more details on this).

See section 2.4 for more on zones.

Beneath this there is *pageset* or Per-CPU-Page (PCP) data for each CPU, a cache which sits between node free lists and physical allocations. Examining each field:-

- **count** – The number of base pages currently present in this CPU's Per-CPU-Page cache (higher order pages are summed according to the base page count they span).
- **high** – The maximum number of pages that can be held in this CPU's cache before they start to get flushed to free lists.
- **batch** – The number of pages that are placed in the cache from free lists when populated or freed back to them when the cache exceeds high.
- **vm stats threshold** – The page increment to PCP statistics required before the statistics are actually updated.

There are separate page sets for each migrate type and orders up to and including `PAGE_ALLOC_COSTLY_ORDER` (hardcoded to 3). These statistics aggregate all orders and migrate types. See section 2.7.3 for more on pagesets.

If we wish to examine the current state of the buddy allocator, `/proc/buddyinfo` is a good place to start:-

Node 0, zone DMA	0	0	0	0	0	0	0	0	0	1	1
Node 0, zone DMA32	3	1	2	2	1	1	3	3	3	3	3
Node 0, zone Normal	2	1	1	0	3	2	1	1	1	1	1

Listing 14-25: Example output from `/proc/buddyinfo`

This very simply outputs free pages for each node, zone and order available in free lists (and thus excluding Per-CPU-Pages). Reading from left to right, we can observe counts for order-0 to order-10. So this example data translates to:-

Table 14-4: Example OOM free buddy pages

Zone	Order										
	0	1	2	3	4	5	6	7	8	9	10
DMA	0	0	0	0	0	0	0	0	1	1	3
DMA32	3	1	2	2	1	1	3	3	3	3	737
Normal	2	1	1	0	3	2	1	1	1	1	1206

See section 2.7 for more details on the buddy allocator.

If we have sufficient permissions, we can see a more detailed breakdown in `/proc/pagetypeinfo` (spacing adjusted slightly for readability):-

Page block order: 9

Pages per block: 512

Free pages count	per migrate type	at order	0	1	2	3	4	5	6	7	8	9
10												
Node 0, zone	DMA, type	Unmovable	0	0	0	0	0	0	0	0	1	0
0												
Node 0, zone	DMA, type	Movable	0	0	0	0	0	0	0	0	0	1
3												
Node 0, zone	DMA, type	Reclaimable	0	0	0	0	0	0	0	0	0	0
0												
Node 0, zone	DMA, type	HighAtomic	0	0	0	0	0	0	0	0	0	0
0												
Node 0, zone	DMA32, type	Unmovable	1	0	0	0	0	0	1	1	1	1
0												
Node 0, zone	DMA32, type	Movable	2	1	2	2	1	1	2	2	2	2
737												
Node 0, zone	DMA32, type	Reclaimable	0	0	0	0	0	0	0	0	0	0
0												
Node 0, zone	DMA32, type	HighAtomic	0	0	0	0	0	0	0	0	0	0
0												
Node 0, zone	Normal, type	Unmovable	1	0	0	0	1	0	1	0	0	1
0												
Node 0, zone	Normal, type	Movable	0	1	1	0	1	1	0	1	0	0
1206												
Node 0, zone	Normal, type	Reclaimable	1	0	0	0	1	1	0	0	1	0
0												
Node 0, zone	Normal, type	HighAtomic	0	0	0	0	0	0	0	0	0	0
0												
Number of blocks type			Unmovable	Movable	Reclaimable	HighAtomic						
Node 0, zone	DMA		1	7	0	0						
Node 0, zone	DMA32		2	1526	0	0						
Node 0, zone	Normal		22	2530	8	0						

Listing 14-26: Example output from /proc/pagetypeinfo

Here each individual set of free pages is broken down into migrate types, followed by a list of pageblock migrate type designations spanning both freed and allocated memory (recalling that pageblocks are typically order-9 (i.e. 2 MiB) blocks of contiguous physical memory which are divided into migrate type to reduce fragmentation (see section 2.5 for more details).

The free page output is determined in `pagetypeinfo_showfree_print()` and page block designation is determined in `pagetypeinfo_showblockcount_print()`.

14.5.2 A quick tour: Virtual memory

Userland virtual memory, by its nature, is tied to processes as each address space is entirely distinct from any other. Therefore the most useful means of examining virtual memory ranges is on a per-process basis.

Other than the data available in `/proc/$pid/status` and `stat/statm` variants, discussed in section 14.2, the most important per-process interfaces to examine are `/proc/$pid/maps`, `/proc/$pid/smaps` and `/proc/$pid/smmaps_rollup`.

Let's examine these interfaces for an example program (eliding less useful fields and adjusting whitespace):-

```
Name:    mupdf
...
VmPeak:   59488 kB
VmSize:   59488 kB
VmLck:     0 kB
VmPin:     0 kB
VmHWM:   19456 kB
VmRSS:   19412 kB
RssAnon:  10664 kB
RssFile:   7980 kB
RssShmem:   768 kB
```

Listing 14-27: Example output from `/proc/$pid/status`

This indicates that its current, mapped, virtual address space spans 59,488 KiB (`VmSize`), and reached a peak of 59,488 KiB (`VmPeak`). It maps 19,412 KiB (`VmRSS`) of resident memory, having reached a peak of 19,456 KiB (`VmHWM`) so far. Of the mapped memory, 10,664 KiB (`RssAnon`) is anonymous, 7,980 KiB (`RssFile`) file-backed (non-shmem) and 768 KiB (`RssShmem`) shmem (i.e. RAM-backed files).

Examining the `/proc/$pid/maps` of this process (skipping a number of shared library mappings for space):-

```
556cf266e000-556cf26b1000 r--p 00000000 103:05 4805731 /usr/bin/mupdf
556cf26b1000-556cf288f000 r-xp 00043000 103:05 4805731 /usr/bin/mupdf
556cf288f000-556cf2a0d000 r--p 00221000 103:05 4805731 /usr/bin/mupdf
556cf2a0d000-556cf2a1e000 r--p 0039e000 103:05 4805731 /usr/bin/mupdf
556cf2a1e000-556cf49c1000 rw-p 003af000 103:05 4805731 /usr/bin/mupdf
556cf49c1000-556cf49de000 rw-p 00000000 00:00 0
556cf6163000-556cf6b3e000 rw-p 00000000 00:00 0 [heap]
7f3591c3d000-7f3591c3f000 r--p 00000000 103:05 4771151 /usr/lib/libXrender.so
    .1.3.0
...
7f3591dcc000-7f3591dcf000 r--p 00000000 103:05 4733534 /usr/lib/libpcre2-8.so
    .0.11.2
...
7f3592000000-7f3592026000 r--p 00000000 103:05 4721961 /usr/lib/libc.so.6
...
7f3592240000-7f359224d000 rw-p 00000000 00:00 0
```

```

...
7f359224e000-7f3592250000 r--p 00000000 103:05 4766188      /usr/lib/libXdmcp.so
    .6.0.0
...
7f359228e9000-7f35928ea000 r--p 00000000 103:05 4721936      /usr/lib/ld-linux-x86
    -64.so.2
...
7ffe32c2e000-7ffe32c4f000 rw-p 00000000 00:00 0             [stack]
7ffe32de8000-7ffe32dec000 r--p 00000000 00:00 0             [vvar]
7ffe32dec000-7ffe32dee000 r-xp 00000000 00:00 0             [vdso]
fffffffff600000-fffffffff601000 --xp 00000000 00:00 0       [vsyscall]

```

Listing 14-28: Example output from `/proc/$pid/maps`

This shows virtual address ranges for mapped memory, both anonymous and file-backed, each describing a struct `vm_area_struct`.

After the address range protection flags are shown – `r` indicating read, `w` indicating write, `x` indicating executable and `p` or `s` indicating whether the mapping is private or shared, respectively.

If the mapping is file-backed, then the next columns shows the offset into the file, the major:minor device numbers for the device upon which the file is mounted and its inode number.

If a file-backed mapping, then the file path is shown in the final column. If anonymous, but named (i.e. possessing a non-NULL struct `vm_area_struct->anon_name`) then this is shown in square brackets.

In addition certain anonymous mappings are given names based on criteria:-

vDSO – If the VMA has no associated struct `mm_struct` object referenced in its struct `vm_area_struct->mm` field, this indicates that this is a kernel mapping, and the only way this can be the case is if this is a vDSO mapping, i.e. a virtual Dynamic Shared Object – this is memory shared between userland and the kernel to make certain system calls more performant.

Heap – If the mapped range starts at or below struct `mm_struct->brk` and ends at or above struct `mm_struct->start_brk`, then the mapping is designated as the heap.

Stack – if the mapped range starts at or below struct `mm_struct->start_stack` and ends at or above this value, then it is designated the process's stack (as determined by `is_stack()`).

This is distinct from mappings which have been designated as user-defined stacks by e.g. `MAP_GROWSDOWN` but is rather intended to indicate the process's overall stack.

Each VMA's output is implemented in `show_map_vma()`, which is also used by `/proc/$pid/smaps`.

This provides the basic details about mappings, but `/proc/$pid/smaps` provides per-VMA statistics in addition to this (eliding out of scope huge page and page size fields):-

```

556cf266e000-556cf26b1000 r--p 00000000 103:05 4805731          /usr/
    bin/mupdf
Size:                268 kB
...
Rss:                268 kB
Pss:                268 kB
Pss_Dirty:          0 kB
Shared_Clean:       0 kB
Shared_Dirty:       0 kB
Private_Clean:      268 kB
Private_Dirty:      0 kB
Referenced:         268 kB
Anonymous:          0 kB
LazyFree:           0 kB
...
Swap:               0 kB
SwapPss:            0 kB
Locked:             0 kB
...
VmFlags: rd mr mw me sd

```

Listing 14-29: Example `/proc/$pid/smaps` output

After showing a line with the same data shown in `/proc/$pid/maps` for this VMA, a number of statistics describing the range follow (each expressed in kilobytes):-

- **Size** – Shows the size of the virtual mapping, equal to `struct vm_area_struct->vm_end - struct vm_area_struct->vm_start`.
- **Rss** – Resident Set Size – The amount of memory which is resident, i.e. present in memory, whether file-backed or anonymous. This constitutes all memory in the range that resolves to a physical page of memory.
- **Pss** – Proportional Set Size – This is the sum of the memory occupied by all resident pages of memory, each page divided by the number of times it is mapped into a process.

Therefore, memory which is not shared will be accounted the same as Rss, but each page of shared memory will be divided by the *map count*. If a file is mapped via `MAP_PRIVATE`, then the portion of the mapping that has been copied into anonymous memory will be accounted the same as anonymous memory, i.e. not divided by map count, whereas the portion of the mapping which maps into the page cache *will* have each page scaled by each page's map count.

This can get complicated, as files can be mapped at different offsets meaning map counts may vary page-per-page, in addition to the `MAP_PRIVATE` case.

Considering a `MAP_PRIVATE` mapping spanning 10 pages of 4 KiB each, with half of its pages copied to private anonymous memory and half still

mapping the shared page cache pages for the underlying file, which has been mapped twice.

The `Rss` value will be 40 KiB, however `Pss` will be equal to 5 pages of 4 KiB anonymous memory, summing to 20 KiB, in addition to a further 20 KiB of memory shared between two mappings, summing to 10 KiB for a total of 30 KiB `Pss`.

This value is most useful when considering multiple mappings - by summing the `PSS` values for each process, the total sum will be equal to all of the mapped memory in the system.

Therefore, `PSS` can be seen as the amount of resident memory utilised by a `VMA`, assuming we account shared memory as equally divided between all mappings of that memory.

Note that memory which is due to be migrated and thus has a *migration entry* assigned to it is considered to have a map count of 1.

`PSS` values are updated in `smaps_page_accumulate()`, invoked from `smaps_account()`.

- `Pss_Dirty` – The Proportional Set Size value for all dirty pages (i.e. memory which has been changed but not yet written back to disk, if file-backed).
- `Shared_Clean` – The amount of resident memory which is shared and not dirty. Determined in `smaps_page_accumulate()`.
- `Shared_Dirty` – The amount of resident memory which is shared and dirty. Determined in `smaps_page_accumulate()`.
- `Private_Clean` – The amount of resident memory which is private (i.e. possesses a map count of 1) and not dirty. Determined in `smaps_page_accumulate()`.

This is only meaningful for `MAP_PRIVATE` mappings of files. For anonymous memory, simply reading newly mapped memory will yield a mapping to the zero page which results in no increase in residential memory usage.

However, reading from a `MAP_PRIVATE` mapping of a file will result in unmodified memory which does occupy residential memory being Copy-on-Written. As soon as that is written to, it will be accounted as `Private_Dirty`.

- `Private_Dirty` – The amount of resident memory which is private (i.e. possesses a map count of 1) and dirty. Determined in `smaps_page_accumulate()`. Private memory can never be cleaned, as there is no writeback of this memory.
- `Referenced` – The number of resident memory that has either been determined to be accessed in the *idle page tracking* framework (out of scope for the book), has the hardware-determined `_PAGE_ACCESSED` PTE flag set checked by `pte_young()` indicating it has been accessed recently, or is marked referenced (i.e. its `struct page` has the `PG_referenced` flag set, indicating that reclaim has determined the page to have been accessed recently (reducing the chance of reclaim)).

If this memory is considered for reclaim, the existence of the sticky PTE bit will guarantee this memory will be marked referenced (see the reclaim chapter for more details on the reclaim algorithm, especially section 11.2).

Therefore, this value can be taken to be the sum of all memory which has been recently accessed and thus gives an insight into how likely it is for the range to be reclaimed and how recently it has been used.

This value is calculated in `smaps_account()`.

- **Anonymous** – The amount of anonymous, resident memory, as determined ultimately through `folio_test_anon()`, which checks to see whether `struct page->mapping` has the `PAGE_MAPPING_ANON` flag set.
This value is calculated in `smaps_account()`.
- **LazyFree** – The total amount of ‘lazy’ freed memory, as determined by it being anonymous but not swap-backed nor dirty.
This is memory that has been indicated that it should be freed via the `madvise()` system call specifying the `MADV_FREE` flag.
This causes the memory to be prioritised for consideration by reclaim, however if they are written to before reclaim occurs they will no longer be freed.
This value is calculated in `smaps_account()`.
- **Swap** – The amount of memory in the range which is not resident but rather swapped out.
This is determined in `smaps_pte_entry()` except for `shmem` mappings (e.g. `tmpfs`) which requires a more complicated calculation as performed in `shmem_swap_usage()` and ultimately `shmem_partial_swap_usage()`.
- **SwapPss** – The amount of memory in the range which is swapped out, as a proportional value using the same algorithm as described above for `Pss`.
This is determined in `smaps_pte_entry()`.
- **Locked** – The Proportional Set Size for all memory which has been `mlock()`’d (see section 8.2.1 for more on the use of this feature).
Determined in `smaps_page_accumulate()`.
- **VmFlags** – A list of mnemonics which indicate which of the `struct vm_area_struct` (VMA) flags have been set as described below.
Determined in `show_smap_vma_flags()`.

Examining the mapping between VMA flags and `VmFlags` mnemonics shown in the `VmFlags` field (eliding `arm64`-specific flags):-

Table 14-5: VmFlags mappings

Mnemonic	VMA flag
rd	VM_READ
wr	VM_WRITE
ex	VM_EXEC
sh	VM_SHARED
mr	VM_MAYREAD
mw	VM_MAYWRITE
me	VM_MAYEXEC
ms	VM_MAYSHARE
gd	VM_GROWSDOWN
pf	VM_PFNMAP
lo	VM_LOCKED
io	VM_IO
sr	VM_SEQ_READ
rr	VM_RANDOM_READ
dc	VM_DONTCOPY
de	VM_DONTEXPAND
ac	VM_ACCOUNT
nr	VM_NORESERVE
ht	VM_HUGETLB
sf	VM_SYNC
ar	VM_ARCH_1
wf	VM_WIPEONFORK
dd	VM_DONTDUMP
sd	VM_SOFTDIRTY
mm	VM_MIXEDMAP
hg	VM_HUGEPAGE
nh	VM_NOHUGEPAGE
mg	VM_MERGEABLE
um	VM_UFFD_MISSING
uw	VM_UFFD_WP
ui	VM_UFFD_MINOR

See section 4.4.1 for a detailed description of each of these flags.

Each of these are determined and output in `show_smmap()`, with a large number of the fields calculated in `smmap_gather_stats()`, which walks the page tables of the range using `smmap_pte_range()` for non-huge mappings which calls `smmap_pte_entry()` and ultimately `smmap_account()`.

Since these values are obtained from walking the page tables with the process address space lock held, they are not subject to the same scalability inaccuracies as other memory counters.

Often the distinction between individual VMA statistics is not important and all we wish to determine are the *overall* statistics for the process. Therefore for the convenience linux exposes the `/proc/$pid/smmap_rollup` interface (previously referenced in section 14.2):-

```
556cf266e000-7ffe32dee000 ---p 00000000 00:00 0 [
    rollup]
Rss:                19916 kB
```

```

Pss:                14553 kB
Pss_Dirty:          11360 kB
Pss_Anon:           10976 kB
Pss_File:           3193 kB
Pss_Shmem:           384 kB
Shared_Clean:        5204 kB
Shared_Dirty:         768 kB
Private_Clean:       2968 kB
Private_Dirty:       10976 kB
Referenced:          19916 kB
Anonymous:           10976 kB
LazyFree:             0 kB
AnonHugePages:        8192 kB
ShmemPmdMapped:       0 kB
FilePmdMapped:        0 kB
Shared_Hugetlb:       0 kB
Private_Hugetlb:      0 kB
Swap:                 0 kB
SwapPss:              0 kB
Locked:               0 kB

```

Listing 14-30: Example `/proc/$pid/smmaps_rollup` output

This aggregates all of the `/proc/$pid/smmaps` values across the process, less the fields determined in `show_smmap()` (i.e. `KernelPageSize`, `MMUPageSize`, `THPeligible` and `ProtectionKey`). It additionally adds three additional values – `Pss_Anon`, `Pss_File` and `Pss_Shmem`. The output is generated in `show_smmaps_rollup()`.

The `Pss_Anon` value indicates the Proportional Set Size (PSS) of anonymous memory. This comprises of ordinary anonymous memory which will be equivalent to the RSS value, as well as file-backed memory which has been mapped via `MAP_PRIVATE` but has not yet been written to, which is therefore anonymous but maps shared page cache data, scaled by the number of times mapped.

The `Pss_File` value indicates the aggregate PSS of non-shmem (RAM-backed memory such as tmpfs) file-backed memory mapped using `MAP_SHARED`.

Finally `Pss_Shmem` indicates the aggregate PSS of shmem file-backed memory mapped using `MAP_SHARED`. The sum of `Pss_Anon`, `Pss_File` and `Pss_Shmem` is equal to `Pss`.

14.5.3 A quick tour: Page table introspection

It is possible, using the `/proc/$pid/pagemap`, `/proc/kpageflags` and `/proc/kpagecount` interfaces, to determine a great deal of data about page table entries for virtual memory mappings. This is documented in *pagemap*.

If we offset into the `pagemap` using virtual page offset, we can then use the contents of this entry (which, if present, will contain the PFN) to obtain data from `kpagecount` and `kpageflags`.

Let's examine a program that interacts with these interfaces:-

```

1 #include <fcntl.h>
2 #include <stdint.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/mman.h>
6 #include <unistd.h>
7 #include <linux/kernel-page-flags.h>
8
9 #define PAGEMAP_SOFTDIRTY (1UL << 55)
10 #define PAGEMAP_EXCLUSIVE (1UL << 56)
11 #define PAGEMAP_FILE      (1UL << 61)
12 #define PAGEMAP_SWAPPED   (1UL << 62)
13 #define PAGEMAP_PRESENT   (1UL << 63)
14
15 #define PAGEMAP_SWAP_OFFSET_SHIFT (5)
16 #define PAGEMAP_SWAP_TYPE_MASK ((1UL << 5) - 1)
17 /* 'Bits 0-54 page frame number (PFN) if present */
18 #define PAGEMAP_PFN_MASK ((1UL << 55) - 1)
19
20 static uint64_t read_u64(const char *path, uint64_t offset)
21 {
22     uint64_t ret;
23     int fd = open(path, O_RDWR);
24
25     if (fd < 0) {
26         perror("open");
27         exit(1);
28     }
29
30     if (lseek(fd, offset, SEEK_SET) != offset) {
31         perror("lseek");
32         exit(1);
33     }
34
35     if (read(fd, &ret, sizeof(ret)) != sizeof(ret)) {
36         perror("read");
37         exit(1);
38     }
39
40     return ret;
41 }

```

Listing 14-31: Example page table introspection (header)

We start by establishing some constants based on the documented values within the `/proc/$pid/pagemap` fields, and implement a simple function for re-

trieving a 64-bit value from a specific offset within a file which we will reuse for each of the interfaces.

```

43 static uint64_t read_pagemap(const void *ptr)
44 {
45     const uint64_t virt_page_num = (uint64_t)ptr / getpagesize();
46     /* There is 'one 64-bit value for each virtual page'. */
47     const uint64_t offset = virt_page_num * sizeof(uint64_t);
48
49     return read_u64("/proc/self/pagemap", offset);
50 }
51
52 static uint64_t read_kpageflags(uint64_t pfn)
53 {
54     return read_u64("/proc/kpageflags", pfn * sizeof(uint64_t));
55 }
56
57 uint64_t read_mapcount(uint64_t pfn)
58 {
59     return read_u64("/proc/kpagecount", pfn * sizeof(uint64_t));
60 }

```

Listing 14-32: Example page table introspection (interface readers)

Here we utilise the `read_u64()` helper function to read from each of the interfaces at the specified locations.

After this, we implement functions to read through each of these values and describe the obtained data:-

```

62 static void describe_swapped(uint64_t pagemap)
63 {
64     /* Swap type and offset encoded in same location as PFN would be. */
65     const uint64_t type = pagemap & PAGEMAP_SWAP_TYPE_MASK;
66     const uint64_t offset =
67         (pagemap & PAGEMAP_PFN_MASK) >> PAGEMAP_SWAP_OFFSET_SHIFT;
68
69     printf("swapped: type=%lu, offset=%lu\n", type, offset);
70 }
71
72 static void describe_kpageflags(uint64_t pfn)
73 {
74     uint64_t kpageflags = read_kpageflags(pfn);
75
76     /*
77      * There are a huge number of flags declared in
78      * uapi/include/linux/kernel-page-flags.h, we examine a subset.
79      */
80
81     #define PRINT_FLAGS(flag) \

```

```

82         if (kpageflags & (1UL << KPF_##flag)) \
83             printf(" [" #flag "]");
84
85         PRINT_FLAGS(ACTIVE);
86         PRINT_FLAGS(DIRTY);
87         PRINT_FLAGS(LOCKED);
88         PRINT_FLAGS(LRU);
89         PRINT_FLAGS(MMAP);
90         PRINT_FLAGS(NOPAGE);
91         PRINT_FLAGS(REFERENCED);
92         PRINT_FLAGS(SWAPBACKED);
93         PRINT_FLAGS(UPTODATE);
94         PRINT_FLAGS(WRITEBACK);
95
96 #undef PRINT_FLAGS
97 }
98
99 static void describe_addr(const void *ptr)
100 {
101     uint64_t pfn, mapcount;
102     const uint64_t pagemap = read_pagemap(ptr);
103
104     printf("%p: ", ptr);
105
106     if (!(pagemap & PAGEMAP_PRESENT)) {
107         if (pagemap & PAGEMAP_SWAPPED) {
108             describe_swapped(pagemap);
109             return;
110         }
111
112         printf("<not present>\n");
113         return;
114     }
115
116     printf("%s: ", (pagemap & PAGEMAP_FILE) ? "file" : "anon");
117
118     pfn = pagemap & PAGEMAP_PFN_MASK;
119     printf("pfn=%lu", pfn);
120
121     if (pagemap & PAGEMAP_SOFTDIRTY)
122         printf(" [SD]");
123
124     if (pagemap & PAGEMAP_EXCLUSIVE)
125         printf(" [EX]");
126
127     mapcount = read_mapcount(pfn);
128     printf(" mapped=%lu", mapcount);

```

```

129
130     describe_kpageflags(pfn);
131
132     printf("\n");
133 }

```

Listing 14-33: Example page table introspection (describers)

We observe that if the page is not present, then it might be swapped, and if not swapped then it simply isn't present at all, before extracting various flags and utilising the flags declared in `include/uapi/linux/kernel-page-flags.h` (note that there are additional, less stable, flags available in `include/linux/kernel-page-flags.h`) to obtain struct page-specific flags.

These flags are sometimes directly page flags, and sometimes a pseudo-flag determined from data relating to that page. These flag values are prefixed by `KPF_` and specified in `fs/proc/page.c`.

Finally, examining an example program which maps anonymous memory, then a file, then writes to file observing the output at each instance:-

```

135 int main(void)
136 {
137     int fd;
138     char *ptr;
139     const long page_size = sysconf(_SC_PAGESIZE);
140
141     ptr = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
142               MAP_ANON | MAP_PRIVATE, -1, 0);
143     if (ptr == MAP_FAILED) {
144         perror("mmap");
145         return EXIT_FAILURE;
146     }
147
148     /* Won't be present yet. */
149     describe_addr(ptr);
150     ptr[0] = 'x';
151     /* Anonymous page description. */
152     describe_addr(ptr);
153
154     fd = open("test.txt", O_RDWR);
155     if (fd < 0) {
156         perror("open");
157         return EXIT_FAILURE;
158     }
159
160     ptr = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
161               MAP_SHARED, fd, 0);
162     if (ptr == MAP_FAILED) {
163         perror("mmap");

```



```

164         return EXIT_FAILURE;
165     }
166
167     close(fd);
168
169     /* Won't be present yet. */
170     describe_addr(ptr);
171     printf("test contents: %s", ptr);
172     /* Now we have read from it, will be present. */
173     describe_addr(ptr);
174     ptr[0] = 'x';
175     /* Now we have updated, should see as dirty. */
176     describe_addr(ptr);
177
178     return EXIT_SUCCESS;
179 }

```

Listing 14-34: Example page table introspection (main)

14.6 Memory tunables

There are a number of VM tunables, located in `/proc/sys/vm` or accessible via the `sysctl` CLI specified via `vm.<tunable name>`.

The `procfs` documentation on these is extensive (see also the *sysctl kernel documentation*), so we will examine a core subset of these in detail:-

- `admin_reserve_kbytes` – `sysctl_admin_reserve_kbytes` – The amount of memory which should be reserved for administrators (i.e. those with the capability of `CAP_SYS_ADMIN`) in `OVERCOMMIT_NEVER` mode. This defaults to the smaller of 8 MiB, and can never exceed 8 MiB. and $\frac{1}{32}$ rd of initial free pages, as determined in `init_admin_reserve()`.
This is checked in `__vm_enough_memory()`, where if the user does not possess the system administration capability, this is subtracted from the permitted allocation size.
In other overcommit modes, this has no effect. See section 4.1 for more details on overcommit modes.
- `user_reserve_kbytes` – Similar to the `sysctl_admin_reserve_kbytes` tunable, this is designed to customise the behaviour of the system in `OVERCOMMIT_NEVER` overcommit mode (see section 4.1 for a detailed examination of how overcommit functions in the kernel).
This value specifies the amount of memory to reserve to prevent any one user from causing the system to grind to a halt.
This value is specified in the `sysctl_user_reserve_kbytes` global value and initially set in `init_user_reserve()` which sets it to the minimum of either $\frac{1}{32}$ rd of initial free pages or 128 MiB.
- `compact_memory`, `compact_unevictable_allowed`, `compaction_proactiveness` – The first, when written to, triggers the `sysctl_compaction_handler()`

function and the state of the latter two tunables are stored in `sysctl_compact_unevictable_allowed` and `sysctl_compaction_proactiveness` respectively.

Writing to `compact_memory` will trigger compaction immediately, invoking `compact_nodes()` to compact memory in all online nodes within the system.

the `compact_unevictable_allowed` tunable determines whether unevictable memory can be compacted, with it permitted by default for non-realtime kernels.

The `compaction_proactiveness` tunable determines how *proactive* compaction is in compacting memory. When the `kcompactd` kernel thread is running. This is expressed in a percentage value, defaulting to 20%. This value impacts compaction via the `should_proactive_compact_node()`. If it is set to 0%, then no compaction will occur at all. See section ?? for a detailed explanation as to how this impacts compaction – broadly speaking the higher this value, the more proactive the kernel is about compacting pages.

When the `compaction_proactiveness` tunable is changed, `compaction_proactiveness_sysctl_handler()` is invoked, which wakes up the `kcompactd` kernel process for each node to apply the change.

See the compaction chapter for a detailed discussion of compaction as a whole.

- `dirty_background_bytes/dirty_background_ratio`, `dirty_bytes/dirty_ratio` – These values dictate how writeback behaves (see section ?? for a detailed discussion of writeback), i.e. at what point modified data gets written back to disk.

The bytes and ratio versions of each are mutually exclusive – when one is set, the other is zeroed. By default the bytes variants are zero, the ratio values used.

The background versions of these tunables determine at what point the per-node background writeback process starts running, writing back data to disk. If bytes are specified, it is simply the number of bytes of dirty data which can be written back to disk.

If a ratio is specified, the value is expressed as a percentage of dirty memory as a proportion of *dirtyable* memory, i.e. memory that could be dirtied (comprising existing file pages and free pages, calculated in `global_dirtyable_memory()`).

The non-background values determine the point at which writing to clean pages will cause the writing process to sleep until sufficient writeback is performed. Again, see section ?? for a detailed discussion of the writeback algorithm for both background and sleep-inducing writeback. The number of dirty pages is equal to the `NR_FILE_DIRTY` statistic (used in `global_node_page_state()`), i.e. the `nr_dirty` value shown in `/proc/zoneinfo` or, across the system, the Dirty value shown in `/proc/meminfo`.

The state of `dirty_background_bytes` is stored in the `dirty_background_bytes` global value, `dirty_background_ratio` is stored in `dirty_background_ratio`,

`dirty_bytes` is stored in `vm_dirty_bytes` and `dirty_ratio` is stored in `vm_dirty_ratio`.

Each of these values, when modified, are processed by the `dirty_background_bytes_handler()`, `dirty_background_ratio_handler()`, `dirty_bytes_handler()` and `dirty_ratio_handler()` functions respectively.

- `dirty_expire_centisecs`, `dirty_writeback_centisecs` – While the above dirty tunables determine the memory conditions under which writeback begins, it does not determine at what *time* this begins.

It would be rather silly to immediately begin writing back dirtied pages the moment they are altered, as users will often perform multiple changes to a page before they are finished.

Equally, it would be ludicrous for the kernel to have to wait until it reaches the point at which there are so many dirty pages that writeback must occur (to avoid mass data loss), before writing out data back to disk.

Therefore, we add some delay before considering dirtied pages for *periodic* writeback (`dirty_expire_centisecs`) and wake up the writeback kernel threads every so often to check to see what needs writing back (`dirty_writeback_centisecs`). Note that young dirtied memory might still be considered for writeback when dirty counts are high.

Both values are expressed in centiseconds, i.e. one hundredths of a second, therefore a value of 100 centiseconds is equivalent to one second, and a value of 1 centisecond is equivalent to 0.01 seconds.

The `dirty_expire_centisecs` value defaults to 30,000 (i.e. 30 seconds), and `dirty_writeback_centisecs` defaults to 5,000 (i.e. 5 seconds).

`dirty_expire_centisecs` is stored in the global `dirty_expire_interval` value, and `dirty_writeback_centisecs` is stored in the global `dirty_writeback_interval` value, which triggers `dirty_writeback_centisecs_handler()` when changed.

See section ?? for a detailed exploration of the linux writeback algorithm.

- `drop_caches` – This interface permits a user to cause page cache and slab pages to be dropped. There is no good reason to do this in ordinary operation, however it can be useful to do so for the purposes of testing performance in the presence or absence of in-memory caching of files. Outputting 1 to this interfaces drops all entries in the page cache, 2 drops reclaimable slab pages, and 3 performs both.

Adding 4 to these values (i.e. setting bit 2) will cause the operation to only output to the kernel log once.

Note that this does not drop caches for dirtied data at any stage, therefore it will not cause data loss, but equally, when used for the purposes of performance testing, it is important to ensure dirtied pages have been written back to disk otherwise these will remain in the page cache.

We examine this interface and how it functions within the kernel in detail in section 9.9.4. The principle function which invokes this action is `drop_caches_sysctl_handler()`.

- `lowmem_reserve_ratio` – Determines the low memory reserves per-zone for allocations which are falling through to the zone below, expressed as fractions of memory of zones above that *could* have been used to service the allocation.

This is something of a tricky topic, so see section 2.4.1 for a detailed description of how this functionality works and how the tunable interacts with it.

- `max_map_count` – Determines the maximum number of VMAs that a process may map at any one time.

This value is specified in the `sysctl_max_map_count` global value, and defaults to `DEFAULT_MAX_MAP_COUNT`, which is equal to 65,536 less `MAPCOUNT_ELF_CORE_MARGIN`.

This limit exists as the ELF file format creates a section per VMA in core dumps, and the number of sections is stored as an unsigned short (i.e. a 16-bit value). There are additionally some informational sections added, hence the additional margin.

This value is compared against `struct mm_struct->map_count` which tracks the number of mapped VMAs for a process address space. This check is performed in a number of places in the kernel, most prominently in `do_mmap()`.

- `min_free_kbytes` – Specifies the minimum amount of free memory which must be maintained on the system at all times, which is used to calculate zone minimum watermarks.

This is important, as it is critical to maintain *some* memory at all times, otherwise the kernel or critical userland processes might not be able to operate correctly resulting in a hung or useless system.

The default value is determined by `calculate_min_free_kbytes()`, which attempts to set this value according to available system memory, calculating this to be equal to $4\sqrt{p}$ where p is the amount of memory in kilobytes which exceeds the high water mark if each zone. On my machine this yields a 67,584 KiB value.

The minimum watermark for each zone is determined by scaling this value against the proportion of total memory each zone manages.

This is stored in the `min_free_kbytes` global value. When the tunable is changed, it triggers `min_free_kbytes_sysctl_handler()` which triggers `setup_per_zone_wmarks()` in turn to update watermarks.

- `mmap_min_addr` – This specifies the minimum address which can be mapped in a process. This is important, as if for instance an allocation was permitted at virtual address zero, then null handling would become problematic.

This value is stored in `mmap_min_addr`, however when input it is kept in `dac_mmap_min_addr`. If the `CONFIG_LSM_MMMap_MIN_ADDR` value is set, then this will set a hard floor to what value can be specified here. The update is performed in `update_mmap_min_addr()`, invoked from `mmap_min_addr_handler()`.

On my 64-bit x86-64 system this defaults to 64 KiB.

Update is only permitted if the user has the `CAP_SYS_RAWIO` capability.

- `oom_dump_tasks`, `oom_kill_allocating_task` – If the `oom_dump_tasks` tunable is set to a non-zero value, then all tasks are dumped on invocation of the out of memory killer, otherwise this output is suppressed (can be useful for truly huge systems). Defaults to 1.

If `oom_kill_allocating_task` is specified, then the task which caused the allocation which triggered the OOM killer is killed instead of the one which is determined to consume the most memory.

This is not usually a very good idea as the task that happens to trigger the OOM is not in way likely to be the one who caused it, but rather an innocent bystander. This of course defaults to 0.

- `overcommit_memory`, `overcommit_kbytes/overcommit_ratio` – Specifies the overcommit mode in `overcommit_memory` – where 0 indicates `OVERCOMMIT_GUESS`, 1 indicates `OVERCOMMIT_ALWAYS`, and 2 indicates `OVERCOMMIT_NEVER`.

In `OVERCOMMIT_GUESS` mode, which is the default, allocations are permitted as long as they do not exceed the total amount of installed RAM and swap (specifying `MAP_NORESERVE` bypasses even this check).

In `OVERCOMMIT_ALWAYS` mode, there are no checks whatsoever and mappings of any valid range are permitted, and in `OVERCOMMIT_NEVER` mode, accountable committed memory is tracked and compared to the limit. The limit is determined to be total swap plus either `overcommit_bytes` of memory or `overcommit_ratio` of all installed RAM, which is expressed as a percentage (defaulting to 50%).

The limit can be observed in `/proc/meminfo` under `CommitLimit`, and the total committed count as `Committed_AS`.

See section 4.1 for a detailed discussion of overcommit.

Stored in the `sysctl_overcommit_memory`, `sysctl_overcommit_kbytes` and `sysctl_overcommit_ratio` global values.

- `page-cluster` – Determines the number of pages which are retrieved from the swap at one time, measured in consecutive pages within the swap. The actual value specifies a page order, i.e. a power-of-two number of base pages, which for machines with anything but the tiniest amount of memory defaults to 3.

This is stored in the global `page_cluster` variable.

- `panic_on_oom` – If set to 0, then the OOM killer performs its usual job of killing the most egregious memory hog (see the OOM killer chapter for details on how it does this). If set to 1 then it will either panic when memory has been exhausted in the system, or proceed as usual if constrained by either memory policy or cgroup factors (both out of scope for the book).

Finally, if set to 2, linux will always panic in an out of memory condition. This value of course defaults to 0.

This is stored in the global `sysctl_panic_on_oom` variable.

- `percpu_pagelist_high_fraction` – This determines the fraction of pages in each zone which can be stored on Per-CPU-Pages (or PCPs), which are caches maintained on each CPU to prevent the need for a lock for

freeing and allocating pages (see section 2.7.3 for more details on how this mechanism works).

The value itself specifies the denominator of the fraction of pages in each zone which will determine the high value for each PCP, i.e. the maximum number of pages which reside there.

If set to zero, which is its default, the kernel will instead determine this value heuristically based on the low watermark for the zone and the number of CPUs in the system. Otherwise if it is specified, then this value simply divides zone managed pages by the denominator specified in the tunable and divides again by the number of CPUs.

This is stored in the global `percpu_pagelist_high_fraction` variable. The calculation of the high count for PCPs is determined in `zone_highsize()`. When the value is changed, the `percpu_pagelist_high_fraction_sysctl_handler()` function is called, which performs sanity checks and ensures the value does not cause undue imbalance.

- `stat_interval/stat_refresh` – VM statistics are stored in per-PCU caches in order to avoid the updating of node and zone statistics causing performance impact by needing to acquire a lock. The statistics are then flushed back to the global state at a regular interval.

The `stat_interval` tunable specifies this interval, expressed in seconds. A statistics refresh can be triggered manually by writing to `stat_refresh`.

The `stat_interval` tunable value is stored in the global `sysctl_stat_interval` value, converted to kernel jiffies (a measure of time within the kernel).

Invoking a manual statistics refresh causes `vmstat_refresh()` to be called.

- `swappiness` – This value expresses in rather heuristic terms, the *relative* cost of swapping memory out vs. dropping pages from the page cache on reclaim.

The value ranges from 0 to 200, with lower values indicating swapping is more expensive, and higher values indicating it is less so, with 100 indicating that both are of equal performance cost.

This is a tunable in the true sense of the word, as it rather requires a value to be empirically determined. By default this is set to a value of 60, sensibly indicating that swapping out/in is considerably more expensive than dropping references to the page cache.

This value is stored in the global `vm_swappiness` variable. It is utilised by `get_scan_count()` during reclaim to determine how to decide between the two.

See the chapter on reclaim for a detailed examination of how this impacts the algorithm.

- `watermark_boost_factor`, `watermark_scale_factor` – When pageblocks have different migrate types mixed into them, this can cause fragmentation (see section 2.5 for a detailed explanation of migrate types). In order to counteract the impact of this, a boost factor can be applied to the watermarks in a zone when this kind of mixing has occurred.

The boost factor determines the boost to be a factor of the high watermark of that zone, which is multiplied by 10,000, and defaults to 15,000 (meaning that the default boost is 1.5x the high water mark of a zone). The `watermark_scale_factor` determines the *distances* between watermarks in each zone, as determined in `__setup_per_zone_wmarks()`, multiplied by 10,000. The default value is 10, which implies a distance between minimum, low, and high watermarks as 0.1% of the available memory in the node. The maximum permitted value is 3,000 i.e. 30% of available memory.

The `watermark_boost_factor` tunable state is stored in the `watermark_boost_factor` global, and `watermark_scale_factor` is stored in `watermark_scale_factor`.

The boost is applied in `boost_watermark()`.

See section 2.4 for more details on nodes, zones and watermarks.

- `zone_reclaim_mode` – This tunable determines whether to attempt reclaim when a zone drops below the required watermark level for the allocation, rather than the default of simply trying the next available zone. This tunable also determines how to go about reclaim in this situation. Amusingly, the global value which stores this value is `node_reclaim_mode`, and can be set to the *combinable* flags `RECLAIM_ZONE` (1) to try to reclaim when a zone runs out of memory, `RECLAIM_WRITE` (2) to write out dirty pages during reclaim and `RECLAIM_UNMAP` (4) to unmap, i.e. swap out pages during reclaim.

The check for this tunable is performed in `get_page_from_freelist()`, where `node_reclaim_enabled()` determines whether this is enabled at all.

14.7 Sharing memory

The use of virtual memory means linux can map the same physical memory from multiple processes simultaneously. This can be used to explicitly share memory between processes, often used as an efficient means of Inter-Process Communication (IPC).

Memory can be shared in a number of different ways, all fundamentally functioning in the same way – a file-backed mapping (usually a `shmem` mapping, i.e. RAM-backed, see section ?? for more details) which each process has obtained access to simultaneously.

Shared memory can be used for Inter-Process Communication (IPC), however its important to note that simply having the same memory mapped between two processes isn't sufficient to establish usable IPC – synchronisation between the sharing processes is still required by some means.

14.7.1 System V shared memory

The system V shared memory interface is the oldest existing shared memory interface available in linux and provides a lot of tooling around the mechanism to permit the construction of robust IPC.

Examining a simple server/client architecture, which simply establishes a shared memory mapping and has each write to it:-

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/shm.h>
7 #include <unistd.h>
8
9 #define KEY (1234)
10 #define SIZE (4096)
11 #define NUM_LOOPS (10)
12
13 int main(void)
14 {
15     int i, shmid;
16     char *shm;
17
18     shmid = shmget(KEY, SIZE, IPC_CREAT | 0666);
19     if (shmid < 0) {
20         perror("shmget");
21         return EXIT_FAILURE;
22     }
23
24     shm = shmat(shmid, NULL, 0);
25     if (shm == (void *)-1) {
26         perror("shmat");
27         return EXIT_FAILURE;
28     }
29
30     memset(shm, 0, SIZE);
31
32     for (i = 0; i < NUM_LOOPS; i++) {
33         shm[0] = 'a' + i;
34         printf("server: %s\n", shm);
35         sleep(1);
36     }
37
38     return EXIT_SUCCESS;
39 }

```

Listing 14-35: Example System V shared memory server

A key is established using `shmget()`, specifying `IPC_CREAT` to create a new shared memory ‘segment’. Specifying `IPC_CREAT` indicates that the segment should be created rather than simply accessed. The permissions are set for this region in the same fashion as those that are specified by `chmod()`.

A pointer to the region is then obtained via `shmat()` which *attaches* to it, i.e. mapping it into the process. The second parameter specifies the desired address or, as here, `NULL` to indicate that it doesn't matter which address this is mapped at and the final specifies flags none of which we require here.

At this point we have created a shared memory mapping which can be accessed by specifying the same key in the client:-

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/ipc.h>
5 #include <sys/shm.h>
6 #include <unistd.h>
7
8 #define KEY (1234)
9 #define SIZE (4096)
10 #define NUM_LOOPS (10)
11
12 int main(void)
13 {
14     int i, shmid;
15     char *shm;
16
17     shmid = shmget(KEY, SIZE, 0666);
18     if (shmid < 0) {
19         perror("shmget");
20         return EXIT_FAILURE;
21     }
22
23     shm = shmat(shmid, NULL, 0);
24     if (shm == (void *)-1) {
25         perror("shmat");
26         return EXIT_FAILURE;
27     }
28
29     for (i = 0; i < NUM_LOOPS; i++) {
30         shm[1] = 'a' + i;
31         printf("client: %s\n", shm);
32         sleep(1);
33     }
34
35     return EXIT_SUCCESS;
36 }
```

Listing 14-36: Example System V shared memory client

This largely reflects the server, only without specifying `IPC_CREAT`, to indicate that we are not creating the shared memory segment, only accessing it.

Note that this is a very brief overview of how the mechanism works, and in a production version it'd be sensible to utilise the semaphores and other machinery the interface exposes to synchronise access to the shared data.

This approach heavily abstracts the underlying mechanisms, however the overall approach remains the same as any other shared memory approach, i.e. a shared file is established which multiple processes map in order to access the shared memory.

This is implemented in the kernel specifically via the system call `shmget()` which ultimately invokes `newseg()` which uses `shmem_kernel_file_setup()` to generate an unlinked tmpfs file (i.e. one that is present in tmpfs but has no file entry in any directory) which acts as the shared file.

14.7.2 POSIX shared memory

POSIX shared memory is an updated means of sharing memory available in linux, intended essentially to be the successor to the System V method. Again, many different building blocks required for using shared memory as an IPC mechanism are provided in addition to simply sharing memory between processes.

This resembles the system V interface, only utilising a file descriptor which is `mmap()`'d into shared processes, and a `shmem` (i.e. RAM-backed) file is established at `/dev/shm/<name>` where `name` is specified in `shm_open()`:-

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/mman.h>
6 #include <sys/shm.h>
7 #include <unistd.h>
8
9 #define SHM_NAME "test"
10 #define SIZE (4096)
11 #define NUM_LOOPS (10)
12
13 int main(void)
14 {
15     int i, fd;
16     char *ptr;
17
18     fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
19     if (fd < 0) {
20         perror("shm_open");
21         return EXIT_FAILURE;
22     }
23
24     if (ftruncate(fd, SIZE)) {
25         perror("ftruncate");

```

```

26         return EXIT_FAILURE;
27     }
28
29     ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
30     if (ptr == MAP_FAILED) {
31         perror("mmap");
32         return EXIT_FAILURE;
33     }
34
35     memset(ptr, 0, SIZE);
36
37     for (i = 0; i < NUM_LOOPS; i++) {
38         ptr[0] = 'a' + i;
39         printf("server: %s\n", ptr);
40         sleep(1);
41     }
42
43     /* Wait until client is done. */
44     while (ptr[1] != 'a' + (NUM_LOOPS - 1)) {
45         sleep(1);
46     }
47
48     if (shm_unlink(SHM_NAME)) {
49         perror("shm_unlink");
50         return EXIT_FAILURE;
51     }
52
53     return EXIT_SUCCESS;
54 }

```

Listing 14-37: Example POSIX shared memory server

Note that, similar to system V shared memory, we must specify that this function creates the file, only we use the generic `O_CREAT` as used in `open()`, to indicate that the file should be created if it does not already exist. The permissions of the file are specified in the final parameter.

The size of the shared file is set via `ftruncate()`, and can then be used as a shared memory mapping. The shared file will stay present in the `/dev/shm` directory until it is removed via `shm_unlink()`.

Examining the client:-

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/mman.h>
5 #include <sys/shm.h>
6 #include <unistd.h>
7
8 #define SHM_NAME "test"

```

```

9 #define SIZE (4096)
10 #define NUM_LOOPS (10)
11
12 int main(void)
13 {
14     int i, fd;
15     char *ptr;
16
17     fd = shm_open(SHM_NAME, O_RDWR, 0666);
18     if (fd < 0) {
19         perror("shm_open");
20         return EXIT_FAILURE;
21     }
22
23     ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
24     if (ptr == MAP_FAILED) {
25         perror("mmap");
26         return EXIT_FAILURE;
27     }
28
29     for (i = 0; i < NUM_LOOPS; i++) {
30         ptr[1] = 'a' + i;
31         printf("client: %s\n", ptr);
32         sleep(1);
33     }
34
35     return EXIT_SUCCESS;
36 }

```

Listing 14-38: Example POSIX shared memory client

This largely mirrors the server, only with `O_CREAT` left unspecified. This implementation overall is less of an abstraction over the filesystem, rather more obviously establishing a file which is simply `mmap()`'d by processes sharing the memory.

The implementation relies upon `/dev/shm` being mapped as a `tmpfs` file system which is required by the standard.

14.7.3 Anonymous shared memory across forked processes

When a process forks a child process, the child process obtains access to the memory mapped by the parent on a copy-on-write basis, meaning that no memory is shared by default.

However, through the use of an anonymous shared mapping via `mmap()`, memory can easily be shared between parent and child:-

```

1 #include <stdio.h>
2 #include <stdlib.h>

```

```

3 #include <unistd.h>
4 #include <sys/mman.h>
5 #include <sys/wait.h>
6
7 #define NUM_LOOPS (10)
8
9 int main(void)
10 {
11     int i;
12     pid_t pid;
13     const long page_size = sysconf(_SC_PAGESIZE);
14     char *ptr = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
15                     MAP_ANON | MAP_SHARED, -1, 0);
16
17     if (ptr == MAP_FAILED) {
18         perror("mmap");
19         return EXIT_FAILURE;
20     }
21
22     pid = fork();
23     if (pid == -1) {
24         perror("fork");
25         return EXIT_FAILURE;
26     }
27
28     /* Child. */
29     if (pid == 0) {
30         for (i = 0; i < NUM_LOOPS; i++) {
31             ptr[1] = 'a' + i;
32             printf("child: %s\n", ptr);
33             sleep(1);
34         }
35
36         return EXIT_SUCCESS;
37     }
38
39     /* Parent. */
40     for (i = 0; i < NUM_LOOPS; i++) {
41         ptr[0] = 'a' + i;
42         printf("parent: %s\n", ptr);
43         sleep(1);
44     }
45
46     if (waitpid(pid, NULL, 0) == -1) {
47         perror("waitpid");
48         return EXIT_FAILURE;
49     }

```

```

50
51     return EXIT_SUCCESS;
52 }

```

Listing 14-39: Example anonymous shared mapping across fork

Here we establish a shared, anonymous mapping using a combination of `MAP_ANON` and `MAP_SHARED`. Note that under the covers this actually creates an unlinked tmpfs file (i.e. one which exists in tmpfs but has no entry in the file system, see section 5.0.2 for more details).

Therefore, when we fork, this pointer is shared between the parent and child processes. This example code iterates through letters, outputting both in parent and child to demonstrate that the memory is, in fact, shared.

14.7.4 Sharing memory via memfd

Another means of sharing memory between processes is to generate a memfd, which generates an unlinked shared tmpfs file much like the shared anonymous `mmap()` and system V shared memory approaches shown above, however permitting further special operations on that file descriptor.

Examining a simple server for this approach:-

```

1  #define _GNU_SOURCE
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/mman.h>
7  #include <unistd.h>
8
9  #define NAME "test"
10 #define SIZE (4096)
11 #define NUM_LOOPS (10)
12
13 int main(void)
14 {
15     int i, fd;
16     pid_t pid;
17     char *ptr;
18
19     fd = memfd_create(NAME, MFD_ALLOW_SEALING);
20     if (fd < 0) {
21         perror("memfd_create");
22         return EXIT_FAILURE;
23     }
24
25     if (ftruncate(fd, SIZE)) {
26         perror("ftruncate");

```

```

27         return EXIT_FAILURE;
28     }
29
30     /* Do not permit the size to change henceforth. */
31     if (fcntl(fd, F_ADD_SEALS,
32             F_SEAL_SHRINK | F_SEAL_GROW | F_SEAL_SEAL) < 0) {
33         perror("fcntl");
34         return EXIT_FAILURE;
35     }
36
37     ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
38     if (ptr == MAP_FAILED) {
39         perror("mmap");
40         return EXIT_FAILURE;
41     }
42
43     memset(ptr, 0, SIZE);
44
45     pid = getpid();
46     printf("Running at /proc/%d/fd/%d\n", pid, fd);
47
48     /* Wait for client... */
49     while (ptr[1] < 'a') {
50         sleep(1);
51     }
52
53     for (i = 0; i < NUM_LOOPS; i++) {
54         ptr[0] = 'a' + i;
55         printf("server: %s\n", ptr);
56         sleep(1);
57     }
58
59     return EXIT_SUCCESS;
60 }

```

Listing 14-40: Example memfd shared mapping server

This establishes a file descriptor via `memfd_create()`, specifying `MFD_ALLOW_SEALING` to permit the *sealing* of the file descriptor, which allows for the limiting of operations which can be performed upon it.

We then establish the size of the shared RAM-backed memory file via `ftruncate()`, before using `fcntl()` to specify `F_SEAL_GROW` which is a seal that disallows the growth in size of the shared file and `F_SEAL_SEAL` which disallows the client from adjust seals further.

Finally we are able to `mmap()` the file descriptor before outputting the location of the file descriptor entry in the process's `procs fd` directory. This is a simplified example for illustration's sake, in reality you'd transmit the file descriptor using another IPC method such as a UNIX socket.

Finally we can pass this path to the client:-

```

1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/mman.h>
5 #include <unistd.h>
6
7 #define SIZE (4096)
8 #define NUM_LOOPS (10)
9
10 int main(int argc, char **argv)
11 {
12     int i, fd;
13     char *ptr;
14
15     if (argc < 2) {
16         fprintf(stderr, "usage: %s [path to fd]\n", argv[0]);
17         return EXIT_FAILURE;
18     }
19
20     fd = open(argv[1], O_RDWR);
21     if (fd == -1) {
22         perror("open");
23         return EXIT_FAILURE;
24     }
25
26     ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
27     if (ptr == MAP_FAILED) {
28         perror("mmap");
29         return EXIT_FAILURE;
30     }
31
32     close(fd);
33
34     for (i = 0; i < NUM_LOOPS; i++) {
35         ptr[1] = 'a' + i;
36         printf("client: %s\n", ptr);
37         sleep(1);
38     }
39
40     return EXIT_SUCCESS;
41 }

```

Listing 14-41: Example memfd shared mapping client

This simply maps the file descriptor and accesses it as it would any file, modifying data in the mapping so we can observe both server and client updating data.