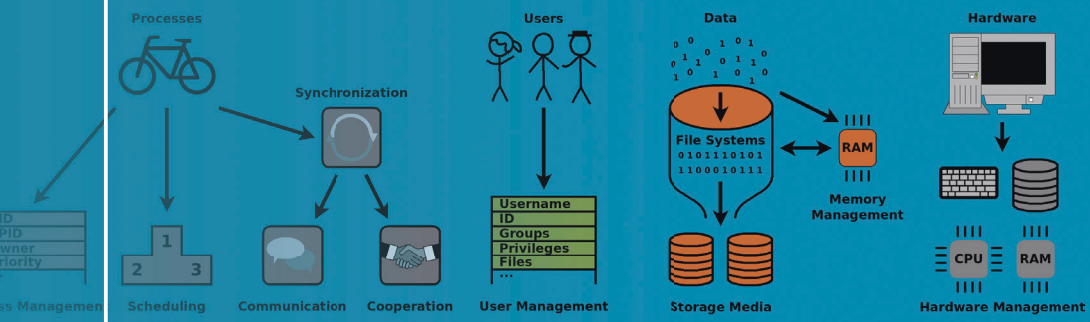

LEHRBUCH

Christian Baun

Operating Systems / Betriebssysteme

Bilingual Edition: English – German / Zweisprachige Ausgabe: Englisch – Deutsch

3rd edition/3. Auflage

Springer Vieweg

Operating Systems/Betriebssysteme

Christian Baun

Operating Systems/ Betriebssysteme

Bilingual Edition: English – German/ Zweisprachige Ausgabe: Englisch – Deutsch

3rd edition/3. Auflage

Springer Vieweg

Christian Baun
FB 2: Faculty of Computer Science
and Engineering
Frankfurt University of Applied Sciences
Frankfurt am Main, Deutschland

ISBN 978-3-658-48059-2 ISBN 978-3-658-48060-8 (eBook)
https://doi.org/10.1007/978-3-658-48060-8

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über https://portal.dnb.de abrufbar.

Planung/Lektorat: David Imgrund
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Preface to the third Edition

This edition one again includes new content that, on the one hand, addresses current developments and, on the other, supports the fundamental aims of the book. Although the volume grown considerably since the first edition, the aim of this work remains to present the core functionalities of operating systems in a compact and easy-to-understand manner, and to develop an understanding of how the computer and its components function together with the operating system.

One of the new sections in Chapter 3 includes a detailed description of the boot process of Linux operating systems, with all the relevant steps, from turning on the computer to handing over control to the user.

The description of the memory hierarchy in Chapter 4 has been expanded, particularly with regard to the performance parameters of the individual cache levels.

Chapter 5 now includes explanation of the translation lookaside buffer, which accelerates the translation from virtual to physical memory addresses. Additionally, for the first time, the chapter contains a description of the slab allocator, which is used by the Linux operating system kernel to efficiently manage the kernel's frequently used small objects and other data structures in the main memory.

The description of the process structure in memory in Chapter 8 has been expanded and improved. Furthermore, Chapter 8 now includes a description of Earliest Eligible Virtual Deadline First (EEVDF), the novel process scheduling method in the Linux kernel, which has replaced

Vorwort zur 3. Auflage

Auch diese Auflage enthält neue Inhalte, die einerseits aktuelle Entwicklungen aufgreifen, andererseits die grundsätzlichen Ziele dieses Buches unterstützen. Auch wenn seit der ersten Auflage der Umfang deutlich angewachsen ist, bleibt das Ziel dieses Werks, die Kernfunktionalitäten der Betriebssysteme kompakt und gut verständlich zu vermitteln sowie ein Verständnis zu entwickeln, wie der Computer und seine Komponenten im Zusammenspiel mit dem Betriebssystem funktionieren.

Neu hinzugekommen ist unter anderem in Kapitel 3 eine umfangreiche Beschreibung des Bootprozesses von Linux-Betriebssystemen mit allen relevanten Schritten vom Einschalten des Computers bis zur Übergabe der Kontrolle an die Benutzer.

Die Beschreibung der Speicherhierarchie in Kapitel 4 wurde insbesondere im Hinblick auf die Leistungsparameter der Cache-Ebenen ausgebaut.

In Kapitel 5 ist eine Beschreibung der Arbeitsweise des Übersetzungspuffers (Translation Lookaside Buffer) neu hinzugekommen, der die Adressumwandlung von virtuellen in physische Speicheradressen beschleunigt. Zusätzlich enthält Kapitel 5 nun erstmals eine Beschreibung des Slab Allocators. Diesen verwendet der Linux-Betriebssystemkern zur effizienten Verwaltung häufig genutzter, kleiner Objekte sowie anderer Kernel-Datenstrukturen im Hauptspeicher.

Die Struktur der Prozesse im Speicher in Kapitel 8 wurde ausgebaut und verbessert. Zudem berücksichtigt Kapitel 8 nun auch Earliest Eligible Virtual Deadline First (EEVDF), das neue Verfahren zum Prozess-Scheduling im Linux-

Completely Fair Scheduling after more than 15 years.

Chapter 9 features numerous additional explanations that enhance the understanding of the listings. For example, this edition includes a detailed explanation of how the permissions are specified using `mkfifo` and the file mode creation mask (`umask`). The sizes of anonymous and named pipes in modern Linux operating systems are discussed for the first time. Additionally, a comprehensive description of the differences between blocking and non-blocking sockets has also been added.

At this point, I would like to thank my editor David Imgrund for his support. I would also like to thank Matthias Kadlubowski, Jonas Göltl, Mert Kaan Demirel and Wei Yin Shing from the Frankfurt UAS for their helpful comments and suggestions for improvement. Finally, I thank my wife, Katrin Baun, for her strong encouragement and support.

Frankfurt am Main
March 2025

Prof. Dr. Christian Baun

Betriebssystemkern, das nach über 15 Jahren das Completely Fair Scheduling abgelöst hat.

In Kapitel 9 sind zahlreiche Beschreibungen hinzugekommen, die das Verständnis der Listings erleichtern. Neu hinzugekommen ist eine umfangreiche Erklärung des Zusammenspiels der bei `mkfifo` definierten Zugriffsrechte mit der Dateierzeugungsmaske (`umask`). Die Größen anonymer und benannter Pipes in modernen Linux-Betriebssystemen werden erstmals thematisiert. Auch eine umfangreiche Beschreibung der Unterschiede von blockierenden und nichtblockierenden Sockets ist neu hinzugekommen.

An dieser Stelle möchte ich meinem Lektor David Imgrund für seine Unterstützung danken. Zudem danke ich Matthias Kadlubowski, Jonas Göltl, Mert Kaan Demirel und Wei Yin Shing von der Frankfurt UAS für hilfreiche Verbesserungsvorschläge. Meiner Frau Katrin Baun danke ich für das Korrekturlesen und die viele Motivation und Unterstützung.

Preface to the second Edition

This edition includes some additional topics and some didactical enhancements.

Chapter 5 now also describes the five-level paging implemented by the latest server CPUs.

In Chapter 6, the sections about the file systems ext4 and Minix have been expanded. In addition, new illustrations about the structure of the NTFS file system, the Master File Table (MFT), and the Copy-on-Write working principle have been added, among other things. Furthermore, this edition, for the first time, contains sections about the modern file systems exFAT, ZFS, Btrfs, and ReFS.

Chapter 8 contains new content about process management for better understanding. Among other things, a new illustration of process switching and the relationship between user space, virtual memory, and user context is better explained in Section 8.3. In addition, descriptions of process scheduling in Linux operating systems have also been added in Section 8.6.12.

The sections covering interprocess communication and cooperation of processes have been expanded in Chapter 9. The descriptions of TCP sockets have been expanded, and examples of UDP sockets have been added. The descriptions of system calls and library functions for interprocess communication have been expanded a lot, and examples of working with the POSIX interface for shared memory areas, message queues, and semaphores are new in this edition. Finally, the section covering the cooperation of processes and threads with mutexes has been completely reworked.

Vorwort zur 2. Auflage

Diese Auflage enthält einige neue Themen und didaktische Verbesserungen.

Kapitel 5 berücksichtigt nun auch das fünfstufige Paging, das neueste Server-Prozessoren implementieren.

In Kapitel 6 wurden die Abschnitte zu den Dateisystemen ext4 und Minix erweitert. Neu hinzugekommen sind unter anderem Abbildungen zur Struktur des Dateisystems NTFS und der Einträge in der Master File Table (MFT) sowie zur Arbeitsweise von Copy-on-Write. Zudem enthält diese Auflage erstmals Abschnitte zu den modernen Dateisystemen exFAT, ZFS, Btrfs und ReFS.

Kapitel 8 enthält zum Thema Prozessverwaltung neue Inhalte, die das Verständnis erleichtern. Unter anderem gibt es eine neue Abbildung zu Prozesswechseln und der Zusammenhang zwischen Userspace, virtuellem Speicher und Benutzerkontext ist in Abschnitt 8.3 besser erklärt. Neu sind auch die Beschreibungen des Prozess-Schedulings in Linux-Betriebssystemen in Abschnitt 8.6.12.

Erweitert wurden die Themen Interprozesskommunikation und Kooperation von Prozessen in Kapitel 9. Die Beschreibungen zu TCP-Sockets wurden ausgebaut und Beispiele zu UDP-Sockets sind neu hinzugekommen. Die Beschreibungen der Systemaufrufe und Bibliotheksfunktionen zur Interprozesskommunikation wurden insgesamt ausgebaut und Beispiele zur Arbeit mit der Schnittstelle POSIX für gemeinsame Speicherbereiche, Nachrichtenwarteschlangen und Semaphoren sind neu in dieser Auflage dazugekommen. Der Abschnitt zur Kooperation von Prozessen und Threads mit Mutexen wurde komplett überarbeitet.

The command-line instructions for compiling and running the programs are now included in the output of the program examples in chapters 7, 8, and 9. This makes it easier for beginners to follow the individual steps from the source code to the output of the programs.

Several sections in Chapter 10 have also been reworked. In particular, the sections covering the partitioning and the emulators have been extended.

At this point, I would like to thank my editor David Imgrund for his support. I would also like to thank Jörg Abke from the TH Aschaffenburg, Michael Eggert from the Würzburg-Schweinfurt University of Technology, and Henry Cocos, Peter Ebinger, Oliver Hahm, Benedikt Möller, Anton Rösler, and Amalie-Margarete Wilke from the Frankfurt UAS for their helpful comments and suggestions for improvement. Finally, I thank my wife, Katrin Baun, for her strong encouragement and support.

Bei den Ausgaben der Programmbeispiele in den Kapitel 7, 8 und 9 sind die Kommandozeilenbefehle zum Kompilieren und Ausführen nun mit dabei. Dies erleichtert Einsteigern das Nachvollziehen der einzelnen Schritte vom Quellcode zur Ausgabe der Programme.

Überarbeitet wurden auch mehrere Abschnitte in Kapitel 10. In erster Linie die Themen Partitionierung und Emulatoren sind nun inhaltlich erweitert.

An dieser Stelle möchte ich meinem Lektor David Imgrund für seine Unterstützung danken. Zudem danke ich Jörg Abke von der TH Aschaffenburg, Michael Eggert von der Technischen Hochschule Würzburg-Schweinfurt sowie Henry Cocos, Peter Ebinger, Oliver Hahm, Benedikt Möller, Anton Rösler und Amalie-Margarete Wilke von der Frankfurt UAS für hilfreiche Verbesserungsvorschläge. Meiner Frau Katrin Baun danke ich für das Korrekturlesen und die viele Motivation und Unterstützung.

Frankfurt am Main
July 2023

Prof. Dr. Christian Baun

Preface to the first Edition

Operating systems are an important topic of practical computer science and, to a lesser extent, also of technical computer science. They are the interface between the hardware of a computer system and its users and their software processes. Furthermore, operating systems manage the hardware components of a computer system and the stored data.

This compact book on the broad topic of operating systems was written to provide an overview of the most essential task areas and core functionalities of operating systems, and thus assist the readers in learning how operating systems work, how they implement essential functionalities, and how they use and control the most important hardware components of a computer system.

Also, this book intends to support those readers who wish to learn not only technical aspects of operating systems, but also want to improve their language skills in English or German.

The motivation to write this bilingual textbook arose from recent changes in universities. Many universities have migrated individual modules or even whole study programs to English to be more attractive for international students and to improve the language skills of the graduates.

The example programs in this book were all written in the C programming language and tested under the free operating system Debian GNU/Linux. In principle, they should run under any other Unix-like operating system.

The example programs and the errata list will be published here:
`https://christianbaun.de`

If you notice any points of criticism while working with this book, or if you have suggestions for improvement for future editions,

Vorwort zur 1. Auflage

Betriebssysteme sind ein wichtiges Thema der praktischen Informatik und zum geringeren Teil auch der technischen Informatik. Sie sind die Schnittstelle zwischen der Hardware eines Computers und seinen Benutzern und deren Softwareprozessen. Zudem verwalten Betriebssysteme die Hardwarekomponenten eines Computers und die gespeicherten Daten.

Dieses kompakte Werk über das breite Thema Betriebssysteme wurde mit dem Ziel geschrieben, dem Leser einen Überblick über die wichtigsten Aufgabenbereiche und Kernfunktionalitäten von Betriebssystemen zu verschaffen und so das Verständnis dafür zu wecken, wie Betriebssysteme funktionieren, wie sie die wichtigsten Funktionalitäten erbringen und wie sie die wichtigsten Hardwarekomponenten eines Computers nutzen und steuern.

Zudem soll dieses Buch diejenigen Leser unterstützen, die sich nicht nur fachlich im Thema Betriebssysteme, sondern auch sprachlich in der englischen oder in der deutschen Sprache weiterbilden möchten.

Die Motivation, dieses zweisprachige Lehrbuch zu schreiben, ergab sich aus den Veränderungen des Hochschulalltags in jüngerer Vergangenheit. Zahlreiche Hochschulen haben einzelne Module oder ganze Studiengänge auf die englische Sprache umgestellt, um für ausländische Studenten attraktiver zu sein, und um die sprachlichen Fähigkeiten der Absolventen zu verbessern.

Die Programmbeispiele in diesem Werk wurden alle in der Programmiersprache C geschrieben und unter dem freien Betriebssystem Debian GNU/Linux getestet. Prinzipiell sollten sie unter jedem anderen Unix-(ähnlichen) Betriebssystem laufen.

Die Programmbeispiele und die Errata-Liste wird hier veröffentlicht:
`https://christianbaun.de`

Wenn Ihnen bei der Arbeit mit diesem Werk Kritikpunkte auffallen, oder Sie Verbesserungsvorschläge für zukünftige Auflagen haben, würde

I would be happy to receive an email from you: christianbaun@gmail.com

At this point, I want to thank my editor Sybille Thelen for her support. Also, many thanks to Turhan Arslan, and especially Jens Liebehenschel and Torsten Wiens, for proofreading. I thank my parents Dr. Marianne Baun and Karl-Gustav Baun, as well as my parents-in-law Anette Jost and Hans Jost and, in particular, my wife, Katrin Baun, for their motivation and support in good times and in difficult times.

Frankfurt am Main
März 2020

Prof. Dr. Christian Baun

ich mich über eine Email von Ihnen sehr freuen: christianbaun@gmail.com

An dieser Stelle möchte ich meiner Lektorin Sybille Thelen für ihre Unterstützung danken. Zudem danke ich Turhan Arslan und ganz besonders Jens Liebehenschel und Torsten Wiens für das Korrekturlesen. Meinen Eltern Dr. Marianne Baun und Karl-Gustav Baun sowie meinen Schwiegereltern Anette Jost und Hans Jost und ganz besonders meiner Frau Katrin Baun danke ich für die Motivation und Unterstützung in guten und in schwierigen Zeiten.

Contents

Inhaltsverzeichnis

1

Introduction

This book intends to provide a compact, but not a comprehensive overview of operating systems and their components. The aim is to assist its readers in gaining a basic understanding of the way operating systems and their components work. No prior technical knowledge is required.

The Chapters 2 and 3 provide an introduction to the fundamentals of information technology (IT) and operating systems. These topics are necessary to understand operating systems and the contents of this book.

An understanding of the main hardware components of a computer is essential to understand how operating systems work. Therefore, Chapter 4 deals with the basics of computer architecture. This chapter focuses on the way the CPU, the memory, and the bus systems work.

Chapter 5 describes the fundamental concepts of memory management and the way modern operating systems organize the cache and the main memory.

Chapter 6 deals with another form of memory management. This chapter describes the technical basics of classic and modern file systems using various examples.

The user processes interact with the functions of the operating system kernel via system calls. These are called directly or via library functions. A description of the functioning of system calls and how to use them is included in Chapter 7.

The focus of Chapter 8 is on process management. The topics of this chapter are, among others, the implementation of processes by the operating system, and the criteria by which they are assigned to the CPU.

Einleitung

Dieses Buch will einen Überblick über das Thema Betriebssysteme und deren Komponenten schaffen, ohne dabei den Anspruch auf Vollständigkeit zu erheben. Das Ziel ist es, den Leserinnen und Lesern ein grundlegendes Wissen über die Funktionsweise von Betriebssystemen und deren Komponenten zu vermitteln. Technische Vorkenntnisse sind dabei nicht erforderlich.

In den Kapiteln 2 und 3 findet eine Einführung in die Grundlagen der Informationstechnik und der Betriebssysteme statt. Dies ist nötig, um die Thematik Betriebssysteme und den Inhalt dieses Buchs verstehen zu können.

Ein Verständnis der notwendigsten Hardwarekomponenten eines Computers ist elementar, um die Arbeitsweise von Betriebssystemen zu verstehen. Darum beschäftigt sich Kapitel 4 mit den Grundlagen der Rechnerarchitektur. Schwerpunkte dieses Kapitels sind die Arbeitsweise des Hauptprozessors, des Speichers und der Bussysteme.

Kapitel 5 beschreibt die grundlegenden Konzepte der Speicherverwaltung und die Art und Weise, wie moderne Betriebssysteme den Cache und den Hauptspeicher verwalten.

Eine andere Form der Speicherverwaltung thematisiert Kapitel 6. Dieses Kapitel beschreibt die technischen Grundlagen klassischer und moderner Dateisysteme anhand ausgewählter Beispiele.

Die Interaktion der Benutzerprozesse mit den Funktionen des Betriebssystemkerns geschieht via Systemaufrufe. Diese können direkt oder über Bibliotheksfunktionen aufgerufen werden. Eine Beschreibung der Arbeitsweise von und mit Systemaufrufen enthält Kapitel 7.

Der Fokus von Kapitel 8 liegt auf der Prozessverwaltung. Schwerpunkte sind einerseits die Art und Weise, wie Prozesse im Betriebssystem realisiert werden und nach welchen Kriterien diese den Prozessor zugewiesen bekommen.

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_1

Possible ways for protecting critical sections and the various aspects of interprocess communication are described in Chapter 9.

Finally, Chapter 10 provides an introduction to the virtualization concepts that are relevant for operating systems.

Möglichkeiten zum Schutz kritischer Abschnitte und die verschiedenen Aspekte der Interprozesskommunikation beschreibt Kapitel 9.

Abschließend findet in Kapitel 10 eine Einführung in die aus Sicht der Betriebssysteme relevanten Virtualisierungskonzepte statt.

2

Fundamentals of Computer Science

A basic understanding of information technology (IT) is required to understand how operating systems work. These fundamentals include the variants of information representation, and the representation of numbers, orders of magnitude, and the way information (especially textual information) are represented in computers.

Grundlagen der Informationstechnik

Um die Funktionsweise von Betriebssystemen zu verstehen, ist ein grundlegendes Verständnis der Informationstechnik (IT) nötig. Bei diesen Grundlagen handelt es sich um die Möglichkeiten der Informationsdarstellung und Repräsentation von Zahlen, Größenordnungen und die Art und Weise, wie Informationen (speziell Texte) in Rechnern dargestellt werden.

2.1

Bit

A *bit* is the smallest possible unit of information, and every piece of information is bound to an information carrier [43]. An information carrier, which can have one of two states, can represent one bit of data. The value of one or more bits is called *state*. One bit can represent two states. Different scenarios can represent one bit of data. Examples are:

- The position of a switch with two states.
- The switching state of a transistor.
- The presence of an electrical voltage or charge.
- The presence of magnetization.

Bit

Ein *Bit* ist die kleinstmögliche Einheit der Information und jede Information ist an einen Informationsträger gebunden [43]. Ein Informationsträger, der sich in genau einem von zwei Zuständen befinden kann, kann die Datenmenge 1 Bit darstellen. Den Wert eines oder mehrerer Bits nennt man *Zustand*. Ein Bit kann zwei Zustände darstellen. Verschiedene Sachverhalte können die Datenmenge 1 Bit darstellen. Beispiele sind:

- Die Stellung eines Schalters mit zwei Zuständen.
- Der Schaltzustand eines Transistors.
- Das Vorhandensein einer elektrischen Spannung oder Ladung.
- Das Vorhandensein einer Magnetisierung.

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_2

- The value of a variable with logical truth values.

If more than two states are required to store information, multiple bits (*bit sequences*) are needed. With n bits, it is possible to represent 2^n different states (see Table 2.1). With 2 bits, $2^2 = 4$ different states can be represented, namely `00`, `01`, `10`, and `11`. With 3 bits, $2^3 = 8$ different states (`000`, `001`, `010`, `011`, `100`, `101`, `110`, and `111`) can be represented. With each additional bit, the number of representable states (bit sequences) doubles [43].

- Der Wert einer Variable mit den logischen Wahrheitswerten.

Benötigt man zur Speicherung einer Information mehr als zwei Zustände, sind Folgen von Bits (*Bitfolgen*) nötig. Mit n Bits kann man 2^n verschiedene Zustände darstellen (siehe Tabelle 2.1). Also kann man mit 2 Bits $2^2 = 4$ verschiedene Zustände repräsentieren, nämlich `00`, `01`, `10` und `11`. Mit 3 Bits kann man schon $2^3 = 8$ verschiedene Zustände (`000`, `001`, `010`, `011`, `100`, `101`, `110` und `111`) repräsentieren. Jedes zusätzliche Bit verdoppelt die Anzahl der möglichen darstellbaren Zustände (Bitfolgen) [43].

Table 2.1: The Number of representable States doubles with each additional Bit

Bits	States	Bits	States	Bits	States
1	$2^1 = 2$	9	$2^9 = 512$	17	$2^{17} = 131{,}072$
2	$2^2 = 4$	10	$2^{10} = 1{,}024$	18	$2^{18} = 262{,}144$
3	$2^3 = 8$	11	$2^{11} = 2{,}048$	19	$2^{19} = 524{,}288$
4	$2^4 = 16$	12	$2^{12} = 4{,}096$	20	$2^{20} = 1{,}048{,}576$
5	$2^5 = 32$	13	$2^{13} = 8{,}192$	21	$2^{21} = 2{,}097{,}152$
6	$2^6 = 64$	14	$2^{14} = 16{,}384$	22	$2^{22} = 4{,}194{,}304$
7	$2^7 = 128$	15	$2^{15} = 32{,}768$	23	$2^{23} = 8{,}388{,}608$
8	$2^8 = 256$	16	$2^{16} = 65{,}536$	24	$2^{24} = 16{,}777{,}216$

2.2

Representation of Numbers

Numbers can be represented in various ways. One task of IT is to map numbers from the *real world* to the computer. In this context, the distinction between *value* and *representation* is significant.

In mathematics, numbers are distinguished as elements of different sets of values (natural numbers, integers, real numbers, complex numbers, ...). The value of a number is also called *abstract number*, and its value is independent of the representation (for example, $0.5 = 1/2$).

However, operations of a computer are not executed on values, but on bit sequences. This is why the representation of numbers is particularly interesting for IT. The representation is determined by the place value system (positional notation). Relevant for IT are the decimal sys-

Repräsentation von Zahlen

Zahlen kann man auf unterschiedliche Arten darstellen. Eine Aufgabe der IT ist es, Zahlen aus der *realen Welt* im Computer abzubilden. Wichtig ist dabei die Unterscheidung zwischen *Wert* und *Darstellung*.

In der Mathematik unterscheidet man Zahlen als Elemente verschiedener Wertemengen (natürliche Zahlen, ganze Zahlen, reelle Zahlen, komplexe Zahlen, usw.). Den Wert einer Zahl nennt man auch *abstrakte Zahl* und der Wert ist unabhängig von der Darstellung (zum Beispiel $0{,}5 = 1/2$).

Operationen eines Rechners werden aber nicht auf Werten, sondern auf Bitfolgen ausgeführt. Darum ist für die IT besonders die Darstellung der Zahlen interessant. Die Darstellung wird vom verwendeten Stellenwertsystem (Positionssystem) bestimmt. Die für die IT wichtigen

tem, the binary system, the octal system, and the hexadecimal system.

Stellenwertsysteme sind das Dezimalsystem, das Dualsystem, das Oktalsystem und das Hexadezimalsystem.

2.2.1

Decimal System

The decimal system uses the number 10 as base. Each digit D at the position i has the value $D \times 10^i$. One example is:

Dezimalsystem

Das Dezimalsystem verwendet als Basis die Zahl 10. Jede Ziffer D an der Stelle i hat den Wert $D \times 10^i$. Ein Beispiel ist:

$$2013 = 2 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 3 \times 10^0$$

Computer systems distinguish between two electrical states. Therefore, the number 2 as base; hence, the binary system is ideal for IT.

Computer-Systeme unterscheiden prinzipiell zwischen zwei elektrischen Zuständen. Darum ist aus Sicht der IT als Basis die Zahl 2 und damit das Dualsystem optimal geeignet.

2.2.2

Binary System

The binary system uses the number 2 as base. Numbers are represented by using the digits of the values zero and one. Number representations in the binary system are called binary numbers. All positive natural numbers, including zero, can be represented by sequences of symbols from the set $\{0, 1\}$. If n is the number of bits, x_0 is the *least significant bit* (LSB), and x_{n-1} is the *most significant bit* (MSB).

Since long series of zeros and ones quickly become confusing to humans, the octal system or the hexadecimal system are often used to represent bit sequences.

The conversion of numbers between different place value systems is simple. For clarification, the place value system of the respective number is subscripted in the following examples.

When converting binary numbers to decimal numbers, the digits are multiplied by their place values, and the results are summed up.

Dualsystem

Das Dualsystem verwendet als Basis die Zahl 2. Zahlen werden nur mit den Ziffern der Werte Null und Eins dargestellt. Zahldarstellungen im Dualsystem heißen Dualzahlen oder Binärzahlen. Alle positiven natürlichen Zahlen inklusive der Null können durch Folgen von Symbolen aus der Menge $\{0, 1\}$ repräsentiert werden. Wenn n der Anzahl der Bits entspricht, ist x_0 das niederwertigste Bit (*Least Significant Bit* – LSB) und x_{n-1} das höchstwertigste Bit (*Most Significant Bit* – MSB).

Da lange Reihen von Nullen und Einsen für Menschen schnell unübersichtlich werden, verwendet man zur Darstellung von Bitfolgen häufig das Oktalsystem oder das Hexadezimalsystem.

Die Umrechnung der Stellenwertsysteme ist einfach möglich. Zur Verdeutlichung ist das Stellenwertsystem der jeweiligen Zahl in den folgenden Beispielen tiefgestellt beigefügt.

Bei der Umwandlung von Dualzahlen in Dezimalzahlen werden die Ziffern mit ihren Stellenwertigkeiten ausmultipliziert und die Ergebnisse addiert.

$$10100100_2 = 2^7 + 2^5 + 2^2 = 164_{10}$$

Multiple ways exist to convert decimal numbers to binary numbers. One method is presented in Table 2.2. The decimal number is divided by the base value 2, and the result, as well as the remainder (value zero or one), are noted. In the next round (row of the table), the result of the division is again divided by the base value, and the result, as well as the remainder, are both noted. This algorithm continues until the result of the division is zero.

Die Umwandlung von Dezimalzahlen in Dualzahlen ist unter anderem mit dem in Tabelle 2.2 gezeigten Verfahren möglich. Dabei wird die Dezimalzahl durch die Basis 2 dividiert und das Ergebnis und der Rest (Wert Null oder Eins) werden notiert. Das Ergebnis der Division wird in der nächsten Runde (Zeile der Tabelle) erneut durch die Basis dividiert und erneut werden das Ergebnis und der Rest notiert. Dieser Restwertalgorithmus wird so lange weitergeführt, bis das Ergebnis der Division Null ist.

Table 2.2: Converting the decimal number 164_{10} to the binary number 10100100_2

k	*Quotient* k DIV 2	*Remainder* k MODULO 2
164	82	$0 = x_0$
82	41	$0 = x_1$
41	20	$1 = x_2$
20	10	$0 = x_3$
10	5	$0 = x_4$
5	2	$1 = x_5$
2	1	$0 = x_6$
1	0	$1 = x_7$

2.2.3

Octal System

The octal system uses the number 8 as base and represents groups of 3 bits with a single digit.

When converting from binary numbers to octal numbers, the bit sequence is subdivided, beginning at the least significant bit, in groups of three. Each group of three corresponds to a single position of the octal number.

Oktalsystem

Das Oktalsystem verwendet als Basis die Zahl 8 und kann Gruppen von 3 Bits mit einem Zeichen darstellen.

Bei der Umwandlung von Dualzahlen in Oktalzahlen wird die Bitkette vom niederwertigsten Bit beginnend in Dreiergruppen unterteilt. Jede Dreiergruppe entspricht einer Stelle der Oktalzahl.

$$164_{10} = 10|100|100_2 = 244_8$$

The conversion of octal numbers to binary numbers is done analogously. One digit in the octal system corresponds to three digits in the binary system.

Die Umwandlung von Oktalzahlen in Dualzahlen erfolgt analog. Eine Stelle im Oktalsystem entspricht drei Stellen im Dualsystem.

2.2.4

Hexadecimal System

The hexadecimal system uses the number 16 as base. The representation of positive natural numbers is done with the 16 symbols from the set $\{0, 1, \ldots 8, 9, A, B, C, D, E, F\}$. A single character can represent a group of 4 bits (tetrad, half-byte or nibble).

Hexadezimalsystem

Das Hexadezimalsystem verwendet als Basis die Zahl 16. Die Darstellung positiver natürlicher Zahlen erfolgt mit den 16 Ziffern und Buchstaben aus der Menge $\{0, 1, \ldots 8, 9, A, B, C, D, E, F\}$. Ein Zeichen kann eine Gruppe von 4 Bits (Tetrade, Halbbyte oder Nibble) darstellen.

Table 2.3: Different Representations of positive natural Numbers

Decimal representation	Binary representation	Octal representation	Hexadecimal representation
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

When converting binary numbers to hexadecimal numbers, the bit sequence is subdivided into tetrads, starting with the least significant bit. Each tetrad corresponds to one digit of the hexadecimal number.

Bei der Umwandlung von Dualzahlen in Hexadezimalzahlen wird die Bitkette vom niederwertigsten Bit beginnend in Tetraden unterteilt. Jede Tetrade entspricht einer Stelle der Hexadezimalzahl.

$$164_{10} = 1010|0100_2 = A4_{16}$$

The conversion of hexadecimal numbers to binary numbers is done analogously. One digit in the hexadecimal system corresponds to four digits in the binary system.

Table 2.3 contains an overview of the various representations of the first 16 positive natural numbers in the decimal system, the binary system, the octal system, and the hexadecimal system.

Die Umwandlung von Hexadezimalzahlen in Dualzahlen geschieht analog. Eine Stelle im Hexadezimalsystem entspricht vier Stellen im Dualsystem.

Tabelle 2.3 enthält eine Übersicht der verschiedenen Darstellungen der ersten 16 positiven natürlichen Zahlen im Dezimalsystem, Dualsystem, Oktalsystem und Hexadezimalsystem.

2.3

File and Storage Dimensions

For performance reasons, computers usually do not carry out read and write operations on single bits, but instead, work with bit sequences whose lengths are multiples of eight. A group of 8 bits is called *byte*. The value of a byte can be represented either by 8 bits or two hexadecimal digits.

A file is an arbitrarily long sequence of bytes that contain related data. All information (numbers, texts, music, programs, ...) a computer should work with has to be represented as a sequence of bytes and stored as a file [43].

Since most files have a size of several thousand or millions of bytes, different size units for shortened number representation exist. For data storage with binary addressing, storage capacities of 2^n bytes (powers of two) are used (see Table 2.4).

Datei- und Speichergrößen

Computer lesen und schreiben aus Geschwindigkeitsgründen meist nicht einzelne Bits, sondern arbeiten mit Bitfolgen, deren Längen Vielfache von Acht sind. Eine Gruppe von 8 Bits nennt man *Byte*. Den Wert eines Bytes kann man entweder durch 8 Bits oder zwei Hexadezimalziffern darstellen.

Eine Datei ist eine beliebig lange Folge von Bytes und enthält inhaltlich zusammengehörende Daten. Alle Informationen (Zahlen, Texte, Musik, Programme, usw.), mit denen ein Computer arbeiten soll, müssen als Folge von Bytes repräsentiert werden können und als Datei gespeichert werden [43].

Da sich die Größenordnungen der meisten Dateien im Bereich mehrerer Tausend oder Millionen Bytes befinden, gibt es verschiedene Größeneinheiten zur verkürzten Zahlendarstellung. Für Datenspeicher mit binärer Adressierung ergeben sich Speicherkapazitäten von 2^n Byte, also Zweierpotenzen (siehe Tabelle 2.4).

Table 2.4: File and Memory Sizes

Name	Symbol	Bytes
Kilobyte	kB	$2^{10} = 1{,}024$
Megabyte	MB	$2^{20} = 1{,}048{,}576$
Gigabyte	GB	$2^{30} = 1{,}073{,}741{,}824$
Terabyte	TB	$2^{40} = 1{,}099{,}511{,}627{,}776$
Petabyte	PB	$2^{50} = 1{,}125{,}899{,}906{,}842{,}624$
Exabyte	EB	$2^{60} = 1{,}152{,}921{,}504{,}606{,}846{,}976$
Zettabyte	ZB	$2^{70} = 1{,}180{,}591{,}620{,}717{,}411{,}303{,}424$
Yottabyte	YB	$2^{80} = 1{,}208{,}925{,}819{,}614{,}629{,}174{,}706{,}176$

The units in Table 2.4 are typically used by operating systems to specify the storage capacities of the main memory and the storage devices. However, the manufacturers of hard disk drives, CD/DVDs, and USB storage drives prefer using decimal prefixes for the calculation of the storage capacity and its specification on the packaging. This means, for example, 10^9 instead of 2^{30} for GB is used and 10^{12} instead of 2^{40} for TB. For this reason, for example, when using a DVD-R disc with a specified capacity of 4.7 GB, several applications will show its correct capacity of only 4.38 GB.

Die Maßeinheiten in Tabelle. 2.4 haben sich für die Größenangabe von Hauptspeicher und Speichermedien in Betriebssystemen eingebürgert. Die Hersteller von Festplatten, CD/DVDs und USB-Speichermedien bevorzugen zur Berechnung der Kapazität und zur Angabe auf der Verpackung aber lieber Dezimal-Präfixe, also zum Beispiel den Faktor 10^9 anstatt 2^{30} für GB und 10^{12} anstatt 2^{40} für TB. Aus diesem Grund wird zum Beispiel bei einem DVD-Rohling mit einer angegebenen Kapazität von 4,7 GB in vielen Anwendungen korrekterweise nur die Kapazität 4,38 GB angezeigt.

$$10^9 = 1{,}000{,}000{,}000\,, \quad 2^{30} = 1{,}073{,}741{,}824$$

In this case, the difference in capacity between the power of two and the power of ten is approximately 7.37 %.

For larger storage systems, the difference is even more significant. For example, of a hard disk drive with a propagated storage capacity of 1 TB, only about 930 GB can be used.

Der Kapazitätsunterschied zwischen Zweierpotenz und Zehnerpotenz ist in diesem Fall ca. 7,37 %.

Bei größeren Speichern ist der Unterschied noch größer. So können von einer Festplatte mit angeblich 1 TB Speicherkapazität tatsächlich nur etwa 930 GB verwendet werden.

$$10^{12} = 1{,}000{,}000{,}000{,}000\,, \quad 2^{40} = 1{,}099{,}511{,}627{,}776$$

The difference in capacity between the power of two and the power of ten, in this case, is already approximately 9.95 %. With each further unit of measurement (PB, EB, ZB, ...), the difference in capacity between the power of two and the power of ten grows.

The International Electrotechnical Commission (IEC) proposed in 1996 to label the popular size factors that are based on powers of two with the lowercase letter "i", and to reserve the established designations for the powers of 10. This proposal did not succeed so far, and the alternative names Kibibyte (KiB), Mebibyte (MiB), Gibibyte (GiB), Tebibyte (TiB), Pebibyte (PiB), Exbibyte (EIB), and Zebibyte (ZiB) did not gain much popularity outside the academic world.

Der Kapazitätsunterschied zwischen Zweierpotenz und Zehnerpotenz ist in diesem Fall schon ca. 9,95 % und mit jeder weiteren Maßeinheit (PB, EB, ZB, usw.) wächst der Kapazitätsunterschied zwischen Zweierpotenzen und Zehnerpotenzen weiter.

Die International Electrotechnical Commission (IEC) schlug 1996 vor, die populären Größenfaktoren, die auf den Zweierpotenzen basieren, mit einem kleinen „i" zu kennzeichnen und die etablierten Bezeichnungen der Maßeinheiten für die Zehnerpotenzen zu reservieren. Dieser Vorschlag konnte sich bislang nicht durchsetzen und die daraus resultierenden alternativen Bezeichnungen Kibibyte (KiB), Mebibyte (MiB), Gibibyte (GiB), Tebibyte (TiB), Pebibyte (PiB), Exbibyte (EiB) und Zebibyte (ZiB) sind außerhalb des akademischen Bereichs nicht besonders populär.

2.4

Information Representation

Data is represented as sequences of zeros and ones that encode arbitrary information. In order to represent text and numbers with data, the characters of the alphabet (uppercase and lowercase), punctuation marks such as period, comma, and semicolon, as well as some special characters like `+`, `%`, `&`, and `$` are encoded as bit sequences. Special characters, such as space (`SP`), carriage return (`CR`), and tabulator (`TAB`) are also required. The most established encoding is the *American Standard Code for Information Interchange (ASCII).*

Informationsdarstellung

Daten sind Folgen von Nullen und Einsen, die beliebige Informationen repräsentieren. Um Text und Zahlen durch Daten darzustellen, kodiert man die Zeichen des Alphabets (Groß- und Kleinschreibung), Satzzeichen wie Punkt, Komma und Semikolon, sowie einige Spezialzeichen wie zum Beispiel `+`, `%`, `&` und `$` in Bitfolgen. Zudem sind Sonderzeichen wie Leerzeichen (`SP`), Wagenrücklauf (`CR`) und Tabulator (`TAB`) nötig. Die am besten etablierte Kodierung ist der *American Standard Code for Information Interchange (ASCII).*

2.4.1

ASCII Encoding

The ASCII encoding, also called US-ASCII, is a 7-bit character encoding. I.e., each character is assigned to a bit sequence of 7 bits. There are $2^7 = 128$ different bit sequences, which is also the number of characters defined by this character encoding (see Table 2.5). Of these 128 characters, 95 characters are printable, and 33 characters are not. The printable characters are (starting with space):

ASCII-Kodierung

Die ASCII-Kodierung, häufig auch US-ASCII genannt, ist eine 7-Bit-Zeichenkodierung. Das heißt, dass jedem Zeichen eine Bitfolge aus 7 Bits zugeordnet ist. Es existieren also $2^7 = 128$ verschiedene Bitfolgen und exakt so viele Zeichen definiert die Zeichenkodierung (siehe Tabelle 2.5). Von den 128 Zeichen sind 95 Zeichen druckbar und 33 Zeichen nicht druckbar. Die druckbaren Zeichen sind (beginnend mit dem Leerzeichen):

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

The non-printable characters 00_{16} to 20_{16} and $7F_{16}$, for example, backspace (`BS`) and carriage return (`CR`), are control characters originally used to control a teleprinter. ASCII is, therefore, not only a standard for storing data but also suitable for data transmission. The beginning and the end of a data transmission are highlighted by Start of Text (`STX`) and End of Text (`ETX`). The transmission can be controlled with non-printable characters such as Acknowledge (`ACK`) and Negative Acknowledge (`NAK`). With Bell (`BEL`), a sender can transmit an alarm signal to the receiver, e.g., in the event of an error.

The 8th bit can be used as a parity bit for error detection. In this case, it has the value 0, if the number of one-bits at the remaining seven bit positions is even. Otherwise, it has the value 1.

Due to improved protocols for data transmission, the 8th bit is no longer required for error detection when transmitting ASCII-encoded texts. Therefore, in order to be able to encode additional characters, US-ASCII was upgraded with numerous extensions and became an 8-bit character encoding. If each character has a sequence of 8 bits assigned, then $2^8 = 256$ different bit sequences are available. This resulted in 128 additional characters in comparison to the standard US-ASCII. Since these 128 additional characters are not sufficient to encode all internationally required special characters, several

Die nicht druckbaren Zeichen 00_{16} bis 20_{16} und $7F_{16}$, zum Beispiel Backspace (`BS`) und Carriage Return (`CR`), sind Steuerzeichen, die ursprünglich zur Steuerung eines Fernschreibers verwendet wurden. ASCII ist also nicht nur ein Standard zur Datenablage, sondern auch zur Datenübertragung geeignet. Den Anfang und das Ende einer Datenübertragung markiert man mit Start of Text (`STX`) bzw. End of Text (`ETX`). Die Steuerung der Übertragung ist mit nicht druckbaren Zeichen wie Acknowledge (`ACK`) und negative Acknowledge (`NAK`) möglich. Mit Bell (`BEL`) kann ein Sender, zum Beispiel bei einem Fehler, ein Alarmsignal an den Empfänger senden.

Das 8. Bit kann als Paritätsbit zur Fehlererkennung verwendet werden. In diesem Fall hat es den Wert 0, wenn die Anzahl der Einsen an den übrigen sieben Bitpositionen gerade ist und ansonsten den Wert 1.

Durch verbesserte Protokolle zur Datenübertragung benötigt man das 8. Bit bei der Datenübertragung von ASCII-kodierten Texten nicht mehr zur Fehlererkennung. Darum wurde, um zusätzliche Zeichen kodieren zu können, US-ASCII mit zahlreichen Erweiterungen zu einer 8-Bit-Zeichenkodierung erweitert. Wird jedem Zeichen eine Bitfolge aus 8 Bits zugeordnet, sind $2^8 = 256$ verschiedene Bitfolgen verfügbar. Es sind also 128 Zeichen mehr verfügbar, als bei US-ASCII. Da diese 128 zusätzlichen Zeichen nicht ausreichen, um alle international benötigten Sonderzeichen zu kodieren, existieren ver-

Table 2.5: The ASCII Character Encoding (US-ASCII)

Dez	Hex	Char.	Dez	Hex	Char.	Dez	Hex	Char.	Dez	Hex	Char.
000	00	NUL	032	20	Space	064	40	@	096	60	‘
001	01	SOH	033	21	!	065	41	A	097	61	a
002	02	STX	034	22	"	066	42	B	098	62	b
003	03	ETX	035	23	#	067	43	C	099	63	c
004	04	EOT	036	24	$	068	44	D	100	64	d
005	05	ENQ	037	25	%	069	45	E	101	65	e
006	06	ACK	038	26	&	070	46	F	102	66	f
007	07	BEL	039	27	'	071	47	G	103	67	g
008	08	BS	040	28	(	072	48	H	104	68	h
009	09	TAB	041	29	)	073	49	I	105	69	i
010	0A	LF	042	2A	*	074	4A	J	106	6A	j
011	0B	VT	043	2B	+	075	4B	K	107	6B	k
012	0C	FF	044	2C	,	076	4C	L	108	6C	l
013	0D	CR	045	2D	-	077	4D	M	109	6D	m
014	0E	SO	046	2E	.	078	4E	N	110	6E	n
015	0F	SI	047	2F	/	079	4F	O	111	6F	o
016	10	DLE	048	30	0	080	50	P	112	70	p
017	11	DC1	049	31	1	081	51	Q	113	71	q
018	12	DC2	050	32	2	082	52	R	114	72	r
019	13	DC3	051	33	3	083	53	S	115	73	s
020	14	DC4	052	34	4	084	54	T	116	74	t
021	15	NAK	053	35	5	085	55	U	117	75	u
022	16	SYN	054	36	6	086	56	V	118	76	v
023	17	ETB	055	37	7	087	57	W	119	77	w
024	18	CAN	056	38	8	088	58	X	120	78	x
025	19	EM	057	39	9	089	59	Y	121	79	y
026	1A	SUB	058	3A	:	090	5A	Z	122	7A	z
027	1B	ESC	059	3B	;	091	5B	[	123	7B	{
028	1C	FS	060	3C	<	092	5C	\	124	7C	|
029	1D	GS	061	3D	=	093	5D	]	125	7D	}
030	1E	RS	062	3E	>	094	5E	^	126	7E	~
031	1F	US	063	3F	?	095	5F	_	127	7F	DEL

ASCII extensions for different languages and regions exist.

These extensions are compatible with the original US-ASCII. All characters specified in US-ASCII are encoded in the different extensions by the same bit sequences. The first 128 characters of all ASCII extensions are, therefore, identical to the original ASCII table. Extensions such as ISO Latin 9 (ISO 8859-15) contain language-specific characters (e.g., German umlauts) and special characters (e.g., the Euro symbol €), which are not included in the standard Latin alphabet.

schiedene ASCII-Erweiterungen für die verschiedenen Sprachen und Regionen.

Die Erweiterungen sind mit dem ursprünglichen US-ASCII kompatibel. Alle im US-ASCII definierten Zeichen werden in den verschiedenen Erweiterungen durch die gleichen Bitfolgen kodiert. Die ersten 128 Zeichen einer ASCII-Erweiterung sind also mit der ursprünglichen ASCII-Tabelle identisch. Die Erweiterungen wie zum Beispiel ISO Latin 9 (ISO 8859-15) enthalten sprachspezifische Zeichen (zum Beispiel Umlaute) und Sonderzeichen (zum Beispiel das Euro-Symbol €), die nicht im lateinischen Grundalphabet enthalten sind.

One disadvantage of these ASCII extensions is that not all extensions are available on all operating systems. If two communication partners do not use the identical extension, the special characters in the text are displayed incorrectly.

Ein Nachteil der ASCII-Erweiterungen ist, dass nicht in allen Betriebssystemen alle Erweiterungen verfügbar sind. Wenn zwei Kommunikationspartner nicht die identische Erweiterung verwenden, werden unter anderem die Sonderzeichen im Text falsch angezeigt.

2.5

Unicode

Unicode

In order to avoid problems caused by different character encodings, the multi-byte encoding Unicode (ISO 10646) was developed. It is extended continuously and is supposed to contain all known characters in the future.

Several Unicode standards exist. The most popular one is UTF-8. The first 128 characters are encoded with a single byte and are identical to US-ASCII. The encodings of the other characters use between 2 and 6 bytes. Currently, UTF-8 contains over 100,000 characters.

With UTF-8, each byte beginning with a `0`-bit corresponds to a 7-bit US-ASCII character. Each byte beginning with a `1`-bit belongs to a character encoded with multiple bytes. If a character that is encoded with UTF-8 consists of $n \geq 2$ bytes, the first byte begins with n `1`-bits, and each of the following $n - 1$ bytes begins with the bit sequence `10` (see Table 2.6).

Um die Probleme durch unterschiedliche Zeichenkodierungen zu vermeiden, wurde die Mehrbyte-Kodierung Unicode (ISO 10646) entwickelt. Diese wird laufend erweitert und soll in Zukunft alle bekannten Schriftzeichen enthalten.

Es existieren verschiedene Unicode-Standards. Am populärsten ist UTF-8. Die ersten 128 Zeichen werden mit einem Byte codiert und sind mit US-ASCII identisch. Die Kodierungen der anderen Zeichen verwenden zwischen 2 und 6 Bytes. Aktuell enthält UTF-8 über 100.000 Zeichen.

Bei UTF-8 entspricht jedes mit `0` beginnende Byte einem 7-Bit US-ASCII-Zeichen. Jedes mit `1` beginnende Byte gehört zu einem aus mehreren Bytes kodierten Zeichen. Besteht ein mit UTF-8 kodiertes Zeichen aus $n \geq 2$ Bytes, beginnt das erste Byte mit n Einsen und jedes der $n-1$ folgenden Bytes mit der Bitfolge `10` (siehe Tabelle 2.6).

Table 2.6: Multi-Byte Character Encoding with UTF-8

Code length	Bits for encoding	Format
1 byte	7 bits	`0xxxxxxx`
2 bytes	11 bits	`110xxxxx 10xxxxxx`
3 bytes	16 bits	`1110xxxx 10xxxxxx 10xxxxxx`
4 bytes	21 bits	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
5 bytes	26 bits	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
6 bytes	31 bits	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

2.6

Representation of Strings

For encoding continuous text, the individual characters are concatenated to a string. The text "`Betriebssysteme kompakt.`" becomes the following string.

```
B, e, t, r, i, e, b, s, s, y, s, t,
e, m, e,  , k, o, m, p, a, k, t, .
```

All characters (including space) are replaced by the decimal character numbers of the ASCII table.

```
066 101 116 114 105 101 098 115 115 121 115 116
101 109 101 000 107 111 109 112 097 107 116 046
```

Alternatively, the hexadecimal character numbers of the ASCII table can be specified.

```
42 65 74 72 69 65 62 73 73 79 73 74
65 6D 65 00 6B 6F 6D 70 61 6B 74 2E
```

The conversion of the characters to binary numbers results in their representation as a bit sequence.

```
01000010 01100101 01110100 01110010 01101001 01100101
01100010 01110011 01110011 01111001 01110011 01110100
01100101 01101101 01100101 00000000 01101011 01101111
01101101 01110000 01100001 01101011 01110100 00101110
```

Darstellung von Zeichenketten

Um einen fortlaufenden Text zu kodieren, werden die einzelnen Zeichen zu einer Zeichenkette (*String*) aneinandergefügt. Der Text „`Betriebssysteme kompakt.`“ wird zur nachstehenden Zeichenfolge.

Alle Zeichen (auch das Leerzeichen) werden durch die dezimalen Zeichennummern der ASCII-Tabelle ersetzt.

Alternativ kann man die hexadezimalen Zeichennummern der ASCII-Tabelle angeben.

Konvertiert man die Zeichennummern in Dualzahlen, erhält man die Repräsentation als Bitfolge.

3

Fundamentals of Operating Systems

Grundlagen der Betriebssysteme

Following a classification of operating systems in the field of computer science, this chapter describes basic concepts and distinction criteria of operating systems. These include the different operation methods batch mode and time-sharing, as well as single- and multitasking. It follows a description of the most relevant characteristics of real-time operating systems and distributed operating systems. Furthermore, a comparison of the various architectural concepts of operating system kernels takes place, and the basic structure of the operating systems is described using a layer model.

Nach einer Einordnung des Themas Betriebssysteme in die Informatik behandelt dieses Kapitel grundlegende Begriffe und Unterscheidungskriterien der Betriebssysteme. Dazu gehören die unterschiedlichen Betriebsarten Stapelbetrieb und Dialogbetrieb sowie Einzel- und Mehrprogrammbetrieb. Es folgt eine Beschreibung der wichtigsten Eigenschaften von Echtzeitbetriebssystemen sowie von verteilten Betriebssystemen. Im weiteren Verlauf des Kapitels werden die unterschiedlichen Architekturkonzepte von Betriebssystemkernen gegenübergestellt und der prinzipielle Aufbau der Betriebssysteme anhand eines Schichtenmodells dargestellt.

3.1

Operating Systems in Computer Science

Einordnung der Betriebssysteme in die Informatik

Computer science consists of four main pillars: practical, technical, theoretical computer science, and mathematics. Like any other subject, computer science is also influenced by other subjects. Studying computer science or a comparable field always includes at least one minor subject to satisfy personal interests and job market demands. Minor subjects are often electronic engineering, business administration/economics, or medicine, rarely geography, or linguistics. The list of minor subjects presented here is not complete. The same applies to the computer science topics shown in Figure 3.1.

Die Informatik gliedert sich in die vier Teilgebiete praktische, technische und theoretische Informatik sowie Mathematik. Wie jedes Fachgebiet ist auch die Informatik beeinflusst von anderen Fachgebieten. Ein Studium der Informatik oder eine vergleichbare Ausbildung berücksichtigen daher auch immer mindestens ein Nebenfach, um den persönlichen Neigungen und dem Arbeitsmarkt gerecht zu werden. Nebenfächer sind häufig E-Technik, BWL/VWL oder Medizin, seltener Geographie oder Sprachwissenschaften. Die an dieser Stelle präsentierte Auflistung der Nebenfächer hat keinen Anspruch auf Vollständigkeit. Das gleiche gilt für die in Abbildung 3.1 gezeigten Themen der Informatik.

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_3

Operating systems are entirely comprised of software. For this reason, the subjects of operating systems primarily include elements of practical computer science. Because one of the main tasks of operating systems is the control of the computers, the operating systems topic always includes elements of technical computer science, too.

Figure 3.1 gives an overview of the subjects of computer science and some of their topics.

Bei Betriebssystemen handelt es sich ausschließlich um Software. Darum umfasst das Thema Betriebssysteme primär Inhalte aus der praktischen Informatik. Da eine der Hauptaufgaben von Betriebssystemen die Steuerung der jeweiligen Computer ist, gehören zum Thema Betriebssysteme auch immer Inhalte der technischen Informatik.

Abbildung 3.1 zeigt eine Übersicht über die Teilgebiete der Informatik und dazugehöriger Themen.

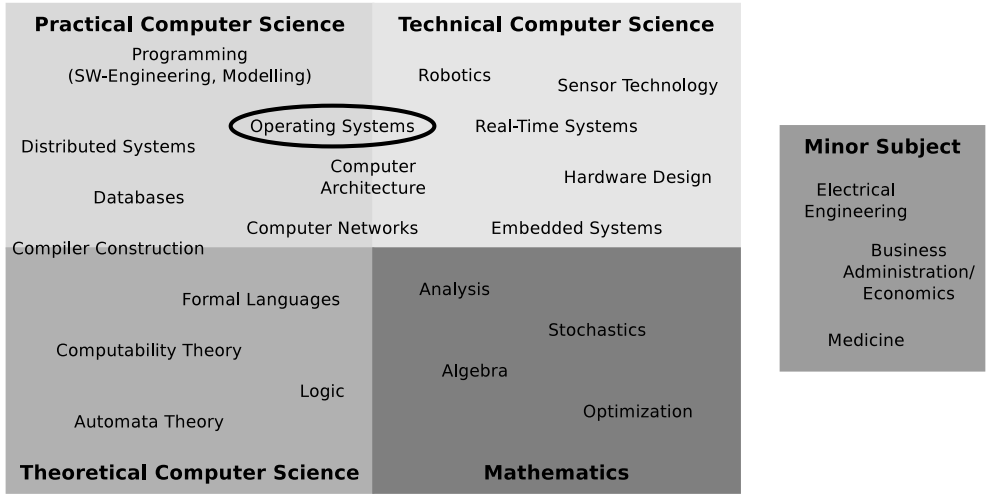

Figure 3.1: Operating Systems in Computer Science

3.2

Positioning and Core Functionalities of Operating Systems

Each computer has an operating system installed directly above the hardware layer (see Figure 3.2), which implements the essential features by using the operating system kernel and various system processes. These are processes that provide services for the operating system. From the hierarchical perspective, the user processes that process the users' tasks are positioned above the operating system.

Positionierung und Kernfunktionalitäten von Betriebssystemen

Auf jedem Computer ist direkt über der Hardware-Ebene ein Betriebssystem (siehe Abbildung 3.2) installiert, das mit Hilfe des Betriebssystemkerns (englisch: *Kernel*) und verschiedener Systemprozesse (das sind Prozesse, die Dienstleistungen für das Betriebssystem erbringen) die grundlegenden Kernfunktionalitäten anbietet. Aus hierarchischer Sicht über dem Betriebssystem befinden sich die Benutzerprozesse, die die Aufträge der Benutzer abarbeiten.

One exception to the principle that an operating system runs directly above the hardware layer is the paravirtualization concept (see Section 10.5). There, a hypervisor runs directly on the hardware, which allocates hardware resources to the guest systems.

Eine Ausnahme vom Grundsatz, dass direkt über der Hardware-Ebene ein Betriebssystem läuft, ist das Virtualisierungskonzept Paravirtualisierung (siehe Abschnitt 10.5). Dort läuft direkt auf der Hardware ein Hypervisor, der die Hardwareressourcen unter den Gastsystemen verteilt.

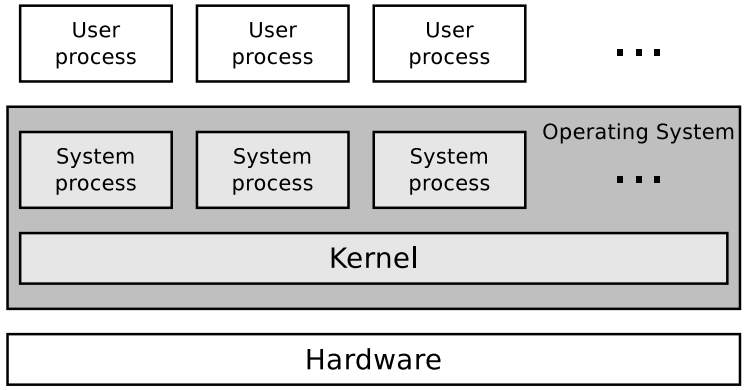

Figure 3.2: Position of the Operating System

An overview of the main features (tasks) of the operating systems is given in Figure 3.3. These are the management of ...

- hardware components of a computer
- data in different memory layers (cache, main memory, swap space, and storage drives) of the computer.
- processes and the provision of functions for interprocess communication.
- different users and user groups.

Eine Übersicht über die Kernfunktionalitäten (Aufgabenbereiche) der Betriebssysteme zeigt Abbildung 3.3. Diese sind die Verwaltung der ...

- Hardwarekomponenten eines Computers.
- Daten in den unterschiedlichen Datenspeichern (Cache, Hauptspeicher, Auslagerungsspeicher und Speicherlaufwerke) des Computers.
- Prozesse und die Bereitstellung von Funktionalitäten zur Interprozesskommunikation.
- unterschiedlichen Benutzer und Benutzergruppen.

Consequently, most of the core functionalities of the operating systems, described in this section and shown in Figure 3.3, are also the subjects this book describes. Basic knowledge about the hardware components, which are controlled by the operating system, is provided in Chapter 4. The different ways of memory man-

Konsequenterweise sind die allermeisten in diesem Abschnitt beschriebenen und in Abbildung 3.3 gezeigten Kernfunktionalitäten der Betriebssysteme auch die Themen, die dieses Buch vermitteln will. Grundlegendes Wissen zu den Hardwarekomponenten, die durch die Betriebssysteme verwaltet werden, vermittelt Kapitel 4.

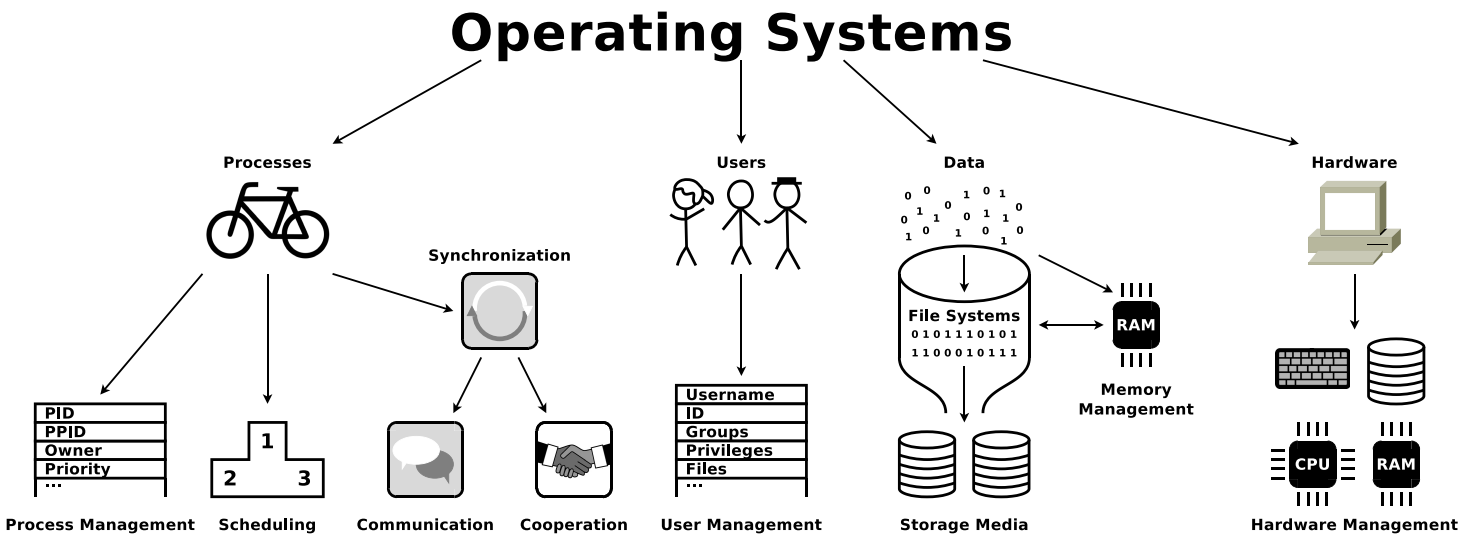

Figure 3.3: Essential Features of Operating Systems

agement are described in the Chapters 5 and 6. Process management and the various forms of interprocess communication are discussed in the chapters 7, 8, and 9.

Die verschiedenen Formen der Speicherverwaltung sind die Inhalte der Kapitel 5 und 6. Die Verwaltung der Prozesse und die unterschiedlichen Formen der Interprozesskommunikation vermitteln die Kapitel 7, 8 und 9.

3.3

Evolution of Operating Systems

Entwicklung der Betriebssysteme

The list of computer system generations, whose chronological order is less strict than Table 3.1 suggests, shows why the first operating systems were not developed before the 1950s.

Die Auflistung der Generationen von Computersystemen, deren zeitliche Grenzen unschärfer sind, als es Tabelle 3.1 vermuten lässt, macht deutlich, warum es erst ab den 1950er Jahren erste Betriebssysteme gab.

Table 3.1: Generations of Computer Systems

Gen.	Time period	Technological progress
0	until 1940	(Electro-)mechanical calculating machines
1	1940 – 1955	Electron tubes, relays, jack panels
2	1955 – 1965	Transistors, batch processing
3	1965 – 1980	Integrated circuits, time-sharing
4	1980 – 2000	Very large-scale integration, PCs/Workstations
5	2000 until ?	Distributed systems, mobile systems, multi-core CPUs, virtualization

Some of the requirements for a computer, i.e., for a universal electronic calculating machine, are stored programs, conditional jumps, and a separation of memory and CPU.

The first generation of computer systems was developed during the second world war. Konrad Zuse, in 1941, constructed the Z3 in Germany, which was the first working computer ever. Other first-generation computers were developed during the 1940s in the USA (e.g., Atanasoff-Berry computer, Mark I, ENIAC) and England (Colossus). Computers of this generation were machines with sometimes more than 10,000 electron tubes or relays, which were slow and error-prone compared to modern computers. For these reasons, these computers were not used for universal purposes, but only for specialized tasks such as flight path calculations or cryptanalysis. Operating systems and also programming languages were still unknown at that time, and the programs were implemented

Zu den Anforderungen an einen Computer, also an eine universelle, elektronische Rechenmaschine gehören: Gespeicherte Programme, bedingte Sprünge und die Trennung von Speicher und Prozessor.

Die erste Generation von Computersystemen entstand während des zweiten Weltkriegs. Konrad Zuse konstruierte 1941 in Deutschland mit der Z3 den ersten funktionsfähigen Computer der Welt. Andere frühe Computer der ersten Generation entstanden im weiteren Verlauf der 1940er Jahre in den USA (z.B. Atanasoff-Berry-Computer, Mark I, ENIAC) sowie in England (Colossus). Computer dieser Generation waren Maschinen mit teilweise mehr als 10.000 Röhren oder Relais, die im Vergleich zu modernen Computern langsam und fehleranfällig arbeiteten. Computer dieser Generation wurden aus den genannten Gründen nicht für universelle Zwecke, sondern nur für spezielle Aufgaben wie zum Beispiel Flugbahnberechnungen oder zur Kryptoanalyse verwendet. Betriebssysteme und

by the users (programmers) directly on the hardware, using patch bays.

In the 1940s, Konrad Zuse developed the programming language *Plankalkül* in addition to his computers. Since the language was not practically used (the first compilers for it were developed decades later), it is mainly of historical relevance.

3.3.1 Second Generation of Computers

In the early 1950s, punch cards (see Figure 3.4) replaced the patch panels. Each punch card usually represents one line of program code with 80 characters or the corresponding amount of binary data. The fact that the line length of e-mails and text files today typically is 80 characters is due to the punch card standard. 12 hole positions are used for the encoding of each character. Numbers are encoded with a single hole in the corresponding line. Letters and special characters are encoded by punching multiple holes in the column.

Following the introduction of transistors in the mid-1950s, computer systems became more reliable. On computers of the second generation, programs in the programming languages Fortran or COBOL were written on form sheets by programmers, punched by coders into punch cards, and then handed over to an operator. The operator coordinated the order of the programs, equipped the computer with the punch cards, ensured that the compiler was loaded from a magnetic tape, and handed over the program output as a printout to the client. This way of software development and program execution was inefficient due to the numerous intermediate steps and the idle time that occurred during user and process changes.

Collecting the programs on magnetic tapes resulted in more efficient program execution because this accelerated the reading process. Operating systems used in this second generation

auch Programmiersprachen waren zu jener Zeit noch unbekannt und die Programme wurden von den Benutzer (Programmierern) über Steckfelder gesteckt, also direkt in der Hardware implementiert.

In den 1940er Jahren entwickelte Konrad Zuse zusätzlich zu seinen Computern die Programmiersprache *Plankalkül*. Da die Sprache praktisch nicht eingesetzt wurde (Compiler dafür wurden erst Jahrzehnte später entwickelt), hat sie primär historische Bedeutung.

3.3.1 Zweite Generation von Computern

Ab Anfang der 1950er Jahre lösten Lochkarten (siehe Abbildung 3.4) die Steckfelder ab. Jede Lochkarte stellt üblicherweise eine Zeile Programmtext mit 80 Zeichen oder entsprechend viele binäre Daten dar. Dass die Zeilenlänge von E-Mails und Textdateien heute noch typischerweise 80 Zeichen beträgt, geht auf die Lochkarte zurück. 12 Lochpositionen stehen für die Kodierung jedes Zeichens zur Verfügung. Ziffern kodiert man mit einem einzelnen Loch in der entsprechenden Zeile. Buchstaben und Sonderzeichen kodiert man, indem mehrere Löcher in die Spalte gestanzt werden.

Durch die Einführung der Transistoren ab Mitte der 1950er Jahre wurden die Computersysteme zuverlässiger. Auf Computern dieser zweiten Generation wurden Programme in den Programmiersprachen Fortran oder COBOL von Programmierern auf Formblätter aufgeschrieben, von Eingebern bzw. Codierern in Lochkarten gestanzt und anschließend einem Operator übergeben. Der Operator koordinierte die Reihenfolge der Programme, bestückte den Computer mit den Lochkarten, sorge dafür, dass der Compiler von einem Magnetband geladen wurde und übergab die Programmausgabe als Ausdruck an den Auftraggeber. Diese Form der Softwareentwicklung und Programmausführung war bedingt durch die zahlreichen Zwischenschritte und den beim Benutzer- und Prozesswechsel entstehenden Leerlauf ineffizient.

Das Sammeln der Programme auf Magnetbändern führte zu einer effizienteren Programmausführung, weil so das Einlesen beschleunigt wurde. Betriebssysteme, die in dieser zweiten Generation verwendet wurden, ermöglich-

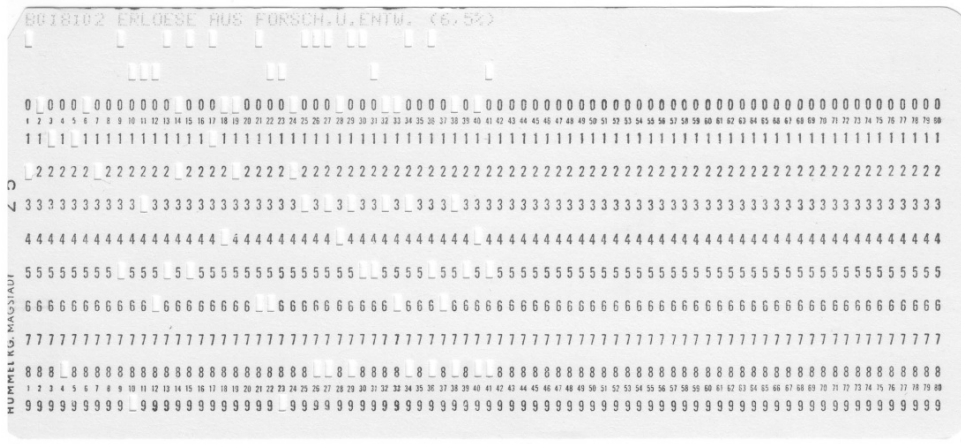

Figure 3.4: Example of a Punch Card

tion only implemented *batch processing* (see Section 3.4.1) and *singletasking* (see Section 3.4.2).

ten ausschließlich *Stapelverarbeitung* (siehe Abschnitt 3.4.1) und *Einzelprogrammbetrieb* (siehe Abschnitt 3.4.2).

3.3.2

Third Generation of Computers

From the early 1960s on, integrated circuits allowed the production of more powerful, smaller, and cheaper computers. The operating systems for batch processing were improved in the 1960s to process multiple jobs at the same time. This feature is called *multitasking* (see Section 3.4.2), and one of its preconditions is the existence of at least a simple form of *memory management*. (see Chapter 5).

One task of memory management is *memory protection* (see Chapter 5.3.2). For this purpose, the main memory is split into smaller parts to separate programs from each other while they are running. As a result, bugs or crashes of single programs do not affect the stability of other programs or the overall system.

Further important features of operating systems that were developed for the third generation of computer systems are:

- *File systems* (see Chapter 6) that allow quasi-simultaneous file access and organize the way data is stored on the drives.

Dritte Generation von Computern

Ab den frühen 1960er Jahren ermöglichten integrierte Schaltungen leistungsfähigere, kleinere und auch in der Herstellung billigere Computer. Die Betriebssysteme zur Stapelverarbeitung wurden in den 1960er Jahren dahingehend erweitert, dass sie mehrere Aufträge gleichzeitig abarbeiten konnten. Diese Fähigkeit heißt *Mehrprogrammbetrieb* (siehe Abschnitt 3.4.2) und erforderte erstmals eine einfache *Speicherverwaltung* (siehe Kapitel 5).

Eine Aufgabe der Speicherverwaltung ist der sogenannte *Speicherschutz* (siehe Kapitel 5.3.2). Dabei wird der Arbeitsspeicher aufgeteilt, um laufende Programme voneinander zu trennen. Dadurch beeinträchtigen Programmierfehler oder der Absturz einzelner Programme nicht die Stabilität anderer Programme oder des Gesamtsystems.

Weitere wichtige Funktionalitäten von Betriebssystemen, die für die dritte Generation von Computersystemen entwickelt wurden, sind:

- *Dateisysteme* (siehe Kapitel 6), die quasigleichzeitige Dateizugriffe erlauben und

die Art und Weise der Datenspeicherung auf den Laufwerken organisieren.

- *Swapping* (see Chapter 7), i.e., the process of storage and retrieval of data into/from main memory from/into swap space (usually on hard disk drives or SSDs).

- *Swapping* (siehe Kapitel 7), also das Ein- und Auslagern von Daten in den/vom Arbeitsspeicher vom/in den Auslagerungsspeicher (meist auf Festplatten oder SSDs).

- *Scheduling* (see Section 8.6), is the automatic creation of an execution plan (*schedule*), which is used to allocate time-limited resources to users or processes.

- *Scheduling* (siehe Abschnitt 8.6), also die automatische Erstellung eines Ablaufplanes (englisch: *Schedule*), der Benutzern bzw. Prozessen zeitlich begrenzte Ressourcen zuteilt.

During the 1970s, computers were enhanced so that several users were able to work simultaneously on a mainframe via *terminals*. This working method is called *time-sharing* (see Section 3.4.1).

The development of the microprocessor at the end of the 1970s led to the development of the home computer (e.g., Apple II) and the IBM personal computer (PC) in the early 1980s. Further well-known computer systems of the third generation are CDC 6600, IBM System/360, DEC PDP-8, and CRAY 1.

Some operating systems for third generation computer systems are IBM OS/360, Multics (the predecessor of Unix), Unics (later Unix) from the Bell Laboratories, DEC VMS for DEC VAX computers, and version 6/7 Unix.

Während der 1970er Jahre wurden die Computer dahingehend ausgebaut, dass mehrere Benutzer gleichzeitig über Dialogstationen (*Terminals*) an einem Großrechner arbeiten konnten. Diese Form der Arbeit heißt *Dialogbetrieb* (siehe Abschnitt 3.4.1).

Die Entwicklung des Mikroprozessors Ende der 1970er Jahre führte in den frühen 1980er Jahren zur Entwicklung der Heimcomputer (z.B. Apple II) und des IBM Personal Computers (PC). Weitere bekannte Vertreter der dritten Generation sind CDC 6600, IBM System/360, DEC PDP-8 und CRAY 1.

Einige Betriebssysteme aus der dritten Generation von Computersystemen sind IBM OS/360, Multics (der Vorgänger von Unix), Unics (später Unix) aus den Bell Laboratories, DEC VMS für DEC VAX-Rechner und Version 6/7 Unix.

3.3.3

Fourth Generation of Computers

From the early 1980s onwards, the growing performance of CPUs, and increasing memory capacity combined with decreasing acquisition costs, led to the establishment of personal computers and workstations in private contexts, as well as university and business areas. One important objective of operating systems of this generation is the provision of intuitive user interfaces for users who want to know little or nothing about the underlying hardware.

Vierte Generation von Computern

Die zunehmende Leistungsfähigkeit der Prozessoren und zunehmende Speicherkapazität bei sinkenden Anschaffungskosten führten ab den frühen 1980er Jahren zu einer Etablierung der Personal Computer und Workstations im privaten, universitären und unternehmerischen Umfeld. Eine zunehmend wichtige Aufgabe von Betriebssystemen dieser Generation ist die Bereitstellung intuitiver Benutzeroberflächen für die Benutzer, die immer weniger von der zu Grunde liegenden Hardware wissen wollen.

Some fourth generation operating systems are disk operating systems such as QDOS (*Quick and Dirty Operating System*), MS-DOS, IBM PC-DOS, Apple DOS, and Atari DOS. Further examples are desktop operating systems like AmigaOS, Atari TOS, Windows, and Mac OS, as well as Unix-like systems such as Microsoft Xenix, SGI IRIX, SunOS, IBM AIX, NeXTSTEP, the GNU project and Linux.

Einige Betriebssysteme der vierten Generation sind Disk Operating Systeme wie QDOS (*Quick and Dirty Operating System*), MS-DOS, IBM PC-DOS, Apple DOS und Atari DOS. Weitere Beispiele sind Desktop-Betriebssysteme wie AmigaOS, Atari TOS, Windows und Mac OS sowie Unix-ähnliche Systeme wie Microsoft Xenix, SGI IRIX, SunOS, IBM AIX, NeXTSTEP, das GNU-Projekt und Linux.

3.4

Operating Modes

Operating systems can be classified according to the operating modes batch processing and time-sharing, single-user and multi-user mode, as well as singletasking and multitasking.

Betriebsarten

Die Betriebssysteme lassen sich anhand der Betriebsarten Stapelbetrieb und Dialogbetrieb, Einzelprogrammbetrieb und Mehrprogrammbetrieb sowie Einzelbenutzerbetrieb und Mehrbenutzerbetrieb klassifizieren.

3.4.1

Batch Processing and Time-sharing

In the *batch processing* mode, each program must be provided complete with all input data before the execution can begin. Even modern computer systems allow batch processing, for example, in the form of batch files or shell scripts. This form of batch processing is a useful tool, especially for the processing of routine tasks. Batch processing usually works in a non-interactive way. A started process is executed without any user interaction until it terminates or an error occurs. One objective of batch processing is to maximize CPU utilization.

Figure 3.5 shows the acceleration by automation that batch processing provides. In batch processing, the user change does not waste computing resources. However, even with this variant of batch operation, the CPU is still not optimally utilized, because it is in idle state during input/output operations.

The system shown in Figure 3.6 is an example of a sophisticated IT system from the 1960s with batch processing and optimal computing power utilization. In this scenario, a frontend and

Stapelbetrieb und Dialogbetrieb

Stapelverarbeitung (englisch: *Batch Processing*) heißt auch *Stapelbetrieb* oder *Batchbetrieb*. Bei dieser Betriebsart muss jedes Programm mit allen Eingabedaten vollständig vorliegen, bevor die Abarbeitung beginnen kann. Auch heutige Systeme ermöglichen Stapelverarbeitung, zum Beispiel in Form von Batch-Dateien oder Shell-Skripten. Speziell zur Ausführung von Routineaufgaben ist diese Form des Stapelbetriebs ein nützliches Werkzeug. Üblicherweise ist Stapelbetrieb interaktionslos. Nach dem Start eines Programms wird dieses bis zum Ende oder Auftreten eines Fehlers ohne Interaktion mit dem Benutzer abgearbeitet. Ein Ziel des Stapelbetriebs ist die maximale Prozessorausnutzung.

Abbildung 3.5 zeigt die Beschleunigung durch Automatisierung, die Stapelbetrieb ermöglicht. Durch den Stapelbetrieb geht keine Rechenleistung durch Benutzerwechsel verloren. Allerdings wird auch bei dieser dargestellten Variante des Stapelbetriebs der Hauptprozessor noch nicht optimal ausgenutzt, denn dieser liegt während der Ein-/Ausgabe brach.

Das in Abbildung 3.6 dargestellte System ist ein Beispiel für ein komplexes System zur Datenverarbeitung aus den 1960er Jahren mit Stapelbetrieb und optimaler Ausnutzung der Re-

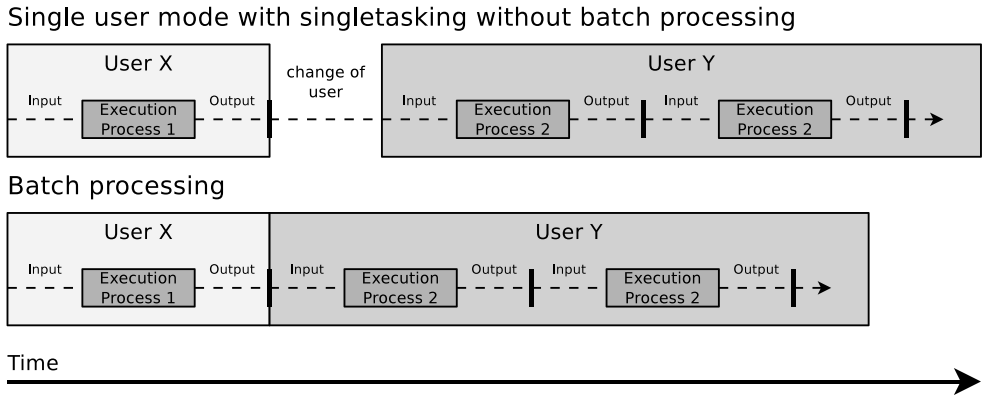

Figure 3.5: Comparison of Single-user Mode with Singletasking, without and with Batch Processing [119]

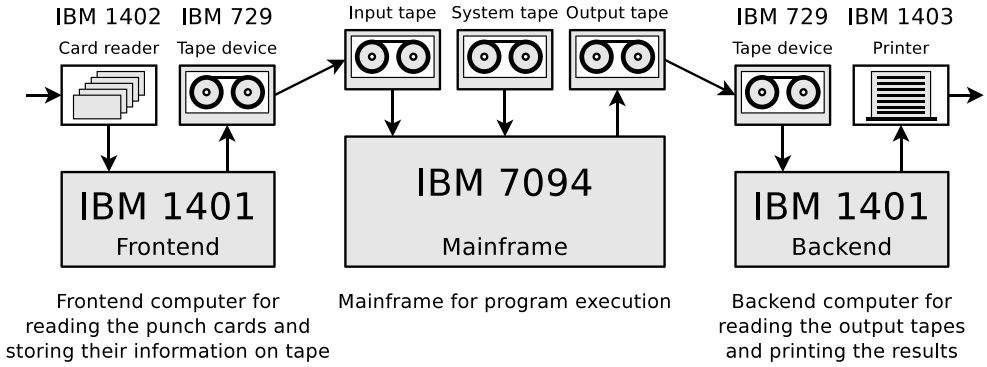

Figure 3.6: Example of an IT System with Batch Processing and optimal Utilization of Computing Power [112]

a backend both support the mainframe. The frontend stores the programs of the users on magnetic tape. The backend reads the program outputs from magnetic tape and prints them. The read performance of magnetic tape is better in comparison to punch cards, and data can be stored much faster on tape than printed. In effect, the time-consuming input/output tasks were taken away from the mainframe and handled by the frontend and the backend instead. Removing input/output workload from the CPU by using additional hardware is called *spooling*. By spooling, the input/output is done concurrently with the processing of the other tasks.

chenleistung. In diesem Szenario unterstützten ein Vorrechner und ein Nachrechner den eigentlichen Großrechner (Hauptrechner). Der Vorrechner schreibt die gesammelten Programme der Benutzer auf Magnetband. Der Nachrechner liest die gesammelten Ausgaben der Programme vom Magnetband und druckt diese aus. Von Band kann schneller eingelesen werden als von Lochkarten, und auf Band kann schneller geschrieben werden als auf Papier. Vorrechner und Nachrechner befreien den Großrechner somit von langsamer Ein-/Ausgabearbeit. Die Entlastung des Hauptprozessors durch zusätzliche Hardware für Ein-/Ausgabeoperationen ist unter dem Begriff *Spooling* bekannt. Durch Spoo-

Today, even modern computers have, in addition to the CPU, specific I/O processors with DMA (*Direct Memory Access*) capability. These write data directly into the main memory and fetch the results from there. There are also so-called *spooling processes* for printing.

In *time-sharing* (*interactive mode*), multiple users can work on a computer system simultaneously and competitively by sharing the available computing resources of the CPU. A challenging aspect of this operation mode is the fair distribution of the computing time. It is done with the help of *time slices*. The time slices can be allocated to the programs according to different scheduling methods (see Section 8.6).

Although the users work concurrently and interactively via terminals on a computer, their programs are independent of each other due to the multitasking (see Section 3.4.2) mode.

Auch moderne Computersysteme haben neben dem Hauptprozessor spezielle, DMA-fähige (*Direct Memory Access*) Ein-/Ausgabeprozessoren. Diese schreiben Daten direkt in den Hauptspeicher und holen von dort die Ergebnisse. Zudem existieren sogenannte *Spoolingprozesse* zum Drucken.

Bei der Betriebsart *Dialogbetrieb* (englisch: *Time-Sharing*), die auch *Zeitteilbetrieb* heißt, arbeiten mehrere Benutzer an einem Computersystem gleichzeitig und konkurrierend, indem sie sich die verfügbare Rechenzeit des Hauptprozessors teilen. Eine Herausforderung hierbei ist die faire Verteilung der Rechenzeit. Dieses geschieht mit Hilfe von *Zeitscheiben* (englisch: *Time Slices*). Die Verteilung der Zeitscheiben an die Programme kann nach unterschiedlichen Scheduling-Verfahren (siehe Abschnitt 8.6) erfolgen.

Obwohl die Benutzer gleichzeitig über Terminals an einem Computer interaktiv arbeiten, sind deren Programme durch den Mehrprogrammbetrieb (siehe Abschnitt 3.4.2) unabhängig voneinander.

3.4.2

Singletasking and Multitasking

The *singletasking* mode only allows the execution of one program at a time.

The technical term *task* means the same as the term *process*, or *job*, from the user's point of view.

Quasi-parallel program or process execution is called *multitasking*. Multiple programs can be executed at the same time (with multiple CPUs or CPU cores) or in a quasi-parallel way. The processes are activated alternately at short intervals. This creates the impression of concurrency. One drawback of multitasking is the overhead that is caused by process switching.

Multitasking is useful despite the overhead because processes often need to wait for external events. Reasons for this include, for example, in-

Einzelprogrammbetrieb und Mehrprogrammbetrieb

Beim *Einzelprogrammbetrieb* (englisch: *Singletasking*) läuft zu jedem Zeitpunkt nur ein einziges Programm.

Der Begriff *Task* ist gleichzusetzen mit *Prozess* oder aus Anwendersicht *Aufgabe* bzw. *Auftrag*.

Die quasi-parallele Programm- bzw. Prozessausführung heißt *Mehrprogrammbetrieb* (englisch: *Multitasking*). Mehrere Programme können gleichzeitig (bei mehreren Prozessoren bzw. Rechenkernen) oder zeitlich verschachtelt (quasiparallel) ausgeführt werden. Die Prozesse werden in kurzen Abständen abwechselnd aktiviert. Dadurch entsteht der Eindruck der Gleichzeitigkeit. Ein Nachteil des Mehrprogrammbetriebs ist das Umschalten von Prozessen, welches einen Verwaltungsaufwand (Overhead) verursacht.

Mehrprogrammbetrieb ist trotz des Verwaltungsaufwand sinnvoll, denn Prozesse müssen häufig auf äußere Ereignisse warten.

put or output operations of users or waiting for a message from another program. Through multiprogram operation, processes that wait for incoming e-mails, successful database operations, data to be written to the hard disk drive, or something similar, can be placed into the background, and other processes can be executed sooner.

The overhead that is caused by the quasi-parallel execution of programs due to program switching is insignificant compared to the performance gain.

3.4.3 Single-user and Multi-user

In *single-user mode*, the computer can only be used by one user at a time. Several single-user operating systems with single- and multitasking functionality exist. Examples of operating systems that only offer single-user operation are MS-DOS, Microsoft Windows 3x/95/98, and OS/2.

In *multi-user mode*, multiple users can work with the computer at the same time. The users share the resources of the system. They must be shared as fairly as possible by using appropriate scheduling methods (see Section 8.6). The individual users must be identified (by passwords), and the operating system must prevent attempts to access data and processes of different users. Examples of operating systems that provide multi-user operation are Linux and other Unix-like systems, Mac OS X, and the server editions of the Microsoft Windows NT family. These include *Terminal Server* and *MultiPoint Server* editions.

Table 3.2 shows a classification of popular operating systems into the categories singletasking, multitasking, single-user, and multi-user operation mode.

Particularly interesting is the so-called *half multi-user operating systems*. The

Gründe sind zum Beispiel Benutzereingaben, Eingabe/Ausgabe-Operationen von Peripheriegeräten oder das Warten auf eine Nachricht eines anderen Programms. Durch Mehrprogrammbetrieb können Prozesse, die auf ankommende E-Mails, erfolgreiche Datenbankoperationen, geschriebene Daten auf der Festplatte oder ähnliches warten, in den Hintergrund geschickt werden und andere Prozesse kommen früher zum Einsatz.

Der Verwaltungsaufwand, der bei der quasiparallelen Abarbeitung von Programmen durch die Programmwechsel entsteht, ist im Vergleich zum Geschwindigkeitszuwachs zu vernachlässigen.

Einzelbenutzerbetrieb und Mehrbenutzerbetrieb

Beim *Einzelbenutzerbetrieb* (englisch: *Single-User Mode*) steht der Computer immer nur einem einzigen Benutzer zur Verfügung. Es existieren Single-User-Betriebssysteme mit Single- und mit Multitasking-Funktionalität. Beispiele für Betriebssysteme, die ausschließlich Einzelbenutzerbetrieb bieten, sind MS-DOS, Microsoft Windows 3x/95/98 und OS/2.

Beim *Mehrbenutzerbetrieb* (englisch: *Multi-User Mode*) können mehrere Benutzer gleichzeitig mit dem Computer arbeiten. Die Benutzer teilen sich hierbei die Systemleistung. Die Systemressourcen müssen mit Hilfe geeigneter Scheduling-Methoden (siehe Abschnitt 8.6) möglichst gerecht verteilt werden. Die verschiedenen Benutzer müssen (durch Passwörter) identifiziert und Zugriffe auf Daten und Prozesse anderer Benutzer durch das Betriebssystem verhindert werden. Beispiele für Betriebssysteme, die Mehrbenutzerbetrieb ermöglichen, sind Linux und andere Unix-ähnliche Systeme, Mac OS X, sowie die Server-Versionen der Microsoft Windows NT-Familie. Dazu gehören auch die Versionen *Terminal Server* und *MultiPoint Server*.

Tabelle 3.2 enthält eine Einordnung bekannter Betriebssysteme in die Betriebsarten Einzelprogrammbetrieb und Mehrprogrammbetrieb sowie Einzelbenutzerbetrieb und Mehrprogrammbetrieb.

Eine Besonderheit sind die sogenannten *halben Multi-User-Betriebssysteme*. Zu die-

Table 3.2: Classification of popular operating systems into different operation modes

	Single-user	Multi-user
Singletasking	MS-DOS, FreeDOS, PC-DOS, DR-DOS, GEOS, Palm OS	—
Multitasking	Windows 3x/95/98, Mac OS 8x/9x, BeOS, Haiku, OS/2, ArcaOS, eComStation, Risc OS, AmigaOS, KolibriOS, MenuetOS, Plan 9	Linux/UNIX, Minix, FreeBSD, OpenBSD, NetBSD, DragonFly BSD, Mac OS X, OpenIndiana, Solaris, ReactOS, Server editions of the Windows NT family

desktop/workstation editions of Microsoft Windows NT/2000/XP/Vista/7/8/10/11 belong to this category. When using these operating systems, different users can only work on the system one after the other, but the data of these users are protected from each other. With inofficial extensions, it is also possible to extend the desktop/workstation editions so that multiple users can simultaneously log on to the system via the Remote Desktop Protocol (RDP).

ser Kategorie gehören beispielsweise die Desktop/Workstation-Versionen von Microsoft Windows NT/2000/XP/Vista/7/8/10/11. Bei diesen Betriebssystemen können verschiedene Benutzer nur nacheinander am System arbeiten, aber die Daten der verschiedenen Benutzer sind voreinander geschützt. Mit inoffiziellen Erweiterungen ist es auch möglich die Desktop/Workstation-Versionen dahingehend zu erweitern, dass mehrere Benutzer sich gleichzeitig via Remote Desktop Protocol (RDP) am System anmelden können.

3.5

8/16/32/64-Bit Operating Systems

Each operating system internally uses memory addresses of a certain length. Modern operating systems are usually 64-bit operating systems. Many Linux distributions, as well as Microsoft Windows, additionally are available as 32-bit operating systems, especially for older hardware. An operating system can only address as many memory units as the address space is capable of. Therefore, a 64-bit operating system can address more memory than a 32-bit operating system. However, the size of the address space depends, in terms of the hardware, on the address bus (see Section 4.1.3).

8-bit operating systems can address 2^8 memory units. Examples of such operating systems are GEOS, Atari DOS, or Contiki [26].

8/16/32/64-Bit-Betriebssysteme

Jedes Betriebssystem arbeitet intern mit Speicheradressen einer bestimmten Länge. Moderne Betriebssysteme werden üblicherweise als 64-Bit-Betriebssysteme angeboten. Zahlreiche Linux-Distributionen sowie Microsoft Windows sind speziell für ältere Hardware zusätzlich noch als 32-Bit-Betriebssysteme verfügbar. Ein Betriebssystem kann nur so viele Speichereinheiten ansprechen, wie der Adressraum zulässt. Darum kann ein 64-Bit-Betriebssystem mehr Speicher ansprechen als ein 32-Bit-Betriebssystem. Die Größe des Adressraums hängt hardwareseitig allerdings vom Adressbus (siehe Abschnitt 4.1.3) ab.

8-Bit-Betriebssysteme können 2^8 Speichereinheiten adressieren. Beispiele für solche Betriebssysteme sind GEOS, Atari DOS oder Contiki [26].

16-bit operating systems can address 2^{16} memory units. Examples of such operating systems are MS-DOS, Windows 3x, and OS/2 1x.

32-bit operating systems can address 2^{32} memory units. Examples of such operating systems are Windows 95/98/NT/Vista/7/8/10, OS/2 2/3/4, eComStation, BeOS, Linux, and Mac OS X until revision 10.7.

64-bit operating systems can address 2^{64} memory units. Examples of such operating systems are Windows 7/8/10 (64-bit), Windows 11, Linux (64-bit), and Mac OS X (64-bit).

16-Bit-Betriebssysteme können 2^{16} Speichereinheiten adressieren. Beispiele für solche Betriebssysteme sind MS-DOS, Windows 3x und OS/2 1x.

32-Bit-Betriebssysteme können 2^{32} Speichereinheiten adressieren. Beispiele für solche Betriebssysteme sind Windows 95/98/NT/Vista/7/8/10, OS/2 2/3/4, eComStation, BeOS, Linux und Mac OS X bis einschließlich 10.7.

64-Bit-Betriebssysteme können 2^{64} Speichereinheiten adressieren. Beispiele für solche Betriebssysteme sind Windows 7/8/10 (64 Bit), Windows 11, Linux (64 Bit) und Mac OS X (64 Bit).

3.6

Real-Time Operating Systems

Real-time operating systems are multitasking operating systems with additional real-time features for the compliance of time conditions. The essential quality criteria of real-time operating systems are reaction time and the ability to adhere to deadlines.

Existing real-time operating systems can be classified into *hard real-time operating systems* and *soft real-time operating systems*.

Echtzeitbetriebssysteme

Echtzeitbetriebssysteme sind Betriebssysteme die Mehrprogrammbetrieb mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen bieten. Wesentliche Qualitätskriterien von Echtzeitbetriebssystemen sind die Reaktionszeit und die Einhaltung von Zeitschranken (englisch: *Deadlines*).

Die existierenden Echtzeitbetriebssysteme können in die beiden Gruppen *harte Echtzeitbetriebssysteme* und *weiche Echtzeitbetriebssysteme* unterschieden werden.

3.6.1

Hard and Soft Real-Time Operating Systems

Hard real-time operating systems must strictly adhere to deadlines. Delays cannot be accepted under any circumstances, as they can lead to catastrophic consequences and high costs. The results of a process execution may be useless if they are available too late. Some examples of applications for hard real-time operating systems, where the need to adhere to time constraints is evident, include welding robots, reactor control systems, vehicle anti-lock braking systems, aircraft flight control systems, and intensive care monitoring systems. Examples of hard real-time operating systems are QNX4, VxWorks, LynxOS, and the Linux extension

Harte und weiche Echtzeitbetriebssysteme

Harte Echtzeitbetriebssysteme müssen Zeitschranken unbedingt einhalten. Verzögerungen können unter keinen Umständen akzeptiert werden, denn sie können zu katastrophalen Folgen und hohen Kosten führen. Die Ergebnisse einer Prozessausführung sind unter Umständen nutzlos, wenn die Bearbeitung des Prozesses zu spät erfolgt. Einige Einsatzbeispiele für harte Echtzeitbetriebssysteme, bei denen die Notwendigkeit der Einhaltung von Zeitschranken eindeutig ist, sind Schweißroboter, Systeme in der Reaktorsteuerung, Antiblockiersysteme bei Fahrzeugen, Systeme zur Flugzeugsteuerung und Überwachungssysteme auf der Intensivstati-

RTLinux [127], which is a real-time microkernel that runs the entire Linux operating system as a process alongside the real-time processes.

For soft real-time operating systems, certain tolerances are acceptable when adhering to deadlines. Delays lead to acceptable costs. Some examples of applications for soft real-time operating systems are telephone systems, parking ticket or ticket vending machines, or multimedia applications such as audio/video on demand.

Soft real-time behavior for processes with high priority is guaranteed by all modern desktop operating systems such as Microsoft Windows, Apple Mac OS X, or Linux. However, these operating systems cannot guarantee hard real-time behavior due to unpredictable events such as swapping (see Section 5.3.2) or interrupts caused by hardware components.

on. Beispiele für harte Echtzeitbetriebssysteme sind QNX4, VxWorks, LynxOS und die Linux-Erweiterung RTLinux [127]. Dabei handelt es sich um einen Echtzeit-Mikrokern, der das komplette Linux-Betriebssystem als einen Prozess neben den Echtzeitprozessen betreibt.

Bei weichen Echtzeitbetriebssystemen sind gewisse Toleranzen bei der Einhaltung von Zeitschranken erlaubt. Verzögerungen führen zu akzeptablen Kosten. Einige Einsatzbeispiele für weiche Echtzeitbetriebssysteme sind Telefonanlagen, Parkschein- oder Fahrkartenautomaten oder Multimedia-Anwendungen wie Audio/Video on Demand.

Weiches Echtzeitverhalten können alle aktuellen Desktop-Betriebssysteme wie zum Beispiel Microsoft Windows, Apple Mac OS X oder Linux für Prozesse mit hoher Priorität garantieren. Wegen des unberechenbaren Zeitverhaltens durch Swapping (siehe Abschnitt 5.3.2), Unterbrechungen (englisch: *Interrupts*) durch Hardwarekomponenten, etc. kann von diesen Betriebssystemen aber kein hartes Echtzeitverhalten garantiert werden.

3.6.2 Architectures of Real-Time Operating Systems

Real-time operating systems can also be categorized according to their architecture.

In real-time operating systems that implement the *Thin kernel* architecture, the operating system kernel itself runs as a process with the lowest priority. The real-time kernel does the scheduling, and real-time processes are executed with the highest priority. The purpose of this strategy is to minimize reaction time.

If a real-time operating system has a so-called *Nano kernel*, further operating system kernels can be executed in addition to the real-time kernel (see Figure 3.7).

The technical terms *Pico kernel*, *Femto kernel*, and *Atto kernel* are examples of marketing terms used by vendors of real-time systems to emphasize the compact size of their real-time kernels. The size of the operating system kernel is generally seen as a quality factor since the reduction of the real-time kernel to the essential

Architekturen von Echtzeitbetriebssystemen

Neben der Unterscheidung anhand der Einhaltung von Zeitschranken, können Echtzeitbetriebssysteme auch anhand Ihrer Architektur in verschiedene Gruppe eingeteilt werden.

Bei Echtzeitbetriebssystemen mit der Architektur *Thin-Kernel* läuft der Betriebssystemkern selbst als Prozess mit niedrigster Priorität. Der Echtzeitkern übernimmt das Scheduling. Die Echtzeit-Prozesse werden mit der höchsten Priorität ausgeführt. Der Zweck dieses Vorgehens ist es, die Reaktionszeit zu minimieren.

Verfügt ein Echtzeitbetriebssystemen über einen sogenannten *Nano-Kernel*, bedeutet das, das neben dem Echtzeitkern weitere Betriebssystemkerne laufen können (siehe Abbildung 3.7).

Die Fachbegriffe *Pico-Kernel*, *Femto-Kernel* und *Atto-Kernel* sind Beispiele für Marketingbegriffe der Hersteller von Echtzeitsystemen, um die geringe Größe ihrer Echtzeitkerne hervorzuheben. Die Größe des Betriebssystemkerns wird allgemein als Qualitätskriterium angesehen, da die Reduktion des Echtzeitkerns auf die wich-

features is supposed to minimize the probability of bugs.

tigsten Funktionalitäten die Wahrscheinlichkeit von Fehlern minimieren soll.

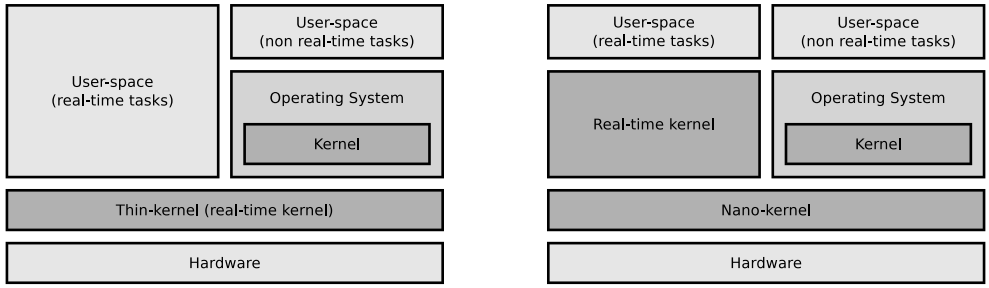

Figure 3.7: Architectures of Real-Time Operating Systems with Thin Kernel and Nano Kernel

3.7

Distributed Operating Systems

A *distributed operating system* is a distributed system that controls the processes on multiple independent computers. The individual nodes remain hidden from the users and their applications (see Figure 3.8). The system appears as a single large computer system. In the field of distributed systems, this principle is called *Single System Image* [19].

Verteilte Betriebssysteme

Ein *verteiltes Betriebssystem* ist ein verteiltes System, das die Prozesse auf mehreren unabhängigen Computern steuert. Die einzelnen Knoten bleiben den Benutzern und deren Prozessen verborgen (siehe Abbildung 3.8). Das System erscheint als ein einzelner großer Computer. Dieses Prinzip ist im Bereich der verteilten Systeme auch unter dem Fachbegriff des *Single System Image* [19] bekannt.

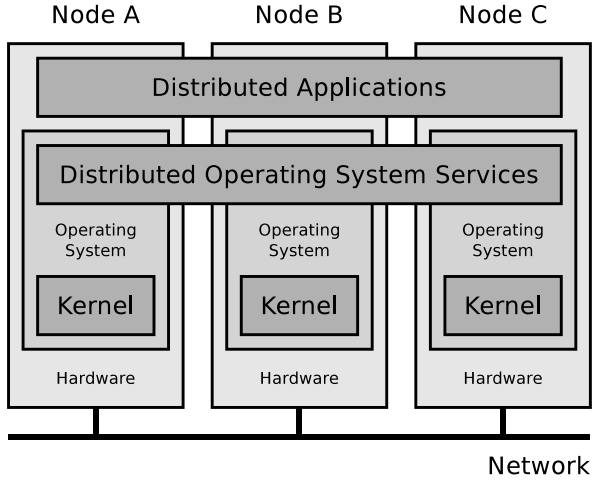

Figure 3.8: Distributed Operating Systems Architecture

Some examples of distributed operating systems are:

- *Amoeba.* This operating system was developed from the mid-1980s to the mid-1990s under the guidance of Andrew S. Tanenbaum at the Free University of Amsterdam [117]. Particularly remarkable is the fact that the programming language Python was initially developed for Amoeba [105].

- *Inferno* was initially developed at the Bell Laboratories and is based on the Unix operating system Plan 9. Unlike other distributed operating systems, Inferno does not replace the operating system but relies on an existing host operating system. It may be Microsoft Windows, Linux, or different Unix operating systems. The current version (Inferno 4th edition) supports numerous hardware architectures and was released as free software in 2005. Applications are developed in the Limbo language. Same as Java, Limbo produces byte code, which is executed by a virtual machine. Inferno has minimum hardware requirements. For example, it requires only 1 MB of main memory [34].

- *Rainbow* was developed at the University of Ulm. This distributed operating system was developed in the mid-2000s. It implements the concept of shared memory with a uniform address space, in which all computers that are part of the cluster can store data as objects. For applications, it is transparent on which computer in the cluster the objects are physically located. Applications can access desired objects via uniform addresses from any computer. If the object is physically located in the memory of a remote computer, Rainbow does the transmission and local deployment to the requesting computer in an automated and transparent way [103].

Einige Beispiele für verteilte Betriebssysteme sind:

- *Amoeba.* Dieses Betriebssystem wurde von Mitte der 1980er Jahre bis Mitte der 1990er Jahre unter der Leitung von Andrew S. Tanenbaum an der Freien Universität Amsterdam entwickelt [117]. Besonders hervorzuheben ist, dass die Programmiersprache Python ursprünglich für Amoeba entwickelt wurde [105].

- *Inferno* wurde ursprünglich in den Bell Laboratories entwickelt und basiert auf dem Unix-Betriebssystem Plan 9. Im Gegensatz zu anderen verteilten Betriebssystemen ersetzt Inferno nicht das Betriebssystem, sondern setzt auf einem bereits existierenden Host-Betriebssystem auf. Dabei kann es sich um Microsoft Windows, Linux oder verschiedene Unix-Betriebssysteme handeln. Die aktuelle Version (Inferno 4th edition) unterstützt zahlreiche Hardwarearchitekturen und wurde 2005 als freie Software veröffentlicht. Anwendungen werden in der Sprache Limbo programmiert. Diese produziert genau wie Java einen Bytecode, den eine virtuelle Maschine ausführt. Das Betriebssystem hat nur minimale Anforderungen an die Hardware. So benötigt das Betriebssystem nur 1 MB Arbeitsspeicher [34].

- *Rainbow* von der Universität Ulm. Dieses ab Mitte der 2000er Jahre entwickelte verteilte Betriebssystem realisiert das Konzept eines gemeinsamen Speichers mit einem für alle verbunden Computer einheitlichen Adressraum, in dem Datenobjekte abgelegt werden. Für Anwendungen ist es transparent, auf welchem Computer im Cluster sich Objekte physisch befinden. Anwendungen können über einheitliche Adressen von jedem Computer auf gewünschte Objekte zugreifen. Sollte sich das Objekt physisch im Speicher eines entfernten Computers befinden, sorgt Rainbow automatisch und transparent für eine Übertragung und lokale Bereitstellung auf dem bearbeitenden Computer [103].

- *Sprite* was developed at the University of California in Berkeley from the mid-1980s to the mid-1990s. This operating system connects workstations so that they appear as a single time-sharing system to users (see Section 3.4.1) [89]. Interestingly, `pmake`, a parallel version of the `make` tool, had initially been developed for Sprite.

The distributed operating system concept was not successful. Existing distributed operating systems did not go beyond the research project stage. One reason was that the idea of replacing the established operating systems did not make sense. A positive aspect is the numerous tools and technologies that have been developed as by-products – such as address spaces that are distributed over several independent computers, the Python programming language, or the `pmake` tool, which are well established in computer science today.

There are libraries for the development of applications for clusters, such as the Message Passing Interface (MPI) [42], OpenSHMEM [22], or Unified Parallel C (UPC) [32]. MPI implements a hardware-independent message passing, i.e., communication that is based on message exchange (see Figure 3.9). The solutions OpenSHMEM and UPC both allow the construction of a partitioned global address space over independent computers. Using such libraries is a more lightweight solution compared to the development and deployment of entirely new operating systems.

- *Sprite*, das von Mitte der 1980er Jahre bis Mitte der 1990er Jahre an der University of California in Berkeley entwickelt wurde. Dieses Betriebssystem verbindet Workstations in einer Art und Weise, dass sie für die Benutzer wie ein einzelnes System mit Dialogbetrieb (siehe Abschnitt 3.4.1) erscheinen [89]. Interessant ist auch, dass `pmake`, eine parallele Version des Werkzeugs `make`, ursprünglich für Sprite entwickelt wurde.

Das Konzept der verteilten Betriebssysteme konnte sich nicht durchsetzen. Die existierenden verteilten Betriebssysteme kamen nicht über das Stadium von Forschungsprojekten hinaus. Ein Grund war, dass das Ersetzen der etablierten Betriebssysteme nicht sinnvoll erschien. Positiv anzumerken sind die zahlreichen Werkzeuge und Technologien, die quasi als Nebenprodukte entstanden sind – wie Adressräume, die sich über mehrere unabhängige Computer verteilen, die Programmiersprache Python oder das Werkzeug `pmake`, die heute in der Informatik einen festen Platz haben.

Um Anwendungen für Verbünde von Computern zu entwickeln, existieren Bibliotheken wie zum Beispiel das Message Passing Interface (MPI) [42], OpenSHMEM [22] oder Unified Parallel C (UPC) [32]. MPI stellt ein von der Hardware unabhängiges Message Passing, also Kommunikation basierend auf dem Versand von Nachrichten, bereit (siehe Abbildung 3.9). Die beiden Lösungen OpenSHMEM und UPC ermöglichen den Aufbau eines partitionierten globalen Adressraums über unabhängige Computer. Der Einsatz solcher Bibliotheken ist eine leichtgewichtigere Lösung als die Entwicklung und Installation vollständig neuer Betriebssysteme.

3.8

Kernel Architectures

The *kernel* contains the essential functions of the operating system. It is the interface to the hardware of the computer.

Architektur des Betriebssystemkerns

Der *Betriebssystemkern* (englisch: *Kernel*) enthält die grundlegenden Funktionen des Betriebssystems. Er ist die Schnittstelle zur Hardware des Computers.

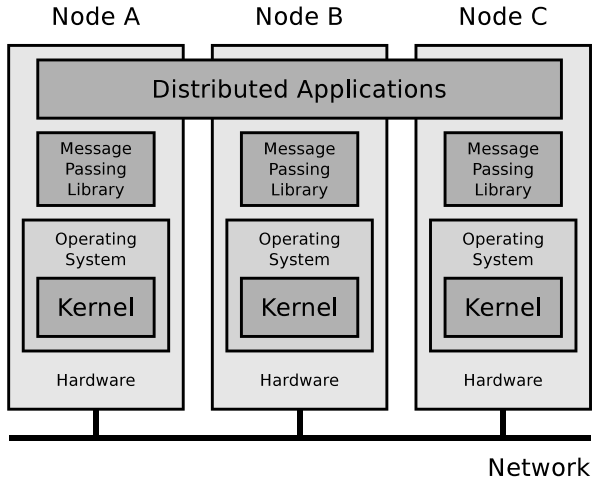

Figure 3.9: Modern Architecture of Distributed Systems

The basic features are the same for all operating systems. They include the provision of system calls, functions for user administration and process management including the definition of the execution order (*scheduling*) and interprocess communication, a process switch (*dispatcher*), essential device drivers, functions for memory management as well as file systems for storing files on storage devices.

Each fully functional operating system must implement the above features. The developers of the operating systems have some options for the placement of the corresponding functions. These can be implemented either inside the kernel or outside the kernel in processes that are called *service* or *server* in this context. There are three different architectures: *monolithic kernel*, *microkernel*, and *hybrid kernel*.

One effect of the placement is that functions located inside the kernel have full hardware access. They run in the address space of the kernel, the so-called *kernel mode* (see Section 7.1). On the other hand, if a function runs outside the kernel address space, it only has access to its virtual memory, the so-called *user mode*.

Die grundlegenden Funktionalitäten sind bei allen Betriebssystemen gleich. Zu diesen gehört die Bereitstellung von Systemaufrufen, Funktionen zur Benutzerverwaltung und Prozessverwaltung inklusive Festlegung der Ausführungsreihenfolge (englisch: *Scheduling*) und Interprozesskommunikation, ein Prozessumschalter (englisch: *Dispatcher*), die nötigen Gerätetreiber, Funktionen zur Speicherverwaltung und Dateisysteme zur Verwaltung von Dateien auf Speicherlaufwerken.

Jedes vollständige Betriebssystem muss die genannten Funktionalitäten erbringen. Die Entwickler der Betriebssysteme haben gewisse Freiheiten bei der Positionierung der entsprechenden Funktionen. Diese können entweder vom Betriebssystemkern selbst oder von Prozessen, die in diesem Kontext auch *Dienst* oder *Server* heißen, außerhalb des Kerns erbracht werden. Dementsprechend werden die drei Architekturen *monolithischer Kern*, *minimaler Kern* und *hybrider Kern* unterschieden.

Eine konkrete Auswirkung der Positionierung ist, dass Funktionen, die sich im Betriebssystemkerns befinden, vollen Hardwarezugriff haben. Sie laufen im Adressraum des Kerns, dem sogenannten *Kernelmodus* (siehe Abschnitt 7.1). Wird eine Funktion hingegen außerhalb des Adressraums des Kerns ausgeführt, kann diese nur auf ihren virtuellen Speicher, den sogenannten *Benutzermodus*, zugreifen.

3.8.1

Monolithic Kernels

A monolithic kernel contains all functions for providing the features of an operating system (see Figure 3.10).

One advantage of this architectural principle is that it requires the least number of process switches compared to all other architectural principles. Thus, the execution speed of an operating system with a monolithic kernel is better in comparison to a microkernel or hybrid kernel. Also, the stability of operating systems with a monolithic kernel often has grown over the years. As a consequence, in practice, operating systems with a microkernel are not inherently more stable than those with a monolithic kernel.

A drawback is that crashed kernel components cannot be restarted separately and may cause the entire operating system to crash. Also, kernel extensions cause a high development effort, because, for each compilation of the extension, the entire kernel needs to be recompiled. One option to reduce the mentioned drawbacks is using so-called modules as in the Linux kernel.

It is possible to outsource many hardware and file system drivers into modules. However, the modules are executed in kernel mode, and not in user mode. Therefore, the Linux kernel is a monolithic kernel. Further examples of operating systems with a monolithic kernel are variants of the Berkeley Software Distribution (BSD), MS-DOS, FreeDOS, Windows 95/98/ME, Mac OS (up to version 8.6) and OS/2.

Monolithische Kerne

Ein monolithischer Betriebssystemkern enthält alle Funktionen zur Erbringung der Funktionalitäten eines Betriebssystems (siehe Abbildung 3.10).

Ein Vorteil dieses Architekturprinzips ist, dass im Vergleich zu allen anderen Architekturen weniger Prozesswechsel nötig sind. Dadurch ist die Ausführungsgeschwindigkeit eines Betriebssystems mit einem monolithischen Kern besser als mit einem minimalen oder einem hybriden Kern. Zudem haben Betriebssysteme mit einem monolithischen Kern häufig eine durch jahrelange Entwicklungstätigkeit gewachsene Stabilität. Dadurch sind Betriebssysteme mit einem minimalen Kern in der Praxis nicht zwangsläufig stabiler als diejenigen mit einem monolithische Kern.

Nachteilig ist, dass abgestürzte Komponenten des Kerns nicht separat neu gestartet werden können und eventuell das gesamte Betriebssystem zum Absturz bringen. Zudem ist der Entwicklungsaufwand für Erweiterungen am Kern höher, da dieser bei jedem Kompilieren komplett neu übersetzt werden muss. Eine Möglichkeit, die genannten Nachteile abzumildern, ist die Verwendung von sogenannten Modulen wie beim Linux-Betriebssystemkern.

Bei Linux können bestimmte Funktionen wie beispielsweise Hardware- und Dateisystem-Treiber in Module auslagert werden. Diese werden jedoch im Kernelmodus und nicht im Benutzermodus ausgeführt. Darum ist der Linux-Kern ein monolithischer Kern. Weitere Beispiele für Betriebssysteme mit monolithischem Kern sind verschiedene Varianten der Berkeley Software Distribution (BSD), MS-DOS, FreeDOS, Windows 95/98/ME, Mac OS (bis Version 8.6) und OS/2.

3.8.2

Microkernels

Microkernels typically contain only the essential functions for memory management, process management, process synchronization, and interprocess communication. Device drivers, file system drivers, and all other functions run as

Minimale Kerne

In minimalen Betriebssystemkernen, die auch *Mikrokern* oder *Mikrokernel* heißen, befinden sich üblicherweise nur die nötigsten Funktionen zur Speicher- und Prozessverwaltung sowie zur Synchronisation und Interprozesskommunikation. Gerätetreiber, Treiber für Dateisysteme und

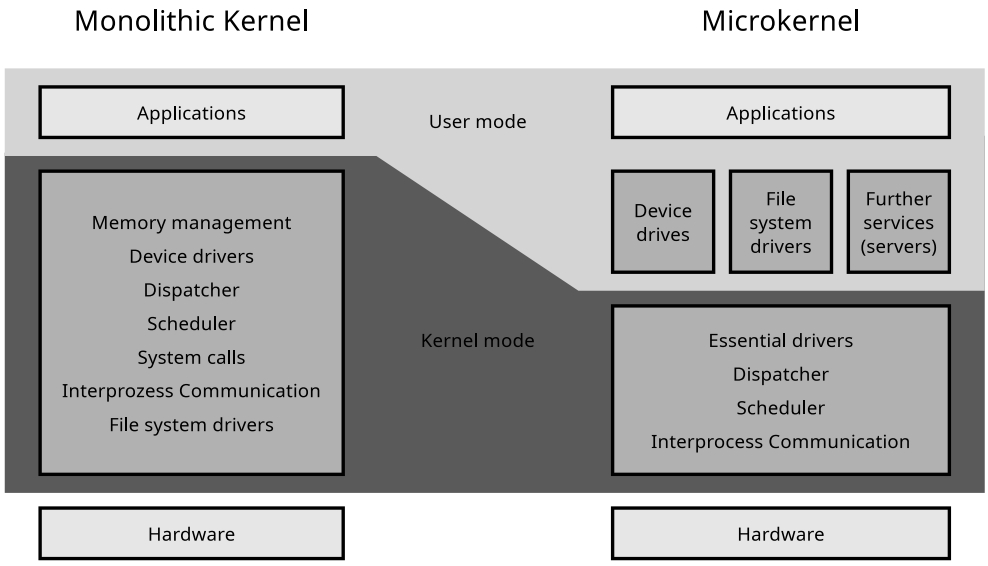

Figure 3.10: Architecture of Monolithic Kernels and Microkernels

so-called *services* or *servers* outside the kernel in user mode (see Figure 3.10).

The benefits of this design principle are that functions that have been outsourced from the kernel can be exchanged more easily. In theory, a microkernel provides better stability and security because fewer functions run in kernel mode.

Drawbacks are that the microkernel architecture offers the worst performance compared to all other designs because it requires the highest number of process switches. Also, developing a new operating system kernel is a complex and time-consuming task that requires considerable financial resources or the enthusiasm of voluntary developers. From the users' or customers' perspective, the architecture of the kernel is seldom a critical factor, because it has no impact on the functionality of the operating system.

One example that illustrates the complexity of developing a new microkernel is GNU Hurd, which is a kernel that is being developed for the operating system GNU since the early 1990s.

alle weiteren Funktionalitäten laufen als sogenannte *Dienste* bzw. *Server* außerhalb des Kerns im Benutzermodus (siehe Abbildung 3.10).

Die Vorteile dieses Architekturprinzips sind, dass aus dem Kern ausgelagerte Funktionalitäten leichter austauschbar sind. Theoretisch bietet ein minimaler Kern eine bessere Stabilität und Sicherheit, weil weniger Funktionen im Kernelmodus laufen.

Nachteilig ist, dass das Architekturprinzip der minimalen Kerne im Vergleich zu allen anderen Architekturen die geringste Ausführungsgeschwindigkeit bietet, weil es gleichzeitig die größte Anzahl an benötigten Prozesswechseln aufweist. Zudem ist die Entwicklung eines neuen Betriebssystemkerns eine komplexe und zeitintensive Entwicklung die große finanzielle Ressourcen oder großen Enthusiasmus freiwilliger Entwickler benötigt. Aus Sicht der Benutzer oder Kunden eines Betriebssystems ist die Architektur des Betriebssystemkerns auch kaum ein entscheidendes Kriterium, da die Architektur keine Auswirkung auf den Funktionsumfang hat.

Ein Beispiel, das die Komplexität der Entwicklung eines neuen minimalen Kerns veranschaulicht, ist der Kernel GNU Hurd, der seit den frühen 1990er Jahren für das Betriebssystem

Ultimately, it was the absence of a working Hurd kernel, and the troubles with its development, that motivated a Finnish computer science student named Linus Torvalds in 1991 to begin developing his own (for simplicity's sake: monolithic) kernel and thus the Linux operating system.

Operating systems with a microkernel are, for instance, AmigaOS, MorphOS, Tru64, the real-time operating system QNX Neutrino, Symbian, Minix, GNU Hurd, and the distributed operating system Amoeba.

3.8.3

Hybrid Kernels

A compromise between monolithic kernels and microkernels are hybrid kernels, which are also called *macrokernels*. For performance reasons, these contain components that are never located inside microkernels. It is not specified which additional components are compiled into the kernel of systems with hybrid kernels.

The advantages of hybrid kernels are that they theoretically provide better performance than microkernels and that they offer better stability in comparison to monolithic kernels.

The example of Windows NT 4 illustrates the advantages and drawbacks of hybrid kernels. In this operating system from 1996, the kernel contains the Graphics Device Interface [81]. One positive effect was an improvement in the performance of the graphics subsystem. A drawback, however, was that buggy graphics drivers often caused crashes, and since the market share of NT 4 was smaller than that of Windows 95/98, the vendors of graphics cards did not focus on improving the graphics drivers for NT 4.

Operating systems with hybrid kernels are, for instance, Windows since NT 3.1, ReactOS, Apple Mac OS X, BeOS, ZETA, Haiku, Plan 9, and DragonFly BSD.

GNU entwickelt wird. Letztlich war es auch das Fehlen eines funktionierenden Hurd-Kerns und die Probleme bei dessen Entwicklung, die 1991 einen finnischen Studenten der Informatik mit dem Namen Linus Torvalds dazu motivierten, die Entwicklung an einem eigenen (der Einfachheit halber monolithischen) Betriebssystemkern und damit am Betriebssystem Linux zu starten.

Beispiele für Betriebssysteme mit eine minimalen Kern sind AmigaOS, MorphOS, Tru64, das Echtzeitbetriebssystem QNX Neutrino, Symbian, Minix, GNU Hurd und das verteilte Betriebssystem Amoeba.

Hybride Kerne

Ein Kompromiss zwischen monolithischen Kernen und minimalen Kernen sind die hybriden Kerne, die auch *Makrokernel* heißen. Diese enthalten aus Geschwindigkeitsgründen Komponenten, die bei minimalen Kernen außerhalb des Kerns liegen. Es ist nicht spezifiziert, welche Komponenten bei Systemen mit hybriden Kernen zusätzlich in den Kernel einkompiliert sind.

Vorteile hybrider Kerne sind, dass sie theoretisch eine höhere Geschwindigkeit als minimale Kerne ermöglichen und das sie im Vergleich zu monolithischen Kernen eine höhere Stabilität bieten.

Die Vor- und Nachteile hybrider Kerne zeigt das Beispiel von Windows NT 4. Bei diesem Betriebssystem aus dem Jahr 1996 enthält der Betriebssystemkern das Grafiksystem [81]. Ein positiver Effekt war eine verbesserte Leistung bei der Grafikausgabe. Nachteilig war allerdings, dass fehlerhafte Grafiktreiber zu häufigen Abstürzen führen. Da der Marktanteil von NT 4 geringer war als der von Windows 95/98, lag der Fokus der Hersteller von Grafikkarten auch nicht in der Verbesserung der Grafiktreiber für NT 4.

Beispiele für Betriebssysteme mit hybriden Kernen sind Windows seit NT 3.1, ReactOS, Apple Mac OS X, BeOS, ZETA, Haiku, Plan 9 und DragonFly BSD.

3.9

Structure (Layers) of Operating Systems

In literature (e.g., in [112] and [119]), the components of operating systems are often structured by models of several layers. Thereby, the operating systems are logically structured with layers that surround each other. The layers enclose each other and contain more and more abstract functions from inside to outside (see Figure 3.11).

Schichtenmodell

In der Literatur (z.B. bei [115] und bei [119]) ist es ein etabliertes Verfahren, die Komponenten von Betriebssystemen mit Schichtenmodellen zu visualisieren. Dabei werden die Betriebssysteme mit ineinander liegenden Schichten logisch strukturiert. Die Schichten umschließen sich gegenseitig und enthalten von innen nach außen immer abstraktere Funktionen (siehe Abbildung 3.11).

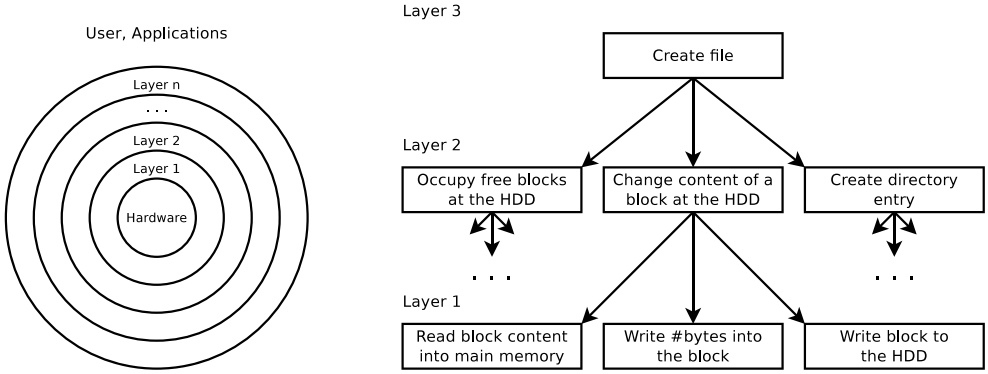

Figure 3.11: The Level of Abstraction inside the Layers grows from Inside to Outside [119]

Three layers are required as a minimum to achieve a meaningful representation. The innermost layer includes those parts of the operating system that depend on the hardware. Operating systems can be ported to different computer architectures only by customizing the innermost layer. The central layer includes basic input/output services for devices and data. These are libraries and interfaces [119]. The outermost layer includes the applications and the user interface. Typically, operating systems are illustrated with more than three logical layers, as in Figure 3.12.

Each layer communicates with neighboring layers via well-defined interfaces, calls functions of the next inner layer, and provides functions to the next outer layer. All functions (so-called *services*) which a layer offers, and the rules which have to be complied to, are called *protocol*. In practice, the constraint that communi-

Das Minimum für eine sinnvolle Darstellung sind drei Schichten. Die innerste Schicht enthält diejenigen Teile des Betriebssystems, die abhängig von der Hardware sind. Betriebssysteme können nur durch Anpassungen in der innersten Schicht an unterschiedliche Computerarchitekturen angepasst werden. Die mittlere Schicht enthält grundlegende Ein-/Ausgabe-Dienste für Geräte und Daten. Dabei handelt es sich um Bibliotheken und Schnittstellen [119]. Die äußerste Schicht enthält die Anwendungsprogramme und die Benutzerschnittstelle. Meist stellt man Betriebssysteme wie in Abbildung 3.12 mit mehr als drei logischen Schichten dar.

Jede Schicht kommuniziert mit benachbarten Schichten über wohldefinierte Schnittstellen, kann Funktionen der nächst inneren Schicht aufrufen und stellt Funktionen der nächst äußeren Schicht zur Verfügung. Alle Funktionen (die sogenannten *Dienste*), die eine Schicht anbietet, und die Regeln, die dabei einzuhalten sind, hei-

cation is only possible with neighboring layers is not always met. In Linux, for instance, the applications (processes) of the users (layer 4 in Figure 3.12) can use the library functions of the standard library glibc (interface D) or directly use the system calls (interface C) of the kernel.

ßen zusammen *Protokoll*. In der Praxis wird die Einschränkung, dass Kommunikation nur mit benachbarten Schichten möglich ist, nicht immer durchgehalten. Bei Linux beispielsweise können Anwendungen (Prozesse) der Benutzer (Schicht 4 in Abbildung 3.12) die Bibliotheksfunktionen der Standardbibliothek glibc (Schnittstelle D) oder direkt die Systemaufrufe (Schnittstelle C) des Betriebssystemkerns nutzen.

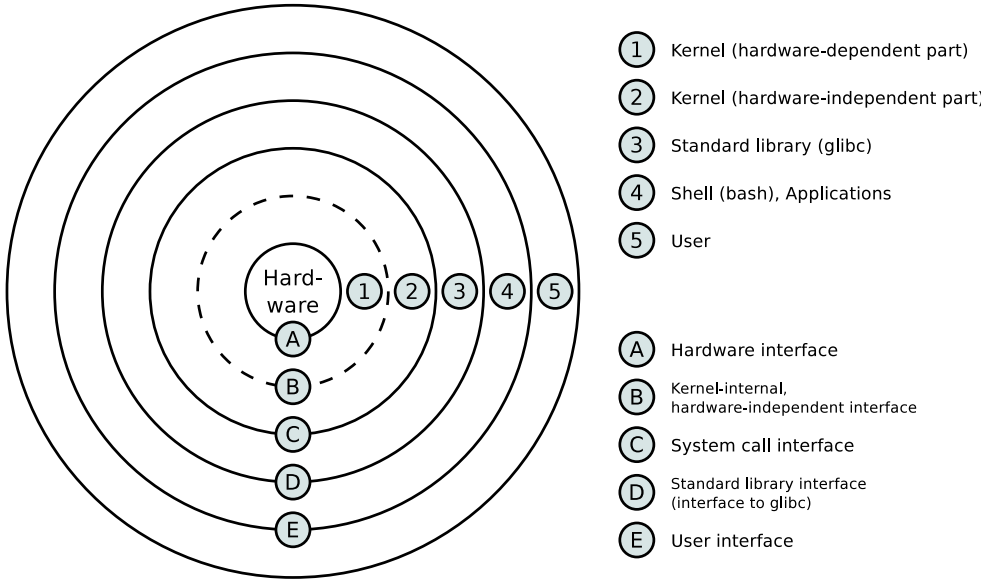

Figure 3.12: Illustration of the Components of Linux, and generally Unix-like Operating Systems, with a multilayer Model

3.10

Booting the Operating System

Starting a computer and its operating system is called the *boot process*, or *bootstrapping*. It includes steps from initializing the hardware components to passing over control to the operating system and its users or processes and providing a user interface.

The individual steps of the boot process of modern Linux/UNIX operating systems are as follows:

Start des Betriebssystems

Der Start eines Computers und seines Betriebssystems heißt *Bootvorgang*, *Bootprozess* oder *Bootstrapping*. Er umfasst zahlreiche Schritte von der Initialisierung der Hardwarekomponenten bis zur Übergabe der Kontrolle an die Benutzer bzw. deren Prozesse sowie die Bereitstellung einer Benutzerschnittstelle.

Die einzelnen Schritte des Bootvorgangs von modernen Linux/UNIX-Betriebssystemen sind:

1. Switch on the computer
2. Start the firmware and perform a self-test
3. Start the boot loader
4. Start the operating system kernel and the temporary root file system
5. Mount the real root file system
6. Start init and the system processes
7. Pass control to the users

The steps of the boot process are described in detail in the following sections.

1. Computer einschalten
2. Firmware starten und Selbsttest durchführen
3. Bootloader starten
4. Betriebssystemkern starten und temporäres Root-Dateisystem einbinden
5. Echtes Root-Dateisystem einbinden
6. init und Systemprozesse starten
7. Kontrolle an die Benutzer übergeben

Die genannten Schritte des Bootvorgangs werden in den folgenden Abschnitte näher beschrieben.

3.10.1

Switch on the Computer

In older computers, such as those that were state of the art until the 2000s, the mainboard is powered up when the computer is turned on, and the processor starts the fetch-decode-execute cycle (see Section 4.1.2).

In more modern computers, an autonomous subsystem is usually running continuously, such as the *Intel Management Engine* (since 2008) or the *AMD Platform Security Processor* (since 2013). Such subsystems are independent microcontrollers on the mainboard or in the chipset or special processor cores in the CPU with their own operating system. They usually run whenever a sufficiently charged battery or a permanent power source is available. They allow a computer to be monitored and woken up via the network (*Wake-on-LAN*) and provide remote administration capabilities (*remote management*). [21, 36]

Computer einschalten

Bei älteren Computern, wie sie bis in die 2000er Jahre Stand der Technik waren, wird das Mainboard durch Einschalten des Computers mit Strom versorgt und der Prozessor beginnt mit dem Von-Neumann-Zyklus (siehe Abschnitt 4.1.2).

Bei moderneren Computern läuft üblicherweise dauerhaft ein autonomes Subsystem wie die *Intel Management Engine* (seit 2008) oder der *AMD Platform Security Processor* (seit 2013). Solche Subsysteme sind eigenständige Mikrocontroller auf dem Mainboard oder im Chipsatz oder alternativ spezielle Prozessorkerne im Hauptprozessor mit eigenem Betriebssystem. Sie laufen üblicherweise immer dann, wenn ein ausreichend geladener Akku oder eine permanente Stromquelle vorhanden ist. Sie ermöglichen das Überwachen und Aufwecken eines Rechners über das Netzwerk (englisch: *Wake-on-LAN*) und bieten weitere Möglichkeiten zur Fernadministration (englisch: *Remote-Management*). [21, 36]

3.10.2

Start the Firmware and perform a self-test

After switching on the computer, the firmware is started. Older computer systems with x86-compatible processors from the early 1980s to the end of the 2000s have a so-called *Basic Input/Output System* (BIOS) as firmware. Newer systems have a *Unified Extensible Firmware Interface* (UEFI) [40]. The firmware performs a self-test of the system, the so-called *Power-on self-test* (POST). Among other things, during this procedure, the CPU, the buffer memory (*cache*), and the main memory are checked for correct operation [115].

During the POST, the firmware also checks the presence of graphical output hardware, input/output devices, and storage drives. Status and error messages are typically communicated during the POST through acoustic signals (beeps) or visual signals on the screen. For most older computers, but also for many modern computers, it is possible to monitor the self-test process using expansion cards, so-called mainboard diagnostic cards (*POST cards*) with 7-segment LCDs, or similar expansions.

Firmware starten und Selbsttest durchführen

Nach dem Einschalten des Computers wird die Firmware geladen. Ältere Computersysteme mit x86-kompatiblen Prozessoren von Anfang der 1980er Jahre bis Ende der 2000er Jahre verwenden als Firmware das sogenannte *Basic Input/Output System* (BIOS). Neuere Systeme verwenden das *Unified Extensible Firmware Interface* (UEFI) [40]. Die Firmware führt einen Selbsttest des Systems, den sogenannten *Power-on self-test* (POST), durch. Dabei werden u.a. die korrekte Funktion des Prozessors, des Pufferspeichers (englisch: *Cache*) und des Hauptspeichers überprüft [115].

Zudem prüft die Firmware beim POST das Vorhandensein einer Hardware zur grafischen Ausgabe sowie von Ein-/Ausgabegeräten und Speicherlaufwerken. Status- und Fehlermeldungen werden während des POST klassischerweise durch akustische Signale (Pieptöne) oder visuelle Signale auf dem Bildschirm mitgeteilt. Für die meisten älteren, aber auch viele moderne Computer gibt es die Möglichkeit, den Ablauf des Selbsttests mit Erweiterungskarten, sogenannten Mainboard-Diagnosekarten (englisch: *POST cards*) mit 7-Segment LCD-Displays oder vergleichbaren Erweiterungen zu überwachen.

3.10.3

Start the Boot Loader

Once the computer has booted and passed self-test, the firmware will search for the first *boot device*, also known as the *boot drive*. The drive order is defined by the user in the firmware or follows the default boot order. The boot device can be a drive (e.g., SSD, hard disk drive, USB memory stick) or a network resource if the firmware supports the *Preboot Execution Environment* (PXE). The firmware starts the *boot loader* from the selected boot device. This is a program that loads the operating system. The area of the boot drive where the boot loader is located depends on the partitioning scheme used.

Bootloader starten

Nach dem Start des Computers und dem erfolgreichen Selbsttest sucht die Firmware nach dem ersten *Bootgerät*, auch *Bootlaufwerk* genannt. Die Reihenfolge der Laufwerke ergibt sich aus der vom Benutzer in der Firmware definierten Boot-Reihenfolge oder entspricht der standardmäßigen Boot-Reihenfolge. Beim Bootgerät kann es sich um ein Laufwerk (z.B. SSD, Festplatte, USB-Speicherstick) oder eine Netzwerkressource handeln, wenn die Firmware *Preboot Execution Environment* (PXE) unterstützt. Die Firmware startet den *Bootloader* vom ausgewählten Bootgerät. Dabei handelt es sich um ein Programm, das das Betriebssystem lädt. In welchem Bereich des Bootlaufwerks der Bootloader

- When using a classic PC partition table, the boot loader is stored in the 512 byte long *Master Boot Record* (MBR)
- When using a GUID partition table (GPT), the boot loader is stored in the EFI system partition (ESP).

The firmware loads the boot loader from the boot device and writes it into the main memory. Some popular boot loaders include:

- *GRand Unified Bootloader* (GRUB). A modern, free software boot loader that supports numerous operating systems, hardware architectures, and file systems. Standard for Linux and various UNIX operating systems since the early 2000s.
- *Linux Loader* (LILO). Obsolete free software boot loader. Standard for Linux operating systems until the early 2000s. It cannot handle file systems but directly accesses the blocks of the storage medium on which the desired operating system kernel is stored.
- *Windows Boot Manager* (BOOTMGR). Proprietary boot loader from Microsoft for operating systems since Windows Vista.
- *NT-Loader* (NTLDR). Proprietary boot loader from Microsoft for the Windows NT/XP/2000/2003 operating systems.
- *Clover*. Modern free software bootloader supporting Linux, MacOS, and Windows.

Modern boot loaders such as GRUB and Clover allow users to boot operating systems with different parameters or in a protected mode.

liegt, hängt vom verwendetem Partitionsschema ab.

- Bei Nutzung einer klassischen PC-Partitionstabelle liegt der Bootloader innerhalb des 512 Bytes großen *Master Boot Record* (MBR).
- Bei Nutzung einer GUID-Partitionstabelle (englisch: *GUID Partition Table – GPT*) liegt der Bootloader in der EFI-Systempartition (englisch: *EFI System Partition – ESP*).

Die Firmware lädt den Bootloader vom Bootlaufwerk und schreibt ihn in den Arbeitsspeicher. Einige bekannte Bootloader sind:

- *GRand Unified Bootloader* (GRUB). Moderner quelloffener Bootloader mit Unterstützung für zahlreiche Betriebssysteme, Hardwarearchitekturen und Dateisysteme. Standard für Linux- und verschiedene UNIX-Betriebssysteme seit Anfang der 2000er Jahre.
- *Linux Loader* (LILO). Veralteter quelloffener Bootloader. Standard für Linux-Betriebssysteme bis Anfang der 2000er Jahre. Kann nicht mit Dateisystemen umgehen, sondern greift direkt auf die Blöcke des Datenträgers zu, in denen der gewünschte Betriebssystemkern liegt.
- *Windows Boot Manager* (BOOTMGR). Proprietärer Bootloader von Microsoft für Betriebssysteme ab Windows Vista.
- *NT-Loader* (NTLDR). Proprietärer Bootloader von Microsoft für die Betriebssysteme Windows-NT/XP/2000/2003.
- *Clover*. Moderner quelloffener Bootloader mit Unterstützung für Linux, MacOS und Windows.

Moderne Bootloader wie GRUB und Clover ermöglichen den Benutzern den Start des Betriebssystems mit verschiedenen Parametern

If there are several operating systems on the computer, modern boot loaders also allow one of these operating systems to be selected. GRUB even provides a command-line interpreter, the so-called GRUB shell. It provides command-line tools for repairing the GRUB boot loader configuration and manually controlling the boot process.

oder in einem abgesicherten Modus. Beim Vorhandensein mehrerer Betriebssysteme auf dem Computer erlauben moderne Bootloader auch die Auswahl eines dieser Betriebssysteme. GRUB bietet sogar einen Kommandozeileninterpreter, die sogenannte GRUB-Shell. Diese stellt Informationen und Kommandozeilenwerkzeug zur Verfügung, um aus GRUB heraus die Konfiguration des Bootloaders zu reparieren und den Bootprozess manuell zu steuern.

3.10.4

Start the Operating System Kernel and the temporary Root File System

Once the user has manually or automatically selected an operating system or kernel using the boot loader, the kernel it unpacked and written into the main memory.

In Linux operating systems, the kernel is typically a compressed file with the file name `vmlinuz-<Version>-<Architecture>` and is stored in the `/boot` folder. In the Microsoft Windows NT operating systems family, the file is called `Ntoskrnl.exe` and is stored in the folder `\Windows\System32`. [101]

After loading the operating system kernel, the boot loader loads and mounts an initial RAM disk (`initrd`) or an initial RAM file system (`initramfs`) into the main memory. This is a temporary root file system loaded into the main memory. In Linux/UNIX operating systems, the root file system, also known as the root directory, is identified by a slash character (`/`). The temporary root file system loaded by `initrd` or `initramfs` implements a minimal Linux environment in the main memory. Its primary purpose is to provide the kernel with more device drivers, file systems drivers, and programs to mount the actual root file system of the operating system into the main memory.

Betriebssystemkern starten und temporäres Root-Dateisystem einbinden

Hat der Benutzer manuell oder der Bootloader automatisch ein Betriebssystem bzw. einen bestimmten Betriebssystemkern ausgewählt, wird der Betriebssystemkern vom Bootloader entpackt und in den Arbeitsspeicher geladen.

Der Betriebssystemkern liegt bei Linux-Betriebssystemen typischerweise als komprimierte Datei mit dem Dateinamen `vmlinuz-<Version>-<Architektur>` im Ordner `/boot`. Bei Betriebssystemen der Microsoft Windows NT-Familie heißt die Datei `Ntoskrnl.exe` und sie liegt im Ordner `\Windows\System32`. [101]

Nach dem Laden des Betriebssystemkerns lädt der Bootloader bei einem Linux-Betriebssystem ist eine initiale RAM-Disk (englisch: `initrd`) oder ein initiales RAM-Dateisystem (englisch: `initramfs`) in den Hauptspeicher und bindet es ein (englisch: *mount*). Dabei handelt es sich um ein temporäres, in den Arbeitsspeicher geladenes Root-Dateisystem. Das Root-Dateisystem, das auch Wurzelverzeichnis heißt, ist unter Linux- und UNIX-Betriebssystemen mit dem Schrägstrich (*Slash*, `/`) gekennzeichnet. Das durch `initrd` oder `initramfs` geladene temporäre Root-Dateisystem realisiert eine minimale Linux-Umgebung im Arbeitsspeicher. Sie dient in erster Linie dazu, dem Betriebssystemkern weitere Gerätetreiber, Treiber für Dateisysteme und Programme bereitzustellen, um das echte Root-Dateisystem des Betriebssystems in den Arbeitsspeicher zu laden.

In Linux operating systems, the initial RAM disk is also usually stored as a compressed file in the `/boot` folder, with the file name `initrd.img-<Version>-<Architecture>`.

Die initiale RAM-Disk liegt bei Linux-Betriebssystemen typischerweise ebenfalls als komprimierte Datei im Ordner `/boot` und hat den Dateinamen `initrd.img-<Version>-<Architektur>`.

3.10.5

Mount the real Root File System

From the temporary root file system, the operating system kernel accesses the drive containing the real root file system, validates its consistency (absence of errors), and corrects any errors in the file system. The operating system kernel then mounts the real root file system to replace the temporary root file system. Any other existing file systems are also mounted during this stage of the boot process (e.g., `/home`).

Echtes Root-Dateisystem einbinden

Aus dem temporären Root-Dateisystem heraus greift der Betriebssystemkern auf das Laufwerk mit dem echten Root-Dateisystem zu, überprüft dessen Konsistenz (Fehlerfreiheit) und korrigiert ggf. Fehler im Dateisystem. Daraufhin bindet der Betriebssystemkern das echte Root-Dateisystem ein und ersetzt damit das temporäre Root-Dateisystem. Auch weitere eventuell vorhandene Dateisysteme werden in diesem Schritt des Bootvorgangs eingebunden (z.B. unter `/home`).

3.10.6

Mount the real Root File System

After the kernel has been started and the root file system has been mounted, the kernel starts `init` as the first user-mode process. The process `init` has the process ID 1. All other user-mode processes in the user mode descend from `init` (see Section 8.4). Numerous system processes and services are also started automatically in this phase of the boot process.

Until the late 2000s, Linux operating systems, like many other UNIX operating systems, included a System V-style implementation of `init`, also called *sysvinit*. Since the early 2010s, most popular Linux distributions have used the more advanced *systemd*, which has many advantages. Benefits of systemd include a faster system boot by starting system processes (services) in parallel, an integrated logging system called `journald` for unified event logging and analysis, and the ability to restart failed services automatically. Alternative `init` implementations include *OpenRC* and *runit*.

Echtes Root-Dateisystem einbinden

Nach dem Start des Betriebssystemkerns und der Einbindung des Root-Dateisystems startet der Kern den Prozess `init` als ersten Prozess im Benutzermodus. Der Prozess `init` hat die Prozessnummer 1. Von ihm stammen alle weiteren Prozesse im Benutzermodus ab (siehe Abschnitt 8.4). In dieser Phase des Bootvorgangs werden auch zahlreiche Systemprozesse und Dienste automatisiert gestartet.

Bis Ende der 2000er Jahre verwendeten Linux-Betriebssysteme wie viele andere UNIX-Betriebssysteme eine Implementierung von `init` im Stil des Standards System V, die auch *sysvinit* heißt. Seit Anfang der 2010er Jahre verwenden die meisten populären Linux-Distributionen das moderne *systemd*, das viele Vorteile bietet. Dazu gehört ein massiv schnellerer Systemstart durch den parallelisierten Start der Systemprozesse (Dienste), ein integriertes Logging-System namens `journald` zur einheitlichen Protokollierung und Analyse von Ereignissen, und die Fähigkeit, ausgefallene Dienste automatisch neu zu starten. Weitere alternative Implementierungen von `init` sind unter anderem *OpenRC* und *runit*.

3.10.7

Pass Control to the Users

As the final step in the boot process, the kernel passes control to the user and their user-mode processes. The kernel continues to run in main memory in *kernel mode*, managing the hardware resources and system calls (see Section 7).

The `getty` processes are also started at this stage. They provide a text-based login for users via one or more (virtual) consoles. Typical Linux systems have six virtual consoles, which can be accessed using the keys `Ctrl+Alt+F1` to `Ctrl+Alt+F6`. For historical reasons, these virtual consoles are called TTY. The abbreviation stands for *teletypewriter* (teleprinter). Although electronic terminals and virtual consoles have replaced teleprinters, the general mode of operation has remained the same, which is why the name has never been changed. The operating system starts a separate instance of the `getty` process for each of the virtual consoles (TTY1 to TTY6).

If the operating system uses a graphical login manager (e.g., GDM, LightDM, or XDM), it will also been started during this final step of the boot process. The graphical login manager typically runs on TTY7 and can be accessed on many Linux distributions using the `Ctrl+Alt+F7` key combination.

Kontrolle an die Benutzer übergeben

Als abschließenden Schritt des Bootvorgangs übergibt der Betriebssystemkern die Kontrolle an die Benutzer und deren Prozesse. Der Betriebssystemkern läuft weiter im Hauptspeicher im *Kernelmodus* und verwaltet die Hardwareressourcen und Systemaufrufe (siehe Kapitel 7).

In diesem Schritt werden auch die `getty`-Prozesse gestartet. Diese ermöglichen eine textbasierte Anmeldung der Benutzer über eine oder mehrere (virtuelle) Konsolen. Auf einem typischen Linux-System gibt es sechs virtuelle Konsolen, die über die Tasten `Strg+Alt+F1` bis `Strg+Alt+F6` aufgerufen werden. Die virtuellen Konsolen heißen in Linux/UNIX-Betriebssystemen TTY. Die Abkürzung steht für *Teletypewriter* (Fernschreiber) und hat historische Gründe. Zwar wurden die Fernschreiber zwischenzeitlich durch elektronische Terminals und virtuelle Konsolen abgelöst, aber die grundsätzliche Arbeitsweise blieb die gleiche, weshalb der Name nie geändert wurde. Für die virtuellen Konsolen (TTY1 bis TTY6) startet das Betriebssystem jeweils eine eigene Instanz des Prozesses `getty`.

Verwendet das Linux-Betriebssystem einen grafischen Login-Manager (z.B. GDM, LightDM oder XDM), wird auch dieser in diesem letzten Schritt des Bootvorgangs gestartet. Üblicherweise läuft er auf TTY7 und kann dementsprechend auf vielen Linux-Distributionen mit der Tastenkombination `Strg+Alt+F7` aufgerufen werden.

4

Fundamentals of Computer Architecture

Grundlagen der Rechnerarchitektur

A legitimate question is: Why does it make sense for a compact work on operating systems, such as this one, to include a description of how the CPU, memory, and bus systems work? Operating systems and their tools belong to the software topic. The answer to the above question is just the reason for using operating systems. Operating systems assist users, and their processes in using the hardware and an understanding of the essential hardware components of a computer is mandatory to understand the functioning of operating systems.

Eine berechtigte Frage ist: Warum ist es sinnvoll, dass ein kompaktes Buch wie dieses zum Thema Betriebssysteme auch eine Beschreibung der Arbeitsweise des Hauptprozessors, des Speichers und der Bussysteme enthält? Immerhin gehören die Betriebssysteme und deren Werkzeuge zum Themenkomplex Software. Die Antwort auf die oben genannte Frage ergibt sich aus dem Grund der Verwendung von Betriebssystemen. Betriebssysteme erleichtern den Benutzern und deren Prozessen die Nutzung der Hardware und ein Verständnis der notwendigsten Hardwarekomponenten eines Computers ist elementar, um die Arbeitsweise der Betriebssysteme zu verstehen.

4.1

Von Neumann Architecture

Von-Neumann-Architektur

The *Von Neumann architecture* was developed in the 1940s by John von Neumann. It describes the structure of a universal computer that is not limited to a fixed program and has input/output devices. In a computer whose structure and functioning follow the principles of the Von Neumann architecture, data and programs are binary coded and stored in the same memory. Konrad Zuse already implemented essential concepts of the Von Neumann architecture in 1937 in the Zuse Z1. The Z1 used binary numbers, was able to process floating-point numbers, and loaded the running program during its execution from a storage medium (punched tape).

Die in den 1940er Jahren von John von Neumann entwickelte *Von-Neumann-Architektur* beschreibt den Aufbau eines Universalrechners, der nicht an ein festes Programm gebunden ist und über Ein-/Ausgabegeräte verfügt. In einem Computer, dessen Aufbau und Arbeitsweise den Regeln der Von-Neumann-Architektur entspricht, werden Daten und Programme binär kodiert und liegen im gleichen Speicher. Wesentliche Ideen der Von-Neumann-Architektur wurden bereits 1937 von Konrad Zuse in der Zuse Z1 realisiert. So arbeitete die Z1 bereits intern mit binären Zahlen, konnte mit Fließkommazahlen umgehen und las das laufende

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_4

Programm während der Abarbeitung von einem Speichermedium (Lochstreifen) ein.

4.1.1

Central Processing Unit

Most components of a computer are passive and are controlled by the *Central Processing Unit* (CPU). During program execution, the CPU (see Figure 4.1) executes the machine instructions of the running program step by step. Programs are sequences of machine instructions that are stored in successive memory addresses. A CPU consists of two components which are: *arithmetic logic unit* and *control unit*. Furthermore, *memory* and *input/output devices* are required.

Hauptprozessor

Die meisten Komponenten eines Computers sind passiv und werden durch den *Hauptprozessor*, englisch *Central Processing Unit* (CPU) gesteuert. Bei der Programmausführung setzt der Prozessor (siehe Abbildung 4.1) die Maschineninstruktionen des aktuell laufenden Programms Schritt für Schritt um. Programme sind Folgen von Maschineninstruktionen, die in aufeinander folgenden Speicheradressen abgelegt sind. Ein Prozessor besteht aus den beiden Komponenten *Rechenwerk* und *Steuerwerk*. Zudem sind *Speicher* und *Ein-/Ausgabegeräte* nötig.

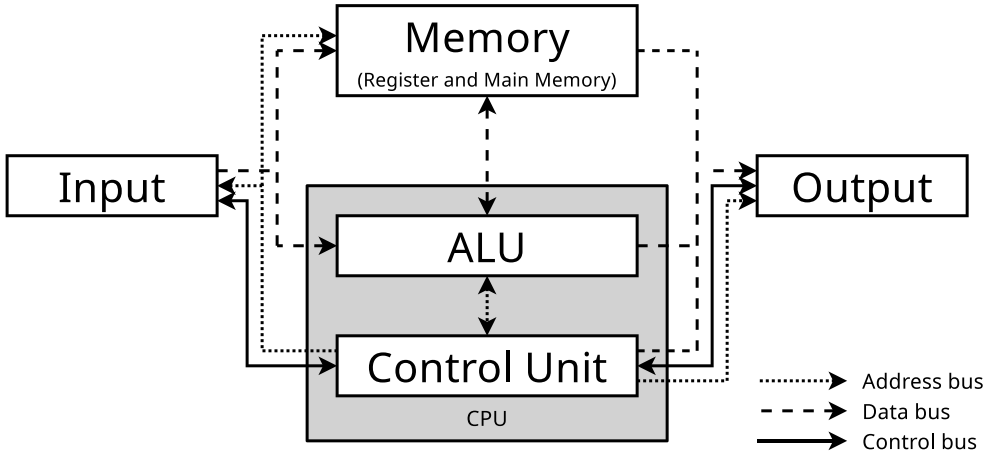

Figure 4.1: Components of the CPU in the Von Neumann Architecture

The *control unit* interprets instructions, coordinates the other components of the CPU, and controls the input/output devices and the control bus.

The *arithmetic logic unit* (ALU) manipulates data and addresses by carrying out the logical (NOT, AND, OR, XOR, ...) and mathematical (ADD, SUB, ...) operations.

Das Steuerwerk, das auch *Befehlswerk* (englisch: Control Unit) und seltener *Leitwerk* genannt wird, interpretiert Befehle, koordiniert die anderen Komponenten des Prozessors und steuert die Ein-/Ausgabe-Einheiten sowie den Steuerbus.

Das Rechenwerk, dessen englische Bezeichnung *Arithmetic Logic Unit* (ALU) ist, realisiert die Manipulation der Daten und Adressen, indem es die logischen (NOT, AND, OR, XOR, ...) und mathematischen (ADD, SUB, ...) Operationen ausführt.

The memory consists of *registers* for the short-term storage of operands and addresses, and of *main memory*, which is used according to the Von Neumann architecture for storing programs and data.

Der Speicher besteht aus *Registern* zur kurzfristigen Speicherung von Operanden und Adressen sowie *Hauptspeicher*, der gemäß der Von-Neumann-Architektur zur Speicherung der Programme und Daten verwendet wird.

4.1.2

Fetch-Decode-Execute Cycle

The CPU repeats the *fetch-decode-execute cycle* from the moment the system is started until it stops. Each phase of the cycle may take several clock cycles to complete. The phases are:

1. `FETCH`: Fetch the command to be executed from memory and copy it into the instruction register.
2. `DECODE`: The control unit resolves the command into switching instructions for the arithmetic logic unit.
3. `FETCH OPERANDS`: If parameters (operands) for the command exist, they are fetched from memory.
4. `EXECUTE`: The arithmetic logic unit carries out the command.
5. `UPDATE PROGRAM COUNTER`: The program counter register is set to the next command.
6. `WRITE BACK`: The result of the command is stored in a register, or main memory, or sent to an output device.

Modern CPUs and computer systems still operate according to the principle of the Von Neumann cycle and implement a Von Neumann computer. However, a significant difference between modern computers and the Von Neumann architecture is the bus. A single bus that connects the input/output devices directly with the CPU is nowadays impossible.

Von-Neumann-Zyklus

Den *Von-Neumann-Zyklus* (englisch: *Fetch-Decode-Execute Cycle*) wiederholt der Prozessor vom Systemstart bis zu dem Zeitpunkt, an dem der Computer gestoppt wird. Jede Phase des Zyklus kann mehrere Takte in Anspruch nehmen. Die Phasen sind:

1. `FETCH`: Den abzuarbeitenden Befehl aus dem Speicher in das Befehlsregister (*Instruction Register*) kopieren.
2. `DECODE`: Das Steuerwerk löst den Befehl in Schaltinstruktionen für das Rechenwerk auf.
3. `FETCH OPERANDS`: Eventuell verfügbare Parameter (Operanden) für den Befehl aus dem Speicher holen.
4. `EXECUTE`: Das Rechenwerk führt den Befehl aus.
5. `UPDATE PROGRAM COUNTER`: Der Befehlszähler (*Program Counter*) wird auf den nächsten Befehl gesetzt.
6. `WRITE BACK`: Das Ergebnis des Befehls wird in einem Register oder im Hauptspeicher gespeichert oder zu einem Ausgabegerät gesendet.

Auch moderne Prozessoren und Rechnersysteme arbeiten nach dem Prinzip des Von-Neumann-Zyklus und realisieren einen Von-Neumann-Rechner. Eine deutliche Abweichung moderner Computer vom Konzept des Von-Neumann-Rechner ist aber der Bus. Ein einzelner Bus, der die Eingabe-/Ausgabe-Geräte

direkt mit dem Hauptprozessor verbindet, ist heute nicht mehr möglich.

4.1.3

Bus Lines

The *data bus* transmits data between the CPU, main memory, and I/O devices. The number of data bus lines specifies the number of bits that can be transferred per clock cycle. Usually, the number of lines is equal to the size of the registers of the arithmetic logic unit inside the CPU. The data bus bandwidth of modern CPUs is 64 bits. The CPU can, therefore, transfer 64 bits of data within a clock cycle to the main memory and away from it.

Memory addresses and I/O devices are accessed (addressed) via the *address bus*. The number of bus lines specifies the maximum number of addressable memory addresses. If there are 32 bus lines, 32 bits long memory addresses are supported. Such a CPU can address 2^{32} Bits = 4 GB of memory in total. Table 4.1 gives an overview of the address bus and data bus sizes of some CPUs.

Busleitungen

Der *Datenbus* überträgt Daten zwischen Hauptprozessor, Arbeitsspeicher und Peripherie. Die Anzahl der Datenbusleitungen legt fest, wie viele Bits pro Takt übertragen werden können. Üblicherweise ist die Anzahl der Datenbusleitungen identisch mit der Größe der Arbeitsregister im Rechenwerk des Prozessors. Die Datenbusbreite moderner Prozessoren ist 64 Bits. Der Prozessor kann somit 64 Datenbits innerhalb eines Taktes zum und vom Arbeitsspeicher weg übertragen.

Speicheradressen und Peripherie-Geräte werden über den *Adressbus* angesprochen (adressiert). Die Anzahl der Busleitungen legt die maximale Anzahl der adressierbaren Speicheradressen fest. Sind 32 Busleitungen vorhanden, sind 32 Bits lange Speicheradressen möglich. Insgesamt kann ein solcher Prozessor 2^{32} Bits = 4 GB Speicher adressieren. Tabelle 4.1 enthält eine Übersicht über die Adressbus- und Datenbusbreiten einiger Prozessoren.

Table 4.1: Sizes of the Address Bus and the Data Bus of various CPUs

Processor (CPU)	Address bus	max. addressable Memory	Data bus
4004, 4040	4 Bits	2^4 = 16 Bytes	4 Bits
8008, 8080	8 Bits	2^8 = 256 Bytes	8 Bits
8085	16 Bits	2^{16} = 65 kB	8 Bits
8088	20 Bits	2^{20} = 1 MB	8 Bits
8086 (XT)	20 Bits	2^{20} = 1 MB	16 Bits
80286 (AT)	24 Bits	2^{24} = 16 MB	16 Bits
80386SX	32 Bits	2^{32} = 4 GB	16 Bits
80386DX, 80486SX/DX/DX2/DX4	32 Bits	2^{32} = 4 GB	32 Bits
Pentium I/MMX/II/III/IV/D/M, Celeron	32 Bits	2^{32} = 4 GB	64 Bits
Pentium Core Solo/Duo, Core 2 Duo/Extreme	32 Bits	2^{32} = 4 GB	64 Bits
Pentium Pro, Dual-Core, Core 2 Quad, Core i7	36 Bits	2^{36} = 64 GB	64 Bits
Itanium	44 Bits	2^{44} = 16 TB	64 Bits
AMD Phenom-II, Itanium 2, AMD64	48 Bits	2^{48} = 256 TB	64 Bits

The *control bus* transmits the commands (e.g., read and write instructions) from the CPU and status messages from the I/O devices. The control bus also contains lines that are used by I/O devices to transmit the interrupt requests

Der *Steuerbus* überträgt die Kommandos (z.B. Lese- und Schreibanweisungen) vom Prozessor und Statusmeldungen von den Peripheriegeräten. Der Steuerbus enthält auch Leitungen, über die E/A-Geräte dem Prozessor Unterbrechungs-

to the CPU. Typically, the control bus has a maximum size of 10 lines.

anforderungen (Interrupts) signalisieren. Typischerweise ist der Steuerbus höchstens 10 Leitungen breit.

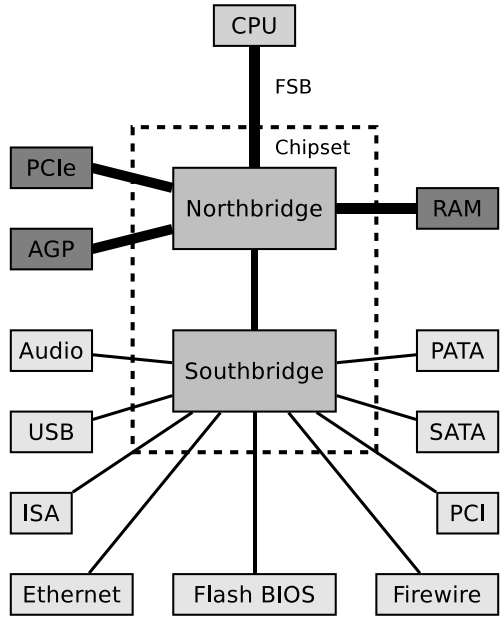

Figure 4.2: Northbridge and Southbridge are used to connect the Components with the CPU

In modern computers, the *chipset* connects the CPU to the rest of the system. It consists of the *northbridge* and the *southbridge* (see Figure 4.2). The northbridge is located close to the CPU for rapid data transfer to and away from it. One of the tasks of the northbridge is to connect the main memory and the graphics adapter(s) to the CPU. The southbridge is used to connect low-speed components.

The bus between CPU and chipset, which contains the address bus, data bus, and control bus, is called *Front-Side-Bus* (FSB) in modern computer systems.

In contrast to the Von Neumann architecture, I/O devices are not directly connected to the CPU. Modern computer systems contain different serial and parallel bus systems that are designed for particular applications. Point-to-point connections are used more and more often. I/O controllers operate between the devices and the CPU. Some selected bus systems are shown in Table 4.2.

In modernen Computern ist der *Chipsatz* das verbindende Element. Dieser besteht aus *Northbridge* und *Southbridge* (siehe Abbildung 4.2). Die Northbridge liegt dicht am Prozessor, um Daten schnell zu dieser bzw. von ihr weg übertragen zu können. Konkret gehört zum Aufgabenbereich der Northbridge die Anbindung des Hauptspeichers und der Grafikkarte(n) an den Prozessor. Die Southbridge ist für die Anbindung langsamerer Komponenten zuständig.

Der Bus zwischen Prozessor und Chipsatz, der den Adressbus, Datenbus und Steuerbus enthält, heißt in modernen Computersystemen *Front-Side-Bus* (FSB).

Anders als im Konzept der Von-Neumann-Architektur sind E/A-Geräte nicht direkt mit dem Prozessor verbunden. Moderne Computer enthalten verschiedene serielle und parallele Bussysteme, die für die jeweilige Anwendungszwecke ausgelegt sind. Immer häufiger werden Punkt-zu-Punkt-Verbindungen eingesetzt. E/A-Controller arbeiten als Vermittler zwischen den

Geräten und dem Prozessor. Einige ausgewählte Bussysteme enthält Tabelle 4.2.

Table 4.2: Some Bus Systems

	Internal computer busses	External computer busses
Parallel busses	PATA (IDE), PCI, ISA, SCSI	PCMCIA, SCSI
Serial busses	SATA, PCI-Express	Ethernet, FireWire, USB, eSATA

For performance and cost reasons, more and more parts of the chipset, such as the memory controller, are relocated into the CPU. Figure 4.3 illustrates this development using some CPUs from the year 2008 as examples. Two of the three CPUs shown in the figure already contain the memory controller, which in previous generations had been a part of the northbridge. Because of this relocation, the northbridge only contains the controller for PCI Express (PCIe).

Aus Geschwindigkeits- und Kostengründen werden zunehmend Teile des Chipsatzes wie beispielsweise der Speichercontroller in den Hauptprozessor verlagert. Abbildung 4.3 zeigt diese Entwicklung exemplarisch anhand einiger Prozessoren aus dem Jahr 2008. Zwei der drei in der Abbildung gezeigten Prozessoren enthalten bereits den Speichercontroller, der bei vorhergehenden Prozessor-Generationen noch ein Teil der Northbridge war. Durch diese Verlagerung enthält die Northbridge nur noch den Controller für PCI-Express (PCIe).

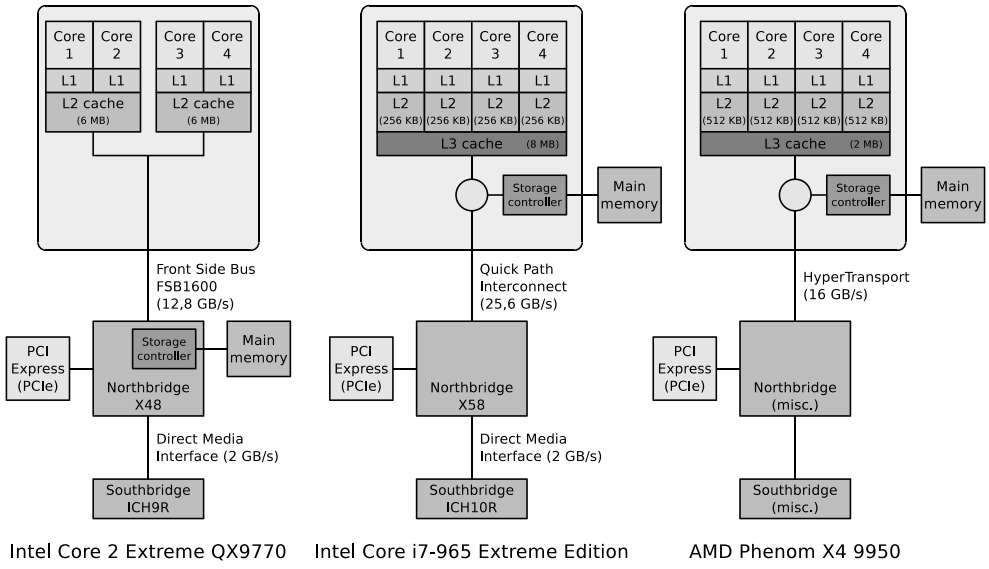

Figure 4.3: The Memory Controller is nowadays often relocated from the Northbridge into the CPU

In some modern computer systems, the entire northbridge is relocated into the CPU. One benefit of this modified architecture is that the price of the overall system is reduced. Figure 4.4 demonstrates this trend. It shows the placement

Bei einigen modernen Computersystemen ist die Funktionalität der Northbridge bereits komplett in den Prozessor verlagert. Ein Vorteil dieser veränderten Architektur sind die geringeren Kosten für das Gesamtsystem. Exemplarisch für

of the functionalities in the chipset generations Intel P55 and P67 from 2009 and 2011. Since that time, the southbridge is also called *Platform Controller Hub* (PCH).

diese Entwicklung zeigt Abbildung 4.4 die Verteilung der Funktionalitäten bei den Chipsatzgenerationen Intel P55 und P67 aus den Jahren 2009 und 2011. Ab dieser Zeit heißt die Southbridge auch *Platform Controller Hub* (PCH).

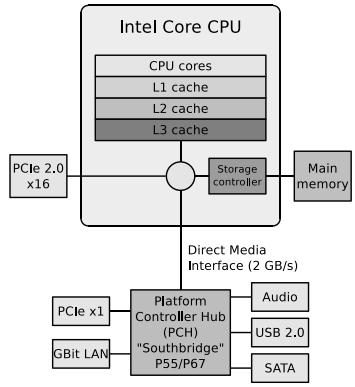

Figure 4.4: Relocation of the Northbridge into the CPU

4.2

Input/Output Devices

Devices that are connected to computer systems are categorized into *character devices* and *block devices*, according to their minimum transfer unit size.

Character devices always communicate with the CPU upon the arrival or request of each character. Examples for such devices are mouse, keyboard, printer, terminal, and magnetic tape. When using block devices, data transfer only takes place when an entire block (e.g., 1-4 kB) is present. Examples of such devices are hard disk drive (HDD), solid-state drive (SSD), CD/DVD drive, and floppy drive. Most block devices support direct memory access (DMA) to transfer data to the main memory without CPU involvement.

If, for example, a record is to be read from a hard disk drive, the following steps are carried out:

1. The CPU receives a request from a process to read a record from a hard disk drive.

Ein-/Ausgabegeräte

Geräte an Computersystemen werden bezüglich der kleinsten Übertragungseinheit in *zeichenorientierte* und *blockorientierte Geräte* unterschieden.

Zeichenorientierte Geräte kommunizieren bei Ankunft bzw. Anforderung jedes einzelnes Zeichens immer mit dem Prozessor. Beispiele für solche Geräte sind Maus, Tastatur, Drucker, Terminal und Magnetband. Bei blockorientierten Geräten findet Datenübertragung erst dann statt, wenn ein vollständiger Block (z.B. 1-4 kB) vorliegt. Beispiele für solche Geräte sind Festplatte, SSD, CD-/DVD-Laufwerk und Disketten-Laufwerk. Die meisten blockorientierten Geräte unterstützen Direct Memory Access, um Daten ohne Beteiligung des Prozessors in den Hauptspeicher zu übertragen.

Soll zum Beispiel ein Datensatz von einer Festplatte gelesen werden, sind folgende Schritte nötig:

1. Der Prozessor bekommt von einem Prozess die Anforderung, einen Datensatz von einer Festplatte zu lesen.

2. The CPU sends an I/O command to the controller by using the driver.
3. The controller locates the record on the hard disk drive.
4. The process receives the requested record.

There are three concepts of how processes can read data into a computer:

- *Busy Waiting.* The device driver sends the request to the device and waits by an *infinite loop* until the controller indicates that the data is available. As soon as the data is made available, it is written into the memory, and the process execution continues. One advantage of this concept is that no additional hardware is required. One drawback is that it slows down the simultaneous execution of multiple processes because the CPU must check periodically whether the data is available.

  One example of a Busy Waiting implementation is the *Programmed Input/Output* (PIO) protocol. The CPU accesses the memory areas of the devices via read and write commands and then copies data between the devices and the main memory (see Figure 4.5). PIO was used among other things for PATA hard disk drives in PIO mode, for the legacy serial interfaces (ports) and the legacy parallel ports, as well as for the PS/2 interface for mouse and keyboard.

- *Interrupt-driven.* The driver initializes the I/O operation and waits for an interrupt from the controller. The driver is *sleeping*. The CPU is not blocked while waiting for the interrupt, and the operating system can assign the CPU to another process. If an interrupt occurs, the driver is *woken up* and gets the CPU assigned. In

2. Der Prozessor schickt dem Controller mit Hilfe des Treibers einen I/O-Befehl.
3. Der Controller lokalisiert den Datensatz auf der Festplatte.
4. Der Prozess erhält die angeforderten Daten.

Es gibt drei Konzepte, wie Prozesse im Computer Daten einlesen können:

- *Busy Waiting* (geschäftiges bzw. aktives Warten). Der Gerätetreiber sendet die Anfrage an das Gerät und wartet in einer *Endlosschleife*, bis der Controller anzeigt, dass die Daten bereit stehen. Stehen die Daten bereit, werden sie in den Speicher geschrieben und die Ausführung des Prozesses geht weiter. Ein Vorteil dieses Konzepts ist es, dass keine zusätzliche Hardware nötig ist. Ein Nachteil ist, dass es die gleichzeitige Abarbeitung mehrerer Prozesse verlangsamt, weil der Prozessor regelmäßig prüfen muss, ob die Daten bereit stehen.

  Ein Beispiel für eine Realisierung von Busy Waiting ist das Zugriffsprotokoll *Programmed Input/Output* (PIO). Dabei greift der Prozessor mit Lese- und Schreibbefehlen auf die Speicherbereiche der Geräte zu und kopiert so Daten zwischen den Geräten und dem Hauptspeicher (siehe Abbildung 4.5). Eingesetzt wurde PIO unter anderem bei PATA-Festplatten im PIO-Modus, bei der seriellen Schnittstelle und bei der parallelen Schnittstelle, sowie bei der PS/2-Schnittstelle für Maus und Tastatur.

- *Interrupt-gesteuert.* Der Treiber initialisiert die E/A-Aufgabe und wartet auf einen Interrupt, also auf eine Unterbrechung durch den Controller. Das bedeutet, dass der Treiber quasi *schläft*. Der Prozessor ist während des Wartens auf den Interrupt nicht blockiert und das Betriebssystem kann den Prozessor einem anderen

the next step, the CPU fetches the data from the controller (by instruction of the device driver) and stores it in the main memory. Then, the interrupted process gets assigned the CPU by the operating system, and process execution continues.

Prozess zuweisen. Kommt es zum Interrupt, wird der Treiber dadurch *geweckt* und bekommt den Prozessor zugewiesen. Im nächsten Schritt holt der Prozessor (auf Anweisung des Gerätetreibers) die Daten vom Controller und legt diese in den Hauptspeicher. Anschließend weist das Betriebssystem den Prozessor dem unterbrochenen Prozess zu, der seine Abarbeitung fortsetzen kann.

Advantages of this concept are that the CPU does not get blocked and that the simultaneous execution of multiple processes is not slowed down. One drawback is that additional hardware in the form of an *interrupt controller* and corresponding *interrupt lines* in the control bus are required for the transmission of the *interrupts*.

Die Vorteile dieses Konzepts sind, dass der Prozessor nicht blockiert wird und dass die gleichzeitige Abarbeitung mehrerer Prozesse nicht verlangsamt wird. Nachteilig ist, dass zusätzliche Hardware in Form eines *Interrupt-Controllers* und entsprechender *Interrupt-Leitungen* im Steuerbus für das Senden der *Interrupts* nötig sind.

- *Direct Memory Access* (DMA). In this concept, data is transferred directly between the main memory and the I/O device using a *DMA controller* (see Figure 4.5). After the data transfer has been completed, the DMA controller triggers an interrupt. I/O devices that use DMA for data transfer include SSDs, hard disk drives, sound cards, network interface cards, and TV/DVB tuner cards.

- *Direct Memory Access* (DMA). Bei diesem Konzept werden Daten über einen *DMA-Controller* direkt zwischen Arbeitsspeicher und E/A-Gerät übertragen (siehe Abbildung 4.5). Nach der Datenübertragung löst der DMA-Controller einen Interrupt aus. Typische E/A-Geräte, bei denen DMA zum Datenaustausch verwendet wird, sind SSD-Laufwerke, Festplatten, Soundkarten, Netzwerkkarten und TV/DVB-Karten.

  An example of a protocol that specifies how data is transferred between the DMA controller and the main memory is *Ultra-DMA* (UDMA). It is the successor protocol of the PIO mode.

  Ein Beispiel für ein Protokoll, das festlegt, wie Daten zwischen DMA-Controller und Arbeitsspeicher übertragen werden, ist *Ultra-DMA* (UDMA). Dabei handelt es sich um den Nachfolger des PIO-Modus.

  One advantage of DMA in comparison to the other concepts is that fetching data causes no CPU workload, and the simultaneous execution of multiple processes is not slowed down. The need for additional hardware in the form of a DMA controller is not a drawback, as it is state of the art that a DMA controller is integrated into the chipset since the late 1980s.

  Ein Vorteil von DMA gegenüber den übrigen Konzepten ist, dass der Prozessor vollständig entlastet und die gleichzeitige Abarbeitung mehrerer Prozesse nicht verlangsamt wird. Die Notwendigkeit zusätzlicher Hardware in Form eines DMA-Controllers ist kein Nachteil, da es seit Ende der 1980er Jahre üblich ist, dass ein DMA-Controller im Chipsatz integriert ist.

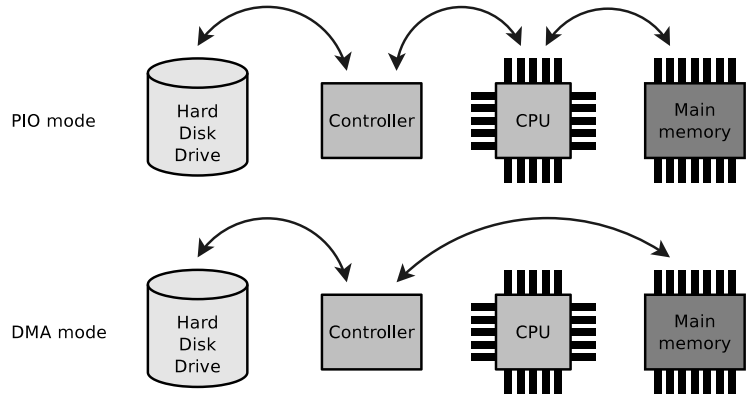

Figure 4.5: Simplified Illustration of how DMA Mode and PIO Mode work

4.3

Digital Data Storage

The Von Neumann architecture, unlike the so-called Harvard architecture, for example, does not differentiate between memory to store programs and memory to store data. The operating system can use the entire main memory that is connected to the computer for all purposes. However, modern computer systems contain different types of memory. These are connected either directly to the CPU by buses or by controllers.

The Harvard architecture, in comparison, logically and physically separates the instruction memory from data memory. Examples for implementations of this architecture concept are Mark I (1944) and the 8-bit Microcontroller AVR from Atmel, which is used on Arduino single-board computers.

Table 4.3 shows an overview of selected data storage technologies and their characteristics. The data storage technologies differ, among other things, in the way read and write operations are performed (electronic, magnetic, mechanical, optical, or magnetic-optical) and in the access mode (random or sequential). *Random access* means that the medium does not need to be searched sequentially from the beginning – such as with magnetic tapes – to locate

Digitale Datenspeicher

Die Von-Neumann-Architektur unterscheidet anders als beispielsweise die sogenannte Harvard-Architektur nicht in Speicher für Programme und Speicher für sonstige Daten. Das Betriebssystem kann den gesamten mit dem Computer verbundenen Hauptspeicher für alle Zwecke verwenden. Dennoch enthalten moderne Computersysteme unterschiedliche Speicher. Diese sind durch Busse entweder direkt mit dem Prozessor oder über Controller mit diesem verbunden.

Im Vergleich dazu ist bei der Harvard-Architektur der Befehlsspeicher für die Programme logisch und physisch vom Datenspeicher getrennt. Beispiele für Implementierungen dieses Architekturkonzepts sind der Mark I von 1944 sowie die 8-Bit-Mikrocontroller AVR von Atmel, die unter anderem auf den Arduino Einplatinencomputern verwendet werden.

Tabelle 4.3 enthält eine Übersicht über ausgewählte Datenspeicher und deren Eigenschaften. Die Datenspeicher unterscheiden sich unter anderem in der Art und Weise, wie Lese- und Schreibzugriffe ausgeführt werden (elektronisch, magnetisch, mechanisch, optisch oder magnetoptisch) und in der Zugriffsart (wahlfrei oder sequentiell). *Wahlfreier Zugriff* heißt, dass das Medium nicht – wie z.B. bei Bandlaufwerken – von Beginn an sequentiell durchsucht werden

a specific record (file). The heads of magnetic disks or the laser of a CD/DVD drive can move to any position of the medium within a known maximum time.

Also, Table 4.3 shows for each data storage technology if it has moving parts, and whether it stores data persistently or not. Moving parts reduce lifetime, increase energy consumption, and cause waste heat. A persistent (non-volatile) storage holds the data for an extended period, even if no electrical energy is applied. In contrast, the data that is stored in volatile storage is lost when no electrical energy is applied. The most common example of volatile memory is main memory.

muss, um eine bestimmte Stelle (Datei) zu finden. Die Köpfe von Magnetplatten oder der Laser eines CD/DVD-Laufwerks können in einer bekannten maximalen Zeit zu jedem Punkt des Mediums springen.

Zudem gibt Tabelle 4.3 an, welche Datenspeicher bewegliche Teile enthalten und ob sie Daten persistent speichern. Bewegliche Teile reduzieren die Lebensdauer, erhöhen den Energieverbrauch und die Abwärme. Ein persistenter bzw. nichtflüchtiger Datenspeicher hält die Daten auch ohne Stromzufuhr für einen längeren Zeitraum. Im Gegensatz dazu sind die in einem flüchtigen Speicher abgelegten Daten beim Wegfall der Stromzufuhr verloren. Das aus dem Alltag bekannteste Beispiel für einen flüchtigen Speicher ist der Hauptspeicher.

4.4

Memory Hierarchy

The different storage technologies in a computer system form a hierarchy that is often represented in the literature (e.g., [49, 50, 116]) as a pyramid (see Figure 4.6). Therefore, the term *storage pyramid* can be used as well. The reason for the existence of the memory hierarchy are the price/performance and price/capacity ratio. The faster the memory is, the more expensive and limited is its capacity. The categorization as *primary storage*, *secondary storage*, and *tertiary storage* is shown in Figure 4.6 is motivated by the memory's connection to the CPU and the computer. The CPU has direct access to the primary storage [50]. Secondary storage is accessed via a controller, and the same applies to tertiary storage. Primary storage and secondary storage are permanently connected to the computer. This does not apply to tertiary storage. Tertiary storage is typically used as archive storage.

Furthermore, tertiary storage is categorized into *nearline storage* and *offline storage*. Nearline storage is automatically and without human intervention connected to the system (e.g., tape library). In the case of offline storage, the media are stored in cabinets or storage rooms and must be connected manually to the system. Ex-

Speicherhierarchie

Die unterschiedlichen Speicher bilden eine Hierarchie, die in der Literatur (zum Beispiel bei [49, 50, 116]) häufig als Pyramide dargestellt ist (siehe Abbildung 4.6). Darum ist auch der Begriff *Speicherpyramide* passend. Der Grund für die Speicherhierarchie sind das Verhältnis von Preis/Leistung und Preis/Speicherkapazität. Je schneller ein Speicher ist, desto teurer und knapper ist er. Die in Abbildung 4.6 gezeigte Unterscheidung in *Primärspeicher*, *Sekundärspeicher* und *Tertiärspeicher* hängt mit der Anbindung des Speichers an den Prozessor und den Computer zusammen. Auf Primärspeicher greift der Prozessor direkt zu [50]. Der Sekundärspeicher wird über einen Controller angesprochen. Auch beim Tertiärspeicher ist für den Zugriff ein Controller nötig. Zudem ist er im Gegensatz zum Primärspeicher und Sekundärspeicher nicht permanent mit dem Computer verbunden. Seine Hauptaufgabe ist die Archivierung.

Tertiärspeicher wird zudem noch unterschieden in *Nearlinespeicher* und *Offlinespeicher*. Nearlinespeicher werden automatisch und ohne menschliches Zutun dem System bereitgestellt. Typische Beispiele für Nearlinespeicher sind Band-Bibliotheken (englisch: *Tape-Library*), bei denen automatisch die Speicherbänder den Laufwerken zugeführt werden. Bei Offlinespeichern

Table 4.3: Some digital Data Storage Technologies and their Characteristics

Storage technology	Write operation	Read operation	Access method	Movable parts	Persistent storage
Punched tape	mechanic		sequential	yes	yes
Punch card	mechanic		sequential	yes	yes
Magnetic tape	magnetic		sequential	yes	yes
Magnetic stripe card	magnetic		sequential	yes	yes
Drum memory	magnetic		random	yes	yes
Magnetic-core memory	magnetic		random	no	yes
Bubble memory	magnetic		random	no	yes
Main memory (DRAM)	electric		random	no	no
Compact cassette (Datasette)	magnetic		sequential	yes	yes
Floppy disk	magnetic		random	yes	yes
Hard disk drive	magnetic		random	yes	yes
Magneto optical disc (MO-Disk)	magneto-optical	optical	random	yes	yes
CD-ROM/DVD-ROM	mechanic	optical	random	yes	yes
CD-R/CD-RW/DVD-R/DVD-RW	optical		random	yes	yes
MiniDisc	magneto-optical	optical	random	yes	yes
Flash memory (USB drive, SSD, CF/SD card)	electric		random	no	yes

amples of offline storage are CD/DVD media and removable hard disk drives.

werden die Medien in Schränken oder Lagerräumen aufbewahrt und müssen von Hand in das System integriert werden. Beispiele für Offlinespeicher sind CD/DVD-Medien sowie Wechselfestplatten.

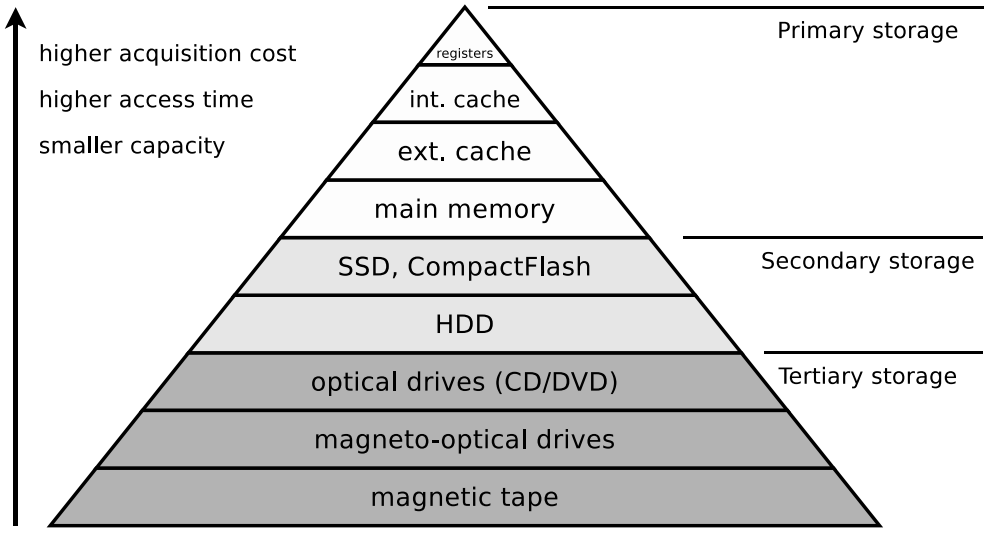

Figure 4.6: The Memory Hierarchy

When a record is accessed for the first time, a copy is created that travels in the memory hierarchy to the top layer (see Figure 4.7). If the record is modified, the modification must be passed down (written back) at some point in time. During the write-back operation, the copies of the record must be updated at all layers to avoid inconsistencies, because modifications cannot be passed directly to the lowest layer (to the original record).

Beim ersten Zugriff auf ein Datenelement wird eine Kopie erzeugt, die entlang der Speicherhierarchie nach oben wandert (siehe Abbildung 4.7). Wird das Datenelement verändert, müssen die Änderungen irgendwann nach unten durchgereicht (zurückgeschrieben) werden. Beim Zurückschreiben müssen die Kopien des Datenblocks auf allen Ebenen aktualisiert werden, um Inkonsistenzen zu vermeiden, denn Änderungen können nicht direkt auf die unterste Ebene (zum Original) durchgereicht werden.

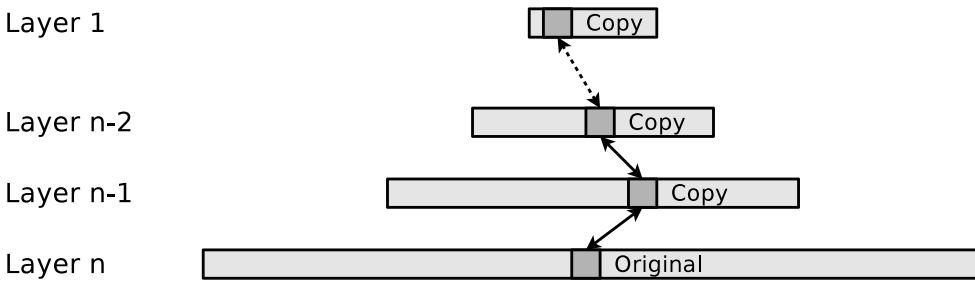

Figure 4.7: Functioning of the Memory Hierarchy

4.4.1

Register

Data inside *registers* can be accessed by the CPU immediately. They operate with the same clock speed as the CPU itself. Some registers that are relevant for the operation of a computer system are:

- *Address registers* are used to store the memory addresses of operands and instructions. Examples of address registers include the *base register*, also called *segment register*, and the *index register* for offset (see Section 5.3.1).
- *Data registers*, also called *accumulators*, store operands for the ALU and their results. Examples include the 32-bit registers *EAX*, *ECX*, *EDX*, and *EBX*, and the corresponding 64-bit registers *RAX*, *RBX*, *RCX*, and *RDX* (see Section 7.3).
- The *program counter*, also called *Instruction Pointer*, stores the memory address of the next command (see Section 4.1.2).
- The *instruction register* stores the command that is currently being executed (see Section 4.1.2).
- The *stack register*, also called *stack pointer*, stores the memory address at the top of the stack (see Section 8.1).

Register

Die *Register* enthalten die Daten, auf die der Prozessor sofort zugreifen kann. Sie sind genauso schnell getaktet wie der Prozessor selbst. Einige für den Betrieb eines Computersystems relevanten Register sind:

- Die *Adressregister* dienen zur Speicherung der Speicheradressen von Operanden und Befehlen. Beispiele für Adressregister sind das *Basisadressregister*, das auch *Segmentregister* genannt wird, und das *Indexregister* für den Offset (siehe Abschnitt 5.3.1).
- Die *Datenregister*, die auch *Akkumulatoren* heißen, speichern Operanden für die ALU und deren Resultate. Beispiele sind die 32-Bit-Register *EAX*, *ECX*, *EDX* und *EBX* und die entsprechenden 64-Bit-Register *RAX*, *RBX*, *RCX* und *RDX* (siehe Abschnitt 7.3).
- Der *Befehlszähler*, der auch *Program Counter* oder *Instruction Pointer* heißt, enthält die Speicheradresse des nächsten Befehls (siehe Abschnitt 4.1.2).
- Das *Befehlsregister*, das auch *Instruction Register* heißt, speichert den aktuellen Befehl (siehe Abschnitt 4.1.2).
- Das *Stapelregister*, das auch *Stack Pointer* heißt, enthält die Speicheradresse am Ende des Stacks (siehe Abschnitt 8.1).

4.4.2

Cache

The *cache* (buffer memory) stores copies of parts of the main memory to accelerate access to this data.

These are usually several levels of cache memory. The *First Level Cache* (L1 cache) is integrated into the CPU. The *Second Level Cache* (L2 cache) operates at a lower clock speed

Cache

Der Pufferspeicher (englisch: *Cache*) enthält Kopien von Teilen des Arbeitsspeichers, um den Zugriff auf diese Daten zu beschleunigen.

Er ist üblicherweise in mehrere Ebenen unterteilt. Der *First Level Cache* (L1-Cache) ist direkt in den Prozessor integriert. Der *Second Level Cache* (L2-Cache) ist mit einer geringe-

and was initially located outside the CPU. Since 1999/2000, hardware manufacturers have integrated the L2 cache into the CPU, too. For this reason, a *Third Level Cache* (L3 cache) was established as a CPU-external cache.

In modern CPUs (e.g., Intel Core i-series and AMD Phenom II), the L3 cache is also integrated into the CPU (see Figure 4.3). In multicore CPUs with integrated L3 cache, the cores share the L3 cache, while each core has its own L1 cache and L2 cache.

Some CPU architectures (e.g., Intel Itanium 2 and some Intel Haswell CPUs) even implement a CPU-external *Fourth Level Cache* (L4 cache).

Typical cache level capacities are:

- L1 cache: 4 kB to 256 kB
- L2 cache: 256 kB to 4 MB
- L3 cache: 1 MB to 16 MB
- L4 cache: 64 MB to 128 MB

The differences in performance between the individual cache levels are significant and vary across existing processor architectures. For example, Table 4.4 shows the latencies for read and write operations for a processor from 2015.

ren Geschwindigkeit getaktet und befand sich ursprünglich außerhalb des Prozessors. Seit den Jahren 1999/2000 integrierten die Hersteller zunehmend den L2-Cache in die Prozessoren. Dies führte zur Etablierung einer weiteren Cache-Ebene, nämlich des *Third Level Cache* (L3-Cache) als Prozessor-externen Cache.

Bei modernen Prozessoren (z.B. Intel Core-i-Serie und AMD Phenom II) ist auch der L3-Cache in den Prozessor integriert (siehe Abbildung 4.3). Bei Mehrkernprozessoren mit integriertem L3-Cache teilen sich die Kerne den L3-Cache, während jeder Kern einen eigenen L1-Cache und L2-Cache hat.

Einige Prozessor-Architekturen (z.B. Intel Itanium 2 und einige Intel Haswell CPUs) haben sogar einen Prozessor-externen *Fourth Level Cache* (L4-Cache).

Typische Kapazitäten der verschiedenen Cache-Ebenen sind:

- L1-Cache: 4 kB bis 256 kB
- L2-Cache: 256 kB bis 4 MB
- L3-Cache: 1 MB bis 16 MB
- L4-Cache: 64 MB bis 128 MB

Die Geschwindigkeitsunterschiede der einzelnen Cache-Ebenen sind drastisch und für die existierenden Prozessorarchitekturen unterschiedlich. Exemplarisch sind in Tabelle 4.4 die Latenzen für Lese- und Schreibzugriffe für einen Prozessor aus dem Jahr 2015 angegeben.

Table 4.4: Latencies of the primary memory levels of an Intel i7-6700 Skylake with 4ĠHz [1, 13]

Memory	Latency in clock cycles	Latency[a] in ns
L1 cache	5	1.25
L2 cache	12	3
L3 cache	42	10,5
Main memory[b]	L3 cache + DRAM latency	10.5 + 51

[a] Latency in ns = latency in clock cycles / processor clock frequency in GHz
[b] DDR4-2400 CL15 (PC-19200)

Cache Write Policies

When writing to the cache, there are two write policies *write-through* and *write-back*.

Cache-Schreibstrategien

Beim Schreiben in den Cache wird zwischen den beiden Schreibstrategien *Write-Through* und *Write-Back* unterschieden.

With write-through, modifications are immediately propagated to lower memory layers. One advantage of this strategy is that data consistency is ensured. A drawback, however, is the lower performance. Figure 4.8 shows a simplified version of the principle. A process that wants to perform a write operation writes the data into the cache in step (1) and sends the write operation to the controller. The controller commands the writing of the data into the storage in step (2). After the data has been successfully written, the controller reports the successful writing of the data to the process in step (3).

With write-back, modifications are not propagated until the corresponding page is removed from the cache. The principle is illustrated in Figure 4.9. A process wishing to perform a write operation writes the data into the cache in step (1) and sends the write operation to the controller. The controller already reports the successful writing of the data to the process in step (2). The writing of the data into the storage in step 3 is therefore asynchronous to the write instruction in the process. An advantage of write-back is better performance, but there is a potential risk of data loss. Modifications to the data in the cache will be lost in the event of a system failure. For each page in the cache, a *dirty bit* is stored in the cache, to indicate whether the page has been modified or not.

The concept of having a dirty bit, indicating whether data has been modified, also exists for the virtual memory in the page tables (see Section 5.3.3) and the segment tables (see Section 5.3.4).

Bei Write-Through werden Änderungen sofort an tiefere Speicherebenen weitergegeben. Ein Vorteil dieser Strategie ist, dass die Konsistenz der Daten gesichert ist. Ein Nachteil ist allerdings die geringere Geschwindigkeit. Das Prinzip wird in Abbildung 4.8 auf vereinfachte Weise dargestellt. Ein Softwareprozess, der eine Schreibanweisung abarbeiten möchte, schreibt in Schritt 1 die Daten in den Cache und übergibt dem Controller die Schreibanweisung. In Schritt 2 weist der Controller das Schreiben der Daten auf dem Datenspeicher an. Nachdem die Daten erfolgreich geschrieben wurde, meldet der Controller in Schritt 3 das erfolgreiche Schreiben der Daten an den Prozess.

Bei Write-Back werden Änderungen erst dann weitergegeben, wenn die betreffende Seite aus dem Cache verdrängt wird. Das Prinzip zeigt Abbildung 4.9. Ein Softwareprozess, der eine Schreibanweisung abarbeiten möchte, schreibt in Schritt 1 die Daten in den Cache und übergibt dem Controller die Schreibanweisung. In Schritt 2 meldet der Controller bereits das erfolgreiche Schreiben der Daten an den Prozess. Das Schreiben der Daten auf dem Datenspeicher in Schritt 3 erfolgt somit asynchron zur Schreibanweisung im Softwareprozess. Der Vorteil der höheren Geschwindigkeit bei Write-Back wird mit der potentiellen Gefahr von Datenverlust erkauft. Änderungen an Daten im Cache gehen beim Systemausfall verloren. Für jede Seite im Cache wird ein *Dirty-Bit* (ebenfalls im Cache) verwaltet, das angibt, ob die Seite geändert wurde.

Das Konzept des Dirty-Bit, das anzeigt ob Daten verändert wurden, gibt es auch beim virtuellen Speicher in den Seitentabellen (siehe Abschnitt 5.3.3) und den Segmenttabellen (siehe Abschnitt 5.3.4).

4.4.3

Main Memory

The *main memory* is a random access memory, and therefore it is also just called *Random Access Memory* (RAM). Another characteristic of the main memory is that it is a volatile memory. Its capacity in modern computer systems is usually several gigabytes.

Hauptspeicher

Der *Hauptspeicher*, der auch *Arbeitsspeicher* oder *Random Access Memory* (RAM) heißt, ist wie der Name es beschreibt, ein Speicher mit wahlfreiem Zugriff. Eine weitere Besonderheit des Hauptspeichers ist, dass er ein flüchtiger Speicher ist. Seine Kapazität auf modernen Computersystemen ist üblicherweise mehrere Gigabyte.

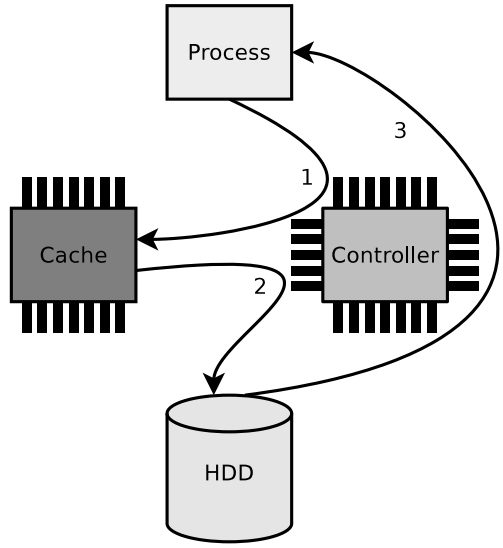

Figure 4.8: Cache Write Policy Write-through

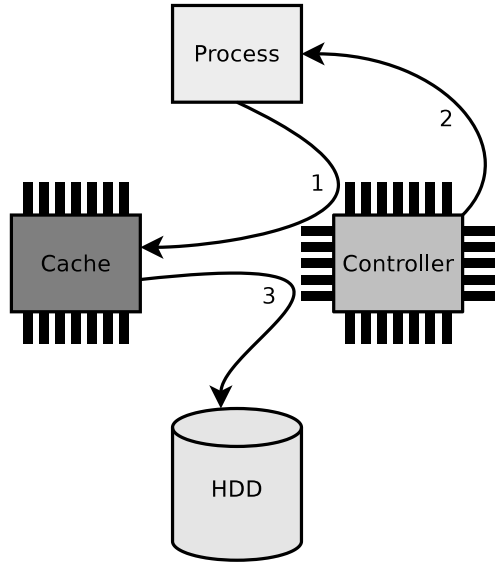

Figure 4.9: Cache Write Policy Write-back

All requests from the CPU that cannot be answered by the cache, are sent to the main memory in the next step.

Alle Anfragen des Hauptprozessors, die nicht vom Cache beantwortet werden können, werden im nächsten Schritt an den Hauptspeicher gestellt.

4.4.4

Hard Disk Drives

Hard disk drives are about 100 times less expensive per bit in comparison to main memory and offer about 100 times more capacity. However, a disadvantage of this storage technology is that accessing hard disk drives is about 1000 times slower than accessing main memory. The reason for the reduced *access time* is that hard disk drives are mechanical devices. They contain one or more disks (also called *platters*), which rotate at 4200, 5400, 7200, 10800, or 15000 revolutions per minute. For each side of each magnetic disk, there is an arm with a *read-and-write* head. It magnetizes areas of the disk surface and thus reads or writes data. The distance between the magnetic disk and the head is a few nanometers.

Usually, hard disk drives include a cache with a capacity of typically 16-32 MB, which buffers the read and write operations.

The surfaces of the magnetic disks are magnetized by the heads in circular *tracks*. All tracks on all disks at a specific arm position are part of a *cylinder*. The tracks are divided into logical units (segments of a circle) called *blocks* or *sectors*. Typically, a sector contains up to 512 bytes or 4 kB of payload. Sectors are the smallest addressable units on hard disk drives. If data is to be modified, the entire sector must be read and rewritten.

The software does not address sectors but so-called *clusters* (see Section 6.1) as the smallest addressable unit. Clusters are groups of sectors with a fixed size (e.g., 4 or 8 kB).

Festplatten

Festplatten sind pro Bit etwa um Faktor 100 preisgünstiger als Hauptspeicher und bieten etwa Faktor 100 mehr Kapazität. Ein Nachteil dieses Datenspeichers ist jedoch, dass Zugriffe auf Festplatten im Vergleich zum Hauptspeicher um ca. Faktor 1000 langsamer sind. Der Grund für die geringere *Zugriffsgeschwindigkeit* ist, dass Festplatten mechanische Geräte sind. Sie enthalten eine oder mehrere Scheiben (englisch: *Platters*), die mit 4200, 5400, 7200, 10800 oder 15000 Umdrehungen pro Minute rotieren. Für jede Seite jeder Magnetplatte existiert ein Schwungarm mit einem *Schreib-/Lesekopf*. Dieser magnetisiert Bereiche der Scheibenoberfläche und schreibt bzw. liest so die Daten. Zwischen Magnetplatte und Kopf befindet sich ein Luftpolster von wenigen Nanometern.

Üblicherweise verfügen Festplatten über einen Cache mit einer Kapazität von typischerweise 16-32 MB, der die Schreib- und Lesezugriffe puffert.

Die Oberflächen der Magnetplatten werden in kreisförmigen *Spuren* (englisch: *Tracks*) von den Köpfen magnetisiert. Alle Spuren auf allen Platten bei einer Position des Schwungarms bilden einen *Zylinder* (englisch: *Cylinder*). Die Spuren sind in logische Einheiten (Kreissegmente) unterteilt, die *Blöcke* oder *Sektoren* heißen. Typischerweise enthält ein Sektor bis zu 512 Bytes oder 4 kB Nutzdaten. Sektoren sind die kleinsten adressierbaren Einheiten auf Festplatten. Müssen Daten geändert werden, muss der ganze Sektor gelesen und neu geschrieben werden.

Softwareseitig sprechen die Betriebssysteme nicht Sektoren, sondern sogenannte *Cluster* (siehe Abschnitt 6.1) als kleinste Zuordnungseinheit an. Cluster sind Verbünde von Sektoren mit fester Größe (z.B. 4 oder 8 kB).

4.4.5

Addressing Data on Hard Disk Drives

Hard disk drives with capacities of up to 8 GB use the so-called *Cylinder-Head-Sector* – CHS addressing. This form of addressing faces several

Adressierung der Daten auf Festplatten

Festplatten mit Größen bis 8 GB verwenden die sogenannte Zylinder-Kopf-Sektor-Adressierung (*Cylinder-Head-Sector-Addressing* – CHS). Diese Form der Adressierung unterliegt mehre-

limitations. The *Parallel ATA* (PATA) interface uses 28 bits for CHS addressing that specifies ...

- 16 bits for the cylinders (up to 65536)
- 4 bits for the heads (up to 16)
- 8 bits for sectors per track (up to 255, because sector number 0 is not used)

The BIOS uses 24 bits for CHS addressing that specifies ...

- 10 bits for the cylinders (up to 1024)
- 8 bits for the heads (up to 255, because head number 0 is not used)
- 6 bits for sectors per track (up to 63, because sector number 0 is not used)

The *Basic Input/Output System* (BIOS) is the firmware of computer systems with x86-compatible processors. It is executed immediately after the computer is switched on and, among other things, initiates the start of the operating system. Modern computers with 64-bit processors typically use the successor to the BIOS, which is the *Unified Extensible Firmware Interface* (UEFI).

For each boundary, the lower value is deciding. For this reason, BIOS versions released before 1995 can usually address a maximum of 504 MB. This value is calculated by multiplying cylinders, heads, and sectors per track – each sector stores 512 bytes of payload.

ren Einschränkungen. Die Schnittstelle *Parallel ATA* (PATA) verwendet 28 Bits für die CHS-Adressierung und davon ...

- 16 Bits für die Zylinder (maximal 65536)
- 4 Bits für die Köpfe (maximal 16)
- 8 Bits für die Sektoren/Spur (maximal 255, da Sektornummer 0 nicht verwendet wird)

Das BIOS verwendet 24 Bits für die CHS-Adressierung und davon ...

- 10 Bits für die Zylinder (maximal 1024)
- 8 Bits für die Köpfe (maximal 255, da Kopfnummer 0 nicht verwendet wird)
- 6 Bits für die Sektoren/Spur (maximal 63, da Sektornummer 0 nicht verwendet wird)

Das *Basic Input/Output System* (BIOS) ist die Firmware von Computersystemen mit x86-kompatiblen Prozessoren. Es wird direkt nach dessen Einschalten des Computers ausgeführt und leitet unter anderem den Start des Betriebssystems ein. Auf modernen Computern mit 64-Bit-Prozessoren wird heute in der Regel der Nachfolger des BIOS, das *Unified Extensible Firmware Interface* (UEFI) verwendet.

Bei den Grenzen ist der jeweils niedrigere Wert entscheidend. Darum können BIOS-Versionen, die vor 1995 erschienen sind, üblicherweise maximal 504 MB adressieren. Der Wert berechnet sich aus der Multiplikation der Zylinder, Köpfe und Sektoren pro Spur. Jeder Sektor speichert 512 Bytes Nutzdaten.

$$1024\,\text{cylinders} \times 16\,\text{heads} \times 63\,\text{sectors/track} \times 512\,\text{bytes/sector} = 528{,}482{,}304\,\text{bytes}$$

$$528{,}482{,}304\,\text{bytes}/1024/1024 = 504\,\text{MB}$$

As it does not make sense for economic reasons to integrate 8 disks with 16 heads in 2.5" or 3.5" format hard disk drives, BIOS versions from the mid-1990s onwards used the so-called *Extended CHS* (ECHS). In this case, not the physical but logical heads are addressed. By increasing the number of heads to up to 255 and decreasing the number of cylinders by the

Da es bei Festplatten im Format 2,5" oder 3,5" aus ökonomischen Gründen nicht sinnvoll ist, 8 Scheiben mit 16 Köpfen einzubauen, verwendeten BIOS-Versionen ab Mitte der 1990er Jahre das sogenannte *Erweiterte CHS* (Extended CHS). Dabei werden nicht die physischen, sondern logische Köpfe adressiert. Durch eine Erhöhung der Anzahl der Köpfe auf maximal 255 und eine Verringerung der Zylinder um den

same factor, up to 7844 GB of capacity can be addressed.

gleichen Faktor sind Kapazitäten bis 7844 GB möglich.

$$1024\,\text{cylinders} \times 255\,\text{heads} \times 63\,\text{sectors/track} \times 512\,\text{bytes/sector} = 8{,}422{,}686{,}720\,\text{bytes}$$

$$8{,}422{,}686{,}720\,\text{bytes}/1024/1024/1024 = 7844\,\text{MB}$$

Hard disk drives with a capacity greater than 7844 GB use *Logical Block Addressing* – LBA. In this form of addressing, all sectors are numbered, starting with 1. Since the usable area per track increases towards the outside, the tracks contain more and more sectors towards the outside. In contrast to this, with CHS addressing all tracks have the same number of sectors. This highlights a major drawback of CHS addressing when compared to LBA addressing. The CHS addressing method wastes storage capacity since the data density decreases from the inner tracks to the outer tracks.

To ensure compatibility, the first 7844 GB of all hard disk drives > 7844 GB can be addressed via CHS addressing.

Festplatten mit einer Kapazität von mehr als 7844 GB verwenden logische Blockadressierung (*Logical Block Addressing* – LBA). Bei dieser Form der Adressierung werden alle Sektoren von 1 beginnend durchnummeriert. Da die nutzbare Fläche pro Spur nach außen hin zunimmt, enthalten die Spuren nach außen hin immer mehr Sektoren. Im Gegensatz dazu sind bei CHS-Adressierung alle Spuren in gleich viele Sektoren unterteilt. Daraus ergibt sich ein großer Nachteil der CHS-Adressierung gegenüber der LBA-Adressierung. CHS-Adressierung verschwendet Speicherkapazität, da die Datendichte mit jeder weiteren Spur nach außen hin immer weiter abnimmt.

Aus Kompatibilitätsgründen können bei allen Festplatten > 7844 GB die ersten 7844 GB via CHS-Adressierung adressiert werden.

4.4.6

Access Time of HDDs

Access time is an important performance indicator of conventional hard disk drives. Two components influence the access time of a hard disk drive:

- The *Average Seek Time* is the time that it takes for the arm to reach a track. For modern hard disk drives, this time is between 5 and 15 ms.

- The *Average Rotational Latency Time* is the delay caused by the rotational speed until the wanted sector is located below the read/write head. Once the head has reached the right track, on average a half rotation of the disk must be waited for the correct sector to be under the head. The Average Rotational Latency Time is the half Rotational Latency Time. This

Zugriffszeit bei Festplatten

Die Zugriffszeit ist ein wichtiges Kriterium für die Geschwindigkeit von konventionellen Festplatten. Zwei Faktoren sind für die Zugriffszeit einer Festplatte verantwortlich:

- Die *Suchzeit* (englisch: *Average Seek Time*) ist die Zeit, die der Schwungarm braucht, um eine Spur zu erreichen. Bei modernen Festplatten liegt diese Zeit zwischen 5 und 15 ms.

- Die *durchschnittliche Zugriffsverzögerung durch Umdrehung* (englisch: *Average Rotational Latency Time*) ist die Verzögerung durch die Drehgeschwindigkeit, bis der Schreib-/Lesekopf den gewünschten Block erreicht. Sobald der Kopf die richtige Spur erreicht hat, muss im Durchschnitt eine halbe Umdrehung der Scheibe abgewartet werden, bis sich der richtige Sektor un-

period depends entirely on the rotational speed of the disks. It is between 2 and 7.1 ms for modern hard disk drives, and is calculated by the following formula:

ter dem Kopf befindet. Die durchschnittliche Zugriffsverzögerung durch Umdrehung entspricht der halben Zugriffsverzögerung durch Umdrehung. Diese Zeitspanne hängt ausschließlich von der Drehgeschwindigkeit der Scheiben ab. Sie liegt bei modernen Festplatten zwischen 2 und 7,1 ms und wird mit der folgenden Formel berechnet:

$$\text{Average Rotational Latency Time [ms]} = \frac{1000\frac{[\text{ms}]}{[\text{sec}]} \times 60\frac{[\text{sec}]}{[\text{min}]} \times 0.5}{\frac{\text{revolutions}}{[\text{min}]}} = \frac{30{,}000\frac{[\text{ms}]}{[\text{min}]}}{\frac{\text{revolutions}}{[\text{min}]}}$$

4.4.7

Solid-State Drives

Solid-state drives (SSD) only contain flash memory, and consequently, unlike hard disk drives, have no moving parts. This is the cause of many advantages over hard disk drives. Examples are shorter access time, less energy consumption, less weight, and better mechanical robustness. Also, there are no noise emissions. Since the position of the data in the semiconductor is irrelevant for access time, defragmentation (see Section 6.8) of SSDs does not influence access time. Furthermore, write operations during defragmentation would unnecessarily reduce the lifetime of the memory cells.

Drawbacks of SSDs, when compared to hard disk drives, are the higher price (this disadvantage will decline in the coming years, as the manufacturing costs of semiconductor drives are continuously decreasing) in comparison to hard disk drives of the same capacity and the fact that it is difficult to securely erase or overwrite data as all write accesses are distributed from the internal controller of the drive to the existing memory cells using a *wear-leveling* algorithm. Another drawback is the already mentioned limited number of write/erase cycles.

Because flash memory cells have a limited lifetime, the drives' built-in controllers use wear-

Solid-State Drives

Solid-State Drives (SSD) enthalten ausschließlich Flash-Speicher und damit im Gegensatz zu Festplatten keine beweglichen Teile. Daraus ergeben sich verschiedene Vorteile gegenüber Festplattenspeicher. Beispiele sind die kürzere Zugriffszeit, der geringere Energieverbrauch, das geringere Gewicht und eine höhere mechanische Robustheit. Zudem gibt es keine Geräuschentwicklung. Da die Position der Daten im Halbleiter für die Zugriffsgeschwindigkeit irrelevant ist, ist das Defragmentieren (siehe Abschnitt 6.8) von SSDs im Hinblick auf die Zugriffsgeschwindigkeit sinnlos. Zudem würden die Schreibzugriffe beim Defragmentieren die Lebenszeit der Speicherzellen unnötig reduzieren.

Nachteile von SSDs gegenüber Festplattenspeicher sind der höhere Preis (dieser Nachteil wird sich in den kommenden Jahren zunehmend abschwächen, da die Herstellungskosten von Halbleiterspeicher kontinuierlich sinken) im Vergleich zu Festplatten gleicher Kapazität sowie die Tatsache, dass ein sicheres Löschen bzw. Überschreiben von Daten schwierig ist, da alle Schreibzugriffe vom internen Controller des Laufwerks auf die vorhandenen Speicherzellen anhand eines *Wear-Leveling*-Algorithmus verteilt werden. Ein weiterer Nachteil ist die bereits erwähnte, eingeschränkte Anzahl an Schreib-/Löschzyklen.

Da die Flash-Speicherzellen nur eine eingeschränkte Lebensdauer haben, verwenden die

leveling algorithms that evenly distribute write operations to the existing memory cells. Modern operating systems also include file systems that are designed to minimize write operations when flash memory is used. Examples of such file systems are the Journaling Flash File System (JFFS), JFFS2, Yet Another Flash File System (YAFFS), and LogFS. JFFS includes a wear-leveling algorithm. This is useful for embedded systems where flash memory is directly connected without a controller that has wear leveling implemented.

Flash memory stores data as electrical charges. In contrast to main memory, there is no electricity required to keep the data in memory. Each flash memory cell (see Figure 4.10) is a floating gate transistor with three connectors (electrodes):

- *Gate*
- *Drain*
- *Source*

The gate is the control electrode. It consists of the control gate and the floating gate that stores the data in the form of electrons. The floating gate is surrounded by an insulator and stores electrical charge like a capacitor. Ideally, the charge stored in it remains stable for years. A positively doped (p) semiconductor separates the two negatively doped (n) electrodes drain and source. Equal to an npn transistor, the npn passage is not conductive without a base current.

eingebauten Controller der Laufwerke Wear-Leveling-Algorithmen, die die Schreibzugriffe auf die verfügbaren Speicherzellen gleichmäßig verteilen. Moderne Betriebssysteme enthalten zudem Dateisysteme, die speziell für Flash-Speicher ausgelegt sind, und darum Schreibzugriffe minimieren. Beispiele für solche Dateisysteme sind Journaling Flash File System (JFFS), JFFS2, Yet Another Flash File System (YAFFS) und LogFS. JFFS enthält einen eigenen Wear-Leveling-Algorithmus. Das ist bei eingebetteten Systemen sinnvoll, wo Flash-Speicher direkt ohne einen Controller mit eigenem Wear Leveling angeschlossen ist.

In Flash-Speicher werden Daten als elektrische Ladungen gespeichert. Im Gegensatz zum Hauptspeicher ist jedoch kein Strom nötig, um die Daten im Speicher zu halten. Jede Flash-Speicherzelle (siehe Abbildung 4.10) ist ein Floating-Gate-Transistor mit den drei Anschlüssen (Elektroden):

- *Tor*
- *Senke*
- *Quelle*

Das Gate ist die Steuerelektrode. Es besteht aus dem Steueranschluss (englisch: *Control-Gate*) und der Ladungsfalle (englisch: *Floating-Gate*), die die Daten in Form von Elektronen speichert. Die Ladungsfalle ist von einem Isolator umgeben und speichert Ladung wie ein Kondensator. Die in ihr gespeicherte Ladung bleibt im Idealfall über Jahre stabil. Ein positiv dotierter (p) Halbleiter trennt die beiden negativ dotierten (n) Elektroden Drain und Source. Wie beim npn-Transistor ohne Basisstrom leitet der npn-Übergang nicht.

4.4.8

Reading Data from Flash Memory Cells

Above a certain positive voltage level (threshold value) at the gate, an n-type channel is created in the p-area. Electric current can flow through this channel between source and drain.

If the floating gate stores electrons, the threshold is changed, and a higher positive volt-

Daten aus Flash-Speicherzellen auslesen

Ab einer bestimmten positiven Spannung, dem sogenannten Schwellwert (englisch: *Threshold*), an Gate entsteht im p-Bereich ein n-leitender Kanal. Durch diesen kann elektrischer Strom zwischen Source und Drain fließen.

Sind Elektronen in der Ladungsfalle, verändert das den Threshold. Es ist eine höhere po-

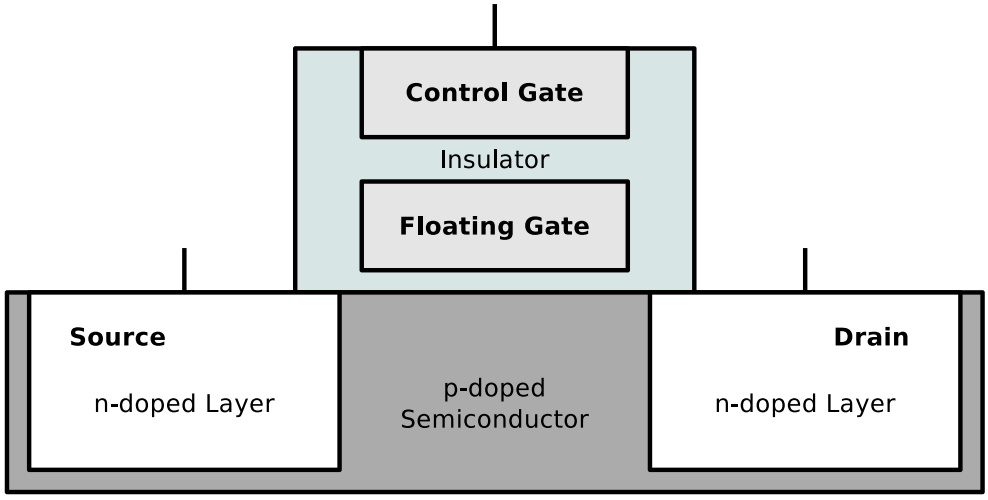

Figure 4.10: Components of a Flash Memory Cell

age at the gate is required so that current can flow between source and drain. This way, the stored value of the flash memory cell is readout.

Writing Data into Flash Memory Cells

Data is stored inside flash memory cells by using *Fowler-Nordheim tunneling*. It allows electrons to pass through an insulating oxide layer. If a sufficiently high positive voltage is applied to the control gate, electrons can flow between source and drain.

If the high enough positive voltage is applied (6 to 20V), some electrons tunnel through the insulator into the floating gate. The method shown in Figure 4.11 is also called *Channel Hot Electron Injection*.

Erasing Data in Flash Memory Cells

To erase a flash memory cell, a significant negative voltage (-6 to -20V) is applied at the control gate. As a result, electrons tunnel in the reverse direction out of the floating gate (see Figure 4.12). The insulating layer, which surrounds the floating gate, suffers from each erase cycle, and at some point, the insulating layer is no longer sufficient to hold the charge in the

sitive Spannung am Gate nötig, damit Strom zwischen Source und Drain fließen kann. So wird der gespeicherte Wert der Flash-Speicherzelle ausgelesen.

Daten in Flash-Speicherzellen schreiben

Schreibzugriffe auf Flash-Speicherzellen werden durch den *Fowler-Nordheim-Tunneleffekt* realisiert. Dieser lässt Elektronen durch eine isolierende Oxidschicht passieren. Wird eine ausreichend große positive Spannung am Control-Gate angelegt, können Elektronen zwischen Source und Drain fließen.

Ist die positive Spannung am Control-Gate groß genug (6 bis 20V), werden einige Elektronen durch den Isolator in das Floating-Gate getunnelt. Das Verfahren, das auch in Abbildung 4.11 dargestellt ist, heißt auch *Channel Hot Electron Injection*.

Daten in Flash-Speicherzellen löschen

Um eine Flash-Speicherzelle zu löschen, wird eine hohe negative Spannung (-6 bis -20V) am Control-Gate angelegt. Die Elektronen werden dadurch in umgekehrter Richtung aus dem Floating-Gate herausgetunnelt (siehe Abbildung 4.12). Die isolierende Schicht, die das Floating-Gate umgibt, leidet allerdings bei jedem Löschvorgang, und ab einem gewis-

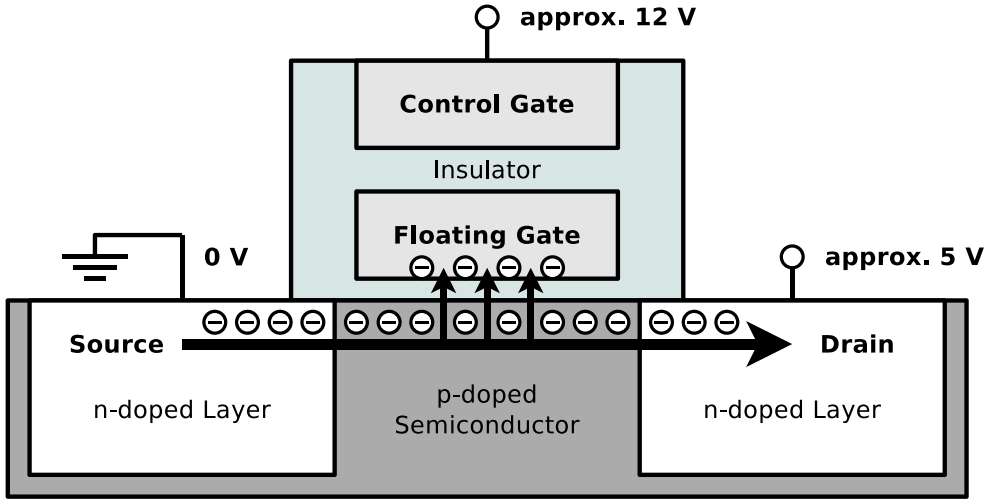

Figure 4.11: Electrons tunnel into the Floating Gate of a Flash Memory Cell

floating gate. For this reason, flash memory survives only a limited number of program/erase cycles. The exact number of possible write/erase cycles depends, among other things, on how many bits a memory cell can store.

sen Punkt ist die isolierende Schicht nicht mehr ausreichend, um die Ladung im Floating-Gate zu halten. Aus diesem Grund überlebt Flash-Speicher nur eine eingeschränkte Anzahl Schreib-/Löschzyklen. Die exakte Anzahl möglicher Schreib-/Löschzyklen hängt unter anderem davon ab, wie viele Bits eine Speicherzelle gleichzeitig speichern kann.

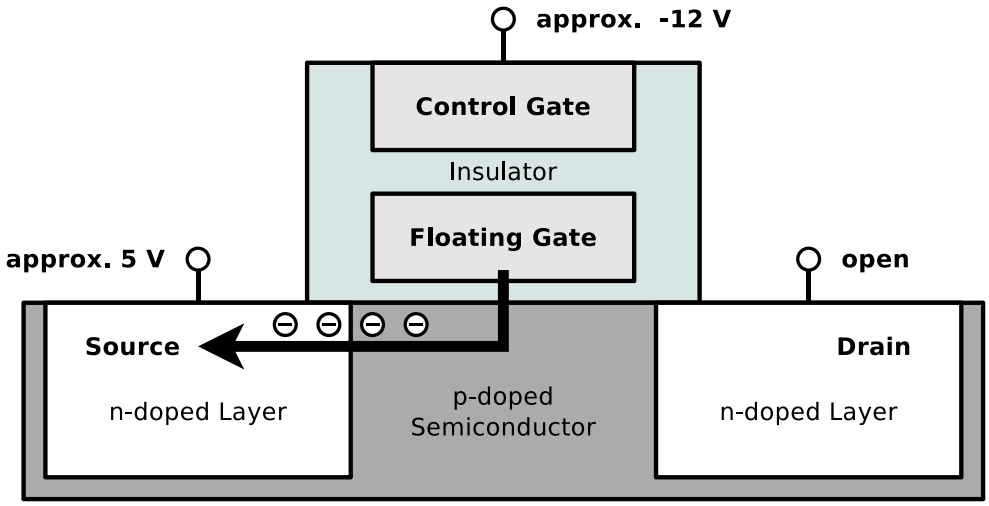

Figure 4.12: Tunneling Electrons out of the Floating Gate of a Flash Memory Cell

Functioning of Flash Memory

The memory cells are arranged in groups of *pages* and *blocks*. Depending on the structure of a flash memory, a block always includes a fixed number of pages. Write and erase operations can only be carried out for entire pages or blocks. For this reason, write and erase operations for flash memories are more complex than read operations. If data in a page is to be modified, the entire block must be erased. For doing this, the block is copied into a memory buffer, in which the data is modified. The block is then erased from the flash memory, and the modified block in the memory buffer is written into the flash memory.

There are two types of flash memory: *NOR memory* and *NAND memory*. The circuit symbol indicates the internal connection of the memory cells and has an impact on the capacity and access time (latency) of the storage.

NOR memory

In NOR memory, each memory cell has its data line. This design allows random access to the memory for read and write operations. The resulting advantage is the shorter access time of NOR memory in comparison to NAND memory. Drawbacks are the more complex and, therefore, more expensive structure, the higher power consumption in comparison to NAND memory, and the typically low storage capacity (less than 32 MB).

NOR memory does not include pages. The memory cells are grouped into blocks. Typical block sizes are 64, 128, or 256 kB. For erase operations, random access is impossible. Erase operations can only be carried out for entire blocks.

Typical applications for NOR memory are industrial environments and components for storing the firmware of a computer system.

NAND memory

In NAND memory, the memory cells are grouped into pages with a typical size of 512 to 8192 bytes each. Each page has a dedicated

Arbeitsweise von Flash-Speicher

Die Speicherzellen sind in Gruppen zu *Seiten* und *Blöcken* angeordnet. Je nach dem Aufbau eines Flash-Speichers enthält ein Block immer eine feste Anzahl an Seiten. Schreib- und Löschoperationen können nur für komplette Seiten oder Blöcke durchgeführt werden. Aus diesem Grund sind Schreib- und Löschoperationen bei Flash-Speicher aufwendiger als Leseoperationen. Sollen Daten in einer Seite verändert werden, muss der gesamte Block gelöscht werden. Dafür wird der Block in einen Pufferspeicher kopiert, in dem die Daten verändert werden. Anschließend wird der Block im Flash-Speicher gelöscht und der veränderte Block vom Pufferspeicher in den Flash-Speicher geschrieben.

Es existieren zwei Arten von Flash-Speicher: *NOR-Speicher* und *NAND-Speicher*. Das jeweilige Schaltzeichen bezeichnet die interne Verbindung der Speicherzellen und beeinflusst die Kapazität und Zugriffsgeschwindigkeit des Speichers.

NOR-Speicher

Bei NOR-Speicher ist jede Speicherzelle über eine eigene Datenleitung angeschlossen. Diese Bauweise ermöglicht den wahlfreien Lese- und Schreibzugriff auf den Speicher. Der resultierende Vorteil ist die bessere Zugriffszeit von NOR-Speicher gegenüber NAND-Speicher. Nachteilig sind der komplexere und somit kostspieligere Aufbau, der höhere Stromverbrauch im Vergleich zu NAND-Speicher und die üblicherweise geringe Kapazität (kleiner als 32 MB).

NOR-Speicher enthält keine Seiten. Die Speicherzellen sind direkt zu Blöcken zusammengefasst. Typische Blockgrößen sind 64, 128 oder 256 kB. Bei Löschoperationen ist kein wahlfreier Zugriff möglich, sondern es muss immer ein vollständiger Block gelöscht werden.

Typische Einsatzbereiche von NOR-Speicher sind industrielle Anwendungen sowie Bausteine zur Speicherung der Firmware eines Computers.

NAND-Speicher

Bei NAND-Speicher sind die Speicherzellen zu Seiten zusammengefasst, die üblicherweise 512 bis 8192 Bytes groß sind. Jede Seite ist über eine eigene Datenleitung angeschlossen. Mehrere Sei-

data line. Multiple pages form a block. Typical block sizes are 32, 64, 128, or 256 pages.

One advantage of NAND memory when compared to NOR memory is that it includes fewer data lines, which results in space-saving and more cost-effective production. A drawback is that random access is impossible, which has a negative impact on access time. Read and write operations can only be performed for entire pages. Erase operations can only be carried out for entire blocks.

Typical applications for NAND memory are solid-state drives (SSD), USB flash memory drives, and memory cards.

Different types of NAND memory exist – *Single-Level Cell* (SLC), *Multi-Level Cell* (MLC), *Triple-Level Cell* (TLC), and *Quad-Level Cell* (QLC). Depending on which group a memory belongs to, each of its memory cells can store 1, 2, 3, or 4 bits (see Figure 4.13).

For an SLC memory cell, only two charge levels (states) have to be distinguished to ensure functionality. For an MLC memory cell, there are four charge levels. For a TLC memory cell, there are eight charge levels, and for a QLC memory cell, there are 16 charge levels that have to be distinguished. It means that it is no longer sufficient to check, as with SLC, whether an electric current can flow between source and drain at a certain positive voltage at the gate. It must also be checked how much voltage is needed at the gate, because there can be several different charge levels at the floating gate.

In a direct comparison of SLC-, MLC-, TLC-, and QLC memory with identical capacity, SLC memory is the most expensive one. In exchange, it provides the best write performance and tends to have the longest lifetime because it survives more write and erase cycles than the other NAND types.

ten umfassen einen Block. Typische Blockgrößen sind 32, 64, 128 oder 256 Seiten.

Ein Vorteil von NAND-Speicher gegenüber NOR-Speicher ist die geringere Anzahl von Datenleitungen, die daraus resultierende Platzersparnis und die preisgünstigere Herstellung. Ein Nachteil ist, dass kein wahlfreier Zugriff möglich ist, was die Zugriffszeit negativ beeinflusst. Lese- und Schreibzugriffe sind nur für ganze Seiten möglich. Löschoperationen sind nur für ganze Blöcke möglich.

Typische Einsatzbereiche von NAND-Speicher sind SSD-Laufwerke, USB-Sticks und Speicherkarten.

NAND-Speicher existiert in den Ausprägungen *Single-Level Cell* (SLC), *Multi-Level Cell* (MLC), *Triple-Level Cell* (TLC) und *Quad-Level Cell* (QLC). Je nachdem, zu welcher Gruppe ein Speicher gehört, können seine einzelnen Speicherzellen 1, 2, 3 oder 4 Bits speichern (siehe Abbildung 4.13).

Während bei einer SLC-Speicherzelle nur zwei Ladungsniveaus (Zustände) unterschieden werden müssen, damit die Funktionalität gegeben ist, müssen bei einer MLC-Speicherzelle vier Ladungsniveaus, bei einer TLC-Speicherzelle acht Ladungsniveaus und bei einer QLC-Speicherzelle 16 Ladungsniveaus unterschieden werden. Dies bedeutet, dass es nicht mehr genügt, wie bei SLC zu überprüfen, ob bei einer bestimmten positiven Spannung am Gate ein elektrischer Strom zwischen Source und Drain fließen kann. Es muss auch überprüft werden, wie viel Spannung am Gate dafür nötig ist, weil es mehrere unterschiedliche Ladungsniveaus in der Ladungsfalle geben kann.

Vergleicht man SLC-, MLC-, TLC- und QLC-Speicher gleicher Kapazität miteinander, dann ist der SLC-Speicher am teuersten, bietet aber dafür die höchste Schreibgeschwindigkeit und tendenziell die höchste Lebensdauer, weil er mehr Schreib-/Löschzyklen übersteht als die übrigen Ausprägungen von NAND-Speicher.

4.5 RAID

RAID

The list of selected hardware components below this paragraph shows that in recent decades, the

Die folgende Auflistung einiger ausgewählter Hardwarekomponenten zeigt, dass in den letz-

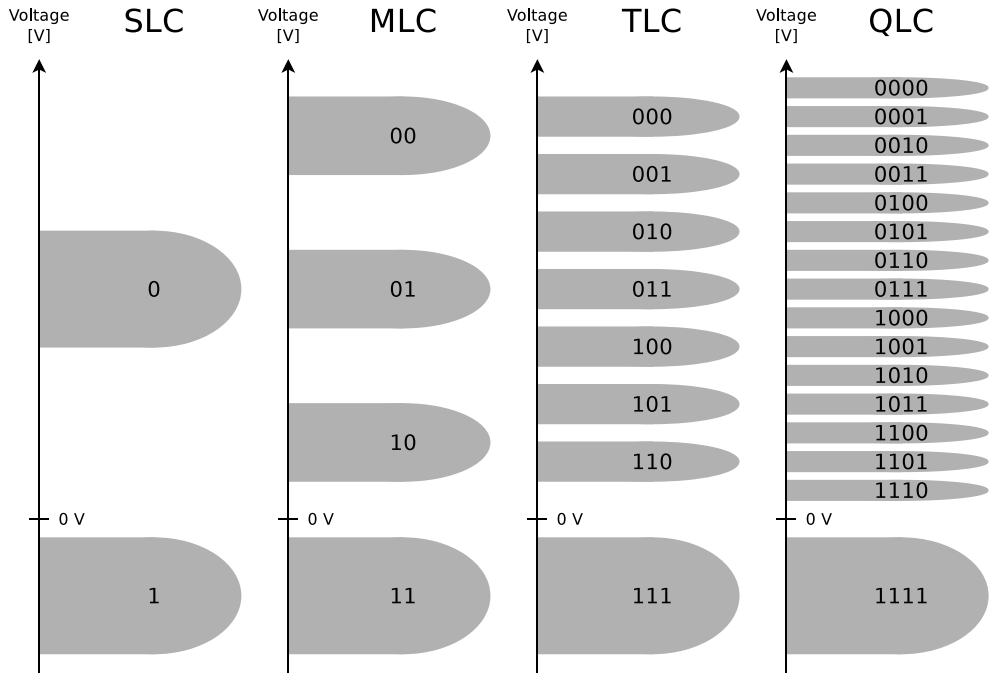

Figure 4.13: The different Types of NAND Flash Memory identify different Numbers of Charge Levels

speed of CPUs (see Table 4.5) has grown much faster than the data access times (latency) of hard disk drives (see Table 4.6).

This gap will continue to grow in the future. Since hard disk drives contain moving parts, their performance cannot be improved infinitely. Their physical, material, and economic limitations must be accepted. SSDs offer better read and write performance than hard disk drives, but they also have performance and capacity limitations. One way to overcome these limitations and improve reliability at the same time is to use multiple drives (hard disk drives or SSDs) in a so-called *RAID* (*Redundant Array of Independent Disks*).

A RAID consists of multiple drives, but for users and their processes, the RAID behaves like a single large drive. The so-called *RAID-Level* specifies how the data is distributed across the drives of a RAID system. Table 4.7 shows

ten Jahrzehnten die Geschwindigkeit der Prozessoren (siehe Tabelle 4.5) sehr viel schneller angewachsen ist als die Zugriffsgeschwindigkeiten der Festplatten (siehe Tabelle 4.6).

Dieser Abstand vergrößert sich in Zukunft weiter. Da Festplatten aus beweglichen Teilen bestehen, lässt sich ihre Geschwindigkeit nicht beliebig verbessern. Die physikalischen, materiellen und wirtschaftlichen Grenzen müssen akzeptiert werden. SSDs bieten höhere Lese- und Schreibgeschwindigkeiten als Festplatten, aber auch bei ihnen gibt es Beschränkungen bzgl. Geschwindigkeit und Kapazität. Eine Möglichkeit, diese Beschränkungen zu umgehen und gleichzeitig die Datensicherheit zu erhöhen, ist das gleichzeitige Verwenden mehrerer Laufwerke (Festplatten oder SSDs) in einem sogenannten *RAID* (*Redundant Array of Independent Disks*).

Ein RAID-Verbund besteht zwar aus mehreren Laufwerken, doch die Benutzer und deren Prozesse nehmen den Verbund als ein einziges großes Laufwerk wahr. Das sogenannte *RAID-Level* spezifiziert die Art und Weise der Vertei-

Table 4.5: Performance parameters of selected processors

Year	Model	Cores	Clock frequency[a]
1971	Intel 4004	1	0.740 MHz
1982	Intel 80286	1	12 MHz
1989	Intel 486DX	1	25 MHz
1994	Intel Pentium P54C	1	100 MHz
1997	AMD K6-2	1	550 MHz
2000	Intel Pentium III Coppermine	1	1000 MHz
2004	Intel Prescott-2M	1	3730 MHz
2007	AMD Athlon 64 X2 6000+	2	3000 MHz
2010	Intel Core i7 980X Extreme	6	3330 MHz
2012	AMD Opteron 6386 SE	16	2830 MHz
2018	AMD Threadripper 2990WX	32	4200 MHz
2021	AMD Threadripper 3990X	64	4300 MHz
2023	AMD Epyc Bergamo 9754	128	3100 MHz

[a] By the beginning of the millennium, it became apparent that further increases in clock frequency had significant drawbacks in terms of power consumption and waste heat. Increasing the clock speed of single-core CPUs was no longer practical, especially for mobile systems such as laptops. In 2005, AMD and Intel introduced the first cost-effective multi-core CPUs. By 2006, multi-core CPUs had become standard for desktops, laptops, and servers. Since then, performance improvements in CPUs have been primarily achieved by integrating additional processor cores.

Table 4.6: Performance parameters of selected hard disk drives

Year	Model	Capacity	Latency[a]
1973	IBM 3340	30 MB	30 ms
1989	Maxtor LXTl00S	96 MB	29 ms
1998	IBM DHEA-36481	6 GB	16 ms
2006	Maxtor STM320820A	320 GB	14 ms
2011	WD WD30EZRSDTL	3 TB	8 ms
2018	Seagate ST14000DM001	14 TB	4-5 ms
2023	WD HC580 0F62795	24 TB	?[a]

[a] For modern hard disk drives, the manufacturers typically no longer specify data access times, as performance is no longer a crucial criterion for this technology. Due to the falling price of SSDs, hard disk drives are primarily used for cost-effective storage of large amounts of data, e.g., in video surveillance systems or for network-attached storage applications. SSDs are used in scenarios where speed is critical. As a result, further improvements in the speed of individual hard disk drives are very unlikely.

an overview of the characteristics of the RAID levels 0 to 6. The most common RAID levels in practice are RAID 0, RAID 1, and RAID 5.

Several different ways exist to implement a RAID. The most powerful variant is *Hardware-RAID*. In this case, a RAID controller is used.

lung der Daten über die Laufwerke eines RAID-Systems. Eine Übersicht mit den Eckdaten der RAID-Level 0 bis 6 enthält Tabelle 4.7. Die in der Praxis gebräuchlichsten RAID-Level sind RAID 0, RAID 1 und RAID 5.

Es gibt verschiedene Möglichkeiten, einen RAID-Verbund technisch zu realisieren. Die leistungsfähigste Variante ist das *Hardware-RAID*.

Table 4.7: Overview of the most common RAID Levels

RAID Level	n (Number of Drives)	net Capacity[a]	allowed to fail[b]	Performance (read)[c]	Performance (write)[c]
0	≥ 2	n	0 (none)	$n * X$	$n * X$
1	≥ 2	1	$n-1$ drives	$n * X$	X
2	≥ 3	$n - [\log_2 n]$	1 drive	variable	variable
3	≥ 3	$n-1$	1 drive	$(n-1) * X$	$(n-1) * X$
4	≥ 3	$n-1$	1 drive	$(n-1) * X$	$(n-1) * X$
5	≥ 3	$n-1$	1 drive	$(n-1) * X$	$(n-1) * X$
6	≥ 4	$n-2$	2 drives	$(n-2) * X$	$(n-2) * X$

[a] If the drives are of different sizes, a RAID 1 array only provides the capacity of the smallest drive.
[b] Indicates how many drives may fail without data loss.
[c] X is the performance of a single drive while reading or writing. The maximum possible performance, in theory, is often limited by the controller or the performance of the CPU.

Such controllers are usually PCI-Express (PCIe) expansion cards, which have interfaces for connecting internal and, possibly, external drives. Furthermore, Hardware-RAID controllers have a powerful processor that calculates the parity information required, depending on the RAID level used, and monitors the state of the connected RAIDs. One advantage of Hardware-RAID over alternative implementations is that it is independent of the operating system. The RAID controller operates transparently for the operating system. It configures and manages the RAID. The operating system uses the RAID in the same manner as a single large drive. One further advantage is that using such a the RAID does not generate CPU load. A drawback is the relatively high price of some hundred Euro for the controller.

Another option for implementing a RAID is the so-called *Host-RAID*. Hence, either an inexpensive RAID controller without a dedicated processor or the mainboard chipset of the computer system provides the RAID functionality. With such a solution, only RAID 0 or RAID 1 is usually recommendable because otherwise, the CPU needs to calculate parity information. This has a negative impact on the overall performance of the computer system. Another drawback is the potentially troublesome dependency on a hardware component that may be difficult to replace in the event of a failure. It cannot be guaranteed that a Host-RAID also

Dabei kommt ein RAID-Controller zum Einsatz. Solche Controller sind meist Steckkarten für PCI-Express (PCIe), die Schnittstellen zum Anschluss von internen und eventuell auch externen Laufwerken bieten. Zudem verfügen Hardware-RAID-Controller über einen leistungsfähigen Prozessor, der die je nach verwendetem RAID-Level benötigten Paritätsinformationen berechnet und den Zustand der angeschlossenen RAID-Verbünde überwacht. Ein Vorteil eines Hardware-RAID gegenüber alternativen Implementierungsvarianten ist die Betriebssystemunabhängigkeit. Der RAID-Controller arbeitet für das Betriebssystem transparent. Er konfiguriert und verwaltet den RAID-Verbund. Das Betriebssystem sieht den RAID-Verbund als ein einzelnes großes Laufwerk. Ein weitere Vorteil ist, dass der Betrieb der RAID-Verbünde den Hauptprozessor nicht belastet. Nachteilig ist der vergleichsweise hohe Preis von einigen hundert Euro für den Controller.

Eine weitere Möglichkeit zur Realisierung eines RAID-Verbunds ist das sogenannte *Host-RAID*. Dabei erbringt entweder ein preiswerter RAID-Controller ohne eigenen Prozessor oder der Chipsatz des Computers die RAID-Funktionalität. Mit solch einer Lösung ist meist nur der Aufbau eines RAID 0 oder RAID 1 empfehlenswert, denn das Berechnen von Paritätsinformationen muss der Hauptprozessor erledigen. Dies wirkt sich negativ auf die Gesamtleistung des Computers aus. Ein weiterer Nachteil ist, dass man sich in ein potentiell problematisches Abhängigkeitsverhältnis von einer Hardware begibt, die im Fehlerfall eventuell nur

functions on a newer motherboard or with another Host-RAID controller. The advantages of host RAID are the low purchase price and, as with Hardware-RAID, operating system independence.

The third possibility is the so-called *Software-RAID*. Modern operating systems like Linux, Windows and Mac OS X provide the software for connecting drives to a RAID without a RAID controller. In Linux, the command `mdadm` exists. When Software-RAID is used, no costs arise for additional hardware. A drawback is a dependency on the operating system because a Software-RAID only functions within the operating system family in which it was created. Just as with Host-RAID, only the construction of a RAID 0 or RAID 1 is recommendable when using Software-RAID, because otherwise, the calculation of parity information needs to be performed by the CPU.

schwer wieder zu beschaffen ist. Dass ein Host-RAID auch auf einem neueren Mainboard oder mit einem anderen Host-RAID-Controller funktioniert, ist keineswegs garantiert. Die Vorteile von Host-RAID sind der geringe Anschaffungspreis und ebenso wie bei Hardware-RAID die Betriebssystemunabhängigkeit.

Als dritte Möglichkeit existiert das sogenannte *Software-RAID*. Moderne Betriebssysteme wie Linux, Windows und Mac OS X ermöglichen das softwaremäßige Zusammenschließen von Laufwerken zu einem RAID auch ohne einen entsprechenden Controller. Unter Linux existiert hierfür das Kommando `mdadm`. Bei Software-RAID entstehen keine Kosten für zusätzliche Hardware. Nachteilig ist die Betriebssystemabhängigkeit, denn ein Software-RAID-Verbund funktioniert nur innerhalb der Betriebssystemfamilie, in der er erzeugt wurde. Genau wie beim Host-RAID ist beim Software-RAID nur der Aufbau eines RAID 0 oder RAID 1 empfehlenswert, denn das Berechnen von Paritätsinformationen muss der Hauptprozessor erledigen.

4.5.1 RAID 0

RAID 0

This RAID level implements so-called *Striping*. It increases the data rate but does not provide redundancy. In a RAID 0, the connected drives are partitioned into blocks of equal size. For sufficiently large input/output jobs (more than 4 or 8 kB), operations can be performed in parallel on several or all drives (see Figure 4.14).

In the event of a drive failure, it is impossible to completely reconstruct the data. RAID 0 should therefore only be used for scenarios where the availability of the data is irrelevant, or where backups are created at regular intervals.

Dieses RAID-Level realisiert das sogenannte *Striping*. Es verbessert die mögliche Datentransferrate, bietet aber keine Redundanz. In einem RAID 0 werden die verbundenen Laufwerke in Blöcke gleicher Größe unterteilt. Bei ausreichend großen Ein-/Ausgabeaufträgen (größer 4 oder 8 kB), können die Zugriffe parallel auf mehreren oder allen Laufwerken durchgeführt werden (siehe Abbildung 4.14).

Fällt allerdings ein Laufwerk aus, können die Daten nicht mehr vollständig rekonstruiert werden. RAID 0 eignet sich darum nur für solche Anwendungsbereiche, wo die Sicherheit (Verfügbarkeit) der Daten bedeutungslos ist oder eine geeignete Form der Datensicherung existiert.

4.5.2 RAID 1

RAID 1

The focus of this RAID level is on availability. It implements a *mirroring* of the data. All drives in the array store the same data (see Figure 4.15). If the drives are of different sizes,

Der Fokus dieses RAID-Levels liegt auf der Sicherheit. Es realisiert eine *Spiegelung* der Daten (englisch: *Mirroring*). Dabei enthalten alle Laufwerke im Verbund die identischen Da-

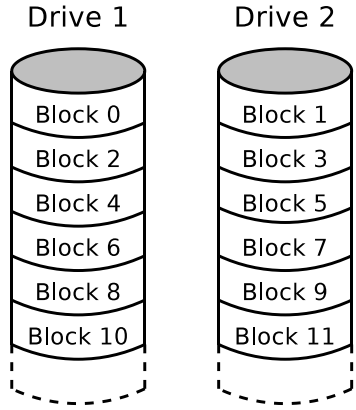

Figure 4.14: RAID 0 implements Block-level Striping

a RAID 1 array only provides the capacity of the smallest drive. The failure of a single drive does not cause data loss because the remaining drives store identical data. A loss of data only occurs if all drives fail.

ten (siehe Abbildung 4.15). Sind die Laufwerke unterschiedlich groß, bietet der Verbund mit RAID 1 höchstens die Kapazität des kleinsten Laufwerks. Der Ausfall eines Laufwerks führt nicht zu Datenverlust, weil die übrigen Laufwerke die identischen Daten vorhalten. Zum Datenverlust kommt es nur beim Ausfall aller Laufwerke.

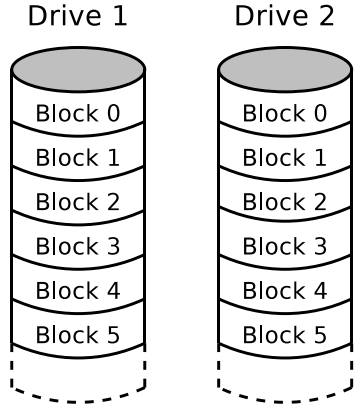

Figure 4.15: RAID 1 implements Block-level Mirroring

RAID 1 cannot improve the data rate for write operations. However, read performance can be improved by an intelligent distribution of the requests to the attached drives.

Since every modification of data is written by all drives, corrupted file operations and attacks

Eine Verbesserung der Schreibgeschwindigkeit ist mit RAID 1 nicht möglich. Die Lesegeschwindigkeit kann allerdings durch eine intelligente Verteilung der Zugriffe auf die angeschlossenen Laufwerke gesteigert werden.

Da jede Datenänderung auf allen Laufwerken geschrieben wird, finden fehlerhafte Dateiopera-

by viruses or other malware take place on all drives. For this reason, neither RAID 1 nor any other RAID level can replace regular backups of important data.

tionen sowie Angriffe durch Viren oder andere Schadsoftware auf allen Laufwerken statt. Aus diesem Grund sollte weder ein RAID 1 noch ein anderes RAID-Level die regelmäßige Sicherung wichtiger Daten ersetzen.

4.5.3

RAID 2

RAID 2

This RAID level implements bit-level striping with *Hamming code error correction*. It means that the data is distributed bit by bit to the drives. All bits whose position numbers are powers of two (1, 2, 4, 8, 16, etc.) are the parity bits. The distribution of the payload data and the parity bits over multiple drives (see Figure 4.16) improves the data rate for reading and writing.

Dieses RAID-Level realisiert Bit-Level Striping mit *Hamming-Code-Fehlerkorrektur*. Das bedeutet, dass die Daten bitweise auf die Laufwerke verteilt werden. Alle Bits, deren Positionsnummer Potenzen von zwei sind (1, 2, 4, 8, 16, usw.), sind die Prüfbits. Die Verteilung der Nutzdaten und der Prüfbits über mehrere Laufwerke (siehe Abbildung 4.16) verbessert den Datendurchsatz beim Lesen und Schreiben.

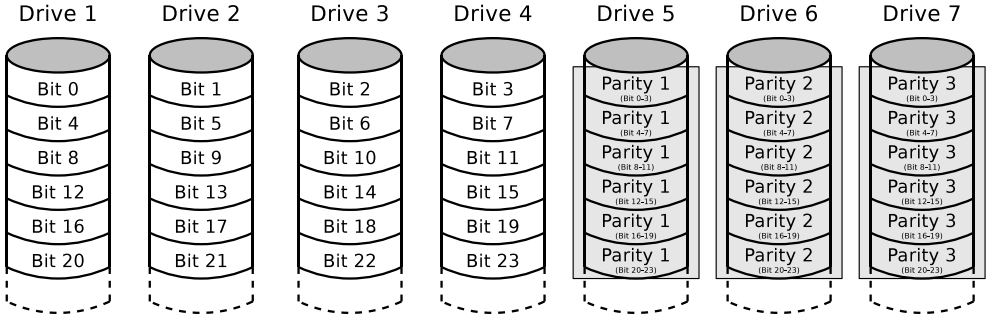

Figure 4.16: RAID 2 implements Bit-level Striping

RAID 2 has mostly been used by mainframe computers. Due to the large number of drives required, RAID 2 was never significantly popular. This RAID level was useful at a time when the drives didn't have an internal Hamming code-based error correction. Since modern hard disk drives and SSDs already detect and correct individual bit errors via Hamming code error correction, RAID 2 no longer makes sense and is no longer used in practice.

Der primäre Anwendungsbereich von RAID 2 waren Großrechner. Wegen der großen Anzahl benötigter Laufwerke war RAID 2 zu keiner Zeit nennenswert populär. Sinnvoll war dieses RAID-Level zu einer Zeit, als die Laufwerke noch keine interne Hamming-Code-Fehlerkorrektur enthielten. Da moderne Festplatten und SSDs bereits via Hamming-Code-Fehlerkorrektur einzelne Bitfehler erkennen und korrigieren, ist RAID 2 nicht länger sinnvoll und wird in der Praxis nicht mehr verwendet.

4.5.4

RAID 3

RAID 3

Byte-Level Striping with parity information is implemented by RAID 3. Parity infor-

Byte-Level Striping mit Paritätsinformationen realisiert RAID 3. Die Paritätsinformationen

mation is stored on a dedicated parity drive (see Figure 4.17). Each write operation on the RAID causes write operations on the parity drive, which is a bottleneck. Furthermore, the parity drive statistically fails more frequently because, in contrast to the data drives, each write operation on the RAID array also causes a write operation on the parity drive. For these reasons, RAID 3 was mostly replaced by RAID 5 in practice.

werden auf einem Paritätslaufwerk gespeichert (siehe Abbildung 4.17). Jede Schreiboperation auf dem RAID führt auch zu Schreiboperationen auf dem Paritätslaufwerk, was einen Engpass (*Flaschenhals*) darstellt. Zudem fällt das Paritätslaufwerk statistisch häufiger aus, weil es im Gegensatz zu den Datenlaufwerken bei jedem Schreibzugriff auf den RAID-Verbund auch einen Schreibzugriff auf das Paritätslaufwerk kommt. Aus diesen Gründen wurde RAID 3 in der Praxis meist von RAID 5 verdrängt.

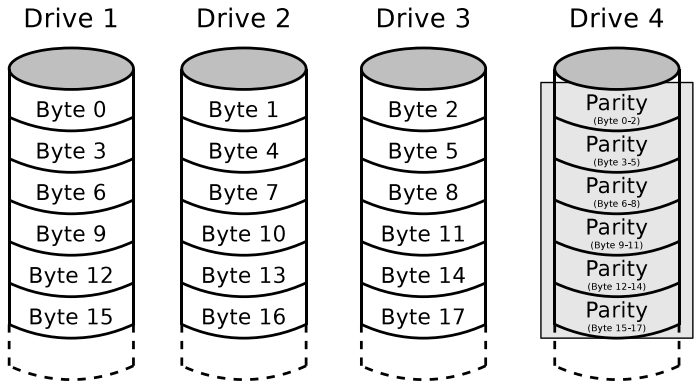

Figure 4.17: RAID 3 implements Byte-level Striping with a dedicated Parity Drive

Table 4.8 shows the calculation of parity information using the scenario shown in Figure 4.17 with three data drives and one parity drive.

Tabelle 4.8 zeigt die Berechnung der Paritätsinformationen anhand des in Abbildung 4.17 gezeigten Szenarios mit drei Datenlaufwerken und einem Paritätslaufwerk.

Table 4.8: Calculation of the Parity Information in RAID 3

Bits on the Payload Drives		Sum		Sum is…		Bit on the Parity Drive
$0+0+0$	$\Longrightarrow$	0	$\Longrightarrow$	even	$\Longrightarrow$	0
$1+0+0$	$\Longrightarrow$	1	$\Longrightarrow$	odd	$\Longrightarrow$	1
$1+1+0$	$\Longrightarrow$	2	$\Longrightarrow$	even	$\Longrightarrow$	0
$1+1+1$	$\Longrightarrow$	3	$\Longrightarrow$	odd	$\Longrightarrow$	1
$1+0+1$	$\Longrightarrow$	2	$\Longrightarrow$	even	$\Longrightarrow$	0
$0+1+1$	$\Longrightarrow$	2	$\Longrightarrow$	even	$\Longrightarrow$	0
$0+1+0$	$\Longrightarrow$	1	$\Longrightarrow$	odd	$\Longrightarrow$	1
$0+0+1$	$\Longrightarrow$	1	$\Longrightarrow$	odd	$\Longrightarrow$	1

4.5.5

RAID 4

Block-Level Striping with parity information is implemented by RAID 4. Just like RAID 3, the parity information is stored on a dedicated parity drive (see Figure 4.18). The difference to RAID 3 is that not single bits or bytes, but blocks (*chunks*) of the same size are stored. Besides this, the drawbacks mentioned for RAID 3 also apply for RAID 4.

RAID 4

Block-Level Striping mit Paritätsinformationen realisiert RAID 4. Genau wie bei RAID 3 werden auch hier die Paritätsinformationen auf einem Paritätslaufwerk gespeichert (siehe Abbildung 4.18). Der Unterschied zu RAID 3 ist, dass nicht einzelne Bits oder Bytes, sondern Blöcke (englisch: *Chunks*) gleicher Größe geschrieben werden. Ansonsten gelten die für RAID 3 genannten Nachteile auch für RAID 4.

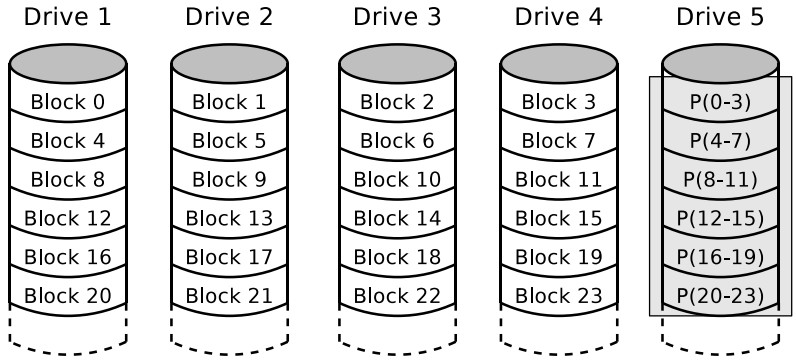

Figure 4.18: RAID 4 implements Block-level Striping with a dedicated Parity Drive

Just like RAID 3, RAID 4 is rarely used in practice, because RAID 5 does not have the drawbacks described above.

A RAID 4 with just a single data drive appears to be similar to a RAID 1 at first glance because, in this case, the same data is stored on the data drive and the parity drive. However, such a RAID 4 array does not provide an improved data rate for read operations. Furthermore, the write speed is decreased by the need to read and recalculate parity information for each write operation. This effort for reading and recalculating does not exist when mirroring with RAID 1 is used.

Application examples for RAID 4 are the NAS servers FAS2020, FAS2050, FAS3040, FAS3140, FAS6080 from NetApp.

Genau wie RAID 3 wird auch RAID 4 in der Praxis selten eingesetzt, weil RAID 5 nicht die bereits beschriebenen Nachteile aufweist.

Ein RAID 4 mit nur einem Datenlaufwerk weist auf den ersten Blick Ähnlichkeit mit einem RAID 1 auf, da in diesem Fall auf dem Datenlaufwerk und auf dem Paritätslaufwerk die gleichen Daten gespeichert sind. Allerdings bietet ein solcher RAID-4-Verbund keine verbesserte Lesegeschwindigkeit. Zudem wird die Schreibgeschwindigkeit durch das bei jedem Schreibzugriff nötige Einlesen und Neuberechnen der Paritätsinformationen verringert. Dieser Aufwand zum Einlesen und Berechnen existiert bei der Spiegelung mit RAID 1 nicht.

Anwendungsbeispiele für RAID 4 sind die NAS-Server FAS2020, FAS2050, FAS3040, FAS3140, FAS6080 der Firma NetApp.

4.5.6

RAID 5

Block-Level Striping with distributed parity information is implemented by RAID 5, where the payload and parity information are both distributed to all drives (see Figure 4.19). The advantages of this technique are an improved data rate for reading and writing, and enhanced availability without having a single parity drive causing a bottleneck.

RAID 5

Block-Level Striping mit verteilten Paritätsinformationen realisiert RAID 5. Dabei werden die Nutzdaten und Paritätsinformationen auf alle Laufwerke verteilt (siehe Abbildung 4.19). Die Vorteile dieses Vorgehens sind neben einem verbesserten Datendurchsatz beim Lesen und Schreiben auch eine Verbesserung der Datensicherheit, ohne dass ein einzelnes Paritätslaufwerk einen Flaschenhals verursacht.

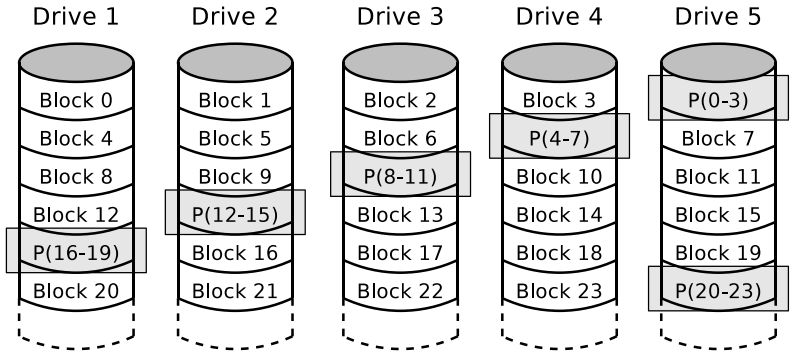

Figure 4.19: RAID 5 implements Block-level Striping with distributed Parity Information

In the same way, as with RAID 4, the payload data is not stored as individual bits or bytes, but as blocks (*Chunks*) of the same size. The blocks are typically 512 bytes and 8 kB in size. Depending on the specific application, it may be useful to use larger blocks, for example, for database systems or E-mail servers.

Parity information is calculated by combining the blocks of a row (see Figure 4.19) with XOR. The calculation below, for example, returns the parity information of the blocks 16 to 19:

Genau wie bei RAID 4 werden die Nutzdaten nicht als einzelne Bits oder Bytes, sondern als Blöcke (englisch: *Chunks*) gleicher Größe geschrieben. Typischerweise sind die Blöcke 512 Bytes und 8 kB groß. Je nach konkreter Anwendung kann es sinnvoll sein, größere Blöcke zu verwenden, zum Beispiel bei Datenbanken oder Email-Servern.

Die Berechnung der Paritätsinformationen erfolgt, indem die Blöcke einer Zeile (siehe Abbildung 4.19) mit XOR verknüpft werden. Die folgende Berechnung erzeugt zum Beispiel die Paritätsinformationen der Blöcke 16 bis 19:

$$\text{Parity}(16\text{–}19) = \text{Block 16 XOR Block 17 XOR Block 18 XOR Block 19}$$

4.5.7

RAID 6

Block-Level Striping with double distributed parity information is implemented by RAID 6. The functioning of this RAID level is the same as RAID 5, but it can handle the simultaneous failure of up to two drives because the parity information is stored twice (see Figure 4.20). The improved availability compared to RAID 5 is obtained by a lower throughput when writing data, because the effort for writing the parity information is higher.

RAID 6

Block-Level Striping mit doppelt verteilten Paritätsinformationen realisiert RAID 6. Dieses RAID-Level funktioniert im Prinzip genau so wie RAID 5, verkraftet aber den gleichzeitigen Ausfall von bis zu zwei Laufwerken, weil die Paritätsinformationen doppelt vorgehalten werden (siehe Abbildung 4.20). Die verbesserte Datensicherheit gegenüber RAID 5 wird durch einen niedrigeren Datendurchsatz beim Schreiben erkauft, da der Schreibaufwand für die Paritätsinformationen höher ist.

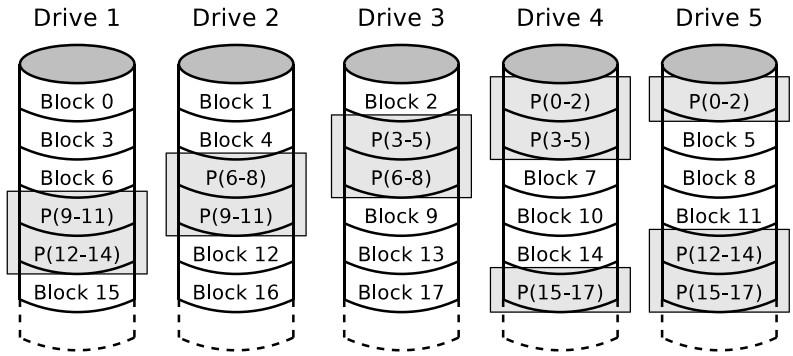

Figure 4.20: RAID 6 implements Block-level Striping with double distributed Parity Information

4.5.8

RAID Combinations

In addition to the popular RAID levels, several RAID combinations exist. It means that at least two RAIDs with possibly different RAID levels are combined to a larger array. One example of such a combination is RAID 10 (see Figure 4.21), where at least two RAID 1 are connected to a huge RAID 0. Further examples of possible combinations are:

- RAID 00: Multiple RAID 0 are connected to a RAID 0
- RAID 01: Multiple RAID 0 are connected to a RAID 1

RAID-Kombinationen

Zusätzlich zu den bekannten RAID-Leveln existieren verschiedene RAID-Kombinationen. Dabei werden mindestens zwei RAID-Verbünde mit eventuell sogar unterschiedlichen RAID-Leveln zu einem größeren Verbund zusammengefasst. Ein Beispiel für eine solche Kombination ist RAID 10 (siehe Abbildung 4.21), bei dem mindestens zwei RAID 1 zu einem großen RAID 0 verbunden sind. Einige weitere Beispiele für mögliche Kombinationen sind:

- RAID 00: Mehrere RAID 0 werden zu einem RAID 0 verbunden
- RAID 01: Mehrere RAID 0 werden zu einem RAID 1 verbunden

- RAID 05: Multiple RAID 0 are connected to a RAID 5
- RAID 15: Multiple RAID 1 are connected to a RAID 5
- RAID 50: Multiple RAID 5 are connected to a RAID 0
- RAID 51: Multiple RAID 5 are connected to a RAID 1

Depending on the combination and the RAID levels used, the availability and performance can be improved. However, this also increases the number of drives that are required.

- RAID 05: Mehrere RAID 0 werden zu einem RAID 5 verbunden
- RAID 15: Mehrere RAID 1 werden zu einem RAID 5 verbunden
- RAID 50: Mehrere RAID 5 werden zu einem RAID 0 verbunden
- RAID 51: Mehrere RAID 5 werden zu einem RAID 1 verbunden

Je nach Kombination und den verwendeten RAID-Leveln ist es auf diese Weise möglich, die Ausfallsicherheit und Geschwindigkeit noch weiter zu steigern. Gleichzeitig steigt damit aber auch die Anzahl der benötigten Laufwerke.

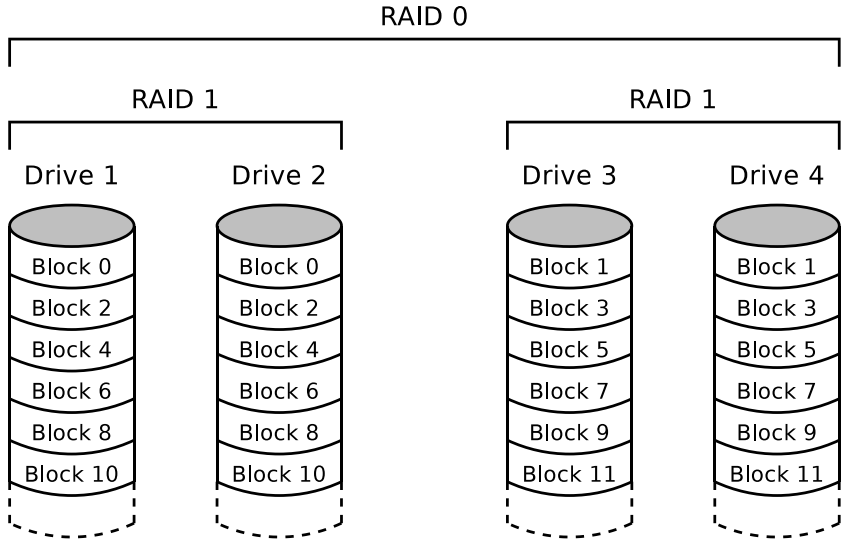

Figure 4.21: RAID 10 implements Block-level Mirroring and Striping

5

Memory Management

Speicherverwaltung

Section 4.3 so far clarified that memory stores the data and programs to be executed and that it forms a hierarchy in the computer system (see also Section 4.4 on the memory pyramid). This chapter introduces several concepts for memory addressing and memory management by an operating system. The operating system assigns portions of the memory to programs during process creation (see Section 8.4) and during process execution at their request. Furthermore, the operating system frees parts of the allocated memory when processes no longer need them.

In Abschnitt 4.3 wurde bislang geklärt, dass der Speicher die Daten und auszuführenden Programme aufnimmt und im Computersystem eine Hierarchie bildet (siehe auch Abschnitt 4.4 zur Speicherpyramide). Dieses Kapitel beschreibt verschiedene mögliche Konzepte der Speicheradressierung und Speicherverwaltung durch ein Betriebssystem. Konkret weist das Betriebssystem den Programmen bei der Prozesserzeugung (siehe Abschnitt 8.4) und während der Prozessausführung auf deren Anforderung hin Teile des Speichers zu. Zudem gibt das Betriebssystem Teile des zugewiesenen Speichers frei, wenn diese von Prozessen nicht länger benötigt werden.

5.1

Memory Management Concepts

Konzepte zur Speicherverwaltung

There are different concepts for the management of memory. This section describes how the three concepts *static* and *dynamic partitioning* and *Buddy memory allocation* work, as well as their advantages and disadvantages.

At this point, it shall be said that static and dynamic partitioning, in contrast to the Buddy memory allocation, are no longer used in practice by modern operating systems. Nevertheless, a discussion of static and dynamic partitioning is useful for understanding the memory management of modern operating systems (see Section 5.3).

Für die Verwaltung des Speichers gibt es verschiedene Konzepte. Dieser Abschnitt beschreibt die Funktionsweise sowie Vor- und Nachteile der drei Konzepte *statische* und *dynamische Partitionierung* sowie *Buddy-Speicherverwaltung*.

An dieser Stelle soll vorweggenommen werden, dass statische und dynamische Partitionierung im Gegensatz zur Buddy-Speicherverwaltung in der Praxis in modernen Betriebssystemen nicht mehr verbreitet sind. Dennoch ist eine Auseinandersetzung mit der statischen und dynamischen Partitionierung sinnvoll, um die Speicherverwaltung moderner Betriebssysteme in der Praxis (siehe Abschnitt 5.3) zu verstehen.

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_5

5.1.1

Static Partitioning

This memory management concept splits the main memory into partitions of equal size or different size (see Figure 5.1). A drawback of this procedure is that internal fragmentation occurs in any case.

Internal fragmentation of the memory occurs when a memory partition is assigned to a process but is not entirely filled with data. Other processes cannot use the unused part of the allocated memory partition.

Consequently, this memory management method is inefficient. With partitions of different sizes, this problem is less severe, but not solved. Another drawback is that the number of partitions limits the number of possible processes [111].

Statische Partitionierung

Bei diesem Konzept der Speicherverwaltung wird der Hauptspeicher in Partitionen gleicher oder unterschiedlicher Größe unterteilt (siehe Abbildung 5.1). Ein Nachteil dieses Verfahrens ist, dass zwangsläufig interne Fragmentierung entsteht.

Interne Fragmentierung des Speichers entsteht, wenn ein Speicherbereich einem Prozess zugeordnet, aber nicht vollständig mit Daten gefüllt ist. Der ungenutzte Teil des zugeordneten Speicherbereich ist für andere Prozesse nicht verfügbar.

Diese Form der Speicherverwaltung ist somit ineffizient. Dieses Problem ist bei Partitionen unterschiedlicher Größe weniger stark ausgeprägt, aber keinesfalls gelöst. Ein weiterer Nachteil ist, dass die Anzahl der Partitionen die Anzahl möglicher Prozesse limitiert [114].

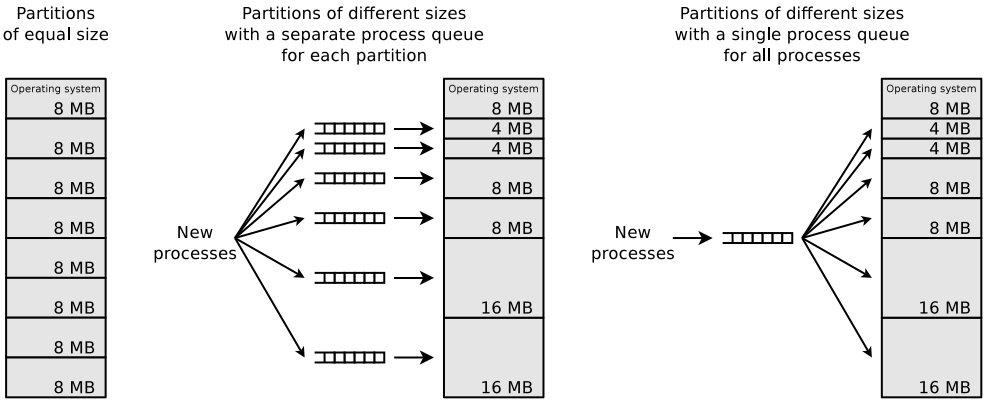

Figure 5.1: Various Options for Memory Management with Static Partitioning

In the best case, a process gets a partition that fits as precisely as possible to cause as little internal fragmentation as possible. If partitions of different sizes are used, there are two methods of assigning partitions to processes. Either the operating system manages a separate process queue for each partition, or it manages a single queue for all partitions.

A queue is a data structure used to cache data objects in a specific order. Usually, queues are implemented as linked lists or as ring buffers.

Im Idealfall erhalten Prozesse eine möglichst passgenaue Partition, um möglichst wenig interne Fragmentierung zu verursachen. Werden Partitionen unterschiedlicher Größe verwendet, gibt es zwei Möglichkeiten, um Prozessen Partitionen zuzuweisen. Entweder verwaltet das Betriebssystem eine eigene Prozesswarteschlange für jede Partition oder es verwaltet eine einzelne Warteschlange für alle Partitionen.

Eine Warteschlange (englisch: *Queue*) ist eine Datenstruktur zur Zwischenspeicherung von Datenobjekten in einer bestimmten Reihenfol-

A drawback of multiple queues is that some partitions may be used less often or even never.

One example of an operating system that uses static partitioning is IBM OS/360 MFT from the 1960s.

ge. Üblicherweise werden Warteschlangen als verkettete Listen oder als Ringpuffer realisiert.

Ein Nachteil mehrerer Warteschlangen ist, dass bestimmte Partitionen seltener oder eventuell sogar nie verwendet werden.

Ein Beispiel für ein Betriebssystem, das statische Partitionierung verwendet, ist IBM OS/360 MFT aus den 1960er Jahren.

5.1.2

Dynamic Partitioning

Using this concept of memory management, the operating system assigns a partition of the exact required size to each process. It inevitably leads to external fragmentation (see Figure 5.2).

External fragmentation of memory occurs when memory management causes gaps between memory partitions that are already assigned to processes. If these gaps are tiny, it is unlikely that they can be allocated to processes in the future. One way to solve the issue of external fragmentation is regular and time-consuming defragmentation. However, this is only possible if the partitions can be relocated. References in processes must not become invalid by relocating partitions.

Entirely solved is the issue of external fragmentation in virtual memory (see Section 5.3.2).

One example of an operating system that uses dynamic partitioning is IBM OS/360 MVT from the 1960s.

Different *allocation concepts* for memory with dynamic partitioning are possible (see Figure 5.3). In this context, it is crucial how the operating system finds a suitable free memory area in the event of a memory request. Memory is represented by the operating system in the form of a linked list with memory partitions. Each entry contains a description of whether the area is free or assigned to a process, the start address, length, and a pointer to the next entry. A more detailed description of the allocation concepts is provided by [108] and [111].

Dynamische Partitionierung

Bei diesem Konzept der Speicherverwaltung weist das Betriebssystem jedem Prozess eine zusammenhängende Partition mit exakt der benötigen Größe zu. Dabei kommt es zwangsläufig zu externer Fragmentierung (siehe Abbildung 5.2).

Externe Fragmentierung des Speichers entsteht, wenn bei der Speicherverwaltung zu Lücken zwischen den Speicherbereichen, die bereits Prozessen zugeordnet sind, kommt. Sind diese Lücken sehr klein, ist es unwahrscheinlich, dass Sie in Zukunft Prozessen zugeordnet werden können. Eine Möglichkeit, um das Problem der externen Fragmentierung zu lösen, ist die regelmäßige und zeitaufwendige Defragmentierung des Speichers. Dieses ist aber nur dann möglich, wenn die Partitionen verschiebbar sind. Verweise in Prozessen dürfen durch ein Verschieben von Partitionen nicht ungültig werden.

Vollständig gelöst ist das Problem der externen Fragmentierung beim virtuellen Speicher (siehe Abschnitt 5.3.2).

Ein Beispiel für ein Betriebssystem, das dynamische Partitionierung verwendet, ist IBM OS/360 MVT aus den 1960er Jahren.

Unterschiedliche *Zuteilungskonzepte* für Speicher bei dynamischer Partitionierung sind denkbar (siehe Abbildung 5.3). Dabei geht es um die Frage, wie das Betriebssystem bei einer Speicheranforderung einen geeigneten freien Speicherbereich sucht. Die Darstellung des Speichers durch das Betriebssystem erfolgt in Form einer verketteten Liste mit Speicherbereichen. Jeder Eintrag enthält eine Bezeichnung, ob der Bereich frei oder durch einen Prozess belegt ist, die Startadresse, die Länge und einen Zeiger auf den nächsten Eintrag. Eine ausführlichere Beschreibung der Zuteilungskonzepte bieten [107] und [114].

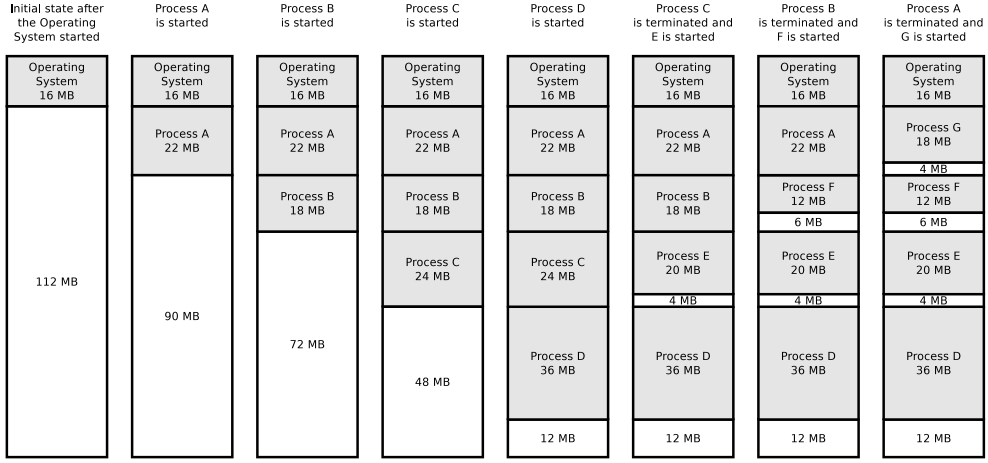

Figure 5.2: External Fragmentation is inevitable in Memory Management with dynamic Partitioning

First Fit

A memory management with the allocation concept *First Fit* searches for a matching free memory partition starting at the beginning of the address space for each memory request. An advantage of First Fit is its speed in comparison to other concepts because it does not inspect all free memory areas in the linked list for every request. The search ends with the first matching free memory partition. One drawback is that First Fit does not utilize the memory as efficiently as possible, because in most scenarios the first free partition of sufficient size is not the one that causes the least external fragmentation, i.e., the one that fits best.

First Fit

Eine Speicherverwaltung mit dem Zuteilungskonzept *First Fit* sucht bei jeder Speicheranforderung ab dem Anfang des Adressraums einen passenden freien Speicherbereich. Ein Vorteil von First Fit ist seine Geschwindigkeit im Vergleich mit anderen Konzepten, weil es nicht in jedem Fall alle freien Speicherbereiche in der verketteten Liste untersucht. Mit dem ersten passenden freien Speicherbereich ist die Suche beendet. Ein Nachteil ist, dass First Fit den Speicher nicht optimal ausnutzt, weil in den meisten Fällen der erste freie Speicherbereich passender Größe nicht derjenige sein wird, der am wenigsten externe Fragmentierung verursacht, der also am besten passt.

Next Fit

With the allocation concept *Next Fit*, the operating system does not always search for a matching partition from the beginning of the address space, but instead from the position of the last allocation onwards. One characteristic symptom of this procedure is that it *fragments* the large area of free memory at the end of the address space faster than other allocation concepts. This means that memory must be defragmented more often when using this method.

Next Fit

Beim Zuteilungskonzept *Next Fit* durchsucht das Betriebssystem den Adressraum nicht immer von Anfang an, sondern ab der letzten Zuweisung nach einem passenden freien Speicherbereich. Symptomatisch für dieses Verfahren ist, dass es den großen Bereich freien Speichers am Ende des Adressraums schneller *zerstückelt* als andere Konzepte. Dadurch muss bei diesem Verfahren der Speicher häufiger defragmentiert werden.

Best Fit

If memory management works according to the concept *Best Fit*, the operating system searches for the free partition that fits the memory request best, i.e., for the partition where the least external fragmentation occurs. One characteristic of this method is that it creates tiny memory partitions (*mini-fragments*). Best Fit uses available memory most efficiently compared to the other memory allocation concepts. A drawback is that its performance is worst compared to all other allocation methods because it checks all free partitions in the linked list for each memory request.

Best Fit

Arbeitet die Speicherverwaltung nach dem Konzept *Best Fit*, sucht das Betriebssystem immer den freien Bereich, der am besten zur Speicheranforderung passt, also denjenigen Block, bei dem am wenigsten externe Fragmentierung entsteht. Symptomatisch für dieses Verfahren ist, dass es sehr kleine Speicherbereiche (*Minifragmente*) erzeugt. Von allen Zuteilungskonzepten nutzt Best Fit den vorhanden Speicher am besten aus. Ein Nachteil ist, dass es im Vergleich mit allen anderen Konzepten am langsamsten arbeitet, weil es bei jeder Speicheranforderung alle freien Speicherbereiche in der verketteten Liste untersucht.

5.1.3

Buddy Memory Allocation

The buddy memory management starts with just a single partition that covers the entire memory. If a process requests a memory area, the requested memory capacity is rounded up to the next higher power of two, and a matching free partition is searched. If no partition of this size exists, the operating system searches for a partition of twice the size and splits it into two parts of equal size, so-called *buddies*. One part is then assigned to the requesting process. If there are no partitions of double the size, the operating system looks for a partition of quadruple the size, and so on.

If a memory partition is deallocated, the operating system checks whether two partitions of the same size can be recombined to a larger memory partition. However, only previously made partitionings are reversed.

Figure 5.4 shows the working method of the buddy memory management based on some memory requests on memory with 1 MB capacity. This capacity may appear very small when considering the storage capacities of modern computer memory, but it is sufficient for this example.

At first, the buddy memory management handles the memory as a single large partition. In step (1), process A requests a 65 kB of memory from the operating system. The next higher

Buddy-Speicherverwaltung

Bei der Buddy-Speicherverwaltung gibt es zu Beginn nur einen Bereich, der den gesamten Speicher abdeckt. Fordert ein Prozess einen Speicherbereich an, wird dessen Speicherkapazität zur nächsthöheren Zweierpotenz aufgerundet und ein entsprechender, freier Bereich gesucht. Existiert kein Bereich dieser Größe, sucht das Betriebssystem nach einem Bereich doppelter Größe und unterteilt diesen in zwei Hälften, sogenannte *Buddies*. Eine Hälfte wird daraufhin dem anfordernden Prozess zugewiesen. Existiert auch kein Bereich doppelter Größe, sucht das Betriebssystem einen Bereich vierfacher Größe, usw.

Wird ein Speicherbereich freigegeben, prüft das Betriebssystem, ob sich zwei Hälften gleicher Größe wieder zu einem größeren Speicherbereich zusammenfassen lassen. Es werden aber nur zuvor vorgenommene Unterteilungen rückgängig gemacht.

Abbildung 5.4 zeigt die Arbeitsweise der Buddy-Speicherverwaltung anhand einiger Speicheranforderungen und -freigaben auf einen Speicher mit 1 MB Gesamtkapazität. Diese Kapazität erscheint unter Berücksichtigung der Speicherkapazitäten moderner Datenspeicher sehr gering, ist aber für ein Beispiel ausreichend.

Zu Beginn behandelt die Buddy-Speicherverwaltung den Speicher wie einen großen Bereich. In Schritt (1) fordert ein Prozess A vom Betriebssystem einen 65 kB

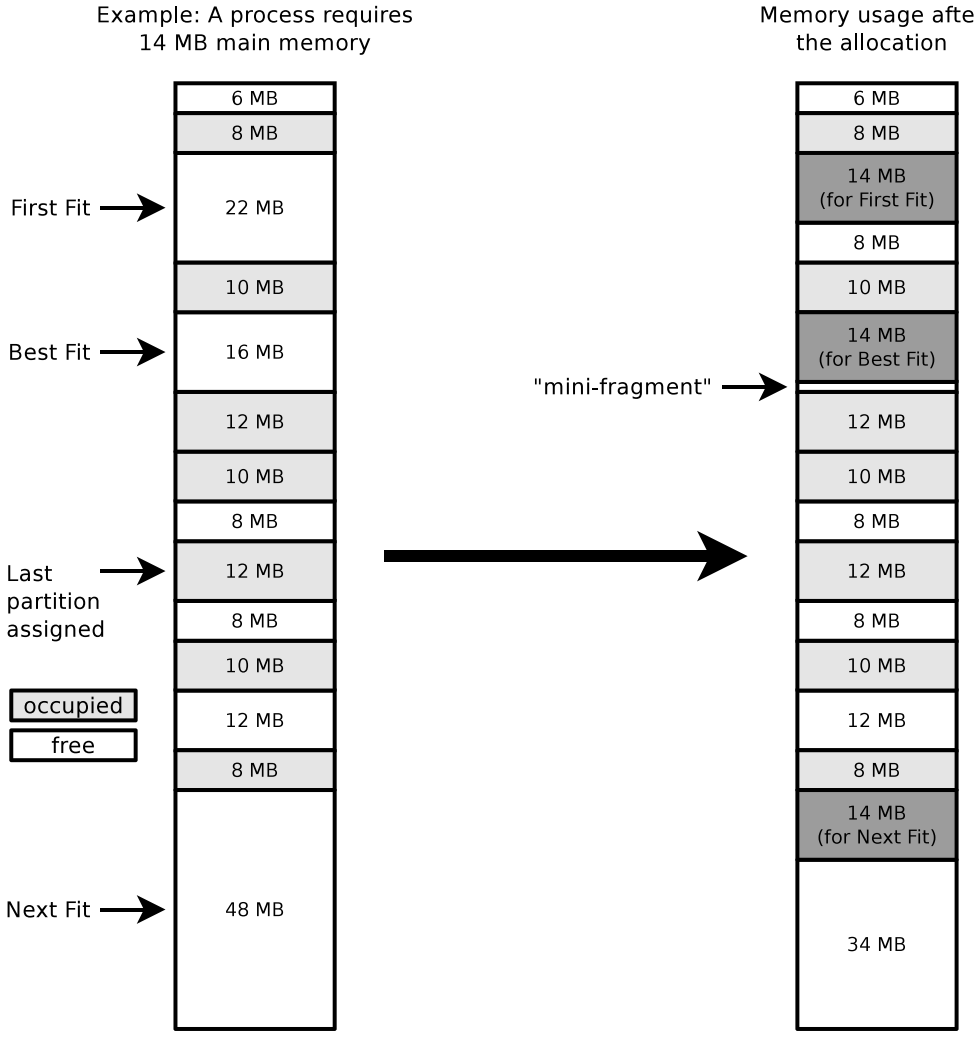

Figure 5.3: Example of a Memory Allocation for dynamic Partitioning with different Allocation Concepts

power of two is 128 kB. Since no such partition is available, the operating system searches for a free partition whose capacity corresponds to the next higher power of two (256 kB). Such a partition also does not exist, and there is no partition of 512 kB in size. However, a 1024 kB partition is available. It is now split until a 128 kB partition exists, that can be assigned to the requesting process A.

großen Speicherbereich an. Die nächsthöhere Zweierpotenz ist 128 kB. Da kein solcher Bereich frei ist, sucht das Betriebssystem nach einem freien Bereich dessen Kapazität der nächsthöhere Zweierpotenz (256 kB) entspricht. Ein solcher Bereich ist genau so wenig verfügbar wie ein 512 kB großer Bereich. Es ist aber ein 1024 kB großer Bereich verfügbar. Dieser wird nun so oft unterteilt, bis ein 128 kB großer

The procedure for the memory request in step (2) has a similar effect. Here, process B requests the 30 kB of memory from the operating system. The next higher power of two is 32 kB. Since no such partition is available, the operating system searches for a free partition whose capacity corresponds to the next higher power of two (64 kB). Such a partition is not available, so the first non-assigned partition with a size of 128 kB is split until a partition with a size of 32 kB becomes available, which can be assigned to the requesting process B.

Analogous to the requests (1) and (2), the allocation of a 128 kB memory partition to process C is done in step (3) so that its 94 kB of data can be stored. Here, again, a partition with the size of the next higher power of two must first be split into two parts.

Process D requests a partition with a size of 34 kB in step (4). The next higher power of two is 64 kB, and this request can be fulfilled directly. Process E requires a partition of 256 kB for the fulfillment of its request for 136 kB of memory in step (5). Since no such area is available, the 512 kB area is split.

Deallocating the memory of process D in step (6) does not cause the partition boundaries to change. A combination with the free neighboring partition that has a size of 32 kB is impossible because buddy memory management can only reverse partitionings in the correct order.

After deallocating the memory of process B in step (7), the two 32 kB partitions can be recombined to a 64 kB partition. Thus, the previous partitioning is reversed. Next, the 64 kB partition can be recombined with its neighbor of the same size to form a 128 kB partition.

Deallocating the memory of the processes C, A, and F in the steps (8-10) causes that previously made partitionings of partitions are reversed. Finally, the initial state is re-established, and the buddy memory management treats the memory as a single area.

Bereich verfügbar ist, der dem anfragenden Prozess A zugewiesen werden kann.

Ähnlich ist der Ablauf bei der Speicheranforderung in Schritt (2). Hier fordert ein Prozess B vom Betriebssystem einen 30 kB großen Speicherbereich an. Die nächsthöhere Zweierpotenz ist 32 kB. Da kein solcher Bereich frei ist, sucht das Betriebssystem nach einem freien Bereich dessen Kapazität der nächsthöheren Zweierpotenz (64 kB) entspricht. Auch ein solcher Bereich ist nicht frei verfügbar, also wird der erste 128 kB große Bereich so oft unterteilt, bis ein 32 kB großer Bereich verfügbar ist, der dem anfragenden Prozess B zugewiesen werden kann.

Analog zu den Anfragen (1) und (2) erfolgt die Zuweisung eines 128 kB großen Speicherbereich zu Prozess C in Schritt (3), um dessen 94 kB Daten zu speichern. Auch hier muss zuerst ein Bereich mit der Größe der nächsthöheren Zweierpotenz in zwei Teile unterteilt werden.

Prozess D fordert in Schritt (4) einen 34 kB großen Speicherbereich an. Die nächsthöhere Zweierpotenz ist 64 kB und diese Anfrage kann direkt erfüllt werden. Prozess E benötigt zur Erfüllung seiner Anfrage nach einem 136 kB großen Speicher in Schritt (5) einen 256 kB großen Bereich. Da kein solcher zur Verfügung steht, wird der 512 kB große Bereich unterteilt.

Die Freigabe des Speichers von Prozess D in Schritt (6) führt zu keiner Veränderung der Bereichsgrenzen. Eine Verbindung mit dem freien 32 kB großen benachbarten Bereich ist unmöglich, da die Buddy-Speicherverwaltung Unterteilungen nur dann rückgängig machen kann, wenn die korrekte Reihenfolge eingehalten wird.

Nach der Freigabe des Speichers von Prozess B in Schritt (7) können die beiden 32 kB großen Bereiche wieder zu einem 64 kB großen Bereich vereinigt werden. Die zuvor vorgenommene Unterteilung wird also rückgängig gemacht. Danach kann der 64 kB große Bereich wieder mit seinem gleich großen Nachbarn zu einem 128 kB großen Bereich vereinigt werden.

Die Freigaben des Speichers der Prozesse C, A und F in den Schritten (8-10) führen dazu, dass immer wieder zuvor gemachte Unterteilungen von Bereichen rückgängig gemacht werden. Am Ende ist der Anfangszustand erneut erreicht und die Buddy-Speicherverwaltung sieht den Speicher wie einen zusammenhängenden Bereich.

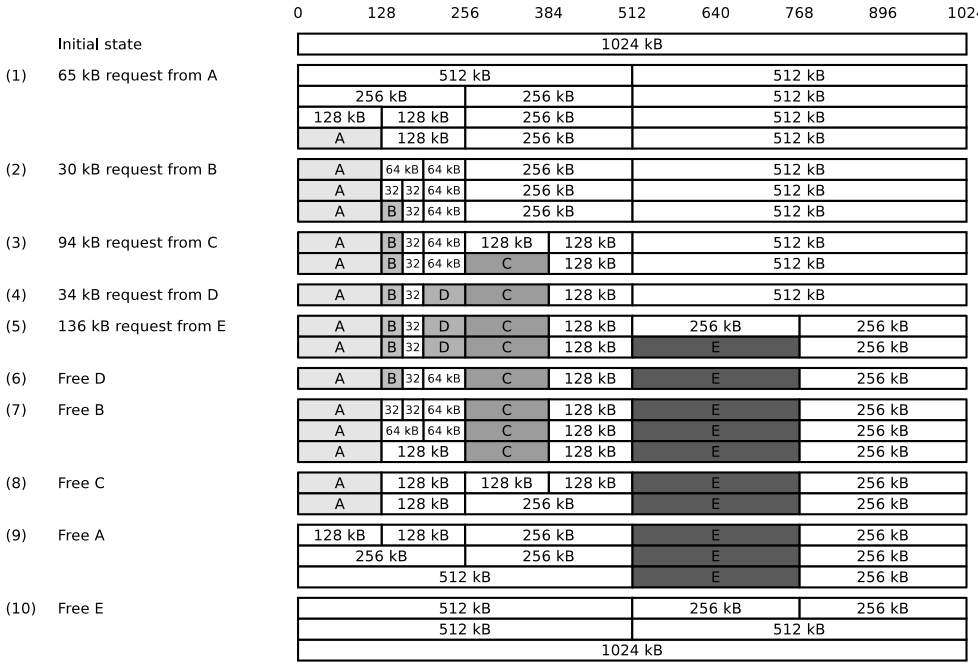

Figure 5.4: Working Method of Buddy Memory Management

Figure 5.4 shows a significant drawback of the buddy memory management method. Internal and external fragmentation occurs.

Even though the buddy method has already been introduced in 1965 by Kenneth Knowlton [62] and in 1968 by Donald Knuth [63] for memory management on computer systems, it is still relevant in practice today. For example, the Linux kernel uses a variant of buddy memory management for assigning memory pages to processes. In this case, the operating system manages a free-list for each supported partition size.

The file `/proc/buddyinfo` provides up-to-date information about the memory management in Linux.

Abbildung 5.4 zeigt einen deutlichen Nachteil der Buddy-Speicherverwaltung. Es kommt zu interner und externer Fragmentierung.

Obwohl das Buddy-Verfahren schon 1965 von Kenneth Knowlton [62] und 1968 von Donald Knuth [63] zum Zweck der Speicherverwaltung bei Computern beschrieben wurde, ist es auch heute noch in der Praxis relevant. Der Linux-Betriebssystemkern verwendet beispielsweise eine Variante der Buddy-Speicherverwaltung für die Zuweisung der Speicherseiten zu den Prozessen. Das Betriebssystem verwaltet in diesem Fall für jede möglich Blockgröße eine Frei-Liste.

Die aktuellsten Informationen zum Zustand der Speicherverwaltung unter Linux enthält die Datei `/proc/buddyinfo`.

```
# cat  /proc/buddyinfo
Node 0,  zone      DMA      1      1      1      0      2      1      1      0      1      1      3
Node 0,  zone    DMA32    208    124   1646    566    347    116    139    115     17      4    212
Node 0,  zone   Normal     43     62    747    433    273    300    254    190     20      8    287
```

The `DMA` row shows the partitioning of the first 16 MB memory of the system. The `DMA32` row shows the partitioning of the memory greater than 16 MB and smaller than 4 GB in the sys-

Die Zeile `DMA` zeigt die Aufteilung der ersten 16 MB im System. Die Zeile `DMA32` zeigt die Aufteilung des Speichers größer 16 MB und kleiner

tem, and the `Normal` row shows the partitioning of the memory greater than 4 GB.

The first column indicates the number of free chunks ("*buddies*") of size $2^0 \times$ page size [bytes]. The command `getconf PAGESIZE` returns the page size in bytes on the command line in Linux.

```
$ getconf PAGESIZE
4096
```

For the majority of common hardware architectures (e.g., x86), the pages are 4096 bytes = 4 kB in size. Therefore, for x86-compatible CPUs, the first column in the `/proc/buddyinfo` file, which contains solely numbers, indicates the number of free chunks of size 4 kB. The second column contains the number of free chunks of size 8 kB, the third column the chunks of size 16 kB, etc. The last column (number 11) indicates how many free 4096 kB = 4 MB large chunks exist in the system.

The information shown in this work from the `buddyinfo` file was taken from a 64-bit Linux operating system. On 32-bit systems, the rows have different names, and they represent different parts of the memory. In such systems, there is no `DMA32` row present in the output. The `Normal` row indicates the partitioning of memory greater than 16 MB and smaller than 896 MB in the system. Furthermore, there is a row labeled `HighMem`, which indicates the partitioning of the memory greater than 896 MB in the system.

Spalte 1 enthält die Anzahl der freien Blöcke („*Buddies*“) mit der Größe $2^0 *$ Seitengröße [Bytes]. Die Seitengröße in Bytes liefert auf der Kommandozeile unter Linux der Befehl `getconf PAGESIZE`.

Auf den meisten gängigen Hardwarearchitekturen (u.a. x86) sind die Seiten 4096 Bytes = 4 kB groß. Dementsprechend informiert bei x86-kompatiblen Prozessoren die erste Spalte in der Datei `/proc/buddyinfo`, die ausschließlich Zahlen enthält, über die Anzahl der freien Blöcke mit der Größe 4 kB. Die zweite Spalte mit ausschließlich numerischen Werten enthält die Anzahl der freien Blöcke mit der Größe 8 kB, die dritte Spalte diejenigen Blöcke mit der Größe 16 kB, usw. Die letzte Spalte (Nummer 11) gibt an, wie viele freie 4096 kB = 4 MB große Blöcke im System existieren.

Die in diesem Buch gezeigten Informationen aus der Datei `buddyinfo` stammen von einem 64-Bit-Linux-Betriebssystem. Auf 32-Bit-Systemen sind die Benennungen der Zeilen sowie die Speicherbereiche, die diese beschreiben, anders. Dort gibt es die Zeile `DMA32` gar nicht. Die Zeile `Normal` beschreibt die Aufteilung des Speichers größer 16 MB und kleiner 896 MB im System. Zusätzlich gibt es eine Zeile `HighMem`, die die Aufteilung des Speichers größer 896 MB im System beschreibt.

5.2

Further Memory Management Concepts

The Linux operating system kernel does not exclusively use the buddy method to manage physical memory, but also implements the so-called *slab allocator* (see Figure 5.5) as another memory management concept for efficiency reasons. The slab allocator works on top of the buddy method and allows a more efficient allocation of small memory areas (smaller than a

Weitere Konzepte zur Speicherverwaltung

Der Linux-Betriebssystemkern verwendet nicht ausschließlich das Buddy-Verfahren zur Verwaltung des physischen Speichers, sondern implementiert aus Effizienzgründen noch den sogenannten *Slab Allocator* (siehe Abbildung 5.5) als weiteres Konzept zur Speicherverwaltung. Der Slab Allocator baut auf dem Buddy-Verfahren und ermöglicht eine effizientere Zuweisung klei-

page) to data objects that are frequently used by different components of the operating system kernel [5]. The slab allocator implements dynamic memory as a collection of caches exclusively for the operating system kernel. User space processes cannot access it [115].

ner Speicherbereiche (kleiner als eine Seite) an Datenobjekte, die von verschiedenen Komponenten des Betriebssystemkerns häufig verwendet werden [5]. Der Slab Allocator realisiert einen dynamischen Arbeitsspeicher bzw. eine Sammlung von Caches nur für den Betriebssystemkern. Prozesse aus dem Userspace können darauf nicht zugreifen [115].

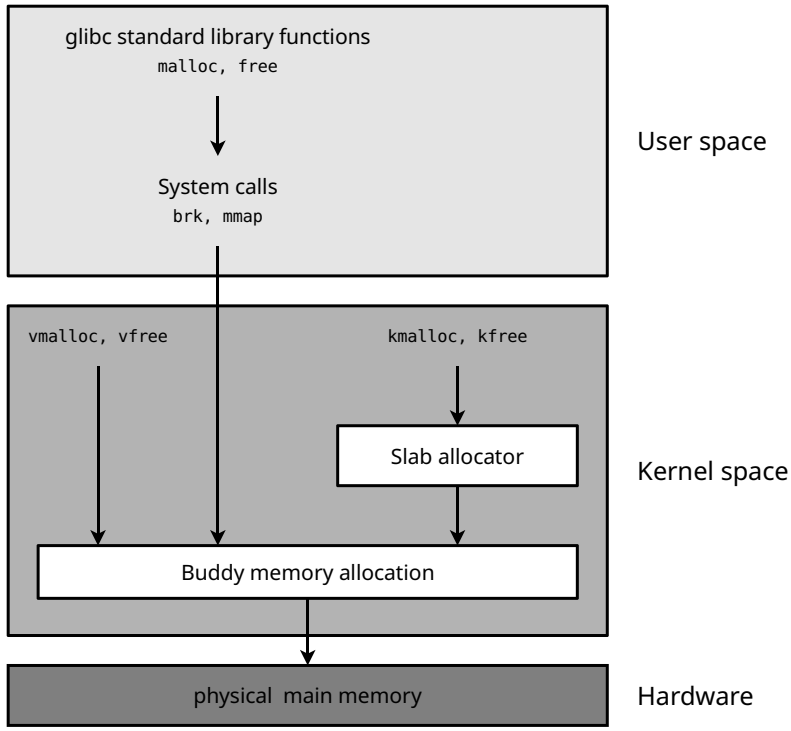

Figure 5.5: In addition to Buddy Memory Management, the Linux Operating System Kernel also implements the Slab Allocator to efficiently manage smaller Memory Areas [53]

The buddy algorithm provides the slab allocator with memory areas (buddies of different sizes), and the slab allocator divides these into smaller regions (so-called *slabs*). The slab allocator divides the slabs into objects.

Storing the objects, which are usually much smaller than a page in size, does not cause significant internal fragmentation due to the slab allocator, unlike the buddy method. Examples of data that the Linux operating system kernel stores as objects are network buffers, process information (`task_struct`), directory en-

Der Buddy-Algorithmus stellt dem Slab Allocator Speicherbereiche (Buddys verschiedener Größe) zur Verfügung und der Slab Allocator unterteilt diese in kleinere Bereiche (sogenannte *Slabs*). Die Slabs unterteilt der Slab Allocator in Objekte.

Die Speicherung der Objekte, die in der Regel viel kleiner als eine Seite groß sind, verursacht durch den Slab Allocator im Gegensatz zum Buddy-Verfahren keine nennenswerte interne Fragmentierung. Beispiele für Daten, die der Linux-Betriebssystemkern als Objekte speichert, sind Netzwerkpuffer, Prozess-

tries (*dentries*), and file system inodes (see Section 6.2). Storing these frequently used data as objects in slab caches allows faster access as the objects are available pre-initialized in the main memory [91].

The size of the individual objects corresponds to the size of the data structure to be managed. The size of a slab is typically a single page. Slabs can also be several pages in size if they are designed for larger objects. Caches are collections of slabs, managing each objects of a particular type. Caches grow dynamically by adding new slabs if the existing slabs run out of free capacity and more objects are requested. [67]

The file `/proc/meminfo` shows the total amount of the memory managed by the slab allocator.

```
$ cat /proc/meminfo | grep Slab
Slab:            3772832 kB
```

The file `/proc/slabinfo` contains the most up-to-date information about the caches managed by the slab allocator. This file contains, among other things, the names of the individual caches, the number of objects within the caches and their size in bytes, and the number of objects and pages per slab. The command line tool `slabtop` can print out the caches and their usage in an ordered format.

informationen (`task_struct`), Verzeichniseinträge (englisch: *Dentries = Directory entries*) und Dateisystem-Inodes (siehe Abschnitt 6.2). Durch die Speicherung dieser häufig verwendeten Daten als Objekte in Slab-Caches ist ein schnellerer Zugriff möglich, da die Objekte vorinitialisiert im Hauptspeicher bereitliegen [91].

Die Größe der einzelnen Objekte entspricht der Größe der Datenstruktur, die verwaltet werden soll. Die Größe eines Slabs entspricht typischerweise eine Seite. Slabs können aber auch mehrere Seiten groß sein, wenn sie für größere Objekte ausgelegt sind. Caches sind Sammlungen von Slabs, die jeweils Objekte eines bestimmten Typs verwalten. Caches wachsen dynamisch, indem neue Slabs hinzugefügt werden, weil die bestehenden Slabs vollständig gefüllt sind und weitere Objekte benötigt werden. [67]

Die Datei `/proc/meminfo` enthält die Gesamtgröße des vom Slab Allocator verwalteten Speichers.

Die aktuellsten Informationen über die vom Slab Allocator verwalteten Caches enthält die Datei `/proc/slabinfo`. Diese Datei enthält u.a. die Namen der einzelnen Caches, die Anzahl der Objekte innerhalb der Caches und deren Größe in Bytes sowie die Anzahl der Objekte und Seiten pro Slab. Das Kommandozeilentool `slabtop` kann die Caches und deren Auslastung sortiert ausgeben.

5.3

Memory Addressing in Practice

Speicheradressierung in der Praxis

As described in the sections 3.5 and 4.1.3, modern computers with 32-bit or 64-bit operating systems can manage high numbers of memory addresses. One of the tasks of an operating system is to organize the requests of the processes to the memory. Two concepts of memory addressing – *real mode* and *protected mode* – are well established in practice and are described in this section.

Wie in den Abschnitten 3.5 und 4.1.3 gezeigt, können moderne Computer mit 32-Bit- oder 64-Bit-Betriebssystemen große Anzahl von Speicheradressen verwalten. Eine der Aufgaben eines Betriebssystems ist, die Zugriffe der Prozesse auf den Speicher zu organisieren. Zwei Konzepte der Speicheradressierung – *Real Mode* und *Protected Mode* – sind in der Praxis etabliert und werden in diesem Abschnitt vorgestellt.

5.3.1

Real Mode

The *real mode*, also called *real address mode*, is one of two possible operating modes of x86-compatible CPUs. This memory addressing method implements direct access to the main memory addresses by the processes (see Figure 5.6). Since the real mode does not protect against accidental or unauthorized access, each process can have access to the entire addressable memory. Therefore, this simple concept is inadequate for multitasking operating systems (see Section 3.4.2). Furthermore, the concept cannot be used to implement a *swap space* that is transparent for the processes.

Real Mode

Der *Real Mode*, der auch *Real Address Mode* heißt, ist eine von zwei möglichen Betriebsarten x86-kompatibler Prozessoren. Dieses Konzept der Speicheradressierung realisiert einen direkten Zugriff auf die Speicheradressen des Hauptspeichers durch die Prozesse (siehe Abbildung 5.6). Da der Real Mode keinen Zugriffsschutz bietet, kann jeder Prozess auf den gesamten adressierbaren Speicher zugreifen. Darum ist dieses einfache Konzept ungeeignet für Betriebssysteme mit Mehrprogrammbetrieb (siehe Abschnitt 3.4.2). Zudem wäre das Konzept ungeeignet, um einen für die Prozesse transparenten *Auslagerungsspeicher* (englisch: *Swap*) zu realisieren.

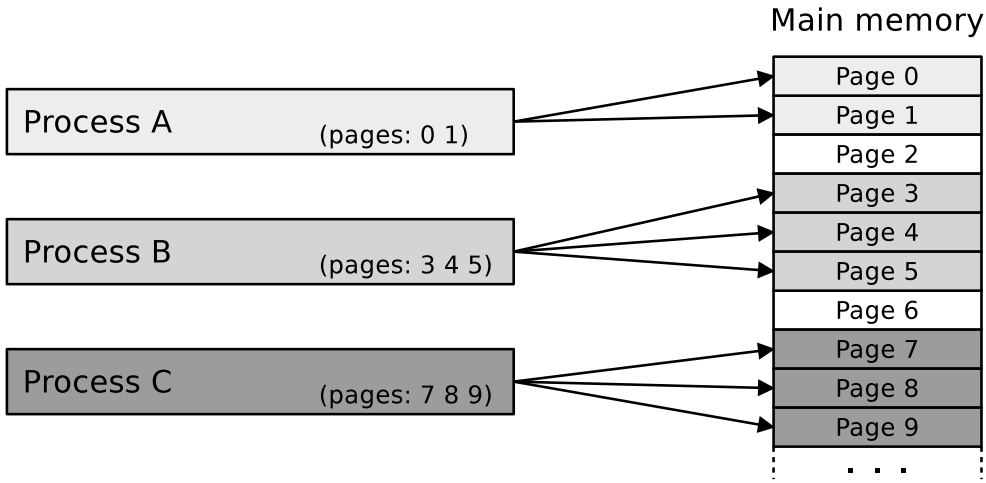

Figure 5.6: Real Mode Memory Management

Another reason why real mode is inappropriate for modern operating systems is that only 1 MB of memory can be addressed. This limitation goes back to the maximum memory expansion of an Intel 8086 processor, whose address bus contains only 20 bus lines (see Table 4.1).

In the operating system MS-DOS and the operating systems compatible with it, such as IBM PC-DOS, DR-DOS, and FreeDOS as well as in early versions of Microsoft Windows, the real mode memory addressing method is used.

Ein weiterer Grund, warum der Real Mode für moderne Betriebssysteme ungeeignet ist, ist die Beschränkung des adressierbaren Hauptspeichers auf maximal 1 MB. Diese Einschränkung geht zurück auf den maximaler Speicherausbau eines Intel 8086 Prozessors, dessen Adressbus nur 20 Busleitungen (siehe Tabelle 4.1) umfasst.

Beim Betriebssystem MS-DOS und den dazu kompatiblen Betriebssystemen wie zum Beispiel IBM PC-DOS, DR-DOS und FreeDOS sowie bei frühen Versionen von Microsoft Windows ist der Real Mode die verwendete Form der Spei-

With these operating systems only the first 640 kB, the so-called *lower memory* of the 1 MB large memory can be used by the operating system itself and the currently running program. The remaining 384 kB, the so-called *upper memory*, contains among other things the BIOS of the graphics adapter, the memory window to the graphics adapter's memory, and the BIOS ROM of the motherboard.

Microsoft Windows 2.0 exclusively runs in real mode. Windows 2.1 and 3.0 both can operate in either real mode or protected mode. Windows 3.1 and later versions only implement protected mode.

The technical term *real mode* was introduced with the Intel 80286 CPU. In real mode, the CPU accesses the main memory the same way as an 8086 CPU does. Every x86-compatible CPU starts at system start or after a reset in real mode. Modern operating systems switch to *protected mode* during the boot process.

Organization and Addressing of Memory in Real Mode

Real mode splits the available memory into segments of equal size. the size of each segment is 64 kB. The addressing of the memory is done via segment and offset. These are two 16 bits long values separated by a colon. Segment and offset are stored in the two 16 bits long registers *segment register* and *offset register* (see Figure 5.7). The segment register stores the segment number, and the offset register points to an address between 0 and 2^{16} (=65,536), relative to the address in the segment register.

In literature, the segment register is sometimes called *base address register* [112, 119] or *segment pointer* [108]. The offset register is sometimes called *index register* [108].

The Intel 8086 has four segment registers (see Figure 5.8), namely:

- `CS` (code segment): This register points to the beginning of the segment that contains the program code (instructions) of the currently running program.

cheradressierung. Bei diesen Betriebssystemen stehen vom 1 MB großen Speicher nur die ersten 640 kB, der sogenannte *untere Speicher*, für das Betriebssystem selbst und das aktuell laufende Programm zur Verfügung. Die restlichen 384 kB, der sogenannte *obere Speicher*, enthalten unter anderem das BIOS der Grafikkarte, das Speicherfenster zum Grafikkartenspeicher und das BIOS ROM des Mainboards.

Microsoft Windows 2.0 läuft ausschließlich im Real Mode. Windows 2.1 und 3.0 können entweder im Real Mode oder im Protected Mode laufen. Windows 3.1 und spätere Versionen laufen ausschließlich im Protected Mode.

Die Bezeichnung *Real Mode* wurde mit dem Intel 80286 Prozessor eingeführt. Im Real Mode greift der Prozessor wie ein 8086 Prozessor auf den Hauptspeicher zu. Jeder x86-kompatible Prozessor startet beim Systemstart bzw. Reset im Real Mode. Moderne Betriebssysteme wechseln während des Starts in den *Protected Mode*.

Organisation und Adressierung des Speichers im Real Mode

Im Real Mode wird der verfügbare Speicher in gleich große Segmente unterteilt. Jedes Segment ist 64 kB groß. Die Adressierung des Speichers geschieht via Segment und Offset. Dabei handelt es sich um zwei 16 Bits lange Werte, die durch einen Doppelpunkt voneinander getrennt sind. Segment und Offset werden in den zwei 16 Bits großen Registern *Segmentregister* und *Offsetregister* (siehe Abbildung 5.7) gespeichert. Das Segmentregister speichert die Nummer des Segments und das Offsetregister zeigt auf eine Adresse zwischen 0 und 2^{16} (=65.536) relativ zur Adresse im Segmentregister.

In der Literatur heißt das Segmentregister an einigen Stellen *Basisregister* [115, 119] oder Segmentzeiger (englisch: *Segment Pointer* [107]). Das Offsetregister heißt in der Literatur an einigen Stellen *Indexregister* [107].

Beim Intel 8086 existieren vier Segmentregister (siehe Abbildung 5.8), nämlich:

- `CS` (Code Segment): Dieses Register zeigt auf den Anfang des Segments, das den Programmcode des aktuell laufenden Programms enthält.

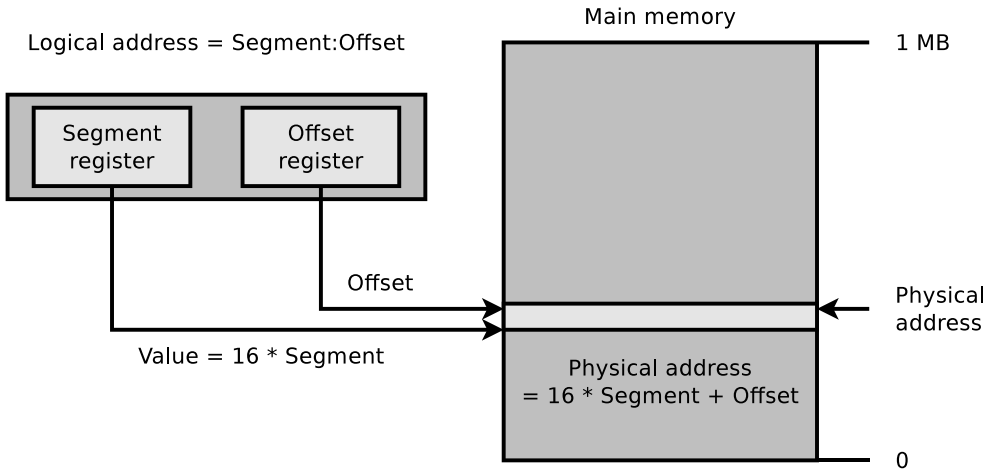

Figure 5.7: Addressing in Real Mode

- `DS` (data segment): This register points to the beginning of the segment that contains the global data of the currently running program.
- `SS` (stack segment): This register points to the beginning of the segment that contains the stack for the local data of the currently running program.
- `ES` (extra segment): This register points to the beginning of the segment that contains further data of the currently running program.

Starting with the Intel 80386, two additional segment registers (`FS` and `GS`) for additional extra segments exist, which is not discussed here any further. The offset registers used for the addressing inside the segments are:

- `IP` (instruction pointer): This register points to the memory address with the next machine instruction to be executed in the code segment (see Section 4.4.1).
- `SI` (source index): This register points to a memory address inside the data segment.

- `DS` (Data Segment): Dieses Register zeigt auf den Anfang des Segments, das die globalen Daten des aktuell laufenden Programms enthält.
- `SS` (Stack Segment): Dieses Register zeigt auf den Anfang des Segments, das den Stack für die lokalen Daten des aktuell laufenden Programms enthält.
- `ES` (Extra Segment): Dieses Register zeigt auf den Anfang des Segments, das weitere Daten des aktuell laufenden Programms enthält.

Ab dem Intel 80386 existieren zwei weitere Segmentregister (`FS` und `GS`) für zusätzliche Extra-Segmente, auf die hier nicht weiter eingegangen wird. Die Offsetregister, die zur Adressierung innerhalb der Segmente dienen, sind:

- `IP` (Instruction Pointer): Dieses Register zeigt auf die Speicheradresse mit dem nächsten auszuführenden Maschinenbefehl im Codesegment (siehe Abschnitt 4.4.1).
- `SI` (Source Index): Dieses Register zeigt auf eine Speicheradresse im Datensegment.

- `SP` (stack pointer): This register points to the memory address that contains the end of the stack in the stack segment (see Section 4.4.1).
- `DI` (destination index): This register points to a memory address in the extra segment.

- `SP` (Stack Pointer): Dieses Register zeigt auf die Speicheradresse, die das Ende des Stacks im Stacksegment enthält (siehe Abschnitt 4.4.1).
- `DI` (Destination Index): Dieses Register zeigt auf eine Speicheradresse im Extrasegment.

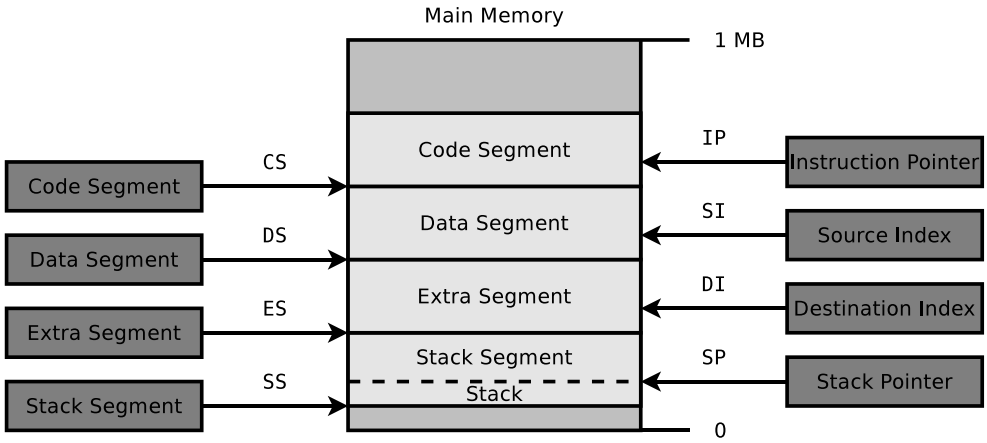

Figure 5.8: Addressing in Real Mode – Segment Registers since the Intel 8086

Figure 5.7 shows that the segments implement a simple form of *memory protection*.

Although the segment and offset registers are 16 bits in size each, the memory addresses in real mode have a length of only 20 bits, regardless of the CPU generation used. The reason for this is the size of the address bus of the Intel 8086, which has only 20 lines.

Abbildung 5.7 zeigt anschaulich, dass die Segmentbereiche einen einfachen *Speicherschutz* realisieren.

Auch wenn die Segment- und Offsetregister jeweils 16 Bits groß sind, haben die Speicheradressen im Real Mode, unabhängig von der verwendeten Prozessorgeneration nur eine Länge von 20 Bits. Der Grund sind die 20 Busleitungen im Adressbus des Intel 8086.

5.3.2

Protected Mode and Virtual Memory

The analysis of the memory management concepts in Section 5.1 and the real mode helps to understand the demands of modern operating systems on memory management. These are:

- *Relocation*: If a process is replaced from the main memory, it is unknown at which

Protected Mode und virtueller Speicher

Die Auseinandersetzung mit den Konzepten zur Speicherverwaltung in Abschnitt 5.1 sowie mit dem Real Mode hilft die Anforderungen moderner Betriebssysteme an die Speicherverwaltung zu verstehen. Diese sind:

- *Relokation*: Wird ein Prozess aus dem Hauptspeicher verdrängt, ist nicht be-

address it will later be reloaded into the main memory. For this reason, processes preferably must not refer to physical memory addresses (as in real mode).

kannt, an welcher Adresse er später wieder in den Hauptspeicher geladen wird. Aus diesem Grund sollen Prozesse nach Möglichkeit keine Referenzen auf physische Speicheradressen (wie beim Real Mode) enthalten.

- *Protection*: Memory areas must be protected against accidental or unauthorized access by other processes. As a consequence, the operating system must implement memory protection by verifying all requests for memory addresses by the processes.

- *Schutz*: Speicherbereiche müssen vor unbeabsichtigtem oder unzulässigem Zugriff durch andere Prozesse geschützt sein. Als Konsequenz daraus muss das Betriebssystem einen Speicherschutz realisieren, indem es alle Zugriffe der Prozesse auf Speicheradressen überprüft.

- *Shared use*: Despite memory protection, it must be possible for processes to collaborate (i.e., access shared data) via *shared memory segments* (see Section 9.3.1).

- *Gemeinsame Nutzung*: Trotz Speicherschutz muss es möglich sein, dass Prozesse über gemeinsame Speicherbereiche, sogenannte *Shared Memory*-Segmente (siehe Abschnitt 9.3.1), auf gemeinsame Daten zugreifen.

- *Increased capacity*: Until the mid-1980s, 1 MB of memory may have been sufficient. However, modern operating systems need to be able to address more memory. Especially with multitasking (see Section 3.4.2), which all modern operating systems offer, it is also useful when operating systems can address more main memory than physically exists. Modern operating systems implement a swap memory, in which they can swap out processes that are currently not assigned to a CPU or a CPU core when the main memory is fully occupied.

- *Größere Speicherkapazität*: Bis Mitte der 1980er Jahre mögen 1 MB Speicher ausreichend gewesen sein. Moderne Betriebssysteme müssen aber in der Lage sein, mehr Speicher zu adressieren. Speziell beim Mehrprogrammbetrieb (siehe Abschnitt 3.4.2), den alle modernen Betriebssysteme bieten, ist es auch sinnvoll, wenn Betriebssysteme mehr Hauptspeicher adressieren können, als physisch existiert. Moderne Betriebssysteme realisieren einen Auslagerungsspeicher, der (speziell in der deutschsprachigen Literatur) auch Hintergrundspeicher [39, 50], oder einfach *Swap* heißt. In diesen können die Betriebssysteme bei vollem Hauptspeicher diejenigen Prozesse auslagern, die gegenwärtig keinen Zugriff auf einen Prozessor bzw. einen Prozessorkern haben.

Protected mode meets all requirements mentioned in this section. It is the second possible operating mode for x86-compatible CPUs and was introduced with the Intel 80286 in the early 1980s.

Alle in diesem Abschnitt genannten Anforderungen realisiert der *Schutzmodus*, der auch in der deutschsprachigen Literatur häufiger unter der englischen Bezeichnung *Protected Mode* beschrieben wird. Bei diesem handelt es sich um die zweite mögliche Betriebsart x86-kompatibler

In protected mode, the processes do not use physical main memory addresses. Instead, they use *virtual memory*, which is independent of the memory technology used and the given expansion capabilities. It consists of logical memory addresses that are numbered in ascending order from address 0 on. Each process runs in its copy of the physical address space, which is isolated from other processes, and each process can only access its virtual memory. A simplified illustration of this concept is shown in Figure 5.9. The mapping of the virtual memory to physical memory is called *mapping*.

Im Protected Mode verwenden die Prozesse keine physischen Hauptspeicheradressen, sondern einen *virtuellen Speicher*, der unabhängig von der verwendeten Speichertechnologie und den gegebenen Ausbaumöglichkeiten ist. Er besteht aus logischen Speicheradressen, die von der Adresse 0 aufwärts durchnummeriert sind. Jeder Prozess läuft ausschließlich in seiner eigenen, von anderen Prozessen abgeschotteten Kopie des physischen Adressraums und jeder Prozess darf nur auf seinen eigenen virtuellen Speicher zugreifen. Eine einfache Darstellung dieses Konzepts zeigt Abbildung 5.9. Die Abbildung des virtuellen Speichers auf den physischen Speicher heißt *Mapping*.

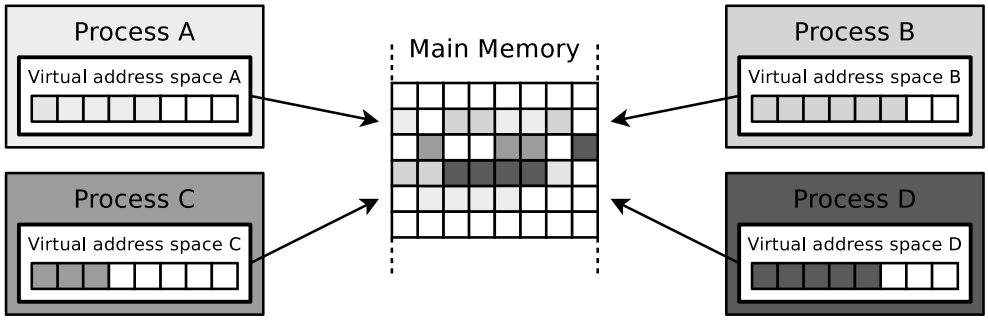

Figure 5.9: Mapping of the Virtual Memory to the Physical Memory in Protected Mode

Figure 5.9 shows another advantage of the virtual memory concept, namely that it utilizes the available main memory more efficiently because the processes do not need to be stored in the main memory in a consecutive area. By the way, a continuous placement of the processes in the main memory would bring no improvement because the main memory is electronic memory with random access (see Section 4.3). Therefore the access to each memory address takes the same amount of time.

An additional advantage of the virtual memory concept is that more memory can be addressed and used as it is physically present in the system. *Swapping* is done transparently for the processes (see Figure 5.10).

In protected mode, the CPU supports two different implementation variants of the virtual

Abbildung 5.9 zeigt einen weiteren Vorteil des virtuellem Speichers, nämlich die bessere Ausnutzung des existierenden Hauptspeichers, weil die Prozesse nicht am Stück im Hauptspeicher liegen müssen. Eine zusammenhängende Anordnung der Prozesse im Hauptspeicher würde auch keinen Vorteil bringen, da der Hauptspeicher ein elektronischer Speicher mit wahlfreiem Zugriff (siehe Abschnitt 4.3) ist und der Zugriff auf jede Speicheradresse gleich schnell ist.

Ein weiterer Vorteil des virtuellen Speichers ist, dass mehr Speicher angesprochen und verwendet werden kann, als physisch im System existiert. Das *Auslagern* (englisch: *Swapping*) geschieht für die Prozesse transparent (siehe Abbildung 5.10).

Im Protected Mode unterstützt die CPU zwei unterschiedliche Methoden zur Realisierung des

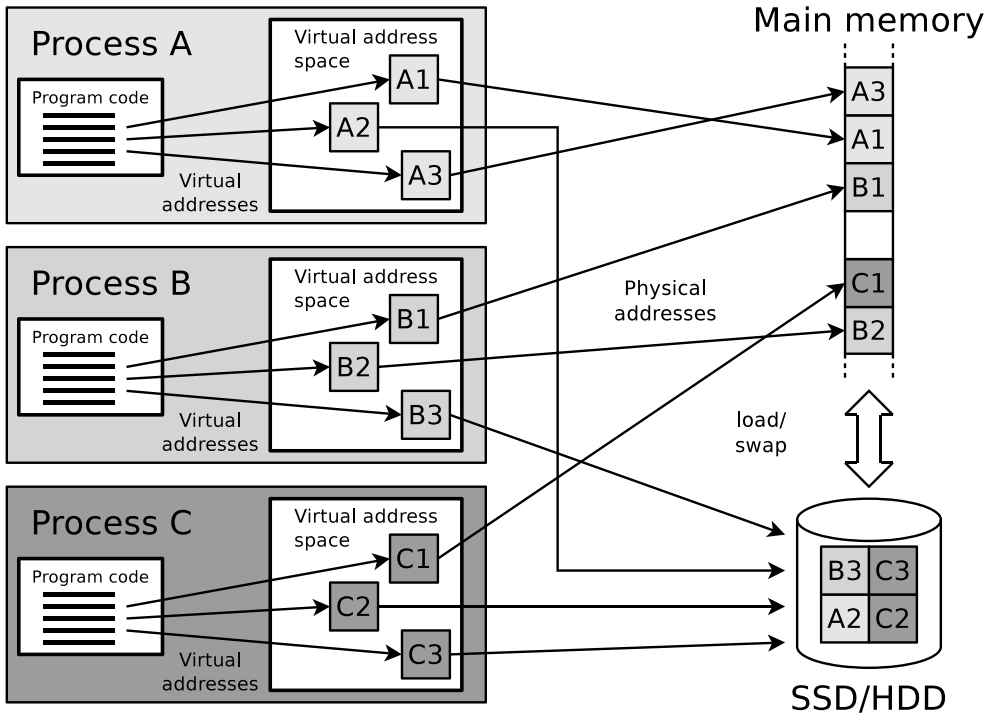

Figure 5.10: Virtual Memory supports *Swapping* that is performed transparently for the Processes [108, 119]

memory concept. These are *segmentation* and *paging*.

virtuellen Speichers. Dabei handelt es sich um die *Segmentierung* und das *Paging*.

5.3.3

Page-based Memory Management (Paging)

Page-based memory management, so-called *paging*, maps the virtual pages of the processes to physical pages in the main memory. These pages are the same size. The page size of most hardware architectures is 4 kB. Exceptions are Alpha and UltraSPARC CPUs, which have a page size of 8 kB [84]. CPUs featuring the ARM architecture support different page sizes (usually 4 kB, 64 kB and 1 MB). Most operating systems (e.g., Linux and Windows) use 4 kB page size by default on such CPUs. Apple's operating systems (iOS and Mac OS X), which run on Apple Silicon CPUs implementing an ARM64 architecture, use 16 kB page size by default. The page

Seitenorientierter Speicher (Paging)

Beim *seitenorientierten Speicher*, dem sogenannten *Paging*, werden die virtuellen Seiten der Prozesse auf physische Seiten im Hauptspeicher abgebildet. Die Seiten heißen in der deutschsprachigen Literatur auch *Kacheln* [123] und haben die gleiche Größe. Die Seitengröße ist bei den allermeisten Hardwarearchitekturen 4 kB. Ausnahmen sind Alpha-Prozessoren und UltraSPARC-Prozessoren. Dort ist die Seitengröße 8 kB [84]. Prozessoren mit einer ARM-Architektur unterstützen verschiedene Seitengrößen (meist 4 kB, 64 kB und 1 MB). Die meisten Betriebssysteme (z.B. Linux und Windows) verwenden auf solchen Prozessoren standardmä-

size is also variable for Intel Itanium (IA-64). Possible sizes are 4, 8, 16, or 64 kB.

The operating system maintains a page table for each process. It indicates where the individual pages of the process are located. The virtual memory addresses consist of two parts. The most significant part is the page number, and the less significant part is the offset, i.e., an address within a page. The length of the virtual addresses depends on the number of bus lines in the address bus (see Section 4.1.3) and is therefore 16, 32, or 64 bits.

Since the processes do not need to be stored in main memory in a consecutive area, external fragmentation is irrelevant (see Figure 5.11). Internal fragmentation can only occur on the last page of each process.

ßig 4 kB Seitengröße. Die Betriebssysteme von Apple (iOS und Mac OS X), die auf den Prozessoren von Apple Silicon mit einer ARM64-Architektur laufen, verwenden standardmäßig 16 kB Seitengröße. Auch beim Intel Itanium (IA-64) ist die Seitengröße variabel. Mögliche Größen sind 4, 8, 16 oder 64 kB.

Das Betriebssystem verwaltet für jeden Prozess eine Seitentabelle (englisch: *Page Table*). In dieser steht, wo sich die einzelnen Seiten des Prozesses befinden. Die virtuellen Speicheradressen bestehen aus zwei Teilen. Der werthöhere Teil enthält die Seitennummer und der wertniedrigere Teil den Offset, also eine Adresse innerhalb einer Seite. Die Länge der virtuellen Adressen hängt von der Anzahl der Busleitungen im Adressbus (siehe Abschnitt 4.1.3) ab und ist darum 16, 32 oder 64 Bits.

Da die Prozesse nicht am Stück im Hauptspeicher liegen müssen, spielt externe Fragmentierung keine Rolle (siehe Abbildung 5.11). Interne Fragmentierung kann nur in der letzten Seite jedes Prozesses auftreten.

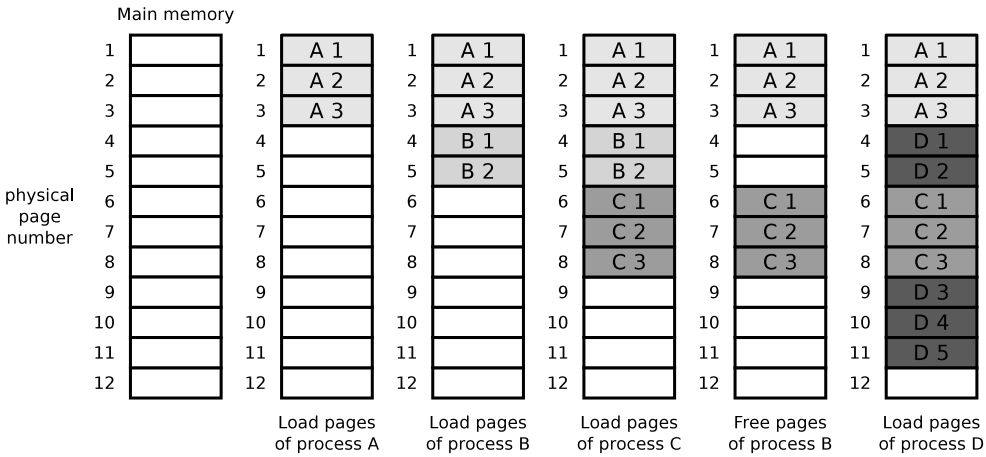

Figure 5.11: External Fragmentation is irrelevant in Paging because the Processes do not need to be stored in Main Memory in a consecutive Area [108]

Virtual memory addresses are translated by the CPU with the Memory Management Unit (MMU) and the page table into physical addresses (see Figure 5.12). Each time a virtual address is requested, the operating system determines whether the assigned physical address is in main memory or in swap memory (usually

Virtuelle Speicheradressen übersetzt der Hauptprozessor mit der Memory Management Unit (MMU) und der Seitentabelle in physische Adressen (siehe Abbildung 5.12). Das Betriebssystem prüft bei jedem Zugriff auf eine virtuelle Adresse, ob sich deren zugeordnete physische Adresse im Hauptspeicher oder im

on an SSD or hard disk drive). If the data is located in swap memory, the operating system has to copy the data into main memory. If the main memory has no more free capacity, the operating system must relocate further data from the main memory into swap memory [112, 119].

All modern CPUs include a Memory Management Unit. However, this was not the case for computer systems in the 1980s. For some computer systems, it was possible to equip them with external MMUs and thus implement virtual memory. Examples are the Motorola CPUs 68010 and 68020 for which the manufacturer offered the external MMUs 68451 (used, for example, in Unix workstations of the Sun-2 generation by Sun Microsystems) and 68851 (used, for example, in the Apple Macintosh II). Another example is the external MMU 8722 from MOS Technology for the Commodore C128 home computer.

Auslagerungsspeicher (meist auf einer SSD oder Festplatte) befindet. Befinden sich die Daten im Auslagerungsspeicher, muss das Betriebssystem die Daten in den Hauptspeicher einlesen. Ist der Hauptspeicher voll, muss das Betriebssystem andere Daten aus dem Hauptspeicher in den Auslagerungsspeicher verdrängen [115, 119].

Alle modernen Hauptprozessoren enthalten eine Memory Management Unit. Bei den gängigen Computersystemen in den 1980er Jahren war das aber nicht bei allen Prozessorfamilien der Fall. Einige Computersysteme konnten aber mit externen MMUs nachgerüstet werden und so ebenfalls einen virtuellen Speicher realisieren. Beispiele sind die Prozessoren 68010 und 68020 für die der Hersteller Motorola die externen MMUs 68451 (wurde u.a. in den Unix-Workstations der Generation Sun-2 von Sun Microsystems eingesetzt) und 68851 (wurde u.a. im Apple Macintosh II eingesetzt) anbot. Ein weiteres Beispiel ist die externe MMU 8722 von MOS Technology für den Commodore C128 Heimcomputer.

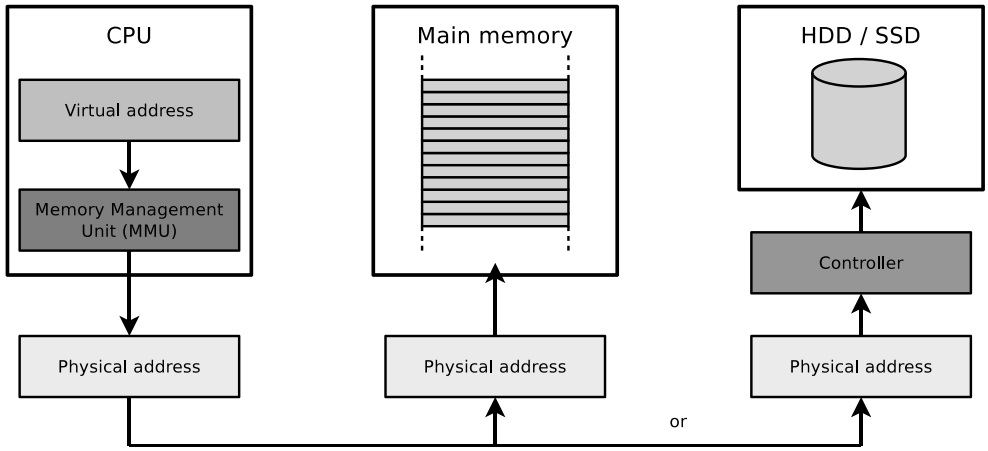

Figure 5.12: Translation of virtual Memory Addresses into physical Addresses by the Memory Management Unit [119]

The page size specified by the hardware architecture has an impact on the amount of internal fragmentation and the length of the page tables. Small pages cause less internal fragmentation but more extended page tables and, therefore, more overhead. Bigger pages cause shorter page tables, but more internal fragmentation in the last page of every process.

Die durch die Hardwarearchitektur vorgegebene Seitengröße beeinflusst den Grad der internen Fragmentierung und die Länge der Seitentabellen. Kurze Seiten verursachen weniger interne Fragmentierung, aber dafür längere Seitentabellen und damit mehr Verwaltungsaufwand. Längere Seiten verursachen kürzere Seitentabel-

len, aber dafür mehr interne Fragmentierung in der letzten Seite jedes Prozesses.

Page Table Size

The following formula specifies the maximum page table size.

Größe der Seitentabellen

Die maximale Größe der Seitentabellen gibt die folgende Formel an.

$$\text{maximum page table size} = \frac{\text{virtual address space}}{\text{page size}} \times \text{size of each page table entry}$$

Therefore, the maximum page table size on a 32-bit operating system on a hardware architecture with 4 kB page size is 4 kB:

Dementsprechend ist die maximale Größe der Seitentabellen bei 32-Bit-Betriebssystemen auf einer Hardwarearchitektur mit 4 kB Seitengröße:

$$\frac{4\,\text{GB}}{4\,\text{kB}} \times 4\,\text{bytes} = \frac{2^{32}\,\text{bytes}}{2^{12}\,\text{bytes}} \times 2^2\,\text{bytes} = 2^{22}\,\text{bytes} = 4\,\text{MB}$$

In 64-bit operating systems, the page tables of the individual processes can be considerably larger. However, since most processes do not require several gigabytes of memory, the management overhead for the page tables on modern computers is low.

Bei 64-Bit-Betriebssystemen können die Seitentabellen der einzelnen Prozesse deutlich größer sein. Da aber die meisten im Alltag laufenden Prozesse nicht mehrere Gigabyte Speicher benötigen, fällt der Overhead durch die Verwaltung der Seitentabellen auf modernen Computern gering aus.

Page Table Structure and Address Translation

Each page table record (see Figure 5.13) contains a *present-bit* that indicates whether the page is stored inside the main memory and a *dirty-bit* that specifies whether the page has been modified. The dirty bit is also called *modified-bit* [108, 112, 119] in literature.

There is also a *reference-bit* that indicates whether the page has been referenced. Read and write operations are taken into account. Such a bit is fundamental to the functioning of some page replacement strategies, such as clock or second chance (see Section 5.4) [35]. In addition to the described bits, each page table record contains further control bits. These specify, among other things, whether processes in user mode have read-only or write access to the page (*read/write-bit*), whether processes in user mode are allowed to access the page (*user/supervisor-bit*), whether modifications are written immediately (*write-through*) or when they are removed (*write-back*) by default (*write-through-bit*) and whether the page is allowed to

Struktur der Seitentabellen und Adressumwandlung

Jeder Eintrag in einer Seitentabelle (siehe Abbildung 5.13) enthält ein *Present-Bit*, das angibt, ob die Seite im Hauptspeicher liegt und ein *Dirty-Bit*, das anzeigt, ob die Seite verändert wurde. Das Dirty-Bit heißt in der Literatur auch *Modified-Bit* [107, 115, 119].

Zudem gibt es noch ein *Reference-Bit*, das angibt, ob es einen Zugriff auf die Seite gab. Dabei werden auch lesende Zugriffe berücksichtigt. Ein solches Bit ist für das Funktionieren einiger möglicher Seitenersetzungsstrategien wie zum Beispiel Clock bzw. Second Chance elementar (siehe Abschnitt 5.4) [35]. Außer den beschrieben Bits enthält jeder Eintrag noch weitere Steuerbits. Diese definieren unter anderem, ob Prozesse im Benutzermodus nur lesend oder auch schreibend auf die Seite zugreifen dürfen (*Read/Write-Bit*), ob Prozesse im Benutzermodus auf die Seite zugreifen dürfen (*User/Supervisor-Bit*), ob Änderungen sofort (*Write-Through*) oder erst beim verdrängen (*Write-Back*) durchgeschrieben werden (*Write-Through-Bit*) und ob die Seite in den

be loaded into the cache or not (*cache-disable-bit*). Finally, each record contains the physical page address. It is concatenated with the offset of the virtual address.

Cache geladen werden darf oder nicht (*Cache-Disable-Bit*). Schlussendlich enthält jeder Eintrag noch die physische Seitenadresse. Diese wird mit dem Offset der virtuellen Adresse verknüpft.

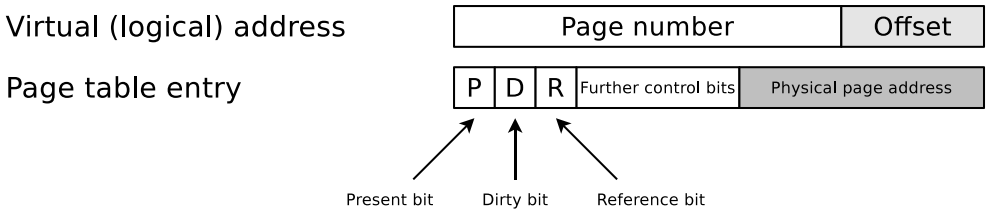

Figure 5.13: Page Table Structure

The address translation mechanism for single-level paging is shown in Figure 5.14. The figure also shows that two registers in the CPU enable the Memory Management Unit to access the page table of the running process. These are the *Page-Table Base Register* (PTBR) that contains the physical address where the page table starts in main memory, and the *Page-Table Length Register* (PTLR) that indicates the length of the page table of the running process.

Single-level paging is sufficient for 16-bit architectures, but already on 32-bit architectures, the operating systems implement multi-level paging. This results from the size of the page tables in multitasking operation mode [108, 119].

At the beginning of this section, a calculation demonstrated that even in 32-bit operating systems with 4 kB page size, the page table of each process might be up to 4 MB in size. In modern 64-bit operating systems, page tables can be much larger. For using less main memory, modern operating systems implement multi-level paging [123]. Thereby, the page table is split into several smaller tables. When calculating a physical address, the operating system checks the pages level by level. If necessary, individual pages of the different levels can be relocated to the swap memory to free up storage capacity in main memory.

Modern hardware architectures allow operating systems to split the page table in up to five smaller tables. Table 5.1 contains

Das Prinzip der Adressumwandlung beim einstufigen Paging zeigt Abbildung 5.14. Die Abbildung zeigt auch, dass es zwei Register im Hauptprozessor der Memory Management Unit ermöglichen, auf die Seitentabelle des laufenden Prozesses zuzugreifen. Dabei handelt es sich um das *Page-Table Base Register* (PTBR), das die physische Adresse enthält, wo die Seitentabelle im Hauptspeicher anfängt und das *Page-Table Length Register* (PTLR), das die Länge der Seitentabelle des laufenden Prozesses enthält.

Einstufiges Paging ist auf 16-Bit-Architekturen ausreichend, aber bereits auf 32-Bit-Architekturen realisieren die Betriebssysteme ein mehrstufiges Paging. Der Grund dafür ist die Größe der Seitentabellen beim Multitasking [107, 119].

Zu Beginn dieses Abschnitts wurde berechnet, dass schon bei 32-Bit-Betriebssystemen mit 4 kB Seitengröße die Seitentabelle jedes Prozesses 4 MB groß sein kann. Bei modernen 64-Bit-Betriebssystemen können die Seitentabellen noch wesentlich größer sein. Um den Hauptspeicher zu schonen, realisieren moderne Betriebssysteme ein mehrstufiges Paging [123]. Die Seitentabelle wird dabei in mehrere kleinere Tabellen aufgespalten. Bei der Berechnung einer physischen Adresse durchläuft das Betriebssystem die Teilseiten Stufe für Stufe. Einzelne Teilseiten können bei Bedarf auf den Auslagerungsspeicher verdrängt werden, um Platz im Hauptspeicher zu schaffen.

Moderne Hardwarearchitekturen ermöglichen Betriebssystemen die Unterteilung der Seitentabelle in bis zu fünf kleinere Tabellen. Einige

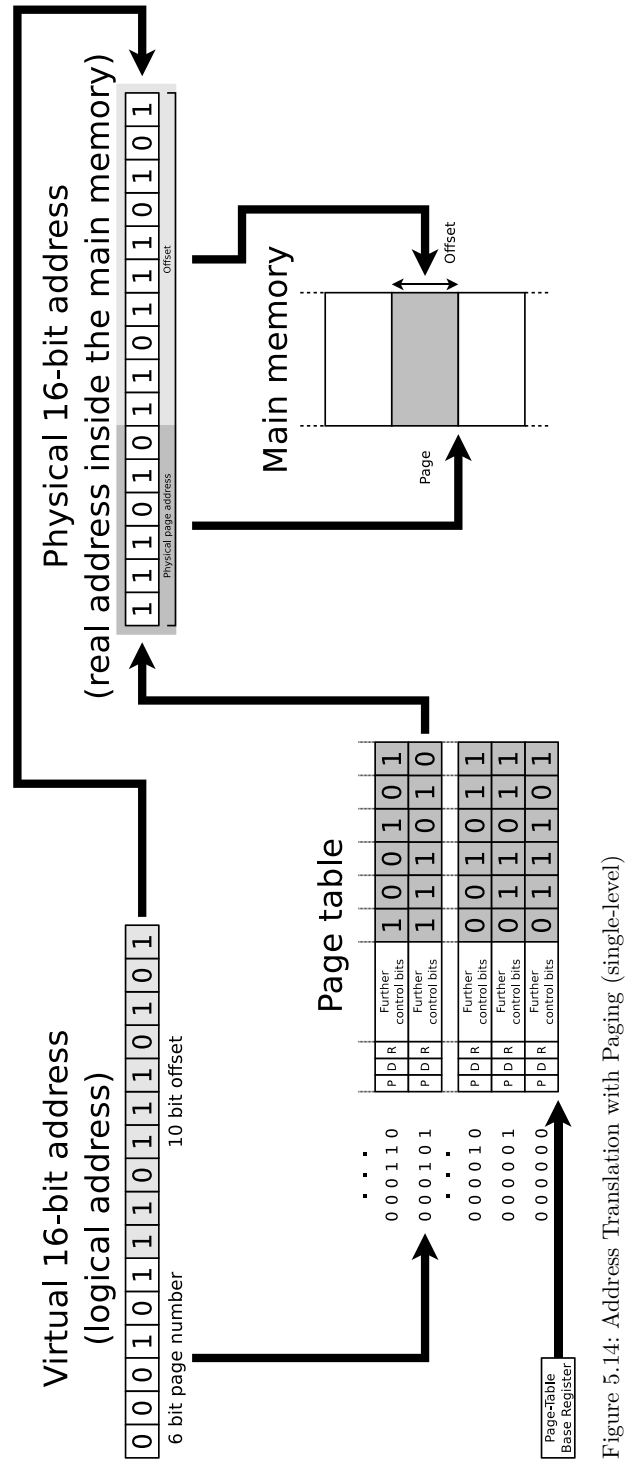

Figure 5.14: Address Translation with Paging (single-level)

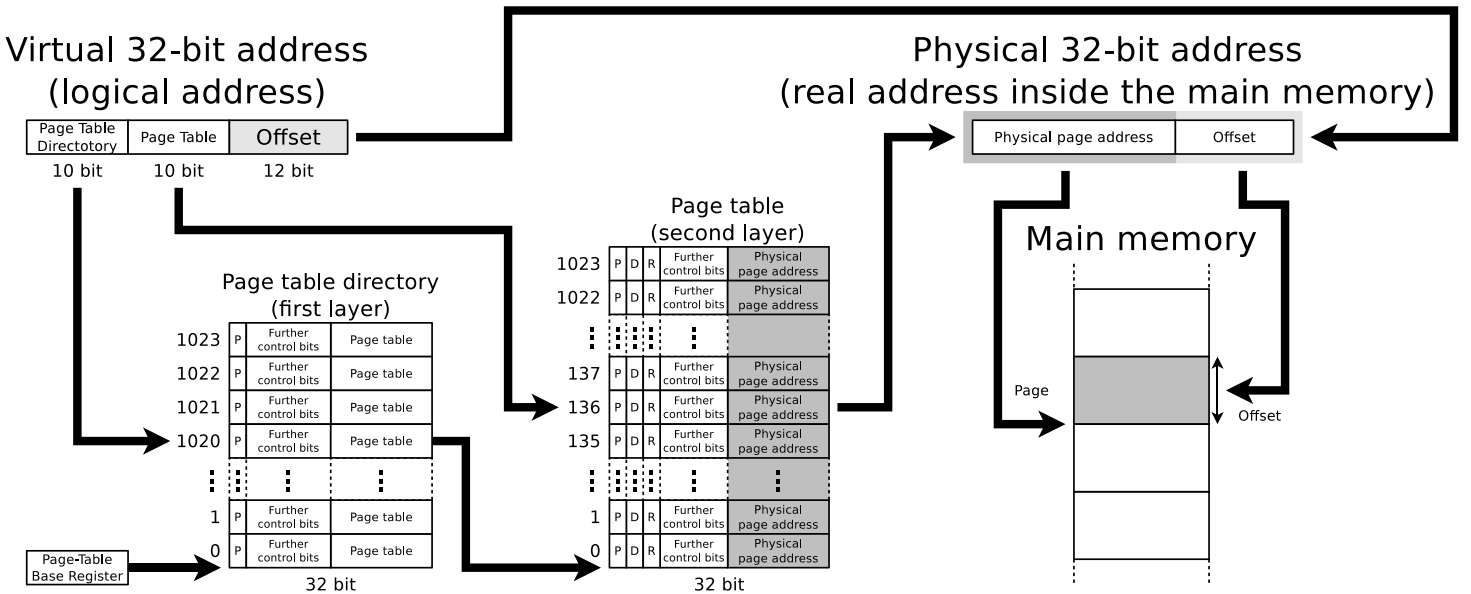

Figure 5.15: Address Translation with Paging (two-levels)

some selected architectures and their paging methods.

ausgewählte Architekturen und deren Art des Paging enthält Tabelle 5.1.

Table 5.1: Selected Architectures and their Paging Methods

Architecture	Page table structure	Virtual address length	Partitioning[a]
IA32 (x86-32)	2 levels	32 bits	10+10+12
ARMv7-A[d]	2 levels	32 bits	12+8+12
IA32 with PAE[b]	3 levels	32 bits	2+9+9+12
Sparc (32-Bit)	3 levels	32 bits	8+6+6+12
ARMv8-A ($\leq$ 512 GB)[e]	3 levels	39 Bits	9+9+9+12
PPC64	3 levels	41 bits	10+10+9+12
AMD64 (x86-64)	4 levels	48 bits	9+9+9+9+12
ARMv8-A ($\leq$ 256 TB)[e]	4 levels	48 bits	9+9+9+9+12
5-level paging[c] (x86-64)	5 levels	57 bits	9+9+9+9+9+12

[a] The last number indicates the offset length in bits. The other numbers indicate the lengths of the page tables.
[b] PAE = Physical Address Extension. With this paging extension of the Pentium Pro CPU, more than 4 GB of RAM can be addressed by the operating system. However, the memory usage per process is still limited to 4 GB.
[c] Intel Ice Lake Xeon Scalable CPUs [55] and AMD EPYC 8004/9004 series CPUs [2] implement 5-level paging.
[d] e.g. ARM Cortex A5, A7, A8, A9, A12, A15, A17 [3].
[e] e.g. ARM Cortex A35, A53, A57, A72, A73 [4, 70, 71, 121].

Address Translation Acceleration through the Translation Buffer

Most processor architectures implement a *Translation Lookaside Buffer* (TLB) in the MMU to accelerate the address translation. The TLB can store several hundred or more address translations depending on the memory capacity. It is a cache for page table records and stores frequently used address translations. Accesses to the TLB are faster than accesses to the main memory [69, 107, 115] The TLB allows address translations to be performed in one or very few clock cycles instead of the 50 to 200 clock cycles (in the sum of all cache levels plus main memory) that are required for a main memory request, depending on the CPU architecture, memory latency, number and capacity of the individual cache levels (see Table 4.4).

Similar to the cache memory (see Section 4.4.2), the translation buffer of modern CPU architectures usually has two levels with different speeds and memory capacities.

Beschleunigung der Adressumwandlung durch den Übersetzungspuffer

Die meisten Prozessorarchitekturen implementieren in der MMU einen *Übersetzungspuffer* (englisch: *Translation Lookaside Buffer* – TLB), um die Adressumwandlung zu beschleunigen. Der TLB kann je nach Speicherkapazität einige hundert oder mehr Adressumwandlungen speichern. Er ist ein Cache für Seitentabelleneinträge und speichert häufig verwendete Adressumwandlungen. Zugriffe auf den TLB sind schneller als Zugriffe auf den Hauptspeicher. [69, 107, 115] Durch den TLB können Adressumwandlungen in einem oder sehr wenigen Taktzyklen erfolgen, anstatt (in der Summe aller Cache-Ebenen plus Hauptspeicher) ca. 50 bis 200 Taktzyklen (siehe Tabelle 4.4), die je nach Prozessorarchitektur, Speicherlatenz, Anzahl und Kapazität der Cache-Ebenen für einen Hauptspeicherzugriff nötig sind.

Ähnlich wie beim Pufferspeicher (siehe Abschnitt 4.4.2) besteht der Übersetzungspuffer aktueller Prozessorarchitekturen üblicherweise aus zwei Ebenen. Diese Ebenen weisen unterschiedliche Geschwindigkeiten und Speicherkapazitäten auf.

The first-level translation buffer (*L1-TLB*) often exists twice, once for instructions (*Instruction TLB*) and once for data (*Data TLB*). Typical capacities for modern CPUs are 32 to 128 entries.

The capacity of the second-level translation buffer (*L2-TLB*) typically ranges from 512 to 2048 entries in modern CPUs.

Issue Handling in Paging

The present-bit in each page table record specifies whether the page is in the main memory or not. If a process tries to request a page that is not located in the physical main memory, a *page fault exception* occurs, which is often called *page fault* [108], *page error* [17, 112] or *hard (page) fault* [113] in literature, and less often *page fault interrupt* [39].

The handling of a page fault is illustrated in Figure 5.16. In step (1), a process tries to request a page in its virtual memory. The *present-bit* in each page table record indicates whether the page is in main memory or not. Since the page is not in main memory, a page fault exception occurs. In step (2), a software interrupt (exception) is triggered to switch from user mode to kernel mode (see Section 3.8). The operating system interrupts the execution of the process in user mode and forces the execution of an *exception handler* in kernel mode. For handling a page fault, the operating system first allocates the page in step (3) using the controller and the associated device driver on the swap memory (usually on an SSD or hard disk drive). In step (4), the operating system copies the page from the swap space into an available main memory page, and it updates the page table in step (5). Finally, the operating system returns control over the CPU to the program in step (6). Afterward, the program again executes the instruction that caused the page fault [101].

In contrast to the hard page fault principle already described, does a *soft page fault* or *soft fault* not require the page to be reloaded into main memory. It is already in main memory

Der Übersetzungspuffer erster Ebene (*L1-TLB*) existiert häufig doppelt, einmal für Anweisungen (*Instruction TLB*) und einmal für Daten (*Data TLB*). Typische Kapazitäten aktueller Prozessoren sind 32 bis 128 Einträge.

Die Kapazität des Übersetzungspuffers zweiter Ebene (*L2-TLB*) umfasst bei aktuellen Prozessoren typischerweise 512 bis 2048 Einträge.

Behandlung von Problemen beim Paging

Das Present-Bit in jedem Eintrag der Seitentabelle gibt an, ob sich die Seite im Hauptspeicher befindet oder nicht. Versucht ein Prozess auf eine Seite zuzugreifen, die nicht im physischen Hauptspeicher liegt, kommt es zu einer *Page Fault Ausnahme* (englisch: Page Fault Exception [107]), die in der deutschsprachigen Literatur häufig *Seitenfehler* [17, 115] oder *harter Seitenfehler* [115] und seltener *Page Fault Interrupt* [39] heißt.

Den Ablauf eines Seitenfehlers und dessen Behandlung zeigt Abbildung 5.16. In Schritt (1) versucht ein Prozess auf eine Seite in seinem virtuellen Speicher zuzugreifen. Das *Present-Bit* in jedem Eintrag der Seitentabelle gibt an, ob die Seite im Hauptspeicher ist oder nicht. Da die Seite nicht im Hauptspeicher ist, kommt es zum Seitenfehler. In Schritt (2) wird ein Software-Interrupt (Exception) ausgelöst, um vom Benutzermodus in den Kernelmodus (siehe Abschnitt 3.8) zu wechseln. Das Betriebssystem unterbricht die Programmausführung des Prozesses im Benutzermodus und erzwingt das Ausführen eines *Exception-Handlers* im Kernelmodus. Zur Behandlung eines Seitenfehlers lokalisiert das Betriebssystem in Schritt (3) die Seite zuerst mit Hilfe des Controllers und des entsprechenden Gerätetreibers auf dem Auslagerungsspeicher (meist auf einer SSD oder Festplatte). In Schritt (4) kopiert das Betriebssystem die Seite vom Auslagerungsspeicher in eine freie Hauptspeicherseite und aktualisiert in Schritt (5) die Seitentabelle. Abschließend gibt das Betriebssystem in Schritt (6) die Kontrolle über den Prozessor wieder an das Programm zurück. Dieses führt die Anweisung, die zum Seitenfehler führte, erneut aus [101].

Im Gegensatz zum bereits beschriebenen *harten Seitenfehler* muss bei einem *weichen Seitenfehler* die Seite nicht in den Hauptspeicher nachgeladen werden. Sie befindet sich bereits im

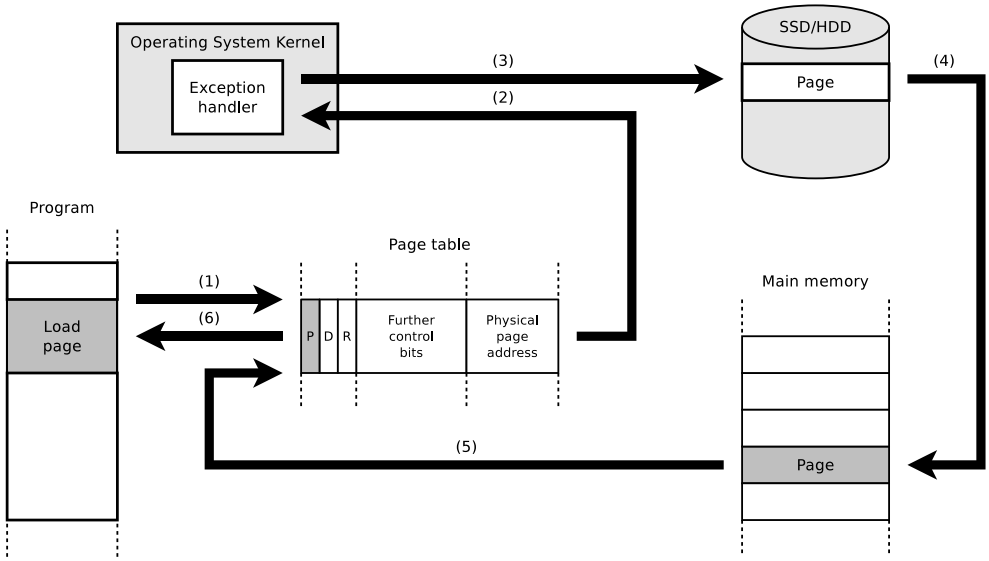

Figure 5.16: Handling of a Page Fault by the Operating System

but not assigned to the requesting process. One example of a situation that causes a soft page fault to occur is when a shared memory page that is mapped into the virtual memory of another process is to be modified. In this case, the page cannot be shared anymore. The operating system needs to copy the page and adjust the page table entry. Soft page fault handling has little impact on the performance of a process compared to hard page fault handling [113].

Another exception that the operating system must be able to handle when paging is used is a *segmentation fault* or *segmentation violation*, which is also called *access violation exception* or *general protection fault exception*. Regardless of its name, this exception is caused by a paging problem that has nothing to do with segmentation (see Section 5.3.4). Such an exception is triggered when a process tries to request a virtual memory address that it is not allowed to access. In some legacy Windows operating systems, segmentation faults often caused system crashes and resulted in a *blue screen*. In Linux, the signal `SIGSEGV` is returned as a result [47].

Hauptspeicher, ist allerdings nicht dem anfragenden Prozess zugeordnet. Ein Beispiel für eine Situation, in der ein weicher Seitenfehler auftritt, ist wenn in einer Seite, die in den virtuellen Speicher eines weiteren Prozesses eingebunden ist, ein Schreibzugriff durchgeführt werden soll. In konkreten Fall kann die Seite nicht mehr zwischen mehreren Prozessen geteilt werden. Das Betriebssystem muss eine Kopie der Seite erstellen und den Seitentabelleneintrag anpassen. Im Gegensatz zu harten Seitenfehlern verlangsamen weiche Seitenfehler die Prozessabarbeitung nur geringfügig [115].

Eine andere Ausnahme, die das Betriebssystem im Zusammenhang mit dem Paging behandeln muss, ist die sogenannte *Schutzverletzung* (englisch: *Segmentation Fault* oder *Segmentation Violation*), die auch *Access Violation Ausnahme* oder *General Protection Fault Ausnahme* heißt. Trotz des Namens handelt es sich bei dieser Ausnahme um ein Paging-Problem, das nichts mit Segmentierung (siehe Abschnitt 5.3.4) zu tun hat. Ausgelöst wird eine solche Ausnahme, wenn ein Prozess versucht, auf eine virtuelle Speicheradresse zuzugreifen, auf die er nicht zugreifen darf. Bei einigen Windows-Betriebssystemen aus der Vergangenheit waren Schutzverletzungen häufig ein Grund für Sys-

temabstürze und hatten einen *Blue Screen* zur Folge. Unter Linux wird als Ergebnis das Signal `SIGSEGV` erzeugt [47].

5.3.4

Segment-based Memory Management (Segmentation)

With *segment-oriented memory* or *segmentation*, the virtual memory of the processes consists of segments of different size.

The operating system maintains a segment table for each process. Each segment table record includes the segment size and its start address in main memory. Virtual addresses of processes are converted into physical addresses by using segment tables.

One drawback of segment-based memory is that for each segment to be stored in the main memory, a memory block of this size must be available in a consecutive area.

Internal fragmentation does not occur in a segment-based memory. External fragmentation occurs as in the dynamic partitioning concept (see Section 5.1.2), but with a lesser impact [108].

Segment Table Structure and Address Translation

Just like in paging, each segment table record includes a *present-bit* indicating whether the page is inside the main memory (see Figure 5.17). When a program tries to request a segment that is not located in main memory, the operating system raises a *segment not present* exception.

Also, each record includes a *dirty-bit* that indicates whether the page has been modified and further control bits that, among other things, specify access privileges. Furthermore, each record includes an entry specifying the segment size and segment base. It is combined with the offset of the virtual address. The address conversion method of a segment-based memory is shown in Figure 5.18.

The *maximum segment size* is specified by the offset length of the virtual addresses. For example, if the length of the offset is 12 bits,

Segmentorientierter Speicher (Segmentierung)

Beim *segmentorientierten Speicher*, der sogenannten *Segmentierung*, besteht der virtuelle Speicher der Prozesse aus Segmenten unterschiedlicher Länge.

Das Betriebssystem verwaltet für jeden Prozess eine Segmenttabelle (englisch: *Segment Table*). Jeder Eintrag der Segmenttabelle enthält die Länge des Segments und seine Startadresse im Hauptspeicher. Virtuelle Adressen der Prozesse werden mit Hilfe der Segmenttabellen in physische Adressen umgerechnet.

Ein Nachteil der Segmentierung ist, dass für jedes Segment, das im Hauptspeicher abgelegt werden soll, ein entsprechend großer zusammenhängender Speicherbereich frei sein muss.

Interne Fragmentierung gibt es bei der Segmentierung nicht. Externe Fragmentierung entsteht wie beim Konzept der dynamischen Partitionierung (siehe Abschnitt 5.1.2), ist aber nicht so deutlich ausgeprägt [107].

Struktur der Segmenttabellen und Adressumwandlung

Genau wie beim Paging enthält auch jeder Eintrag in einer Segmenttabelle ein *Present-Bit*, das angibt, ob die Seite im Hauptspeicher liegt (siehe Abbildung 5.17). Versucht ein Programm auf ein Segment zuzugreifen, das nicht im Hauptspeicher liegt, löst das Betriebssystem eine *Segment not present*-Ausnahme aus.

Zudem enthält jeder Eintrag ein *Dirty-Bit*, das angibt, ob die Seite verändert wurde, und weitere Steuerbits, die unter anderem Zugriffsrechte definieren. Zudem enthält jeder Eintrag die Länge des Segments und die Segmentbasis. Diese wird mit dem Offset der virtuellen Adresse verknüpft. Das Prinzip der Adressumwandlung bei Segmentierung zeigt Abbildung 5.18.

Die Länge des Offsets der virtuellen Adressen definiert die *maximale Segmentgröße*. Ist der Offset zum Beispiel 12 Bits lang, ist die maximale

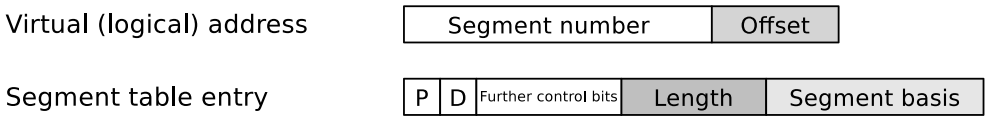

Figure 5.17: Segment Table Structure

the maximum segment size for this memory management method is $2^{12} = 4096$ bits and thus 512 bytes.

Segmentgröße bei dieser Form der Speicherverwaltung $2^{12} = 4096$ Bits und damit 512 Bytes.

5.3.5

State of the Art of Virtual Memory

Modern operating systems (for x86-compatible CPUs) operate in protected mode and only use paging. The segments Data, Code, Extra, and Stack, whose start addresses in main memory specify the corresponding registers (see Section 5.3.1), cover the entire address space (see Figure 5.19). The example shows the address space of a 32-bit operating system. This way, the entire address space of each process can be addressed via the offset. Segmentation is, therefore, no longer effectively used. This mode of operation is called the *flat memory model*.

The fact that segmentation in the flat memory model no longer provides memory protection is not a problem because of additional paging. One advantage of the flat memory model method is that it eases the porting of operating systems to other CPU architectures that do not support segment-based memory management.

Stand der Technik beim virtuellen Speicher

Moderne Betriebssysteme (für x86-kompatible Prozessoren) arbeiten im Protected Mode und verwenden ausschließlich Paging. Die Segmente Data, Code, Extra und Stack, deren Startadressen im Hauptspeicher die entsprechenden Register (siehe Abschnitt 5.3.1) definieren, decken den gesamten Adressraum ab (siehe Abbildung 5.19). Das Beispiel zeigt den Adressraum eines 32-Bit-Betriebssystems. Damit ist der vollständige Adressraum jedes Prozesses über den Offset adressierbar. Segmentierung wird somit effektiv nicht mehr verwendet. Diese Arbeitsweise heißt *Flat Memory-Modell*.

Dass die Segmentierung beim Flat Memory-Modell keinen Speicherschutz mehr bietet ist wegen des nachgeschalteten Pagings kein Nachteil. Ein Vorteil des Flat Memory-Modells ist, dass Betriebssysteme leichter auf andere Prozessorarchitekturen ohne Segmentierung portiert werden können.

5.3.6

Kernel Space and User Space

Operating systems separate the virtual address space of each process into *kernel space*, also called *kernel address space*, and *user space*, also called *user address space*. Kernel space is the memory area for the kernel and its extensions or modules (drivers). User space is the area for the currently running process that the operating system extends using swap memory (Linux/Unix: *swap*, Windows: *page file*).

Kernelspace und Userspace

Die Betriebssysteme unterteilen den virtuellen Adressraum jedes Prozesses in den *Kernelspace*, der auch *Kernel Address Space* heißt, und in den *Userspace*, der auch *User Address Space* heißt. Der Kernelspace ist der Bereich für den Betriebssystemkern und seine Erweiterungen bzw. Module (Treiber). Der Userspace ist der Bereich für den aktuell ausgeführten Prozess, den das Betriebssystem um den Auslagerungsspeicher (Linux/Unix: *Swap*, Windows: *Page-File*) vergrößert.

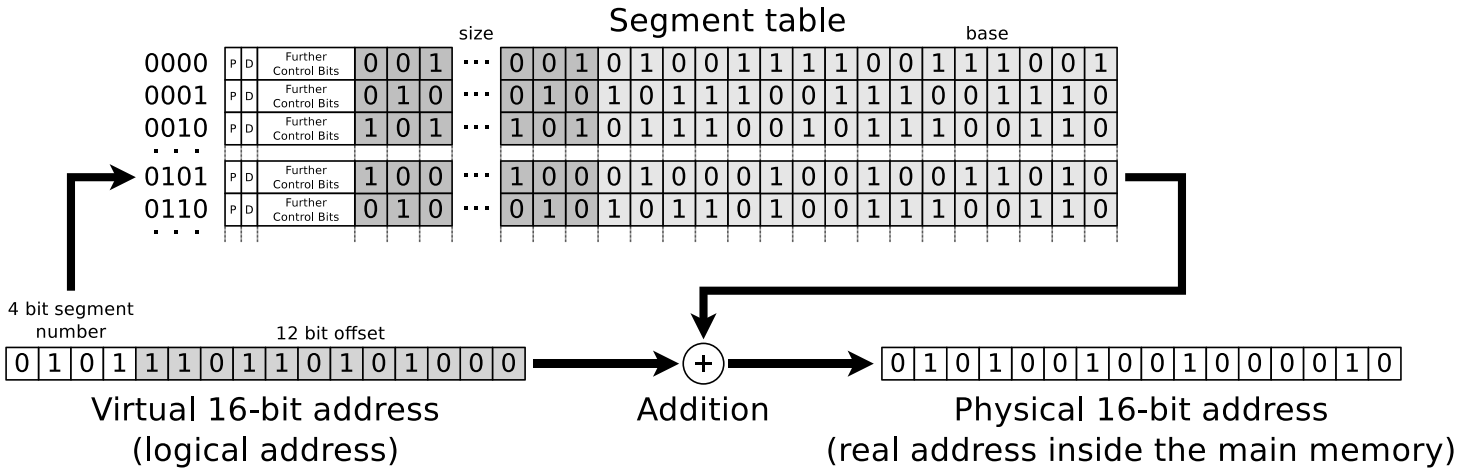

Figure 5.18: Address Translation with Segment-based Memory

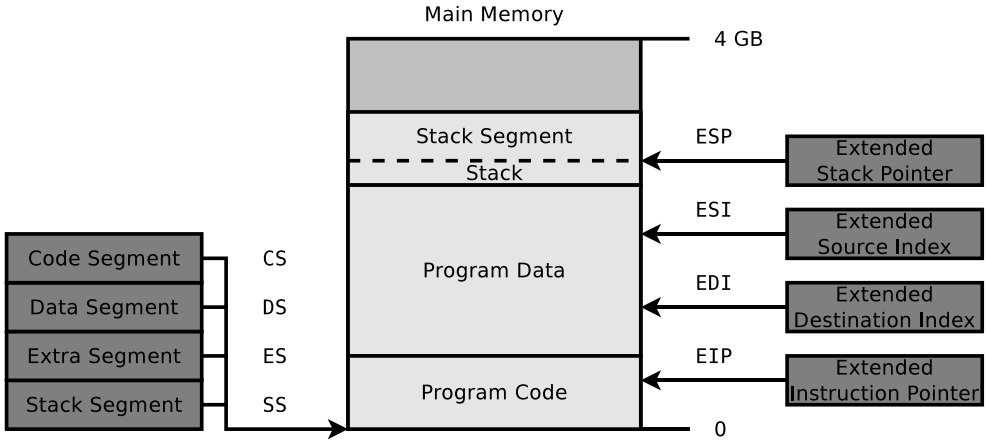

Figure 5.19: Data, Code, Extra, and Stack Segment cover the entire Address Space in the Flat Memory Model

The virtual address space (virtual memory) of 32-bit CPUs is limited to 4 GB per process. The Linux kernel on 32-bit systems by default reserves 25% for the kernel and 75% for processes in user mode. For Windows on 32-bit systems, the default ratio is 50:50 [17]. This method is called *3G/1G* or *2G/2G* split. With a *3G/1G* split on the Intel IA32 architecture (x86-32), the virtual memory addresses `0x00000000` to `0xBFFFFFFF` are user space. The running user mode process can request these addresses. This area can be requested by processes in user mode, but also kernel mode. The virtual addresses `0xC0000000` to `0xFFFFFF` comprise the kernel space. Only processes in kernel mode are allowed to request these addresses.

64-bit CPUs with a 64-bit address bus are able to address 16 exabytes of memory in theory. However, the popular 64-bit CPUs have an address bus that is only 36, 44, or 48 bits wide (see Section 4.1.3). Nevertheless, the 48 bits wide address bus of the AMD64 (x86-64) architecture allows addressing 256 TB of memory. These CPUs operate with a 64 bits wide data bus, and the 64-bit operating systems with 64 bits long memory addresses, but only the 48 least significant bits are used for addressing.

Der virtuelle Adressraum (virtuelle Speicher) von 32-Bit-Prozessoren ist auf 4 GB pro Prozess beschränkt. Der Betriebssystemkern von Linux reserviert auf 32-Bit-Systemen standardmäßig 25% für den Betriebssystemkern und 75% für die Prozesse im Benutzermodus. Bei Windows auf 32-Bit-Systemen ist das Verhältnis standardmäßig 50:50 [17]. In diesem Zusammenhang spricht man auch vom sogenannten *3G/1G*- oder *2G/2G*-Split. Bei einem *3G/1G*-Split auf der Intel IA32-Architektur (x86-32) bilden die virtuellen Speicheradressen `0x00000000` bis `0xBFFFFFFF` der Userspace. Diese Adressen stehen dem laufenden Prozess im Benutzermodus zur Verfügung. Auf diesen Bereich können Prozesse im Benutzermodus, aber auch im Kernelmodus zugreifen. Die virtuellen Speicheradressen `0xC0000000` bis `0xFFFFFFFF` bilden der Kernelspace. Auf diese Adressen können nur Prozesse im Kernelmodus zugreifen.

64-Bit-Prozessoren mit einem 64 Bits breiten Adressbus könnten theoretisch 16 Exabyte Speicher adressieren. Bei den gängigen 64-Bit-Prozessoren ist der Adressbus aber nur 36, 44 oder 48 Bits breit (siehe Abschnitt 4.1.3). Ein 48 Bits breiter Adressbus bei der Architektur AMD64 (x86-64) ermöglicht es immerhin, 256 TB Speicher zu adressieren. Diese Prozessoren arbeiten mit einem 64 Bits breiten Datenbus und die 64-Bit-Betriebssysteme zwar mit 64 Bits langen Speicheradressen, bei der Adressierung

If the address bus is only 36 or 44 bits wide, only the same number of least significant bits of the memory addresses are used.

If a 32-bit process runs in a 64-bit operating system, it will be allowed to use the first 4 GB of the entire user space.

A 64-bit process running in a 64-bit operating system with the AMD64 (x86-64) architecture, i.e., with a 48 bits wide address bus, can use the *lower half* of the 256 TB address space in user mode. These are the memory addresses `0x00000000 000000000` to `0x00007FFF FFFFFFFF`. These addresses now form the user space. The *upper half* of the 256 TB address space is used as kernel space by the kernel. These are the memory addresses `0xFFFF8000 000000000` to `0xFFFFFFFF FFFFFFFF`.

werden aber nur die 48 niederwertigsten Bits verwendet. Ist der Addressbus nur 36 oder 44 Bits breit, werden auch nur entsprechend viele niederwertigste Bits der Speicheradressen verwendet.

Wird ein 32-Bit-Prozess in einem 64-Bit-Betriebssystem ausgeführt, erhält es die ersten 4 GB des Userspace vollständig.

Einem 64-Bit-Prozess in einem 64-Bit-Betriebssystem bei der Architektur AMD64 (x86-64), also bei einem 48 Bits breiten Adressbus, steht die *untere Hälfte* des 256 TB großen Addressraums im Benutzermodus zur Verfügung. Dabei handelt es sich um die Speicheradressen `0x00000000 000000000` bis `0x00007FFF FFFFFFFF`. Diese Adressen bilden nun den Userspace. Die *obere Hälfte* des 256 TB großen Addressraums steht dem Betriebssystemkern als Kernelspace zur Verfügung. Dabei handelt es sich um die Speicheradressen `0xFFFF8000 000000000` bis `0xFFFFFFFF FFFFFFFF`.

5.4

Page Replacement Strategies

The efficiency of a memory management strategy for main memory and cache is indicated by how successful it is in keeping those pages in memory that are requested frequently. Since fast memory is a limited resource, memory management must try to identify those pages in memory that are unlikely to be requested soon and replace them first if memory capacity is requested.

In case of a request to a memory, two results are possible. If the requested data is available in memory, it is a *hit*. If the requested data is not available in memory, it is a *miss*. Based on this, the efficiency of memory management can be evaluated using the metrics *hit rate* and *miss rate*.

Dividing the number of requests to the memory with the result *hit* by the number of all requests results in a value between zero and one. It is the *hit rate*. The higher the *hit rate*,

Seitenersetzungsstrategien

Die Effizienz einer Speicherverwaltung für Hauptspeicher und Cache zeigt sich dadurch, wie erfolgreich sie darin ist, diejenigen Seiten im Speicher zu halten, auf die häufig zugegriffen wird. Da schneller Speicher ein knappes Gut ist, muss die Speicherverwaltung versuchen, diejenigen Seiten im Speicher zu identifizieren, auf die in naher Zukunft vermutlich nicht zugegriffen wird und diese bei weiteren Speicheranforderungen zu verdrängen.

Bei einer Anfrage an einen Speicher sind prinzipiell zwei Ergebnisse möglich. Liegen die angefragten Daten im Speicher, handelt es sich um einen Treffer (englisch: *Hit*). Alternativ sind die angefragten Daten nicht im Speicher vorhanden (englisch: *Miss*). Darauf basierend kann die Effizienz einer Speicherverwaltung mit Hilfe der beiden Kennzahlen *Hitrate* und *Missrate* bewerten werden.

Die Division der Anzahl der Anfragen an den Speicher mit dem Ergebnis Hit durch die Anzahl aller Anfragen liefert als Ergebnis einen Wert zwischen Null und Eins. Dabei handelt es sich um die Hitrate. Je höher die Hitrate ist, desto

the more efficient is the memory management in the given application scenario.

The *miss rate* is calculated by dividing the number of requests to memory with the result *miss* by the number of all requests. Alternatively: miss rate = 1 − hit rate.

The following subsections describe the working methods of some page replacement strategies, their advantages and drawbacks, and typical areas of application. For each page replacement strategy, this book presents one example with a specific request sequence to an initially empty memory of finite capacity. To be precise, the requests are requests for pages inside the virtual address space of a process. If a requested page is not already in the cache, the operating system must request it from the main memory or even from swap memory.

The following sections provide descriptions and examples of the optimal strategy, from Least Recently Used, Least Frequently Used, FIFO, Clock that is also called Second Chance, and of Random. This is not a comprehensive list of the existing page replacement strategies. Further methods like Time To Live or Most Frequently Used (MFU) are not discussed in this book due to the limited space, and because they play a marginal role in practice.

effizienter arbeitet die Speicherverwaltung im konkreten Anwendungsszenario.

Die Missrate ergibt sich aus der Division der Anzahl der Anfragen an den Speicher mit dem Ergebnis Miss durch die Anzahl aller Anfragen oder alternativ: Missrate = 1 − Hitrate.

Die folgenden Unterabschnitte beschreiben die Arbeitsweise einiger Seitenersetzungsstrategien, deren Vor- und Nachteile sowie typische Einsatzbereiche. Dieses Buch präsentiert zu jeder Seitenersetzungsstrategie ein Beispiel mit einer bestimmten Anfragesequenz auf einen initial leeren Speicher mit endlicher Kapazität. Konkret stellen diese *Anfragen* Anforderungen an Seiten im virtuellen Adressraum eines Prozesses dar. Wenn eine angefragte Seite nicht schon im Cache ist, muss das Betriebssystem sie aus dem Hauptspeicher oder gar aus dem Auslagerungsspeicher (Swap) nachladen.

Die folgenden Abschnitte enthalten Beschreibungen und Beispiele der optimalen Strategie, von Least Recently Used und Least Frequently Used, FIFO, Clock bzw. Second Chance und Random. Es handelt sich hierbei um keine vollständige Liste der existierenden Seitenersetzungsstrategien. Weitere Verfahren wie Time To Live oder Most Frequently Used (MFU) werden aus Platzgründen – und weil sie in der Praxis eine untergeordnete Rolle spielen – in diesem Buch nicht weiter behandelt.

5.4.1

Optimal Strategy

The *optimal page replacement strategy* always replaces the page that is not requested for the longest time in the future. The downside of this method is that it is impossible to implement because the future is unpredictable. In practice, operating systems can only evaluate memory requests from the past and try to make conclusions about future requirements from the collected historical data [39]. The optimal strategy is only used for the evaluation and comparison with other page replacement strategies. It delivers a result that the memory management of operating systems is supposed to most closely approximate in practice [123].

Optimale Strategie

Die *optimale Seitenersetzungsstrategie* verdrängt immer diejenige Seite, auf die am längsten in der Zukunft nicht zugegriffen wird. Ein offensichtlicher Nachteil dieses Verfahrens ist, dass es wegen der Unvorhersehbarkeit der Zukunft unmöglich zu implementieren ist. In der Praxis können die Betriebssysteme nur die Speicherzugriffe in der Vergangenheit auswerten und versuchen, daraus Rückschlüsse auf zukünftige Anforderungen zu ziehen [39]. Relevant ist die optimale Strategie einzig zum Vergleich mit anderen Seitenersetzungsstrategien und der Bewertung von deren Effizienz. Sie liefert ein Ergebnis, dem die Speicherverwaltung von Betriebssystemen in der Praxis möglichst nahe kommen sollen [123].

Requests:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page 1:	1	1	1	1	1	1	1	1	1	3	3	3
Page 2:		2	2	2	2	2	2	2	2	2	4	4
Page 3:			3	4	4	4	5	5	5	5	5	5

Figure 5.20: Optimal Page Replacement Strategy Example

Figure 5.20 shows that the specified sequence of 12 requests to a memory that is initially empty and has a capacity of just three pages, results in seven miss events when using the optimal strategy. Consequently, the hit rate is ≈ 0.42, and the miss rate is ≈ 0.58. The pages that are highlighted in gray in Figure 5.20 are the new pages that are loaded into memory, i.e., the miss events.

Abbildung 5.20 zeigt, dass die angegebene Sequenz aus 12 Anfragen auf einen initial leeren Speicher mit einer Kapazität von nur drei Seiten unter Verwendung der optimalen Strategie zu sieben Miss-Ereignissen führt. Konsequenterweise ist die Hitrate $\approx 0{,}42$ und die Missrate $\approx 0{,}58$. Diejenigen Seiten, die in Abbildung 5.20 grau hinterlegt sind, sind die neu in den Speicher geladenen Seiten, also die Miss-Ereignisse.

5.4.2

Least Recently Used

The page replacement strategy *Least Recently Used* (LRU) replaces the page that has not been requested for the longest time. For this purpose, the operating system manages a queue in which all page numbers are referenced. If a page is loaded into memory or referenced, it is relocated to the front of the queue. If the memory has no more free capacity and a miss occurs, the page at the end of the queue is replaced. A drawback of this method is that it ignores the number of requests.

Figure 5.21 shows that the sequence of requests that is already familiar from Section 5.4.1 to a memory, that is initially empty and has a capacity of just three pages, results in ten miss events when using LRU. Hence, the hit rate is ≈ 0.17, and the miss rate is ≈ 0.83.

Least Recently Used

Die Seitenersetzungsstrategie *Least Recently Used* (LRU) verdrängt diejenige Seite, auf die am längsten nicht zugegriffen wurde. Dafür verwaltet das Betriebssystem eine Warteschlange, in der die Seitennummern eingereiht sind. Wird eine Seite in den Speicher geladen oder auf diese zugegriffen, wird sie am Anfang der Warteschlange eingereiht. Ist der Speicher voll und es kommt zum Miss, lagert das Betriebssystem die Seite am Ende der Warteschlange aus. Ein Nachteil dieses Verfahrens ist, dass es nicht die Zugriffshäufigkeit berücksichtigt.

Abbildung 5.21 zeigt, dass die aus Abschnitt 5.4.1 bereits bekannte Anfragesequenz auf einen initial leeren Speicher mit einer Kapazität von drei Seiten bei Verwendung von LRU zu zehn Miss-Ereignissen führt. Die Hitrate ist somit $\approx 0{,}17$ und die Missrate $\approx 0{,}83$.

5.4.3

Least Frequently Used

When using the *Least Frequently Used* (LFU) strategy, the operating system replaces the page that was requested the least often. For implementing this strategy, the operating system

Least Frequently Used

Bei der Strategie *Least Frequently Used* (LFU) verdrängt das Betriebssystem diejenige Seite, auf die am wenigsten häufig zugegriffen wurde. Um diese Strategie zu realisieren, verwaltet das Betriebssystem für jede Seite in der Seitenta-

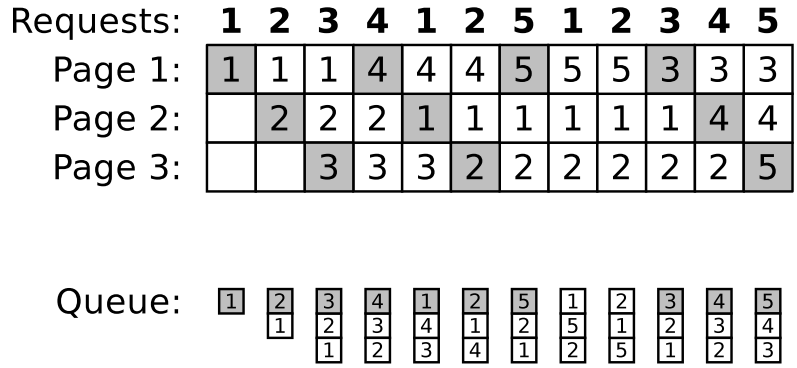

Figure 5.21: Least Recently Used Page Replacement Strategy Example

manages a reference counter for each page in the page table, which stores the number of requests.

If memory that is managed with LFU has no more free capacity and a miss occurs, then the page with the lowest reference counter value is replaced. One advantage of this method is that it takes into account how often pages are requested. However, one drawback is that pages that have often been requested in the past may block the memory.

belle einen Referenzzähler, der die Anzahl der Zugriffe speichert.

Sind alle Speicherplätze in einem mit LFU verwalteten Speicher belegt und kommt es zum Miss, wird diejenige Seite aus dem Speicher verdrängt, deren Referenzzähler den niedrigsten Wert hat. Ein Vorteil dieses Verfahrens ist, dass es die Zugriffshäufigkeit der Seiten berücksichtigt. Ein sich daraus ergebender Nachteil ist allerdings, dass Seiten, auf die in der Vergangenheit häufig zugegriffen wurde, den Speicher blockieren können.

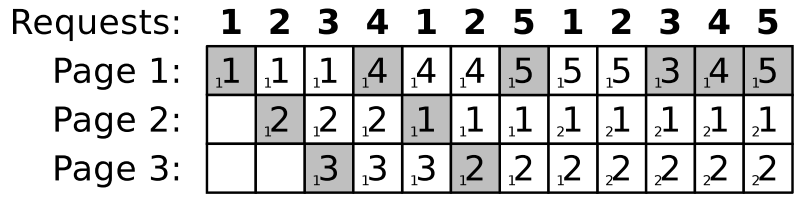

Figure 5.22: Least Frequently Used Page Replacement Strategy Example

Figure 5.22 shows the sequence of requests, which is already familiar from Section 5.4.1. For a memory that is initially empty and has a capacity of just three pages, the sequence sequence results in ten miss events when using LFU. Hence, the hit rate is ≈ 0.17, and the miss rate is ≈ 0.83.

Abbildung 5.22 zeigt, dass die aus Abschnitt 5.4.1 bereits bekannte Anfragesequenz auf einen initial leeren Speicher mit einer Kapazität von drei Seiten bei Verwendung von LFU zu 10 Miss-Ereignissen führt. Die Hitrate in diesem Szenario ist $\approx 0{,}17$ und die Missrate $\approx 0{,}83$.

5.4.4

First In First Out

The page replacement strategy *First In First Out* (FIFO) always replaces the page that has been in memory for the longest time. A common assumption when using FIFO is that, as with any other page replacement strategy, expanding the memory while keeping the sequence of requests results in less or at worst the same number of miss events. This assumption was proven wrong for FIFO in 1969 by Laszlo Belady. He demonstrated that with specific request patterns, FIFO causes more miss events in expanded memory [9]. This phenomenon is called *Belady's anomaly* in literature. Since the discovery of Belady's anomaly, FIFO, if used at all, usually follows another page replacement strategy.

First In First Out

Die Seitenersetzungsstrategie *First In First Out* (FIFO) verdrängt immer diejenige Seite, die sich am längsten im Speicher befindet. Eine logische Annahme bei der Verwendung von FIFO ist, dass wie bei allen anderen Seitenersetzungsstrategien eine Vergrößerung des Speichers bei gleichbleibender Anfragesequenz zu weniger oder schlechtestenfalls gleich vielen Miss-Ereignissen führt. Diese Annahme wurde für FIFO 1969 durch Laszlo Belady wiederlegt. Er zeigte, dass bei bestimmten Zugriffsmustern FIFO bei einem vergrößerten Speicher zu mehr Miss-Ereignissen führt [9]. Dieses Phänomen heißt in der Literatur *Beladys Anomalie*. Seit der Entdeckung von Beladys Anomalie wird FIFO, wenn es denn überhaupt noch verwendet wird, meist einer anderen Seitenersetzungsstrategie nachgeschaltet.

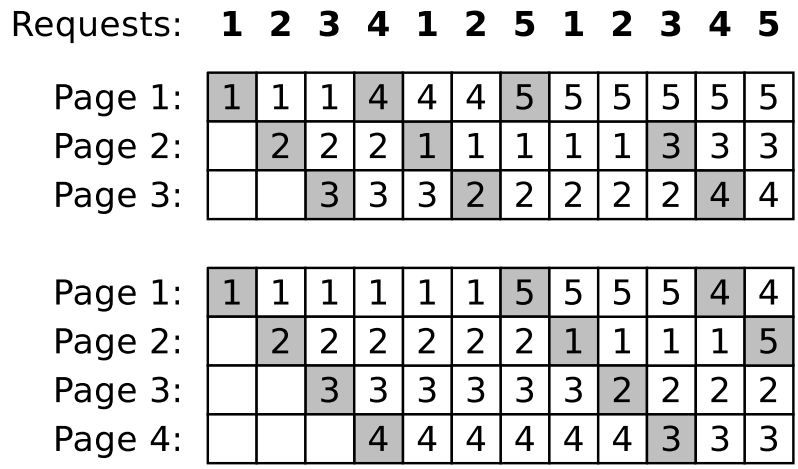

Figure 5.23: FIFO Page Replacement Strategy Example with a Sequence of Requests that triggers Belady's Anomaly

Figure 5.23 shows an example of FIFO, where Belady's anomaly is visible. For an initially empty memory with a capacity of four pages, using FIFO with the given sequence of requests results in 10 miss events. For a smaller memory with a capacity of three pages, the sequence of requests results in only nine miss events.

Abbildung 5.23 zeigt ein Beispiel für FIFO, bei dem Beladys Anomalie sichtbar ist. Bei einem initial leeren Speicher mit einer Kapazität von vier Seiten führt die Verwendung von FIFO bei der gegebenen Anfragesequenz zu 10 Miss-Ereignissen. Bei einem kleineren Speicher mit einer Kapazität von drei Seiten führt die Anfragesequenz nur zu neun Miss-Ereignissen.

5.4.5

Clock / Second Chance

When using the *Clock* strategy, which is also called *Second Chance*, the operating system uses the *reference-bit* (see Section 5.3.3) in the page Table [39, 119] when selecting the next page to be replaced. If a page is loaded into memory, the reference-bit has the value zero. If a page is requested, then the operating system modifies the value of the reference-bit to the value one.

The operating system also manages a pointer that refers to the last requested page. In case of a miss, the memory is scanned, starting from the position of the pointer, for the first page, whose reference-bit has value zero. This page is replaced then. For all pages, which are examined during the scan, where the reference-bit has value one, the value is set to zero. This approach ensures that when searching for a proper page to replace, pages that were requested at least once in the past after being loaded into the memory, get a second chance.

Clock / Second Chance

Bei *Clock*, das auch *Second Chance* heißt, berücksichtigt das Betriebssystem bei der Auswahl der nächsten zu ersetzenden Seite das *Reference-Bit* (siehe Abschnitt 5.3.3) in der Seitentabelle [39, 119]. Wird eine Seite in den Speicher geladen, hat das Reference-Bit den Wert Null. Wird auf eine Seite zugegriffen, ändert das Betriebssystem den Wert des Reference-Bits auf den Wert Eins.

Zudem verwaltet das Betriebssystem einen Zeiger, der auf die zuletzt zugegriffene Seite zeigt. Beim Miss wird der Speicher ab dem Zeiger nach der ersten Seite durchsucht, deren Reference-Bit den Wert Null hat. Diese Seite wird ersetzt. Bei allen bei der Suche durchgesehenen Seiten, bei denen das Reference-Bit den Wert Eins hat, wird der Wert auf Null zurückgesetzt. Beim Suchen nach einer geeigneten Seite zum Verdrängen erhalten also Seiten, auf die in der Vergangenheit nach dem Laden in den Speicher mindestens ein weiteres Mal zugegriffen wurde, eine zweite Chance.

Requests:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Page 1:	$_0 1^X$	$_0 1$	$_0 1$	$_0 4^X$	$_0 4$	$_0 4$	$_0 5^X$	$_0 5$	$_0 5$	$_0 3^X$	$_0 4^X$	$_0 4$
Page 2:		$_0 2^X$	$_0 2$	$_0 2$	$_0 1^X$	$_0 1$	$_0 1$	$_1 1^X$	$_1 1$	$_1 1$	$_0 1$	$_0 5^X$
Page 3:			$_0 3^X$	$_0 3$	$_0 3$	$_0 2^X$	$_0 2$	$_0 2$	$_1 2^X$	$_1 2$	$_0 2$	$_0 2$

Figure 5.24: Clock (Second Chance) Page Replacement Strategy Example

Figure 5.24 shows the sequence of requests that is already familiar from Section 5.4.1. For an initially empty memory with a capacity of three pages, the sequence results in ten miss events when using clock (second chance). Hence, the hit rate is ≈ 0.17, and the miss rate is ≈ 0.83.

Examples of operating systems that use this page replacement strategy or variants are Linux, BSD-Unix, VAX/VMS (originally from Digital Equipment Corporation), and Microsoft Windows NT 4.0 on uniprocessors systems [99, 102].

Abbildung 5.24 zeigt, dass die aus Abschnitt 5.4.1 bereits bekannte Anfragesequenz auf einen initial leeren Speicher mit einer Kapazität von drei Seiten bei Verwendung von Clock bzw. Second Chance zu 10 Miss-Ereignissen führt. Die Hitrate ist somit $\approx$ 0,17 und die Missrate $\approx$ 0,83.

Beispiele für Betriebssysteme, die diese Ersetzungsstrategie oder Varianten davon verwenden, sind Linux, BSD-Unix, VAX/VMS (ursprünglich von der Digital Equipment Corporation) und Microsoft Windows NT 4.0 auf Uniprozessor-Systemen [99, 102].

5.4.6

Random

This page replacement strategy replaces pages randomly. Since this method does not require any additional administrative information in the page table, it is more resource-efficient than other strategies. Furthermore, unlike the Second Chance strategy, for example, it is not necessary to search for a page that can be replaced. One drawback is that pages that are often used are replaced from memory, which in consequence results in reduced performance due to later reloading.

Examples of operating systems that use the random page replacement strategy are IBM OS/390 and Windows NT 4.0 on multiprocessor systems [90, 99]. Another example is the Intel i860 RISC CPU. It uses *random* as the page replacement strategy for the cache [95].

Random

Bei der Seitenersetzungsstrategie Random werden zufällige Seiten verdrängt. Da dieses Verfahren ohne zusätzliche Verwaltungsinformationen in der Seitentabelle auskommt, ist es ressourcenschonender als andere Strategien. Zudem muss nicht wie zum Beispiel bei Second Chance langwierig nach einer Seite gesucht werden, die verdrängt werden kann. Nachteilig ist dafür, dass unter Umständen häufig verwendete Seiten aus dem Speicher verdrängt werden, was wiederum durch das spätere Nachladen zu Geschwindigkeitsverlusten führt.

Beispiele für Betriebssysteme, die die Ersetzungsstrategie Random einsetzen, sind IBM OS/390 und Windows NT 4.0 auf Multiprozessor-Systemen [90, 99]. Ein weiteres Einsatzbeispiel ist der Intel i860 RISC-Prozessor. Dieser verwendet Random als Ersetzungsstrategie für den Cache [95].

6

File Systems

Dateisysteme

Not only the management of main memory and cache (see Chapter 5) but also the management of mass storage (e.g., solid-state drives and hard disk drives) with file systems belongs to the operating system's tasks. File systems organize the storage of files on data storage devices. Files are sequences of bytes that represent data belonging together. File systems manage the names and attributes (metadata) of the files and form a namespace, i.e., a hierarchy of directories and files.

File systems abstract the complexity of data storage on mass storage, so that processes and users can abstractly address files using file names and do not need to access the individual storage addresses of the files directly. File systems are supposed to implement this comfort improvement with as little effort (overhead) as possible for metadata.

This chapter begins with a description of the technical principles of Linux file systems, followed by a description of the basics of MS-DOS and Windows file systems. Finally, the chapter discusses the topics journaling, extents, copy-on-write, file system cache, and defragmentation.

Neben der Verwaltung des Hauptspeichers und des Cache (siehe Kapitel 5) gehört auch die Verwaltung des Massenspeichers (zum Beispiel Solid-State Drives und Festplatten) mit Dateisystemen zu den Aufgaben der Betriebssysteme. Dateisysteme organisieren die Ablage von Dateien auf Datenspeichern. Dateien sind Folgen von Bytes, die inhaltlich zusammengehörende Daten repräsentieren. Dateisysteme verwalten die Namen und Attribute (Metadaten) der Dateien und bilden einen Namensraum, also eine Hierarchie von Verzeichnissen und Dateien.

Dateisysteme abstrahieren die Komplexität der Datenhaltung auf Massenspeicher dahingehend, dass die Prozesse und Benutzer auf Dateien abstrakt über deren Dateinamen ansprechen können und nicht direkt auf die einzelnen Speicheradressen der Dateien zugreifen müssen. Diesen Zugewinn an Komfort sollen die Dateisysteme mit möglichst wenig Aufwand (Overhead) für Verwaltungsinformationen realisieren.

Zu Beginn dieses Kapitels erfolgt eine Beschreibung der technischen Grundlagen der Linux-Dateisysteme, gefolgt von einer Beschreibung der technischen Grundlagen der Dateisysteme von MS-DOS und Windows. Abschließend enthält das Kapitel eine Auseinandersetzung mit den Themen Journaling, Extents, Copy-on-Write, Dateisystem-Cache und Defragmentierung.

6.1

Technical Principles of File Systems

Technische Grundlagen der Dateisysteme

File systems address *clusters* and not storage device blocks. Each file occupies an integer num-

Dateisysteme adressieren *Cluster* und nicht Blöcke des Datenträgers. Jede Datei belegt eine

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_6

ber of clusters. In literature, clusters are often called *blocks* [108, 112] and less often *zones* [118]. There is a tendency to confuse these with the sectors of hard disk drives, which sometimes are called blocks in literature, too. The optimal cluster size is essential for the efficiency of the file system (see Table 6.1). The bigger the clusters are, the more storage capacity is lost because of internal fragmentation.

A simple example demonstrates the consequences of different cluster sizes. If a 1 kB large file is to be stored on a file system with 4 kB large clusters, 3 kB are lost due to internal fragmentation. If the same file is to be stored on a file system with 64 kB clusters, 63 kB are lost. The cluster size can be specified with certain restrictions when creating the file system. In Linux, the following applies: cluster size $\leq$ page size. As described in Section 5.1.3, the page size depends on the hardware architecture used and is usually 4 kB.

ganzzahlige Anzahl von Clustern. In der Literatur heißen die Cluster häufig *Blöcke* [107, 115] und seltener *Zonen* [118]. Das führt leicht zu Verwechslungen mit den Sektoren der Laufwerke, die in der Literatur auch manchmal Blöcke heißen. Die Clustergröße ist wichtig für die Effizienz des Dateisystems (siehe Tabelle 6.1). Je größer die Cluster sind, desto mehr Speicher geht durch interne Fragmentierung verloren.

Ein einfaches Beispiel zeigt die Konsequenzen der unterschiedlichen Clustergrößen. Soll beispielsweise eine 1 kB große Datei auf einem Dateisystem mit 4 kB großen Clustern gespeichert werden, gehen 3 kB durch interne Fragmentierung verloren. Soll die gleiche Datei auf einem Dateisystem mit 64 kB großen Clustern gespeichert werden, gehen 63 kB verloren. Die Clustergröße kann mit gewissen Einschränkungen beim Anlegen des Dateisystems definiert werden. Unter Linux gilt: Clustergröße $\leq$ Seitengröße. Wie in Abschnitt 5.1.3 beschrieben, hängt die Seitengröße von der verwendeten Hardwarearchitektur ab und ist meist 4 kB.

Table 6.1: Benefits and Drawbacks of increasing or shrinking the Cluster Size of a File System

	The smaller the clusters are ...	The bigger the clusters are ...
Advantage	the less capacity is lost due to internal fragmentation	the less overhead is caused by large files
Drawback	the more overhead is caused by large files	the more capacity gets lost due to internal fragmentation

6.2

Block Addressing in Linux File Systems

If a *file* is created in a Linux file system, an *inode* (*index node*) is created as well. It stores all metadata of a file except the filename. Metadata are, among other things, the size, user ID (UID), group ID (GID), access rights, and date of a file. Each inode is identified by its unique inode number in the file system. In addition to the well-defined storage of metadata, the inode of each file also refers to clusters that contain the actual data of the file.

Blockadressierung bei Linux-Dateisystemen

Wird in einem Linux-Dateisystem eine *Datei* angelegt, wird immer auch ein *Indexknoten*, genannt *Inode* (*Index Node*), angelegt. Dieser speichert alle Verwaltungsdaten (*Metadaten*) einer Datei außer dem Dateinamen. Metadaten sind unter anderem Dateigröße, Benutzerzugehörigkeit (User-ID = UID), Gruppenzugehörigkeit (Group-ID = GID), Zugriffsrechte und Datum einer Datei. Jeder Inode wird über seine im Dateisystem eindeutige Inode-Nummer identifiziert. Neben der wohldefinierten Ablage der

Since each inode has a unique inode number, and the address space is finite, it is possible that in a file system with many small files, no free inode numbers are available. As a result, no further files can be created in the file system even though it still has free storage capacity. The command `df -i` is useful in such situations. It indicates the address space of inodes of all file systems which are mounted by the Linux system and how many inode numbers are still available on the individual file systems.

All Linux file systems implement the functional principle of inodes. Even a *directory* is just a file that contains the filename and the inode number of each file that is assigned to the directory [17, 50].

The traditional working method of Linux file systems is *block addressing*. This term is misleading, because file systems always address *clusters* and not blocks (of the volume), as mentioned before. However, since the term has been established in literature for decades, it is used in this book as well. Linux file systems that use block addressing include Minix and ext2/3.

Figure 6.1 shows block addressing using the example of the Linux file systems ext2/3. Each inode stores up to 12 cluster numbers directly, in addition to the metadata of the file (except the file name). If a file requires additional clusters, then these are indirectly addressed with clusters whose contents are further cluster numbers of the file.

Figure 6.1 illustrates that with a file system that has a cluster size of 1 kB, each inode can directly address only 12 kB of a file. If a file requires more storage capacity, further clusters are indirectly addressed. The file systems ext2 and ext3 use 32 bit (= 4 Byte) large cluster numbers. Each cluster of 1 kB size can, therefore, store a maximum of 256 additional cluster addresses [50]. Consequently, an ext2/3 file system with clusters that are 1 kB in size has to use an additional cluster just for storing cluster

Da jeder Inode eine eindeutige Inode-Nummer hat, und der Adressraum endlich ist, kann es vorkommen, dass in einem Dateisystem, das viele kleine Dateien speichert, keine Inode-Nummern mehr frei sind. Im Ergebnis können dann keine weiteren Dateien im Dateisystem erstellt werden, obwohl noch freie Speicherkapazität vorhanden ist. Hilfreich ist bei solchen Situationen das Kommando `df -i`. Dieses zeigt, wie groß der Adressraum an Inodes in den eingebunden Dateisystemen des Linux-Systems ist und wie viele Inode-Nummern auf den einzelnen Dateisystemen noch verfügbar sind.

Alle Linux-Dateisysteme basieren auf dem Funktionsprinzip der Inodes. Auch ein *Verzeichnis* ist nichts anderes als eine Datei, die als Inhalt für jede dem Verzeichnis zugewiesene Datei den Dateinamen und die zugehörigen Inode-Nummer enthält [17, 50].

Linux-Dateisysteme arbeiten traditionell nach dem Prinzip der *Blockadressierung*. Eigentlich ist dieser Begriff irreführend, da die Dateisysteme immer *Cluster* adressieren und nicht Blöcke (des Datenträgers), wie bereits erwähnt. Weil der Begriff aber seit Jahrzehnten in der Literatur etabliert ist, wird er auch in diesem Buch verwendet. Linux-Dateisystemen, die Blockadressierung verwenden, sind unter anderem Minix und ext2/3.

Abbildung 6.1 zeigt die Blockadressierung am Beispiel der Linux-Dateisysteme ext2/3. Jeder Inode speichert außer den Metadaten der Datei (mit Ausnahme des Dateinamens) bis zu 12 Clusternummern direkt. Benötigt eine Datei mehr Cluster, wird indirekt adressiert mit Clustern, deren Inhalt die weiteren Clusternummern der Datei sind.

In Abbildung 6.1 ist zu sehen, dass bei einer Clustergröße im Dateisystem von 1 kB jeder Inode nur 12 kB einer Datei direkt adressieren kann. Benötigt eine Datei mehr Speicherplatz, müssen weitere Cluster indirekt adressiert werden. Die Dateisysteme ext2 und ext3 verwenden 32 Bit (= 4 Byte) large Clusternummern. Jeder 1 kB große Cluster kann also maximal 256 Adressen weiterer Cluster speichern [50]. Somit muss ein ext2/3-Dateisystem mit 1 kB großen Clustern für alle Dateien mit einer Dateigröße

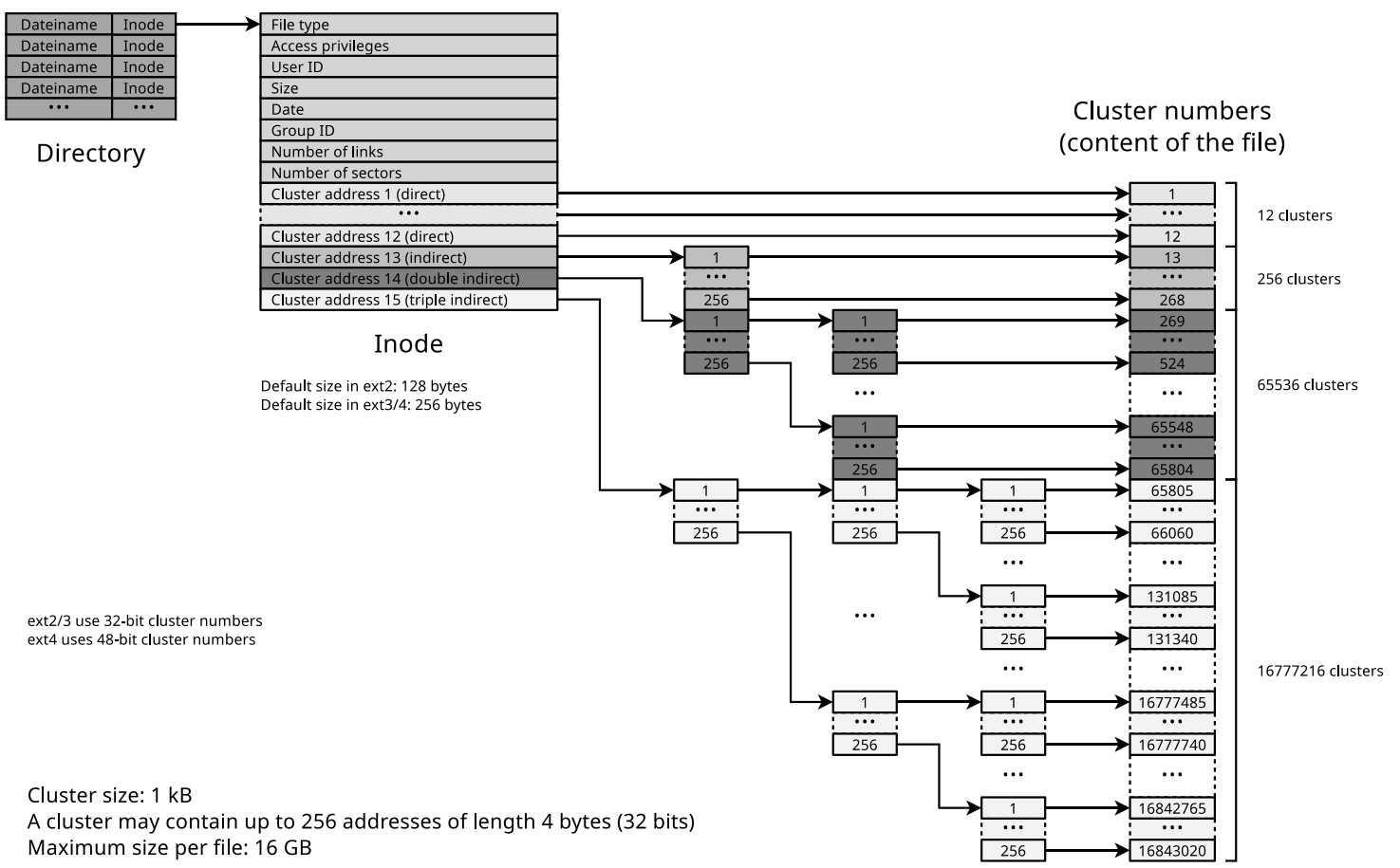

Figure 6.1: Block Addressing using the Example ext2/3

numbers for all files with a file size of $> 12\,\text{kB}$ and $\leq 268\,\text{kB}$. For even larger files, the file system must address two times indirectly or even three times indirectly. In this way, an ext2/3 file system with 1 kB large clusters can provide a maximum file size of 16 GB. With files of this size, however, the administrative effort is enormous. Imagine how many clusters have to be adjusted if a file that has a size of several gigabytes becomes resized or if a large file is moved to another file system with block addressing.

At this point, it also becomes clear that the cluster size has a strong impact on overhead. With 2 kB or 4 kB cluster size, all clusters up to a file size of 24 kB or 48 kB are addressed directly. On the other hand, the capacity loss due to internal fragmentation grows, because, in every file system, there are files that are smaller than the cluster size.

The following two sections describe the functioning of the Minix and ext2/3 file systems, which work according to the block addressing concept and have played an important role in the history of Linux.

6.2.1 Minix

The name *Minix* stands first and foremost for the Unix-like operating system, which has been in development by Andrew Tanenbaum for educational purposes since the mid-1980s. However, Minix is also the name of the standard Linux file system until 1992.

The Minix file system is particularly famous for its low overhead. Although it hardly plays a role anymore in practice with Linux today, a description of its logical structure and its characteristics is useful for an introduction to the file system topic.

A Minix file system contains just six areas and represents the storage as a linear chain of clusters of equal size (1-8 kB).

von $> 12\,\text{kB}$ und $\leq 268\,\text{kB}$ einen weiten Cluster zur Speicherung von Clusternummern verwalten. Bei noch größeren Dateien muss das Dateisystem entsprechend zweifach-indirekt oder sogar dreifach-indirekt adressieren. Auf diese Weise kann ein ext2/3-Dateisystem mit 1 kB große Clustern eine maximale Dateigröße von 16 GB realisieren. Bei derart großen Dateien ist allerdings auch der Verwaltungsaufwand enorm. Man bedenke, wie viele Cluster angepasst werden müssen, wenn sich die Größe einer mehrere Gigabyte großen Datei ändert oder wenn eine große Datei gar auf ein anderes Dateisystem mit Blockadressierung verschoben wird.

An dieser Stelle wird auch deutlich, wie groß der Einfluss der Clustergröße auf den Verwaltungsaufwand ist. Bei 2 kB bzw. 4 kB Clustergröße können bei Dateien bis zu einer Dateigröße von 24 kB bzw. 48 kB alle Cluster direkt adressiert werden. Dafür steigt der Kapazitätsverlust durch interne Fragmentierung, weil es in der Praxis auch immer Dateien im Dateisystem gibt, die kleiner als die Clustergröße sind.

Die folgenden beiden Abschnitte beschreiben die Funktionsweise der Dateisysteme Minix und ext2/3, die nach dem Prinzip der Blockadressierung arbeiten und in der Entwicklungsgeschichte von Linux eine große Rolle gespielt haben.

Minix

Der Name *Minix* steht in erster Linie für das Unix-ähnliche Betriebssystem, das seit Mitte der 1980er Jahre federführend von Andrew Tanenbaum ursprünglich als Lehrsystem entwickelt wurde. Gleichzeitig bezeichnet Minix aber auch das Minix-Dateisystem, das bis 1992 das Standard-Dateisystem von Linux war.

Das Minix-Dateisystem zeichnet sich besonders durch seinen geringen Verwaltungsaufwand aus. Auch wenn es in der Praxis unter Linux heute kaum noch eine Rolle spielt, ist eine Beschreibung seiner gut verständlichen Struktur und seiner Eigenschaften für einen Einstieg in das Thema Dateisysteme sinnvoll.

Ein Minix-Dateisystem enthält nur sechs Bereiche und der Datenspeicher wird bei Minix als lineare Kette gleichgroßer Cluster (1-8 kB) dargestellt.

Area 1	Area 2	Area 3	Area 4	Area 5	Area 6
Boot block (1 cluster)	Superblock (1 cluster)	Inodes bitmap (1 cluster)	Cluster bitmap (1 cluster)	Inodes (15 clusters)	Data (remaining clusters)

Figure 6.2: Minix File System Structure

Figure 6.2 shows the structure of a Minix file system. The *boot block* contains the boot loader that starts the operating system. The *superblock* contains information about the file system. For example, it indicates the number of inodes and clusters. The *inodes bitmap* contains a list of all inodes with the information if the inode is occupied (value: 1) or unused (value: 0). The *cluster bitmap* contains a list of all clusters with the information whether the cluster is occupied (value: 1) or unused (value: 0). The next area contains the inodes with the metadata of the stored files. Finally, there is the area that contains the contents of the files and directories. It is by far the largest part of the file system.

The Minix file system version that was state-of-the-art in the early 1990s utilized 16 bits large cluster numbers and 1 kB large clusters. For this reason, it was possible to create $2^{16} = 65{,}536$ inodes (files) per file system. The maximum file system size was 64 MB. This limitation results from the number of possible cluster numbers and the cluster size.

Abbildung 6.2 zeigt die Struktur eines Minix-Dateisystems. Der *Bootblock* enthält den Boot-Loader, der das Betriebssystem startet. Der *Superblock* enthält Informationen über das Dateisystem. Zum Beispiel ist hier die Anzahl der Inodes und Cluster angegeben. Die *Inodes-Bitmap* enthält eine Liste aller Inodes mit der Information, ob der Inode belegt (Wert: 1) oder frei (Wert: 0) ist. Das *Cluster-Bitmap* enthält eine Liste aller Cluster mit der Information, ob der Cluster belegt (Wert: 1) oder frei (Wert: 0) ist. Der darauf folgende Bereich enthält die Inodes mit den Metadaten der gespeicherten Dateien. Abschließend folgt der Bereich, der die Inhalte der Dateien und Verzeichnisse enthält. Dies ist der mit Abstand größte Bereich im Dateisystem.

Die Anfang der 1990er Jahre aktuelle Version des Minix-Dateisystems arbeitete mit 16 Bits langen Clusternummern und 1 kB großen Clustern. Aus diesem Grund konnten pro Dateisystem $2^{16} = 65.536$ Inodes (also Dateien) angelegt werden. Die maximale Dateisystemgröße waren 64 MB. Diese Limitierung ergibt sich aus der Anzahl der möglichen Clusternummern und der Clustergröße.

$$2^{16}\,\text{cluster numbers} * 1\,\text{kB cluster size} = 65{,}536\,\text{kB} = 64\,\text{MB}$$

Versions 2 and 3 of the Minix file system support more cluster numbers and bigger clusters, but at the time of their release, they did not play a significant role under Linux any longer.

The free operating system Minix was relevant in the academic context only during most of its development but has seen an unexpected usage growth since around 2015. It is used intensively on most desktops and server systems with Intel processors, even if their users and administrators do not notice it. The reason for this is that the Minix 3 operating system runs as firmware called Intel Active Management Technology (AMT) in Intel chipsets (until around 2008 in the Northbridge, later in the Platform

Die Versionen 2 und 3 des Minix-Dateisystems haben großzügigere Parameter, spielten zum Zeitpunkt ihres Erscheinens aber unter Linux schon keine signifikante Rolle mehr.

Das freie Betriebssystem Minix war die allermeiste Zeit während seiner Entwicklung nur im akademischen Kontext relevant, konnte aber seit ca. 2015 einen erstaunlichen Zuwachs an Nutzung verzeichnen. Es wird intensiv auf den meisten Desktops und Server-Systemen mit Intel-Prozessoren eingesetzt, auch wenn deren Benutzer und Administratoren davon nichts mitbekommen. Der Grund dafür ist, das in Intel-Chipsätzen (bis ca. 2008 in der Northbridge, danach im Platform Controller Hub – siehe Ab-

Controller Hub – see Section 4.1.3) with the so-called Management Engine (ME) since version 11. This fact is quite critical since we are talking about a co-processor with a permanently running operating system [36]. Furthermore, the system has full access to the main memory and most hardware components, including the Ethernet interfaces. Moreover, users and administrators cannot control or remove the management engine on most systems, and it runs as long as the computer is plugged in, even when the CPU is shut down [21]. The management engine is shipped by Intel as a cryptographically signed binary and is poorly documented. Which file system is used by the presumably heavily modified Minix 3 in the Intel management engine, however, is not known.

schnitt 4.1.3) mit der sogenannten Management Engine (ME) ab Version 11 intern das Betriebssystem Minix 3 als Firmware mit dem Namen Intel Active Management Technology (AMT) läuft. Dieser Umstand ist durchaus kritisch zu sehen, da es sich hierbei um einen Co-Prozessor mit dauerhaft laufendem Betriebssystem handelt [36]. Das System hat vollständigen Zugriff auf den Hauptspeicher und einen großen Teil der Hardware inklusive der Ethernet-Schnittstellen. Zudem kann die Management Engine von den Benutzern und Administratoren auf den allermeisten Systemen weder kontrolliert, noch entfernt werden, und läuft solange der Computer am Strom ist, also auch wenn der Hauptprozessor abgeschaltet ist [21]. Die Management Engine wird von Intel als kryptografisch signierte Binärdatei ausgeliefert und ist kaum dokumentiert. Welches Dateisystem das von Intel eingesetzte und vermutlich stark veränderte Minix 3 in der Management Engine verwendet, ist allerdings nicht bekannt.

6.2.2 ext2/3/4

ext2/3/4

The file systems ext2 (*Second Extended Filesystem*), ext3 (*Third Extended Filesystem*), and ext4 (*Fourth Extended Filesystem*) combine the file system clusters to so-called *block groups* (see Figure 6.3) of the same size. The information about the metadata and free clusters of each block group is maintained in the respective block group. One advantage of block groups is that the inodes, and thus the metadata, are physically stored close to the clusters they address. This form of data organization can be particularly beneficial when used on hard disk drives (see Section 4.4.4) because it reduces seek times and *fragmentation* [20].

Bei den Dateisystemen ext2 (*Second Extended Filesystem*), ext3 (*Third Extended Filesystem*) und ext4 (*Fourth Extended Filesystem*) werden die Cluster des Dateisystems in sogenannten *Blockgruppen* (siehe Abbildung 6.3) gleicher Größe zusammengefasst. Die Informationen über die Metadaten und freien Cluster jeder Blockgruppe werden in der jeweiligen Blockgruppe verwaltet. Ein Vorteil der Blockgruppen ist, dass die Inodes, und damit die Metadaten, physisch nahe bei den Clustern liegen, die sie adressieren. Diese Form der Datenorganisation kann besonders auf Festplatten (siehe Abschnitt 4.4.4) vorteilhaft sein, weil somit Suchzeiten und *Fragmentierung* reduziert werden [20].

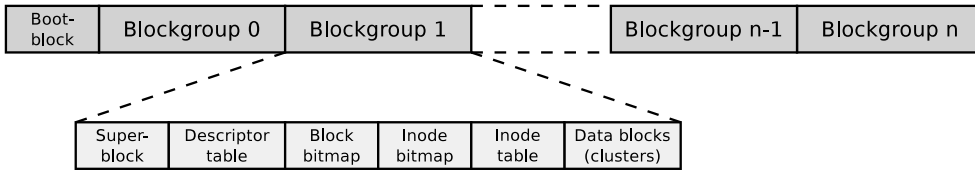

Figure 6.3: By using Block Groups, the File Systems ext2, ext3, and ext4 store the Inodes (Metadata) physically close to the Clusters, they address

With flash memory, for example, an SSD (see Section 4.4.7), the internal controller distributes the data to the existing memory cells by using a *wear-leveling* algorithm. Therefore on an SSD with the ext2/3/4 file systems, the metadata is not necessarily physically close to the clusters it addresses. However, this does not matter, because the position of the data in the individual memory cells of a flash memory is irrelevant for access time. Therefore, fragmentation in flash memories has no negative effects in general.

The maximum size of a block group is eight times the cluster size in bytes. For example, if, as shown in Figure 6.1, the cluster size of a file system is 1024 bytes, each block group can contain up to 8192 clusters. The default cluster size is 4096 bytes. In such a scenario, each block group may contain up to 32,768 clusters, and the maximum block size is 32,768 clusters * 4096 bytes cluster size = 128 MB.

Before the first block group (see Figure 6.3) the 1 kB large *boot block* is located. It contains the boot manager, which starts the operating system.

Each block group includes a *copy of the superblock* at the very beginning. It contains information about the file system, such as the number of inodes and clusters, as well as the number of free inodes and free clusters in each block group. The replicas of the superblock allow for the original superblock to be repaired in the event of an error.

Each *descriptor table* contains information about its block group. This information includes the cluster numbers of the areas *block bitmap* and *inode bitmap*, as well as the number of free clusters and free inodes in the block group. The *block bitmap* and the *inode bitmap* are each one cluster in size. These two areas indicate the occupation status of the clusters and inodes in each block group. Following the two bitmaps, each block has an inode table area that contains the inodes of the block group. The remaining clusters of each block group are used for the contents of the files.

Bei Flash-Speicher, zum Beispiel bei einer SSD (siehe Abschnitt 4.4.7), verteilt der interne Controller die Daten anhand eines *Wear-Leveling*-Algorithmus auf die vorhandenen Speicherzellen. Darum sind bei einer SSD auch mit den Dateisystemen ext2/3/4 die Metadaten nicht zwingend physisch nahe bei den Clustern, die sie adressieren. Das spielt aber keine Rolle, weil die Position der Daten in den einzelnen Speicherzellen eines Flash-Speichers für die Zugriffsgeschwindigkeit irrelevant ist und darum Fragmentierung bei Flash-Speichern generell keine negativen Auswirkungen hat.

Die maximale Größe einer Blockgruppe entspricht der achtfachen Clustergröße in Bytes. Beträgt beispielsweise so wie in Abbildung 6.1 die Clustergröße eines Dateisystems 1024 Bytes, kann jede Blockgruppe maximal 8192 Cluster umfassen. Die standardmäßige Clustergröße ist 4096 Bytes. In einem solchen Fall kann jede Blockgruppe maximal 32.768 Cluster umfassen und die maximale Blockgröße ist 32.768 Cluster * 4096 Bytes Clustergröße = 128 MB.

Vor der ersten Blockgruppe (siehe Abbildung 6.3) befindet sich der 1 kB große *Bootblock*. Dieser enthält den Bootmanager, der das Betriebssystem startet.

Jede Blockgruppe enthält zu Beginn eine *Kopie des Superblocks*. Der Superblock enthält Informationen über das Dateisystem wie zum Beispiel die Anzahl der Inodes und Cluster im Dateisystem und die Anzahl der freien Inodes und Cluster in jeder Blockgruppe. Die Kopien des Superblocks erlauben im Fehlerfall eine Reparatur des Original-Superblocks.

Jede *Deskriptor-Tabelle* enthält Informationen über die jeweilige Blockgruppe, in der sie sich befindet. Zu diesen Informationen gehören unter anderem die Clusternummern der Bereiche *Block-Bitmap* und *Inode-Bitmap*, sowie die Anzahl der freien Cluster und Inodes in der Blockgruppe. Die *Block-Bitmap* und der *Inode-Bitmap* sind jeweils einen Cluster groß. Diese beiden Bereiche enthalten die Information, welche Cluster und welche Inodes in der jeweiligen Blockgruppe belegt sind. Am Anschluss an die beiden Bitmaps folgt in jeder Blockgruppe ein Bereich mit der Inode-Tabelle, die die Inodes der Blockgruppe enthält. Die restlichen Cluster jeder Blockgruppe sind für die Inhalte der Dateien nutzbar.

As described at the beginning of Section 6.2, a directory is just a (text) file that contains the file name and the inode number of each file that is assigned to the directory.

The file system ext2 in 1992/93 replaced Minix as the standard file system of Linux and became replaced by ext3 around the year 2001. The only difference between ext2 and ext3 is that the latter has a journal (see Section 6.4) in which the write operations are collected before they are carried out. The most significant evolution of ext4 is the extent-based addressing (see Section 6.5.1).

In comparison to its successors ext3/4 and other modern file systems, ext2 seems obsolete. Nevertheless, its use on flash drives (e.g., USB flash memory drives) with a capacity of just a few gigabytes can be useful, because the lack of a journal and the low metadata overhead have a positive effect on the lifespan of memory cells [85].

Table 6.2 shows the maximum file size and file system size for clusters of different sizes when using ext2 and ext3 in Linux. For ext4 (see Section 6.5.1), the boundaries are different because of the extent-based addressing. As described in Section 6.2, the maximum file size depends on the cluster size. The same applies to the maximum file system size. Section 6.2 already explained how the maximum file size is calculated for clusters of 1 kB. The maximum file system size is calculated from the number of possible cluster numbers and the cluster size, the same way as with Minix in Section 6.2. For 32 bits large cluster numbers and 1 kB large clusters, the maximum file system size is calculated as follows:

Wie zu Beginn von Abschnitt 6.2 bereits beschrieben, ist ein Verzeichnis auch nichts anderes als eine (Text-)Datei, die als Inhalt für jede dem Verzeichnis zugewiesene Datei den Dateinamen und die zugehörige Inode-Nummer enthält.

Das Dateisystem ext2 löste 1992/93 Minix als Standard-Dateisystem unter Linux ab und wurde selbst erst um das Jahr 2001 von ext3 abgelöst. Der einzige Unterschied zwischen den beiden Dateisystemen ist, dass ext3 ein sogenanntes Journal (siehe Abschnitt 6.4) führt, in dem die Schreibzugriffe vor ihrer Ausführung gesammelt werden. Die maßgebliche Weiterentwicklung von ext4 ist die Extent-basierte Adressierung (siehe Abschnitt 6.5.1).

Gegenüber seinen Nachfolgern ext3/4 sowie gegenüber anderen, modernen Dateisystem wirkt ext2 veraltet. Dennoch kann seine Verwendung zum Beispiel auf Flash-Speichern (z.B. USB-Sticks) mit wenigen Gigabyte Kapazität sinnvoll sein, weil das Fehlen eines Journals und der geringe Aufwand für Verwaltungsinformationen sich positiv auf die Lebensdauer der Speicherzellen auswirkt [85].

Tabelle 6.2 enthält die maximale Dateigröße und Dateisystemgröße von ext2 und ext3 bei unterschiedlich großen Clustern unter Linux. Bei ext4 (siehe Abschnitt 6.5.1) gelten wegen der Extent-basierten Adressierung andere Grenzen. Wie in Abschnitt 6.2 schon beschrieben wurde, hängt die maximale Dateigröße von der Clustergröße ab. Das gleiche gilt auch für die maximale Dateisystemgröße. Dort wurde auch schon rechnerisch gezeigt, wie sich die maximale Dateigröße für 1 kB große Cluster ergibt. Die maximale Dateisystemgröße berechnet sich genau wie bei Minix in Abschnitt 6.2 aus der Anzahl der möglichen Clusternummern und der Clustergröße. Bei 32 Bits langen Clusternummern und 1 kB großen Clustern berechnet sich die maximale Dateisystemgröße wie folgt:

$$2^{32} * 1\,\text{kB}\,\text{cluster size} = 4{,}294{,}967{,}296\,\text{kB} = 4{,}194{,}304\,\text{MB} = 4096\,\text{GB} = 4\,\text{TB}$$

Table 6.2: Maximum File Size and File System Size of ext2 and ext3 for Clusters of different Sizes

Cluster size	Maximum file size	Maximum partition size
1 kB	16 GB	4 TB
2 kB	256 GB	8 TB
4 kB	2 TB	16 TB

6.3

File Systems with a File Allocation Table

The *File Allocation Table* (FAT) file system bases on the data structure of the same name. The FAT is a table of fixed size that contains a record for each cluster of the file system, indicating whether the cluster is unused, the media is damaged at this location, or a file occupies the cluster. In the latter case, the FAT record stores the address of the next cluster that belongs to that file, or it indicates the end of the file with the entry EOF (*End-Of-File*). The clusters of a file thus form a linked list in the FAT, the *cluster chain*.

Different versions of the FAT file system exist, of which FAT12, FAT16, FAT32, in particular, had significant popularity, as they had been the standard file systems of the operating system families DOS and Windows for decades. The more modern FAT versions are still standard for portable flash memory drives such as USB flash memory drives and SD cards. The number in the version naming of a FAT file system indicates the length of the cluster numbers.

Dateisysteme mit Dateizuordnungstabellen

Das Dateisystem *File Allocation Table* (FAT) basiert auf der gleichnamigen Datenstruktur, deren deutsche Bezeichnung *Dateizuordnungstabelle* ist. Die FAT ist eine Tabelle fester Größe, die für jeden Cluster des Dateisystems einen Eintrag enthält, der angibt ob der Cluster frei, das Medium an dieser Stelle beschädigt oder der Cluster von einer Datei belegt ist. Im letzten Fall speichert der FAT-Eintrag die Adresse des nächsten Clusters, der zu dieser Datei gehört oder er zeigt das Ende der Datei mit dem Eintrag EOF (*End-Of-File*) an. Die Cluster einer Datei bilden in der FAT somit eine verkettete Liste, die sogenannte *Clusterkette*.

Es existieren verschiedene Versionen des FAT-Dateisystems, wovon besonders FAT12, FAT16, FAT32 eine signifikante Verbreitung hatten, da Sie über Jahrzehnte die Standard-Dateisysteme der Betriebssystemfamilien DOS und Windows waren. Die jüngeren FAT-Versionen sind nach wie vor Standard bei mobilen Flash-Speichern wie USB-Sticks und SD-Karten. Die Zahl im Versionsnamen eines FAT-Dateisystems gibt die Länge der Clusternummern an.

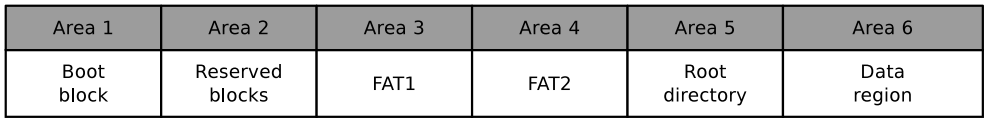

Figure 6.4: Structure of a File System that implements a File Allocation Table (FAT)

Figure 6.4 shows the structure of a file system with (at least) one FAT. The *boot sector* contains executable x86 machine code, which is supposed to start the operating system, and some information about the file system. This information includes:

Abbildung 6.4 zeigt die Struktur eines Dateisystems mit (mindestens) einer FAT. Im *Bootsektor* liegen ausführbarer x86-Maschinencode, der das Betriebssystem starten soll, und Informationen über das Dateisystem. Zu diesen Informationen gehören:

- block size of the storage device (512, 1024, 2048 or 4096 bytes)
- number of blocks per cluster
- number of blocks (sectors) on the storage device
- description (name) of the storage device
- FAT version number

Especially with magnetic, rotating storage media such as hard disk drives (see Section 4.4.4) and floppy disks, the blocks are usually called sectors. In this case, the term sector size instead of block size is more appropriate.

Between the boot block and the (first) FAT, optional *reserved sectors* may be located, e.g., for the boot manager. The FAT stores a record for each cluster in the file system, which informs whether the cluster is unused, occupied, or defective. The consistency of the FAT is essential for the functioning of the file system. Therefore, a copy of the FAT usually exists to secure a full FAT as a backup in case of data loss.

When creating a new FAT file system, it is possible to specify whether the FAT is to be stored only once or twice. In the age of floppy disks, not having a backup FAT was an easy way to (slightly) increase the storage capacity of the storage device.

The *root directory* follows after the FAT. There, every file and every directory is represented by a record.

In the file systems FAT12 and FAT16, the root directory is located directly behind the FAT and has a fixed size. The maximum number of directory entries is therefore limited. With FAT32, the root directory can reside at any position in the data area and has a variable size. The last and by far the largest area in the file system contains the actual data.

- die Blockgröße des Speichermediums (512, 1024, 2048 oder 4096 Bytes),
- die Anzahl der Blöcke pro Cluster,
- die Anzahl der Blöcke (Sektoren) auf dem Speichermedium,
- eine Beschreibung des Speichermediums und
- eine Angabe der FAT-Version.

Insbesondere bei magnetischen, rotierenden Datenträgern wie Festplatten (siehe Abschnitt 4.4.4) und Disketten werden die Blocks meist als Sektoren bezeichnet. In diesem Fall wäre der Begriff Sektorgröße anstatt Blockgröße passender.

Zwischen Bootsektor und (erster) FAT können sich optionale *reservierte Sektoren*, z.B. für den Bootmanager, befinden. In der FAT sind alle Cluster im Dateisystem, inklusive der Angabe, ob der Cluster frei, belegt oder defekt ist, erfasst. Die Konsistenz der FAT ist für die Funktionalität des Dateisystems elementar. Darum existiert üblicherweise eine Kopie der FAT, um bei Datenverlust noch eine vollständige FAT als Backup zu haben.

Bei der Anlage eines neuen FAT-Dateisystems kann angegeben werden, ob die FAT nur ein- oder zweimal vorgehalten werden soll. Im Zeitalter der Disketten war der Verzicht auf eine Kopie der FAT eine einfache Möglichkeit, um die Speicherkapazität der Datenträger (wenn auch nur geringfügig) zu erhöhen.

Im Anschluss an die FAT befindet sich das *Stammverzeichnis* (Wurzelverzeichnis). Dort ist jede Datei und jedes Verzeichnis durch einen Eintrag repräsentiert.

Bei den Dateisystemen FAT12 und FAT16 befindet sich das Stammverzeichnis direkt hinter der FAT und hat eine feste Größe. Die maximale Anzahl an Verzeichniseinträgen ist somit begrenzt. Bei FAT32 kann sich das Stammverzeichnis an einer beliebigen Position im Datenbereich befinden und hat eine variable Größe. Der letzte und mit Abstand größte Bereich im Dateisystem enthält die eigentlichen Daten.

Figure 6.5 shows the structure of the root directory. The example (for a file system with 1 kB clusters) presents a fragment with the entries of three files. In addition to the file names, the root directory contains a date entry for each file, as well as the file size, and the number of the first cluster of the file. The size of the first file `CODE.C` is 1240 bytes, and therefore it needs two clusters. The record in the root directory refers to the first cluster (`401`) of the file, and the FAT record of this first cluster contains the number of the next cluster (`402`). The second cluster of the file is also the last one. For this reason, the record for cluster 402 in the FAT has the value `EOF` (*End-Of-File*). The second file `OS.DAT` is implemented in the same way as the first one. The third file `FILE.TXT` is much smaller than a cluster, and therefore in the record of its cluster (`581`) in the FAT is the value `EOF`.

Abbildung 6.5 zeigt die Struktur des Stammverzeichnisses. Das Beispiel (für ein Dateisystem mit 1 kB großen Clustern) zeigt einen Ausschnitt mit den Einträgen von drei Dateien. Neben den Dateinamen enthält das Stammverzeichnis für jede Datei einen Datumseintrag sowie die Dateigröße und die Nummer des ersten Clusters der Datei. Die erste Datei `CODE.C` ist 1240 Bytes groß und benötigt somit zwei Cluster. Der Eintrag im Stammverzeichnisses verweist auf den ersten Cluster (`401`) der Datei und in der FAT ist im Eintrag dieses ersten Clusters die Nummer des nächsten Clusters (`402`) angegeben. Der zweite Cluster der Datei ist auch gleichzeitig der letzte Cluster. Aus diesem Grund enthält in der FAT der Eintrag für Cluster 402 den Wert `EOF` (*End-Of-File*). Die Realisierung der zweiten Datei `OS.DAT` ist analog zur ersten Datei. Die dritte Datei `FILE.TXT` ist deutlich kleiner als ein Cluster und darum steht im Eintrag ihres Clusters (`581`) in der FAT auch der Wert `EOF`.

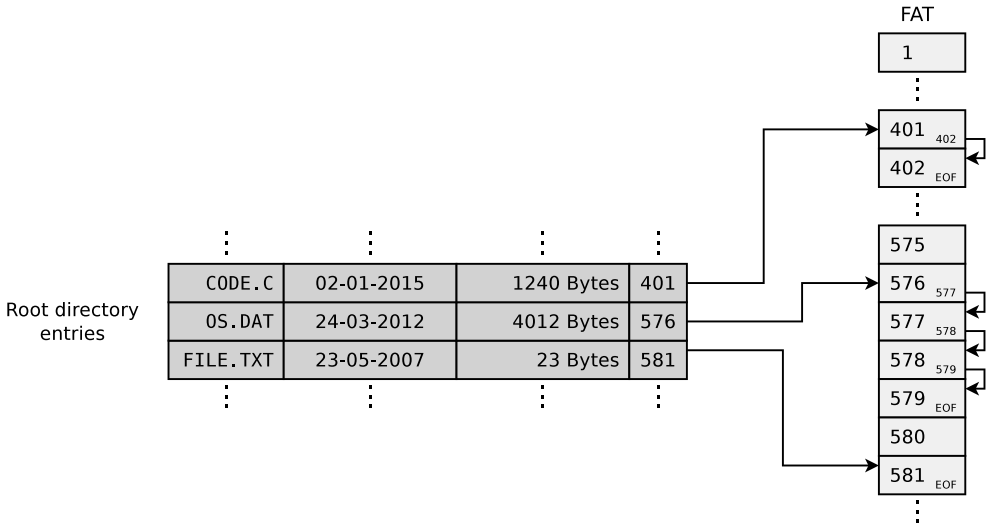

Figure 6.5: Root Directory and FAT

For the sake of simplicity, the example in Figure 6.5 lacks any fragmentation of the files (see Section 6.8). It is also very rare in practice that the clusters of files are in the correct order next to each other.

A more detailed description of the structure of root directory records is shown in Figure 6.6.

Aus Gründen der Verständlichkeit wurde beim Beispiel in Abbildung 6.5 auf eine Fragmentierung (siehe Abschnitt 6.8) der Dateien verzichtet. Dass die Cluster der Dateien in der korrekten Reihenfolge direkt nebeneinander liegen, kommt in der Praxis selten vor.

Eine detailliertere Übersicht über die Struktur der Einträge im Stammverzeichnis enthält

The naming convention for the files in schema 8.3 is obvious. Eight characters are available for the file name and three characters for the file name extension.

Abbildung 6.6. Deutlich erkennbar ist die Konvention zur Benennung der Dateien im Schema 8.3. Dabei stehen acht Zeichen für den Dateinamen und drei Zeichen für die Dateinamenserweiterung zur Verfügung.

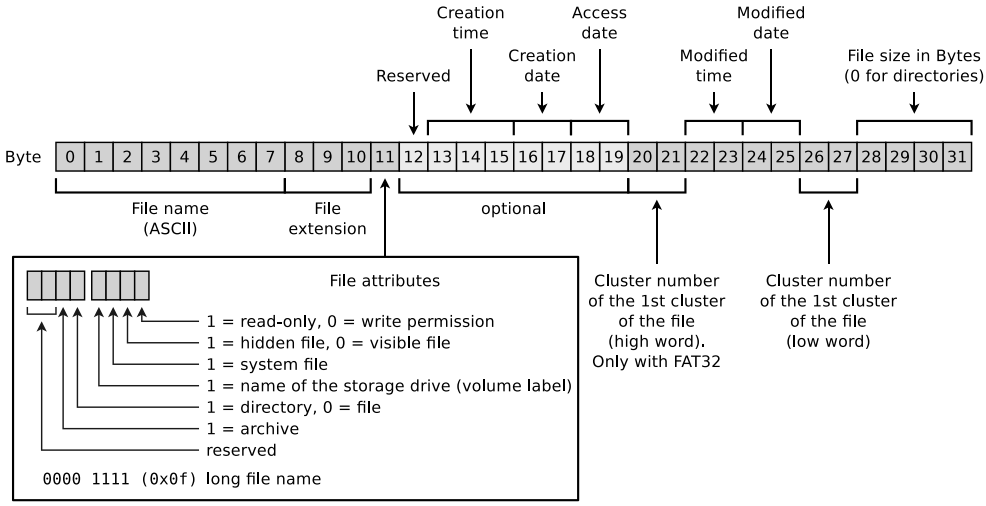

Figure 6.6: Structure of the Root Directory Records

Typical issues with FAT-based file systems are lost clusters and cross-linked clusters. Figure 6.7 shows four files in the root directory. Each record refers to the first cluster of the file. Using the FAT and the cluster numbers to the next cluster, the cluster chain with the contents of the files is determined. For the files 1 and 2 in Figure 6.7, there are no issues. Here the cluster chains from the first cluster to the cluster with the entry `EOF` are correct. The record of cluster 12 in the FAT contains an error. The value of the record is 29 and not 15, causing some clusters of file 3 to be lost. The reference to cluster 29 also harms another file, because cluster 29 is already a part of the cluster chain of file 4. In practice, a read operation on file 3 would mostly produce incorrect results. Write operations on file 3 would also, in most cases, unintentionally modify the contents of file 4. The longer the time is, the more of such lost and cross-linked clusters exist in the file system, and the higher is the risk of data loss. It is possible to use utilities such as `chkdsk` under DOS and Windows and `fsck.fat` under Linux to fix such issues.

Typische Probleme von Dateisystemen, die auf einer FAT basieren, sind verlorene Cluster (englisch: *lost clusters*) und querverbundene Cluster (englisch: *cross-linked clusters*). Abbildung 6.7 zeigt vier Dateien im Stammverzeichnis. Jeder Eintrag verweist auf den ersten Cluster der jeweiligen Datei. Mit Hilfe der FAT und den Clusternummern zum jeweils nächsten Cluster ergibt sich die Clusterkette mit den Inhalten der Dateien. Bei den Dateien 1 und 2 in Abbildung 6.7 gibt es keine Probleme. Hier sind die Clusterketten vom ersten Cluster bis zum Cluster mit dem Eintrag `EOF` korrekt. Beim Eintrag von Cluster 12 in der FAT gibt es einen Fehler. Der Wert des Eintrags ist 29 und nicht 15. Dadurch sind einige Cluster von Datei 3 nicht mehr angebunden. Der Verweis auf Cluster 29 hat zudem negative Auswirkungen auf eine weitere Datei, denn Cluster 29 ist bereits ein Teil der Clusterkette von Datei 4. Ein Lesezugriff auf Datei 3 würde in der Praxis meist zu falschen Ergebnissen führen. Schreibzugriffe auf Datei 3 würden in den meisten Fällen auch ungewollt Inhalte von Datei 4 verändern. Je länger solche verlorenen und querverbundenen Cluster im

Dateisystem existieren, umso größer ist die Gefahr eines Datenverlusts. Zur Behebung solcher Fehler existieren Dienstprogramme wie `chkdsk` unter DOS und Windows sowie `fsck.fat` unter Linux.

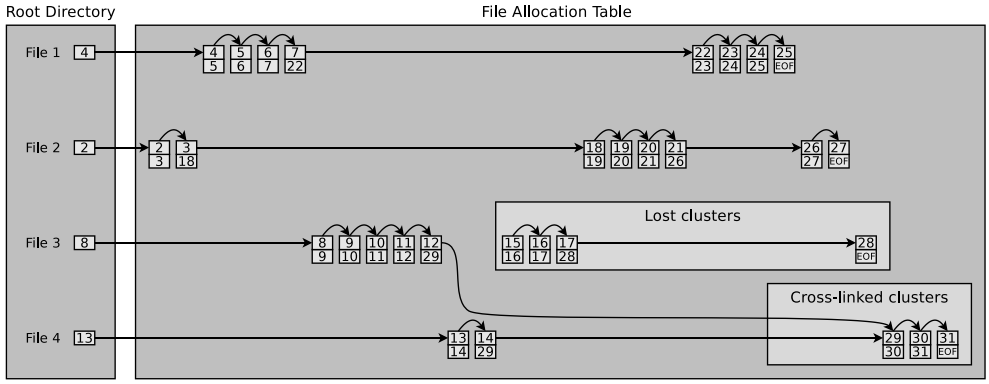

Figure 6.7: Lost and cross-linked Clusters are common Problems of File Systems with a FAT [15]

6.3.1

FAT12

In FAT12, which was released in 1980 with the operating system QDOS (Quick and Dirty Operating System), later renamed MS-DOS, the cluster numbers are 12 bits large. Therefore, a maximum of $2^{12} = 4096$ clusters can be addressed. Possible cluster sizes are 512 bytes, 1 kB, 2 kB, and 4 kB. The maximum file system size is 16 MB.

For 12 bits large cluster numbers and 4 kB large clusters, the maximum file system size is calculated as follows: 2^{12} cluster numbers $\times$ 4 kB cluster size = 16,384 kB = 16 MB.

FAT12

Bei FAT12, das 1980 mit dem Betriebssystem QDOS (Quick and Dirty Operating System), später umbenannt in MS-DOS, erschienen ist, sind die Clusternummern 12 Bits lang. Damit können maximal $2^{12} = 4096$ Cluster adressiert werden. Mögliche Clustergrößen sind 512 Bytes, 1 kB, 2 kB und 4 kB. Die maximale Dateisystemgröße beträgt 16 MB.

Bei 12 Bits langen Clusternummern und 4 kB großen Clustern berechnet sich die maximale Dateisystemgröße wie folgt: 2^{12} Clusternummern $*$ 4 kB Clustergröße = 16.384 kB = 16 MB.

6.3.2

FAT16

Due to the limitations of FAT12, especially concerning the maximum file system size of no more than 16 MB, FAT16 was released in 1983, with which $2^{16} = 65{,}536$ clusters per file system can be addressed. Because 12 clusters are reserved, there are only 65,524 clusters available [18]. The

FAT16

Aufgrund der Einschränkungen von FAT12, besonders im Hinblick auf die maximale Dateisystemgröße von nur 16 MB, erschien schon 1983 FAT16, mit dem $2^{16} = 65.536$ Cluster pro Dateisystem adressiert werden können. Weil 12 Cluster reserviert sind, sind effektiv nur 65.524 Clus-

clusters can be between 512 bytes and 256 kB in size.

Table 6.3 shows the default cluster sizes of Windows 2000/XP/Vista/7/8/10 with different file system dimensions.

Oddly, not all operating systems support all cluster sizes of FAT16. MS-DOS and Windows 95/98/Me, for example, do not support clusters > 32 kB. Therefore the maximum file system size for these operating systems is limited to 2 GB. Windows 2000/XP/7/8/10, for example, do not support clusters > 64 kB. For these operating systems, the maximum file system size is therefore limited to 4 GB. 128 kB and 256 kB large clusters are only supported by Windows NT 4.0 on storage devices with a block size (sector size) > 512 bytes [76].

The cluster size can be specified when creating the file system. It allows creating large clusters, even on small partitions. In practice, however, this rarely makes sense. The primary application of FAT16 is mobile storage devices like USB-flash memory drives or partitions on such storage devices with a capacity of up to 2 GB. For larger file systems, it is not practical to use FAT16 because of the memory loss when using large clusters due to internal fragmentation.

ter verfügbar [18]. Die Cluster können zwischen 512 Bytes und 256 kB groß sein.

Tabelle 6.3 enthält die Standard-Clustergrößen unter Windows 2000/XP/Vista/7/8/10 bei unterschiedlich großen Dateisystemgrößen.

Kurioserweise unterstützten auch nicht alle Betriebssysteme alle Clustergrößen bei FAT16. MS-DOS und Windows 95/98/Me beispielsweise unterstützen keine Cluster > 32 kB. Damit ist auch die maximale Dateisystemgröße bei diesen Betriebssystemen auf 2 GB beschränkt. Windows 2000/XP/7/8/10 beispielsweise unterstützen keine Cluster > 64 kB. Bei diesen Betriebssystemen ist somit die maximale Dateisystemgröße auf 4 GB beschränkt. 128 kB und 256 kB große Cluster unterstützt ausschließlich Windows NT 4.0 auf Datenspeichern mit einer Blockgröße (Sektorgröße) > 512 Bytes [76].

Die Clustergröße kann beim Erzeugen des Dateisystems festgelegt werden. Somit können auch auf kleinen Partitionen große Cluster realisiert werden. In der Praxis ist das aber nur selten sinnvoll. Das primäre Einsatzgebiet von FAT16 sind mobile Datenträger wie USB-Sticks oder Partitionen auf diesen Datenträgern mit einer Kapazität von bis zu 2 GB. Bei größeren Dateisystemen ist ein Einsatz wegen des Speicherverlustes bei großen Clustern durch interne Fragmentierung nicht sinnvoll.

Table 6.3: Maximum File System Size of FAT16 for Clusters of different Size

Partition size	Cluster size
up to 31 MB	512 bytes
32 MB - 63 MB	1 kB
64 MB - 127 MB	2 kB
128 MB - 255 MB	4 kB
256 MB - 511 MB	8 kB
512 MB - 1 GB	16 kB
1 GB - 2 GB	32 kB
2 GB - 4 GB	64 kB
4 GB - 8 GB	128 kB
8 GB - 16 GB	256 kB

6.3.3

FAT32

Because of growing hard disk drive capacities, and because clusters ≥ 32 kB waste storage resources, the FAT32 file system was released in 1997. The cluster numbers of this file system are 32 bits long. However, since 4 bits are reserved, only $2^{28} = 268{,}435{,}456$ clusters can be addressed. The cluster size can have values between 512 bytes and 32 kB. Table 6.4 shows the standard cluster sizes of Windows 2000/XP/Vista/7/8/10 for different file system dimensions. As with other FAT file systems, the cluster size can be specified when creating the file system.

The maximum file size in FAT32 is 4 GB. The reason for this is that only 4 bytes are available to indicate the file size (see Figure 6.6). FAT32 is mostly used with mobile storage devices such as USB flash memory drives or on partitions of such storage devices with a capacity > 2 GB.

FAT32

Als Reaktion auf steigende Festplattenkapazitäten und weil Cluster ≥ 32 kB viel Speicher verschwenden, erschien 1997 das Dateisystem FAT32. Bei diesem sind die Clusternummern 32 Bits lang. Da allerdings 4 Bits reserviert sind, können nur $2^{28} = 268.435.456$ Cluster adressiert werden. Die Cluster können zwischen 512 Bytes und 32 kB groß sein. Tabelle 6.4 enthält die Standard-Clustergrößen unter Windows 2000/XP/Vista/7/8/10 bei unterschiedlich großen Dateisystemgrößen. Genau wie bei den anderen FAT-Dateisystemen kann die Clustergröße beim Erzeugen des Dateisystems festgelegt werden.

Die maximale Dateigröße unter FAT32 ist 4 GB. Der Grund dafür ist, dass nur 4 Bytes für die Angabe der Dateigröße zur Verfügung stehen (siehe Abbildung 6.6). Das primäre Einsatzgebiet von FAT32 sind mobile Datenträger wie USB-Sticks oder Partitionen darauf mit einer Kapazität > 2 GB.

Table 6.4: Default Cluster Size of FAT32 for different Partition Sizes

Partition size	Cluster size
up to 63 MB	512 bytes
64 MB - 127 MB	1 kB
128 MB - 255 MB	2 kB
256 MB - 8 GB	4 kB
8 GB - 16 GB	8 kB
16 GB - 32 GB	16 kB
32 GB - 2 TB	32 kB

6.3.4

VFAT

The *Virtual File Allocation Table* (VFAT) is a file system extension for FAT12/16/32 that allows longer file names. It was released in 1997. Because of this extension, file names that do not follow the schema 8.3 became supported under Windows for the first time. With VFAT, file names can have a length of up to 255 characters, and Unicode (see Section 2.5) is used as character encoding method.

VFAT

Eine 1997 erschienene Erweiterung für die Dateisysteme FAT12/16/32, die längere Dateinamen ermöglicht, ist *Virtual File Allocation Table* (VFAT). Damit wurden unter Windows erstmals Dateinamen unterstützt, die nicht dem Schema 8.3 folgen. Mit VFAT können Dateinamen bis zu einer Länge von 255 Zeichen realisiert werden. Die Kodierung der Zeichen erfolgt via Unicode (siehe Abschnitt 2.5).

The VFAT extension is an interesting example of the implementation of new functionality without losing backward compatibility. VFAT distributes long file names to a maximum of 20 pseudo-directory records. The first four bits in the *file attributes* field of a VFAT record in the FAT have the value 1 (see Figure 6.6). One special feature is that upper/lower case characters (case-sensitive) are displayed. Still, characters are treated case-insensitive, just like the traditional DOS and Windows operating system families that operate case-insensitively.

VFAT stores a unique file name in the 8.3 format for each file. Operating systems without the VFAT extension ignore the pseudo-directory entries and only show the shortened file name. This way, Microsoft operating systems without VFAT can access files with long file names. However, the short file names must be unique. This condition is satisfied by how the shortened file names are created. All special characters and dots inside the file name get erased, all lowercase letters get converted to uppercase letters, and only the first six letters are used. Next follows the string `~1` before the dot. The first three characters after the dot are kept, and the rest is erased. If a file with the same name already exists, `~1` becomes `~2`, etc. For example, the file with the filename `A very long file name.test.pdf` is displayed in MS-DOS as `AVERYL~1.pdf` This is the case if there is no file with the same shortened file name in the file system. If such a file exists, the number is incremented.

VFAT ist ein interessantes Beispiel für die Realisierung einer neuen Funktionalität unter Beibehaltung der Abwärtskompatibilität. Lange Dateinamen verteilt VFAT auf maximal 20 Pseudo-Verzeichniseinträge. Bei einem VFAT-Eintrag in der FAT, haben die ersten vier Bits im Feld *Dateiattribute* den Wert 1 (siehe Abbildung 6.6). Eine Besonderheit ist, dass Groß-/Kleinschreibung zwar angezeigt, aber wie traditionell unter den Betriebssystemfamilien DOS und Windows üblich, ignoriert wird.

VFAT speichert für jede Datei einen eindeutigen Dateinamen im Format 8.3. Betriebssysteme ohne die VFAT-Erweiterung ignorieren die Pseudo-Verzeichniseinträge und zeigen nur den verkürzten Dateinamen an. Dadurch können Microsoft-Betriebssysteme ohne VFAT auf Dateien mit langen Dateinamen zugreifen. Allerdings müssen die kurzen Dateinamen eindeutig sein. Dies wird durch die Art und Weise der Erzeugung der verkürzten Dateinamen gewährleistet. Dafür werden alle Sonderzeichen und Punkte innerhalb des Dateinamens gelöscht, alle Kleinbuchstaben werden in Großbuchstaben umgewandelt und es werden nur die ersten sechs Buchstaben beibehalten. Danach folgt die Zeichenkette `~1` vor dem Punkt. Die ersten drei Zeichen hinter dem Punkt werden beibehalten und der Rest gelöscht. Existiert schon eine Datei gleichen Namens, wird `~1` zu `~2`, usw. So wird beispielsweise die Datei mit dem Dateinamen `Ein ganz langer Dateiname.test.pdf` unter MS-DOS als `EINGAN~1.pdf` dargestellt. Dies ist zumindest dann der Fall, wenn noch keine Dateimit dem gleichen verkürzten Dateinamen im Dateisystem existiert. Wenn doch, wird entsprechend hochgezählt.

6.3.5 exFAT

The latest and most up-to-date file system based on the FAT concept is the *Extended File Allocation Table* (exFAT) released in 2006. Microsoft designed it particularly for use on flash storage media. For this reason, it does not implement a journal (see Section 6.4). A journal would increase the number of write accesses, thus shortening the flash memory lifetime.

exFAT

Das jüngste und modernste auf dem Konzept der FAT basierende Dateisystem ist das 2006 erschienene *Extended File Allocation Table* (exFAT). Dieses wurde von der Firma Microsoft speziell für den Einsatz auf Flash-Speichermedien entwickelt. Aus diesem Grund verwendet es kein Journal (siehe Abschnitt 6.4). Ein Journal würde die Anzahl der Schreibzugriffe erhöhen, was die Lebensdauer von Flashspeicher verkürzt.

As in FAT32, the cluster numbers in exFAT are 32 bits long. However, because unlike FAT32, no bits are reserved, up to 2^{32} = 4,294,967,296 clusters can be addressed. The clusters can be between 512 bytes and 32 MB in size. Table 6.5 shows the default cluster sizes in Windows XP/Vista/7/8/10 for different file system sizes. The cluster size can be specified when the file system is created. The maximum file size for exFAT is 16 EB (2^{64} bytes).

Wie bei FAT32 sind auch in exFAT die Clusternummern 32 Bits lang. Da aber im Gegensatz zu FAT32 keine reservierten Bits enthalten sind, können auch bis zu 2^{32} = 4.294.967.296 Cluster adressiert werden. Die Cluster können zwischen 512 Bytes und 32 MB groß sein. Tabelle 6.5 enthält die Standard-Clustergrößen unter Windows XP/Vista/7/8/10 bei unterschiedlich großen Dateisystemgrößen. Die Clustergröße kann beim Erzeugen des Dateisystems festgelegt werden. Die maximale Dateigröße bei exFAT ist 16 EB (2^{64} Bytes).

Table 6.5: Default Cluster Size of exFAT for different Partition Sizes [82]

Partition size	Cluster size
up to 256 MB	4 kB
256 MB - 32 GB	32 kB
32 GB - 256 GB	128 kB

The exFAT file system lacks numerous capabilities of modern file systems, such as integrated compression and encryption, software RAID, and quotas. To improve data security, however, the file system includes automatic error correction of metadata with checksums [79].

Zahlreiche Fähigkeiten moderner Dateisysteme wie z.B. integrierte Kompression und Verschlüsselung, Software-RAID und Kontingente (*Quotas*) fehlen bei exFAT. Zur Verbesserung der Datensicherheit enthält das Dateisystem aber eine automatische Fehlerkorrektur der Metadaten mit Prüfsummen [79].

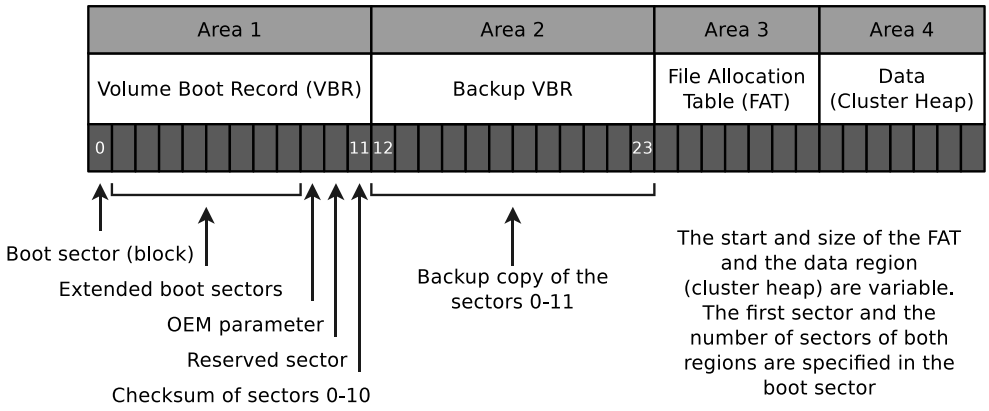

Figure 6.8: Structure of the exFAT File System

Figure 6.8 shows the structure of an exFAT file system. The first area is called *Volume Boot Record* (VBR) and comprises 12 blocks on the disk, containing the most relevant metadata of the file system. It includes the boot sector index

Abbildung 6.8 zeigt die Struktur eines exFAT-Dateisystems. Der erste Bereich, der sogenannte *Volume Boot Record* (VBR), der 12 Blöcke auf dem Datenträger umfasst, enthält die wichtigsten Metadaten des Dateisystems. Dazu gehören

and a checksum over the remaining 11 blocks of the VBR.

The boot sector contains executable x86 machine code used to start the operating system and some information about the file system. This information includes:

- the position of the first block (sector) of the FAT and its size (the number of sectors),
- the position of the first block of the data area (*Cluster Heap*) and its size, and
- the position of the first block of the root directory.

A backup copy of the VBR follows the VBR. Next comes the FAT and the data area (cluster heap). The cluster numbers of the first cluster of the FAT and the data area and their size (the number of sectors) are specified in the boot sector [78].

In contrast to the other FAT file system revisions, the root directory has no fixed position in exFAT. Instead, it is located within the data area and is usually not stored in one piece but fragmented [106].

The fact that exFAT is significantly less popular than FAT32 with VFAT is due, among other things, to the point that the source code was not made available until 2013. In addition, only in 2019 declared the vendor that the patent becomes free of royalties. Before this happened, the patent situation for free (re-)implementations and their use was partly unclear.

This file system has been supported in current Windows releases since Windows 7, and there are updates for some older Windows versions (e.g., XP and Vista). Mac OS X includes exFAT file system drivers since release 10.6.4 and the Linux kernel since release 5.4.

der Bootsektor und eine Prüfsumme über die übrigen 11 Blöcke des VBR.

Im Bootsektor liegen ausführbarer x86-Maschinencode, der das Betriebssystem starten soll, und Informationen über das Dateisystem. Zu diesen Informationen gehören:

- die Position des ersten Blocks (Sektors) der FAT und deren Größe (Anzahl der Sektoren),
- die Position des ersten Blocks des Datenbereichs (*Cluster Heap*) und dessen Größe und
- die Position des ersten Blocks des Stammverzeichnisses.

Anschließend an den VBR folgt eine Sicherheitskopie des VBR. Darauf folgen die FAT und der Datenbereich (Cluster Heap). Die Clusternummern des ersten Clusters der FAT und der Datenbereichs sowie deren Größe (Anzahl der Sektoren) sind im Bootsektor definiert [78].

Das Stammverzeichnis (Wurzelverzeichnis) hat im Gegensatz zu den übrigen FAT-Dateisystemversionen keine feste Position. Es befindet sich innerhalb des Datenbereichs und liegt dort üblicherweise nicht am Stück vor, sondern fragmentiert [106].

Das exFAT verglichen mit FAT32 mit VFAT eine deutlich geringere Verbreitung hat, liegt u.a. daran, dass der Quellcode bis 2013 nicht vorlag. Zudem erfolgte die vollständige patentrechtliche Freigabe durch den Hersteller erst 2019. Bis zu diesem Zeitpunkt waren die patentrechtliche Situation für freie (Re-)Implementierungen und deren Einsatz in Teilen ungeklärt.

Unterstützung für dieses Dateisystem ist Bestandteil von modernen Windows-Versionen seit Windows 7. Für einige ältere Windows-Versionen (u.a. XP und Vista) existieren offizielle Updates. In Mac OS X sind exFAT-Dateisystemtreiber seit Version 10.6.4 und im Linux-Kernel seit Version 5.4 enthalten.

6.4

Journaling File Systems

If files (or directories – they are just files, too) are created, relocated (moved), renamed, erased, or modified, write operations in the file system are carried out. Write operations shall convert data from a consistent state to another consistent state. If a failure occurs during a write operation, the consistency of the file system must be checked. If a file system has a size of several gigabytes, the consistency check can last several hours or days. Skipping the consistency check is not a reasonable alternative because this increases the risk of data loss.

To narrow down the data to be checked during the consistency check, modern file systems implement a *journal* that collects the write operations before carrying them out. Consequently, such file systems are called *journaling file systems*. The advantage of such file systems is that after a crash, only those files (clusters) and metadata must be checked, for which a record exists in the journal. The drawback is additional write operations. A journal increases the number of write operations because modifications are written into the journal before they are carried out.

Various journaling concepts exist. *Metadata journaling* works according to the cache write strategy *write-back* (see Section 4.4.2). The file system only collects metadata (inode) modifications in the journal. Modifications to the clusters of the files are carried out by the Linux kernel at first only in the *page cache* (see Section 6.7) in the main memory. After a certain time or a manual request by the user, the system call `sync` commits the modifications in the page cache to the storage device. One advantage of this journaling concept is that consistency checks only take a few seconds. One drawback is that only the consistency of the metadata is ensured after a crash. Data loss due to a system crash is still possible. This journaling concept is optional for the file systems ext3/4. NTFS

Journaling-Dateisysteme

Sollen Dateien (oder Verzeichnisse, die ja auch nichts anderes sind als Dateien) erstellt, verschoben, umbenannt, gelöscht oder einfach verändert werden, sind Schreibzugriffe im Dateisystem nötig. Schreibzugriffe sollen Daten von einem konsistenten Zustand in einen neuen konsistenten Zustand überführen. Kommt es während eines Schreibzugriffs zum Ausfall, muss die Konsistenz des Dateisystems überprüft werden. Ist ein Dateisystem mehrere Gigabyte groß, kann die Konsistenzprüfung mehrere Stunden oder Tage dauern. Die Konsistenzprüfung zu überspringen ist keine sinnvolle Alternative, weil so das Risiko von Datenverlust steigt.

Um bei der Konsistenzprüfung die zu überprüfenden Daten einzugrenzen, führen moderne Dateisysteme ein sogenanntes *Journal* (in der deutschsprachigen Literatur heißt es manchmal *Logbuch* [6]), in dem die Schreibzugriffe vor ihrer Durchführung gesammelt werden. Konsequenterweise heißen solche Dateisysteme auch *Journaling-Dateisysteme*. Der Vorteil, dass nach einem Absturz nur diejenigen Dateien (Cluster) und Metadaten überprüft werden müssen, die im Journal stehen, wird durch den Nachteil zusätzlicher Schreibzugriffe erkauft. Ein Journal erhöht die Anzahl der Schreibzugriffe, weil Änderungen erst ins Journal geschrieben und danach durchgeführt werden.

Es existieren verschiedene Journaling-Konzepte. Das *Metadaten-Journaling* arbeitet nach der Cache-Schreibstrategie *Write-Back* (siehe Abschnitt 4.4.2). Das Dateisystem erfasst im Journal nur Änderungen an den Metadaten (Inodes). Änderungen an den Clustern der Dateien führt der Linux-Kernel erst nur im *Page Cache* (siehe Abschnitt 6.7) im Hauptspeicher durch. Nach einer bestimmten Zeit oder nach einer manuellen Anweisung durch den Benutzer überträgt der Systemaufruf `sync` die Änderungen im Page Cache auf den Datenspeicher. Vorteilhaft bei dieser Form des Journalings ist, dass eine Konsistenzprüfung nur wenige Sekunden dauert. Nachteilig ist, dass nur die Konsistenz der Metadaten nach einem Absturz garantiert ist. Datenverlust durch einen Systemabsturz ist weiterhin mög-

(see Section 6.5.2) and XFS only implement metadata journaling.

Using *full journaling* means that the file system collects all modifications to the metadata and clusters of files in the journal. In contrast to metadata journaling, this also ensures the consistency of the files. The drawback of this method is that all write operations are carried out twice. Also, this journaling concept is optional for the file systems ext3/4.

Most Linux file systems, by default, use a compromise between the two journaling concepts discussed, namely *ordered-journaling*. Thereby the file system collects only metadata modifications in the journal. File modifications are carried out in the file system first, and next, the relevant metadata modifications are written into the journal. One advantage of this concept is that, as with metadata journaling, consistency checks only take a few seconds. Furthermore, write speed is similar to that of metadata journaling. However, even with this concept, only the consistency of the metadata is ensured. During a system crash, as long as incomplete transactions are contained in the journal, new files and attachments to existing files get lost because, the clusters are not yet allocated to the inodes. A write operation that intends to overwrite a file may result in inconsistent content after a crash, and the file probably cannot be repaired because no backup of the original version was made. Despite these compromises in terms of protection against data loss, ordered journaling is the default concept for ext3/4 file systems, and it is the only journaling concept that is supported by JFS.

lich. Diese Form des Journalings ist optional bei den Dateisystemen ext3/4. NTFS (siehe Abschnitt 6.5.2) und XFS bieten ausschließlich Metadaten-Journaling.

Beim *vollständigen Journaling* erfasst das Dateisystem alle Änderungen an den Metadaten und alle Änderungen an den Clustern der Dateien im Journal. Dadurch ist im Gegensatz zum Metadaten-Journaling auch die Konsistenz der Dateien garantiert. Nachteilig ist, dass alle Schreibzugriffe doppelt ausgeführt werden. Auch diese Form des Journalings ist optional bei den Dateisystemen ext3/4.

Die meisten Linux-Dateisysteme verwenden standardmäßig einen Kompromiss aus den beiden bislang besprochenen Journaling-Konzepten, nämlich das *Ordered-Journaling*. Dabei erfasst das Dateisystem im Journal nur Änderungen an den Metadaten. Änderungen an den Clustern von Dateien werden erst im Dateisystem durchgeführt und danach die Änderungen an den betreffenden Metadaten ins Journal geschrieben. Ein Vorteil dieses Konzepts ist, dass so wie beim Metadaten-Journaling die Konsistenzprüfungen nur wenige Sekunden dauert. Zudem werden ähnlich hohe Schreibgeschwindigkeit wie beim Metadaten-Journaling erreicht. Allerdings ist auch bei diesem Konzept nur die Konsistenz der Metadaten garantiert. Beim Systemabsturz mit nicht abgeschlossenen Transaktionen im Journal sind neue Dateien und Dateianhänge verloren, da die Cluster noch nicht den Inodes zugeordnet sind. Ein Schreibvorgang, der darauf abzielt, eine Datei zu überschreiben, führt nach einem Absturz möglicherweise zu inkonsistentem Inhalt, und die Datei kann wahrscheinlich nicht repariert werden, da die ursprüngliche Version nicht gesichert wurde. Trotz dieser Kompromisse bei der Datensicherheit ist Ordered-Journaling das standardmäßig verwendete Konzept bei den Dateisystemen ext3/4 und es ist das einzige von JFS unterstützte Journaling-Konzept.

6.5

Extent-based Addressing

Extent-basierte Adressierung

One issue with file systems that operate according to the block addressing concept (see

Ein Problem von Dateisystemen, die nach dem Adressierungsschema der Blockadressierung (sie-

Section 6.2) is overhead. This issue worsens as the storage capacity of storage devices increases.

Since each inode addresses only a few cluster numbers directly when block addressing is used, large files occupy numerous additional clusters for indirect addressing. Figure 6.9 shows the problem of block addressing using ext3 as an example, which is that each inode in this file system (with 4 kB large clusters) is able to address a maximum of 48 kB of data directly.

he Abschnitt 6.2) arbeiten, ist der Aufwand für die Verwaltungsinformationen. Dieses Problem verschärft sich durch die steigende Speicherkapazität der Datenspeicher.

Da jeder Inode bei Blockadressierung nur eine sehr geringe Anzahl Clusternummern direkt adressiert, belegen große Dateien zahlreiche zusätzliche Cluster zur indirekten Adressierung. Abbildung 6.9 zeigt am Beispiel von ext3 das Problem der Blockadressierung, nämlich dass jeder Inode bei diesem Dateisystem (mit 4 kB großen Clustern) maximal 48 kB direkt adressieren kann.

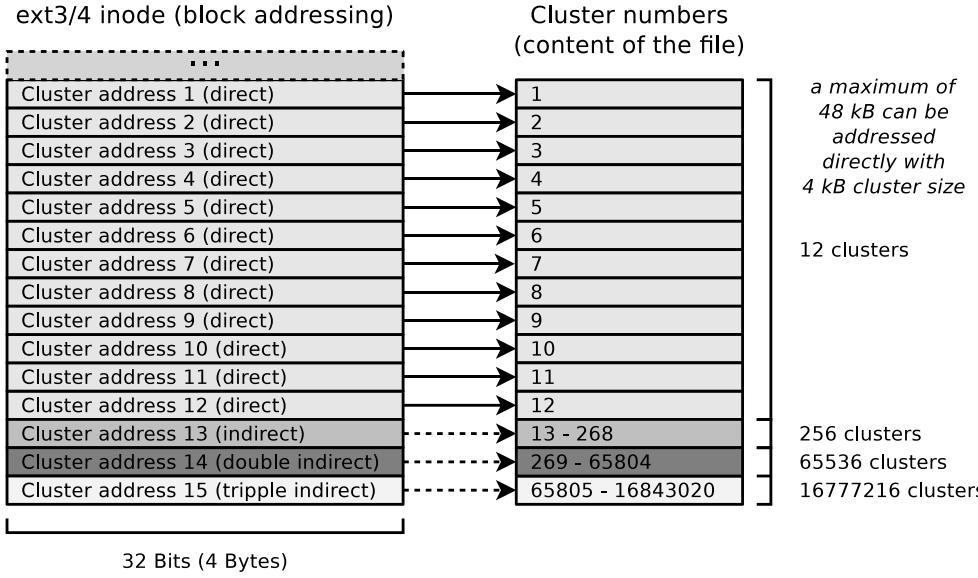

Figure 6.9: Block Addressing by Example of ext3. Each Inode can directly address 48 kB

One solution to reduce the growing overhead for addressing, despite the rising file and file system dimensions, is using *extents*.

With extent-based addressing, the inodes do not address individual clusters. Instead, the inodes map areas of files that are as large as possible to areas (the so-called extents) on the storage device in one piece (see Figure 6.10). Instead of many individual cluster numbers, this form of addressing requires only three values:

- The first cluster number of the area (extent) in the file

Eine Lösung, um den zunehmendem Verwaltungsaufwand für die Adressierung trotz steigender Datei- und Dateisystemgrößen zu reduzieren, sind *Extents*.

Bei Extent-basierter Adressierung adressieren die Inodes nicht einzelne Cluster. Stattdessen bilden sie möglichst große Dateibereiche auf zusammenhängende Bereiche (die sogenannten Extents) auf dem Datenspeicher ab (siehe Abbildung 6.10). Statt vieler einzelner Clusternummern erfordert diese Form der Adressierung nur drei Werte:

- Die erste Clusternummer des Bereichs (Extents) in der Datei

- The size of the area in the file (in clusters)
- The number of the first cluster on the storage device

- Die Größe des Bereichs in der Datei (in Clustern)
- Die Nummer des ersten Clusters auf dem Speichergerät

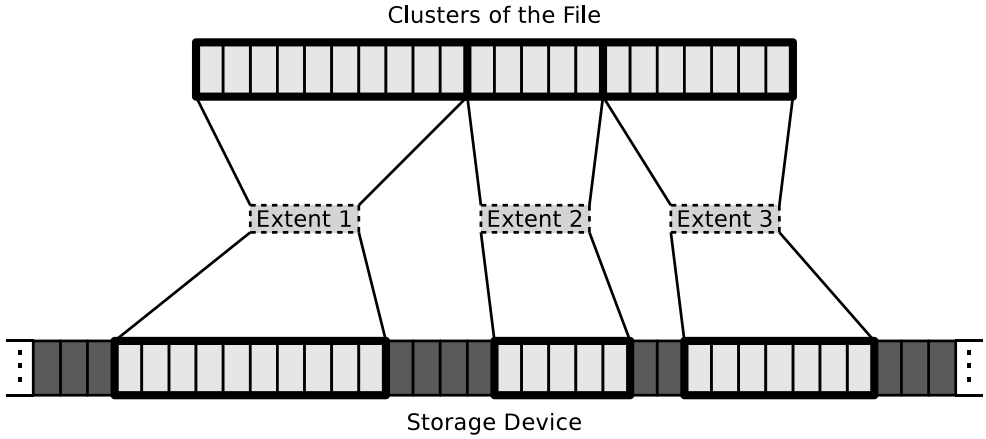

Figure 6.10: Using Extents, the Inodes do not address individual Clusters, but instead Areas on the Storage Device

Examples of file systems that address the cluster of files via extents are JFS, XFS, Btrfs, NTFS, and ext4. Due to the limited space in this book, the focus in the following sections will be on ext4 and NTFS.

Beispiele für Dateisysteme, die via Extents die Cluster der Dateien adressieren, sind JFS, XFS, Btrfs, NTFS und ext4. Aus Platzgründen geht dieses Buch in den folgenden Abschnitten nur auf ext4 und NTFS ein.

6.5.1

ext4

The file system ext4 (*fourth extended filesystem*) is a journaling file system like ext3. Because ext4 has 48 bits large cluster numbers (in ext3, the cluster numbers are 32 bits large), it supports much larger file systems than its predecessor ext3. Since 2008 it is marked as a stable part of the Linux kernel. The most significant difference to ext3 is extent-based addressing. Figure 6.11 illustrates the reduced overhead for large files with the example of ext4.

With block addressing in ext2/3 (see Figure 6.9), each inode has 15 fields of four bytes each, i.e., a total of 60 bytes, available

ext4

Das Dateisystem ext4 (*Fourth Extended Filesystem*) ist wie ext3 ein Journaling-Dateisystem. Da bei ext4 die Länge der Clusternummern auf 48 Bits vergrößert wurde, kann ext4 deutlich größere Dateisysteme als der Vorgänger ext3 verwalten. Seit dem Jahr 2008 ist es ein als stabil gekennzeichneter Teil des Linux-Betriebssystemkerns. Der bedeutendste Unterschied zu ext3 ist die Adressierung mit Extents. Abbildung 6.11 zeigt am Beispiel von ext4 anschaulich den reduzieren Verwaltungsaufwand bei großen Dateien.

Bei Blockadressierung mit ext2/3 (siehe Abbildung 6.9) sind in jedem Inode 15 je vier Bytes große Felder, also insgesamt 60 Bytes, zur Adres-

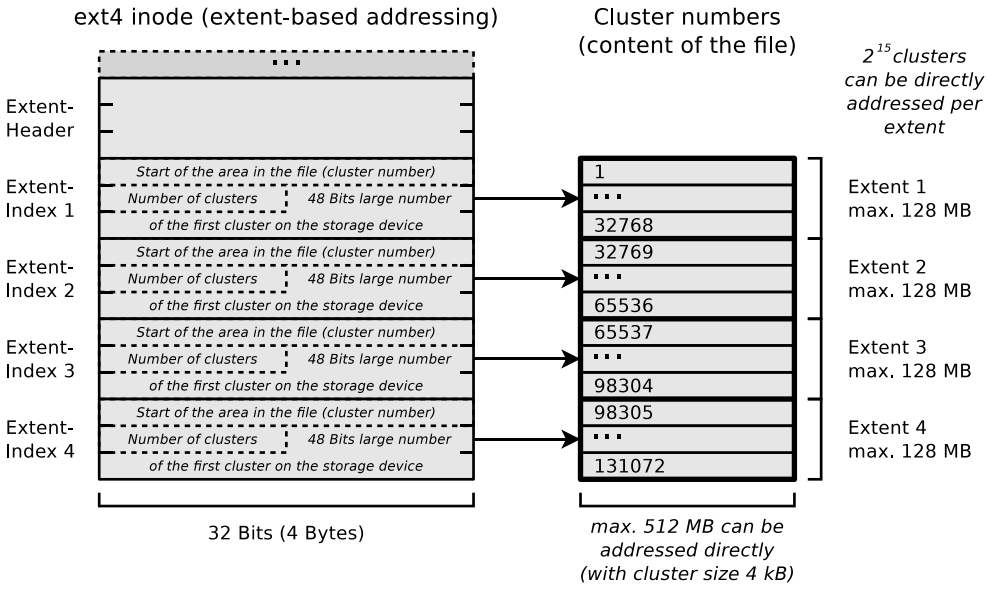

Figure 6.11: Extent-based addressing demonstrated by the Example of ext4. Each Inode can address a maximum of 512 MB directly

for addressing clusters. By contrast, ext4 (see Figure 6.11) uses these 60 bytes for one extent header (12 bytes) and for addressing four extents (12 bytes each).

For every extent, in the inode of the file...

- in a 4 bytes long data field, the first cluster number of the area in the file is stored,
- in a 2 bytes long data field, the number of clusters of the extent, and
- in a 6 bytes long data field, the first cluster number on the storage device.

Extents cannot be larger than 128 MB (2^{15} bits) because ext4, just like its predecessors ext2 and ext3, organizes the file system's clusters into so-called block groups (see Section 6.2.2) which have a maximum size of 128 MB. The remaining sixteenth bit (the most significant bit) in the data field specifying

sierung von Clustern verfügbar. Im Gegensatz dazu verwendet ext4 (siehe Abbildung 6.11) diese 60 Bytes für einen Extent-Header (12 Bytes) und zur Adressierung von vier Extents (jeweils 12 Bytes).

Für jeden Extent wird im Inode der Datei...

- in einem 4 Bytes langen Datenfeld die erste Clusternummer des Bereichs in der Datei gespeichert,
- in einem 2 Bytes langen Datenfeld die Anzahl der Cluster des Bereichs, und
- in einem 6 Bytes langen Datenfeld die Nummer des ersten Clusters auf dem Speichergerät.

Extents können nicht größer als 128 MB (2^{15} Bits) sein, weil ext4, genau wie seine Vorgänger ext2 und ext3, die Cluster des Dateisystems in sogenannten Blockgruppen (siehe Abschnitt 6.2.2) mit einer maximalen Größe von 128 MB organisiert. Das übrig gebliebene sechzehnte Bit (es ist das Bit mit dem höchs-

the number of clusters of the extent indicates whether the extent already has data written to it. With this information, the file system can preallocate storage capacity using the so-called *persistent preallocation* to speed up subsequent writes as well as to eliminate future storage capacity bottlenecks for individual applications [38, 72].

With four extents, an ext4 inode can directly address 512 MB. If a file is larger than 512 MB, ext4 creates a tree of extents. The functional principle is analogous to indirect block addressing.

ten Stellenwert) im Datenfeld das die Anzahl der Cluster des Extents definiert, gibt an, ob der Extent bereits mit Daten beschrieben ist. Mit dieser Information kann das Dateisystem mit Hilfe der sogenannten *Persistent Preallocation* Speicherkapazität vorbelegen, um spätere Schreibzugriffe zu beschleunigen sowie um zukünftige Engpässe bzgl. Speicherkapazität für einzelne Anwendungen auszuschließen [38, 72].

Mit vier Extents kann ein ext4-Inode 512 MB direkt adressieren. Ist eine Datei größer als 512 MB, realisiert ext4 einen Baum aus Extents. Das Funktionsprinzip ist analog zur indirekten Blockadressierung.

6.5.2 NTFS

In the early 1990s, Microsoft started developing the *New Technology File System* (NTFS) as a successor to the FAT file systems for the Windows NT operating system family. Since that time, the following versions of NTFS have been released:

- NTFS 1.0: Included in Windows NT 3.1
- NTFS 1.1: Included in Windows NT 3.5/3.51
- NTFS 2.x: Included in Windows NT 4.0 up to SP3
- NTFS 3.0: Included in Windows NT 4.0 since SP3/2000
- NTFS 3.1: Included in Windows XP/2003/Vista/7/8/10/11

Recent versions are backward compatible with older versions and offer more features than the FAT file systems. NTFS includes transparent compression and encryption via Triple-DES and AES since version 2.x and the support for *quotas* since version 3.x.

NTFS

Ab Anfang der 1990er Jahre wurde das *New Technology File System* (NTFS) als Nachfolger der FAT-Dateisysteme von Microsoft für die Betriebssystemfamilie Windows NT entwickelt. Seit dieser Zeit sind folgende Versionen von NTFS erschienen:

- NTFS 1.0: Bestandteil von Windows NT 3.1
- NTFS 1.1: Bestandteil von Windows NT 3.5/3.51
- NTFS 2.x: Bestandteil von Windows NT 4.0 bis SP3
- NTFS 3.0: Bestandteil von Windows NT 4.0 ab SP3/2000
- NTFS 3.1: Bestandteil von Windows XP/2003/Vista/7/8/10/11

Neuere Versionen sind zu früheren Versionen abwärtskompatibel und bieten im Gegensatz zu den FAT-Dateisystemen einen vergrößerten Funktionsumfang. Dazu gehörten transparente Kompression und Verschlüsselung via Triple-DES und AES ab Version 2.x und die Unterstützung für Kontingente (*Quota*) ab Version 3.x.

Further enhancements of NTFS compared to its predecessor FAT include a maximum file size of 16 TB and a maximum file system size of 256 TB. Possible cluster sizes are 512 bytes to 64 kB. Just like VFAT (see Section 6.3.4), NTFS supports filenames of up to 255 Unicode characters in length. It implements interoperability with the MS-DOS operating system family by storing a unique file name in the format 8.3 for each file.

One of the characteristic elements of NTFS is the *Master File Table* (MFT). It contains references that assign extents and clusters to files. Furthermore, the metadata of the files are located in the MFT. It includes the file size, creation date, date of last modification, file type and sometimes the file content as well. The content of small files ≤ 900 bytes is stored directly in the MFT [80].

When a partition is formatted, a fixed space is reserved for the MFT. It is usually 12.5% of the partition size. If the MFT area has no more free capacity, the file system uses additional free space in the partition for the MFT. That may cause a fragmentation of the MFT, but this has no negative effect when using flash memory drives (see Section 4.4.7).

The clusters can be between 512 kB and 64 kB in size. Table 6.6 shows the standard cluster sizes of Windows 2000/XP/Vista/7/8/10/11 with different file system sizes. As with FAT file systems, the cluster size can be specified when the file system is created.

Weitere Verbesserungen von NTFS im Vergleich zu seinem Vorgänger FAT sind unter anderem eine maximale Dateigröße von 16 TB und eine maximale Dateisystemgröße von 256 TB. Mögliche Clustergrößen sind 512 Bytes bis 64 kB. Genau wie VFAT (siehe Abschnitt 6.3.4) speichert NTFS Dateinamen bis zu einer Länge von 255 Unicode-Zeichen und genau wie VFAT realisiert NTFS eine Kompatibilität zur Betriebssystemfamilie MS-DOS, indem es für jede Datei einen eindeutigen Dateinamen im Format 8.3 speichert.

Charakteristisch für den Aufbau von NTFS ist die *Hauptdatei* (englisch: *Master File Table* – MFT). Diese enthält Referenzen, die Extents und Cluster zu Dateien zuordnen. Zudem befinden sich die Metadaten der Dateien in der MFT. Dazu gehören die Dateigröße, das Datum der Erstellung, das Datum der letzten Änderung, der Dateityp und eventuell auch der Dateiinhalt. Der Inhalt kleiner Dateien ≤ 900 Bytes wird direkt in der MFT gespeichert [80].

Beim Formatieren einer Partition wird für die MFT ein fester Bereich reserviert. Üblicherweise sind das 12,5% der Partitionsgröße. Ist der Bereich voll, verwendet das Dateisystem zusätzlichen freien Speicher in der Partition für die MFT. Dabei kommt es meist zu einer Fragmentierung der MFT, was aber bei Flash-Speicher (siehe Abschnitt 4.4.7) keine negativen Auswirkungen hat.

Die Cluster können zwischen 512 kB und 64 kB groß sein. Tabelle 6.6 enthält die Standard-Clustergrößen unter Windows 2000/XP/Vista/7/8/10/11 bei unterschiedlich großen Dateisystemgrößen. Genau wie bei den FAT-Dateisystemen kann die Clustergröße beim Erzeugen des Dateisystems festgelegt werden.

Table 6.6: Default Cluster Size of NTFS for different Partition Sizes

Partition size	Cluster size
up to 16 TB	4 kB
16 TB - 32 TB	8 kB
32 TB - 64 TB	16 kB
64 TB - 128 TB	32 kB
128 TB - 256 TB	64 kB

Figure 6.12 shows the NTFS file system structure with the MFT and the area reserved for the MFT by default. Starting from the logical cen-

Abbildung 6.12 zeigt die Struktur des Dateisystems NTFS mit der MFT und dem standardmäßig reservierten Bereich für die MFT. Ab

ter of the partition, a backup copy of the most important metadata in the MFT exists. The figure also shows the structure of MFT records for files whose contents can be stored directly inside the MFT and for files whose contents are stored inside extents. In NTFS, the extents are also called *Data Runs*. For the addressing with extents in NTFS, the *Virtual Cluster Number* (VCN) indicates the first cluster number of an extent and the *Logical Cluster Number* (LCN) indicates the first cluster number of an extent on the storage device.

Directories in NTFS are files (MFT entries) whose file contents are the numbers of the MFT records (files) assigned to the particular directory.

der logischen Mitte der Partition befindet sich eine Sicherheitskopie der wichtigsten Metadaten in der MFT. Die Abbildung zeigt auch die Struktur von MFT-Einträgen für Dateien, deren Inhalt direkt in der MFT gespeichert werden kann, sowie für Dateien, deren Inhalte in Extents gespeichert sind. Die Extents heißen bei NTFS auch *Data Runs*. Bei der Adressierung mit Extents in NTFS bezeichnet *Virtual Cluster Number* (VCN) die erste Clusternummer eines Extents, und *Logical Cluster Number* (LCN) steht für die Nummer des ersten Clusters eines Extents auf dem Speichergerät.

Auch Verzeichnisse sind bei NTFS Dateien (MFT-Einträge), deren Dateiinhalt die Nummern der MFT-Einträge (Dateien) sind, die dem jeweiligen Verzeichnis zugeordnet sind.

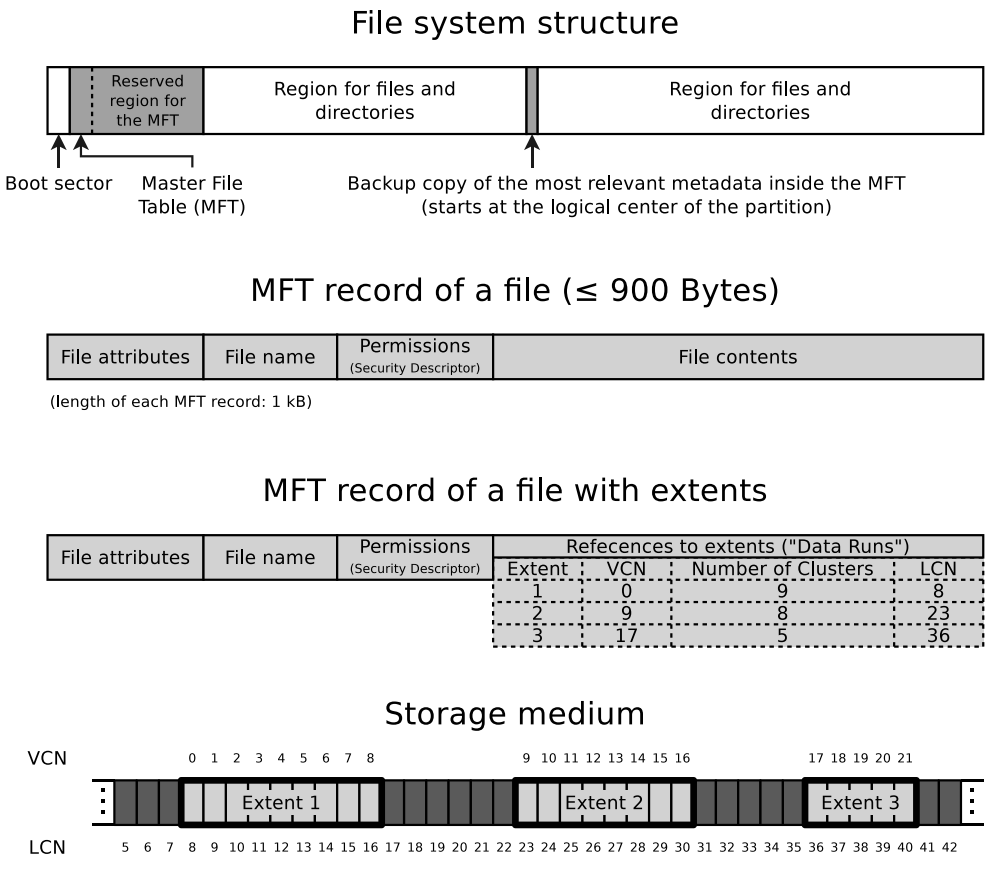

Figure 6.12: Structure of the NTFS File System with the Master File Table (MFT)

6.6

Copy-on-Write

A file system that implements the *Copy-on-Write* (CoW) principle does not modify the content of a file during a write operation. Instead, it writes the modified content as a new file by using free clusters (see Figure 6.13). Afterward, the metadata is modified for the new file. Until the metadata has been updated, the original file is kept and can be used after a system crash. Two benefits of CoW are better data security in comparison to journaling file systems. Also, older versions of modified files are kept by the file system and are available for the users. Examples of file systems that implement CoW are Btrfs, ZFS and ReFS.

Copy-on-Write

Arbeitet ein Dateisystem nach dem Prinzip *Copy-on-Write* (CoW) ändert es bei einem Schreibzugriff nicht den Inhalt der Originaldatei, sondern schreibt den veränderten Inhalt als neue Datei in freie Cluster (siehe Abbildung 6.13). Anschließend werden die Metadaten auf die neue Datei angepasst. Bis die Metadaten angepasst sind, bleibt die Originaldatei erhalten und kann nach einem Systemabsturz weiter verwendet werden. Zwei Vorteile von CoW sind eine bessere Datensicherheit im Vergleich zu Journaling-Dateisystemen und dass ältere Versionen geänderter Dateien vom Dateisystem vorgehalten werden, die für den Benutzer zur Verfügung stehen. Beispiele für Dateisysteme, die CoW unterstützen, sind Btrfs, ZFS und ReFS.

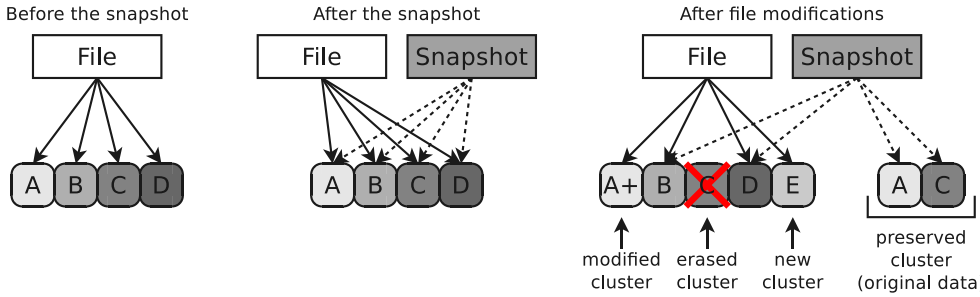

Figure 6.13: Using Copy-on-Write, old File Versions are preserved and can be restored

6.6.1

ZFS

Sun Microsystems initially developed the ZFS file system. It was free software under the Common Development and Distribution License (CDDL) from 2005 until its acquisition by Oracle in 2009. However, its incompatibility with the GNU General Public License (GPL) prevented an integration into the Linux kernel and a distribution outside the UNIX operating systems Solaris or OpenSolaris and FreeBSD. As of 2010, Oracle developed ZFS further and distributed it as proprietary software. At the same time, several initiatives and projects to

ZFS

Das Dateisystem ZFS wurde ursprünglich von Sun Microsystems entwickelt und war von 2005 bis zur Übernahme durch Oracle 2009 freie Software unter der Lizenz Common Development and Distribution License (CDDL). Deren Inkompatibilität zur Softwarelizenz GNU General Public License (GPL) verhinderte allerdings eine Integration in den Linux-Kernel und eine Verbreitung außerhalb der UNIX-Betriebssysteme Solaris bzw. OpenSolaris und FreeBSD. Ab 2010 wurde ZFS als proprietäre Software durch Oracle weiterentwickelt und vertrieben. Gleich-

utilize or reimplement ZFS on Linux and other free operating systems arose. One example is zfs-fuse, a prototype file system driver for the FUSE (Filesystem in Userspace) interface of the Linux kernel. The most advanced projects merged in 2013 to become the OpenZFS project. The result is a port and further development of the latest free release of the ZFS source code from Sun/Oracle with a focus on Linux and FreeBSD, but in principle, it can also run in other operating systems.

In addition to CoW, ZFS implements several state-of-the-art features. Multiple physical volumes can be combined into logical units (so-called pools), and different RAID levels can be implemented via built-in software RAID (see Section 4.5) to improve reliability. ZFS thus also includes the functionality of a Logical Volume Manager. A Logical Volume Manager (LVM) implements an abstraction layer between storage devices, individual partitions, and file systems. An LVM can create and modify virtual partitions (*logical volumes*) during runtime. Logical volumes can span across multiple storage devices and provide software RAID. The vast majority of modern operating systems include an LVM.

Furthermore, ZFS implements integrated data compression and encryption, automatic error correction with checksums and snapshots. By the so-called ZFS Intent Log (ZIL), ZFS also implements a feature that is similar to journaling (see Section 6.4). The ZIL can reside within a pool or on its dedicated fast drive. When using the ZIL, data is first written into the ZIL before being written into the file system. Thus, in the same way, as in journaling, data is written twice.

The maximum file size and the maximum file system size in ZFS is 16 EB (2^{64} bytes). However, because ZFS implements 128 bits long addresses, much larger file systems are possible in theory [14].

zeitig entstanden verschiedene Ansätze und Projekte zur Nutzbarmachung bzw. Reimplementierung von ZFS unter Linux und anderen freien Betriebssystemen. Ein Beispiel dafür ist zfs-fuse, ein prototypischer Dateisystemtreiber für die FUSE-Schnittstelle (Filesystem in Userspace) des Linux-Kernels. Die am weitesten fortgeschrittenen Projekte haben sich 2013 im Projekt OpenZFS zusammengetan. Das Resultat ist eine Portierung und Weiterentwicklung der letzten freien Version des ZFS-Quellcodes von Sun/Oracle mit dem Fokus auf Linux und FreeBSD, das aber prinzipiell auch unter weiteren Betriebssystemen lauffähig ist.

Außer CoW implementiert ZFS mehrere moderne Funktionen. Mehrere physische Datenträger können zu logischen Einheiten (sogenannten Pools) zusammengefasst und verschiedene RAID-Level via eingebautem Software-RAID (siehe Abschnitt 4.5) zur Erhöhung der Ausfallsicherheit realisiert werden. ZFS enthält somit auch die Funktionalität eines Logical Volume Managers. Ein Logical Volume Manager (LVM) realisiert eine Abstraktionsebene zwischen Speicherlaufwerken, einzelnen Partitionen und den Dateisystemen. Durch einen LVM können virtuelle Partitionen (*logische Volumes*) erzeugt und während der Laufzeit verändert werden. Die logischen Volumes können sich über mehrere Speicherlaufwerke erstrecken und ermöglichen Software-RAID. Die allermeisten modernen Betriebssysteme enthalten einen LVM.

Zusätzlich implementiert ZFS u.a. integrierte Datenkompression und Verschlüsselung, automatische Fehlerkorrektur mit Prüfsummen und platzsparende Schnappschüsse (englisch: *Snapshots*). Mit dem sogenannten ZFS Intent Log (ZIL) implementiert ZFS auch eine zum Journaling (siehe Abschnitt 6.4) vergleichbare Funktionalität. Das ZIL kann sich innerhalb eines Pools befinden oder auf einem eigenen schnellen Laufwerk. Bei Verwendung des ZIL werden Daten zuerst in das ZIL geschrieben und dann in das Dateisystem. Es werden also wie beim Journaling auch Daten doppelt geschrieben.

Die maximale Dateigröße bei ZFS ist 16 EB (2^{64} Bytes). Der gleiche Wert gilt für die maximale Dateisystemgröße. Dadurch, dass ZFS intern mit 128 Bits langen Adressen arbeitet, sind theoretisch deutlich größere Dateisysteme möglich [14].

6.6.2

Btrfs

The Btrfs file system has been developed as free software under the GPL software license since 2007 and included in the Linux kernel since 2013. The state-of-the-art features of this file system include copy-on-write, snapshots, built-in data compression, and automatic error correction with checksums. Btrfs also implements the functions of a logical volume manager and provides integrated software RAID (see Section 4.5).

Btrfs could become the future standard file system. Reasons are the features mentioned already, the seamless integration into the Linux kernel, the active support of the development by several companies, and the integration as standard file system into several Linux distributions (e.g., Fedora and openSUSE) and commercial server systems and storage systems (e.g., the NAS servers of Synology mostly use Btrfs). Btrfs implements extents for the cluster addressing.

By default, the cluster size in Btrfs is equal to the page size (usually 4 kB) [98]. Larger clusters up to a maximum of 64 kB are also possible. However, it must be a multiple of the block size (sector size) on the storage device and a power of two. The maximum file size and the maximum file system size in Btrfs is 16 EB (2^{64} bytes).

Btrfs

Btrfs ist ein freies Dateisystem, das seit 2007 als freie Software unter der Softwarelizenz GPL entwickelt wird und seit 2013 im Linux-Kernel enthalten ist. Zu den modernen Fähigkeiten dieses Dateisystems gehören Copy-on-Write, platzsparende Snapshots, integrierte Datenkompression und automatische Fehlerkorrektur mit Prüfsummen. Auch Btrfs enthält die Funktionalität eines Logical Volume Managers und bietet integriertes Software-RAID (siehe Abschnitt 4.5).

Wegen der genannten positiven Eigenschaften, der nahtlosen Integration in den Linux-Kernel, der aktiven Unterstüzung der Entwicklung durch zahlreiche Unternehmen und der Integration als Standarddateisystem in mehrere Linux-Distributionen (z.B. Fedora und openSUSE) und kommerzielle Server- und Speichersysteme (z.B. die NAS-Server der Firma Synology verwenden meist Btrfs) könnte es in Zukunft das nächste Standard-Dateisystem werden. Die Adressierung der Cluster geschieht bei Btrfs mit Hilfe von Extents.

Die Größe der Cluster in Btrfs entspricht standardmäßig der Seitengröße (meist 4 kB) [98]. Auch größere Cluster bis maximal 64 kB sind möglich. Es muss sich allerdings um ein Vielfaches der Blockgröße (Sektorgröße) auf dem Datenträger und eine Zweierpotenz handeln. Die maximale Dateigröße bei Btrfs ist 16 EB (2^{64} Bytes). Der gleiche Wert gilt für die maximale Dateisystemgröße.

6.6.3

ReFS

Windows Server 2012/2016/2019/2022/2025 and selected versions of Windows 8/10/11 include the *Re*silient *F*ile *S*ystem (ReFS). Most of these operating system versions only allow the use of ReFS for specific applications, such as software RAID. With Windows 11 and subsequent versions, the existing restrictions are expected to disappear over time.

ReFS is considered the future standard file system for the Windows operating system family and the successor to NTFS. However, ReFS

ReFS

Windows Server 2012/2016/2019/2022/2025 und ausgewählte Versionen von Windows 8/10/11 enthalten das *Re*silient *F*ile *S*ystem (ReFS). Die meisten der genannten Betriebssystem-Versionen erlauben die Verwendung von ReFS nur für wenige Anwendungszwecke, wie zum Beispiel Software-RAID. Mit Windows 11 und nachfolgenden Versionen ist zu erwarten, dass die bestehenden Einschränkungen nach und nach wegfallen.

ReFS gilt als zukünftiges Standard-Dateisystem der Windows-Betriebssystemfamilie und als Nachfolger

lacks some of the features of modern file systems. For example, unlike NTFS, it does not yet implement compression or encryption. It is also impossible to convert existing NTFS file systems to ReFS.

The clusters can be 4 kB or 64 kB in size. The maximum file size and maximum file system size is 35 PB. ReFS stores files and directories as objects within key-value tables implemented as B+ trees [87].

von NTFS. Einige Fähigkeiten moderner Dateisysteme fehlen aber bislang bei ReFS. So implementiert es beispielsweise im Gegensatz zu NTFS bislang weder Kompression, noch Verschlüsselung. Auch eine Konvertierung bestehender NTFS-Dateisysteme in ReFS ist bislang nicht möglich.

Die Cluster können entweder 4 kB oder 64 kB groß sein. Die maximale Dateigröße ist 35 PB. Der gleiche Wert gilt für die maximale Dateisystemgröße. Dateien und Verzeichnisse speichert ReFS als Objekte in Schlüssel-Wert-Tabellen, die als B+ Bäume realisiert sind [87].

6.7

Accelerating Data Access with a Cache

Modern operating systems accelerate file accesses with a *cache* in the main memory, called *page cache* or *buffer cache* [67, 119]. If a file is requested for reading, the kernel first tries to allocate the file in the page cache. If the file is not present there, it is loaded into the page cache. The page cache is never as big as the amount of stored files on the storage drives of the computer system. That is why rarely requested files must be replaced from the page cache. If a file has been modified in the cache, the modifications must be passed down (written back) the memory hierarchy, at latest, when the file is replaced. Optimal use of the page cache is impossible because data accesses are *non-deterministic*, i.e., unpredictable.

Modern operating systems usually do not pass down write operations immediately. They operate according to the *write-back* principle (see Section 4.4.2). For example, the MS-DOS and Windows operating system families up to and including version 3.11 use the *Smartdrive* utility to implement a page cache. All later versions of Windows include a *cache manager* that implements a page cache [77]. Linux automatically buffers as many files in the page cache as there is free space in the main memory. The command `free -m` returns an overview of the memory usage in the command line interpreter

Datenzugriffe mit einem Cache beschleunigen

Moderne Betriebssysteme beschleunigen Dateizugriffe mit einem *Cache* im Hauptspeicher, der *Page Cache* oder *Buffer Cache* genannt wird [67, 119]. Wird eine Datei lesend angefragt, schaut der Betriebssystemkern zuerst, ob die Datei im Page Cache vorliegt. Bei einem negativem Ergebnis wird sie in diesen geladen. Der Page Cache ist nie so groß, wie die Menge der gespeicherten Dateien auf den Speicherlaufwerken des Computersystems. Darum müssen selten nachgefragte Dateien aus dem Page Cache verdrängt werden. Wurde eine Datei im Cache verändert, müssen die Änderungen spätestens beim Verdrängen in der Speicherhierarchie nach unten durchgereicht (zurückgeschrieben) werden. Ein optimales Verwenden des Page Cache ist nicht möglich, da Datenzugriffe nicht *deterministisch*, also nicht vorhersagbar sind.

In der Regel geben moderne Betriebssystemen Schreibzugriffe nicht direkt weiter. Sie arbeiten nach dem Funktionsprinzip des *Write-Back* (siehe Abschnitt 4.4.2). Die Betriebssystemfamilien MS-DOS und Windows bis einschließlich Version 3.11 verwenden beispielsweise das Programm *Smartdrive*, um einen Page Cache zu realisieren. Auch all späteren Versionen von Windows enthalten einen *Cache Manager*, der einen Page Cache verwaltet [77]. Linux puffert automatisch so viele Dateien im Page Cache wie Platz im Hauptspeicher frei ist. Das Kommando `free -m` gibt im Kommandozeileninterpreter (*Shell*) un-

(*shell*) in Linux and informs how much main memory is currently used for the page cache.

One advantage of using a page cache is that it increases the performance of the system when accessing files. One drawback is that modifications to files are lost if the system crashes.

ter Linux eine Übersicht der Speicherbelegung aus und informiert darüber, wie viel Speicherkapazität des Hauptspeichers gegenwärtig für den Page Cache verwendet wird.

Ein Vorteil, der durch den Einsatz eines Page Cache entsteht, ist die höhere System-Geschwindigkeit bei Dateizugriffen. Ein Nachteil ist, dass bei einem Systemabsturz Dateiänderungen verloren gehen.

6.8

Defragmentation

As described in Section 6.1, each cluster in the file system can only be assigned to a single file. If a file is larger than one cluster, it is split and distributed among several clusters. *Fragmentation* inevitably occurs over time in each file system. It means that logically related clusters, i.e., the clusters of a file, are not located physically close to each other. In the age of hard disk drives and small caches, the fragmentation of data often had a negative impact on the performance of computer systems, because if the clusters of a file are distributed over the hard disk drive, the heads need to perform more time-consuming position changes when accessing the file. Therefore, during the development of file systems such as ext2 (see Section 6.2.2), one objective was to avoid frequent head movements in the hard disk drives by a smart arrangement of the metadata. For solid-state drives (see Section 4.4.7), the position of the clusters does not affect the access time (latency).

Even with hard disk drives, defragmentation in practice seldomly results in a better overall system performance. It is because a continuous arrangement of the clusters would only accelerate continuous forward reading of that file, because in such case, no more seek times occur.

In general, defragmenting a storage device makes sense only if the seek times are long. With operating systems that do not use a page cache, long seek times have an explicitly negative effect.

Defragmentierung

Wie in Abschnitt 6.1 beschrieben, darf jeder Cluster im Dateisystem nur einer Datei zugeordnet sein. Ist eine Datei größer als ein Cluster, wird sie auf mehrere verteilt. Zwangsläufig kommt es über die Zeit in jedem Dateisystem zur *Fragmentierung*. Das heißt, dass logisch zusammengehörende Cluster, also die Cluster einer Datei, nicht räumlich beieinander sind. Im Zeitalter der Festplatten und geringer Caches konnte sich die Fragmentierung der Daten negativ auf die Leistungsfähigkeit eines Computers auswirken, denn liegen die Cluster einer Datei über die Festplatte verteilt, müssen die Festplattenköpfe (siehe Abschnitt 4.4.4) bei Zugriffen auf die Datei eine höhere Anzahl zeitaufwendiger Positionswechsel durchführen. Darum war es bei der Entwicklung mancher Dateisysteme wie zum Beispiel ext2 (siehe Abschnitt 6.2.2) ein Ziel, häufige Bewegungen der Schwungarme durch eine geschickte Anordnung der Metadaten zu vermeiden. Bei Solid-State Drives (siehe Abschnitt 4.4.7) spielt die Position der Cluster keine Rolle für die Zugriffsgeschwindigkeit.

Auch bei Festplatten führt Defragmentierung in der Praxis nur selten zu einer besseren Gesamtleistung des Computers. Der Grund dafür ist, dass eine zusammenhängende Anordnung der Cluster einer Datei nur das *fortlaufende Vorwärtslesen* eben dieser Datei beschleunigen würde, da in einem solchen Fall keine Suchzeiten mehr vorkommen würden.

Überhaupt ist das Defragmentieren eines Datenspeichers nur dann sinnvoll, wenn die Suchzeiten groß sind. Bei Betriebssystemen, die keinen Page Cache verwenden, wirken sich hohe Suchzeiten besonders negativ aus.

Operating systems with single program operation mode (see Section 3.4.2) such as MS-DOS can only run one application at a time. On such operating systems, if the running process *hangs* because it waits for the results of read and write requests, the system speed is significantly reduced. For this reason, periodic defragmentation of the connected hard disk drives can be useful for singletaslking operating systems.

In multitasking operating systems, several programs are always executed in parallel, or at least in a quasi-parallel way. In practice, processes can seldom read large amounts of data in succession, without other processes requesting read/write operations that are carried out "*in between*" by the operating system scheduler (see Section 8.6). To prevent programs, that are running at the same time, from interfering too much with each other, operating systems read more data than requested for each read request, and keep a stock of data in the page cache (see Section 6.7). The positive effect of the page cache on the overall performance of the system dramatically exceeds the short-term benefits of defragmentation.

Writing data to a storage device always leads to fragmentation. The defragmentation of hard disk drives has, in such operating systems, mainly a *benchmarking effect* that is not relevant in practice. For this reason, defragmenting the hard disk drives of multitasking operating systems is not useful concerning the overall system performance and the lifespan of the storage devices.

Bei Betriebssystemen mit Einzelprogrammbetrieb (siehe Abschnitt 3.4.2) wie zum Beispiel MS-DOS kann immer nur eine Anwendung laufen. Wenn bei solchen Betriebssystemen der laufende Prozess *hängt*, weil er auf die Ergebnisse von Lese- und Schreibanforderungen wartet, verringert dies die Systemgeschwindigkeit signifikant. Aus diesem Grund kann bei Betriebssystemen mit Einzelprogrammbetrieb das regelmäßige Defragmentieren der angeschlossenen Festplatten sinnvoll sein.

Bei modernen Betriebssystemen mit Mehrprogrammbetrieb laufen immer mehrere Programme parallel oder zumindest quasi-parallel ab. In der Praxis können Prozesse fast nie große Datenmengen am Stück lesen, ohne dass durch das Scheduling (siehe Abschnitt 8.6) des Betriebssystems andere Anwendungen ihre Lese- und Schreibanweisungen „*dazwischenschieben*“. Damit sich gleichzeitig laufende Programme nicht zu sehr gegenseitig behindern, lesen Betriebssysteme bei jeder Leseanweisung mehr Daten ein als angefordert und sie halten einen Vorrat an Daten im Page Cache (siehe Abschnitt 6.7). Die auf die Gesamtleistung des Systems positive Wirkung des Page Cache überwiegt bei weitem die kurzzeitigen Vorteile einer Defragmentierung.

Durch das Schreiben von Daten auf einen Datenträger kommt es zwangsläufig zu Fragmentierung. Das Defragmentieren der Festplatten hat bei solchen Betriebssystemen primär einen *Benchmark-Effekt*, der für die Praxis nicht relevant ist. Aus diesem Grund ist bei Betriebssystemen mit Mehrprogrammbetrieb das Defragmentieren der Festplatten im Hinblick auf die Gesamtleistung des Systems und auf die Lebensdauer der Datenspeicher nicht sinnvoll.

7

System Calls

Systemaufrufe

As described in Section 5.3.2, all processes outside the operating system kernel are only allowed to access their virtual memory. If a user mode process wants to perform a more privileged task, such as a hardware access, the creation or management of a process or a file, it must request this from the kernel via a *system call*.

This chapter starts with a description of user mode and kernel mode and how operating systems implement them. An introduction to system calls follows, and how to invoke them directly or indirectly via library functions.

Wie in Abschnitt 5.3.2 beschrieben, dürfen alle Prozesse außerhalb des Betriebssystemkerns ausschließlich auf ihren eigenen virtuellen Speicher zugreifen. Will ein Prozess im Benutzermodus eine höher privilegierte Aufgabe wie zum Beispiel einen Hardwarezugriff durchführen, einen Prozess oder eine Datei erzeugen oder verwalten, muss er dies dem Betriebssystemkern durch einen *Systemaufruf* (englisch: *System Call*) mitteilen.

Zu Beginn dieses Kapitels erfolgt eine Beschreibung des Benutzermodus und des Kernelmodus und wie diese realisiert sind. Anschließend folgt eine Einführung in das Thema Systemaufrufe und wie diese direkt oder indirekt via Bibliotheksfunktionen aufgerufen werden.

7.1

User Mode and Kernel Mode

Benutzermodus und Kernelmodus

x86-compatible CPUs implement four *privilege levels*, also called *rings* (see Figure 7.1). The privilege levels are intended to improve stability and security. Each process is permanently assigned to a ring and cannot free itself from it. The rings are implemented using the register *Current Privilege Level* (CPL). It stores the ring number of the currently running process [54].

The kernel (see Section 3.8) runs in ring 0, the *kernel mode*. Processes running in kernel mode have full hardware access. The kernel can also address physical memory directly in real mode (see Section 5.3.1). The other processes run in

x86-kompatible Prozessoren enthalten vier *Privilegienstufen*, die auch *Ringe* heißen (siehe Abbildung 7.1). Die Privilegienstufen sollen die Stabilität und Sicherheit verbessern. Jeder Prozess wird in einem Ring ausgeführt und kann sich nicht selbstständig aus diesem befreien. Die Realisierung der Ringe geschieht mit Hilfe des Registers *Current Privilege Level* (CPL). Dieses speichert die Ringnummer des aktuell laufenden Prozesses [54].

Im Ring 0, dem sogenannten *Kernelmodus*, läuft der Betriebssystemkern (siehe Abschnitt 3.8). Prozesse, die im Kernelmodus laufen, haben vollen Zugriff auf die Hardware. Der Kern kann auch physischen Speicher direkt im Real Mode (siehe Abschnitt 5.3.1) adressieren.

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_7

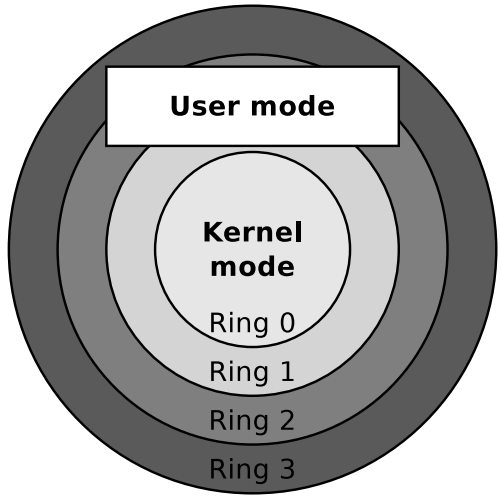

Figure 7.1: The four Privilege Levels (Rings) of x86-compatible CPUs

ring 3, the *user mode*. These processes can only access virtual memory (see Section 5.3.2).

Modern operating systems only use two privilege levels. One reason for this is that some once-popular hardware architectures, such as the Alpha CPU, the PowerPC architecture, and the MIPS architecture, only implement two levels. [23, 100]

Im Ring 3, dem sogenannten *Benutzermodus*, laufen die übrigen Prozesse. Diese arbeiten ausschließlich mit virtuellem Speicher (siehe Abschnitt 5.3.2).

Moderne Betriebssysteme verwenden ausschließlich zwei Privilegienstufen. Ein Grund dafür ist, dass einige vormals populäre Hardwarearchitekturen wie zum Beispiel der Alpha-Prozessor, die PowerPC-Architektur und die MIPS-Architektur, nur zwei Stufen enthalten. [23, 100]

7.2

System Calls and Libraries

A system call can be invoked directly or through a library (see Figure 7.2). A system call is a function call in the operating system that triggers a switch from user mode to kernel mode. It is called a *context switch*.

During a context switch, a process passes the control over the CPU to the kernel and is suspended until the request is processed. After the system call, the kernel returns the control over the CPU to the process in user mode.

Systemaufrufe und Bibliotheken

Ein Systemaufruf kann direkt oder über den Umweg einer Bibliothek aufgerufen werden (siehe Abbildung 7.2). Bei einem Systemaufruf handelt es sich um einen Funktionsaufruf im Betriebssystemkern, der einen Sprung vom Benutzermodus in den Kernelmodus auslöst. In diesem Kontext spricht man vom sogenannten *Moduswechsel*.

Beim Moduswechsel gibt ein Prozess die Kontrolle über den Hauptprozessor an den Betriebssystemkern ab und ist so lange unterbrochen, bis die Anfrage bearbeitet ist. Nach dem Systemaufruf gibt der Kern den Prozessor wieder

The process then continues its execution at the point where the context switch was previously requested. The functionality of a system call is always provided in the kernel and thus outside of the address space of the calling process.

an den Prozess im Benutzermodus ab. Der Prozess führt daraufhin seine Abarbeitung an der Stelle fort, an der er den Moduswechsel zuvor angefordert hat. Die Leistung eines Systemaufrufs wird immer im Kern und damit außerhalb des Adressraums des aufrufenden Prozesses erbracht.

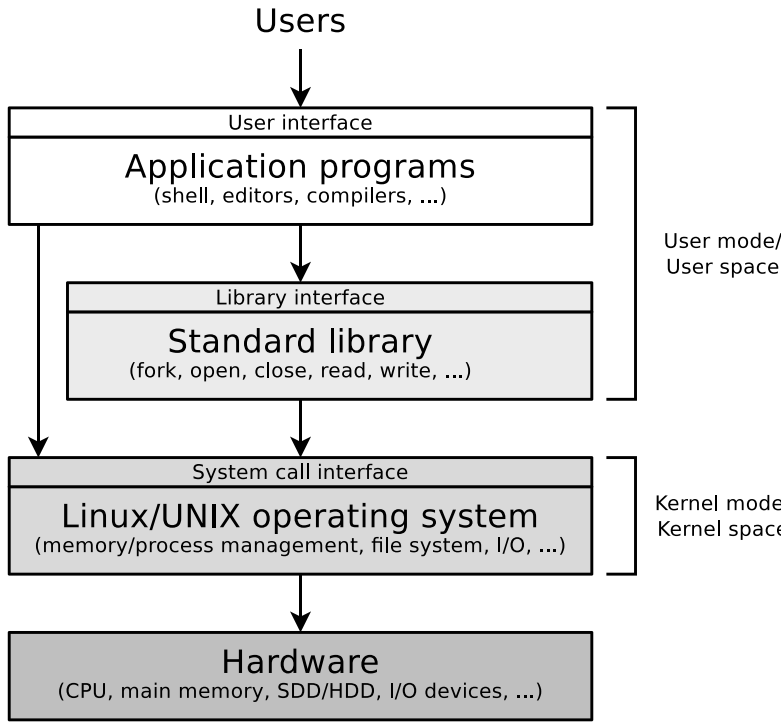

Figure 7.2: System Calls are the Interface to the Operating System for User Mode Processes

One example of a system call is `ioctl`. Processes in Linux call device-specific instructions with this system call. It enables processes to communicate and control character-oriented devices (e.g., mouse, keyboard, printer, and terminals) as well as block-oriented devices (e.g., SSD/hard disk drives and CD/DVD drive). The syntax of `ioctl` is:

```
ioctl (file descriptor,
       request code number,
       integer value or pointer);
```

Ein Beispiel für einen Systemaufruf ist `ioctl`. Damit realisieren Prozesse unter Linux gerätespezifische Anweisungen. Dieser Systemaufruf ermöglicht Prozessen die Kommunikation und Steuerung von zeichenorientierten Geräten (z.B. Maus, Tastatur, Drucker und Terminals) sowie blockorientierten Geräten (z.B. SSD/Festplatte und CD-/DVD-Laufwerk). Die Syntax von `ioctl` ist:

```
ioctl (File-Deskriptor,
       Aktionsanforderung,
       Integer-Wert oder Zeiger);
```

Some typical application scenarios for this system call are the formatting of a floppy disk track, initializing a modem or sound card, ejecting a CD from the drive, reading the status and link information of the Wi-Fi interface, or accessing sensors and actuators via a bus, such as the serial Inter-Integrated Circuit (I^2C) data bus.

Modern operating system kernels include several hundred system calls [66, 125]. A selection from the fields process management, file management, and directory management is shown in Table 7.1.

Einige typische Einsatzszenarien dieses Systemaufrufs sind das Formatieren einer Diskettenspur, das Initialisieren eines Modems oder einer Soundkarte, das Auswerfen einer CD auf dem Laufwerk, das Auslesen von Status- und Verbindungsinformationen der WLAN-Schnittstelle oder der Zugriff auf Sensoren und Aktoren über einen Bus wie den seriellen Datenbus Inter-Integrated Circuit (I^2C).

Die Kerne moderner Betriebssysteme bieten mehrere hundert Systemaufrufe an [66, 125]. Eine Auswahl unter anderem aus den Aufgabenbereichen Prozess-, Datei- und Verzeichnisverwaltung enthält Tabelle 7.1.

Table 7.1: A Selection of System Calls of the Linux Kernel

System call	Function
`chdir`	Change current directory
`chmod`	Change file permissions of a file
`chown`	Change the owner of a file
`close`	Close an open file
`execve`	Replace the calling process with a new one and keep the process ID (PID)
`exit`	Terminate the calling process
`fork`	Create a new child process
`getpid`	Request the process ID (PID) of the calling process and print it out
`getppid`	Request the parent process ID (PPID) of the calling process and print it out
`kill`	Send a signal to a process
`link`	Create a directory entry (link) to a file
`lseek`	Specify the read/write file offset
`mkdir`	Create a new directory
`mount`	Attach a file system to the file system hierarchy
`open`	Open a file for reading/writing
`read`	Read data from a file into the buffer
`rmdir`	Remove an empty directory
`stat`	Determine the status of a file
`umount`	Detach a file system
`uname`	Request information about the running kernel and print it out
`unlink`	Erase a directory entry
`time`	Print the number of seconds since January 1st, 1970 (*UNIX time*)
`waitpid`	Wait for the termination of a child process
`write`	Write data from the buffer into a file

A list with the names of the system calls in Linux can be found in the source code of the Linux kernel. For kernel version 2.6.x, this list is located in the file `arch/x86/kernel/syscall_table_32.S`.

For kernel version 3.x, 4.x, and 5.x, the list of system calls is either located in the files `arch/x86/syscalls/syscall_[64|`

Eine Liste mit den Namen der Systemaufrufe unter Linux befindet sich in den Quellen des Linux-Kerns. Bei Kernel-Version 2.6.x ist die Liste in der Datei `arch/x86/kernel/syscall_table_32.S`.

Beim Linux-Kernel 3.x, 4.x und 5.x befinden sich die Systemaufrufe in den Dateien `arch/x86/syscalls/syscall_[64|32].tbl`

32].tbl or in arch/x86/entry/syscalls/syscall_[64|32].tbl.

Invoking system calls directly from self-developed programs is usually not recommended. The reason is that the portability of such software is poor since not all system calls are identical for different operating system families. Furthermore, it is not guaranteed that future releases of the operating system kernel do not contain modifications to individual system calls. For this reason, for the development of self-written software, it is better to use library functions, which are logically located between the user processes and the kernel. These functions are also called *wrapper functions* [35] in literature. All modern operating systems include such libraries.

The library is used for the communication between the user processes and the kernel, and the context switching between user mode and kernel mode. Examples of such libraries are the UNIX C standard library, the Linux GNU C library glibc, and the Windows Native API ntdll.dll.

The program example in Listing 7.1 shows how convenient it is to use the library function of the same name instead of the system call using a simple C source code, and the system call getpid.

```
1 #include <syscall.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <sys/types.h>
5
6 int main(void) {
7   unsigned int ID1, ID2;
8
9   // System call
10  ID1 = syscall(SYS_getpid);
11  printf ("Result of the system call: %d\n", ID1);
12
13  // Wrapper function of the glibc, which calls the system call
14  ID2 = getpid();
15  printf ("Result of the wrapper function: %d\n", ID2);
16
17  return(0);
18 }
```

Listing 7.1: Calling a Library Function is as simple as directly invoking a System Call

Compiling this source code with the GNU C compiler (gcc) in Linux and executing it produces the same result for the system call and

oder arch/x86/entry/syscalls/syscall_[64|32].tbl.

Aus eigenen Programmen Systemaufrufe direkt aufrufen ist in der Praxis meist nicht empfehlenswert. Der Grund dafür ist, dass solche Software schlecht portabel ist, da nicht alle Systemaufrufe bei den verschiedenen Betriebssystemfamilien identisch sind. Zudem ist nicht garantiert, dass eine neue Version des Betriebssystemkerns nicht auch Veränderungen an einzelnen Systemaufrufen enthält. Aus diesem Grund sollte bei der Entwicklung eigener Software lieber auf die Funktionen einer Bibliothek zurückgegriffen werden, die sich logisch zwischen den Benutzerprozessen und dem Betriebssystemkern befindet. Diese Funktionen heißen in der Literatur auch *Wrapper-Funktionen* [35]. Alle modernen Betriebssysteme enthalten solche Bibliotheken.

Die Bibliothek ist zuständig für die Vermittlung der Kommunikation zwischen den Benutzerprozessen mit dem Betriebssystemkern und für das Anweisen der Moduswechsel zwischen Benutzer- und Kernelmodus. Beispiele für solche Bibliotheken sind die UNIX C Standard Library, die Linux GNU C-Bibliothek glibc und die Native API ntdll.dll von Windows.

Das Programmbeispiel in Listing 7.1 zeigt anhand eines einfachen C-Quellcodes und des Systemaufrufs getpid, wie einfach es ist, anstatt des Systemaufrufs die gleichnamige Bibliotheksfunktion zu verwenden.

Das Übersetzen dieses Quellcodes mit dem GNU C Compiler (gcc) unter Linux und anschließende Ausführen führt beim Systemaufruf

the library function, namely the process number of the running process is returned:

und der Bibliotheksfunktion zum gleichen Ergebnis, nämlich der Ausgabe der Prozessnummer des laufenden Prozesses:

```
$ gcc Listing_7_1_Systemcall.c -o Listing_7_1_Systemcall
$ ./Listing_7_1_Systemcall
Result of the system call: 3452
Result of the wrapper function: 3452
```

7.3

System Call Processing

For a detailed understanding of how the kernel processes a system call that is wrapped by a library function, the individual steps are described in this section. The following example and its description are taken from [112]. Figure 7.3 shows the processing of the system call `read` when it gets invoked by the library function of the same name. `read` reads a certain number (`nbytes`) of bytes from the file (`fd`) and stores it in a buffer (`buffer`).

Ablauf eines Systemaufrufs

Zum besseren Verständnis, wie der Betriebssystemkern einen durch eine Bibliotheksfunktion gekapselten Systemaufruf abarbeitet, enthält dieser Abschnitt eine Beschreibung der einzelnen Arbeitsschritte. Das folgende Beispiel und seine Beschreibung ist aus [115] entnommen. Abbildung 7.3 zeigt die Abarbeitung des Systemaufrufs `read`, wenn er durch die gleichnamige Bibliotheksfunktion aufgerufen wird. `read` liest eine bestimmte Menge (`nbytes`) von Bytes aus der Datei (`fd`) und schreibt sie in einen Puffer (`buffer`).

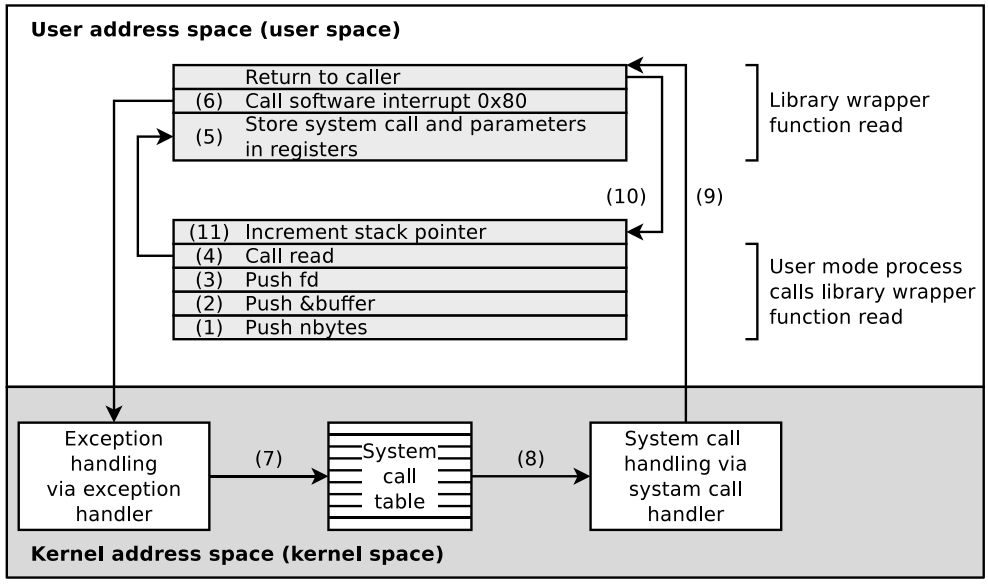

Figure 7.3: Processing of a System Call Step-by-Step [112]

The user mode process stores the parameters on the stack in the steps 1-3 (see Section 8.3), and invokes the user mode process, the library function for `read`, in step 4. The library function stores the system call number in the *accumulator register* `EAX` (or in the 64-bit register `RAX`) in step 5. The library wrapper function stores the parameters of the system call in the registers `EBX`, `ECX`, and `EDX` (or in the 64-bit registers `RBX`, `RCX`, and `RDX`).

In step 6, the software interrupt (*exception*) 0x80 (decimal: 128) is triggered to switch from user mode to kernel mode. The software interrupt interrupts program execution in user mode and enforces the execution of an exception handler in kernel mode.

The operating system kernel maintains the *system call table*, which is a list of all system calls. In this list, each system call is assigned to a unique number and an internal kernel function.

The called exception handler is a function in the kernel that reads out the content of the `EAX` (or in the 64-bit register `RAX`) register. In step 7, the exception handler function calls the corresponding kernel function from the system call table with the arguments that are stored in the registers `EBX`, `ECX`, and `EDX` (or in the 64-bit registers `RBX`, `RCX`, and `RDX`). In step 8, the system call is executed, and in step 9, the exception handler returns control back to the library that triggered the software interrupt. Then the function returns to the user mode process in step 10 in the same way as a normal function would have done. To complete the system call, the user mode process must clean up the stack in step 11, just like after every function call. The user process can then continue to run.

In den Schritten 1-3 legt der Benutzerprozess die Parameter auf den Stack (siehe Abschnitt 8.3) und in Schritt 4 ruft er die Bibliotheksfunktion für `read` auf. Die Bibliotheksfunktion speichert in Schritt 5 die Nummer des Systemaufrufs im *Akkumulator Register* `EAX` (bzw. im entsprechenden 64-Bit-Register `RAX`). Die Parameter das Systemaufrufs speichert die Bibliotheksfunktion in den Registern `EBX`, `ECX` und `EDX` (bzw. in den entsprechenden 64-Bit-Registern `RBX`, `RCX` und `RDX`).

In Schritt 6 wird der Softwareinterrupt (englisch: *Exception*) 0x80 (dezimal: 128) ausgelöst, um vom Benutzermodus in den Kernelmodus zu wechseln. Der Softwareinterrupt unterbricht die Programmausführung im Benutzermodus und erzwingt das Ausführen eines Exception-Handlers im Kernelmodus.

Der Betriebssystemkern verwaltet die *System Call Table*, eine Liste mit allen Systemaufrufen. Jedem Systemaufruf ist dort eine eindeutige Nummer und eine Kernel-interne Funktion zugeordnet.

Der aufgerufene Exception-Handler ist eine Funktion im Kern, die den Inhalt des Registers `EAX` (bzw. im 64-Bit-Register `RAX`) ausliest. Die Exception-Handler-Funktion ruft in Schritt 7 die entsprechende Funktion im Kern aus der System Call Table mit den in den Registern `EBX`, `ECX` und `EDX` (bzw. in den 64-Bit-Registern `RBX`, `RCX` und `RDX`) gespeicherten Argumenten auf. In Schritt 8 startet der Systemaufruf und in Schritt 9 gibt der Exception-Handler die Kontrolle an die Bibliothek zurück, die den Softwareinterrupt ausgelöst hat. Die Funktion kehrt danach in Schritt 10 zum Benutzerprozess so zurück, wie es auch eine normale Funktion getan hätte. Um den Systemaufruf abzuschließen, muss der Benutzerprozess in Schritt 11 genau wie nach jedem Funktionsaufruf den Stack aufräumen. Anschließend kann der Benutzerprozess weiterarbeiten.

8

Process Management Prozessverwaltung

Process management, as described in Section 3.8, is one of the essential functions of operating systems.

Every *process* in the operating system is an instance of a currently running program. Moreover, processes are dynamic objects that represent sequential activities in the computer system.

On a modern computer system, multiple processes are being executed all the time. In multitasking operation mode, the CPU switches back and forth between the processes.

Each process includes, in addition to the program code, its *process context*, which is independent of the contexts of other processes. Operating systems manage three types of context information: *hardware context*, *system context*, and *user context*.

Wie in Abschnitt 3.8 bereits beschrieben wurde, ist die Prozessverwaltung eine der grundlegenden Funktionalitäten eines Betriebssystems.

Jeder *Prozess* im Betriebssystem ist eine Instanz eines Programms, das ausgeführt wird. Zudem sind Prozesse dynamische Objekte und sie repräsentieren sequentielle Aktivitäten im Computer.

Auf einem modernen Computersystem sind immer mehrere Prozesse in Ausführung. Der Hauptprozessor wird beim Mehrprogrammbetrieb im raschen Wechsel zwischen den Prozessen hin- und hergeschaltet.

Jeder Prozess umfasst außer dem Programmcode noch seinen *Prozesskontext*, der von den Kontexten anderer Prozesse unabhängig ist. Betriebssysteme verwalten drei Arten von Kontextinformationen: *Hardwarekontext*, *Systemkontext* und *Benutzerkontext*.

8.1

Process Context

The *hardware context* contains the contents of the registers in the CPU during process execution. Some of these registers have already been described in Section 4.4.1. Registers whose content the operating system needs to back up in the event of a process change are, among others:

- Program counter (instruction pointer) – stores the memory address of the next command to be executed
- Stack pointer – stores the memory address at the end of the stack

Prozesskontext

Der *Hardwarekontext* sind die Inhalte der Register im Hauptprozessor zum Zeitpunkt der Prozessausführung. Einige dieser Register wurden bereits in Abschnitt 4.4.1 vorgestellt. Register, deren Inhalt das Betriebssystem bei einem Prozesswechsel sichern muss, sind unter anderem:

- Befehlszähler (*Program Counter*, *Instruction Pointer*) – enthält die Speicheradresse des nächsten auszuführenden Befehls
- Stack pointer – enthält die Speicheradresse am Ende des Stacks

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_8

- Base pointer – points to an address in the stack
- Instruction register – stores the instruction which is currently executed
- Accumulator – stores operands for the ALU and their results
- Page-table base register – stores the address of the page table of the running process (see Section 5.3.3)
- Page-table length register – stores the length of the page table of the running process (see Section 5.3.3)

The *system context* is the information, the operating system stores about a process. Examples are:

- Process table record
- Process ID (PID)
- Parent Process ID (PPID)
- Process state
- Priority
- Identifiers = access credentials to resources
- Quotas = allowed usage quantity of individual resources
- Runtime
- Open files
- Assigned devices

- Base pointer – zeigt auf eine Adresse im Stack
- Befehlsregister (*Instruction Register*) – speichert den aktuellen Befehl
- Akkumulator – speichert Operanden für die ALU und deren Resultate
- Page-Table Base Register – enthält die Adresse, bei der die Seitentabelle des laufenden Prozesses anfängt (siehe Abschnitt 5.3.3)
- Page-Table Length Register – enthält die Länge der Seitentabelle des laufenden Prozesses (siehe Abschnitt 5.3.3)

Der *Systemkontext* umfasst die Informationen, die das Betriebssystem über einen Prozess speichert. Beispiele sind:

- Eintrag in der Prozesstabelle
- Prozessnummer (PID)
- Elternprozessnummer (PPID)
- Prozesszustand
- Priorität
- Zugriffsrechte auf Ressourcen
- Erlaubte Nutzungsmengen (englisch: *Quotas*) einzelner Ressourcen
- Laufzeit
- Geöffnete Dateien
- Zugeordnete Geräte

The *user context* contains the pages in the allocated virtual memory address space (see Section 5.3.2).

For managing the processes, the operating system implements, the *process table*, which is a list of all existing processes. It contains a record for each process, which is called *process control block* (see Figure 8.1). If the CPU switches from one process to another one, the operating system stores the hardware context and the system context of the running process in the process table record. If a process gets assigned to the CPU, its context is restored by using the content of the process control block (see Figure 8.2).

Der *Benutzerkontext* sind die Seiten im zugewiesenen Adressraum des virtuellen Speichers (siehe Abschnitt 5.3.2).

Zur Verwaltung der Prozesse führt das Betriebssystem mit der *Prozesstabelle* eine Liste aller existierenden Prozesse. Diese enthält für jeden Prozess einen Eintrag, den *Prozesskontrollblock* (siehe Abbildung 8.1). Dort speichert das Betriebssystem beim Prozesswechsel den Hardwarekontext und den Systemkontext des jeweiligen Prozesses. Erhält ein Prozess Zugriff auf den Hauptprozessor, wird sein Kontext mit dem Inhalt des Prozesskontrollblocks wiederhergestellt (siehe Abbildung 8.2).

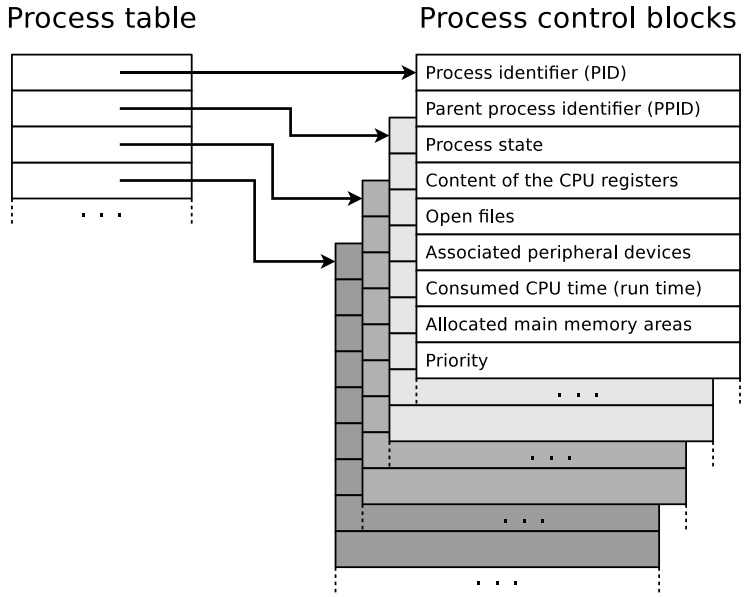

Figure 8.1: The Process Table contains Pointers to the Process Control Blocks

8.2

Process States

Every process is at any moment in a particular *state*. The number of different states depends on the process state model of the operating system. Generally, the process states *running*, for the process to which the CPU is assigned to,

Prozesszustände

Jeder Prozess befindet sich zu jedem Zeitpunkt in einem bestimmten *Zustand*. Wie viele unterschiedliche Zustände es gibt, hängt vom Zustands-Prozessmodell des verwendeten Betriebssystems ab. Prinzipiell genügen der Pro-

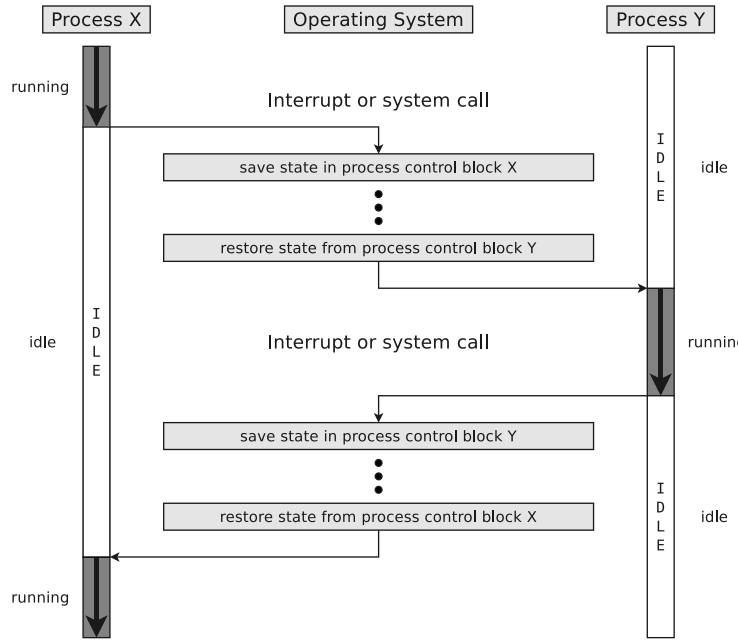

Figure 8.2: If a Process gets the CPU assigned, its Context gets restored by using the Content of the Process Control Block

and *idle*, for the processes that are waiting for the assignment of the CPU (see Figure 8.3) are sufficient [108].

zesszustand *rechnend* (englisch: *running*) für den Prozess, dem der Hauptprozessor zugeteilt ist und der Zustand *untätig* (englisch: *idle*) für die Prozesse, die auf die Zuteilung des Prozessors warten (siehe Abbildung 8.3) [107].

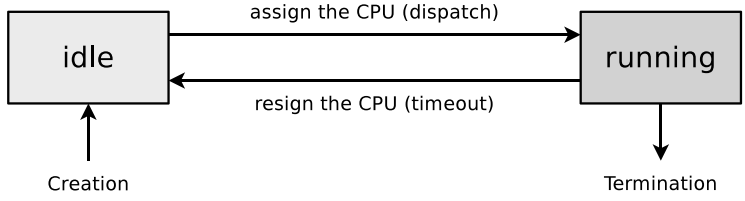

Figure 8.3: Generally, two Process States are sufficient for simple Process Management

For implementing a *2-state process model*, as shown in Figure 8.3, the processes in *idle* state must be stored in a queue (see Figure 8.4) in which they wait for execution. The queue is sorted by an algorithm that takes the process priority and/or the waiting time into account.

Um ein in Abbildung 8.3 gezeigtes *2-Zustands-Prozessmodell* zu realisieren, müssen die Prozesse im Zustand *untätig* in einer Warteschlange (siehe Abbildung 8.4) gespeichert werden, in der sie auf ihre Ausführung warten. Die Liste wird nach einem Algorithmus sortiert, der sinnvollerweise die Prozesspriorität und/oder die Wartezeit berücksichtigt.

The priority (proportional computing power) in Linux has a value from −20 to +19 (in integer steps). The value −20 is the highest priority, 19 is the lowest priority [67]. The default priority is 0. Standard users can assign priorities from 0 to 19. The system administrator (`root`) can also assign negative values to processes. The priority of a process can be specified in Linux when starting the process with the command `nice`. Modifying the priority of an already existing process can be done with the `renice` command.

Die Priorität (anteilige Rechenleistung) hat unter Linux einen Wert von −20 bis +19 (in ganzzahligen Schritten). Der Wert −20 ist die höchste Priorität und 19 die niedrigste Priorität [67]. Die Standardpriorität ist 0. Normale Benutzer können Prioritäten von 0 bis 19 vergeben. Der Systemverwalter (`root`) darf Prozessen auch negative Werte zuweisen. Die Priorität eines Prozesses kann unter Linux beim Start des Prozesses mit dem Kommando `nice` angebeben werden. Die Veränderung der Priorität eines bereits existierenden Prozesses ist mit dem Kommando `renice` möglich.

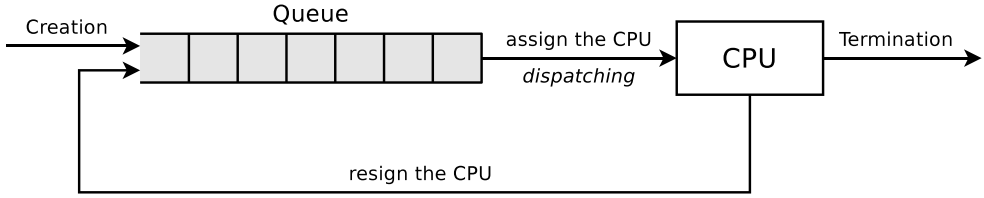

Figure 8.4: The Processes in *idle* State are stored in a Queue

The model, shown in Figure 8.4, also demonstrates the working method of the *dispatcher*. Its task is to carry out the state transitions. The execution order of the processes is specified by the *scheduler*, which uses a *scheduling algorithm* (see Section 8.6).

The process model with two states has the advantage that it is simple to implement, but it also has a conceptual flaw. It assumes that all processes are ready to run at any time. In practice, this is not the case. In a multitasking operating system, there are always processes that are *blocked*, which can be caused by different reasons. Possible reasons include, for example, that a process is waiting for the input/output of a device, the result of another process, the occurrence of a synchronization event, or user interaction. Therefore, it makes sense to categorize idle processes into two groups, namely those processes which are in the *ready* state and those which are in the *blocked* state. Figure 8.5 shows the resulting *3-state process model*.

One way to implement a 3-state process model, as shown in Figure 8.5, is by using two

Das Modell in Abbildung 8.4 zeigt auch die Arbeitsweise des *Dispatchers*. Dessen Aufgabe ist die Umsetzung der Zustandsübergänge. Die Ausführungsreihenfolge der Prozesse legt der *Scheduler* fest, der einen *Scheduling-Algorithmus* (siehe Abschnitt 8.6) verwendet.

Das 2-Zustands-Prozessmodell hat zweifellos den Vorteil, dass es sehr einfach realisierbar ist, es hat aber auch einen konzeptionellen Fehler. Dieser besteht in der Annahme, dass alle Prozesse jederzeit zur Ausführung bereit sind. In der Praxis ist das allerdings nicht der Fall, denn es gibt in einem Betriebssystem mit Mehrprogrammbetrieb auch immer Prozesse, die *blockiert* sind, was unterschiedliche Gründe haben kann. Mögliche Gründe können zum Beispiel sein, dass ein Prozess auf die Ein-/Ausgabe eines Geräts, das Ergebnis eines anderen Prozesses, das Eintreten eines Synchronisationsereignisses oder die Reaktion des Benutzers wartet. Darum ist es sinnvoll, die untätigen Prozesse in zwei Gruppen zu unterscheiden, nämlich in diejenigen Prozesse, die im Zustand *bereit* (englisch: *ready*) sind und diejenigen im Zustand *blockiert* (englisch: *blocked*). Abbildung 8.5 zeigt das resultierende *3-Zustands-Prozessmodell*.

Eine Möglichkeit, um ein 3-Zustands-Prozessmodell wie in Abbildung 8.5 zu realisie-

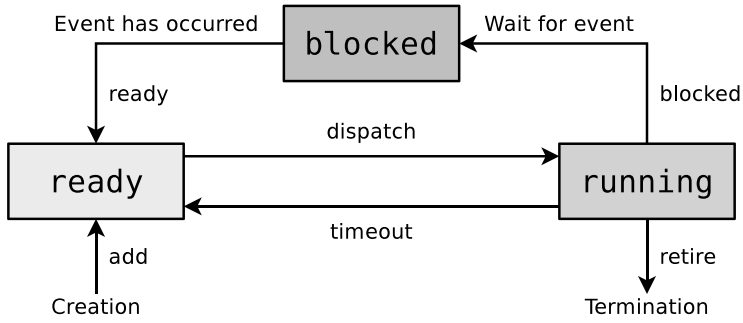

Figure 8.5: The 3-State Process Model takes into Account that not all Processes are ready to run at any Time

queues. One queue is used for processes in the *ready* state and the other queue for processes in the *blocked* state (see Figure 8.6).

ren, ist die Verwendung von zwei Warteschlangen. Eine Warteschlange würde die Prozesse im Zustand *bereit* aufnehmen und die andere die Prozesse im Zustand *blockiert* (siehe Abbildung 8.6).

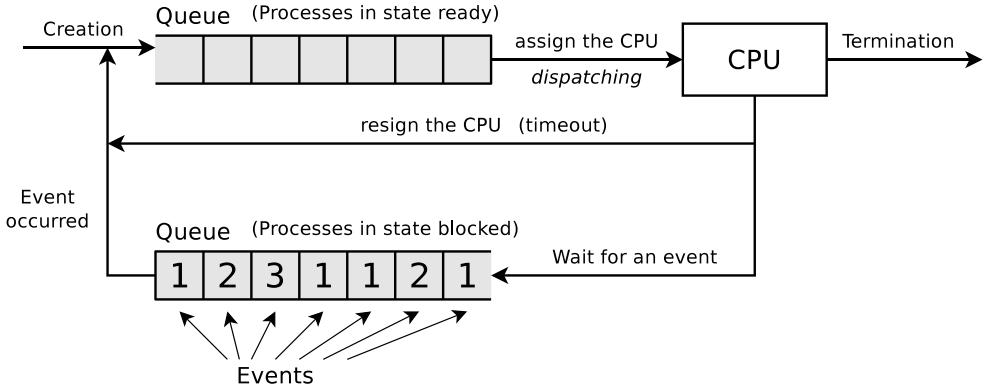

Figure 8.6: In Principle, two Queues are sufficient for implementing the 3-State Process Model [108]

However, it makes more sense to use multiple queues for the blocked processes, as shown in Figure 8.7. Modern operating systems such as Linux work according to this concept. For each event for which at least a single process is waiting, the operating system creates a queue. Each time an event occurs, all processes in the corresponding queue are transferred to the queue that contains the processes in the *ready* state. One advantage of this concept is that the operating system does not have to check for all existing processes in the *blocked* state, whether

Sinnvoller ist allerdings so wie in Abbildung 8.7 dargestellt die Verwendung mehrerer Warteschlangen für die blockierten Prozesse. Nach diesem Konzept arbeiten moderne Betriebssysteme wie zum Beispiel Linux in der Praxis. Für jedes Ereignis, auf das mindestens ein Prozess wartet, legt das Betriebssystem eine Warteschlange an. Tritt ein Ereignis ein, werden alle in der entsprechenden Warteschlange befindlichen Prozesse in die Warteschlange mit den Prozessen im Zustand *bereit* überführt. Ein Vorteil dieses Konzepts ist, dass das Betriebssystem nicht für alle existierenden Prozesse im Zustand

an event that has occurred if it is relevant for them [108].

blockiert überprüfen muss, ob ein eingetretenes Ereignis auf sie zutrifft [107].

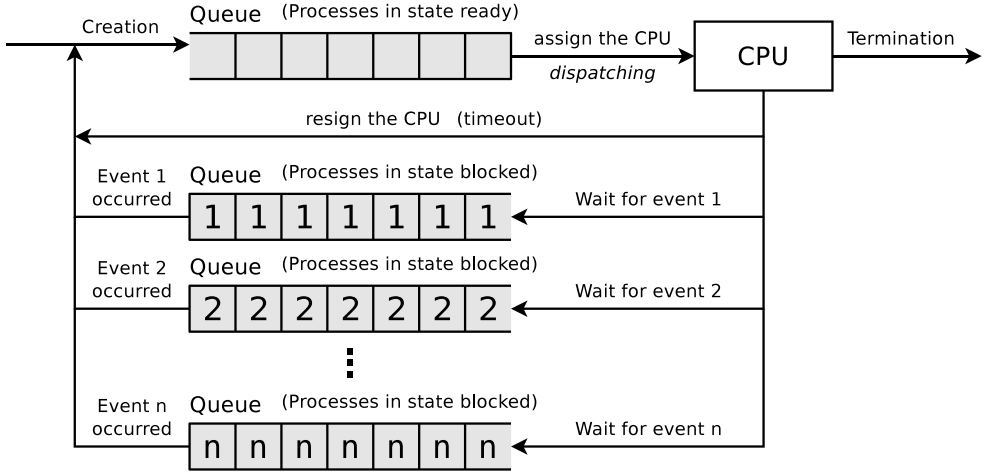

Figure 8.7: Modern Operating Systems use multiple Queues for blocked Processes [108]

During state transition, the process control block of the affected process is removed from the old state list and inserted into the new state list. No list exists for processes in the `running` state.

Because it can be useful to limit the number of processes on low-end computers with few resources to save memory, and to specify the degree of multitasking, it is useful to expand the 3-state process model by two additional process states to the *5-state process model*. In the *new* state, there are processes whose process control block has already been created by the operating system, but which are not yet added to the queue for processes in the *ready* state.

A process that is in the *exit* state has already finished execution or was terminated, but for various reasons, its record in the process table, and thus its process control block, has not yet been removed.

If not enough physical memory is available for all processes, parts of processes must be swapped to the swap memory. If the operating system can outsource processes that are in the *blocked* state, the 5-state process model must be extended by an additional process state, *suspended*, to the *6-state process model*

Beim Zustandsübergang wird der Prozesskontrollblock des betreffenden Prozesses aus der alten Zustandsliste entfernt und in die neue Zustandsliste eingefügt. Für Prozesse im Zustand `rechnend` existiert keine Liste.

Weil es auf Computern mit geringer Ressourcenausstattung sinnvoll sein kann, die Anzahl der ausführbaren Prozesse zu limitieren, um Speicher zu sparen und den Grad des Mehrprogrammbetriebs festzulegen, ist es sinnvoll, das 3-Zustands-Prozessmodell um zwei weitere Prozesszustände zum *5-Zustands-Prozessmodell* zu erweitern. Im Zustand *neu* (englisch: *new*) sind diejenigen Prozesse, deren Prozesskontrollblock das Betriebssystem zwar bereits erzeugt hat, die aber noch nicht in die Warteschlange für Prozesse im Zustand *bereit* eingefügt sind.

Ein Prozess im Zustand *beendet* (englisch: *exit*) ist bereits fertig abgearbeitet oder wurde abgebrochen, sein Eintrag in der Prozesstabelle und damit sein Prozesskontrollblock wurden aus verschiedenen Gründen aber noch nicht entfernt.

Ist nicht genügend physischer Hauptspeicher für alle Prozesse verfügbar, müssen Teile von Prozessen auf den Auslagerungsspeicher (*Swap*) ausgelagert werden. Soll das Betriebssystem in der Lage sein, bei Bedarf Prozesse auszulagern, die im Zustand *blockiert* sind, muss das 5-Zustands-Prozessmodell um einen weiteren

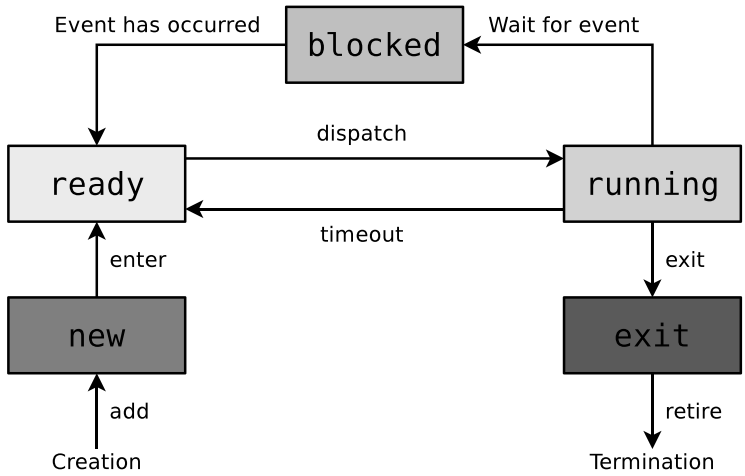

Figure 8.8: The 5-State Process Model allows to specify the Degree of Multitasking and takes into Account that Processes may have finished Execution but have not yet been erased

(see Figure 8.9). As a result, additional main memory capacity is available for the processes in the states *running* and *ready*.

Prozesszustand, *suspendiert* (englisch: *suspended*), zum *6-Zustands-Prozessmodell* (siehe Abbildung 8.9) erweitert werden. Dadurch steht den Prozessen in den Zuständen *rechnend* und *bereit* mehr Hauptspeicher zur Verfügung.

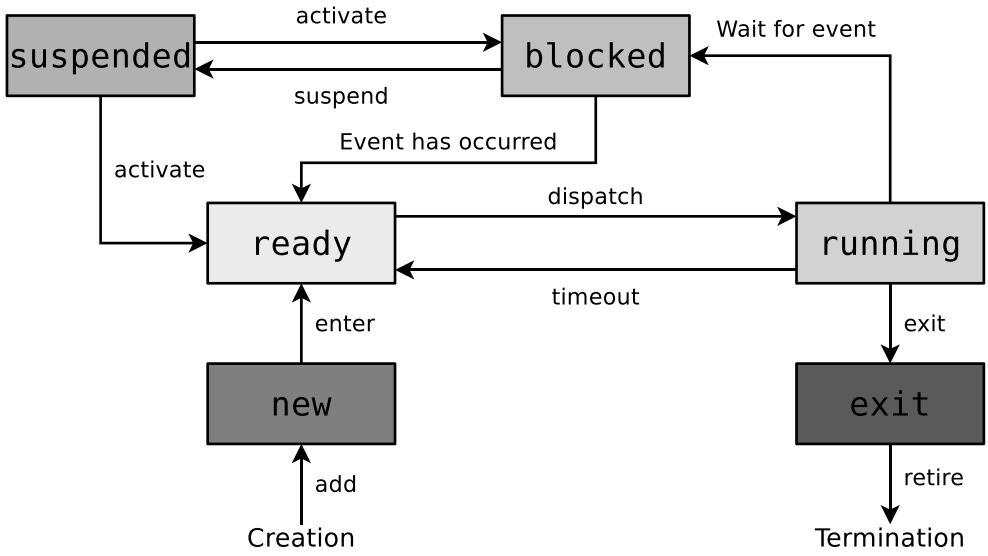

Figure 8.9: With the 6-State Process Model, it is possible to outsource Parts of Processes to the Swap Memory

If a process has been suspended, it is better to use the freed up space in main memory to activate an outsourced process instead of assigning it to a new process. However, this is only useful if the activated process is no longer blocked. The 6-state process model cannot classify the outsourced processes into blocked suspended, and ready suspended processes. Therefore, it makes sense to extend the 6-state process model with an additional process state to the *7-state process model* (see Figure 8.10).

Wurde ein Prozess suspendiert, ist es besser, den frei gewordenen Platz im Hauptspeicher zu verwenden, um einen ausgelagerten Prozess zu aktivieren, als ihn einem neuen Prozess zuzuweisen. Das ist aber nur dann sinnvoll, wenn der aktivierte Prozess nicht mehr blockiert ist. Im 6-Zustands-Prozessmodell fehlt die Möglichkeit, die ausgelagerten Prozesse in blockierte und nicht-blockierte ausgelagerte Prozesse zu unterscheiden. Darum ist es sinnvoll, das 6-Zustands-Prozessmodell um einen weiteren Prozesszustand zum *7-Zustands-Prozessmodell* zu erweitern (siehe Abbildung 8.10).

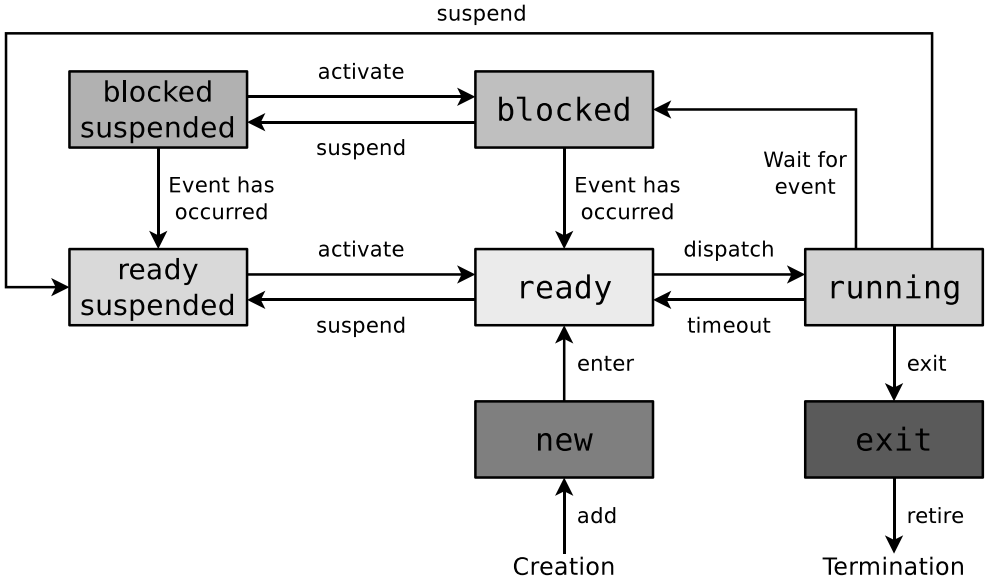

Figure 8.10: The 7-State Process Model differentiates between blocked suspended and ready suspended Processes

The process model of Linux (see Figure 8.11) is very similar to the 7-state process model described here, as shown in Figure 8.10. The most obvious difference is that the state *running* is divided into the two states *user running* for processes in user mode and *kernel running* for processes in kernel mode. Furthermore, the state *finished* in Linux is called *zombie*. A *zombie process* has completed execution (via system call `exit`). However, its record in the process table still exists until the parent process has requested the return value (via system call `wait`).

Das Prozessmodell von Linux (siehe Abbildung 8.11) kommt dem in diesem Abschnitt entwickelten 7-Zustands-Prozessmodell in Abbildung 8.10 schon sehr nahe. Der offensichtlichste Unterschied ist, dass der Zustand *rechnend* in der Praxis in die beiden Zustände *benutzer rechnend* (englisch: *user running*) für Prozesse im Benutzermodus und *kernel rechnend* (englisch: *kernel running*) für Prozesse im Kernelmodus unterteilt ist. Zudem heißt der Zustand *beendet* unter Linux *Zombie*. Ein *Zombie-Prozess* ist zwar fertig abgearbeitet (via Systemaufruf `exit`), aber sein Eintrag in der Prozesstabelle

existiert noch so lange, bis der Elternprozess den Rückgabewert (via Systemaufruf `wait`) abgefragt hat.

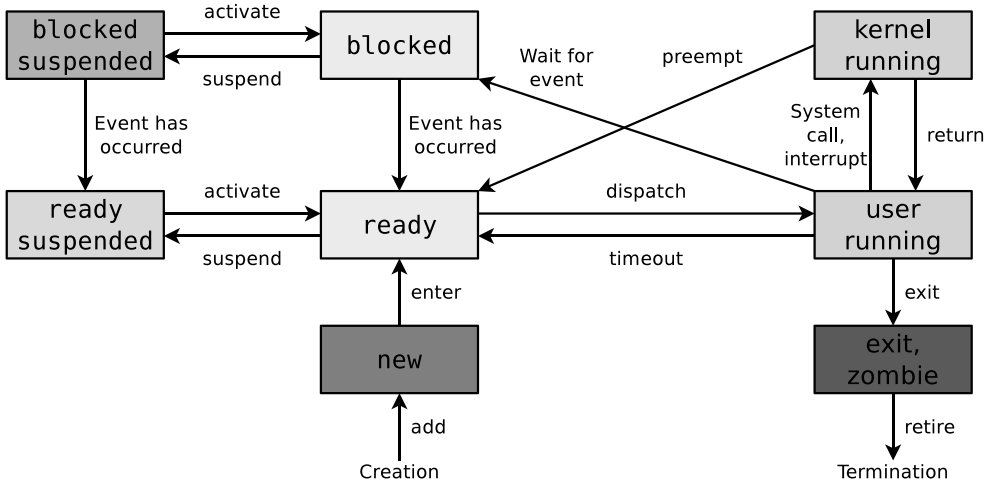

Figure 8.11: Process State Model of Linux with eight States

8.3

Structure of a Process in Memory

The structure of processes in the main memory (see Figure 8.12) is described in this section by using the Linux operating system as an example [29]. By default, Linux uses 25% of the virtual address space on a 32-bit system for the operating system kernel (kernel mode) and 75% for the processes in user mode. In such a system, each running process can use up to 3 GB of memory. Section 5.3.6 already described the limitations of 64-bit systems. The structure of processes on 64-bit systems is identical to that on 32-bit systems. The only difference is that the address space is larger and so is the possible expansion of the processes in memory.

The user space in the memory structure of the processes (see Figure 8.12) is the user context (see Section 8.1). It is the virtual address space

Struktur eines Prozesses im Speicher

Dieser Abschnitt beschreibt die Struktur von Prozessen im Hauptspeicher (siehe Abbildung 8.12) anhand des Betriebssystems Linux [29]. Bei der standardmäßigen Aufteilung des virtuellen Adressraums auf einem 32-Bit-System reserviert Linux standardmäßig 25% für den Betriebssystemkern (Kernelmodus) und 75% für die Prozesse im Benutzermodus. Auf solchen Systemen kann jeder laufende Prozess bis zu 3 GB Speicher verwenden. In Abschnitt 5.3.6 wurden bereits die Grenzen von 64-Bit-Systemen beschrieben. Die Struktur der Prozesse auf 64-Bit-Systemen unterscheidet sich nicht von der auf 32-Bit-Systemen. Lediglich der Adressraum ist größer und damit die mögliche Ausdehnung der Prozesse im Speicher.

Der Userspace in der dargestellten Speicherstruktur der Prozesse (siehe Abbildung 8.12) entspricht dem Benutzerkontext (siehe Abschnitt 8.1). Das ist der vom Betriebssystem

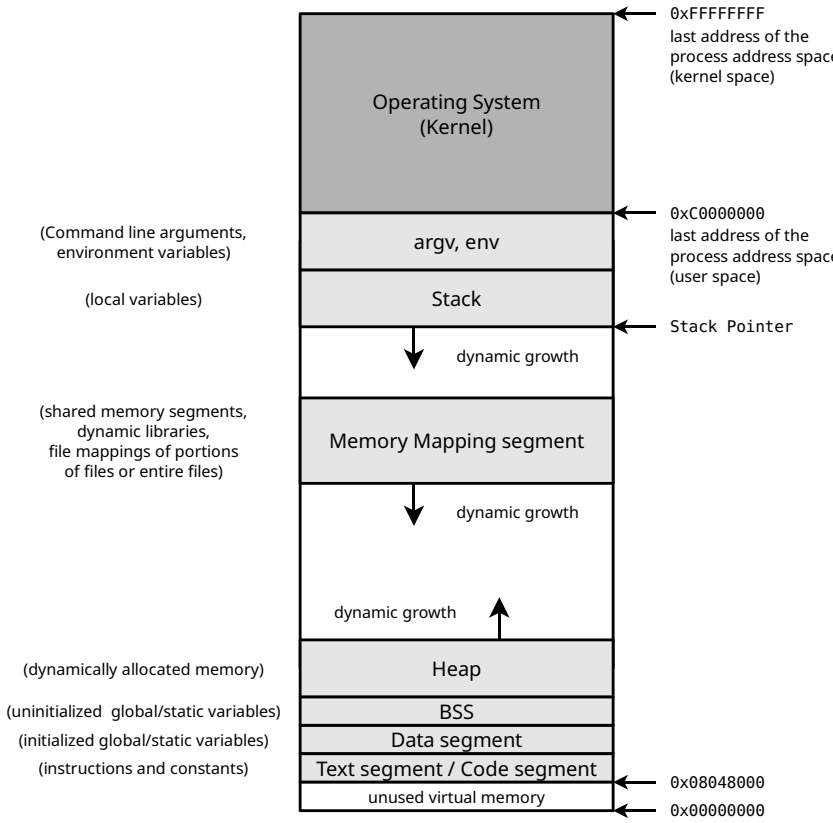

Figure 8.12: Structure of a Linux Process in the Memory of a 32-Bit System

(virtual memory) allocated by the operating system (see Section 5.3.2).

The *text segment*, also called the *code segment*, contains the program code (machine code) and read-only data such as constants [37, 69]. An example of such a constant is `const int MAX = 100;` It can be shared by several processes and therefore only needs to be stored once in physical memory [47, 112]. Consequently, it is usually read-only [119]. The content of the text segment is read from the program file during process creation by the system call `exec` (see Section 8.5).

In order to understand the contents of the other memory areas, it is helpful to explain the different types of variables and their scopes. For global variables, the declaration, i.e. the assignment of name and data type, takes place outside

zugewiesene virtuelle Adressraum bzw. virtuelle Speicher (siehe Abschnitt 5.3.2).

Das *Textsegment*, auch *Codesegment* genannt, enthält den ausführbaren Programmcode (Maschinencode) und ausschließlich lesbare Daten wie Konstanten [37, 69]. Ein Beispiel für eine solche Konstante ist `const int MAX = 100;` Es kann von mehreren Prozessen geteilt werden und muss somit nur einmal im physischen Speicher vorgehalten werden [47, 115]. Darum ist es üblicherweise nur lesbar (englisch: *read only*) [119]. Den Inhalt des Textsegments liest der Systemaufruf `exec` (siehe Abschnitt 8.5) bei der Prozesserzeugung aus der Programmdatei.

Für das Verständnis der Inhalte der übrigen Speicherbereiche ist es sinnvoll die verschiedenen Arten von Variablen und deren Gültigkeitsbereiche zu wiederholen. Bei globalen Variablen findet die Deklaration, also die Zuweisung von

of functions. On the other hand, local variables are only valid within the function in which they are declared unless they are static. This means that the value of variables that are both local and static is preserved when the function is exited.

The *data segment* contains initialized variables (variables that are assigned an initial value) that are either global or local and static at the same time. One example of such a declaration is `int sum = 0;` The content of the data segment is read from the program file by `exec` during process creation.

The *BSS* (*Block Started by Symbol*) area contains the global variables and local static variables that are not initialized when the process is started, i.e. that they have no initial value assigned to them [50]. An example of such a declaration is `int i;` All variables in the BSS are initialized `exec` with the value 0 [115].

The *Heap* grows dynamically. The process can also allocate memory in this area dynamically at runtime. In C, this is done with the function `malloc` [119]. The standard library function `free` [5] allows to free memory in the heap. Unlike the text segment, data segment and BSS, the heap area can expand during the program's runtime.

The command-line arguments (`argv`) of the program call and the environment variables (`env`) are located in an area that starts at the far end of the user space [47].

The *stack* allows the implementation of nested function calls and operates according to the *Last In First Out* (LIFO) principle. Each function call places a data structure on the stack containing the call parameters, the return address, and a pointer to the calling function on the stack. Functions also add their local variables to the stack. When returning from a function, the data structure of the function is removed from the stack. Consequently, the stack can grow as the program runs. Although not shown in Figure 8.12, there are two separate stacks [119], one for user mode and one for kernel mode.

Name und Datentyp, außerhalb von Funktionen statt. Lokale Variablen hingegen sind nur innerhalb der Funktion gültig, in der sie deklariert werden, außer sie sind auch statisch. Das heißt, der Wert von Variablen, die lokal und statisch zugleich sind, bleibt auch beim Verlassen der Funktion erhalten.

Das *Datensegment* enthält initialisierte Variablen (d. h. Variablen, die einen Anfangswert zugewiesen bekommen), die entweder global sind oder lokal und zugleich statisch. Ein Beispiel für eine solche Deklaration ist `int summe = 0;` Auch den Inhalt des Datensegments liest `exec` bei der Prozesserzeugung aus der Programmdatei.

Der Bereich *BSS* (*Block Started by Symbol*) enthält diejenigen globalen Variablen und lokalen statischen Variablen, die beim Start des Prozesses nicht initialisiert werden, die also keinen Anfangswert zugewiesen bekommen [50]. Ein Beispiel für eine solche Deklaration ist `int i;` Alle Variablen im BSS initialisiert `exec` mit dem Wert 0 [115].

Der *Heap* wächst dynamisch. Hier kann ein Prozess dynamisch zur Laufzeit Speicher allokieren. In C geschieht dies mit der Standard-Bibliotheksfunktion `malloc` [119]. Das Freigeben von Speicher im Heap ermöglicht die Standard-Bibliotheksfunktion `free` [5]. Der Bereich Heap kann also im Gegensatz zum Textsegment, Datensegment und BSS während der Laufzeit eines Programms wachsen.

Kommandozeilenargumente (`argv`) des Programmaufrufs und die Umgebungsvariablen (`env`) liegen in einem Bereich, der am äußersten Ende des Userspace beginnt [47].

Der *Stack* ermöglicht die Realisierung geschachtelter Funktionsaufrufe und arbeitet nach dem Prinzip *Last In First Out* (LIFO). Mit jedem Funktionsaufruf wird eine Datenstruktur auf den Stack gelegt, die die Aufrufparameter, die Rücksprungadresse und einen Zeiger auf die aufrufende Funktion im Stack enthält. Die Funktionen legen auch ihre lokalen Variablen auf den Stack. Beim Rücksprung aus einer Funktion wird die Datenstruktur der Funktion aus dem Stack entfernt. Der Stack kann also während der Laufzeit eines Programms wachsen. Auch wenn es aus Abbildung 8.12 nicht ersichtlich ist, existieren für den Benutzermodus und

Figure 8.12 also shows the *Memory Mapping* area, which is located in the address space between the stack and the heap. This is where shared libraries are loaded. Shared memory areas are also mapped here (see Table 9.11) [115]. In addition, whole files or areas of files can be mapped here with the system call `mmap` (see Table 9.3) [47].

In Linux operating systems, there is a file `/proc/<pid>/maps` for each process. This file contains information about the virtual address ranges of the process `<pid>`, their access privileges, and what they are used for (heap, stack, files, dynamic libraries...).

Since the text segment, data segment, and BSS areas (at least when all global and uninitialized variables are initially set to 0) are read from the program file, their size is already fixed before the process is started. In Linux, the `size` command-line tool returns the size (in bytes) of the text segment, data segment, and BSS of program files [47].

den Kernelmodus zwei voneinander getrennte Stacks [119].

Abbildung 8.12 zeigt auch den Speicherbereich *Memory Mapping*, der sich im Adressraum zwischen Stack und Heap befindet. In diesem Bereich werden dynamische Bibliotheken (englisch: *Shared Libraries*) geladen. Auch gemeinsame Speicherbereiche sind hier abgebildet (siehe Tabelle 9.11) [115]. Zusätzlich können komplette Dateien oder Bereiche von Dateien hier mit dem Systemaufruf `mmap` abgebildet werden (siehe Tabelle 9.3) [47].

Bei Linux-Betriebssystemen existiert für jeden Prozess die Datei `/proc/<pid>/maps`. Diese enthält Informationen über die virtuellen Adressbereiche des Prozesses `<pid>`, deren Zugriffsrechte und wofür sie verwendet werden (Heap, Stack, Dateien, dynamische Bibliotheken...).

Da die Bereiche Textsegment, Datensegment und BSS (zumindest im Zustand, dass alle globalen und nicht initialisierten Variablen initial den Wert 0 haben) beim Start eines Prozesses aus der Programmdatei gelesen werden, steht deren Größe schon vor dem Start eines Prozesses fest. Das Kommando `size` gibt unter Linux die Größe (in Bytes) von Textsegment, Datensegment und BSS von Programmdateien aus [47].

```
$ size /bin/c*
   text    data     bss     dec     hex filename
  46480     620    1480   48580    bdc4 /bin/cat
   7619     420      32    8071    1f87 /bin/chacl
  55211     592     464   56267    dbcb /bin/chgrp
  51614     568     464   52646    cda6 /bin/chmod
  57349     600     464   58413    e42d /bin/chown
 120319     868    2696  123883   1e3eb /bin/cp
 131911    2672    1736  136319   2147f /bin/cpio
```

The process structure in Figure 8.12 appears clear and well organized. In practice, however, the pages that are part of a process (see Figure 8.13), are scattered due to the use of virtual memory. The pages are stored in a fragmented form in main memory and possibly also in swap memory.

In Abbildung 8.12 erscheint die Struktur der Prozesse klar und aufgeräumt. In der Praxis hingegen sind die Seiten, die Teil eines Prozesses sind, wie in Abbildung 8.13 zu sehen, durch den virtuellen Speicher in unzusammenhängender Weise im Hauptspeicher und eventuell auch im Auslagerungsspeicher (Swap) verteilt.

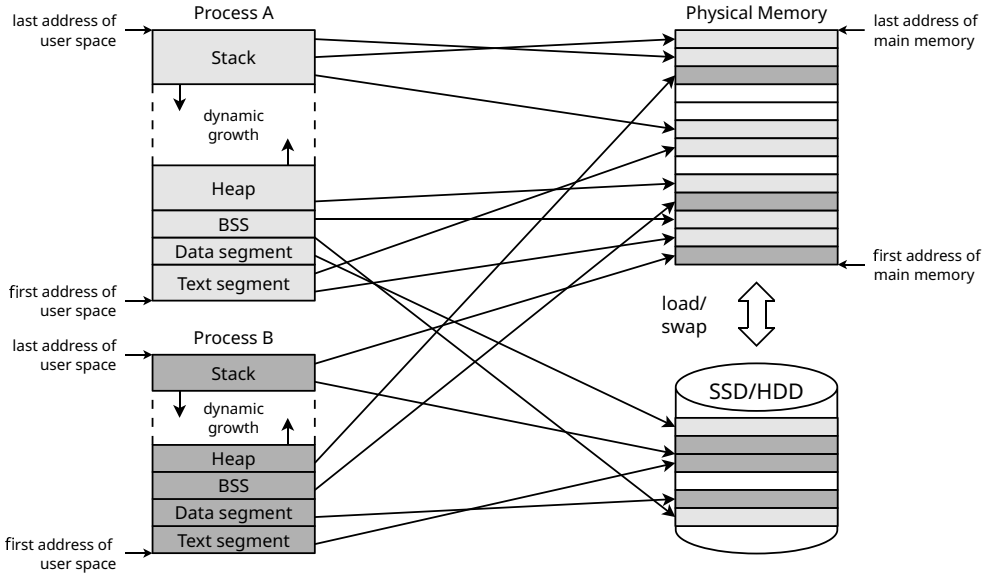

Figure 8.13: Processes are stored in Physical Memory by Virtual Memory, not in a continuous manner and not always in Main Memory [109]

8.4

Process Creation via fork

The system call `fork` is the standard way to create a new process in Linux and other Unix-like operating systems. When a process calls `fork`, the operating system creates an identical copy of that process. The calling process, in this case, is called the *parent process*. The new process is called the *child process*, and after its creation, it has the same program code as the parent process. Even the program counters refer to the same program code line. As with all other processes, the memory areas of the child and parent processes are separated from each other. In other words, the child process and the parent process both have their process context (see Section 8.1).

There is a variant of `fork` named `vfork`, which does not copy the address space of the parent process, and therefore causes less overhead

Prozesse erzeugen mit fork

Der Systemaufruf `fork` ist unter Linux und anderen Unix-(ähnlichen) Betriebssystemen die üblicherweise verwendete Möglichkeit, einen neuen Prozess zu erzeugen. Ruft ein Prozess `fork` auf, erzeugt das Betriebssystem eine identische Kopie dieses Prozesses. Der aufrufende Prozess heißt in diesem Kontext *Elternprozess* (englisch: *Parent Process*) und in der deutschsprachigen Literatur manchmal *Vaterprozess* [35, 97]. Der neu erzeugte Prozess heißt *Kindprozess* und er hat nach der Erzeugung den gleichen Programmcode wie der Elternprozess. Auch die Befehlszähler haben den gleichen Wert, verweisen also auf die gleiche Zeile im Programmcode. Die Speicherbereiche von Kindprozess und Elternprozess sind, wie bei allen anderen Prozessen auch, streng voneinander getrennt. Kurz gesagt: Kindprozess und Elternprozess besitzen ihren eigenen Prozesskontext (siehe Abschnitt 8.1).

Mit `vfork` existiert eine Variante von `fork`, die nicht den Adressraum des Elternprozesses kopiert, und somit weniger Verwaltungsaufwand

than `fork`. Using `vfork` is useful if the child process is to be replaced by another process immediately after its creation (see Section 8.5) [47].

A process that is based on the C source code in Listing 8.1 calls the standard library function for the system call `fork`. This results in the creation of an exact copy of the calling process. In the source code, the two processes can only be identified using the return value of `fork`. If the return value is negative, an error occurred when attempting to create the process. If the return value is a positive integer, it is the parent process, and the return value is the process ID (PID) of the child process. If the return value has the value 0, it is the child process.

als `fork` verursacht. Die Verwendung von `vfork` ist sinnvoll, wenn der Kindprozess direkt nach seiner Erzeugung durch einem anderen Prozess ersetzt werden soll (siehe Abschnitt 8.5) [47].

Ein Prozess, der auf dem C-Quellcode in Listing 8.1 basiert, ruft die Standard-Bibliotheksfunktion für den Systemaufruf `fork` auf. Dadurch wird eine exakte Kopie des Prozesses erzeugt. Im Quellcode ist eine Unterscheidung der beiden Prozesse nur anhand des Rückgabewerts von `fork` möglich. Ist der Rückgabewert negativ, gab es beim Versuch der Prozesserzeugung einen Fehler. Ist der Rückgabewert eine positive ganze Zahl, handelt es sich um den Elternprozess und der Rückgabewert entspricht der Prozessnummer (PID) des Kindprozesses. Hat der Rückgabewert den Wert 0, handelt es sich um den Kindprozess.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6   // Create a child process
7   int return_value = fork();
8
9   if (return_value < 0) {
10    // Return value of fork = negative number --> error
11    // Memory or processes table might be full
12    printf("An error occurred.\n");
13  }
14
15  if (return_value > 0) {
16    // Return value of fork = positive number --> parent process
17    // Return value = PID of the ne child process
18    printf("Parent: Here is the parent process.\n");
19    printf("Parent: PID of the child: %i\n", return_value);
20  }
21
22  if (return_value == 0) {
23    // Return value of fork = 0 --> child process
24    printf("Child: Here is the child process.\n");
25  }
26 }
```

Listing 8.1: The System Call `fork` and the Standard Library Function of the same Name create a Copy of the calling Process

After compiling the program with the GNU C compiler (`gcc`) and running it, the output, in most cases, has a form like this:

Nach dem Übersetzen des Programms mit dem GNU C Compiler (`gcc`) und dem anschließenden Ausführen, hat die Ausgabe in den meisten Fällen eine Form wie diese:

```
$ gcc Listing_8_1_fork.c -o Listing_8_1_fork
$ ./Listing_8_1_fork
Parent: Here is the parent process.
Parent: PID of the child: 49724
Child: Here is the child process.
```

By creating more new child processes with `fork`, an unrestricted deep tree of processes is created. The output of the `pstree` command gives an overview of this *process hierarchy*. It returns an overview of the running processes in Linux according to their parent/child relationships. Another useful command is `ps -ef`. It returns all running processes of the system, including the columns PID (Process ID) and PPID (Parent Process ID), which inform about the parent/child relationships.

The example in Listing 8.2 demonstrates by using the C programming language that parent and child processes are always independent of each other and have different memory areas. When the program is executed, it creates an identical copy as a child process using the standard library function for the `fork` system call. Parent process and child process each use a `for` loop to increment a counter variable `i`, starting from value 0 to value 5,000,000.

Durch das Erzeugen immer neuer Kindprozesse mit `fork` entsteht ein beliebig tiefer Baum von Prozessen. Eine Übersicht über diese *Prozesshierarchie* ermöglicht das Kommando `pstree`. Dieses gibt die laufenden Prozesse unter Linux entsprechend ihrer Eltern-/Kind-Beziehungen aus. Ein weiteres hilfreiches Kommando ist `ps -ef`. Dieses gibt alle laufenden Prozesse im System inklusive der Spalten PID (Process ID) und PPID (Parent Process ID) aus, die Aufschluss über die Eltern-/Kind-Beziehungen geben.

Das Beispiel in Listing 8.2 in der Programmiersprache C demonstriert, dass Eltern- und Kindprozess immer unabhängig voneinander sind und unterschiedliche Speicherbereiche verwenden. Wird das Programm ausgeführt, erzeugt es mit Hilfe der Standard-Bibliotheksfunktion für den Systemaufruf `fork` eine identische Kopie als Kindprozess. Elternprozess und Kindprozess zählen jeweils mit Hilfe einer `for`-Schleife eine Zählvariable `i` vom Wert 0 bis zum Wert 5.000.000 hoch.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6   int i;
7   if (fork()) {
8     // Parent process source code
9     for (i = 0; i < 5000000; i++) {
10      printf("\n Parent: %i", i);
11    }
12  } else {
13    // Child process source code
14    for (i = 0; i < 5000000; i++) {
15      printf("\n Child : %i", i);
16    }
17  }
18 }
```

Listing 8.2: Parent and Child Processes are independent of each other

The output of the program demonstrates the process switching. The value of the loop variable

An der Ausgabe des Programms sind die Prozesswechsel zu sehen. Der Wert der Schleifenva-

i proves that parent and child processes are independent of each other. Furthermore, the result of the execution cannot be reproduced since the exact times of the process switching depend on various factors such as the number of running processes, the process priorities, and the number of CPU cores. Therefore, they are unpredictable.

riablen i beweist, dass Eltern- und Kindprozess unabhängig voneinander sind. Zudem ist das Ergebnis der Ausführung nicht reproduzierbar, da die genauen Zeitpunkte der Prozesswechsel von verschiedenen Faktoren wie der Anzahl der laufenden Prozesse, der Prozessprioritäten und der Anzahl der Rechenkerne abhängen und somit unvorhersehbar sind.

```
$ gcc Listing_8_2_fork.c -o Listing_8_2_fork
$ ./Listing_8_2_fork
Child : 0
Child : 1
...
Child : 21019
Parent: 0
...
Parent: 50148
Child : 21020
...
Child : 129645
Parent: 50149
...
Parent: 855006
Child : 129646
...
```

For obtaining a similar output on a multi-core processor system, it is necessary to enforce the execution of Listing 8.2 on a single processor core. This can be achieved in Linux operating systems with the taskset command, which allows, among other things, to specify the CPU allocation of processes. The following command assigns the process from Listing 8.2 to the CPU core number 1.

Um auf einem System mit einem Mehrkernprozessor eine vergleichbare Ausgabe zu erhalten, ist es nötig die Ausführung von Listing 8.2 auf einem einzelnen Prozessorkern zu erzwingen. Dieses ist unter Linux-Betriebssystemen mit dem Kommando taskset möglich, das es u.a. erlaubt den CPU-Bezug von Prozessen zu definieren. Das folgende Kommando weist den aus Listing 8.2 resultierenden Prozess dem Prozessorkern mit der Nummer 1 zu.

```
$ taskset --cpu-list 1 ./Listing_8_2_fork
```

The previous examples of process generation have demonstrated that every child process has a parent process. Furthermore, the process state *zombie* in Section 8.2 shows that a process does not terminate before its parent has requested the return value. Each process in the operating system is, therefore, a child process to which a single parent process must be assigned to at any time. It raises the question of what an operating system is supposed to do when a parent process terminates before a child process. The answer to this question is shown by the output of the

Die bisherigen Beispiele zur Prozesserzeugung haben gezeigt, dass jeder Kindprozess einen Elternprozess hat. Zudem ist anhand des Prozesszustands *Zombie* in Abschnitt 8.2 ersichtlich, das ein Prozess erst dann endgültig beendet ist, wenn sein Elternprozess den Rückgabewert abgefragt hat. Jeder Prozess im Betriebssystem ist also ein Kindprozess, dem zu jedem Zeitpunkt exakt einen Elternprozess zugewiesen sein muss. Das wirft die Frage auf, was ein Betriebssystem machen muss, wenn ein Elternprozess vor einem Kindprozess terminiert. Die Antwort darauf lie-

program example in Listing 8.3, which like the previous examples, is implemented in the C programming language.

fert das Programmbeispiel in Listing 8.3, das wie die vorherigen in der Programmiersprache C realisiert ist.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 void main() {
6   int pid_of_child;
7
8   pid_of_child = fork();
9
10  // An error occured --> program abort
11  if (pid_of_child < 0) {
12    perror("\n fork() caused an error!");
13    exit(1);
14  }
15
16  // Parent process
17  if (pid_of_child > 0) {
18    printf("\n Parent: PID: %i", getpid());
19    printf("\n Parent: PPID: %i", getppid());
20  }
21
22  // Child process
23  if (pid_of_child == 0) {
24    printf("\n Child:  PID: %i", getpid());
25    printf("\n Child:  PPID: %i", getppid());
26  }
27 }
```

Listing 8.3: Parent and Child Processes are independent of each other

The example in Listing 8.3 creates a child process by calling `fork`. Parent process and child process use the library functions for the system calls `getpid` and `getppid` to request the own process ID (PID) and the parent process ID (PPID). In most cases, the output looks like this:

After compiling the program with the GNU C compiler (`gcc`) and running it, the output, in most cases, has a form like this:

In Listing 8.3 erzeugt der laufende Prozess mit `fork` einen Kindprozess. Elternprozess und Kindprozess erfragen mit den Bibliotheksfunktionen für die Systemaufrufe `getpid` und `getppid` die eigene Prozessnummer (PID) und die Prozessnummer (PPID) des jeweiligen Elternprozesses.

Nach dem Übersetzen des Programms mit dem GNU C Compiler (`gcc`) und dem anschließenden Ausführen, hat die Ausgabe in den meisten Fällen eine Form wie diese:

```
$ gcc Listing_8_3_fork.c -o Listing_8_3_fork
$ ./Listing_8_3_fork
Parent: PID: 20952
Parent: PPID: 3904
Child:  PID: 20953
Child:  PPID: 20952
```

As expected, the output shows that the PPID (in this case, it has the value 20952) of the child process is the same as the PID of the parent process. However, in rare cases, the output has the following form:

Wie zu erwarten zeigt die Ausgabe, dass die PPID (in diesem Fall hat sie den Wert 20952) des Kindprozesses der PID des Elternprozesses entspricht. In seltenen Fällen hat die Ausgabe die folgende Form:

```
$ gcc Listing_8_3_fork.c -o Listing_8_3_fork
$ ./Listing_8_3_fork
Parent: PID: 20954
Parent: PPID: 3904
Child:  PID: 20955
Child:  PPID: 1
```

In such a case, the parent process terminated before the child process, and, as usual, the child process got the `init` process assigned as its new parent process. The `init` process is the first process in Linux and UNIX-like operating systems. It *adopts* orphaned processes automatically.

Since Linux Kernel 3.4 (2012) and Dragonfly BSD 4.2 (2015), it is also possible that other processes than `init` become the new parent process of an orphaned process.

A potential threat of the system call `fork` is *fork bombs*. A fork bomb is a malware that calls `fork` or a library function in an infinite loop. The program creates copies of the calling process until there is no more free memory in the operating system, and the computer becomes unusable. The source code in Listing 8.4 implements a fork bomb in the C programming language.

In einem solchen Fall wurde der Elternprozess vor dem Kindprozess beendet und wie üblich wurde dem Kindprozess der Prozess `init` als neuer Elternprozess zugeordnet. Der `init`-Prozess ist der erste Prozess unter Linux und verwandten Betriebssystemen. Er *adoptiert* elternlose Prozesse automatisch.

Seit Linux Kernel 3.4 (2012) und Dragonfly BSD 4.2 (2015) können zumindest bei diesen Betriebssystemen auch andere Prozesse als `init` neue Elternprozesse eines verwaisten Kindprozesses werden.

Eine potentielle Gefahr des Systemaufrufs `fork` sind *Forkbomben*. Eine Forkbombe ist ein Schadprogramm, das `fork` oder eine entsprechende Bibliotheksfunktion in einer Endlosschleife aufruft. Das Programm wird so lange Kopien des aufrufenden Prozesses erzeugen, bis kein Speicher im Betriebssystem mehr frei ist und der Computer unbenutzbar ist. Der Quellcode in Listing 8.4 realisiert eine Forkbombe in der Programmiersprache C.

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5   while(1)
6     fork();
7 }
```

Listing 8.4: Fork Bomb Example

Fork bombs are a potential threat, especially in situations where a large number of users have simultaneous access to a computer system. The most straightforward way to handle this threat is to limit the maximum number of processes per user and the maximum memory usage per user.

Forkbomben sind besonders dort eine potentielle Gefahr, wo eine große Zahl von Benutzern gleichzeitig auf einen Computer zugreift. Die einfachste Möglichkeit, um mit dieser Gefahr umzugehen ist die maximale Anzahl an Prozessen pro Benutzer und den maximalen Speicherverbrauch pro Benutzer einzuschränken.

8.5

Replacing Processes via exec

With the system call `fork`, it is possible to create a child process that is an identical copy of the calling process. However, if a whole new process and not a copy is to be created, then the system call `exec` must be used to replace one process by another. In this case, the new process gets the PID of the calling process (see Figure 8.14).

Prozesse ersetzen mit exec

Mit dem Systemaufruf `fork` ist es möglich, einen Kindprozess als identische Kopie des aufrufenden Prozesses zu erzeugen. Soll aber ein ganz neuer Prozess und keine Kopie erstellt werden, muss der Systemaufruf `exec` verwendet werden, um einen Prozess durch einen anderen zu ersetzen. In diesem Fall erbt der neue Prozess die PID des aufrufenden Prozesses (siehe Abbildung 8.14).

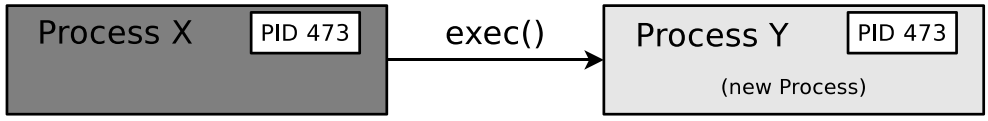

Figure 8.14: Replacing one Process with another is done by calling `exec`

If a program shall be started from a process, such as a command-line interpreter (*shell*), a new process must be created first with `fork` and then replaced with `exec`. If no new process is created with `fork` before `exec` is called, the parent process gets lost. Figure 8.15 shows the different ways of process creation in Linux [39]:

- *Process forking*: A running process creates a new, identical process with `fork`.
- *Process chaining*: A running process creates a new process with `exec()` and terminates itself this way because it gets replaced by the new process.
- *Process creation*: A running process creates a new, identical process with `fork`, which replaces itself via `exec()` by a new process.

The following example shows the effects of calling `exec` in the command line interpreter.

Soll aus einem Prozess wie beispielsweise aus einem Kommandozeileninterpreter (*Shell*) heraus ein Programm gestartet werden, muss zuerst mit `fork` ein neuer Prozess erzeugt und dieser anschließend mit `exec` ersetzt werden. Wird vor einem Aufruf von `exec` kein neuer Prozess mit `fork` erzeugt, geht der Elternprozess verloren. Abbildung 8.15 fasst die existierenden Möglichkeiten der Prozesserzeugung unter Linux zusammen [39]:

- Die *Prozessvergabelung* (englisch: *Process forking*): Ein laufender Prozess erzeugt mit `fork` einen neuen, identischen Prozess.
- Die *Prozessverkettung* (englisch: *Process chaining*): Ein laufender Prozess erzeugt mit `exec` einen neuen Prozess und beendet (terminiert) sich damit selbst, weil der neue Prozess ihn ersetzt.
- Die *Prozesserzeugung* (englisch: *Process creation*): Ein laufender Prozess erzeugt mit `fork` einen neuen, identischen Prozess, der sich selbst mit `exec` durch einen neuen Prozess ersetzt.

Das folgende Beispiel zeigt die Auswirkungen eines Aufrufs von `exec` im Kommandozeilen-

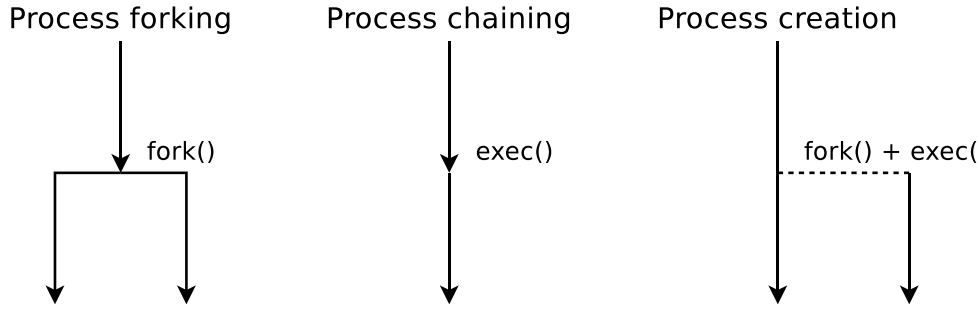

Figure 8.15: The System Calls `fork` and `exec` are used in Linux for Process creation

The first call of the command `ps -f` returns two processes as an output. First, the UNIX shell bash with process number 26756 and the executed command itself, whose PPID is 26756, and it matches the PID of the bash, as expected. In the next step, another bash is started in the already running bash. The output of the command `ps -f` now shows the initial bash, the `ps` command, and the new bash instance. The PPID 26756 of the new bash matches the PID of the initial bash, and the PPID 1278 of the `ps` command matches the PID of the new bash instance as expected.

The subsequent call of the command `exec` with the `ps -f` command as argument creates a new bash that is replaced by the `ps -f` command. This command got the PID 1278 of the replaced bash and its parent relationship (PPID 26756). It terminates as soon as the parent process of the `ps` command requests its return value. After that, the operating system no longer has a process with process number 1278.

interpreter. Der erste Aufruf des Kommandos `ps -f` zeigt zwei Prozesse. Einmal die UNIX-Shell Bash mit der Prozessnummer 26756 und das aufgerufene Kommando selbst, dessen PPID 26756 ist, und wie erwartet mit der PID der Bash übereinstimmt. Im nächsten Schritt wird eine weitere Bash innerhalb der bereits laufenden gestartet. Die neue Ausgabe des Kommandos `ps -f` zeigt nun die ursprüngliche Bash, das `ps`-Kommando und die neue Instanz der Bash an. Die PPID 26756 der neuen Bash stimmt mit der PID der ursprünglichen Bash überein und die PPID 1278 des `ps`-Kommandos stimmt wie erwartet mit der PID der neuen Bash-Instanz überein.

Der anschließende Aufruf des Kommandos `exec` mit dem Kommando `ps -f` als Argument führt dazu, dass die neue Bash durch das Kommando `ps -f` ersetzt wird. Dieses Kommando hat nun auch die PID 1278 der ersetzten Bash und deren Elternbeziehung (PPID 26756) geerbt. Sobald der Elternprozess des `ps`-Kommandos dessen Rückgabewert abgefragt hat, wird es terminieren. Danach existiert einstweilen im Betriebssystem kein Prozess mehr, der die Prozessnummer 1278 hat.

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
bnc       1265 26756  0 13:25 pts/1    00:00:00 ps -f
bnc      26756  1694  0 10:17 pts/1    00:00:00 bash
$ bash
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
bnc       1278 26756  1 13:25 pts/1    00:00:00 bash
bnc       1293  1278  0 13:25 pts/1    00:00:00 ps -f
bnc      26756  1694  0 10:17 pts/1    00:00:00 bash
$ exec ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
bnc       1278 26756  0 13:25 pts/1    00:00:00 ps -f
bnc      26756  1694  0 10:17 pts/1    00:00:00 bash
```

Another example that demonstrates how to work with the system call `exec` is the program in Listing 8.5. The program structure is very similar to the previous examples of `fork`. In the example, the running process creates a child process with the library function for the system call `fork`. The return value is stored in the integer variable `pid`. As in the previous examples, the return value of `fork` is used in the source code to determine whether it is the parent process or the child process. The parent process returns its process ID and the child's process ID as output. It gets its process ID by using the library function for the system call `getpid`. The process ID of the child is obtained by the value of the integer variable `pid`. Before the parent process terminates, it requests its process number again.

The child process outputs its process ID, and that of the parent process, which it can request from the operating system using the library functions for the system calls `getpid` and `getppid`. Finally, the child process replaces itself by calling `execl` with the `date` command as an argument. The example uses the library function `execl` because the system call `exec` does not exist as a library function, but there are several variants, one of which is `execl` [50, 97].

Ein weiteres Beispiel zur Arbeitsweise mit dem Systemaufruf `exec` ist das Programm in Listing 8.5. Der Aufbau hat eine große Ähnlichkeit mit den vorangegangenen Beispielen zu `fork`. Im Programmbeispiel erzeugt der laufende Prozess mit der Bibliotheksfunktion für den Systemaufruf `fork` einen Kindprozess. Den Rückgabewert speichert das Programm in der Integer-Variable `pid`. Wie bei den vorherigen Beispielen wird anhand des Rückgabewerts von `fork` im Quellcode erkannt, ob es sich jeweils um den Elternprozess oder den Kindprozesses handelt. Der Elternprozess gibt seine eigene Prozessnummer und die Prozessnummer des Kindes aus. Die eigene Prozessnummer erfährt er mit Hilfe der Bibliotheksfunktion für den Systemaufruf `getpid`. Die Prozessnummer des Kindes erfährt er durch den Wert der Integer-Variable `pid`. Bevor sich der Elternprozess beendet, erfragt er noch einmal seine eigene Prozessnummer.

Der Kindprozess gibt seine eigene Prozessnummer und die des Elternprozesses aus, die er mit Hilfe der Bibliotheksfunktionen für die Systemaufrufe `getpid` und `getppid` vom Betriebssystem erfragt. Abschließend ersetzt sich der Kindprozess selbst durch einen Aufruf von `execl` mit dem Kommando `date` als Argument. Im Beispiel wurde die Bibliotheksfunktion `execl` verwendet, weil der Systemruf `exec` nicht als Bibliotheksfunktion existiert, aber dafür mehrere Varianten, von der eine `execl` ist [50, 97].

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
```

```
5 int main () {
6   int pid;
7   // Create a child process
8   pid = fork();
9
10  // An error occured --> program abort
11  if (pid < 0) {
12    printf("fork() caused an error!\n");
13    exit(1);
14  }
15
16  // Parent process
17  if (pid > 0) {
18    printf("[Parent] Own PID:           %d\n", getpid());
19    printf("[Parent] PID of the child:  %d\n", pid);
20  }
21
22  // Child process
23  if (pid == 0) {
24    printf("[Child]  Own PID:           %d\n", getpid());
25    printf("[Child]  PID of the parent: %d\n", getppid());
26
27    // Current program is replaced by "date"
28    // "date" will be the process name in the process table
29    execl("/bin/date", "date", "-u", NULL);
30  }
31
32  printf("[Parent] Program abort.\n");
33  return 0;
34 }
```

Listing 8.5: The Library Function `execl` calls the System Call `exec` and in this way replaces a Process by a new one

After compiling the program with the GNU C compiler (`gcc`) and running it, the output has, in most cases, the following form:

Nach dem Übersetzen des Programms mit dem GNU C Compiler (`gcc`) und dem anschließenden Ausführen, hat die Ausgabe in den meisten Fällen die folgende Form:

```
$ gcc Listing_8_5_exec.c -o Listing_8_5_exec
$ ./Listing_8_5_exec
[Parent] Own PID:           45413
[Parent] PID of the child:  45414
[Parent] Program abort.
[Child]  Own PID:           45414
[Child]  PID of the parent: 45413
Do 19. Jan 08:32:34 UTC 2023
```

In rare cases, the parent process terminates before the child process, and the process `init` with process ID 1 is assigned to the child process as the new parent process.

In seltenen Fällen wird auch bei diesem Programmbeispiel der Elternprozess vor dem Kind-Prozess beendet und dem Kindprozess der Prozess `init` mit der Prozessnummer 1 als neuer Elternprozess zugeordnet.

```
$ gcc Listing_8_5_exec.c -o Listing_8_5_exec
$ ./Listing_8_5_exec
[Parent] Own PID:           46699
[Parent] PID of the child:  46700
[Parent] Program abort.
[Child]  Own PID:           46700
[Child]  PID of the parent: 1
Do 19. Jan 08:33:58 UTC 2023
```

8.6

Process Switching and Process Scheduling

Tasks of multitasking operating systems are, among others, *dispatching* and *scheduling*. Dispatching is the switching of the CPU during process switching. Scheduling is the determination of the point in time when process switching occurs, and the calculation of the execution order of the processes.

During process switching, the dispatcher removes the CPU from the running process and assigns it to the process, which is the first one in the queue. For transitions between the states `ready` and `blocked`, the dispatcher removes the corresponding process control blocks from the status lists and accordingly inserts them newly. Transitions from or to the state `running` always switch the process that the CPU currently executes. If a process switches into the state running or from the state running to another state, the dispatcher needs to back up the process context (register contents) of the executed process in the process control block. Next, the dispatcher assigns the CPU to another process and imports the context (register contents) of the process, which is executed next, from its process control block.

The scheduler in the kernel specifies the execution order of the processes in the state `ready`. However, no scheduling strategy is optimally suited for each system, and no scheduling strategy can take into account all scheduling criteria in an optimal way, such as CPU load, response time (*latency*), turnaround time, throughput, efficiency, real-time behavior (com-

Prozesswechsel und Scheduling von Prozessen

Zu den Aufgaben von Betriebssystemen mit Mehrprogrammbetrieb gehört das *Dispatching* und das *Scheduling*. Dispatching ist das Umschalten des Hauptprozessors beim Prozesswechsel. Scheduling ist das Festlegen des Zeitpunkts des Prozesswechsels und der Ausführungsreihenfolge der Prozesse.

Beim Prozesswechsel entzieht der Dispatcher im Betriebssystemkern dem rechnenden Prozess den Prozessor und teilt ihn dem Prozess zu, der in der Warteschlange an erster Stelle steht. Bei Übergängen zwischen den Zuständen `bereit` und `blockiert` werden vom Dispatcher die entsprechenden Prozesskontrollblöcke aus den Zustandslisten entfernt und neu eingefügt. Übergänge aus oder in den Zustand `rechnend` bedeuten immer einen Wechsel des gegenwärtig rechnenden Prozesses auf dem Prozessor. Beim Prozesswechsel in oder aus dem Zustand `rechnend` muss der Dispatcher den Prozesskontext, also die Registerinhalte des gegenwärtig ausgeführten Prozesses im Prozesskontrollblock speichern (retten), danach den Prozessor einem anderen Prozess zuteilen und den Prozesskontext (Registerinhalte) des jetzt auszuführenden Prozesses aus seinem Prozesskontrollblock wieder herstellen.

Der Scheduler im Betriebssystemkern legt die Ausführungsreihenfolge der Prozesse im Zustand `bereit` fest. Allerdings ist kein Verfahren für jedes System optimal geeignet und kein Schedulingverfahren kann alle Scheduling-Kriterien wie Prozessor-Auslastung, Antwortzeit (*Latenz*), Durchlaufzeit (*Turnaround Time*), Durchsatz, Effizienz, Echtzeitverhalten

pliance with deadlines), waiting time, overhead, fairness, consideration of priorities, and even resource utilization. When choosing a scheduling strategy, a compromise between the scheduling criteria must always be found. There are two classes of scheduling strategies:

- *Non-preemptive scheduling* or *cooperative scheduling.* A process, which is assigned to the CPU by the scheduler, remains in control of the CPU until its execution is finished, or until it gives control back voluntarily. The most problematic aspect of such scheduling strategies is that a process may occupy the CPU as long as it wants to. Examples of operating systems that use this scheduling method are Windows 3.x and Mac OS 8/9.

- *Preemptive scheduling.* The CPU may be removed from a process before its execution is complete. If the CPU is removed from a process, this process is paused until the scheduler assigns the CPU to it again. A drawback of preemptive scheduling is the higher overhead. However, the advantages of preemptive scheduling outweigh the drawbacks because operating systems can react to events with it and consider processes priorities. Examples of operating systems that use this scheduling method are Linux, Mac OS X, and Windows from version 95 on.

The following example shows how significant the impact of the scheduling method used can be on the overall performance of a computer system. Two processes P_A and P_B shall be executed one after the other and without interruption. Process P_A needs 27 ms CPU time, and process P_B needs 3 ms CPU time. The operating system scheduler must determine the execution order of the two processes. The Tables 8.1 and 8.2 show the possible scenarios and their consequences.

(Termineinhaltung), Wartezeit, Verwaltungsaufwand (*Overhead*), Fairness, Berücksichtigen von Prioritäten und gleichmäßiger Ressourcenauslastung optimal berücksichtigen. Bei der Auswahl eines Schedulingverfahrens muss immer ein Kompromiss zwischen den Scheduling-Kriterien gefunden werden. Die existierenden Schedulingverfahren werden in zwei Klassen unterschieden:

- *Nicht-präemptives (nicht-verdrängendes) Scheduling* bzw. *kooperatives Scheduling.* Dabei behält ein Prozess, der vom Scheduler den Hauptprozessor zugewiesen bekommen hat, die Kontrolle über diesen bis zu seiner vollständigen Fertigstellung oder bis er die Kontrolle freiwillig wieder abgibt. Problematisch bei solchen Schedulingverfahren ist, dass ein Prozess den Prozessor so lange belegen kann wie er will. Beispiele für Betriebssysteme, die diese Form des Schedulings verwenden sind Windows 3.x und Mac OS 8/9.

- *Präemptives (verdrängendes) Scheduling.* Dabei kann einem Prozess der Prozessor vor seiner Fertigstellung entzogen werden. In einem solchen Fall pausiert der Prozess so lange, bis der Scheduler ihm erneut den Prozessor zuteilt. Ein Nachteil des verdrängenden Schedulings ist der höhere Verwaltungsaufwand. Allerdings überwiegen die Vorteile des präemptivem Schedulings, weil Betriebssysteme damit auf Ereignisse reagieren und Prozesse mit einer höheren Priorität berücksichtigen können. Beispiele für Betriebssysteme, die diese Form des Schedulings verwenden sind Linux, Mac OS X, Windows 95 und neuere Versionen.

Wie groß der Einfluss des verwendeten Schedulingverfahrens auf die Gesamtleistung eines Computers sein kann, zeigt das folgende Beispiel. Zwei Prozesse P_A und P_B sollen nacheinander und ohne Unterbrechung ausgeführt werden. Prozess P_A benötigt den Hauptprozessor 27 ms und Prozess P_B benötigt ihn nur 3 ms. Der Scheduler des Betriebssystems muss die Reihenfolge festlegen, in der die beiden Prozesse ausgeführt werden. Die Tabellen 8.1 und 8.2 zei-

gen die beiden möglichen Szenarien und deren Auswirkungen.

Table 8.1: Impact of the Execution Order on average Runtime

Execution order	Runtime P_A	Runtime P_B	Average runtime
P_A,P_B	27 ms	30 ms	$\frac{27+30}{2} = 28.5$ ms
P_B,P_A	30 ms	3 ms	$\frac{30+3}{2} = 16.5$ ms

Table 8.2: Impact of the Execution Order on average Waiting Time

Execution order	Waiting time P_A	Waiting time P_B	Average waiting time
P_A,P_B	0 ms	27 ms	$\frac{0+27}{2} = 13.5$ ms
P_B,P_A	3 ms	0 ms	$\frac{3+0}{2} = 1.5$ ms

The results in the Tables 8.1 and 8.2 show that the runtime and waiting time of the process with the high resource demand only worsen slightly when a process with a short runtime is executed before. If a process with a long runtime runs before a process with a short runtime, the runtime and waiting time of the process with little resource requirements gets significantly worse.

The *waiting time* of a process is the time the process had to wait in the `ready` list for getting assigned to the CPU.

The following sections and Table 8.3 describe some classic and modern scheduling methods and their characteristics.

Die Ergebnisse in den Tabellen 8.1 und 8.2 zeigen, dass sich Laufzeit und Wartezeit des Prozesses mit hohem Ressourcenaufwand nur wenig verschlechtern, wenn ein Prozess mit kurzer Laufzeit zuvor ausgeführt wird. Läuft ein Prozess mit einer langen Laufzeit aber vor einem Prozess mit kurzer Laufzeit, verschlechtern sich die Laufzeit und die Wartezeit des Prozesses mit geringem Ressourcenbedarf deutlich.

Die *Wartezeit* eines Prozesses ist die Zeit, die der Prozess in der `bereit`-Liste auf die Zuteilung des Prozessors gewartet hat.

Die folgenden Abschnitte und Tabelle 8.3 stellen einige klassische und moderne Schedulingverfahren und deren Eigenschaften vor.

Table 8.3: An Overview of classic and modern Scheduling Methods

	Scheduling NP[a]	Scheduling P[b]	Fair[c]	CPU runtime must be known	Takes priorities into account
Priority-driven scheduling	X	X	no	no	yes
First Come First Served	X		yes	no	no
Round Robin		X	yes	no	no
Shortest Job First	X		no	yes	no
Shortest Remaining Time First		X	no	yes	no
Longest Job First	X		no	yes	no
Longest Remaining Time First		X	no	yes	no
Highest Response Ratio Next	X		yes	yes	no
Earliest Deadline First	X	X	yes	no	no
Fair-Share		X	yes	no	no
Static multilevel scheduling		X	no	no	yes
Multilevel feedback scheduling		X	yes	no	yes
O(1) scheduler		X	yes	no	yes
Completely Fair Scheduler		X	yes	no	yes
Earliest Eligible Virtual Deadline First		X	yes	no	yes

[a] NP = non-preemptive scheduling.
[b] P = preemptive scheduling.
[c] A scheduling method is fair when no process can starve because it waits endlessly for the assignment of the CPU.

8.6.1

Priority-driven Scheduling

In priority-controlled scheduling, the process with the highest priority in the `ready` state always gets assigned to the CPU next. The processes are thus processed according to their importance or urgency. Priority may depend on various criteria, such as required resources, user rank, demanded real-time criteria, etc.

Priority-controlled scheduling can be implemented as preemptive scheduling or non-preemptive, and the priority values can be assigned statically or dynamically. Static priorities remain unchanged throughout the lifetime of a process and are often used in real-time systems (see Section 3.6). Dynamic priorities are adjusted from time to time. In this case, the scheduling method used is called *multilevel feedback scheduling* (see Section 8.6.11).

One risk of static priority-driven scheduling is that low priority processes may starve [17]. They might wait infinitely long for the assignment of the CPU because processes with a higher priority continuously appear in the `ready` list. Static priority-driven scheduling is, therefore, not fair.

Figure 8.16 shows a possible implementation of priority-driven scheduling. Thereby, a queue exists for each priority value.

Prioritätengesteuertes Scheduling

Beim prioritätengesteuerten Scheduling bekommt immer der Prozess mit der höchsten Priorität im Zustand `bereit` als nächstes den Prozessor zugewiesen. Die Prozesse werden somit nach ihrer Wichtigkeit bzw. Dringlichkeit abgearbeitet. Die Priorität kann von verschiedenen Kriterien abhängen, zum Beispiel benötigte Ressourcen, Rang des Benutzers, geforderte Echtzeitkriterien usw.

Prioritätengesteuertes Scheduling kann verdrängend (*präemptiv*) oder nicht-verdrängend (*nicht-präemptiv*) realisiert sein und die Vergabe der Prioritäten kann statisch oder dynamisch erfolgen. Statische Prioritäten ändern sich während der gesamten Lebensdauer eines Prozesses nicht und werden häufig in Echtzeitsystemen (siehe Abschnitt 3.6) verwendet. Dynamische Prioritäten werden von Zeit zu Zeit angepasst. In diesem Fall heißt das verwendete Schedulingverfahren *Multilevel-Feedback Scheduling* (siehe Abschnitt 8.6.11).

Eine Gefahr beim statischen prioritätengesteuertem Scheduling ist, dass Prozesse mit niedriger Priorität verhungern können [17]. Sie warten also endlos lange auf die Zuteilung des Prozessors, weil immer wieder Prozesse mit einer höheren Priorität in der `bereit`-Liste ankommen. Das statische prioritätengesteuerte Scheduling ist somit nicht fair.

Abbildung 8.16 zeigt eine mögliche Realisierung für das prioritätengesteuerte Scheduling. Hierbei existiert für jede mögliche Priorität eine Warteschlange.

8.6.2

First Come First Served

The scheduling method *First Come First Served* (FCFS) works according to the principle *First In First Out* (FIFO). The processes get assigned to the CPU according to their order of arrival. Running processes are not interrupted by FCFS. It is, therefore, a non-preemptive scheduling method.

Since all processes are processed according to their arrival order, the method is fair. However, the average waiting time may be high under certain circumstances, since processes with short

First Come First Served

Das Schedulingverfahren *First Come First Served* (FCFS) funktioniert nach dem Prinzip *First In First Out* (FIFO). Die Prozesse bekommen den Prozessor entsprechend ihrer Ankunftsreihenfolge zugewiesen. Laufende Prozesse werden bei FCFS nicht unterbrochen. Es handelt sich somit um nicht-verdrängendes Scheduling.

Da alle Prozesse entsprechend ihrer Ankunftsreihenfolge berücksichtigt werden, ist das Verfahren fair. Allerdings kann die mittlere Wartezeit unter Umständen hoch sein, da Prozesse

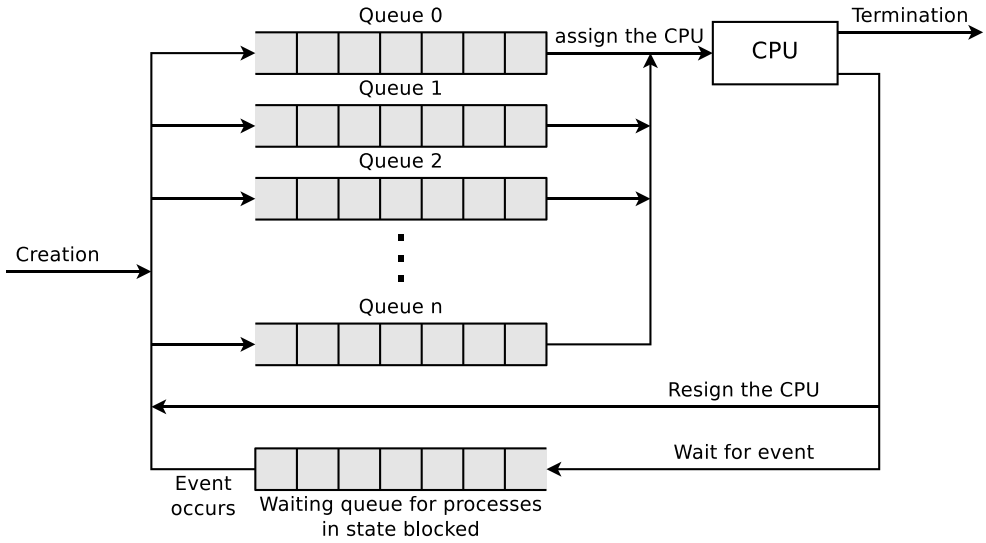

Figure 8.16: Priority-driven Scheduling can be implemented with several Queues for different Priority Values [108]

execution times may need to wait for a long time if processes with long execution times have arrived before. For example, FCFS can be used for batch processing (see Section 3.4.1) [112].

mit kurzer Abarbeitungszeit eventuell eine lange Zeit warten müssen, wenn vor ihnen Prozesse mit langer Abarbeitungszeit eingetroffen sind. FCFS eignet sich unter anderem für Stapelverarbeitung (siehe Abschnitt 3.4.1) [115].

8.6.3

Round Robin

Round Robin (RR) specifies time-slices with a fixed duration. The processes are queued in a cyclic queue and processed according to the FIFO principle (see Figure 8.17). The first process of the queue is assigned to the CPU for the duration of a time slice. After this time slice has elapsed, the process is removed from the CPU, and it is positioned at the end of the queue. Round Robin is, therefore, a preemptive scheduling method. If a process has been completed, it is removed from the queue. New processes are inserted at the end of the queue.

Round Robin

Bei *Round Robin* (RR), das in der Literatur auch *Zeitscheibenverfahren* [17, 119] heißt, werden Zeitscheiben (englisch: *Time Slices*) mit einer festen Dauer definiert. Die Prozesse werden in einer zyklischen Warteschlange eingereiht und nach dem Prinzip FIFO abgearbeitet (siehe Abbildung 8.17). Der erste Prozess der Warteschlange erhält für die Dauer einer Zeitscheibe Zugriff auf den Prozessor. Nach dem Ablauf der Zeitscheibe wird dem Prozess der Zugriff auf den Prozessor wieder entzogen und er wird am Ende der Warteschlange eingereiht. Bei Round Robin handelt sich somit um verdrängendes Scheduling. Wird ein Prozess erfolgreich beendet, wird er aus der Warteschlange entfernt. Neue Prozesse werden am Ende der Warteschlange eingereiht.

CPU time is distributed in a fair way among the processes. Round Robin with a time slice size of ∞ behaves like FCFS.

Die Zugriffszeit auf den Prozessor verteilt dieses Verfahren fair auf die Prozesse. Round Robin mit der Zeitscheibengröße ∞ verhält sich wie FCFS.

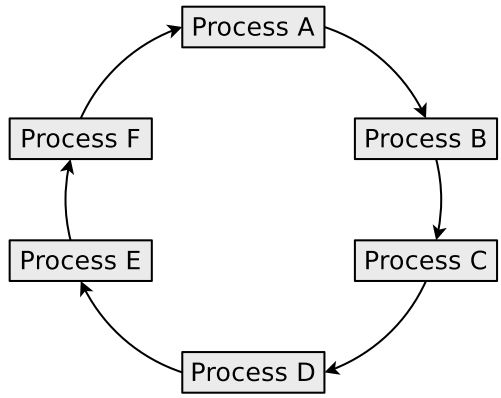

Figure 8.17: The Round Robin Scheduling Method arranges the Processes in a cyclic Queue

Round Robin prefers processes with short execution times, because the longer the execution time of a process is, the more rounds are required for its execution to complete. The size of the time slices influences the performance of the system. The length is usually in a single-digit or double-digit millisecond range [119]. The shorter the time slices, the more process switching must take place, and each process switching causes overhead. The longer the time slices are, the more the perceived simultaneousness is lost. The system hangs. For an interactive system, users might say it is "jerky". This scheduling method can be used for interactive systems [112].

Round Robin bevorzugt Prozesse mit kurzer Abarbeitungszeit, denn je länger die Bearbeitungsdauer eines Prozesses ist, desto mehr Runden sind für seine vollständige Ausführung nötig. Die Größe der Zeitscheiben ist wichtig für die Systemgeschwindigkeit. Sie liegt üblicherweise im ein- oder zweistelligen Millisekundenbereich [119]. Je kürzer die Zeitscheiben sind, desto mehr Prozesswechsel müssen stattfinden und jeder Prozesswechsel verursacht einen Verwaltungsaufwand (*Overhead*). Je länger die Zeitscheiben sind, desto mehr geht die „gefühlte Parallelität" verloren. Das System hängt. Bei einem interaktiven System würden man sagen: „es ruckelt". Dieses Schedulingverfahren eignet sich unter anderem für interaktive Systeme [115].

8.6.4

Shortest Job First / Shortest Process Next

With *Shortest Job First* (SJF), which is also called *Shortest Process Next* (SPN), the process with the shortest execution time is assigned to the CPU first. It is a non-preemptive scheduling method. SJF is not fair because processes with a short execution time are preferred, and processes with a long execution time may only

Shortest Job First / Shortest Process Next

Bei *Shortest Job First* (SJF), das auch *Shortest Process Next* (SPN) heißt, erhält der Prozess mit der kürzesten Abarbeitungszeit als erster Zugriff auf den Prozessor. Es handelt sich um nicht-verdrängendes Scheduling. SJF ist nicht fair, denn Prozesse mit kurzer Abarbeitungszeit werden bevorzugt und Prozesse mit langer Abarbeitungszeit erhalten eventuell erst nach einer

get assigned to the CPU after a long waiting period, or they starve.

The main problem of SJF is that for each process, it is necessary to know how long it takes until its termination, i.e., how long its execution time is. In practice, it is seldom the case that this information exists [119]. In scenarios where the execution time of processes can be estimated, possibly by recording and analyzing previous processes, SJF can be used for batch processing (see Section 3.4.1), or for interactive processes that have a short execution time in general [17, 39, 112].

8.6.5

Shortest Remaining Time First

Preemptive SJF is called *Shortest Remaining Time First* (SRTF). If a new process is created, the remaining execution time of the currently running process is compared to each process in the list of waiting processes. If the currently running process has the shortest remaining execution time, it continues to run. If one or more processes in the list of waiting processes have a shorter execution time or remaining execution time, the process with the shortest remaining execution time is assigned to the CPU. The processes in the list of waiting processes are only compared to the currently running process when a new process is created. As long as no new process is created, no running process is interrupted.

This scheduling method has the same drawbacks as the preemptive variant SJF. The (remaining) execution time of the processes must be determined or estimated, which in practice is usually unrealistic. This scheduling method is not fair, since processes with a long execution time may starve.

8.6.6

Longest Job First

With *Longest Job First* (LJF), the process with the longest execution time is assigned to the CPU first. It is a non-preemptive scheduling method. LJF is not fair, because processes with a long execution time are preferred, and processes with a short execution time may only

langen Wartezeit Zugriff auf den Prozessor oder sie verhungern.

Das größte Problem von SJF ist, dass für jeden Prozess bekannt sein muss, wie lange er bis zu seiner Terminierung braucht, also wie lange seine Abarbeitungszeit ist. Das ist in der Praxis praktisch nie der Fall [119]. In Szenarien, wo die Abarbeitungszeit der Prozesse, eventuell durch die Erfassung und Analyse vorheriger Prozesse abgeschätzt werden kann, eignet sich SJF für Stapelverarbeitung (siehe Abschnitt 3.4.1) oder für interaktive Prozesse, die generell eine kurze Abarbeitungszeit haben [17, 39, 115].

Shortest Remaining Time First

Verdrängendes SJF heißt *Shortest Remaining Time First* (SRTF). Wird ein neuer Prozess erstellt, wird die Restlaufzeit des aktuell rechnenden Prozesses mit jedem Prozess in der Liste der wartenden Prozesse verglichen. Hat der gegenwärtig rechnende Prozesses die kürzeste Restlaufzeit, darf er weiterrechnen. Haben ein oder mehr Prozesse in der Liste der wartenden Prozesse eine kürzere Abarbeitungszeit bzw. Restlaufzeit, erhält der Prozess mit der kürzesten Restlaufzeit Zugriff auf den Prozessor. Die Prozesse in der Liste der wartenden Prozesse werden nur dann mit dem aktuell rechnenden Prozess verglichen, wenn ein neuer Prozess erstellt wird. Solange kein neuer Prozess eintrifft, wird auch kein rechnender Prozess unterbrochen.

Die Nachteile von SJF gelten auch für die verdrängenden Variante. Die (Rest-)laufzeit der Prozesse muss bekannt sein oder abgeschätzt werden, was in der Praxis in den allermeisten Fällen unrealistisch ist. Das Verfahren ist nicht fair, da Prozesse mit langer Abarbeitungszeit verhungern können.

Longest Job First

Bei *Longest Job First* (LJF) erhält der Prozess mit der längsten Abarbeitungszeit als erster Zugriff auf den Prozessor. Es handelt sich um nicht-verdrängendes Scheduling. LJF ist nicht fair, da Prozesse mit langer Abarbeitungszeit bevorzugt werden und Prozesse mit kurzer Ab-

by assigned to the CPU after a very long waiting time, or starve.

Just as with SJF and SRTF, also with LJF, it must be determined for each process how long its execution time is. If the execution time of the processes can be estimated, LJF, in the same way as SJF, can be used for batch processing (see Section 3.4.1).

8.6.7 Longest Remaining Time First

Preemptive LJF is called *Longest Remaining Time First* (LRTF). If a new process is created, as with SRTF, the remaining execution time of the currently running process is compared to each process in the list of waiting processes. If the currently running process has the longest remaining execution time, it continues to run. If one or more processes in the list of waiting processes have a longer execution time or remaining execution time, the process with the longest remaining execution time is assigned to the CPU. The processes in the list of waiting processes are only compared with the currently running process when a new process is created. As long as no new process is created, no running process is interrupted.

This scheduling method has the same drawbacks as the preemptive variant LJF. The (remaining) execution time of the processes must be determined or estimated, which in practice is usually unrealistic. This scheduling method is not fair, since processes with a short execution time may starve.

8.6.8 Highest Response Ratio Next

The scheduling method *Highest Response Ratio Next* (HRRN) is a fair variant of SJF/SRTF/LJF/LRTF because it takes the age of each process into account to avoid starvation. The deciding criterion for the execution order of the processes is a *response ratio*, which the scheduler calculates for each process using this equation:

arbeitungszeit möglicherweise erst nach sehr langer Wartezeit Zugriff auf den Prozessor erhalten oder verhungern.

Genau wie bei SJF und SRTF muss auch bei LJF für jeden Prozess bekannt sein, wie lange er den Prozessor bis zu seiner Abarbeitung benötigt. Wenn die Abarbeitungszeit der Prozesse abgeschätzt werden kann, eignet sich LJF genau wie SJF für Stapelverarbeitung (siehe Abschnitt 3.4.1).

8.6.7 Longest Remaining Time First

Verdrängendes LJF heißt *Longest Remaining Time First* (LRTF). Wird ein neuer Prozess erstellt, wird genau wie bei SRTF die Restlaufzeit des gegenwärtig rechnenden Prozesses mit jedem Prozess in der Liste der wartenden Prozesse verglichen. Hat der rechnende Prozess die längste Restlaufzeit, darf er weiterrechnen. Haben ein oder mehr Prozesse in der Liste der wartenden Prozesse eine längere Abarbeitungszeit bzw. Restlaufzeit, erhält der Prozess mit der längsten Restlaufzeit Zugriff auf den Prozessor. Die Prozesse in der Liste der wartenden Prozesse werden nur dann mit dem rechnenden Prozess verglichen, wenn ein neuer Prozess erstellt wird. Solange kein neuer Prozess eintrifft, wird auch kein rechnender Prozess unterbrochen.

Die Nachteile von LJF gelten auch für die verdrängende Variante. Die (Rest-)laufzeit der Prozesse muss bekannt sein oder abgeschätzt werden, was in der Praxis in den allermeisten Fällen unrealistisch ist. Das Verfahren ist nicht fair, da Prozesse mit kurzer Abarbeitungszeit verhungern können.

8.6.8 Highest Response Ratio Next

Das Schedulingverfahren *Highest Response Ratio Next* (HRRN) ist eine faire Variante von SJF/SRTF/LJF/LRTF, denn es berücksichtigt das Alter der Prozesse, um ein Verhungern zu vermeiden. Das für die Ausführungsreihenfolge der Prozesse entscheidende Kriterium ist ein *Antwortquotient* (englisch: *Response Ratio*), den der Scheduler für jeden Prozess mit Hilfe der folgenden Formel berechnet:

$$\text{response ratio} = \frac{\text{estimated execution time [s]} + \text{waiting time [s]}}{\text{estimated execution time [s]}}$$

The response ratio has the value 1 when a process is created, and it rises quickly for short processes. This scheduling method operates efficiently if the response ratio of all processes in the system is low. After the termination or blocking of a process, the process with the highest response ratio is assigned to the CPU. This method prevents any process from starving. Thus, HRRN is a fair scheduling method.

Since with HRRN, just like with SJF/SRTF/LJF/LRTF, statistical recordings of the past must be used to estimate the execution times of the processes, this scheduling method is hardly usable in practice [17].

Der Antwortquotienten hat bei der Erzeugung eines Prozesses den Wert 1 und steigt bei kurzen Prozessen schnell an. Das Schedulingverfahren arbeitet dann effizient, wenn der Antwortquotient aller Prozesse im System niedrig ist. Nach der Beendigung oder bei der Blockade eines Prozesses bekommt der Prozess mit dem höchsten Antwortquotient den Prozessor zugewiesen. Damit ist sichergestellt, dass kein Prozess verhungert. Das macht HRRN zu einem fairen Verfahren.

Da bei HRRN genau wie bei SJF/SRTF/LJF/LRTF die Laufzeiten der Prozesse durch statistische Erfassungen der Vergangenheit abgeschätzt werden müssen, ist das Verfahren in der Praxis meist nicht einsetzbar [17].

8.6.9

Earliest Deadline First

Earliest Deadline First

The aim of the scheduling method *Earliest Deadline First* (EDF) is to comply with the deadlines of the processes. The processes in the `ready` state are arranged according to their deadlines. The process with the closest deadline is assigned to the CPU next. The queue is reviewed and reorganized whenever a new process switches to the `ready` state, or when an active process terminates.

EDF can be implemented as a preemptive or non-preemptive scheduling method. Preemptive EDF can be used in real-time operating systems (see Section 3.6), non-preemptive EDF can be used for batch processing (see Section 3.4.1), for example.

Der Fokus beim Schedulingverfahren *Earliest Deadline First* (EDF) ist, dass die Termine zur Fertigstellung (*Deadlines*) der Prozesse eingehalten werden. Die Prozesse im Zustand `bereit` werden anhand ihrer jeweiligen Deadline geordnet. Der Prozess, dessen Deadline am nächsten ist, bekommt als nächstes den Prozessor zugewiesen. Eine Überprüfung und gegebenenfalls Neuorganisation der Warteschlange findet immer dann statt, wenn ein neuer Prozess in den Zustand `bereit` wechselt oder ein aktiver Prozess terminiert.

EDF kann verdrängend oder nicht-verdrängend realisiert werden. Verdrängendes EDF eignet sich beispielsweise für Echtzeitbetriebssysteme (siehe Abschnitt 3.6) und nicht-verdrängendes EDF für Stapelverarbeitung (siehe Abschnitt 3.4.1).

8.6.10

Fair-Share Scheduling

Fair-Share-Scheduling

The scheduling method *fair-share* distributes the available resources among groups of processes in a fair manner. The computing time is

Das Schedulingverfahren *Fair-Share* verteilt die verfügbaren Ressourcen zwischen Gruppen von Prozessen in einer fairen Art und Weise. Die

allocated to the users and not to the processes. As a result, the computing time, which a user receives, is independent of the number of its processes (see Figure 8.18). The resource shares, received by the users, are called *shares* [58].

Rechenzeit wird den Benutzern und nicht den Prozessen zugeteilt. Das führt dazu, dass die Rechenzeit die ein Benutzer erhält, unabhängig von der Anzahl seiner Prozesse ist (siehe Abbildung 8.18). Die Ressourcenanteile, die die Benutzer erhalten, heißen *Shares* [58].

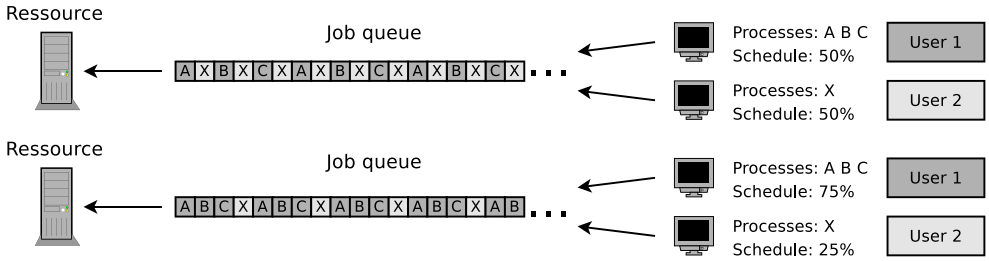

Figure 8.18: Fair-Share fairly distributes Resources between Groups of Processes

Fair-Share is often used in distributed systems as a scheduling method by the job scheduler. The task of a job scheduler (e.g., Maui Cluster Scheduler, Moab Cluster Suite, Oracle Grid Engine, Slurm Workload Manager) is to distribute the compute jobs among the resources within a site. If a job scheduler distributes jobs among the sites of a distributed system, it is called a *meta scheduler* [12].

One example of an operating system that includes a fair share scheduler is Oracle Solaris Version 9 and 10 [74, 88].

Fair-Share wird häufig in verteilten Systemen als Schedulingverfahren vom verwendeten Job-Scheduler eingesetzt. Die Aufgabe eines Job-Schedulers (z.B. Maui Cluster Scheduler, Moab Cluster Suite, Oracle Grid Engine, Slurm Workload Manager) ist die Verteilung der Rechenaufträge auf Ressourcen innerhalb eines Standorts. Verteilt ein Job-Scheduler Aufgaben zwischen den Standorten eines verteilten Systems, heißt dieser *Meta-Scheduler* [12].

Ein Beispiel für ein Betriebssystem, das einen Fair-Share-Scheduler enthält, ist Oracle Solaris Version 9 und 10 [74, 88].

8.6.11

Multilevel Scheduling

Some of the scheduling methods presented so far (SJF, SRTF, LJF, LRTF, and HRRN) are generally unusable in practice because it is impossible to predict the execution times of the processes in advance, and estimations based on statistical data from the past do not provide exact results. The other methods discussed (priority-driven scheduling, FCFS, RR, EDF, and fair-share) are all trade-offs between the different scheduling criteria. One way to address this issue is to implement static or dynamic multilevel scheduling.

Multilevel-Scheduling

Einige der bislang vorgestellten Schedulingverfahren (SJF, SRTF, LJF, LRTF und HRRN) sind in der Praxis meist untauglich, weil die Laufzeiten der Prozesse im Voraus nicht bekannt sind und Abschätzungen durch statistische Erfassung aus der Vergangenheit keine exakten Ergebnisse liefern. Von den übrigen vorgestellten Verfahren (Prioritätengesteuertes Scheduling, FCFS, RR, EDF und Fair-Share) erfordert jedes Kompromisse bezüglich der unterschiedlichen Scheduling-Kriterien. Eine Möglichkeit, damit umzugehen, ist die Implementierung von statischem oder alternativ von dynamischem Multilevel-Scheduling.

The *static multilevel scheduling* method splits the list of processes with the state `ready` into multiple sublists. A different scheduling strategy can be used for each sublist. The sublists have different priorities or time multiplexes (e.g., 80%:20% or 60%:30%:10%). Therefore, static multilevel scheduling can be used to separate time-critical from non-time-critical processes. One example of a useful classification of the processes into different process categories (sublists) with different scheduling methods is shown in Table 8.4.

Beim *statischen Multilevel-Scheduling* wird die `bereit`-Liste in mehrere Teillisten unterteilt. Für jede Teilliste kann eine andere Scheduling-Strategie verwendet werden. Die Teillisten haben unterschiedliche Prioritäten oder Zeitmultiplexe (z.B. 80%:20% oder 60%:30%:10%). Somit ist statisches Multilevel-Scheduling geeignet, um zeitkritische von zeitunkritischen Prozessen zu trennen. Ein Beispiel für eine sinnvolle Unterteilung der Prozesse in verschiedene Prozessklassen (Teillisten) mit verschiedenen Scheduling-Strategien enthält Tabelle 8.4.

Table 8.4: Static Multilevel Scheduling Example with different Process Classes

Priority	Process category	Scheduling method
top	Real-time processes (time-critical)	Priority-driven scheduling
middle	Interactive processes	Round Robin
lowest	Compute-intensive batch processes	First Come First Served

One drawback of static multilevel scheduling is that for correct functioning, each process must be inserted into the correct process category when it is created. Furthermore, static multilevel scheduling is not fair because processes with a low priority can starve [119].

Multilevel feedback scheduling solves these issues by sanctioning processes that have been running for a long time by reducing their priority level [39]. Multilevel feedback scheduling works just like static multilevel scheduling with multiple queues. Each queue has a different priority or time multiplex. Each new process is added to the top queue and gets the highest priority by default (see Figure 8.19).

Each queue uses the scheduling method Round Robin, which eliminates the need for complicated estimations of (remaining) execution times. If a process returns the CPU voluntarily, it is added to the same queue again. If a process has utilized its entire time slice, it is added to the next lower queue with a lower priority. The priorities are, therefore, dynamically assigned with this procedure.

Multilevel feedback scheduling prefers new processes to older (longer running) processes. The scheduling method also prefers processes

Ein Nachteil von statischem Multilevel-Scheduling ist, dass für eine korrekte Arbeitsweise jeder Prozess bei seiner Erzeugung in die passende Prozessklasse eingefügt werden muss. Zudem ist das statischen Multilevel-Scheduling nicht fair, weil Prozesse mit einer niedrigen Priorität verhungern können [119].

Diese Probleme löst das *Multilevel-Feedback-Scheduling*, indem es Prozesse, die schon länger aktiv sind, durch eine Reduzierung von deren Priorität bestraft [39]. Multilevel-Feedback-Scheduling arbeitet genau wie statisches Multilevel-Scheduling mit mehreren Warteschlangen. Jede Warteschlange hat eine andere Priorität oder einen anderen Zeitmultiplex. Jeder neue Prozess wird in die oberste Warteschlange eingefügt. Damit hat er automatisch die höchste Priorität (siehe Abbildung 8.19).

Innerhalb jeder Warteschlange wird Round Robin eingesetzt. Dadurch sind keine komplizierten Abschätzungen der (Rest-)laufzeiten nötig. Gibt ein Prozess den Prozessor freiwillig wieder ab, wird er wieder in die gleiche Warteschlange eingereiht. Hat ein Prozess seine volle Zeitscheibe genutzt, kommt er in die nächst tiefere Warteschlange mit einer niedrigeren Priorität. Die Prioritäten werden bei diesem Verfahren somit dynamisch vergeben.

Multilevel-Feedback-Scheduling bevorzugt neue Prozesse gegenüber älteren (länger laufenden) Prozessen. Das Verfahren bevorzugt auch

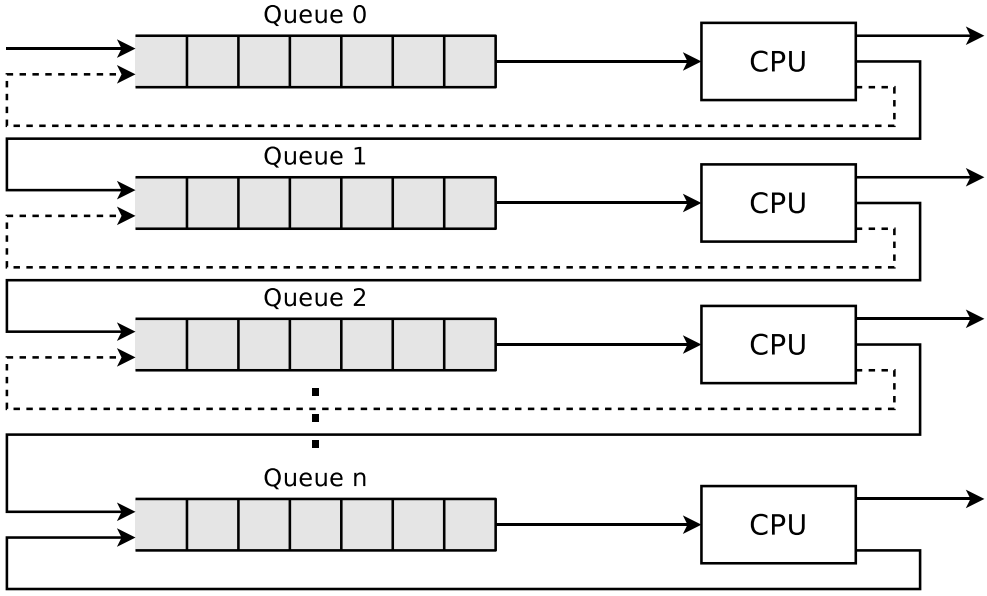

Figure 8.19: In Multilevel Feedback Scheduling, each new Process is added to the top Queue with the highest Priority Level [108]

with many input/output operations, because after releasing the CPU voluntarily, these processes are added again to the original queue, and thus keep their priority level. Older, longer running processes are delayed, which is in practice mostly acceptable.

Many modern operating systems such as Linux (up to kernel 2.4) [69], Mac OS X [65], FreeBSD [75], NetBSD [83], and the Microsoft Windows NT family [17, 69] use variants of the multilevel feedback scheduling for the process scheduling.

Prozesse mit vielen Ein-/Ausgabeoperationen, weil diese nach einer freiwilligen Abgabe des Prozessors wieder in die ursprüngliche Warteliste eingeordnet werden und dadurch ihre Priorität behalten. Ältere, länger laufende Prozesse werden verzögert, was in der Praxis meist akzeptabel ist.

Viele moderne Betriebssysteme wie zum Beispiel Linux (bis Kernel 2.4) [69], Mac OS X [65], FreeBSD [75], NetBSD [83], und die Microsoft Windows NT-Familie [17, 69] verwenden für das Scheduling der Prozesse Varianten des Multilevel-Feedback-Scheduling.

8.6.12

Scheduling of Linux Operating Systems

The Linux operating system kernel has been subject to significant development efforts over the last few decades. Even the process scheduling has been redesigned several times.

Scheduling von Linux-Betriebssystemen

Der Linux-Betriebssystemkern hat in den letzten Jahrzehnten große Weiterentwicklungen erfahren. Auch das Prozess-Scheduling wurde dabei mehrfach neu entwickelt.

Process Scheduling up to Linux Kernel 2.4

Up to kernel 2.4, Linux implements a multilevel feedback scheduling variant with three process classes [69, 107]:

- *Timesharing.* This category includes the normal user processes without real-time demand. The scheduling works preemptively. The scheduling algorithm decides the duration of the next time slice for each process, depending on its priority.

- *Realtime with FIFO.* Processes of this category have the maximum process priority, and the scheduling is non-preemptive. Thus, processes assigned to this category do not get a time slice assigned to them. Such processes are interrupted by the operating system kernel only when...
  - another process with a higher priority in the category *Realtime with FIFO* switches to the *ready* state,
  - the currently running process switches to *blocked* state, or
  - the currently running process voluntarily releases the assigned CPU core.

- *Realtime with Round Robin.* Processes of this category get time slices one after the other. Thus, the scheduling works preemptively. The scheduling algorithm decides the duration of the next time slice for each process, depending on the process priority, and appends the process at the end of the waiting queue when the time slice has expired.

Because the Linux operating system kernel also implements threads, the kernel's scheduling refers to threads, not processes. A process can be treated as a single thread, but a process can also contain multiple threads that share

Prozess-Scheduling bis Linux Kernel 2.4

Bis Kernel 2.4 implementiert Linux eine Variante des Multilevel-Feedback-Scheduling mit drei Prozessklassen [69, 107]:

- *Timesharing.* Diese Klasse enthält die normalen Benutzerprozesse ohne Echtzeitanforderungen. Das Scheduling arbeitet präemptiv. Der Scheduling-Algorithmus ermittelt für jeden Prozess die Länge der nächsten Zeitscheibe, abhängig von der Prozesspriorität.

- *Realtime mit FIFO.* Prozesse dieser Klasse haben die höchste Prozesspriorität und das Scheduling ist nicht-präemptiv. Prozessen, die dieser Klasse zugeordnet sind, wird somit keine Zeitscheibe zugeordnet. Solche Prozesse werden nur dann vom Betriebssystemkern unterbrochen, wenn...
  - ein anderer Prozess mit einer höheren Priorität in der Klasse *Realtime mit FIFO* in den Zustand *bereit* wechselt,
  - der aktuell laufende Prozess in den Zustand *blockiert* wechselt, oder
  - der aktuell laufende Prozess den zugewiesenen Prozessorkern freiwillig abgibt.

- *Realtime mit Round Robin.* Prozesse dieser Klasse erhalten nacheinander eine Zeitscheibe. Das Scheduling arbeitet somit präemptiv. Der Scheduling-Algorithmus ermittelt für jeden Prozess die Länge der nächsten Zeitscheibe, abhängig von der Prozesspriorität, und fügt den Prozess nach Ablauf der Zeitscheibe wieder am Ende der Warteschlange ein.

Da der Linux-Betriebssystemkern auch Threads realisiert, bezieht sich das Scheduling des Kernels eigentlich auf Threads und nicht auf Prozesse. Ein Prozess kann als ein einzelner Thread angesehen werden, aber ein Prozess kann auch mehrere Threads enthalten, die sich

resources. For convenience, this section only mentions processes.

The keyword *Realtime* must not be misinterpreted. Linux is not a hard but a soft real-time operating system (see Section 3.6.1) [69].

For processes in the categories *Timesharing* and *Realtime with Round Robin*, the time slice length is reduced with each further time slice. If a process returns the assigned CPU core before the end of the current time slice, the time slice length is preserved for the next allocation of a CPU core [114, 115].

The manual mapping of individual processes to the three process categories, and thus to the different scheduling strategies, is done in this and the more modern Linux schedulers O(1) and CFS with the command line tool `chrt`. It implements the command line parameters `--other`, `--fifo`, and `--rr` for allocating individual processes based on their process number (PID) to each process category presented in this section.

Process Scheduling in Linux Kernel 2.6 to 2.6.22 with the O(1) Scheduler

The Linux operating system kernel revisions 2.6 (December 2003) up to 2.6.22 (July 2007) implement the so-called *O(1) scheduler*. However, the three process categories have remained the same.

The operating system kernel maintains for every CPU core the data structure *runqueue*, which contains, among other things, two priority arrays and two pointers to these arrays for the scheduling [11]. Each of the two arrays contains a chained list for each process priority that maintains the processes with that process priority (see Figure 8.20).

The O(1) scheduler implements the two process categories *Timesharing* for the normal (non-real-time) user processes and *Realtime* for processes that demand soft real-time operation [69].

The two process categories differ in the value ranges of the dynamic priority levels. The O(1) scheduler implements 140 dynamic priority levels (see Figure 8.20). The smaller the value, the

Ressourcen teilen. Zur Vereinfachung wird im Verlauf dieses Abschnitts nur von Prozessen gesprochen.

Das Schlüsselwort *Realtime* sollte an dieser Stelle nicht falsch interpretiert werden. Linux realisiert kein hartes sondern ein weiches Echtzeitbetriebssystem (siehe Abschnitt 3.6.1) [69].

Bei Prozessen in den Klassen *Timesharing* und *Realtime mit Round Robin* wird mit jeder weiteren Zeitscheibe die Länge der Zeitscheiben verkürzt. Gibt ein Prozess den zugewiesenen Prozessorkern vor dem Ende der laufenden Zeitscheibe wieder ab, bleibt die Länge der Zeitscheibe bei der nächsten Zuweisung eines Prozessorkerns gleich [114, 115].

Die manuelle Zuordnung einzelner Prozesse zu den drei Prozessklassen und damit zu den unterschiedlichen Scheduling-Strategien geschieht bei diesem und den späteren Linux-Schedulern O(1) und CFS mit dem Kommandozeilenwerkzeug `chrt`. Es ermöglicht mit den Kommandozeilenparametern `--other`, `--fifo` und `--rr` die Zuweisung einzelner Prozesse anhand ihrer Prozessnummer (PID) zu den einzelnen in diesem Abschnitt vorgestellten Prozessklassen.

Prozess-Scheduling in Linux Kernel 2.6 bis 2.6.22 mit dem O(1)-Scheduler

Die Linux-Betriebssystemkerne in den Versionen 2.6 (Dezember 2003) bis 2.6.22 (Juli 2007) implementieren den sogenannten *O(1)-Scheduler*. Die drei Prozessklassen haben sich dabei nicht geändert.

Der Betriebssystemkern verwaltet für jeden Prozessorkern die Datenstruktur *Ausführungswarteschlange* (englisch: *Runqueue*), die u.a. zwei Arrays (englisch: *Priority Arrays*) und zwei Zeiger auf diese Arrays für das Scheduling enthält [11]. Jedes der beiden Arrays enthält für jede Prozesspriorität eine verkettete Liste, die die Prozesse mit der jeweiligen Prozesspriorität verwaltet (siehe Abbildung 8.20).

Der O(1)-Scheduler implementiert die beiden Prozessklassen *Timesharing* für die normalen Benutzerprozesse ohne Echtzeitanforderungen und *Realtime* für Prozesse mit weichen Echtzeitanforderungen [69].

Der Unterschied zwischen beiden Prozessklassen sind die unterschiedlichen Wertebereiche der dynamischen Prioritätsstufen. Beim O(1)-Scheduler gibt es 140 dynamische Prioritätsstu-

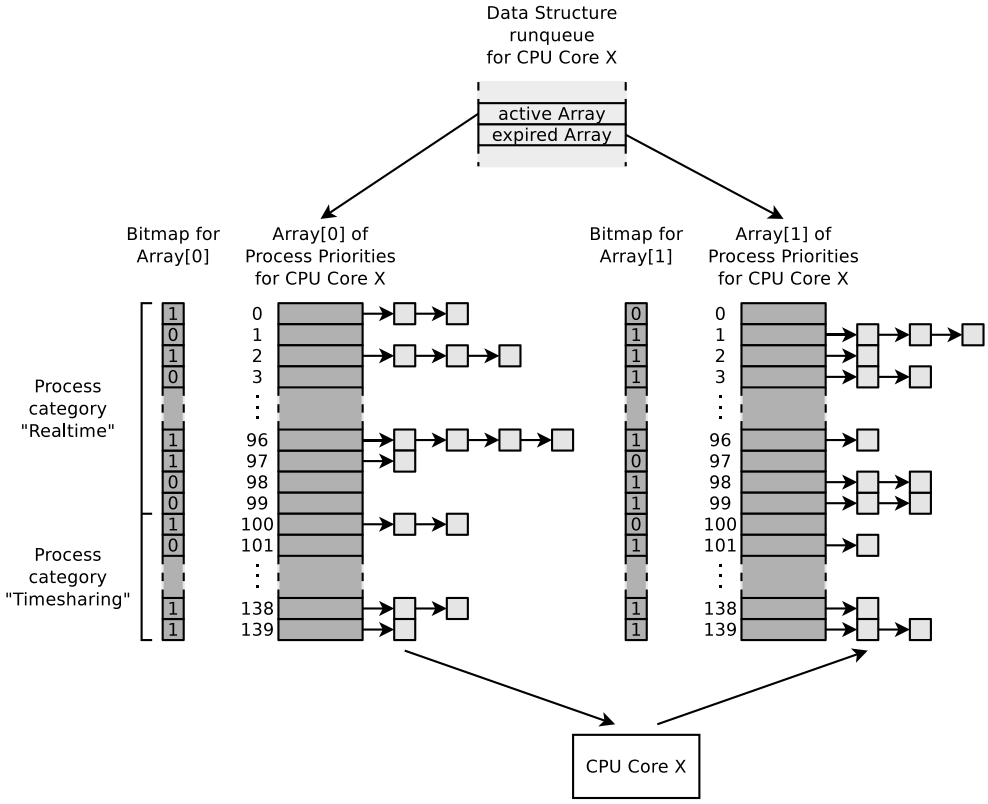

Figure 8.20: Working Method of the Linux O(1)-Scheduler

higher the priority. The value range for real-time processes is 0-99, and the value range for timesharing processes is 100-139.

The process in the active array with the highest dynamic priority level gets the CPU core assigned next for a time slice. Searching the process with the highest dynamic priority level is done in a resource-saving way because, for each array, the operating system kernel maintains a bitmap with one bit for each priority level [5]. For each priority level for which at least one process exists in the array, the corresponding bit position in the bitmap has the value 1. Otherwise, the bit position in the bitmap has the value 0. For searching the next process with the highest dynamic priority level, only the bitmap must be searched for the highest priority level that currently has processes assigned to it [52].

fen (siehe Abbildung 8.20). Je kleiner der Wert, desto höher ist die Priorität. Der Wertebereich für Realtime-Prozesse ist 0-99 und der Wertebereich für Timesharing-Prozesse ist 100-139.

Der Prozess im aktiven Array mit der höchsten dynamischen Prioritätsstufe bekommt als nächstes den Prozessorkern für eine Zeitscheibe zugewiesen. Die Suche nach dem Prozess mit der höchsten dynamischen Prioritätsstufe geschieht ressourcenschonend, denn für jedes Array verwaltet der Betriebssystemkern eine Bitmap mit einem Bit für jede Prioritätsstufe [5]. Für jede Prioritätsstufe, für die mindestens ein Prozess im Array existiert, hat die entsprechende Bitposition im Bitmap den Wert 1. Ansonsten hat die Bitposition in der Bitmap den Wert 0. Für die Suche nach dem nächsten Prozess mit der höchsten dynamischen Prioritätsstufe muss nur die Bitmap dahingehend durchsucht werden, wel-

As soon as the array marked as active is completely emptied, the scheduler switches the arrays. Doing so is simply swapping the pointers to the two arrays, which does not cause significant overhead. Furthermore, by switching the arrays after all processes in the state *ready* have been executed for a time slice, a fair execution of all processes is ensured [115].

The O(1) scheduler favors in the timesharing process category interactive processes over processes that run for a longer time. In particular, the scheduler favors processes that frequently utilize a CPU core for only a short time and then voluntarily release it. For this purpose, the scheduler awards a bonus or malus in the value range +5 to -5 to the process priority level. Time-critical processes whose task is input/output and user interaction are thus automatically executed preferentially [69].

The initial priority level of a process of the timesharing process category results from the static process priority (`nice`-value). For timesharing processes, the 40 static process priorities in Linux are mapped to the dynamic priority levels from 100 to 139.

The 40 static process priorities in Linux have the value range −20 to +19 (in integer values). The value −20 is the highest priority, and 19 is the lowest one. The default priority level is 0. Normal users can assign priorities from 0 to 19. The system administrator (`root`) may also assign negative values to processes.

Consequently, the 0(1) scheduler inserts a normal user process with the default `nice`-value of 0, during the initial assignment into the lists, to the dynamic process priority 119. Furthermore, a process can improve its position in the scheduling by up to five priority levels because of interactive behavior, or it can lower its position by up to five priority levels because of its CPU-intensive behavior. The distinction between interactive and CPU-intensive processes is a precondition for obtaining the bonus or malus of up to five priority levels. The scheduler achieves this by monitoring the runtime behavior [67].

ches die höchste Prioritätsstufe ist, der aktuell Prozesse zugeordnet sind [52].

Sobald das als aktiv markierte Array vollständig geleert ist, wechselt der Scheduler die Arrays. Dabei handelt es sich lediglich um einen Tausch der Zeiger auf die beiden Arrays, was keinen signifikanten Verwaltungsaufwand verursacht. Durch das Wechseln der Arrays nach einem Durchlauf aller Prozesse im Zustand *bereit* ist eine faire Abarbeitung aller Prozesse gewährleistet [115].

Der O(1)-Scheduler bevorzugt bei der Prozessklasse Timesharing interaktive Prozesse vor Prozessen die länger laufen. Konkret bevorzugt der Scheduler solche Prozesse, die häufig einen Prozessorkern nur kurze Zeit belegen und ihn dann freiwillig wieder abgeben. Um dieses zu gewährleisten vergibt der Scheduler einen Bonus oder Malus im Wertebereich +5 bis-5 auf die Prioritätsstufe. Zeitkritische Prozesse, deren Aufgabe Ein-/Ausgabe und Benutzerinteraktion ist, werden somit automatisch bevorzugt abgearbeitet [69].

Die initiale Prioritätsstufe eines Prozesses der Prozessklasse Timesharing ergibt sich aus der statischen Prozesspriorität (`nice`-Wert). Die 40 möglichen statischen Prozessprioritäten unter Linux werden bei Timesharing-Prozessen einfach auf die dynamischen Prioritätsstufen von 100 bis 139 abgebildet.

Die 40 statischen Prozessprioritäten haben unter Linux den Wertebereich −20 bis +19 (in ganzzahligen Schritten). Der Wert −20 ist die höchste Priorität und 19 die niedrigste Priorität. Die Standardpriorität ist 0. Normale Benutzer können Prioritäten von 0 bis 19 vergeben. Der Systemverwalter (`root`) darf Prozessen auch negative Werte zuweisen.

Das heißt für den 0(1)-Scheduler, dass ein normaler Benutzerprozess mit dem standardmäßigen `nice`-Wert von 0 bei der initialen Eingruppierung in die Listen für die dynamische Prozesspriorität 119 eingefügt wird. Zudem kann ein Prozess seine Position beim Scheduling durch interaktives Verhalten um bis zu fünf Prioritätsstufen verbessern oder durch prozessorlastiges Verhalten um bis zu fünf Prioritätsstufen verschlechtern. Voraussetzung zur Gewährung des Bonus oder Malus von bis zu fünf Prioritätsstufen ist die Unterscheidung von interaktiven und prozessorlastigen Prozessen. Dieses realisiert der

Processes of process classification Realtime keep their priority levels [5] and can belong either to process category *Realtime with FIFO* (non-preemptive) or to *Realtime with Round Robin* (preemptive).

The time slice length of a timesharing process is determined for each run from its dynamic priority level, hence from the assigned static process priority (the `nice`-value) and the consequences of its process interactivity. The time slice length is calculated as follows [11]:

Prozesse der Prozessklasse Realtime behalten ihre Prioritätsstufen [5] und können entweder zur Prozessklasse *Realtime mit FIFO* (nichtpräemptiv) oder zu *Realtime mit Round Robin* (präemptiv) gehören.

Die Zeitscheibendauer eines Timesharing-Prozesses ergibt sich für jeden Durchlauf aus seiner dynamischen Prioritätsstufe, also aus der zugeordneten statischen Prozesspriorität (dem `nice`-Wert) und den Konsequenzen seiner Prozessinteraktivität. Die Zeitscheibendauer berechnet sich wie folgt [11]:

$$\text{time slice lengt [ms]} = \begin{cases} (140 - \text{priority level}) * 5 & \text{(if priority level 120-139)} \\ (140 - \text{priority level}) * 20 & \text{(if priority level 100-119)} \end{cases}$$

Since only the system administrator (`root`) may assign processes negative values as static process priority (`nice`-values from -20 to -1), which result in the dynamic priority levels 100-119, obviously such processes are massively preferred when calculating the time slice length. Table 8.5 shows for some priority levels the time slice length calculated by the 0(1) scheduler.

Da nur der Systemverwalter (`root`) Prozessen auch negative Werte als statische Prozesspriorität (`nice`-Werte von -20 bis -1) zuweisen darf, die in den dynamischen Prioritätsstufen 100-119 resultieren, ist es offensichtlich, dass solche Prozesse bei der Berechnung der Zeitscheibendauer massiv bevorzugt werden. Tabelle 8.5 zeigt für einige Prioritätsstufen die Zeitscheibendauern beim 0(1)-Scheduler.

Table 8.5: Time Slice Lengths of normal User Processes (Process Category Timesharing) of the O(1) Scheduler

Priority level	`nice` value	Time slice length
100	-20	800 ms
101	-19	780 ms
102	-18	760 ms
...	...	...
117	-3	460 ms
118	-2	440 ms
119	-1	420 ms
120	0	100 ms
121	1	95 ms
122	2	90 ms
123	3	85 ms
...	...	...
137	17	15 ms
138	18	10 ms
139	19	5 ms

The allocated time slice length does not need to be utilized in one run, and, depending on the situation, this may not be possible. For example,

Die zugewiesene Zeischeibendauer muss nicht in einem Durchlauf verbraucht werden und je nach Situation ist das auch gar nicht möglich.

if there is an interrupt due to the presence of a process with a higher priority level or due to an interrupt handling, the process that is in execution state keeps the remaining time slice length and gets reassigned to the list of the same priority level.

For systems with multi-core CPUs or systems with multiple CPUs, there may be a strong imbalance in the distribution of processes among the CPU cores [5, 69]. To compensate such imbalances implements the O(1) scheduler a *load balancer* which, for each runqueue, monitors the equal distribution of the processes over the available runqueues (i.e., CPU cores) when the pointers to the arrays get switched as well as at regular intervals. If necessary, the scheduler allocates individual processes to other runqueues.

The scheduler's name indicates the time complexity of the algorithms behind it. The time complexity is O(1), which implies that the scheduler always requires the same CPU time for its operation. The time needed to calculate the execution order of the processes automatically is independent of the number of threads or processes the scheduler needs to handle [115].

Process Scheduling in Linux Kernel 2.6.23 to 6.5.13 with the CFS Scheduler

The Linux operating system kernels revisions 2.6.23 (October 2007) up to 6.5.13 (November 2023) implement the so-called *Completely Fair Scheduler* (CFS). This scheduler is much more simplified because it does not need a data structure such as the runqueue, does not compute time slices, and does not switch back and forth between arrays for active and expired processes.

The purpose of this scheduler is to allocate a similarly large (fair) share of CPU time to all processes assigned to a CPU core [69]. Thus, for n processes, each process is to be allocated a share of $1/n$ of the available CPU time. For example, if five processes are assigned to a CPU core, the scheduler's goal is to allocate 20% of the CPU time to each process [40].

Wenn es zum Beispiel durch die Ankunft eines Prozesses mit einer höheren Prioritätsstufe oder wegen der Behandlung eines Interrupts zu einer Unterbrechung kommt, behält der rechnende Prozess die noch übrige Restdauer seiner Zeitscheibe und wird wieder in die Liste der gleichen Prioritätsstufe eingereiht.

Bei Systemen mit Mehrkernprozessoren oder bei Systemen mit mehreren Prozessoren kann es zu einem starken Ungleichgewicht in der Verteilung der Prozesse auf die Prozessorkerne kommen [5, 69]. Um diesem entgegenzuwirken, implementiert der O(1)-Scheduler einen *Lastverteiler* (englisch: *Load Balancer*), der für jede Ausführungswarteschlange beim Wechsel der Zeiger auf die Arrays und in regelmäßigen Abständen die gleichmäßige Verteilung der Prozesse auf die verfügbaren Prozessorkerne bzw. Ausführungswarteschlangen kontrolliert und gegebenenfalls einzelne Prozesse anderen Ausführungswarteschlangen (also Prozessorkernen) zuweist.

Die Benennung des Schedulers weist auf die Zeitkomplexität der zugrundeliegenden Algorithmen hin. Diese ist O(1), was bedeutet, dass der Scheduler für seine Funktion immer die gleiche Prozessorzeit benötigt. Die nötige Zeit zur automatischen Berechnung der Ausführungsreihenfolge der Prozesse ist unabhängig von der Anzahl der verwalteten Threads bzw. Prozesse [115].

Prozess-Scheduling in Linux Kernel 2.6.23 bis 6.5.13 mit dem CFS-Scheduler

Die Linux-Betriebssystemkerne in den Versionen 2.6.23 (Oktober 2007) bis 6.5.13 (November 2023) implementieren den sogenannten *Completely Fair Scheduler* (CFS). Dieser Scheduler stellt eine Vereinfachung dar, da er ohne eine Datenstruktur wie die Ausführungswarteschlange (Runqueue) auskommt, keine Zeitscheiben berechnet und nicht zwischen Arrays für aktive und abgelaufene Prozesse hin- und herwechselt.

Ziel des Schedulers ist es, allen Prozessen, die einem Prozessorkern zugeordnet sind, einen ähnlich großen (fairen) Anteil Rechenzeit zuzuteilen [69]. Bei n Prozessen soll also jedem Prozess ein Anteil von $1/n$ der verfügbaren Rechenzeit zugeordnet werden. Sind also z.B. fünf Prozesse einem Prozessorkern zugeordnet, ist es das Ziel des Schedulers jedem Prozess 20% der Rechenzeit zuzuweisen [40].

The Linux operating system kernel implements a CFS scheduler for each CPU core and a variable *vruntime* (virtual runtime) for each process. The variable's value represents a virtual CPU runtime in nanoseconds and informs about how much CPU time the particular process has already been utilized. The process with the lowest vruntime gets access to the CPU core next and is permitted to utilize it until its vruntime value has reached a fair level, i.e., until it has reached the intended share of $1/n$ of the available CPU time. The scheduler attempts to achieve an as equal as possible vruntime value for all processes [69].

The organization of the processes is done by a red-black tree [30, 104] (i.e., a self-balancing binary search tree). In it, the processes are arranged based on their vruntime values (see Figure 8.21). The values are the keys of the inner nodes. The leaf nodes (NIL nodes) do not have keys and do not contain any data. NIL stands for *none*, *nothing*, *null*, etc., which means a null value or null pointer, depending on the programming language.

The processes assigned to a CPU core are arranged in the red-black tree in such a way that the processes with the highest demand for CPU time (lowest vruntime) are on the left-hand side of the tree and the processes with the lowest demand (highest vruntime) for CPU time are on the right-hand side. For fairness, the CFS scheduler assigns the CPU core next to the process on the far left of the red-black tree. If the process is removed from the CPU core before it terminates and is inserted into the red-black tree again, its vruntime value is increased by the time the process was allowed to compute on the CPU core. The nodes (processes) in the red-black tree thus wander continuously from right to left, and a fair distribution of the CPU resources is ensured [57].

The CFS scheduler considers the static process priorities (`nice`-value) of the processes in such a way that a low `nice`-value (high static process priority) results in an effectively longer CPU time and a high `nice`-value (low static process priority) results in an effectively shorter CPU time. The vruntime values are

Der Linux-Betriebssystemkern realisiert für jeden Prozessorkern einen CFS-Scheduler und für jeden Prozess eine Variable *vruntime* (virtual runtime). Der Wert der Variablen repräsentiert eine virtuelle Prozessorlaufzeit in Nanosekunden und informiert darüber, wie lange der jeweilige Prozess schon gerechnet hat. Der Prozess mit der niedrigsten vruntime bekommt als nächstes Zugriff auf den Prozessorkern und darf so lange rechnen, bis sein vruntime-Wert wieder auf einem fairen Niveau ist, also sich dem angestrebten Anteil von $1/n$ der verfügbaren Rechenzeit angenähert hat. Der Scheduler strebt einen möglichst gleichen vruntime-Wert für alle Prozesse an [69].

Die Verwaltung der Prozesse geschieht mit Hilfe eines Rot-Schwarz-Baums (also eines selbstbalancierenden binären Suchbaums) [30, 104]. Dort werden die Prozesse anhand der vruntime-Werte einsortiert (siehe Abbildung 8.21). Die Werte sind die Schlüssel der inneren Knoten. Die Blattkonten (NIL-Knoten) haben keine Schlüssel und enthalten keine Daten. NIL ist dabei stellvertretend für *none*, *nothing*, *null*, etc. also je nach Programmiersprache für einen Null-Wert oder Null-Pointer.

Die einem Prozessorkern zugeordneten Prozesse sind im Rot-Schwarz-Baum dahingehend einsortiert, dass die Prozesse mit dem höchsten Bedarf an Prozessorzeit (niedrigste vruntime) auf der linken Seite des Baums sind und die Prozesse mit dem geringsten Bedarf (höchste vruntime) an Prozessorzeit auf der rechten Seite. Aus Fairnessgründen weist der CFS-Scheduler dem Prozess ganz links im Rot-Schwarz-Baum als nächstes den Prozessorkern zu. Wird der Prozess vor seiner Beendigung vom Prozessorkern verdrängt und erneut in den Rot-Schwarz-Baum eingefügt, erhöht sich der Wert um die Zeit, die der Prozess auf dem Prozessorkern rechnen durfte. Die Knoten (Prozesse) im Rot-Schwarz-Baum wandern somit kontinuierlich von rechts nach links und eine faire Verteilung der Ressource Rechenleistung ist gewährleistet [57].

Die statischen Prozessprioritäten (`nice`-Werte) der Prozesse berücksichtigt der CFS-Scheduler in der Art und Weise, dass ein niedriger `nice`-Wert (hohe statische Prozesspriorität) zu einer effektiv höheren Rechenzeit führt und ein hoher `nice`-Wert (niedrige statische Prozesspriorität) zu einer effektiv niedrigeren Rechen-

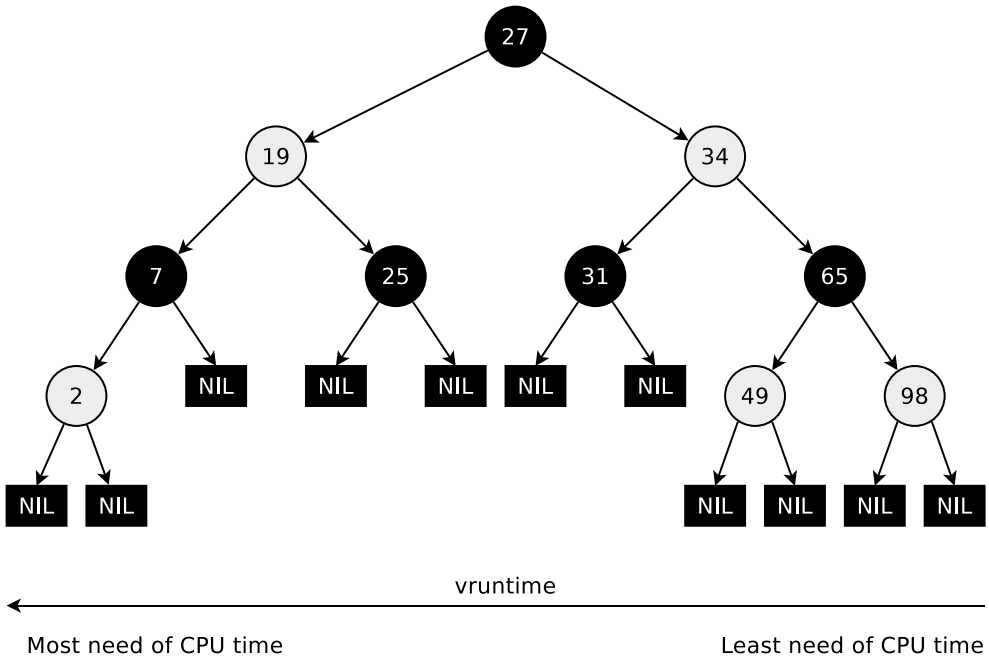

Figure 8.21: The CFS scheduler manages the processes assigned to a CPU core using a red-black tree based on their vruntime values [57]

thus weighted differently depending on the `nice` value. In other words, the virtual clock can run at different speeds [40, 68].

Process Scheduling since Linux Kernel 6.6 with the EEVDF Scheduler

The Linux operating system kernels since revision 6.6 (October 2023) implement the so-called *Earliest Eligible Virtual Deadline First* (EEVDF).

EEVDF combines the fairness concept of CFS (Completely Fair Scheduler) with deadline-based scheduling like EDF (Earliest Deadline First) from real-time systems. Both CFS and EEVDF...

- aim to provide all processes a similar (fair) share of computing time of the CPU core they are assigned to.

zeit. Die vruntime-Werte werden also abhängig vom `nice`-Wert unterschiedlich gewichtet. Anders gesagt: Die virtuelle Uhr kann unterschiedlich schnell laufen [40, 68].

Prozess-Scheduling seit Linux Kernel 6.6 mit dem EEVDF-Scheduler

Die Linux-Betriebssystemkerne seit Version 6.6 (Oktober 2023) implementieren den sogenannten *Earliest Eligible Virtual Deadline First* (EEVDF).

Der EEVDF-Scheduler kombiniert das Fairness-Konzept von CFS (Completely Fair Scheduler) mit Deadline-basiertem Scheduling wie EDF (Earliest Deadline First) aus Echtzeitsystemen. CFS und EEVDF beide...

- zielen darauf ab, allen Prozessen einen gleichen (fairen) Anteil der Rechenzeit des CPU-Kerns, dem sie zugewiesen sind, bereitzustellen.

- use the static process priorities (`nice` values) to make the virtual clock (`vruntime`) run at different speeds.

EEVDF introduces some new values for each process. These are *lag*, *eligibility*, *eligible time* and *virtual deadline*.

With EEVDF, the kernel keeps a *lag* value for each process. The lag is the difference between the ideal (computed) CPU time the process should have received and the CPU time it received. A negative lag indicates that the process has been allocated too much CPU time. A positive lag indicates that the process has not received its fair share of CPU time. [28]

The scheduler calculates the lag for each process using this equation [24]:

- verwenden die statischen Prozessprioritäten (`nice`-Werte), um die virtuelle Uhr (`vruntime`) unterschiedlich schnell laufen zu lassen.

EEVDF führt für jeden Prozess einige neue Werte ein. Diese sind *Verzögerung* (englisch: *Lag*), *Eignung* (englisch: *Eligibility*), *Zeit bis zur Eignung* (englisch: *Eligible time*) und *virtuelle Deadline*.

Mit EEVDF verwaltet der Kernel einen *Lag*-Wert für jeden Prozess. Der Lag ist die Differenz von idealer (berechneter) CPU-Zeit, die er hätte bekommen sollen, und der erhaltenen Zeit. Ein negativer Lag sagt aus, dass dem Prozess zu viel CPU-Zeit zugewiesen wurde. Ein positiver Lag hingegen sagt aus, dass der Prozess seinen fairen Anteil Zeit nicht erhalten hat. [28]

Der Scheduler berechnet den Lag-Wert für jeden Prozess mit Hilfe der folgenden Formel [24]:

$$\text{Process lag} = \text{Process current } \texttt{vruntime} - \text{average of every process's } \texttt{vruntime}$$

Only processes with a lag value that is positive or zero are *eligible* to run. The motivation for maintaining lag and eligibility is to achieve more fairness in process scheduling. [59]

The following example (taken from [27]) visualizes the calculation of the process lag values:

Nur Prozesse mit einem nicht-negativem Lag sind zur Ausführung *geeignet* (englisch: *eligible*). Der Nutzen der Verwaltung von Lag und Eignung ist eine größere Fairness beim Prozess-Scheduling. [59]

Das folgende Beispiel (entnommen von [27]) zeigt anschaulich die Berechnung der Lag-Werte:

Process	A	B	C
lag [ms]	0	0	0
Eligible	yes	yes	yes

In the example, three processes are assigned to the same CPU core and start at the same time. Therefore, they all start with a lag of value zero. Since none of the processes has a negative lag, all of them can run. We assume in this example that all processes have the same static priority (`nice` value), and that the time slice length for each process is 30 ms. We also assume, that the scheduler decides process A runs first, and A runs for the entire time slice. This results in the following new lag values:

Each process got 1/3 of the total CPU time (10 ms of 30 ms). Process A ran for 30 ms, so its new lag is -20 ms. Processes B and C did not run,

Im Beispiel sind drei Prozesse demselben CPU-Kern zugewiesen und starten zur gleichen Zeit. Das bedeutet, zu Beginn haben sie alle einen Lag mit dem Wert Null. Da kein Prozess einen negativen Lag aufweist, sind alle zur Ausführung geeignet. Im Beispiel nehmen wir an, dass alle Prozesse die gleiche statische Priorität (`nice`-Wert) haben und dass die Zeitscheibenlänge für jeden Prozess 30 ms beträgt. Zudem nehmen wir an, dass der Scheduler festlegt, dass Prozess A zuerst läuft, und A läuft über die gesamte Zeitscheibe. Somit ergeben sich folgende neue Lag-Werte:

Jeder Prozess hat 1/3 der gesamten CPU-Zeit (10 ms von 30 ms) erhalten. Prozess A ist 30 ms gelaufen, also beträgt sein neuer Lag-Wert

Process	A	B	C
lag [ms]	-20	10	10
Eligible	no	yes	yes

so they now each have 10 ms lag. We assume in this example that the scheduler decides process B runs next, and B runs for the entire time slice. This results in the following lag values:

-20 ms. Die Prozesse B und C sind nicht gelaufen und verfügen nun jeweils über einen Lag-Wert von 10 ms. Für den weiteren Verlauf des Beispiels nehmen wir an, dass der Scheduler festlegt, dass Prozess B als nächstes läuft, und B läuft über die gesamte Zeitscheibe. Somit ergeben sich folgende neue Lag-Werte:

Process	A	B	C
lag [ms]	-10	-10	20
Eligible	no	no	yes

Each process again got 1/3 of the total CPU time (10 ms of 30 ms). Process B ran for 30 ms, so its new lag is -10 ms. Process C did not run and now has a 20 ms lag. The scheduler's next decision will be process C, as it is now the only eligible process.

Jeder Prozess hat erneut 1/3 der gesamten CPU-Zeit (10 ms von 30 ms) erhalten. Prozess B ist 30 ms gelaufen, also beträgt sein neuer Lag-Wert -10 ms. Prozess C ist nicht gelaufen und verfügt nun über einen Lag-Wert von 20 ms. Der Scheduler wird Prozess C als nächstes auswählen, weil er aktuell der einzige geeignete Prozess ist.

The example demonstrates that the sum of all lag values of the processes assigned to a CPU core will always be zero.

Das Beispiel zeigt, dass die Summe aller Lag-Werte der Prozesse, die einem CPU-Kern zugeordnet sind, stets dem Wert Null entspricht.

While lag values and eligibility are primarily about fairness, the EEVDF scheduler maintains a *virtual deadline* and the *eligible time* for each process to improve the latency for real-time and interactive processes.

Während Lag-Werte und die Verwaltung der Eignung hauptsächlich auf die Fairness abzielen, verwaltet der EEVDF-Scheduler auch für jeden Prozess eine *virtuelle Deadline* und die *Zeit bis zur Eignung*, um die Latenz von Echtzeit- und interaktiven Prozessen zu verbessern.

The process with the earliest eligible deadline that is eligible to run will run next. The virtual deadline is calculated by the scheduler using the eligible time of the process and its time slice length (this depends on the static priority = `nice` value). The eligible time indicates when a process becomes eligible again.

Der Prozess mit der kürzesten Deadline, der für eine Ausführung in Frage kommt, läuft als nächstes. Die virtuelle Deadline berechnet der Scheduler anhand der Zeit bis zur Eignung des Prozesses und dessen Zeitscheibenlänge (diese ist abhängig von der statischen Priorität = `nice`-Wert). Die Zeit bis zur Eignung gibt an, wann ein Prozess wieder zur Ausführung zugelassen wird.

9

Interprocess Communication

Interprozess-kommunikation

Processes do not only perform read and write operations on data. They also often need to call each other, wait for each other, and coordinate with each other. In short, processes must be able to interact with each other. This functionality is called *interprocess communication* (IPC), and this chapter describes the various ways in which processes can transmit information to others and access shared resources.

Prozesse müssen nicht nur Lese- und Schreibzugriffe auf Daten ausführen, sondern sie müssen sich auch häufig gegenseitig aufrufen, aufeinander warten und sich untereinander abstimmen. Kurz gesagt: Prozesse müssen miteinander interagieren können. Diese Funktionalität heißt *Interprozesskommunikation* (IPC) und das vorliegende Kapitel beschreibt die verschiedenen Möglichkeiten, wie Prozesse Informationen an andere Prozesse weiterreichen und auf gemeinsame Ressourcen zugreifen können.

9.1

Critical Sections and Race Conditions

Kritische Abschnitte und Wettlaufsituationen

If multiple processes run in parallel, a distinction is made between critical and uncritical sections. In *uncritical sections*, the processes do not access shared data at all or only carry out read operations on shared data. In *critical sections*, the processes carry out read and write operations on shared data. For avoiding errors, critical sections must not be processed by multiple processes at the same time. For enabling processes to access shared memory, and the data stored on it, without conflict, the operating system must provide a mechanism for *mutual exclusion*.

Laufen mehrere parallel ausgeführte Prozesse, unterscheidet man kritische von unkritischen Abschnitten. Bei *unkritischen Abschnitten* greifen die Prozesse gar nicht oder nur lesend auf gemeinsame Daten zu. Bei *kritischen Abschnitten* hingegen greifen die Prozesse lesend und schreibend auf gemeinsame Daten zu. Um Fehler zu vermeiden, dürfen kritische Abschnitte nicht von mehreren Prozessen gleichzeitig durchlaufen werden. Damit Prozesse konfliktfrei auf einen gemeinsam genutzten Speicher und die darauf abgelegten Daten zugreifen können, muss das Betriebssystem einen Mechanismus zum *wechselseitigen Ausschluss* (englisch: *Mutual Exclusion*) bereitstellen.

One example of a critical section from [112] is the scenario of a printer spooler in Table 9.1. The processes P_A and P_B each want to print a document (see Figure 9.1). The spooler uses two

Ein Beispiel für einen kritischen Abschnitt aus [115] ist das folgende Szenario eines Drucker-Spoolers in Tabelle 9.1. Die beiden Prozesse P_A und P_B möchten jeweils ein Dokument ausdru-

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_9

variables whose values are stored in a text file and can, therefore, be accessed by all processes. The variable `out` stores the number of the next document to be printed in the spooler directory, and the variable `in` stores the number of the next free entry in the spooler directory. Each process must store its document in the spooler directory and then increment the value of the variable `in` by one.

Process P_A first reads the value of the variable `in`, to obtain the next free number (it is the number 16). A process switching occurs, and process P_B reads the value of the variable `in` to obtain the next free number. The variable still has the value 16. Process P_B then adds its print job into the memory location with the number 16. As soon as the print job has been transferred, process P_B increments the counter variable of the spooling directory by one. Next, the process switching takes place again, and P_A still assumes that the next free memory location is the one with the number 16. Therefore, P_A also writes its print job into the memory location with the number 16, which overwrites the print job from process P_B. Finally, process P_A also increments the counter variable of the spooler directory by one.

cken (siehe Abbildung 9.1). Der Spooler arbeitet mit zwei Variablen, deren Werte in einer Textdatei gespeichert werden und somit für alle Prozesse erreichbar sind. Die Variable `out` speichert die Nummer des nächsten auszudruckenden Dokuments im Spooler-Verzeichnis und die Variable `in` speichert die Nummer des nächsten freien Eintrags im Spooler-Verzeichnis. Jeder Prozess muss sein Dokument in das Spooler-Verzeichnis schreiben und danach den Wert der Variable `in` um den Wert 1 erhöhen.

Prozess P_A liest im ersten Schritt den Wert der Variable `in`, um die Nummer des nächsten freien Speicherplatzes (es ist die Nummer 16) zu erhalten. Daraufhin kommt es zum Prozesswechsel und Prozess P_B liest im ersten Schritt den Wert der Variable `in`, um die Nummer des nächsten freien Speicherplatzes zu erfahren. Die Variable speichert unverändert den Wert 16. Daraufhin schreibt Prozess P_B seinen Druckauftrag in den Speicherplatz mit der Nummer 16. Sobald der Druckauftrag übermittelt ist, erhöht Prozess P_B die Zählvariable des Spooler-Verzeichnisses um den Wert 1. Nun kommt es erneut zum Prozesswechsel und P_A nimmt immer noch an, das der nächste freie Speicherplatz derjenige mit der Nummer 16 ist. Aus diesem Grund schreibt P_A seinen Druckauftrag ebenfalls in den Speicherplatz mit der Nummer 16 und überschreibt damit den Druckauftrag von Prozess P_B. Abschließend erhöht auch Prozess P_A die Zählvariable des Spooler-Verzeichnisses um den Wert 1.

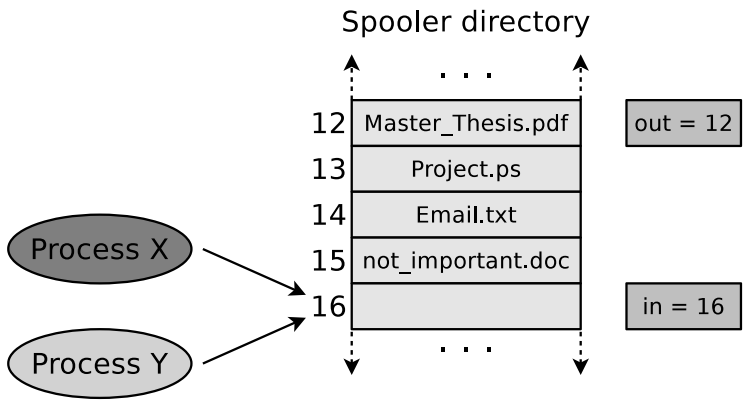

Figure 9.1: Two Processes are in a Race Condition [112]

Table 9.1: Example of a Critical Section and its potential Consequences [112]

Process P_A		*Process P_B*
`next_free_slot = in;` (Result: 16)		
	Process switching	
		`next_free_slot = in;` (Result: 16)
		`Store record in next_free_slot;` (Result: 16)
		`in = next_free_slot+1;` (Result: 17)
	Process switching	
`Store record in next_free_slot;` (Result: 16)		
`in = next_free_slot+1;` (Result: 17)		

As a result, the spooling directory is in a consistent state, but the record of process P_B was overwritten by process P_A and was lost. Such a situation is called a *race condition*. It is an *unintended race condition* of two processes that want to modify the value of the same record. In such a situation, the result of a process depends on the execution sequence or timing of other events.

Race conditions often are the root cause of bugs that are hard to locate and fix because their occurrence and symptoms depend on different events. The symptoms may be different or disappear with each test run. One way to avoid race conditions is by using the blocking concept (see Section 9.2.2) or the *semaphore* concept (see Section 9.4.1).

Unintended race conditions are not trivial and can have tragic consequences depending on the situation. A well-known example of possible results caused by a race condition is the linear particle accelerator Therac-25, which was used for radiation therapy of cancer tumors. This device caused some accidents in the United States in the mid-1980s. Of the patients that were killed, two died because of a race condition, which resulted in inconsistent settings of the device, causing an increased radiation dose. The

Im Ergebnis ist das Spooler-Verzeichnis zwar in einem konsistenten Zustand, aber der Eintrag von Prozess P_B wurde von Prozess P_A überschrieben und ging verloren. Eine solche Situation heißt *Race Condition*. Sie beschreibt die *unbeabsichtigte Wettlaufsituation* zweier Prozesse, die den Wert der gleichen Speicherstelle ändern wollen. Das Ergebnis eines Prozesses hängt dabei von der Reihenfolge oder dem zeitlichen Ablauf anderer Ereignisse ab.

Race Conditions sind ein häufiger Grund für schwer auffindbare Programmfehler, denn das Auftreten und die Symptome hängen von unterschiedlichen Ereignissen ab. Bei jedem Testdurchlauf können die Symptome unterschiedlich sein oder verschwinden. Eine Vermeidung ist unter anderem durch das Konzept der Sperren (siehe Abschnitt 9.2.2) oder das Konzept der *Semaphore* (siehe Abschnitt 9.4.1) möglich.

Unbeabsichtigte Wettlaufsituationen sind keine Petitesse und können je nach Anwendungsfall durchaus tragische Auswirkungen nach sich ziehen. Ein bekanntes Beispiel für tragische Auswirkungen durch eine Race Condition ist der Elektronen-Linearbeschleuniger Therac-25, der zur Strahlentherapie von Krebstumoren eingesetzt wurde. Dieser verursachte Mitte der 1980er Jahre in den vereinigten Staaten Unfälle. Von den getöteten Patienten starben zwei nachweislich durch eine Race Condition, die zu inkonsis-

control process did not synchronize correctly with the user interface process. The bug only occurred when the operator operated the device too fast. During testing, the bug did not occur because it required experience (routine) to operate the device this fast [64].

tenten Einstellungen des Gerätes und damit zu einer erhöhten Strahlendosis führte. Ursächlich war eine fehlerhafte Synchronisierung mit dem Prozess der Eingabeaufforderung. Der Fehler trat allerdings nur dann auf, wenn die Bedienung zu schnell erfolgte. Bei Tests trat der Fehler nicht auf, weil es Erfahrung (Routine) erforderte, um das Gerät so schnell zu bedienen [64].

9.2

Synchronization of Processes

Interprocess communication has a functional and a temporal aspect. From a functional point of view, interprocess communication enables *communication* and *cooperation* between processes (see Figure 9.2).

Synchronisation von Prozessen

Bei der Prozessinteraktion unterscheidet man den funktionalen und den zeitlichen Aspekt. Aus funktionaler Sicht ermöglicht die Prozessinteraktion die *Kommunikation* und die *Kooperation* zwischen Prozessen (siehe Abbildung 9.2).

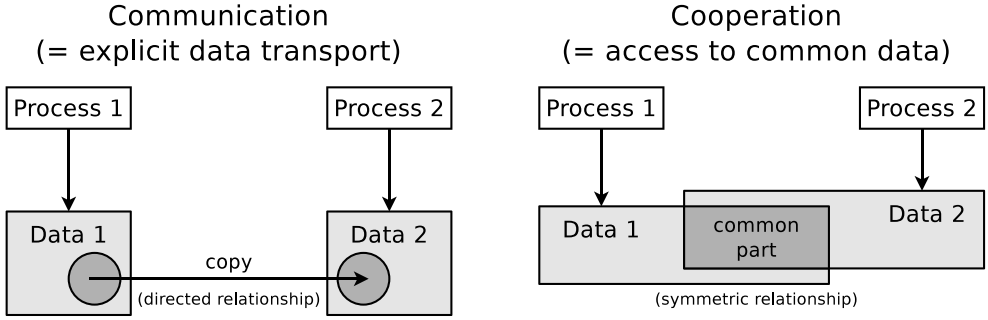

Figure 9.2: Interprocess Communication enables Communication and Cooperation between Processes [46]

Furthermore, there is a temporal aspect of interprocess communication, which is *synchronization*. Communication and cooperation both are based on synchronization because synchronization between the interacting partners is mandatory for obtaining correct results. Synchronization is, thus, the most elementary form of interaction (see Figure 9.3). For this reason, this chapter first describes the synchronization of processes, and then different forms of communication and cooperation.

Zudem gibt es bei der Prozessinteraktion auch einen zeitlichen Aspekt, nämlich die *Synchronisation*. Kommunikation und Kooperation basieren beide auf der Synchronisation, denn sie benötigen eine zeitliche Abstimmung zwischen den Interaktionspartnern, um korrekte Ergebnisse zu erhalten. Synchronisation ist somit die elementarste Form der Interaktion (siehe Abbildung 9.3). Aus diesem Grund beschreibt dieses Kapitel zuerst die Synchronisation von Prozessen und anschließend unterschiedliche Formen der Kommunikation und der Kooperation.

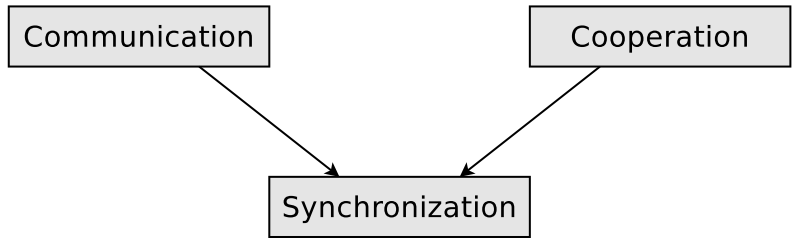

Figure 9.3: Synchronization is the most elementary Form of Interaction between Processes [46]

9.2.1

Specification of the Execution Order of Processes

One way to specify the order in which the processes are run, and to synchronize them accordingly, is by using signals. One example of signaling is shown in Figure 9.4. In this scenario, Section X of process P_A has to be executed before Section Y of process P_B. The `signal` operation indicates when process P_A has finished Section X. As a consequence, process P_B may have to wait for the signal of process P_A.

Definition der Ausführungsreihenfolge durch Signalisierung

Eine Möglichkeit, die Ausführungsreihenfolge der Prozesse zu definieren und somit die Prozesse zu synchronisieren, ist die Verwendung von Signalen. Ein Beispiel für Signalisierung zeigt Abbildung 9.4. Bei diesem Szenario soll Abschnitt X von Prozess P_A vor Abschnitt Y von Prozess P_B ausgeführt werden. Die Operation `signal` signalisiert, wenn Prozess P_A den Abschnitt X abgearbeitet hat. Als Konsequenz muss Prozess P_B eventuell auf das Signal von Prozess P_A warten.

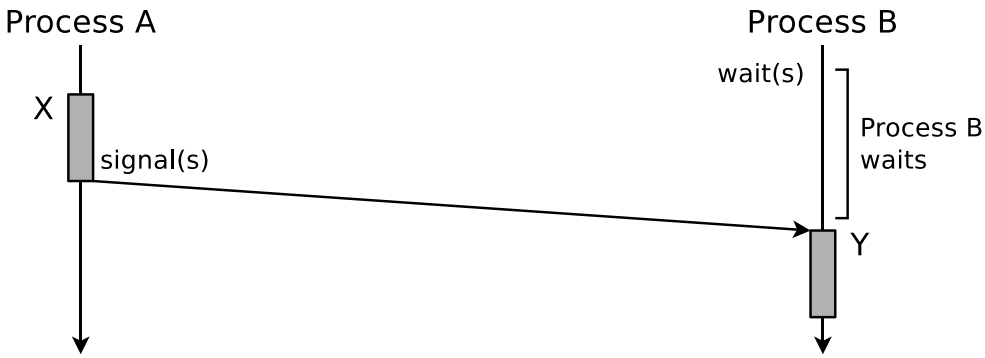

Figure 9.4: Signalization is used to specify an Execution Order

The most simple form of signaling is the *busy waiting* method, which in literature is also called *polling* [120]. Figure 9.5 shows busy waiting at the global signal variable `s` [108]. The variable can be stored in a local file for the sake of simplicity. The process whose critical section shall be executed first, sets the signal variable with the `signal` operation after it has finished its

Die einfachste Form der Realisierung ist das *aktive Warten* (englisch: *Busy Waiting*), das in der Literatur auch *Polling* heißt [120]. Abbildung 9.5 zeigt aktives Warten an der globalen Signalvariable `s` [107]. Diese kann sich der Einfachheit halber in einer lokalen Datei befinden. Der Prozess, dessen kritischer Abschnitt zuerst ausgeführt werden soll, setzt die Signalvariable

critical section. The other process periodically checks whether the signal variable is set. If the signal variable is set, then the `wait` operation resets it.

The names `signal` and `wait` for the required operations are generic identifiers. The operating systems provide utilities, such as system calls and library functions, to implement the functionalities that are described in this section.

A drawback of busy waiting is that it wastes CPU resources because the `wait` operation occupies the CPU at regular intervals. This technique is also called *spinlock* [39].

mit der `signal`-Operation, sobald er seinen kritischen Abschnitt fertig abgearbeitet hat. Der andere Prozess prüft in regelmäßigen Abständen, ob die Signalvariable gesetzt ist. Ist die Signalvariable gesetzt, setzt die `wait`-Operation diese zurück.

Die Namen `signal` und `wait` für die benötigen Operationen sind generische Bezeichner. Die existierenden Betriebssysteme stellen Werkzeuge wie Systemaufrufe und Bibliotheksfunktionen zur Verfügung, um die in diesem Abschnitt beschrieben Funktionalitäten nachzubilden.

Ein Nachteil des aktiven Wartens ist, dass Rechenzeit verschwendet wird, weil die `wait`-Operation den Prozessor in regelmäßigen Abständen belegt. Diese Technik heißt auch *Warteschleife* [39].

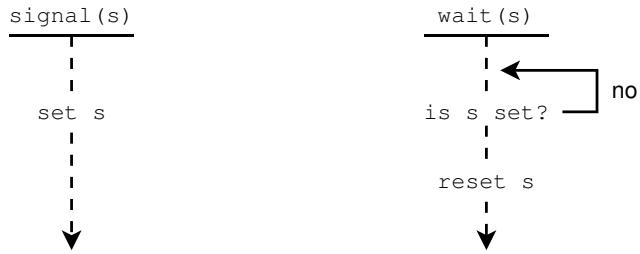

Figure 9.5: Working Principle of Busy Waiting (Spinlock)

A better concept for synchronizing processes that does not waste CPU resources is shown in Figure 9.6. In the literature, it is often called *passive waiting* [120]. The `signal` operation sets the signal variable `s` and checks if there is a process waiting. If this is the case, the `signal` operation unblocks it. The `wait` operation checks whether the signal variable is set. Is this the case, the `wait` operation resets the signal variable. If the signal variable is not set, the `wait` operation blocks the process.

One way to specify an execution order with passive waiting in Linux and other UNIX-like operating systems is by using the function `sigsuspend`. With this function, a process blocks itself until another process sends an appropriate signal to it (usually `SIGUSR1` or `SIGUSR2`) with the command `kill` and thus signals that it shall continue working.

Ein besseres Konzept zur Synchronisation von Prozessen, das keine Rechenzeit des Prozessors vergeudet, zeigt Abbildung 9.6. In der Literatur heißt es häufig *passives Warten* [120]. Die `signal`-Operation setzt die Signalvariable `s` und überprüft, ob es einen wartenden Prozess gibt. Ist das der Fall, deblockiert die `signal`-Operation diesen. Die `wait`-Operation prüft, ob die Signalvariable gesetzt ist. Ist das Fall, setzt die `wait`-Operation die Signalvariable zurück. Ist die Signalvariable nicht gesetzt, blockiert die `wait`-Operation den Prozess.

Eine Möglichkeit, um mit den Mitteln von Linux und anderen Unix-ähnlichen Betriebssystem eine Ausführungsreihenfolge mit passivem Warten festzulegen, ist die Funktion `sigsuspend`. Mit dieser blockiert sich ein Prozess so lange selbst, bis ein anderer Prozess ihm mit dem Kommando `kill` ein passendes Signal (meist `SIGUSR1` oder `SIGUSR2`) sendet und somit signalisiert, dass er weiterarbeiten soll.

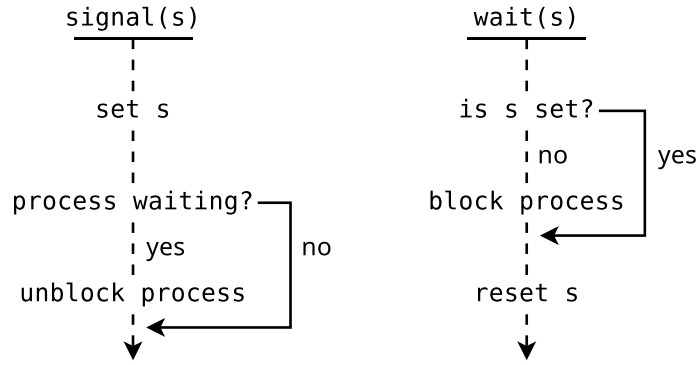

Figure 9.6: Passive Waiting does not waste CPU Resources

With the system call `kill`, and the library function of the same name, as well as the command of the same name in the shell, it is possible to send signals to processes.

Alternative system calls and function calls of the same name, by which a process can block itself until it is woken up again by a system call, are `pause` and `sleep` [48].

An in-depth discussion of the options for controlling the execution order of processes by signals is given in [47, 48, 50].

Mit dem Systemaufruf `kill` und der gleichnamigen Bibliotheksfunktion sowie dem gleichnamigen Kommando in der Shell ist es möglich, Signale an Prozesse zu senden.

Alternative Systemaufrufe und gleichnamige Funktionsaufrufe, mit denen sich ein Prozess selbst so lange blockieren kann, bis er durch einen Systemaufruf wieder geweckt wird, sind `pause` und `sleep` [48].

Eine intensive Auseinandersetzung mit den Möglichkeiten, die Ausführungsreihenfolge von Prozessen durch Signale zu beeinflussen, bieten [47, 48, 50].

9.2.2

Protecting Critical Sections by Blocking

Signaling always specifies an execution order. However, if it is only necessary to ensure that there is no overlap in the execution of the critical sections, the two atomic operations `lock` and `unlock` can be used, as shown in Figure 9.7. The operation `lock` blocks the other process, and `unlock` indicates that the other process may enter its critical section.

The synchronization concept of Figure 9.7 shows how to protect critical sections without having to specify an execution order. It is called *blocking* in literature. The working method of the operations `lock` and `unlock` for locking and unlocking processes is shown in Figure 9.8.

Schutz kritischer Abschnitte durch Sperren

Beim Signalisieren wird immer eine Ausführungsreihenfolge festlegt. Soll aber einfach nur sichergestellt werden, dass es keine Überlappung in der Ausführung der kritischen Abschnitte gibt, können die beiden atomaren Operationen `lock` und `unlock` wie in Abbildung 9.7 eingesetzt werden. Die Operation `lock` blockiert hier den anderen Prozess und `unlock` teilt ihm mit, dass er in seinen kritischen Abschnitt eintreten darf.

Das in Abbildung 9.7 gezeigte Synchronisationskonzept zum Schutz kritischer Abschnitte ohne Definition einer Ausführungsreihenfolge heißt in der Literatur auch *Sperren* [122]. Die Arbeitsweise der beiden Operationen `lock` und

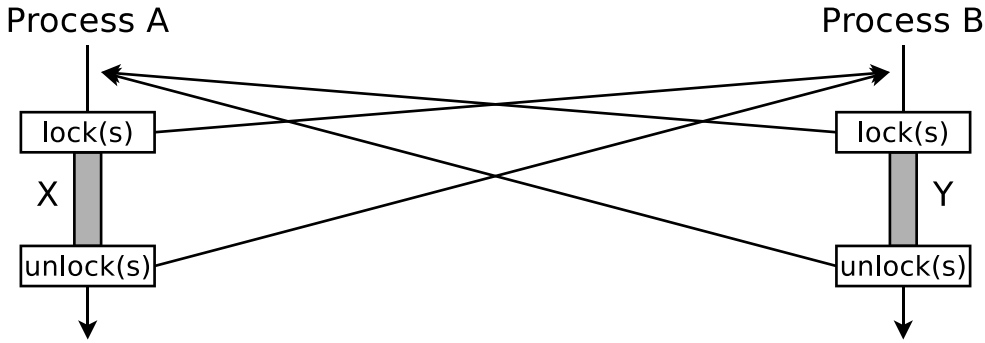

Figure 9.7: Locking secures critical Sections [122]

`unlock` zum Sperren und Freigeben von Prozessen zeigt Abbildung 9.8.

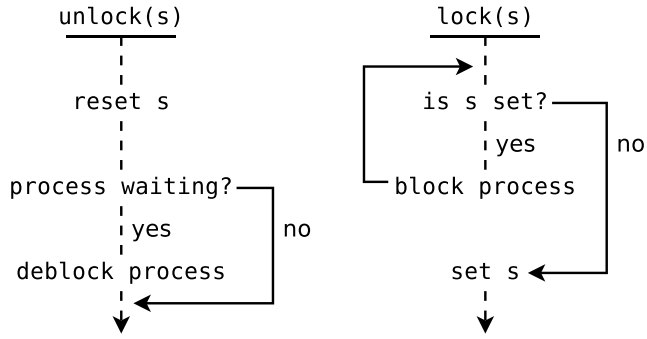

Figure 9.8: Blocking (Locking) prevents the overlapping Execution of two critical Sections

The operations `lock` and `unlock` for locking and unlocking processes can also be implemented in Linux and other Unix-like operating systems with system calls and library functions such as `sigsuspend`, `kill`, `pause`, and `sleep`.

An even more straightforward way to implement locking is using the signals `SIGSTOP` (signal number 19) and `SIGCONT` (signal number 18). With the first signal, another process in the system can be stopped, and with the second signal, it can be reactivated.

If the locking must not be implemented with signals, a local file can also serve as a locking mechanism for mutual exclusion. In this concept, each process must verify before entering its critical section whether it can open

Auch die beiden Operationen `lock` und `unlock` zum Sperren und Freigeben von Prozessen können unter Linux und anderen Unix-ähnlichen Betriebssystem mit Systemaufrufen und Bibliotheksfunktion wie `sigsuspend`, `kill`, `pause` und `sleep` realisiert werden.

Eine noch einfachere Möglichkeit zur Realisierung von Sperren ist die Verwendung der beiden Signale `SIGSTOP` (Signalnummer 19) und `SIGCONT` (Signalnummer 18). Mit dem ersten Signal ist es möglich, einen anderen Prozess im System zu stoppen und mit dem zweiten Signal kann er reaktiviert werden.

Soll das Sperren nicht mit Signalen realisiert werden, kann auch eine lokale Datei als Sperrmechanismus für wechselseitigen Ausschluss dienen. Dafür muss nur jeder Prozess vor dem Eintritt in seinen kritischen Abschnitt prüfen, ob er die

the file exclusively with the system call `open` or with a standard library function like `fopen`. If this is not the case, it must pause for a certain time (e.g., with the system call `sleep`) and then try again (*busy waiting*), or with `sleep` or `pause` and hope that the process that has already opened the file unblocks it with a signal at the end of its critical section (*passive waiting*). An in-depth discussion of this concept of process synchronization is provided by [35, 47, 119].

Datei mit dem Systemaufruf `open`, oder alternativ mit einer Standard-Bibliotheksfunktion wie `fopen`, exklusiv öffnen kann. Wenn das nicht der Fall ist, muss er für eine bestimmte Zeit pausieren (z.B. mit dem Systemaufruf `sleep`) und es danach erneut versuchen (*aktives Warten*) oder alternativ sich mit `sleep` oder `pause` selbst pausieren und hoffen, dass der Prozess, der bereits die Datei geöffnet hat, ihn nach Abschluss seines kritischen Abschnitts mit einem Signal deblockiert (*passives Warten*). Eine intensive Auseinandersetzung mit diesem Konzept zu Prozesssynchronisation bieten [35, 47, 119].

9.2.3

Starvation and Deadlock

When defining an execution order, or when blocking processes, *starvation* or a *deadlock* may occur. Starvation is a situation in which a process does not remove a lock, and therefore one or more processes are waiting endlessly for the release. If two or more processes wait for resources that are locked mutually, then this is called deadlock. Because all processes that are involved in the deadlock have to wait infinitely long, no one can initiate an event that resolves the situation. According to [25], the following conditions must be fulfilled simultaneously for a deadlock to occur [112]:

- *Mutual Exclusion.* At least one resource is occupied by exactly one process, or is available. It means that the resource cannot be shared.
- *Hold and wait.* A process that already occupies at least one resource requests additional resources that are already occupied by another process.
- *No preemption.* The operating system can not deallocate resources which a process occupies, only the process can release them by itself.

Verhungern und Deadlock

Bei der Definition einer Ausführungsreihenfolge oder dem Blockieren von Prozessen kann es leicht zum *Verhungern* (englisch: *Starvation*) oder zu einer *Verklemmung* (englisch: *Deadlock*) kommen. Das Verhungern beschreibt eine Situation, in der ein Prozess eine Sperre nicht wieder aufhebt und darum ein oder mehr Prozesse endlos auf die Freigabe warten. Beim Deadlock warten zwei oder mehr Prozesse auf die von ihnen gesperrten Ressourcen. Sie sperren sich somit gegenseitig. Da alle am Deadlock beteiligten Prozesse endlos lange warten, kann keiner ein Ereignis auslösen, das die Situation auflöst. Damit ein Deadlock entstehen kann, müssen laut [25] folgende Bedingungen gleichzeitig erfüllt sein [115]:

- *Wechselseitiger Ausschluss.* Mindestens eine Ressource wird von genau einem Prozess belegt oder ist verfügbar. Sie ist also nicht gemeinsam nutzbar.
- *Anforderung weiterer Betriebsmittel.* Ein Prozess, der bereits mindestens eine Ressource belegt, fordert weitere Ressourcen an, die von einem anderen Prozess bereits belegt sind.
- *Ununterbrechbarkeit.* Die Ressourcen, die ein Prozess besitzt, können nicht vom Betriebssystem entzogen, sondern nur durch ihn selbst freigegeben werden.

- *Circular wait*. A cyclic chain of processes exists. Each process requests a resource that the next process in the chain occupies.

If one of these conditions is not fulfilled, no deadlock can occur.

Various methods for deadlock detection exist. Two of them are introduced in the next two sections. Such methods generally have the drawback that they cause overhead. In principle, deadlocks can occur in all popular operating systems and are often tolerated, depending on the importance of the computer system. If, for example, the maximum number of inodes in a file system is allocated, no new files can be created. If the process table is full, no more new processes can be created [17, 112]. If main memory and swap memory are full, the operating system becomes unusable [39]. The probability that such things happen in daily life is low, but it is also not impossible. Such potential deadlocks are more likely to be accepted in the vast majority of applications than the overhead that is required to detect and avoid them. In this context, literature knows the technical term *ostrich algorithm*, which recommends ignoring the potential deadlock [39, 112].

Resource Graphs

Resource allocation graphs are one way to visualize the relations between processes and resources, using directed graphs, and to model potential deadlocks when the sequence of resource requests from the processes is known in advance. Processes are represented as circles in the graph and resources as rectangles. An edge from a process to a resource means that the process is blocked because it waits for the resource. An edge from a resource to a process means that the process occupies the resource [17].

Figure 9.9 shows the components and relations in the resource graph and a circular wait situation. The two processes wait for each other.

- *Zyklische Wartebedingung*. Es gibt eine zyklische Kette von Prozessen. Jeder Prozess fordert eine Ressource an, die der nächste Prozess in der Kette besitzt.

Fehlt eine der genannten Bedingungen, kann kein Deadlock entstehen.

Es gibt verschiedene Verfahren zur Deadlock-Erkennung. Zwei davon stellen die folgenden beiden Abschnitte vor. Solche Verfahren haben generell den Nachteil, dass Sie einen Verwaltungsaufwand erzeugen. In allen bekannten Betriebssystemen sind Deadlocks prinzipiell möglich und häufig werden sie abhängig von der Wichtigkeit des Computersystems akzeptiert. Sind beispielsweise alle möglichen Inodes im Dateisystem vergeben, können keine neuen Dateien mehr angelegt werden. Ist die Prozesstabelle voll, können keine neuen Prozesse mehr erzeugt werden [17, 115]. Sind der Hauptspeicher und der Auslagerungsspeicher voll, wird das Betriebssystemen unbenutzbar [39]. Die Wahrscheinlichkeit, dass solche Dinge im Alltag passieren, ist gering, aber es ist auch nicht unmöglich. Solche potentiellen Deadlocks werden in den allermeisten Anwendungsfällen eher akzeptiert als der zur Erkennung und Vermeidung nötige Verwaltungsaufwand. In diesem Zusammenhang spricht man auch von einer Politik der *Deadlock-Ignorierung* oder *Vogel-Strauß-Politik* (englisch: *Ostrich Algorithm*) [39, 115].

Betriebsmittel-Graphen

Betriebsmittel-Graphen (englisch: *Resource Allocation Graphs*) sind eine Möglichkeit, um mit gerichteten Graphen die Beziehungen von Prozessen und Ressourcen darzustellen und potentielle Deadlocks zu modellieren, wenn der zeitliche Ablauf der Ressourcenanforderungen durch die Prozesse im Voraus bekannt ist. Prozesse sind im Graph als Kreise und Ressourcen als Rechtecke dargestellt. Eine Kante von einem Prozess zu einer Ressource bedeutet, dass der Prozess blockiert ist, weil er auf die Ressource wartet. Eine Kante von einer Ressource zu einem Prozess bedeutet, dass der Prozess die Ressource belegt [17].

Abbildung 9.9 zeigt die Komponenten und Beziehungen im Betriebsmittel-Graph, außerdem wird ein Zyklus dargestellt. Die beiden Prozesse warten aufeinander. Diese Deadlock-Situation

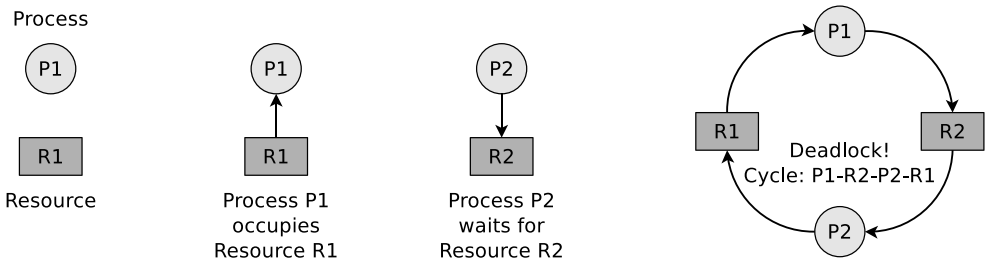

Figure 9.9: Resource Allocation Graphs for visualizing Deadlocks

This deadlock situation can only be resolved by aborting one of the processes involved [6].

Deadlock Detection with Petri Nets

Petri nets allow the graphical modeling of concurrent systems and are another way of visualizing deadlock situations. For reasons of limited space, and because of the similarity to resource graphs, this book contains no further information about petri nets. An in-depth discussion of this topic is provided by [69].

Deadlock Detection with Matrices

A drawback of deadlock detection with resource allocation graphs is that only individual resources can be displayed with it. If multiple copies (instances) of a resource exist, then graphs are not suitable for deadlock visualization or detection. If multiple instances of a resource exist, a matrix-based method can be used, which requires two vectors and two matrices and is described in [112].

The two vectors are the *existing resource vector*, which indicates how many instances of each resource class exist, and the *available resource vector*, which indicates how many instances of each resource class are not yet occupied by processes.

The two matrices are the *current allocation matrix*, which indicates how many instances of the individual resource classes each process occupies, and the *request matrix*, which indicates how many resources the individual processes want to occupy.

kann nur durch den Abbruch eines der beteiligten Prozesse aufgelöst werden [6].

Deadlock-Erkennung mit Petrinetzen

Petrinetze ermöglichen die grafische Modellierung nebenläufiger Systeme und sind eine weitere Möglichkeit der Darstellung von Deadlock-Situationen. Aus Platzgründen und wegen der großen Ähnlichkeit zu Betriebsmittel-Graphen enthält dieses Buch keine weiteren Informationen zu Petrinetzen. Eine intensive Auseinandersetzung mit diesem Thema bietet [69].

Deadlock-Erkennung mit Matrizen

Ein Nachteil der Deadlock-Erkennung mit Betriebsmittel-Graphen ist, dass man damit nur einzelne Ressourcen darstellen kann. Gibt es mehrere Kopien (Instanzen) einer Ressource, sind Graphen zur Darstellung bzw. Erkennung von Deadlocks ungeeignet. Existieren von einer Ressource mehrere Instanzen, kann ein in [115] beschriebenes, matrizenbasiertes Verfahren verwendet werden, das zwei Vektoren und zwei Matrizen benötigt.

Die beiden Vektoren sind der *Ressourcenvektor* (englisch: *Existing Resource Vector*), der anzeigt, wie viele Instanzen von jeder Ressourcenklasse existieren und der *Ressourcenrestvektor* (englisch: *Available Resource Vector*), der anzeigt, wie viele Instanzen von jeder Ressourcenklasse noch nicht durch Prozesse belegt sind.

Die beiden Matrizen sind die *Belegungsmatrix* (englisch: *Current Allocation Matrix*), die anzeigt, wie viele Instanzen der einzelnen Ressourcenklassen jeder Prozess belegt und die *Anforderungsmatrix* (englisch: *Request Matrix*),

The values of the following example for deadlock detection with matrices are taken from [112]. Similar examples are provided by [17] and [108]. In the following scenario, the existing resource vector, the current allocation matrix, and the request matrix are given.

$$\text{Existing resource vector} = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

$$\text{Current allocation matrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

$$\text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

In the existing resource vector, each column represents a resource class. Thus, the resource vector indicates that four instances of resource class 1 exist, two instances of class 2, three instances of class 3, and one instance of class 4.

Also, in the current allocation matrix, the columns represent the resource classes, and each row represents a process. The current allocation matrix specifies that process 1 occupies one instance of class 3, process 2 occupies two instances of class 1 and one instance of class 4, and process 3 occupies one instance of class 2 and two instances of class 3.

The request matrix indicates how many instances of each resource class (represented again by the columns) each process (represented again by the rows) still requests, so that it can be processed completely, and release all its occupied resources afterward.

By using the existing resource vector and the current allocation matrix, it is possible to calculate the available resource vector. For this purpose, the records in the existing resource vector are subtracted from the summed columns in the current allocation matrix. For this example, the following available resource vector is calculated:

die anzeigt, wie viele Ressourcen die einzelnen Prozesse noch anfordern.

Die Werte des folgenden Beispiels zur Deadlock-Erkennung mit Matrizen sind aus [115] entnommen. Ähnliche Beispiele enthalten [17] und [107]. Gegeben sind im Szenario der Ressourcenvektor, die Belegungsmatrix und die Anforderungsmatrix.

$$\text{Ressourcenvektor} = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

$$\text{Belegungsmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

$$\text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Im Ressourcenvektor ist jede Spalte stellvertretend für eine Ressourcenklasse. Dementsprechend sagt der Ressourcenvektor aus, dass vier Instanzen der Ressourcenklasse 1 existieren, zwei Instanzen der Klasse 2, drei Instanzen der Klasse 3 und eine Instanz der Klasse 4.

Auch in der Belegungsmatrix beschreiben die Spalten die Ressourcenklassen und jede Zeile steht für einen Prozess. Die Belegungsmatrix gibt somit an, dass Prozess 1 eine Instanz von Klasse 3 belegt, Prozess 2 belegt zwei Instanzen vom Klasse 1 und eine Instanz vom Klasse 4 und Prozess 3 belegt eine Instanz von Klasse 2 und zwei Instanzen vom Klasse 3.

Die Anforderungsmatrix gibt an, wie viele Instanzen jeder Ressourcenklasse (erneut dargestellt durch die Spalten) jeder Prozess (erneut dargestellt durch die Zeilen) noch anfordert, damit er komplett abgearbeitet wird und anschließend alle seine belegten Ressourcen freigeben kann.

Mit Hilfe des Ressourcenvektors und der Belegungsmatrix ist es möglich, den Ressourcenrestvektor zu berechnen. Dafür werden die Werte im Ressourcenvektor von den summierten Spalten in der Belegungsmatrix subtrahiert. Für unser Beispiel ergibt sich der folgende Ressourcenrestvektor:

$$\text{Available resource vector} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

This means that only two instances of class 1 and one instance of class 2 are available. The resource classes 3 and 4 are fully occupied. A comparison of the available resource vector with the individual rows of the request matrix shows which processes can continue to run at this point because the operating system can fulfill their resource requests.

Process 1 is blocked because no instance of class 4 is available, and process 2 is blocked because no instance of class 3 is available. Only process 3 is not blocked. It is the only process that can continue to run. As soon as it has finished execution, the operating system frees its resources. This modifies the available resource vector and the request matrix as follows:

$$\text{Available resource vector} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

$$\text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

According to the new available resource vector, two instances of the resource classes 1, 2, and 3 are now available. Processes still occupy all resources of class 4. Therefore, no instances of class 4 are available now. A comparison of the newly available resource vector with the individual rows of the request matrix shows that process 1 is still blocked because no resource of class 4 is available. However, process 2 can continue to run, and as soon as it has finished execution, the operating system also frees its resources. This modifies the available resource vector and the request matrix as follows:

$$\text{Available resource vector} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Konkret bedeutet das, dass nur zwei Instanzen von Klasse 1 und noch eine Instanz von Klasse 2 frei sind. Die Ressourcenklassen 3 und 4 sind vollständig belegt. Ein Vergleich des Ressourcenrestvektors mit den einzelnen Zeilen der Anforderungsmatrix zeigt, welche Prozesse zum aktuellen Zeitpunkt weiterlaufen können, weil das Betriebssystem ihre Ressourcenanforderungen bedienen kann.

Prozess 1 ist blockiert, weil keine Instanz vom Klasse 4 frei ist und Prozess 2 ist blockiert, weil keine Instanz vom Klasse 3 frei ist. Nur Prozess 3 ist nicht blockiert. Er kann als einziger Prozess weiterlaufen. Sobald er fertig ausgeführt ist, gibt das Betriebssystem seine Ressourcen frei. Dadurch ändern sich der Ressourcenrestvektor und die Anforderungsmatrix wie folgt:

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

$$\text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ - & - & - & - \end{bmatrix}$$

Nach dem neuen Ressourcenrestvektor sind nun jeweils zwei Instanzen der Ressourcenklassen 1, 2 und 3 frei. Alle Ressourcen von Klasse 4 sind nach wie vor von Prozessen belegt. Entsprechend sind zum aktuellen Zeitpunkt keine Instanzen der Klasse 4 verfügbar. Ein Vergleich des neuen Ressourcenrestvektors mit den einzelnen Zeilen der Anforderungsmatrix zeigt, dass Prozess 1 immer noch blockiert ist, weil keine Ressource von Klasse 4 frei ist. Prozess 2 hingegen kann weiterlaufen und sobald er fertig ausgeführt ist, gibt das Betriebssystem auch seine Ressourcen frei. Dadurch ändern sich der Ressourcenrestvektor und die Anforderungsmatrix wie folgt:

$$\text{Ressourcenrestvektor} = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

$$\text{Request matrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix} \qquad \text{Anforderungsmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ - & - & - & - \\ - & - & - & - \end{bmatrix}$$

From this point in time on, process 1 is also no longer blocked, since the operating system can now fulfill its resource requests (two instances of class 1 and one instance of class 4). No deadlock occurs in the presented scenario.

Ab diesem Zeitpunkt ist auch Prozess 1 nicht mehr blockiert, da das Betriebssystem nun seine Ressourcenanforderungen (zwei Instanzen der Klasse 1 und eine Instanz der Klasse 4) bedienen kann. Damit ist klar, dass es im vorgestellten Szenario nicht zum Deadlock kommt.

9.3

Communication of Processes

Kommunikation von Prozessen

Operating systems enable *interprocess communication* via shared memory, message queues, pipes, and sockets. The following sections describe these four communication concepts.

Betriebssysteme ermöglichen den Prozessen die *Interprozesskommunikation* durch die Nutzung von gemeinsamem Speicher, Nachrichtenwarteschlangen, Pipes und Sockets. Die folgenden Abschnitte beschreiben diese vier Kommunikationskonzepte.

9.3.1

Shared Memory (System V)

Gemeinsamer Speicher (System V)

Interprocess communication via a *shared memory* is memory-based communication. The shared memory segments used for this purpose are memory areas, which can be accessed directly by multiple processes (see Figure 9.10). Processes need to coordinate access operations by themselves and ensure that their memory requests are mutually exclusive. A receiver process cannot read anything from shared memory until the sender process has finished writing. If access operations are not coordinated carefully, inconsistencies occur.

The shared memory concept described in this section follows the *System V* standard. Apart from that, there are also the shared memory segments according to the standard *POSIX* – Portable Operating System Interface (see Section 9.3.2).

The Linux kernel maintains a *shared memory table* with information about the existing shared

Interprozesskommunikation über einen *gemeinsamen Speicher* (englisch: *Shared Memory*) ist speicherbasierte Kommunikation. Die dabei verwendeten gemeinsamen Speichersegmente sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können (siehe Abbildung 9.10). Die Prozesse müssen die Zugriffe selbst koordinieren und sicherstellen, dass ihre Speicherzugriffe sich gegenseitig ausschließen. Ein lesender Prozess darf nichts aus dem gemeinsamen Speicher lesen, bevor der schreibende Prozess fertig geschrieben hat. Ist die Koordinierung der Zugriffe nicht sorgfältig, kommt es zu Inkonsistenzen.

Das in diesem Abschnitt beschriebene Konzept zur Realisierung und Nutzung gemeinsamer Speicherbereiche entspricht dem Standard *System V*. Daneben existieren auch die gemeinsamen Speichersegmente gemäß dem Standard *POSIX* – Portable Operating System Interface (siehe Abschnitt 9.3.2).

Der Betriebssystemkern von Linux speichert zur Verwaltung der gemeinsamen Speicherbe-

Figure 9.10: Shared Memory Segments are Memory Areas which multiple Processes can access directly

memory segments to manage them. This information includes the start address in memory, size, owner (username and group), and privileges (see Figure 9.11). A shared memory segment is always addressed by its index number in the shared memory table.

reiche eine *Shared Memory-Tabelle* mit Informationen über die existierenden Segmente. Zu diesen Informationen gehören: Anfangsadresse im Speicher, Größe, Besitzer (Benutzername und Gruppe) und Zugriffsrechte (siehe Abbildung 9.11). Ein gemeinsames Speichersegment wird immer über seine Indexnummer in der Shared Memory-Tabelle angesprochen.

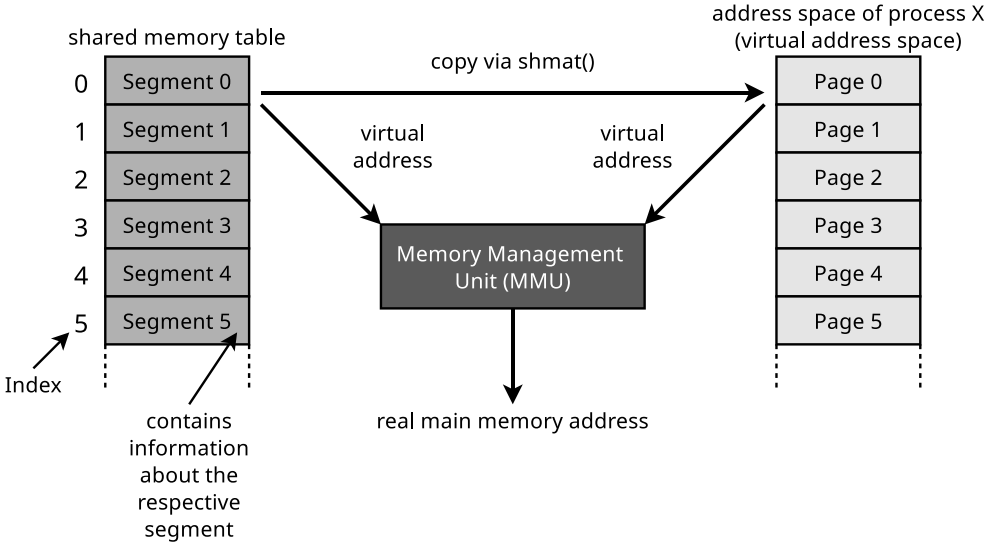

Figure 9.11: The Kernel manages the Information about the Shared Memory Segments in the Shared Memory Table

If a shared memory segment is no longer attached to a process, it is not automatically deleted by the operating system. It is preserved until the removal is instructed by the system call `shmctl`, or until the operating system is rebooted. Linux provides four system calls for working with shared memory (see Table 9.2).

The program example in Listing 9.1 shows in the programming language C the creation of a

Ist ein gemeinsames Speichersegment an keinen Prozess mehr gebunden, wird es nicht automatisch vom Betriebssystem gelöscht, sondern bleibt erhalten, bis die Löschung durch den Systemaufruf `shmctl` angewiesen wird oder bis zum Neustart des Betriebssystems. Linux stellt vier Systemaufrufe für die Arbeit mit gemeinsamem Speicher bereit (siehe Tabelle 9.2).

Das Programmbeispiel in Listing 9.1 zeigt in der Programmiersprache C die Erzeugung eines

Table 9.2: Linux System Calls for working with System V Shared Memory Segments

System call	Purpose
`shmget`	Create a shared memory segment or access an existing one
`shmat`	Attach a shared memory segment to a process
`shmdt`	Detach/release a shared memory segment from a process
`shmctl`	Request the status (including permissions) of a segment, modify or erase it

System V shared memory segment in Linux with the function `shmget` (see line 18), appending the virtual address space of the calling process with `shmat` (see line 27), writing a string into the segment (see line 36), reading this string from the segment (see line 45), detaching the segment from the virtual address space of the calling process with `shmdt` (see line 51), and finally erasing (deleting) the segment from the operating system with `shmctl` (see line 60).

The example is quite unrealistic because the same process does both: the writing and the reading part. However, it demonstrates the implementation of the operations mentioned above in a compact way. Also, it is simple to relocate parts of the source code to a second process. It would make sense for a second process to gain access to an existing segment and have read or write access to it.

If calling the function `shmget` in line 18 has been successful, the return value is the shared memory ID, which is unique in the operating system. If the return value of the function is -1, the kernel was unable to create the segment. A shared memory key for the new segment is specified in line 9. The constant `MAXMEMSIZE` in line 6 specifies the memory capacity of the segment. In this case, it is 20 bytes. The parameter `IPC_CREAT` in line 18 indicates that a possibly existing segment with the same shared memory key must not be overwritten, but only its shared memory ID must be returned. The parameter `0600` in the same line specifies the privileges. In this example, only the user who creates the segment has read and write access to it.

For trying to append the segment to the virtual address space of the calling process, with the function `shmat` in line 27, the vari-

gemeinsamen System V-Speichersegments unter Linux mit der Funktion `shmget` (in Zeile 18), das Anhängen an den virtuellen Adressraum des aufrufenden Prozess mit `shmat` (in Zeile 27), das Schreiben einer Zeichenkette in das Segment (in Zeile 36), das Auslesen dieser Zeichenkette aus dem Segment (in Zeile 45), das Lösen des Segments vom virtuellen Adressraum des aufrufenden Prozess mit `shmdt` (in Zeile 51) und abschließend das Entfernen (Löschen) das Segments aus dem Betriebssystem mit `shmctl` (in Zeile 60).

Das Beispiel geht an der Realität vorbei, weil der schreibende und der lesende Prozess identisch sind. Es zeigt aber auf kompakte Art und Weise die Realisierung der oben genannten Schritte. Zudem ist es einfach, einzelne Codebereiche in einen zweiten Prozess zu verlagern. Sinnvollerweise würde ein zweiter Prozess den Zugriff auf ein existierendes Segment und einen Lese- oder Schreibzugriff darauf erhalten.

War der Aufruf der Funktion `shmget` in Zeile 18 erfolgreich, ist der Rückgabewert die im Betriebssystem eindeutige Shared Memory-ID. Wenn der Rückgabewert der Funktion -1 ist, konnte der Betriebssystemkern das Segment nicht anlegen. Ein Shared Memory-Key für das neue Segment wird in Zeile 9 definiert. Die Konstante `MAXMEMSIZE` in Zeile 6 enthält die Speicherkapazität des Segments, in diesem Fall: 20 Bytes. Der Parameter `IPC_CREAT` in Zeile 18 gibt an, dass ein eventuell existierendes Segment mit dem gleichen Shared Memory-Key nicht überschrieben, sondern nur seine Shared Memory-ID zurückgeliefert werden soll. Der Parameter `0600` in der gleichen Zeile definiert die Zugriffsrechte. In diesem Fall darf nur der Benutzer, der das Segment anlegt, auf dieses lesend und schreibend zugreifen.

Beim Versuch das Segment mit der Funktion `shmat` in Zeile 27 an den virtuellen Adressraum des aufrufenden Prozess anzuhängen, ist als Pa-

able `return_shmget` is specified as parameter. It contains the return value of `shmget` with the unique shared memory ID. If the return value of the function is −1, the kernel was unable to append the segment. Checking the return value of `shmat` looks a bit different from the other functions discussed in this section because the return value is a pointer that must not be compared to a natural number [124].

When writing the string into the segment with the function `sprintf` in line 36, the return value of `return_shmget` is also specified with the variable `return_shmget` as the target of the write operation. Also, the return value of the function indicates whether the write operation was successful because the return value contains the number of written characters. In case of an error, it is < 0.

The read access to the segment with the function `printf` in line 45 is self-explanatory.

Also, when trying to detach the segment from the process using the function `shmdt` in line 51, and when trying to erase the segment with the function `shmctl` and the parameter `IPC_RMID` in line 60, the variable `return_shmget` that contains the shared memory ID is specified as parameter. The return value for these functions is also −1 in the event of an error.

rameter die Variable `return_shmget` angegeben. Diese enthält den Rückgabewert von `shmget` mit der eindeutigen Shared Memory-ID. Wenn der Rückgabewert der Funktion −1 ist, konnte der Betriebssystemkern das Segment nicht anhängen. Die Überprüfung des Rückgabewerts sieht bei `shmat` etwas anders aus als bei den in diesem Abschnitt besprochenen Funktionen, weil der Rückgabewert ein Zeiger ist, der nicht mit einer natürlichen Zahl verglichen werden darf [124].

Beim Schreiben der Zeichenkette in das Segment mit der Funktion `sprintf` in Zeile 36 ist ebenfalls mit der Variable `return_shmget` der Rückgabewert von `shmget` als Ziel der Schreibanweisung angegeben. Auch hier gibt wieder der Rückgabewert der Funktion Auskunft darüber, ob der Schreibzugriff erfolgreich durchgeführt wurde, denn der Rückgabewert enthält die Anzahl der geschriebenen Zeichen und im Fehlerfall ist er < 0.

Der Lesezugriff auf das Segment mit der Funktion `printf` in Zeile 45 ist selbsterklärend.

Auch beim Versuch, das Segment mit der Funktion `shmdt` in Zeile 51 vom Prozess zu lösen und beim Versuch, das Segment mit der Funktion `shmctl` und dem Parameter `IPC_RMID` in Zeile 60 zu löschen, ist als Parameter die Variable `return_shmget` angegeben, die die Shared Memory-ID enthält. Auch bei diesen Funktionen ist der Rückgabewert im Fehlerfall −1.

```
1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <sys/shm.h>
5 #include <stdio.h>
6 #define MAXMEMSIZE 20
7
8 int main(int argc, char **argv) {
9   int key = 12345;
10  int return_shmget;
11  int return_shmdt;
12  int return_sprintf;
13  int return_printf;
14  int return_shmctl;
15  char *shmpointer;
16
17  // Create shared memory segment
18  return_shmget = shmget(key, MAXMEMSIZE, IPC_CREAT | 0600);
19  if (return_shmget < 0) {
20    printf("Unable to create the shared memory segment.\n");
21    exit(1);
```

```
22   }
23
24   printf("The shared memory segment has been created.\n");
25
26   // Attach the shared memory segment
27   shmpointer = shmat(return_shmget, 0, 0);
28   if (shmpointer==(char *)-1) {
29     printf("Unable to attach the shared memory segment.\n");
30     exit(1);
31   } else {
32     printf("The shared memory segment has been attached.\n");
33   }
34
35   // Write a string into the shared memory segment
36   return_sprintf = sprintf(shmpointer, "Hello World.");
37   if (return_sprintf < 0) {
38     printf("The write operation failed.\n");
39     exit(1);
40   } else {
41     printf("%i characters were written.\n", return_sprintf);
42   }
43
44   // Read the string from the shared memory segment
45   if (printf ("Content of the segment: %s\n", shmpointer) < 0) {
46     printf("The read operation failed.\n");
47     exit(1);
48   }
49
50   // Detach the shared memory segment
51   return_shmdt = shmdt(shmpointer);
52   if (return_shmdt < 0) {
53     printf("Unable to detach the shared memory segment.\n");
54     exit(1);
55   } else {
56     printf("The shared memory segment has been detached.\n");
57   }
58
59   // Erase the shared memory segment
60   return_shmctl = shmctl(return_shmget, IPC_RMID, 0);
61   if (return_shmctl == -1) {
62     printf("Unable to erase the shared memory segment.\n");
63     exit(1);
64   } else {
65     printf("The shared memory segment has been erased.\n");
66   }
67
68   exit(0);
69 }
```

Listing 9.1: Program Example for System V Shared Memory Segments

Compiling the program with the GNU C compiler (gcc) and running it should result in the following output:

Das Übersetzen des Programms mit dem GNU C Compiler (gcc) und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_1_shared_memory_systemv.c -o Listing_9_1_shared_memory_systemv
$ ./Listing_9_1_shared_memory_systemv
The shared memory segment has been created.
The shared memory segment has been attached.
12 characters were written.
Content of the segment: Hello World.
The shared memory segment has been detached.
The shared memory segment has been erased.
```

It is interesting to observe the different states the segment that the program creates and then erases is in during runtime. These states can be traced from the Linux command line interface using the ipcs utility, which returns information about existing System V shared memory segments.

Interessant ist die Beobachtung der verschiedenen Zustände, in denen sich das im Programm erstellte und abschließend gelöschte Segment während der Laufzeit befindet. Diese Zustände kann man mit dem Programm ipcs, das Informationen über bestehende gemeinsame System V-Speichersegmente liefert, auf der Kommandozeile von Linux nachvollziehen.

After creating the segment with the shmget function, the permissions and the size, that has been specified in the program example, are visible in the output of ipcs. With the -m option, ipcs is instructed to return only the shared memory areas. The nattch column indicates the number of processes that have already attached to the segment.

Nach der Erzeugung des Segments mit der Funktion shmget sind in der Ausgabe von ipcs die im Programmbeispiel festgelegten Zugriffsrechte und die Größe sichtbar. Mit der Option -m wird ipcs angewiesen, nur die gemeinsamen Speicherbereiche auszugeben. Die Spalte nattch gibt an, wie viele Prozesse das Segment bereits an sich gebunden haben.

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00003039 630718524  bnc        600        20         0
```

The shared memory key is specified in hexadecimal notation. A conversion to the decimal system with the command printf proves that the shared memory key, which was specified in the program example, has actually been used.

Der Shared Memory-Key ist in Hexadezimalschreibweise angegeben. Eine Konvertierung ins Dezimalsystem mit dem Kommando printf zeigt, dass der im Programmbeispiel festgelegte Shared Memory-Key berücksichtigt wurde.

```
$ printf "%d\n" 0x00003039        # Convert from hexadecimal to decimal
12345
```

After appending the segment using the shmat function, the output of ipcs has changed only in the nattch column. Here it becomes evident that up to now, one process has attached the segment to its virtual memory.

Nach dem Anhängen des Segments mit der Funktion shmat verändert sich die Ausgabe von ipcs nur in der Spalte nattch. Hier wird deutlich, dass bislang ein Prozess das Segment an seinen virtuellen Speicher angebunden hat.

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00003039 630718524  bnc        600        20         1
```

Writing into the memory segment, and reading from it, is not indicated in the output of `ipcs`, but detaching the segment from the process with the function `shmdt` (again in the column `nattch`). After erasing the segment with `shmctl`, the output of `ipcs` does not show anything of the segment anymore.

Erasing shared memory segments is also possible using the `ipcrm` command from the command line interface. This is particularly helpful when shared memory segments are created but not removed during the development of software. The command below removes the shared memory segment with the ID `630718524` from the operating system:

```
$ ipcrm shm 630718524
resource(s) deleted
```

Das Schreiben in das Speichersegment und das Lesen daraus schlägt sich nicht in der Ausgabe von `ipcs` nieder – das Lösen des Segments vom Prozess mit der Funktion `shmdt` hingegen schon (erneut in der Spalte `nattch`). Nach dem Löschen des Segments mit `shmctl` ist auch in der Ausgabe von `ipcs` nichts mehr davon zu sehen.

Das Löschen gemeinsamer Speichersegmente ist auch von der Eingabeaufforderung aus mit dem Kommando `ipcrm` möglich. Das ist besonders hilfreich, wenn bei der Entwicklung eigener Software zwar Segmente erzeugt, aber nicht mehr entfernt wurden. Das folgende Kommando entfernt das Segment mit der Shared Memory-ID `630718524` aus dem Betriebssystem:

9.3.2

Shared Memory (POSIX)

An alternative method of implementing shared memory segments is offered by the POSIX standard. The POSIX interface was developed a few years after System V and is therefore considered more modern and easier to understand. However, a drawback can be that in legacy Linux distributions and other Unix-like operating systems, the POSIX interprocess communication is only sometimes completely included or available. However, this drawback has become more and more attenuated in practice since the support of the POSIX interface in Linux and other Unix-like operating systems has been state-of-the-art for several years.

Table 9.3 includes the C function calls for POSIX memory segments specified in the header file `mman.h` Besides these function calls, other functions are part of the POSIX interface for memory segments. Among these are `close` for closing the descriptor of a memory segment and

POSIX-Speichersegmente

Eine alternative Möglichkeit, um gemeinsame Speichersegmente zu realisieren, bietet der Standard POSIX. Die POSIX-Schnittstelle ist einige Jahre nach System V entstanden und gilt dementsprechend als moderner und leichter zugänglich. Ein Nachteil kann in bestimmten Einsatzszenarien allerdings sein, dass besonders in älteren Linux-Distributionen und anderen älteren Unix-ähnlichen Betriebssystemen die POSIX-Interprozesskommunikation nicht immer komplett oder gar nicht verfügbar ist. Dieser Nachteil wird in der Praxis aber immer mehr abgeschwächt, da eine Unterstützung der POSIX-Schnittstelle in Linux und anderen Unix-ähnlichen Betriebssystemen schon mehrere Jahre Stand der Technik ist.

Tabelle 9.3 enthält die in der Header-Datei `mman.h` definierten C-Funktionsaufrufe für POSIX-Speichersegmente. Außer diesen Funktionsaufrufen zählen auch noch weitere Funtionen zur POSIX-Schnittstelle für Speichersegmente. Eine Auswahl sind `close`, um den De-

`ftruncate` for specifying the size of a memory segment.

skriptor eines Speichersegments zu schließen, und `ftruncate`, um die Größe eines Speichersegments zu definieren.

Table 9.3: C Function Calls for POSIX Memory Segment Management

Function call	Purpose
`shm_open`	Create a memory segment or access an existing one
`mmap`	Attach a memory segment to a process
`munmap`	Detach/release a memory segment from a process
`shm_unlink`	Erase a memory segment

The program example in Listing 9.2 shows the life cycle of a POSIX memory segment in Linux in the programming language C. Therefore, it is similar to the program example in Listing 9.1.

POSIX memory segment names must start with a slash (`/`), as shown in the example, and must not contain another slash. In the program example in Listing 9.2, the name of the new memory segment is `/mymemory` (see line 13).

The program creates a memory segment with the function `shm_open` (in line 16). The parameter `O_CREAT` specifies that any existing memory segment of the same name shall not be overwritten, but only its descriptor shall be returned. The parameter `0600` in the same line specifies the access privileges exactly as in the program example in Listing 9.1. The parameter `O_RDWR` specifies that the memory segment shall be opened for reading and writing.

With the function `ftruncate` (in line 26), the program expands the memory segment to the specified size, and with `mmap` (in line 35), the memory segment is appended to the virtual address space of the calling process.

Writing the string into the segment with the function `sprintf` (in line 50) and the read operation to the segment with the function `printf` (in line 59) are self-explanatory. The return value of `sprintf` indicates whether the write operation was successful because the return value contains the number of characters written, and in case of an error, it has value -1.

Das Programmbeispiel in Listing 9.2 zeigt in der Programmiersprache C den Lebenszyklus eines POSIX-Speichersegments unter Linux und ist dem Programmbeispiel in Listing 9.1 dementsprechend sehr ähnlich.

Die Namen von POSIX-Speichersegmenten müssen wie im Beispiel mit einem Schrägstrich – Slash (`/`) beginnen und dürfen im weiteren Verlauf keinen weiteren Slash enthalten. Im Programmbeispiel in Listing 9.2 ist der Name des neuen Speichersegments `/mymemory` (siehe Zeile 13).

Das Programm erstellt mit der Funktion `shm_open` (in Zeile 16) ein Speichersegment. Der Parameter `O_CREAT` definiert, dass ein eventuell existierendes Speichersegment mit dem gleichen Namen nicht überschrieben, sondern nur dessen Deskriptor zurückgeliefert werden soll. Der Parameter `0600` in der gleichen Zeile definiert genau wie im Programmbeispiel in Listing 9.1 die Zugriffsrechte. Der Parameter `O_RDWR` definiert, dass das Speichersegment lesend und schreibend geöffnet werden soll.

Mit der Funktion `ftruncate` (in Zeile 26) vergrößert das Programm das Speichersegment auf die definierte Größe und mit `mmap` (in Zeile 35) wird das Speichersegment an den virtuellen Adressraum des aufrufenden Prozess angehängt.

Das Schreiben der Zeichenkette in das Segment mit der Funktion `sprintf` (in Zeile 50) und der Lesezugriff auf das Segment mit der Funktion `printf` (in Zeile 59) sind selbsterklärend. Der Rückgabewert von `sprintf` gibt Auskunft darüber, ob der Schreibzugriff erfolgreich durchgeführt wurde, denn der Rückgabewert enthält die Anzahl der geschriebenen Zeichen und im Fehlerfall ist er -1.

Finally, the program detaches the segment from the virtual address space of the calling process with `munmap` (in line 65), closes the descriptor with `close` (in line 74) and removes (erases) the segment from the operating system with `shm_unlink` (in line 83).

Abschließend löst das Programm das Segment vom virtuellen Adressraum des aufrufenden Prozess mit `munmap` (in Zeile 65), schließt den Deskriptor mit `close` (in Zeile 74) und entfernt (löscht) das Segment aus dem Betriebssystem mit `shm_unlink` (in Zeile 83).

```
1 #include <sys/mman.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <unistd.h>
7 #define MAXMEMSIZE 20
8
9 int main(int argc, char **argv) {
10   int shmem_fd;          // File descriptor of the segment
11   int rc_sprintf;        // Return value of sprintf
12   void* shmem_pointer;
13   const char name[] = "/mymemory"; {
14
15   // Create shared memory segment
16   shmem_fd = shm_open(name, O_CREAT | O_RDWR, 0600);
17   if (shmem_fd < 0) {
18     printf("Unable to create the shared memory segment.\n");
19     perror("shm_open");
20     exit(1);
21   } else {
22     printf("Shared memory segment %s was created.\n", name);
23   }
24
25   // Adjust the size of the memory segment
26   if (ftruncate(shmem_fd, MAXMEMSIZE) < 0) {
27     printf("Unable to adjust the size.\n");
28     perror("ftruncate");
29     exit(1);
30   } else {
31     printf("The size of %s was adjusted.\n", name);
32   }
33
34   // Attach the shared memory segment
35   shmem_pointer = mmap(0,
36                        MAXMEMSIZE,
37                        PROT_WRITE,
38                        MAP_SHARED,
39                        shmem_fd,
40                        0);
41   if (shmem_pointer < 0) {
42     printf("Unable to attach %s.\n", name);
43     perror("mmap");
44     exit(1);
45   } else {
```

```
46      printf("%s was attached.\n", name);
47    }
48
49    // Write a string into the shared memory segment
50    rc_sprintf = sprintf(shmem_pointer, "Hello World.");
51    if (rc_sprintf < 0) {
52      printf ("The write operation failed.\n");
53      exit(1);
54    } else {
55      printf("%i characters were written.\n", rc_sprintf);
56    }
57
58    // Read the string from the shared memory segment
59    if (printf("Content of %s: %s\n", name, shmem_pointer) < 0) {
60      printf ("The read operation failed.\n");
61      exit(1);
62    }
63
64    // Detach the shared memory segment
65    if (munmap(shmem_pointer, MAXMEMSIZE) < 0) {
66      printf("Unable to detach %s.\n", name);
67      perror("munmap");
68      exit(1);
69    } else {
70      printf("%s was detached from the process.\n", name);
71    }
72
73    // Close the shared memory segment
74    if (close(shmem_fd) < 0) {
75      printf("Unable to close %s.\n", name);
76      perror("close");
77      exit(1);
78    } else {
79      printf("%s was closed.\n", name);
80    }
81
82    // Erase the shared memory segment
83    if (shm_unlink(name) < 0) {
84      printf("Unable to erase %s.\n", name);
85      perror("shm_unlink");
86      exit(1);
87    } else {
88      printf("%s was erased.\n", name);
89    }
90
91    exit(0);
92  }
```

Listing 9.2: Program Example for POSIX Shared Memory Segments

Compiling the program in Linux using the GNU C compiler (`gcc`) with the option `-lrt` Das Übersetzen des Programms unter Linux mit dem GNU C Compiler (`gcc`) mit der Op-

for linking to the library (`librt`) and running it should result in the following output:

tion `-lrt` zur Verknüpfung mit der Bibliothek (`librt`) und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_2_shared_memory_posix.c -o Listing_9_2_shared_memory_posix -lrt
$ ./Listing_9_2_shared_memory_posix
Shared memory segment /mymemory was created.
The size of /mymemory was adjusted.
/mymemory was attached.
12 characters were written.
Content of /mymemory: Hello World.
/mymemory was detached from the process.
/mymemory was closed.
/mymemory was erased.
```

In Linux, the presence of POSIX memory segments, their memory usage, access privileges, and their contents can be examined in the directory `/dev/shm`. The program example in Listing 9.2 creates the following memory segment:

Unter Linux können die Existenz von POSIX-Speichersegmenten, deren Speicherverbrauch, Zugriffsrechte und auch der Inhalt im Verzeichnis `/dev/shm` kontrolliert werden. Das Programmbeispiel in Listing 9.2 erzeugt folgendes Speichersegment:

```
$ ls -l /dev/shm/
-rw------- 1 bnc bnc 20  8. Okt 11:46 mymemory
```

As expected, the memory segment has a size of 20 bytes because that's the size specified by `ftruncate` (in line 26). The content of the file `mymemory` is the data written into the memory segment with `sprintf` (in line 50) .

Wie zu erwarten ist das Speichersegment 20 Bytes groß, weil das die Größe ist, die mit `ftruncate` (in Zeile 26) definiert wurde. Der Inhalt der Datei `mymemory` sind die Daten, die mit `sprintf` (in Zeile 50) in das Speichersegment geschrieben wurden.

```
$ cat /dev/shm/mymemory
Hello World.
```

9.3.3

Message Queues (System V)

Nachrichtenwarteschlangen (System V)

Message queues are linked lists in which processes can store, and from where processes can fetch messages according to the FIFO principle (see Figure 9.12).

One advantage in comparison to shared memory segments is that the processes which access a message queue do not need to coordinate their access operations by themselves. The operating system does it.

Usually, a process that wants to send a message into a message queue lacking free capacity,

Nachrichtenwarteschlangen (englisch: *Message Queues*) sind verkettete Listen, in die Prozesse nach dem FIFO-Prinzip Nachrichten ablegen und aus denen sie Nachrichten abholen können (siehe Abbildung 9.12).

Ein Vorteil gegenüber gemeinsamen Speicherbereichen ist, dass die auf eine Nachrichtenwarteschlange zugreifenden Prozesse ihre Zugriffe nicht selbst koordinieren müssen. Dieses ist Aufgabe des Betriebssystems.

Normalerweise ist ein Prozess, der eine Nachricht in eine Nachrichtenwarteschlange ohne

is blocked until there is enough free capacity available. The same applies to read operations. If a process tries to fetch a message from an empty message queue, the process is blocked until a message is present.

Alternatively, non-blocking read and write operations to message queues are also possible. In such a case, when trying to write into a full message queue or read from an empty message queue, the function would return an error message, and the process continues to run and needs to handle the situation.

Exactly as with the shared memory segments, the termination of a process does not affect the data that is already stored in a message queue. It means that messages in the message queue are preserved even after the process that created them has finished.

Each message sent to a message queue gets a message type (a positive integer) assigned. Processes receiving messages from message queues can filter them according to their type. This allows, for example, using a single message queue to implement multiple logical message queues differentiated by different message types.

freie Kapazität senden möchte, so lange blockiert, bis wieder Kapazität frei ist. Ähnlich ist es bei Lesezugriffen. Versucht ein Prozess eine Nachricht aus einer leeren Nachrichtenwarteschlange zu empfangen, ist der Prozess so lange blockiert, bis eine Nachricht vorliegt.

Alternativ sind auch nicht-blockierende Lese- und Schreibzugriffe auf Nachrichtenwarteschlangen möglich. In einem solchen Fall würde beim Versuch, in eine volle Nachrichtenwarteschlange zu schreiben oder aus einer leeren Nachrichtenwarteschlange zu lesen, die jeweilige Funktion eine Fehlermeldung zurückgeben und der Prozess weiterarbeiten. Der Prozess muss dann selbst mit der Situation umgehen.

Genau wie bei den gemeinsamen Speicherbereichen hat die Beendigung eines Prozesses keinen Einfluss auf die Daten, die sich bereits in einer Nachrichtenwarteschlange befinden. Nachrichten bleiben also auch nach der Beendigung des Erzeuger-Prozesses in der Nachrichtenwarteschlange erhalten.

Jeder Nachricht, die in eine Nachrichtenwarteschlange gesendet wird, wird ein Nachrichtentyp (eine positive ganze Zahl) zugewiesen. Prozesse, die Nachrichten aus Nachrichtenwarteschlangen empfangen, können diese Nachrichten anhand des Nachrichtentyps filtern. Damit ist es zum Beispiel möglich über eine einzige Nachrichtenwarteschlange mehrere logische, nach verschiedenen Nachrichtentypen unterschiedene, Nachrichtenwarteschlangen zu realisieren.

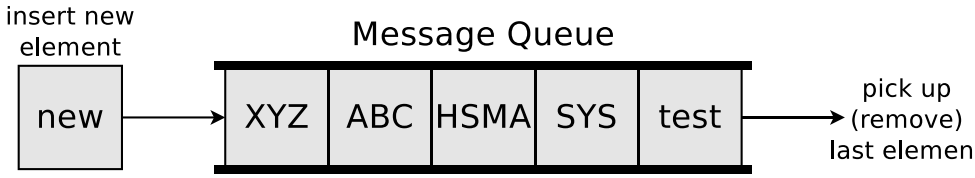

Figure 9.12: Message Queues implement Interprocess Communication according to the FIFO Principle

The interprocess communication concept described in this section is *System V message queues*. Apart from that, POSIX message queues also exist (see Section 9.3.4).

Das in diesem Abschnitt beschriebene Konzept zur Interprozesskommunikation sind *System V-Nachrichtenwarteschlangen*. Daneben existieren auch POSIX-Nachrichtenwarteschlangen (siehe Abschnitt 9.3.4).

Linux provides four system calls for working with System V message queues (see Table 9.4).

Linux stellt vier Systemaufrufe für die Arbeit mit System V-Nachrichtenwarteschlangen bereit (siehe Tabelle 9.4).

Table 9.4: Linux System Calls for working with System V Message Queues

System call	Purpose
`msgget`	Create a message queue or access an existing one
`msgsnd`	Write (send) a message into a message queue
`msgrcv`	Read (receive) a message from a message queue
`msgctl`	Request the status (i.a. permissions) of a message queue, modify or erase it

The program example in Listing 9.3 demonstrates in the programming language C the creation of a System V message queue in Linux using the function `msgget` (see line 23), writing a message into the message queue with `msgsnd` (see line 42), reading this message from the message queue with `msgrcv` (see line 57), and finally erasing the message queue from the operating system with `msgctl` (see line 68).

This example is also quite unrealistic because the same process does both: the sending and the receiving part. However, this example also demonstrates the implementation of the operations mentioned above in a compact way. Furthermore, it is simple to relocate parts of the source code to a second process. It would make sense for a second process to gain access to an existing message queue and write to it or read from it.

If the message queue has successfully been created using the `msgget` function in line 23, the return value is the message queue ID, which is unique in the operating system. If the return value has the value `-1`, then the attempt to create the message queue has failed. Possible reasons are, for example, that the main memory and swap memory have no more free capacity or that the maximum number of message queues in the operating system has been reached. The key of the new message queue is specified in line 15. Just as in Listing 9.1, the parameter `IPC_CREAT` in line 23 indicates that a possibly existing message queue with the same message queue key must not be overwritten, but only its message queue ID must be returned. The parameter `0600` in the same line also specifies the privileges. In this example, only the user

Das Programmbeispiel in Listing 9.3 zeigt in der Programmiersprache C die Erzeugung einer System V-Nachrichtenwarteschlange unter Linux mit der Funktion `msgget` (in Zeile 23), das Schreiben einer Nachricht in die Nachrichtenwarteschlange mit `msgsnd` (in Zeile 42), das Auslesen dieser Nachricht aus der Nachrichtenwarteschlange mit `msgrcv` (in Zeile 57) und abschließend das Entfernen (Löschen) der Nachrichtenwarteschlange aus dem Betriebssystem mit `msgctl` (in Zeile 68).

Auch dieses Beispiel geht an der Realität vorbei, weil der sendende und der empfangende Prozess identisch sind. Aber auch dieses Beispiel zeigt auf kompakte Art und Weise die Realisierung der oben genannten Schritte. Zudem ist es einfach, einzelne Codebereiche in einen zweiten Prozess zu verlagern. Sinnvollerweise würde ein zweiter Prozess den Zugriff auf eine existierende Nachrichtenwarteschlange und das Schreiben oder Lesen daraus enthalten.

War die Erzeugung der Nachrichtenwarteschlange mit der Funktion `msgget` in Zeile 23 erfolgreich, ist der Rückgabewert die im Betriebssystem eindeutige Message Queue-ID. Hat der Rückgabewert den Wert `-1`, ist der Versuch, die Nachrichtenwarteschlange anzulegen, fehlgeschlagen. Mögliche Gründe sind beispielsweise, dass der Hauptspeicher und Auslagerungsspeicher vollständig belegt sind, oder dass die maximale Anzahl an Nachrichtenwarteschlangen im Betriebssystem bereits erreicht wurde. Ein Message-Queue-Key für die neue Nachrichtenwarteschlange wird in Zeile 15 definiert. Genau wie bei Listing 9.1 legt auch hier der Parameter `IPC_CREAT` in Zeile 23 fest, dass eine eventuell existierende Nachrichtenwarteschlange mit der gleichen Message-Queue-Key nicht überschrieben, sondern nur ihre Message Queue-ID zurück-

who creates the message queue has read and write access to it.

When attempting to write a message to the message queue using the `msgsnd` function, the variable `rc_msgget` is specified as the target of the write statement. This variable contains the return value of the `msgget` function. The `msgsnd` function writes the contents of a send buffer, which receives a string in line 36. The structure of the send buffer is specified in the lines 9-12. The structure that is specified in this example includes a character string of up to 80 characters per send buffer instance and a number that indicates the message type. In line 34, the value 1 is specified as the message type. If the return value of the function is a positive integer, then the write operation was successful, and the return value is equal to the number of characters sent. In the event of an error, the return value is -1.

Without the parameter `IPC_NOWAIT` in line 42, if the message queue has no more free capacity, the process would be blocked until sufficient free capacity is available. In the example in Listing 9.3, this is unlikely to happen. However, in situations where no new message queue is created but an existing one is used, this is possible.

In line 57, the program tries to read the first message of type 1 from the message queue using the function `msgrcv`. As parameters of the function, the variable `rc_msgget` with the return value of `msgget` is used, and among other things, a receive buffer, which specifies the structure of the message and corresponds to the structure of the send buffer, is specified. The message type is indicated in line 51. It is again 1. The parameter `IPC_NOWAIT` ensures that when calling the `msgrcv` function, the process does not wait in blocked state for the arrival of a message in case of an empty message queue, but instead aborts with an error message.

If `mtype` has the value 0, the first message is fetched from the message queue. If the value

geliefert werden soll. Der Parameter `0600` in der gleichen Zeile definiert auch hier die Zugriffsrechte. In diesem Fall darf nur der Benutzer, der die Nachrichtenwarteschlange anlegt, auf diese lesend und schreibend zugreifen.

Beim Versuch, mit der Funktion `msgsnd` eine Nachricht in die Nachrichtenwarteschlange zu schreiben, ist mit der Variable `rc_msgget` der Rückgabewert von `msgget` als Ziel der Schreibanweisung angegeben. Geschrieben wird der Inhalt eines Sendepuffers, dem in Zeile 36 eine Zeichenkette übergeben wird. Die Struktur des Sendepuffers ist in den Zeilen 9-12 definiert. Die im Beispiel definierte Struktur sieht nicht nur eine maximal 80 Zeichen lange Zeichenkette pro Sendepufferinstanz vor, sondern auch eine Zahl, die den Nachrichtentyp definiert. In Zeile 34 wird als Nachrichtentyp der Wert 1 definiert. Ist der Rückgabewert der Funktion eine positive ganze Zahl, dann war der Schreibzugriff erfolgreich und der Rückgabewert entspricht der Anzahl der gesendeten Zeichen. Im Fehlerfall ist der Rückgabewert -1.

Ohne den Parameter `IPC_NOWAIT` in Zeile 42 wäre der Prozess, wenn die Nachrichtenwarteschlange keine freie Kapazität mehr hat, so lange blockiert, bis ausreichend freie Kapazität vorliegt. In unserem Beispiel in Listing 9.3 ist es unwahrscheinlich, dass so etwas passieren kann. Aber in Fällen, in denen nicht eine neue Nachrichtenwarteschlange erzeugt, sondern auf eine bereits existierende zugegriffen wird, ist es durchaus möglich.

In Zeile 57 versucht das Programm, mit der Funktion `msgrcv` die erste Nachricht vom Nachrichtentyp 1 aus der Nachrichtenwarteschlange zu lesen. Als Parameter der Funktion sind neben der Variable `rc_msgget` mit dem Rückgabewert von `msgget` unter anderem ein Empfangspuffer angegeben, der die Struktur der Nachricht definiert und der Struktur des Sendepuffers entspricht. Der Nachrichtentyp ist erneut 1 und wird in Zeile 51 definiert. Der Parameter `IPC_NOWAIT` beim Aufruf von `msgrcv` stellt sicher, dass der Prozess im Fall einer leeren Nachrichtenwarteschlange nicht im blockierten Zustand auf das Vorhandensein einer Nachricht wartet, sondern stattdessen mit einer Fehlermeldung abbricht.

Hat `mtype` den Wert 0, wird die erste Meldung aus der Nachrichtenwarteschlange gelesen.

of `mtype` is a positive integer, the first message of this type is fetched. If the value of `mtype` is negative, the first message of a type less than or equal to the absolute value of `mtype` is fetched.

Also, when trying to erase the message queue with the function `msgctl` and the parameter `IPC_RMID` in line 68, the variable `rc_msgget`, which contains the message queue ID, is specified as a parameter. The return value for these functions is also -1 in case of an error.

Ist der Wert von `mtype` eine positive ganze Zahl, wird die erste Meldung dieses Typs gelesen. Ist der Wert von `mtype` negativ, wird die erste Nachricht gelesen, deren Typ kleiner oder gleich dem absoluten Wert von `mtype` ist.

Auch beim Versuch, die Nachrichtenwarteschlange mit der Funktion `msgctl` und dem Parameter `IPC_RMID` in Zeile 68 zu löschen, ist als Parameter die Variable `rc_msgget` angegeben, die die Message Queue-ID enthält. Auch bei diesen Funktionen ist der Rückgabewert im Fehlerfall -1.

```
1 #include <stdlib.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdio.h>
5 #include <sys/msg.h>
6 #include <string.h>
7
8 // Template of a buffer for msgsnd and msgrcv
9 typedef struct msgbuf {
10   long mtype;
11   char mtext[80];
12 } msg;
13
14 int main(int argc, char **argv) {
15   int key = 12345;
16   int rc_msgget;
17   int rc_msgctl;
18   int rc_msgrcv;
19   // Create a receive buffer and a send buffer
20   msg sendbuffer, receivebuffer;
21
22   // Create a message queue
23   rc_msgget = msgget(key, IPC_CREAT | 0600);
24   if(rc_msgget < 0) {
25     printf("Unable to create the message queue.\n");
26     perror("msgget");
27     exit(1);
28   } else {
29     printf("Message queue %i with ID %i has been created.\n",
30             key, rc_msgget);
31   }
32
33   // Specifiy the message type
34   sendbuffer.mtype = 1;
35   // Write the message into the send buffer
36   strncpy(sendbuffer.mtext, "Testmessage", 11);
37
38   // Write a message into the message queue
39   if (msgsnd(rc_msgget,
```

```
40              &sendbuffer,
41              sizeof(sendbuffer.mtext),
42              IPC_NOWAIT) == -1) {
43    printf("Unable to write the message.\n");
44    perror("msgsnd");
45    exit(1);
46  } else {
47    printf("Message written: %s\n", sendbuffer.mtext);
48  }
49
50  // Fetch the first message of type 1 from the message queue
51  receivebuffer.mtype = 1;
52
53  rc_msgrcv = msgrcv(rc_msgget,
54                     &receivebuffer,
55                     sizeof(receivebuffer.mtext),
56                     receivebuffer.mtype,
57                     MSG_NOERROR | IPC_NOWAIT);
58  if (rc_msgrcv < 0) {
59    printf("Unable to fetch a message from the queue.\n");
60    perror("msgrcv");
61    exit(1);
62  } else {
63    printf("Fetched message: %s\n", receivebuffer.mtext);
64    printf("Message length: %i characters.\n", rc_msgrcv);
65  }
66
67  // Erase the message queue
68  rc_msgctl = msgctl(rc_msgget, IPC_RMID, 0);
69  if (rc_msgctl < 0) {
70    printf("The message queue has been erased.\n");
71    perror("msgctl");
72    exit(1);
73  } else {
74    printf("Message queue %i with ID %i has been erased.\n",
75            key, rc_msgget);
76  }
77
78  exit(0);
79 }
```

Listing 9.3: Program Example for System V Message Queues

Compiling the program with the GNU C compiler (gcc) in Linux and running it afterward should result in the following output:

Das Übersetzen des Programms mit dem GNU C Compiler (gcc) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_3_message_queue_systemv.c -o Listing_9_3_message_queue_systemv
$ ./Listing_9_3_message_queue_systemv
Message queue 12345 with ID 131072 has been created.
Message written: Testmessage
Fetched message: Testmessage
Message length: 80 characters.
Message queue 12345 with ID 131072 has been erased.
```

It is interesting to observe the different states in which the message queue that the program creates and then erases is in during runtime. These states can be traced from the Linux command line interface using the `ipcs` utility, which returns information about existing message queues.

After creating the message queue with the `msgget` function, the permissions, and the size that have been specified in the program example, are visible in the output of `ipcs`. Using the `-q` option, `ipcs` is instructed only to return the message queues. The columns `used-bytes` and `messages` indicate that the message queue is still empty.

Interessant ist die Beobachtung der verschiedenen Zustände, in denen sich die im Programm erstellte und abschließend gelöschte Nachrichtenwarteschlange während der Laufzeit befindet. Diese Zustände kann man mit dem Programm `ipcs`, das Informationen über bestehende Nachrichtenwarteschlange liefert, auf der Kommandozeile von Linux nachvollziehen.

Nach der Erzeugung der Nachrichtenwarteschlange mit der Funktion `msgget` sind in der Ausgabe von `ipcs` die im Programmbeispiel festgelegten Zugriffsrechte und die Größe sichtbar. Mit der Option `-q` wird `ipcs` angewiesen, nur die Nachrichtenwarteschlangen auszugeben. Aus den Spalten `used-bytes` und `messages` geht hervor, dass die Nachrichtenwarteschlange noch leer ist.

```
$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 131072     bnc        600        0            0
```

Just like the shared memory key for the shared memory segments (see Section 9.3.3), the message queue key is also specified in hexadecimal notation. A conversion to the decimal system with the command `printf` shows that the message queue key, which is specified in the program example, matches the one that is actually used.

Genau wie zuvor schon der Shared Memory-Key bei den gemeinsamen Speicherbereichen (siehe Abschnitt 9.3.3) ist auch der Message Queue-Key in Hexadezimalschreibweise angegeben. Eine Konvertierung ins Dezimalsystem mit dem Kommando `printf` zeigt, dass der im Programmbeispiel festgelegte Message Queue-Key mit dem tatsächlich verwendeten übereinstimmt.

```
$ printf "%d\n" 0x00003039        # Convert from hexadecimal to decimal
12345
```

After writing the message into the message queue with the function `msgsnd`, the output of `ipcs` has changed in the columns `used-bytes` and `messages`. The size of all messages in the message queue and the number of messages are indicated here.

Nach dem Schreiben der Nachricht in die Nachrichtenwarteschlange mit der Funktion `msgsnd` verändert sich die Ausgabe von `ipcs` in den Spalten `used-bytes` und `messages`. Hier sind die Größe aller Nachrichten in der Nachrichtenwarteschlange und die Anzahl der Nachrichten angegeben.

```
$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00003039 131072     bnc        600        80           1
```

After erasing the message queue with `msgctl`, the output of `ipcs` does not show anything of the message queue anymore.

Erasing message queues is also possible using the `ipcrm` command from the command line interface, just as with shared memory areas. The following command removes the message queue with the ID `131072` from the operating system:

Nach dem Löschen der Nachrichtenwarteschlange mit `msgctl` ist auch in der Ausgabe von `ipcs` nichts mehr davon zu sehen.

Das Löschen von Nachrichtenwarteschlangen ist genau wie bei gemeinsamen Speicherbereichen auch von der Eingabeaufforderung aus mit dem Kommando `ipcrm` möglich. Das folgende Kommando entfernt die Nachrichtenwarteschlange mit der Message Queue-ID `131072` aus dem Betriebssystem:

```
$ ipcrm msg 131072
resource(s) deleted
```

9.3.4

Message Queues (POSIX)

POSIX-Nachrichtenwarteschlangen

An alternative way of implementing interprocess communication with message queues is offered by the POSIX standard.

Table 9.5 contains the C function calls for POSIX message queues specified in the header file `mqueue.h`

Eine alternative Möglichkeit, um Interprozesskommunikation mit Nachrichtenwarteschlangen zu realisieren, bietet der Standard POSIX.

Tabelle 9.5 enthält die in der Header-Datei `mqueue.h` definierten C-Funktionsaufrufe für POSIX-Nachrichtenwarteschlangen.

Table 9.5: C Function Calls for POSIX Message Queues

Function call	Purpose
`mq_open`	Create a message queue or access an existing one
`mq_send`	Write (send) a message into a message queue. Blocking operation, except the message queue is opened with the parameter `O_NONBLOCK`
`mq_timedsend`	Write (send) a message into a message queue. Blocking operation with specified timeout
`mq_receive`	Read (receive) a message from a message queue. Blocking operation, except the message queue is opened with the parameter `O_NONBLOCK`
`mq_timedreceive`	Read (receive) a message from a message queue. Blocking operation with specified timeout
`mq_getattr`	Query the attributes of a message queue. e.g., number of messages in the queue, maximum message length, maximum number of messages
`mq_setattr`	Modify the attributes of a message queue
`mq_notify`	The process shall be notified as soon as a message is available
`mq_close`	Close a message queue
`mq_unlink`	Erase a message queue

The program example in Listing 9.4 demonstrates the life cycle of a POSIX message queue on Linux using the C programming language. It is quite similar to the program example in Listing 9.3.

As shown in the example, POSIX message queue names must start with a slash (`/`) and must not contain another slash [92]. In the program example in Listing 9.4, the name of the new message queue is `/myqueue` (see line 8).

The program creates a message queue with the function `mq_open` (in line 25). The parameter `O_CREAT` specifies that any existing message queue with the same name should not be overwritten, but only its descriptor should be returned. The parameter `0600` in the same line specifies the access privileges in the same way as in the program example in Listing 9.3. The parameter `O_RDWR` specifies that the message queue shall be opened for reading and writing. The `O_NONBLOCK` parameter specifies that the message queue shall be opened in a non-blocking way [60]. This form of opening strongly influences writing messages to the message queue or reading messages from it.

The program tries to write a message into the message queue using the function `mq_send` (in line 38). The message to be sent is a string (see line 9). In addition, a non-negative integer value is specified as the message priority. The minimum possible message priority is the value zero.

The program requests an overview of the message queue attributes using the function `mq_getattr` in line 50. Among these attributes is the number of messages inside the message queue (`mq_curmsgs`).

Line 67 contains an attempt to read the oldest message with the highest priority from the message queue using the function `mq_receive`. Parameters of the function include the variable `mymq_descriptor` with the return value of `mq_open` and a receive buffer. If the function's return value is a positive integer, the read

Das Programmbeispiel in Listing 9.4 zeigt in der Programmiersprache C den Lebenszyklus einer POSIX-Nachrichtenwarteschlange unter Linux und ist dem Programmbeispiel in Listing 9.3 dementsprechend sehr ähnlich.

Die Namen von POSIX-Nachrichtenwarteschlangen müssen wie im Beispiel mit einem Schrägstrich – Slash (`/`) beginnen und dürfen im weiteren Verlauf keinen weiteren Slash enthalten [92]. Im Programmbeispiel in Listing 9.4 ist der Name der neuen Nachrichtenwarteschlange `/myqueue` (siehe Zeile 8).

Das Programm erstellt mit der Funktion `mq_open` (in Zeile 25) eine Nachrichtenwarteschlange. Der Parameter `O_CREAT` definiert, dass eine eventuell existierende Nachrichtenwarteschlange mit dem gleichen Namen nicht überschrieben, sondern nur ihr Deskriptor zurückgeliefert werden soll. Der Parameter `0600` in der gleichen Zeile definiert genau wie im Programmbeispiel in Listing 9.3 die Zugriffsrechte. Der Parameter `O_RDWR` definiert, dass die Nachrichtenwarteschlange lesend und schreibend geöffnet werden soll. Der Parameter `O_NONBLOCK` weist das nichtblockierende Öffnen der Nachrichtenwarteschlange an [60]. Diese Form des Zugriffs hat starken Einfluss auf das Schreiben von Nachrichten in die Nachrichtenwarteschlange bzw. Lesen von Nachrichten daraus.

Mit der Funktion `mq_send` (in Zeile 38) versucht das Programm eine Nachricht in die Nachrichtenwarteschlange zu schreiben. Die zu sendende Nachricht ist eine Zeichenkette (siehe Zeile 9). Zudem wird ein ganzzahliger nichtnegativer Wert als Nachrichtenpriorität angegeben. Die niedrigst mögliche Nachrichtenpriorität ist der Wert Null.

In Zeile 50 fordert das Programm mit der Funktion `mq_getattr` eine Übersicht der Eigenschaften der Nachrichtenwarteschlange an. Dazu gehört auch die Anzahl der aktuell in der Nachrichtenwarteschlange befindlichen Nachrichten (`mq_curmsgs`).

In Zeile 67 versucht das Programm, mit der Funktion `mq_receive` die älteste Nachricht mit der höchsten Priorität aus der Nachrichtenwarteschlange zu lesen. Als Parameter der Funktion sind neben der Variable `mymq_descriptor` mit dem Rückgabewert von `mq_open` unter anderem ein Empfangspuffer angegeben. Ist der Rückga-

operation was successful, and the return value is equal to the number of characters received. In case of an error, the return value is −1.

Finally, the program closes the message queue with `mq_close` (in line 80) and removes it with `mq_unlink` (in line 89).

Calling `mq_close` is not necessary for the program example in Listing 9.4 to work correctly because when a process that has access to a named POSIX message queue terminates, the queue is automatically closed. However, it is not good programming style to omit `mq_close`. The library function `mq_unlink` causes the deletion of a POSIX message queue. This is carried out as soon as there is no more reference to the message queue, i.e., when the last process that opened the message queue has called `mq_close` or has terminated.

bewert der Funktion eine positive ganze Zahl, war der Lesezugriff erfolgreich und der Rückgabewert entspricht der Anzahl der empfangenen Zeichen. Im Fehlerfall ist der Rückgabewert −1.

Abschließend schließt das Programm die Nachrichtenwarteschlange mit `mq_close` (in Zeile 80) und entfernt diese mit `mq_unlink` (in Zeile 89).

Der Aufruf von `mq_close` ist für die korrekte Funktion des Programmbeispiels in Listing 9.4 nicht nötig, da bei der Beendigung eines Prozesses mit Zugriff auf eine benannte POSIX-Nachrichtenwarteschlange, diese automatisch geschlossen wird. Es ist aber kein guter Programmierstil, wenn `mq_close` weggelassen wird. Die Bibliotheksfunktion `mq_unlink` weist das Löschen einer POSIX-Nachrichtenwarteschlange an. Dieses geschieht sobald es keine Referenz mehr auf die Nachrichtenwarteschlange gibt, also wenn der letzte Prozess, der diese geöffnet hat, `mq_close` aufgerufen hat, oder beendet ist.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <mqueue.h>
5
6 int main(int argc, char **argv) {
7   int rc_mq_receive;        // Return code) of mq_receive
8   const char mq_name[] = "/myqueue";
9   char message[] = "Hello World";
10
11  mqd_t mymq_descriptor;    // Message queue descriptor
12  int msg_prio = 0;         // Specify message priority
13  char recv_buffer[80];     // Receive buffer
14
15  struct mq_attr attr;
16  attr.mq_flags = 0;        // Flags
17  attr.mq_maxmsg = 10;      // Maximum number of messages
18  attr.mq_msgsize = 80;     // Maximum message size (bytes)
19  attr.mq_curmsgs = 0;      // Number of messages in the queue
20
21  // Fill the memory area of the receive buffer with zeros
22  memset(&recv_buffer, 0, sizeof(recv_buffer));
23
24  // Create a message queue
25  mymq_descriptor = mq_open(mq_name,
26                            O_RDWR | O_CREAT | O_NONBLOCK,
27                            0600,
28                            &attr);
29  if (mymq_descriptor < 0) {
30    printf("Unable to create the message queue.\n");
```

```
31    perror("mq_open");
32    exit(1);
33  } else {
34    printf("Message queue %s can be used now.\n", mq_name);
35  }
36
37  // Write a message into the message queue
38  if (mq_send(mymq_descriptor,
39              message,
40              strlen(message),
41              msg_prio) == -1) {
42    printf("Unable to write the message.\n");
43    perror("mq_send");
44    exit(1);
45  } else {
46    printf("Message written: %s\n", message);
47  }
48
49  // Query the attributes of a message queue
50  if (mq_getattr(mymq_descriptor, &attr) == -1) {
51    printf("Unable to query the attributes.\n");
52    perror("mq_getattr");
53  }
54
55  // Query the number of messages in the message queue
56  if (attr.mq_curmsgs == 0) {
57    printf("%s does not contain messages.\n", mq_name);
58  } else if (attr.mq_curmsgs == 1) {
59    printf("%s contains %d messages.\n",
60            mq_name,
61            attr.mq_curmsgs);
62  } else {
63    printf("Messages in %s: %d\n", mq_name, attr.mq_curmsgs);
64  }
65
66  // Fetch the oldest message of the highest priority
67  rc_mq_receive = mq_receive(mymq_descriptor,
68                             recv_buffer,
69                             sizeof(recv_buffer),
70                             &msg_prio);
71  if (rc_mq_receive < 0) {
72    printf("Unable to fetch a message from the queue.\n");
73    perror("mq_receive");
74  } else {
75    printf("Fetched message: %s\n", recv_buffer);
76    printf("Message length %i characters.\n", rc_mq_receive);
77  }
78
79  // Close the message queue
80  if (mq_close(mymq_descriptor) < 0) {
81    printf("Unable to close %s.\n", mq_name);
```

```
82      perror("mq_close");
83      exit(1);
84    } else {
85      printf("Message queue %s has been closed.\n", mq_name);
86    }
87
88    // Erase the message queue
89    if (mq_unlink(mq_name) < 0) {
90      printf("Unable to erase %s.\n", mq_name);
91      perror("mq_unlink");
92      exit(1);
93    } else {
94      printf("Message queue %s has been erased.\n", mq_name);
95    }
96
97    exit(0);
98 }
```

Listing 9.4: Program Example for POSIX Message Queues

Compiling the program in Linux using the GNU C compiler (`gcc`) with the option `-lrt` for linking to the library (`librt`) and running it should result in the following output:

Das Übersetzen des Programms unter Linux mit dem GNU C Compiler (`gcc`) mit der Option `-lrt` zur Verknüpfung mit der Bibliothek (`librt`) und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_4_message_queue_posix.c -o Listing_9_4_message_queue_posix -lrt
$ ./Listing_9_4_message_queue_posix
Message queue /myqueue can be used now.
Message written: Hello World
/myqueue contains 1 messages.
Fetched message: Hello World
Message length 11 characters.
Message queue /myqueue has been closed.
Message queue /myqueue has been erased.
```

In Linux, the presence of POSIX message queues and their access privileges can be examined in the directory `/dev/mqueue`. The program example in Listing 9.2 creates the following message queue:

Unter Linux können die Existenz von POSIX-Nachrichtenwarteschlangen und deren Zugriffsrechte im Verzeichnis `/dev/mqueue` kontrolliert werden. Das Programmbeispiel in Listing 9.4 erzeugt folgende Nachrichtenwarteschlange:

```
$ ls -l /dev/mqueue/
-rw------- 1 bnc bnc 80  7. Okt 18:17 myqueue
```

The content of the file `myqueue` includes some information about the state of the message queue. Of particular interest is the record `QSIZE`. It contains the size of all messages in the message queue and the number of messages.

Der Inhalt der Datei `myqueue` enthält einige Informationen über den aktuellen Zustand der Nachrichtenwarteschlange. Besonders hervorzuheben ist der Eintrag `QSIZE`. Hier sind die Größe aller Nachrichten in der Nachrichtenwarteschlange und die Anzahl der Nachrichten angegeben.

```
$ cat /dev/mqueue/myqueue
QSIZE:13         NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

The default sizes for `mq_maxmsg` (maximum number of messages per queue) and `mq_msgsize` (maximum message size in bytes), which are set in the program example in Listing 9.4, in the lines 17 and 18, can also be obtained from the following files in Linux:

Die Standardgrößen für `mq_maxmsg` (maximale Anzahl an Nachrichten pro Warteschlange) und für `mq_msgsize` (maximale Nachrichtengröße in Bytes), die im Programmbeispiel in Listing 9.4 in den Zeilen 17 und 18 gesetzt werden, können unter Linux auch in den folgenden Dateien eingesehen werden:

```
$ cat  /proc/sys/fs/mqueue/msgsize_default
8192
$ cat  /proc/sys/fs/mqueue/msg_default
10
```

The same directory contains additional files that provide information about the system-wide limitations.

Im gleichen Verzeichnis befinden sich noch weitere Dateien, die über die systemweiten Maximalwerte informieren.

9.3.5 Pipes

Kommunikationskanäle

Modern operating systems provide interprocess communication via two different types of pipes: anonymous pipes and named pipes.

Moderne Betriebssysteme ermöglichen Interprozesskommunikation über zwei verschiedene Arten von Pipes, nämlich anonyme und benannte Pipes.

Anonymous Pipes

An *anonymous pipe*, also called *unnamed pipe* [120], is a buffered communication channel between two processes, and it works according to the FIFO principle.

If a process tries to write data into a full pipe, then the process is blocked until there is free capacity in the pipe. The same happens when reading data. If a process tries to read data from an empty pipe, the process is blocked until data is present.

Creating a pipe in Linux is done with the system call `pipe`. Thereby, the operating system kernel creates an inode (see Section 6.2) and two *file descriptors* (*handles*).

Processes access the handles with `read` and `write` system calls (or standard library functions) for reading data from or writing data into the pipe.

Anonymous pipes are always unidirectional. This means that communication is only possible in one direction (see Figure 9.13). One process can write into the pipe, and one process can

Anonyme Pipes

Eine *anonyme Pipe* (englisch: *unnamed Pipe* [120]) ist ein gepufferter Kommunikationskanal zwischen zwei Prozessen und arbeitet nach dem FIFO-Prinzip.

Versucht ein Prozess Daten in eine volle Pipe zu schreiben, ist der Prozess so lange blockiert, bis es wieder freien Platz in der Pipe gibt. Ähnlich ist es bei Lesezugriffen. Versucht ein Prozess aus einer leeren Pipe Daten zu lesen, ist der Prozess so lange blockiert, bis Daten vorliegen.

Das Anlegen einer Pipe geschieht unter Linux mit dem Systemaufruf `pipe`. Dabei legt der Betriebssystemkern einen Inode (siehe Abschnitt 6.2) und zwei *Zugriffskennungen* (englisch: *Handles*) bzw. *Deskriptoren* an.

Lese- und Schreibzugriffe auf eine Pipe sind über die beiden Zugriffskennungen mit den Systemaufrufen `read` und `write` oder alternativ mit den entsprechenden Standard-Bibliotheksfunktionen möglich.

Anonyme Pipes sind immer unidirektional. Das heißt, dass die Kommunikation nur in eine Richtung funktioniert (siehe Abbildung 9.13). Ein Prozess kann in die Pipe schreiben und

read from it. If communication in both directions is supposed to be possible at the same time, two pipes are necessary – one for each communication direction.

ein Prozess aus ihr lesen. Soll Kommunikation in beide Richtungen gleichzeitig möglich sein, sind zwei Pipes nötig, eine für jede mögliche Kommunikationsrichtung.

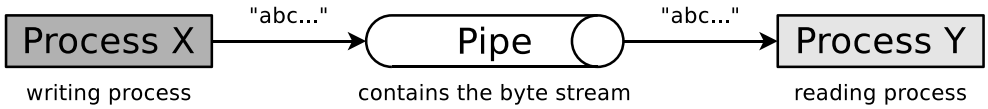

Figure 9.13: Interprocess Communication via Anonymous Pipes is only unidirectionally

Anonymous pipes can only be used to communicate between closely related processes. Only processes that are related via `fork` are allowed to communicate via anonymous pipes because when creating child processes with `fork`, the child processes inherit access to the file descriptors. If the last process, which has access to an anonymous pipe, terminates, the pipe is erased by the kernel.

At any time, a pipe can only be accessed by a single process. In contrast to shared memory segments, the kernel automatically ensures mutual exclusion of access operations.

The program example in Listing 9.5 demonstrates interprocess communication via anonymous pipes in Linux. In line 11, the program tries to create an anonymous pipe with the `pipe` function. Previously, it creates the two required file descriptors `testpipe[0]` for reading and `testpipe[1]` for writing in line 8. In line 20, the program tries to create a child process with the function `fork`. If the process creation has been successful, the parent process in line 34 closes the read channel of the pipe with the function `close` and writes a string into the write channel with the function `write` (in line 39). The child process also blocks the write channel of the pipe in line 47 with the function `close` and reads the string from the read channel in line 53 with the function `read`. A receive buffer is required for reception, which is set up in line 50 and has a capacity of 80 characters. Finally, the child process returns the received string to the command line.

Anonyme Pipes ermöglichen Kommunikation nur zwischen eng verwandten Prozessen. Nur Prozesse, die via `fork` eng verwandt sind, können über anonyme Pipes kommunizieren, denn bei der Erzeugung von Kindprozessen mit `fork` erben die Kindprozesse auch den Zugriff auf die Zugriffskennungen. Mit der Beendigung des letzten Prozesses, der Zugriff auf eine anonyme Pipe hat, wird diese vom Betriebssystemkern entfernt.

Zu jedem Zeitpunkt kann immer nur ein Prozess auf eine Pipe zugreifen. Im Gegensatz zu gemeinsamen Speichersegmenten stellt der Betriebssystemkern den wechselseitigen Ausschluss der Zugriffe automatisch sicher.

Das Programmbeispiel in Listing 9.5 zeigt wie Interprozesskommunikation via anonyme Pipes unter Linux möglich ist. In Zeile 11 versucht das Programm, eine anonyme Pipe mit der Funktion `pipe` anzulegen. Zuvor legt es in Zeile 8 die beiden benötigten Zugriffskennungen `testpipe[0]` zum Lesen und `testpipe[1]` zum Schreiben an. In Zeile 20 versucht das Programm, einen Kindprozess mit der Funktion `fork` zu erzeugen. War die Prozesserzeugung erfolgreich, schließt der Elternprozess in Zeile 34 den Lesekanal der Pipe mit der Funktion `close` und schreibt in Zeile 39 mit der Funktion `write` eine Zeichenkette in den Schreibkanal. Der Kindprozess blockiert in Zeile 47 den Schreibkanal der Pipe ebenfalls mit der Funktion `close` und liest in Zeile 53 mit der Funktion `read` die Zeichenkette aus dem Lesekanal. Zum Empfang ist ein Empfangspuffer nötig, der in Zeile 50 erzeugt wird und im Beispiel eine Kapazität von 80 Zeichen hat. Abschließend gibt der Kindprozess die empfangene Zeichenkette auf der Kommandozeile aus.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 void main() {
6   int pid_of_child;
7   // Create file descriptors (handles) for read and write
8   int testpipe[2];
9
10  // Create the pipe testpipe
11  if (pipe(testpipe) < 0) {
12    printf("Unable to create the Pipe.\n");
13    // Terminate the program (failure)
14    exit(1);
15  } else {
16    printf("The pipe testpipe has been created.\n");
17  }
18
19  // Create a child process
20  pid_of_child = fork();
21
22  // An error occured during fork
23  if (pid_of_child < 0) {
24    printf("An error occured during fork!\n");
25    // Terminate the program (failure)
26    exit(1);
27  }
28
29  // Parent process
30  if (pid_of_child > 0) {
31    printf("Parent process. PID: %i\n", getpid());
32
33    // Block the read channel of the pipe
34    close(testpipe[0]);
35
36    char message[] = "Test message";
37
38    // Write data into the write channel of the pipe
39    write(testpipe[1], &message, sizeof(message));
40  }
41
42  // Child process
43  if (pid_of_child == 0) {
44    printf("Child process. PID: %i\n", getpid());
45
46    // Block the write channel of the pipe
47    close(testpipe[1]);
48
49    // Create a receive buffer
50    char buffer[80];
51
```

```
52      // Read data from the read channel of the pipe
53      read(testpipe[0], buffer, sizeof(buffer));
54
55      // Return received data to the command line
56      printf("Data received: %s\n", buffer);
57    }
58 }
```

Listing 9.5: Program Example for Anonymous Pipes

Compiling the program with the GNU C compiler (`gcc`) in Linux and running it afterward should result in the following output:

Das Übersetzen des Programms mit dem GNU C Compiler (`gcc`) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_5_anonymous_pipe.c -o Listing_9_5_anonymous_pipe
$ ./Listing_9_5_anonymous_pipe
The pipe testpipe has been created.
Parent process. PID: 31465
Child process. PID: 31466
Data received: Test message
```

Named Pipes

Besides anonymous pipes, modern operating systems also provide *named pipes* [120]. These can also simply be called *FIFO*.

Processes that are not closely related to each other are able to communicate via named pipes. Furthermore, named pipes enable bidirectional communication between two processes.

Each named pipe is represented by a special file in the file system. For this reason, unlike anonymous pipes, they remain intact even when no process is using them.

In Linux, a pipe is created using the system call `mkfifo` or a standard library function of the same name. The program example in Listing 9.6, written in the programming language C, demonstrates how interprocess communication using named pipes is possible in Linux. In line 10, the program tries to create a named pipe called `testfifo` with all possible permissions using the function `mkfifo`. The function call creates a special file in the file system called `testfifo` in the current working directory. The first letter in the output of the command `testfifo` indicates that `testfifo` is a named pipe.

Benannte Pipes

Außer den anonymen Pipes ermöglichen moderne Betriebssysteme auch *benannte Pipes* (englisch: *named pipe* [120]). Diese heißen auch einfach *FIFO*.

Über benannte Pipes können auch nicht eng miteinander verwandte Prozesse kommunizieren. Zudem ermöglichen benannte Pipes bidirektionale Kommunikation zwischen zwei Prozessen.

Jede benannte Pipe ist durch einen Eintrag im Dateisystem repräsentiert. Aus diesem Grund bleiben sie im Gegensatz zu anonymen Pipes auch dann erhalten, wenn kein Prozess auf sie zugreift.

Das Anlegen einer Pipe geschieht unter Linux mit dem Systemaufruf `mkfifo` bzw. einer gleichnamigen Standard-Bibliotheksfunktion. Das Programmbeispiel in Listing 9.6 zeigt in der Programmiersprache C, wie Interprozesskommunikation mit benannten Pipes unter Linux möglich ist. In Zeile 10 versucht das Programm eine benannte Pipe mit dem Namen `testfifo` und größtmöglichen Zugriffsrechten mit der Funktion `mkfifo` anzulegen. Der Funktionsaufruf erzeugt im aktuellen Verzeichnis einen Dateisystemeintrag mit dem Namen `testfifo`. Der erste Buchstabe in der Ausgabe des Kommandos `ls` zeigt, das `testfifo` eine benannte Pipe ist.

The first character of each file in the output of `ls -l` indicates what type of file it is. If it is a normal file, there is just a hyphen at this point. The letter `d` indicates a directory. Sockets have the letter `s`, and pipes have the letter `p`. Device files for character-oriented devices have the letter `c`, and block-oriented devices have the letter `b`. Symbolic links, i.e., files that refer to other files, are identified by the letter `l`.

Das erste Zeichen jedes Dateisystemeintrags in der Ausgabe von `ls -l` gibt an, um welche Art von Datei es sich handelt. Im Fall einer normalen Datei würde sich an dieser Stelle nur ein Bindestrich befinden. Ein Verzeichnis identifiziert der Buchstabe `d`. Sockets haben den Buchstaben `s` und benannte Pipes den Buchstaben `p`. Gerätedateien für zeichenorientierte Geräte haben den Buchstaben `c`, und für blockorientierte Geräte steht der Buchstabe `b`. Symbolische Links (Verknüpfungen), also Dateien, die auf andere Dateien verweisen, identifiziert der Buchstabe `l`.

```
$ ls -la testfifo
prw-r--r-- 1 bnc bnc 0 Nov 12 10:54 testfifo
```

The symbolic notation of the permissions `rw-r--r--` of the named pipe in the output of the command `ls` is represented in octal notation by the digits `644`. However, the function `mkfifo` in line 11 uses the permissions `0666` in octal notation. The different number of digits and the different values are due to the following reasons:

Die symbolische Notation der Zugriffsrechte `rw-r--r--` der benannten Pipe in der Ausgabe des Kommandos `ls` entspricht in Oktalnotation den Ziffern `644`. Beim Aufruf der Funktion `mkfifo` in Zeile 11 hingegen sind die Zugriffsrechte `0666` in Oktalnotation. Das die Anzahl der Stellen und die Werte verschieden sind, hat folgende Gründe:

- When using the four-digit octal notation for permissions, the leading digit (the first `0`) is a placeholder for the so-called sticky bit, the setgid bit, and the setuid bit. These additional attributes are rarely used and irrelevant in the context of Listing 9.6. For this reason, the meaning of the leading zero in the four-digit octal notation is ignored at this point.

- Bei der Oktalnotation der Zugriffsrechte mit vier Ziffern ist die erste Ziffern (die führende `0`) ein Platzhalter für das sogenannte Sticky-Bit, das Setgid Bit und das Setuid Bit. Diese erweiterten Dateirechte kommen eher selten zum Einsatz und spielen im Kontext von Listing 9.6 keine Rolle. Die Bedeutung der führenden Null bei der Oktalnotation mit vier Ziffern kann also hier ignoriert werden.

- The operating system removes (*masks*) the file mode creation mask set with `umask`. The default setting of `umask` depends on the operating system used and can be customized by the system administrator. For example, the `umask` default value for the Linux distribution Debian is `0022`, and for Ubuntu it is `0002`. The current file mode creation mask can be queried by executing the command `umask` without any parameters on the commandline.

- Das Betriebssystem entfernt die mit der Dateierzeugungsmaske `umask` gesetzten Zugriffsrechte (*maskiert*). Die Standardeinstellung von `umask` hängt vom verwendeten Betriebssystem ab und kann vom Systemadministrator verändert werden. Die `umask`-Standardwerte der Linux-Distributionen Debian und Ubuntu sind beispielsweise `0022` bzw. `0002`. Die aktuell eingestellte Dateierzeugungsmaske liefert ein Aufruf des Kommandos `umask` ohne Parameter in der Kommandozeile.

```
$ umask
0022
```

If `umask` has the value `0022`, the named pipe from Listing 9.6 has the permissions `rw-r--r--`, as in the output of `ls`. The calculation is as follows:

Permissions specified in mkfifo:	`rw-rw-rw-` (666)
Deduction by `umask`:	`----w--w-` (022)
Result (permission of the named pipe):	`rw-r--r--` (644)

On a system where `umask` is set to `0002`, the permissions of the named pipe would be `rw-rw-r--`.

In line 20, the program tries to create a child process with `fork`. If the process creation is successful, the parent process opens the pipe for writing with `open` (line 38) and writes a string to the write channel with `write` (line 41). Then, the parent process closes the pipe with `close` in line 44.

The child process opens the pipe for reading in line 59, also with the function `open`, and reads the string from the read channel with the function `read` (line 62). A receive buffer is required for reception, which is created in line 56 and has a capacity of 80 characters in this example. Then the child process returns the received string to the command line and closes the pipe with the function `close` in line 70. Finally, the child process erases the pipe with `unlink` in line 73.

Hat `umask` den Wert `0022`, erhält die benannte Pipe aus Listing 9.6 die Zugriffsrechte `rw-r--r--`, wie in der Ausgabe von `ls`. Die Berechnung ist wie folgt:

Auf einem System, bei dem `umask` den Wert `0002` hat, wären die Zugriffsrechte der benannten Pipe dementsprechend `rw-rw-r--`.

In Zeile 20 versucht das Programm, einen Kindprozess mit der Funktion `fork` zu erzeugen. War die Prozesserzeugung erfolgreich, öffnet der Elternprozess in Zeile 38 die Pipe für Schreibzugriffe mit der Funktion `open` und schreibt in Zeile 41 mit der Funktion `write` eine Zeichenkette in den Schreibkanal. Anschließend schließt der Elternprozess den Zugriff auf die Pipe mit der Funktion `close` in Zeile 44.

Der Kindprozess öffnet in Zeile 59 die Pipe für Lesezugriffe ebenfalls mit der Funktion `open` und liest in Zeile 62 mit der Funktion `read` die Zeichenkette aus dem Lesekanal. Zum Empfang ist ein Empfangspuffer nötig, der in Zeile 56 erzeugt wird und im Beispiel eine Kapazität von 80 Zeichen hat. Anschließend gibt der Kindprozess die empfangene Zeichenkette auf der Kommandozeile aus und schließt den Zugriff auf die Pipe mit der Funktion `close` in Zeile 70. Abschließend entfernt der Kindprozess die Pipe mit der Funktion `unlink` in Zeile 73.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <sys/stat.h>
6
7  void main() {
8    int pid_of_child;
9
10   // Create the pipe testfifo
11   if (mkfifo("testfifo",0666) < 0) {
12     printf("Unable to create the pipe.\n");
13     // Terminate the program (failure)
14     exit(1);
15   } else {
```

```
16      printf("The pipe testfifo has been created.\n");
17    }
18
19    // Create a child process
20    pid_of_child = fork();
21
22    // An error occured during fork
23    if (pid_of_child < 0) {
24      printf("An error occured during fork!\n");
25      // Terminate the program (failure)
26      exit(1);
27    }
28
29    // Parent process
30    if (pid_of_child > 0) {
31      printf("Parent process. PID: %i\n", getpid());
32
33      // Create the file descriptor (handle) for the pipe
34      int fd;
35      char message[] = "Test message";
36
37      // Open the pipe for writing
38      fd = open("testfifo", O_WRONLY);
39
40      // Write data into the pipe
41      write(fd, &message, sizeof(message));
42
43      // Close the pipe
44      close(fd);
45
46      exit(0);
47    }
48
49    // Child process
50    if (pid_of_child == 0) {
51      printf("Child process. PID: %i\n", getpid());
52
53      // Create the file descriptor (handle) for the pipe
54      int fd;
55      // Create a receive buffer
56      char buffer[80];
57
58      // Open the pipe for reading
59      fd = open("testfifo", O_RDONLY);
60
61      // Read data from the pipe
62      if (read(fd, buffer, sizeof(buffer)) > 0) {
63        // Return received data to the command line
64        printf("Data received: %s\n", buffer);
65      } else {
66        printf("Unable to read data from the pipe.\n");
```

```
67      }
68
69      // Close the pipe
70      close(fd);
71
72      // Erase the pipe
73      if (unlink("testfifo") < 0) {
74        printf("Unable to erase the pipe.\n");
75        // Terminate the program (failure)
76        exit(1);
77      } else {
78        printf("The pipe has been erased.\n");
79      }
80
81      exit(0);
82    }
83 }
```

Listing 9.6: Program Example for Named Pipes (FIFO)

Compiling the program with the GNU C compiler (`gcc`) in Linux and then running it afterward should produce the following output:

Das Übersetzen des Programms mit dem GNU C Compiler (`gcc`) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_6_named_pipe_fifo.c -o Listing_9_6_named_pipe_fifo
$ ./Listing_9_6_named_pipe_fifo
The pipe testfifo has been created.
Parent process. PID: 13807
Child process. PID: 13808
Data received: Test message
The pipe has been erased.
```

Size of anonymous and named Pipes in Linux

Because the operating system will block processes that try to write to a full pipe or read from an empty pipe, it is useful to be familiar with the sizes of anonymous and named pipes.

The standard size of anonymous pipes in modern Linux operating systems is 64 kB. With the system call `fcntl`, it is possible to query the default size (via parameter `F_GETPIPE_SZ`) or to change it dynamically at runtime (via parameter `F_SETPIPE_SZ`). The maximum size allowed for anonymous pipes is specified in the file `/proc/sys/fs/pipe-max-size`. In modern Linux operating systems, it is usually 1 MB. [60]

Größe anonymer und benannter Pipes in Linux

Da das Betriebssystem Prozesse blockiert, die versuchen in eine volle Pipe zu schreiben bzw. aus einer leeren Pipe zu lesen, ist es sinnvoll, die Größen anonymer und benannter Pipes zu kennen.

Die Standardgröße anonymer Pipes in modernen Linux-Betriebssystemen ist 64 kB. Mit dem Systemaufruf `fcntl` ist es möglich, die Standardgröße abzufragen (via Parameter `F_GETPIPE_SZ`) oder dynamisch während der Laufzeit zu ändern (via Parameter `F_SETPIPE_SZ`). Die maximal erlaubte Größe anonymer Pipes ist in der Datei `/proc/sys/fs/pipe-max-size` definiert und entspricht bei modernen Linux-Betriebssystemen meist 1 MB. [60]

The size of named pipes is typically 65536 bytes in a system with a page size of 4096 bytes. Unlike anonymous pipes, it is impossible to change the size dynamically.

The maximum number of bytes that can be written atomically to a named pipe is specified by the system variable `PIPE_BUF`. In Linux operating systems, it corresponds to the page size of the hardware architecture used.

Pipes in the Command Line Interface

On the command line, pipes can be inserted into command sequences with the vertical line `|`, the so-called *pipe character*. This character tells the operating system to use the output of the command to the left of the character as an input when executing the command to the right of the character. Such concatenations of commands are often used to filter the output with the `grep` command. An example is:

```
$ lsof | grep pipe
```

The `lsof` command returns a list of all open files in a Linux operating system, including the existing named pipes. The `grep pipe` command, which filters the output of `lsof`, returns only those lines from the output of `lsof` that contain the keyword `pipe`.

The steps taken by the operating system in the above example can be reconstructed in detail. In the Linux command-line shell, it is possible to create a named pipe using the command `mknod` [50]. The first of the three commands below creates a named pipe `fifoexample` in the current directory. The output of the `lsof` command is then redirected to the named pipe. It is essential that this process has the ampersand character `&` appended to it to keep it running in the background and not terminate it immediately. The process that starts the command must exist until the data has been read from the named pipe. The third command, `grep`, filters the contents of the pipe and returns only those lines on the command line that contain the search pattern `pipe`.

Die Größe benannter Pipes ist typischerweise 65536 Bytes in einem System mit 4096 Bytes Seitengröße. Im Gegensatz zu anonymen Pipes ist keine dynamische Änderung der Größe möglich.

Die maximale Anzahl von Bytes, die atomar in benannte Pipes geschrieben werden können, definiert die Systemvariable `PIPE_BUF` und entspricht in Linux-Betriebssystemen der Seitengröße der verwendeten Hardwarearchitektur.

Mit Pipes auf der Kommandozeile arbeiten

Auf der Kommandozeile können Pipes in Folgen von Kommandos mit dem senkrechten Strich, dem sogenannten *Pipe-Zeichen* `|`, realisiert werden. Dieses weist das Betriebssystem an, die Ausgabe des Kommandos links neben dem Zeichen als Eingabe bei der Ausführung des Kommandos rechts vom Zeichen zu verwenden. Häufig werden solche Verkettungen von Anweisungen zum Filtern von Ausgaben mit dem Kommando `grep` verwendet. Ein Beispiel ist:

Das Kommando `lsof` gibt in einem Linux-Betriebssystem eine Liste aller aktuell offenen Dateien, also auch die existierenden benannten Pipes aus. Durch das Kommando `grep pipe`, das die Ausgabe von `lsof` filtert, werden nur diejenigen Zeilen aus der Ausgabe von `lsof` ausgegeben, die das Schlüsselwort `pipe` enthalten.

Die einzelnen Schritte, die das Betriebssystem im obigen Beispiel abarbeitet, sind im Detail nachvollziehbar. In der Shell unter Linux ist es mit dem Kommando `mknod` möglich, eine benannte Pipe zu erzeugen [50]. Das erste Kommando unter den drei folgenden legt eine benannte Pipe `fifoexample` im aktuellen Verzeichnis an. Anschließend wird die Ausgabe des Kommandos `lsof` in die benannte Pipe umgeleitet. Wichtig ist, dass dieser Prozess mit dem *kaufmännischen Und* `&` in den Hintergrund geschickt und nicht sofort beendet wird. Der Prozess, den das Kommando startet, muss so lange existieren, bis die Daten aus der benannten Pipe ausgelesen sind. Das dritte Kommando `grep` filtert den Inhalt der Pipe und gibt nur die Zeilen mit dem Schlüsselwort `pipe` auf der Kommandozeile aus.

```
$ mknod fifoexample p
$ lsof > fifoexample &
$ grep pipe < fifoexample
```

9.3.6

Sockets

If communication not only between processes on one computer but across computer boundaries has to be implemented, *sockets* are the appropriate form of interprocess communication. A user mode process can request a socket from the operating system, and send and receive data via this socket afterward. The operating system maintains all used sockets and the related connection information. The individual processes of a computer are addressed using port numbers. These numbers are assigned during connection establishment.

Another positive attribute of sockets is that they enable communication between processes running in different operating systems.

Depending on the transport layer protocol used for interprocess communication, two types of sockets are possible: connectionless sockets and connection-oriented sockets.

Connectionless sockets, also called *datagram sockets*, use the User Datagram Protocol (UDP) transport layer protocol. One advantage of connectionless sockets is that UDP has less overhead than TCP. A drawback is that individual UDP segments can arrive in the wrong sequence or get lost. Segments are the messages used by transport layer protocols.

Connection-oriented sockets, also called *stream sockets*, use the transport layer protocol Transmission Control Protocol (TCP). It has better reliability than UDP at the cost of additional overhead, because it requests lost segments again, and ensures the correct order of the segments.

Figure 9.14 shows how interprocess communication with connection-oriented sockets works. The required functions are:

Sockets

Soll Kommunikation nicht nur zwischen Prozessen auf einem Computer, sondern über Rechnergrenzen hinweg möglich sein, sind *Sockets* die geeignete Form der Interprozesskommunikation. Ein Benutzerprozess kann einen Socket vom Betriebssystem anfordern, und über diesen anschließend Daten versenden und empfangen. Das Betriebssystem verwaltet alle benutzten Sockets und die zugehörigen Verbindungsinformationen. Die Adressierung der einzelnen Prozesse auf einem Computer geschieht mit Hilfe von Portnummern. Deren Vergabe erfolgt beim Verbindungsaufbau.

Eine weitere positive Eigenschaft von Sockets ist, dass sie Kommunikation zwischen Prozessen ermöglichen, die in verschiedenen Betriebssystemen laufen.

Je nach verwendetem Transportprotokoll, das zur Kommunikation zwischen den Prozessen verwendet wird, unterscheidet man verbindungslose und verbindungsorientierte Sockets.

Verbindungslose Sockets, die auch *Datagram Sockets* heißen, verwenden das Transportprotokoll User Datagram Protocol (UDP). Ein Vorteil verbindungsloser Sockets ist der geringere Verwaltungsaufwand (*Overhead*) von UDP im Vergleich zu TCP. Ein Nachteil ist, dass einzelne UDP-Segmente einander überholen oder verloren gehen können. Segmente sind die Nachrichteneinheiten, mit denen Transportprotokolle kommunizieren.

Verbindungsorientierte Sockets, die auch *Stream Sockets* heißen, verwenden das Transportprotokoll Transmission Control Protocol (TCP). Dieses bietet zum Preis des höheren Verwaltungsaufwands eine höhere Verlässlichkeit als UDP, da es verlorene Segmente neu anfordert und die korrekte Reihenfolge der Segmente sicherstellt.

Den Ablauf der Interprozesskommunikation mit verbindungsorientierten Sockets zeigt Abbildung 9.14. Die benötigten Funktionen sind:

- `socket` for creating a socket (sender and client both need a socket) on the local system
- `bind` for binding a socket at the server to a port number
- `listen` for making a socket on the server ready for reception by telling the operating system to set up a queue for connections with clients
- `accept` for accepting incoming connection requests from clients on the server
- `connect` for sending a connection request from the client to a server
- `send` and `recv` for exchanging data between client and server
- `close` for closing a socket on the local system

The program examples in the Listings 9.7 and 9.8 show how interprocess communication with sockets and the TCP transport protocol are possible when using Linux. The server program in Listing 9.7 creates a socket in line 34 with the function `socket`, which uses the transport layer protocol TCP (`SOCK_STREAM`) and the network layer protocol IPv4 (`AF_INET`). The return value of the function is the socket descriptor (`sd`), which is a positive integer.

Socket addresses are stored in the structure `sockaddr_in` (see line 15) when using the programming language C. It contains the variables `sin_family` for the address family, and `sin_port` for the port number, as well as the structure `sin_addr` for the address that contains a variable `s_addr`.

In line 43, the server assigns a port number to the socket with the function `bind`. This port number is specified as a parameter right after the file name (`argv[1]`) in the command line

- `socket` zum Erstellen eines Sockets (Sender und Client benötigen jeweils einen Socket) auf dem lokalen System.
- `bind`, um einen Socket auf Serverseite an eine Portnummer zu binden.
- `listen`, um einen Socket auf Serverseite empfangsbereit zu machen, indem das Betriebssystem eine Warteschlange für Verbindungen mit Clients einrichtet.
- `accept`, um auf Serverseite eintreffende Verbindungsanforderungen von Clients zu akzeptieren.
- `connect`, um vom Client eine Verbindungsanforderung an einen Server zu senden.
- `send` und `recv`, um Daten zwischen Client und Server auszutauschen.
- `close`, um einen Socket auf dem lokalen System zu schließen.

Die Programmbeispiele in Listing 9.7 und Listing 9.8 zeigen wie Interprozesskommunikation mit Sockets und dem Transportprotokoll TCP unter Linux möglich ist. Das Server-Programm in Listing 9.7 legt in Zeile 34 mit der Funktion `socket` einen Socket an, der das Transportprotokoll TCP (`SOCK_STREAM`) und das Vermittlungsprotokoll IPv4 (`AF_INET`) verwendet. Der Rückgabewert der Funktion ist der Socket-Deskriptor (`sd`), eine positive ganze Zahl.

Socket-Adressen werden in der Programmiersprache C in der Struktur `sockaddr_in` (siehe Zeile 15) gespeichert. Diese enthält die Variablen `sin_family` für die Adressfamilie und `sin_port` für die Portnummer sowie die Struktur `sin_addr` für die Adresse, die wiederum eine Variable `s_addr` enthält.

In Zeile 43 verknüpft der Server mit der Funktion `bind` den Socket mit einer Portnummer. Diese wird als Parameter direkt nach dem Dateinamen (`argv[1]`) beim Start des Programms

when the program is started, and it is stored in line 26 in the variable `portnumber`. The function `atoi` in line 26 converts the port number, which is a string, into the data type integer.

Setting up a queue for incoming connection requests is done with the function `listen` in line 51. The second parameter next to the socket descriptor is the maximum number of connections waiting (for those, the `accept` function has not yet been called). In our example, the queue can hold up to five connection requests. Once `listen` has been executed, the socket *listens* on the port. It waits for incoming connection requests.

The function `accept` in line 58 fetches the first connection request in the queue. The return value in the variable `new_socket` is the socket descriptor of the new socket. If the queue contains no connection requests, the process is blocked until a connection request is received. Once a connection request has been accepted with `accept`, the connection to the client is established.

In line 70, the server reads a message from the new socket with the function `read` and writes it into a buffer (`buffer`), which was created in line 17 and filled with zero bytes. After the contents of the buffer have been printed out on the command line in line 76, a message is sent back to the client for verification. It is done with the function `write` in line 81 that writes the message into the new socket.

In the lines 87 and 92, the new socket and the *listening* socket are terminated with `close`.

in der Kommandozeile angegeben und in Zeile 26 in der Variable `portnumber` gespeichert. Die Funktion `atoi` in Zeile 26 konvertiert die Portnummer, die als Zeichenkette vorliegt, in den Datentyp Integer.

Das Einrichten einer Warteschlange für eintreffende Verbindungsanforderungen geschieht mit der Funktion `listen` in Zeile 51. Der zweite Parameter neben dem Socket-Deskriptor ist die maximale Anzahl wartender Verbindungen (für die `accept` noch nicht aufgerufen wurde). In unserem Beispiel kann die Warteschlange fünf Verbindungsanforderungen aufnehmen. Sobald `listen` ausgeführt wurde, *lauscht* der Socket am Port. Er wartet auf eintreffende Verbindungsanforderungen.

Die erste Verbindungsanforderung in der Warteschlange holt die Funktion `accept` in Zeile 58. Der Rückgabewert in der Variable `new_socket` ist der Socket-Deskriptor des neuen Sockets. Enthält die Warteschlange keine Verbindungsanforderungen, ist der Prozess blockiert, bis eine Verbindungsanforderung eintrifft. Nachdem eine Verbindungsanforderung mit `accept` angenommen wurde, ist die Verbindung mit dem Client vollständig aufgebaut.

In Zeile 70 liest der Server mit der Funktion `read` eine Nachricht aus dem neuen Socket und schreibt diese in einen Puffer (`buffer`), der in Zeile 17 erzeugt und mit Null-Bytes gefüllt wurde. Nachdem der Inhalt des Puffers auf der Kommandozeile in Zeile 76 ausgegeben wurde, wird an den Client zur Bestätigung eine Nachricht zurückgesendet. Dies geschieht, indem das Programm mit der Funktion `write` in Zeile 81 die Nachricht in den neuen Socket schreibt.

In den Zeilen 87 und 92 werden der neue Socket und der *lauschende* Socket mit `close` beendet.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[])
10 {
11   int sd;
12   int new_socket;
```

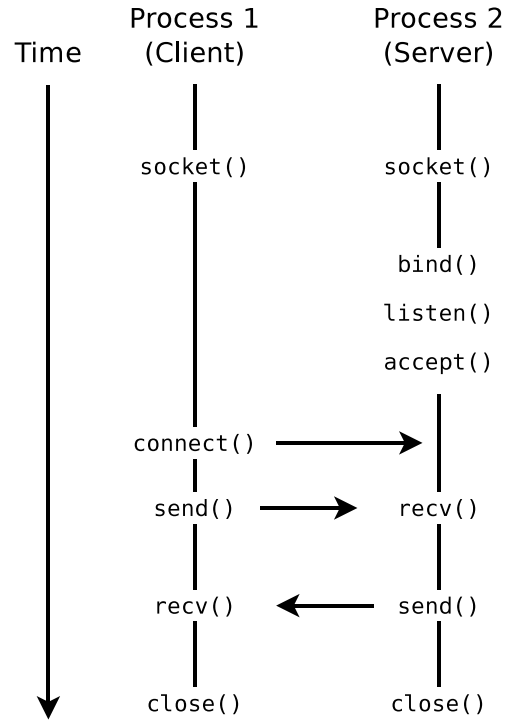

Figure 9.14: Connectionless Communication via Sockets and the Transport Layer Protocol TCP

```
13  int portnumber;
14  int addrlen;
15  struct sockaddr_in addr;
16
17  char buffer[1024] = { 0 };
18
19  // The port number must be specified as an argument
20  if (argc < 2) {
21    printf("IP-Address and/or port number are/is missing.\n");
22    exit(1);
23  }
24
25  // The argument following the file name is the port number
26  portnumber = atoi(argv[1]);
27
28  // Store the socket address in the structure  sockaddr_in
29  addr.sin_family = AF_INET;
30  addr.sin_addr.s_addr = INADDR_ANY;
31  addr.sin_port = htons(portnumber);
32
33  // Create socket
34  sd = socket(AF_INET, SOCK_STREAM, 0);
```

```
35  if (sd < 0) {
36    printf("Unable to create the socket.\n");
37    exit(1);
38  } else {
39    printf("The socket has been created.\n");
40  }
41
42  // Bind socket to a port number
43  if (bind(sd, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
44    printf("The port is not available.\n");
45    exit(1);
46  } else {
47    printf("The socket has been bound to the port number.\n");
48  }
49
50  // Create a queue for connection requests
51  if (listen(sd, 5) == 0) {
52    printf("Waiting for connection requests.\n");
53  } else {
54    printf("An error occurred during the listen function.\n");
55  }
56
57  addrlen = sizeof(addr);
58  new_socket = accept(sd, (struct sockaddr *) &addr, &addrlen);
59  if (new_socket < 0) {
60    printf("Connection request failed.\n");
61    exit(1);
62  } else {
63    printf("Connection to a client established.\n");
64  }
65
66  // Fill the contents of the buffer with zero bytes
67  memset(buffer, 0, sizeof(buffer));
68
69  // Receive message
70  if (read(new_socket, buffer, sizeof(buffer)) < 0) {
71    printf("The read operation has failed.\n");
72    exit(1);
73  }
74
75  // Print the received message
76  printf("Received message: %s\n", buffer);
77
78  char reply[]="Server: Message received.\n";
79
80  // Send message
81  if (write(new_socket, reply, sizeof(reply)) < 0) {
82    printf("The write operation has failed.\n");
83    exit(1);
84  }
85
```

```
86   // Close the socket
87   if (close(new_socket) == 0) {
88     printf("The connected socket has been closed.\n");
89   }
90
91   // Close the socket
92   if (close(sd) == 0) {
93     printf("The socket has been closed.\n");
94   }
95
96   exit(0);
97 }
```

Listing 9.7: Program Example for TCP Sockets (Server)

The program example in Listing 9.8 implements a client that can exchange data with the server of Listing 9.7.

The client program also creates a socket in line 28 with the function `socket` that uses the transport layer protocol TCP (`SOCK_STREAM`) and the network layer protocol IPv4 (`AF_INET`).

The IP address is specified in the example as a parameter right after the file name (`argv[1]`) in the command line when the program is started. The second parameter, after the file name (`argv[2]`), contains the port number of the server.

In the lines 40-42, the program stores the address family, IP address, and port number of the server in the variable `addr`, which represents the structure `sockaddr_in`.

With `connect` in line 45, the client sends a connection request to the server. If the return value of the function is `0`, the connection request was successful.

In line 57, a string is read at the command line interface and stored in the variable `buffer`, which has previously been filled with zero bytes in line 16. The client program writes the contents of the variable `buffer` in line 60 into the new socket with `write`, and sends it to the server.

After the content of the variable in line 66 has been overwritten again with zero bytes, the client program receives a message from the server in line 69 with `read`, stores it in the variable `buffer`, and returns it to the command line.

Das Programmbeispiel in Listing 9.8 realisiert einen Client, der mit dem Server in Listing 9.7 Daten austauschen kann.

Auch das Client-Programm erzeugt in Zeile 28 mit der Funktion `socket` einen Socket, der das Transportprotokoll TCP (`SOCK_STREAM`) und das Vermittlungsprotokoll IPv4 (`AF_INET`) verwendet.

Die IP-Adresse wird im Beispiel als Parameter direkt nach dem Dateinamen (`argv[1]`) beim Start des Programms in der Kommandozeile angegeben. Der zweite Parameter nach dem Dateinamen (`argv[2]`) enthält die Portnummer des Servers.

In den Zeilen 40-42 speichert das Programm die Adressfamilie, IP-Adresse und Portnummer des Servers in der Variable `addr`, die der Struktur `sockaddr_in` entspricht.

Mit `connect` in Zeile 45 sendet der Client eine Verbindungsanforderung an den Server. Wenn der Rückgabewert der Funktion dem Wert `0` entspricht, war die Verbindungsanforderung erfolgreich.

In der Zeile 57 wird eine Zeichenkette auf der Eingabeaufforderung eingelesen und in der Variable `buffer` gespeichert, die zuvor in Zeile 16 mit Null-Bytes gefüllt wurde. Den Inhalt der Variable `buffer` schreibt die Client-Anwendung in Zeile 60 mit `write` in den neuen Socket und sendet ihn damit an den Server.

Nachdem der Inhalt der Variable in Zeile 66 erneut mit Null-Bytes überschrieben wurde, empfängt die Client-Anwendung in Zeile 69 mit `read` eine Nachricht des Servers, speichert sie in der Variable `buffer` und gibt sie auf der Eingabeaufforderung aus.

Finally, the function `close` closes the socket in line 77.

Abschließend beendet die Funktion `close` in Zeile 77 den Socket.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[])
10 {
11   int sd;
12   int portnumber;
13   struct sockaddr_in addr;
14
15   // Fill the buffer with zero bytes
16   char buffer[1024] = { 0 };
17
18   // If hostname and/or port number are missing...
19   if (argc < 3) {
20     printf("Host name and/or port number are missing.\n");
21     exit(1);
22   }
23
24   // Second argument after the file name = port number
25   portnumber = atoi(argv[2]);
26
27   // Create socket
28   sd = socket(AF_INET, SOCK_STREAM, 0);
29   if (sd < 0) {
30     printf("Unable to create the socket.\n");
31     exit(1);
32    } else {
33     printf("The socket has been created.\n");
34   }
35
36   // Fill the memory area of sockaddr_in with zeros
37   memset(&addr, 0, sizeof(addr));
38
39   // Store the socket address in the structure sockaddr_in
40   addr.sin_family = AF_INET;
41   addr.sin_port = htons(portnumber);
42   addr.sin_addr.s_addr = inet_addr(argv[1]);
43
44   // Send a connection request to the server
45   if (connect(sd,
46               (struct sockaddr *) &addr,
47               sizeof(addr)) < 0) {
48     printf("Connection request failed.\n");
49     exit(1);
```

```
50   } else {
51     printf("Connection to the server established.\n");
52   }
53
54   printf("Please enter the message: ");
55
56   // Read the message from the command line interface
57   fgets(buffer, sizeof(buffer), stdin);
58
59   // Send message
60   if (write(sd, buffer, strlen(buffer)) < 0) {
61     printf("The write operation has failed.\n");
62     exit(1);
63   }
64
65   // Fill the buffer with zero bytes
66   memset(buffer, 0, sizeof(buffer));
67
68   // Receive message
69   if (read(sd, buffer, sizeof(buffer)) < 0) {
70     printf("The read operation has failed.\n");
71     exit(1);
72   } else {
73     printf("%s\n",buffer);
74   }
75
76   // Close the socket
77   if (close(sd) == 0) {
78     printf("The socket has been closed.\n");
79   }
80
81   exit(0);
82 }
```

Listing 9.8: Program Example for TCP Sockets (Client)

Compiling the program with the GNU C compiler (gcc) in Linux and running it afterward should result in the following output:

Das Übersetzen der Programme mit dem GNU C Compiler (gcc) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe auf dem Server:

```
$ gcc Listing_9_7_tcp_socket_server.c -o Listing_9_7_tcp_socket_server
$ ./Listing_9_7_tcp_socket_server 50003
The socket has been created.
The socket has been bound to the port number.
Waiting for connection requests.
Connection to a client established.
Received message: This is a test message.

The connected socket has been closed.
The socket has been closed.
```

The output on the client is as follows:

Die Ausgabe auf dem Client ist wie folgt:

```
$ gcc Listing_9_8_tcp_socket_client.c -o Listing_9_8_tcp_socket_client
$ ./Listing_9_8_tcp_socket_client 127.0.0.1 50003
The socket has been created.
Connection to the server established.
Please enter the message: This is a test message.
Server: Message received.

The socket has been closed.
```

The output of the client and server shows that both processes were running in the same operating system, and therefore on the same computer system.

During the development of a server such as in Listing 9.7, tools for testing are handy because a client may not yet exist, or its proper functioning cannot yet be guaranteed. In such a case, classic command-line tools such as the telnet client or the more modern tool Netcat, also called `nc`, are helpful. The following output shows how how to use `nc` with the IP address and port number as command line parameters. Additional information about the connection and the amount of data transferred is obtained using the command line parameters `-v` or `-vv`.

Anhand der Ausgabe von Client und Server ist ersichtlich, dass sich bei der Ausführung beide Prozesse im gleichen Betriebssystem und damit auf dem gleichen Computersystem befanden.

Während der Entwicklung eines Servers wie in Listing 9.7, sind Werkzeuge zum Testen hilfreich, da ein Client vielleicht noch nicht existiert oder dessen korrekte Funktion noch nicht garantiert werden kann. In einem solchen Fall sind klassische Kommandozeilenwerkzeuge wie der Telnet-Client oder besser das modernere Werkzeug Netcat, auch `nc` genannt, hilfreich. Die folgende Ausgabe zeigt den Aufruf von `nc` mit der IP-Adresse und Portnummer als Kommandozeilenparametern. Zusätzliche Informationen zur Verbindung und zu den übertragenen Datenmengen liefern die Kommandozeilenparameter `-v` bzw. `-vv`.

```
$  nc 127.0.0.1 50003
This is a test message.
Server: Message received.
```

Another helpful command-line tool for this purpose is `netstat` (*network statistics*). This tool allows monitoring of the open ports on the local system and the network connections to remote machines. The following output from `netstat` shows not only the TCP socket of the server waiting for connection requests but also the active connection between the server and the netcat client `nc`.

Ein weiteres hilfreiches Kommandozeilenwerkzeug in diesem Zusammenhang ist `netstat` (*network statistics*). Dieses Werkzeug ermöglicht es zu überprüfen, welche Ports auf dem lokalen System geöffnet sind und welche Netzwerkverbindungen zu entfernten Rechnern bestehen. Die folgende Ausgabe von `netstat` zeigt nicht nur den TCP-Socket des auf Verbindungsanforderungen wartenden Servers, sondern auch die aktive Verbindung zwischen dem Server und dem netcat-Client `nc`.

```
$ netstat -tap | grep 50003
tcp  0 0  0.0.0.0:50003    0.0.0.0:*        LISTEN       110353/./Listing_9_
tcp  0 0  localhost:53394  localhost:50003  ESTABLISHED  110385/nc
tcp  0 0  localhost:50003  localhost:53394  ESTABLISHED  110353/./Listing_9_
```

Communication flow with connectionless sockets (see Figure 9.15) differs slightly from connection-oriented sockets. The program examples in Listing 9.9 and Listing 9.10 demonstrate how interprocess communication with sockets using the UDP transport layer protocol works in Linux. For the server in Listing 9.9, the relevant differences to the TCP server in Listing 9.7 are the absence of the `listen` and `accept` functions, and the use of the functions `sendto` and `recvfrom` instead of `send` and `recv`.

Der Ablauf der Kommunikation mit verbindungslosen Sockets (siehe Abbildung 9.15) unterscheidet sich nur geringfügig von verbindungsorientierten Sockets. Die Programmbeispiele in Listing 9.9 und Listing 9.10 zeigen wie Interprozesskommunikation mit Sockets und dem Transportprotokoll UDP unter Linux möglich ist. Beim Server in Listing 9.9 sind die relevanten Unterschiede zum TCP-Server in Listing 9.7 der Wegfall der Funktionen `listen` und `accept` sowie die Verwendung der Funktionen `sendto` und `recvfrom` anstatt `send` und `recv`.

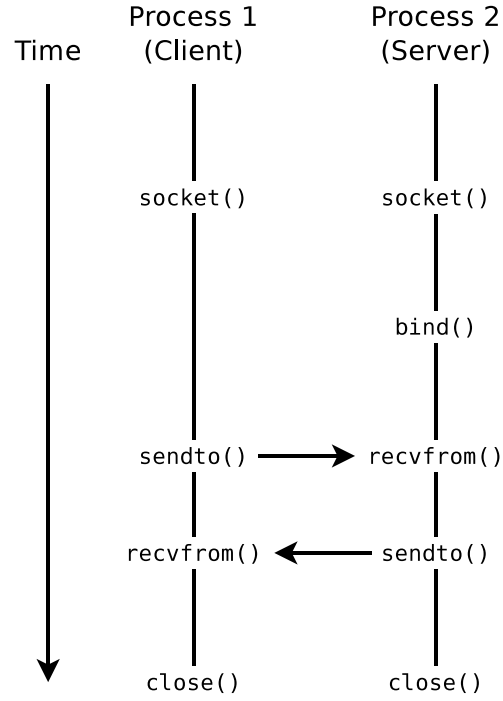

Figure 9.15: Connection-oriented Communication via Sockets and the Transport Layer Protocol UDP

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[])
10 {
```

```
11  int sd;
12  int portnumber;
13  int addrlen;
14  struct sockaddr_in addr, client_addr;
15
16  // Fill the buffer with zero bytes
17  char buffer[1024] = { 0 };
18
19  // The port number must be specified as an argument
20  if (argc < 2) {
21    printf("The port number is missing.\n");
22    exit(1);
23  }
24
25  // The argument following the file name is the port number
26  portnumber = atoi(argv[1]);
27
28  // Fill the memory area of sockaddr_in with zeros
29  memset(&addr, 0, sizeof(addr));
30  // Fill the memory area of sockaddr_in with zeros
31  memset(&client_addr, 0, sizeof(client_addr));
32
33  // Store the socket address in the structure sockaddr_in
34  addr.sin_family = AF_INET;
35  addr.sin_addr.s_addr = INADDR_ANY;
36  addr.sin_port = htons(portnumber);
37
38  // Create socket
39  sd = socket(AF_INET, SOCK_DGRAM, 0);
40  if (sd < 0) {
41    printf("Unable to create the socket.\n");
42    exit(1);
43  } else {
44    printf("The socket has been created.\n");
45  }
46
47  // Bind socket to a port number
48  if (bind(sd,
49           (struct sockaddr *) &addr,
50           sizeof(addr)) < 0) {
51    printf("The port is not available.\n");
52    exit(1);
53  } else {
54    printf("The socket has been bound to the port number.\n");
55  }
56
57  addrlen = sizeof(client_addr);
58
59  // Receive message
60  if (recvfrom(sd,
61               (char *)buffer,
```

```
62                sizeof(buffer),
63                0,
64                (struct sockaddr *) &client_addr,
65                &addrlen) < 0) {
66     printf("The read operation has failed.\n");
67     exit(1);
68   }
69
70   // Print the received message
71   printf("Received message: %s\n",buffer);
72
73   char reply[]="Server: Message received.\n";
74
75   // Send message
76   if (sendto(sd,
77              (const char *)reply,
78              sizeof(reply),
79              0,
80              (struct sockaddr *) &client_addr,
81              addrlen) < 0) {
82     printf("The write operation has failed.\n");
83     exit(1);
84   }
85
86   // Close the socket
87   if (close(sd) == 0) {
88     printf("The socket has been closed.\n");
89   }
90
91   exit(0);
92 }
```

Listing 9.9: Program Example for UDP-Sockets (Server)

For the client in Listing 9.10, the relevant differences to the TCP client in Listing 9.8 are the absence of the function `connect` for establishing a connection and the use of the functions `sendto` and `recvfrom` instead of `send` and `recv`.

Beim Client in Listing 9.10 sind die relevanten Unterschiede zum TCP-Client in Listing 9.8 der Wegfall der Funktion `connect` zum Verbindungsaufbau sowie die Verwendung der Funktionen `sendto` und `recvfrom` anstatt `send` und `recv`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 int main(int argc, char *argv[])
10 {
11   int sd;
12   int portnumber;
```

```
13  int addrlen;
14  struct sockaddr_in addr;
15
16  // Fill the buffer with zero bytes
17  char buffer[1024] = { 0 };
18
19  // If hostname and/or port number are missing...
20  if (argc < 3) {
21    printf("Host name and/or port number are missing.\n");
22    exit(1);
23  }
24
25  // Second argument after the file name = port number
26  portnumber = atoi(argv[2]);
27
28  // Create socket
29  sd = socket(AF_INET, SOCK_DGRAM, 0);
30  if (sd < 0) {
31    printf("Unable to create the socket.\n");
32    exit(1);
33  } else {
34    printf("The socket has been created.\n");
35  }
36
37  // Fill the memory area of sockaddr_in with zeros
38  memset(&addr, 0, sizeof(addr));
39
40  // Store the socket address in the structure sockaddr_in
41  addr.sin_family = AF_INET;
42  addr.sin_port = htons(portnumber);
43  addr.sin_addr.s_addr = inet_addr(argv[1]);
44
45  printf("Please enter the message: ");
46
47  // Read the message from the command line interface
48  fgets(buffer, sizeof(buffer), stdin);
49
50  addrlen = sizeof(addr);
51
52  // Send message
53  if (sendto(sd,
54             (const char *)buffer,
55             strlen(buffer),
56             0,
57             (struct sockaddr *) &addr,
58             addrlen) < 0) {
59    printf("The write operation has failed.\n");
60    exit(1);
61  }
62
63  // Fill the buffer with zero bytes
```

```
64   memset(buffer, 0, sizeof(buffer));
65
66   // Receive message
67   if (recvfrom(sd,
68                (char *)buffer,
69                sizeof(buffer),
70                0,
71                (struct sockaddr *) &addr,
72                &addrlen) < 0) {
73     printf("The read operation has failed.\n");
74     exit(1);
75   } else {
76     printf("%s\n",buffer);
77   }
78
79   // Close the socket
80   if (close(sd) == 0) {
81     printf("The socket has been closed.\n");
82   }
83
84   exit(0);
85 }
```

Listing 9.10: Program Example for UDP-Sockets (Client)

Compiling the program with the GNU C compiler (gcc) in Linux and running it afterward should create the following output at the server site:

Das Übersetzen der Programme mit dem GNU C Compiler (gcc) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe auf dem Server:

```
$ gcc Listing_9_9_udp_socket_server.c -o Listing_9_9_udp_socket_server
$ ./Listing_9_9_udp_socket_server 50004
The socket has been created.
The socket has been bound to the port number.
Received message: This is a test message.

The socket has been closed.
```

The output on the client is as follows:

Die Ausgabe auf dem Client ist wie folgt:

```
$ gcc Listing_9_10_udp_socket_client.c -o Listing_9_10_udp_socket_client
$ ./Listing_9_10_udp_socket_client 127.0.0.1 50004
The socket has been created.
Please enter the message: This is a test message.
Server: Message received.

The socket has been closed.
```

The output of the client and server shows that both processes were running in the same operating system, and therefore on the same computer system.

Anhand der Ausgabe von Client und Server ist ersichtlich, dass sich bei der Ausführung beide Prozesse im gleichen Betriebssystem und damit auf dem gleichen Computersystem befanden.

If the server from Listing 9.9 is running, it is possible to work with it or perform some tests using tools such as Netcat (`nc`), in the same way as with the TCP server from Listing 9.7. The command-line parameter `-u` specifies that the transport layer protocol UDP will be used for the communication.

Läuft der Server aus Listing 9.9, kann genau wie beim TCP-Server aus Listing 9.7 mit geeigneten Werkzeugen wie Netcat (`nc`) mit diesem gearbeitet werden bzw. dieser getestet werden. Der Kommandozeilenparameter `-u` legt fest, das via Transportprotokoll UDP kommuniziert werden soll.

```
$  nc -u 127.0.0.1 50004
This is a test message.
Server: Message received.
```

Again, the command-line tool `netstat` is handy at this point because it allows monitoring the open UDP ports via the command-line parameter `-u`. The following output from `netstat` shows not only the UDP socket of the server but also the socket of the netcat client `nc` that is interacting with the server.

Auch an dieser Stelle ist das Kommandozeilenwerkzeug `netstat` hilfreich, denn es ermöglicht auch die Kontrolle der geöffneten UDP-Ports via Kommandozeilenparameter `-u`. Die folgende Ausgabe des Aufrufs von `netstat` zeigt nicht nur den UDP-Socket des Servers, sondern auch den Socket des netcat-Client `nc`, der mit dem Server interagiert.

```
$ netstat -tapu | grep 50004
udp  0 0  localhost:48274  localhost:50004  ESTABLISHED  127343/nc
udp  0 0  0.0.0.0:50004    0.0.0.0:*                     127279/./Listing_9_
```

The UDP and TCP examples in this section implement *non-blocking sockets*. In many everyday situations, however, *blocking sockets* is the more practical alternative. Examples include web servers or chat servers that require parallel handling of connections to multiple clients. A blocking socket would block the process until an operation such as establishing a connection or sending or receiving data is completed [120]. Sockets are blocking by default. With the function `fcntl` for querying and modifying file descriptors and the parameter `O_NONBLOCK` a socket is declared as non-blocking.

Die in diesem Abschnitt enthaltenen Beispiele zu UDP und TCP implementieren *nichtblockierende Sockets*. In vielen alltäglichen Situationen sind aber *blockierende Sockets* die sinnvollere Alternative. Beispiele sind Webserver oder Chat-Server, die Verbindungen zu mehreren Clients parallel verwalten müssen. Ein blockierender Socket würde dazu führen, dass der jeweilige Prozess blockiert ist, bis eine Operation wie zum Beispiel der Aufbau einer Verbindung oder das Senden bzw. der Empfang von Daten abgeschlossen ist [120]. Standardmäßig sind Sockets blockierend. Mit der Funktion `fcntl` zur Abfrage und Änderung von Dateideskriptoren und dem Parameter `O_NONBLOCK` wird ein Socket als nicht-blockierend definiert.

Once a socket becomes non-blocking, the behavior of the `connect`, `accept`, `send`, and `recv` functions for TCP sockets and the `sendto` and `recvfrom` functions for UDP sockets changes. [126]

Sobald ein Socket nicht-blockierend ist, ändert sich das Verhalten der in diesem Abschnitt besprochenen Funktionen `connect`, `accept`, `send` und `recv` bei TCP-Sockets sowie von `sendto` und `recvfrom` bei UDP-Sockets. [126]

- Function `connect` executed on a...
  - blocking TCP socket: The process waits until the connection is established.

- Funktion `connect` ausgeführt auf einen...
  - blockierenden TCP-Socket: Der Prozess wartet bis die Verbindung aufgebaut ist.

  - non-blocking TCP socket: The error message `EINPROGRESS` is returned, and the process continues to work. However, the process must check whether the connection has been successfully established before sending data via the socket (e.g., with the `select` function).

  - nicht-blockierenden TCP-Socket: Die Fehlermeldung `EINPROGRESS` wird zurückgegeben und der Prozess arbeitet weiter. Der Prozess muss allerdings vor dem Senden von Daten über den Socket prüfen (z.B. mit der Funktion `select`), ob die Verbindung erfolgreich aufgebaut ist.

- Function `accept` executed on a...
  - blocking TCP socket: The process waits until there is a connection request in the queue that can be accepted.
  - non-blocking TCP socket: If no connection request is present, the error message `EAGAIN` or `EWOULDBLOCK` is returned immediately. The process must check again later whether there is a connection request and invoke `accept`

- Funktion `accept` ausgeführt auf einen...
  - blockierenden TCP-Socket: Der Prozess wartet bis eine Verbindungsanforderung in der Warteschlange ist, die akzeptiert werden kann.
  - nicht-blockierenden TCP-Socket: Beim Fehlen einer Verbindungsanfrage wird sofort die Fehlermeldung `EAGAIN` oder `EWOULDBLOCK` zurückgegeben. Der Prozess muss zu einem späteren Zeitpunkt erneut prüfen, ob eine Verbindungsanforderung vorliegt und `accept` aufrufen.

- Function `send` executed on a...
  - blocking TCP socket: The process is blocked until the receiver acknowledges the transmission. If the transmission was successful, the return value of `send` corresponds to the number of bytes transmitted. If the send buffer of the network stack does not have enough capacity, `send` blocks the process until there is enough free capacity.
  - non-blocking TCP socket: The process continues to run. If the send buffer capacity is insufficient, the error message `EAGAIN` or `EWOULDBLOCK` is returned immediately. The process must check again later whether the transmission is possible.

- Funktion `send` ausgeführt auf einen...
  - blockierenden TCP-Socket: Der Prozess ist solange blockiert, bis die Übertragung vom Empfänger bestätigt ist. Bei einer erfolgreichen Übertragung entspricht der Rückgabewert von `send` der Anzahl der übertragenen Bytes. Wenn der Sendepuffer des Netzwerk-Stacks keine ausreichende Kapazität vorweist, blockiert `send` den Prozess solange, bis ausreichend freie Kapazität vorliegt.
  - nicht-blockierende TCP-Socket: Der Prozess arbeitet sofort weiter. Wenn der Sendepuffer keine ausreichende Kapazität vorweist, wird sofort die Fehlermeldung `EAGAIN` oder `EWOULDBLOCK` zurückgegeben. Der Prozess muss zu einem späteren Zeitpunkt erneut prüfen, ob die Übertragung möglich ist.

- Function `recv` executed on a...
  - blocking TCP socket: The process waits for data in the receive buffer

- Funktion `recv` ausgeführt auf einen...
  - blockierenden TCP-Socket: Der Prozess wartet auf Daten im Empfangs-

and is blocked until data is available. If the transfer was successful, the return value of `recv` corresponds to the number of bytes received.

– non-blocking TCP socket: On successful transmission, the return value of `recv` corresponds to the number of bytes received. If the receive buffer does not contain any data that can be read now, the error message `EAGAIN` or `EWOULDBLOCK` is returned immediately. The process can check again later whether there is data in the receive buffer.

puffer und ist solange blockiert. Bei einer erfolgreichen Übertragung entspricht der Rückgabewert von `recv` der Anzahl der empfangenen Bytes.

– nicht-blockierende TCP-Socket: Bei einer erfolgreichen Übertragung entspricht der Rückgabewert von `recv` der Anzahl der empfangenen Bytes. Wenn der Empfangspuffer keine Daten enthält, die zum jetzigen Zeitpunkt gelesen werden können, wird sofort die Fehlermeldung `EAGAIN` oder `EWOULDBLOCK` zurückgegeben. Der Prozess kann zu einem späteren Zeitpunkt erneut prüfen, ob Daten im Empfangspuffer vorliegen.

- Function `sendto` executed on a...
  – blocking UDP socket: Since UDP does not implement any acknowledgment from the receiver for receiving a transmission, calling `sendto` may only block the process in rare cases. If the send buffer of the network stack has insufficient capacity, `sendto` blocks the process until there is sufficient free capacity.
  – non-blocking UDP socket: The process continues to run. If the send buffer has insufficient capacity, the error message `EAGAIN` or `EWOULDBLOCK` is returned immediately. The process must check again later whether the transmission is possible.

- Funktion `sendto` ausgeführt auf einen...
  – blockierenden UDP-Socket: Da UDP keine Bestätigung für den Empfang vom Empfänger vorsieht, kann ein Aufruf von `sendto` nur in seltenen Fällen den Prozess blockieren. Wenn der Sendepuffer des Netzwerk-Stacks keine ausreichende Kapazität vorweist, blockiert `sendto` den Prozess solange, bis ausreichend freie Kapazität vorliegt.
  – nicht-blockierende UDP-Socket: Der Prozess arbeitet sofort weiter. Wenn der Sendepuffer keine ausreichende Kapazität vorweist, wird sofort die Fehlermeldung `EAGAIN` oder `EWOULDBLOCK` zurückgegeben. Der Prozess muss zu einem späteren Zeitpunkt erneut überprüfen, ob die Übertragung möglich ist.

- Function `recvfrom` executed on a...
  – blocking UDP socket: The process waits for data in the receive buffer and is blocked until data is available. If the transfer was successful, the return value of `recvfrom` corresponds to the number of bytes received.
  – non-blocking UDP socket: On a successful transmission, the return value

- Funktion `recvfrom` ausgeführt auf einen...
  – blockierenden UDP-Socket: Der Prozess wartet auf Daten im Empfangspuffer und ist solange blockiert. Bei einer erfolgreichen Übertragung entspricht der Rückgabewert von `recvfrom` der Anzahl der empfangenen Bytes.
  – nicht-blockierende UDP-Socket: Bei einer erfolgreichen Übertra-

of `recvfrom` corresponds to the number of bytes received. If the receive buffer does not contain any data that can be read now, the error message `EAGAIN` or `EWOULDBLOCK` is returned immediately. The process can check again later whether there is data in the receive buffer.

gung entspricht der Rückgabewert von `recvfrom` der Anzahl der empfangenen Bytes. Wenn der Empfangspuffer keine Daten enthält, die zum jetzigen Zeitpunkt gelesen werden können, wird sofort die Fehlermeldung `EAGAIN` oder `EWOULDBLOCK` zurückgegeben. Der Prozess kann zu einem späteren Zeitpunkt erneut prüfen, ob Daten im Empfangspuffer vorliegen.

9.4

Cooperation of Processes

Critical sections can be protected using the concept of locking/blocking (see Section 9.2.2) so that there is no overlap in the execution. Two more sophisticated concepts that allow multiple processes working together are semaphores and mutexes.

Kooperation von Prozessen

Der Schutz kritischer Abschnitte, so dass es keine Überlappung in der Ausführung gibt, ist mit dem einfachen Konzept der Sperren möglich (siehe Abschnitt 9.2.2). Zwei komplexere Konzepte, die die Kooperation mehrerer Prozesse ermöglichen, sind Semaphoren und Mutexe.

9.4.1

Semaphore according to Dijkstra

For securing critical sections, not only the locking/blocking (see Section 9.2.2) concept exists, but also the *semaphore* [33] concept, that was developed by Edsger Dijkstra in the 1960s. In the German literature, semaphores are rarely also called *coordination variables* [6, 10].

A semaphore is an integer, non-negative counter variable. Unlike locks, which only allow for a single process to enter the critical section at a time, a semaphore may allow multiple processes to enter the critical section. Processes that are waiting to pass the semaphore are in `blocked` state and wait to be transferred into `ready` state by the operating system when the semaphore allows access to the critical Section [119].

The semaphore concept includes the two access operations P and V, shown in Figure 9.16. These are atomic, i.e. cannot be interrupted. The P operation checks the value of the counter variable and attempts to reduce its value. If the

Semaphor nach Dijkstra

Zur Sicherung kritischer Abschnitte kann außer den bekannten Sperren (siehe Abschnitt 9.2.2) auch das bereits in den 1960er Jahren von Edsger Dijkstra entwickelte Konzept der *Semaphoren* [33] eingesetzt werden. In der deutschsprachigen Literatur heißen Semaphoren selten auch *Koordinationsvariablen* [6, 10].

Eine Semaphore ist eine ganzzahlige, nichtnegative Zählersperre. Im Gegensatz zu Sperren, die immer nur einem Prozess das Betreten des kritischen Abschnitts erlauben können, ist es mit einer Semaphore möglich, mehreren Prozessen das Betreten des kritischen Abschnitts zu erlauben. Prozesse, die darauf warten, die Semaphore passieren zu dürfen, sind im Zustand `blockiert` und warten darauf, vom Betriebssystem in den Zustand `bereit` überführt zu werden, wenn die Semaphore den Weg freigibt [119].

Das Konzept der Semaphoren beinhaltet die beiden in Abbildung 9.16 dargestellten Zugriffsoperation P und V. Diese sind atomar, also nicht unterbrechbar. Die P-Operation prüft den Wert der Zählsperre (bzw. Zählvariable) und ver-

value is zero, the process gets blocked. If the value is greater than zero, it is decremented by one.

The *P* originally comes from the Dutch words *probeeren* (English: try) and *passeeren* (English: pass by) [69, 112, 119].

The V operation first increments the value of the counter variable by one. If processes are already waiting until they can pass the counter variable, the longest waiting process gets unblocked. The process that has just been unblocked then proceeds with its P operation and first decrements the value of the counter variable.

The *V* originally comes from the Dutch words *verhogen* (English: raise) and *vrijgeven* (English: release) [69, 112, 119].

The mentioned FIFO procedure, in which the longest waiting process is fetched from the queue first, corresponds to the *strong semaphore* [108] concept. It is the typical form of semaphores that operating systems provide. One advantage of this approach is that waiting processes cannot starve.

Another concept is the *weak semaphore* [108]. In this concept, processes are not fetched from the queue depending on the time they have been blocked, but instead on the process priority, for example. Such semaphores are useful, among other things, when real-time operation shall be implemented (see Section 3.6).

sucht, ihren Wert zu verringern. Ist der Wert 0, wird der Prozess blockiert. Ist der Wert > 0, wird er um den Wert 1 erniedrigt.

Das *P* geht ursprünglich auf die holländischen Wörter *probeeren* (deutsch: probieren bzw. versuchen) und *passeeren* (deutsch: passieren) zurück [69, 115, 119].

Die V-Operation erhöht als erstes den Wert der Zählsperre um eins. Warten bereits Prozesse darauf, die Zählersperre passieren zu dürfen, wird der am längsten wartende Prozess deblockiert. Der soeben deblockierte Prozess setzt dann seine P-Operation fort und erniedrigt als erstes den Wert der Zählersperre.

Das *V* geht ursprünglich auf die holländischen Wörter *verhogen* (deutsch: erhöhen) und *vrijgeven* (deutsch: freigeben) zurück [69, 115, 119].

Die beschriebene FIFO-Vorgehensweise, bei der der am längsten wartende Prozess als erstes aus der Warteschlange geholt wird, entspricht dem Konzept der *starken Semaphore* [107]. Dieses ist die typische Form der Semaphore, die Betriebssysteme bereitstellen. Ein Vorteil dieser Vorgehensweise ist, dass wartende Prozesse nicht verhungern können.

Ein anderes Konzept ist die *schwache Semaphore* [107]. Hier werden Prozesse nicht abhängig vom Zeitpunkt der Blockierung aus der Warteschlange geholt, sondern es ist zum Beispiel die Prozesspriorität entscheidend. Solche Semaphoren sind unter anderem dort hilfreich, wo Echtzeitbetrieb (siehe Abschnitt 3.6) realisiert werden soll.

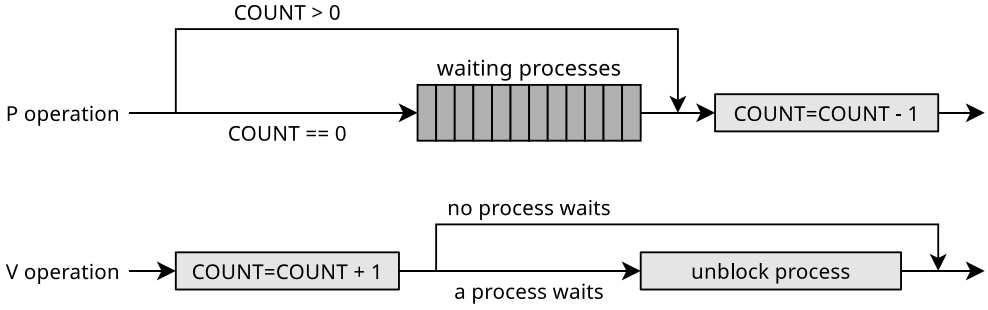

Figure 9.16: The P Operation attempts to decrement the Value of a Semaphore, and the V Operation increments it [119]

Producer/Consumer Example

One scenario that illustrates the usefulness of semaphores is the producer/consumer example. It involves a producer process that generates data and sends that data to another process (the consumer).

A finite buffer is used to minimize waiting times of the consumer. The producer inserts data into the buffer, and the consumer removes data from it (see Figure 9.17). If the buffer is full, the producer must be blocked. If the buffer is empty, the consumer must be blocked. To avoid inconsistencies, mutual exclusion is mandatory.

Erzeuger/Verbraucher-Beispiel

Ein Szenario, das den Nutzen von Semaphoren anschaulich zeigt, ist das Erzeuger/Verbraucher-Beispiel. Dabei sendet ein Daten erzeugender Prozess (der Erzeuger) diese Daten an einen anderen Prozess (den Verbraucher).

Ein endlicher Zwischenspeicher (Puffer) soll Wartezeiten des Verbrauchers minimieren. Der Erzeuger legt Daten in den Puffer und der Verbraucher entfernt sie aus diesem (siehe Abbildung 9.17). Ist der Puffer voll, muss der Erzeuger blockieren. Bei einem leeren Puffer muss der Verbraucher blockieren. Um Inkonsistenzen zu vermeiden, ist gegenseitiger Ausschluss zwingend erforderlich.

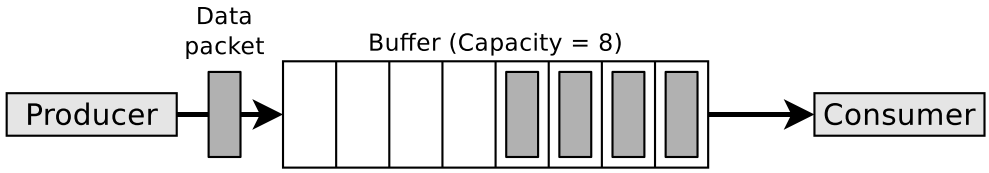

Figure 9.17: In the Creator/Consumer Example, one Process inserts Data into the Buffer, and another Process removes Data from it

Three semaphores `empty`, `full` and `mutex` are required to synchronize the access operations to the buffer.

The semaphore `empty` counts the free positions in the buffer. The producer process decrements it with the P operation, and the consumer process increments it with the V operation. If `empty` has value 0, the buffer is full, and the creator process is blocked.

The semaphore `filled` counts the data packets, i.e., the occupied positions in the buffer. The producer process increments it with the V operation, and the consumer process decrements it with the P operation. If `filled` has value 0, the buffer is empty, and the receiver process gets blocked.

The semaphores `filled` and `empty` are used in opposite to each other.

The *binary semaphore* `mutex` is used to ensure mutual exclusion. Binary semaphores are initialized with value 1 and ensure that two or more processes cannot enter their critical sections at the same time.

Zur Synchronisation der Zugriffe auf den Puffer werden drei Semaphoren `leer`, `filled`, und `mutex` benötigt.

Die Semaphore `empty` zählt die freien Plätze im Puffer. Der Erzeuger-Prozess erniedrigt sie mit der P-Operation, der Verbraucher-Prozess erhöht sie mit der V-Operation. Wenn `empty` den Wert 0 hat, ist der Puffer vollständig belegt und der Erzeuger-Prozess blockiert.

Die Semaphore `filled` zählt die Datenpakete, also die belegten Plätze im Puffer. Der Erzeuger-Prozess erhöht sie mit der V-Operation, der Verbraucher-Prozess erniedrigt sie mit der P-Operation. Wenn `filled` den Wert 0 hat, ist der Puffer leer und der Verbraucher-Prozess blockiert.

Die Semaphoren `filled` und `empty` werden gegenläufig zueinander eingesetzt.

Die *binäre Semaphore* `mutex` ist für den wechselseitigen Ausschluss zuständig. Binäre Semaphoren werden mit dem Wert 1 initialisiert und garantieren, dass zwei oder mehr Prozesse nicht gleichzeitig in ihre kritischen Bereiche eintreten können.

Listing 9.11 shows a possible solution to the producer/consumer example using the three semaphores described in this section.

Further Popular Interprocess Communication Problems

Besides the producer/consumer example described in this work, many other examples for the use of semaphores with the purpose of process cooperation exist in literature. Three well-known examples that cannot be discussed in depth in this book due to space limitations are:

- *Readers-writers problem*
- *Dining philosophers problem*
- *Sleeping barber problem*

The *readers-writers problem* [16, 35, 39, 108, 112, 123] is about the concurrent access of several processes to a shared memory location or database.

In the *dining philosophers problem* [35, 94, 112], several processes must be synchronized so that they can access limited resources, e.g., input/output devices, in a way that no process starves.

In the *problem of the sleeping barber* [33, 39, 114, 120], multiple processes must be synchronized, so that race conditions do not occur when accessing different resources.

Listing 9.11 zeigt eine mögliche Lösung des Erzeuger/Verbraucher-Beispiels mit den in diesem Abschnitt beschriebenen drei Semaphoren.

Weitere klassische Probleme der Interprozesskommunikation

Außer dem Erzeuger/Verbraucher-Beispiel gibt es noch zahlreiche andere in der Literatur populäre Beispiele für den Einsatz von Semaphoren und Mutexen zur Kooperation von Prozessen. Drei bekannte Beispiele, die aus Platzgründen in diesem Buch nicht intensiv besprochen werden können, sind:

- *Leser-/Schreiberproblem*
- *Philosophenproblem*
- *Problem des schlafenden Friseurs*

Beim *Leser-/Schreiberproblem* [16, 35, 39, 107, 115, 123] geht es um den gleichzeitigen Zugriff mehrerer Prozesse auf eine gemeinsame Speicherstelle oder eine Datenbank.

Beim *Philosophenproblem* [35, 94, 115] müssen mehrere Prozesse so synchronisiert werden, dass Sie auf knappe Ressourcen, wie z.B. Ein-/Ausgabegeräte, zugreifen können, ohne das einzelne Prozesse verhungern.

Beim *Problem des schlafenden Friseurs* [33, 39, 114, 120], müssen mehrere Prozesse so synchronisiert werden, dass es beim Zugriff auf verschiedene Ressourcen nicht zu Race Conditions kommt.

```
1  typedef int semaphore;                  // semaphores are of type integer
2  semaphore filled = 0;                   // counts the occupied locations in the buffer
3  semaphore empty  = 8;                   // counts the empty locations in the buffer
4  semaphore mutex  = 1;                   // controls access to the critial sections
5
6  void producer (void) {
7      int data;
8      while (TRUE) {                      // infinite loop
9          createDatapacket(data);         // create data packet
10         P(empty);                       // decrement the counter for empty positions
11         P(mutex);                       // enter the critical section
12         insertDatapacket(data);         // insert a data packet into the buffer
13         V(mutex);                       // leave the critical section
14         V(filled);                      // increment the counter for occupied positions
15     }
16 }
17
18 void consumer (void) {
19     int data;
20     while (TRUE) {                      // infinite loop
21         P(filled);                      // decrement the counter for occupied positions
22         P(mutex);                       // enter the critical section
23         removeDatapacket(data);         // remove a data packet from the buffer
24         V(mutex);                       // leave the critical section
25         V(empty);                       // increment the counter for empty positions
26         consumeDatapacket(data);        // consume a data packet
27     }
28 }
```

Listing 9.11: Implementation of the Producer/Consumer Example in Pseudocode [112]

9.4.2

Semaphores (System V)

The semaphore concept of Linux differs in some points from the semaphore concept of Dijkstra. In Linux and other Unix-like operating systems, the counter variable can be incremented or decremented with a P or V operation by more than one. It is also possible to perform atomic access operations on several different semaphores. This way, several P operations can be combined and executed only if none of the P operations block [119]. This is only possible when the semaphores are in the same *semaphore group*. The concept described in this section for cross-process protection of critical sections is also called *System V semaphore* in the literature. In addition, POSIX semaphores also exist (see Section 9.4.3).

The Linux kernel maintains a *semaphore table* (see Figure 9.18), which contains references to arrays of semaphores. Each array contains a single or a group of semaphores and is identified by the index of the table. Semaphores are addressed using the table index and the position in the group.

Furthermore, the Linux operating system kernel maintains for each semaphore group a data structure `semid_ds` with the metadata of the semaphore group. These metadata include the number of semaphores in the semaphore group, the access privileges, the time of the last access (P or V operation), and the time of the last modification [8, 47].

Linux provides three system calls for working with semaphores (see Table 9.6).

Semaphoren (System V)

Das Konzept der Semaphoren unter Linux weicht in einigen Punkten vom Konzept der Semaphoren nach Dijkstra ab. Unter Linux und anderen Unix-ähnlichen Betriebssystemen kann die Zählvariable mit einer P- oder V-Operation um mehr als den Wert 1 erhöht bzw. erniedrigt werden. Es können auch Zugriffsoperationen auf mehreren verschiedenen Semaphoren atomar durchgeführt werden. So können zum Beispiel mehrere P-Operationen zusammengefasst und nur dann durchgeführt werden, wenn keine der P-Operationen blockiert [119]. Das ist aber nur dann möglich, wenn sich die Semaphoren in einer *Semaphorengruppe* befinden. Das in diesem Abschnitt beschriebene Konzept zum prozessübergreifenden Schutz kritischer Abschnitte heißt in der Literatur auch *System V-Semaphoren*. Daneben existieren auch die POSIX-Semaphoren (siehe Abschnitt 9.4.3).

Der Linux-Betriebssystemkern verwaltet eine *Semaphorentabelle* (siehe Abbildung 9.18), die Verweise auf Arrays mit Semaphoren enthält. Jedes Array enthält eine einzelne oder eine Gruppe von Semaphoren und wird über den Index der Tabelle identifiziert. Einzelne Semaphoren werden über den Tabellenindex und die Position in der Gruppe angesprochen.

Zudem verwaltet der Linux-Betriebssystemkern für jede Semaphorengruppe eine Datenstruktur `semid_ds` mit den Metadaten der Semaphorengruppe. Diese sind unter anderem die Anzahl der Semaphoren in der Semaphorengruppe, die Zugriffsrechte, der Zeitpunkt des letzten Zugriffs (P- oder V-Operation) und der letzten Änderung [8, 47].

Linux stellt drei Systemaufrufe für die Arbeit mit Semaphoren bereit (siehe Tabelle 9.6).

Table 9.6: Linux System Calls for working with System V Semaphores

System call	Purpose
`semget`	Create a new semaphore, or a group of semaphores, or open an existing semaphore
`semctl`	Request or modify the value of an existing semaphore, or a group of semaphores, or erase a semaphore
`semop`	Carry out P and V operations on semaphores

The program example in Listing 9.12 shows how the protection of critical sections with Sys-

Das Programmbeispiel in Listing 9.12 zeigt, wie der Schutz kritischer Abschnitte mit Sys-

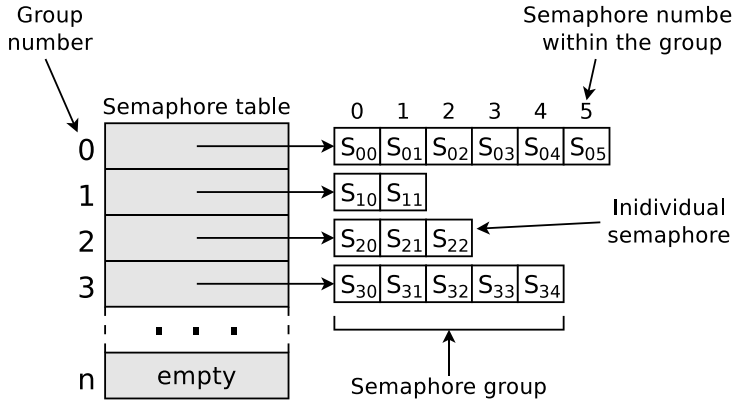

Figure 9.18: The Semaphore Table in Linux refers to Arrays of multiple System V Semaphores [119]

tem V semaphores can be done in Linux. In the example, two processes use loops to print single characters in the command-line. One process prints only the letter `A` and the other process prints only the letter `B`. The desired result is an alternating character sequence. Assuming each process has five loop iterations, the desired output is the string `ABABABABAB`.

The program creates in line 20 and 32 two semaphore groups with the names (keys) `12345` and `54321` with `semget`. Each semaphore group contains only one semaphore, as declared by the second parameter. The parameters `IPC_CREAT` and `IPC_EXCL` combined specify that a new semaphore group is created and that any existing group with the same key is not used. If a semaphore group with the same key already exists, `semget` returns an error message [8, 47]. If in the example only the parameter `IPC_CREAT` (without `IPC_EXCL`) was specified, a possibly already existing semaphore group of the same name would be used by the program [126]. The parameter `0600` specifies the access privileges. In this case, only the user who creates the semaphore group can carry out read and write operations on it.

The program sets the initial values of the semaphores with `semctl` (in line 52 and 60) with the parameter `SETVAL`. The semaphore of semaphore group `12345` will be set to value 1

tem V-Semaphoren unter Linux möglich ist. Konkret sollen im Beispiel zwei Prozesse mit Hilfe von Schleifen einzelne Zeichen auf der Kommandozeile ausgeben. Ein Prozess gibt ausschließlich den Buchstaben `A` aus und der andere Prozess ausschließlich den Buchstaben `B`. Das gewünschte Ergebnis ist eine abwechselnde Zeichenausgabe. Geht man davon aus, dass es pro Prozess fünf Schleifendurchläufe gibt, ist die gewünschte Ausgabe die Zeichenfolge `ABABABABAB`.

Das Programm erstellt zwei Semaphorengruppen mit den Namen (Keys) `12345` und `54321` mit `semget` (in Zeile 20 und 32). In jeder Semaphorengruppe befindet sich – definiert durch den zweiten Parameter – nur ein Semaphor. Die Parameter `IPC_CREAT` und `IPC_EXCL` zusammen definieren, dass eine neue Semaphorengruppe erstellt wird und nicht auf eine eventuell existierende Gruppe mit identischem Key zugegriffen wird. Für den Fall, dass schon eine Semaphorengruppe mit dem gleichen Key existiert, gibt `semget` eine Fehlermeldung zurück [8, 47]. Wäre im Beispiel nur der Parameter `IPC_CREAT` (ohne `IPC_EXCL`) angegeben, würde eine eventuell schon existierende Semaphorengruppe gleichen Namens vom Programm verwendet werden [126]. Der Parameter `0600` definiert die Zugriffsrechte. In diesem Fall darf nur der Benutzer, der die Semaphorengruppe anlegt, auf diese lesend und schreibend zugreifen.

Die initialen Werte der Semaphoren setzt das Programm mit `semctl` (in Zeile 52 und 60) mit dem Parameter `SETVAL`. Die Semaphore der Semaphorengruppe `12345` erhält den Wert 1 und

and the semaphore of semaphore group `54321` will be set to value 0 to specify the execution order of the involved processes. In the lines 69 and 75, the program again requests the values of the two semaphores with `semctl` but this time with the parameter `GETVAL` for printing them on the command-line for verification [48].

In line 80, the program attempts to create a child process using the `fork` function. If the process creation was successful, there is now a child process that tries with `semop` (in line 92) to carry out a P operation on the semaphore in semaphore group `54321` i.e., decrement its value by one. This is impossible at the beginning of the program because, in line 60, the semaphore got the initial value 0.

The parent process attempts to carry out a P operation on the semaphore in the semaphore group `12345` by using `semop` (in line 107). This can be done at the beginning of the program because this semaphore received the initial value 1 in line 52. This way, it is ensured that after the `fork` runs the parent process first.

During its first loop run, the parent process prints the character `A` in the command-line and afterward increments the value of the semaphore in group `54321` with `semop` (in line 115). Another loop run for the parent process is now impossible because the semaphore in the group `12345` has the value zero. However, the child process can now carry out the P operation with `semop` (in line 92) and print the character `B` in the command-line. Then, the child process increments with `semop` (in line 99) the value of the semaphore in group `12345` and the parent process can proceed. This way, the desired output sequence is provided.

Finally, the program removes both semaphores (semaphore groups) with `semctl` and the parameter `IPC_RMID` (in line 125 and 136). The command `wait` (in line 120) ensures that the child process has finished before.

In a Linux system, the output of the command `ipcs` gives an overview of the System V semaphores currently existing. The `-s` option instructs `ipcs` to return only the semaphore-related information. The `sleep` instructions in lines 95 and 111 are used only to slow down

die Semaphore der Semaphorengruppe `54321` erhält den Wert 0, um später die Startreihenfolge der beteiligten Prozesse zu definieren. In den Zeilen 69 und 75 fragt das Programm die Werte der beiden Semaphoren erneut mit `semctl`, aber diesmal mit dem Parameter `GETVAL` ab, um Sie zur Kontrolle auf der Kommandozeile auszugeben [48].

In Zeile 80 versucht das Programm, einen Kindprozess mit der Funktion `fork` zu erzeugen. War die Prozesserzeugung erfolgreich, existiert nun ein Kindprozess, der versucht mit `semop` (in Zeile 92) eine P-Operation auf die Semaphore in der Gruppe `54321` auszuführen, also deren Wert um eins zu dekrementieren. Zu Beginn des Programms ist das nicht möglich, da diese Semaphore in Zeile 60 den initialen Wert 0 erhalten hat.

Der Elternprozess versucht mit `semop` (in Zeile 107) eine P-Operation auf die Semaphore in der Gruppe `12345` auszuführen. Zu Beginn des Programms ist das möglich, da diese Semaphore in Zeile 52 den initialen Wert 1 erhalten hat. Damit steht fest, dass nach dem `fork` zuerst der Elternprozess läuft.

Der Elternprozess gibt bei seinem ersten Schleifendurchlauf das Zeichen `A` auf der Kommandozeile aus und inkrementiert daraufhin mit `semop` (in Zeile 115) den Wert der Semaphore in der Gruppe `54321`. Ein weiterer Schleifendurchlauf des Elternprozesses ist nun nicht möglich, da die Semaphore in der Gruppe `12345` den Wert null hat. Allerdings kann nun der Kindprozess die P-Operation mit `semop` (in Zeile 92) ausführen und das Zeichen `B` auf der Kommandozeile ausgeben. Daraufhin inkrementiert der Kindprozess mit `semop` (in Zeile 99) den Wert der Semaphore in der Gruppe `12345` und der Elternprozess kann weiterlaufen. So ist die gewünschte Ausgabesequenz sichergestellt.

Abschließend entfernt das Programm die beiden Semaphoren(gruppen) mit `semctl` und dem Parameter `IPC_RMID` (in Zeile 125 und 136). Das Kommando `wait` (in Zeile 120) garantiert, dass der Kindprozess zuvor beendet ist.

Eine Übersicht über die existierenden System V-Semaphoren in einem Linux-System liefert die Ausgabe des Kommandos `ipcs`. Mit der Option `-s` wird `ipcs` angewiesen, nur die Semaphoren auszugeben. Die `sleep`-Anweisungen in den Zeilen 95 und 111 dienen nur der Ver-

the program execution so that the semaphores can be observed with `ipcs` and to provide some randomness in the execution speed of these very simple processes. To be precise, `sleep(rand() %3);` causes a pseudorandom waiting time between 0 and 2 seconds.

Erasing System V semaphores is also possible using the `ipcrm` command from the command line interface, just like with shared memory areas and message queues.

langsamung der Programmausführung, damit die Semaphoren mit `ipcs` beobachtet werden können und eine gewisse Zufälligkeit in der Abarbeitungsgeschwindigkeit dieser sehr einfachen Prozesse gegeben ist. Konkret verursacht `sleep(rand() %3);` eine pseudozufällige Wartezeit zwischen 0 und 2 Sekunden.

Das Löschen von System V-Semaphoren ist genau wie bei gemeinsamen Speicherbereichen und Nachrichtenwarteschlangen auch von der Eingabeaufforderung aus mit dem Kommando `ipcrm` möglich.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <sys/sem.h>
6
7  void main() {
8    int pid_of_child;
9    int key1=12345;
10   int key2=54321;
11   int rc_semget1;        // Return code of semget
12   int rc_semget2;        // Return code of semget
13   int rc_semctl;         // Return code of semctl
14   int output;
15
16   // Prevent buffering of standard output (stdout)
17   setbuf(stdout, NULL);
18
19   // Create new semaphore group 12345 with one semaphore
20   rc_semget1 = semget(key1, 1, IPC_CREAT | IPC_EXCL | 0600);
21   if (rc_semget1 < 0) {
22     printf("Unable to create the semaphore group.\n");
23     perror("semget");
24     // Program abort
25     exit(1);
26   } else {
27     printf("Semaphore group %i with key %i has been created.\n",
28            rc_semget1, key1);
29   }
30
31   // Create new semaphore group 54321 with one semaphore
32   rc_semget2 = semget(key2, 1, IPC_CREAT | IPC_EXCL | 0600);
33   if (rc_semget2 < 0) {
34     printf("Unable to create the semaphore group.\n");
35     perror("semget");
36     // Program abort
37     exit(1);
38   } else {
39     printf("Semaphore group %i with key %i has been created.\n",
```

```
40            rc_semget2, key2);
41  }
42
43  // Definition of the P operation.
44  // Decrement the value of a semaphore by one.
45  struct sembuf p_operation = {0, -1, 0};
46
47  // Definition of the V operation.
48  // Increment the value of a semaphore by one.
49  struct sembuf v_operation = {0, 1, 0};
50
51  // Set first semaphore of group 12345 to initial value 1
52  rc_semctl = semctl(rc_semget1, 0, SETVAL, 1);
53  if (rc_semctl < 0) {
54    printf("Unable to set the value of %i.\n", key1);
55    perror("semctl SETVAL");
56    exit(1);
57  }
58
59  // Set first semaphore of group 54321 to initial value 0
60  rc_semctl = semctl(rc_semget2, 0, SETVAL, 0);
61  if (rc_semctl < 0) {
62    printf("Unable to set the value of %i.\n", key2);
63    perror("semctl SETVAL");
64    exit(1);
65  }
66
67  // Return initial value of the first semaphore of
68  // semaphore group 12345 for verification
69  output = semctl(rc_semget1, 0, GETVAL, 0);
70  printf("Value of the semaphore group %i: %i\n",
71         rc_semget1, output);
72
73  // Return initial value of the first semaphore of
74  // semaphore group 54321 for verification
75  output = semctl(rc_semget2, 0, GETVAL, 0);
76  printf("Value of the semaphore group %i: %i\n",
77         rc_semget2, output);
78
79  // Create a child process
80  pid_of_child = fork();
81
82  // An error occured --> program abort
83  if (pid_of_child < 0) {
84    perror("Fork caused an error!\n");
85    exit(1);
86  }
87
88  // Child process
89  if (pid_of_child == 0) {
90    for (int i=0;i<5;i++) {
```

```
91        // P Operation semaphore 54321
92        semop(rc_semget2, &p_operation, 1);
93        // Critical section (start)
94        // Pause. Wait between 0 and 2 seconds
95        sleep(rand() % 3);
96        printf("B");
97        // Critical section (end)
98        // V Operation semaphore 12345
99        semop(rc_semget1, &v_operation, 1);
100     }
101     exit(0);
102   }
103
104   // Parent process
105   if (pid_of_child > 0) {
106     for (int i=0;i<5;i++) {
107       semop(rc_semget1, &p_operation, 1);
108       // P Operation semaphore 12345
109       // Critical section (start)
110       // Pause. Wait between 0 and 2 seconds
111       sleep(rand() % 3);
112       printf("A");
113       // Critical section (end)
114       // V Operation semaphore 54321
115       semop(rc_semget2, &v_operation, 1);
116     }
117   }
118
119   // Wait for the termination of the child process
120   wait(NULL);
121
122   printf("\n");
123
124   // Remove semaphore group 12345
125   rc_semctl = semctl(rc_semget1, 0, IPC_RMID, 0);
126     if (rc_semctl < 0) {
127       printf("Unable to remove the semaphore group.\n");
128       perror("semctl");
129       exit(1);
130   } else {
131       printf("Semaphore group %i has been removed.\n",
132              rc_semget1);
133   }
134
135   // Remove semaphore group 54321
136   rc_semctl = semctl(rc_semget2, 0, IPC_RMID, 0);
137     if (rc_semctl < 0) {
138       printf("Unable to remove the semaphore group.\n");
139       perror("semctl");
140       exit(1);
141   } else {
```

```
142         printf("Semaphore group %i has been removed.\n",
143                 rc_semget2);
144     }
145
146     exit(0);
147 }
```

Listing 9.12: Program Example for System V Semaphores

Compiling the program with the GNU C compiler (`gcc`) and running it should result in the following output:

Das Übersetzen des Programms mit dem GNU C Compiler (`gcc`) unter Linux und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_12_semaphore_systemv.c -o Listing_9_12_semaphore_systemv
$ ./Listing_9_12_semaphore_systemv
Semaphore group 12 with key 12345 has been created.
Semaphore group 13 with key 54321 has been created.
Value of the semaphore group 12: 1
Value of the semaphore group 13: 0
ABABABABAB
Semaphore group 12 has been removed.
Semaphore group 13 has been removed.
```

Without mutual exclusion using the semaphores in Listing 9.12, there would be a more or less random output sequence of the two processes. Possible output sequences would be, for example `ABBABABA`, `ABBAABAB`, or `ABABABBA`. Without mutual exclusion using the semaphores and without the `sleep` instructions, the output sequence would be, in most cases, `AAAAABBBBB`, and in relatively rare cases `BBBBBAAAAA` or something like `AABAAABBBB`. For any case, the output sequence would not be predicted with certainty because it would depend on the workload of the overall system (number of processes and their resource consumption) and the scheduling behavior of the operating system and would thus be partially subject to randomness.

It is interesting to observe the semaphore groups using the `ipcs` utility, which returns information about the number of semaphores included, the access privileges, the ownership, and the associated semaphore IDs and semaphore keys.

Exactly as with shared memory segments (see Section 9.3.3) and message queues (see Section 9.3.3), the operating system kernel automatically assigns the IDs (`semid`) to the semaphores. By contrast, the semaphore key

Ein Verzicht auf den gegenseitigen Ausschluss mittels der Semaphoren in Listing 9.12 würde zu einer mehr oder weniger zufälligen Ausgabereihenfolge der beiden Prozesse führen. Mögliche Ausgabesequenzen wären z.B. `ABBABABABA`, `ABBAABABAB` oder `ABABABABBA`. Ohne gegenseitigen Ausschluss mittels der Semaphoren und ohne die `sleep`-Anweisungen wäre die Ausgabesequenz in den meisten Fällen `AAAAABBBBB` und in relativ seltenen Fällen `BBBBBAAAAA` oder so ähnlich wie `AABAAABBBB`. In jedem Fall wäre die Ausgabesequenz nicht sicher vorhersagbar, weil abhängig von der Auslastung des Gesamtsystems (Anzahl und Ressourcenverbrauch der übrigen Prozesse) und vom Schedulingverhalten des Betriebssystems und damit teilweise dem Zufall unterworfen.

Interessant ist die Beobachtung der Semaphorengruppen mit dem Programm `ipcs`, das Informationen über die Anzahl der enthaltenen Semaphoren, die Zugriffsrechte, den Besitzer und die zugehörigen Semaphor-IDs und Semaphor-Keys liefert.

Genau wie bei gemeinsamen Speichersegmenten (siehe Abschnitt 9.3.3) und Nachrichtenwarteschlangen (siehe Abschnitt 9.3.3) legt der Betriebssystemkern die IDs (`semid`) der Semaphoren automatisch fest. Der Semaphor-Key (`key`) hingegen ist im Programquelltext defi-

(key) is specified in the source code and is returned by ipcs in hexadecimal notation.

```
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00003039 12         bnc        600        1
0x0000d431 13         bnc        600        1
```

In addition to the parameters (GETVAL, SETVAL and IPC_RMID) for semctl that have already been discussed in this section, many other parameters for requesting and modifying information about existing semaphores or semaphore groups exist. Table 9.7 contains a selection of these command parameters.

niert und wird von ipcs im Hexadezimalsystem angegeben.

Neben den in diesem Abschnitt bereits vorgestellten Kommandoparametern (GETVAL, SETVAL und IPC_RMID) für semctl existieren noch zahlreiche weitere Kommandoparameter, mit denen Informationen über existierende Semaphoren oder Semaphorengruppen abgefragt und verändert werden können. Eine Auswahl dieser Kommandoparameter enthält Tabelle 9.7.

Table 9.7: Parameters of semctl [8]

Parameter	Purpose
GETALL	Request the values of all semaphores in a semaphore group
GETNCNT	Request the number of processes waiting until the semaphore's value exceeds zero
GETPID	Request the process number (PID) of the process that last carried out a P or V operation on the semaphore
GETVAL	Request the value of a semaphore
GETZCNT	Request the number of processes waiting until the semaphore's value is zero
IPC_INFO	Request information about semaphore-specific limits (e.g., maximum number of semaphores per group)
IPC_RMID	Erase a semaphore group
IPC_STAT	Request the content of the data structure semid_ds of a particular semaphore group, which the operating system kernel maintains for each semaphore group. The data structure contains the metadata of the semaphore group. These metadata include the number of semaphores in the semaphore group, the access privileges, the time of the last access (P or V operation), and the time of the last modification
IPC_SET	Modify the ownership and access privileges
SEM_INFO	Almost identical to IPC_INFO. It also returns the number of semaphores and semaphore groups currently in the system SETALL
SETALL	Set all semaphores of a semaphore group to a given value
SETVAL	Set a semaphore to a given value

9.4.3

Semaphores (POSIX)

An alternative method of implementing semaphores is offered by the POSIX standard. The POSIX semaphore interface is considered

POSIX-Semaphoren

Eine alternative Möglichkeit, um Semaphoren zu realisieren, bietet der Standard POSIX. Die Schnittstelle von POSIX-Semaphoren gilt im Vergleich zu System V als intuitiver und damit

more intuitive and, therefore, easier to learn than System V, but this is a subjective feeling.

A substantial simplification of working with POSIX semaphores is, among other things, that in contrast to System V semaphores, they are not organized in semaphore groups but are independent non-negative integer counter variables precisely as in the concept [33] developed by Edsger Dijkstra. Furthermore, a POSIX semaphore gets its initial value assigned immediately at creation with the library function `sem_open` or with `sem_init`. When using System V semaphores, the two system calls `semget` and `semctl` or the library functions of the same name are required for this purpose [47].

POSIX semaphores can be *named* or *unnamed semaphores*. Both variants are suitable for interprocess and intraprocess communication, i.e., for cooperation between processes and for cooperation only between the threads of a process [86]. Named POSIX semaphores are represented in Linux in the `/dev/shm/` directory as files with filenames in the form `sem.<name>`. For unnamed POSIX semaphores, this is not the case. POSIX semaphores cannot be examined with the `ipcs` command-line tool and they cannot be manually removed with `ipcrm`

Table 9.8 contains the C function calls for POSIX semaphores specified in the header file `semaphore.h`.

leichter erlernbar, was aber subjektives Empfinden ist.

Eine konkrete Vereinfachung der Arbeit mit POSIX-Semaphoren ist u.a., dass sie im Gegensatz von System V-Semaphoren nicht in Semaphorengruppen organisiert sind, sondern genau wie beim von Edsger Dijkstra entwickelten Konzept [33], eigenständige ganzzahlige, nichtnegative Zählersperren sind. Zudem wird einer POSIX-Semaphore bei der Erzeugung mit der Bibliotheksfunktion `sem_open` oder alternativ mit `sem_init` sofort der initiale Wert zugewiesen. Bei System V-Semaphoren sind hierfür die beiden Systemaufrufe `semget` und `semctl` oder die gleichnamigen Bibliotheksfunktionen nötig [47].

POSIX-Semaphoren können als *benannte* (englisch: *named Semaphor*) oder alternativ als *unbenannte Semaphoren* (englisch: *unnamed Semaphor*) realisiert sein. Beide Varianten eigenen sich für Inter- und für Intraprozesskommunikation, also für Kooperation zwischen Prozessen und für Kooperation ausschließlich zwischen den Threads eines Prozesses [86]. Benannte POSIX-Semaphoren sind unter Linux im Verzeichnis `/dev/shm/` als Dateien mit Datennamen in der Form `sem.<name>` repräsentiert. Bei unbenannten POSIX-Semaphoren ist das nicht der Fall. Generell können POSIX-Semaphoren nicht mit dem Kommandozeilenwerkzeug `ipcs` untersucht und auch nicht mit `ipcrm` manuell entfernt werden.

Tabelle 9.8 enthält die in der Header-Datei `semaphore.h` definierten C-Funktionsaufrufe der POSIX-Semaphoren.

Table 9.8: C Function Calls for POSIX Semaphores

Function call	Purpose
`sem_init`	Create a *unnamed* semaphore and set its initial value
`sem_open`	Create a *named* semaphore and set its initial value
`sem_post`	V Operation
`sem_wait`	P Operation. Blocking instruction
`sem_trywait`	P Operation. Executed only if it does not block the calling process
`sem_timedwait`	P Operation. Blocking instruction with specified timeout
`sem_getvalue`	Request the value of a semaphore
`sem_destroy`	Delete *unnamed* semaphore
`sem_close`	Close *named* semaphore
`sem_unlink`	Delete *named* semaphore

The program example in Listing 9.13 demonstrates how protecting critical sections with named POSIX semaphores can be done in Linux. Exactly as in the program example in Listing 9.12, two processes use loops to print single characters (either `A` or `B`) in five loop runs on the command-line. Again, the desired result is the character sequence `ABABABAB`.

The program creates two named POSIX semaphores `/mysem1` and `/mysem2` using the library function `sem_open` (in line 21 and 32) and the parameter `O_CREAT`, and sets their initial values to 1 and 0. The parameter `0600` specifies the same access privileges as in the program example in Listing 9.12. Also, in this example, it would be possible to use the parameter `O_CREAT` in addition to the parameter `O_EXCL` to prevent that a possibly existing semaphore of the same name is used.

POSIX semaphore names must start with a slash (`/`), as shown in the example in Listing 9.13, and must not contain another slash.

In the lines 44 and 49, the program requests the values of the two semaphores with `sem_getvalue` for printing them on the command-line for verification.

In line 53, the program attempts to create a child process with the function `fork`. If the process creation was successful, there is now a child process that tries with `sem_wait` (in line 65) to carry out a P operation on the semaphore `/mysem2` i.e., to decrement its value by one. This is impossible at the beginning of the program because this semaphore was set to initial value 0 in line 32.

The parent process tries using `sem_wait` (in line 82) to carry out a P operation on the semaphore `/mysem1`. This can be done at the beginning of the program because that semaphore is set to initial value 1 in line 21. Thus, just like in the program example in Listing 9.12, it is ensured that the parent process runs first after the `fork`.

During its first loop run, the parent process prints the character `A` on the command-line and, after that, increments with `sem_post` (in line 90) the value of the semaphore `/mysem2`. Another loop run of the parent process is impossible be-

Das Programmbeispiel in Listing 9.13 zeigt, wie der Schutz kritischer Abschnitte mit benannten POSIX-Semaphoren unter Linux möglich ist. Genau wie beim Programmbeispiel in Listing 9.12 sollen zwei Prozesse mit Hilfe von Schleifen einzelne Zeichen (entweder `A` oder `B`) in fünf Schleifendurchläufen auf der Kommandozeile ausgeben. Auch hier ist das gewünschte Ergebnis wieder die Zeichenfolge `ABABABABAB`.

Das Programm erstellt zwei benannte POSIX-Semaphoren `/mysem1` und `/mysem2` mit der Bibliotheksfunktion `sem_open` (in Zeile 21 und 32) und dem Parameter `O_CREAT`, und weist diesen die initialen Werte 1 und 0 zu. Der Parameter `0600` definiert die gleichen Zugriffsrechte wie im Programmbeispiel in Listing 9.12. Auch hier wäre es möglich, zusätzlich zum Parameter `O_CREAT` den Parameter `O_EXCL` anzugeben, um das Öffnen einer eventuell existierenden Semaphore gleichen Namens zu unterbinden.

Die Namen benannter POSIX-Semaphoren müssen wie im Beispiel in Listing 9.13 mit einem Schrägstrich – Slash (`/`) beginnen und dürfen im weiteren Verlauf keinen weiteren Slash enthalten.

In den Zeilen 44 und 49 fragt das Programm die Werte der beiden Semaphoren mit `sem_getvalue` ab, um Sie zur Kontrolle auf der Kommandozeile auszugeben.

In Zeile 53 versucht das Programm, einen Kindprozess mit der Funktion `fork` zu erzeugen. War die Prozesserzeugung erfolgreich, existiert nun ein Kindprozess, der versucht mit `sem_wait` (in Zeile 65) eine P-Operation auf die Semaphore `/mysem2` auszuführen, also deren Wert um eins zu dekrementieren. Zu Beginn des Programms ist das nicht möglich, da diese Semaphore in Zeile 32 den initialen Wert 0 erhalten hat.

Der Elternprozess versucht mit `sem_wait` (in Zeile 82) eine P-Operation auf die Semaphore `/mysem1` auszuführen. Zu Beginn des Programms ist das möglich, da diese Semaphore in Zeile 21 den initialen Wert 1 erhalten hat. Damit steht genau wie beim Programmbeispiel in Listing 9.12 fest, dass nach dem `fork` zuerst der Elternprozess läuft.

Der Elternprozess gibt bei seinem ersten Schleifendurchlauf das Zeichen `A` auf der Kommandozeile aus und inkrementiert daraufhin mit `sem_post` (in Zeile 90) den Wert der Semaphore `/mysem2`. Ein weiterer Schleifendurchlauf des

cause the value of the semaphore `/mysem1` is zero. Now, the child process can carry out the P operation on the semaphore `/mysem2` with `sem_wait` (in line 65) and print the character `B` on the command-line. Next, the child process increments with `sem_post` (in line 73) the value of the semaphore `/mysem1` and the parent process can proceed. This ensures the desired output sequence.

Finally, the program closes both semaphores with `sem_close` (in line 100 and 109) and removes them with `sem_unlink` (in line 118 and 127).

Calling `sem_close` is not necessary for the program example in Listing 9.13 to work correctly because when a process that has access to a named POSIX semaphore terminates, the semaphore is automatically closed. However, it is not good programming style to omit `sem_close`. The library function `sem_unlink` causes the deletion of a POSIX semaphore. This is carried out as soon as there is no more reference to the semaphore, i.e., when the last process that opened the semaphore did call `sem_close` or has terminated.

The command `wait` (in line 95) ensures, just as in the program example in Listing 9.12, that the child process terminates before the semaphores are removed.

Elternprozesses ist nun nicht möglich, da die Semaphore `/mysem1` den Wert null hat. Allerdings kann nun der Kindprozess die P-Operation auf die Semaphore `/mysem2` mit `sem_wait` (in Zeile 65) ausführen und das Zeichen `B` auf der Kommandozeile ausgeben. Daraufhin inkrementiert der Kindprozess mit `sem_post` (in Zeile 73) den Wert der Semaphore `/mysem1` und der Elternprozess kann weiterlaufen. So ist die gewünschte Ausgabesequenz sichergestellt.

Abschließend schließt das Programm die beiden Semaphoren mit `sem_close` (in Zeile 100 und 109) und entfernt diese mit `sem_unlink` (in Zeile 118 und 127).

Der Aufruf von `sem_close` ist für die korrekte Funktion des Programmbeispiels in Listing 9.13 nicht nötig, da automatisch bei der Beendigung eines Prozesses mit Zugriff auf eine benannte POSIX-Semaphore diese geschlossen wird. Es ist aber kein guter Programmierstil, wenn `sem_close` weggelassen wird. Die Bibliotheksfunktion `sem_unlink` weist das Löschen einer POSIX-Semaphore an. Dieses geschieht sobald es keine Referenz mehr auf die Semaphore gibt, also wenn der letzte Prozess, der diese geöffnet hat, `sem_close` aufgerufen hat, oder beendet ist.

Das Kommando `wait` (in Zeile 95) garantiert genau wie beim Programmbeispiel in Listing 9.12, dass der Kindprozess vor dem Entfernen der Semaphoren beendet ist.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <semaphore.h>
6 #include <fcntl.h>
7
8 void main() {
9   const char sem1_name[] = "/mysem1";
10  const char sem2_name[] = "/mysem2";
11  int pid_of_child;
12  int returncode_close, returncode_unlink;
13  int output;
14
15  sem_t *mutex_sem1, *mutex_sem2;
16
17  // Prevent buffering of standard output (stdout)
18  setbuf(stdout, NULL);
19
20  // Create new named semaphore /mysem1 with initial value 1
```

```
21  mutex_sem1 = sem_open(sem1_name, O_CREAT, 0600, 1);
22  if (mutex_sem1 == SEM_FAILED) {
23    printf("Unable to create the semaphore.\n");
24    perror("sem_open");
25    // Program abort
26    exit(1);
27  } else {
28    printf("Semaphore %s has been created.\n", sem1_name);
29  }
30
31  // Create new named semaphore /mysem2 with initial value 0
32  mutex_sem2 = sem_open(sem2_name, O_CREAT, 0600, 0);
33  if (mutex_sem2 == SEM_FAILED) {
34    printf("Unable to create the semaphore.\n");
35    perror("sem_open");
36    // Program abort
37    exit(1);
38  } else {
39    printf("Semaphore %s has been created.\n", sem2_name);
40  }
41
42  // Return the initial value of the semaphore /mysem1
43  // for verification
44  sem_getvalue(mutex_sem1, &output);
45  printf("Value of %s: %i\n", sem1_name, output);
46
47  // Return the initial value of the semaphore /mysem2
48  // for verification
49  sem_getvalue(mutex_sem2, &output);
50  printf("Value of %s: %i\n", sem2_name, output);
51
52  // Create a child process
53  pid_of_child = fork();
54
55  // An error occured --> program abort
56  if (pid_of_child < 0) {
57    perror("Fork caused an error!\n");
58    // Program abort
59    exit(1);
60  }
61
62  // Child process
63  if (pid_of_child == 0) {
64    for (int i=0;i<5;i++) {
65      sem_wait(mutex_sem2);
66      // P Operation semaphore /mysem2
67      // Critical section (start)
68      // Pause. Wait between 0 and 2 seconds
69      sleep(rand() % 3);
70      printf("B");
71      // Critical section (end)
```

```
72        // V Operation semaphore /mysem1
73        sem_post(mutex_sem1);
74      }
75
76      exit(0);
77    }
78
79    // Parent process
80    if (pid_of_child > 0) {
81      for (int i=0;i<5;i++) {
82        sem_wait(mutex_sem1);
83        // P Operation semaphore /mysem1
84        // Critical section (start)
85        // Pause. Wait between 0 and 2 seconds
86        sleep(rand() % 3);
87        printf("A");
88        // Critical section (end)
89        // V Operation semaphore /mysem2
90        sem_post(mutex_sem2);
91      }
92    }
93
94    // Wait for the termination of the child process
95    wait(NULL);
96
97    printf("\n");
98
99    // Close semaphore /mysem1
100   returncode_close = sem_close(mutex_sem1);
101     if (returncode_close < 0) {
102       printf("%s could not be closed.\n", sem1_name);
103       exit(1);
104   } else {
105       printf("%s has been closed.\n", sem1_name);
106   }
107
108   // Close semaphore /mysem2
109   returncode_close = sem_close(mutex_sem2);
110     if (returncode_close < 0) {
111       printf("%s could not be closed.\n", sem2_name);
112       exit(1);
113   } else {
114       printf("%s has been closed.\n", sem2_name);
115   }
116
117   // Remove semaphore /mysem1
118   returncode_unlink = sem_unlink(sem1_name);
119     if (returncode_unlink < 0) {
120       printf("%s could not be removed.\n", sem1_name);
121       exit(1);
122   } else {
```

```
123         printf("%s has been removed.\n", sem1_name);
124     }
125
126     // Remove semaphore /mysem2
127     returncode_unlink = sem_unlink(sem2_name);
128       if (returncode_unlink < 0) {
129         printf("%s could not be removed.\n", sem2_name);
130         exit(1);
131     } else {
132         printf("%s has been removed.\n", sem2_name);
133     }
134
135     exit(0);
136 }
```

Listing 9.13: Program Example for System V Semaphores

Compiling the program in Linux using the GNU C compiler (`gcc`) with the option `-lpthread` for linking to the POSIX thread library (`libpthread`) and running it should result in the following output:

Das Übersetzen des Programms unter Linux mit dem GNU C Compiler (`gcc`) mit der Option `-lpthread` zur Verknüpfung mit der POSIX-Thread-Bibliothek (`libpthread`) und das anschließende Ausführen führt im Erfolgsfall zu folgender Ausgabe:

```
$ gcc Listing_9_13_semaphore_posix_named.c \
      -o Listing_9_13_semaphore_posix_named -lpthread
$ ./Listing_9_13_semaphore_posix_named
Semaphore /mysem1 has been created.
Semaphore /mysem2 has been created.
Value of /mysem1: 1
Value of /mysem2: 0
ABABABABAB
/mysem1 has been closed.
/mysem2 has been closed.
/mysem1 has been removed.
/mysem2 has been removed.
```

In Linux, the presence of POSIX semaphores and their access privileges can be examined in the directory `/dev/shm`.

Unter Linux können die Existenz benannter POSIX-Semaphoren und deren Zugriffsrechte im Verzeichnis `/run/shm` kontrolliert werden.

```
$ ls -l /run/shm/
-rw-------  1 bnc  bnc    32 20. Jan 16:20 sem.mysem1
-rw-------  1 bnc  bnc    32 20. Jan 16:20 sem.mysem2
```

In comparison to named POSIX semaphores, such as in the program example in Listing 9.13, unnamed POSIX semaphores are even more lightweight since they are not represented in the file system. Creating such semaphores is done with the library function `sem_init` and removing them is done with `sem_destroy`. Unnamed semaphores are perfectly suited for protecting

Im Vergleich zu benannten POSIX-Semaphoren, wie im Programmbeispiel in Listing 9.13, sind unbenannte POSIX-Semaphoren noch leichtgewichtiger, da sie nicht im Dateisystem repräsentiert sind. Die Erstellung solcher Semaphoren geschieht mit der Bibliotheksfunktion `sem_init` und das Entfernen mit `sem_destroy`. Ideal geeignet

critical sections of the threads of a process, i.e., for intra-process communication. Cross-process usage of unnamed POSIX semaphores – i.e., interprocess communication – is possible but not very convenient.

There are two requirements for this: the correct specification when creating the semaphore with `sem_init` and all processes involved having access to the semaphore. When creating an unnamed POSIX semaphore with `sem_init` the parameter `pshared` is used to specify whether the semaphore will be available exclusively in the process context of the calling process (i.e., used in a multi-threaded scenario) or if it will be used to protect critical sections across processes (i.e., be accessible by different processes). This requires that the semaphore resides in a System V memory segment (see Section 9.3.1) or in a POSIX memory segment (see Section 9.3.2) that can be accessed by the processes involved [86]. This restriction applies even if the involved processes are closely related because a child process inherits a copy of the unnamed semaphore through `fork` but cannot automatically use it to reach the semaphore of the parent process.

sind unbenannte Semaphoren zum Schutz kritischer Abschnitte der Threads eines Prozesses, also für Intraprozesskommunikation. Die Prozessübergreifende Nutzung unbenannter POSIX-Semaphoren – also Interprozesskommunikation – ist zwar möglich, aber nicht sehr komfortabel.

Zwei Dinge sind hierzu Voraussetzung, nämlich die korrekte Angabe bei der Erstellung der Semaphore mit `sem_init` und die Möglichkeit des Zugriffs für alle beteiligten Prozesse. Beim Erstellen einer unbenannten POSIX-Semaphore mit `sem_init` wird mit dem Parameter `pshared` definiert, ob die Semaphore ausschließlich im Prozesskontext des aufrufenden Prozesses zur Verfügung stehen soll (also in einem Szenario mit mehreren Threads zum Einsatz kommt), oder ob sie zum prozessübergreifenden Schutz kritischer Abschnitte verwendet werden soll (also für verschiedene Prozesse erreichbar sein soll). Damit das möglich ist, muss sich die Semaphore in einem System V-Speichersegment (siehe Abschnitt 9.3.1) oder in einem POSIX-Speichersegment (siehe Abschnitt 9.3.2) befinden, auf das die beteiligten Prozesse Zugriff haben [86]. Diese Einschränkung gilt auch dann, wenn die beteiligten Prozesse eng miteinander verwandt sind, denn ein Kindprozess erbt durch `fork` zwar eine Kopie der unbenannten Semaphore, aber kann dadurch nicht automatisch die Semaphore des Elternprozesses erreichen.

9.4.4

Mutex

Semaphores offer the feature of counting. However, if this feature is not required, a simplified semaphore version, the *mutex*, can be used instead. The mutex concept is used to protect critical sections, which can only be accessed by a single process at any given moment. The term *mutex* is derived from *mut*ual *ex*clusion.

A mutex can only have the two states *occupied* and *not occupied*. Because of this, the functionality of a mutex is identical to that of a *binary semaphore*.

All popular standard libraries implement functions for creating and working with mutexes for thread cooperation. This section gives an overview of these functions from different li-

Mutex

Wird die Möglichkeit einer Semaphore zu zählen nicht benötigt, kann die vereinfachte Version einer Semaphore, der *Mutex*, verwendet werden. Mit dem Konzept des Mutex ist der Schutz kritischer Abschnitte möglich, auf die zu jedem Zeitpunkt immer nur ein Prozess zugreifen darf. Der Begriff *Mutex* ist abgeleitet von *Mut*ual *E*xclusion, also *wechselseitiger Ausschluss*.

Ein Mutex kann nur die beiden Zustände *belegt* und *nicht belegt* annehmen. Aus diesem Grund ist die Funktionalität eines Mutex mit der einer *binären Semaphore* identisch.

Alle gängigen Standardbibliotheken implementieren Funktionen um Mutexe zur Kooperation von Threads zu erstellen und um mit diesen zu arbeiten. Dieser Abschnitt enthält eine Über-

braries. Unfortunately, for the cooperation of processes, these functions are either inappropriate or require shared memory segments.

Table 9.9 contains the function calls from the C standard library for working with mutexes specified in the header file `threads.h` [31, 41, 61].

sicht über diese Funktionen aus unterschiedlichen Bibliotheken. Für die Zusammenarbeit von Prozessen sind diese Funktionen entweder gar nicht, oder nur über den Umweg eines gemeinsamen Speichersegmenten geeignet.

Tabelle 9.9 enthält die in der Header-Datei `threads.h` definierten Funktionsaufrufe aus der C-Standard-Bibliothek zur Arbeit mit Mutexen [31, 41, 61].

Table 9.9: C Function Calls for Mutexes

Function call	Purpose
`mtx_init`	Create mutex
`mtx_unlock`	Release mutex. Similar to the V operation of semaphores.
`mtx_lock`	Lock mutex. Blocking instruction. Similar to the P operation of semaphores
`mtx_trylock`	Lock mutex. Will only be carried out if it does not block the calling thread
`mtx_timedlock`	Lock mutex. Blocking instruction with specified timeout
`mtx_destroy`	Delete mutex

Table 9.10 contains the function calls from the C library `pthread.h`, which implements POSIX threads [12, 39, 93, 115, 120, 126].

Tabelle 9.10 enthält die Funktionsaufrufe aus der Bibliothek `pthread.h` zur Realisierung von POSIX-Threads. [12, 39, 93, 115, 120, 126]

Table 9.10: C Function Calls for POSIX Mutexes

Function call	Purpose
`pthread_mutex_init`	Create mutex
`pthread_mutex_unlock`	Release mutex. Similar to the V operation of semaphores.
`pthread_mutex_lock`	Lock mutex. Blocking instruction. Similar to the P operation of semaphores
`pthread_mutex_trylock`	Lock mutex. Will only be carried out if it does not block the calling thread
`pthread_mutex_timedlock`	Lock mutex. Blocking instruction with specified timeout
`pthread_mutex_destroy`	Delete mutex

Table 9.11 contains the function calls specified in the header file `threads.h` for working with mutexes. It is a part of the C standard library of the Solaris operating system from Oracle (formerly Sun Microsystems) [44, 73, 96, 110].

For the two last-mentioned options (`pthread.h` and `threads.h` in Solaris), it is also possible to use mutexes for the cooperation of threads of different processes. For that to be possible, however, just as before with the POSIX semaphores, the corresponding mutex must reside in a System V memory segment (see Section 9.3.1) or in a POSIX memory

Tabelle 9.11 enthält die in der Header-Datei `threads.h` definierten Funktionsaufrufe zur Arbeit mit Mutexen aus der C-Standard-Bibliothek des Betriebssystems Solaris von Oracle (vormals Sun Microsystems) [44, 73, 96, 110].

Bei den beiden letztgenannten Möglichkeiten (`pthread.h` und `threads.h` unter Solaris) ist es auch möglich, Mutexe zur Kooperation von Threads unterschiedlicher Prozesse zu verwenden. Damit das möglich ist, muss sich aber genau wie zuvor bei den POSIX-Semaphoren der entsprechende Mutex in einem System V-Speichersegment (siehe Abschnitt 9.3.1) oder

Table 9.11: C Function Calls for Mutexes in the Solaris Operating System

Function call	Purpose
`mutex_init`	Create mutex
`mutex_unlock`	Release mutex. Similar to the V operation of semaphores.
`mutex_lock`	Lock mutex. Blocking instruction. Similar to the P operation of semaphores
`mutex_trylock`	Lock mutex. Will only be carried out if it does not block the calling thread
`mutex_destroy`	Delete mutex

segment (see Section 9.3.2) which the involved processes can access [39].

If a thread wants to access the critical section, it calls `mtx_lock` (alternatively: `pthread_mutex_lock` or `mutex_lock`). If the critical section is not locked, the thread can enter. If the critical section is locked, the thread is blocked until the thread in the critical section is finished and calls `mtx_unlock` (alternatively: `pthread_mutex_unlock` or `mutex_unlock`). If multiple threads are waiting for the critical section, randomness decides [115]. In addition, there are non-blocking functions (`mtx_trylock`, `pthread_mutex_trylock` and `mutex_trylock`) and depending on the standard library used, functions with a timeout (`mtx_timedlock` and `pthread_mutex_timedlock`) are also available.

in einem POSIX-Speichersegment (siehe Abschnitt 9.3.2) befinden, auf das die beteiligten Prozesse Zugriff haben [39].

Will ein Thread auf den kritischen Abschnitt zugreifen, ruft er `mtx_lock` (alternativ: `pthread_mutex_lock` oder `mutex_lock`) auf. Ist der kritische Abschnitt nicht gesperrt, kann der Thread eintreten. Ist der kritische Abschnitt gesperrt, wird der Thread blockiert, bis der Thread im kritischen Abschnitt fertig ist und `mtx_unlock` (alternativ: `pthread_mutex_unlock` oder `mutex_unlock`) aufruft. Warten mehrere Threads auf den kritischen Abschnitt, entscheidet der Zufall [115]. Zudem gibt es auch nichtblockierende Funktionen (`mtx_trylock`, `pthread_mutex_trylock` und `mutex_trylock`) und je nach Standardbibliothek auch solche mit Timeout (`mtx_timedlock` und `pthread_mutex_timedlock`).

9.4.5

Monitor

Another concept to protect critical sections is the *monitor* concept, which was developed by Per Brinch Hansen [45] and Tony Hoare [51] in the 1970s. A monitor consists of a data structure and access operations to it, which are summarized as a module [6]. Processes can call the access operations of the monitor, but cannot access the internal data structure of the monitor [35]. The compiler implements the monitor as a resource with exclusive access, using the operating system concepts mutex or semaphore. The compiler ensures that only one process at a time can access the monitor and thus the critical section. It is done transparently for software developers.

Monitor

Ein weiteres Konzept zum Schutz kritischer Abschnitte ist der von Per Brinch Hansen [45] und Tony Hoare [51] in den 1970er Jahren entwickelte *Monitor*. Ein Monitor besteht aus einer Datenstruktur und Zugriffsoperationen darauf, die als Modul zusammengefasst sind [6]. Prozesse können die Zugriffsoperationen des Monitors aufrufen, aber nicht auf die interne Datenstruktur des Monitors zugreifen [35]. Der Compiler realisiert den Monitor als exklusives Betriebsmittel, indem er auf der Ebene des Betriebssystems die Konzepte Mutex oder Semaphore verwendet. Der Compiler stellt sicher, dass zu jedem Zeitpunkt nur ein Prozess auf den Monitor und damit auf dem kritischen Abschnitt zugreifen kann. Für den Softwareentwickler geschieht dies transparent.

A monitor offers similar functionality as a mutex or a binary semaphore. Furthermore, using them is more simple and less error-prone than using semaphores.

Examples for programming languages that offer the monitor concept include Java and Python [7]. In Java, if a method or code section is labeled with the keyword `synchronized`, the runtime environment creates a monitor and ensures exclusive access to it [56]. The access to a monitor or its release is done with the methods `wait`, `notify`, and `notifyAll`. Other languages, such as C and PHP, lack the monitor concept. Here, the software developers have to use semaphores or mutexes to protect critical sections.

For reasons of limited space and because of the similarity to the mutex concept, this work contains no further information about monitors. An in-depth discussion of this topic is provided by [17, 108, 112, 120].

Monitore bieten eine vergleichbare Funktionalität wie Mutexe oder binäre Semaphoren. Zudem ist die Arbeit mit ihnen im Vergleich zu Semaphoren einfacher und weniger fehleranfällig.

Beispiele für Programmiersprachen, die das Monitor-Konzept anbieten, sind Java und Python [7]. Wenn unter Java eine Methode oder ein Codebereich mit dem Schlüsselwort `synchronized` gekennzeichnet ist, erzeugt die Laufzeitumgebung einen Monitor und garantiert den exklusiven Zugriff darauf [56]. Der Zugriff auf einen Monitor bzw. dessen Freigabe geschieht unter Java mit den Methoden `wait`, `notify` und `notifyAll`. Bei anderen Sprachen wie beispielsweise C und PHP fehlt das Monitor-Konzept. Hier müssen die Softwareentwickler zum Schutz kritischer Abschnitte auf Semaphoren oder Mutexe zurückgreifen.

Aus Platzgründen und wegen der großen Ähnlichkeit zum Konzept des Mutex enthält dieses Buch keine weiteren Informationen zu Monitoren. Eine intensive Auseinandersetzung mit diesem Thema bieten [17, 107, 115, 120].

10

Virtualization

Virtualization is a technique that combines resources from a logical perspective so that their utilization can be optimized. The technical term virtualization includes several completely different concepts and technologies.

This chapter introduces virtualization techniques that are relevant to operating systems. These are partitioning, hardware emulation, application virtualization, full virtualization, paravirtualization, hardware virtualization, and operating system virtualization. Other virtualization techniques, such as network or storage virtualization, are not discussed in this book.

Most of the virtualization techniques presented in this chapter allow building and running *virtual machines* (VM). Each VM runs in an isolated environment on physical hardware and behaves like a full-fledged computer system with dedicated components. In a VM, an operating system with applications can run just like on a physical computer. Applications running in a VM do not notice this. Requests from the operating system instances are transparently intercepted by the virtualization software, and converted for the existing physical or emulated hardware.

Virtualisierung

Virtualisierung ist eine Herangehensweise in der Informationstechnik, die Ressourcen so in einer logischen Sicht zusammenfasst, dass ihre Auslastung optimiert werden kann. Das Schlagwort Virtualisierung umfasst mehrere grundsätzlich verschiedene Konzepte und Technologien.

Dieses Kapitel stellt die aus Sicht der Betriebssysteme interessanten Virtualisierungstechniken vor. Bei diesen handelt es sich um Partitionierung, Hardware-Emulation, Anwendungsvirtualisierung, vollständige Virtualisierung, Paravirtualisierung, Hardware-Virtualisierung und Betriebssystem-Virtualisierung. Andere Virtualisierungstechniken wie Netzwerk- oder Speichervirtualisierung behandelt dieses Buch nicht.

Die meisten in diesem Kapitel vorgestellten Virtualisierungstechniken ermöglichen die Erzeugung *virtueller Maschinen* (VM). Jede VM läuft in einer abgeschotteten Umgebung auf einer physischen Hardware und verhält sich wie ein vollwertiger Computer mit eigenen Komponenten. In einer VM kann ein Betriebssystem mit Anwendungen genau wie auf einem realen Computer laufen. Die Anwendungen, die in einer VM laufen, bemerken diesen Umstand nicht. Anforderungen der Betriebssystem-Instanzen werden von der Virtualisierungssoftware transparent abgefangen und auf die real vorhandene oder emulierte Hardware umgesetzt.

10.1

Partitioning

If *partitioning* is used, the total amount of resources can be split to create subsystems of a computer system. Each subsystem can contain an executable operating system instance and can be used in the same way as an indepen-

Partitionierung

Beim Virtualisierungskonzept *Partitionierung* können Teilsysteme auf den Gesamtressourcen eines Computersystems definiert werden. Jedes Teilsystem kann eine lauffähige Betriebssysteminstanz enthalten und verhält sich wie ein

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8_10

dent computer system. The resources (CPU, main memory, storage, ...) are managed by the firmware of the computer that allocates them to the virtual machines. No additional software is required for the implementation of the virtualization functionality (see Figure 10.1).

Partitioning is used, for example, in IBM mainframes (zSeries) or mid-range systems (pSeries) with Power5/6/7/8/9 series CPUs. On such systems, resource allocation is possible during operation without having to reboot. Several hundred to thousands of Linux instances can be run simultaneously on a modern mainframe computer.

eigenständiger Computer. Die Ressourcen (Prozessor, Hauptspeicher, Datenspeicher, etc.) werden von der Firmware des Computers verwaltet, die diese den virtuellen Maschinen zuteilt. Eine zusätzliche Software zur Realisierung der Virtualisierungsfunktionalität ist nicht nötig (siehe Abbildung 10.1).

Partitionierung kommt zum Beispiel bei IBM Großrechnern (zSerie) oder Midrange-Systemen (pSerie) mit Prozessoren der Serien Power5/6/7/8/9 zum Einsatz. Eine Änderung der Ressourcenzuteilung ist auf solchen Systemen im laufenden Betrieb ohne Neustart möglich. Auf einem aktuellen Großrechner können mehrere hundert bis tausend Linux-Instanzen gleichzeitig laufen.

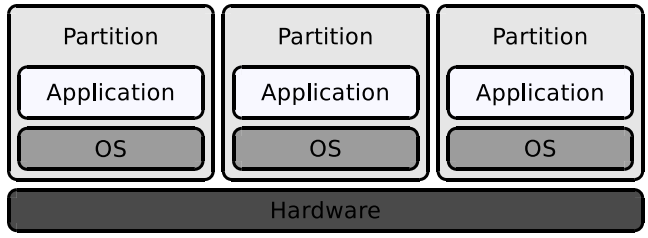

Figure 10.1: Full Partitioning creates Subsystems that behave like standalone Computers

Current x86-compatible CPUs with extensions for virtualization applications, such as Intel Vanderpool (VT-x) and AMD Pacifica (AMD-V), support only the partitioning of the CPU itself and not of the entire system.

In practice, partitioning is only used on mainframes and some servers. It is not used for desktop environments.

Aktuelle x86-kompatible Prozessoren mit Erweiterungen für Virtualisierungsanwendungen, wie zum Beispiel Intel Vanderpool (VT-x) und AMD Pacifica (AMD-V), unterstützen lediglich die Partitionierung des Prozessors selbst und nicht des Gesamtsystems.

Außerhalb von Großrechnern und Servern spielt Partitionierung in der Praxis keine Rolle. Im Desktop-Umfeld wird es nicht verwendet.

10.2

Hardware Emulation

A completely different technology than virtualization is *emulation*. It simulates the entire hardware of a computer system for running an *unmodified operating system*, designed for a *different hardware architecture* (see Figure 10.2).

Hardware-Emulation

Eine vollständig andere Technologie als Virtualisierung ist die *Emulation*. Diese bildet die komplette Hardware eines Rechnersystems nach, um ein *unverändertes Betriebssystem*, das für eine *andere Hardwarearchitektur* ausgelegt ist, zu betreiben (siehe Abbildung 10.2).

One drawback of emulators is their lower performance compared with virtualization solutions.

Ein Nachteil von Emulatoren ist die geringere Ausführungsgeschwindigkeit im Vergleich mit Virtualisierungslösungen.

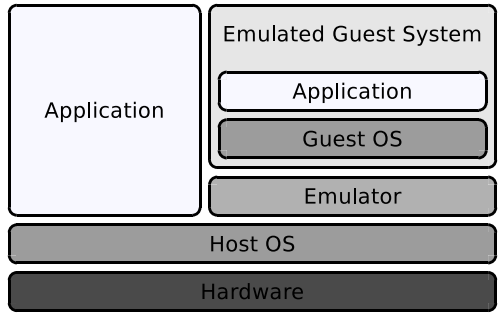

Figure 10.2: Emulation simulates the entire Hardware of a Computer System

Examples for emulators include:

- Bochs. Emulates computers with the CPU architectures x86 and AMD64.
- Basilisk. Emulates the 680x0 CPU architecture to run those versions of Mac OS (up to and including version 8.1) that were intended for 68K Macintosh computers.
- DOSBox. Emulates a computer with x86 CPU and other hardware that is required or helpful for running MS-DOS and Windows 3x.
- JSLinux. Emulates a computer with a 32-bit x86 or with a 64-bit RISC-V CPU. The emulator is written in JavaScript and can run the operating systems Linux, FreeDOS, and Windows 2000 in the browser.
- JSNES. Emulates the NES video game console and is written in JavaScript to run in a browser window.
- MAME. The Multiple Arcade Machine Emulator (MAME) emulates the hardware of classic arcade games.

Beispiele für Emulatoren sind:

- Bochs. Emuliert Computer mit den Prozessorfamilien x86 und AMD64.
- Basilisk. Emuliert die 680x0 Prozessorfamilie, um diejenigen Version von Mac OS (bis einschließlich Version 8.1) zu betreiben, die auf 68K-Macintoshs laufen.
- DOSBox. Emuliert einen Computer mit x86-Prozessor und weitere Hardware, die zum Betrieb von MS-DOS und Windows 3x benötigt wird oder hilfreich ist.
- JSLinux. Emuliert einen Computer mit einem x86-kompatiblen 32-Bit-Prozessor oder einem RISC-V-Prozessor. Der Emulator wurde in JavaScript entwickelt und ermöglicht den Betrieb von Linux, FreeDOS und Windows 2000 im Browser.
- JSNES. Emuliert die NES-Spielkonsole, ist komplett in JavaScript entwickelt und läuft somit im Browser.
- MAME. Der Multiple Arcade Machine Emulator (MAME) emuliert die Hardware klassischer Videospielautomaten.

- PearPC. Emulates a computer with a PowerPC CPU to run the Mac OS X versions for PowerPC CPUs.
- QEMU. Emulates a computer and supports different CPU architectures such as x86, AMD64, and PowerPC.
- SheepShaver. Emulates computers with the PowerPC and 680x0 CPU architecture for running Mac OS 7/8/9.
- Hercules. Emulates the IBM System/360, System/370, System/390, and System z mainframe series.
- Virtual PC. Emulates x86 compatible CPUs in the MacOS X version on the PowerPC architecture.

- PearPC. Emuliert einen Computer mit einem PowerPC-Prozessor, um diejenigen Version von Mac OS X zu betreiben, die auf PowerPC-Prozessoren lauffähig sind.
- QEMU. Emuliert einen Computer und unterstützt verschiedene Prozessorfamilien wie zum Beispiel x86, AMD64 und PowerPC.
- SheepShaver. Emuliert Computer mit den Prozessorfamilien PowerPC und 680x0, um Mac OS 7/8/9 zu betreiben.
- Hercules. Emuliert IBM-Großrechner der Serien System/360, System/370, System/390 und System z.
- Virtual PC. Emuliert in der Version für MacOS X auf der PowerPC-Prozessorarchitektur x86-kompatible Prozessoren.

10.3

Application Virtualization

In *application virtualization*, individual applications run in a virtual environment that provides all the components the application needs. The virtual machine is located between the application and the operating system. An example for this virtualization concept is the Java Virtual Machine (JVM). It is the part of the Java runtime environment (JRE) that executes Java byte code (see Figure 10.3). The compiler `javac` compiles source code into architecture-independent `.class` files that contain byte code, which can run in the Java VM. The `java` program starts a Java application inside an instance of the Java VM.

One advantage of application virtualization is its platform independence. Programs, written in Java, run on all operating systems and hardware architectures for which a port of the JVM exists.

Anwendungsvirtualisierung

Bei der *Anwendungsvirtualisierung* werden einzelne Anwendungen in einer virtuellen Umgebung ausgeführt, die alle Komponenten bereitstellt, die die Anwendung benötigt. Die virtuelle Maschine befindet sich zwischen der auszuführenden Anwendung und dem Betriebssystem. Ein Beispiel für dieses Virtualisierungskonzept ist die Java Virtual Machine (JVM). Diese ist der Teil der Java-Laufzeitumgebung (JRE), der für die Ausführung des Java-Bytecodes verantwortlich ist (siehe Abbildung 10.3). Der Compiler `javac` übersetzt Quellcode in architekturunabhängige `.class`-Dateien, die Bytecode enthalten, der in der Java VM lauffähig ist. Das `java`-Programm startet eine Java-Anwendung in einer Instanz der Java VM.

Ein Vorteil der Anwendungsvirtualisierung ist die Plattformunabhängigkeit. So laufen in Java geschriebene Programme auf allen Betriebssystemen und Hardwarearchitekturen, für die eine

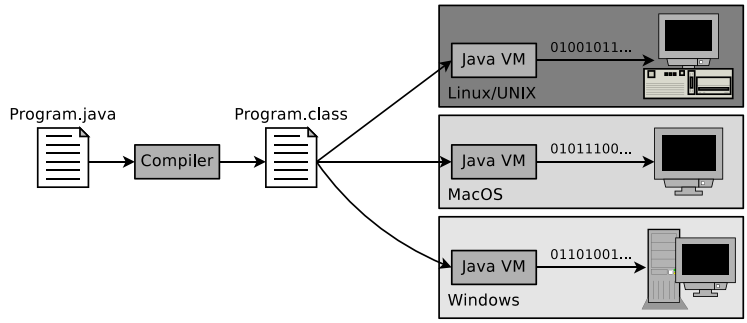

Figure 10.3: The Java Virtual Machine is one Example of Application Virtualization

A drawback is the lower performance compared to native program execution.

Another example of application virtualization is VMware ThinApp, which was sold under the name Thinstall until 2008. The product packs Windows applications into single `exe` files. Applications become portable and can be used without a local installation in the Windows operating system. For example, applications can be run directly from a USB flash memory drive. Furthermore, no entries are made in the Windows registry, and no environment variables and DLL files are created on the system. User preferences and created documents are stored inside a separate sandbox.

Portierung der JVM existiert. Ein Nachteil ist die geringere Ausführungsgeschwindigkeit gegenüber nativer Programmausführung.

Ein weiteres Beispiel für Anwendungsvirtualisierung ist VMware ThinApp, das bis 2008 unter dem Namen Thinstall vertrieben wurde. Diese Lösung ist in der Lage, Windows-Anwendungen in einzelne `exe`-Dateien zu packen. Dadurch ist es möglich, Anwendungen ohne lokale Installation unter Windows-Betriebssystemen auszuführen. Anwendungen können so zum Beispiel direkt von einem USB-Stick ausgeführt werden. Zudem erfolgen keine Einträge in der Windows Registry und es werden keine Umgebungsvariablen und DLL-Dateien auf dem System erstellt. Benutzereinstellungen und erstellte Dokumente speichert diese Lösung in einer eigenen Sandbox.

10.4

Full Virtualization

Full virtualization software solutions offer each virtual machine a complete virtual PC environment, including its own BIOS. Each guest operating system has its own virtual machine with virtual resources like CPU(s), main memory, storage devices, network adapters, etc. The heart of the solution is a so-called *Virtual Machine Monitor* (VMM) that runs *hosted* as an application in the host operating system (see Figure 10.4). The VMM is also called a *Type-2 hypervisor* in literature.

Vollständige Virtualisierung

Vollständige Virtualisierungslösungen bieten einer virtuellen Maschine eine vollständige, virtuelle PC-Umgebung inklusive eigenem BIOS. Jedes Gastbetriebssystem erhält eine eigene virtuelle Maschine mit virtuellen Ressourcen wie Prozessor(en), Hauptspeicher, Laufwerke, Netzwerkkarten, etc. Kern der Lösung ist ein sogenannter *Virtueller Maschinen-Monitor* (VMM), der *hosted* als Anwendung im Host-Betriebssystem läuft (siehe Abbildung 10.4). Der VMM heißt in der Literatur auch *Typ-2-Hypervisor*.

The purpose of the VMM is to allocate hardware resources to virtual machines. Some hardware components are emulated because they were not initially designed for concurrent access by multiple operating systems. One example is network adapters. A side benefit of emulating popular hardware is that it avoids driver issues in the guest operating systems.

Die Aufgabe des VMM ist die Zuweisung der Hardwareressourcen an die virtuellen Maschinen. Teilweise emuliert er auch Hardwarekomponenten, die nicht für den gleichzeitigen Zugriff mehrerer Betriebssysteme ausgelegt ist, wie zum Beispiel Netzwerkkarten. Ein sich nebenbei ergebender Vorteil der Emulation populärer Hardware ist die Vermeidung von Treiberproblemen in den Gastbetriebssystemen.

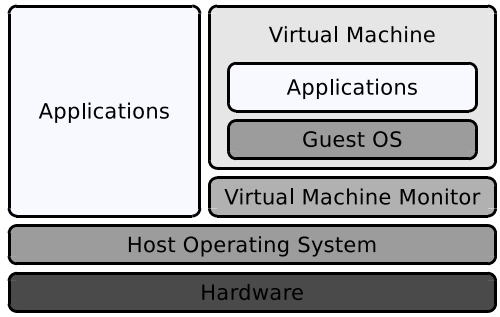

Figure 10.4: In Full Virtualization, the Type-2 Hypervisor allocates Hardware Resources to the Virtual Machines

For an understanding of how the VMM works, a discussion of the virtualization basics of the x86 architecture is useful. x86-compatible CPUs traditionally contain four privilege levels, as described in Section 7.1. Each process is permanently assigned to a ring (see Figure 7.1), and cannot unassign itself from it. In ring 0 (*kernel mode*) is the operating system kernel running (see Section 3.8) and in ring 3 (*user mode*) are the remaining processes running (see Figure 10.5). Only processes in kernel mode have full hardware access. If a process in user mode wants to carry out a task, with higher privileges, for example, accessing hardware, it can notify the kernel via a system call (see Chapter 7). The user mode process generates a software interrupt that is intercepted and handled in ring 0.

Zum Verständnis der Arbeitsweise des VMM ist eine Auseinandersetzung mit den Virtualisierungsgrundlagen der x86-Architektur sinnvoll. x86-kompatible Prozessoren enthalten traditionell, wie in Abschnitt 7.1 bereits beschrieben, vier Privilegienstufen. Jeder Prozess wird in einem Ring (siehe Abbildung 7.1) ausgeführt und kann sich nicht selbstständig aus diesem befreien. Im Ring 0 (*Kernelmodus*) läuft der Betriebssystemkern (siehe Abschnitt 3.8) und im Ring 3 (*Benutzermodus*) laufen die übrigen Prozesse (siehe Abbildung 10.5). Nur Prozesse im Kernelmodus haben vollen Zugriff auf die Hardware. Will ein Prozess im Benutzermodus eine höher privilegierte Aufgabe, zum Beispiel einen Hardwarezugriff durchführen, kann er das dem Betriebssystemkern durch einen Systemaufruf (siehe Kapitel 7) mitteilen. Der Prozess im Benutzermodus erzeugt einen Softwareinterrupt, der in Ring 0 abgefangen und dort behandelt wird.

Full virtualization makes use of the fact that operating systems on x86-compatible CPUs commonly use only two privilege levels. The VMM runs together with the applications in ring 3, and the virtual machines are located in the less privileged ring 1. The VMM intercepts

Vollständige Virtualisierung nutzt die Tatsache, dass Betriebssysteme auf x86-kompatible Prozessoren meist nur zwei Privilegienstufen verwenden. Der VMM läuft wie die Anwendungen in Ring 3 und die virtuellen Maschinen befinden sich im weniger privilegierten Ring 1. Der

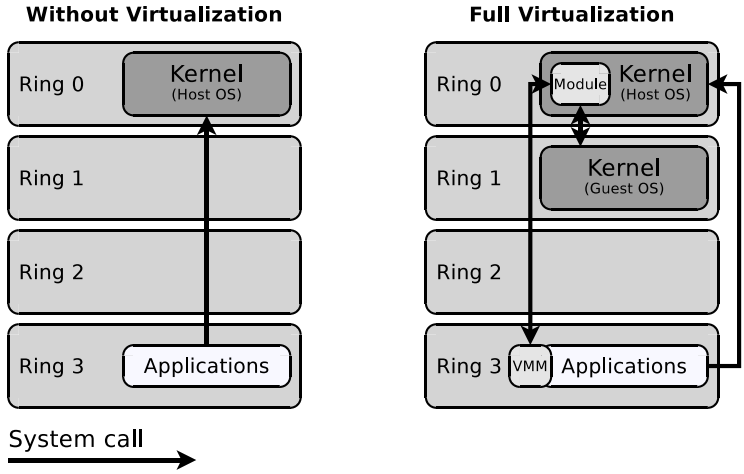

Figure 10.5: Usage of the Privilege Levels of x86-compatible CPUs without Virtualization and with Full Virtualization

the software interrupts of the guest operating system, interprets, and handles them using its interrupt handlers. Virtual machines can only access the hardware via the VMM. This ensures synchronized access to the system resources used by multiple operating systems.

Positive characteristics of the full virtualization concept are that only few modifications to the host and guest operating systems are required. Because the VMM only forwards requests to the essential hardware resources and does not emulate them, guest operating systems can run almost with native performance. Another advantage of this virtualization concept is that each guest operating system has its kernel, which provides flexible deployment options. For example, it is possible to run different versions of a kernel for testing purposes or different operating systems in various virtual machines on the same physical hardware for testing or development purposes.

One drawback is the frequent process switching because each process switching consumes computing time. If an application in the guest operating system requests the execution of a privileged instruction, the VMM intercepts this

VMM fängt die Softwareinterrupts der Gastbetriebssystem ab, interpretiert und behandelt sie mit Hilfe seiner Routinen zur Unterbrechungsbehandlung. Virtuelle Maschinen erhalten nur über den VMM Zugriff auf die Hardware. Das garantiert einen kontrollierten Zugriff auf die von mehreren Betriebssystemen gemeinsam genutzten Systemressourcen.

Positive Aspekte des Konzepts der vollständigen Virtualisierung sind, dass nur geringe Änderungen an den Host- und Gastbetriebssystemen erforderlich sind. Da der VMM Zugriffe auf die wichtigsten Hardwareressourcen nur durchreicht und diese Hardware nicht emuliert, können die Gastbetriebssysteme mit einer fast nativen Verarbeitungsgeschwindigkeit ausgeführt werden. Ein weiterer Vorteil dieses Virtualisierungskonzepts ist, dass jedes Gastbetriebssystem seinen eigenen Betriebssystemkern enthält, was flexible Einsatzmöglichkeiten mit sich bringt. So ist es beispielsweise möglich, zu Testzwecken verschiedene Versionen eines Betriebssystemkerns oder verschiedene Betriebssysteme in diversen virtuellen Maschinen auf einer physischen Hardware zu Test- oder Entwicklungszwecken zu betreiben.

Nachteilig sind die häufigen Prozesswechsel, denn jeder Prozesswechsel verbraucht Rechenzeit. Fordert eine Anwendung im Gastbetriebssystem die Ausführung einer privilegierten Aufgabe an, fängt der VMM diese Anforderung ab

request and instructs the kernel of the host operating system to carry it out.

Some examples of virtualization products that use the VMM concept are Kernel-based Virtual Machine (KVM), Mac-on-Linux, Microsoft Virtual PC (the version that runs on the x86 architecture), Oracle VirtualBox, Parallels Desktop, Parallels Workstation, VMware Server, VMware Workstation, and VMware Fusion.

10.5

Paravirtualization

In *paravirtualization*, the guest operating systems use an abstract management layer, the hypervisor, to access the physical resources. This type of hypervisor is a *Type-1 hypervisor*, which directly runs on the physical hardware, without a host operating system between them. The hypervisor distributes the hardware resources among the guest systems in the same way as an operating system does it among the running processes.

The hypervisor runs in the privileged ring 0. A host operating system is mandatory because of the device drivers. The host operating system no longer runs in ring 0 but in the less privileged ring 1. Since the kernel of the host operating system can no longer execute privileged instructions due to its position in ring 1, the hypervisor provides *hypercalls*. These are similar to system calls, but their interrupt numbers are different. When an application requests a system call, a replacement function in the hypervisor is called. The hypervisor then orders the execution of the corresponding system call at the kernel of the host operating system (see Figure 10.6).

One challenge of paravirtualization is that the kernels of guest operating systems need to be modified in a way that any system call for direct hardware access is replaced by the corresponding hypercall, which is typically only possible for operating systems that are free software. One benefit of this virtualization concept is that the interception and verification of system calls

und weist deren Ausführung beim Betriebssystemkern des Host-Betriebssystems an.

Beispiele für Virtualisierungslösungen, die auf dem Konzept des VMM basieren, sind Kernel-based Virtual Machine (KVM), Mac-on-Linux, Microsoft Virtual PC (in der Version für x86), Oracle VirtualBox, Parallels Desktop, Parallels Workstation, VMware Server, VMware Workstation und VMware Fusion.

Paravirtualisierung

Bei der *Paravirtualisierung* verwenden die Gastbetriebssysteme eine abstrakte Verwaltungsschicht, den Hypervisor, um auf physische Ressourcen zuzugreifen. Der Hypervisor ist in diesem Fall ein sogenannter *Typ-1-Hypervisor*, der direkt (englisch: *bare metal*) auf der Systemhardware ohne ein dazwischenliegendes Host-Betriebssystem läuft und die Hardwareressourcen unter den Gastsystemen verteilt, so wie ein Betriebssystem dies unter den laufenden Prozessen tut.

Der Hypervisor läuft im privilegierten Ring 0. Ein Host-Betriebssystem ist wegen der Gerätetreiber zwingend nötig. Dieses läuft nicht mehr in Ring 0, sondern im weniger privilegierten Ring 1. Da der Kern des Host-Betriebssystems durch seine Position in Ring 1 keine privilegierten Anweisungen mehr ausführen kann, stellt der Hypervisor sogenannte *Hypercalls* zur Verfügung. Diese sind vergleichbar mit Systemaufrufen, aber die Nummern der Softwareinterrupts sind verschieden. Fordert eine Anwendung die Ausführung eines Systemaufrufs an, wird eine Ersatzfunktion im Hypervisor aufgerufen. Der Hypervisor weist dann die Ausführung des entsprechenden Systemaufrufs beim Kern des Host-Betriebssystems an (siehe Abbildung 10.6).

Problematisch ist, dass in den Kernen der Gast-Betriebssysteme alle Systemaufrufe für Hardware-Zugriffe durch die entsprechenden Hypercall-Aufrufe ersetzt werden müssen, was in der Regel nur bei Betriebssystemen möglich ist, die als freie Software vorliegen. Ein Vorteil dieses Virtualisierungskonzepts ist, dass das Abfangen und Prüfen der Systemaufrufe durch

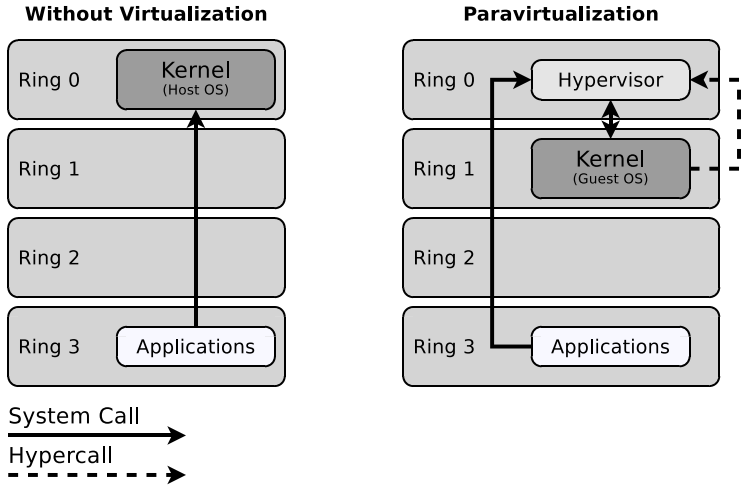

Figure 10.6: A Host Operating System is mandatory for Paravirtualization because of the Device Drivers

by the hypervisor only cause little performance loss.

Examples of virtualization products that implement paravirtualization include Xen, Citrix Xenserver, Virtual Iron, and VMware ESX Server.

den Hypervisor nur zu geringen Geschwindigkeitseinbußen führt.

Beispiele für Virtualisierungslösungen, die auf dem Konzept der Paravirtualisierung basieren, sind Xen, Citrix Xenserver, Virtual Iron und VMware ESX Server.

10.6

Hardware Virtualization

Up-to-date x86-compatible CPUs from Intel and AMD implement extensions for hardware virtualization. One advantage of these extensions is that unmodified operating systems can be used as guest operating systems. The solutions from Intel and AMD are similar, but incompatible to each other. Since 2006, AMD64 CPUs include the Secure Virtual Machine Command Set (SVM). This feature is called *AMD-V* and had previously been named *Pacifica*.

The solution from Intel is called *VT-x* and was previously called *Vanderpool*.

The extension resulted in a modification of the privilege levels by adding ring -1 for the hypervisor (see Figure 10.7). The hypervisor or VMM runs in ring -1, and at all times has total control over the CPU and the other hardware

Hardware-Virtualisierung

Aktuelle x86-kompatible Prozessoren von Intel und AMD enthalten Erweiterungen, um Hardware-Virtualisierung zu ermöglichen. Ein Vorteil dieser Erweiterungen ist, dass unveränderte Betriebssysteme als Gast-Systeme ausgeführt werden können. Die Lösungen von Intel und AMD sind ähnlich, aber inkompatibel zueinander. Seit 2006 enthalten AMD64-Prozessoren den Secure-Virtual-Machine-Befehlssatz (SVM). Diese Lösung heißt *AMD-V* und war vorher als *Pacifica* bekannt.

Die Lösung von Intel heißt *VT-x* und war zuvor unter dem Stichwort *Vanderpool* bekannt.

Die Erweiterung führte zu einer Überarbeitung der Privilegienstruktur, da ein neuer Ring -1 für den Hypervisor hinzugefügt wurde (siehe Abbildung 10.7). Der Hypervisor bzw. VMM läuft im Ring -1 und besitzt jederzeit die volle

resources, because ring -1 has a higher privilege than ring 0.

The virtual machines run in ring 0 and are called Hardware Virtual Machine (HVM) in such a scenario.

An advantage of hardware virtualization is that guest operating systems do not need to be modified. This allows for proprietary operating systems, such as Windows, to be run as guest operating systems.

Kontrolle über den Prozessor und die übrigen Hardwareressourcen, da mit Ring -1 ein höheres Privileg als Ring 0 existiert.

Die virtuellen Maschinen laufen in Ring 0 und heißen in einem solchen Kontext auch Hardware Virtual Machine (HVM).

Ein Vorteil der Hardware-Virtualisierung ist, dass Gastbetriebssysteme nicht angepasst sein müssen. Dadurch laufen auch proprietäre Betriebssysteme wie zum Beispiel Windows als Gastsysteme.

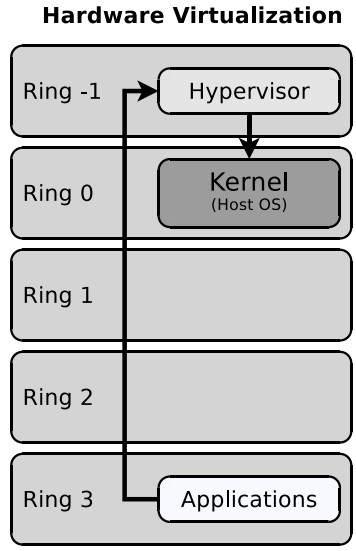

Figure 10.7: With Hardware Virtualization, the Hypervisor runs in the new Ring -1, and the Host Operating System returns to Ring 0

Some examples of virtualization products that support hardware virtualization are Xen since version 3, Windows Server since version 2008 (Hyper-V), VirtualBox and KVM.

Beispiele für Virtualisierungslösungen, die Hardware-Virtualisierung unterstützen, sind Xen seit Version 3, Windows Server ab Version 2008 (Hyper-V), VirtualBox und KVM.

10.7

Operating System-level Virtualization

Betriebssystem-Virtualisierung

With *operating system virtualization*, several identical system environments can be run under the same kernel, isolated from each other. These system environments are usually called *containers* and, in rare cases, *jails* (see Figure 10.8).

Bei der *Betriebssystem-Virtualisierung* können unter ein und demselben Betriebssystemkern mehrere voneinander abgeschottete identische Systemumgebungen laufen, die in der Regel *Container* und seltener *Jails* genannt werden

When launching a virtual machine, in contrast to full virtualization, paravirtualization, and emulation, no additional operating system is started. Instead, an isolated runtime environment is created. Thus, all containers use the same kernel.

Applications that run in a container can only get in touch with applications within the same container. One advantage of this virtualization concept is the reduced overhead because the kernel manages the hardware as usual. Depending on the specific application, the restriction of having just a single kernel may be a limitation, since operating system virtualization does not allow different operating systems to be used in parallel on the same physical hardware. Only independent instances of the same operating system can be started.

(siehe Abbildung 10.8). Beim Start einer virtuellen Maschine wird also im Gegensatz zur vollständigen Virtualisierung, Paravirtualisierung und Emulation kein zusätzliches Betriebssystem gestartet, sondern eine isolierte Laufzeitumgebung erzeugt. Aus diesem Grund verwenden alle Container denselben Betriebssystemkern.

Anwendungen, die in einem Container laufen, sehen nur Anwendungen im gleichen Container. Ein Vorteil dieses Virtualisierungskonzepts ist der geringe Verwaltungsaufwand, da der Betriebssystemkern in gewohnter Weise die Hardware verwaltet. Je nach konkretem Anwendungsfall ist die Beschränkung auf nur einen Betriebssystemkern eine Einschränkung, da es bei Betriebssystem-Virtualisierung nicht möglich ist, verschiedene Betriebssysteme gleichzeitig auf einer physischen Hardware zu verwenden. Es werden nur unabhängige Instanzen eines Betriebssystems gestartet.

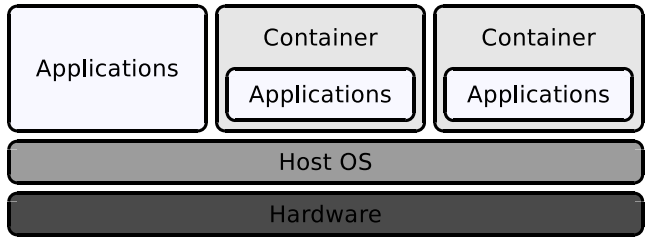

Figure 10.8: With Operating System Virtualization, the Operating System can create identical System Environments (Containers) that are isolated from each other

This virtualization concept is particularly useful in situations where applications need to run in isolated environments with high security. One application example is Internet service providers that offer (virtual) root servers or web services on multicore CPU architectures. Another use case is the automated installation of sophisticated application software, such as a web server or a database, without having to consider package dependencies on the operating system and without having to modify it.

Examples of virtualization products that implement operating system virtualization include Docker, the Solaris operating system from Oracle (previously Sun Microsystems), OpenVZ for Linux, Linux-VServer, the FreeBSD operating

Dieses Virtualisierungskonzept ist besonders da hilfreich, wo Anwendungen in isolierten Umgebungen mit hoher Sicherheit betrieben werden sollen. Ein Anwendungsbeispiel sind Internet-Service-Provider, die (virtuelle) Root-Server oder Webdienste auf Mehrkernprozessorarchitekturen anbieten. Ein weiterer Anwendungsfall ist die automatisierte Installation komplexer Anwendungssoftware wie eines Web-Servers oder einer Datenbank, ohne auf Paketabhängigkeiten auf dem Betriebssystem Rücksicht nehmen zu müssen und ohne dieses zu verändern.

Beispiele für Virtualisierungslösungen, die Betriebssystem-Virtualisierung realisieren, sind Docker, das Betriebssystem Solaris von Oracle (vormals Sun Microsystems), OpenVZ für Linux, Linux-VServer, das Betriebssystem FreeBSD,

system, Virtuozzo (the commercial variant of OpenVZ), and FreeVPS.

Virtuozzo (die kommerzielle Variante von OpenVZ) und FreeVPS.

Glossary

Adressbus Memory addresses and I/O devices are addressed via the address bus

AES The symmetric Advanced Encryption Standard belongs to the group of block ciphers and is the successor of DES and Triple-DES. The key length for AES is 128, 192, or 256 bits. Depending on the key length, the method uses 10, 12, or 14 encryption rounds

ASCII American Standard Code for Information Interchange. 7-bit character encoding

see Swap

Batch processing is an operation mode in which each program must be provided entirely with all input data before the execution can begin. Usually, batch processing works in a non-interactive way

see User mode

Glossar

Adressbus Speicheradressen und Peripherie-Geräte werden über den Adressbus angesprochen (adressiert)

AES Das symmetrische Verschlüsselungsverfahren Advanced Encryption Standard gehört zur Gruppe der Blockchiffren und ist der Nachfolger von DES und Triple-DES. Die Schlüssellänge bei AES ist 128, 192 oder 256 Bit. Je nach Schlüssellänge sieht das Verfahren 10, 12 oder 14 Verschlüsselungsrunden vor

ASCII American Standard Code for Information Interchange. 7-Bit-Zeichenkodierung

Auslagerungsspeicher Ein Speicherbereich (Datei oder Partition) auf einer Festplatte oder SSD, in den das Betriebssystem bei vollem Hauptspeicher diejenigen Prozesse auslagert, die gegenwärtig keinen Zugriff auf einen Prozessor bzw. einen Prozessorkern haben

Batchbetrieb siehe Stapelbetrieb

Benutzermodus Hier laufen alle Prozesse außer dem Betriebssystemkern. Prozesse im Benutzermodus arbeiten ausschließlich mit virtuellem Speicher

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8

see Kernel

Betriebssystemkern Zentrale Komponente eines Betriebssystems, dessen Funktionen die Kernfunktionalitäten wie Benutzer-, Hardware-, Prozess- und Datenverwaltung ermöglichen

BIOS The *Basic Input/Output System* is the firmware of computer systems with x86-compatible CPUs. It is executed immediately after the computer is powered on and, among other things, starts the operating system. On modern computer systems with 64-bit CPUs, the successor of the BIOS, the *Unified Extensible Firmware Interface* (UEFI), is used nowadays

BIOS Das *Basic Input/Output System* ist die Firmware von Computersystemen mit x86-kompatiblen Prozessoren. Es wird direkt nach dessen Einschalten des Computers ausgeführt und leitet unter anderem den Start des Betriebssystems ein. Auf modernen Computern mit 64 Bit-Prozessoren wird heute in der Regel der Nachfolger des BIOS, das *Unified Extensible Firmware Interface* (UEFI) verwendet

Binary system Place value system that uses 2 as base

Binärsystem siehe Dualsystem

Bit Smallest possible unit of information. Two possible states

Bit Kleinstmögliche Informationseinheit. Zwei mögliche Zustände

Block see Sector

Block Siehe Sektor

BSS Block Started by Symbol. This memory area of a process contains the global variables and the variables that are local and static and are not initialized at the start of the process. The content of the BSS segment is read from the program file when the process is created

BSS Block Started by Symbol. Speicherbereich eines Prozesses, der die globalen Variablen und die lokalen statischen Variablen enthält, die beim Start des Prozesses nicht initialisiert werden. Der Inhalt des BSS wird bei der Prozesserzeugung aus der Programmdatei gelesen

Btrfs A free file system that is part of the Linux kernel since 2013. Because of its many advanced features such as snapshots, built-in compression, built-in software RAID (see Section 4.5) and copy-on-write (see Section 6.6), Btrfs may become the next Linux standard file system in the future

Btrfs Ein freies Dateisystem, das seit 2013 im Linux-Kernel enthalten ist. Wegen der zahlreichen modernen Fähigkeiten wie Schnappschüsse (englisch: *Snapshots*), integrierte Datenkompression, integriertes Software-RAID (siehe Abschnitt 4.5) und Copy-on-Write (siehe Abschnitt 6.6) könnte Btrfs in Zukunft das nächste Standard-Dateisystem unter Linux werden

Buffer cache see Page cache

siehe Page Cache

Bus Physical interconnection between hardware components. Computer systems contain several different bus systems

Bus Physische Verbindung zwischen Hardwarekomponenten. Computersysteme enthalten zahlreiche unterschiedliche Bussysteme

Byte Group of 8 bits

Byte Gruppe von 8 Bits

Cache A fast buffer memory. Modern computer systems have several caches in the CPU, on the mainboard, and in storage devices. Modern operating systems also implement a file system cache (page cache) in main memory

Cache Ein schneller Puffer-Speicher. Moderne Computersysteme enthalten verschiedene Caches im Prozessor, auf dem Mainboard und in den Speicherlaufwerken. Moderne Betriebssysteme realisieren zudem einen Dateisystem-Cache (Page Cache) im Hauptspeicher

Cluster Groups of sectors/blocks of a fixed size. The smallest allocation unit that modern operating systems address on storage devices. Each cluster in the file system can only be assigned to a single file

Cluster Verbünde von Sektoren/Blöcken mit fester Größe. Die kleinste Zuordnungseinheit, die moderne Betriebssysteme auf Speicherlaufwerken ansprechen. Jede Datei belegt eine ganzzahlige Anzahl von Clustern

Container Isolated system environment that is used to isolate processes in operating system virtualization

Container Abgeschottete Systemumgebung bei Betriebssystem-Virtualisierung, um Prozesse zu isolieren

Context switch A switch from user mode to kernel mode

siehe Moduswechsel

Control bus Commands (e.g., read and write instructions) from the CPU and status messages from the I/O devices are transmitted by the control bus. It also contains lines that are used by I/O devices to transmit interrupt requests to the CPU

siehe Steuerbus

CPU Central Processing Unit. Central component of a computer system. The CPU executes the machine instructions of the currently running program step by step

siehe Prozessor

Cylinder All tracks on all disks of a hard disk drive at a specific arm position

siehe Zylinder

see File system

Dateisystem Komponente des Betriebssystems, die die strukturierte Ablage der Dateien auf Speicherlaufwerken organisiert

see Page cache

Dateisystem-Cache Ein vom Betriebssystemkern verwalteter Cache im Hauptspeicher, um häufig verwendete Dateien auf lokalen Speicherlaufwerken

Data bus Transmits data between CPU, main memory and I/O devices

Datenbus Überträgt Daten zwischen CPU, Arbeitsspeicher und Peripherie

Data segment The memory area of a process that contains the global variables and the variables that are local and static and are initialized at the start of a process. The content of the data segment is read from the program file when the process is created

Datensegment Speicherbereich eines Prozesses, der die globalen Variablen und die lokalen statischen Variablen enthält, die beim Start des Prozesses initialisiert werden. Der Inhalt des Datensegments wird bei der Prozesserzeugung aus der Programmdatei gelesen

Deadlock Two or more processes wait for resources that are locked by each other

Deadlock Zwei oder mehr Prozesse warten auf die von ihnen gesperrten Ressourcen und sperren sich gegenseitig

Decimal system Place value system that uses 10 as base

Dezimalsystem Stellenwertsystem mit der Basis 10

see Time-sharing

Dialogbetrieb Mehrere Benutzer können gleichzeitig über Dialogstationen an einem Großrechner arbeiten

Dispatcher The kernel component that carries out the state transitions of the processes

Siehe Prozessumschalter

DMA Direct Memory Access. Data is directly transferred between the main memory and I/O device via a DMA controller

DMA Daten werden über einen DMA-Controller direkt zwischen Arbeitsspeicher und E/A-Gerät übertragen

Driver A set of kernel functions that allows accessing a hardware component

siehe Treiber

see Binary system

Dualsystem Stellenwertsystem mit der Basis 2

see Real-time operation mode

Echtzeitbetrieb Mehrprogrammbetrieb mit zusätzlichen Echtzeit-Funktionen (z.B. Einhaltung von Zeitschranken)

see Single-user mode

Einzelbenutzerbetrieb Der Computer steht immer nur einem einzigen Benutzer zur Verfügung

see Singletasking

Einzelprogrammbetrieb Zu jedem Zeitpunkt kann nur ein einziges Programm laufen

Emulator Simulates the entire hardware of a computer system to run an unmodified operating system that is designed for a different hardware architecture

Emulator Bildet die komplette Hardware eines Rechnersystems nach, um ein unverändertes Betriebssystem, das für eine andere Hardwarearchitektur ausgelegt ist, zu betreiben

FAT File Allocation Table. A data structure that contains a record for each cluster of the file system, indicating whether the cluster is free, the medium is defective at this point, or a file occupies the cluster

FAT File Allocation Table. Datenstruktur, die für jeden Cluster des Dateisystems einen Eintrag enthält, der vermerkt, ob der Cluster frei, das Medium an dieser Stelle beschädigt oder der Cluster von einer Datei belegt ist

Front-Side-Bus The bus between CPU and chipset. It includes the address bus, the data bus, and the control bus

Front-Side-Bus Der Bus zwischen Prozessor und Chipsatz. Er enthält den Adressbus, den Datenbus und den Steuerbus

GNU GNU's not Unix. Since the mid-1980s, Richard Stallman has been the driving force behind this project, which aims to develop a free, Unix-like operating system. Since that time, a large number of free software tools and compilers have been developed, which were essential for the development of the Linux operating system

GNU GNU's not Unix. Ein seit Mitte der 1980er Jahre maßgeblich von Richard Stallman vorangetriebenes Projekt mit dem Ziel, ein freies, Unix-ähnliches Betriebssysteme zu entwickeln. Seit dieser Zeit ist eine große Anzahl an freien Werkzeugen und Compilern entstanden, ohne die die Entwicklung des Betriebssystems Linux kaum möglich gewesen wäre

see Main memory

Hauptspeicher Der Speicher eines Computers der gemäß der Von-Neumann-Architektur zur Speicherung der Programme und Daten verwendet wird

see Swap

Hintergrundspeicher siehe Auslagerungsspeicher

Heap The dynamic memory of a process. Processes can request memory areas here and free them later in any order

Heap Der dynamische Speicher eines Prozesses. Prozesse können hier Speicherbereiche anfordern und in beliebiger Reihenfolge wieder freigeben

Hexadecimal system Place value system that uses 16 as base

Hexadezimalsystem Stellenwertsystem mit der Basis 16

Hybrid kernel A kernel that contains more functions than necessary, i.e., more functions than a microkernel

Hybridkernel Ein Betriebssystemkern, der mehr Funktionen enthält als zwingend nötig, also mehr Funktionen enthält als ein Mikrokernel

Hypervisor A software for working with virtual machines. The hypervisor allocates the hardware resources to the virtual machines and emulates hardware components where it makes sense and where parallel access by different operating systems is not possible

Hypervisor Software zur Realisierung virtueller Maschinen. Der Hypervisor weist die Hardwareressourcen den virtuellen Maschinen zu und emuliert Hardwarekomponenten dort, wo es sinnvoll ist und wo ein gleichzeitiger Zugriff durch verschiedene Betriebssysteme nicht möglich ist

Inode Index node. A data structure in the file system that exists for each file and that contains all metadata of the file, except the file name

Inode Indexknoten. Eine Datenstruktur im Dateisystem, die für jede Datei erzeugt wird und die alle Verwaltungsdaten (Metadaten) der Datei außer dem Dateinamen enthält

ISO International Organization for Standardization

ISO Internationale Organisation für Normung

Interrupt A signal that instructs the CPU to interrupt the running process

Interrupt Eine Unterbrechungsanforderung, die den Prozessor anweist, den laufenden Prozess zu unterbrechen

JFS The Journaled File System is a file system that had initially been developed by IBM. It is free software since 1999 and part of the Linux kernel since 2002

JFS Das Journaled File System ist ein ursprünglich von IBM entwickeltes Dateisystem, das seit 1999 freie Software und seit 2002 im Linux-Kernel enthalten ist

Journal Data structure in which modern file systems often collect write operations before they are carried out

Journal Datenstruktur, in der in moderne Dateisysteme häufig Schreibzugriffe vor ihrer Durchführung sammeln

see Page

Kachel siehe Seite

Kernel Central component of an operating system that implements the essential functions such as user, hardware, process and data management

siehe Betriebssystemkern

Kernel mode This is where the kernel runs. Processes running in kernel mode have full hardware access

Kernelmodus Hier läuft der Betriebssystemkern. Prozesse, die im Kernelmodus laufen, haben vollen Zugriff auf die Hardware

Linux A free, Unix-like operating system. The essential components of Linux are the Linux kernel, whose development was initiated by Linus Torvalds in the early 1990s, and a collection of free software tools and compilers, the so-called GNU tools

Linux Ein freies, Unix-ähnliches Betriebssystem. Die wichtigsten Komponenten von Linux sind der von Linus Torvalds Anfang der 1990er Jahre initiierte Linux-Betriebssystemkern sowie eine Sammlung freier Werkzeuge und Compiler, die sogenannten GNU-Tools

Main memory The computer memory which is used according to the Von Neumann architecture for storing programs and data

siehe Hauptspeicher

see Multi-user mode

Mehrbenutzerbetrieb Mehrere Benutzer können gleichzeitig mit dem Computer arbeiten

see Multitasking

Mehrprogrammbetrieb Mehrere Programme können gleichzeitig mit dem Computer arbeiten

Microkernel A minimal kernel that only contains the most essential functions for memory and process management, synchronization, and interprocess communication. Device drivers, file system drivers, and all other functions run outside the kernel in user mode

Mikrokernel Ein minimaler Betriebssystemkern, der nur die nötigsten Funktionen zur Speicher- und Prozessverwaltung sowie zur Synchronisation und Interprozesskommunikation enthält. Gerätetreiber, Treiber für Dateisysteme und alle weiteren Funktionalitäten laufen außerhalb des Kerns im Benutzermodus

Message queue Form of interprocess communication. Linked list in which processes store messages according to the FIFO principle and from which they can fetch messages

siehe Nachrichtenwarteschlange

see Context switch

Moduswechsel Sprung vom Benutzermodus in den Kernelmodus

Multi-user mode Multiple users can work with the computer at the same time

siehe Mehrbenutzerbetrieb

Multitasking Multiple programs can run on the computer at the same time

siehe Mehrprogrammbetrieb

Mutex Enables mutual exclusion to protect critical sections that can only be accessed by a single process at a time. A simplified version of the semaphore concept

Mutex Ermöglicht wechselseitigen Ausschluss zum Schutz kritischer Abschnitte, auf die zu jedem Zeitpunkt immer nur ein Prozess zugreifen darf. Vereinfachte Version des Semaphoren-Konzepts

see Message queue

Nachrichtenwarteschlange Form der Interprozesskommunikation. Verkettete Liste, in die Prozesse nach dem FIFO-Prinzip Nachrichten ablegen und aus der sie Nachrichten abholen können

Nibble Group of 4 bits, also called half-byte

Nibble Gruppe von 4 Bits bzw. ein Halbbyte

Octal system Place value system that uses the number 8 as base

Oktalsystem Stellenwertsystem mit der Basis 8

Octet see Byte

Oktett siehe Byte

Page Cache A cache in main memory that is managed by the kernel to buffer the files that are frequently used on local storage drives

siehe Dateisystem-Cache

Page A set of memory locations, which is specified by the hardware architecture. The main memory is partitioned into pages of equal size

siehe Seite

Page table The operating system maintains a page table for each process for converting the virtual addresses of the process into physical memory addresses

siehe Seitentabelle

Paging A concept of organization that implements the virtual memory of processes as pages of the same length. The operating system manages a page table for each process

Paging Organisationskonzept, das den virtuellen Speicher der Prozesse in Form von Seiten gleicher Länge realisiert. Das Betriebssystem verwaltet für jeden Prozess eine Seitentabelle

PID Process ID. Unique process identifier assigned by the operating system kernel to each process when it is created

PID Process ID. Eindeutige Prozessnummer, die der Betriebssystemkern jedem Prozess bei dessen Erzeugung zuweist

Pipe Form of interprocess communication. A pipe is a communication channel that implements a buffered data stream between two processes

Pipe Form der Interprozesskommunikation. Eine Pipe ist ein Kommunikationskanal, die einen gepufferten Datenstrom zwischen zwei Prozessen realisiert

PPID Parent Process ID. Process identifier of the parent process of a process

PPID Parent Process ID. Prozessnummer des Elternprozesses eines Prozesses

Program A sequence of instructions in a programming language to perform tasks using a computer

Programm Folge von Anweisungen in einer Programmiersprache, um Aufgaben mithilfe eines Computers zu bearbeiten

Protected Mode In this memory management concept, the processes do not use physical main memory addresses, but virtual memory instead

Protected Mode In diesem Konzept der Speicherverwaltung verwenden die Prozesse keine physischen Hauptspeicheradressen, sondern virtuellen Speicher

Protocol Agreement of communication rules

Protokoll Vereinbarung von Kommunikationsregeln

Process An instance of a program that is executed

Prozess Eine Instanz eines Programms, das ausgeführt wird

see CPU

Prozessor Zentrale Komponente eines Computers. Der Hauptprozessor führt die Maschineninstruktionen des aktuell laufenden Programms Schritt für Schritt aus

see Dispatcher

Prozessumschalter Komponente des Betriebssystemkerns, die die Zustandsübergänge der Prozesse durchführt

RAID Redundant Array of Independent Disks. Multi-drive array that is experienced by users and their processes as a single large drive, offering better performance and/or availability than a single drive.

RAID Redundant Array of Independent Disks. Verbund mehrerer Speicherlaufwerke, der durch die Benutzer und deren Prozesse wie ein einziges großes Laufwerk wahrgenommen wird und eine bessere Geschwindigkeit und/oder Ausfallsicherheit bietet als ein einzelnes Laufwerk

RAM Random access memory. The main memory of a computer system

RAM Random Access Memory. Der Hauptspeicher (Arbeitsspeicher) eines Computersystems. Ein Speicher mit wahlfreiem Zugriff

Real Mode In this memory addressing concept, the processes use physical main memory addresses

Real Mode In diesem Konzept der Speicheradressierung verwenden die Prozesse physische Hauptspeicheradressen

Real-time operation mode Multitasking with additional real-time functions (e.g., compliance with time limits)

siehe Echtzeitbetrieb

Register Memory areas in the CPU. The CPU contains data registers, address registers, stack registers, and several special registers

Register Speicherbereiche im Prozessor. Der Hauptprozessor enthält unter anderem Datenregister, Adressregister, Stapelregister und diverse Spezialregister

ReFS Windows 8/10 and Windows Server 2012/2016 include the *Re*silient *F*ile *S*ystem (ReFS). However, these operating systems only allow using ReFS for a few purposes, such as software RAID. Nevertheless, ReFS is expected to be the future standard file system of the Windows operating system family and the successor of NTFS. Microsoft is quite secretive about the way ReFS works. The author of this work is not aware of a comprehensive specification of this file system. For this reason, ReFS is not further discussed in this book

ReFS Windows 8/10 und Windows Server 2012/2016 enthalten das *Re*silient *F*ile *S*ystem (ReFS). Die Betriebssysteme erlauben die Verwendung von ReFS aber bislang nur für wenige Anwendungszwecke, wie zum Beispiel Software-RAID. Dennoch gilt ReFS als zukünftiges Standard-Dateisystem der Windows-Betriebssystemfamilie und als Nachfolger von NTFS. Die Firma Microsoft hält sich bezüglich der Arbeitsweise von ReFS sehr bedeckt. Eine vollständige Spezifikation dieses Dateisystems ist dem Autor dieses Werks nicht bekannt. Aus diesem Grund wird ReFS in diesem Werk nicht weiter thematisiert

ROM A non-volatile read-only memory

ROM Read Only Memory. Ein nicht-flüchtiger Lesespeicher

Sector Smallest accessible unit on magnetic storage devices such as hard disk drives and floppy disks. The size is typically 512 bytes or 4096 bytes for modern drives

Sektor Kleinste zugreifbare Einheit auf magnetischen Datenspeichern wie zum Beispiel Festplatten und Disketten. Typischerweise 512 Byte oder bei modernen Laufwerken 4096 Byte groß

Scheduling Automatic calculation of an execution plan (schedule) of processes by the operating system

Scheduling Automatische Erstellung eines Ablaufplanes der Prozesse durch das Betriebssystem

see Page	**Seite** Eine durch die Hardwarearchitektur definierte Menge von Speicherstellen. Der Hauptspeicher wird in gleich große Speicherseiten unterteilt.
see Page table	**Seitentabelle** Das Betriebssystem verwaltet für jeden Prozess eine Seitentabelle, mit deren Hilfe es die virtuellen Adressen des Prozesses in physische Speicheradressen umwandelt
Segmentation A concept of organization that implements the virtual memory of processes as segments of different lengths. The operating system manages a segment table for each process	**Segmentierung** Organisationskonzept, das den virtuellen Speicher der Prozesse in Form von Segmenten unterschiedlicher Länge realisiert. Das Betriebssystemen verwaltet für jeden Prozess eine Segmenttabelle
Semaphore An integer, non-negative counter variable that can allow multiple processes to enter a critical section	**Semaphore** Ganzzahlige, nichtnegative Zählersperre, die mehreren Prozessen das Betreten eines kritischen Abschnitts erlauben kann
Single-user mode The computer can only be used by a single user at any time	siehe Einzelbenutzerbetrieb
Singletasking Only a single program can be run at any time	siehe Einzelprogrammbetrieb
Socket Platform-independent, standardized interface between the implementation of the network protocols in the operating system and the applications. A socket consists of a port number and an IP address	**Socket** Plattformunabhängige, standardisierte Schnittstelle zwischen der Implementierung der Netzwerkprotokolle im Betriebssystem und den Anwendungen. Ein Socket besteht aus einer Portnummer und einer IP-Adresse
SSD Solid State Drive. Non-volatile electronic storage device	**SSD** Solid State Drive. Nichtflüchtiges elektronisches Speichermedium
Stack Enables nested function calls. With each function call, the call parameters, return address, and a pointer to the calling function are added to the stack. Furthermore, the functions add their local variables onto the stack	**Stack** Ermöglicht geschachtelte Funktionsaufrufe. Mit jedem Funktionsaufruf werden die Aufrufparameter, Rücksprungadresse und ein Zeiger auf die aufrufende Funktion auf den Stack gelegt. Zudem legen die Funktionen ihre lokalen Variablen auf den Stack.

see Batch processing	**Stapelbetrieb** Betriebsart, bei der das Programm mit allen Eingabedaten vollständig vorliegen muss, bevor die Abarbeitung beginnen kann. Üblicherweise ist Stapelbetrieb interaktionslos.
see Control Bus	**Steuerbus** Dieser überträgt die Kommandos (z.B. Lese- und Schreibanweisungen) vom Prozessor und Statusmeldungen von den Peripheriegeräten. Zudem enthält er Leitungen, über die E/A-Geräte dem Prozessor Unterbrechungsanforderungen (Interrupts) signalisieren
Swap A storage area (file or partition) on a hard disk drive or SSD to which the operating system relocates those processes that currently have no access to a processor or processor core when the main memory lacks free capacity	siehe Auslagerungsspeicher
Swapping Process of outsourcing and retrieving data in/from the main memory from/to the swap memory (usually a file or partition on a hard disk drive or SSD)	**Swapping** Prozess des Ein- und Auslagern von Daten in den/vom Arbeitsspeicher vom/in den Auslagerungsspeicher (meist eine Datei oder Partition auf einer Festplatte oder SSD)
System Call A function call in the operating system kernel to perform a more privileged task, e.g., hardware access. It causes a context switch, which is a switch from user mode to kernel mode	**Systemaufruf** Ein Funktionsaufruf im Betriebssystemkern, um eine höher privilegierte Aufgabe, wie zum Beispiel einen Hardwarezugriff, durchführen. Dabei kommt es zum Moduswechsel, also zum Sprung vom Benutzermodus in den Kernelmodus
TCP Transmission Control Protocol. Connection-oriented Transport Layer Protocol	**TCP** Transmission Control Protocol. Verbindungsorientiertes Transportprotokoll
Terminal Interface for the users for working on a mainframe computer	**Terminal** Dialogstationen zur Arbeit an einem Großrechner
Time-sharing Multiple users can work simultaneously on one mainframe computer via terminals	Siehe Dialogbetrieb

Time slice A period with a fixed duration that is available periodically. During a time slice, a process can use a resource

siehe Zeitscheibe

see Driver

Treiber Eine Sammlung von Funktionen des Betriebssystemkerns, um auf eine Hardware zugreifen zu können

TLB Translation Lookaside Buffer. A buffer memory that stores address translations from virtual to physical memory addresses, thereby accelerates frequently used address translations

TLB Translation Lookaside Buffer. Ein Übersetzungspuffer, der Adressumwandlungen von virtuellen in physische Speicheradressen speichert und somit häufig verwendete Adressumwandlungen beschleunigt

Triple-DES An improvement of the symmetric encryption method Data Encryption Standard (DES). Both methods belong to the block cipher category. Data to be encrypted is first split into blocks of the same size. Then one block is encrypted after the other. Triple-DES implements multiple executions of DES with three different keys. First, each data block is encrypted with the first key, then it is decrypted with the second key, and finally, it is encrypted with the third key. The key length of 168 bits is three times as long as DES (56 bits)

Triple-DES Eine Verbesserung des symmetrischen Verschlüsselungsverfahrens Data Encryption Standart (DES). Beide Verfahren gehören zur Gruppe der Blockchiffren. Dabei werden zu verschlüsselnden Daten zuerst in Blöcke gleicher Größe unterteilt. Danach wird ein Block nach dem anderen verschlüsselt. Triple-DES basiert auf der mehrfachen Ausführung von DES mit drei verschiedenen Schlüsseln. Zuerst wird jeder Datenblock mit dem ersten Schlüssel chiffriert, dann mit dem zweiten Schlüssel dechiffriert und abschließend mit dem dritten Schlüssel chiffriert. Die Schlüssellänge ist mit 168 Bit dreimal so groß wie bei DES (56 Bit)

UID User ID. The unique identifier (number) of a user in the operating system

UID User-ID. Die eindeutige Kennung (Nummer) eines Benutzers im Betriebssystem

UDP User Datagram Protocol. Connectionless Transport Layer protocol

UDP User Datagram Protocol. Verbindungsloses Transportprotokoll

Unicode Multi-byte character encoding

Unicode Mehrbyte-Zeichenkodierung

Unix A multitasking and multi-user operating system that was developed by the Bell Labs in the C programming language since the end of the 1960s. The inventors are Ken Thompson and Dennis Ritchie. Many proprietary and free (e.g., Linux) operating systems base on the concepts of UNIX

Unix Ein Betriebssystem mit Mehrprogrammbetrieb und Mehrbenutzerbetrieb, das ab Ende der 1960er Jahre von den Bell Labs in der Programmiersprache C entwickelt wurde. Als Erfinder gelten Ken Thompson und Dennis Ritchie. Zahlreiche proprietäre und freie (z.B. Linux) Betriebssysteme bauen auf den Konzepten von UNIX auf

Virtual machine An isolated environment that acts like a full-fledged computer with dedicated components

Virtuelle Maschine Eine abgeschottete Umgebung, die sich wie ein vollwertiger Computer mit eigenen Komponenten verhält

XFS A file system that was initially developed by Silicon Graphics (SGI), is free software since 2000 and is included in the Linux kernel since 2001. In general, XFS is regarded to perform well and operate efficiently

XFS Ein ursprünglich von der Firma Silicon Graphics (SGI) entwickeltes Dateisystem, das seit 2000 freie Software und seit 2001 im Linux-Kernel enthalten ist. Allgemein gilt XFS als sehr leistungsfähig und ausgereift

see Time slice

Zeitscheibe Ein Zeitabschnitt fester Dauer, der periodisch zur Verfügung steht. Während einer Zeitscheibe kann ein Prozess eine Ressource verwenden

see Time-sharing

Zeitteilbetrieb Siehe Dialogbetrieb

see Cylinder

Zylinder Alle Spuren auf allen Platten einer Festplatte bei einer Position des Schwungarms

Bibliography

[1] 7-Zip LZMA Benchmark Intel i7-6700 Skylake.
https://www.7-cpu.com/cpu/Skylake.html

[2] 4th Gen AMD EPYC Processor Architecture. White Paper. Fourth Edition. May 2024.
https://www.amd.com/content/dam/amd/en/documents/products/epyc/4th-gen-amd-epyc-processor-architecture-whitepaper.pdf

[3] ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. White Paper. Version C.d. 2018.
https://developer.arm.com/documentation/ddi0406/latest/

[4] ARMv8-A Address Translation. White Paper. Version 1.1. 2019.
https://documentation-service.arm.com/static/5efa1d23dbdee951c1ccdec5

[5] Achilles A. (2006) Betriebssysteme. Eine kompakte Einführung mit Linux. Springer, Heidelberg

[6] Baumgarten U, Siegert H-J. (2007) Betriebssysteme. 6. Auflage. Oldenbourg Verlag, München

[7] Beazley D, Jones B. (2013) Python Cookbook. Third Edition. O'Reilly

[8] Beck M, Böhme H, Dziadzka M, Kunitz U, Magnus R, Verworner D. (1997) Linux Kernelprogrammierung. 4. Auflage. Addison-Wesley

[9] Belady L, Nelson R, Shedler G. (1969) An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine. Communications of the ACM. Volume 12. Number 6

[10] Blöchl B, Meyberg C. (2002) Repetitorium der Informatik. 1. Auflage. Oldenbourg Verlag, München

[11] Bovet D, Cesati M. (2006) Understanding the Linux Kernel. 3. Auflage. O'Reilly

[12] Bengel G, Baun C, Kunze M, Stucky K-U. (2015) Masterkurs Parallele und Verteilte Systeme. Springer Vieweg, Wiesbaden

[13] Berestovskyy A. (2021) Smarter CPU Testing – How to Benchmark Kaby Lake & Haswell Memory Latency.
https://nexthink.com/blog/smarter-cpu-testing-kaby-lake-haswell-memory

[14] Bonwick J, Ahrens M, Henson V, Maybee M, Shellenbaum M. (2003) The Zettabyte File System. Proceedings of the 4th ACM Usenix Conference on File and Storage Technologies

[15] Bower, T. (2009) Operating Systems Study Guide.
http://faculty.salina.k-state.edu/tim/ossg/index.html

[16] Bräunl T. (1993) Parallele Programmierung. Vieweg, Braunschweig/Wiesbaden

[17] Brause R. (2017) Betriebssysteme. 4. Auflage. Springer, Berlin/Heidelberg

[18] Bunting S. (2008) EnCase Computer Forensics: The Official EnCE: EnCase Certified Examine. Second Edition. Wiley

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8

[19] Buyya R, Cortes T, Jin H. (2001) Single System Image (SSI). The International Journal of High Performance Computing Applications. Volume 15, Number 2

[20] Card R, Ts'o T, Tweedie S. (1995) Design and Implementation of the Second Extended Filesystem. Proceedings of the First Dutch International Symposium on Linux

[21] Carikli D. (2018) The Intel Management Engine: an attack on computer users' freedom. Free Software Foundation. `https://static.fsf.org/nosvn/blogs/Intel_ME_Carikli_article_PRINT_2.pdf`

[22] Chapman B, Curtis T, Pophale S, Poole S, Kuehn J, Koelbel C, Smith L. (2010) Introducing OpenSHMEM. Proceedings of the 4th ACM Conference on Partitioned Global Address Space Programming Model

[23] Chisnall D. (2008) The Definitive Guide to the Xen Hypervisor. Prentice Hall

[24] Chunyu (2024) Thinking about eevdf.
`https://chunyu.sh/blog/thinking-about-eevdf/`

[25] Coffman E, Elphick M, Shoshani A. (1971) System Deadlocks. Computing Surveys. Volume 3, Number 2, Juni 1971, P. 67-78

[26] Contiki: The Open Source OS for the Internet of Things.
`http://www.contiki-os.org`

[27] Corbet J. (2024) Completing the EEVDF scheduler.
`https://lwn.net/Articles/925371/`

[28] Corbet J. (2023) An EEVDF CPU scheduler for Linux.
`https://lwn.net/Articles/969062/`

[29] Corbet J. (2004) Reorganizing the address space.
`https://lwn.net/Articles/91829/`

[30] Cormen T, Leiserson C, Rivest R, Stein C. (2010) Algorithmen – Eine Einführung. 3. Auflage. Oldenbourg Verlag, München

[31] Crawford T, Prinz P. (2015) C in a Nutshell: The Definitive Reference. 2. Auflage. O'Reilly

[32] Dinan J, Balaji P, Lusk E, Sadayappan P, Thakur R. (2010) Hybrid Parallel Programming with MPI and Unified Parallel C. Proceedings of the 7th ACM international conference on Computing frontiers

[33] Dijkstra E. (1965) Cooperating sequential processes.
`https://www.cs.utexas.edu/~EWD/ewd01xx/EWD123.PDF`

[34] Dorward S, Pike R, Presotto D, Ritchie D, Trickey H, Winterbottom P. (1997) The Inferno Operating System. Bell Labs Technical Journal. Volume 2, Number 1

[35] Ehses E, Köhler L, Riemer P, Stenzel H, Victor F. (2005) Betriebssysteme. Pearson, München

[36] Ermolov M, Goryachy M. (2017) How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine. Black Hat Europe. London

[37] Erickson J. (2009) Hacking: die Kunst des Exploits. dpunkt.verlag, Heidelberg

[38] Fairbanks K. (2012) An analysis of Ext4 for digital forensics. Digital Investigation. Band 9. S.118-130
`https://doi.org/10.1016/j.diin.2012.05.010`

[39] Glatz E. (2006) Betriebssysteme. dpunkt.verlag, Heidelberg

[40] Glatz E. (2019) Betriebssysteme. 4. Auflage, dpunkt.verlag, Heidelberg

[41] Goll J, Dausmann M. (2014) C als erste Programmiersprache: Mit den Konzepten von C11. 8. Auflage. Springer Fachmedien, Wiesbaden

[42] Gropp W, Lusk E, Doss N, Skjellum A. (1996) A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing. Volume 22, Number 6

[43] Gumm H, Sommer M. (2011) Einführung in die Informatik. Oldenburg, München

[44] Hamilton M. (1999) Hamilton, M: Software Development: Building Reliable Systems. Pearson

[45] Brinch Hansen P. (1973) Shared Classes.
`http://brinch-hansen.net/papers/1973b.pdf`

[46] Heiß H.-U. (1999) Folienskript Betriebssysteme. Universität Paderborn
`http://www2.cs.uni-paderborn.de/fachbereich/AG/heiss/lehre/bs/`

[47] Herold H. (1996) UNIX-Systemprogrammierung. 2. Auflage. Addison-Wesley

[48] Herold H. (1999) Linux-Unix Kurzrefenz. 2. Auflage. Addison-Wesley

[49] Herold H, Lurz B, Wohlrab J. (2012) Grundlagen der Informatik. 2. Auflage. Pearson, München

[50] Hieronymus A. (1993) UNIX-Systemarchitektur und Programmierung. Vieweg, Braunschweig/Wiesbaden

[51] Hoare C. (1974) Monitors: An Operating System Structuring Concept. Communications of the ACM. Volume 17, Number 10, October 1974. P. 549-557

[52] Hönig T. (2006) Der O(1)-Scheduler im Kernel 2.6. Linux-Magazin 2/2004
`https://www.linux-magazin.de/ausgaben/2004/02/die-reihenfolge-zaehlt/`

[53] Huang A. (2022) Slab Allocator in Linux Kernel.
`https://de.slideshare.net/slideshow/slab-allocator-in-linux-kernel/253184071`

[54] Intel 80386 Programmer's Reference Manual 1986.
`http://css.csail.mit.edu/6.858/2012/readings/i386.pdf`

[55] 5-Level Paging and 5-Level EPT. White Paper. Revision 1.1. 2017.
`https://software.intel.com/content/dam/develop/public/us/en/documents/5-level-paging-white-paper.pdf`

[56] Jobst F. (2015) Programmieren in Java. Hanser, München

[57] Jones M. (2009) Inside the Linux 2.6 Completely Fair Scheduler – Providing fair access to CPUs since 2.6.23. IBM.
`https://developer.ibm.com/tutorials/l-completely-fair-scheduler/`

[58] Kay J, Lauder P. (1988) A Fair Share Scheduler. Communications of the ACM. Volume 31, Number 1, January 1988. P. 44-55

[59] Kernel development community (2024) EEVDF Scheduler.
`https://docs.kernel.org/next/scheduler/sched-eevdf.html`

[60] Kerrisk M. (2010) The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press

[61] Kirch U, Prinz P. (2019) C – kurz & gut. 2. Auflage. O'Reilly

[62] Knowlton K. (1965) A Fast storage allocator. Communications of the ACM. Volume 8, Number 10, October 1965. P. 623-624

[63] Knuth D. (1968) The Art of Computer Programming – Volume 1. First Edition. Addison-Wesley

[64] Leveson N, Turner C. (1993) An Investigation of the Therac-25 Accidents. IEEE Computer. Volume 26, Number 7, June 1993. P. 18-41

[65] Levin J. (2013) Mac OS X and iOS Internals. To the Apple's Core. Wiley

[66] LINUX System Call Quick Reference.
`http://www.digilife.be/quickreferences/qrc/linux%20system%20call%20quick%20reference.pdf`

[67] Love R. (2005) Linux-Kernel-Handbuch. 1. Auflage. Addison-Wesley

[68] Love R. (2010) Linux-Kernel-Handbuch. 3. Auflage. Addison-Wesley

[69] Mandl P. (2014) Grundkurs Betriebssysteme. 4. Auflage. Springer Vieweg, Wiesbaden

[70] Marinas, C. Memory Layout on AArch64 Linux. The Linux Kernel documentation.
`https://www.kernel.org/doc/html/v6.4/arm64/memory.html`

[71] Marinas, C. (2012) Linux on AArch64 ARM 64-bit Architecture..
`https://events.static.linuxfound.org/images/stories/pdf/lcna_co2012_marinas.pdf`

[72] Mathur A, Cao M, Bhattacharya S, Dilger A, Tomas, A, Vivier L. (2007) The new ext4 filesystem: current status and future plans. Proceedings of the Linux symposium. Band 2. S.21-33

[73] Mendoza A, Skawratananond C, Walker A. (2006) Unix to Linux Porting: A Comprehensive Reference. Prentice Hall

[74] McDougall R, Mauro J. (2007) Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture. Second Edition. Prentice Hall

[75] McKusick M, Neville-Neil G, Watson R. (2014) The Design and Implementation of the FreeBSD Operating System. 2. Auflage. Addison Wesley/Pearson

[76] Microsoft Corporation. Default cluster size for NTFS, FAT, and exFAT.
`https://support.microsoft.com/en-us/kb/140365`

[77] Microsoft Corporation. File Caching.
`https://msdn.microsoft.com/de-de/library/windows/desktop/aa364218(v=vs.85).aspx`

[78] Microsoft Corporation. exFAT file system specification.
`https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification`

[79] Microsoft Corporation. File System Functionality Comparison.
`https://docs.microsoft.com/en-gb/windows/win32/fileio/filesystem-functionality-comparison`

[80] Microsoft Corporation. How NTFS Works (2003)
`https://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx`

[81] Microsoft Corporation. MS Windows NT Kernel-mode User and GDI White Paper.
`https://technet.microsoft.com/library/cc750820.aspx`

[82] Microsoft Corporation. Standard-Clustergröße für NFTS, FAT und exFAT.
`https://support.microsoft.com/en-us/help/140365/`

[83] Rasiukevicius M. (2009) Thread scheduling and related interfaces in NetBSD 5.0.
`https://www.netbsd.org/~rmind/pub/netbsd_5_scheduling_apis.pdf`

[84] Musumeci G, Loukides M. (2002) System Performance Tuning. Second Edition. O'Reilly

[85] Mordvinova O, Kunkel J, Baun C, Ludwig T, Kunze M. (2009) USB Flash Drives as an Energy Efficiency Storage Alternative. IEEE/ACM Grid 2009, Proceedings of the 10th International Conference on Grid Computing

[86] Nehmer J, Sturm P. (2001) Systemsoftware. Grundlagen moderner Betriebssysteme. dpunkt Verlag, Heidelberg

[87] Nordvik R, Georges H, Toolan F, Axelsson S. (2019) Reverse engineering of ReFS. Digital Investigation. Band 30, S.127-147
`https://doi.org/10.1016/j.diin.2019.07.004`

[88] Oracle (2013) System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones.
`https://docs.oracle.com/cd/E22645_01/pdf/817-1592.pdf`

[89] Ousterhout J, Cherenson R, Douglis F, Nelson N, Welch B. (1988) The Sprite Network Operating System. IEEE Computer. Volume 21, Number 2

[90] Pancham P, Chaudhary D, Gupta R. (2014) Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance. International Journal of Computer Applications. Volume 98, Number 19

[91] Quade J, Kunst E.-K. (2006) Linux-Treiber entwickeln. Eine systematische Einführung in Gerätetreiber für den Kernel 2.6. 2 Auflage. dpunkt Verlag, Heidelberg

[92] Quade J, Mächtel M. (2012) Moderne Realzeitsysteme kompakt. Eine Einführung mit Embedded Linux. dpunkt Verlag, Heidelberg

[93] Rauber T, Rünger G. (2013) Parallel Programming: for Multicore and Cluster Systems. 2 Auflage. Springer, Berlin/Heidelberg

[94] Richter L. (1985) Betriebssysteme. 2. Auflage. Teubner, Stuttgart

[95] Rhodehamel M. (1989) The Bus Interface and Paging Units of the i860 Microprocessor. Proceedings of the IEEE International Conference on Computer Design. P. 380-384

[96] Robbins K, Robbins S. (1996) Practical UNIX Programming. A Guide to Concurrency, Communication, and Multithreading. Prentice Hall.

[97] Rochkind M. (1988) UNIX Programmierung für Fortgeschrittene. Hanser, München

[98] Rodeh O, Bacik J, Mason C. (2013) BTRFS: The Linux B-tree filesystem. ACM Transactions on Storage (TOS). Band 9, Nummer 3

[99] Russinovich M. (1998) Inside Memory Management, Part 2. http://windowsitpro.com/systems-management/inside-memory-management-part-2

[100] Russinovich M., Solomon D. (2005) Microsoft Windows Internals. Fourth Edition. Microsoft

[101] Silberschatz A, Galvin P, Gagne G. (2013) Operating System Concepts. Ninth Edition. Wiley

[102] Stuart B. (2009) Principles of Operating Systems. First Edition. Course Technology, Boston

[103] Schmitt T, Kämmer N, Schmidt P, Weggerle A, Gerhold S, Schulthess P. (2011) Rainbow OS: A distributed STM for in-memory data clusters. IEEE MIPRO 2011, Proceedings of the 34th International Convention

[104] Sedgewick R. (1992) Algorithmen in C. Addison-Wesley.

[105] Severance C. (2015) Guido van Rossum: The Early Years of Python. IEEE Computer. Volume 48, Number 2

[106] Shullich R. (2010) Reverse Engineering the Microsoft exFAT File System. https://www.sans.org/reading-room/whitepapers/forensics/paper/33274

[107] Stallings W. (2003) Betriebssysteme – Prinzipien und Umsetzung. 4. Auflage. Pearson, München

[108] Stallings W. (2000) Operating Systems – Internals and Design Principles. Fourth Edition. Prentice-Hall, New Jersey

[109] Introduction to Computer Architecture. (2011) http://cseweb.ucsd.edu/classes/wi11/cse141/Slides/19_VirtualMemory.key.pdf

[110] Sun Microsystems (1994) Multithreaded Programming Guide. SunSoft, Mountain View https://docs.oracle.com/cd/E19457-01/801-6659/801-6659.pdf

[111] Tanenbaum A. (2001) Modern Operating Systems. Second Edition. Prentice Hall

[112] Tanenbaum A. (2008) Modern Operating Systems. Third Edition. Pearson

[113] Tanenbaum A, Bos H (2014) Modern Operating Systems. Fourth Edition. Prentice Hall

[114] Tanenbaum A. (2002) Moderne Betriebssysteme. 2. Auflage. Pearson, München

[115] Tanenbaum A. (2009) Moderne Betriebssysteme. 3. Auflage. Pearson, München

[116] Tanenbaum A, Goodman J. (2001) Computerarchitektur. Pearson, München

[117] Tanenbaum A, Sharp G. The Amoeba Distributed Operating System. http://www.cs.vu.nl/pub/amoeba/Intro.pdf

[118] Tanenbaum A, Woodhull A. (2006) Operating Systems: Design and Implementation. Third Edition. Prentice Hall

[119] Vogt C. (2001) Betriebssysteme. Spektrum Akademischer Verlag, Heidelberg

[120] Vogt C. (2012) Nebenläufige Porgrammierung. Hanser, München

[121] Wang K. C. (2023) ARMv8 Architecture and Programming. Embedded and Real-Time Operating Systems. Springer, S.505–792 https://doi.org/10.1007/978-3-031-28701-5_11

[122] Werner M. (2016) Folienskript Betriebssysteme. Technische Universität Chemnitz
`https://osg.informatik.tu-chemnitz.de/lehre/os/`

[123] Wettstein H. (1984) Architektur von Betriebssystemen. 2. Auflage. Hanser, München

[124] Willemer A. (2004) Wie werde ich UNIX-Guru? Rheinwerk Verlag, Bonn
`http://openbook.rheinwerk-verlag.de/unix_guru/`

[125] Windows X86 System Call Table (NT/2000/XP/2003/Vista/2008/7/8).
`http://j00ru.vexillium.org/ntapi`

[126] Wolf J. (2005) Linux-UNIX-Programmierung: Das umfassende Handbuch. 1. Auflage. Galileo Computing, Bonn

[127] Yodaiken V. (1999) The RTLinux Manifesto.
`http://www.yodaiken.com/papers/rtlmanifesto.pdf`

Index

C. Baun, *Operating Systems/Betriebssysteme*,
https://doi.org/10.1007/978-3-658-48060-8