

Peter Mandl

Grundkurs Betriebssysteme

Architekturen, Betriebsmittelverwaltung,
Synchronisation, Prozesskommunikation

2. Auflage

► Mit Online-Service

STUDIUM



Peter Mandl

Grundkurs Betriebssysteme

Peter Mandl

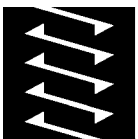
Grundkurs Betriebssysteme

Architekturen, Betriebsmittelverwaltung,
Synchronisation, Prozesskommunikation

2., überarbeitete und aktualisierte Auflage

Mit 164 Abbildungen und 6 Tabellen

STUDIUM



**VIEWEG+
TEUBNER**

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
<<http://dnb.d-nb.de>> abrufbar.

Das in diesem Werk enthaltene Programm-Material ist mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Der Autor übernimmt infolgedessen keine Verantwortung und wird keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials oder Teilen davon entsteht.

Höchste inhaltliche und technische Qualität unserer Produkte ist unser Ziel. Bei der Produktion und Auslieferung unserer Bücher wollen wir die Umwelt schonen: Dieses Buch ist auf säurefreiem und chlorfrei gebleichtem Papier gedruckt. Die Einschweißfolie besteht aus Polyäthylen und damit aus organischen Grundstoffen, die weder bei der Herstellung noch bei der Verbrennung Schadstoffe freisetzen.

- 1. Auflage 2008
- 2., überarbeitete und aktualisierte Auflage 2010

Alle Rechte vorbehalten

© Vieweg+Teubner | GWV Fachverlage GmbH, Wiesbaden 2010

Lektorat: Christel Roß | Walburga Himmel

Vieweg+Teubner ist Teil der Fachverlagsgruppe Springer Science+Business Media.

www.viewegteubner.de



Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Umschlaggestaltung: KünkelLopka Medienentwicklung, Heidelberg
Druck und buchbinderische Verarbeitung: STRAUSS GMBH, Mörlenbach
Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.
Printed in Germany

ISBN 978-3-8348-0809-7

Vorwort

Betriebssysteme stellen Dienste für Anwendungssysteme bereit und werden auch nur für diese entwickelt. Ohne Anwendung bräuchte man nämlich auch kein Betriebssystem. Heutige Betriebssysteme wie Unix, Linux, Windows oder Großrechnerbetriebssysteme sind sehr komplexe Programmsysteme, in denen viele Jahre Entwicklungsleistung stecken und die auch ständig weiterentwickelt werden müssen. Sie nutzen viele Techniken und Strategien zur Gewährleistung einer hohen Leistungsfähigkeit und zur Bereitstellung von optimalen Services für die Anwendungen.

Das vorliegende Lehrbuch befasst sich mit den Grundlagen von Betriebssystemen. In kompakter Form werden wichtige Grundkonzepte, Verfahren und Algorithmen dargestellt, die in modernen Betriebssystemen eingesetzt werden. Das Buch behandelt die folgenden Themenkomplexe:

1. Einführung
2. Betriebssystemarchitekturen und Betriebsarten
3. Interruptverarbeitung
4. Prozesse und Threads
5. CPU-Scheduling
6. Synchronisation und Kommunikation
7. Hauptspeicherverwaltung
8. Geräte- und Dateiverwaltung

Der Schwerpunkt liegt bei sog. Mehrzweckbetriebssystemen oder Universalbetriebssystemen, die überwiegend für betriebliche Informationssysteme eingesetzt werden, weniger bei Realzeit- bzw. Embedded Systemen, die mehr in technischen Fragestellungen relevant sind. Einige grundlegende Konzepte gelten aber für alle Typen von Betriebssystemen.

Kapitel 1 enthält eine Einführung in grundlegende Konzepte und Aufgaben von Betriebssystemen sowie eine historische Betrachtung. Kapitel 2 stellt wichtige Architekturvarianten und Betriebsarten von Betriebssystemen vor. In Kapitel 3 werden die Konzepte zur Interruptbearbeitung und zur Behandlung von Systemaufrufen diskutiert.

Im Kapitel 4 wird erläutert, wie heutige Betriebssysteme die Ressourcen „Prozess“ und „Thread“ verwalten. Was Threads sind und wie sie in höheren Programmiersprachen wie Java und C# verwendet werden können, wird ebenfalls dargestellt. Kapitel 5 beschreibt verschiedene Strategien zur Verwaltung von Prozessen und Threads (*CPU-Scheduling* = Ablaufplanung) und erläutert, wie ein Betriebssystem einen Prozess-/Threadwechsel durchführt (*Dispatching* = Arbeitsvorbereitung, Disposition). Verschiedene Scheduling-Strategien werden vorgestellt und mit Fallbeispielen unterlegt.

Das Thema *Synchronisation und Kommunikation* wird in Kapitel 6 behandelt, wobei die grundlegenden Prinzipien paralleler bzw. nebenläufiger Bearbeitung von Objekten durch Prozesse und Threads erläutert werden. Als Mechanismen zur Synchronisation nebenläufiger Prozesse und Threads werden Locks, Semaphore, Mutexe und Monitore behandelt. Es wird gezeigt, wie man diese Mechanismen in höheren Sprachen wie Java und C# einsetzen kann. Kommunikationsmöglichkeiten zwischen Prozessen oder Threads werden ebenfalls vorgestellt, wobei z.B. auf das Pipe-Konzept eingegangen wird.

In Kapitel 7 werden schließlich die Konzepte der *Hauptspeicherverwaltung* ausführlich erläutert, wobei die virtuelle Speichertechnik im Mittelpunkt steht. Seitenersetzungsstrategien werden vorgestellt und Ansätze für die Verwaltung großer Adressräume skizziert.

Ein grundlegender Einblick in einige Konzepte der *Geräte- und Dateiverwaltung* innerhalb von Betriebssystemen folgt schließlich in Kapitel 8. Es wird erläutert, wie die Geräteverwaltung ins Betriebssystem eingebettet ist und wie Dateisysteme prinzipiell funktionieren.

Im Anhang sind einige Ergänzungen enthalten. Insbesondere werden einige Datenstrukturen, die in Betriebssystemen häufig verwendet werden, im Anhang einführend erläutert.

In diesem Buch wird ein praxisnaher Ansatz gewählt. Der Stoff wird mit vielen Beispielen aus aktuell relevanten Betriebssystemen und Programmiersprachen angereichert. Als Beispiel-Betriebssysteme werden vorwiegend Windows, Unix und Linux herangezogen. Zu jedem Kapitel sind Übungsaufgaben beigelegt. Für das Verständnis einiger Programmbeispiele sind grundlegende Kenntnisse von Programmiersprachen (C, C++, Java und C#) nützlich, jedoch können die wesentlichen Konzepte auch ohne tiefere Programmierkenntnisse verstanden werden. Vom Leser werden ansonsten keine weiteren Grundkenntnisse vorausgesetzt.

Der Inhalt des Buches entstand aus mehreren Vorlesungen über Betriebssysteme an der Hochschule für Angewandte Wissenschaften – FH München. Bedanken möchte ich mich sehr herzlich bei meinen KollegInnen *Kerstin Schmidt* und *Georg Lackermair* sowie bei meinem Tutor *David Fehr* für die tatkräftige Unterstützung und für die Diskussionen im Rahmen der Vorlesungen und Übungen. Auch bei meinen Studentinnen und Studenten, die mir Feedback zum Vorlesungsstoff gaben, möchte ich mich herzlich bedanken. Dem Verlag, insbesondere Frau Sybille Thelen, danke ich für die Unterstützung im Projekt und für die allzeit problemlose Zusammenarbeit.

In der vorliegenden 2. Auflage wurden textliche Überarbeitungen sowie einige Aktualisierungen vorgenommen und es wurden zahlreiche Fehler behoben. Der Aufbau des Buches blieb bis auf die Aufteilung von Kapitel 1 in drei Einzelkapitel unverändert. Die Einführung in wichtige Datenstrukturen wurde in den Anhang verlegt. Für die zahlreichen textlichen Hinweise und die inhaltlichen Verbesserungsvorschläge möchte ich meinem Kollegen Herrn Prof. Dr. Christian Vogt recht herzlich danken.

Fragen und Korrekturvorschläge richten Sie bitte an mandl@cs.hm.edu.

Für begleitende Informationen zur Vorlesung siehe www.prof-mandl.de.

München, im Juli 2009

Peter Mandl

Inhaltsverzeichnis

1	Einführung	1
1.1	Computersysteme	1
1.1.1	Einführung	2
1.1.2	Aufgabe von Betriebssystemen	2
1.1.3	Hardwaremodelle	3
1.1.4	CPU-Registersatz	6
1.1.5	Beispiele für Microprozessor-Architekturen	8
1.1.6	Multicore-Prozessoren und Hyperthreading-CPU's	12
1.1.7	Unser einfaches Modell der Hardware	13
1.2	Betriebssystem-Geschichte	14
1.2.1	Historische Entwicklung	15
1.2.2	Geschichte von Microsoft Windows und Unix	17
1.3	Übungsaufgaben	21
2	Betriebssystemarchitekturen und Betriebsarten	23
2.1	Betriebssystemarchitekturen	23
2.1.1	Klassische Architekturen	23
2.1.2	Mikrokern-Architektur	25
2.1.3	Verteilte Systeme und Middleware	26
2.1.4	Betriebssystem-Virtualisierung	28
2.1.5	Architektur von Unix und Windows	29
2.2	Betriebsarten	32
2.2.1	Parallelisierung der Verarbeitung	33
2.2.2	Teilhaber- versus Teilnehmerbetrieb	34
2.2.3	Application-Server-Betrieb	37
2.2.4	Terminalserver-Betrieb	39
2.2.5	Handheld-Computing	40

2.3 Übungsaufgaben.....	41
3 Interruptverarbeitung.....	43
3.1 Interrupts.....	44
3.1.1 Überblick.....	44
3.1.2 Interrupt-Bearbeitung.....	45
3.1.3 Interrupt-Verarbeitung bei IA32-Prozessoren.....	50
3.1.4 Interrupt-Bearbeitung unter Windows.....	53
3.1.5 Interruptverarbeitung unter Linux	57
3.2 Systemaufrufe	60
3.3 Übungsaufgaben.....	64
4 Prozesse und Threads	65
4.1 Prozesse	66
4.1.1 Prozessmodell	66
4.1.2 Prozessverwaltung	67
4.1.3 Prozesslebenszyklus.....	70
4.2 Threads	71
4.2.1 Threadmodell.....	71
4.2.2 Implementierung von Threads	71
4.2.3 Vor-/Nachteile und Einsatzgebiete von Threads	74
4.3 Programmierkonzepte für Threads	76
4.3.1 Threads in Java.....	76
4.3.2 Threads in C#	80
4.4 Prozesse und Threads in konkreten Betriebssystemen	84
4.4.1 Prozesse und Threads unter Windows.....	84
4.4.2 Prozesse und Threads unter Unix und Linux.....	92
4.5 Übungsaufgaben.....	97
5 CPU-Scheduling	99
5.1 Scheduling-Kriterien.....	100
5.2 Scheduling-Verfahren.....	103
5.2.1 Verdrängende und nicht verdrängende Verfahren	103

5.2.2	Überblick über Scheduling-Verfahren.....	103
5.2.3	Multi-Level-Scheduling mit Prioritäten	106
5.2.4	Round-Robin-Scheduling mit Prioritäten	107
5.3	Vergleich ausgewählter Scheduling-Verfahren	109
5.3.1	CPU-Scheduling im ursprünglichen Unix	113
5.3.2	CPU-Scheduling unter Linux	115
5.3.3	CPU-Scheduling unter Windows.....	122
5.3.4	Scheduling von Threads in Java	129
5.3.5	Zusammenfassung	129
5.4	Übungsaufgaben	131
6	Synchronisation und Kommunikation.....	133
6.1	Grundlegendes zur Synchronisation.....	134
6.1.1	Nebenläufigkeit, atomare Aktionen und Race Conditions.....	134
6.1.2	Kritische Abschnitte und wechselseitiger Ausschluss	137
6.1.3	Eigenschaften nebenläufiger Programme	139
6.2	Synchronisationskonzepte	141
6.2.1	Sperren.....	141
6.2.1	Semaphore.....	144
6.2.3	Monitore	149
6.3	Synchronisationstechniken moderner Betriebssysteme.....	154
6.3.1	Synchronisationstechniken unter Unix	154
6.3.2	Synchronisationstechniken unter Windows.....	154
6.3.3	POSIX-Synchronisationstechniken	155
6.4	Synchronisationsmechanismen in Programmiersprachen	155
6.4.1	Die Java-Synchronisationsprimitive „synchronized“	156
6.4.2	Warten auf Bedingungen in Java	163
6.4.3	Weitere Synchronisationsmechanismen in Java	166
6.4.4	C#-Monitore	169
6.4.5	Die C#-Synchronisationsprimitive „lock“	177
6.4.6	C#-Mutex-Objekte	178

6.4.7	C#-Lese- und Schreibsperrern	180
6.4.8	C#-Interlocked-Klasse	181
6.4.9	Warten auf Bedingungen in C#	183
6.5	Kommunikation von Prozessen und Threads	183
6.5.1	Grundbegriffe der Kommunikation.....	184
6.5.2	Möglichkeiten für eine rechnerinterne Kommunikation	189
6.5.3	Fallbeispiel: Pipes	193
6.5.4	Rechnerübergreifende Interprozesskommunikation.....	197
6.6	Übungsaufgaben.....	199
7	Hauptspeicherverwaltung	201
7.1	Grundlegende Betrachtungen.....	202
7.1.1	Speicherhierarchien.....	202
7.1.2	Lokalität	203
7.1.3	Adressen und Adressräume	204
7.1.4	Techniken der Speicherverwaltung	206
7.2	Virtueller Speicher.....	209
7.2.1	Grundbegriffe und Funktionsweise.....	209
7.2.2	Optimierung der Speicherverwaltung.....	218
7.2.3	Seitenersetzung und Verdrängung (Replacement).....	223
7.2.4	Vergleich von Seitenersetzungsverfahren.....	232
7.2.5	Speicherbelegungs- und Vergabestrategien (Placement).....	233
7.2.6	Entladestrategie (Cleaning).....	236
7.2.7	Eine weitere Technik: Segmentadressierung.....	237
7.2.8	Shared Memory	239
7.3	Speicherverwaltung in ausgewählten Systemen.....	240
7.3.1	Linux-Speicherverwaltung.....	240
7.3.2	Windows-Speicherverwaltung	244
7.4	Übungsaufgaben.....	250
8	Geräte- und Dateiverwaltung.....	253
8.1	Aufgaben und Überblick	254

8.1.1	Grundlegendes	254
8.1.2	Gerätearten.....	258
8.1.3	Geräteanbindung unter Unix.....	261
8.1.4	Memory-Mapped Ein-/Ausgabe und DMA	262
8.2	Dateiverwaltung.....	264
8.2.1	Allgemeines.....	264
8.2.2	Fallbeispiel: Dateisysteme unter Unix.....	266
8.2.3	Fallbeispiel: Dateisysteme unter Windows	268
8.3	Storage-Systeme	273
8.3.1	RAID-Plattensysteme.....	273
8.3.2	NAS und SAN.....	278
8.4	Übungsaufgaben	280
9	Schlussbemerkung	281
10	Lösungen zu den Übungsaufgaben.....	283
10.1	Einführung	283
10.2	Betriebssystemarchitekturen und Betriebsarten	284
10.2	Interruptverarbeitung.....	287
10.4	Prozesse und Threads.....	291
10.5	CPU-Scheduling	295
10.6	Synchronisation und Kommunikation.....	303
10.7	Hauptspeicherverwaltung.....	310
10.8	Geräte- und Dateiverwaltung.....	318
Anhang		323
A1	Zahlennamen	323
A2	Metrische Grundeinheiten	324
A3	Wichtige Datenstrukturen für Betriebssysteme	325
	Zeiger- und Referenztypen.....	326
	Sequenzen, Listen.....	328
	Stapelspeicher, Stack.....	334
	Warteschlangen, Queues.....	336

Hashtabellen.....	338
A4 Java-Implementierung des Dining-Philosophers-Problems.....	340
A5 C#-Implementierung des Dining-Philosophers-Problems	344
Literaturhinweise	349
Sachwortverzeichnis	351

1 Einführung

Dieses Kapitel geht auf die grundlegenden Aufgaben von Betriebssystemen ein. Das Kapitel beginnt mit einer Einführung in den Aufbau eines Computersystems. Dabei wird die Von-Neumann-Maschine als Basis unserer heutigen Rechnersysteme vorgestellt, und es werden beispielhaft aktuelle Microprozessor-Architekturen wie Intel Pentium und Intel Itanium skizziert. Weiterhin wird ein kurzer Abriss der Geschichte der Betriebssystementwicklung vor allem am Beispiel von Windows und Unix gegeben. Betriebssysteme stellen gewissermaßen Betriebsmittelverwalter dar, welche die Anwendungen mit den verfügbaren Betriebsmitteln wie Speicher und Prozessorzeit sowie Geräte und Dateien versorgen. Darauf wird in diesem Kapitel ebenfalls eingegangen.

Zielsetzung des Kapitels

Der Studierende soll die grundlegenden Aufgaben von Betriebssystemen verstehen und erläutern können. Weiterhin soll die historische Entwicklung von Betriebssystemen nachvollzogen werden können.

Wichtige Begriffe

Von-Neumann-Maschine, CPU-Register, Betriebsmittel, Programmzähler, PSW, Universalbetriebssystem, Mehrzweckbetriebssystem.

1.1 Computersysteme

In diesem Abschnitt soll zunächst ein Überblick über die Aufgaben von Computersystemen gegeben werden, so weit sie für das Verständnis von Betriebssystemen von Belang sind. Wichtig sind vor allem die grundlegenden Zusammenhänge der Rechnerhardware. Wir werden später noch auf einige Aspekte der Hardware im Zusammenhang mit der Speicherverwaltung und der Interrupt-Verarbeitung eingehen, jedoch auch dort auf einem relativ abstrakten Level.¹

¹ Wer sich für Hardware und Rechnerarchitekturen interessiert, dem sei (Herrmann 2002) wärmstens empfohlen.

1.1.1 Einführung

Computersysteme (Synonym: Rechnersysteme) bestehen – etwas vereinfacht dargestellt – aus Hardware, System- und Anwendungssoftware. Unter *Systemsoftware* versteht man zum einen das Betriebssystem (Operating System) und zum anderen Hilfsprogramme wie Compiler, Interpreter, Editoren usw. Anwendungssoftware wie Bankanwendungen, Buchhaltungssysteme, Browser usw. nutzen die Systemsoftware für einen ordnungsgemäßen Ablauf. In Abbildung 1-1 wird die grobe Zusammensetzung eines Computersystems skizziert.

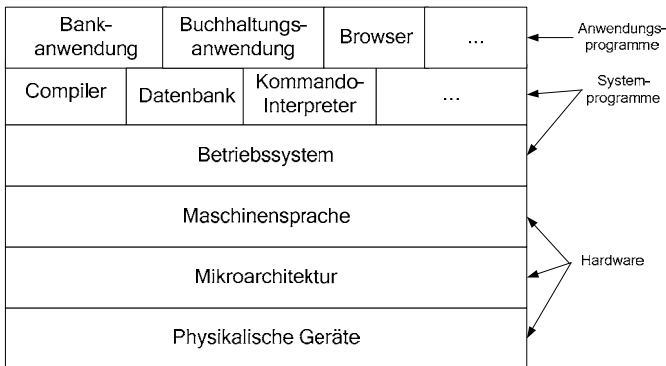


Abbildung 1-1: Hard- und Software eines Computersystems (Tanenbaum 2002)

Aus der Abbildung wird auch deutlich, dass man zur Hardware neben den physikalischen Geräten auch die sog. Mikroarchitektur einschließlich der Mikroprogramme zählt, wobei es hier keine Rolle spielt, ob letztere software- oder hardware-technisch realisiert sind. Weiterhin wird die Maschinensprache der Hardware zugeordnet. Jedes Computersystem hat seine eigene Maschinensprache, die aus Maschinenbefehlen besteht. Die Menge der Maschinenbefehle eines Rechnersystems wird als Befehlssatz bezeichnet. Jeder Befehl ist durch ein Mikroprogramm realisiert. Mikroprogramme werden oft auch als Firmware bezeichnet. Man unterscheidet Testbefehle, Sprungbefehle (BRANCH), Transportbefehle (MOV), arithmetische (ADD) und logische Befehle (CMP, SHIFT) usw. Die Bezeichnungen für die Befehle werden vom Hardware-Architekten des Rechnersystems festgelegt und daher hier nur beispielhaft aufgeführt.

1.1.2 Aufgabe von Betriebssystemen

Das Betriebssystem soll den Anwendungsprogrammierer bzw. den Anwender von Details der Rechnerhardware entlasten. Modern strukturierte Betriebssysteme *kapseln* den Zugriff auf die Betriebsmittel, der Zugriff funktioniert also nur über

Betriebssystemfunktionen. Das Betriebssystem stellt somit eine *virtuelle Maschine* über der Hardware bereit.

Die wesentliche Aufgabe des Betriebssystems ist also die *Betriebsmittelverwaltung*. Als Betriebsmittel versteht man Hardware- und Software-Ressourcen und zwar u.a. die Prozessoren, die Prozesse (Software), Speicher, Dateien und Geräte. Dies sind *reale* Betriebsmittel, aber auch sog. *virtuelle* Betriebsmittel wie virtueller Speicher, virtuelle Prozessoren und virtuelle Koprozessoren werden durch das Betriebssystem verwaltet.

Man kann Betriebsmittel auch nach verschiedenen Kriterien klassifizieren. Genannt wurde bereits die Unterscheidung nach Hardware- und Softwarebetriebsmittel. Ein Hardwarebetriebsmittel ist z.B. der Prozessor, ein Softwarebetriebsmittel eine Nachricht oder ein Prozess. Weiterhin unterscheidet man *entziehbare* und *nicht entziehbare*, sowie *exklusiv* oder *gemeinsam* („shared“) nutzbare Betriebsmittel.

Beispiele:

- Prozessoren sind entziehbare Betriebsmittel
- Drucker sind nicht entziehbare Betriebsmittel
- Prozessoren sind zu einer Zeit nur exklusiv nutzbar
- Magnetplatten sind „shared“, also gemeinsam nutzbar

Das Betriebssystem muss dafür Sorge tragen, dass exklusive Betriebsmittel *konfliktfrei* genutzt werden. Die Entscheidung hierfür trifft üblicherweise ein in Software implementierter Scheduling-Algorithmus, der im Betriebssystem implementiert ist.

1.1.3 Hardwaremodelle

Das Betriebssystem verwaltet also die Rechnerhardware und sonstige softwaretechnische Ressourcen. Die zu Grunde liegende Rechnerhardware sollte daher in den Grundzügen klar sein, um die Aufgaben und die Funktionsweise von Betriebssystemen zu verstehen. Ein Systemprogrammierer, der hardwarenahe Programme entwickelt, muss natürlich von der Hardware etwas mehr verstehen als wir hier für das grundlegende Verständnis von Betriebssystemen benötigen.

Man unterscheidet zwei grundlegende Computermodelle:

- Von-Neumann-Rechner
- Harvard-Rechner

Wie Abbildung 1-2 zeigt besteht der Von-Neumann-Rechner, der von John von Neumann im Jahre 1946 entwickelt wurde, aus den vier Funktionseinheiten Leitwerk (Control Unit, CU), Rechenwerk (Processing Unit, PU), Speicher (Memory)

sowie Ein-/Ausgabe (Input/Output, I/O). Das Leitwerk holt die Maschinenbefehle² nacheinander in den Speicher und führt sie aus. Damit stellt das Leitwerk den „Befehlsprozessor“ dar. Das Rechenwerk, auch Arithmetisch Logische Einheit (ALU) genannt, stellt den „Datenprozessor“ dar und führt logische und arithmetische Operationen aus. Die ALU wird auch als Verarbeitungs- oder Ausführungseinheit bezeichnet. Im Speicher liegen die Maschinenbefehle und die zu verarbeitenden Daten. Maschinenbefehle und Daten sind also in einem gemeinsamen Speicher.

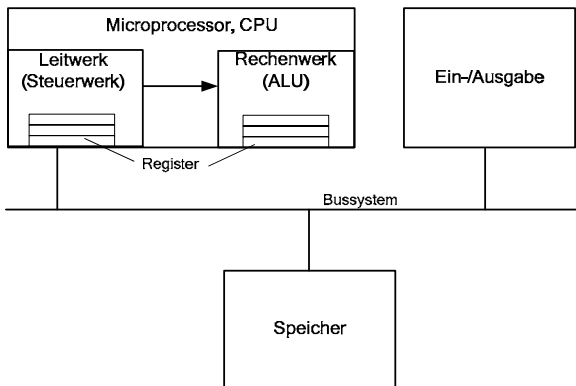


Abbildung 1-2: Architektur eines Von-Neumann-Rechners

Die Ein-/Ausgabe ist die Verbindung externer Geräte (Tastatur, Maus, Festplatten, ...) mit dem Rechenwerk, stellt also die Schnittstelle nach außen dar. Alle Komponenten werden über ein Transportsystem, auch als Bussystem bezeichnet, miteinander verbunden.

Rechenwerk und Leitwerk werden heute in der Regel in einem Prozessor, der als Zentraleinheit (CPU, Central Processing Unit) oder Rechnerkern bezeichnet wird, zusammengefasst. Wir gehen im Weiteren auch davon aus. Die in der Abbildung gezeigten Register sind kleine und schnelle Speicherbereiche, auf die weiter unten noch eingegangen wird.

Der Harvard-Rechner, der nach der Struktur des Mark-I-Rechners (entwickelt von 1939 bis 1944 an der Harvard University) benannt ist, hat im Unterschied zum Von-Neumann-Rechner zwei getrennte Speicher, einen für die Daten und einen für die Maschinenbefehle. Beide Speicher werden auch über einen getrennten Bus mit der CPU verbunden. Für die Übertragung von Maschinenbefehlen und Daten ste-

² Eine Folge von zusammengehörigen Maschinenbefehlen nennen wir übrigens ein Programm.

hen damit doppelt so viele Leitungen zur Verfügung, weshalb der Harvard-Rechner auch effizienter ist. Diese Effizienz muss aber mit einem Hardware-Mehraufwand erkauft werden. Zudem muss man sich überlegen, wie man Daten und Maschinenbefehle trennen kann. Heute wird die Harvard-Architektur daher nur sehr selten eingesetzt. Den Standard stellt die Von-Neumann-Architektur dar, die in modernen Rechnern um einige Bausteine erweitert wurde.

Eine wichtige Erweiterung für Rechnerarchitekturen sind *Caches*. Hier handelt es sich um schnelle Pufferspeicher, die in unterschiedlichen Speicherebenen zum Einsatz kommen. Ein Cache speichert Daten bzw. Codeteile und dient der Zugriffsoptimierung. Heute werden auch Hierarchien von Caches verwendet. Je näher ein Cache an der CPU liegt, desto höher ist die Zugriffsgeschwindigkeit. Man spricht hier auch von Level- n -Caches (Level-1, Level-2 oder L1, L2, usw.). Je kleiner n ist, desto schneller ist der Cache. Heutige CPUs haben meist zwei oder drei Cache-Levels.

CPU und Hauptspeicher sind über ein schnelles Bussystem – auch CPU-Bus oder Systembus genannt – miteinander verbunden, wobei ein Bussystem im Prinzip eine Menge von parallelen Datenleitungen darstellt. Heutige PCs verfügen über 32 bzw. 64 (und in Zukunft sicher 256 und mehr) Datenleitungen und können damit – vereinfacht ausgedrückt – zwischen CPU und Hauptspeicher genau so viele Bit parallel übertragen. Der CPU-Bus besteht meist aus drei Teilen: Der *Datenbus*, der *Adressbus* und der *Steuerbus*. Über den Datenbus werden Daten zwischen Hauptspeicher und CPU übertragen, der Adressbus dient der Adressierung des Hauptspeichers, und über den Steuerbus werden Steuerinformationen ausgetauscht.

Externe Systeme sind über einen Ein-/Ausgabe-Bus (E-/A-Bus) untereinander und über den CPU-Bus mit der CPU und dem Hauptspeicher verbunden. An externen Systemen gibt es u.a. Plattenspeicher, Bildschirm, Keyboard und Maus, sowie Drucker und Netzwerkadapter. Viele externe Geräte verfügen wiederum über eigene Controller, die mit der CPU zum Zwecke des Datenaustauschs kommunizieren.

Eine Rechneranlage arbeitet taktgesteuert. Das sog. System-Clock-Signal wird von einem Clock-Generator erzeugt. Die Ausführung eines Maschinenbefehls erfordert in der Regel eine festgelegte Anzahl an Bearbeitungsphasen, wobei jede in einem Takt abgearbeitet wird. Bei den meisten Prozessoren sind es mindestens drei bis vier Phasen, die gemeinsam als Maschinenzyklus bezeichnet werden: Diese Phasen sind das Holen vom Speicher, das Dekodieren, das Ausführen des Befehls und das Zurückschreiben der Ergebnisse in den Speicher oder in ein Register. Bei der Entwicklung neuer Prozessoren versuchte und versucht man ständig, die Anzahl der Takte durch eine parallele Ausführung der Phasen zu reduzieren.

Die Rechnerleistung wird über verschiedene Kenngrößen dargestellt. Hierzu gehören die Größe des Hauptspeichers (gemessen in Megabyte bzw. Gigabyte), die Größe der Plattenbereiche (gemessen in Gigabyte) und die Rechengeschwindigkeit der CPU gemessen in MIPS (Millions Instructions Per Second, also Maschinenbefehlsausführungen pro Sekunde).

Oft wird auch noch die Taktfrequenz zur Bewertung herangezogen, die aber heute für sich alleine keine allzu großen Schlüsse zulässt. Es kommt nämlich immer darauf an, was das Rechnersystem in einem Taktzyklus ausrichten kann. Ein handelsüblicher PC hat heute z.B. einen Hauptspeicher von vier GByte, einen Plattenspeicher von 100 und mehr GByte und eine Rechnergeschwindigkeit von mehreren Tausend MIPS.

1.1.4 CPU-Registersatz

Jede CPU enthält einen Registersatz mit einigen kleinen aber schnellen Speichern (sog. Register). Diese schnellen Speicher werden benötigt, um die Maschinenbefehle auszuführen. Alle Register, die durch Software direkt angesprochen werden können, repräsentieren das Programmiermodell eines Prozessors. Darüberhinaus gibt es weitere Register, sog. Hilfsregister, die nur prozessorintern verwendet werden. Je nach Maschinenbefehl wird ein Register oder es werden sogar mehrere Register verwendet. Die Maschinenbefehle schreiben ihre Operanden oft in Register oder lesen Operanden aus Registern. Man nennt die Register je nach Typ Integerregister, Universalregister, Gleitkommaregister, Datenregister, Segmentregister usw.

Weitere, spezielle Register sind der Program Counter (PC, Programmzähler, Befehlszähler), der die Hauptspeicheradresse des nächsten auszuführenden Befehls enthält, das Instruction Register (IR, Befehlsregister), das den aktuellen auszuführenden Maschinenbefehl enthält und das Stack-Pointer-Register (SP), das auf den aktuellen Stack³ verweist. Darüber hinaus gibt es Statusregister, auch *PSW* (= *Program Status Word*) genannt, welche für Vergleichsoperationen benutzt werden, und weitere Kontrollbits wie etwa den aktuellen Modus (Benutzermodus, Kernelmodus), in dem sich die Rechneranlage befindet, enthalten.

³ Was ein Stack ist, wird weiter unten erläutert.

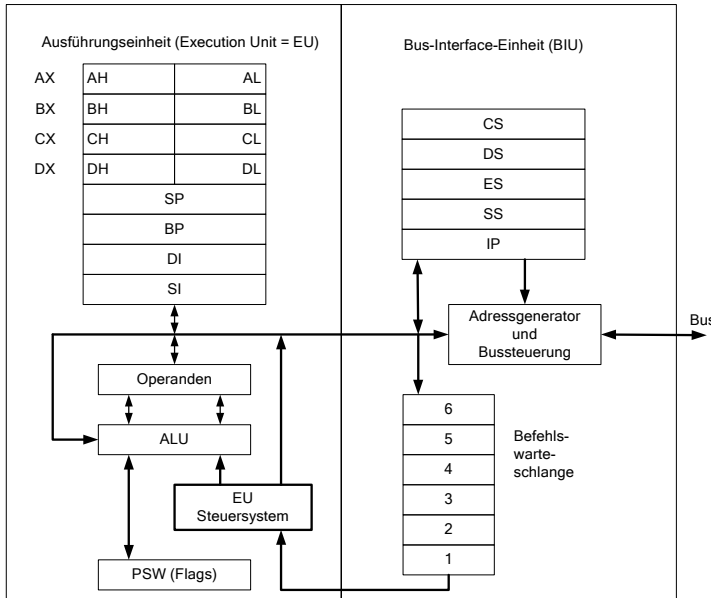


Abbildung 1-3: Blockschaltbild eines Intel 8086-Prozessors

Abbildung 1-3 zeigt das vereinfachte Blockschaltbild inkl. Registersatz des mittlerweile in die Jahre gekommenen Prozessors 8086 von Intel (I-8086). Ohne ins Detail zu gehen, soll der Registersatz des Intel 8086 grob skizziert werden:

- Der Intel 8086 besitzt 16-Bit breite *Register* mit den Bezeichnungen AX, BX, CX und DX, die jeweils auch als zwei 8-Bit-Register (AH, AL,...) verwendbar sind.
- Weiterhin hat er zu Adressierungszwecken das Register CS für die Adressierung des Codesegments, DS für das Datensegment, ES für ein Extrasegment und SS für das Stacksegment. Alle vier Register verweisen jeweils auf die Anfangsadresse eines Speichersegments im Hauptspeicher, da Intel-Prozessoren den Hauptspeicher in Segmenten adressieren.
- Der Befehlszähler heißt beim Intel 8086 *Instruction Pointer* oder kurz *IP*. Mit seinem Inhalt und dem Inhalt des CS wird die Adresse des nächsten Befehls ermittelt.
- Hinzu kommt noch der *Stack-Pointer SP*, der zusammen mit dem Register SS die aktuelle Adresse im Stack adressiert.

DI und SI sind Index-Register, die ebenfalls zur Adressierung verwendet werden können. Das PSW-Register hat die oben erläuterte Bedeutung.

Exkurs: Stack

Unter einem Stack im o.g. Sinne versteht man einen reservierten Teil des Adressraums und ein dazugehöriges Prozessor-Spezialregister, das als Stack-Pointer oder Stapelzeiger bezeichnet wird. Der Stack-Pointer zeigt immer auf die Adresse des oberen Endes des Stapels (Top of Stack). Im Stapelspeicher legt das Betriebssystem die Aufrufinformation (Aufrufrahmen) aller Unterprogramme ab, die noch nicht beendet sind, sowie einige zusätzliche, zwischengespeicherte Daten. Wie wir noch sehen werden, legt das Betriebssystem bei Unterbrechungen oder Funktionsaufrufen die Rücksprunginformationen auf den Stack. Damit kann nach Abarbeitung einer Unterbrechungsroutine oder der aufgerufenen Funktion wieder der Weg zurück zur unterbrochenen Codestelle gefunden werden. Heutige Programmiersprachen nutzen ebenfalls einen Stack für diese Zwecke. In der Softwaretechnik wird unter einem Stapelspeicher auch allgemein eine Datenstruktur mit der oben dargestellten Funktionalität bezeichnet (siehe auch Anhang).

1.1.5 Beispiele für Mikroprozessor-Architekturen

Das oben dargestellte Rechnersystem ist natürlich nur ein sehr einfaches Modell. Heutige Rechnersysteme sind weit komplexer und verfügen über weitere Bausteine wie etwa Pipelines, Gleitkomma-Arithmetik und Caches, und man spricht bei neueren Architekturen auch von superskalaren Architekturen, bei denen die Befehlsbearbeitung mehr und mehr parallelisiert wird, um die Leistung zu verbessern. Betrachten wir beispielhaft einige gängige Mikroprozessoren:

Pentium-Prozessor. Der *Pentium-Prozessor* der Firma Intel⁴ verfügt über eine superskalare Architektur. Er hat einen 64-Bit breiten CPU-Bus, der die CPU mit einem Code-Cache (hier Level-1- bzw. L1-Cache genannt) und einem Daten-Cache (hier Level-2- bzw. L2-Cache genannt) verbindet. Der Intel-80386-Mikroprozessor hatte aus Platzgründen noch keinen Cache. Erst mit der 80486-Architektur wurde ein L1-Cache in der CPU und ein L2-Cache außerhalb der CPU auf dem Motherboard realisiert. Die Maschinenbefehle werden vom Hauptspeicher über den L1-Cache und über einen Prefetch-Puffer in die erforderlichen Register geladen.

⁴ Pentium ist eigentlich eine ganze Serie von Prozessoren mit unterschiedlicher Ausstattung (Pentium Pro, Pentium II, Pentium III, Pentium III Xeon, Pentium 4,...). Andere 32-Bit-Prozessoren von Intel sind z.B. *Celeron M*, *Celeron D*, *Core 2 Duo*, *Core 2 Extreme Quad Core*. Letzterer besteht z.B. aus mehr als 290 Millionen Transistoren und verfügt über 4 Rechnerkerne.

Bei den hier erwähnten Caches handelt es sich um schnelle Speicher (Hardware-Cache) zwischen der CPU und dem Hauptspeicher. Um ein Gefühl für die Leistungsfähigkeit derartiger Caches zu bekommen, sei hier erwähnt, dass das Verhältnis zwischen den Zugriffszeiten auf externe Speichermedien (Platten) und dem Hauptspeicher eine Größenordnung von etwa 100.000 hat, das Verhältnis zwischen Hauptspeicher und Cache liegt zwischen 10 und 100.

	31	15	0	
EAX		AH	AL	AX
EBX		BH	BL	BX
ECX		CH	CL	CX
EDX		DH	DL	DX

Registerbezeichnungen:
 [E]AX: Akkumulator
 [E]BX: Base Register
 [E]CX: Count Register
 [E]DX: Data Register

Abbildung 1-4: Intel-Pentium-Prozessor, allgemeine Register

Zwei ALUs dienen der Durchführung von arithmetisch-logischen und Shift-Operationen. Eine Floating Point Unit (Gleitkommaeinheit) führt Gleitkomma-Operationen aus.

Der allgemeine Registersatz des Intel Pentium (siehe Abbildung 1-4) hat 32-Bit-Register, die auch als 16-Bit- und als 8-Bit-Register verwendbar sind (AX = 16-Bit-Register, AH = 8-Bit-Register). Die allgemeinen Register können alle für arithmetische und logische Operationen verwendet werden.

Itanium-Prozessor. Der neuere *Itanium-Prozessor* von Intel⁵ (vgl. Abbildung 1-5) verfügt über eine echte 64-Bit-Architektur und ist der erste Prozessor mit der sog. IA64-Architektur. Er ist nur noch über einen Firmware-Emulationsmoduls mit der IA32-Architektur kompatibel. Er ist ausgestattet mit 128 Integerregistern (General Purpose Register), 128 Gleitkommaregistern, einigen Ausführungseinheiten (branch units) für die Integerberechnung, die Gleitkommaberechnung sowie einige andere Bausteine zur Optimierung, die hier im Detail keine Rolle spielen.

Die Integerregister sind 64 Bit und die Gleitkommaregister 82 Bit breit. Der L1- und der L2-Cache sind hier in der CPU, während sich ein L3-Cache außerhalb befindet. Die Caches sind natürlich schon viel größer als beim Intel Pentium.

Der Itanium-Prozessor wurde in *EPIC*-Technologie entwickelt und arbeitet mit einer zehnstufigen Hardware-Pipeline. Pro Taktzyklus kann der Prozessor bis zu

⁵ Itanium ist auch eine Serie unterschiedlich ausgestatteter 64-Bit-Prozessoren. Mittlerweile ist auch das Folgeprodukt von Intel, der Prozessor Itanium II, auf dem Markt. Der Itanium-Prozessor besteht je nach Ausprägung aus vielen Millionen Transistoren (z.B. Madison 9M mit 590 Millionen oder Montevino und Montvale mit 1,77 Milliarden).

sechs Maschinenbefehle ausführen. EPIC steht für *Explicit Parallel Instruction Computing* und ist wie *RISC* und *CISC* eine Prozessor-Architekturvariante. *RISC* steht für *Reduced Instruction Set Computer* (siehe SPARC-Prozessoren von Sun) im Gegensatz zu *CISC* (*Complex Instruction Set Computer*, siehe Intel-Prozessoren). EPIC unterstützt eine explizite Parallelisierung des zu verarbeitenden Befehlsstroms. Man geht bei diesem Konzept davon aus, dass der Compiler den Code möglichst optimal vorbereitet, sodass zu verarbeitende Befehle zu Bündeln von jeweils drei Befehlen zusammengefasst werden, die möglichst unabhängig voneinander ausgeführt werden können. Dabei wird angenommen, dass der Compiler das zu verarbeitende Programm besser kennt als der Prozessor und Sprungvorhersagen auch besser ermitteln kann. Die Verlagerung der Parallelisierungsfunktionalität dient auch der Komplexitätsreduzierung.

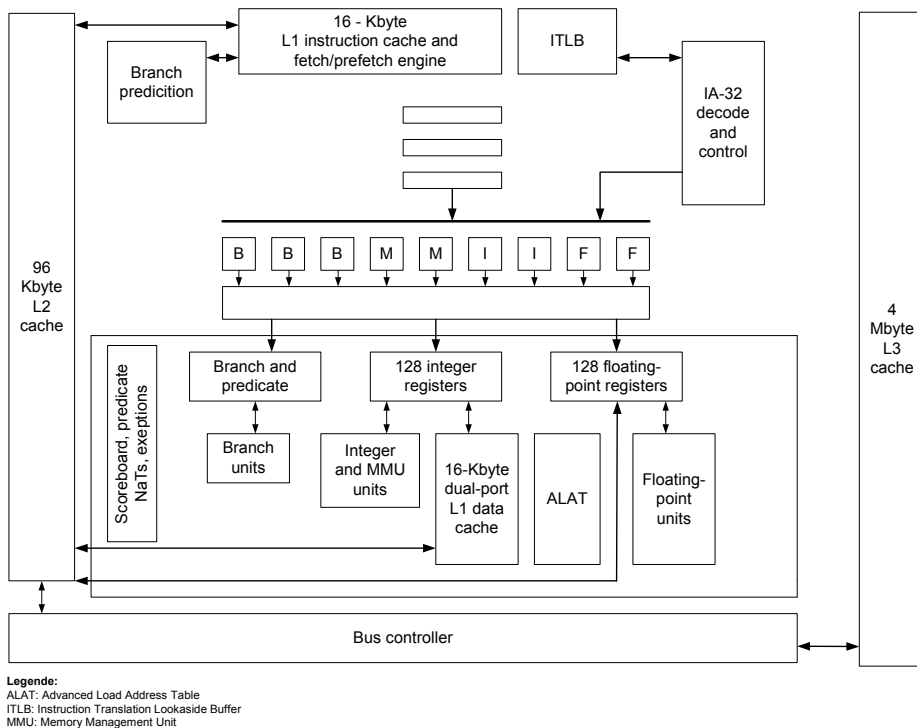


Abbildung 1-5: Blockschaltbild des Intel-Itanium-Prozessors

UltraSparc-Prozessor. Die Firma Sun bietet mit ihrem RISC-Prozessor *UltraSparc III* ebenfalls einen superskalaren Mikroprozessor⁶ mit einer 64-Bit-Architektur, einer 14-stufigen Pipeline und einem 64-Kbyte großen L1-Cache an, jedoch keine echten L2-Cache und auch keine L3-Cache. RISC-Prozessoren verfügen über einen vergleichsweise einfachen (reduzierten) Befehlssatz. RISC und CISC sind zwei grundsätzliche Philosophien im Computerbau.

Weitere moderne Mikroprozessoren sind u.a. der *AMD Athlon* (in PCs verwendet), *AMD Opteron* (in Servern verwendet, Codename *Sledgehammer*), der *MIPS64*, der *IBM Power 4*, der *DEC Alpha* sowie Prozessoren für Großrechner wie etwa der *IBM S/370*. Weiterhin sind noch Vektorrechner interessant, und natürlich sind in heutigen Serverrechnern meist mehrere Mikroprozessoren enthalten, die parallel arbeiten. Vektorrechner sind Rechnersysteme, die auf die Berechnung von Vektoren, also auf Funktionen über Datenstrukturen aus der Vektoralgebra spezialisiert sind. Sie beherrschen also Operationen auf ganzen Vektoren (im Gegensatz zu einfachen Skalaren).

Anmerkungen zu 64-Bit-Architekturen von Intel und AMD

Die IA64- und die IA32-Architektur sind Intel-Architekturen (IA), welche die prinzipiellen Arbeitsweisen von Prozessoren und deren Befehlssätze festlegen. Die IA64-Architektur löst die IA32-Architektur ab. Die IA32-Architektur wird oft auch als i386- oder x86-Architektur bezeichnet und ist die Basis aller Intel-80386-Prozessoren. Die IA64-Architektur wurde speziell für den Intel-Itanium/Itanium2-Prozessor entwickelt.

Das Attribut „64-Bit“ wird in der Computertechnik vielfältig benutzt und trägt daher auch zur Verwirrung bei. Man spricht u.a. von 64-Bit-CPU's, 64-Bit-Hauptspeichern, 64-Bit-Architekturen und 64-Bit-Betriebssystemen. Man kann grundsätzlich sagen, dass 64-Bit-Betriebssysteme 64-Bit-Architekturen unterstützen. Sinngemäß gilt dies auch für die Attribute „16-Bit“, „32-Bit“ und „128-Bit“. Wenn man von einer 64-Bit-Architektur spricht, so meint man allerdings in erster Linie, dass die CPU auf die parallele Verarbeitung von 64 Bit ausgelegt ist. Davon sind die Daten- und Adressbusbreite (64 Datenleitungen, 64 Adressleitungen), die Registergrößen und auch der Befehlssatz betroffen. In erster Linie bezieht man sich aber auf die Adressierung. Ein 64-Bit-Mikroprozessor verfügt über 64 Datenleitungen, mit denen Daten parallel zwischen ALU und Hauptspeicher übertragen werden können. Es ist aber *nicht* unbedingt der Fall, dass alle CPUs mit 64-Bit-Datenpfaden auch 64-Bit-Adresspfade zur Adressierung des Hauptspeichers zur

⁶ Der UltraSparc III besteht aus ca. 29 Millionen Transistoren.

Verfügung stellen. Ein Pentium-Prozessor verfügt z.B. über 64-Bit-Datenpfade, aber nur 32-Bit-Adressleitungen. Auch 64-Bit-Prozessoren nutzen nicht immer 64 Adressleitungen für die Hauptspeicheradressierung.

Zwei wichtige Mitbewerber am Markt der 64-Bit-Mikroprozessoren sind die Unternehmen AMD und Intel. Diese Unternehmen haben für die Entwicklung von 64-Bit-CPU's unterschiedliche Ansätze gewählt. Während Intel mit seiner IA64-Architektur eine Abkoppelung von der 32-Bit-Architektur unternahm, ging AMD mit seiner AMD64-Architektur, die auch als *x64-Architektur*⁷ bezeichnet wird, eine Kompatibilitätsstrategie im Hinblick auf die Kompatibilität zu x86-fähigen Programmen. Die älteren 32-Bit-Prozessoren wurden von AMD um 64-Bit-Register erweitert. Die x64-Architektur ist dadurch zu vorhandener 32-Bit- und sogar 16-Bit-Software abwärtskompatibel. Es gibt entsprechende Kompatibilitätsmodi, die man im Prozessor einstellen kann. Die Register der x64-Architektur sind nur Erweiterungen der 32-Bit-Architektur. Dementsprechend heißen diese Register auch so ähnlich wie die Register der 80x86-Prozessoren (aus EAX wird z.B. RAX).

Linux war das erste Betriebssystem, das eine Unterstützung für den 64-Bit-Modus der x64-Prozessoren bot. Microsoft zog erst später im Betriebssystem Windows XP nach (April 2005), vermarktete die x64-Unterstützung aber nicht besonders intensiv. Mittlerweile unterstützen auch Open-Source-Derivate von BSD-Unix die x64-Architektur.⁸

1.1.6 Multicore-Prozessoren und Hyperthreading-CPU's

Multicore-Prozessoren (auch: Mehrkernprozessor) sind Mikroprozessoren mit mehr als einer vollständigen CPU. Viele Ressourcen mit Ausnahme des Busses und einiger Caches sind repliziert. Als *Dualcore-Prozessor* (Doppelkernprozessor) bezeichnet man einen Multicore-Prozessor mit zwei CPU's. Mikroprozessoren mit einem Hauptprozessor bezeichnet man zur Abgrenzung als Singlecore (Einzelkernprozessor). Bei vier Kernen spricht man von einem *Quadcore-Prozessor* usw.

Multicore-Prozessoren sind kostengünstiger zu entwickeln als mehrere einzelne CPU's. Man kann mit einem Dualcore-Prozessor je nach Anwendung die 1,3- bis 1,7-fache Leistung erbringen als mit einem Singlecore-Prozessor.

Der Einsatz von Multicore-Prozessoren dient auch dazu, das Abwärmeproblem bei Prozessoren zu lösen, da z.B. ein Dualcore-Prozessor unwesentlich mehr Abwärme

⁷ Früher nannte man die Architektur auch AMD64- und noch früher x86-64-Architektur.

⁸ Auf die einzelnen Betriebssysteme kommen wir noch in der historischen Betrachtung.

erzeugt als ein Prozessor mit einem Rechnerkern. Beispiele für Multicore-Prozessoren:

- *Der Athlon 64 X2* ist AMDs Dualcore-Version des Athlon 64.
- *SUN UltraSPARC T1* (Niagara) mit vier, sechs oder acht SPARC-Kernen.
- *Intel Core* ist der Name einer Prozessorfamilie von Intel. Die Variante mit einem CPU-Kern wird *Core Solo*, die Variante mit zwei CPU-Kernen (Dualcore-Prozessor) wird *Core Duo* genannt. Die Intel Core CPUs sind die Nachfolger des *Pentium M* und basieren auch auf dessen Technologie. Sie ersetzen den *Pentium M* auch in Intels *Centrino*-Mobiltechnologie und sind auf niedrige Wärmeentwicklung und geringen Stromverbrauch bei hoher Rechenleistung optimiert.

Im Gegensatz dazu sind sog. *Hyperthreading-CPU*s mehrfädige (engl. multithreading) Singlecore-Prozessoren mit mehreren Programmzählern und Registersätzen sowie Interrupt-Controllern, die sich gegenüber dem Betriebssystem aber als Multicore-Prozessor darstellen. Das Betriebssystem sieht also zwei oder mehrere CPUs (Rechnerkerne), obwohl auf der physikalischen CPU nur einer vorhanden ist. Die Rechnerkerne sind logische Bausteine. Es handelt sich bei Hyperthreading-CPU s also nicht um echte Multicore-Prozessoren.⁹

Hyperthreading wurde von Intel eingeführt. Intel arbeitet sowohl mit Multicore-Prozessoren als auch mit Hyperthreading. Heute werden z.B. Pentium 4 von Intel als Hyperthreading-Prozessoren realisiert. Aus Sicht des Betriebssystems hat damit z.B. eine Dualcore-Maschine mit Hyperthreading vier Rechnerkerne.

1.1.7 Unser einfaches Modell der Hardware

Ein Betriebssystemprogrammierer muss sich intensiv mit der Hardware beschäftigen. Für unsere weitere Betrachtung der Aufgaben und der Funktionsweise von Betriebssystemen verwenden wir ein vereinfachtes Modell der Hardware (vgl. Abbildung 1-6). Für uns sind neben den Mehrzweck- und Gleitkommaregistern einige Steuerregister von Bedeutung (PSW, Befehlsregister). Auch Register zur Adressierung von Daten und Code, wie etwa der Programmzähler, der Stackpointer sowie Basisregister bzw. die Segmentregister unter Intel- und AMD-Prozessoren kommen in unserer Betrachtung vor. Neben der ALU und den Bussystemen spielen auch noch die Pufferspeicher (Cache) eine Rolle. Wir bezeichnen Sie mit L1, L2 und L3. Je nach Prozessortyp sind diese für Code und Daten ge-

⁹ Diese Technik hat auch Vorteile bei der Lizenzierung von Software. Oracle (Datenbankhersteller) rechnet z.B. seine Lizenzen nach der Anzahl der physikalischen Rechnerkerne ab, was bei Hyperthreading zu Einsparungen führt.

meinsam oder separat verfügbar. Der L1-Cache ist immer in der CPU integriert, L2- und L3-Caches können auch außerhalb der CPU auf der Hauptplatine eines Rechners platziert werden. Wie wir bei der Speicherverwaltung noch sehen werden, spielen die Bausteine MMU (Memory Management Unit) und TLB (Translation Lookaside Buffer) eine wichtige Rolle. Weiterhin ist der Interrupt-Controller für die Ein-/Ausgabe von Bedeutung (siehe Kapitel 3).

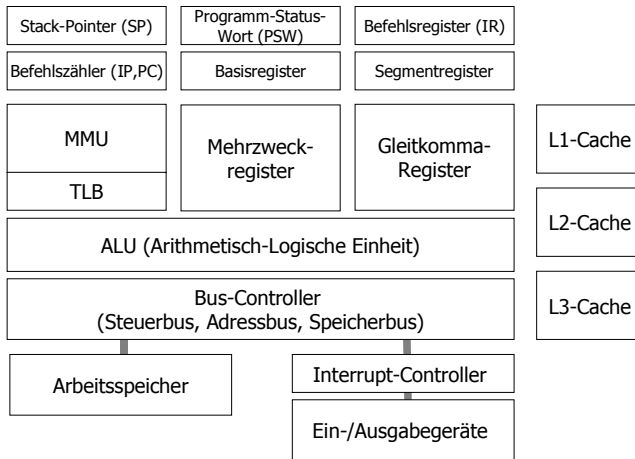


Abbildung 1-6: Vereinfachtes Hardwaremodell

1.2 Betriebssystem-Geschichte

Bevor wir die Entwicklungsgeschichte der Betriebssysteme betrachten, wollen wir eine Kategorisierung versuchen und den Begriff des *Mehrzweckbetriebssystems*, das wir im Weiteren in den Fokus unserer Betrachtung stellen, einführen.

Betriebssystemkategorien. Man kann heutige Betriebssysteme grob in mehrere Kategorien einordnen, und wie man sieht, werden manche Betriebssysteme auch für verschiedene Zwecke verwendet:

- Großrechner oder Mainframes (High-End-Systeme wie IBM OS/390 (heute IBM zSeries¹⁰) und Siemens Fujitsu BS 2000/OSD)
- Serverbetriebssysteme (Unix, Windows NT/2000/2003/XP,...)
- PC-Betriebssysteme (Windows-Derivate, Linux,...)
- Echtzeitbetriebssysteme (VxWorks, QNX, Embedded Linux, NetBSD,...)

¹⁰ zSeries heißt die aktuelle Architektur der Mainframes der Firma IBM. Gegenüber der Vorgängerarchitektur S/390 unterscheidet sich zSeries u.a. durch die 64-Bit-Adressierung.

- Embedded Systems (VxWorks, QNX, Embedded Linux, NetBSD, Microsoft Windows CE)
- Handheld-Computer (Palm OS, Windows CE, Symbian)
- Smartcard-Betriebssysteme (Chipkarte mit spezieller Java Virtual Machine, JVM)

Typisch für Echtzeitsysteme ist, dass hier die Forderung nach einer garantierten maximalen Reaktionszeit besteht, in der auf ein externes Ereignis (Signal eines technischen Prozesses) reagiert werden muss.

Embedded Systems (eingebettete Systeme) können als Spezialfall von Echtzeitsystemen betrachtet werden (Baumgarten 2006). Als Embedded System bezeichnet man ein Rechner- bzw. Steuerungssystem, das in Geräten (Telefone, DVD-Player, Waschmaschinen, Fernseher), Fahrzeugen (Flugzeuge, Autos) oder Robotern eingebaut ist und dort seine Aufgaben meist unsichtbar verrichtet. Embedded Systems kann man weitgehend als geschlossene Systeme betrachten, die eine dedizierte Aufgabe übernehmen. Das Betriebssystem ist klein ausgelegt, da meist wenige Ressourcen verfügbar sind. Oft verfügen Embedded Systems aufgrund der knappen Ressourcen nicht über ein Betriebssystem. Die Software bedient dann direkt die Hardwareschnittstellen. Embedded Systems sind oft auch Echtzeitsysteme.

Weiterhin kann man noch Netzwerkbetriebssysteme (wie Novell Netware) und verteilte Betriebssysteme als spezielle Betriebssystemkategorien sehen.

Universalbetriebssysteme. Unter die Kategorie *Universal-* oder *Mehrzweckbetriebssystemen* (engl.: general purpose operating system) fallen Betriebssysteme wie Unix und Windows, aber auch Großrechnerbetriebssysteme, die für verschiedenste Zwecke, meist aber nicht für strenge Realzeitanforderungen einsetzbar sind. Diese Betriebssysteme findet man häufig in Unternehmen. Sie sind besonders gut für betriebliche Informations- und Steuerungssysteme und für Systeme in der Verwaltung geeignet.

1.2.1 Historische Entwicklung

Sehr interessant ist die rasche Entwicklung der Betriebssysteme in den letzten Jahren. Betrachtet man die Entwicklung von Betriebssystemen seit 1945, so kann man mehrere Generationen abgrenzen (Tanenbaum 2002). Wir sehen vor allem vier Generationen, die man aber auch durchaus noch weiter unterteilen könnte. Diese vier Generationen sollen nun kurz vorgestellt werden (siehe auch Abbildung 1-6), wobei die verwendeten Begriffe zur Aufzählung der in den einzelnen Generationen möglichen Betriebsarten (wie Multiprogramming und Timesharing) weiter unten noch näher erläutert werden. Wir beginnen mit der Nachkriegsgeneration.

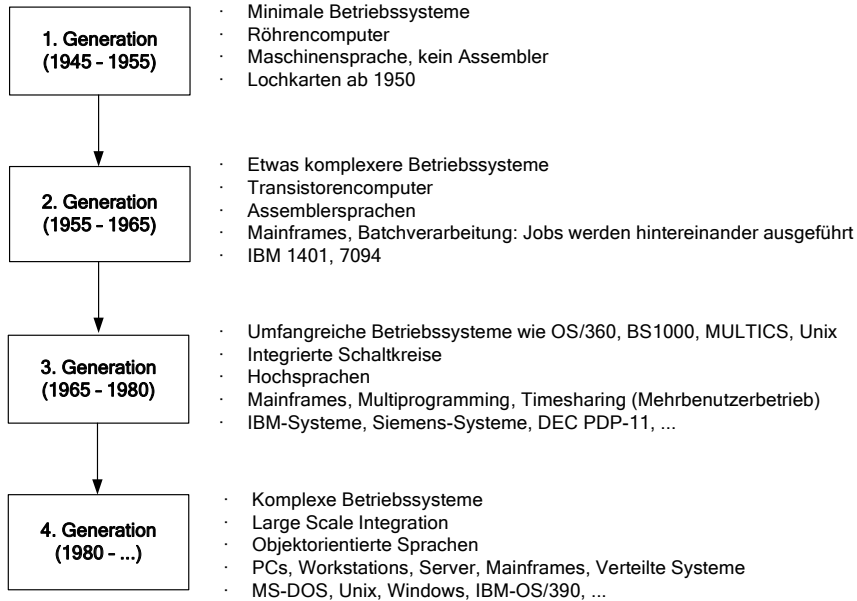


Abbildung 1-6: Entwicklung von Betriebssystemen

1. Generation (ca. 1945 – 1955). Diese Generation war geprägt durch Röhrencomputer und der Programmierung in reiner Maschinensprache (kein Assembler, keine Hochsprache) Elektronenröhren wurden als Schaltelemente verwendet (ca. 20.000 Elektronenröhren). Ab ca. 1950 wurden auch Lochkarten eingesetzt.

2. Generation (ca. 1955 – 1965). In dieser Generation wurden bereits Transistoren¹¹ verwendet. Die Verarbeitung wurde Stapelverarbeitung (Batch-Verarbeitung) genannt, wobei die IBM-Systeme 1401 und 7094 hier sehr bekannt waren. Jobs wurden von Lochkarten auf ein Magnetband eingelesen und dann hintereinander abgearbeitet. Ein Programm nach dem anderen wurde ausgeführt, die Ergebnisse auf Band gespeichert und am Ende ausgedruckt. Die Hardware wurde über ein recht einfaches Betriebssystem gesteuert.

¹¹ Ein Transistor ist ein Halbleiterbauelement zum Schalten und Verstärken von elektrischen Signalen ohne mechanische Bewegung. Nach Moore's Law verdoppelt sich die Anzahl der Transistoren in Prozessoren ca. alle 18 Monate.

3. Generation (ca. 1965 – 1980). Durch Integrated Circuits (ICs), kleinere integrierte Schaltungen, wurden die Rechner immer kleiner¹². Bekannte Systeme zu dieser Zeit waren IBM System/360 (Serie von Rechnern) und IBM System/370, 3080 und 3090. Multiprogramming (Mehrprogrammbetrieb, Multitasking) wurde eingeführt. Mehrere Programme liefen also bereits gleichzeitig oder quasi-gleichzeitig im Rechnersystem ab. Während der E/A-Wartezeit wurde die CPU für einen neuen Job vergeben. Das Spooling, also das Übernehmen von auszuführenden Jobs von einem Plattenspeicher, wurde entwickelt. Die Ergebnisse wurden von den Jobs wieder auf einen Plattenspeicher geschrieben. Später kam das sog. Timesharing (mit Mehrbenutzerbetrieb) als Variante des Multiprogrammings hinzu. Der Online-Zugang über Terminal war nun möglich, und die CPU wurde auf alle Benutzer aufgeteilt. Am M.I.T. entwickelte man die Betriebssysteme CTSS¹³ und MULTICS¹⁴. DEC entwickelte die Minicomputer DEC PDP-1 und PDP-11 auf denen Unix als Betriebssystem eingeführt wurde.

4. Generation (ca. 1980 – heute). In dieser Generation wurden Personal Computer und Workstations entwickelt und eingesetzt. Durch Large-Scale-Integration (LSI-Schaltungen) mit Tausenden von Transistoren auf einem Silizium-Chip wurde die Hardware immer kleiner und schneller. Die Betriebssysteme IBM OS/360, MS-DOS, Unix, Unix BSD, Unix System V, IBM OS/2, Microsoft Windows-Derivate und Linux kamen auf dem Markt. Die Benutzerfreundlichkeit stieg immer mehr durch komfortable Bediensysteme, sog. grafische Oberflächen (X-Windows, Motif, OS/2 Presentation Manager).

1.2.2 Geschichte von Microsoft Windows und Unix

Das Betriebssystem mit der größten Verbreitung ist Windows, das aus MS-DOS (Microsoft Disk Operating System) hervorging. MS-DOS wurde 1981 im IBM-Auftrag von Microsoft entwickelt. Microsoft kaufte es seinerseits von der Firma Seattle Computer Products. Es wurde zunächst auf Basis eines Intel-8088-Prozessors als 8-Bit-Betriebssystem (es unterstützte also einen 8-Bit-Datenbus) auf dem Markt gebracht und bis MS-DOS V8.0 weiterentwickelt.

¹² Integrierte Schaltkreise (IC) sind auf einem einzigen Stück Halbleitersubstrat untergebrachte elektronische Schaltungen.

¹³ CTSS ist die Abkürzung für Compatible Timer Sharing System.

¹⁴ MULTICS steht abkürzend für "MULTIplexed Information and Computing Service".

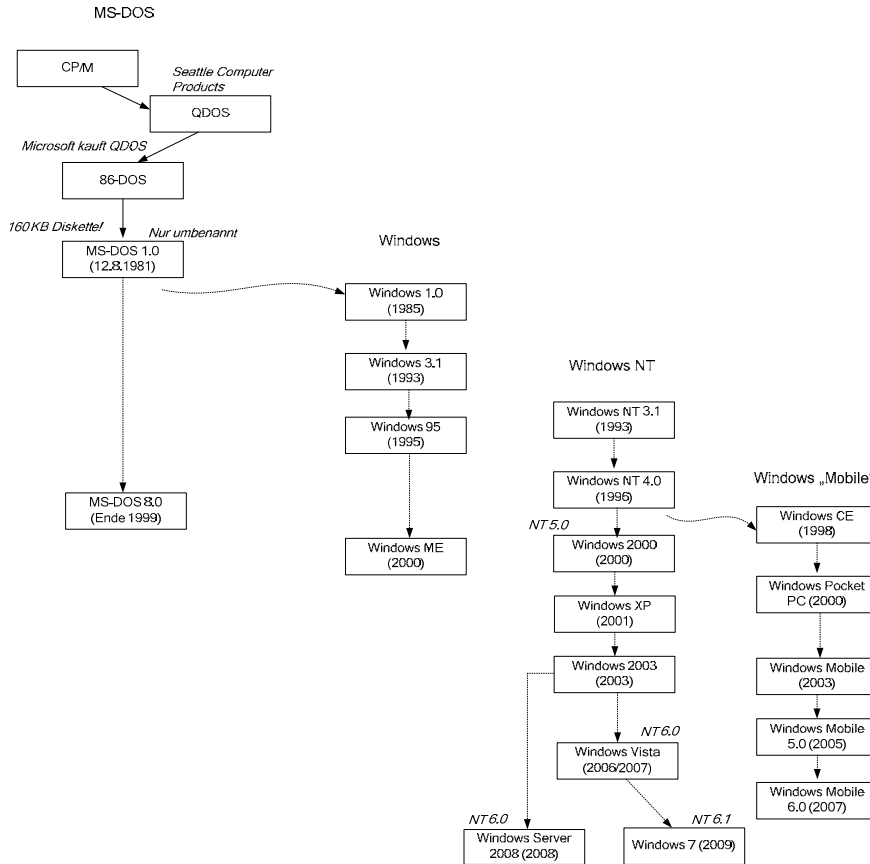


Abbildung 1-7: Entwicklung des Betriebssystems Windows

MS-DOS war ein einfaches Betriebssystem mit Kommandozeilen-orientierter Benutzeroberfläche. Die Basis war QDOS (quick and dirty operating system), Betriebssystem für den Mikroprozessor Intel 8088. QDOS wurde 1981 von Microsoft übernommen und zu MS-DOS weiterentwickelt und wurde von Tim Paterson bei der Firma Seattle Computer Products entwickelt. Das Betriebssystem war CP/M sehr ähnlich, hatte aber einige Verbesserungen bei der Dateiverwaltung, z.B. das FAT-Dateisystem, das wir in Kapitel 8 noch betrachten werden. Die Ursprungsversion von MS-DOS unterstützte noch keine Festplatte, sondern nur Floppy-Laufwerke.

Windows war zunächst nur eine grafische Oberfläche für MS-DOS, entwickelte sich anfangs sehr langsam voran und war durch die Kompatibilität zu MS-DOS

sehr eingeschränkt und auch fehleranfällig. Erst mit Windows NT und Windows 95 gelang der Durchbruch. Heute sind mit Windows 2003/2008 bzw. XP/Vista und Windows 7 leistungsfähige Betriebssysteme auf dem Markt, die sowohl im Arbeitsplatzbereich, als auch im Serverbereich eingesetzt werden. Die Geschichte von Windows und dessen Ursprung in MS-DOS bzw. QDOS ist in Abbildung 1-7 etwas vereinfacht skizziert. Das heutige Windows setzt immer noch auf Windows NT (New Technology) auf. Die Planung von Microsoft ist nicht genau durchschaubar. So ist noch nicht ersichtlich, wann ein Betriebssystem auf dem Markt kommt, das nicht mehr auf NT basiert.

Windows NT hatte im Jahre 2000 einen Codeumfang von ca. 29 Mio. LOC (Lines of Code). Bei Windows Vista spricht man bereits von einem Codeumfang von 50 Mio. LOC. Dem gegenüber wirkt Linux, ein Unix-Derivat, mit ca. 2,2 Mio. LOC sehr schlank (Tanenbaum 2002).

Unix entstand aus dem Betriebssystem MULTICS zunächst unter einer PDP-7 und wurde von Ken Thompson und Dennis Ritchie in den Bell Labs entwickelt. Danach entstanden zwei inkompatible Versionen: BSD Unix aus der Berkeley University als Vorgänger von Sun OS und System V von AT&T. Das BSD-System findet man heute in Derivaten wie FreeBSD, OpenBSD, BSDI, Mac OS X und NetBSD wieder.

Standardisierungsbemühungen der IEEE¹⁵ brachten gegen Ende der 80er Jahre einen Standard namens POSIX (Portable Operating System Interface) hervor, der eine einheitliche Schnittstelle eines kompatiblen Unix-Systems definiert. Andere Standardisierungsbemühungen gibt es von der *Open Group*¹⁶, die durch diverse Unix-Hersteller unterstützt wird. Weiterhin gilt das *System V*, das ursprünglich von den *USL* (Unix System Laboratories) von AT&T stammt, als Standard gehandelt. Wir haben also derzeit drei Standardisierungsbemühungen im Unix-Umfeld (siehe Abbildung 1-8).

Die neueste Standardisierung von Unix-Systemen ist die sog. *Single Unix Specification*, die seit dem Jahr 2003 in einer Version 3 vorliegt und auch durch einen internationalen Standard der ISO/IEC¹⁷ genormt ist. Dieser Standard wurde gemeinsam

¹⁵ IEEE = Institute for Electrical and Electronic Engineers, vergleichbar mit VDI in Deutschland.

¹⁶ Die Open Group ist ein Konsortium aus mehreren Unternehmen, die (u.a.) Unix-Systeme herstellen.

¹⁷ ISO = International Standardisation Organisation; die Unix-Norm heißt 9945:2003. IEC = International Electrotechnical Commission; IEC ist ein internationales elektrotechnisches

von IEEE und der Open Group entwickelt. Den Normierungen sind im Wesentlichen Schnittstellenspezifikationen für die Unix-Kommandos und Systemdienste unterworfen. Die Unix-Hersteller müssen nachweisen, dass sie die Standards erfüllen, und dürfen dann ihre Unix-Produkte als standardkonform bezeichnen. Eine Zertifizierung nach dem ISO-Standard erlaubt z.B. die Führung des Labels *Unix 03*.

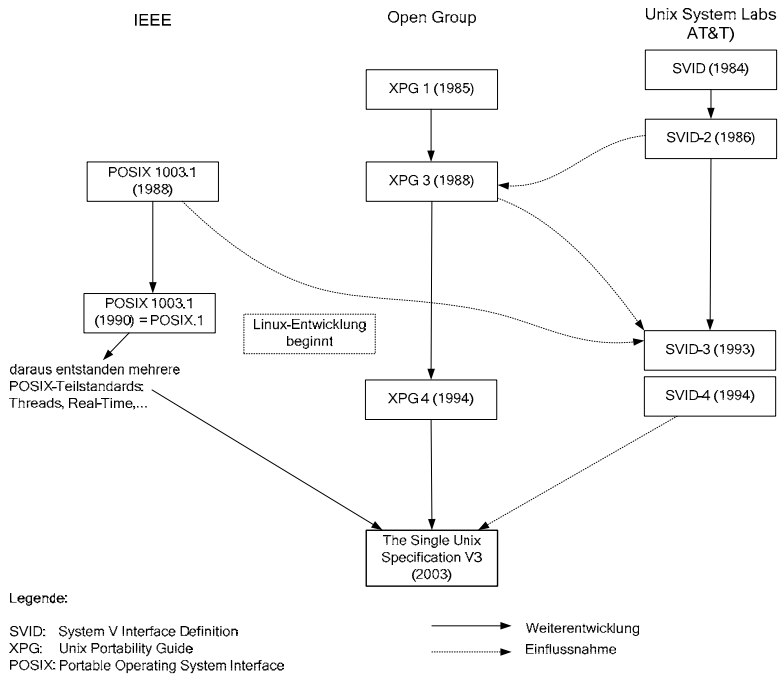


Abbildung 1-8: Unix-Normierung

Linus Torvalds begann schließlich 1991 aus dem Unix-Clone MINIX von Tanenbaum das Betriebssystem Linux als Open-Source-Unix zu entwickeln. Linux V1.0 wurde schließlich im Jahre 1994 und Linux 2.0 im Jahre 1996 freigegeben. Heute ist Linux das populärste Unix-Derivat und ist zertifiziert nach POSIX 1003.1a. Weiterhin implementiert Linux auch die Systemaufrufe von SVID Release 4 und BSD-Unix.

Normierungsgremium mit Sitz in Genf für Normen im Bereich der Elektrotechnik und Elektronik. Einige Normen werden gemeinsam mit ISO entwickelt, darum ISO/IEC.

1.3 Übungsaufgaben

1. Nennen Sie fünf Betriebssystemkategorien!
2. Was ist ein Von-Neumann-Rechner und wie unterscheidet er sich von einem Harvard-Rechner?
3. Wozu braucht man in Computern CPU-Register?
4. Was ist ein Program Status Word (PSW) und wozu wird es verwendet?
5. Was ist ein Mehrzweck- oder Universalbetriebssystem?

2 Betriebssystemarchitekturen und Betriebsarten

Dieses Kapitel führt in Betriebssystemenarchitekturen und Betriebsarten für Betriebssysteme ein. Häufig verwendete Begriffe zu historischen und aktuell üblichen Betriebsarten wie Multiprogramming, Multitasking und Timesharing werden gegeneinander abgegrenzt. Spezielle Betriebsarten wie Teilnehmer- und Teilhaberbetrieb sowie der Application-Server-Betrieb werden eingeführt. Der heute wieder an Bedeutung gewinnende Terminalbetrieb, allerdings im Gegensatz zu früher mit grafischer Oberfläche anstelle von blockorientierten, alphanumerischen Terminals, wird ebenfalls kurz vorgestellt. Dabei wird erläutert, was ein Terminalserver ist. Einige Spezialkonzepte wie virtuelle Maschinen (bzw. Virtualisierung), die heute auch an Bedeutung zunehmen, runden diese Einführung ab.

Zielsetzung des Kapitels

Der Studierende soll die gängigen Architekturvarianten verstehen und erläutern können. Weiterhin sollen die verschiedenen Betriebsarten wiedergegeben werden können.

Wichtige Begriffe

Kernel, Betriebssystemkernel, Timesharing, Stapelsystem, Teilnehmer- und Teilhaberbetrieb, Transaktionsmonitor, Application-Server, Multiprocessing, Multitasking, Multiuserbetriebssystem, Mehrzweckbetriebssystem, Verteiltes System, Terminalserver.

2.1 Betriebssystemarchitekturen

2.1.1 Klassische Architekturen

Betriebssysteme sind sehr komplexe Softwaresysteme und intern in Komponenten bzw. Module strukturiert. Etwas ältere Betriebssysteme besitzen meist einen rein monolithischen Betriebssystemkern (Kernel), wie dies in Abbildung 2-1 dargestellt ist.

Alle Module eines monolithischen Kerns laufen in einem privilegierten Modus, dem sog. Kernelmodus ab (siehe unten) und haben den kompletten Kerneladressraum im Zugriff. Meist findet man einen sog. Dienstverteiler, der die Systemaufrufe der Anwendungen entgegennimmt und an die einzelnen Kernelmodule weiterleitet. Die ursprüngliche Struktur des Betriebssystems wird ggf. im Zuge der

Weiterentwicklung aufgegeben. Das alte MS-DOS, aber auch ältere Unix-Derivate und auch Linux verfügen über eine derartige Architektur.

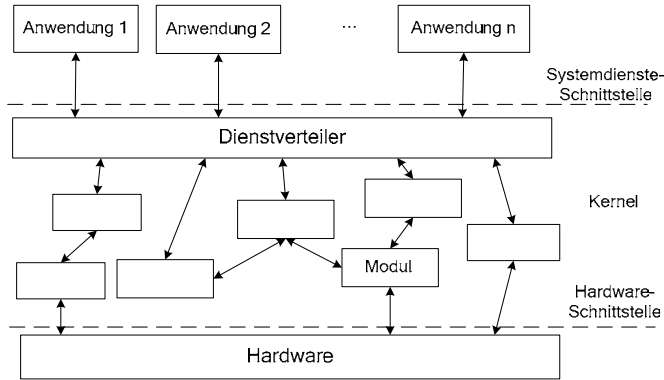


Abbildung 2-1: Monolithischer Kernel

Als Weiterentwicklung bzw. Verbesserung des monolithischen Kerns wurden neuere Betriebssysteme üblicherweise in Schichten strukturiert, wie dies in Abbildung 2-2 beispielhaft dargestellt ist. Dadurch gelang es, den Kernel flexibler und überschaubarer zu gestalten. Einen Standard für die Schichtung gibt es allerdings nicht, da dies vom jeweiligen Design des Systems abhängt. Höhere Schichten nutzen die Module der darunter liegenden Schicht und nur in äußersten Ausnahmefällen und wohlbegründet (z.B. aufgrund von Leistungsargumenten) tiefere Schichtenfunktionalität direkt. Die unterste Schicht dient meist dem Zugriff auf die konkrete Hardware. Abhängigkeiten von der Hardware sind also in dieser Schicht gekapselt, was eine Portierung auf eine andere Hardwareplattform erleichtert. Die Schnittstelle zur Hardware wird oft noch – zumindest teilweise – in Assemblersprache programmiert.

Typische Komponenten bzw. Schichten des Kernels sind Memory-Manager (Speicherverwaltung), Prozess-Manager (Prozessverwaltung), File-Manager (Dateiverwaltung), E/A-Manager (Ein-/Ausgabeverwaltung), Netzwerk-Manager (Netzwerkzugang).

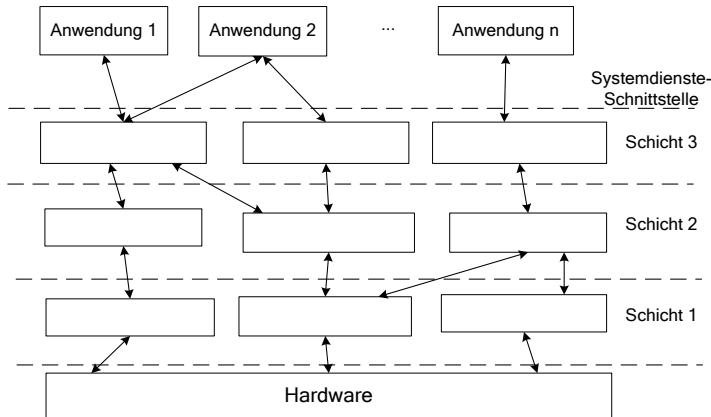


Abbildung 2-2: Schichtenorientierter Kernel

Je nach Architektur sind die Komponenten des Kernels unterschiedlich realisiert und laufen zum Teil im privilegierten oder im nicht-privilegierten Modus, man spricht hier auch von Benutzer- oder Kernelmodus, ab. Die beiden Modi sind für heutige Betriebssysteme typisch. Sie haben folgende Bedeutung:

- Im *Benutzermodus* (Usermodus) laufen üblicherweise Anwendungsprogramme ab. In diesem Modus ist ein Zugriff auf kernelspezifische Code- und Datenbereiche nicht möglich.
- Im *Kernelmodus* (privilegierter Modus) werden Programmteile des Betriebssystems ausgeführt, die einem gewissen Schutz unterliegen. Damit kann das Betriebssystem auch eine Abschottung von Datenstrukturen und Codeteilen des Betriebssystems vor Zugriffen aus Anwendungsprogrammen heraus vornehmen. Wir werden im Weiteren noch auf die Nutzung dieser Modi eingehen.

Als Beispiele heute weit verbreiteter Betriebssysteme mit Schichtenarchitektur sind verschiedene Unix-Derivate (Ableitungen von Unix sind HP UX, Sun Solaris,...) und Windows-Derivate (2000, 2003, XP, Vista, 7, 2008) zu nennen.

2.1.2 Mikrokern-Architektur

Im Gegensatz zu monolithischen Architekturen wird eine Betriebssystemarchitektur, die einen leichtgewichtigen Kernel enthält, als Mikrokern-Architektur (auch Mikrokern-Architektur) bezeichnet. Der Kernel wird bei dieser Architekturvariante dadurch entlastet, dass Funktionalität in Anwendungsprozesse, sog. Serverprozesse, in den Benutzermodus ausgelagert wird. Nicht die gesamte Kernelfunktionalität läuft also im Kernelmodus. Der Kernel übernimmt hier im Wesentlichen die Abwicklung der Kommunikation zwischen Client- und Serverprozessen. Bei-

spiele für Mikrokern-Betriebssysteme sind Mach von der Carnegie Mellon University (CMU), Hurd, L4 sowie Amoeba und Chorus (Tanenbaum 2009). Insbesondere Mach wird heute noch weiterentwickelt und stellt auch die Basis für das Betriebssystem MAC OS X dar.

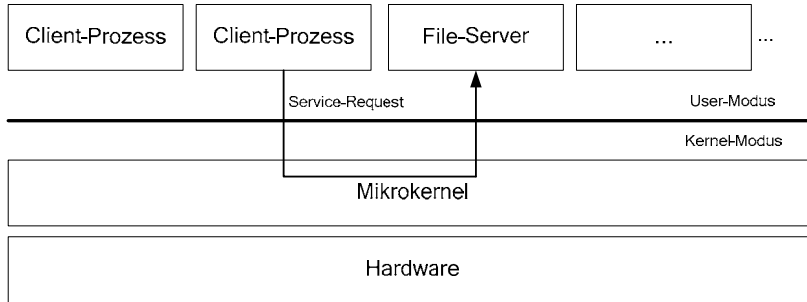


Abbildung 2-3: Mikrokern-Architektur

Die Mikrokern-Architektur entstand aus dem Trend, den Kernel „leichter“ zu machen. Typische Serverprozesse können hier z.B. ein File-Server oder ein Memory-Server sein. Clientprozesse greifen bei dieser Betriebssystemarchitektur auf den Mikrokern über eine Service-Request-Schnittstelle zu. Der Mikrokern leitet die Requests an die entsprechenden Serverprozesse, die in Anwendungsprozessen ablaufen, weiter und stellt umgekehrt den Clientprozessen die Ergebnisse zu. Generell kann man sagen, dass die Mikrokern-Architektur nicht sehr leistungsfähig ist und daher auch in der kommerziellen Praxis in ihrer reinen Form nicht angewendet wird. Auch kommerzielle Mach-Implementierungen sind ähnlich wie Windows z.B. so ausgelegt, dass zumindest der Memory-Manager, der Netzwerk-Manager, der E/A-Manager und der File-Manager im Kernelmodus ablaufen (Solomon 2005). Abbildung 2-3 verdeutlicht diese Architekturvariante (nach Tanenbaum 2002).

2.1.3 Verteilte Systeme und Middleware

Die Mikrokern-Architektur vereinfacht auch eine Verteilung der Serverprozesse auf mehrere Rechner in einem Netzwerk. Ein Mikrokern lässt sich also vom Konzept her relativ einfach zu einem *verteilten Betriebssystem* ausbauen.

Im praktischen Einsatz findet man kaum echt verteilte Betriebssysteme (Abbildung 2-4). Vielmehr setzt man sog. *Kommunikations-Middleware*¹ wie CORBA (Common Object Request Broker Architecture), Java RMI (Java Remote Method Invocation) oder Java EJB (Enterprise Java Beans) ein, um verteilte Anwendungssysteme zu realisieren. Kommunikations-Middleware wird in der Regel im Benutzermodus betrieben und stellt dem Anwendungsprogramm komfortable Dienste zur Kommunikation mit anderen, in einem Netzwerk verteilten Bausteinen zur Verfügung.

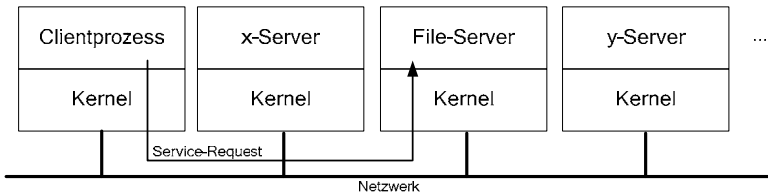


Abbildung 2-4: Echt verteiltes Client-/Server-Betriebssystem

Jeder Rechnerknoten verfügt über ein komplettes Betriebssystem, wobei durchaus unterschiedliche Betriebssysteme zum Einsatz kommen können. In der Regel ist die Middleware in den kommunizierenden Anwendungsprozessen implementiert (siehe Abbildung 2-5). Der Anwendungsprogrammierer muss sich damit befassen, wo sein Service-Request ausgeführt wird (keine echte Verteilungstransparenz). Dem Benutzer einer Anwendung, die auf mehrere Rechnersysteme verteilt ist, kann die Verteilung aber verborgen werden. Klassisches Beispiel für verteilte Anwendungen, in der auch verschiedene Middleware-Komponenten zum Einsatz kommen, sind heute WWW-Anwendungen.

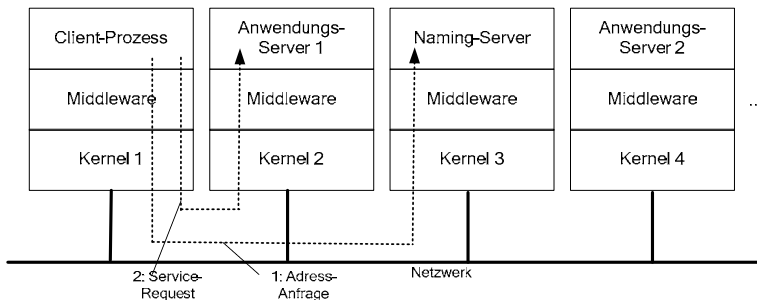


Abbildung 2-5: Einsatz von Kommunikations-Middleware

¹ Es gibt auch andere Arten von Middleware, die sich nicht mit der Kommunikation befassen. Beispiel hierfür ist Middleware für den Zugriff auf persistente (dauerhafte) Datenhaltung.

2.1.4 Betriebssystem-Virtualisierung

Die Client-/Server-Bewegung brachte mit sich, dass die Anzahl der Server in den Unternehmen extrem gestiegen ist. Im Zuge der Konsolidierung der Serverlandschaft ist in den letzten Jahren das Konzept der Virtualisierung wieder modern geworden. Dabei können Gast-Betriebssysteme auf einem Host- bzw. Wirtsbetriebssystem zum Ablauf kommen. Sinn und Zweck von virtuellen Maschinen ist es, mehrere logische Betriebssysteme auf einer gemeinsamen Hardware ablaufen zu lassen. Virtualisierung erreicht man entweder direkt auf der Hardware-Ebene oder über den Einsatz einer Virtualisierungssoftware. Bei der Virtualisierung über Software stellt ein Wirtsbetriebssystem für die Gastbetriebssysteme Betriebssystemcontainer bereit. Alternativ dazu kann ein sog. VMM (Virtual Machine Monitor, auch Hypervisor genannt) für die Gastbetriebssysteme eine vollständige Hardware emulieren. Den Gast-Betriebssystemen werden die erforderlichen Ressourcen wie CPUs, Hauptspeicher, Ein-/Ausgabe-Geräte usw. vom Host-Betriebssystem entweder direkt oder über einen VMM zugeordnet. Prozessorhersteller wie *Intel* unterstützen die Virtualisierung bereits im Prozessor (Intel Virtualization Technology, IVT).

Schon in den frühen Mainframe-Systemen hat man immer wieder das Konzept der virtuellen Maschinen eingesetzt. Typischer Vertreter für diese Architektur war und ist das Betriebssystem *z/VM* von *IBM*. *z/VM*, das früher auch als *VM/CMS* bezeichnet wurde, war schon in frühen Jahren in der Lage, eine Vielzahl von virtuellen Maschinen ablaufen zu lassen. Die Geschichte von *z/VM* begann mit *CTSS* in den 50er Jahren. Heute dient *z/VM* als Basissystem für mehrere 1000 Linux-Systeme auf einem Großrechner.

Das Grundkonzept wird in Abbildung 2-6 skizziert. Als virtuelle Basismaschine (Host) kann z.B. das Produkt *VMware*² dienen. Wie das Bild zeigt, können auf der Basismaschine konkrete Betriebssysteme wie *Windows* oder *Linux* aufsetzen. Die Basismaschine simuliert die Hardware und stellt den einzelnen Gast-Betriebssystemen eine Ablaufumgebung bereit. Die Betriebssysteme laufen völlig isoliert voneinander.

Mit *VMware*, *z/VM* oder ähnlichen Systemen lassen sich also mehrere Maschinen mit verschiedenen Betriebssystemen gleichzeitig virtualisieren. Die virtualisierten Betriebssysteme sind in Abhängigkeit vom Speicherausbau typischerweise etwas langsamer als vergleichbare Installationen auf identischer Hardware. Wie bereits

² *VMware Workstation* ist ein Produkt der Firma *VMware*. Produkte sind u.a. *VMware GSX Server*, *VMware ESX Server* und *VMware Infrastructure 3*. Siehe auch www.vmware.com.

einführend erwähnt, nutzt man diese Technik heute, um die Anzahl der Serverrechner im Unternehmen zu reduzieren. Man spricht von einer virtuellen Infrastruktur, von der man sich neben Kosteneinsparungen auch eine einfachere Administrierbarkeit verspricht.

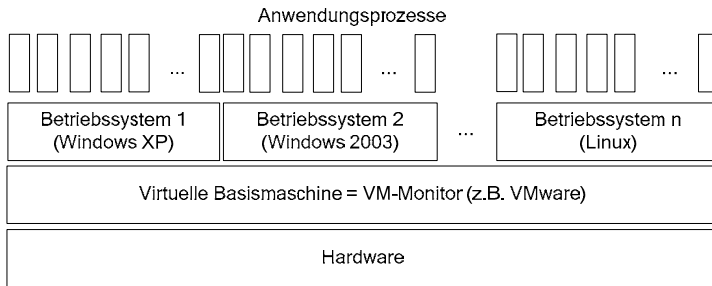


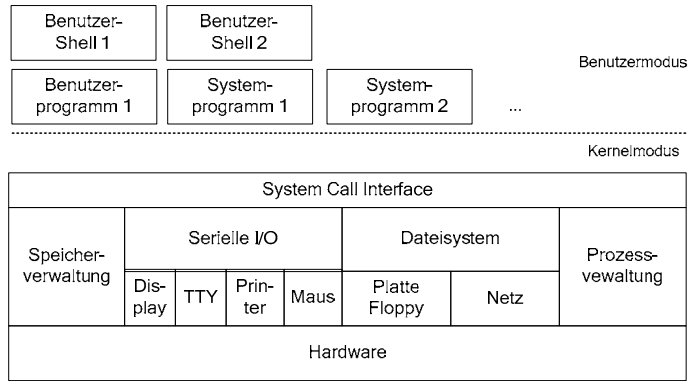
Abbildung 2-6: Virtualisierung von Betriebssystemen

Eine andere Art virtueller Maschinen entwickelte sich in den letzten Jahren im Bereich der Laufzeitumgebungen von Programmiersprachen. Moderne Sprachen wie Java und .NET-Sprachen erzeugen heute keinen Maschinencode mehr, sondern einen Zwischencode, der dann von einer virtuellen Maschine wie der JVM (Java Virtual Machine) oder der CLR (Common Language Runtime) unter .NET interpretiert und in Maschinenbefehle umgesetzt wird. Diese virtuellen Maschinen sorgen für eine Entkopplung der Anwendungssoftware vom konkreten Betriebssystem. Zum Ablauf einer Anwendung ist nur noch die virtuelle Maschine notwendig, die heute üblicherweise im Anwendungsprozess und nicht im Kernel liegt.

2.1.5 Architektur von Unix und Windows

Im Folgenden sind die vereinfachten Betriebssystemarchitekturen von Unix und Windows (2003/XP/2008/Vista/7) kurz dargestellt. Bei Unix ist eine klare System-call-Schnittstelle vorhanden, welche den Zugang zur Kernelfunktionalität ermöglicht. Jedes Benutzerprogramm wird in der Regel von einer sog. Shell aus gestartet. Eine Shell stellt einen Kommando-Interpreter dar. Beispiele hierfür sind die Korn-Shell und die Bourne-Shell. Innerhalb der Shell können dann weitere Programme gestartet werden. Wie man aus der Abbildung 2-7 ersehen kann, gibt es im Unix-Kernel Komponenten wie die Speicherverwaltung, das Dateisystem, die Prozessverwaltung usw.

Windows (bzw. dessen Vorgänger) wurde so konzipiert, dass auch Nicht-Windows-Anwendungen wie OS/2- und POSIX-Anwendungen in diesem Betriebssystem ablaufen können. Bei Windows ist die Architektur daher etwas komplizierter.



den im Wesentlichen die Implementierungen der Systemdienste über einen Dispatching-Mechanismus zugänglich gemacht. Diese Dienste werden wiederum über die Windows API aus dem Usermodus heraus aufgerufen. Weitere Hauptkomponenten sind u.a. der Process- und Thread-Manager, der E/A-Manager und der Memory-Manager.

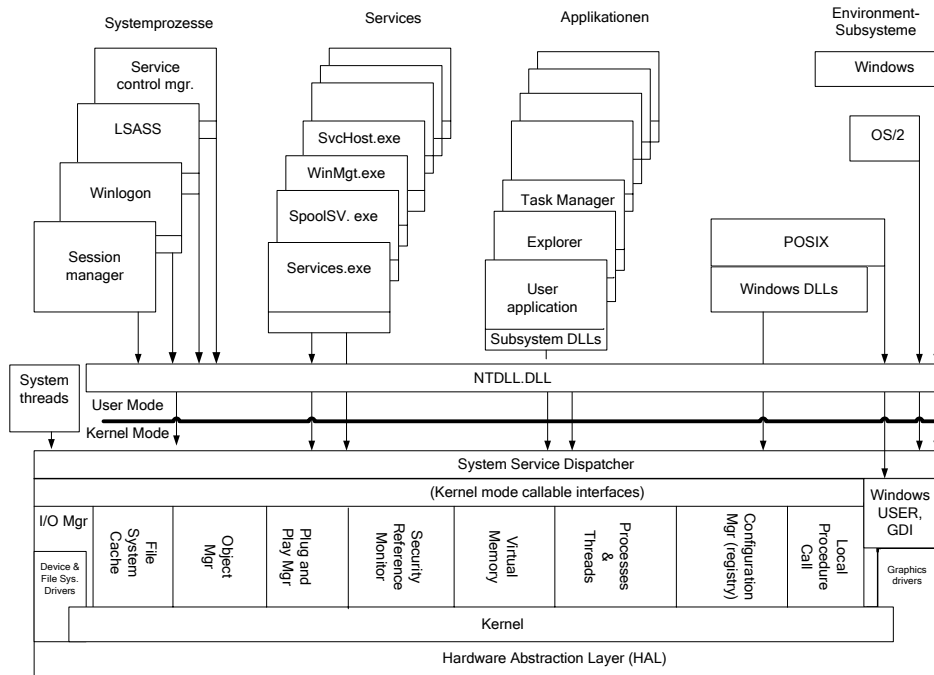


Abbildung 2-8: Architektur von Windows (Solomon 2005)

Einige System-Threads, die nur im Kernelmodus ohne eigenen Adressraum ablaufen, sind in Abbildung 2-8 ebenfalls sichtbar. Diese System-Threads laufen alle in einem speziellen Prozess mit der Bezeichnung *system* ab. Wir werden in Kapitel 4 noch mehr über die Zusammenhänge von Prozessen und Threads erläutern. Im Mikrokern (im Bild als Kernel bezeichnet) sind grundlegende Funktionen wie das Thread-Scheduling und Synchronisationsmechanismen implementiert.

Wie die Architekturskizzen zeigen, sind sowohl moderne Unix- als auch Windows-Derivate geschichtete Betriebssysteme mit einer gekapselten Hardware-schnittstelle. Dieser Aspekt wird auch heute immer wichtiger, da man nicht mehr das ganze Betriebssystem neu entwickeln möchte, wenn man die Hardwareplattform (z.B. durch Unterstützung eines neuen Prozessors) austauscht. Betriebssysteme wie Unix und Windows unterstützen heute eine ganze Fülle von verschiede-

nen Hardwareplattformen. Ohne eine Abstraktionsschicht wie die HAL wäre dies nicht mit vertretbarem Aufwand möglich.

2.2 Betriebsarten

In der Terminologie der Betriebssysteme verwendet man einige Begriffe, die man auch allgemein mit dem Oberbegriff „Betriebsarten“ umschreibt. Die wichtigsten Begriffe (zum Teil sind sie heute nicht mehr so gebräuchlich) sollen in diesem Abschnitt aufgeführt werden. Einen Überblick über die gebräuchliche Terminologie gibt die Abbildung 2-9.

Beim *Benutzerzugang* unterscheidet man den offenen und den geschlossenen Betrieb. Diese Unterscheidung ist heute nicht mehr so üblich. Hiermit meint man den Zugang zum System. Offene Betriebssysteme verfügen über eine offen zugängliche Benutzerschnittstelle. In diese Kategorie fallen alle Mehrzweckbetriebssysteme. Geschlossene Betriebssysteme verfügen nur über eine dedizierte Schnittstelle für ganz spezielle Anwender, z.B. für Administratoren, Operatoren oder Entwickler. Embedded Systems können heute zum Teil als geschlossene Systeme betrachtet werden. Früher waren es die Systeme ohne Online-Zugang, die nur über die Möglichkeit der Stapelverarbeitung verfügten.

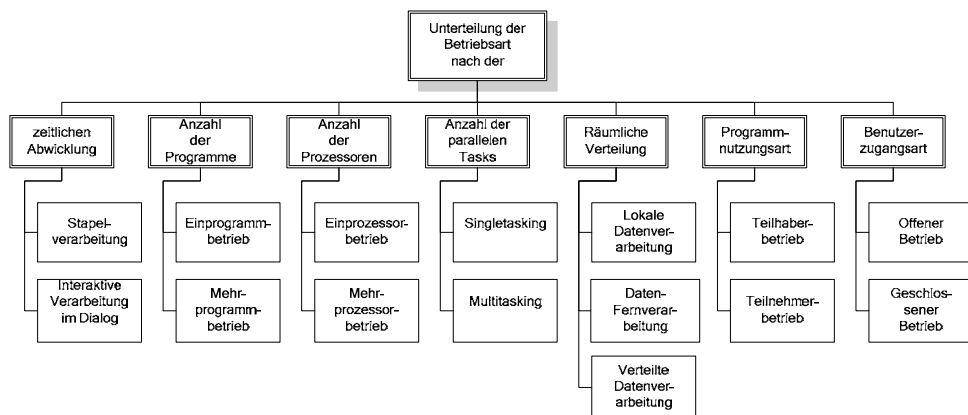


Abbildung 2-9: Betriebsarten, Terminologie

Die Unterscheidung nach der *räumlichen Verteilung* stellt in den Vordergrund, ob ein Betriebssystem an ein Netzwerk angeschlossen werden kann oder nicht. Bei der lokalen Datenverarbeitung wird ein Auftrag eines Anwenders in räumlicher Nähe zum System eingegeben. Von Datenfernverarbeitung spricht man, wenn der Anwender mit dem System über Datenübertragungsleitungen und größere Entfernungen hinweg kommuniziert. Unter verteilter Datenverarbeitung versteht man,

dass Teile eines Programms auf räumlich getrennten Hardwarekomponenten ablaufen. Heutige Mehrzweckbetriebssysteme sind durchweg in der Lage, an einer verteilten Verarbeitung teilzunehmen. Sie sind über Netzwerkschnittstellen an lokale oder entfernte Netzwerke angebunden.

Im Stapelbetrieb, auch Batchprocessing genannt (Unterscheidung nach der zeitlichen Abwicklung), werden die Aufträge (Jobs) an das Betriebssystem zunächst in eine Warteschlange des Betriebssystems eingetragen und dann z.B. unter Berücksichtigung von Prioritäten oder der Reihe nach abgearbeitet. Insbesondere in Mainframe-Anwendungen ist der Stapelbetrieb z.B. für Abrechnungs- oder Buchungsläufe immer noch weit verbreitet. Dagegen wird bei der Dialogverarbeitung eine Kommunikation des Benutzers mit dem System ermöglicht. Der Auftrag an das System muss zum Startzeitpunkt nicht vollständig formuliert sein, sondern wird sozusagen im Dialog ergänzt.

Die Unterscheidungskriterien, die sich mit der Parallelisierung der Verarbeitung (Anzahl der Programme, Prozesse und Tasks) und mit der Nutzungsart (Teilnehmer- und Teilhaberbetrieb) befassen, werden im Weiteren noch ausgeführt. Zudem werden zwei in letzter Zeit immer mehr aufkommende Betriebsarten, der *Application-Server-Betrieb* und der *Terminalserverbetrieb*, kurz erläutert.

2.2.1 Parallelisierung der Verarbeitung

Multiprogramming (*Mehrprogrammbetrieb*, *Multi Processing*) ist ein Begriff, der bereits mehrfach verwendet wurde. Dieser Begriff bezeichnet im Gegensatz zum *Einprogrammbetrieb* (*Single Processing*) die Möglichkeit der „gleichzeitigen“ oder aber auch „quasi-gleichzeitigen“ Ausführung von Programmen in einem Betriebssystem. Unter einem Einprogrammbetrieb versteht man, dass zu einer Zeit nur ein Programm abläuft. Parallelverarbeitung wird in diesem Fall nicht unterstützt. Diese Betriebsart ist heute, außer in einfachen Betriebssystemen (z.B. bei einfachen Embedded Systems), nicht mehr üblich. Die meisten modernen Mehrzweckbetriebssysteme wie *Windows*, *Unix*, *IBM/390*, *BS2000 OSD* usw. unterstützen heute den Mehrprogrammbetrieb, wobei die Anzahl der nebenläufigen Programme meist wesentlich höher als die Anzahl der vorhandenen CPUs ist. Auch mit einer CPU ist ein Mehrprogrammbetrieb möglich und auch üblich (siehe heutige PCs). Wesentliches Merkmal ist hier, dass ein Programm zu einer Zeit auf einem Prozessor, den das Betriebssystem zugewiesen hat, ausgeführt wird.

Einprozessorsysteme verwalten genau einen Prozessor (CPU). Die meisten Mehrzweckbetriebssysteme sind heute *Mehrprozessorsysteme*, die mehrere CPUs oder

sonstige Spezialprozessoren verwalten. Spezialprozessoren sind z.B. E/A-Prozessoren oder auch Grafikkarten und Netzwerkkarten.

Die Begriffe *Singletasking* und *Multitasking* werden ebenfalls häufig verwendet. Die Begriffe sind eng verwandt mit den Begriffen Einprogramm- und Mehrprogrammbetrieb. Im Singletasking-Betrieb ist nur ein Programm aktiv, das sämtliche Betriebsmittel des Systems nutzen kann. Die alten PC-Betriebssysteme wie MS-DOS unterstützen z.B. nur Singletasking. Im Multitasking-Betrieb können mehrere Programme nebenläufig ausgeführt werden. Die erforderlichen Betriebsmittel werden nach verschiedenen Strategien (Prioritäten, Zeitscheibenverfahren) zugeteilt. Die Zuordnung des Prozessors nach Zeitintervallen an die nebenläufigen Programme wird als *Timesharing* bezeichnet.

2.2.2 Teilhaber- versus Teilnehmerbetrieb

Im Hinblick auf die Programmnutzungsart unterscheidet man *Teilhaber-* und *Teilnehmerbetrieb*, wobei der Teilhaberbetrieb in der Regel eine Systemsoftware mit der Bezeichnung *Transaktionsmonitor* benötigt. Die Unterscheidung ist auf den ersten Blick nicht ganz einleuchtend und soll daher erläutert werden.

Die ursprünglich typische Nutzungsart von Online-Systemen, die von Benutzern über eine Dialogschnittstelle verwendet werden konnten, war der Teilnehmerbetrieb. Im Teilnehmerbetrieb erhält jeder Anwender seinen eigenen Benutzerprozess³ sowie weitere Betriebsmittel vom Betriebssystem zugeteilt. Der Benutzer meldet sich über einen Login-Dialog beim System an und bekommt die Betriebsmittel dediziert zugeordnet. Dies ist auch heute noch die übliche Vorgehensweise in modernen Betriebssystemen. Unter Unix und Windows arbeitet man üblicherweise nach dem Login-Vorgang im Teilnehmerbetrieb.

Größere Anwendungen, die viele Benutzer unterstützen, sind aber durch eine dedizierte Betriebsmittelzuweisung nicht besonders leistungsfähig. Wenn z.B. 100 oder mehr Benutzer gleichzeitig ein Buchhaltungssystem verwenden, dann muss das Betriebssystem im Teilnehmerbetrieb jedem Benutzer einen Prozess zuordnen (siehe Abbildung 2-10). Das Programm wird jedes Mal neu gestartet, und das System muss sehr viel Leistung erbringen, obwohl es vielleicht gar nicht sein müsste. Diese klassischen betrieblichen Informationssysteme wie Buchhaltungs-, Auftrags- oder Kundenverwaltungssysteme haben nämlich meistens nur kurze Dialogschritte zu bearbeiten. Die meiste Zeit sitzt der Anwender vor seinem Bildschirm und

³ Ein Prozess stellt im Betriebssystem eine Ablaufumgebung für ein Programm bereit. Wir werden diesen Begriff noch ausführlich erläutern.

denkt. Der zugeordnete Prozess muss sehr wenig tun, belegt aber viele Betriebsmittel. Hinzu kommt, dass derartige Anwendungen in der Regel auch Datenbanken benötigen. Wenn viele Prozesse Verbindungen zu einer Datenbank unterhalten müssen, reduziert dies die Leistungsfähigkeit eines Systems enorm.

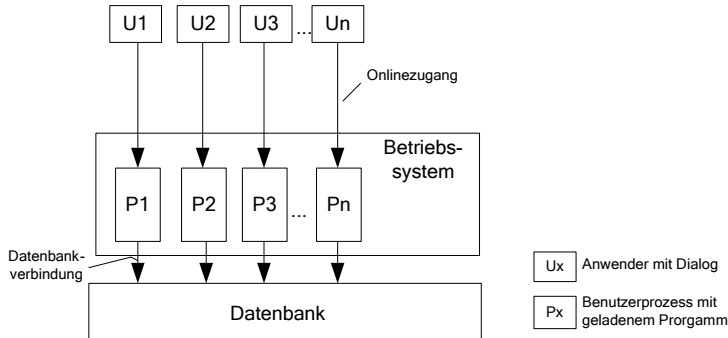


Abbildung 2-10: Teilnehmerbetrieb

Gewissermaßen zur Optimierung dieser Anwendungen wurde der *Teilhhaberbetrieb* erfunden. Im *Teilhhaberbetrieb* werden Prozesse und Betriebsmittel über einen *Transaktionsmonitor* zugeteilt (Abbildung 2-11). Dies ist möglich, da die Benutzer aus Sicht des Betriebssystems die meiste Zeit „denken“, bevor eine Eingabe in das System erfolgt. Die Betriebsart *Teilhhaberbetrieb* ist ideal für dialogorientierte Programme mit vielen parallel arbeitenden Anwendern, die meistens kurze und schnelle Transaktionen ausführen, wie dies etwa bei einem Buchungssystem für Flugbuchungen der Fall ist. Unter einer Transaktion wird hierbei ein atomar auszuführender Service verstanden, der entweder ganz oder gar nicht ausgeführt wird. Werden innerhalb der Ausführung einer Transaktion mehrere Operationen etwa auf einer Datenbank erforderlich, so darf dies nur in einem Stück erfolgen⁴. Das Transaktionskonzept ist für betriebliche Informationssysteme ein sehr wichtiges Konzept, das bei der Entwicklung fehlertoleranter Anwendungssysteme unterstützt.

Ein Transaktionsmonitor ist ein sehr komplexes Dienstprogramm, das oberhalb des Kernels angesiedelt ist und viele Aufgaben wie die Zugangskontrolle, die Verteilung der Anfragen auf bereitgestellte Prozesse und die Optimierung der Zugriffe sowie die Verwaltung der Ressourcen und auch die Zuordnung von Daten-

⁴ Klassische Transaktionen müssen verschiedene Korrektheitskriterien erfüllen, die auch mit dem Akronym ACID (Atomarity, Consistency, Isolation und Durability) umschrieben werden.

bankverbindungen übernimmt. Transaktionsmonitore kommen ursprünglich aus der Mainframe-Welt, sind aber heute auch in Unix- und Windows-Umgebungen stark verbreitet.⁵ Bei der Programmierung einer Transaktionsanwendung geht man etwas anders vor als bei der klassischen Programmierung. Man hängt in der Regel seine Programme, auch Transaktionsprogramme genannt, in den Transaktionsmonitor ein, der die eigentliche Steuerung übernimmt. Transaktionsmonitore stellen also die Ablaufumgebung für die Transaktionsanwendungen bereit. Der Aufruf eines Transaktionsprogramms erfolgt über die Angabe eines Transaktionscodes (TAC), der die Programmteile adressiert.

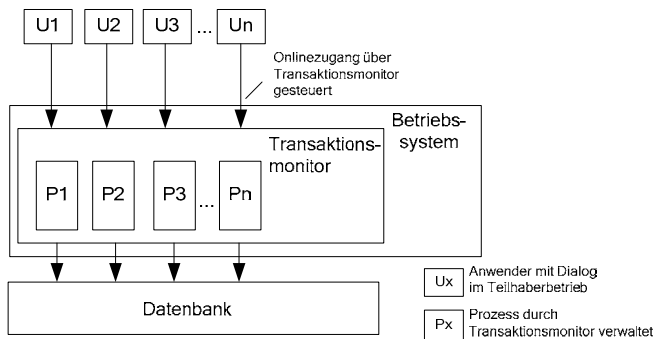


Abbildung 2-11: Teilhaberbetrieb

In Abbildung 2-12 ist ein vereinfachter Ablauf des Aufrufs eines Transaktionsprogramms dargestellt. Das Transaktionsprogramm läuft in der Umgebung des Transaktionsmonitors. Über einen TAC erfolgt die Adressierung. Ein Dispatcher (Verteiler) teilt benötigte Ressourcen wie z.B. einen Prozess zu und gibt den Aufruf samt seiner Input-Daten an das Transaktionsprogramm weiter. Meist werden Daten aus Datenbanken gelesen, verarbeitet und ggf. verändert. Nach der Bearbeitung werden die Ergebnisse für die Ausgabe auf dem Terminal aufbereitet und an das Terminal gesendet.

⁵ Beispiele für Transaktionsmonitore sind IBM CICS, IBM IMS, Siemens UTM. Tuxedo und Encina.

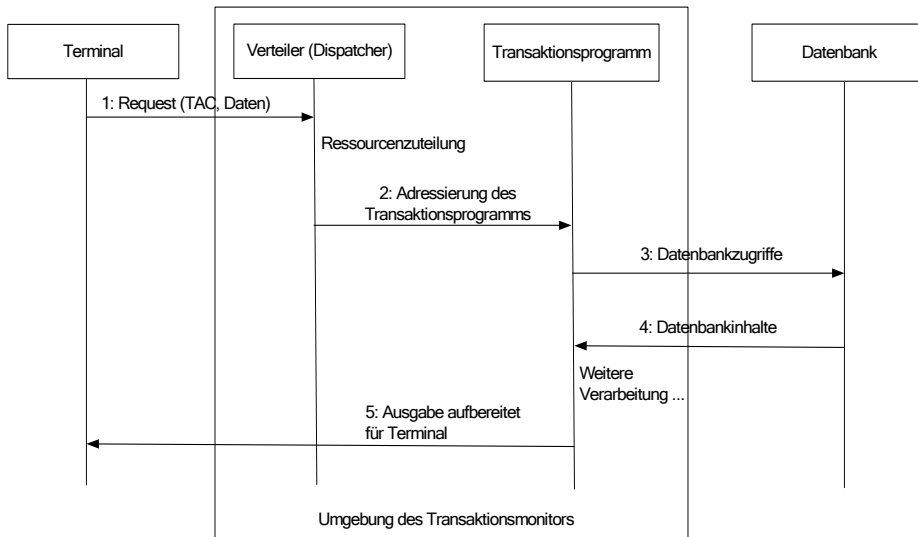


Abbildung 2-12: Ablauf beim Aufruf einer Transaktion im Teilhaberbetrieb

2.2.3 Application-Server-Betrieb

Neuere Entwicklungen im Zuge verteilter Objektsysteme und verteilter Komponentensysteme haben sog. Application-Server als Middleware hervorgebracht, die im Wesentlichen eine Weiterentwicklung der klassischen Transaktionsmonitore darstellen. Während Transaktionsmonitore für die Mainframe-Welt entwickelt wurden, sind Application-Server für verteilte Anwendungssysteme konzipiert. Transaktionsmonitore unterstützten im Wesentlichen eine prozedurale Entwicklung von Transaktionsprogrammen. Application-Server nutzen die Konzepte der Objekt- und vor allem der Komponentenorientierung. Anwendungen werden in sog. Komponenten zerlegt, die über Schnittstellen nach außen verfügen sollen. Application-Server unterstützen vorwiegend das Request-Response-Modell bzw. das Client-/Server-Konzept, aber auch andere Kommunikationsmodelle wie Message-Passing werden ermöglicht. Clientanwendungen nutzen die Dienste der Serverkomponenten, die in der Ablaufumgebung des Application-Servers ablaufen.

In modernen Application-Servern werden für den Anwendungsprogrammierer bestimmte Schnittstellen bereitgestellt, die man nutzen muss, um ein Programm in der Umgebung des Application-Servers ablaufen zu lassen. Typische Standards in diesem Umfeld sind der *Enterprise-Java-Bean-Standard* (EJB) in der Java-Welt und die *.NET-Enterprise-Services* in der Microsoft-Welt. Anwendungsprogramme werden als sog. Komponenten in die Ablaufumgebung des Application-Servers (auch als *Container* bezeichnet) eingebettet. Eine Komponente muss bestimmte Schnitt-

stellen bereitstellen, die der Container für den Aufruf verwendet. Diese Schnittstellen werden dann über eine Netzwerkverbindung von den Clients aufgerufen. Dazu werden Nachrichten über standardisierte Protokolle übertragen. Die Ergebnisse werden an den Client zurückgesendet.

Das Grundprinzip der Arbeitsweise eines Application-Servers ist in Abbildung 2-13 dargestellt. Die Clientanwendungen laufen – im Gegensatz zum klassischen Teilhaber- und auch Teilnehmerbetrieb – auf eigenen Rechnern mit eigenen Betriebssystemen ab. Auf den Clientrechnern muss eine entsprechende Software ablaufen, die den Zugang zum Application-Server ermöglicht. Im Serverrechner läuft der Application-Server ab. In diesem liegen – ähnlich wie Transaktionsprogramme – die Softwarekomponenten der Anwendungen. Zum Ablaufzeitpunkt werden diesen Komponenten nach Bedarf Prozesse oder Threads⁶ zugeordnet, in denen der Programmcode der Komponenten zum Ablauf kommt.

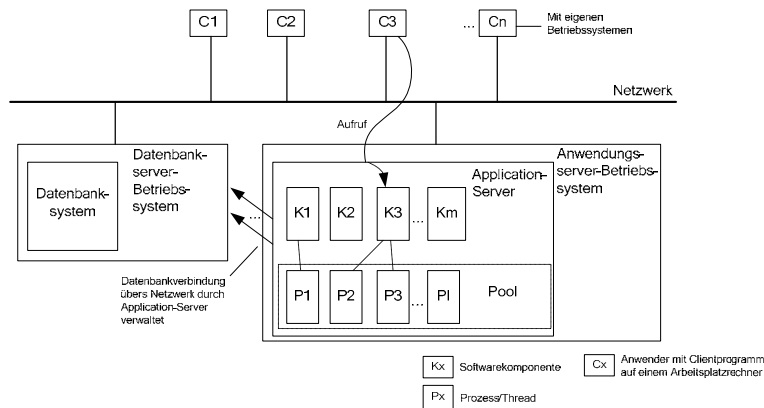


Abbildung 2-13: Application-Server-Betrieb

Der Application-Server verwaltet üblicherweise einen Pool an Prozessen bzw. Threads und ordnet diese dynamisch den Komponenten zu. Das Datenbanksystem liegt meist – aber nicht zwingend – auf einem eigenen Serverrechner. Die Zugriffe auf die Datenbank laufen über das Netzwerk ab. Sie werden über den Application-Server kontrolliert. Application-Server leisten, wie Transaktionsmonitore, umfangreiche Dienste. Sie verwalten die Komponenten, ordnen diese zur Laufzeit Prozessen zu, optimieren die Datenbankzugriffe und erleichtern die Programmierung

⁶ Ein Thread ist ein leichtgewichtiger Prozess (mehr dazu in Kapitel 2).

durch vordefinierte Standardschnittstellen.⁷ Der Ablauf eines Requests ähnelt dem Aufruf eines Teilprogramms im Teilhaberbetrieb, nur dass der Initiator kein Terminal, sondern meist ein Clientrechner ist.

2.2.4 Terminalserver-Betrieb

Ein Terminalserver verwaltet Terminals bzw. Client-Arbeitsplätze. Sinn und Zweck von Terminalserver-Systemen ist es, die Administration verteilter Komponenten zu vereinfachen und besser kontrollieren zu können. Eine zentrale Serverlandschaft, die aus großen Serverfarmen bestehen kann, bedient „dumme“ Clientrechner (sog. Thin Clients), wobei die Anwendungsprogramme vollständig in den Servern ablaufen und die Clientrechner nur noch für Zwecke der Präsentation eingesetzt werden. Die Idee hinter Terminaldiensten ist die Zentralisierung von Betriebsmitteln, um die beteiligten Systeme leichter administrieren zu können. Dies kann als Gegenbewegung zur starken Verteilung der Systeme gesehen werden. Eine leistungsfähige „Terminalserverfarm“ stellt Ressourcen wie Rechenleistung, Hauptspeicher, Plattenspeicher usw. bereit. Der Anwender arbeitet mit der gewohnten Benutzeroberfläche, die Anwendungen laufen aber komplett im Server ab, ohne dass dies der Benutzer bemerken soll. Lediglich Bildschirmänderungen werden im Client dargestellt. Ein Server bedient je nach Leistungsfähigkeit mehrere oder viele Benutzer.

In Abbildung 2-14 ist ein typischer Einsatz von Terminalserver-Software skizziert. Auf den Servern, die in einer sog. Serverfarm organisiert sind, werden die Anwendungen installiert. Über die Terminalserver-Software wird der Zugang zu den Anwendungen geregelt. Auf der Clientseite ist ebenfalls spezielle Software, die hier als Terminalserver-Client bezeichnet wird, erforderlich. TS-Client und TS-Server kommunizieren miteinander über spezielle Kommunikationsmechanismen (z.B. das Remote Desktop Protocol (RDP) von Microsoft) zum Austausch der Informationen, die an der Oberfläche des Clientrechners präsentiert werden sollen. Jede Eingabe an der Oberfläche wird an den Terminalserver (bzw. die Serverfarm) gesendet und von dort bei Bedarf, also wenn sich die GUI verändert, für die Darstellung zurück zum Client transportiert.

⁷ Bekannte Application-Server-Produkte aus dem Java-Umfeld sind u.a. *Oracle IAS*, *Oracle BEA Weblogic AS*, *IBM Websphere AS* und *JBoss AS*.

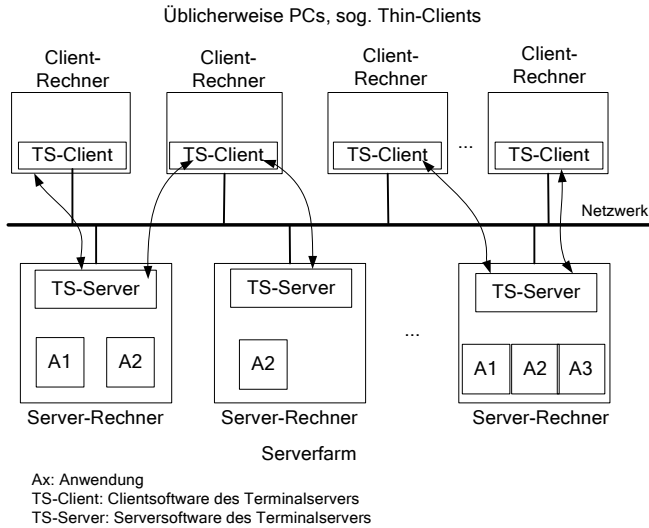


Abbildung 2-14: Typischer Terminalservereinsatz

Terminalserver leisten heute darüber hinaus eine Fülle von Diensten wie z.B. die Verwaltung von Lizenzen, die zentrale Benutzerverwaltung, gewisse Sicherheitsdienste, Dienste zur Softwareverteilung und zur Systemkonfiguration. Sie zeichnen sich vor allem durch eine einfache und zentralisierte Administration aus. Die Softwareverteilung spielt sich z.B. nur im Serverumfeld ab.⁸

2.2.5 Handheld-Computing

Im Bereich der mobilen Kommunikation hat sich in den letzten Jahren das Thema Handheld-Computing stark entwickelt. Mittlerweile existieren PDAs (Personal Digital Assistant), Mobile Phones (Handy) oder sog. Smartphones (Mischung aus PDA und Handy), für die eigene Betriebssysteme entwickelt wurden. Bekannte Betriebssysteme in diesem Umfeld sind *Symbian OS*, *Palm OS* und *Windows CE* bzw. *Windows Mobile*.

Für mobile Betriebssysteme dieser Kategorie gibt es eine Reihe spezieller Anforderungen. Zum einen sind die Geräte sehr klein und verfügen daher auch nur über kleine Displays. Zum anderen ist die Stromversorgung limitiert und zudem sind Ressourcen wie Hauptspeicher und externe Speichersysteme noch recht knapp.

⁸ Typische Terminalserver-Produkte sind die unter Windows standardmäßig vorhandenen Windows-Terminaldienste (www.microsoft.com) und Citrix MetaFrame bzw. XenServer von der Firma Citrix (www.citrix.com).

Grundlegende Konzepte der Betriebsmittelverwaltung werden auch in diesen Betriebssystemen benötigt. Es soll allerdings im Rahmen dieses Buches nicht weiter auf diese Betriebssysteme eingegangen werden. Wer Interesse an diesen Betriebssystemen hat, der findet im Literaturverzeichnis einige weiterführende Werke.

2.3 Übungsaufgaben

1. Was versteht man unter einem Mikrokern?
2. Nennen Sie ein Beispiel für ein Betriebssystem mit Mikrokern-Architektur!
3. Was versteht man unter einem „echt“ verteilten Betriebssystem im Gegensatz zu den heutigen Client-/Server-Systemen?
4. Was versteht man unter Kommunikations-Middleware?
5. Nennen Sie vier Betriebsmittel, welche das Betriebssystem verwaltet! Welche davon sind hardware- und welche software-technische Betriebsmittel?
6. Erläutern Sie den Unterschied zwischen Teilnehmer- und Teilhaberbetrieb! In welcher Betriebsart wird üblicherweise ein Transaktionsmonitor eingesetzt?
7. Welche Aufgaben erfüllt ein Transaktionsmonitor, welche ein Application-Server?
8. Nennen Sie einen Vorteil der Schichtenarchitektur bei Betriebssystemen!
9. Was versteht man unter Multiprogramming?
10. Benötigt man für Multiprogramming mehrere CPUs?
11. Verfügen die Betriebssysteme Unix und Windows 2000/2003/XP über Mikrokern-Architekturen und wenn nein, wie sind sie konzipiert?
12. Was bezeichnet man als Timesharing?
13. Wozu verwendet man Embedded Systems? Nennen Sie ein Beispiel!
14. Wozu dient ein Terminalserver?

3 Interruptverarbeitung

Moderne Betriebssysteme verfügen über mehrere externe Geräte, die zeitnah bedient werden müssen. Wenn ein Gerät Signale oder Daten an die CPU übertragen möchte, erzeugt es zunächst eine Unterbrechungsanforderung, die der richtigen Bearbeitungsroutine im Betriebssystem übergeben werden muss. Dazu müssen aktuell ablaufende Aktivitäten ggf. unterbrochen werden. Nach der Abarbeitung muss wieder der alte Zustand hergestellt werden. Ähnliches geschieht bei der Bearbeitung von Systemdiensten, wie etwa dem Lesen einer Datei. Systemdienste werden über sog. Systemcalls durch die Programme aktiv initiiert. Man spricht hier allgemein von Unterbrechungsanforderungen (Interrupt-Anforderung oder Interrupt-Request) und der zugehörigen Unterbrechungsbearbeitung (Interrupt-Bearbeitung).

Dieses Kapitel befasst sich mit den Aktivitäten bei der Unterbrechungsbearbeitung. Zunächst werden die verschiedenen Interrupt-Typen klassifiziert. Anschließend wird die Interrupt-Bearbeitung erläutert und anhand einiger konkreter Beispiele untersucht. Betrachtet werden die von der Hardware bereitgestellten Mechanismen der Interrupt-Verarbeitung insbesondere bei Intel-Prozessoren. Die Implementierungen der Interrupt-Verarbeitung unter Windows und Linux sollen darstellen, wie diese Aufgabe in konkrete Betriebssysteme eingebettet ist. Schließlich wird noch der Ablauf eines Systemcalls (Software-Interrupt), also einer aktiv von einem Programm angeforderten Unterbrechung aufgezeigt und an einem konkreten Codebeispiel demonstriert.

Zielsetzung des Kapitels

Der Studierende soll verstehen und erläutern können, was ein Interrupt ist, welche Arten von Interrupts es gibt und wie die Interruptbearbeitung durch ein Betriebssystem unterstützt wird. Weiterhin soll er die Abarbeitung eines Systemcalls nachvollziehen können.

Wichtige Begriffe

Synchroner und asynchroner Interrupt, Unterbrechungsanforderung, Interrupt-Request, IRQ, Interrupt Service Routine (ISR), Interrupt-Controller, Interrupt-Sharing, Interrupt-Vektor-Tabelle, Prozess-Kontext, Trap, Systemcall, Software-Interrupt.

3.1 Interrupts

3.1.1 Überblick

Bei der Bearbeitung unvorhergesehener Ereignisse unterscheidet man zwei Verfahren, Polling und Interrupts. Unter *Polling* versteht man das zyklische Abfragen von einer Ereignisquelle bzw. mehreren Ereignisquellen (z.B. E/A-Geräte), um deren Kommunikationsbereitschaft festzustellen bzw. um anliegende Ereignisse oder Kommunikationswünsche der Ereignisquelle abzufragen. Polling hat den Nachteil, dass die CPU ständig arbeiten muss und damit die Effizienz eines Systems beeinträchtigt ist. Die meiste Zeit wird umsonst nachgefragt. Allerdings ist das Verfahren relativ leicht zu implementieren.

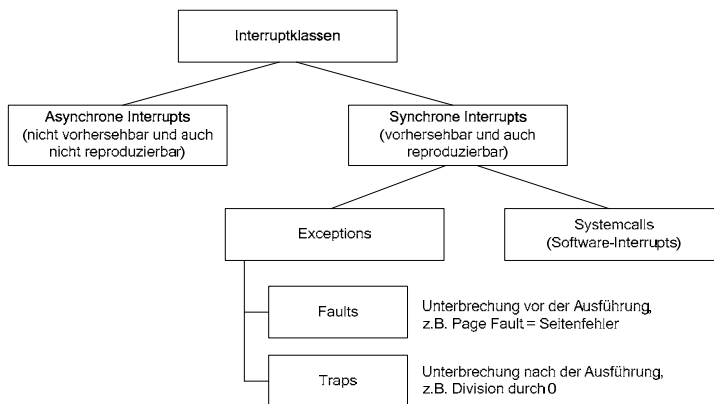


Abbildung 3-1: Interrupt-Klassifizierung

Im Gegensatz zu Polling sind *Interrupts* (Unterbrechungen) sog. Betriebssystembedingungen oder auch asynchrone Ereignisse, die den Prozessor veranlassen, einen vordefinierten Code auszuführen, der außerhalb des normalen Programmflusses liegt (Solomon 2005). Überwachte Ereignisquellen müssen nicht ständig abgefragt werden, sondern die Ereignisquellen melden sich beim Auftreten eines Ereignisses, das behandelt werden muss. Interrupts können grundsätzlich durch *Hardware* oder *Software* (Software-Interrupt, Systemcall) verursacht werden. Software-Interrupts werden auch meist als *Traps* oder *Faults* bezeichnet. Man unterscheidet auch *synchrone* und *asynchrone* Interrupts. Eine Klassifizierung von Interrupts ist in Abbildung 3-1 gegeben.

Synchrone Interrupts. Synchrone Interrupts treten bei synchronen Ereignissen auf. Dies sind Ereignisse, die bei identischen Randbedingungen (Programmausführungen mit gleichen Daten) immer an der gleichen Programmstelle auftreten. Syn-

chrone Interrupts werden auch als *Exceptions* (*Ausnahmen*) bezeichnet. Ein weiterer Typ eines synchronen Ereignisses sind Systemcalls. Synchrone Interrupts sind vorhersehbar und auch bei gleicher Konstellation wiederholbar.

Ausnahmen werden von der CPU selbst ausgelöst und sind für das laufende Programm bestimmt. Eine typische Ausnahme ist die Division-Durch-Null-Ausnahme, die auch oft als Trap bezeichnet wird. Traps werden vom Betriebssystem erst nach der Ausführung erkannt und an das Anwendungsprogramm gemeldet. In der Speicherverwaltung spricht man – wie wir noch sehen werden – z.B. von einem Seitenfehler (Page Fault), wenn ein benötigter Speicherbereich nicht im Hauptspeicher vorhanden ist oder ein nicht zugewiesener Speicherbereich adressiert wird. Dies ist auch ein Ereignis, das einen synchronen Interrupt auslöst. Bei dieser Art von Interrupt spricht man von einem *Fault*. Faults werden vom Betriebssystem vor der eigentlichen Ausführung abgefangen und gemeldet.

Beide Typen können aufgrund eines Softwarefehlers verursacht werden. Dies sind Ausnahmesituationen, die vom Prozessor alleine nicht gelöst werden können. Diese Ausnahmen müssen an das Anwendungsprogramm oder an das Betriebssystem gemeldet werden, das (hoffentlich) eine entsprechende Bearbeitungsroutine zu deren Behandlung bereitstellt.

Asynchrone Interrupts. Asynchrone Interrupts sind die klassischen Interrupt-Typen, die nicht an ein laufendes Programm gebunden sind. Sie treten unabhängig davon auf, was das System gerade macht. Beispiele für asynchrone Interrupts sind die Ankunft einer Nachricht an einem Netzwerkadapter oder die Zustellung eines Plattenspeicherblocks an die CPU. Beide Ereignisse unterbrechen in der Regel für kurze Zeit den Ablauf des laufenden Programms. Asynchrone Interrupts sind nicht vorhersehbar und können auch nicht ohne weiteres reproduziert werden.

3.1.2 Interrupt-Bearbeitung

Bei der Interrupt-Bearbeitung wird die Steuerung an eine definierte Position im Kernel übergeben. Es wird also veranlasst, dass – sofern noch nicht geschehen – vom Benutzermodus in den Kernelmodus gewechselt wird, um den Interrupt zu bearbeiten.

Maskierung. Die Maskierung, also das Ein- oder Ausschalten von Interrupts bestimmter Geräte, kann über ein *Interrupt-Maskenregister* bzw. *Maskenregister* erfolgen (Interrupt Mask Register, IMR). Für jede Interrupt-Quelle wird in dem Register ein Maskierungsbit verwaltet. Wird das Bit auf 1 gesetzt, ist der Interrupt ausgeschaltet. Wenn ein Interrupt verhinderbar ist, so spricht man auch von einem *maskierbaren* Interrupt. E/A-Geräte benutzen typischerweise maskierbare Interrupts.

Meist gibt es eine Steuerleitung, die davon ausgeschlossen wird. Diese nennt man NMI (Non Maskable Interrupt). Tritt ein NMI ein, liegt eine schwerwiegende Ausnahmesituation vor und es kommt zu einer Ausnahmebehandlung.

Die meisten asynchronen Interrupts können maskiert werden. Dies wird zwar im Kernel möglichst vermieden, aber gelegentlich ist es unbedingt erforderlich. Ein Grund ist beispielsweise, um während der Bearbeitung eines Interrupts zu vermeiden, dass eine Unterbrechung durch einen anderen Interrupt erfolgt. Dieses Abschalten von Interrupts sollte aber recht kurzzeitig sein, um die Systemleistung nicht zu beeinträchtigen und damit keine Interrupts verloren gehen. Nicht maskierbare Interrupts werden meist nur zum Signalisieren sehr kritischer Ereignisse, z.B. Speicherparitätsfehlern, benutzt.

Interrupt-Service-Routine (ISR). Das Programmstück, das einen Interrupt bearbeitet, wird als *Interrupt-Service-Routine (ISR, Interrupt-Bearbeitungsroutine)* bezeichnet. In einem Treiberprogramm zur Steuerung eines externen Geräts ist zum Beispiel auch in der Regel eine ISR enthalten. Für jeden Interrupt-Typen gibt es eine ISR. Eine ISR kann aber auch mehreren Interrupt-Typen zugeordnet werden. Das Betriebssystem stellt für alle Interrupts eine passende ISR zur Verfügung. Ein Interrupt ist also ein zur Programmausführung synchrones oder asynchrones Ereignis, das die sequentielle Programmausführung unterbricht und die Kontrolle an ein speziell dafür vorgesehenes Programm, die Interrupt-Service-Routine, übergibt.

Interrupt-Request-Bearbeitung (IRQ-Bearbeitung). Hardwarebedingte Interrupts können in der Regel nicht direkt vom auslösenden Gerät an die CPU signalisiert werden. Ereignisse der Geräte, wie zum Beispiel die Ausführung einer Positionierungsanweisung an den Festplatten-Controller, werden daher zunächst an einen *Interrupt-Controller*¹ (vgl. Abbildung 3-2) wie etwa den Intel 8259A PIC-Baustein gemeldet. Dies geschieht in Form eines sog. *Interrupt-Request* oder *IRQ* (Interruptanforderung). Der Interrupt-Controller erzeugt dann aus einem IRQ eine Unterbrechung der CPU, die mit Hilfe eines passenden Programmstücks (einer ISR) bearbeitet werden muss. Im Interrupt-Controller liegt auch das oben erwähnte Interrupt-Maskenregister. Der 8289A PIC (Programmable Interrupt Controller) ist ein programmierbarer Interrupt Controller, der häufig in X86-Systemen vorzufinden ist.

¹ Nicht alle Rechnersysteme verfügen über einen Interrupt-Controller. In diesem Fall erzeugt jedes einzelne Gerät direkt Interrupts bei der CPU.

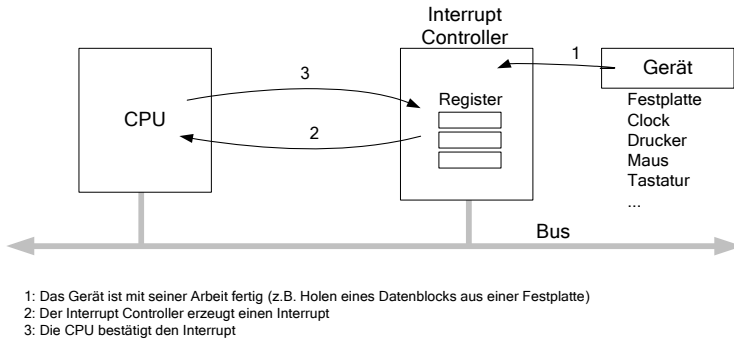


Abbildung 3-2: Zusammenspiel der CPU mit Interrupt-Controller (Tanenbaum 2009)

Interrupt-Vektor-Tabelle. ISRs werden in heutigen Systemen meist nicht direkt, sondern über einen *Interrupt-Vektor* adressiert. Die Interrupt-Vektoren sind meist in einer Interrupt-Vektor-Tabelle gespeichert. Ein Interrupt-Vektor ist also ein Eintrag in dieser Tabelle, der die Speicheradresse der Interrupt-Service-Routine enthält. Die Tabelle liegt an einer vordefinierten Stelle im Kernelspeicher. Der Index zur Adressierung in der Tabelle wird der CPU bei Auftreten einer Unterbrechung implizit durch den Interrupt-Controller anhand der belegten Adressleitungen übermittelt. Jeder Interrupt-Quelle wird ein fester Index auf die Interrupt-Vektor-Tabelle zugeordnet.

Interrupt-Level. Jeder Interrupt hat ein vorgegebenes *Interrupt-Level*, also eine Priorität, welche das Betriebssystem verwaltet. Diese Priorität legt fest, wie der Interrupt in die Gesamtverarbeitung eingebaut werden soll. Bei exakt gleichzeitigem Auftreten mehrerer Interrupt-Anforderungen ist über die Priorität genau festgelegt, in welcher Reihenfolge diese bearbeitet werden sollen. Tritt während einer Interrupt-Bearbeitung noch einmal ein Interrupt auf, so entscheidet die Priorität, ob er ausgeführt werden soll oder nicht. Eine ankommende Interrupt-Anforderung mit niedrigerer oder gleicher Priorität als die gerade laufende Interrupt-Bearbeitung wird nicht angenommen.

Erkennung von Unterbrechungsanforderungen. Das Betriebssystem muss anstehende Interrupts bearbeiten. Wie erkennt aber die CPU eine Unterbrechungsanforderung? Die Prüfung, ob eine Unterbrechung ansteht, ist Teil des Befehlszyklus, wie dies in Abbildung 3-3 vereinfacht skizziert ist. Nach Ausführung eines Ma-

schinenbefehls wird überprüft, ob ein Interrupt-Request anliegt.² Ist dies der Fall, wird in ein spezielles Unterprogramm, die oben genannte ISR oder bei größeren Betriebssystemen in eine entsprechend davor geschaltete Verteilungsroutine, verzweigt.

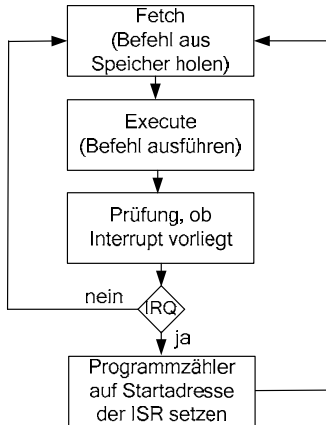


Abbildung 3-3: Prüfung, ob ein Interrupt vorliegt (vereinfachte Befehlsausführung)

Die Bearbeitung eines Interrupts kann sofort oder evtl. erst nach Abarbeitung wichtigerer Arbeiten erfolgen. Wie bereits erwähnt, können Interrupts vom Betriebssystem auch ausgeblendet (maskiert) werden, wenn gerade ein Codestück durchlaufen wird, das gar nicht unterbrochen werden darf. In diesem Fall muss der Interrupt-Controller den Interrupt ggf. so lange wiederholen, bis er angenommen wird. In den meisten Fällen sind Interrupts ebenfalls unterbrechbar.

Nach der Bearbeitung des Interrupts bestätigt die ISR dem Interrupt-Controller die Bearbeitung, damit dieser wieder auf den nächsten Interrupt warten kann. In Abbildung 3-4 ist die Bearbeitung eines Interrupts grob skizziert.

Der Ablauf sieht wie folgt aus:

- Ein Interrupt unterbricht zunächst (sofern der Interrupt-Level dies zulässt) das aktuell laufende Programm, das in unserem Fall ein Anwendungsprogramm eines beliebigen Prozesses ist (1).
- Der aktuelle Prozessorstatus des laufenden Programms wird am Anfang der Interrupt-Bearbeitung gerettet (2)

² Tatsächlich gibt es auch die Möglichkeit, einen Maschinenbefehl zu unterbrechen. Es ist allerdings viel aufwändiger, auch noch den Maschinenbefehlszustand zu sichern und anschließend wieder herzustellen.

- Die Interrupt-Bearbeitungsroutine wird in der Interrupt-Vektor-Tabelle gesucht und anschließend ausgeführt (3)
- Am Ende der Bearbeitung sendet die ISR an den Interrupt-Controller eine Bestätigung (4)
- Anschließend wird der alte Prozessorstatus wieder hergestellt, und es wird an der vorher unterbrochenen Stelle weitergearbeitet (5).

Letzteres kann natürlich durch eine entsprechende Scheduling-Entscheidung der Prozessverwaltung (mehr hierzu in Kapitel 5) etwas anders aussehen. Ein Prozess mit höherer Priorität kann z.B. die CPU erhalten, bevor der unterbrochene Prozess wieder an der Reihe ist. Dies hängt von der Scheduling-Strategie des Betriebssystems ab. Es kann auch sein, dass nach einem schwerwiegendem Interrupt (z.B. einem Spannungsausfall) kein weiteres Programm mehr zum Ablauf kommt.

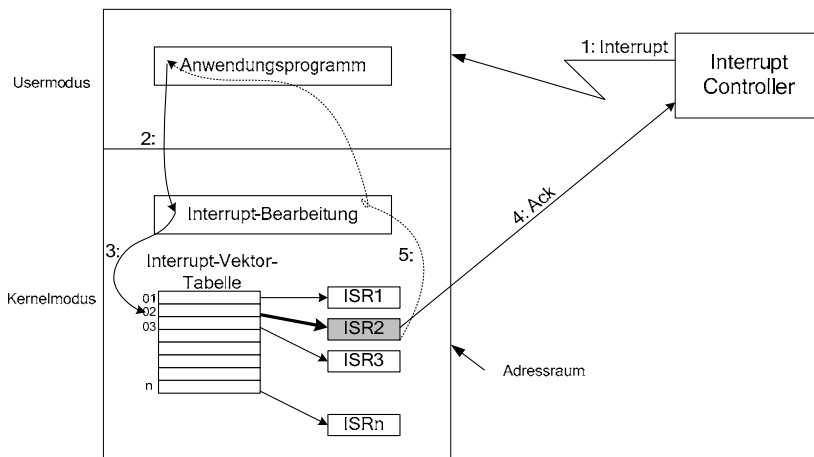


Abbildung 3-4: Interrupt-Bearbeitung

In Abbildung 3-5 ist nochmals der zeitliche Ablauf für die Bearbeitung eines Interrupts dargestellt.

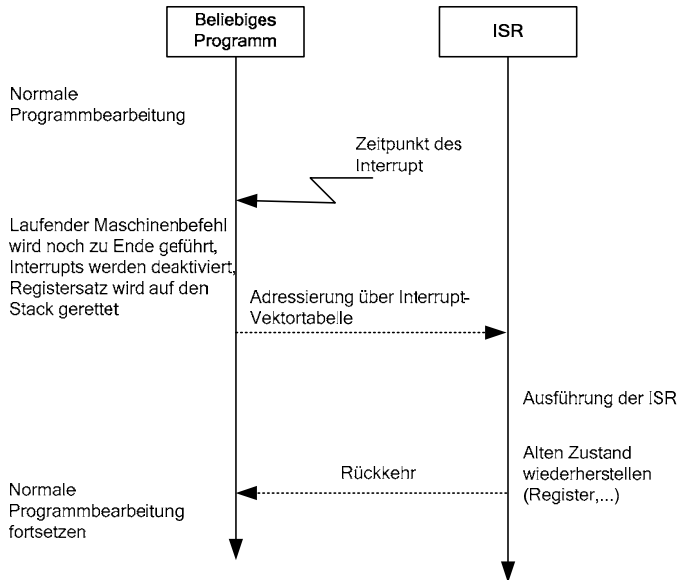


Abbildung 3-5: Zeitlicher Ablauf einer Unterbrechung

Es soll noch erwähnt werden, dass die Interrupt-Bearbeitung für Hardware-Interrupts sehr stark von der Hardware abhängt. Je nach Hardwareplattform gibt es hier zum Teil beträchtliche Unterschiede.

Wenn ein Betriebssystem verschiedene Hardwareplattformen unterstützt, ist der Teil zur Bearbeitung von Interrupts meist sehr spezifisch, meist auch in Assembler programmiert und in der Regel gut gekapselt.

3.1.3 Interrupt-Verarbeitung bei IA32-Prozessoren

Die Interrupt-Verarbeitung in PCs mit Intel IA32-Architektur wird mit Hilfe des Interrupt-Controllers Intel 8259 PIC3 (siehe Abbildung 3-6) und deren Folgeversionen (z.B. Intel 8259 APIC4 für Multiprozessorsysteme) unterstützt. Heutige PCs verfügen oft über zwei PIC-Bausteine, frühere Systeme, wie beispielsweise die

³ PIC steht für Programmable Interrupt Controller. Der PIC-Standard stammt vom Original-IBM-PC.

⁴ APIC steht für Advanced Programmable Interrupt Controller. Intel und weitere Firmen haben für den APIC die Multiprozessor-Spezifikation definiert.

alten PC-XT-Systeme, mussten mit einem PIC auskommen. Heute ist der PIC-Baustein in die CPU integriert⁵.

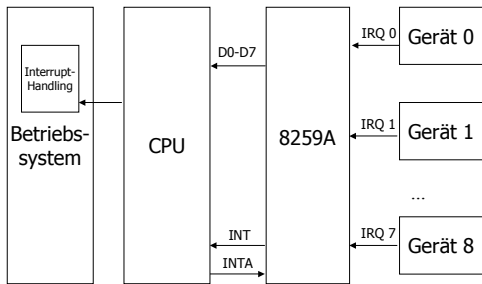


Abbildung 3-6: Baustein Intel 8259 PIC im Kontext eines Computersystems

Ein PIC verfügt über acht Interrupt-Eingänge IRQ0 bis IRQ7. Über die Signale INT und INTA (Interrupt Acknowledge) und die Datenleitungen D0 bis D7 erfolgt die Kommunikation mit der CPU. Der Interrupt-Request-Nummer wird über die Datenleitungen D0 bis D7 an die CPU übertragen. Die CPU übermittelt die Unterbrechungsanforderungen seinerseits an den Interrupt-Handler des Betriebssystems.

Das Innenleben eines PIC 8259A ist in Abbildung 3-7 vereinfacht dargestellt. Man sieht im Bild mehrere Register. Das *Interrupt-Request-Register* speichert die Unterbrechungsanforderungen der angeschlossenen Geräte. Über das *Interrupt-Masken-Register* lassen sich Interrupts ausblenden (maskieren). Im *In-Service-Register* werden die gerade bearbeiteten Unterbrechungsanforderungen gespeichert. Über den Datenbus wird die Interrupt-Request-Nummer an die CPU gesendet.

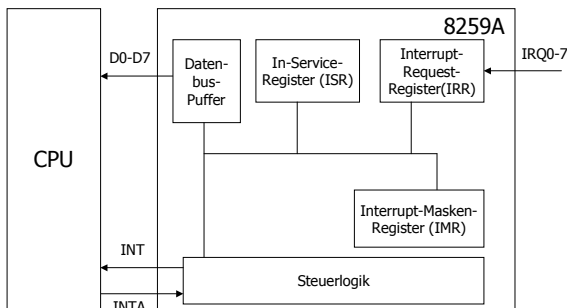


Abbildung 3-7: Innenleben eines Intel 8259 PIC

⁵ In der sog. Southbridge.

3 Interruptverarbeitung

PICs sind aber auch kaskadierbar, so dass mehr als acht nummerierte Interrupt-Eingänge unterstützt werden können (siehe Abbildung 3-8). Über den IRQ-2 wird eine Slave-PIC an eine Master-PIC angeschlossen. Von den externen Geräten führen elektronische Leitungen zum Interrupt-Controller. Die Belegung der Eingänge der Interrupt-Controller muss genau festgelegt werden. Über diese Leitungen werden die Interrupt-Requests zunächst an den Interrupt-Controller und von dort an die CPU weitergeleitet.

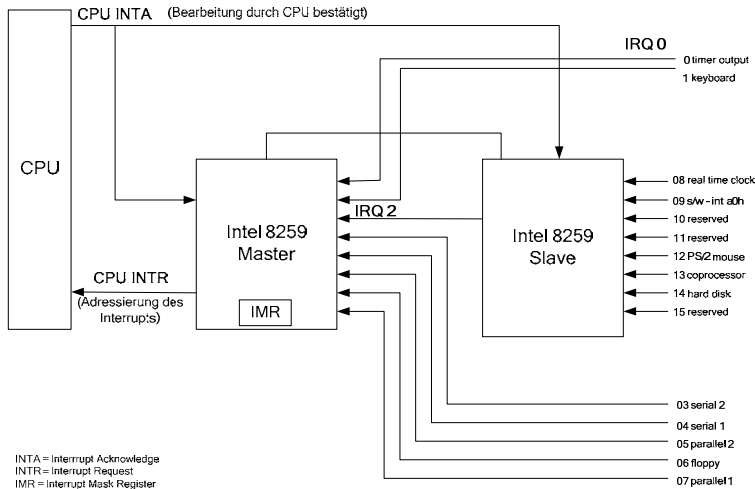


Abbildung 3-8: Baustein Intel 8259 PIC (kaskadiert)

In Intel-basierten PC-Systemen gibt es eine Interrupt-Vektor-Tabelle mit 256 Interrupt-Vektoren. Diese Tabelle wird über eine Interrupt-Nummer adressiert. Die Adressierung erfolgt bei Software-Interrupts mit dem Befehl *int*, speziell dem Befehl *int 0x21*⁶. In IA32-Prozessoren sind für Hardware-Interrupts meist nur 16 Interrupt-Nummern im Bereich von 32 bis 47 für Geräte reserviert, die den zu unterstützenden Geräten zugeordnet werden müssen. Von diesen sind auch noch einige fest reserviert (z.B. für die Tastatur). Daher ist – je nach System – ein *Interrupt-Sharing* erforderlich. Mehrere Geräte müssen sich eine Interrupt-Nummer teilen. Bei Auftreten eines Interrupts muss dann ermittelt werden, welches Gerät den Interrupt tatsächlich ausgelöst hat.

⁶ Beim Befehl *int 0x21* aus dem 0x86-Befehlssatz muss im Register AX die Interrupt-Nummer zur Adressierung des konkreten Vektors in der Interrupt-Vektor-Tabelle stehen. Dieser Software-Interrupt wurde im MS-DOS-System sehr häufig verwendet.

Bei einem Hardware-Interrupt, der durch den 8259-Controller initiiert wird, erfolgt die Adressierung durch eine Abbildung der IRQ-Leitung, über die der Interrupt gemeldet wird, auf die Interrupt-Nummer. Die Nummerierung von IRQs beginnt bei IA32-Prozessoren bei 0 und geht bis 15. Es ist also eine Abbildung auf den Interrupt-Nummernkreis (32 bis 47) durch Addition von 32 erforderlich. Die Aufgabe der Abbildung übernimmt der Interrupt-Controller.

Die Mehrzahl der neueren Computersysteme ist zumindest im PC- und im unteren Serverbereich mit APIC-Bausteinen ausgestattet. APIC-Bausteine unterstützen Multiprozessorsysteme und verfügen über 256 Interruptleitungen. Diese Bausteine unterstützen auch einen PIC-Kompatibilitätsmodus mit 15 Interrupts, die dann allerdings nur an einen Prozessor übermittelt werden.

3.1.4 Interrupt-Bearbeitung unter Windows

Unter Windows werden für die Interrupt-Bearbeitung sog. *Traphandler* bereitgestellt. Der Prozessor überträgt die Steuerung bei Auftreten eines Interrupts an einen Traphandler. Der Begriff Traphandler ist etwas irreführend, im Prinzip handelt es sich hier aber um Interrupt-Service-Routinen. Die Einteilung der Interrupts unter Windows entspricht im Wesentlichen der bereits in Abbildung 3-1 eingeführten Klassifizierung und ist in Abbildung 3-9 skizziert. Obwohl Interrupt-Controller wie der PIC-Baustein bereits eine Interruptpriorisierung vornehmen, ordnet der Windows-Kernel den Interrupts über sog. *Interrupt-Request-Levels* (Einstellung der Unterbrechungsanforderung, IRQL) eigene Prioritäten zu. Sie sind prozessorspezifisch und können Werte zwischen 0 und 31 annehmen, wobei 31 die höchste Priorität ist.

Das IRQL-Konzept darf nicht mit den Thread-Prioritäten, die wir später noch kennenlernen werden, verwechselt werden. Die IRQL-Einstellung legt fest, welche Interrupts ein Prozessor empfangen kann. Wenn eine Ausführungseinheit eines Programms (ein Thread) im Kernelmodus abläuft, werden alle Interrupts, deren IRQL-Zuordnung kleiner oder gleich der aktuellen ist, maskiert (also verboten), bis der aktuelle Thread die IRQL-Einstellung wieder verringert. Nur Interrupts mit höherer IRQL-Zuordnung werden zugelassen.

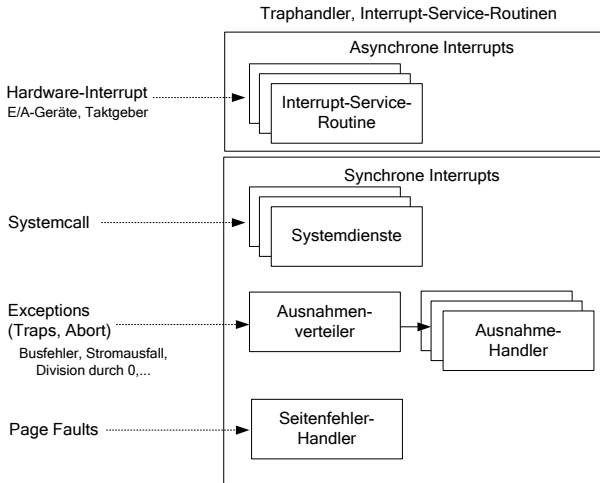


Abbildung 3-9: Interrupt-Verteilung unter Windows (Solomon 2005)

Alle Threads (Kernel- und Usermodus) laufen unter Interrupt-Priorität 0 ab und sind damit durch Hardware-Interrupts unterbrechbar. Nur Interrupts auf einem höheren IRQL dürfen eine Interrupt-Bearbeitung auf niedrigerem IRQL unterbrechen. Über eine *Interrupt-Dispatcher⁷-Tabelle* (IDT) wird festgehalten, welche ISR für welchen IRQL zuständig ist.

Abbildung 3-10 zeigt etwas vereinfacht die IDT, die jedem IRQL eine Bearbeitungsroutine zuordnet. In der IDT ist beispielsweise unter IRQL 31 ein Verweis auf die Routine zum Herunterfahren des Systems zu finden, für IRQL 0 wird keine ISR zugeordnet, da auf diesem Level die normale Programmausführung stattfindet. Bei nicht unterbrechbaren Kernaufgaben können auch bestimmte IRQL maskiert und damit ausgeschaltet werden.

Im IRQ-Level 1 werden sog. APC-Interrupts bearbeitet. Dies ist ein spezieller Windows-Mechanismus, der sog. asynchrone Prozeduraufrufe behandelt. Für jeden Thread wird sowohl im Kernel- als auch im Usermodus eine APC-Queue mit APC-Interrupt-Objekten verwaltet. Setzt beispielsweise ein Thread einen Read-Befehl auf eine Festplatte ab, so wird die Threadbearbeitung zunächst unterbrochen. Wenn der Befehl vom Kernel abgearbeitet ist und die Daten eingelesen wurden, wird ein APC-Interrupt erzeugt und ein APC-Objekt in die APC-Queue des ent-

⁷ Dispatching = Arbeitsvorbereitung, Disposition

sprechenden Threads eingehängt. Die APC-Queue wird nach und nach abgearbeitet (Solomon 2005).

Die Zuordnung von IRQs auf Hardware-Interruptnummern wird in der *Hardware Abstraction Layer* (HAL) durchgeführt, da sie prozessorspezifisch ist. Damit ist die Interrupt-Bearbeitung im Kernel auch relativ unabhängig von Hardware-Besonderheiten.

IRQL	Verweis auf ISR oder Verteileroutine	
31 - High	Routine zum Herunterfahren des Systems	Hardware Interrupts
30 - Power	Routine zum Ausschalten der Hardware	
...	...	
28 - Systemuhr (Clock)	Uhr - Handler	
...	...	
3-n - Hardware-Geräte-Interrupts	ISR für Geräte-Interrupts	Software Interrupts
2 - Thread	Routine für DPC - Verteilung	
1 - APC-Interrupts	Behandlung von APC-Interrupts	
0 - Passiv	Normale Threadausführung, kein ISR	

Abbildung 3-10: Interrupt-Verteilungstabelle im Windows für IA32-Architektur

Die Gerätetreiber können ihre Einstellungen über sog. *Interruptobjekte* vornehmen. Ein Interruptobjekt enthält aus Kernelsicht alle Informationen (ISR-Adresse,...), die er benötigt, um eine ISR mit einer Interrupteinstellung (einem IRQ) zu verknüpfen und die IDT zu belegen.

Wie bereits dargestellt, unterbricht in x86-basierten Rechnersystemen der Interrupt-Controller (meist PIC-8259-Baustein) den Prozessor auf einer Leitung. Der Prozessor fragt dann den Controller ab, um eine Unterbrechungsanforderung (IRQ) zu erhalten. Der Interrupt-Controller übersetzt den IRQ in eine Interruptnummer (Index auf IDT) und mit diesem Index kann über die IDT auf die Adresse der Interrupt-Service-Routine für den entsprechenden Hardware-Interrupt zugegriffen werden. Die ISR kann über diese Adresse aufgerufen werden.

Die Belegung der IRQs ist prozessorabhängig. Anders als für IA32-Prozessoren verwaltet Windows für Prozessoren mit x64- sowie die IA64-Architektur beispielsweise eine etwas andere Belegung der IRQs. Hier werden nur 16 IRQs (siehe Abbildung 3-11) verwendet. Wie man in der Abbildung sieht, gibt es z.B.

deutlich weniger IRQs für Geräte. Auf die genaue Belegung der IRQs und deren Bedeutung soll hier nicht weiter eingegangen werden.

Im Falle von Geräte-Interrupts gibt es unter Windows eine Besonderheit. Die ISR erzeugt nämlich meist eine sog. *DPC-Datenstruktur* (Deferred Procedure Call⁸) und trägt sie in eine prozessorspezifische *DPC-Queue* ein. Dies wird gemacht, um die Bearbeitungszeiten in der ISR möglichst kurz zu halten und damit das System nicht unnötig zu blockieren. Die adressierte DPC-Routine wird dann später mit niedrigerer Priorität auf IRQ-Level 2 aufgerufen und abgearbeitet. Hierfür gibt es einen eigenen Verwaltungsmechanismus.

Entsprechend der Interrupt-Priorisierung werden also zunächst alle anstehenden DPC-Objekte (IRQ-Level 2) und danach alle APC-Objekte (IRQ-Level 1) abgearbeitet, bevor wieder die normale Threadausführung (IRQ-Level 0) weitergeführt wird.

	x64-Architektur	IA64-Architektur
15	High/Profile	High/Profile/Power
14	Inter-processor interrupt / Power	Inter-processor interrupt
13	Clock	Clock
12	Synch	Synch
11	Device n	Device n
		.
4	.	Device 1
		.
3	Device 1	Correctable Machine Check
2	Dispatch/DPC	Dispatch/DPC
1	APC	APC
0	Passive/Low	Passive/Low

Abbildung 3-11: Interrupt-Verteilungstabelle im Windows für 64-Bit-Architekturen (Solomon 2005)

Der Ablauf der Interrupt-Bearbeitung für einen Hardware-Interrupt unter Windows ist grob und vereinfacht in der Abbildung 3-12 dargestellt. Der Interrupt-Controller gibt den Interrupt zunächst an eine IRQ-Komponente im Kernel weiter, der anhand des IRQ den IRQL ermittelt und diesen für die Indizierung der IDT benutzt, um die passende ISR zu ermitteln. Die ISR erstellt dann ein Element für

⁸ Verzögerter Prozeduraufruf.

die DPC-Queue und reiht es in die Queue ein. Die Abarbeitung des Interrupts erfolgt dann, sobald das Element der DPC-Queue an der Reihe ist. Aus dem IRQ wird also der IRQL ermittelt und über diesen wiederum die ISR (IRQ → IRQL → ISR), die letztendlich in diesem Fall nur einen DPC-Eintrag erzeugt.

Wie bereits in der Einführung zu Betriebssystemarchitekturen erläutert, liegen die Gerätetreiber unter Windows gekapselt in der *Hardware Abstraction Layer*, womit die Abhängigkeiten zu unterstützten Hardwareplattformen im Wesentlichen in dieser Schicht konzentriert sind.

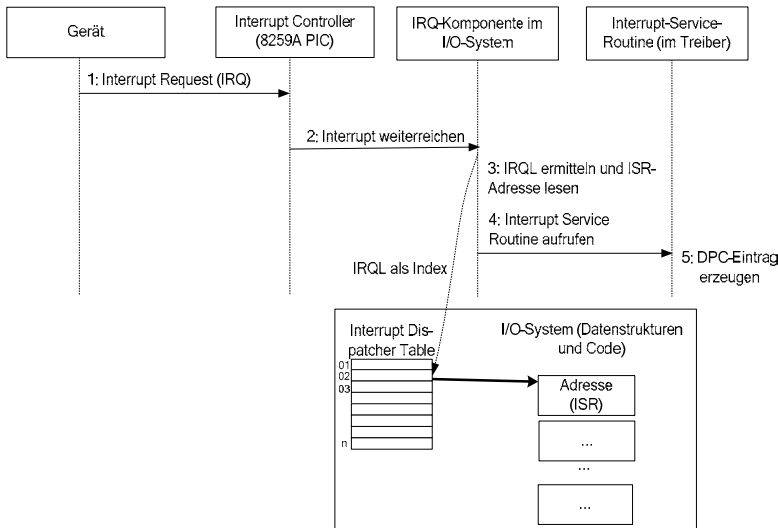


Abbildung 3-12: Bearbeitung eines Geräte-Interrupts im Windows

3.1.5 Interruptverarbeitung unter Linux

In Linux-Systemen wird ebenfalls eine Tabelle mit Referenzen auf die Interrupt-Handler, (Interrupt-Service-Routinen, ISR) verwaltet. Jedem Interrupt wird über einen Index auf die Tabelle ein Interrupt-Handler zugeordnet. Es erfolgt eine Abbildung eines IRQ auf einen Tabellenindex.

Bei Auftreten eines Interrupts wird zunächst in den Kernelmodus gewechselt, sofern nicht schon geschehen, und anschließend werden die Register gesichert. Nach Ausführen der ISR wird zunächst geprüft, ob ein neuer Prozess zur Ausführung kommt oder ob der unterbrochene Prozess fortgesetzt werden soll. Der Registersatz des ausgewählten Prozesses wird dann wiederhergestellt und es wird in den Benutzermodus gewechselt. Interrupt-Handler sind üblicherweise sehr kurze Routinen. Alles was nicht unbedingt sofort erledigt werden muss, wird zunächst no-

tiert und später abgewickelt. Dieser, dem DPC-Mechanismus von Windows verwandte, Mechanismus wird in Linux als *Tasklet* bezeichnet. Tasklets sind kleine Tasks, die in einer Datenstruktur namens *tasklet_struct* mit entsprechenden Informationen zur späteren Verarbeitung verwaltet werden. Vom sog. Tasklet-Scheduler (Scheduling-Mechanismen werden in Kapitel 5 ausführlich erörtert) wird eine Liste aller anstehenden Tasklets geführt und deren Abarbeitung vorgenommen. Die Interrupt-Handler-Tabelle ist im System wie folgt definiert (siehe Mauerer 2004 und Bovet 2005):

```
extern irq_desc_t irq_desc [NR_IRQS];
```

Ein Eintrag in der Tabelle (in C: Array) hat folgenden Aufbau:

```
typedef struct {  
    unsigned int status; // IRQ-Status  
    // Zeiger auf verantwortlichen IRQ-Controller  
    hw_irq_controller *handler;  
    // Zeiger auf Action-Liste  
    struct irqaction *action;  
    // Spezielles Feld zum Aktivieren und Deaktivieren des IRQ  
    unsigned int depth;  
    ...  
} ____cacheline_aligned irq_desc_t;
```

Ein IRQ ist über diese Datenstruktur vollständig beschrieben, wobei Hardware-spezifika verborgen werden. In einer Systemkonstante namens *NR_IRQS* wird die max. Anzahl der Tabelleneinträge festgelegt. Bei einem IA64-Prozessor sind es 256 mögliche IRQs.

Im Feld *depth* wird für den Interrupt-Handler hinterlegt, ob der zugehörige IRQ hardwaretechnisch aktiviert (Wert = 0) ist oder nicht (Wert > 0). Jedesmal wenn irgendwo im Kernel der IRQ deaktiviert wird, wird der Zähler *depth* um 1 erhöht. Bei einer Aktivierung wird er um 1 vermindert. Damit ist auch bekannt, wie oft er zu einem bestimmten Zeitpunkt deaktiviert wurde. Nur wenn das Feld auf 0 steht, darf der Interrupt durch die Hardware ausgelöst werden. Im Feld *status* wird der aktuelle Zustand des Interrupt-Handlers festgehalten. Wenn im Feld *status* z.B. ein Wert „IRQ_DISABLED“ ist, bedeutet dies, dass der Interrupt-Handler abgeschaltet ist, hat das Feld den Wert "IRQ_INPROGRESS" ist der Interrupt-Handler aktiv.

Die beiden Zeiger *action* und *handler* verweisen auf weitere Datenstrukturen im Kernel. Der Zeiger *action* verweist z.B. auf eine Struktur *irqaction*, die als Action-Liste bezeichnet wird und als Elemente sog. *Action-Deskriptoren* enthält. In dieser Liste sind dann je Listeneintrag die Verweise auf die eigentlichen Interrupt-

Service-Routinen abgelegt. Diese Struktur ist im Headerfile *interrupt.h* zu finden und hat folgenden Aufbau:

```
struct irqaction { // Action-Descriptor
    // Verweis auf Interrupt-Service-Routine
    void (*handler)(int, void *, struct pt_regs *);
    // Eigenschaften des Interrupt-Handlers
    unsigned long flags;
    // Name des Interrupt-Handlers
    const char *name;
    // Eindeutige Identifikation des Interrupt-Handlers
    void *dev_id;
    // Verweis auf den Nachfolger in der Liste
    struct irqaction *next;
};
```

Wie man erkennen kann, übergibt man einem Interrupt-Handler beim Aufruf drei Parameter. Der erste Parameter ist vom Typ *int* und enthält als Wert die IRQ-Nummer, der zweite Parameter ist ein Zeiger auf eine Device-Identifikation (Typ *void*) und der dritte Parameter ist ein Zeiger auf die Struktur *pt_regs*, mit den aktuellen Registerinhalten.

Wird ein IRQ-Eintrag in der Interrupt-Vektor-Tabelle für die Bearbeitung mehrerer IRQs verwendet (Interrupt-Sharing), so zeigt der Zeiger *next* auf die *nächste irqaction*-Instanz innerhalb einer einfach verketteten Liste. Der Kernel muss dann durch diese Liste traversieren, bis er die richtige Routine herausgefunden hat. Weiterführende Ausführungen zu den verwendeten Datenstrukturen und deren Nutzung sind in Mauerer 2004 und Bovet 2005 zu finden.

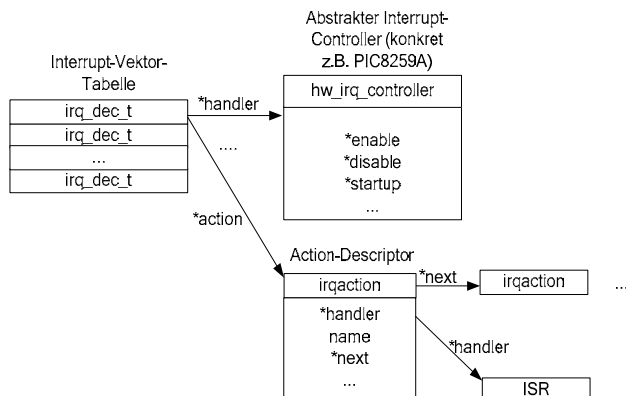


Abbildung 3-13: Kernel-Datenstrukturen für die Interrupt-Bearbeitung unter Linux

Der Zeiger *handler* verweist auf eine Datenstruktur, welche die speziellen Eigenschaften eines IRQ-Controllers abstrahiert. Diese Datenstruktur namens *hw_irq_controller* enthält im Wesentlichen wiederum Zeiger auf sechs Controller-Funktionen (*startup, disable, enable,...*). Die Zusammenhänge der an der Interrupt-Bearbeitung beteiligten Datenstrukturen sind in Abbildung 3-13 skizziert. IRQ-Controller werden durch entsprechende Gerätetreiber dynamisch beim Kernel registriert. Aktuelle Informationen zu den Interrupts können aus dem Verzeichnis */proc/interrupts/* des Systems ausgelesen werden.

3.2 Systemaufrufe

Das Betriebssystem ist von den Anwendungsprozessen abgeschottet, wobei – wie bereits erläutert – das Konzept der virtuellen Maschine benutzt wird. Anwendungsprogramme nutzen die Dienste des Betriebssystems über eine Zugangsschicht. Der Aufruf der Dienste erfolgt über sog. Systemcalls.

Ein Systemcall ist ein Dienstauftrag an das Betriebssystem, bei dessen Ausführung in den Kernelmodus gewechselt wird. Der Kontrollfluss wird dabei meist von einem Anwendungsprogramm an den Kernel übergeben. Linux unterstützt z.B. im Prinzip alle Unix-Entwicklungslinien und deren Systemaufrufe. Darüber hinaus gibt es auch noch Linux-spezifische Systemaufrufe. Beispiele für Systemaufrufe unter Linux sind:

- *fork* zur Prozesserzeugung
- *exit* zum Beenden eines Prozesses
- *open* zum Öffnen einer Datei
- *close* zum Schließen einer Datei
- *read* zum Lesen einer Datei

Die Windows-API (Win-API⁹, früher auch als Win32-API bezeichnet) stellt unter Windows eine Grundmenge an Funktionen für die Anwendungsentwicklung bereit, die u.a. den Zugriff auf Kernel-Ressourcen ermöglichen. Von den vielen API-Funktionen der Win-API, sind einige Systemcalls. Beispiele hierfür sind:

- *CreateProcess* zur Prozesserzeugung
- *ExitProcess* zum Beenden eines Prozesses
- *CreateFile* zum Erzeugen einer Datei
- *ReadFile* zum Lesen einer Datei

⁹ API steht für Application Programming Interface, also eine Programmierschnittstelle, die sowohl prozedural als auch objektorientiert ausgeprägt sein kann.

Damit die Anwendungsprogramme die Adressen von Systemroutinen nicht kennen müssen, wird ein spezieller Aufrufmechanismus verwendet, der auch als Trap bezeichnet wird. Ein Trap ist ein Software-Interrupt, der für einen Übergang vom Benutzermodus in den Kernelmodus sorgt, um den aufgerufenen Systemdienst auszuführen. Traps werden üblicherweise durch einen speziellen Maschinenbefehl des Prozessors, den sog. Supervisor-Call oder SVC unterstützt. Bei Ausführung des Systemcalls über einen Software-Interrupt wird wie folgt verfahren:

- Der aktuelle *Kontext* des laufenden Programms, also die Information, welche den aktuellen Status eines Prozesses beschreibt (später dazu mehr), wird gesichert.
- Der Program Counter wird mit der Adresse der passenden Systemroutine belegt.
- Vom Benutzermodus wird in den Kernelmodus geschaltet.
- Die adressierte Systemroutine wird durchlaufen.
- Anschließend wird wieder der alte Kontext des Prozesses hergestellt und der Program Counter mit der Adresse des Befehls nach dem Systemcall belegt.

Der Trap-Mechanismus hat den Vorteil, dass der Aufruf eines Systemcalls von einem Anwendungsprogramm aus ermöglicht wird, ohne dass die tatsächliche Adresse der Systemroutine bekannt sein muss (Information Hiding). Alle Systemcalls zusammen bilden die Schnittstelle der Anwendungsprogramme zum Betriebssystemkern (Kernel).

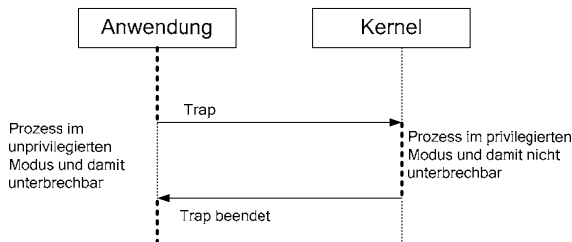


Abbildung 3-14: Traps bei seriellen Kernel

Man unterscheidet serielle (Abbildung 3-14) und nebenläufige Kernel-Implementierungen (siehe Abbildung 3-15). Ein serieller Kernel verarbeitet zu einem Zeitpunkt höchstens einen Systemcall und ist dabei nicht unterbrechbar. Im Gegensatz dazu ist ein nebenläufiger Kernel unterbrechbar und verarbeitet mehrere Systemcalls quasi gleichzeitig.

Heutige Betriebssysteme verfügen weitgehend über einen nebenläufigen Kernel. Sie versuchen aber, mit möglichst kurzen, nicht unterbrechbaren Phasen auszukommen. Dies ist besonders für Multiuser-/Multitasking-Betriebssysteme wichtig.

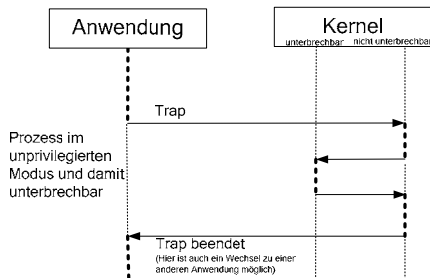


Abbildung 3-15: Traps bei nebenläufigem Kernel

Der Aufruf eines synchronen Software-Interrupts wird bei Intel-x86-Prozessoren über den Maschinenbefehl *int* vollzogen. Diesem wird beim Aufruf die Interrupt-Nummer mitgegeben. Dieser Maschinenbefehl wird auch als SVC-Befehl (Supervisor-Call) bezeichnet und ist wie folgt definiert:

```
int n // Aufruf des Interrupts mit der Nummer n, mit n = 0...255 (dezimal)
```

Die Interrupt-Nummer *n* stellt den Index zum Auffinden des Interrupt-Vektors in der Interrupt-Vektor-Tabelle dar. Bei Aufruf des *int*-Befehls wird im PSW der Kernelmodus gesetzt. Der Aufruf eines Systemcalls unter Unix wird z.B. über diesen Befehl ausgeführt, indem als Parameter *0x80* mitgegeben wird:

```
int $0x80 // Trap, Systemcall
```

Die Einbettung des Befehls in ein C-Programm unter Linux soll an einem einfachen Beispiel gezeigt werden. In diesem Beispiel wird eine Datei mit dem Dateinamen „mandl.txt“ zum Lesen geöffnet. Der C-Code ist, wie die folgenden Zeilen zeigen, sehr einfach:

```
#include ...  
...  
main()  
{  
    open("mandl.txt",1); // Datei zum Lesen öffnen  
}
```

Der vom Compiler erzeugte Maschinencode (gcc-Compiler) sieht wie folgt aus:

```
.LC0:  
    .string "mandl.txt"  
    .text  
.globl main
```

```
main:
    ...
    call open
    ...
```

Im generierten Code wird mit dem Befehl *call open* die Open-Routine der C-Library aufgerufen. In dieser Routine wird vor dem tatsächlichen Systemcall eine Belegung der Parameter vorgenommen. In unserem Beispiel werden einige Register belegt. Schließlich wird der Befehl *int* mit der Interrupt-Nummer 0x80 aufgerufen. Die Rückkehr zum aufrufenden Programm erfolgt über den Aufruf des Maschinenbefehls *iret*. Er stellt den alten Zustand wieder her und verzweigt zur Aufrufstelle. Dies ist in diesem Fall der Befehl, der nach dem Aufruf *call open* folgt.

```
__libc_open:
    ...
    mov 0xc(%esp,1), %ecx // Parameter für Open in Register laden
    mov ...
    mov $0x5, %eax        // Systemcall-Code für open-Funktion
    int $0x80             // Systemcall
    ...
    iret                  // zurück zum Aufrufer
```

Über die Auswertung der Interrupt-Nummer wird die entsprechende Interrupt-Routine in der Interrupt-Vektor-Tabelle gefunden. Die Interrupt-Vektor-Tabelle wird bei der Systeminitialisierung mit den Adressen der ISRs belegt. In Abbildung 3-16 ist die Abarbeitung eines Systemcalls nochmals als Sequenzdiagramm dargestellt. Die im Bild erwähnte Systemcall-Tabelle ist eine Kernetabelle, welche die Adressen aller Systemdienstroutinen enthält. Sie wird unter Linux in einer Datenstruktur mit der Bezeichnung *sys_call_table* verwaltet.

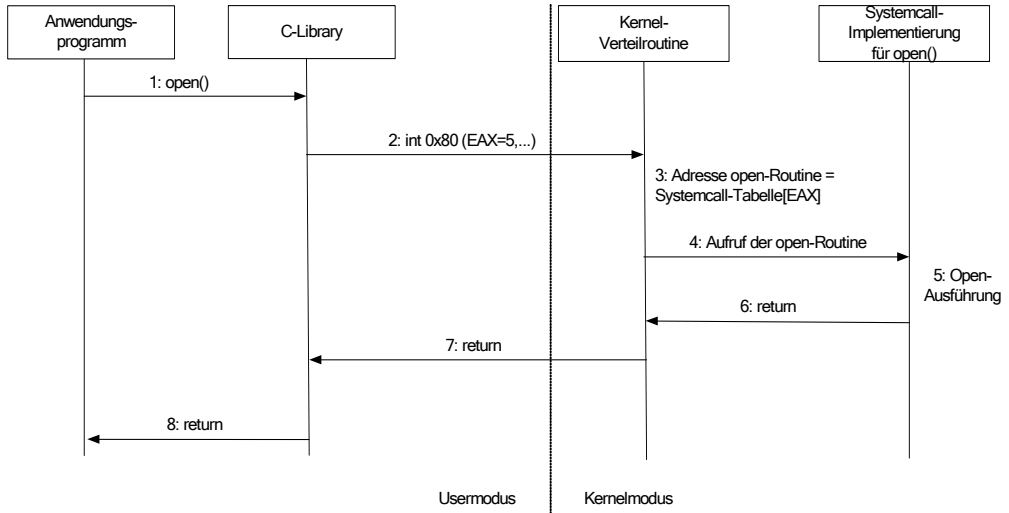


Abbildung 3-16: Ablauf eines Systemcalls unter Linux

3.3 Übungsaufgaben

1. Was ist der Unterschied zwischen Polling und interruptgesteuerter Verarbeitung?
2. Was ist der Unterschied zwischen den Exception-Typen Fault und Trap? Nennen Sie jeweils ein Beispiel!
3. Wozu dient ein Systemcall und wie wird er üblicherweise von einem Betriebssystem wie Windows oder Unix ausgeführt?
4. Was bedeutet „Maskierung“ von Unterbrechungsanforderungen?
5. Wie erkennt die CPU, dass eine Unterbrechungsanforderung ansteht?
6. Was versteht man unter einer Interrupt-Vektor-Tabelle?
7. Was ist eine Interrupt-Service-Routine und wann wird sie aufgerufen?
8. Was bedeutet Interrupt-Sharing?
9. Was versteckt der Trap-Mechanismus zum Aufruf eines Systemdienstes vor dem Anwendungsprogramm?
10. Erläutern Sie die Abwicklung eines Hardware-Interrupts unter Linux. Gehen sie dabei auf Tasklets ein!
11. Erläutern Sie die Abwicklung eines Traps (Systemcalls) bei einem seriellen und bei einem nebenläufigen Kernel!
12. Nennen Sie den Unterschied zwischen einem synchronen und asynchronen Interrupt!
13. Warum verwendet Windows den DPC-Mechanismus?
14. Welche Aufgabe hat ein Interrupt-Controller?

4 Prozesse und Threads

Das Prozess- und das Threadmodell sind wesentliche Konzepte der Betriebssystementwicklung und dienen als grundlegende Bausteine der Parallelverarbeitung. In diesem Kapitel wird ein Überblick über diese beiden Modelle und deren allgemeine sowie spezielle Einbettung in Betriebssysteme gegeben. Prozesse sind Betriebsmittel, die vom Betriebssystem verwaltet werden. Threads werden je nach Implementierung entweder direkt vom Betriebssystem oder von einem Laufzeitsystem einer höheren Programmiersprache (wie etwa der JVM¹) verwaltet.

Der Lebenszyklus von Prozessen und Threads wird anhand von Zustandsautomaten dargestellt. Die Informationen, die für die Verwaltung von Prozessen und Threads im Betriebssystem notwendig sind, werden ebenfalls erläutert. Hierzu werden die Datenstrukturen PCB (Process Control Block) und TCB (Thread Control Block) eingeführt.

Weiterhin wird in diesem Kapitel auf konkrete Betriebssysteme eingegangen und aufgezeigt, wie dort Prozesse und Threads genutzt werden. In den meisten Betriebssystemen wird der Prozessbegriff gleichwertig genutzt, Threads werden allerdings unterschiedlich implementiert. Dies soll in diesem Kapitel ebenfalls erörtert werden. In älteren Betriebssystemen, insbesondere aus dem Großrechnerbereich, wird für den Begriff des Prozesses auch der Begriff Task (siehe BS2000/OSD) verwendet. Auch in Realzeitsystemen oder eingebetteten Systemen spricht man häufig von Tasks. Als Beispielbetriebssysteme, die den Prozessbegriff nutzen, dienen Windows und Unix bzw. Linux. Die Nutzung von Prozessen und Threads wird anhand der Programmiersprachen C, Java und C# vermittelt.

Zielsetzung des Kapitels

Der Studierende soll das Prozess- und das Threadmodell und den Lebenszyklus von Prozessen und Threads innerhalb eines Betriebssystems verstehen und erläutern können. Weiterhin soll der Leser verstehen, welchen Aufwand ein Betriebssystem betreiben muss, um die Betriebsmittel *Thread* und *Prozess* zu verwalten, und wie das Prozess- und das Threadmodell in modernen Betriebssystemen implementiert werden. Die verschiedenen Möglichkeiten der Threadimplementierung mit ihren Vor- und Nachteilen sollten klar sein. Die Implementierungskonzepte sollen am

¹ JVM = Java Virtual Machine, die Laufzeitumgebung für Java-Programme.

Beispiel der Betriebssysteme Windows und Unix erläutert werden können. Zudem soll die Verwendung von Threads in modernen Sprachen wie Java und C# nachvollzogen werden können. Schließlich soll der Leser einschätzen können, wie die Unterstützung der Parallelverarbeitung in Multiprogramming-Betriebssystemen und der Einsatz von Threads in eigenen Anwendungsprogrammen funktioniert. Für welche Anwendungsfälle Threads sinnvoll sind, sollte ebenfalls erläutert werden können.

Wichtige Begriffe

Prozess und Thread, User-Level-Thread und Kernel-Level-Thread, Prozess-Kontext, Thread-Kontext, Kontextwechsel, PCB (Process Control Block) und TCB (Thread Control Block).

4.1 Prozesse

4.1.1 Prozessmodell

Jedes Programm wird in Universalbetriebssystemen üblicherweise in einem (Betriebssystem-) *Prozess* ausgeführt, der zum Ablauf einem Prozessor zugeordnet werden muss.

Prozess. Ein Prozess (in manchen Betriebssystemen auch Task genannt) stellt auf einem Rechnersystem die Ablaufumgebung für ein Programm bereit und ist eine dynamische Folge von Aktionen mit entsprechenden Zustandsänderungen, oder anders ausgedrückt, die Instanziierung eines Programms. Als Prozess bezeichnet man auch die gesamte Zustandsinformation eines laufenden Programms. Ein *Programm* ist im Gegensatz dazu die (statische) Verfahrensvorschrift für die Verarbeitung auf einem Rechnersystem. Üblicherweise wird das Programm von einem Übersetzungsprogramm (Compiler) in eine ausführbare Datei umgewandelt, die vom Betriebssystem in den Adressraum eines Prozesses geladen werden kann. Ein Prozess ist damit ein Programm zur Laufzeit bzw. die konkrete Instanziierung eines Programms innerhalb eines Rechnersystems.

Virtuelle Prozessoren. Das Betriebssystem ordnet im Multiprogramming jedem Prozess einen *virtuellen Prozessor* zu. Bei echter Parallelarbeit wird jedem virtuellen Prozessor jeweils ein *realer Prozessor* zugeordnet. Im *quasi-parallelen* oder *pseudoparallelen* bzw. *nebenläufigen* Betrieb – und das ist in modernen Betriebssystemen der Normalfall – wird jeder reale Prozessor zu einer Zeit immer nur einem virtuellen Prozessor zugeordnet (siehe Abbildung 4-1). Prozess-Umschaltungen werden nach einer definierten Strategie vorgenommen. Bei einem Einprozessorsystem kann zu einer Zeit auch nur ein Prozess zum Ablauf kommen. Bei Mehrprozessorsystemen

können dagegen mehrere Prozesse zu einer Zeit zum Ablauf kommen, da es mehrere reale Prozessoren (CPUs) gibt.

Prozesse konkurrieren miteinander um die in heutigen Rechnersystemen meist knappen Betriebsmittel wie Speicherplatz oder Prozessor (bzw. zugeteilte CPU-Zeit), da die Anzahl der laufenden Prozesse meist größer ist als die Anzahl der Prozessoren. Die Zuordnung der Betriebsmittel übernimmt das Betriebssystem und nutzt dabei verschiedene Strategien, die als Scheduling-Strategien bezeichnet und weiter unten behandelt werden. Vom Betriebssystem wird dabei ein Zeitmultiplexing der verfügbaren Prozessoren durchgeführt.

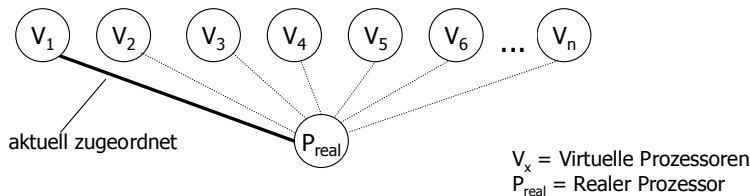


Abbildung 4-1: Virtuelle und reale Prozessoren

4.1.2 Prozessverwaltung

Das Betriebssystem verwaltet für jeden Prozess vielfältige Informationen, die als *Prozess-Kontext* bezeichnet wird.

Prozess-Kontext. Zum Prozess-Kontext zählen wir die Programme und Daten des laufenden Programms, die Informationen, die im Betriebssystem für einen Prozess verwaltet werden und die Inhalte aller Hardware-Register (Befehlszähler, PSW, MMU-Register, Mehrzweckregister,...). Die Registerinhalte bezeichnet man auch als Hardware-Kontext.

Der aktuell laufende Prozess (bei einer Einprozessormaschine kann dies nur einer sein) nutzt die CPU mit ihren Registern zur Bearbeitung. Verliert der gerade aktive Prozess die CPU, so muss ein sog. Kontextwechsel oder ein Kontext-Switching durchgeführt werden. Dabei wird der Hardware-Kontext des laufenden Prozesses gesichert und der Hardware-Kontext des neu aktivierten Prozesses in die Ablaufumgebung geladen.

Ein typischer Prozess mit seinem Prozess-Kontext ist in Abbildung 4-2 dargestellt. Der Adressraum eines Prozesses wird meist vom Compiler in verschiedene Bereiche aufgeteilt. Im Stack (Benutzerstack) werden die lokalen Variablen und die Rücksprunginformation für Methoden- bzw. Prozeduraufrufe abgelegt. Im Heap

werden dynamisch erzeugte Objekte eingetragen und schließlich gibt es auch noch einen Codebereich, in dem der Programmcode liegt.

Der Begriff des Adressraums wird in Kapitel 7 genauer erläutert. Im Vorgriff auf die Betrachtung der Speicherverwaltung sei angemerkt, dass es sich hier um alle Speicheradressen (real oder virtuell sei vorerst nicht relevant) handelt, die ein laufendes Programm in einem Prozess nutzen darf. Man spricht oft auch von Prozess-adressraum. Daten des Adressraums, die konkret von einem Maschinenbefehl benutzt werden, müssen zur Ausführungszeit im Hauptspeicher sein.

Im Kernelstack werden die Prozeduraufrufe, die für den Prozess innerhalb des Kernels, also im Kernelmodus, stattfinden mit Rücksprunginformationen usw. abgelegt. Die MMU-Register speichern die aktuellen Inhalte der Memory Management Unit, die für die Berechnung der physikalischen Adressen zuständig ist. Dies wird im Kapitel 7 noch ausführlicher diskutiert.

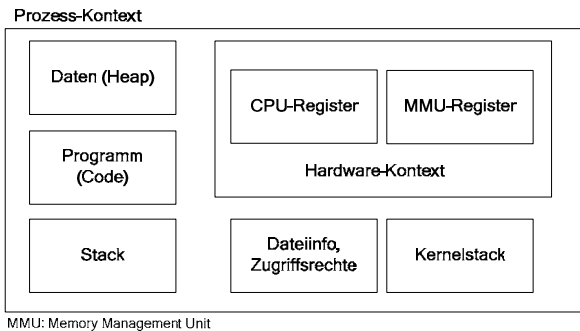


Abbildung 4-2: Prozess-Kontext

Beim Itanium-Prozessor gibt es beispielsweise auch noch eine sog. Register-Stack-Engine (RSE), die es ermöglicht, einem Prozess aus einem dynamischen Registerpool einige Register fest zuzuordnen. Diese Register stehen dann, ähnlich wie lokale Variable höherer Programmiersprachen, dem Prozess zur Verfügung. Nur ein Teil der Integerregister muss für alle Prozesse genutzt werden und kann beim Kontextwechsel vom Betriebssystem gesichert werden.

Prozesstabelle und PCB. Neben verschiedenen Tabellen zur Speicherverwaltung und zur Dateiverwaltung führt das Betriebssystem auch eine Tabelle mit allen aktuellen Prozessen in einer speziellen Datenstruktur. Diese Tabelle enthält die gesamte Information für die Prozessverwaltung. Diese kerneleigene Datenstruktur wird oft auch als *Prozesstabelle* bezeichnet. Ein Eintrag in der Prozesstabelle heißt *Process Control Block (PCB)*.

Der PCB ist eine der wichtigsten Datenstrukturen des Systems. Je nach Betriebssystem gibt es deutliche Unterschiede im Aufbau. Einige Informationen sind aber prinzipiell sehr ähnlich. Hierzu gehört u.a die Information zur Identifikation des Prozesses, die Information zum aktuellen Prozesszustand sowie Informationen zu sonstigen Ressourcen, die dem Prozess zugeordnet sind (Dateien, offene Netzwerkverbindungen). Wichtige Informationen zum Prozess sind also u.a:

- Programmzähler
- Prozesszustand
- Initiale (statische) und dynamische Priorität
- Verbrauchte Prozessorzeit seit dem Start des Prozesses
- Eigene Prozessnummer (PID) und Prozessnummer des erzeugenden Prozesses (Eltern- oder Vaterprozess genannt)
- Zugeordnete Betriebsmittel, z.B. Dateien (Dateideskriptoren) und Netzwerkverbindungen
- Aktuelle Registerinhalte (je nach Prozessor)

Wenn ein Prozess vom System durch einen anderen abgelöst wird, muss der Hardware-Kontext des zu suspendierenden Prozesses für eine erneute Aktivierung aufbewahrt werden. Der Hardware-Kontext eines zu suspendierenden Prozesses wird in seinem PCB gesichert, der Hardware-Kontext des neu zu aktivierenden Prozesses wird aus seinem PCB in die Ablaufumgebung geladen. Der Ablauf einer Prozessumschaltung ist in Abbildung 4-3 skizziert, wobei hier nur das Sichern und Laden der Programmzähler (PC) dargestellt ist. Natürlich müssen beim Kontextwechsel auch alle anderen Registerinhalte wie der Stackpointer (SP), das Program Status Word (PSW) sowie alle Mehrzweckregister usw. bewegt werden.

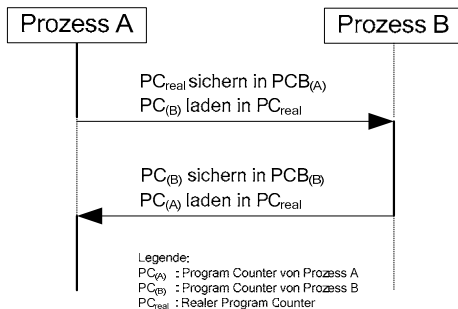


Abbildung 4-3: Prinzipieller Ablauf eines Kontextwechsels

4.1.3 Prozesslebenszyklus

Ein Prozess wird durch einen Systemcall erzeugt (z.B. in Unix durch den Systemcall *fork* oder unter Windows durch den Systemcall *CreateProcess*). Dabei werden der Programmcode und die Programmdaten in den Speicher geladen. Innerhalb des Betriebssystems wird einem Prozess eine eindeutige Identifikation zugeordnet, die als *Process Identification* oder kurz als *PID* bezeichnet wird. Bei der Erzeugung eines Prozesses wird auch ein neuer PCB in der Prozesstabelle angelegt.

Ein Prozess kann normal beim Ende des Programmablaufs oder auch von einem anderen Prozess beendet (terminiert) werden und durchläuft während seiner Lebenszeit verschiedene Zustände. Der Zustandsautomat eines Prozesses ist abhängig von der Implementierung des Betriebssystems.

In der Abbildung 4-4 ist der Zustandsautomat eines Prozesses für ein einfaches Betriebssystem mit vier Zuständen dargestellt. Im Zustand „bereit“ ist der Prozess zur Bearbeitung vorbereitet, im aktiven Zustand hat er eine CPU und im Zustand „blockiert“ wartet er auf Ressourcen, um weitermachen zu können. Im Zustand „beendet“ ist der Prozess dann schon nicht mehr im System vorhanden. Die Zustandsübergänge lassen sich wie folgt beschreiben:

1. Das Betriebssystem wählt den Prozess aus (Aktivieren)
2. Das Betriebssystem wählt einen anderen Prozess aus (Deaktivieren, Preemption, Vorrangunterbrechung)
3. Der Prozess wird blockiert (z.B. wegen Warten auf Input, Betriebsmittel wird angefordert)
4. Der Blockierungsgrund wird aufgehoben (Betriebsmittel verfügbar)
5. Prozessbeendigung oder schwerwiegender Fehler (Terminieren des Prozesses)

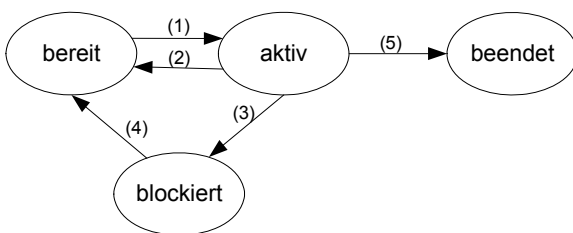


Abbildung 4-4: Einfacher Zustandsautomat eines Prozesses mit vier Zuständen

Die Zustandsautomaten moderner Betriebssysteme sind natürlich etwas komplexer und haben mehr Zustände, aber im Prinzip sind sie ähnlich konzipiert. Der

aktuelle Prozessstatus wird im PCB (oder in einer ähnlich benannten Datenstruktur) verwaltet.

4.2 Threads

4.2.1 Threadmodell

Prozesse sind Betriebsmittel, deren Verwaltung relativ aufwändig ist. Ein ergänzendes, ressourcenschonenderes Konzept hierzu sind die sog. Threads (deutsch: Fäden), die heute in allen modernen Betriebssystemen in unterschiedlicher Weise unterstützt werden.

Thread. Ein Thread stellt eine *nebenläufige Ausführungseinheit* innerhalb eines Prozesses dar. Threads werden im Gegensatz zu den traditionellen (schwergewichtigen, heavy-weight) Prozessen als *leichtgewichtige* (light-weight) Prozesse oder kurz LWP bezeichnet. Alle Threads eines Prozesses teilen sich den gemeinsamen Adressraum dieses Prozesses und können damit gleichermaßen auf Prozessdaten zugreifen und Programmcodes nutzen. Insbesondere global definierte Variable sind im Zugriff aller Threads eines Prozesses. Die „Leichtgewichtigkeit“ von Threads ergibt sich also aus der gemeinsamen Nutzung von dem Prozess zugeordneten Ressourcen.

Thread-Kontext. Gemäß dieser Definition erben Threads offene Dateien und Netzwerkverbindungen des Prozesses. Sie verfügen aber auch über einen eigenen Thread-Kontext mit einem eigenen Speicherbereich für den Stack, um ihre lokalen Variablen dort abzulegen, eigenen Registern und Befehlszählern, also auch einen eigenen Hardware-Kontext.

4.2.2 Implementierung von Threads

Die Implementierung von Threads kann auf Kernebene erfolgen, möglich ist aber auch eine Implementierung auf Benutzerebene. Die Implementierung hängt vom jeweiligen Betriebssystem ab. Im Windows-Betriebssystem (Windows 2000 und folgende) sind Threads z.B. auf Kernebene realisiert, die verschiedenen Unix-Derivate verfügen über mehrere Thread-Implementierungen sowohl auf der Kernel- als auch auf der Benutzerebene. Es gibt auch eine hybride Implementierung, in der mehrere Benutzerthreads auf Kernelthreads abgebildet werden. Dieses Implementierungskonzept wird z.B. von Sun Solaris genutzt (siehe Bengel 2002).

Für Threads werden unabhängig von ihrer konkreten Realisierung eigene Zustandsautomaten geführt, die den Prozess-Zustandsautomaten sehr ähneln. Im Betriebssystemkern oder im Benutzeradressraum wird für jeden Thread auch eine

eigene Datenstruktur zugeordnet, die oft auch als *Thread Control Block (TCB)* bezeichnet wird. Alle TCBs werden in einer Threadtabelle zusammengefasst. Es ist also üblich, dass Threads über einen eigenen Zustand, einen eigenen Befehlszähler, einen eigenen Registersatz (abgelegt im TCB) sowie einen eigenen Stack verfügen. Sie werden jedoch im Hinblick auf die Speicherverwaltung in den Betriebssystemen nicht als eine eigenständige Einheit, sondern immer im Kontext des Prozesses betrachtet.

Implementierung auf Benutzerebene

Bei Threads, die auf Benutzerebene realisiert sind, übernimmt die entsprechende Threadbibliothek (siehe Abbildung 4-5) das Scheduling und Umschalten zwischen den Threads, das Betriebssystem weiß davon nichts. In dieser Bibliothek sind Methoden bzw. Prozeduren bereitgestellt, die ein Erzeugen und Löschen eines Thread ermöglichen. Die Scheduling-Einheit ist in diesem Fall der Prozess. Die Threadtabelle wird im Benutzerspeicher verwaltet.

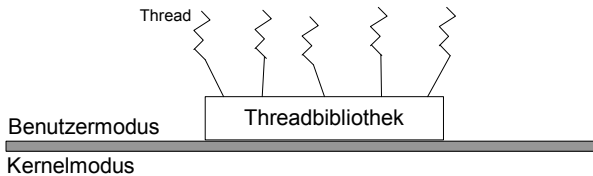


Abbildung 4-5: Threads auf Benutzerebene

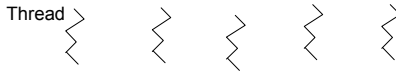
Vorteil dieser Implementierungsvariante ist die hohe Effizienz, da beim Thread-Kontextwechsel kein Umschalten in den Kernelmodus notwendig ist. Nachteilig ist, dass alle Threads eines Prozesses blockieren, wenn ein Systemaufruf innerhalb eines einzelnen Threads blockiert.

Implementierung auf Kernelebene

Bei Kernel-Threads werden die Threads im Kernelmodus verwaltet. Der Kernel stellt die Methoden bzw. Prozeduren (Systemaufrufe) zur Erzeugung und zum Löschen von Threads bereit, und die Scheduling-Einheit ist der Thread und nicht der Prozess. Eine spezielle Threadbibliothek für den Anwendungsprogrammierer ist nicht erforderlich (siehe Abbildung 4-6). Die Threadtabelle wird im Kernelspeicher verwaltet.

Benutzermodus

Kernelmodus

**Abbildung 4-6: Threads auf Kernelebene**

Die Implementierung von Threads auf Kernelebene hat gewisse Vorteile, aber auch Nachteile. Von Vorteil ist beispielsweise, dass das Betriebssystem in der Lage ist, die Zuteilung der Rechenzeit über die Threads zu gestalten und so einen Prozess nicht unnötig zu blockieren. Mit dieser Implementierungsvariante kann man auch Multiprozessorsysteme besser unterstützen, da das Betriebssystem ablaufbereite Threads selbstständig auf die verfügbaren CPUs verteilen kann. Ein weiterer Vorteil für Kernel-Threads ist, dass ein Prozess nicht blockiert ist, wenn ein Thread innerhalb des Prozesses blockiert ist. Ein anderer Thread des Prozesses kann weiterarbeiten. Nachteilig ist beispielsweise, dass im Kernel implementierte Threads nicht so effizient sind, da sie sich bei jedem Thread-Kontextwechsel an den Kernel wenden müssen (Software-Interrupt). Weiterhin ist die größere Systemabhängigkeit von Nachteil.

Die speziellen Thread-Implementierungen der Hersteller sind meist nicht kompatibel zueinander. Daher hat das IEEE einen POSIX-Standard für Threads spezifiziert, der von den meisten Betriebssystemherstellern unterstützt wird. POSIX-Threads sind in POSIX Sektion 1003.1c seit 1995 standardisiert.

Zuordnung von Threads zu Prozessen

Threads und Prozesse können auf unterschiedliche Weise miteinander kombiniert werden. Folgende Kombinationsmöglichkeiten findet man vor:

- 1:1-Beziehung zwischen Thread und Prozess, wie im ursprünglichen Unix-System. In einem Prozess läuft also genau eine Ablaufeinheit (Thread).
- 1:n-Beziehung zwischen Prozess und Thread, wie dies heute unter Windows und Linux üblich ist. Ein Prozess kann also mehrere Threads beherbergen.
- In Betriebssystemen, die man in der Forschung findet, sind auch n:1- und n:m-Beziehungen möglich.

1:n-Beziehungen, bei denen ein Prozess beliebig viele Threads enthalten kann, sind heute in Betriebssystemen üblich, unabhängig davon, ob die Threads im Benutzer- oder im Kernelmodus implementiert sind. Eine 1:1-Beziehung bedeutet im Prinzip, dass es in einem Prozess nur eine Ablaufeinheit gibt. Hier könnte man auch auf den Threadbegriff verzichten.

Zuordnung von User-Level-Threads zu Kernel-Level-Threads

In heutigen Rechnersystemen findet man sowohl Threads auf der Benutzerebene (auch als User-Level-Threads bezeichnet) als auch auf der Kernelebene (Kernel-Level-Threads) vor. Wie bereits dargestellt, sind User-Level-Threads solche Threads, die im Benutzermodus, also in der Regel in einem Laufzeitsystem einer Programmiersprache, verwaltet werden, Kernel-Level-Threads verwaltet dagegen der Kernel.

Man kann bei der Zuordnung von User-Level-Threads zu Kernel-Level-Threads verschiedene Varianten unterscheiden. Eine $m:1$ -Zuordnung bedeutet z.B., dass alle m Threads, die im Usermodus für einen Prozess sichtbar sind, einem vom Kernel verwalteten Thread (Kernelthread) zugeordnet werden. Entsprechend unterscheidet man $1:1$ - und $m:n$ -Zuordnungen.

Wenn das Betriebssystem kein Multithreading unterstützt, findet man in der Regel eine $m:1$ -Zuordnung vor. In diesem Fall wird die CPU nur an Prozesse und nicht an einzelne Threads vergeben, die Scheduling-Einheit ist also der Prozess (siehe hierzu die Ausführungen zum CPU-Scheduling in Kapitel 5). Das Laufzeitsystem der verwendeten Programmiersprache kümmert sich um die Threadverwaltung. Die Zuteilung von CPU-Zeit auf einzelne Threads findet im Benutzermodus statt.

Bei einer $1:1$ -Zuordnung kennt das Betriebssystem dagegen Threads und führt Buch über deren Lebenszyklus. Damit wird auch die Zuteilung der CPU über die Threads gesteuert, und die Zuteilung der CPU auf Threads findet im Kernelmodus statt. Eine $m:n$ -Zuordnung kommt in der Praxis nicht vor.

4.2.3 Vor-/Nachteile und Einsatzgebiete von Threads

Ein Vorteil gegenüber Prozessen ergibt sich durch den wesentlich schnelleren Thread-Kontextwechsel im Vergleich zum Prozess-Kontextwechsel. Daher kommt auch der Begriff *LWP*. Dies lässt sich damit begründen, dass durch Threads meist Speicherbereiche des gleichen Prozesses verwendet werden. Ein Austausch von Speicherbereichen ist daher oft nicht erforderlich, was den Betriebssystem-Overhead reduziert.

Weiterhin unterstützen Threads die Parallelisierung der Prozessarbeit und können sinnvoll bei Nutzung mehrerer CPUs genutzt werden. Sie sind ein fein-körniger Mechanismus zur Parallelisierung von Anwendungen. Man muss Software natürlich auch so schreiben, dass sie gewisse Aufgaben in nebenläufigen Threads ausführt. Threads eines Prozesses sind aber nicht gegeneinander geschützt und müssen sich daher beim Zugriff auf die gemeinsamen Prozess-Ressourcen abstimmen (synchronisieren). Die Software muss also *threadsafe* bzw. *reentrant* (wiedereintritts-

fähig) programmiert werden. Eine Methode einer Objektklasse bzw. eine Prozedur eines Moduls ist dann threadsafe, wenn ein Thread, der sie ausführt, die nebenläufige oder auch die folgende Ausführung eines weiteren Threads in keiner Weise beeinträchtigt. Dies könnte z.B. passieren, wenn eine globale Variable verwendet und auf diese ungeschützt zugegriffen wird.

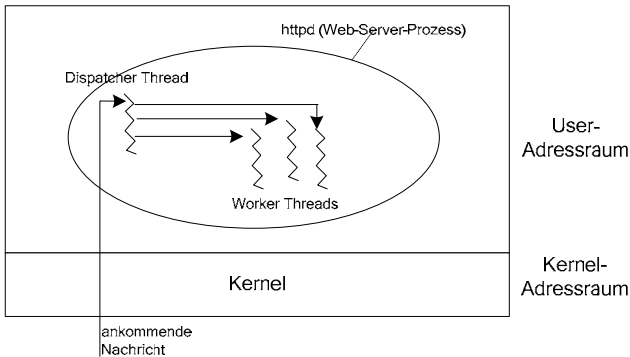


Abbildung 4-7: Einsatz von Threads im Web-Server (Tanenbaum 2009)

Einsatzgebiete für Threads sind Programme, die eine Parallelverarbeitung ermöglichen. Typisch sind hierfür etwa dialogorientierte Anwendungen, bei denen ein Thread auf Eingaben des Anwenders über die Benutzeroberfläche wartet und andere Threads im Hintergrund Rechenaufgaben erledigen können. Rahmenwerke für die Realisierung von graphischen Oberflächen wie *AWT*, *SWT* bzw. *Swing* in Java² nutzen beispielsweise mehrere Threads. Auch ein Web-Browser oder auch ein Textverarbeitungsprogramm wie Microsoft Word sind beispielsweise Programme, die Threads intensiv einsetzen. Ein klassisches Beispiel für die sinnvolle Nutzung von Threads ist auch ein Web-Server, der mit einem Thread auf ankommende Verbindungen von neuen Web-Clients (Browsern) wartet und mit anderen Threads die bestehenden Verbindungen bedient (siehe Abbildung 4-7). Ein Web-server oder ein anderer Server, der auf mehrere ankommende Requests wartet und diese dann ausführt, könnte mit Threads, etwa wie es im folgenden Pseudocode dargestellt ist, formuliert werden.

```
Dispatcher() {
    while (true) {
        r = receive_request();
```

² AWT, SWT und Swing sind Java-Mechanismen, die eine Reihe vordefinierter Java-Klassen zur GUI-Entwicklung bereitstellen.

```
        start_thread(workerThread, r);
    }
}
workerThread(r) {
    a = process_request(r, a);
    reply_request(r,a);
}
```

Im Pseudocode ist der Server ein Verteiler bzw. Zuteiler (Dispatcher = Arbeitsverteiler), der in einer Endlosschleife auf ankommende Requests wartet, einen neuen Thread zur Abarbeitung des Requests erzeugt und dann sofort wieder auf den nächsten Request wartet.

4.3 Programmierkonzepte für Threads

4.3.1 Threads in Java

Die Sprache Java sowie die zugehörige Java Virtual Machine (JVM) unterstützen Threads durch eine eigene in dem Package *java.lang* vorgegebene Basisklasse namens *Threads* sowie durch ein Interface namens *Runnable*. Wenn eine JVM gestartet wird³, werden zunächst ein Main-Thread sowie einige Verwaltungs-Threads wie z.B. der Garbage-Collector-Thread gestartet. Jeder Thread kann weitere Threads erzeugen. Die Klasse *Thread*⁴ ist grob wie folgt definiert:

```
class Thread {
    public static final int MIN_PRIORITY; // Thread-Prioritäten
    public static final int NORM_PRIORITY;
    public static final int MAX_PRIORITY;
    Thread(); // Konstruktor 1
    Thread (Runnable target); // Konstruktor 2
    ...
    // Weitere Konstruktoren
    ...
    // Thread starten, Startmethode wird aktiviert
    public void start();
    // Auf Ende des Threads oder maximal msec
    // Millisekunden warten
    public void join(int msec) throws InterruptedException;
    // Wie oben ohne Zeitbegrenzung
    public void join() throws InterruptedException;
    // Methode, die die Arbeit des Threads ausführt
    public void run() throws InterruptedException;
    // Thread msec Millisekunden anhalten
}
```

³ Hierzu verwendet man das Kommando `java`.

⁴ Im Gegensatz zu C# ist in Java ein Element standardmäßig öffentlich. Das Schlüsselwort *public* muss also nicht angegeben werden.

```

public static void sleep(long millis) throws InterruptedException;
// Thread unterbrechen,
// eine Ausnahme vom Typ InterruptedException wird geworfen
void interrupt();
...
// Test, ob ein Thread noch lebt
public final boolean isAlive();
// Thread-Priorität setzen
public final void setPriority(int newPriority);
// Thread-Priorität ermitteln
public final int getPriority();
// Stack des aktuellen Threads ausgeben
public void dumpStack();
// Namen des Threads setzen
public final void setName(String name);
// Namen des Threads ermitteln
public final String getName();
}

```

Wie in Abbildung 4-8 sichtbar ist, gibt es zwei Möglichkeiten, einen Thread zu erzeugen. Die erste Variante nutzt die Klasse *Thread*. Eine eigene Threadklasse wird von der Klasse *Thread* abgeleitet und die Methode *run* wird überschrieben. Ein Objekt dieser Klasse kann dann instanziiert werden, und mit der in der Klasse *Thread* vordefinierten Methode *start* wird der Thread gestartet. Diese Methode wird von der JVM beim Start des Threads aufgerufen.

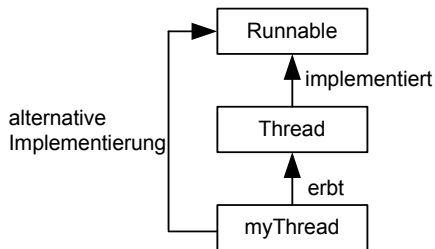


Abbildung 4-8: Klassendiagramm für Java-Threads

Ein nach der ersten Variante formulierter Thread könnte beispielsweise wie folgt aussehen:

Zunächst die eigene Thread-Klasse:

```

import java.lang.Thread;
class myThread extends Thread // Meine Thread-Klasse
{
    String messageText;
    public myThread(String messageText)
    {
        this.messageText = messageText;
    }
}

```



```
// Methode, welche die eigentliche Aktion
public void run()
{
    for (;;) {
        System.out.println("Thread " + getName() + ": " + messageText);
        try {
            sleep(2000);
        }
        catch (Exception e) { /* Ausnahmebehandlung */}
    }
}
```

Der hier angegebene Thread mit dem Namen *myThread* stellt eine Methode *run* bereit. Diese Methode wird beim Start des Threads aufgerufen und wickelt die eigentliche Arbeit ab. Der eigene Thread erbt von der Klasse *Thread*, die im Package *java.lang.Thread*⁵ definiert ist. Der eigene Thread macht nichts anderes, als seinen Namen und einen frei vergebbaren Text auf dem Bildschirm auszugeben und zwar endlos.

Im Folgenden ist noch eine kleine Testklasse für die eigene Thread-Klasse dargestellt:

```
import java.lang.Thread;
...
public class myThread
{
    myThread() { ... } // Konstruktor
    public static void main(String args[])
    {
        myThread t1;
        // Thread erzeugen
        t1 = new myThread("...auf und nieder immer wieder...");
        // Dem Thread einen Namen geben
        t1.setName("Thread 1");
        t1.start(); // Thread starten
        if (t1.isAlive()) {
            try {
                t1.join(10000);
            }
            catch (InterruptedException e) { /* Ausnahmebehandlung */ }
        }
        ...
        System.out.println("Thread " + t1.getName() + " beendet");
    }
}
```

⁵ Packages sind in Java eine Strukturierungsmöglichkeit für zusammengehörige Komponenten.

Das Testprogramm erzeugt einen Thread vom Typ *myThread* und startet ihn mit der Methode *start*. Bei Aufruf der Methode *start* wird implizit die Methode *run* des Threads ausgeführt. Solange der Thread am Leben ist, wird die Methode *run* ausgeführt. Mit *join* kann man auf das Ende eines Threads warten. Im Beispiel wartet der sog. Main-Thread, der beim Start eines Java-Programms von der JVM gestartet wird, auf das Ende des gestarteten Threads vom Typ *myThread*. Mit der Methode *isAlive* kann geprüft werden, ob der Thread am Leben ist. In dem Beispiel wird von dem neuen Thread endlos ein Text ausgegeben. Der Thread wird nie beendet, bis der ganze Prozess (also die laufende JVM) abgebrochen wird.

Die andere Variante, einen Thread zu erzeugen, ist die Definition einer Klasse, die das Interface *Runnable* implementiert. In der Klasse wird ebenfalls die Methode *run* implementiert. Um eine Objektinstanz der Klasse zu erzeugen, wird im Beispiel der gegebene Konstruktor *Thread(Runnable target)* verwendet. Als *Runnable*-Objekt wird die Klasse angegeben, welche das Interface *Runnable* implementiert. Der Grundrahmen für die Definition eines Threads nach dieser Methode sieht wie folgt aus:

```
class myThread implements Runnable {
    public myThread(...) {
        ...
    }
    ...
    public void run() {
        // Die eigentliche Arbeit des Threads
    }
}
```

Die Instanzierung eines Threads vom Typ *myThread* nutzt dann z.B. den zweiten Konstruktor der Klasse *Thread*:

```
...
myThread r = new myThread();
Thread t1 = new Thread(r);
t1.start();
...
```

Beide Varianten sind semantisch vollkommen identisch. Die zweite Methode (über *Runnable*) hat den Vorteil, dass die eigene Klasse noch von einer anderen Klasse abgeleitet werden kann. Bei der ersten Möglichkeit ist aufgrund der fehlenden Mehrfachvererbung in Java keine weitere Vererbung mehr möglich.

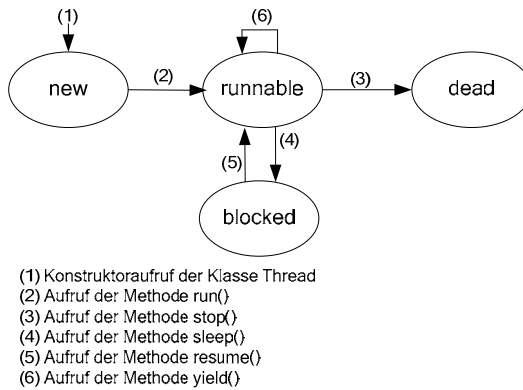


Abbildung 4-9: Zustandsautomat eines Java-Threads

Jeder Thread hat auch eine Priorität (MIN, NORM, MAX). Wird ein Thread von einem anderen Thread erzeugt, so erhält er zunächst die Priorität des Eltern-Threads. Die Priorität kann mit der Methode *setPriority* verändert werden. Die tatsächliche Priorisierung von Java-Threads hängt aber davon ab, wie die JVM im konkreten Betriebssystem implementiert ist. Eine JVM-Implementierung kann die Threadverwaltung z.B. auf evtl. vorhandene Betriebssystem-Threads abbilden oder diese selbst realisieren.

Der Zustandsautomat eines Java-Threads ist recht einfach (siehe Abbildung 4-9). Ein Thread wird mit Aufruf des Konstruktors (new) instanziiert. Die Zustandsübergänge werden durch Methodenaufrufe initiiert. Ein Aufruf der Methode *yield* bewirkt z.B., dass dem laufenden Thread die CPU entzogen wird und ein anderer Thread, der im Zustand *runnable* ist, die CPU zugeteilt bekommt.

4.3.2 Threads in C#

Das Microsoft .NET-Framework ist eine Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen. Sie enthält u.a. die virtuelle Maschine namens CLR (Common Language Runtime) und eine Framework-Klassen-Bibliothek (FCL = Framework Class Library) mit vielen vorgegebenen Typen (Klassen,...).

Die CLR entspricht in Java etwa der JVM und die FCL der J2SE (Java 2 Standard Edition).⁶ Auch in der .NET-Sprache C# werden Threads unterstützt. Hierzu wird den Namensraum (Namespace)⁷ *System.Threading* bereitgestellt.

⁶ Mehr zum .NET-Framework ist u.a. in (Prosis 2002) und (Schwichtenberg 2007) zu finden.

Der Namespace ist grob wie folgt aufgebaut:

```
namespace System.Threading
{
    public delegate void ThreadStart();
    public enum ThreadState {
        Running=0, ...,
        Stopped=16,
        Suspended=64,
        Aborted=256 }
    ...
    public sealed8 class Thread { ... }
    public sealed class Monitor { ... }
    public class ThreadStateException { ... }
    public class ThreadAbortException { ... }
    public class ThreadInterruptedException { ... }
    public class SynchronizationLockException { ... }
}
```

Der Namespace umfasst also neben der eigentlichen Thread-Definition einige Exception-Klassen, eine Aufzählung (Enumeration) von möglichen Thread-Zuständen sowie ein Delegate namens *ThreadStart*. Unter einem Delegate versteht man in C# nichts anderes als eine Referenz auf eine Methode. Dies ist in diesem Zusammenhang die Startmethode eines Threads und muss individuell programmiert und beim Erzeugen eines Threads übergeben werden. Die Klasse *Monitor* dient der Synchronisation und wird in Kapitel 4 noch behandelt.

Ein C#-Thread hat folgende Definition:

```
public sealed class Thread
{
    public Thread(ThreadStart start);
    // Thread starten, Startmethode wird aktiviert
    public void Start();
    // Auf Ende des Threads oder maximal msec Millisekunden warten
    public bool Join(int msec);
    // Thread msec Millisekunden anhalten
    public static void Sleep(int msec);
    // Auslösen einer Ausnahme vom Typ ThreadAbortException
}
```

⁷ Im Unterschied zu Java wird bei C# nochmals zwischen der physikalischen und der logischen Organisation von Typen unterschieden. Während ein Package in Java auch die physikalische Organisation im Filesystem vorgibt, wird die physikalische Gruppierung in C# durch sog. Assemblies (Module) gebildet. In einer Assembly können mehrere Typen (Klassen, Strukturen, Schnittstellen,...) enthalten sein. Ein typisches Assembly ist eine .EXE- oder eine .DLL-Datei. Vgl. hierzu auch (Solymosi 2001).

⁸ Das Schlüsselwort *sealed* entspricht *final* in Java und bedeutet, dass die Klasse nicht vererbbar ist.

```
public void Abort();
// Abort zurücknehmen
public void ResetAbort();
// Thread unterbrechen, eine Ausnahme vom Typ
// ThreadInterruptedException wird geworfen
public void Interrupt();
// Thread suspendieren
public void Suspend();
// Thread wieder anstarten (nach einer Suspension)
public void Resume();
public string Name {get; set;};9 // Name des Threads
// Priorität des Threads
public ThreadPriority Priority {get; set;};
...
}
```

C#-Threads bzw. Threads, die in der CLR, also im Laufzeitsystem erzeugt werden, werden auf Windows-Threads abgebildet. Alle C#-Threads eines Prozesses haben Zugriff auf denselben Adressraum und damit Zugriff auf alle global deklarierten Variablen. Hierzu gehören alle statischen Variablen einer Klasse (siehe Schlüsselwort *static*) sowie die Instanzvariablen eines Objekts. Auch Objekte, die im Heap erzeugt werden, sind für alle Threads eines Prozesses zugreifbar. Jeder Thread erhält einen eigenen Stackbereich mit seinen lokalen Variablen, die vor anderen Threads geschützt sind. Nebenläufige Zugriffe auf gemeinsam verwendete Variablen sind zu synchronisieren.

In C# wird ein Thread wie folgt erzeugt:

```
using System.Threading;
...
class myThread
{
    public void myThread() { .. } // Konstruktor
    public static void Main() {
    {
        // Startmethode festlegen
        ThreadStart startMethod = new ThreadStart(Run);
        // Neuen Thread erzeugen
        Thread myThread = new Thread(startMethod);
        // Thread erhält einen Namen
        myThread.Name = "myThread";
        // Neuer Thread wird gestartet
        myThread.Start();
        // Erzeugender Thread macht etwas anderes
        //Warten, bis sich neuer Thread beendet hat
        myThread.Join();
    }
}
```

⁹ So definiert man in C# Getter/Setter-Methoden. Das Attribut *Name* kann dann einfach direkt gelesen oder verändert werden.

```

    }
    public void Run() // Startmethode des Threads
    {
        // Aktionen des Threads müssen hier programmiert werden
    }
}

```

Im Gegensatz zu Java ist in C# nicht unbedingt eine eigene Klasse zu definieren, um einen Thread zu erzeugen, wie dies im oben angegebenen Beispielcode und im Klassendiagramm aus Abbildung 4-10 gezeigt wird. Es genügt auch die Definition einer Startmethode, die dann bei der Thread-Erzeugung angegeben wird. Der Name der Startmethode (im Beispiel *Run*¹⁰) ist frei vergebbar¹¹.

Wie im Beispiel ersichtlich, wird der Thread durch Aufruf der Methode *Start* erzeugt. Die angegebene Startmethode wird dann in einem neuen Thread ausgeführt. Wenn die Methode *Run* endet, ist auch der Thread zu Ende.

Der Thread, welcher Zugriff auf das Threadobjekt hat, kann z.B. auf die Beendigung des neuen Threads warten (Methode *Join*), ihn unterbrechen (Methode *Interrupt*) oder ihn zur Beendigung auffordern (Methode *Abort*). Der Abbruch eines Threads ist aber mit Vorsicht zu genießen und erfordert eine saubere Exception-Handling-Programmierung im Thread-Code. Alle Ressourcen müssen sauber freigegeben werden. Auch kann ein Thread in der Ausnahmebehandlung seinen Abbruch vereiteln, in dem er die Methode *ResetAbort* ausführt (siehe Prosise 2002).

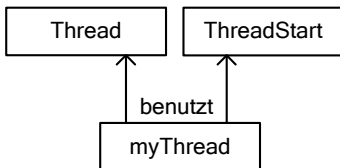


Abbildung 4-10: Klassendiagramm für C#-Threads

Jeder C#-Thread hat innerhalb der CLR eine eigene Priorität im Attribut *Priority* zugeordnet. Der Typ *ThreadPriority* definiert die Gruppe aller möglichen Werte für eine Threadpriorität und gibt die relative Priorität eines Threads gegenüber anderen Threads an. Einem Thread kann einer der folgenden Prioritätswerte zugewiesen werden:

¹⁰ Per Konvention werden Methoden in C# im Gegensatz zur Java-Konvention groß geschrieben.

¹¹ In Java muss die Methode den Namen *run()* haben.

- Highest
- AboveNormal
- Normal
- BelowNormal
- Lowest

Die Priorität kann durch Setzen des Attributes *Priority* zur Laufzeit verändert werden.

Beispiel: `Thread.Priority = ThreadPriority.AboveNormal;`

Dies wirkt sich aber nicht auf den Threadzustand aus, der *Running* sein muss, bevor er vom Betriebssystem geplant werden kann. Die anfängliche Priorität ist die *Normal*-Priorität. Threads werden auf der Grundlage ihrer Priorität für die Ausführung geplant (mehr hierzu in Kapitel 5). Der Planungsalgorithmus, mit dem die Reihenfolge der Threadausführung bestimmt wird, ist bei jedem Betriebssystem unterschiedlich. Das Betriebssystem kann außerdem die Threadpriorität dynamisch anpassen. Die Veränderung der Priorität bewirkt also nicht unbedingt bei jedem Betriebssystem das gleiche und ist somit wenig verlässlich.

4.4 Prozesse und Threads in konkreten Betriebssystemen

4.4.1 Prozesse und Threads unter Windows

Der Windows-Kernel (NTOSKRNL.EXE) verwaltet in einer Komponente namens Objekt-Manager sog. Kernel-Objekte. Hierzu gehören Objekte vom Typ Thread, Prozess, Job und viele andere. Hierbei wird wiederum zwischen sog. Kontroll-Objekten, wie das Prozess-Objekt, und Dispatcher-Objekten, wie das Thread-Objekt unterschieden.

Unter Windows unterscheidet man bei der Prozessverwaltung zwischen Prozessen, Threads und Aufträgen (Jobs). Threads sind eindeutig einem Prozess zugeordnet und stellen die eigentliche Scheduling-Einheit des Betriebssystems dar. Jeder Prozess startet zunächst mit einem Start-Thread.

Prozesse können wiederum gemeinsam in Gruppen verwaltet werden, indem sie zu einem Job zusammengefasst werden. Diesen Mechanismus kann man auch als Ersatz für den Prozessbaum unter Unix ansehen, er ist aber etwas mächtiger. Man kann hier eine Gruppe von Prozessen als gemeinsame Ressource einheitlich ansprechen und verwalten. Beispielsweise lässt sich die maximal für eine Prozessgruppe nutzbare Rechenzeit, der maximal belegbare Speicher oder die maximal zulässige Anzahl an parallelen Prozessen für einen Auftrag einstellen.

Die Beziehungen der Job-, Prozess- und Threadobjekte untereinander können wie folgt beschrieben werden: Ein Thread kann nur zu einem Prozess und ein Prozess kann nur zu einem Auftrag gehören. Alle Nachkommen eines Prozesses gehören zum gleichen Auftrag. Ebenso gehören alle Nachkommen von Threads zum gleichen Prozess. Auch in Windows gibt es also einen Vererbungsmechanismus.

Unter Windows gibt es einige besondere Threads, die auch als Systemthreads bezeichnet werden und daher im Kernelmodus ablaufen. Hierzu gehören der Idle-Thread (Leerlaufthread), der nur läuft, wenn sonst nichts läuft, sowie einige Speicherverwaltungsprozesse.

Lebenszyklus eines Threads. Ein Thread durchläuft im Windows-Betriebssystem verschiedene Zustände. Interessanter sind aber die Zustände der Scheduling-Einheit Thread und der zugehörige Zustandsautomat. Die Threadzustände und die Zustandsübergänge werden im Folgenden erläutert, ohne alle Zustandsübergänge detailliert zu betrachten (vgl. Abbildung 4-11). Der Zustandsautomat wurde im Vergleich zu den Windows-Vorgängerversionen in Windows 2003 um den Zustand *Deferred Ready* erweitert. Ebenso wurden einige Zustandsübergänge verändert. Unsere Betrachtung bezieht sich auf den Zustandsautomaten ab Windows 2003:

- *Init*: Dies ist ein interner Zustand, der während der Erstellung eines Threads eingenommen wird.
- *Ready*: Im diesem Zustand wartet ein Thread auf die Ausführung. Bei der CPU-Zuteilung werden Threads berücksichtigt, die in diesem Zustand sind.
- *Running*: Ein gerade aktiver Thread ist in diesem Zustand. Ein Prozessor ist also zugeteilt. Der Thread bleibt so lange in dem Zustand, bis er vom Kernel unterbrochen wird, um einen Thread mit höherer Priorität auszuführen, bis er sich selbst beendet, sich in einen Wartezustand begibt oder aber bis sein Quantum abläuft.
- *Standby*: In diesem Zustand ist ein Thread, der als nächstes zur Ausführung kommt. Pro Prozessor kann dies nur maximal ein Thread sein. Ab Windows 2003 gibt es keinen Zustandswechsel mehr von *Standby* nach *Ready*.
- *Terminate*: In diesem Zustand ist ein Thread bereits beendet und kann vom System entfernt werden.
- *Waiting*: Ein Thread befindet sich im Zustand *Waiting*, wenn er auf das Eintreffen eines Ereignisses, z.B. auf eine Ein- oder Ausgabe wartet.
- *Transition*: Ein Thread wechselt in den Zustand *Transition* (Übergang) wenn er zwar ausführungsbereit ist, aber sein Kernelstack gerade nicht im Hauptspeicher zugreifbar ist (mehr zur Speicherverwaltung ist in Kapitel 7 zu finden). Bevor der Thread aktiviert werden kann, muss der Kernelstack im Hauptspeicher sein.

- *Deferred Ready*: In diesem Zustand ist ein Thread schon einem bestimmten Prozessor zugeteilt, der Thread ist aber noch nicht aktiv. Der Zustand wurde eingeführt, um im Mehrprozessorbetrieb den Zugriff auf die Scheduling-Datenstrukturen im Kernel (Ready-Queue, siehe unten) zu optimieren. Sperrzeiten wurden damit reduziert (Solomon 2005). Wie wir in Kapitel 5 sehen werden, gibt es für Threads im Zustand *Deferred Ready* je Prozessor eine separate Warteschlange.

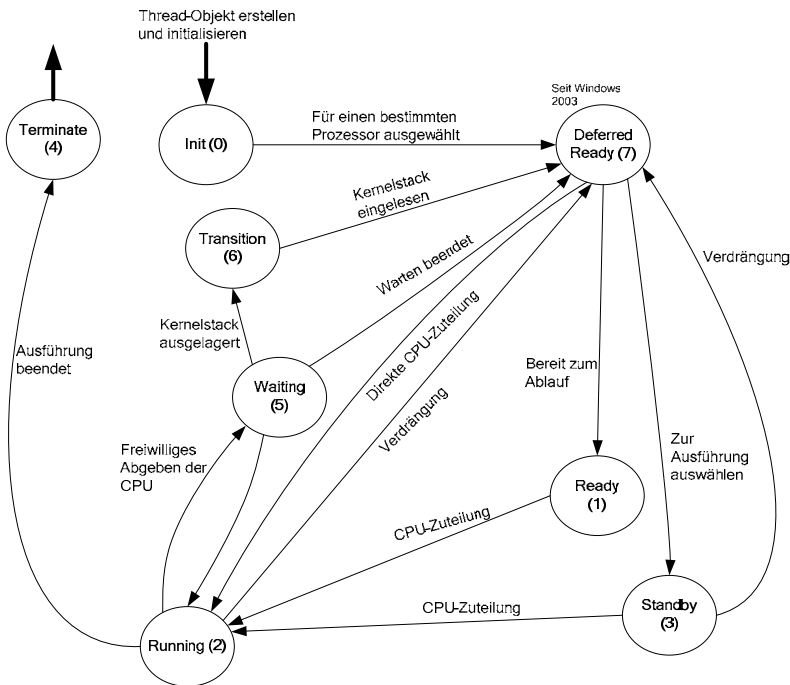


Abbildung 4-11: Zustandsautomat eines Windows-Threads (Solomon 2005)

Gemäß dem Zustandsautomaten gibt es vier Zustandsübergänge, die einen Thread in den Zustand *Running* überführen und die mit einer CPU-Zuteilung einhergehen:

- *Ready* --> *Running*: Direkte Zuordnung eines Threads zu einem Prozessor.
- *Deferred Ready* --> *Running*: Zuordnung des ersten Threads zu einem bereits festgelegten Prozessor aus einer speziellen Warteschlange.
- *Standby* --> *Running*: Dieser Zustandsübergang wird für einen Thread durchgeführt, der schon einem Prozessor zugeordnet ist.

- *Waiting* --> *Running*: Dieser Zustandsübergang kommt vor, wenn ein Thread die CPU abgegeben hat, um auf ein Ereignis zu warten, z.B. weil ein Seitenfehler aufgetreten ist. Nach Eintreffen des Ereignisses kann der Thread sofort wieder auf *Running* gesetzt werden.

Prozesserzeugung unter Windows. Die Prozesserzeugung ist unter Windows relativ komplex und soll am Beispiel des 32-Bit-Windows vereinfacht skizziert werden. Das System verwendet hierfür den Systemcall *CreateProcess* und die Erzeugung des Prozesses erfolgt in verschiedenen Bereichen des Betriebssystems und zwar in der Windows-Ausführungsschicht (für das 32-Bit-Windows die Bibliothek *Kernel32.dll*) und dem Win32-Teilsystemprozess. Folgender Ablauf ist erforderlich:

- Die aufgerufene .exe-Datei wird geöffnet. Hier wird erst einmal festgestellt, ob es eine Win32-Datei ist. Ist es eine andere (z.B. eine POSIX-, eine MS-DOS- oder eine Win16-Anwendung), so müssen noch entsprechende Ablaufumgebungen hergestellt werden. Nur Win32-Anwendungen können direkt geladen werden.
- Das Prozessobjekt wird in der Ausführungsschicht erzeugt. Der erzeugende Prozess erhält ein Objekt-Handle mit zugeordneter PID (Process Identification). Hier werden u.a. der EPROCESS-Block und der Kernelprozessblock erzeugt (siehe unten).
- Ein Start-Thread bzw. das entsprechende Threadobjekt mit Stack und Kontext wird erzeugt.
- Das Win32-Teilsystem wird über eine entsprechende Nachricht informiert und bereitet sich auf den neuen Prozess und den Start-Thread vor.
- Der Adressraum des neuen Prozesses wird initialisiert und der Prozess bzw. der Start-Thread wird gestartet.

Im *Windows-Task-Manager*, einem Windows-Dienstprogramm, kann man sich eine Übersicht über die wesentlichen Prozesse mit speziellen Informationen über die Anzahl der Threads, die für die einzelnen Prozesse erzeugt wurden, die bereits verbrauchte CPU-Zeit der Prozesse usw. ausgeben lassen. Die Tabelle 2-1 zeigt einige Windows-Prozesse, die ständig laufen und mit dem Windows-Task-Manager verfolgt werden können.

Ebenso sind die aktuelle Speichernutzung der Prozesse und die bis dato verursachten Seitenfehler zu sehen. Diese beiden Angaben werden bei der Diskussion der Speicherverwaltung noch erläutert.

Tabelle 2-1: Wichtige Systemprozesse unter Windows

Windows-Prozessname	Aufgabe
csrss.exe	Client-Server-Runtime-Subsystem. Implementiert den Usermodus des Windows-Subsystems und ist auch für die Windows-Console und das Threading verantwortlich.
lsass.exe	Local Security Authentication Server. Dieser Prozess verwaltet die User-Logins und überprüft die Login-Angaben.
mstask.exe	Dieser Prozess verwaltet die geplanten Tasks.
smss.exe	Dieser Prozess verwaltet die User-Sessions.
spoolsv.exe	Dies ist der Spoolserver, der für die Drucker- und Faxeinträge verantwortlich ist.
svchost.exe	Dies ist ein generischer Prozess, den es seit Windows 2000 gibt. Der Prozess sucht beim Systemstart nach Windows-Diensten, die geladen werden müssen und auch nach über DLLs (Dynamic Link Libraries) zu ladende Windows-Dienste. Dienste oder auch zusammengefasste Dienstgruppen werden in eigenen Prozessen jeweils mit dem Namen svchost.exe gestartet. Beim späteren Nachladen eines Dienstes kann ebenfalls ein neuer svchost.exe-Prozess gestartet werden.
services.exe	Dies ist der <i>Service Control Manager</i> , der für das Starten und Stoppen von Systemprozessen und Diensten sowie für die Interaktion mit diesen verantwortlich ist.
Leerlaufprozess	Dies ist der Leerlaufprozess. Für jeden Prozessor läuft ein eigener Thread in diesem Prozess.
winlogon.exe	Dieser Prozess ist verantwortlich für das Ein- und Ausloggen eines Users.
system	Kernelspezifischer Systemprozess. In diesem Prozess laufen z.B. einige Speicherverwaltungs-Threads.

Ein weiterer wichtiger Begriff aus dem Windows-Betriebssystem ist der des Dienstes. Unter einem *Windows-Dienst* versteht man einen Prozess, der nicht an einen interaktiven Benutzer gebunden ist, also vergleichbar mit einem Unix-Dämonprozess. Typische Beispiele für Dienste unter Windows sind der Windows-Zeitgeberdienst, der Plug&Play-Dienst, ein Webserver-Prozess, ein Datenbankserver-Prozess, die Druckwarteschlange oder ein Fax-Server. Dienste können zum Sys-

temstart automatisch erzeugt werden. Dienst Anwendungen müssen eine spezielle Schnittstelle unterstützen, also als Dienst programmiert sein. Über diese Schnittstelle kann ein sog. Service Control Manager (SCM), der Dienststeuerungsmanager, mit der Dienst Anwendung zum Starten, Stoppen des Dienstes usw. kommunizieren. Dienstprogramme sind meist als Konsolenprogramme ohne grafische Oberfläche konzipiert. Sie kommen in einer Prozess-Instanz mit der Bezeichnung *svchost.exe* zum Ablauf.

Für Prozesse, Threads und Jobs verwaltet das Betriebssystem verschiedene Datenstrukturen, die im Weiteren kurz erläutert werden sollen. Die Übersicht in Abbildung 4-12 zeigt, dass einige Datenstrukturen im Prozessadressraum und andere im Kerneladressraum verwaltet werden. Um ein Gefühl für den Umfang der Systemdaten, die unter Windows für Prozesse und Threads verwaltet werden, zu bekommen, sollen im Folgenden einige Datenstrukturen herausgegriffen und (ohne Anspruch auf Vollständigkeit) diskutiert werden.

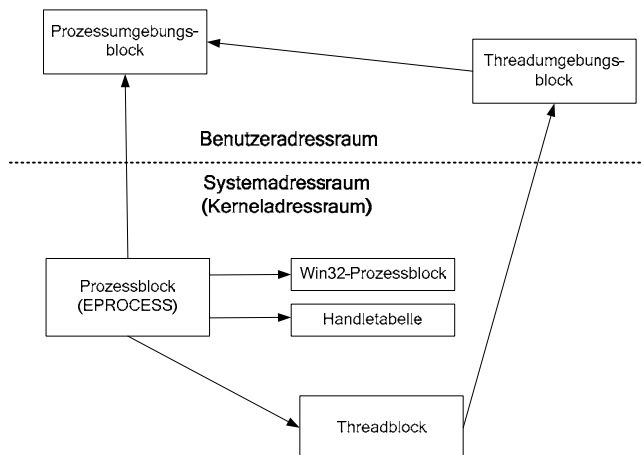


Abbildung 4-12: Datenstrukturen des Betriebssystems, Überblick (Solomon 2005)

Datenstrukturen für die Prozess-Verwaltung. Im Systemadressraum wird der Prozessblock (Struktur EPROCESS-Block, vgl. Abbildung 4-13) angelegt, der den Prozess repräsentiert. Über ihn sind alle anderen Informationen zu dem Prozess, wie die zugehörigen Threads (siehe Verweis auf Threadblock) und auch ein Verweis auf den nächsten EPROCESS-Block in einer zugehörigen verketteten Verwaltungsliste zu finden. Wichtige Informationen im EPROCESS-Block sind u.a. die PID des Prozesses und des übergeordneten Prozesses, der Verweis auf einen Prozessumgebungsblock, die Beschreibung der zugeordneten Teile des Adressraums und ein Verweis auf eine Handletabelle mit allen Objekt-Handles, die dem Prozess

zugeordnet sind. Weiterhin ist ein Verweis auf das Auftragsobjekt, zu dem der Prozess gehört, im EPROCESS-Block enthalten.

Der EPROCESS-Block enthält auch den Kernelprozessblock (Struktur KPROCESS, vgl. Abbildung 4-13). Dieser enthält zusätzliche Informationen zum Prozess und wird unter Windows auch als *PCB* bezeichnet. Informationen, die hier verwaltet werden, sind u.a. die Kernelzeit und die Benutzerzeit, die der Prozess bisher verbraucht hat, ein Prozess-Spinlock¹², der aktuelle Prozesszustand gemäß Zustandsautomat und das Standard-Threadquantum (Zeitscheibe für den Thread).

Der Prozessumgebungsblock (auch PEB) im prozesseigenen Adressraum und speichert Informationen zum geladenen Programm, z.B. zum Heap, zur Heap-Größe, zur Adresse des ausführbaren Codes usw.

Datenstrukturen für die Thread-Verwaltung. Datenstrukturen für die Threadverwaltung sind der ETHREAD-Block bzw. Executive-Thread-Block und der Kernel-Thread-Block (KTHREAD-Block). Der ETHREAD- und der KTHREAD-Block befinden sich im Kerneladressraum und repräsentieren einen Thread auf Betriebssystemebene. Der ETHREAD-Block enthält u.a. Informationen zur Prozesszugehörigkeit (PID sowie ein Verweis auf den EPROCESS-Block), die Startadresse des Threads und einen Verweis auf eine Liste ausstehender Ein-/Ausgabeanforderungen. Der KTHREAD-Block enthält Informationen, auf die der Kernel zur Planung und Synchronisierung aktiver Threads zugreifen muss. Beispielsweise sind hier folgende Informationen enthalten:

- Adresse des Kernelstacks
- Basispriorität und aktuelle Priorität (mehr dazu in Kapitel 5)
- Quantum
- Einreihung in Warteschlangen des Schedulers
- Verweis auf den Thread-Umgebungsblock (TEB)

ETHREAD- und KTHREAD-Block sind in Abbildung 4-12 vereinfacht als Threadblock dargestellt.

¹² Spinlock ist ein einfacher Sperrmechanismus, der für den geordneten Zugriff auf bestimmte Prozessdaten benutzt wird (mehr dazu in Kapitel 6).

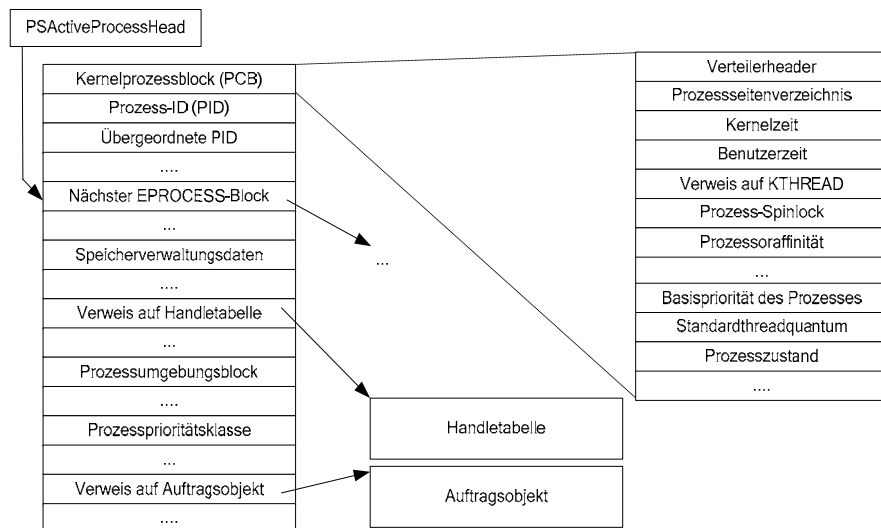


Abbildung 4-13: Datenstruktur EPROCESS (Solomon 2005)

Der TEB liegt wie der PEB im prozesseigenen Adressraum und speichert Kontextdaten für das zu ladende Programm. Im TEB sind u.a. folgende Informationen zu finden:

- Thread-ID
- Stackinformationen (Anfangsadresse, Stackgrenze)
- Informationen zu kritischen Abschnitten, die der Thread gerade besitzt (siehe Kapitel 6)
- Informationen zu angefallenen Exceptions

Im Kernel werden einige für die Prozessverwaltung wichtige globale Variablen (Kernelvariablen und Leistungsindikatoren) verwaltet, die den Zugang auf Prozessdaten ermöglichen sowie einige Informationen zum Leistungsverhalten liefern. Die Variable `PActiveProcessHead` verweist z.B. auf den Kopf der EPROCESS-Liste, die Variable `PIdleProcess` verweist auf den EPROCESS-Block des sog. Leerlaufprozesses.

Unter Windows gibt es noch sog. *schlanke Threads*, die als *Fibers* bezeichnet werden. Man kann mit Win32-API-Funktionen einen Thread zu einem Fiber konvertieren, ihn damit aus dem Betriebssystem-Scheduling herausnehmen und in der Anwendung seine eigene Ausführungsstrategie implementieren. Man kann auch Fiber-Gruppen organisieren. Fibers laufen im Benutzermodus und sind für den CPU-Scheduler nicht sichtbar.

4.4.2 Prozesse und Threads unter Unix und Linux

Unix organisiert die Prozesse in einer baumartigen Prozessstruktur, wobei der sog. *init*-Prozess der Urvater aller zeitlich folgenden Prozesse ist. Ein Prozess wird unter Unix mit einem Systemcall *fork* erzeugt und erhält dabei eine Prozess-Identifikation (PID). Der Prozess *init* hat die PID 1, ein ausgezeichneter Speicher-verwaltungs-Prozess namens *swapper* erhält die PID 0 und ein anderer Speicher-verwaltungsprozess *page* die PID 2.

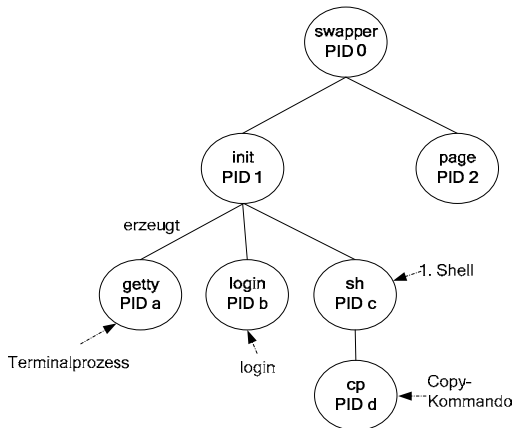


Abbildung 4-14: Prozessbaum unter Unix (Tanenbaum 2002)

Es sei angemerkt, dass dies je nach Unix-Derivat etwas unterschiedlich sein kann (System V, Berkeley Unix, Linux, Sun Solaris), weshalb hier nur allgemein über die Prozessverwaltung unter Unix diskutiert werden kann.

Bei der Erzeugung eines Prozesses erbt der „Kindprozess“ vom „Elternprozess“ die gesamte Umgebung (inkl. Umgebungsvariablen), alle offenen Dateien und Netzwerkverbindungen. Der Kindprozess erhält eine Kopie des Adressraums des Elternprozesses mit den Daten- und Codebereichen. Er kann dann das gleiche Programm ausführen oder lädt sich bei Bedarf durch Aufruf des Systemcalls *execve* ein neues Programm und überlädt somit das Programm des Elternprozesses.

Unter Linux gibt es neben dem klassischen *fork*-Aufruf noch zwei weitere Systemaufrufe zur Prozesserzeugung. Mit dem Systemaufruf *vfork* wird eine schnelle Möglichkeit der Prozesserzeugung ermöglicht. Die Prozessdaten werden nämlich in diesem Fall nicht kopiert, was CPU-Zeit einspart. Diese Art der Prozesserzeugung ist sinnvoll, wenn anschließend ohnehin mit *execve* ein neues Programm geladen wird. Bei dem Systemaufruf *clone* kann angegeben werden, welche Daten-

elemente zwischen Eltern- und Kindprozess geteilt und welche kopiert werden sollen.

Speziell für Unix-Betriebssysteme gibt es eine Spezifikation für Threads, die in einem API-Standard namens POSIX von der IEEE (Portable Operating System Interface for Unix) beschrieben ist und nahezu von allen Unix-Derivaten unterstützt wird. Die POSIX-Threads werden entweder auf Benutzer- oder auf Kernel-ebene realisiert.

Betrachtet man den Unix-Prozessbaum in Abbildung 4-14, so erkennt man einen *getty*-Prozess (*getty* ist eine Abkürzung für *get terminal*) der für jedes (in der Konfigurationsdatei */etc/ttys*) konfigurierte Terminal vom System beim Hochfahren und nach einem Logout gestartet wird. Dieser Prozess verwaltet ein virtuelles oder physikalisches Terminal, wartet auf das Einschalten des Terminals und erzeugt über den *init*-Prozess einen Login-Bildschirm. Nach erfolgreichem Login wird zunächst eine im Benutzerprofil (z.B. aus Datei */etc/passwd*) vorkonfigurierte Shell gestartet. In dieser Shell können dann beliebige Kommandos oder Programme aufgerufen werden. Bei jeder Abarbeitung eines Kommandos wird wieder ein eigener Prozess erzeugt. Unix ist also ziemlich stark mit dem Auf- und Abbau neuer Prozesse beschäftigt.

Abbildung 4-15 zeigt eine typische Prozesshierarchie eines Linux-Systems. Jeder Prozess, außer *init*, hat einen Elternprozess (auch Vaterprozess genannt). Zu den Prozessen mit den Namen *getty* und *login* sind hier beispielhaft einige Anwendungsprozesse aus dem X-Windows-System¹³ dargestellt (siehe *xinit*, ...).

Weiterhin sind die Prozesse *syslogd* und *inetd* unmittelbare „Kinder“ von *init*. *inetd* ist ein in Unix/Linux üblicher Prozess, der auf ankommende Verbindungsaufbauwünsche für einige Serverprozesse wartet und diese dann weitergibt. Der Prozess *syslogd* protokolliert bestimmte Ereignisse, die man vom Anwendungsprogramm aus auch über eine Schnittstelle an ihn senden kann, in eine Loggingdatei.

Die meisten Unix-Prozesse, deren Namen mit einem „d“ enden, sind sog. Dämonprozesse. Dies sind Hintergrundprozesse, die beim Start vom Terminal abgekoppelt werden. Sie werden von der Unix-Shell aus gestartet, indem beim Start an den Namen des Prozesses das Zeichen „&“ angehängt wird. Nur am Rande sei

¹³ Das X-Windows-System ist unter Unix/Linux das System, das die graphische Oberfläche bereitstellt. Es implementiert das X-Display-Protokoll zur Kommunikation zwischen den beteiligten Komponenten. Desktop-Umgebungen wie GNOME und KDE nutzen das X-Windows-System.

erwähnt, dass der Programmierer eines Programms, das als Dämonprozess ablaufen soll, bei der Initialisierung das Signal *SIGHUP* deaktivieren muss, damit der Prozess vom Terminal unabhängig wird.

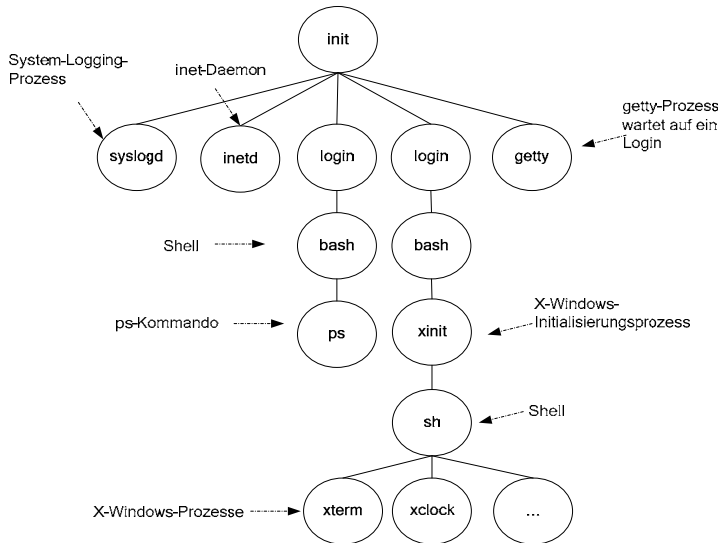


Abbildung 4-15: Prozesshierarchie unter Linux

Lebenszyklus eines Prozesses

In Abbildung 4-16 ist der Lebenszyklus eines Unix-Prozesses etwas vereinfacht in Form eines Zustandsdiagramms dargestellt (siehe Brause 2001). Neben fünf klassischen Zuständen enthält der Lebenszyklus eines Prozesses auch Übergangs- bzw. Zwischenzustände. Der Zustandsübergang vom Zustand *nicht existent* in den Zustand *bereit* wird z.B. über den Zwischenzustand *idle* ausgeführt. Eine Besonderheit stellt der Zwischenzustand *zombie*¹⁴ dar. In diesen Zustand gelangt ein Prozess, der terminieren will. Er verweilt solange in diesem Zustand, bis der Elternprozess eine Nachricht über das Ableben des Kindprozesses erhalten hat und terminiert erst dann.

Der Elternprozess muss über einen Systemaufruf *wait* auf alle seine Kindprozesse warten. Über diesen Systemdienst erhält der Elternprozess die Information über das Ende des Kindprozesses. Erst wenn der Elternprozess *wait* aufgerufen hat, kann der Kindprozess vom Zustand *zombie* in den Zustand *nicht existent* überführt und damit tatsächlich gelöscht werden. Im Zustand *zombie* hat der Kindprozess

¹⁴ „Zombie“ bedeutet soviel wie tot, aber irgendwie doch noch am Leben.

zwar keine Ressourcen mehr belegt, aber er bleibt in der Prozesstabelle. Bei fehlerhafter Programmierung des Elternprozesses kann es sein, dass der *wait*-Aufruf (blockierender Aufruf) nicht ausgeführt wird. Kindprozesse, bei denen der Elternprozess vor Ablauf terminiert, verwaisen und werden dem *init*-Prozess zugeordnet.

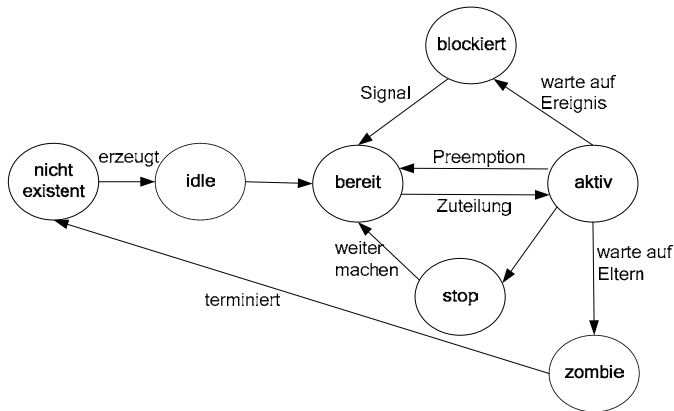


Abbildung 4-16: Vereinfachter Zustandsautomat eines Unix-Prozesses

Beispiel: Der Vorgang der Prozesserzeugung und -terminierung soll noch etwas genauer anhand eines Codebeispiels betrachtet werden:

```

int main()
{
    int ret;          // Returncode von fork
    int status;       // Status des Sohnprozesses
    pid_t pid;        // pid_t ist ein spezieller Datentyp, der eine
                    // PID beschreibt
    ret = fork(); // Sohnprozess wird erzeugt
    if (ret == 0) {
        // Anweisungen, die im Kindprozess ausgeführt werden sollen
        ...
        exit(0);      // Beenden des Sohnprozesses mit Status 0 (ok)
    }
    else {
        // Anweisungen, die im Vaterprozess ausgeführt werden
        // Zur Ablaufzeit kommt hier nur der Elternprozess rein
        //(Returncode = PID des Kindprozesses)
        ...
        // Warten auf das Ende des Kindprozesses
        pid = wait(&status);
        exit(0);      // Beenden des Vaterprozesses mit Status 0 (ok)
    }
}

```

Im Codebeispiel wird deutlich, wie der neue Prozess so programmiert werden kann, dass er etwas anderes tut als der Elternprozess. Der Programmierer unterscheidet die Codeteile für den Eltern- und den Kindprozess am Returncode des *fork*-Aufrufs. Der Kindprozess erhält vom System den Returnwert 0, während der Elternprozess die PID des Kindprozesses erhält. Die Abbildung 4-17 zeigt den prinzipiellen Ablauf einer Prozessverdoppelung unter Unix. Der Prozessadressraum des Elternprozesses wird kopiert und die Kopie dem Kindprozess zur Verfügung gestellt. Damit wird eine Code- und Datenverdoppelung durchgeführt, und die beiden Prozesse arbeiten zunächst mit identischen Kopien weiter. Die Synchronisation der beiden Prozesse erfolgt, indem der Elternprozess einen *wait*-Aufruf absetzt und damit auf das Ende des Kindprozesses wartet.

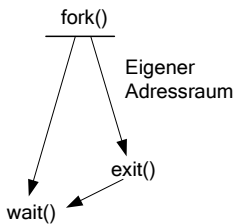


Abbildung 4-17: Prozessverdoppelung unter Unix

Der Kindprozess erbt vom Vaterprozess auch die offenen Dateien, so dass beide auf dieselben Ressourcen zugreifen können. Der Zugriff muss dann aber synchronisiert werden.

Für die Verwaltung der Prozessinformation nutzt Unix Tabellen mit speziellen Datenstrukturen, deren Definitionen in den entsprechenden System-Headerfiles (C-Includes *.h) eingesehen werden können. Im Linux-Kernel wird beispielsweise für jeden Prozess-Kontext (also PCB) eine Struktur *task_struct* verwaltet, die in einer Headerdatei mit Namen *sched.h* definiert ist. In dieser Struktur liegen u.a. folgende Informationen:

- Prozessstatus
- Swap-Status (siehe hierzu die Ausführungen zum Speichermanagement)
- Prozesspriorität
- Anstehende Signale, die für den Prozess bestimmt sind (z.B. Ein-/Ausgabe-Signale)
- Verweise auf Programmcode, Stack und Daten, die dem Prozess zugeordnet sind

Die verwendeten Datenstrukturen sind je nach Unix-Derivat verschieden.

4.5 Übungsaufgaben

1. Was ist in der Prozessverwaltung ein PCB, wozu dient er und welche Inhalte hat er? Nennen Sie dabei drei wichtige Informationen, die im PCB verwaltet werden!
2. Threads werden heute von den meisten Betriebssystemen unterstützt. Was versteht man unter einem Thread?
3. Wie verhalten sich Threads zu Prozessen im Hinblick auf die Nutzung des Prozessadressraums?
4. Beschreiben Sie den groben Ablauf eines Prozess-Kontextwechsels und erläutern Sie, warum ein Thread-Kontextwechsel schneller sein kann als ein Prozess-Kontextwechsel!
5. Was versteht man unter User-Level-Threads im Vergleich zu Kernel-Level-Threads und welche Beziehungen zwischen beiden sind möglich?
6. Was bedeutet eine 1:n-Beziehung zwischen den Betriebsmitteln Prozess und Thread?
7. Erläutern Sie die beiden Möglichkeiten in Java, eigene Threads zu definieren und zu nutzen?
8. In welcher Methode wird die eigentliche Arbeit eines Java-Threads ausgeführt?
9. Was passiert beim Aufruf des Systemcalls *fork* unter Unix?
10. Welche Aufgabe hat ein Thread unter Windows?
11. Wie kann man in C# eigene Threads definieren und nutzen? Welche Bedeutung hat in diesem Zusammenhang die sog. Start-Methode?
12. Kann es unter Windows sein, dass ein Thread mehreren Prozessen zugeordnet ist? Begründen Sie Ihre Entscheidung!
13. Warum ist der Einsatz von Threads sinnvoll?
14. Welche zwei grundsätzlichen Implementierungsmöglichkeiten für Threads gibt es und welche Vor- bzw. Nachteile haben diese jeweils?
15. Beschreiben Sie einen einfachen Zustandsautomaten eines Prozesses!
16. Erläutern Sie die Prozesshierarchie unter Unix!
17. Was ist ein „Zombie-Prozess“ unter Unix?
18. Skizzieren Sie die möglichen Zustandsübergänge eines Windows-Threads (ab Windows 2003) in den Zustand Runnig!

5 CPU-Scheduling

Die verfügbare Rechenzeit muss vom Betriebssystem an die parallel ablaufenden bzw. nebenläufigen Aktivitäten (Prozesse und Threads) zugewiesen werden. Bei Einprozessormaschinen wird eine einzige CPU (ein Rechnerkern oder Rechenkern) für mehrere Aktivitäten genutzt. Bei Mehrprozessormaschinen und heutigen Multicore-Prozessoren stehen mehrere CPUs oder Rechnerkerne zur Verfügung. Im letzteren Fall spricht man von echter Parallelität, da so viele Aktivitäten ausgeführt werden können wie CPUs oder Rechnerkerne zur Verfügung stehen.

Für die Zuteilung der CPUs auf Aktivitäten stehen verschiedene Möglichkeiten zur Verfügung. In diesem Kapitel wird auf die Vergabe-Strategien bzw. Vergabe-Algorithmen für das Betriebsmittel „Rechenzeit“ (CPU-Zeit) eingegangen. Bei dieser Aufgabe spricht man im Betriebssysteme-Jargon von CPU-Scheduling oder kurz vom Scheduling (Scheduling = Ablaufplanung). Kriterien und Ziele für das Scheduling werden betrachtet. Verschiedene Scheduling-Verfahren werden im Einzelnen, aufgegliedert nach Prozess-Klassifizierungen wie Batch-, Dialog- und Realtime-Prozesse, erläutert. Grundsätzlich teilt man die Verfahren in nicht-verdrängende (engl. Fachbegriff: non-preemptive) und verdrängende (engl. Fachbegriff: preemptive) Verfahren ein und meint damit, dass ein Verfahren die CPU einem Prozess aktiv entzieht oder nicht. Alte Betriebssysteme wie MS-DOS und erste Windows-Varianten sind z.B. non-preemptive. Heutige Universalbetriebssysteme sind allerdings preemptive. Probleme mit dem Entzug der CPU haben hier Realzeitsysteme.

Es wird gezeigt, dass das optimale Verfahren der sog. Shortest-Job-First-Algorithmus (SJF) ist. Allerdings ist dieses Verfahren schwer zu realisieren und daher in heutigen Betriebssystemen nicht vorzufinden. Das heute wichtigste Verfahren in Universalbetriebssystemen ist das Round-Robin-Verfahren, ergänzt um eine Prioritätensteuerung. Dieses Verfahren wird etwas näher betrachtet, und es werden wichtige Fragen hierzu diskutiert. Eine Gegenüberstellung von Scheduling-Algorithmen soll zeigen, in welcher Reihenfolge Prozesse ausgeführt werden und welche Verweilzeiten sich im Vergleich ergeben.

Die betrachteten Algorithmen für das CPU-Scheduling wie Shortest Job First (SJF), Shortest Remaining Time Next (SRTN), FIFO (First In First Out) und RR (Round Robin) sind in der einen oder anderen Ausprägung meist eher theoretischer Natur. Es werden daher in diesem Kapitel auch spezielle Implementierungen von Schedu-

ling-Verfahren unter Unix, Linux und Windows, aber auch in der Java Virtual Machine auf Basis der erläuterten Algorithmen überblicksartig angerissen.

Zielsetzung des Kapitels

Der Studierende soll die wesentlichen Scheduling-Verfahren verstehen und erläutern können. Insbesondere sollte das Round-Robin-Verfahren mit und ohne Prioritätensteuerung skizziert und mit anderen Algorithmen verglichen werden können. Der Studierende sollte ferner eine Einschätzung über die praktische Nutzung von Scheduling-Verfahren erwerben und aufzeigen können, was ein Betriebssystem prinzipiell tun muss, um das CPU-Scheduling effektiv zu unterstützen. Es sollte erläutert werden können, welche Informationen und Datenstrukturen ein Betriebssystem grundsätzlich verwalten muss, um eine Scheduling-Entscheidung treffen zu können.

Wichtige Begriffe

Antwortzeit, Durchlaufzeit, Verweilzeit, Durchsatz, preemptive und non-preemptive, Round Robin (RR), SJF, FIFO, SRTN, Priority Scheduling, Multi-Level Scheduling, Run-Queue, Quantum (Zeitscheibe), Priorität.

5.1 Scheduling-Kriterien

Die parallel auszuführenden oder nebenläufigen Aktivitäten werden gelegentlich als Auftrag bzw. Job oder als Prozess bezeichnet. Ersterer ist ein etwas allgemeinerer Begriff für eine auszuführende Aktivität, mit Prozess ist bereits der konkrete Betriebssystemprozess gemeint, in dem die Aktivität bzw. das konkrete Programm zum Ablauf kommt. Für die Betrachtung der verschiedenen Algorithmen spielt dies aber keine Rolle.

Prozesse werden vom *Prozessmanager* des Betriebssystems verwaltet. Üblicherweise laufen in einem Universalbetriebssystem wesentlich mehr Prozesse als Prozessoren und daher muss die Zuordnung eines Prozessors (CPU) an einen Prozess bestimmten Regeln unterworfen werden. Die Komponente im Prozessmanager, die für die Planung der Betriebsmittelzuteilung zuständig ist, heißt *Scheduler*. Die Komponente, die dann einen tatsächlichen Prozesswechsel ausführt, wird als *Dispatcher* (Arbeitsverteilung, Disposition) bezeichnet. Man unterscheidet auch zwischen langfristigem, mittelfristigem und kurzfristigem Scheduling (Ablaufplanung). Mit kurzfristigem (short-term) Scheduling meint man das CPU-Scheduling, also die Vergabe der CPUs an Prozesse, mittelfristiges (medium-term) Scheduling befasst sich mit der Vergabe des Speichers und langfristiges (long-term) Scheduling mit der Verwaltung der im Betriebssystem ankommenden Aufgaben.

Sind mehrere Prozesse ablaufbereit (ready), muss entschieden werden, welcher Prozess als nächstes eine Zuteilung der CPU erhält. Wir gehen der Einfachheit halber zunächst von einer verfügbaren CPU aus, die Prinzipien gelten sinngemäß auch für Multiprozessormaschinen.

Bei der Vergabe der CPU werden bestimmte Scheduling-Ziele angestrebt:

- *Fairness*, d.h. für jeden Prozess eine garantierte Mindestzuteilung
- Effizienz durch möglichst volle *Auslastung* der CPU
- Die *Antwortzeit* soll minimiert werden
- Die *Wartezeit* von Prozessen soll minimiert werden
- Der *Durchsatz* soll optimiert werden
- Die *Durchlaufzeit* (Verweilzeit) eines Prozesses soll minimiert werden
- Die Ausführung eines Prozesses soll vorhersehbar und damit kalkulierbar sein.

Diese Scheduling-Ziele widersprechen sich teilweise. Die schnelle Durchlaufzeit eines Prozesses kann z.B. zu langsameren Antwortzeiten in anderen Prozessen führen. Welche Ziele nun priorisiert werden sollen, hängt nicht zuletzt vom Typ des Betriebssystems und der Anwendung ab. Die betrachteten Ziele sind also zum Teil gegensätzlich, was eine Optimierung erschwert.

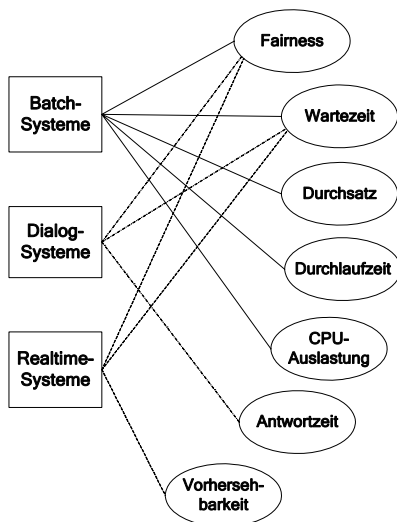


Abbildung 5-1: Typische Scheduling-Ziele für bestimmte Betriebssysteme

Zum Begriff der Fairness sei hier nur darauf hingewiesen, dass man verschiedene Fairness-Varianten kennt, die auch im Rahmen von Scheduling-Strategien diskutiert werden. Man unterscheidet prinzipiell zwischen *weak* und *strong fair*. *weak fair*

bedeutet, dass eine Bedingung auch irgendwann einmal wahr wird. Wann das genau ist, wird allerdings nicht festgelegt. Nach Kredel 1999 ist die Strategie *weak fair* beispielsweise im Java-Scheduler realisiert.

Abbildung 5-1 zeigt, welche Scheduling-Ziele für batch-orientierte, dialog-orientierte und Realtime-Systeme von Bedeutung sind. Für Realtime-Systeme ist z.B. die Vorhersehbarkeit, also wann ein Prozess die CPU erhält, besonders wichtig, um schnell und zeitgerecht auf Ereignisse reagieren zu können. Dies ist bei Dialog- und Batch-Systemen sicherlich von zweitrangiger Bedeutung.

Bei der Zuteilung der CPU unterscheidet man auch zwei grundsätzliche Möglichkeiten:

- Process-based Scheduling: Bei dieser Variante wird die CPU nur Prozessen zugeordnet.
- Thread-based Scheduling: In diesem Fall wird die CPU einzelnen Threads zugeordnet

Die Unterscheidung ist aber für die Betrachtung der verschiedenen Scheduling-Algorithmen nicht relevant, da es sich bei beiden um nebenläufige bzw. parallele Aktivitäten handelt.

Interessant ist auch eine Unterscheidung der Prozesse (Threads) nach CPU-lastigen und Ein-/Ausgabe-intensiven Prozessen. Beide Varianten kommen in Universalbetriebssystemen gewöhnlich nebenläufig vor. Erstere nutzen viel Rechenzeit und warten relativ selten auf Ein-/Ausgabe. Letztere rechnen eher wenig und warten vergleichsweise lange auf die Beendigung von Ein-/Ausgabe-Operationen.

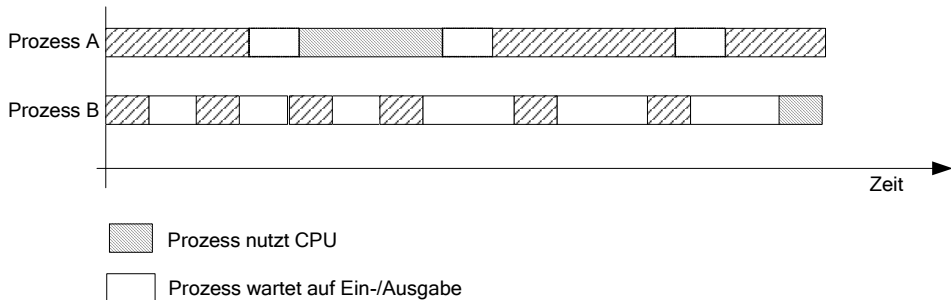


Abbildung 5-2: Ein-/Ausgabe-intensiv versus rechenintensive Prozesse

In Abbildung 5-2 ist z.B. der Prozess B eher Ein-/Ausgabe-intensiv, während der Prozess A überwiegend rechenintensiv ist.

Wir betrachten im Folgenden einige Scheduling-Verfahren, wobei die Unterscheidung zwischen den Begriffen *Prozess* (auch Task) und *Auftrag* (Job) bei der Betrachtung

tung nicht relevant ist. Eine kurze Abgrenzung soll aber vorab noch durchgeführt werden. Der Begriff des Prozesses wurde ja schon oben erläutert. Er stellt eine konkrete Instanzierung eines Programms dar. Ein Auftrag bzw. ein Job wird in Betriebssystemen meist über spezielle Sprachen formuliert (in Mainframes als JCL = Job Control Language, in Unix als Scriptsprache bezeichnet) und benötigt für ihre Ausführung einen Prozess oder sogar mehrere Prozesse. Nicht in allen Betriebssystemen wird dieser Begriff noch verwendet, da er ursprünglich aus der Batchverarbeitung stammt.

5.2 Scheduling-Verfahren

5.2.1 Verdrängende und nicht verdrängende Verfahren

Primär unterscheidet man bei Scheduling-Verfahren zwischen *verdrängendem* (*preemptive*) und *nicht verdrängendem* (*non-preemptive*) Scheduling.

Im non-preemptive, auch „run-to-completion“-Verfahren genannt, darf ein Prozess nicht unterbrochen werden, bis er seine Aufgaben vollständig erledigt hat. Das Verfahren ist natürlich nicht für konkurrierende Benutzer geeignet und auch nicht für Realtime-Anwendungen. MS-DOS und auch die ersten Windows-Systeme unterstützten z.B. nur dieses Verfahren.

Im Gegensatz dazu darf im preemptive Scheduling eine Vorrang-Unterbrechung stattfinden. Rechenbereite Prozesse können somit suspendiert werden. Dies setzt natürlich eine Strategie zur Vergabe der CPU voraus, die vom Betriebssystem unterstützt werden muss und in der Regel auf der Zeitscheibentechnik basiert. Dieses Verfahren ist für die Unterstützung konkurrierender Benutzer geeignet.

Für das preemptive, also verdrängende Scheduling gibt es verschiedene Scheduling-Strategien (Algorithmen), von denen heute in Betriebssystemen einige genutzt werden, andere aber nie aus dem Forschungsstadium herauskamen.

Wir betrachten im Folgenden kurz einige dieser Strategien, klassifiziert nach zu unterstützenden Systemen. Im Prinzip ist es heute so, dass ein Betriebssystem in der Regel eine Kombination von Strategien unterstützt.

5.2.2 Überblick über Scheduling-Verfahren

Die Anforderungen an das Scheduling sind je nach Betriebssystemtyp etwas unterschiedlich gelagert. Rein Batch-orientierte Systeme nutzten z.B. in den 60er und 70er Jahren vorwiegend nicht-verdrängende Verfahren, während bei Dialogsystemen verdrängende Verfahren notwendig sind. Ganz andere Anforderungen stellen

Echtzeitsysteme. Wir unterscheiden daher in Scheduling-Verfahren für Batch-, Dialog- und Realtime-Systeme.

Scheduling für Batch-orientierte Prozesse. Für Batch-orientierte Prozesse findet man u.a. folgende Scheduling-Verfahren zur Unterstützung der CPU-Zuteilung:

- First Come First Serve (*FCFS*)
- Shortest Job First (*SJF*), Shortest Process First (*SPF*) bzw. Shortest Process Next (*SPN*)
- Shortest Remaining Time Next (*SRTN*)

Die Bezeichnungen für die Algorithmen sind für sich sprechend. Die Algorithmen *FCFS* und *SJF/SPF/SPN* sind vom Grundsatz her non-preemptive, was allerdings in heutigen Betriebssystemen so nicht mehr implementiert ist. Unterbrechungen werden in allen modernen Betriebssystemen auch bei Batch-Prozessen zugelassen.

FCFS bearbeitet die im System ankommenden Aufträge in der Reihenfolge ihres Eintreffens. *SJF* sucht sich dagegen immer den Job bzw. Prozess aus, von dem es die kürzeste Bedienzeit erwartet.

SRTN ist eine verdrängende Variante von *SJF*. *SRTN* wählt im Gegensatz zu *SJF* als nächstes immer den Prozess mit der am kürzesten verbleibenden Restrechenzeit im System aus. Dieser Algorithmus bevorzugt wie *SJF* neu im System ankommende Prozesse mit kurzer Laufzeit.

Ein großer Nachteil bei *SJF/SPF/SPN*- und *SRTN*-Strategien ist, dass sie zum *Verhungern* (engl. Starvation) von Prozessen führen können. Es kann also Prozesse geben, die nie eine CPU zugeteilt bekommen und daher nicht ausgeführt werden. Beispielsweise tritt der Fall bei *SJF* ein, wenn ständig Prozesse ins System kommen, die nur kurze Zeit dauern. Länger dauernde Prozesse kommen dann nie zum Zuge, was dem Fairness-Prinzip widerspricht. So gut wie unlösbar ist bei *SJF/SPF/SPN* bzw. *SRTN* die Frage, wie das Betriebssystem herausfindet, ob ein Prozess nur (noch) kurze Zeit oder länger dauern wird. Die Strategie ist daher praktisch nicht zu realisieren.

Scheduling für Dialog-orientierte Prozesse. Für die Unterstützung der CPU-Zuteilung zu Dialog-Prozessen gibt es u.a. folgende Scheduling-Algorithmen:

- Round Robin (*RR*)
- Priority Scheduling (*PS*)
- Shortest Remaining Time First (*SRTF*) bzw. Shortest Remaining Process Time (*SRPT*)
- Garantiertes Scheduling
- Lottery Scheduling

RR ist im Prinzip FCFS (siehe Batch-Strategien) in Verbindung mit einer Zeitscheibe. RR geht davon aus, dass alle Prozesse gleich wichtig sind. Ein Prozess erhält ein bestimmtes *Quantum* (auch *Zeitscheibe* oder engl. *time slice* genannt) und wenn dieses abgelaufen ist, wird der Prozess unterbrochen und ein anderer Prozess erhält die CPU. Der unterbrochene Prozess wird hinten in die Warteschlange eingetragen und kommt erst dann wieder an die Reihe, wenn die anderen Prozesse ihr Quantum verbraucht haben oder aus einem anderen Grund unterbrochen wurden. Die Frage nach der Länge der Zeitscheibe ist von großer Bedeutung für die Leistung des Systems.

PS wählt immer den Prozess mit der höchsten Priorität aus. Dies setzt natürlich die Verwaltung von Prioritäten voraus. Jedem Prozess P_i wird eine Priorität p_i zugewiesen. P_i wird vor P_j ausgewählt, wenn p_i größer als p_j ist. Prozesse mit gleicher Priorität werden meist gemeinsam in einer Warteschlange verwaltet. Innerhalb der Prioritätsklasse kann dann im RR-Verfahren ausgewählt werden (siehe Multilevel-Scheduling).

SRTF/SRPT wählt den Prozess mit der kürzesten Restrechenzeit als nächstes aus und ist für Dialogprozesse optimal, aber auch praktisch nicht realisiert. In Betriebssystemen liegen keine Informationen über die verbleibende Prozesszeit vor. Wie sollte man denn auch den Prozess mit der vermutlich kürzesten Verweilzeit im System herausfinden? Zudem ist ein Verhungern von Prozessen möglich.

Als *garantiertes Scheduling* bezeichnet man ein Verfahren, in dem jedem Prozess der gleiche Anteil der CPU zugeteilt wird. Gibt es also n Prozesse im System, so wird jedem Prozess $1/n$ der CPU-Leistung zur Verfügung gestellt. Bei diesem Verfahren muss festgehalten werden, wie viel CPU-Zeit ein Prozess seit seiner Erzeugung bereits erhalten hat. Diese Zeit wird in Relation zur tatsächlich vorhandenen CPU-Zeit gesetzt. Der Prozess, der das schlechteste Verhältnis zwischen verbrauchter und tatsächlich vorhandener CPU-Zeit hat, wird als nächstes ausgewählt und darf so lange aktiv bleiben, bis er die anderen Prozesse überrundet hat.

Auch die zufällige Auswahl eines Prozesses wie etwa im *Lottery Scheduling* ist eine mögliche Strategie. Jede Scheduling-Entscheidung erfolgt dabei zufällig etwa in der Form eines Lotterieloses. Das System könnte z.B. n mal in der Sekunde eine Verlosung unter den Prozessen durchführen und dem Gewinner der Verlosung die CPU für m Millisekunden bereitstellen.

Scheduling für Realtime-Prozesse. Realtime-Systeme (Echtzeitsysteme, Realzeitsysteme) erfordern ganz andere Strategien bei der Auswahl des nächsten Jobs als batch- und dialogorientierte Systeme. Hier ist vor allem eine schnelle und berechenbare Reaktion auf anstehende Ereignisse wichtig. Es wird zwischen *hard real*

time und *soft real time* unterschieden. Erstere müssen schnell reagieren, bei letzteren ist eine gewisse Verzögerung zumutbar.

Weiterhin wird unterschieden, ob die Scheduling-Entscheidung von Haus aus (bei der Systemprogrammierung) festgelegt wird oder ob die Scheduling-Entscheidung zur Laufzeit gemacht wird. Erstere nennt man *statische*, letztere *dynamische* Algorithmen.

Folgende Scheduling-Algorithmen für Realtime-Systeme sind u.a. bekannt:

- Minimal Deadline First
- Polled Loop
- Interrupt-gesteuert

Bei *Minimal Deadline First* wird der Prozess mit der kleinsten nächsten Zeitschranke (deadline) als erstes ausgewählt. Bei *Polled Loop* werden alle Geräte (Ereignisquellen) zyklisch nach einem anstehenden Ereignis abgefragt, und dieses wird dann gleich bearbeitet. Interrupt-gesteuerte Systeme warten z.B. in einer Warteschleife auf Interrupts von Ereignisquellen und führen dann die geeignete Interrupt-Service-Routine (ISR) aus.

5.2.3 Multi-Level-Scheduling mit Prioritäten

Manche Betriebssysteme unterstützen im Rahmen ihrer Scheduling-Strategie für verschiedene Prozess-Prioritäten auch jeweils eine eigene Warteschlange, in der Prozesse gleicher Priorität eingeordnet werden. Die Warteschlangen dienen dem Scheduler zur Auswahl des nächsten auszuführenden Prozesses.

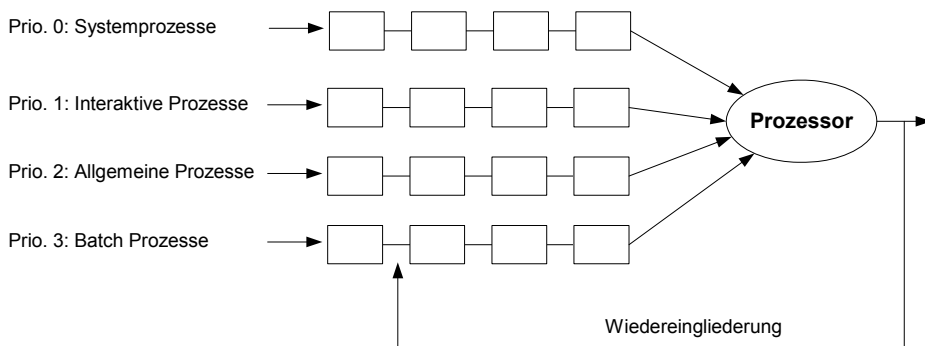


Abbildung 5-3: Multi-Level-Scheduling mit Prioritäten

Im Multi-Level-Scheduling werden mehrere Warteschlangen verwaltet und zwar für jede Prioritätsstufe oder für jeden unterstützten Prozesstypen eine eigene. In

Abbildung 5-3 ist z.B. ein Modell dargestellt, in dem den verschiedenen Prozessstypen (Systemprozesse, interaktive Prozesse,...) jeweils eine Warteschlange zugeordnet ist.

Nachdem ein Prozess abgearbeitet ist, wird er entweder beendet oder wieder in die Warteschlange eingereiht, was wiederum nach einer eigenen Strategie erfolgen kann. Multi-Level-Feedback-Scheduling ist eine Abwandlung zum Multi-Level-Scheduling, in der ein Prozess auch die Warteschlange wechseln kann.

Bei Mehrprozessorsystemen müsste man in der obigen Abbildung entsprechend mehrere Prozessoren zeichnen. Ein ausführbereiter Prozess aus einer Warteschlange wird dann jeweils einem Prozessor zur Ausführung zugeordnet.

5.2.4 Round-Robin-Scheduling mit Prioritäten

Das Round-Robin-Verfahren, ergänzt um eine Prioritätensteuerung, soll aufgrund seiner praktischen Bedeutung nochmals etwas näher betrachtet werden. Es gibt zwei wichtige Aspekte zu betrachten:

1. Wie lange soll die Zeitscheibe für einzelne Prozesse (das Quantum) sein?
2. Wie wird die Priorität einzelner Prozesse eingestellt bzw. ermittelt?

Zur Frage 1) ist anzumerken, dass eine zu kurze Zeitscheibe in der Regel einen hohen Overhead für den ständigen Kontextwechsel zur Folge hat. Ein zu langes Quantum führt möglicherweise zu langen Verzögerungen einzelner Prozesse, was insbesondere bei interaktiven Aufträgen schädlich ist.

In Abbildung 5-4 sind zwei Prozesse A und B skizziert, die jeweils nach einem gleich langen Quantum die CPU im RR-Verfahren erhalten. Der Overhead des Kernels für den Kontextwechsel ist ebenfalls angedeutet.

Nehmen wir an, der Overhead für einen Kontextwechsel dauert 1 ms und die Zeitscheibe ist 10 ms lang und fix. Dann würden 10 % der Rechenleistung nur für den Kontextwechsel benötigt, pro Sekunde also 100 ms. Wäre dagegen die Zeitscheibe 100 ms lang und der Kontextwechsel würde 1 ms dauern, hätte man nur einen Overhead von 1 %. Der richtige Mittelweg muss gefunden werden.

Das Quantum kann statisch festgelegt oder dynamisch verändert werden. Die dynamische Veränderung des Quantums zur Laufzeit ist in heutigen Betriebssystemen genau so üblich wie eine statische Vorbelegung. Die Dauer des Quantums hängt natürlich auch von der Prozessorleistung ab. Je höher die Taktrate, umso mehr Rechenleistung ist verfügbar und umso kürzer wird das Quantum sein. Übliche Größen für Quanten sind heute 10 bis 100 ms.

Zu Frage 2) gibt es mehrere Möglichkeiten. Die Priorität kann statisch beim Prozessstart je nach Prozessstyp festgelegt und zur Laufzeit nicht mehr verändert werden. Es ist aber auch eine dynamisch, adaptive Veränderung der Prioritäten von Prozessen möglich oder sogar eine Kombination beider Varianten, also eine statische Voreinstellung und eine dynamische Anpassung der Priorität. Letztere Variante ist in heutigen Betriebssystemen recht häufig vorzufinden.

Hierbei ist es aber wichtig, dass die Vergabe der CPU einigermaßen gerecht verläuft, was durchaus bei prioritätengesteuertem Scheduling problematisch sein kann. Wenn dauernd Prozesse mit hoher Priorität im System sind, werden möglicherweise Prozesse mit niedriger Priorität vernachlässigt und müssen verhungern (engl: Starvation). Diesem Problem muss vorgebeugt werden, was z.B. durch eine kurzzeitige, dynamische Erhöhung der Priorität für benachteiligte Prozesse erfolgen kann.

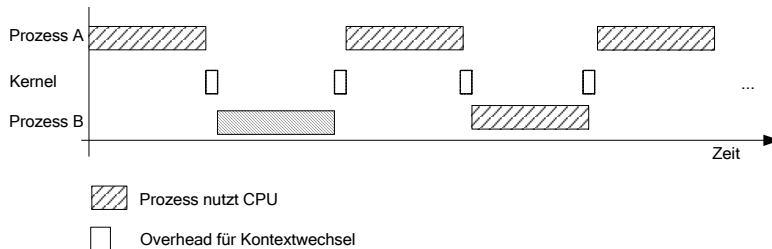


Abbildung 5-4: Zeitverbrauch für Kontextwechsel

In der Praxis stellt folgendes Verfahren einen recht guten Ansatz dar:

- Das Quantum wird initial eingestellt und passt sich dynamisch an, so dass auch Prozesse mit niedriger Priorität die CPU ausreichend lange erhalten.
- Ein-/Ausgabe-intensive Prozesse erhalten ein höheres Quantum, rechenintensive ein kürzeres. Damit können vernünftige Antwortzeiten von Dialogprozessen erreicht werden, und die rechenintensiven Prozesse können gut damit leben.
- Die Prozess-Prioritäten werden statisch voreingestellt und unterliegen einer dynamischen Veränderung. Die aktuelle Priorität wird auch als relative Priorität bezeichnet.

Dieses Verfahren erfordert natürlich eine Verwaltung der Prozessdaten *Quantum* und *Priorität* und eine zyklische Anpassung dieser. Das Quantum der aktiven Prozesse wird taktorientiert herunter gerechnet, und zu vorgegebenen Zeiten muss eine Quantumsneuberechnung durchgeführt werden.

Ebenso ist eine Neuberechnung der Prozess-Prioritäten erforderlich. Der Berechnungsalgorithmus kann auch die verbrauchten Quanten mit einbeziehen.

Dieses etwas vereinfachte Verfahren wird meist in Kombination mit dem Multilevel-Feedback-Scheduling eingesetzt und liefert gute Ergebnisse. Natürlich haben die einzelnen Betriebssysteme ihre Besonderheiten. Beispielsweise vergibt Windows einen Prioritätsbonus für Prozesse, die schon länger keine CPU mehr zugeteilt bekommen haben.

5.3 Vergleich ausgewählter Scheduling-Verfahren

Vergleicht man die betrachteten Scheduling-Algorithmen, so können einige der oben genannten Scheduling-Kriterien herangezogen werden. Interessant sind z.B. die Durchlaufzeit (auch Verweilzeit oder turnaround time), die Wartezeit, die Bedienzeit (auch Servicezeit), die Antwortzeit, der Durchsatz oder die CPU-Auslastung:

- Durchlaufzeit (= Verweilzeit): Gesamte Zeit, in der sich ein Prozess im System befindet (Servicezeiten + Wartezeiten).
- Wartezeit: Zeit, die ein Prozess auf die Ausführung warten muss, also die Summe aller Zeiträume, in denen ein Prozess warten muss.
- Bedienzeit (Servicezeit): Zeit, in der ein Prozess die CPU hält und arbeiten kann.
- Antwortzeit: Zeit, in der ein Anwender auf die Bearbeitung seines Auftrags warten muss.
- Durchsatz: Anzahl an Prozessen, die ein System in einer bestimmten Zeit bearbeiten kann.
- CPU-Auslastung: Auslastung der CPU während der Bearbeitung von Prozessen in Prozent der Gesamtkapazität.

Wir vergleichen in diesem Abschnitt einige der oben erläuterten Scheduling-Algorithmen hinsichtlich des Beurteilungskriteriums „Durchlaufzeit“. Betrachten wir zu den genannten Scheduling-Strategien ein Beispiel nach (Tanenbaum 2009): Nehmen wir an, dass in einem System fünf Aufträge (Jobs) A, B, C, D und E fast gleichzeitig eintreffen. Die einzelnen Aufträge erhalten folgende Prioritäten (Priorität 5 ist die höchste) und Ablaufzeiten (hier in Sekunden):

Job	A	B	C	D	E
Ablaufzeit	10	6	4	2	8
Priorität	3	5	2	1	4

Gesucht sind jeweils die gesamte (V_{all}) und die durchschnittliche (V_{avg}) Verweilzeit im System bei Einsatz folgender Scheduling-Algorithmen:

- 1) Priority Scheduling (nicht verdrängend)
- 2) FCFS unter Berücksichtigung der Reihenfolge-Annahme: A, B, D, C, E (nicht verdrängend)
- 3) Shortest Job First (nicht verdrängend)
- 4) RR mit statischen (also sich nicht verändernden) Prioritäten bei einem Quantum von 2 Sekunden (verdrängend)

Die Jobs treffen ungefähr gleichzeitig im System ein. Die Prozesswechselzeit wird vernachlässigt und die Aufträge werden nacheinander ausgeführt. Eine Verdrängung (Preemption) wird nur im Fall 4 ausgeführt.

Priority Scheduling. Die folgende Tabelle zeigt die Reihenfolge der Auftragsausführung bei Priority Scheduling sowie die jeweiligen Verweilzeiten der einzelnen Jobs:

Job	B	E	A	C	D
Verweilzeit	6	14	24	28	30

Die Summe über alle Verweilzeiten ist $V_{\text{all}} = 6 + 14 + 24 + 28 + 30 = 102 \text{ s}$.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 20,4 \text{ s}$.

Die Abbildung 5-5 verdeutlicht den Ablauf nochmals.

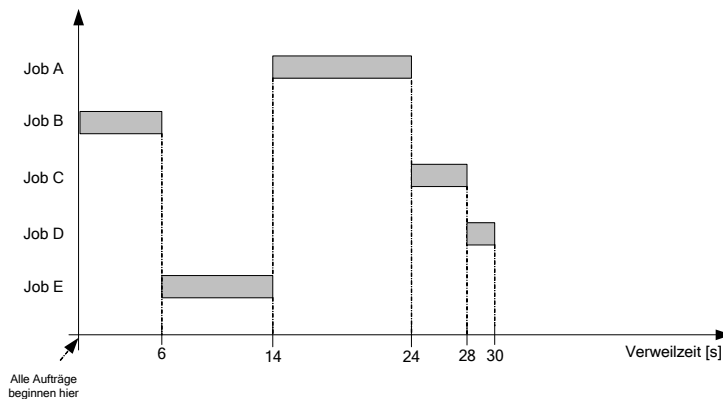


Abbildung 5-5: Beispiel für Priority-Scheduling

FCFS-Scheduling. Die folgende Tabelle zeigt die Reihenfolge der Auftragsausführung bei FCFS-Scheduling sowie die jeweiligen Verweilzeiten der einzelnen Jobs.

Job	A	B	D	C	E
Verweilzeit	10	16	18	22	30

Die Summe über alle Verweilzeiten $V_{\text{all}} = 10 + 16 + 18 + 22 + 30 = 96 \text{ s}$.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 19,2 \text{ s}$.

FCFS ist also in diesem Szenario besser als reines Prioritäten-Scheduling. Die Abbildung 5-6 zeigt den Ablauf bei FCFS-Scheduling.

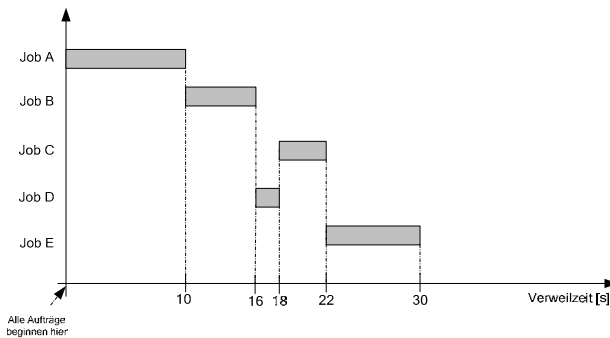


Abbildung 5-6: Beispiel für FCFS-Scheduling

SJF-Scheduling. Die folgende Tabelle zeigt die Reihenfolge der Auftragsausführung bei SJF-Scheduling:

Job	D	C	B	E	A
Verweilzeit	2	6	12	20	30

Die Summe über alle Verweilzeiten $V_{\text{all}} = 2 + 6 + 12 + 20 + 30 = 70 \text{ s}$.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 14,0 \text{ s}$.

SJF ist, wie bereits erläutert, die beste aller Lösungen was die Verweilzeit anbelangt.

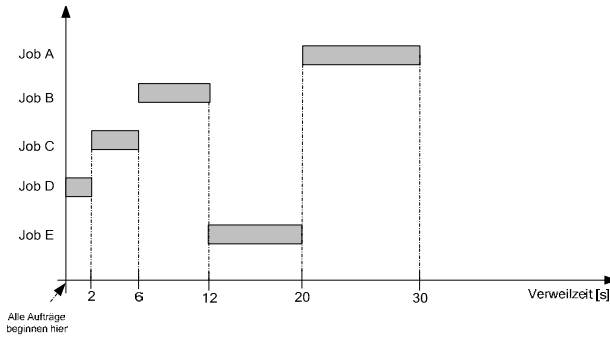


Abbildung 5-7: Beispiel für SJF-Scheduling

RR-Scheduling mit Prioritäten. Die folgende Tabelle zeigt die Reihenfolge der Auftragsausführung bei RR-Scheduling unter Berücksichtigung von Prioritäten und einer Zeitscheibe von 2 s, wobei der Overhead für den Prozesswechsel vernachlässigt wird.

Job	A	B	C	D	E
Verweilzeit	30	20	18	10	26

Die Summe über alle Verweilzeiten $V_{\text{all}} = 30 + 20 + 18 + 10 + 36 = 104$ s.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 20,8$ s.

RR-Scheduling mit Prioritäten ist also die schlechteste aller Varianten, dafür aber auch die gerechteste.

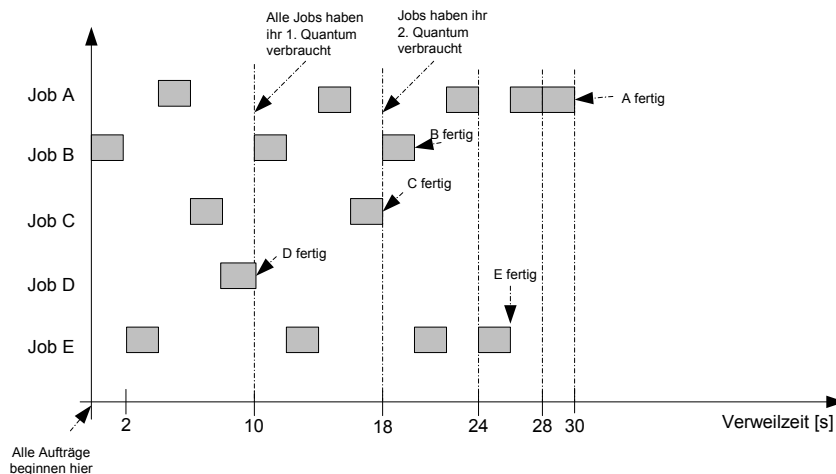


Abbildung 5-8: Beispiel für RR-Scheduling mit Prioritäten

Die in den Beispielen skizzierten Ergebnisse zeigen unterschiedlichste Scheduling-Abläufe. Die Ergebnisse lassen sich auch einfach beweisen. Wir betrachten hierzu fünf beliebige Jobs mit den Bezeichnungen A, B, C, D und E und benennen die erwarteten Ausführungszeiten der Einfachheit halber für die folgende Berechnung mit a, b, c, d und e.

Die gesamte Verweilzeit aller Jobs im System ergibt sich dann wie folgt:

$$V_{\text{all}} = a + (a+b) + (a+b+c) + (a+b+c+d) + (a+b+c+d+e) = 5a + 4b + 3c + 2d + e$$

Die durchschnittliche Verweilzeit ergibt sich dann aus

$$V_{\text{avg}} = (5a + 4b + 3c + 2d + e) / 5$$

Der Job A trägt also am meisten zur durchschnittlichen Verweilzeit bei, der Job B steht an zweiter Stelle usw. Will man die durchschnittliche Verweilzeit minimieren, sollte zunächst der Job mit der kürzesten Verweilzeit ausgeführt werden, da die Ausführungszeit des ersten gestarteten Jobs für die Berechnung mit 5 multipliziert wird und damit am meisten Gewicht hat. Danach sollte der zweitkürzeste Job folgen usw.

Anmerkung. In der Praxis ist es schwierig, die Ausführungszeiten von vorneherein zu bestimmen. Man kann sie nur schätzen, was sich aber als recht schwierig erweist. Man müsste sich die Ablaufzeiten eines Programmes jeweils für den nächsten Ablauf merken und daraus evtl. mit Hilfe eines Alterungs-Verfahrens die als nächstes erwartete Ablaufzeit berechnen. Das Berechnungsbeispiel und auch die oben skizzierten Beispiele dienen also vorwiegend der theoretischen Betrachtung.

5.3.1 CPU-Scheduling im ursprünglichen Unix

Die ursprüngliche Intention von Unix als Multiprogramming-/Multiuser-Betriebssystem war die Unterstützung des Dialogbetriebs, wobei die Anwender auf Terminals mit dem System arbeiten. Daher stand im Vordergrund, den Benutzern kurze Antwortzeiten im interaktiven Betrieb zu bieten. Der am System arbeitende Benutzer sollte somit das Gefühl erhalten, dass das System für ihn alleine arbeitet.

Unterstützte Scheduling-Strategie. Unix-Systeme (zumindest die ursprünglichen Unix-Systeme) verwenden als Scheduling-Strategie *Round-Robin (RR)*, ergänzt um eine *Prioritätensteuerung*. Es wird auch eine Multi-Level-Feedback-Queue verwendet, in der für jede Priorität eine Queue verwaltet wird. Das Scheduling erfolgt verdrängend (preemptive). Als Scheduling-Einheit dient in traditionellen Unix-Systemen der Prozess.

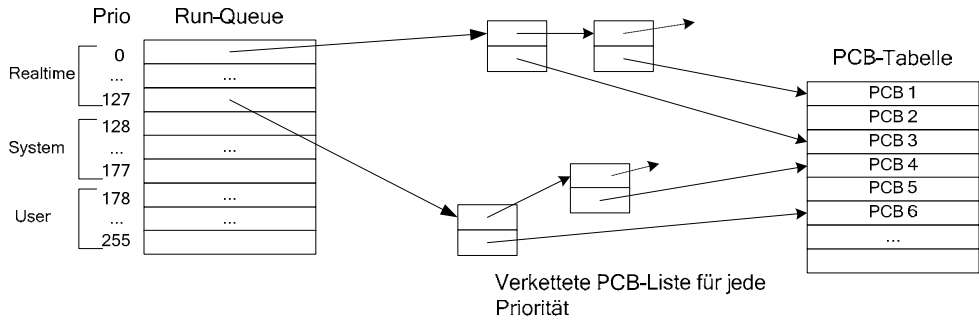


Abbildung 5-9: Run-Queue unter Unix

Abbildung 5-9 zeigt eine Multi-Level-Queue für die Prioritätenverwaltung, wie sie unter Unix typischerweise aussieht. Dies ist die wichtigste Datenstruktur des Schedulers und wird als *Run-Queue* bezeichnet. Es muss allerdings erwähnt werden, dass es bei dieser Funktionalität je nach Unix-Derivat (System V, BSD, HP UX, AIX, Solaris,...) doch einige Unterschiede gibt. Es soll deshalb hier nur das Grundprinzip verdeutlicht werden.

Prioritätsberechnung. Neuere Unix-Derivate verwalten Prioritäten zwischen 0 und 255, wobei 0 die höchste Priorität darstellt. In früheren Unix-Derivaten gab es Prioritäten zwischen -127 bis +127 mit -127 als niedrigste Priorität. Die initiale Priorität eines Prozesses verändert sich im Laufe der Zeit dynamisch. Systemprozesse und bei entsprechender Unterstützung auch Realtime-Prozesse werden bevorzugt behandelt und erhalten höhere Prioritäten. Hier gibt es natürlich unterschiedlichste Varianten der Realisierung in den einzelnen Unix-Derivaten.

Bei Unix System V Release 3 wird die Priorität etwa jede Sekunde neu berechnet, wobei die bisher aufgebrauchte CPU-Zeit in die Berechnung mit eingeht. War diese bisher hoch, sinkt die Priorität, wobei allerdings nur die in der letzten Zeit verbrauchte CPU-Zeit berücksichtigt wird. Durch diese Vorgehensweise werden auch Dialogprozesse, die relativ oft nicht aktiv sind und auf eine Interaktion mit dem Benutzer warten, bevorzugt.

Run-Queue-Verwaltung. Die *Run-Queue* verwaltet alle Prozesse und deren Zugehörigkeiten zu Prioritäten. In der *Run-Queue* ist für jede Priorität ein Zeiger vorhanden, der auf eine Warteschlange von Prozessen mit gleicher Priorität verweist. Die Warteschlangen-Elemente verweisen dann letztendlich auf den eigentlichen PCB in einer PCB-Tabelle. Bei einem Kontextwechsel wird immer der vorderste Prozess in der Queue der höchsten Priorität ausgewählt und nach Abarbeitung einer Zeitscheibe (ursprünglich 100 ms bis 1 s) wieder ans Ende einer Queue ein-

gehängt, wobei vorher seine neue Priorität ermittelt wird. Es ist also möglich, dass der Prozess in eine andere Queue eingehängt wird.

5.3.2 CPU-Scheduling unter Linux

Auch die Scheduling-Mechanismen unter Linux entwickelten sich in den letzten Jahren stark weiter. Ursprünglich lehnte sich das Scheduling stark an Unix an und nutzte Prozesse als Scheduling-Einheit. Im Laufe der Zeit wurden zusätzliche Strategien implementiert. So wurden z.B. die POSIX-Echtzeitstrategien eingebaut.

Linux¹ nutzt heute Threads als Scheduling-Einheit, da bei diesem Betriebssystem Threads auf Kernelebene realisiert sind. Die Begriffe *Prozess* und *Thread* werden unter Linux nicht so stark abgegrenzt wie in anderen Betriebssystemen. Unter Linux kann nämlich in Abweichung zum POSIX-Standard mit dem Systemdienst *clone* ein Kindprozess generiert werden. Beim Aufruf wird mit dem Parameter *sharing_flags* festgelegt, was der Kindprozess erbt. Wenn das Kind den Adressraum mit dem Vater teilt, spricht man von einem Thread. Im Weiteren werden die Begriffe *Thread* und *Prozess* für das Scheduling nicht unterschieden.

Unterstützte Scheduling-Strategien. Linux nutzt einen prioritätengesteuerten Scheduling-Algorithmus, wobei drei Scheduling-Strategien, auch Scheduling-Klassen genannt, unterstützt werden. Die Scheduling-Klassen werden auf drei verschiedene Prozess- bzw. Thread-Klassen angewendet:

- *Timesharing* für die Standard-Benutzerthreads: Diese Strategie ist verdrängend und wird bei „normalen“ Prozessen angewendet. Der Scheduler ermittelt für jeden Prozess nach einem bestimmten Algorithmus eine Zeitscheibe (Quantum), die immer wieder neu berechnet wird.
- *Realtime mit FIFO*: Diese Strategie wird bei Prozessen mit höchster Priorität angewendet und ist nicht verdrängend (non-preemptive), d.h. einem Prozess wird kein Quantum zugeordnet und einem Prozess wird die CPU vom Betriebssystem nicht entzogen.
- *Realtime mit Round Robin*: Dieses verdrängende Verfahren nutzt RR als Scheduling-Strategie. Jeder Prozess erhält ein Quantum. Nach Ablauf seines Quantums wird ein Prozess verdrängt und an das Ende der RR-Liste gehängt.

Realtime-Prozesse sind unter Linux nicht wirklich realtimefähig für harte Echtzeitanforderungen. Eigenschaften wie Deadlines oder garantierte CPU-Zuteilung

¹ Der Linux-Kernel 2.6.x, mit $x < 23$, wird hier als Basis verwendet. Dieser Scheduler wird auch als *O(1)-Scheduler* bezeichnet. Mit Version 2.6.23 wurde ein völlig neuer Scheduler eingeführt, der als *Completely Fair Scheduler (CFS)* bezeichnet wird.

werden nicht unterstützt. Diese Art von Realtime-Prozessen ist im POSIX-Standard P1003.4 genormt. Dieser Standard wird in Linux unterstützt. Realtime-Prozesse haben einfach nur eine höhere Priorität und werden immer gegenüber Timesharing-Prozessen bevorzugt.

Es soll noch erwähnt werden, das Linux bei Mehrprozessorsystemen versucht, einen Prozess immer derselben CPU zuzuordnen. Dadurch müssen die Caches nicht so oft neu geladen werden. Dies bezeichnet man als Prozessoraffinität. Linux unterscheidet hier explizite Prozessoraffinität, die über einen Systemcall eingestellt werden kann, und implizite Prozessoraffinität. Bei letzterer versucht der Kernel die Zuteilung selbst vorzunehmen.

Der Linux-Scheduler trifft zyklisch bzw. wenn er aufgerufen wird, eine globale Entscheidung, welcher Prozess als nächstes die CPU erhält. Bei Multiprozessorsystemen wird entsprechend entschieden, welche Prozesse als nächstes abgearbeitet werden. Grundsätzlich unterstützt Linux eine hohe Interaktivität mit Benutzern. Prozesse, die oft blockiert sind und daher ohnehin die CPU selten nutzen, sollen bei der CPU-Zuteilung begünstigt werden. Dies bedeutet, dass Ein-/Ausgabeintensive (interaktive) vor rechenintensiven (Batch-) Prozessen bevorzugt werden. Der Scheduler erkennt interaktive Prozesse an einer langen durchschnittlichen Schlafzeit.

Prioritätenberechnung. Prozesse mit höherer Priorität bekommen unter Linux generell mehr CPU-Zeit zugeordnet. Intern verwaltet Linux die Prioritätenbereiche in einer Skala von 0 bis 139. Der Linux-Kernel unterstützt zwei Prioritätsbereiche für Prozesse:

- Für die „normalen“ Timesharing-Prozesse liegt die Priorität, die über den sog. *Nice*-Wert definiert ist, zwischen -20 und 19, wobei der Wert 19 die niedrigere Priorität darstellt. Ein Prozess mit einer Priorität von -20 erhält die größte Zeitscheibe. Die Timesharing-Prioritäten -20 bis +19 werden auf die internen Prioritäten 100 bis 139 abgebildet, wobei 100 der höchsten Priorität und damit einem Nice-Wert von -20 entspricht. Die Priorität der Prozesse wird dynamisch verändert. Als Standardwert wird 120 bzw. der Nice-Wert 0 festgelegt.
- Für die alle Realtime-Prozesse wird ein interner Prioritätenbereich zwischen 0 und 99 zugeordnet, wobei 0 die höchste Priorität darstellt.

Der *Nice*-Wert kann auch mit dem Systemdienst *nice* verändert werden.

Der Scheduler bevorzugt Prozesse mit höherer Priorität und versucht eine Liste mit Prozessen gleicher Priorität so lange abzuarbeiten, bis das Quantum von allen Prozessen der Liste verbraucht ist. Erst danach wird eine Prozessliste mit geringe-

rer Priorität bearbeitet (siehe Mauerer 2004). Eine Ausnahmebehandlung erfahren für längere Zeit blockierte Prozesse. Dies bedeutet aber, dass schlecht programmierte Echtzeitprozesse das System lahm legen können. Auch FIFO-Prozesse, die fehlerhaft in einer Endlosschleife laufen, werden nie unterbrochen, da sie keine Zeitscheibe besitzen.

Arbeitsweise des Schedulers. Ziel des Schedulers ist, dass Prozesse mit höherer Priorität mehr CPU-Anteile erhalten. Der Linux-Scheduler wird in eine periodisch aufgerufene Scheduling-Komponente (periodische Scheduling-Funktion) und einen sog. Hauptscheduler unterteilt. Ersterer wird zyklisch in gleichmäßigen Abständen aufgerufen, um das Quantum der aktiven Prozesse zu verringern. Dies wird von einer Kernel-internen Systemuhr getaktet. Der Hauptscheduler wird zusätzlich bei bestimmten Ereignissen (z.B. von der periodischen Scheduling-Funktion) aufgerufen. Beispielsweise wird er aktiv, wenn ein aktiver Prozess sein Quantum verbraucht hat oder ein aktiver Prozess blockiert.

Der Linux-Kernel nutzt also eine getaktete Systemuhr. Der Takt des Linux-Kernels, auch als *Tick* bezeichnet), war bis zur Kernel-Version 2.5 auf 100 Hertz (Hz)² festgelegt und kann ab dem Kernel 2.6 bis auf 1000 Hz erhöht werden. Ein Tick wird auch als *Jiffy* (Zeitintervall) bezeichnet.

Das Quantum wird von der periodischen Scheduling-Funktion bei jedem Tick reduziert, solange, bis es den Wert 0 erreicht hat. Bei 1000 Hz entspricht ein Tick einer Millisekunde ($1/1000 \text{ s} = 0,001 \text{ s} = 1 \text{ ms}$). Das bedeutet auch, dass die kürzestmögliche Länge einer Zeitscheibe ab der Linux-Version 2.6 eine Millisekunde beträgt.

20 Ticks werden als Standardwert für das Quantum verwendet, was bei einer mit 1000 Hz getakteten Systemuhr demnach 20 ms ergibt.

Einem Prozess wird die CPU entzogen, wenn sein Quantum = 0 ist, der Prozess blockiert (E/A, Semaphor,...) oder zum Schedulingzeitpunkt ein vorher blockierter Prozess mit höherer Priorität bereit wird. Damit wird auch nochmals deutlich, dass die sogenannten Realtime-Prozesse beim Scheduling begünstigt werden.

Die Quanten der Prozesse werden im Laufe der Bearbeitung immer kleiner, bis alle, außer ggf. die Quanten von den Ein-/Ausgabe-lastigen Prozessen, auf 0 stehen. Wenn dieser Zustand erreicht ist, verteilt der Scheduler alle Quanten neu. Die

² Der Wert ist in einer Kernel-Konstante mit dem Namen *HZ* festgelegt. Ab Linux-Version 2.6.13 sind auch die Werte 100, 250, 300 und 1000 möglich. Je größer der Wert ist, desto mehr CPU-Zeit wird aufgrund der erhöhten Anzahl an Unterbrechungsbearbeitungen verbraucht.

Quantumberechnung hängt von der statischen Priorität ab. Je höher der Wert der Priorität, also je niedriger die statische Priorität ist, umso kleiner ist das Quantum. Dieser Zusammenhang ist in Abbildung 5-10 skizziert und lässt sich über eine lineare Funktion beschreiben.

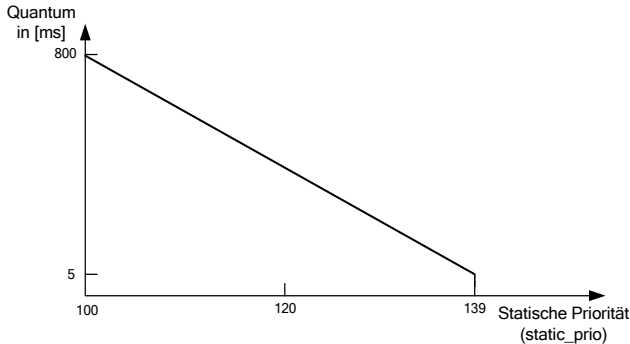


Abbildung 5-10: CPU-Zuteilung auf Basis der statischen Priorität unter Linux

Bei einer statischen Priorität von 100 (interner Wert), was der höchsten Priorität eines Timesharing-Prozesses entspricht, wird ein Quantum von 800 ms ermittelt. Ein Quantum von 5 ms wird dagegen für einen Prozess mit der statischen Priorität 139 berechnet. Nach Bovet 2005 wird das Quantum vom Linux-Kernel über folgende Formel berechnet:

Quantum in [ms] = (140 - statische Priorität) * 20, falls die statische Priorität < 120 ist
bzw.

Quantum in [ms] = (140 - statische Priorität) * 5, falls die statische Priorität ≥ 120 ist

Neben der effektiven (dynamischen) und statischen Priorität sowie der Zeitscheibe verwaltet der Kernel für jeden Prozess einen Bonuswert, der zur Priorität addiert wird. Der Bonus beeinflusst die Auswahl des nächsten Prozesses unmittelbar. Der Wertebereich für den Bonus liegt zwischen -5 und +5. Ein negativer Wert bedeutet eine Verbesserung der Priorität, da eine Verminderung des Prioritätswerts eine Verbesserung der Priorität zur Folge hat. Ein positiver Bonuswert führt zu einer Verschlechterung der effektiven Priorität. Einen Bonus erhalten Prozesse, die viel „schlafen“, also auf Ein-/Ausgabe warten (siehe Abbildung 5-11). Je mehr ein Prozess warten muss, desto höher (also niedrigerer Wert!) wird sein Bonus. Die effektive Priorität wird unter Berücksichtigung des Bonuswerts etwas vereinfacht wie folgt berechnet:

Effektive Priorität = Statische Priorität + Bonus

Mit dem Bonus werden also vor allem interaktive Prozesse begünstigt, die oft auf Eingaben des Benutzers warten. Der maximale Bonus von -5 wird bei einer durch-

schnittlichen Schlafzeit von 1000 ms oder mehr gewährt (siehe Abbildung 5-11). Bei einer Veränderung der effektiven Priorität erfolgt auch eine Eingliederung des Prozesses in die entsprechende Prioritätsqueue.

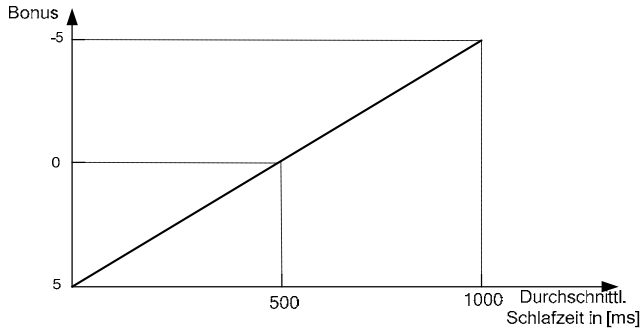


Abbildung 5-11: Bonuszuteilung unter Linux

Ein Prozess wird als interaktiver Prozess behandelt, wenn für ihn folgende Formel zutrifft (siehe Bovet 2005):

$$\text{Effektive Priorität} \leq 3 * \text{statische Priorität} / 4 + 28$$

oder

$$\text{Bonus} - 5 \geq \text{statische Priorität} / 4 - 28$$

Demnach hat die statische Priorität einen hohen Einfluss auf die Behandlung als interaktiver oder als rechenintensiver (Batch-) Prozess. Ein Prozess mit hoher statischer Priorität wird schon bei einer durchschnittlichen Schlafzeit von 200 ms als interaktiver Prozess eingeordnet, ein Prozess mit einer statischen Priorität von 139 kann nie ein interaktiver Prozess sein. Ein Prozess mit einer statischen Priorität von 120 gilt ab einer durchschnittlichen Schlafzeit von 700 ms als interaktiv.

Zusammengefasst kann man also festhalten:

- Die Parameter Quantum, Priorität und Bonus spielen im Linux-Scheduling bei der Quantumberechnung zusammen.
- Rechenintensive Prozesse werden vom Linux-Scheduler weniger bevorzugt als mehr Ein-/Ausgabe-intensive (interaktive) Prozesse. Rechenintensive Prozesse verbrauchen ihr Quantum schnell und erhalten bei einer Neuberechnung ein neues Quantum, das direkt von ihrer statischen Priorität abgeleitet wird. Weiterhin erhalten sie bei intensiver Nutzung der CPU einen schlechten Bonuswert, der die effektive Priorität erhöht und damit verschlechtert.
- Ein-/Ausgabe-intensive Prozesse werden bei einer Neuordnung der Quanten durch das Hinzufügen eines günstigen Bonuswerts, der bei der Be-

rechnung der effektiven Priorität zur statischen Priorität addiert wird, bevorzugt.

- Ob ein Prozess interaktiv oder rechenintensiv ist, hängt von seiner durchschnittlichen Schlafzeit ab.

Run-Queue-Verwaltung. Die zentrale Datenstruktur des Schedulers wird auch in Linux als *Run-Queue* bezeichnet. Für jede CPU wird unter Linux eine eigene Run-Queue verwaltet (siehe Abbildung 5-12). Ein Loadbalancing-Mechanismus verteilt die Prozesse auf die CPUs.

Der Aufbau der Run-Queue ist grob in Abbildung 5-12 skizziert. Die Datenstruktur enthält im Wesentlichen Referenzen (Zeiger) auf die eigentlichen Prozess-Queues. Diese werden repräsentiert durch die Datenstruktur *prio_array* und werden auch als Priority-Array bezeichnet.

Linux unterscheidet eine Prozess-Queue für aktive (*active*) und eine für abgelaufene (*expired*) Prozesse. In der *active Queue* sind alle Prozesse mit einem Quantum > 0 enthalten. In der *expired Queue* werden alle Prozesse, deren Quantum bereits abgelaufen ist, verwaltet. Sobald ein Prozess sein Quantum abgearbeitet hat, wird er in die *expired Queue* umgehängt. Wenn alle Prozesse der *active Queue* ihr Quantum abgearbeitet haben, wird das Quantum für alle Prozesse der *expired Queue* neu ermittelt, und es werden einfach die entsprechenden Zeiger auf die *active* und die *expired Queue*, die in der Run-Queue verwaltet werden, ausgetauscht. Im Detail ist das etwas komplizierter (siehe Mauerer 2004 und Bovet 2005), was aber in unserer Betrachtung vernachlässigt werden soll.

In der Run-Queue sind u.a. noch die folgenden Informationen enthalten:

- Der *Idle*-Zeiger zeigt auf den *Idle*-Prozess, der aktiviert wird, wenn kein anderer Prozess aktivierbar ist.
- Der *Current*-Zeiger zeigt auf den aktuell aktiven Prozess.
- Das Feld *nr_running* enthält die Anzahl aller ablaufbereiten Prozesse.
- In der *active/expired Queue* wird für jede Priorität eine Liste mit Referenzen, also insgesamt 140 Referenzen, verwaltet. In diesen Listen sind jeweils alle Prozesse mit gleicher Priorität eingetragen.

Zudem wird eine Bitmap gepflegt, in der für jede dieser 140 Listen ein Bit verfügbar ist. Ist ein Bit auf 1 gesetzt, so ist mindestens ein Prozess in der korrespondierenden Liste enthalten. Über die Bitmap kann bei der Scheduling-Entscheidung sehr schnell die als nächstes abzuarbeitende Liste und damit der als nächstes zu aktivierende Prozess gefunden werden. In der Liste muss nur überprüft werden, ob ein Bit gesetzt ist oder nicht. Weiterhin ist im Feld *nr_active* die Anzahl der aktuell lauffähigen Prozesse im gesamten *Priority-Array* hinterlegt. Die verwendete Datenstruktur sieht wie folgt aus:

```
struct prio_array {
    int nr_active;           // Anzahl aktiver Prozesse
                             // Prioritäts-Bitmap, 140 Bits reserviert
    unsigned long bitmap[BITMAP_SIZE];
                             // Prioritäts-Queues, MAX_PRIO = 140 (Default)
    struct list_head queue[MAX_PRIO];
};
```

Die einzelnen Prozesse werden in einer Datenstruktur vom Typ `task_struct` (entsprechend dem PCB) verwaltet (siehe Mauerer 2004). Diese Datenstruktur ist wie folgt aufgebaut:

```
struct task_struct {
    ...
    int prio;                // Dynamische Priorität
    int static_prio;         // Statische Priorität
    unsigned long sleep_avg; // Zeit, die der Prozess schläft (inaktiv ist)
    unsigned long last_run;  // Zeitpunkt, zu dem der Prozess zuletzt lief
    unsigned long policy;    // Scheduling-Strategie: Normal (Timesharing),
                             // FIFO, RR
    ...
    unsigned int time_slice; // verbleibendes Quantum
    ...
};
```

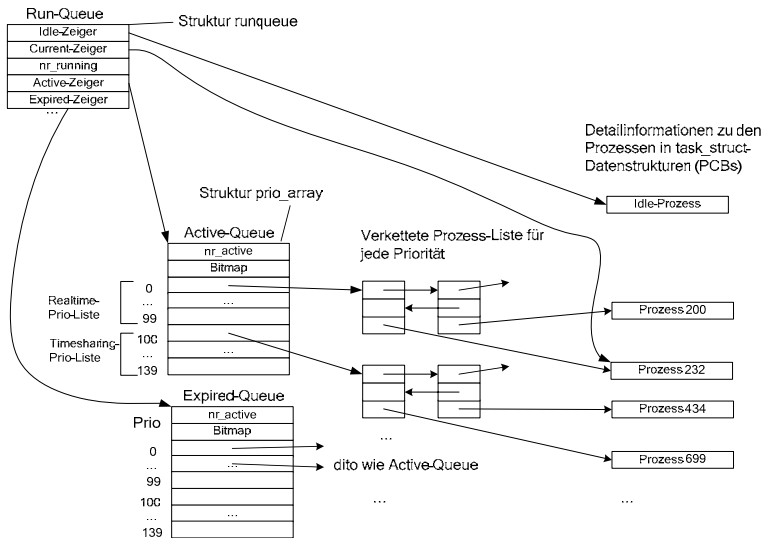


Abbildung 5-12: Run-Queue unter Linux

Unter dynamischer bzw. effektiver Priorität versteht man die aktuelle Priorität eines Prozesses, während die statische Priorität der initialen Einstellung beim Prozessstart entspricht. Eine Veränderung der effektiven Priorität führt zum Einhängen des Prozesses in eine andere Queue.

5.3.3 CPU-Scheduling unter Windows

Das Scheduling unter Windows (ab Windows NT) ist zwar an manchen Stellen sehr speziell, aber es gibt einige Parallelen mit Unix-ähnlichen Betriebssystemen. Windows führt das Scheduling auf Threadebene unabhängig von den Prozessen aus. Als Scheduling-Einheit dienen also Threads (thread-based).

Unterstützte Scheduling-Strategien. Windows verwendet ein *Zeitscheibenverfahren* für das CPU-Scheduling. Es wird nicht berücksichtigt, zu welchem Prozess ein Thread gehört, was auch sinnvoll ist, da Prozesse nur eine Ablaufumgebung bereitstellen. Wenn z.B. ein Prozess *A* zehn Threads hat und Prozess *B* zwei Threads und sonst kein weiterer Benutzerthread im System läuft, so teilt der Scheduler die verfügbare Zeit (abzgl. der Zeit für die Systemthreads) in zwölf Teile auf und vergibt für jeden der zwölf Threads ein Zwölftel der Rechenzeit.

Das Scheduling erfolgt *verdrängend* und *prioritätsgesteuert*. Die Verdrängung findet spätestens nach Ablauf des Quantums (Zeitscheibe) statt. Zudem wird eine mögliche *Prozessoraffinität* von Threads angestrebt, um die Caches optimaler zu nutzen. Threads werden also, wenn möglich, immer der gleichen CPU zugeordnet. Für jeden Thread wird eine *Affinity-Maske* verwaltet, die zu jedem verfügbaren Prozessor angibt, ob er für den Thread zugelassen ist. Die Maske kann explizit über einen Systemcall verändert werden. Windows sorgt aber auch implizit für eine gute Verteilung der Threads auf die verfügbaren Prozessoren.

Prioritätsberechnung. Man unterscheidet unter Windows *interne Prioritäten auf Kernelebene* (Kernelprioritäten) und *Prioritäten auf Laufzeitsystemebene*. Die Prioritätenvergabe für die internen Kernelprioritäten ist wie folgt geregelt (siehe auch Abbildung 5-13):

- Es gibt 32 Prioritätsstufen mit den numerischen Werten von 0 bis 31.
- Die Prioritätsstufen von 16 bis 31 sind sog. *Echtzeitprioritäten* und werden nicht verändert. Dies sind keine Echtzeitprioritäten für harte Echtzeitanforderungen, da Windows kein echtzeitfähiges Betriebssystem ist. Es gibt also keine garantierten Ausführungszeiten und auch keine berechenbaren Verzögerungen, wie es in Echtzeitbetriebssystemen erforderlich ist. Es laufen aber einige Systemthreads unter Echtzeitprioritäten.

- Die Prioritätsstufen 1 bis 15 (einschließlich) werden an die Benutzerprozesse vergeben. Diese können vom Betriebssystem dynamisch angepasst werden.
- Die Prioritätsstufe 0 ist die niedrigste und wird für Systemzwecke genutzt.

Die internen Kernelprioritäten haben nichts mit den Interrupt-Prioritäten der Interrupt-Behandlung unter Windows zu tun. Benutzer-Threads können keine Hardware-Interrupts blockieren, da sie mit einer sehr niedrigen Interrupt-Einstellung ausgeführt werden. Threads, die im Kernelmodus ausgeführt werden, können den Interrupt-Level aber anheben, was bewirkt, dass diese nicht unterbrechbar sind.

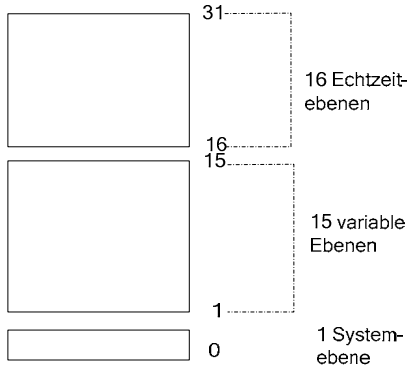


Abbildung 5-13: Prioritätsstufen von Threads in Windows (Solomon 2005)

Im Windows-Laufzeitsystem und damit aus Sicht des Anwendungsprogrammierers werden die Thread-Prioritäten anders dargestellt. Die 32 internen Kernelprioritäten werden in der Windows-API verborgen und auf ein eigenes Prioritätenschema abgebildet. Windows-Threads arbeiten mit zwei Prioritätsebenen:

- Je Prozess wird eine *Prioritätsklasse* zugewiesen. Mögliche Werte sind *idle*, *below normal*, *normal*, *above normal*, *high* und *real-time*. Dieser Prioritätswert wird auch Prozess-Basispriorität genannt.
- Innerhalb der Prioritätsklassen können *von Threads* insgesamt sieben *Relativprioritäten* (auch Threadprioritäten oder Basisprioritäten der Threads genannt) eingenommen werden. Die Bezeichnungen für diese Prioritäten sind *idle*, *lowest*, *below normal*, *normal*, *above normal*, *highest* und *time critical*.

Die Prozess-Basispriorität wird bei der Erzeugung von Threads an diese weitergegeben. Durch Aufruf der Windows-API-Funktion *SetPriorityClass* kann eine Veränderung aller Basisprioritäten der Threads des aufrufenden Prozesses erfolgen. Eine andere Möglichkeit, die Startprioritätsklasse für eine Anwendung festzulegen, funktioniert mit dem Befehl *start* in der Windows-Kommandoeingabe (siehe *start /?*). Als Funktionen zum Lesen und Verändern von Threadprioritäten stehen in der Windows-API *GetThreadPriority* und *SetThreadPriority* zur Verfügung.

Initial erben Threads zunächst die Stufe *normal* der Prioritätsklasse ihres Prozesses als (relative) Thread-Basispriorität. Der Programmierer muss aktiv über einen Aufruf des Systemdienstes *setThreadPriority* eine Veränderung der Prioritäten anstoßen, wenn er das will, ansonsten ist die Thread-Basispriorität gleich der Prozess-Basispriorität.

Tabelle 3-1: Mapping der Windows-API-Prioritäten auf interne Kernelprioritäten

Windows-Thread-Priorität	Windows-Prioritätsklasse					
	real-time	high	above normal	normal	below normal	Idle
time critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Threads verfügen neben der Basispriorität auch noch über eine dynamische Priorität. Diese wird für das CPU-Scheduling verwendet. Die dynamische Threadpriorität wird auch als *aktuelle Threadpriorität* (current priority) bezeichnet. Die aktuelle Threadpriorität wird bei Bedarf vom Kernel angepasst.

In Abbildung 5-14 sind die Zusammenhänge bei der Prioritätenvergabe unter Windows nochmals verdeutlicht.

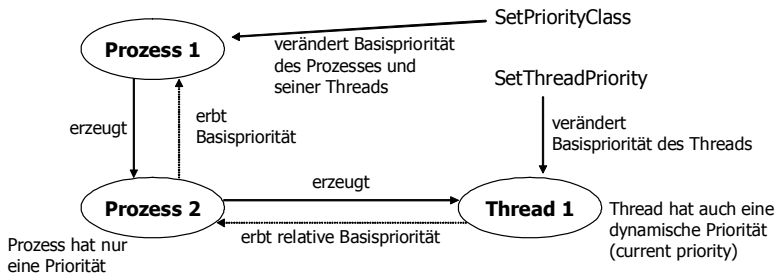


Abbildung 5-14: Vererbung von Prioritäten unter Windows

Die Windows-API bildet die nach außen sichtbaren Prioritäten auf interne Kernelprioritäten ab. In der Tabelle 3-1 sind die Threadprioritäten zeilenweise und die Prioritätsklassen spaltenweise dargestellt. Die Elemente der Tabelle enthalten die zugeordneten internen Kernelprioritäten. Der Prioritätsklasse *normal* ist z.B. für die Threadpriorität *normal* die interne Kernelpriorität 8 zugeordnet.

Arbeitsweise des Schedulers. Der Scheduling-Code ist bei Windows nicht in einer Softwarekomponente gekapselt, sondern im gesamten Kernel auf mehrere Routinen verteilt (Solomon 2005). Ein Threadwechsel wird von der Windows-Prozessverwaltung u.a. bei verschiedenen Ereignissen initiiert:

- Wenn ein Thread mit höherer Priorität bereit ist
- Wenn ein neuer Thread erzeugt wird
- Wenn ein aktiver Thread beendet wird
- Wenn sich ein aktiver Thread in den Wartezustand begibt
- Bei Ablauf des Quantums eines gerade aktiven Threads

Bei diesen Situationen wird überprüft, welcher Thread als nächstes die CPU erhält. Dies ist der Thread mit der höchsten Priorität, der in der entsprechenden Prioritäts-Queue ganz vorne steht.

Nach jedem Clock-Intervall (Tick) wird der aktuelle Thread unterbrochen und sein Restquantum wird reduziert. Die Clock-Intervalle sind in der HAL festgelegt. Bei Intel-x86-basierten Einprozessorsystemen liegen sie üblicherweise bei ca. 10 ms, bei Intel-x86-basierten Multiprozessorsystemen bei ca. 15 ms.

Für jeden Thread wird vom System ein sog. *Quantumszähler* geführt. Dieser wird z.B. bei Windows-Workstations mit Windows XP standardmäßig auf 6 eingestellt, bei Windows-Servern dagegen auf 36 (Solomon 2005). Server erhalten übrigens ein höheres Quantum, damit einmal begonnene Aufgaben auch schnell zu Ende geführt werden können, möglichst bevor das Quantum verbraucht ist.

Nach jedem Clock-Intervall wird der Quantumszähler des laufenden Threads um 3 vermindert. Das bedeutet, dass das Quantum für einen Thread beispielsweise unter Windows XP 2 bzw. 12 Clock-Intervalle lang ist.

Ist der Quantumszähler auf 0, so ist das Quantum abgelaufen und der Thread verliert die CPU. Nach Ablauf des Quantums wird ein Thread also deaktiviert, sofern ein anderer Thread mit gleicher oder höherer Priorität bereit ist. Ist dies nicht der Fall, erhält der unterbrochene Thread die CPU erneut.

Wie wir bei der Betrachtung einiger Szenarien noch sehen werden, wird das Quantum in gewissen Situationen erhöht, um einen evtl. benachteiligten Thread kurzzeitig zu begünstigen. Auch interaktive Threads erhalten ein höheres Quantum.

Unter Windows sind einige dynamische Mechanismen implementiert, die dafür sorgen sollen, dass die Systemleistung verbessert wird. Diese Mechanismen nutzen als hauptsächliche Instrumente die Veränderung der Priorität und des Quantums, wobei in erster Linie Quanten und Prioritäten von Benutzerthreads angepasst werden. Die Instrumente werden auch mit *Priority Boost*, und *Quantum Boost* (auch *Quantum Stretching*) bezeichnet.

Windows erhöht die Priorität eines Threads in mehreren Situationen (Solomon 2005). Eine Situation tritt nach der Bearbeitung einer Ein- oder Ausgabe-Operation ein, eine andere nach dem Warten auf ein Ereignis oder auf ein Semaphore. Auch nachdem interaktive Threads eine Warte-Operation beendet haben oder wenn interaktive Threads wegen einer GUI-Aktivität aufgeweckt werden, kann eine Prioritätsanhebung erfolgen. Schließlich wird eine Prioritätsanhebung durchgeführt, um zu vermeiden, dass ein Thread verhungert.

Zwei Szenarien sollen im Folgenden diskutiert werden:

1. Szenario: Prioritätsanhebung nach Ein-/Ausgabe-Operation (Priority Boost).

Nachdem ein Thread unter Windows eine Ein-/Ausgabe-Operation beendet hat, wird die Priorität angehoben, damit er nach der Wartezeit schnell wieder die CPU zugeteilt bekommt. Die Anhebung wird als *Priority Boost* bezeichnet. Es wird maximal auf die Priorität 15 angehoben. Wie stark die Anhebung tatsächlich ausfällt, hängt davon ab, auf was gewartet wird. Letztendlich entscheidet dies der Gerätetreiber des Gerätes, an dem gewartet wird. Nach einer Platten-Ein-/Ausgabe wird beispielsweise eine Prioritätsanhebung um 1 durchgeführt, nach einer Maus-Eingabe um 6. Die Priorität wird aber anschließend wieder Zug um Zug herabgesetzt. Je abgelaufenes Quantum wird die Priorität des Threads um 1 reduziert, bis wieder die Basispriorität erreicht ist. Bei der Reduktion ist eine Unterbrechung durch höher priorisierte Threads jederzeit möglich. Das Quantum wird dann zu-

nächst aufgebraucht, bevor die Priorität um 1 vermindert wird. Ein erneutes Warten auf Ein-/Ausgabe kann dabei erneut zu einem Priority Boost führen (Abbildung 5-15).

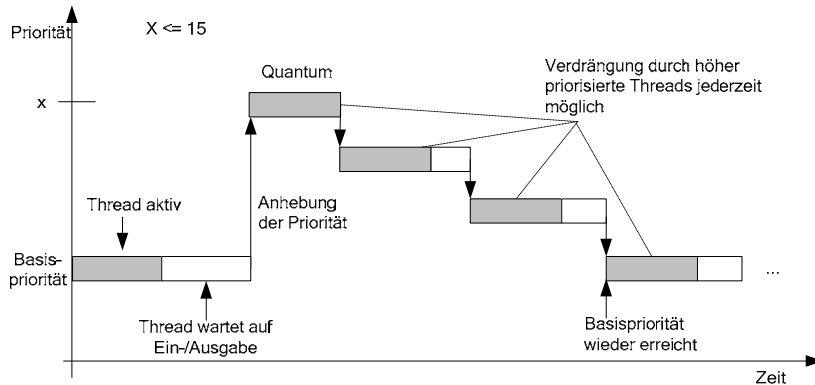


Abbildung 5-15: Prioritätsanhebung bei Windows für wartende Threads

2. Szenario: Rettung verhungender Prozesse. Unter Windows könnten Threads mit niedrigerer Priorität verhungern, da rechenintensive Threads höherer Priorität immer bevorzugt werden. Daher ist unter Windows noch ein weiterer Mechanismus implementiert. Einmal pro Sekunde wird geprüft, ob ein Thread schon länger (ca. 4 Sekunden) nicht mehr die CPU hatte, obwohl er im Zustand „bereit“ ist. Ist dies der Fall, wird seine Priorität auf 15 angehoben und sein Quantum verdoppelt (Windows 2000/XP) oder sogar vervierfacht (Windows 2003). Nachdem er die CPU erhalten hat, wird er wieder auf den alten Zustand gesetzt. Ein Verhungern von Prozessen wird damit vermieden. Diese Prioritäts- und Quantumserhöhung wird bei jeder Überprüfung aber nur jeweils für eine begrenzte Anzahl an Threads durchgeführt. Nachdem 10 Threads angehoben wurden wird der Vorgang abgebrochen. Beim nächsten Mal werden die weiteren Threads berücksichtigt (siehe Abbildung 5-16).

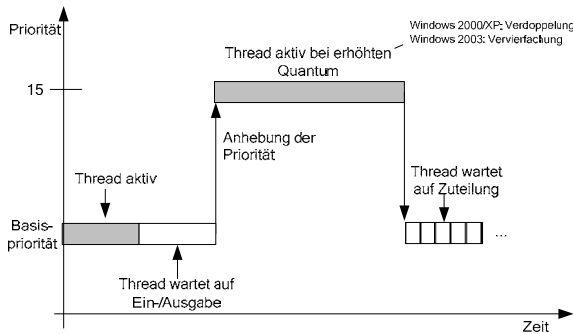


Abbildung 5-16: Rettung verhungender Prozesse durch Prioritäts-/Quantumsanhebung

Ready-Queue-Verwaltung. Die Verwaltung der Threads erfolgt bei Windows in einer *Multi-Level-Feedback-Warteschlange* wie sie in Abbildung 5-17 dargestellt ist. Die verwendete Datenstruktur wird als Ready-Queue bezeichnet. Für jede interne Kernelpriorität wird eine Queue verwaltet, deren Anker in der Ready-Queue-Datenstruktur gespeichert ist. Innerhalb der einzelnen Queues werden die Threads mit gleicher Priorität über eine RR-Strategie zugeteilt. Nach Ablauf des Zeitquantums wird ein aktiver Thread an das Ende einer Queue gehängt.

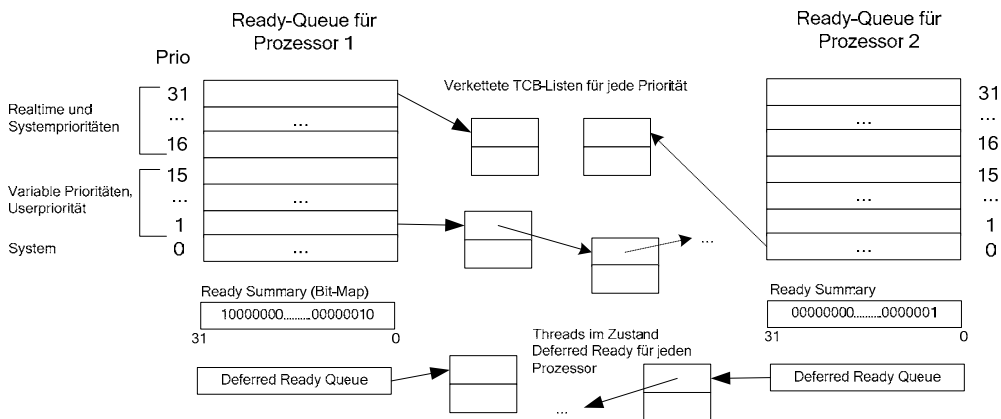


Abbildung 5-17: Multi-Level-Feedback-Queue in Windows

Wie die Abbildung zeigt, ist je Prozessor eine Ready-Queue-Datenstruktur zugeordnet. Weiterhin verwaltet Windows zu jedem Prozessor eine Bit-Maske, in der zu jeder der 32 Queues ein Bit angibt, ob mindestens ein Thread in die Queue eingetragen ist. Seit Windows 2003 wird auch noch eine weitere Warteschlange je Prozessor verwaltet. Dies ist die sog. *Deferred-Ready-Queue*. Alle Threads, die sich

im Zustand *Deferred Ready* (siehe Kapitel 4) befinden, werden in diese Warteschlange eingetragen. Durch die prozessorspezifische Trennung der Queues kann unabhängig voneinander geprüft werden, ob Threads auf einen Prozessor warten. Damit verringert man in Mehrprozessorsystemen die Lockzeiten beim Zugriff auf die Ready-Queue-Datenstruktur.

5.3.4 Scheduling von Threads in Java

Java-Threads laufen in der JVM, also im Java-Laufzeitsystem, ab und erhalten von dieser eine Zuteilung von Rechenzeit. Wie wir bereits gesehen haben, ist der Zustandsautomat für Java-Threads recht einfach (siehe auch Kapitel 4).

Das Java-Thread-Scheduling ist prioritätengesteuert. Einsatzbereite Threads können im RR-Verfahren zugeteilt werden, was aber letztendlich in der Implementierung festgelegt werden kann. Es wird auch eine Zeitscheibe eingesetzt, deren Länge implementierungsabhängig ist. Um sicher zu gehen, dass ein rechenintensiver Thread nicht alle anderen ausbremst, sollte man bei der Programmierung von Multithreading-Anwendungen an entsprechenden Stellen die Methode *yield* aus der Klasse *Thread* aufrufen. Diese Methode bewirkt, dass die CPU dem nächsten ablaufbereiten („runnable“) Thread zugeteilt wird.

In Java gibt es die Prioritätsstufen 1 bis 10, wobei 10 die höchste Priorität darstellt. Für jede Priorität werden die Threads in einer eigenen Queue verwaltet. Als Standardpriorität wird bei der Thread-Erzeugung 5 vergeben. Die Thread-Prioritäten kann man mit der Methode *setPriority* der Klasse *Thread* auch verändern bzw. mit der Methode *getPriority* auslesen. Ein Thread kann nur die vier Zustände *new*, *runnable*, *blocked* und *dead* einnehmen.

Die Zuteilungsregeln sind in der JVM-Spezifikation von Sun Microsystems nicht exakt festgelegt, sondern der Implementierung überlassen. Es ist nur festgelegt, dass ein Thread mit höherer Priorität vor den anderen bevorzugt werden soll, jedoch gibt es keine Garantie hierfür. Der JVM-Implementierer entscheidet, wie die Regeln implementiert werden, was die Portierung von Java-Programmen von einer JVM auf eine andere (eines anderen Herstellers) damit unter Umständen erschwert. Auch die Abbildung der Java-Prioritäten auf Betriebssystemprioritäten ist in der Java-Spezifikation nicht festgelegt.

5.3.5 Zusammenfassung

Zusammenfassend kann festgehalten werden, dass in heutigen Mehrzweckbetriebssystemen aber auch in Laufzeitsystemen von Programmiersprachen wie Java überwiegend Varianten von Zeitscheibenverfahren eingesetzt werden, die Round-

Robin- und Priority-Scheduling nutzen. Meist wird zur Verwaltung mehrerer Prioritäten auch noch eine Multi-Level-Feedback-Queue eingesetzt. Dabei spielt es keine Rolle, ob der Prozess oder der Thread als Scheduling-Einheit dient.

Tabelle 3-2: Ein Vergleich des CPU-Scheduling in Windows und Linux

Kriterium	Windows	Linux
Unterstützte Prozess-typen	Timesharing und Echtzeitprozesse; Unterscheidung über Priorität	Timesharing, Realtime mit FIFO, Realtime mit RR; Unterscheidung über Priorität
Prioritätsstufen	Interne Kernelprioritätsstufen zwischen 0 – 31 (31 ist höchste): • 0 – 15: Timesharing-Threads • 16 – 31: Realtime- und System-Threads Eigene WinAPI-Prioritätsstufen mit Basisprioritäten für Prozesse und Relativprioritäten für Threads	Statische und effektive Prioritäten mit Stufen von 0 – 139 (139 ist niedrigste): • 0 – 99: Realtime-Prozesse • 100 – 139: Timesharing-Prozesse
Scheduling-Strategie für Timesharing- Prozesse	Thread-basiert, Begünstigung von interaktiven vor rechenintensiven Threads; Quanten, prioritätsgesteuert, Round Robin, Multi-Level- Feedback-Queue, 32 Queues	Prozess-basiert, Begünstigung von interaktiven vor rechenintensiven Prozessen (Thread = Prozess); Quanten, prioritätsgesteuert, Round Robin, Multi-Level- Feedback-Queue, 140 Queues
Prioritätenberechnung für Timesharing- Prozesse	Aktuelle Priorität wird zur Laufzeit berechnet; Priority-Boost bei wartenden Threads nach Wartezeit oder bei GUI-Threads	Effektive Priorität wird zur Laufzeit berechnet, abhängig von statischer Priorität und einem Bonus (+/- 5) für lang schlafende Prozesse; Effektive Prio = statische Prio + Bonus
Quantumsberechnung für Timesharing- Prozesse	Quantumszähler = 6 bei Workstati- ons und 36 bei Windows Servern, bei jedem Clock-Intervall wird Zähler um 3 dekrementiert; Clock-Intervall ist ca. 10 ms bei x86- Singleprozessoren (100 Hz) und ca. 15 ms (67 Hz) bei Multiprozessoren; Quantumserhöhung bei GUI- Threads oder zur Vorbeugung vor Verhungern von Threads	Abhängig von statischer Priorität; Quantum = (140 - statische Prio) * 20 [ms] (oder * 5, je nach Priorität); Maximum bei 800 ms (Linux 2.6); Takt der internen Systemuhr bis Linux 2.5 auf 100 Hz einstellbar, ab Linux 2.6 auf 100, 250, 300 und 1000 Hz; bei jedem Tick wird das Quantum um das Clock-Intervall reduziert

Die Prioritäten und die Quanten werden üblicherweise dynamisch nach sehr individuellen Algorithmen ermittelt. Interaktive Prozesse werden bei der CPU-Zuteilung vor den eher rechenintensiven (Batch-orientierten) Prozessen oder Threads bevorzugt. In Tabelle 3-2 ist ein abschließender Vergleich der CPU-Scheduling-Mechanismen der Betriebssysteme Windows und Linux anhand einiger Vergleichskriterien skizziert. Im Fokus des Vergleichs stehen Timesharing-Prozesse. Man kann im Wesentlichen große Übereinstimmungen erkennen.

5.4 Übungsaufgaben

1. Welche Scheduling-Algorithmen sind für Echtzeitbetriebssysteme (Realtime-System) sinnvoll und warum?
2. Welche Aufgaben haben im Prozess-Management der Dispatcher und der Scheduler?
3. Nennen Sie jeweils zwei geeignete Scheduling-Verfahren für Batch- und Dialog-Systeme und erläutern Sie diese kurz!
4. Erläutern Sie den Unterschied zwischen preemptive und non-preemptive Scheduling und nennen Sie jeweils zwei Scheduling-Strategien, die in diese Kategorien passen.
5. Ermitteln Sie für folgende fünf Jobs, die gesamte Verweilzeit und die durchschnittliche Verweilzeit unter Berücksichtigung folgender Scheduling-Strategien (Angaben in ms).

Job	A	B	C	D	E
Ablaufzeit	8	12	20	16	5
Priorität	5	4	3	2	1

- a) Reines Priority Scheduling (höchste Priorität ist 5)
- b) FCFS (First Come First Serve) unter Berücksichtigung der Reihenfolge-Annahme: A, B, D, C, E
- c) SJF (Shortest Job First)
- d) RR (Round Robin) ohne Prioritäten bei einem Quantum von 2 ms, wobei die Reihenfolge der Abarbeitung A, B, C, D, E sein soll

Die Jobs treffen ungefähr gleichzeitig im System ein. Die reine Prozesswechselzeit wird für die Berechnung vernachlässigt, und die Aufträge werden nacheinander ausgeführt. Eine Verdrängung (Preemption) wird nur im Fall d) ausgeführt.

6. Beweisen Sie, dass die SJF-Strategie für das Scheduling die optimale Strategie darstellt!
7. Nennen Sie Vergleichskriterien, nach denen Scheduling-Algorithmen verglichen werden können und erläutern Sie diese!

8. Betrachten Sie folgende Aufträge an ein Betriebssystem mit ihren Ausführungszeiten (in ms):

Job	A	B	C	D	E
Ablaufzeit	8	27	1	5	10

Berechnen Sie für die folgenden Scheduling-Algorithmen die durchschnittliche Wartezeit für die Ausführung:

- FCFS in der Reihenfolge A, B, D, C, E
- SJF (wie das System dies ermittelt, ist nicht von Belang für diese Aufgabe)
- RR mit einem Quantum von 2 ms in der Reihenfolge A, B, D, C, E

Die Jobs treffen ungefähr gleichzeitig im System ein. Der Overhead für den Prozesswechsel soll nicht betrachtet werden.

- Warum sind Windows und Linux keine echten Realtime-Systeme, obwohl sie Realtime-Threads unterstützen?
- Wie wird unter Windows verhindert, dass Threads mit niedriger Priorität verhungern?
- Wie funktioniert der RR-Scheduling-Algorithmus?
- Warum ist der Scheduling-Algorithmus Shortest Remaining Time First (SRTF) kaum zu realisieren?
- Erläutern Sie den Aufbau und die Nutzung der Datenstruktur *Run-Queue* unter Linux!
- Welche Bedeutung haben die statische und effektive (dynamische) Priorität, der Bonus und das Quantum für das Linux-Scheduling?
- Wie ermittelt der Linux-Scheduler den nächsten zu aktivierenden Prozess?
- Was ist ein Priority Boost unter Windows?

6 Synchronisation und Kommunikation

Bei der Parallelverarbeitung mit gleichzeitiger Nutzung gemeinsamer Betriebsmittel durch Prozesse bzw. durch Threads sind einige Herausforderungen zu bewältigen. Wenn man Prozesse oder Threads ohne Abstimmung mit gemeinsam genutzten Betriebsmitteln wie z.B. gemeinsam genutzte Speicherbereiche, arbeiten lässt, kann es zu Inkonsistenzen oder sog. Race Conditions kommen. Lost-Updates oder andere Anomalien können die Folge sein.

Derartige Aufgabenstellungen erfordern die Definition sog. atomarer Aktionen, die nicht unterbrochen werden dürfen. Zugriffe auf gemeinsame Betriebsmittel werden in sog. kritischen (Code-)Abschnitten gekapselt, wofür in der Informatik einige bekannte Konzepte verfügbar sind. Hierzu gehören sog. Synchronisationstechniken wie Locks (Sperrern), Semaphore, Mutexe und Monitore.

In diesem Kapitel wird zunächst die Problemstellung erläutert. Danach werden einige wichtige Lösungskonzepte vorgestellt und anhand von konkreten Implementierungen in den Sprachen Java und C# vertieft. Einfache Beispiele wie das Counter-Problem und das Philosophenproblem werden zur Erläuterung herangezogen.

Anschließend werden einige allgemeine Grundbegriffe der Kommunikation zwischen nebenläufigen Prozessen oder Threads erläutert sowie einige Möglichkeiten der internen Kommunikation dargestellt und am Fallbeispiel von *Pipes* vertieft.

Zielsetzung des Kapitels

Der Studierende soll die Probleme der Parallelverarbeitung verstehen und einschätzen sowie Konzepte zur Vermeidung von Race Conditions sowohl auf Betriebssystemebene als auch auf Anwendungsebene erläutern können. Weiterhin soll er moderne Konzepte von Programmiersprachen zur Unterstützung der Synchronisierung nebenläufiger Prozesse oder Threads verstehen und einschätzen können. Zudem soll der Studierende einen grundlegenden Überblick über Begriffe der lokalen (rechnerinternen) Kommunikation erhalten.

Wichtige Begriffe

Race Condition, Kritischer Abschnitt und wechselseitiger Ausschluss, TSL-Befehl, Lock (Sperre), Spinlock, Semaphore und Mutex, Monitor, Deadlock, Betriebsmit-

telgraphen, halbduplex, duplex, synchron, asynchron, verbindungsorientiert, verbindungslos, Unicast, Multicast, Anycast, Broadcast.

6.1 Grundlegendes zur Synchronisation

6.1.1 Nebenläufigkeit, atomare Aktionen und Race Conditions

Der Begriff *Nebenläufigkeit* bezeichnet bei Betriebssystemen ganz allgemein die parallele oder quasi-parallele Ausführung von Befehlen auf einer CPU oder mehreren CPUs bzw. Rechenkernen. Prozesse bzw. Threads werden in Multiprogramming-Systemen nebenläufig ausgeführt. Durch verdrängende Scheduling-Verfahren kann ein Prozess bzw. Thread jederzeit suspendiert und ein anderer aktiviert werden. Es obliegt dem Betriebssystem, welcher Befehl von welchem Prozess als nächstes ausgeführt wird, was zu einem gewissen Nichtdeterminismus führt. Der Programmierer einer Anwendung kann dies jedenfalls nicht beeinflussen.

Nun kommt es aber häufig vor, dass gewisse Codeteile des Betriebssystems oder auch von Anwendungen nicht unterbrochen bzw. zumindest nicht bei der Abarbeitung beeinflusst werden sollten, da es sonst evtl. zu Inkonsistenzen kommen kann. Solche Codebereiche können als atomare Aktionen aufgefasst werden. Typisch sind diese bei der Bearbeitung gemeinsamer Datenstrukturen oder Dateien (shared data structures, shared files) und auch bei der gemeinsamen Nutzung von Hardware-Komponenten (shared hardware) wie Drucker.

Atomare Aktionen sind also Codebereiche, die in einem Stück, also atomar, ausgeführt werden müssen und (logisch) nicht unterbrochen werden dürfen. Die im Rahmen einer atomaren Aktion bearbeiteten Ressourcen (Objekte, Datenbereiche) müssen dem Prozess bzw. Thread, der die atomare Aktion ausführt, exklusiv zugeordnet sein und dürfen ihm nicht entzogen werden. Eine Unterbrechung durch eine Scheduling-Entscheidung des Betriebssystems ist aber jederzeit möglich. Die Problematik soll anhand von zwei einfachen Beispielen erläutert werden:

Beispiel 1: In Betriebssystemen werden oftmals verkettete Listen z.B. von PCBs (Prozessliste eines Schedulers) verwaltet, die von mehreren Prozessen zugreifbar sind. Hängt ein Prozess nun ein neues Objekt am Anfang der Liste ein und wird dabei durch einen anderen Prozess, der das gleiche tun möchte, unterbrochen, so kann es möglicherweise zu einer Inkonsistenz kommen.

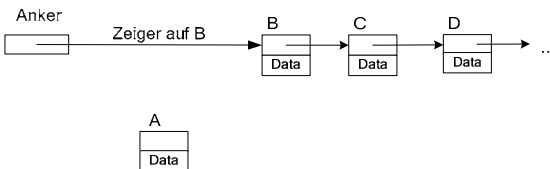
Das Einhängen in die Liste besteht aus zwei Operationen, die aufgrund einer Scheduling-Entscheidung des Betriebssystems jederzeit unterbrechbar sind. Ist A

das neue Objekt, so sind dies die beiden folgenden Operationen, wobei *Anker* die Anfangsadresse der Liste ist:

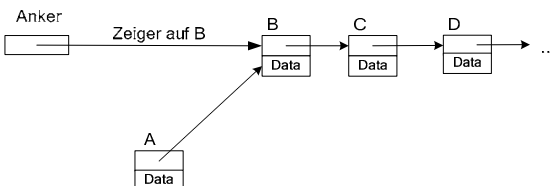
```
A.next := Anker;  
Anker := Adresse(A);
```

Wird die aus den zwei Operationen bestehende zusammengehörige Aktion nach der ersten Operation bzw. während einer der beiden Operationen unterbrochen, so kann ein anderer Prozess, der die gleichen Variablen verwendet, leicht ein Problem verursachen. Der Grund ist, dass die zusammengehörigen Operationen nicht atomar sind, da hierzu mehrere Maschinenbefehle ausgeführt werden, nach denen jeweils eine Unterbrechung stattfinden kann. Daher müssen die zwei Operationen in einer atomaren Aktion (ganz oder gar nicht) ohne Beeinflussung von nebenläufigen Prozessen ausgeführt werden (siehe Abbildung 6-1).

(1) Ursprungsstand: A erzeugt, aber nicht eingehängt



(2) Nach $A.next := Anker$



(3) Nach $Anker := Adresse(A)$

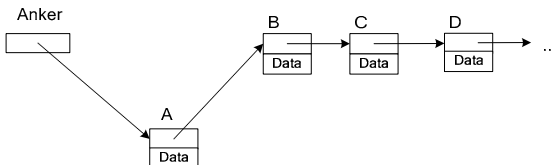


Abbildung 6-1: Einhängen eines neuen Elementes an den Kopf einer verketteten Liste

Ein Problem ergibt sich z.B. bei folgender Ablaufsequenz, wenn die neuen Elemente A_1 und A_2 eingehängt werden sollen, die Operationen aber verzahnt ausgeführt werden:

```
A1.next := Anker;
```

```
A2.next := Anker;  
Anker := Adresse(A1);  
Anker := Adresse(A2);
```

In diesem Fall verliert man das Element A_1 .

Beispiel 2: Ein weiteres, einfaches aber sehr einleuchtendes Beispiel, wie es zu einem Problem durch nebenläufige Ausführung eines Programms kommen kann, ist das Inkrementieren eines einfachen Zählers.

Die Erhöhung eines Zählers in einer nebenläufig bearbeiteten Variablen (also einem Speicherbereich in einem durch mehrere Prozesse bzw. Threads zugänglichen Adressbereich) kann nämlich nicht in einem Maschinenzklus abgearbeitet werden und ist daher unterbrechbar. Wenn nun, wie in Abbildung 6-2 dargestellt, Prozess A den Zähler (in ein Register) einliest und danach unterbrochen wird, dann Prozess B ihn ebenfalls liest, im Anschluss daran (im Register) erhöht und auf den Speicherplatz zurückschreibt (vom Register in den Hauptspeicher), kann die Änderung von B verloren gehen. Diese Inkonsistenz wird auch als *Lost Update* bezeichnet und ist ein Synchronisationsproblem, das nicht nur bei Betriebssystemen, sondern auch im Bereich der Datenbanken häufig diskutiert wird.

Geht man im Beispiel davon aus, dass der Zähler (counter) anfangs auf 0 stand, so steht er nach Abarbeitung der Befehle durch die Prozesse A und B auf 1, sollte aber eigentlich auf 6 stehen. Die Erhöhung durch Prozess B um 5 geht verloren.

Das Lesen, Erhöhen und Zurückschreiben des Zählers müsste in einer atomaren Aktion ausgeführt werden. Das Lost-Update-Problem wird umso akuter, je mehr Prozesse/Threads um den Zähler konkurrieren.

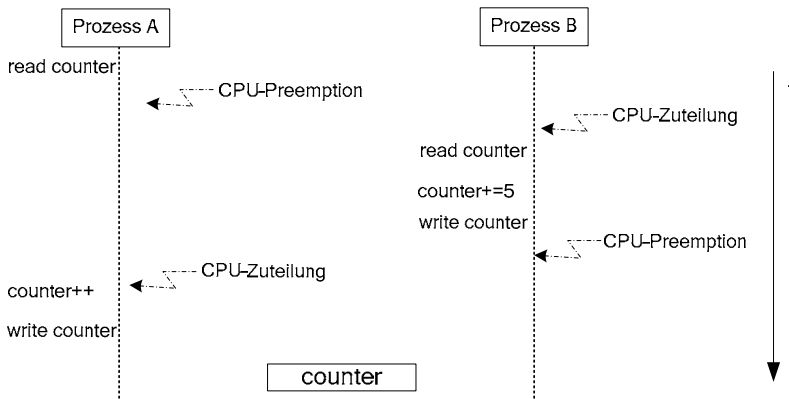


Abbildung 6-2: Lost-Update-Problem beim Inkrementieren eines Zählers

Die erläuterten Probleme treten auf, weil mehrere Prozesse unkontrolliert auf gemeinsam genutzte Ressourcen zugreifen. Man lässt den Prozessen freien Lauf beim Zugriff, und es ist nicht vorhersehbar, wann der Scheduler einen laufenden Prozess unterbricht und einem anderen die CPU zuteilt. Dies kann in kritischen Situationen erfolgen, ohne dass man es beeinflussen kann. Diese Situation bezeichnet man in der Informatik auch als *Race Condition* oder *zeitkritischen Ablauf*.

Race Conditions sind also Situationen, bei denen zwei oder mehr Prozesse gemeinsame Betriebsmittel nutzen und die Endergebnisse der Nutzung von der zeitlichen Reihenfolge der Operationen abhängen (Ehse 2005). Diese Situationen muss man in den Griff bekommen, was z.B. durch eine Kontrolle über die Ausführungsreihenfolge erfolgen kann.

6.1.2 Kritische Abschnitte und wechselseitiger Ausschluss

Programmteile, die nicht unterbrochen werden dürfen, werden auch als *kritische Abschnitte* bezeichnet. Ein kritischer Abschnitt ist ein Codeabschnitt, der zu einer Zeit nur durch einen Prozess bzw. Thread durchlaufen und in dieser Zeit nicht durch andere nebenläufige Prozesse bzw. Threads betreten werden darf. Ein Prozess bzw. Thread, der einen kritischen Abschnitt betritt, darf nicht unterbrochen werden. Sofern das Betriebssystem in dieser Zeit aufgrund einer Scheduling-Entscheidung eine Unterbrechung zulässt, darf der Prozess bzw. Thread, der den kritischen Abschnitt belegt, nicht beeinflusst werden.

Um dies zu erreichen und um Inkonsistenzen zu vermeiden, muss ein kritischer Abschnitt geschützt werden. Dies kann durch *gegenseitigen (oder wechselseitigen) Ausschluss* (engl.: *mutual exclusion*) erreicht werden. Prozesse, die einen kritischen Abschnitt ausführen wollen, müssen warten, bis dieser frei ist. Mit einem wechselseitigen Ausschluss wird also die Illusion einer atomaren Anweisungsfolge geschaffen, denn echt atomar wird sie natürlich nicht ausgeführt. Es kann ja immer noch vorkommen, dass ein nebenläufiger Prozess zwischendurch die CPU erhält.

Es gibt mehrere Möglichkeiten, einen kritischen Abschnitt zu implementieren. Das klassische *Busy Waiting*, also das Warten und ständige Abfragen eines Sperrkennzeichens am Eingang des kritischen Abschnitts, das freigegeben werden muss, bevor man den kritischen Abschnitt betreten kann, ist ineffizient und führt zu einer Verschwendung von CPU-Zeit. Effizienter ist es, einen Prozess, der vor einem kritischen Abschnitt warten muss, schlafen zu legen und erst wieder aufzuwecken, wenn er diesen betreten darf. Hierzu braucht man einen geeigneten Mechanismus, der dies effektiv unterstützt.

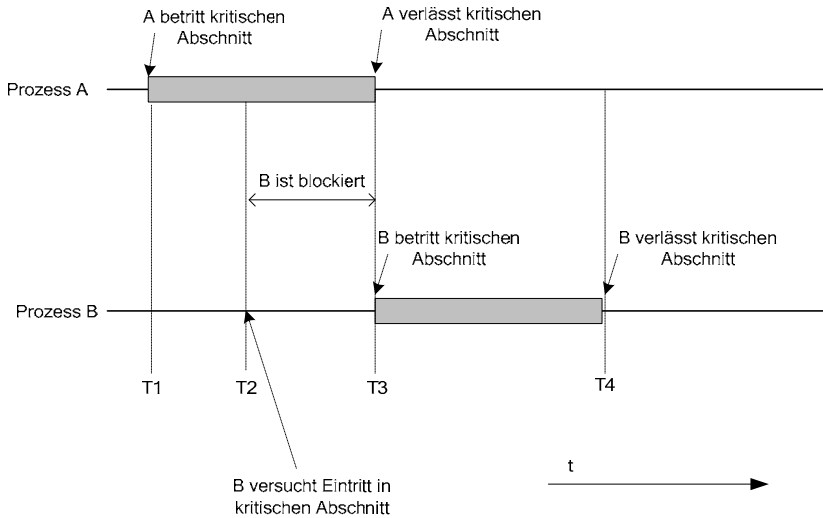


Abbildung 4-3: Gegenseitiger Ausschluss über kritische Abschnitte

In Abbildung 4-3 ist ein klassischer gegenseitiger Ausschluss am Beispiel zweier konkurrierender Prozesse A und B skizziert. Prozess B muss bis zum Zeitpunkt T3 warten, obwohl er schon zum Zeitpunkt T2 bereit wäre. Dann erst darf er den kritischen Abschnitt, den Prozess A zu diesem Zeitpunkt verlässt, betreten. Zu einer Zeit ist also immer nur einer der Prozesse im kritischen Abschnitt.

Nach Dijkstra¹ ist bei kritischen Abschnitten folgendes zu beachten:

- *Mutual exclusion*: Zwei oder mehr Prozesse dürfen sich nicht gleichzeitig im gleichen kritischen Abschnitt befinden.
- Es dürfen *keine Annahmen über die Abarbeitungsgeschwindigkeit und die Anzahl der Prozesse bzw. Prozessoren* gemacht werden. Der kritische Abschnitt muss unabhängig davon geschützt werden.
- Kein Prozess außerhalb eines kritischen Abschnitts darf einen anderen nebenläufigen Prozess *blockieren*.
- *Fairness Condition*: Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen (kein ewiges Warten).

Kritische Abschnitte können sehr anschaulich mit Petrinetzen modelliert werden. Abbildung 6-4 zeigt ein Beispiel-Petrinetz zur Visualisierung eines gegenseitigen Ausschlusses für zwei kritische Abschnitte.

¹ Edsger Wybe Dijkstra (geboren am 11.5.1930, gestorben am 06.08.2002), Niederländischer Computer-Wissenschaftler.

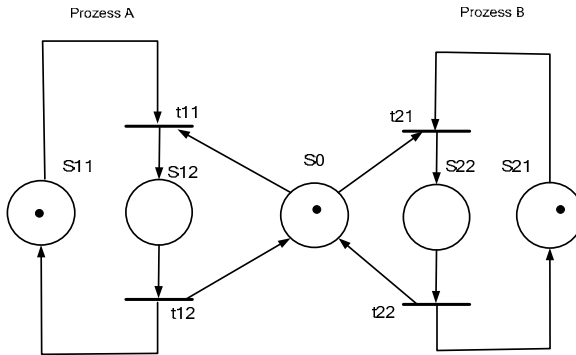


Abbildung 6-4: Beispiel-Petrinetz mit kritischen Abschnitten

Interpretation. Prozess A und Prozess B laufen nebenläufig ab. Die Stelle S_0 realisiert den gegenseitigen Ausschluss, da die Transitionen t_{11} und t_{21} nur alternativ, also einander ausschließend schalten können. Die Eingänge in den kritischen Abschnitt sind die Transitionen t_{11} und t_{21} , die Ausgänge sind t_{12} und t_{22} . S_{12} und S_{22} stellen kritische Abschnitte, die Stellen S_{11} und S_{21} stellen unkritische Abschnitte dar. Ein Prozess, der in den kritischen Abschnitt gelassen werden soll, muss zwei Tokens erhalten. A kann z.B. die Transition t_{11} nur schalten, wenn er neben dem Token aus Stelle S_{11} auch das Token von Stelle S_0 erhält. Es handelt sich im Beispiel allerdings nicht um eine faire Lösung, da es sein kann, dass einer der beiden Prozesse nie in den kritischen Abschnitt gelangt.

Zur Behandlung des Problems des wechselseitigen Ausschlusses gibt es Software- und Hardwarelösungen. In Betriebssystemen sind überwiegend Lösungen mit Unterstützung der Hardware vorgesehen. In der Regel verwendet man in modernen Systemen spezielle Hardwarebefehle zur Unterstützung der Implementierung des gegenseitigen Ausschlusses wie etwa den weiter unten erläuterten TSL-Befehl, der in mehreren Varianten implementiert wurde. Auf diesem Befehlstyp basieren auch die im Weiteren erläuterten höheren Sprachkonstrukte.

6.1.3 Eigenschaften nebenläufiger Programme

Bei nebenläufiger Ausführung von Programmteilen kann es zu verschiedensten Problemen kommen. Prozesse können beispielsweise blockieren, sie können verhungern oder es kann eine Verklemmung (Deadlock) eintreten. Diese Situationen werden im Folgenden kurz erläutert, wobei die Betrachtung sowohl für Prozesse als auch für Threads gleichermaßen gilt.

Blockieren. Ein Prozess P_1 belegt ein Betriebsmittel, ein zweiter Prozess P_2 benötigt dasselbe Betriebsmittel ebenfalls und wird daher blockiert, bis es P_1 freigegeben hat.

Verhungern (Starvation). Ein Prozess erhält trotz Rechenbereitschaft keine CPU-Zeit zugeteilt, z.B. weil ihm immer wieder Prozesse mit höherer Priorität vorgezogen werden.

Verklemmung. Zwei oder mehr Prozesse halten jeder für sich ein oder mehrere Betriebsmittel belegt und versuchen ein weiteres zu belegen, das aber von dem anderen Prozess belegt ist. Es liegt ein Zyklus von Abhängigkeiten vor. Kein Prozess gibt seine Betriebsmittel frei, und alle Prozesse warten daher ewig. Dieser Zustand wird auch als *Deadlock* bezeichnet.

Diese Situationen müssen natürlich vermieden werden. Zur näheren Beschreibung der Korrektheit von nebenläufigen Programmen nutzt man die Begriffe *Sicherheit* und *Lebendigkeit*. Nach Kredel ist ein Programmteil sicher, falls keine fehlerhaften Zustände eintreten (Kredel 1999). Sicherheit bedeutet auch, dass keine Verklemmungen vorkommen. Ein Programmteil ist lebendig, falls irgendwann alle gewünschten Zustände eintreten. Ein Programmteil ist lebendig, wenn er eine faire Chance für die Durchführung erhält. Hierzu muss die Scheduling-Strategie fair sein.

Ein Deadlock kann nur eintreten, wenn folgende vier Bedingungen eintreffen:

- Mutual Exclusion für die benötigten Betriebsmittel
- Prozesse belegen Betriebsmittel und fordern weitere an
- Kein Entzug eines Betriebsmittels ist möglich
- Zwei oder mehrere Prozesse warten in einer Warteschleife (circular waiting) auf weitere Betriebsmittel

In Abbildung 6-5 ist ein Petrinetz für einen Deadlock dargestellt. Wenn beide Tokens gleichzeitig für die Transitionen t_1 und t_2 verschossen werden, entsteht ein Deadlock, da im Folgenden weder die Transition t_3 noch t_4 schalten kann.

Deadlocks lassen sich a priori deshalb nur schwer vermeiden, weil sonst nur jeder Prozess ohne Gefahr eines Deadlocks zum Ablauf kommen dürfte. Dies könnte man beispielsweise erreichen, indem man alle Betriebsmittel im Vorfeld reserviert.

Eine in der Praxis häufig eingesetzte Technik ist das Erkennen und Beseitigen von Deadlocks zur Laufzeit, wobei man hierzu sog. Betriebsmittelbelegungsgraphen einsetzt. Dies sind Graphen, die als Knoten Ressourcen und Prozesse/Threads enthalten, und Kanten, welche die Belegung und Anforderung der Ressourcen durch Prozesse/Threads aufzeigen. Als Maßnahmen zur Beseitigung eines Dead-

locks sind das Abbrechen eines Prozesses/Threads oder das Entziehen eines Betriebsmittels möglich.

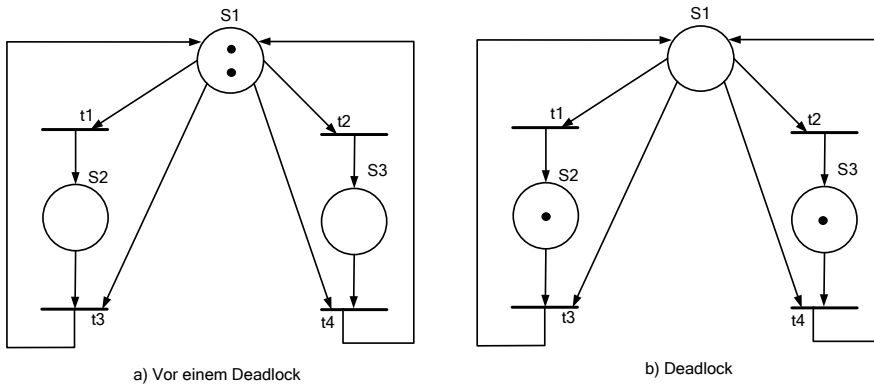


Abbildung 6-5: Petrinetz für einen Deadlock

6.2 Synchronisationskonzepte

6.2.1 Sperren

Wie kann man nun verhindern, dass mehrere Prozesse einen kritischen Abschnitt betreten? Ein einfaches, aber in Betriebssystemen häufig verwendetes Instrument ist die Sperre (auch als *Lock* bezeichnet). Eine Sperre kann man implementieren, indem man z.B. eine Variable (Sperrvariable) einführt, die nur die Werte 0 oder 1 annehmen kann. Ist der Wert der Sperrvariable 1, so gilt die Sperre als belegt, ist der Wert 0, so ist die Sperre nicht belegt. Will nun ein Prozess in den kritischen Abschnitt, muss er die Sperrvariable auf 1 setzen. Dies geht aber nur, wenn sie gerade einen Wert von 0 aufweist. Daher braucht der Prozess zwei Operationen, um die Sperrvariable zu testen und – wenn sie auf 0 steht – auch zu setzen. Problematisch ist, dass die Ausführung dieser beiden Befehle durch eine CPU-Scheduling-Entscheidung unterbrochen werden kann. Ein Fehlerszenario wäre z.B., dass der Prozess, der die Sperrvariable setzen möchte, diese zunächst ausliest, um sie zu testen, und gleich danach verdrängt wird. Wenn jetzt ein anderer Prozess, der die CPU erhält, das Gleiche versucht, könnte er die Sperrvariable testen und auch gleich setzen. Kommt danach der erste Prozess wieder an die Reihe, würde auch er die Sperrvariable im nächsten Schritt setzen, da er immer noch annehmen würde, dass sie auf 0 steht. In diesem Fall würden beide Prozesse, trotz Sperrmechanismus, den kritischen Abschnitt betreten und es käme zu einer Inkonsistenz.

Spinlocks. Wenn in einer Warteschleife immer wieder versucht wird, eine Sperre zu erhalten, bis sie gewährt wird, spricht man auch von einem *Spinlock*. Spin²-Locks werden in Betriebssystemen bei sehr kurzen Wartezeiten verwendet. Zum Setzen der Sperre wird eine Sperrvariable gesetzt. Der wartende Prozess gibt die CPU nicht vor Ablauf seines Quantums frei und verlässt die Schleife erst, wenn die Sperrvariable gesetzt werden konnte. Eine Warteschlange von Prozessen vor der Sperrvariablen wird nicht unterstützt. Spinlocks sind in Betriebssystemen manchmal eine effiziente Methode zum Schützen von gemeinsam genutzten Datenstrukturen, sind aber nicht bei Einprozessormaschinen sinnvoll, da während der Ausführung der Warteschleife kein anderer Prozess oder Thread an der Sperrvariable etwas verändern kann. Spinlocks dienen also als effizienter Sperrmechanismus in Mehrprozessorsystemen.

Reine Softwarelösungen für Sperren. Es gibt einige reine Softwarelösungen zum Setzen von Sperren für kritische Abschnitte. Hier sei z.B. auf die Lösungen von Dekker und Peterson verwiesen (Tanenbaum 2009). Wir wollen diese nicht weiter vertiefen, da sie in Betriebssystemen keine große Rolle spielen.

Interrupts sperren. Eine andere Möglichkeit der Implementierung von Sperren ist das Maskieren aller Interrupts, während ein kritischer Abschnitt ausgeführt wird. In diesem Fall kann der gerade laufende Prozess nicht mehr durch eine CPU-Scheduling-Entscheidung unterbrochen werden. Diese Hardwarelösung ist zwar sehr effektiv, allerdings kann es dabei passieren, dass wichtige Interrupts verlorengelassen werden. Bei Multiprozessormaschinen müssten auch alle CPUs für die Zeit, in dem sich ein Prozess in einem kritischen Abschnitt befindet, Interrupts verbieten. In der Praxis wird diese Methode daher selten eingesetzt.

Hardwareunterstützung durch spezielle Maschinenbefehle. Wie oben erläutert, bedarf es zum Testen und Setzen einer Sperre zweier Aktionen (Testen und Setzen). Wenn man beide Aktionen in einem Maschinenbefehl ununterbrechbar ausführen könnte, wäre das die Lösung. Tatsächlich unterstützen die meisten Prozessoren einen derartigen Befehl, der meist als *TSL* (*Test and Set Lock*) bezeichnet wird. Auf Intel-basierten Maschinen gibt es hierfür z.B. den Befehl *XCHG* zum Austausch der Inhalte zweier Speicherbereiche. Der *TSL*- und der *XCHG*-Befehl arbeiten so, dass die Operationen Lesen und Ersetzen einer Speicherzelle ununterbrechbar (atomar) ausgeführt werden. Mit diesem Befehl lässt sich also das atomare Setzen einer Sperre implementieren.

² Engl. to spin = schnell drehen.

Bei Mehrprozessorsystemen könnte es aber auch hier zu einem Problem kommen, wenn z.B. zwei CPUs einen TSL-Befehl ausführen und die Teiloperationen sich überlappen. Allerdings wird die Ausführung eines TSL-Befehls in einem Speicherzugriffszyklus erledigt. Daher funktioniert der Befehl auch bei Multiprocessing. Die CPU, welche die TSL-Instruktion anstößt, blockiert den Speicherbus solange, bis die Instruktion vollständig ausgeführt ist.

Im Folgenden ist eine Lock-Implementierung mit dem TSL-Befehl skizziert, die kurz besprochen werden soll:

MyLock_lock:

```
TSL R1, LOCK // Lies LOCK und setze Wert von LOCK auf 1
CMP R1, #0    // Vergleiche Registerinhalt von R1 mit 0
JNE MyLock_lock // Erneut versuchen, falls Lock nicht gesetzt
                // werden konnte
RET           // Kritischer Abschnitt kann betreten werden
```

MyLock_unlock:

```
MOVE LOCK, #0 // LOCK auf 0 setzen (freigeben)
RET           // Kritischer Abschnitt kann von anderem
                // Prozess betreten werden
```

Interpretation. Die Implementierungsvariante zeigt zwei die Operationen *MyLock_lock* und *MyLock_unlock*:

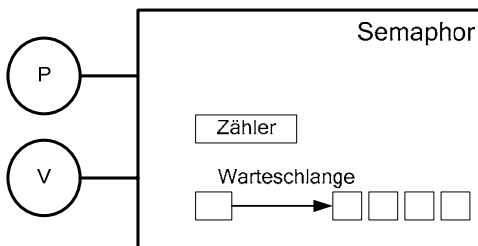
- Die Operation *MyLock_lock* dient dem Testen und Setzen der Sperre, die Operation *MyLock_unlock* dient dem Freigeben der Sperre.
- Das Setzen geschieht über einen TSL-Befehl, in dem eine Speicherzelle namens LOCK gelesen und, sofern sie 0 ist, auch gleich auf 1 gesetzt wird. Das Register R1 und die Speicherzelle mit dem symbolischen Namen LOCK können die Werte 0 und 1 annehmen. Der Inhalt der im Hauptspeicher liegenden Zelle LOCK wird mit dem TSL-Befehl in ein Register R1 geladen, wobei der Wert der Zelle auf 1 verändert wird.
- Das Register R1 (frei gewählte Bezeichnung) enthält nach der Operationsausführung den alten Wert der Speicherzelle, und dieser Wert kann nach dem Ausführen des TSL-Befehls dahingehend überprüft werden, ob der Lock schon vorher gesetzt war oder nicht. Wenn R1 = 1 ist, dann ist die Sperre schon gesetzt gewesen, 0 bedeutet, der laufende Prozess erhält die Sperre.
- Wenn der laufende Prozess die Sperre nicht bekommen hat, versucht er es erneut. Hat der Prozess die Sperre bekommen, so darf er den kritischen Abschnitt betreten.
- Das Freigeben der Sperre geschieht in *MyLock_unlock* über ein einfaches Nullsetzen von LOCK ohne weitere Sicherheitsmaßnahmen.

Es sei noch erwähnt, dass Sperren natürlich nicht besonders gerecht sind, weil ein Prozess möglicherweise nie in den kritischen Abschnitt kommt, was der Fairness Condition von Dijkstra widerspricht.

6.2.1 Semaphore

Das *Semaphor-Konzept*³ ist ein höherwertiges Konzept zur Lösung des Mutual-Exclusion-Problems, das von Dijkstra im Jahre 1962 erstmals entwickelt wurde und das auf Sperrmechanismen aufsetzt. Ein Semaphor verwaltet intern eine Warteschlange für die Prozesse bzw. Threads, die gerade am Eingang eines kritischen Abschnitts warten müssen, und einen Semaphorzähler. Es kommt auf die Initialisierung des *Semaphorzählers* an, wie viele Prozesse in den kritischen Abschnitt dürfen. Für den Eintritt in den kritischen Abschnitt und für den Austritt aus dem kritischen Abschnitt gibt es zwei Operationen:

- P wird beim Eintritt in den kritischen Abschnitt aufgerufen. Der Semaphorzähler wird um 1 reduziert, sofern er größer als 0 ist. Wenn er gerade auf 0 steht, wird der Eintritt verwehrt, der Prozess/Thread wird in die Warteschlange eingereiht und suspendiert.⁴
- V wird beim Verlassen des kritischen Abschnitts aufgerufen. Der Semaphorzähler wird wieder um 1 erhöht, so dass ein weiterer Prozess/Thread in den kritischen Abschnitt darf. Dies muss natürlich vom Entwickler so programmiert werden.



P: P-Operation auch Down-Operation genannt

V: V-Operation, auch Up-Operation genannt

Abbildung 6-6: Semaphor

³ Engl. Semaphore = Signalmast.

⁴ P kommt vom holländischen „passeeren“ und V vom holländischen „vrijgeven“.

In Abbildung 6-6 ist das Semaphore-Konzept grafisch dargestellt. Die P-Operation wird gelegentlich auch als Down-Operation bezeichnet, da sie den Semaphore-Zähler herunterzählt. Die V-Operation wird dagegen als Up-Operation bezeichnet, weil sie den Semaphore-Zähler erhöht.

Ein Semaphore könnte prinzipiell gemäß folgendem Pseudocode implementiert werden, wobei der Semaphore-Zähler hier mit *s* bezeichnet wird:

```
// Initialisierung des Semaphore-Zählers
s = x; // mit x >= 1
void P() { // Dies ist für sich ein kritischer Abschnitt!
    if (s >= 1){
        s = s - 1; // der die P-Operation ausführende Prozess setzt
                    // seinen Ablauf fort.
    } else {
        // der die P-Operation ausführende Prozess wird in seinem
        // Ablauf zunächst gestoppt, in den Wartezustand versetzt
        // und in einer dem Semaphore s zugeordneten Warteliste eingetragen
    }
}
void V() { // Dies ist auch für sich ein kritischer Abschnitt
    s = s + 1;
    if (Warteliste ist nicht leer) {
        // aus der Warteliste wird ein Prozess ausgewählt und aufgeweckt
    }
}
```

Eine typische Codesequenz für die Nutzung eines Semaphors könnte wie folgt aussehen:

```
...
// Ist der kritische Abschnitt besetzt?
// Wenn ja, wartet der Prozess/Thread
P();
// Kritischer Abschnitt beginnt
z = z + 3;
write (z, 3);
// Kritischer Abschnitt endet
// Verlassen des kritischen Abschnitts und
// Aufwecken eines wartenden Prozesses/Threads
V();
...
```

Wie bereits erläutert, kommt es auf die Initialisierung des Semaphorzählers an, wie viele Prozesse in den kritischen Abschnitt dürfen. Im gerade gezeigten Codebeispiel ist eine Initialisierung mit dem Wert 1 sinnvoll, da nur ein Prozess die Operationen im kritischen Abschnitt ausführen sollte. Man unterscheidet hier prinzipiell zwei Grundtypen von Semaphoren: Der erste Typ, der als *binäres Semaphore* bezeichnet wird, kann in seinem Semaphorzähler nur die Werte 0 und 1 annehmen. Der zweite Typ wird als *Zählsemaphor* (*counting semaphore*) bezeichnet. Dies ist die allgemeine Form eines Semaphors, bei der der Zähler beliebig gesetzt werden kann.

Binäre Semaphore werden auch als *Mutex* bezeichnet. Ein Mutex verfügt praktisch über eine Variable, die nur zwei Zustände, nämlich *locked* und *unlocked* annehmen kann. Mutexe sind einfach zu implementieren und effizient.

Wie wir noch sehen werden, gibt es für Semaphore und Mutexe viele praktische Einsatzfälle. Das einführend skizzierte Lost-Update-Problem kann beispielsweise durch die Nutzung eines binären Semaphors auf einfache Weise verhindert werden. Ein Szenario ist in Abbildung 6-7 skizziert.

Wie man sieht wird Prozess B an der Bearbeitung gehindert, da er in der P-Operation zunächst einmal ausgebremst und in den Wartezustand versetzt wird. Der Grund dafür ist, dass Prozess A gerade im kritischen Abschnitt ist. Nachdem Prozess A die V-Operation aufgerufen hat, kann Prozess B (nach erneuter CPU-Zuteilung) ebenfalls in den kritischen Abschnitt gelangen.

Semaphor-Operationen müssen *selbst wieder nicht unterbrechbar*, also atomar sein, weil eine Unterbrechung zu Inkonsistenzen in der Warteschlangenbearbeitung oder im Semaphorzähler führen kann und sogar ein gegenseitiger Ausschluss nicht mehr gewährleistet werden kann. Die korrekte Implementierung eines Semaphors benötigt daher am besten auch eine unteilbare Hardware-Operation wie etwa TSL.

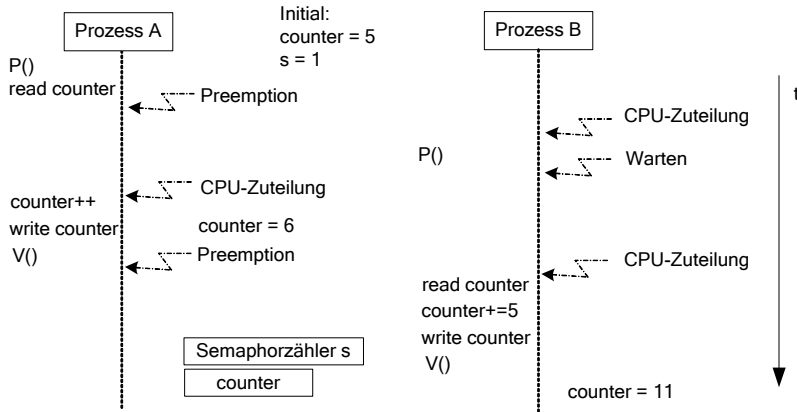


Abbildung 6-7: Semaphore zur Vermeidung des Lost-Update-Problems

Es gibt einige anschauliche Anwendungen, in denen der Einsatz von Synchronisationsmechanismen wie Semaphore sinnvoll ist. Oft werden die folgenden Anwendungen bzw. Beispiele als Anschauungsmaterial diskutiert:

- *Producer-Consumer-Problem*: Hier befasst man sich mit dem Problem, dass eine bestimmte Anzahl von produzierenden Prozessen/Threads Datenstrukturen oder Objekte generiert und diese in einen begrenzten Puffer (*bounded buffer*) einfügt, während eine beliebige Anzahl an Konsumenten-Prozessen/-Threads diese Datenstrukturen oder Objekte ausliest und verarbeitet. Der Zugriff auf den Puffer (Bounded Buffer) ist kritisch und muss synchronisiert werden.
- *Reader-Writer-Problem*: Diese Anwendung dient der Synchronisation einer beliebigen Anzahl von Schreiber-Prozessen/-Threads, die ein oder mehrere Datenstrukturen oder Objekte verändern möchten, während eine beliebige Anzahl an lesenden Prozessen/Threads die Datenstrukturen oder Objekte nur auslesen möchte. Durch das nebenläufige Lesen und Verändern kann es zu Konflikten kommen, die zu Inkonsistenten Ergebnissen führen können. Dies kann durch einen synchronisierten Zugriff vermieden werden.
- *Dining-Philosophers-Problem*: Dieses Beispielproblem dient dazu, aufzuzeigen, wie man es schafft, dass mehrere Prozesse/Threads eine vorgegebene Anzahl an Ressourcen nutzen können, um eine bestimmte Aktion auszuführen.

Auf das Dining-Philosophers-Problem, auch als Problem der „Spaghetti-essenden“ Philosophen bekannt, soll stellvertretend kurz eingegangen werden. Diese Problemstellung ist eine Erfindung von Dijkstra und Hoare aus dem Jahre 1965. Wie man am Datum der Erfindung erkennen kann, sind diese Probleme schon lange bekannt und auch weitgehend gelöst. Bei dieser Problemstellung müssen

genau zwei Betriebsmittel (hier Gabeln) belegt werden, bevor die Arbeit gemacht werden kann.

Fünf Philosophen (fünf Prozesse/Threads) sitzen um einen Tisch herum (oder gehen, wenn sie Hunger haben an den Tisch und zwar genau an ihren vorgegebenen Platz) und teilen sich fünf Gabeln. Sie sollen so essen, dass keiner verhungert und es natürlich auch keinen Deadlock gibt. Jeder Philosoph braucht zum Essen genau zwei Gabeln. Diese sind gemäß Abbildung 6-8 auf dem Tisch angeordnet. Es können also maximal zwei Philosophen gleichzeitig essen. Ein Philosoph denkt und isst abwechselnd. Wenn er hungrig ist, versucht er in beliebiger Reihenfolge die beiden Gabeln links und rechts von seinem Teller zu nehmen. Nachdem er gegessen hat, legt er sie wieder auf seinen Platz zurück, so dass ein anderer Philosoph (einer seiner Nachbarn) sie in gleicher Weise nutzen kann.

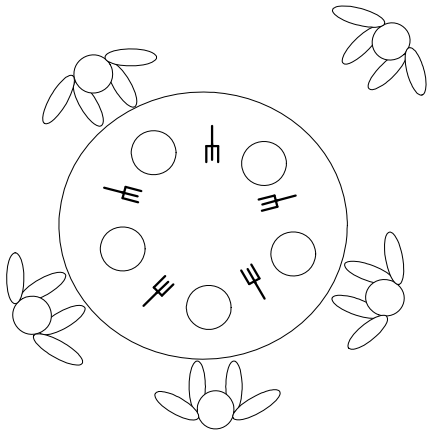


Abbildung 6-8: Dining Philosophers

Nimmt z.B. jeder Philosoph seine linke Gabel, so kann keiner essen und es gibt einen Deadlock, also einen Zustand, in dem keiner der Philosophen mehr essen kann, weil er eben zwei Gabeln benötigt. Wichtig ist also, dass ein Philosoph zwei Gabeln braucht und diese in einer atomaren Aktion⁵ aufnimmt. Wenn der Philosoph bei der Gabelaufnahme unterbrochen werden kann, ist ein Deadlock möglich und wird irgendwann auch eintreten. In Abbildung 6-9 ist eine Deadlock-Situation für das Philosophenproblem in Form eines Betriebsmittelgraphen skizziert. Wenn

⁵ Wir gehen in unserem Beispiel im Weiteren von einer atomaren Aktion aus, obwohl Dijkstra auch eine Lösung vorgibt, die ohne atomare Anforderung auskommt und deadlockfrei ist.

die Situation eintritt, dass jeder Philosoph eine Gabel belegt (P1 belegt G2, P2 belegt G3,...), ist ein Deadlock gegeben.

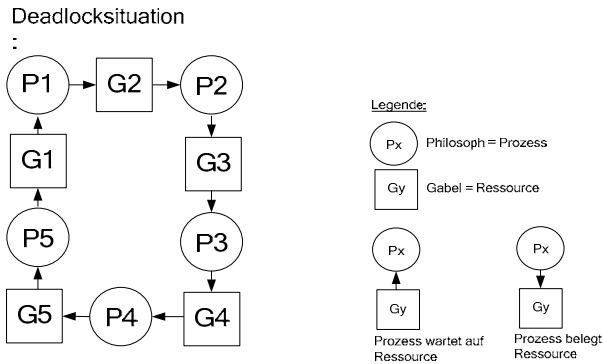


Abbildung 6-9: Deadlock-Situation beim Philosophenproblem

Implementierungen von Lösungen des Problems werden im Anhang sowohl in Java als auch in C# präsentiert.

6.2.3 Monitore

Die Nutzung von Semaphoren ist zwar eine Erleichterung bei der Programmierung von Synchronisationsproblemen, aber immer noch recht fehleranfällig. Man kann als Programmierer eine Semaphor-Operation vergessen oder verwechseln. Schreibt man z.B. versehentlich die Codesequenz

```
V(); ... kritischer Abschnitt ...; P();
```

so führt dies dazu, dass alle Prozesse im kritischen Abschnitt zugelassen sind.

Auch die Codesequenz

```
P(); ... kritischer Abschnitt ...; P();
```

ergibt wenig Sinn, da nach kurzer Zeit kein Prozess mehr in den kritischen Abschnitt darf und die ersten Prozesse möglicherweise auch im kritischen Abschnitt verbleiben.

Hoare⁶ (Hoare 1994) und Per Brinch Hansen⁷ schlugen daher bereits 1974 vor, die Erzeugung und Anordnung der Semaphor-Operationen dem Compiler zu überlassen und konzipierten den abstrakten Datentypen, den sie mit *Monitor* bezeichneten.

⁶ Sir Charles Antony Richard Hoare (geboren am 11.01.1934), Britischer Computerwissenschaftler.

Unter einem Monitor versteht man eine Menge von Prozeduren und Datenstrukturen, die als Betriebsmittel betrachtet werden und mehreren Prozessen zugänglich sind, aber nur von einem Prozess zu einer Zeit benutzt werden können. Anders ausgedrückt ist ein Monitor ein Objekt (im Sinne der objektorientierten Programmierung), das den Zugriff auf gemeinsam genutzte Daten über kritische Bereiche in Zugriffsmethoden realisiert.

Ein Monitor ist demnach eine Hochsprachenprimitive zur Lösung des Synchronisationsproblems. In einem Monitor werden gemeinsam benutzte Daten durch Synchronisationsvariable geschützt. Der Zugriff auf die privaten, zu schützenden Daten ist nur über die definierten Zugriffsmethoden (siehe Abbildung 6-10) zulässig. Man findet Implementierungen des Monitorkonzepts z.B. in der Programmiersprache Concurrent Pascal oder als Bibliothekserweiterungen.

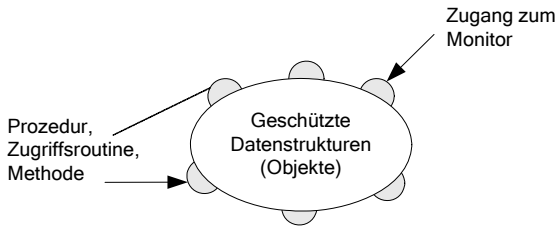


Abbildung 6-10: Abstraktion eines Monitors

Für die Monitor-Realisierung benötigt man zudem eine oder mehrere Bedingungsvariablen und zwei Standard-Operationen *wait* und *signal*, die den Bedingungsvariablen zugeordnet sind. Diese beiden Operationen dienen dazu, einen Prozess/Thread in den Wartezustand zu versetzen (*wait*) oder an einen wartenden Prozess/Thread ein Signal zu senden, das ihn wieder aufweckt (*signal*). Kann ein Prozess nicht weiterarbeiten, weil er auf ein Betriebsmittel (also auf eine Bedingung) warten muss, hat er den Monitor temporär zu verlassen, damit ein anderer Prozess diesen betreten kann. Dafür dient die Operation *wait*, die auf die Bedingungsvariable angewendet wird. Verlässt ein Prozess einen kritischen Abschnitt, auf dessen Zugang ggf. ein anderer Prozess wartet, kann er dem wartenden Prozess mit der Operation *signal* auf die Bedingungsvariable mitteilen, dass der kriti-

⁷ Von Per Brinch Hansen, Dänisch-Amerikanischer Computerwissenschaftler (geboren am 13.11. 1938, gestorben am 31.07.2007) gibt es einige Veröffentlichungen, in denen die wesentlichen Grundzüge der parallelen Programmierung dargestellt sind. Er war auch der Erfinder der Sprache Concurrent Pascal. Die Hochphase der Entwicklung in diesem Umfeld war 1971 bis 1975. Folgende Veröffentlichungen von Brinch Hansen sind u.a. interessant: (Brinch Hansen 1973) S. 226-232 und (Englewood 1995) S. 199-207.

sche Abschnitt nun frei ist. Der wartende Prozess kann nun seine Arbeit fortsetzen. Das temporäre Verlassen des Monitors ist notwendig, da nur ein Prozess im Monitor aktiv sein darf. Es wird implizit durch die Monitor-Implementierung, also ohne zusätzlichen Programmieraufwand, durchgeführt.

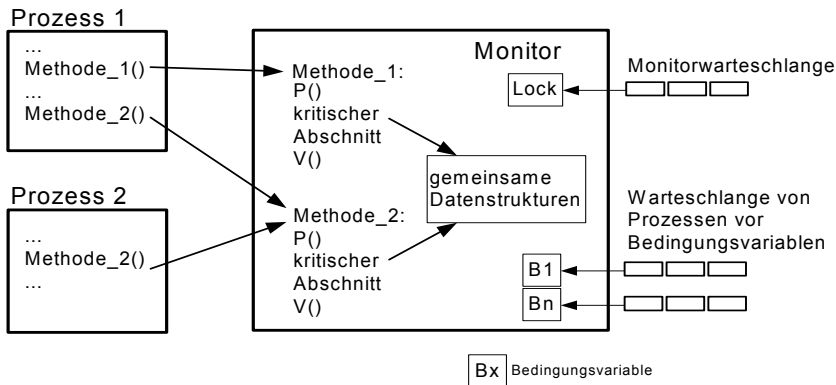


Abbildung 6-11: Grundstruktur eines Monitors

Das mit *signal* generierte „Aufwecksignal“ wird üblicherweise nicht gespeichert. Ist also die Warteschlange der auf das Signal wartenden Prozesse bzw. Threads leer, bleibt das Signal ohne Wirkung.

Man unterscheidet bei *signal/wait* zwei unterschiedliche Implementierungskonzepte. Das *Signal-and-Continue-Konzept* besagt, dass ein signalisierender Prozess nach Absenden eines Signals mit seiner Verarbeitung weiter macht und dann irgendwann einmal, z.B. bei Ablauf seiner Zeitscheibe, vom Scheduler deaktiviert wird. Beim *Signal-and-Wait-Konzept* verlässt der signalisierende Prozess dagegen sofort den Monitor. Abbildung 6-11 zeigt die implizite und für die nutzenden Prozesse verborgene Einbettung der P- und V-Operationen in die Zugriffsmethoden (hier `Methode_1` und `Methode_2`) eines Monitors.

Über eine entsprechende Einbettung des Monitorkonzepts in eine Programmiersprache können Monitore mit beliebigen Zugriffsmethoden programmiert werden. Nach (Tanenbaum 2002)⁸ lässt sich z.B. das Producer-Consumer-Problem mit einem speziell dafür vorgesehenen Monitor wie folgt realisieren (wir nehmen hier die Nutzung des *Signal-and-Wait-Konzepts* an):

Monitor ProducerConsumer

⁸ Beispiel etwas abgewandelt.

```
{
    final static int N = 5;          // Maximale Anzahl an Items im Puffer
    static int count = 0;            // Zähler für Items im Puffer
                                    // Bedingungsvariable: Puffer ist voll

    condition not_full = false;      // (zunächst nicht voll)
    condition not_empty = false;     // Bedingungsvariable: Puffer ist
                                    // leer (Anfangszustand)

    int Puffer[N-1];                // Pufferbereich
    // Operation zum Einfügen eines Items im Puffer
    void insert(item integer) {
        // Warten, bis Puffer wieder nicht mehr voll ist,
        // also bis Bedingungsvariable not_full = true ist
        if (count == N) wait(not_full);
        // Füge Item in Puffer ein
        count+=1;
        // Signalisieren, dass Puffer nicht mehr leer ist
        // also dass not_empty = true ist.
        if (count == 1) {signal(not_empty);}
    }
    // Operation zum Entfernen eines Items aus dem Puffer
    int remove() {
        // Warten, bis Puffer nicht mehr leer ist,
        // also bis Bedingungsvariable not_empty true ist
        if (count == 0) {wait(not_empty);}
        // Entferne Item aus Puffer
        count--;
        // Signalisieren, dass Puffer nicht mehr voll ist,
        // also dass not_full = true ist.
        if (count == (N-1)) {signal(not_full);}
        return (item);
    }
}

// Klasse, die den Monitor verwendet
class UseMonitor
{
    ProducerConsumer mon = new ProducerConsumer();
    ...
    // Produzent, der Items erzeugt
```

```

void producer() {
    while (true) {
        // Erzeuge Item
        // Einfügen des Items in den Puffer, evtl. muss gewartet werden
        mon.insert(item);
    }
}

// Konsument, der Items verwendet
void consumer() {
    while (true) {
        // Hole ein Item aus dem Puffer, warte, wenn Puffer leer ist
        // Konsumiere ein Item
        item = mon.remove();
    }
}
}

```

Der im Pseudocode skizzierte Monitor stellt die zwei Operationen zum Einfügen und Entfernen von sog. „Items“ bereit (*insert* und *remove*) und verwaltet einige private Variablen, u.a. den gemeinsam genutzten Puffer und die Bedingungsvariablen (Condition-Variablen) *not_full* sowie *not_empty*.

Ist der Puffer voll (hier bei fünf Elementen, Items), so kann nichts mehr eingefügt werden und die Bedingungs-Variable *not_full* steht auf *false*. Ist der Puffer leer, kann nichts mehr entnommen werden und die Bedingungsvariable *not_empty* steht auf *false*. Die beiden in diesem Beispiel genutzten Bedingungsvariablen werden über die Operationen *signal* und *wait* entsprechend verändert. Wenn z.B. der Puffer voll ist, wird die Operation *wait(not_full)* aufgerufen, um den aktiven Produzentenprozess, der ein neues Item in den Puffer legen möchte, erst einmal in den Wartezustand zu versetzen. Erst wenn ein Konsumentenprozess wieder ein Item aus dem Puffer entfernt hat, signalisiert der Monitor dies über *signal(not_full)* an den wartenden Produzenten, der daraufhin weiter arbeiten kann.

Bei der Programmierung des Monitors muss darauf geachtet werden, dass die Operationen *signal* und *wait* auf die Bedingungsvariablen je nach Anwendungsfall richtig eingesetzt werden. Die Nutzung des Monitors ist, wie die Klasse *UseMonitor* zeigt, dann recht einfach.

Allerdings gibt es bei der Implementierung des Monitorkonzepts einige Probleme. Insbesondere kann das temporäre Verlassen des Monitors beim *wait*-Aufruf zu erheblichen Implementierungsschwierigkeiten führen.

6.3 Synchronisationstechniken moderner Betriebssysteme

Im Folgenden soll nur eine kurze Einführung zu einigen Techniken, die unter Unix und Windows sowie im POSIX-Standard vorhanden sind, gegeben werden (Tannenbaum 2009, Kredel 1999).

6.3.1 Synchronisationstechniken unter Unix

In den meisten Unix-Derivaten gibt es Semaphore, wobei die Syntax und Semantik der Semaphor-Operationen zum Teil unterschiedlich ist. Folgende Semaphor-Operationen sind typisch:

- *semget* zum Erzeugen einer Menge von Semaphoren und zum Aufnehmen einer Verbindung zu einem Semaphor.
- *semop* für die Semaphor-Operationen zum Hochzählen (V-Operation) und Herunterzählen (P-Operation) und zum Abfragen der Semaphore.
- *semctl* zum Setzen und Abfragen eines Semaphors. Beispielsweise kann mit der Operation abgefragt werden, wie viele Prozesse gerade durch das Warten auf den Semaphor blockiert sind.

Ein Semaphor erhält beim Erzeugen unter Unix eine eigene Identifikation ähnlich einem Filedeskriptor, die bei jeder weiteren Operation verwendet werden muss.

6.3.2 Synchronisationstechniken unter Windows

Unter Windows existiert ebenfalls eine Semaphor-Implementierung, die der Unix-Implementierung semantisch recht ähnlich ist und zur Synchronisation von Threads dient. Typische Methoden sind:

- *CreateSemaphore* dient dazu, ein Semaphor ähnlich wie eine Datei in einem prozess-globalen Namensraum zu erzeugen.
- *WaitForSingleObject* ist mit *P()* und *ReleaseSemaphore* mit *V()* vergleichbar.

Da die Windows-Prozess-Semaphore recht „schwergewichtig“ sind, gibt es zur Synchronisation von Threads auch noch sog. „Leichtgewichtsemaphore“, die auch als „Critical Sections“ bezeichnet werden, wobei u.a. folgende Operationen angeboten werden:

- Mit der Operation *InitializeCriticalSection* wird ein kritischer Bereich angelegt.
- Die P-Operation zum Betreten des kritischen Abschnitts heißt *EnterCriticalSection*.
- Die V-Operation zum Verlassen des kritischen Abschnitts ist *LeaveCriticalSection*.

Zur Multiprozessorsynchronisation gibt es unter Windows spezielle Spinlocks. Ein Thread kann mit einem Spinlock einen Prozessor so lange blockieren, bis er den Spinlock wieder frei gibt. Spinlocks werden aber nur im Kernelmodus benutzt.

Weitere Synchronisationsdienste basieren unter Windows auf sog. Kernel-Objekten, die über spezielle Synchronisationseigenschaften verfügen. Beispiele für Kernel-Objekte sind Ereignisse, Timer und Mutexe sowie Threads. Ein Thread kann z.B. auf einen anderen Thread warten, indem er die Methode *WaitForSingleObject* aus dem Windows-API aufruft. Mit *CreateMutex* wird ein Mutex-Objekt erzeugt, was einem binären Semaphor entspricht, und mit *ReleaseMutex* wird das Objekt wieder freigegeben.

6.3.3 POSIX-Synchronisationstechniken

Die im POSIX-Standard definierte Thread-API ist in der Regel in einer Thread-Bibliothek (meist in C/C++) realisiert. Die sog. Pthreads-API enthält mehr als 50 Methoden, wovon einige der Synchronisation dienen:

- Mit den verfügbaren Methoden *pthread_mutex_init*, *pthread_mutex_unlock*, *pthread_mutex_lock* und *pthread_mutex_trylock* werden z.B. Mutexe realisiert.
- Die Methoden *pthread_cond_wait* und *pthread_cond_signal* dienen dem Warten und Senden von Signalen und damit zur expliziten Synchronisation zwischen Prozessen.

Für eine weiterführende Darstellung von POSIX-Threads und deren Synchronisations-Mechanismen wird auf die Literatur verwiesen (Tanenbaum 2009).

6.4 Synchronisationsmechanismen in Programmiersprachen

In Java werden einige Sprachmechanismen bereitgestellt, die es ermöglichen, nebenläufige Threads zu synchronisieren. In der Version Java 1.5 hat sich einiges zu den Vorgängerversionen verändert. Einige neue Mechanismen sind hinzugekommen. Die Synchronisationsprimitive *synchronized* scheint aber immer noch am Wichtigsten zu sein und wird deshalb im Folgenden behandelt. Weiterhin wird eine Einführung in Sprachmechanismen von .NET C# gegeben.

In C# gibt es eine Fülle von Mechanismen zur Synchronisation von Threads, die im .NET-Framework, und zwar im Namespace *System.Threading* vereinbart sind. Die zugehörigen Basisklassen sind in der Framework Class Library (FCL) zu finden. Hierzu gehören u.a. (siehe Prosise 2002):

- Die *Monitor*-Klasse
- Die Synchronisationsprimitive *lock*
- *Mutex*-Klasse

- Lese- und Schreibsperrern mit *ReaderWriterLock*-Klasse
- *Interlocked*-Klasse

Die wichtigsten Mechanismen sind die Synchronisationsprimitive *lock* sowie die *Monitor*-Klasse.

Jedes C#-Objekt enthält neben einer Methodentabelle, in der alle Referenzen auf die verfügbaren Methoden aufgeführt sind, auch eine Datenstruktur, die gelegentlich als *SyncBlock* (Stärk 2003) bezeichnet wird. Dies ist ein Bereich, in dem Synchronisationsinformationen wie eine Sperre, eine Warteschlange für blockierte Threads, die auf das Objekt warten, und eine Condition-Variable verwaltet werden.

6.4.1 Die Java-Synchronisationsprimitive „synchronized“

Die Synchronisationsprimitive *synchronized* ermöglicht eine Zugriffsserialisierung, die auf Klassen- oder Objektebene erfolgt, je nachdem, ob man sie auf statische Methoden oder Instanzmethoden anwendet. Im Folgenden ist der Rumpf einer synchronisierten Methode dargestellt.

```
public synchronized void method1()  
{  
    ...  
}
```

Auch die Nutzung von *synchronized* in Anwendungsblöcken ist möglich, wobei hier ein Objekt angegeben wird, für das der Zugriff zu serialisieren ist. Verwendet man als Objekt *this*, gilt auch hier die Serialisierung für das ganze Objekt (, hier das Objekt, auf das die Referenz *object1* verweist):

```
synchronized (object1)  
{  
    ...  
}
```

Die Aufrufe einer synchronisierten Methode oder der Zugriff auf das mit *synchronized* gekennzeichnete Objekt werden strikt serialisiert und verzögert, wenn ein anderer Thread gerade im synchronisierten Code ist. Damit lässt sich ein kritischer Abschnitt deklarativ implementieren.

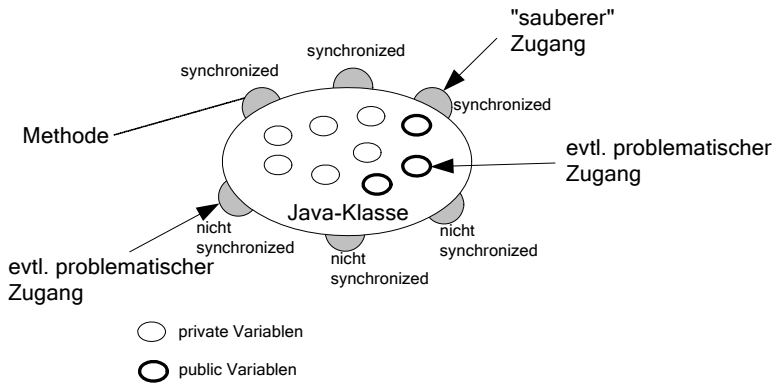


Abbildung 6-12: „Synchronized“ Java-Klassen

Die Java Virtual Machine (JVM) implementiert die gesamte Synchronisationslogik und verwendet als internen Synchronisationsmechanismus eine Monitor-Variante. Es sind jedoch keine reinen Monitore nach Hoare oder Brinch Hansen⁹, da nicht alle öffentlichen (*public*) Methoden zwingend als *synchronized* deklariert sein müssen und auch nicht nur private Variablen zugelassen sind. Betrachtet man die Abbildung 6-12, so erkennt man, dass in Klassen sowohl „synchronized“ Methoden, als auch andere Methoden zulässig sind. Weiterhin gibt es auch die Möglichkeit, öffentliche Variablen zu deklarieren, auf die dann ein beliebiger Zugriff von außen möglich ist. Brinch Hansen kritisiert dies, da der Compiler hier keine Möglichkeit der Überprüfung hat. Allerdings kann man durch disziplinierte Programmierung Probleme umgehen (keine Variable als *public* und alle Methoden mit *synchronized* deklarieren).

Abbildung 6-13 zeigt das grundlegende Konzept des Java-Monitors. Jedem Monitor ist eine Warteschlange zugeordnet, in der Threads warten müssen, falls der Monitor gerade durch einen anderen Thread belegt ist. Belegt bedeutet hier, dass sich ein anderer Thread gerade in einer mit *synchronized* gekennzeichneten Methode befindet.

Ein Java-Objekt (also eine Instanz einer Java-Klasse) ist automatisch ein Java-Monitor, wenn es mindestens eine Methode enthält, in der das Schlüsselwort *syn-*

⁹ Per Brinch Hansen befasst sich in (Brinch Hansen 1999) mit Java und paralleler Programmierung und stellt fest, dass die Java-Monitore nicht korrekt sind und Java gegen Grundprinzipien verstößt. Beispielsweise lässt Java auch als „public“ deklarierte Variablen in Klassen zu, die synchronisierte Methoden haben.

synchronized verwendet wird. Jedem Java-Monitor wird implizit von der JVM eine Sperre (Monitor-Sperre) zugeordnet. Die erforderliche Sperrinformation wird intern für das Objekt (bei Instanzmethoden) oder aber für die ganze Klasse (bei mit *synchronized* gekennzeichneten statischen, also Klassenmethoden) verwaltet. Bedingungsvariablen für Monitorbedingungen werden in Java nicht verwaltet. Alle mit dem Schlüsselwort *synchronized* gekennzeichneten Methoden und Codeblöcke bilden gemeinsam die kritischen Abschnitte des Monitors. Der implizite Semaphor sperrt bei Eintritt eines Threads in einen mit *synchronized* gekennzeichneten Bereich alle kritischen Bereiche des Objekts (oder entsprechend der ganzen Klasse).

Die implizit verwaltete Bedingungsvariable eröffnet zusätzlich die Möglichkeit über ein Signal-and-Continue-Konzept auf Bedingungen zu warten. Die hierzu vorhandenen Methoden *signal* und *notify* bzw. *notifyAll* werden weiter unten noch näher betrachtet.

Die Sperrgranularität ist bei einem Java-Monitor sehr grob, da sie das Objekt oder sogar die Klasse (Klassenmethode) für die gesamte Durchlaufzeit durch den Methodencode komplett sperrt. Dies ist nachteilig für die Laufzeit, der Mechanismus ist also nicht sehr effektiv. Verbessert wird dies durch Nutzung von synchronisierten Anweisungsblöcken, da hier der kritische Bereich kürzer gehalten werden kann. Die Spezifikation der JVM legt die Implementierung an dieser Stelle nicht fest. Es wurde auch beobachtet, dass manche JVM-Implementierungen immer auf Klassenbasis sperren.

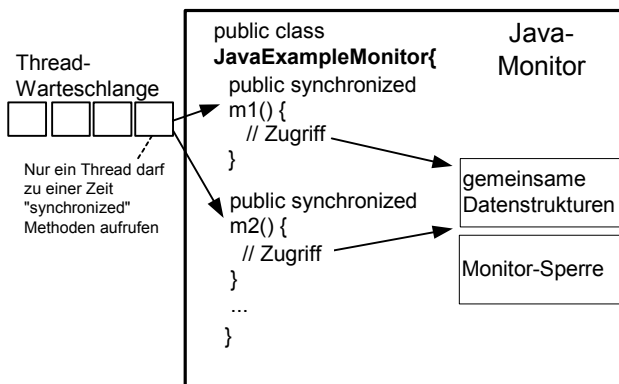


Abbildung 6-13: Grundstruktur eines Java-Monitors

Beispiel: Die Nutzung der Synchronisationsprimitive *synchronized* soll am Beispiel eines Zählerobjekts, das von mehreren Threads nebenläufig hochgezählt wird, dargestellt werden. Das Programm startet drei nebenläufige Threads, die einen

gemeinsamen Zähler jeweils um 1.000.000, also insgesamt auf 3.000.000 zählen. Der Zähler ist in einem Objekt der Klasse *CounterObject* gekapselt. Das Objekt stellt Methoden zum Auslesen (get) und zum Setzen des Zählers (set) bereit. Der kritische Abschnitt umfasst die drei folgenden Operationen:

```
int c = myCounter.get();
c++;
myCounter.set(c);
```

Natürlich könnte man einfach nur eine *addOne*-Methode definieren, die den Zähler um 1 erhöht. Damit würde man sich zwei Operationen sparen. Aber das grundsätzliche Problem bleibt. Drei Operationen bewirken mehr Konflikte beim nebenläufigen Ablauf, und das ist aus didaktischen Gründen sinnvoll.

Der kritische Abschnitt wird im ersten Durchlauf ungeschützt und im zweiten Durchlauf durch *synchronized* geschützt durchlaufen. Hierfür sind die beiden von der Klasse *Thread* abgeleiteten Klassen *CountThread1* und *CountThread2* definiert, die jeweils in ihren *run*-Methoden das Lesen, Hochzählen und Setzen ausführen.

In Abbildung 6-14 ist ein Sequenzdiagramm dargestellt, in dem die drei Threads auf den Zähler zugreifen. Gezeigt wird auch ein Lost-Update, der durch eine verzahnte Ausführung der *get*- und der *set*-Methode in den Threads T₂ und T₃ verursacht wird.

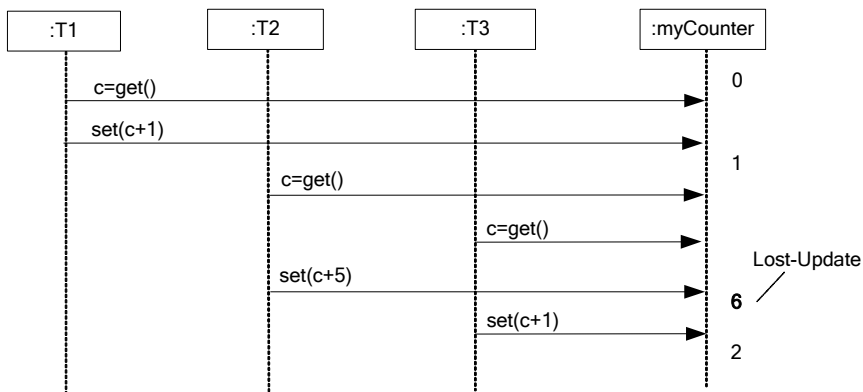


Abbildung 6-14: Sequenzdiagramm für Counter-Problem

Im Folgenden soll der Java-Code für das Beispiel kurz erläutert werden. Zunächst wird die Klasse *CounterObject* dargestellt:

```
package Threads;
import java.io.*;
class CounterObject {
```

```
private int count = 0;
CounterObject() { ... }
void set(int newCount) {
    count = newCount;
}
int get() {
    return count;
}
}
```

Die Klasse *CountThread1* ist von der Klasse *Thread* abgeleitet und dient dem Test des nicht synchronisierten Hochzählens des oben definierten Zählerobjekts. Der kritische Bereich liegt in der Methode *run*.

```
class CountThread1 extends Thread {
    private CounterObject myCounter;
    private int myMaxCount;
    CountThread1(CounterObject c, int maxCount)
    {
        myCounter = c;
        myMaxCount = maxCount;
    }
    public void run() {
        System.out.println("Thread " + getName() + " gestartet");
        for (int i=0;i<myMaxCount;i++){
            // Kritischer Bereich
            int c = myCounter.get();
            c++;
            myCounter.set(c);
            // Ende des kritischen Bereichs
        }
    }
}
```

Die Klasse *CountThread2* ist von der Klasse *Thread* abgeleitet und dient dem Test des synchronisierten Hochzählens des oben definierten Zählerobjekts. Der kritische Bereich liegt in der Methode *run*.

```
class CountThread2 extends Thread {
    private CounterObject myCounter;
    private int myMaxCount;
```

```
CountThread2(CounterObject c, int maxCount) {
    myCounter = c;
    myMaxCount = maxCount;
}

public void run() {
    System.out.println("Thread " + getName() + " gestartet");
    for (int i=0; i<myMaxCount; i++){
        // Kritischer Bereich
        synchronized (myCounter) {
            int c = myCounter.get();
            c++;
            myCounter.set(c);
        }
        // Ende des kritischen Bereichs
    }
}
```

Die Klasse *CountTest* ist eine Testklasse, die in ihrer *main*-Methode jeweils drei Threads für den synchronisierten und drei für den nicht synchronisierten Test aufruft und am Ende der Thread-Bearbeitung den Zähler ausgibt.

```
class CountTest {
    public static void main(String[] args) {
        CounterObject c = new CounterObject();
        int maxCount = 1000000;
        // Wir starten drei nebenläufige Threads und warten auf ihr Ende
        Thread t1 = new CountThread1(c, maxCount);
        Thread t2 = new CountThread1(c, maxCount);
        Thread t3 = new CountThread1(c, maxCount);
        System.out.println("Test ohne Synchronisation geht jetzt los ...");
        try {
            t1.start(); t2.start(); t3.start();
            t1.join(); t2.join(); t3.join();
        }
        catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
        finally {
```

6 Synchronisation und Kommunikation

```
        System.out.println("Counter am Ende, Wert sollte 3.000.000 " +
            "sein und ist " + c.get());
        System.out.println("Test ohne Synchronisation ist zu Ende");
    }
    // Wir starten drei nebenläufige Threads und warten auf ihr Ende
    c.set(0);    // Zähler auf 0 setzen
    t1 = new CountThread2(c, maxCount);
    t2 = new CountThread2(c, maxCount);
    t3 = new CountThread2(c, maxCount);
    System.out.println("Test mit Synchronisation geht jetzt los ...");
    try {
        t1.start(); t2.start(); t3.start();
        t1.join(); t2.join(); t3.join();
    }
    catch (InterruptedException e) {
        System.out.println("Thread interrupted");
    }
    finally {
        System.out.println("Counter am Ende, Wert sollte " +
            "3.000.000 sein und ist " + c.get());
        System.out.println("Test mit Synchronisation ist zu Ende");
    }
}
```

Das Ergebnis der Durchläufe ist, dass im nicht synchronisierten Fall tatsächlich eine ganze Menge an Updates möglicherweise verloren geht. Dies ist etwas zufallsbedingt und die Summe ist daher bei jedem Durchlauf immer etwas anders (Race Condition). Die Ausgaben zweier Testläufe zeigen, dass beim Zählen ohne Synchronisation eine ganze Menge an Lost-Updates auftreten:

Lauf 1:

```
Test ohne Synchronisation geht jetzt los ...
Thread Thread-0 gestartet
Thread Thread-1 gestartet
Thread Thread-2 gestartet
Counter am Ende, Wert sollte 3.000.000 sein und ist 1120906
Test ohne Synchronisation ist zu Ende
Test mit Synchronisation geht jetzt los ...
```

Thread Thread-3 gestartet
Thread Thread-4 gestartet
Thread Thread-5 gestartet
Counter am Ende, Wert sollte 3.000.000 sein und ist 3.000.000
Test mit Synchronisation ist zu Ende

Lauf 2:

Test ohne Synchronisation geht jetzt los ...
Thread Thread-0 gestartet
Thread Thread-1 gestartet
Thread Thread-2 gestartet
Counter am Ende, Wert sollte 3.000.000 sein und ist 2171922
Test ohne Synchronisation ist zu Ende
Test mit Synchronisation geht jetzt los ...
Thread Thread-3 gestartet
Thread Thread-4 gestartet
Thread Thread-5 gestartet
Counter am Ende, Wert sollte 3.000.000 sein und ist 3.000.000
Test mit Synchronisation ist zu Ende

Im Lauf 1 ist der Zähler 1.120.906, es gehen also 1.879.094 Updates verloren, im Lauf 2 fehlen $3.000.000 - 2.171.922$, also 828.078 Updates. Man sieht, dass die Anzahl an Lost-Updates von Lauf zu Lauf variiert, da sie davon abhängt, wie viele Konflikte zufällig entstehen. Im Gegensatz dazu wird mit Synchronisation immer richtig gezählt.

6.4.2 Warten auf Bedingungen in Java

Bei manchen Problemstellungen reicht die Nutzung der Synchronisationsprimitive *synchronized* nicht aus. Dies ist zum Beispiel der Fall, wenn zwischen zwei nebenläufigen Programmteilen nicht vermeidbare Abhängigkeiten bestehen und man daher diese explizit synchronisieren muss.

Zur Realisierung einer expliziten Synchronisation zwischen nebenläufigen Programmteilen mit Wartebedingungen hat Java in der Klasse *Object* einige Methoden definiert, die eine Abstimmung zwischen Threads ermöglichen und die implizit auf die jedem Java-Monitor zugeordnete Bedingungsvariable wirken:

- *wait*: Ruft ein Thread diese Methode auf, so geht er in einen Wartezustand und wartet auf einen *notify/notifyAll*-Aufruf eines anderen Threads.

- *notify*: Mit Aufruf dieser Methode weckt ein Thread mindestens einen gerade wartenden (blockierten) Thread auf. Die Methode garantiert nicht, dass genau ein Thread aufgeweckt wird.
- *notifyAll*: Durch Aufruf dieser Methode werden alle Threads, die aktuell auf das gleiche Objekt warten, aufgeweckt. Diese Methode ist dann sinnvoll, wenn unterschiedliche Threadtypen auf ein Signal warten (Beispiel: Erzeuger-Verbraucher-Problem). In diesen Anwendungsfällen könnte es bei Nutzung von *notify* zu Deadlocks kommen.
- *join*: Wird diese Methode eines Threads aufgerufen, so wartet der aufrufende Thread auf das Ende des angegebenen Threads.

Die Methoden *wait* und *notify* bzw. *notifyAll* dürfen nur innerhalb eines mit *synchronized* geschützten Abschnitts aufgerufen werden. Ein *notify/notifyAll* wirkt nur auf Threads, die in diesem Abschnitt sind, aber sich durch Aufruf der *wait*-Methode selbst blockiert haben.

Bei der Methode *wait* kann man einen Timeout-Parameter mitgeben, der eine Begrenzung der Wartezeit ermöglicht. Ist die Wartezeit abgelaufen, terminiert die Methode auch ohne *notify* eines anderen Threads, und der Thread erhält wieder die Sperre für den kritischen Abschnitt.

Die Nutzung von *wait* und *notify* ist laufzeitkritisch hinsichtlich der Reihenfolge der Aufrufe. In nebenläufigen Ausführungseinheiten kann es nämlich vorkommen, dass ein Thread die Methode *notify*-Aufruf aufruft, bevor der eigentlich wartende Thread eine *wait*-Methode ausführt. In diesem Fall erhält letzterer das Signal möglicherweise nicht mehr. Zudem ist es möglich, dass ein wartender Thread zwar mit *notify* aufgeweckt wird, aber wieder durch den Scheduling-Mechanismus suspendiert wird, bevor er die Wartebedingung erneut abprüfen kann. Ein anderer Thread könnte dann die Bedingung verändern, noch bevor der aufgeweckte Thread die Bedingung abprüfen kann. Der Grund dafür ist, dass in Java-Monitoren keine Bedingungsvariablen vorhanden sind. Die Lösung hierfür ist, den *wait*-Aufruf in einer *while*-Schleife so zu programmieren, so dass die Bedingung auch nach einem *wait*-Aufruf erneut geprüft wird.

Die Nutzung von *wait* und *notify* enthält noch keinen Warteschlangenmechanismus wie er für Semaphore gefordert ist und ist daher auch nicht absolut fair. In der Praxis werden Semaphore aber auf unterschiedliche Weise implementiert. In Java können hierfür die Methoden *wait* und *notify* genutzt werden.

Beispiel: Eine vereinfachte Semaphore-Implementierung nach (Kredel 1999) ohne Warteschlange in Java, welche die Methoden *wait* und *notify* nutzt, soll im Folgenden kurz skizziert werden. Eine Klasse *Sema* enthält neben einem Konstruktor und einem Destruktor (Finalizer) die synchronisierten Semaphore-Operationen P und V.

Bei der Konstruktion mit *Sema(int i)* kann angegeben werden, wie viele Threads nebenläufig bedient werden können:¹⁰

```
package Semaphor;

public class Sema {
    // Anzahl der maximal möglichen Threads im kritischen Abschnitt
    private int init;
    // Anzahl der verfügbaren Plätze im kritischen Abschnitt
    private int s;
    // Anzahl der in der P-Operation mit wait wartenden Threads
    private int del;
    public Sema() {
        this(0);
    }
    // Konstruiere eine Semaphore mit einer Obergrenze
    public Sema(int i) {
        if (i >= 0) { init = i; } else { init = 0; }
        s = init;
        del = 0;
    }
    protected void finalize() throws Throwable {
        if (init != s) {
            int x = init - s;
            System.out.println("sema: " + x + " pending operations.");
        }
        super.finalize();
    }
    // P-Operation
    synchronized void P() {
        while (s <= 0) {
            del++;
            try {
                this.wait();
            } catch (InterruptedException e) {}
            del--;
        }
    }
}
```

¹⁰ Im Anhang ist eine Lösung zum Philosophenproblem skizziert, die die Klasse *Sema* benutzt.

```
    }
    s--;
}
// V-Operation
public synchronized void V() {
    s++;
    if (del > 0) {
        this.notify();
    }
}
}
```

6.4.3 Weitere Synchronisationsmechanismen in Java

In neueren Java-Implementierungen (ab Version 1.5) werden eine Fülle von Synchronisationsmechanismen im Java-Package *java.util.concurrent* zur Verfügung gestellt. Zwei davon sollen erwähnt werden:

- Atomar nutzbare und „*thread-safe*“ Variable unterschiedlichen Typs wie z.B. *AtomicBoolean*, *AtomicLong*, *AtomicInteger*, usw.
- Eine eigene Semaphor-Implementierung

Beispiel: Implementierung des Counter-Problems mit einer *AtomicLong*-Variable. Das Lesen und gleichzeitige Erhöhen des Zählers wird über die bereitgestellte Methode *getAndIncrement* ausgeführt. Weitere Methoden zur Nutzung dieser atomar verwalteten Variablen sind *addAndGet*, *getAndSet*, *getAndDecrement* usw.

```
import java.lang.Thread;
import java.util.concurrent.atomic.AtomicLong;
/**
 * Thread-Klasse, die den Zähler synchronisiert hochzählt
 */
class CountThread extends Thread
{
    // Dies ist der automatisch synchronisierte Zähler
    private AtomicLong myCounter;
    private int myMaxCount;

    /*
     * Konstruktor
     */
    CountThread(AtomicLong c, int maxCount)
```



```
{
    myCounter = c;
    myMaxCount = maxCount;
}
/*
 * run-Methode
 */
public void run()
{
    System.out.println("Thread " + getName() + " gestartet");
    // Synchronisierter Zugriff auf den Counter
    for (int i=0;i<myMaxCount;i++){
        // Kritischer Bereich
        long c = myCounter.getAndIncrement();
        // Ende des kritischen Bereichs
    }
}
}
/**
 * Testklasse ...
 */
class CountTest
{
    public static void main(String[] args)
    {
        AtomicLong c = new AtomicLong();
        final int maxCount = 1000; // Jeder Thread soll 1000 mal um 1 erhöhen
        // Wir starten drei nebenläufige Threads und warten auf ihr Ende
        Thread t1 = new CountThread(c, maxCount);
        Thread t2 = new CountThread(c, maxCount);
        Thread t3 = new CountThread(c, maxCount);
        System.out.println("Test geht jetzt los ...");
        // Der Rest ist bekannt
        ...
    }
}
```

Weiterhin ist ab Java 1.5 eine eigene Semaphore-Implementierung vorhanden, die verschiedene Nutzungsmöglichkeiten bietet. Bei der Konstruktion eines Semaphors kann beispielsweise angegeben werden, ob es eine faire FIFO-organisierte Semaphore sein soll oder nicht. Die Klasse heißt *Semaphore*¹¹ und hat maximal zwei Parameter im Konstruktor:

```
public Semaphore(int permits, boolean fair)
```

Mit dem Parameter *permits* kann die Anzahl der parallel bzw. der zu Beginn zugelassenen Threads eingestellt werden (= Semaphorzähler). Der Parameter *fair* kann auf *true* oder *false* gesetzt werden. *true* bedeutet, dass die Abarbeitung der P-Operationen fair im FIFO-Verfahren abläuft. Die P-Operation heißt hier *acquire* und die V-Operation *release*. Mit jedem *acquire*-Aufruf wird der Semaphorzähler um 1 reduziert, bei Aufruf von *release* entsprechend um 1 erhöht. *permits* kann sogar auf einen negativen Wert eingestellt werden, wobei zunächst *release*- vor *acquire*-Aufrufen notwendig sind. Weitere Methoden sind z.B. *getQueueLength*, *availablePermits* und *isFair*.

Beispiel: Anlegen einer fairen Semaphore in einem kleinen Testprogramm.

```
...
import java.util.concurrent.Semaphore;
public class SemaphorTestClass
{
    public static Semaphore s; // Referenz auf eine Semaphore
    public static void main(String[] args)
    {
        // Semaphortestklasse erzeugen
        SemaphorTestClass semTestClass = new SemaphorTestClass();
        semTestClass.work();
    }
    void work() {
        // Semaphore wird angelegt: Nur 1 Thread darf die Semaphore zu
        // einer Zeit betreten, true bedeutet faire FIFO-Abarbeitung
        s = new Semaphore(1, true);
        ...
    }
}
```

¹¹ Siehe <http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html>, letzter Zugriff am 03.04.2009.

Diese Standard-Semaphore macht die Implementierung der oben dargestellten Beispielprogramme natürlich wesentlich einfacher. Der Leser möge die Programme für das Counter-Problem und das Philosophenproblem selbst mit Hilfe der Java-Semaphore programmieren.

6.4.4 C#-Monitore

C#-Monitore synchronisieren den gleichzeitigen Zugriff von Threads auf eine Ressource (Objekt, Liste, ...). Sie werden in der FCL-Klasse *Monitor* bereitgestellt, die grob wie folgt definiert ist:

```
public sealed12 class Monitor {  
    private Monitor();  
    public static void Enter(object obj);  
    public static void Exit(object obj);  
    public static void Wait(object obj, int msec);  
    public static void Pulse(object obj);  
    public static void PulseAll(object obj);  
    public static bool TryEnter(object obj, int msec);  
    public static bool TryEnter(object obj);  
    ...  
}
```

Die Klasse *Monitor* enthält nur statische, also Klassenmethoden. Zudem ist der Konstruktor mit dem Schlüsselwort *private* definiert. Dies bedeutet in C#, dass keine Objektinstanzen des Typs *Monitor* erzeugt werden können. Der Grund dafür ist, dass in C# jedes Objekt als „Monitor“ genutzt werden kann, weshalb es keine eigenen Monitor-Objekte geben muss.

Die wichtigsten Methoden der Klasse *Monitor* sind *Enter* und *Exit*. *Enter* dient dem Eintritt in den kritischen Abschnitt. Die Methode errichtet eine Sperre für die zu schützende Ressource. Ist die Ressource bei Aufruf von *Enter* gerade von einem anderen Thread belegt, wird der laufende Thread blockiert, bis die Sperre aufgehoben ist. Die Methode *Exit* gibt die Sperre frei. Auch ein bedingter Erwerb von Sperren ist möglich. Hierzu gibt es die Methoden *Wait* und *Pulse* bzw. *PulseAll*.

Die CLR implementiert die gesamte Synchronisationslogik. Auch bei den C#-Monitoren handelt es sich nicht um reine Monitore nach Hoare oder Brinch Hansen. Hier gelten im Wesentlichen die gleichen Argumente wie bei dem Java-Schlüsselwort *synchronized*. Beispielsweise sind nicht alle öffentlichen Methoden

¹² *sealed* bedeutet, es kann keine Subklassen von *Monitor* geben.

einer Klasse in die Synchronisationskontrolle eingeschlossen, und man kann in C# auch nicht nur `private` Variablen zulassen. Der Compiler übernimmt keine weitere Prüfung in dieser Hinsicht.

Gesperrt wird immer auf Objektebene, d.h. in der *Enter*-Methode wird ein Objekt angegeben, für das eine Sperre gesetzt werden soll. Das Sperren und Entsperren wird atomar durchgeführt.

Wie bei der Erläuterung der Java-Synchronisationsmechanismen wollen wir das Monitor-Konzept aus C# anhand des Counter-Beispiels darstellen.

Beispiel: Das bereits weiter oben dargestellte Beispiel des Zählers, der von mehreren Threads nebenläufig hochgezählt wird, soll nun in C# mit Monitoren gezeigt werden. Die Implementierung ist zwar nicht identisch zur Java-Implementierung, aber doch semantisch sehr ähnlich.

Zur Erinnerung: Das Programm startet drei nebenläufige Threads, die einen gemeinsamen Zähler jeweils um 1.000.000 hochzählen, also insgesamt auf 3.000.000 zählen. Der Zähler ist in einem Objekt der Klasse *CounterObject* gekapselt. Das Objekt stellt Methoden zum Auslesen (*get*) und zum Setzen des Zählers (*set*) bereit. Der kritische Abschnitt umfasst die drei folgenden Operationen¹³:

```
int c = myCounter.get();  
c++;  
...  
myCounter.set(c);
```

Auch hier wird der kritische Abschnitt im ersten Durchlauf ungeschützt und im zweiten Durchlauf durch einen Monitor geschützt ausgeführt.

Im Folgenden soll der C#-Code für das Beispiel kurz erläutert werden. Zunächst ist die Klasse *CounterObject* in einem eigenen Namespace *CsharpSynchronisation* dargestellt:

```
using System;  
using System.Threading;  
namespace CsharpSynchronisation  
{
```

¹³ Auch hier gilt: Man könnte natürlich einfach nur z.B. eine *addOne*-Methode definieren, die den Zähler um 1 erhöht. Damit würde man sich zwei Operationen sparen. Aber das grundsätzliche Problem bleibt. Drei Operationen bewirken mehr Konflikte beim nebenläufigen Ablauf, und das ist aus didaktischen Gründen sinnvoll.

```
/// Einfache Zählerklasse
public class CounterObject {
    private int count = 0; // Zählvariable
    public CounterObject()
    {
        // Nichts zu konstruieren
    }
    /// Zähler auf neuen Wert setzen
    public void set(int newCount)
    {
        count = newCount;
    }
    /// Zähler auslesen
    public int get()
    {
        return count;
    }
} // class CounterObject
```

Die Klasse *CountTest* dient zum synchronisierten und nicht-synchronisierten Hochzählen eines einfachen Zählers durch drei nebenläufige Threads. Der kritische Abschnitt ist jeweils kommentiert.

```
public class CountTest
{
    // Zähl-Objekt: der Zugriff auf dieses Objekt wird synchronisiert
    private CounterObject myCounter;
    private int myMaxCount; // Maximaler Wert des Zählers
    public CountTest(CounterObject c, int maxCount)
    {
        myCounter = c;
        myMaxCount = maxCount;
    }
    /// Thread kurz anhalten, um einen Fehler zu provozieren
    void MyWait(int c)
    {
        if ( (c % 100000) == 0)
```

```
{
    Thread.Sleep(0);
}

}

/// Nicht-synchronisiertes Hochzählen: provoziert eine
/// Inkonsistenz bei Nutzung durch mehrere Threads
public void CountUnsynchronized()
{
    Console.WriteLine("Thread "
        + Thread.CurrentThread.Name
        + " gestartet um " + DateTime.Now);
    for (int i=0; i<myMaxCount; i++)
    {
        // Kritischer Bereich (ungesichert)
        int c = myCounter.get();
        c++;
        MyWait(c);
        myCounter.set(c);
        // Ende des kritischen Bereichs
    }
}

/// Startmethode für Threads zum synchronisierten Hochzählen
/// Die Synchronisation wird durch die Monitor-Primitive erreicht
public void CountMonitorSynchronized()
{
    Console.WriteLine("Thread " + Thread.CurrentThread.Name
        + " gestartet um " + DateTime.Now);
    for (int i=0; i<myMaxCount; i++)
    {
        // Kritischer Bereich
        Monitor.Enter (myCounter);
        try {
            int c = myCounter.get();
            c++;
            MyWait(c);
```

```
        myCounter.set(c);
    }
    finally {
        Monitor.Exit(myCounter);
    }
    // Ende des kritischen Bereichs
}

}

/// Unsynchronisierter Test
void TestUnsynchronized()
{
    // Wir starten drei nebenläufige Threads und warten
    // auf ihr Ende
    try {
        // Drei Threads initialisieren und mit Startmethode versehen
        ThreadStart startMethod = new
            ThreadStart(this.CountUnsynchronized);
        Thread t1 = new Thread(startMethod);
        t1.Name = "t1";
        Thread t2 = new Thread(startMethod);
        t2.Name = "t2";
        Thread t3 = new Thread(startMethod);
        t3.Name = "t3";
        Console.WriteLine("Test ohne Synchronisation "
            + " geht jetzt los ...");
        // Start der Threads
        t1.Start(); t2.Start(); t3.Start();
        // Warten auf das Ende der Threads
        t1.Join(); t2.Join(); t3.Join();
    }
    catch (Exception) {
        Console.WriteLine("Thread interrupted");
    }
    finally {
        Console.WriteLine("Counter am Ende, Wert "
```

```
        + "sollte 3.000.000 sein und ist"
        + myCounter.get());
    Console.WriteLine("Test ohne Synchronisation "
        + "ist ist zu Ende");
}
// Warte auf Eingabe von der Konsole
Console.ReadLine();
}
/// Synchronisierter Test
void TestMonitorSynchronized()
{
    // Wir starten drei nebenläufige Threads und warten
    // auf ihr Ende
    try {
        // Drei Threads initialisieren und mit Startmethode versehen
        ThreadStart startMethod = new
            ThreadStart(this.CountMonitorSynchronized);
        Thread t1 = new Thread(startMethod);
        t1.Name = "t1";
        Thread t2 = new Thread(startMethod);
        t2.Name = "t2";
        Thread t3 = new Thread(startMethod);
        t3.Name = "t3";
        Console.WriteLine ("Test mit Monitor-Synchronisation "
            + "geht jetzt los ...");
        // Start der Threads
        t1.Start(); t2.Start(); t3.Start();
        // Warten auf das Ende der Threads
        t1.Join(); t2.Join(); t3.Join();
    }
    catch (Exception) {
        Console.WriteLine("Thread interrupted");
    }
    finally {
        Console.WriteLine("Counter am Ende, Wert "
```



```
        + " sollte 3.000.000" +  
        + " sein und ist" + myCounter.get();  
        Console.WriteLine("Test mit Monitor- Synchronisation "  
        + "ist zu Ende");  
    }  
    // Warten auf Eingabe von der Konsole  
    Console.ReadLine();  
}  
public static void Main(String[] args)  
{  
    // Zähler soll nur bis hierher von einem Thread erhöht werden  
    int maxCount = 1000000;  
    // Wir starten drei nebenläufige Thread, die nicht  
    // synchronisiert arbeiten und warten auf ihr Ende  
    CountTest ct1 = new CountTest(new CounterObject(),  
        maxCount);  
    ct1.TestUnsynchronized();  
    // Nun starten wir drei Monitor-synchronisierte  
    // Threads und versuchen das gleiche  
    CountTest ct2 = new CountTest(new CounterObject(),  
        maxCount);  
    ct2.TestMonitorSynchronized();  
}  
} // class CountTest  
} // namespace CSharpSynchronisation
```

Die Methode *Main* dient als Einsprungspunkt. Zunächst wird ein nicht synchronisiertes Hochzählen mit drei Threads durchgeführt (Methode *TestUnsynchronized*) und anschließend das gleiche synchronisiert mit Monitoren (Methode *TestMonitorSynchronized*). Der Ablauf des Programms führt zu folgenden Ausgaben:

Lauf 1:

```
Test ohne Synchronisation geht jetzt los ...  
Thread t1 gestartet um 01.03.2007 21:24:47  
Thread t2 gestartet um 01.03.2007 21:24:47  
Thread t3 gestartet um 01.03.2007 21:24:47  
Counter am Ende, Wert sollte 3.000.000 sein und ist 1099999
```

```
Test ohne Synchronisation ist zu Ende
Test mit Monitor-Synchronisation geht jetzt los ...
Thread t1 gestartet um 01.03.2007 21:24:51
Thread t2 gestartet um 01.03.2007 21:24:51
Thread t3 gestartet um 01.03.2007 21:24:51
Counter am Ende, Wert sollte 3.000.000 sein und ist 3000000
Test mit Monitor-Synchronisation ist zu Ende
```

Lauf 2:

```
Test ohne Synchronisation geht jetzt los ...
Thread t1 gestartet um 01.03.2007 21:28:28
Thread t2 gestartet um 01.03.2007 21:28:28
Thread t3 gestartet um 01.03.2007 21:28:28
Counter am Ende, Wert sollte 3.000.000 sein und ist 1011988
Test ohne Synchronisation ist zu Ende
Test mit Monitor-Synchronisation geht jetzt los ...
Thread t1 gestartet um 01.03.2007 21:28:31
Thread t2 gestartet um 01.03.2007 21:28:31
Thread t3 gestartet um 01.03.2007 21:28:31
Counter am Ende, Wert sollte 3.000.000 sein und ist 3000000
Test mit Monitor-Synchronisation ist zu Ende
```

Damit ist auch in C# gezeigt, wie wichtig die Synchronisation bei Multithreading-Anwendungen ist. Ohne den Einsatz von Synchronisationsmechanismen kommt es zu vielen Lost-Updates, bei Nutzung der Monitor-Klasse dagegen zu keinem einzigen.

Die Methode *Monitor.Enter* kann blockieren, wenn die Sperre gerade von einem anderen Thread gesetzt ist. Dies möchte man nicht in jedem Fall. Man kann daher auch antesten, ob die Sperre gerade gesetzt werden kann. Hierzu kann die Methode *TryEnter* verwendet werden. Die Methode ist nicht blockierend und liefert einen bool-Wert *true* zurück, wenn eine Sperre vom aktuellen Thread gesetzt wurde, andernfalls wird *false* zurückgegeben. Man kann der Methode auch einen Timeout-Wert in Millisekunden mitgeben, um die Wartezeit zu begrenzen. Wird dieser nicht angegeben, so wird gar nicht gewartet.

Die Nutzung von *TryEnter* kann wie folgt aussehen:

```
if (Monitor.TryEnter(object, 1000))
{
```

```
// Sperre holen und auf Objekt zugreifen
}  
else {  
    // Sperre nicht erhalten, noch mal versuchen  
}
```

Der Vorteil an der Nutzung von *TryEnter* ist, dass der rufende Thread etwas anderes erledigen kann, falls die Sperre nicht erhältlich ist.

6.4.5 Die C#-Synchronisationsprimitive „lock“

Die Synchronisationsprimitive *lock* bietet eine ähnliche Funktionalität wie die Klasse *Monitor* in C#. Man kann damit ebenso kritische Abschnitte für den Zugriff auf ein Objekt definieren. Im Gegensatz zu Java, wo man mit dem ansonsten semantisch recht ähnlichen Schlüsselwort *synchronized* auch den Zugriff für ganze Methoden synchronisieren kann, ist es mit *lock* wie bei der *Monitor*-Klasse nur möglich, den Zugriff auf einzelne Objekte zu synchronisieren. In Folgenden ist ein Beispiel für einen synchronisierten Anweisungsblock dargestellt, wobei auch hier ein Objekt angegeben wird, für das der Zugriff zu serialisieren ist. Möchte man den Zugriff auf mehrere Objekte gleichzeitig synchronisieren, so kann man dies z.B. so programmieren, dass alle zu sperrenden Objekte in einer gemeinsamen Objektklasse zusammengefasst werden. Dies funktioniert natürlich auch bei Einsatz der *Monitor*-Klasse.

```
...  
lock (object1)  
{  
    // Kritischer Abschnitt  
}
```

Die Aufrufe einer synchronisierten Methode werden strikt serialisiert und verzögert, wenn ein anderer Thread gerade im synchronisierten Code ist. Damit lässt sich also ein kritischer Abschnitt deklarativ implementieren. Das Sprachmittel ist demnach höherwertiger als *Monitor*, allerdings nicht so mächtig, da die Methoden *Wait*, *Pulse*, usw., die weiter unten erläutert werden, nicht verfügbar sind.

Beispiel: Das Counter-Beispiel soll hier ansatzweise mit der Synchronisationsprimitive *lock* skizziert werden. Auszutauschen ist lediglich die Methode *TestMonitorSynchronized* durch die Methode *TestLockSynchronized* (hier im Weiteren nicht dargestellt) sowie die Methode *CountMonitorSynchronized* durch *CountLockSynchronized* (siehe das folgende Coding):

```
namespace ...  
{  
    ...
```

```
public void CountLockSynchronized()
{
    Console.WriteLine("Thread " + Thread.CurrentThread.Name
        + " gestartet um " + DateTime.Now);
    for (int i=0;i<myMaxCount;i++)
    {
        // Kritischer Bereich
        lock (myCounter) {
            int c = myCounter.get();
            c++;
            MyWait(c);
            myCounter.set(c);
        }
        // Ende des kritischen Bereichs
    }
}
...
}
```

Für den Standardfall ist die Nutzung von *lock* der einfachere und auch der sicherere Mechanismus, da man nicht wie bei der Klasse *Monitor* den Aufruf *Exit* vergessen kann. Ist ein Warten auf Bedingungen notwendig, ist der Einsatz von C#-Monitoren erforderlich.

6.4.6 C#-Mutex-Objekte

Auch Mutex-Objekte verhindern in C#, dass mehrere Threads zu einem gegebenen Zeitpunkt auf eine Ressource (z.B. ein Objekt) zugreifen können. Mutex-Objekte sind also Monitor-Objekten sehr ähnlich. Sie unterscheiden sich allerdings in folgenden Punkten:

- Mit Mutex-Objekten können auch Threads synchronisiert werden, die nicht zum gleichen Prozess gehören.
- Wenn ein Thread, der gerade ein Mutex-Objekt hält, sich beendet, ohne das Mutex freizugeben, so gibt die Laufzeitumgebung auch das Mutex frei.

Die Nutzung eines C#-Mutex soll beispielhaft skizziert werden:

```
Mutex mutex1 = new Mutex();
...
mutex1.WaitOne();
try {
    ...
}
finally {
    mutex1.ReleaseMutex();
}
```

Mit der Methode *WaitOne* wird die Sperre gesetzt, mit *ReleaseMutex* wird sie wieder freigegeben. Mutexe sind innerhalb eines Prozesses wesentlich langsamer als die *Monitor*-Klasse oder die Primitive *lock*. Der eigentliche Sinn von Mutex ist die Synchronisation von Prozessen über Anwendungsgrenzen hinweg. Hierzu nutzt man sog. *benannte* Mutex-Objekte wie folgt:

```
// Anwendung A
// Der Name des benannten und global bekannten Mutex ist „MandlMutex1“
Mutex mutex = new Mutex("MandlMutex1");
...
mutex.Close();

// Anwendung B
Mutex mutex = new Mutex(false, "MandlMutex1");
...
mutex.Close();
```

Der Windows-Betriebssystemkernel verwaltet die benannten Mutex-Objekte. Bei Aufruf der Methode *mutex.WaitOne* aus einer Anwendung versucht der Betriebssystemkern eine Sperre zu setzen. Ist bereits eine Sperre auf dem benannten Mutex, wird der Thread suspendiert und muss warten. Eine Methode *Close* erlaubt das explizite Freigeben eines Mutex, wenn dieses vom Prozess nicht mehr benötigt wird.

In der Windows-API erfüllen die Methoden *CreateMutex* und *ReleaseMutex* den gleichen Zweck wie die im Beispiel gezeigten Methoden *WaitOne* und *ReleaseMutex* der C#-Klasse *Mutex*.

Beispiel: Das Counter-Beispiel soll auch hier ansatzweise mit einem Mutex gelöst werden. Auszutauschen ist lediglich die Methode *TestMonitorSynchronized* durch die Methode *TestMutexSynchronized* (hier im Weiteren nicht dargestellt) sowie die Methode *CountMonitorSynchronized* durch *CountMutexSynchronized* (siehe das folgende Coding):

```
namespace ...
{
    public class CountTest {
        public void CountMutexSynchronized()
        {
            Mutex mutex = new Mutex(false, "MandlMutex1");
            ...
            for (int i=0; i<myMaxCount; i++)
            {
                // Eintritt in kritischen Pfad, Sperre setzen
                mutex.WaitOne();
                // Kritischer Bereich
                try {
                    int c = myCounter.get();
                    c++;
                }
            }
        }
    }
}
```

```
        MyWait(c);
        myCounter.set(c);
    }
    finally {
        mutex.ReleaseMutex();
    }
    // Ende des kritischen Bereichs
}
mutex.Close();
}
}
```

Tests mit dem Beispielprogramm bestätigten, dass der Einsatz von *Mutex* erheblich langsamer ist als der Einsatz der *Monitor*-Klasse oder der Synchronisationsprimitive *lock*. Dies liegt wohl daran, dass *Mutex* für die prozessübergreifende Synchronisation gedacht ist.

6.4.7 C#-Lese- und Schreibsperrern

Lese- und Schreibsperrern in C# sind Monitoren ähnlich, und man kann mit ihnen ebenfalls das Mutual-Exclusion-Problem lösen. Sie sind aber ein bisschen intelligenter: Mehrere Threads können durch Nutzung dieses Mechanismus nämlich gleichzeitig lesen, aber sich überschneidende Lese- und Schreibaktionen werden synchronisiert. Von der FCL wird hierfür die Klasse *ReaderWriterLock* bereitgestellt.

Der lesende Thread nutzt die Methode *AcquireReaderLock* zum Anfordern und *ReleaseReaderLock* zum Freigeben der Sperre. Beim Anfordern der Sperre ist ein Timeout-Wert angebbar (in Millisekunden). Nach Ablauf der Timeout-Zeit wird das Warten abgebrochen, sofern nicht vorher der Lock zugewiesen wurde. Ein schreibender Thread nutzt auch die Methoden *AcquireWriterLock* und *ReleaseWriterLock*. Weitere Methoden sind *IsReaderLockHeld* oder *IsWriterLockHeld*. Mit diesen Methoden kann man prüfen, ob der laufende Thread bereits den Lock hält.

Beispiel: Es wird wieder das Counter-Beispiel herangezogen und mit C#-Schreibsperrern gelöst. Die Sperre wird als private, statische Variable namens *rwlock* in der Klasse *CountTest* definiert.

```
namespace ...
{
    public class CountTest
    {
        ...
        // Objektweit zugängliche globale Lockvariable für
        // ReaderWriterLock-Mechanismus
        private static ReaderWriterLock rwlock = new ReaderWriterLock();
        ...
        public void CountWriterLockSynchronized()
```

```

{
    ...
    for (int i=0;i<myMaxCount;i++)
    {
        // Eintritt in kritischen Pfad, Sperre setzen
        rwlock.AcquireWriterLock(Timeout.Infinite);
        // Kritischer Bereich
        try {
            int c = myCounter.get();
            c++;
            MyWait(c);
            myCounter.set(c);
        }
        finally {
            rwlock.ReleaseWriterLock();
        }
        // Ende des kritischen Bereichs
    }
}
    ...
}

```

Anmerkung: Im Test wurde beobachtet, dass dieser Synchronisationsmechanismus ebenfalls langsamer ist als der *Monitor*- und der *lock*-Mechanismus. Nur der Einsatz von *Mutex* war noch langsamer.

6.4.8 C#-Interlocked-Klasse

Die *Interlocked*-Klasse entspricht der Funktionalität eines TSL-Befehls auf C#-Ebene. Man kann mit *Interlocked* das Austauschen einer Variablen mit einem anderen Wert in einer nicht unterbrechbaren (atomaren) Operation ausführen. Zudem ist das Hochzählen und Vermindern einer Integervariablen möglich. Die Klassendefinition sieht grob wie folgt aus:

```

public sealed class Interlocked {
    private Interlocked();
    ...
    public static int Increment(ref int);
    public static int Decrement(ref int location);
    public static int Exchange(ref int location1, int value);
    public static int CompareExchange(ref int location1, int value,
        int comparand);
}

```

Folgende Methoden sind u.a. definiert:

- *Increment* zum Hochzählen einer Integervariablen. Inkrementiert den Wert einer angegebenen Variablen und speichert das Ergebnis in einer atomaren Operation.

- *Decrement* zum Vermindern einer Integervariablen. Dekrementiert den Wert einer angegebenen Variablen und speichert das Ergebnis in einer atomaren Operation.
- *Exchange* legt eine 32-Bit-Ganzzahl mit Vorzeichen in einer atomaren Operation auf einen angegebenen Wert fest und gibt den ursprünglichen Wert zurück.
- *CompareExchange* zum Vergleichen und Austauschen einer Integervariablen. Wenn *comparand* und *location1* gleich sind, wird *value* in *location1* gespeichert. Andernfalls wird keine Operation ausgeführt. Die Vergleichs- und die Austauschoperation werden als atomare Operation ausgeführt.

Die zu verändernden Parameter werden in den einzelnen Methoden als Referenzen (*ref*) angegeben. Zwei Threads können bei entsprechender Nutzung der Klasse *Interlocked* nicht zur selben Zeit auf eine Variable zugreifen, auch dann nicht, wenn sie auf verschiedenen CPUs ablaufen.

Beispiel: Für das Counter-Beispiel wird in der Klasse *CountTest* eine private und statische Lockvariable *lockvariable* genutzt. Sie wird mit 0 initialisiert, was im Beispiel bedeutet, dass der Lock nicht gesetzt ist. Die Methode *Exchange* wird verwendet, um die Variable von 0 (nicht gesperrt) auf 1 (gesperrt) zu setzen. *Exchange* liefert den alten Wert der Lockvariable.

```
namespace ...
{
    public class CountTest
    {
        ...
        // Objektweit zugängliche Lockvariable für Interlock-Mechanismus
        private static int lockvariable = 0;
        ...
        public void CountInterlockedSynchronized()
        {
            for (int i=0; i<myMaxCount; i++) {
                int before = 1;
                while (before == 1) {
                    // solange warten, bis der laufende Thread den Lock erhält
                    before = Interlocked.Exchange(ref lockvariable, 1);
                    if (before == 0) {
                        // Lock erhalten
                        // Kritischer Bereich
                        try {
                            int c = myCounter.get();
                            c++;
                            MyWait(c);
                            myCounter.set(c);
                        }
                        finally {
                            // Lock wieder freigeben
                            Interlocked.Decrement(ref lockvariable);
                        }
                    }
                }
            }
        }
    }
}
```



```
        }  
        // Ende des kritischen Bereichs  
    }  
    else {  
        // Thread suspendieren, um Polling zu vermeiden  
        Thread.Sleep(1);  
    }  
}  
}  
}  
...  
}
```

6.4.9 Warten auf Bedingungen in C#

Ähnlich wie in Java gibt es in C# die statischen Methoden *Wait*, *Pulse* (in Java *notify*) und *PulseAll* (in Java *notifyAll*). *Wait* gibt vorübergehend eine vom aufrufenden Thread gehaltene Sperre frei und blockiert den aufrufenden Thread, bis dieser die Sperre erneut erhält. *Pulse* und *PulseAll* dienen der Benachrichtigung der Threads, die mit *Wait* in den Wartezustand gegangen sind. Damit kann die Blockierung eines Threads aufgehoben werden. *PulseAll* ermöglicht allen wartenden Threads die Ausführung. Welcher Thread nun tatsächlich die CPU erhält, hängt von der aktuellen Scheduling-Situation ab.

Die Nutzung von *Wait/Pulse* ist auch in C# laufezeitkritisch hinsichtlich der Reihenfolge der Aufrufe. In nebenläufigen Ausführungseinheiten kann es auch hier wie in Java vorkommen, dass ein Thread die Methode *Pulse* aufruft, bevor der eigentlich wartende Thread eine *wait*-Methode ausführt. In diesem Fall passiert das gleiche wie in Java: Der wartende Thread erhält das Signal möglicherweise nicht mehr. Die Lösung ist hier ebenfalls, den *Wait*-Aufruf in einer while-Schleife zu programmieren. Die Methoden *Wait* und *Pulse* bzw. *PulseAll* sind nur sinnvoll nutzbar, wenn der aktuelle Thread gerade im kritischen Abschnitt ist, also die Sperre hält.

6.5 Kommunikation von Prozessen und Threads

In diesem Abschnitt werden wir die grundlegenden Begriffe und Möglichkeiten der Kommunikation von Prozessen bzw. Threads innerhalb eines Rechnersystems betrachten. Kommunikation ist grundsätzlich der Austausch von Informationen nach bestimmten Regeln. Dies ist zwischen Menschen ähnlich wie zwischen Maschinen. Das Regelwerk fasst man in der Datenkommunikation bzw. bei Rechnernetzen unter dem Begriff *Protokoll* zusammen. Die rechnerübergreifende Kommunikation in Netzen ist zwar nicht Inhalt dieses Kapitels, einige Begriffe gelten aber

sowohl für die rechnerinterne als auch für die rechnerübergreifende Kommunikation.

6.5.1 Grundbegriffe der Kommunikation

Wie bereits angedeutet, können Prozesse und Threads lokal auf dem gleichen Rechner miteinander kommunizieren oder auch rechnerübergreifend in einer verteilten Umgebung. Bei beiden Möglichkeiten gibt es einige Parallelen. Dies veranlasst uns dazu, zunächst einige Grundbegriffe der Kommunikation zu erläutern, die der Einordnung und Unterscheidung verschiedener Kommunikationsformen dienen soll. Man unterscheidet:

- Speicherbasierte versus nachrichtenbasierte Kommunikation
- Synchrone versus asynchrone Kommunikation
- Verbindungsorientierte versus verbindungslose Kommunikation
- Halbduplex- versus Vollduplex-Betrieb
- Varianten der Empfängeradressierung

Diese Unterscheidungsmerkmale sollen im Weiteren diskutiert werden (siehe auch Glatz 2006).

Speicherbasierte versus nachrichtenbasierte Kommunikation

Speicherbasierte Kommunikation erreicht man durch die Verwaltung eines gemeinsamen Speichers, den alle Prozesse/Threads, die miteinander kommunizieren, im Zugriff haben. Bei der *nachrichtenbasierten Kommunikation* tauschen die kommunizierenden Prozesse/Threads definierte Nachrichten aus.

Beide Varianten sind sowohl rechnerübergreifend als auch rechnerlokal möglich. Für die Kommunikation innerhalb eines Rechners kann sowohl das Laufzeitsystem als auch das Betriebssystem zuständig sein. Kommunizierende Threads innerhalb eines Java-Prozesses steuert beispielsweise die JVM, also die Java-Laufzeitumgebung. Bei einer prozessübergreifenden Kommunikation werden meist Dienste des Betriebssystems genutzt.

Synchrone versus asynchrone Kommunikation

Bei der nachrichtenbasierten Kommunikation unterscheidet man weiterhin zwischen *synchroner* sowie *asynchroner* Kommunikation. Synchron bedeutet in diesem Zusammenhang, dass Sender und/oder Empfänger blockieren, um sich abzustimmen. Ein Sender sendet z.B. eine Nachricht über den Aufruf eines Dienstes (send) ab und wartet so lange, bis der Empfänger die Nachricht über einen weiteren Dienst (receive) gelesen hat. Umgekehrt kann es so organisiert sein, dass ein Empfänger zunächst einen receive-Dienst aufruft und damit blockiert, bis ein Sender die Nachricht an ihn sendet.

Abbildung 6-15 zeigt eine synchrone Kommunikation. Der Sender sendet eine Nachricht, die der Empfänger entgegennimmt, verarbeitet und mit einer weiteren Nachricht das Ergebnis der Anfrage sendet. Der Sender blockiert in einem receive-Aufruf solange, bis das Ergebnis des Empfängers ankommt. Hier spricht man von einem synchronen Dienstaufwurf im Rahmen einer sog. *Zweiwegekommunikation*. Andere Varianten sind ebenfalls möglich. Dies hängt davon ab, wie die Interaktionsform genau ausschauen soll. Wenn es nicht notwendig ist, dass ein Empfänger gleich antwortet, so spricht man auch von einer *Einwegekommunikation*. Auch die Entgegennahme der Nachricht und das reine Bestätigen der Ankunft durch eine Bestätigungsnachricht des Empfängers ist eine sinnvolle Variante. Das Ergebnis der Anfrage kann dann später geliefert werden.

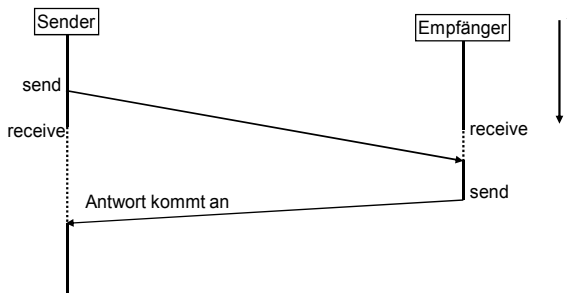


Abbildung 6-15: Synchrone Kommunikation

Bei der *asynchronen* Kommunikation wird im Kommunikationssystem ein Puffer bzw. eine Nachrichtenwarteschlange verwaltet, die es dem Sender ermöglicht, die Nachricht dort abzulegen und mit anderen Aufgaben ohne Blockieren weiterzumachen. Der Empfänger kann bei Gelegenheit die Nachricht aus der Nachrichtenwarteschlange lesen und verarbeiten.

Abbildung 6-16 zeigt ein Beispiel einer asynchronen Kommunikation. Der Sender sendet seine Nachricht, die durch das Kommunikationssystem zwischengepuffert und bei Gelegenheit durch den Empfänger ausgelesen und verarbeitet wird. Der Sender wartet in diesem Beispiel nach einigen anderen Aktionen auf eine Nachricht des gleichen oder eines anderen Partners.

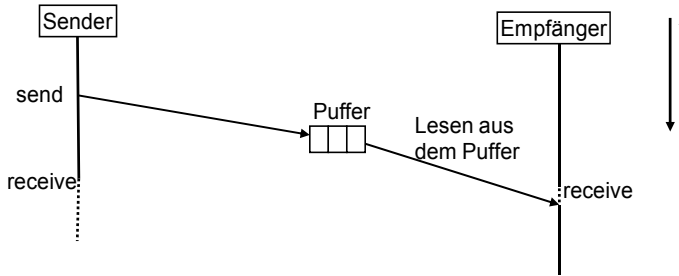


Abbildung 6-16: Asynchrone Kommunikation

Eine weitere Variante zeigt Abbildung 6-17. Hier bestätigt das Kommunikationssystem (Kom-System) dem Sender, dass die Nachricht empfangen wurde. So weiß der Sender, dass die Nachricht im Puffer liegt und irgendwann einmal vom adressierten Empfänger gelesen werden kann. Wie man sieht, ist bei dieser Form der Kommunikation die Verarbeitung vom Senden der Nachricht vollständig entkoppelt. Damit können also lose gekoppelte Anwendungssysteme kommunizieren. Bei hohen Anforderungen an die Zuverlässigkeit der Nachrichtenzustellung muss das Kommunikationssystem sicherstellen, dass die Nachrichten evtl. auch bei schwerwiegenden Fehlerfällen schließlich irgendwann einmal ausgeliefert werden. Ein Beispiel für eine hohe Zuverlässigkeitsanforderung ist die Zustellung einer Abbuchungsnachricht von einem Online-Banking-System an ein Kontoführungssystem einer Bank. Am Rande sei erwähnt, dass man für diese Aufgabe persistente und transaktionsgesicherte Nachrichtenwarteschlangensysteme einsetzt.

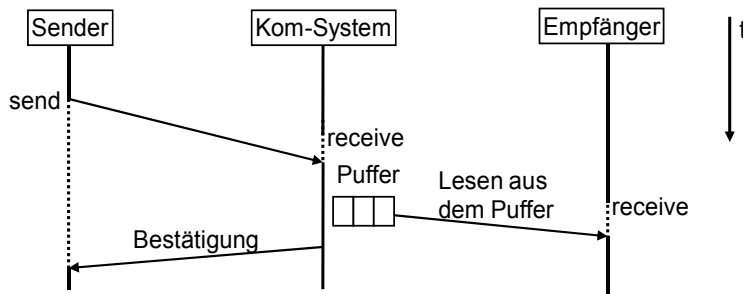


Abbildung 6-17: Asynchrone Kommunikation mit Bestätigung des Empfangs

Je nach Anforderung ist die eine oder andere Variante sinnvoller. Die klassische Client-/Server-Kommunikation ist meist synchron organisiert, sog. Message-Passing-Systeme sind dagegen asynchron organisiert. Letztere benötigt man z.B., wenn der Empfänger einer Nachricht nicht immer erreichbar ist, etwa weil er nur zu bestimmten Zeiten aktiv ist.

Bei der asynchronen, nachrichtenorientierten Kommunikation unterscheidet man nochmals zwei grundlegende Kommunikationsmodelle:

- *Publish-Subscribe-Modell*: Bei diesem Modell sendet ein Publisher Nachrichten bestimmten Typs an mehrere Interessenten (sog. Subscriber). Ein Empfänger kann sich als Subscriber für ein oder mehrere Themen (Topics) registrieren.
- *Point-to-Point-Modell*: In diesem Modell erfolgt eine Punkt-zu-Punkt-Kommunikation zwischen zwei Partnern, wobei die Nachrichten über eine Warteschlange übermittelt werden.

Die Implementierung dieser Mechanismen geschieht heute meist über nachrichtenorientierte Middlewaresysteme.¹⁴

Welche Interaktionsform und welches Kommunikationsmodell angewendet werden, hängt von den Anforderungen an die Kommunikation ab. Wenn es sehr wichtig ist, dass der Sender erfährt, ob die Nachricht beim Empfänger angekommen ist, wird man eine andere Kommunikationsform wählen, als wenn dies keine Rolle spielt. Heutige Client-/Server-Systeme, die über höhere Kommunikationsparadigmen wie z.B. RPC (Remote Procedure Calls) realisiert werden, nutzen meist den sog. synchronen Dienstaufwurf, bei dem der Client eine Anfrage sendet und blockierend auf die Antwort des Servers wartet.

Verbindungsorientierte versus verbindungslose Kommunikation

Bei der nachrichtenbasierten Kommunikation unterscheidet man zudem zwischen *verbindungsorientierter* und *verbindungsloser* Kommunikation. Dieses Unterscheidungsmerkmal bezieht sich darauf, ob zwischen zwei Kommunikationspartnern eine logische Verbindung, ein Verbindungskanal, aufgebaut werden soll oder nicht. Wird über längere Zeit kommuniziert, bietet sich meist eine Verbindung an. In diesem Fall müssen sich Sender und Empfänger Informationen über den anderen Partner, den sog. *Verbindungskontext* merken. Dieser wird in einer Verbindungsaufbauphase ausgetauscht. Hierzu gehören z.B. Adressierungsinformationen. Bei der weiteren Kommunikation ist dann aber nur noch ein Verbindungskennzeichen in der Nachricht anzugeben. Typischerweise besteht eine verbindungsorientierte Kommunikation aus den Kommunikationsphasen Verbindungsaufbau, Datenübertragung und Verbindungsabbau. Bei einer verbindungslosen Kommunikation ist dagegen kein expliziter Verbindungsaufbau notwendig. Die Nachrichten enthalten hier immer die vollständige Adressierungsinformation, damit der Empfänger weiß, woher die Nachricht kommt.

¹⁴ Produkte sind z.B. JBoss MQ oder Websphere MQ.

Halbduplex- versus Vollduplex-Betrieb

Bei der nachrichtenbasierten Kommunikation unterscheidet man auch noch, ob beide Partner gleichzeitig senden können (bidirektional), oder nur einer nach dem anderen (unidirektional). Man spricht im ersten Fall von *Vollduplex*- und im zweiten Fall von *Halbduplex*-Betrieb. Beide Betriebsarten sind in Abbildung 6-18 skizziert. Im Übrigen spricht man von Simplex-Betrieb, wenn die Nachrichtenübertragung immer nur in eine Richtung, ohne einen Richtungswechsel, erfolgt.

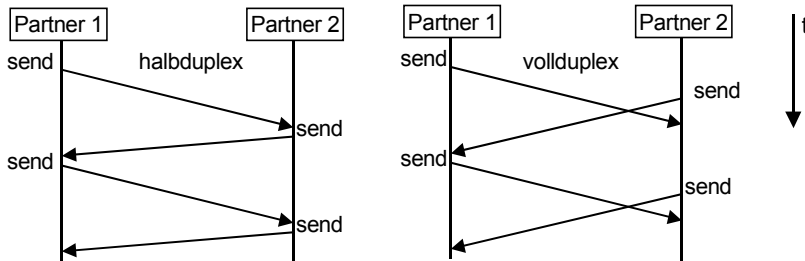


Abbildung 6-18: Halbduplex versus Vollduplex

Varianten der Empfängeradressierung

Schließlich unterscheidet man bei der nachrichtenorientierten Kommunikation noch verschiedene Arten der Adressierung und damit die Anzahl der Kommunikationspartner, denen während der Kommunikation Nachrichten gesendet werden:

- *Unicast*: Dies ist der klassische Fall, von dem wir bisher ausgegangen sind. Ein Partner kommuniziert mit genau einem anderen Partner und muss daher auch nur diesen adressieren (Beziehung 1:1).
- *Multicast*: Hier adressiert der Sender eine definierte Gruppe von Empfängern. Eine Nachricht wird also an die gesamte Gruppe gesendet (Beziehung 1:n, wobei n die Anzahl der Gruppenmitglieder ist).
- *Anycast*: Diese Art der Empfängeradressierung wendet sich auch an eine Gruppe, allerdings nimmt hier mindestens ein Empfänger, aber nicht zwangsweise alle Nachrichten entgegen (Beziehung 1:1...n).
- *Broadcast*: Diese Adressierung sendet Nachrichten an alle möglichen Empfänger, z.B. alle entfernten Partner in einem Netzwerk oder lokale Partnerprozesse auf einem bestimmten Rechner (Beziehung 1:m, wobei m die Anzahl aller möglichen Partner ist).
- Ob nun tatsächlich nur eine physikalische Nachricht gesendet wird, oder ob z.B. bei Multicast eine Nachricht pro Gruppenmitglied durch das Netz oder über das Betriebssystem gesendet wird, ist von der Implementierung des Kommunikationssystems abhängig.

Die Adressierung der Empfänger erfolgt abhängig von den Vorgaben des Kommunikationssystems. Bei der rechnerübergreifenden Kommunikation erhält jeder Empfänger (und natürlich auch der Sender) eine Adresse. Bei TCP-basierter Kommunikation ist dies z.B. die Adresse des Rechners (IP-Adresse), kombiniert mit einem Port, der von der Kommunikationsanwendung reserviert wird (Abbildung 6-19).

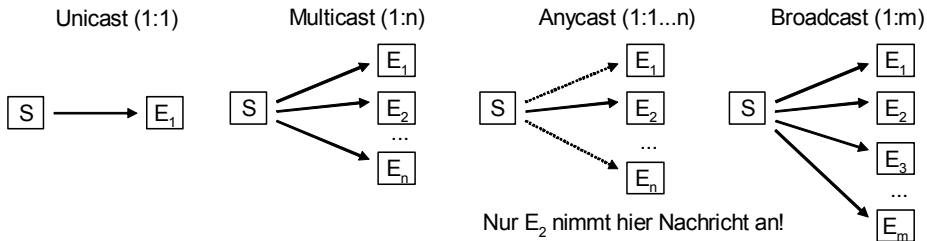


Abbildung 6-19: Varianten der Empfängeradressierung

6.5.2 Möglichkeiten für eine rechnerinterne Kommunikation

Die lokale Kommunikation bezieht sich auf einen Rechnerknoten. Prozesse bzw. deren Threads können mit anderen Prozessen über Prozessgrenzen hinweg kommunizieren. Im Folgenden wird ein grober Überblick über einige Kommunikationsmechanismen gegeben. Anhand eines konkreten Fallbeispiels wird eine häufig verwendete Variante etwas näher betrachtet.

Unter lokaler Kommunikation verstehen wir die nachrichten- oder speicherbasierte Kommunikation zwischen Prozessen oder Threads innerhalb eines Betriebssystems. Hierfür gibt es verschiedene Möglichkeiten.

Typische Kommunikationsmechanismen sind u.a.:

- Gemeinsame Nutzung von Datenbereichen durch Threads innerhalb eines Prozesses (prozessintern)
- Gemeinsame Nutzung von Datenbereichen über Shared Memory (prozessübergreifend)
- Gemeinsame Nutzung von Daten über externe Speicherobjekte wie Dateien (prozessübergreifend)
- Kommunikation über verschiedene Interprozesskommunikationsmechanismen wie Pipes, Message Queues oder lokale Sockets (prozessübergreifend)

Nutzung von Datenbereichen durch Threads in Prozessen:

Die Kommunikation von Threads innerhalb eines Prozesses kann einfach durch die Nutzung globaler Datenbereiche des Prozesses ermöglicht werden, da alle Threads eines Prozesses Zugang zum gesamten Adressraum haben. Man kann also eine oder mehrere globale Variablen bzw. noch besser sauber gekapselte Objekte, für den Austausch von Informationen nutzen.

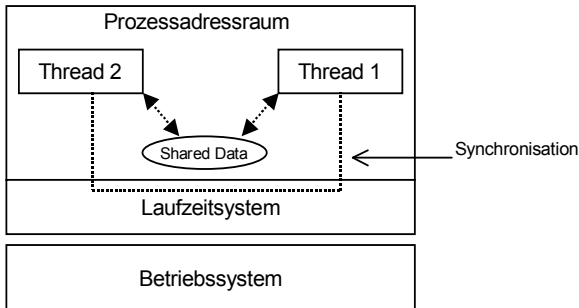


Abbildung 6-20: Prozessinterne Interprozesskommunikation allgemein

Diese Art der Kommunikation ist in Abbildung 6-20 skizziert. Zur Koordination des gemeinsamen Zugriffs auf die *Shared Data* sind allerdings Synchronisationsmaßnahmen erforderlich, um Inkonsistenzen zu vermeiden (Locks, Semaphore,...).

Gemeinsame Nutzung von Datenbereichen über sog. shared Memory

Threads verschiedener Prozesse können über einen gemeinsam genutzten Speicherbereich im Hauptspeicher (Shared Memory) kommunizieren. Dies sind Speicherbereiche, die mit Hilfe des Betriebssystems in mehrere Adressräume kopiert werden. Dabei spielt es keine Rolle, wo die Bereiche in den Prozessadressräumen liegen. Wichtig ist, dass die Daten nur einmal im Hauptspeicher vorhanden sind.

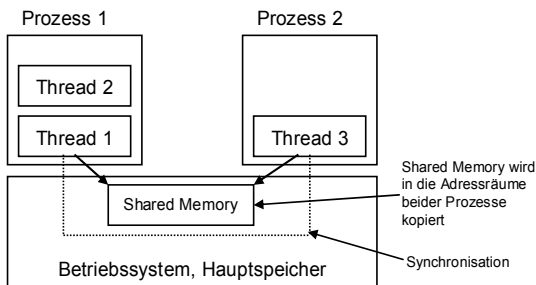


Abbildung 6-21: Interprozesskommunikation über Shared Memory

Für eine ordnungsgemäße Kommunikation sind neben den Mechanismen zum Datenaustausch auch die bereits erläuterten Mechanismen zur Synchronisation nebenläufiger Prozesse und Threads erforderlich. In Abbildung 6-21 kommunizieren der Thread 1 aus Prozess 1 mit dem Thread 3 aus Prozess 2.

Das Betriebssystem stellt Mechanismen bereit, die es einem Anwendungsprogrammierer ermöglichen, einen Shared Memory zu erzeugen, Daten aus diesen zu lesen und in diesen einzutragen. Die Bereitstellung erfolgt über Systemdienste in Form von APIs.

Gemeinsame Nutzung von Daten über externe Speicherobjekte wie Dateien

Eine einfache Variante der Kommunikation ist der Austausch von Dateien (siehe Abbildung 6-22). Ein Prozess kann Daten in Dateien ablegen, die ein anderer Prozess lesen kann. Das Lesen und Schreiben erfordert ebenfalls eine explizite Synchronisation. Betriebssysteme bzw. deren Dateiverwaltungen bieten meist einen Lock-Mechanismus an, der ganze Dateien sperrt. Prozesse bzw. Threads, die auf eine Datei zugreifen, können zunächst versuchen, diesen Lock zu erhalten. Es ist aber auch eine Synchronisation über die klassischen Mechanismen (Semaphore usw.) möglich.

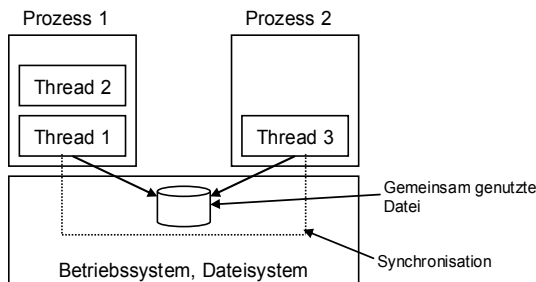


Abbildung 6-22: Interprozesskommunikation über Dateien

Die Kommunikation über Dateiaustausch kann auch rechnerübergreifend verwendet werden und wird heute in der Praxis insbesondere bei unternehmensübergreifender Kommunikation immer noch sehr häufig genutzt.

Kommunikation über verschiedene IPC-Mechanismen

Threads verschiedener Prozesse können auch über Kommunikationsdienste, die das Betriebssystem in Form von APIs bereitstellt, miteinander kommunizieren. Typische Kommunikationsdienste, die in Betriebssystemen wie Windows und Unix angeboten werden, sind *Pipes*, *FIFOs*, *Message Queues*, *Sockets* usw.

Threads verschiedener Prozesse bauen eine Kommunikationsverbindung auf und kommunizieren über diese direkt, wie es in Abbildung 6-23 dargestellt ist.

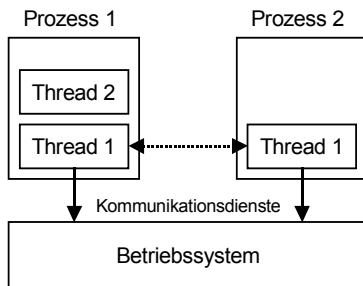


Abbildung 6-23: Prozessübergreifende Interprozesskommunikation allgemein

Pipes sind Einweg-Kommunikationskanäle, die es einem Prozess ermöglichen, Daten über das Betriebssystem als Datenstrom an einen anderen Prozess zu übertragen. Pipes werden zur Laufzeit erzeugt und nach Beendigung der kommunizierenden Prozesse auch wieder zerstört. *FIFOs* (FIFO kommt von „First In, First Out“) sind benannte Pipes, die üblicherweise vom Betriebssystem wie Dateien behandelt werden. Sie erhalten einen Namen und können von beliebigen Prozessen, die eine Berechtigung erhalten, verwendet werden. FIFOs überleben einen Prozess und sogar einen Neustart eines Betriebssystems.

Message Queues sind Nachrichtenwarteschlangen, über die mehrere Prozesse miteinander kommunizieren können. Mit Message Queues kann man Point-to-Point, aber auch Publish-Subscribe-Kommunikation realisieren. Während Pipes üblicherweise einen unbegrenzten Datenstrom realisieren, wird mit Message Queues eine Kommunikation auf Basis abgegrenzter Nachrichten definierter Länge ermöglicht.

Sockets werden üblicherweise zur Kommunikation von entfernten Prozessen auf Basis der Internet-Standardkommunikationsprotokolle TCP (Transport Control Protocol) und UDP (User Datagram Protocol) verwendet. Sie sind aber auch ein einfacher und praktischer, wenn auch nicht so leistungsfähiger Mechanismus für die interne Kommunikation von Prozessen. Über TCP-Sockets kann eine verbindungsorientierte, über UDP-Sockets eine verbindungslose Kommunikation realisiert werden. Die Prozesse werden anhand der gleichen IP-Adresse (IP-Loopback-Adresse) sowie über sog. Portnummern identifiziert. Man spricht hier von Loopback-Kommunikation. Das Betriebssystem sorgt dafür, dass die Daten nur rechnerintern von Prozess zu Prozess übertragen werden. Darauf aufbauend können auch

höhere Kommunikationsdienste wie RPC¹⁵ zur lokalen Kommunikation verwendet werden. Wenn allerdings hohe Leistung gefordert wird, greift man eher auf IPC-Mechanismen wie Shared Memory zurück.

Dies sind nur einige Möglichkeiten zur IPC-Kommunikation. Verschiedene Betriebssysteme bieten hierfür auch unterschiedliche Mechanismen an. Hostsysteme unterstützen beispielsweise ganz spezielle APIs, die hier nicht näher betrachtet werden sollen.

6.5.3 Fallbeispiel: Pipes

Stellvertretend für die genannten IPC-Mechanismen wird im Folgenden der Pipes-Mechanismus beschrieben, der in mehreren Betriebssystemen unterstützt wird und ursprünglich aus der Unix-Welt stammt. Der Begriff *Pipe* ist eine Abkürzung für Pipeline. Wie der Name schon andeutet, handelt es sich hier um einen Datenstrom von einem Prozess zu einem anderen. Daten, die ein Prozess in ein Ende der Pipe schreibt, können auf der anderen Seite von einem anderen Prozess gelesen werden. Eine Pipe stellt damit einen unidirektionalen Datenstrom bereit. Die Daten werden von einem Prozess ohne Nachrichtenbegrenzer in die Schreibseite eingetragen, die von einem lesenden Prozess aus dem anderen Ende (Leseseite) entnommen werden können. Eine bidirektionale Kommunikation zweier Prozesse kann über die Einrichtung von zwei Pipes erreicht werden (siehe Abbildung 6-24).¹⁶

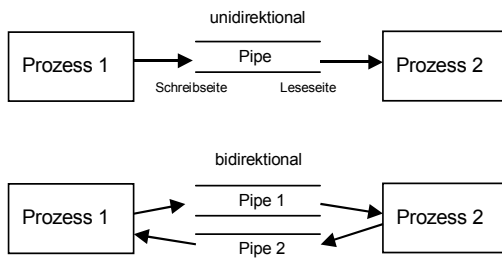


Abbildung 6-24: Pipe als unidirektionaler oder bidirektionaler Datenstrom

Eine Pipe stellt einen Pufferbereich fester Länge dar. Die Länge ist abhängig vom Betriebssystem. Typische Werte liegen bei 5 KB. Eingetragene Daten bleiben so

¹⁵ Remote Procedure Call (RPC): Ein Mechanismus zum entfernten Aufruf von Prozeduren, aus Sicht des Programmierers transparent, so als ob sie lokal vorhanden wären.

¹⁶ Es gibt auch Betriebssysteme, die bidirektionale Pipes anbieten.

lange im Puffer, bis sie ausgelesen werden. Das Auslesen erfolgt in der Reihenfolge, in der die Daten eingetragen werden (FIFO-Puffer).

Für den Zugriff auf eine Pipe werden vom Betriebssystem Synchronisationsmechanismen bereitgestellt. In eine volle Pipe kann z.B. nichts geschrieben und aus einer leeren Pipe nichts gelesen werden. Die Prozesse blockieren beim Versuch, dies zu tun. Aber auch nicht-blockierende Zugriffe sind verfügbar. In diesem Fall empfiehlt sich die Synchronisation zwischen schreibendem und lesendem Prozess über weitere Synchronisationsmechanismen. Man denke nur z.B. daran, ein Erzeuger-/Verbraucher-System über Pipes zu realisieren.

Pipes werden auch genutzt, um die Standardausgabe eines Prozesses mit der Standardeingabe eines weiteren Prozesses zu verbinden. Eine typische Anwendung unter Unix ist die Befehlsverkettung auf der Shellebene. Das Beispiel in Abbildung 6-25 zeigt eine Verkettung der Unix-Shell-Befehle *who*, *sort* und *lpr*. Mit dem Befehl *who* wird ausgegeben, wer sich gerade im System befindet, mit *sort* werden die Ergebnisse von *who* sortiert und mit dem Befehl *lpr* wird die sortierte Liste ausgegeben. Der erste Befehl schreibt sein Ergebnis in die Pipe, indem die Standardausgabe (stdout) umgelenkt wird. Der zweite Befehl legt seine Standardeingabe (stdin) auf die Leseseite dieser Pipe usw. In Unix-Shells ist das Zeichen „|“ das sog. Pipe-Symbol, das dazu dient, zwischen zwei Kommandos, die definitionsgemäß in eigenen Prozessen ablaufen, eine Pipe einzurichten und gleichzeitig die Standardein- bzw. -ausgabe umzulenken.

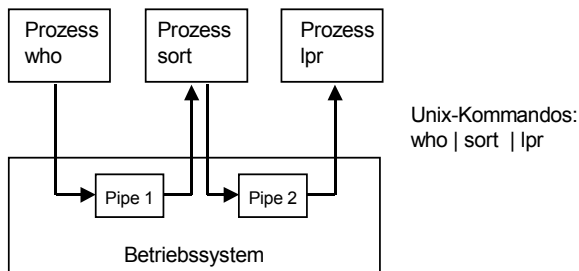


Abbildung 6-25: Pipes für die Übertragung von Daten von Befehl zu Befehl

Für die Nutzung von Pipes in Programmen stellen die Betriebssysteme entsprechende APIs bereit. Unter Unix gibt es zum Arbeiten mit Pipes u.a. die Systemfunktionen *pipe* bzw. *popen*, *close* bzw. *pclose*, *write* und *read* und *dup* bzw. *dup2*. *pipe* bzw. *popen* dienen zum Anlegen einer Pipe, *close* bzw. *pclose* zum Entfernen einer Pipe, *write* und *read* zum Schreiben bzw. Lesen und *dup* bzw. *dup2* zum Ver-

binden der Ein- und Ausgänge mit der Standardeingabe bzw. Standardausgabe eines Prozesses.

Ein Elternprozess erzeugt gewöhnlich eine Pipe, die im System wie eine Datei durch einen *Filedescriptor* identifiziert wird. Weiterhin erzeugt der Elternprozess anschließend zwei Kindprozesse, die die Filedescriptoren und damit den Zugang zur Pipe erben.

Üblicherweise schließt der schreibende Prozess dann die Leseseite der Pipe und der lesende Prozess die Schreibseite. Das folgende Programmstück in der Sprache C soll die Anwendung von Pipes verdeutlichen (Glatz 2006):

```
int main()
{
    int fds[2]; // Filedescriptoren für Pipe
    char *text = "Hi, wie geht es!\n";
    char buffer[5];
    int count, status;
    // Pipe erzeugen
    pipe(fds);
    if (fork() == 0) {
        // 1. Kindprozess, Standardausgabe auf Pipe Schreibseite
        // (Pipe-Eingang) legen und Pipe-Leseseite (Pipe-Ausgang)
        // schließen (wird nicht benötigt)
        dup2(fds[1], 1);
        close(fds[0]);
        write (1, text, strlen(text)+1);
    }
    else {
        if (fork() == 0) {
            // 2. Kindprozess, Pipe-Leseseite (Pipe-Ausgang) auf
            // Standardeingabe umlenken und Pipe-Schreibseite
            // (Pipe-Eingang) schließen
            dup2(fds[0], 0);
            close(fds[1]);
            while (count = read(0, buffer, 4))
            {
                // Pipe in einer Schleife auslesen
            }
        }
    }
}
```

```
        buffer[count] = 0; // String terminieren
        printf("%s", buffer) // und ausgeben
    }
}
else {
    // Im Vaterprozess: Pipe an beiden Seiten schließen und
    // auf das Beenden der Kindprozesse warten
    close(fds[0]);
    close(fds[1]);
    wait(&status);
    wait(&status);
}
exit(0);
}
```

Die Arbeitsweise dieses einfachen Testprogramms ist in Abbildung 6-26 skizziert. Das Programm erzeugt zunächst eine Pipe und anschließend zwei neue Kindprozesse. Für jede Pipe-Seite wird bei Aufruf von *pipe(fds)* ein Filedescriptor zurückgegeben und in das Integer-Array mit dem Namen *fds* eingetragen.

Der erste Kindprozess schließt den Pipe-Ausgang und lenkt die Standardausgabe auf den Pipe-Eingang (Schreibseite) um. Der zweite Kindprozess schließt den Pipe-Eingang und lenkt die Standardeingabe auf den Pipe-Ausgang (Leseseite) um. Der erste Kindprozess schreibt einen String in die Pipe, den der zweite Kindprozess in einer Schleife ausliest und auf Konsole ausgibt.

Der Vaterprozess schließt beide Pipe-Enden und wartet mit zwei *wait*-Aufrufen auf die Beendigung beider Kindprozesse.

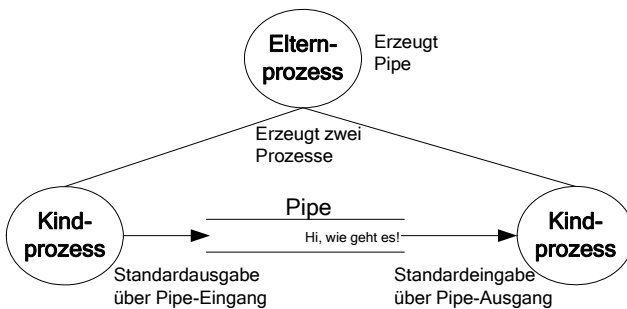


Abbildung 6-26: Ablauf zum Pipes-Beispiel

Die skizzierten, sog. „Unnamed Pipes“ (namenlose Pipes) dienen in erster Linie dazu, verwandte Prozesse miteinander kommunizieren zu lassen. Darüber hinaus gibt es noch sog. „Named Pipes“, die wie Dateien sprechende Namen bekommen. Über diese benannten Pipes können auch Prozesse, die nicht miteinander verwandt sind, kommunizieren. Named Pipes werden auch als FIFOs bezeichnet. Beliebige Prozesse können auf FIFO's zugreifen. FIFO-Implementierungen gibt es ebenfalls in den gängigen Unix-Derivaten und unter Windows. Sie werden üblicherweise auf der Kommandoebene, etwa unter Unix mit dem Befehl *mknod* erzeugt, aber es gibt auch einen entsprechenden Systemaufruf hierfür, um das Erzeugen im Programm vorzunehmen.

Unter Windows gibt es ähnliche Funktionen zur Behandlung von Pipes wie unter Unix. Man unterscheidet auch hier zwischen „Named Pipes“ und „Unnamed Pipes“. Eine „Unnamed Pipe“ wird z.B. mit dem Systemaufruf *CreatePipe* erzeugt. Zum Lesen und Schreiben verwendet man die Systemaufrufe *ReadFile* und *WriteFile*.

6.5.4 Rechnerübergreifende Interprozesskommunikation

Bisher haben wir betrachtet, wie Prozesse innerhalb eines Rechnersystems miteinander kommunizieren. Das Betriebssystem unterstützt in diesem Fall das Senden und Empfangen von Nachrichten über interne Datenbereiche bzw. Nachrichten. Die Interprozesskommunikation zwischen Prozessen unterschiedlicher Rechner ist weit komplexer. Aus Sicht eines Anwendungsprogrammierers ist es allerdings auch hier wünschenswert, wenn die Komplexität so wenig wie möglich beachtet werden muss, also weitgehende oder vollständige Transparenz darüber vorherrscht, auf welchem Rechner die kommunizierenden Partnerprozesse ablaufen und welche Basismechanismen sie nutzen.

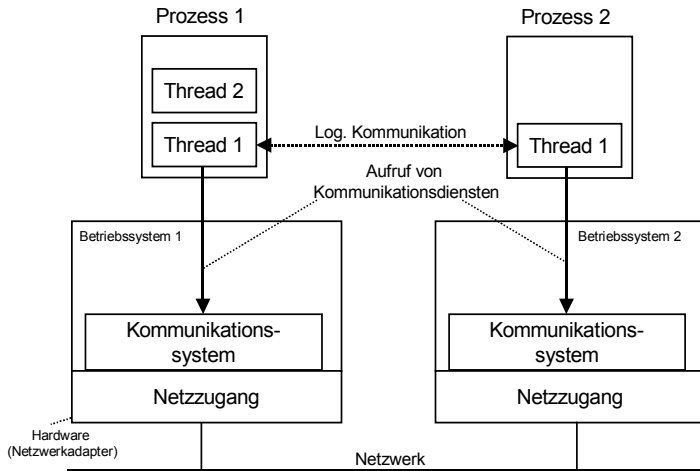


Abbildung 6-27: Rechnerübergreifende Interprozesskommunikation allgemein

Ein grundlegendes Modell zur Beschreibung rechnerübergreifende Kommunikation in Netzen ist das ISO/OSI-Referenzmodell. In heutigen Netzwerken ist auch das TCP/IP-Referenzmodell von großer Bedeutung. Auf Basis dieser Modelle gibt es viele Protokollspezifikationen, die hier nicht weiter vertieft werden sollen (Tanenbaum 2003).

Zwei Prozesse bzw. deren Threads können rechnerübergreifend nur dann miteinander kommunizieren, wenn sie vom jeweiligen Betriebssystem durch entsprechende Kommunikationssysteme unterstützt werden, welche die Dienste für die entfernte Kommunikation bereitstellen. Die Kommunikationssysteme der beteiligten Rechner müssen auch aufeinander abgestimmt sein, also die gleiche Sprache sprechen.

Je komfortabler das Kommunikationssystem ist, umso mehr versteckt es die Details der Kommunikation vor dem Anwendungsprozess und damit vor dem Programmierer. Optimal ist es, wenn schon der Netzwerkzugang vollständig vom Kommunikationssystem, das ein Teil des Betriebssystems ist, gekapselt wird und damit gar nicht für den Prozess sichtbar ist. Dies ist in Abbildung 6-27 skizziert. Threads von Prozessen, die in unterschiedlichen Rechnersystemen liegen, kommunizieren hier miteinander. Die Betriebssysteme der Rechner stellen jeweils Kommunikationssysteme bereit, die den Prozessen Kommunikationsdienste zur Verfügung stellen. Die Nachrichtenübermittlung erfolgt über ein gemeinsames Netzwerk, für das jeder der beiden Rechner einen Netzwerkzugang besitzt. Heutige PCs und PC-Betriebssysteme (Linux, Windows) verfügen beispielsweise schon

standardmäßig über ein TCP/IP-Kommunikationssystem und der Netzwerkzugriff wird über einen Local-Area-Network-Adapter (LAN-Adapter wie z.B. Ethernet-Karten) abgewickelt.¹⁷

6.6 Übungsaufgaben

1. Was bezeichnet man in der Prozessverwaltung als Blockieren, Verklemmen und Verhungern?
2. Bei Betriebssystemen, bei systemnahen Programmen, aber auch bei Anwendungssoftware muss man sich als Entwickler von nebenläufig auszuführenden Aktionen mit kritischen Abschnitten befassen. Was versteht man unter einem kritischen Abschnitt?
3. Welche Maßnahmen sind zu treffen, dass es beim Durchlaufen eines kritischen Abschnitts nicht zu Inkonsistenzen kommt? Gehen Sie dabei auf den Begriff des gegenseitigen Ausschlusses (mutual exclusion) ein!
4. Was sind Semaphore? Gehen Sie dabei kurz auf die Semaphoreoperationen P() und V() ein!
5. Warum müssen die Semaphore-Operationen P() und V() selbst wieder ununterbrechbar sein?
6. Welche Auswirkung hat die folgende, falsche Nutzung eines binären Semaphors?
V(); ... kritischer Abschnitt ...; P();
7. Welche Auswirkung hat die folgende, falsche Nutzung eines binären Semaphors?
P(); ... kritischer Abschnitt ...; P();
8. Welche vier Kriterien sind nach Dijkstra für die Behandlung kritischer Abschnitte zu beachten?
9. Der Ablauf zweier konkurrierender Prozesse wird oben (Abbildung 6-4) im Petrinetz dargestellt. Beantworten Sie hierzu folgende Fragen:
 - a) Über welche Stelle wird der kritische Abschnitt realisiert und warum?
 - b) Welche Transitionen sind Ein- bzw. Ausgänge des kritischen Abschnitts?
 - c) Welche Stelle bewirkt den gegenseitigen Ausschluss?
 - d) Welche Stellen bezeichnen einen unkritischen Abschnitt?
10. Was ist ein Mutex im Sinne der Synchronisation konkurrierender Prozesse?

¹⁷ TCP (Transport Control Protocol) und IP (Internet Protocol) sind die wichtigsten Protokolle im heutigen Internet. Nach diesen Protokollen wurde auch die ganze Protokollfamilie benannt (TCP/IP-Protokolle).

11. Semaphore zur Kontrolle des Eintritts in einen kritischen Abschnitt müssen effizient implementiert werden. Nehmen Sie zu dieser Aussage Stellung und betrachten Sie dabei die Implementierungsvariante mit Polling (Busy Waiting).
12. Ein TSL-Befehl (Test and Set Lock) wird verwendet, um atomare Aktionen zu implementieren.
 - a) Welche Operanden hat ein TSL-Befehl?
 - b) Welche Werte können die Operanden annehmen und was bedeuten sie?
13. Zeigen Sie, wie man den TSL-Befehl für die Implementierung einer Sperre mit den beiden Operationen *lock* und *unlock* verwenden kann. Skizzieren Sie dabei grob die Algorithmen in einem Assembler-ähnlichen Pseudocode!
14. Nennen Sie die vier notwendigen und hinreichenden Bedingungen, die zu einem Deadlock führen!
15. Warum lassen sich Deadlocks bei nebenläufigen Prozessen schwer ganz vermeiden?
16. Welche Alternative zur Behandlung von Deadlocks kennen Sie?
17. Erläutern Sie die Arbeitsweise der Java-Synchronisationsprimitive *synchronized*!
18. Nennen Sie Kritikpunkte der Java-Umsetzung des Monitor-Konzepts von Hoare oder Brinch Hansen. Wo liegen die Schwachstellen und wie kann man sie umgehen?
19. Was sind Race Conditions?
20. Nehmen wir an, zwei Prozesse P1 und P2 greifen auf zwei gemeinsame globale Variablen a und b lesend und verändernd in der folgenden Form zu.

// Globale Variablen

int a = 10;

int b = 0;

Prozess P1:

(1.1) b = 5;

(1.2) a = a + b;

Prozess P2:

(2.1) b = -5;

(2.2) a = a + b;

Zeigen Sie zwei nebenläufige Ablaufsequenzen auf, die jeweils zu einem anderen Wert der Variable a führen! Wie könnte man diese Situation vermeiden?

21. Vergleichen Sie die Funktionsweise des Java-Schlüsselwortes *synchronized* mit der Funktionsweise eines C#-Monitors! Gehen Sie dabei darauf ein, wie die Sperrmechanismen greifen!
22. Was ist der Unterschied zwischen synchroner und asynchroner Kommunikation?
23. Was ist Multicast-Kommunikation im Unterschied zu Broadcast und Anycast?
24. Was versteht man unter einem Halbduplex- im Gegensatz zu einem Vollduplex-Betrieb?
25. Erläutern Sie kurz die Funktionsweise einer Pipe!

7 Hauptspeicherverwaltung

Dieses Kapitel befasst sich mit der Speicherverwaltung und zwar insbesondere mit der Verwaltung des Hauptspeichers. In Multiuser-/Multiprogramming-Betriebssystemen nutzt man heute meist die virtuelle Speichertechnik mit diversen Optimierungsmöglichkeiten im Demand-Paging-Verfahren. Deshalb bildet die virtuelle Speichertechnik den Schwerpunkt des Kapitels, aber auch auf ältere Verfahren wie Overlay-Technik und auf Swapping wird eingegangen.

Zunächst wird ein Überblick über grundlegende Verfahren der Speicherverwaltung gegeben. Anschließend wird das Konzept des virtuellen Speichers vorgestellt. Es wird insbesondere auf die Verwaltung einstufiger und mehrstufiger Seitentabellen eingegangen und wie eine Seitenersetzung durchgeführt wird, wenn kein Platz mehr im Hauptspeicher ist. Weiterhin werden Optimierungsmöglichkeiten im Bereich der Speicherplatznutzung und der Geschwindigkeit bei der Adressumsetzung besprochen. In diesem Zusammenhang wird auf das Konzept des Adressumsetzungspuffers (TLB) und auf das Konzept der invertierten Seitentabellen eingegangen.

Schließlich werden einige Algorithmen zur Seitenersetzung wie FIFO, NFU, NRU, usw. besprochen und miteinander sowie mit der optimalen Strategie von Belady verglichen. Das Kapitel gibt auch eine kurze Einführung in die Adressierung über Segmente sowie in die gemeinsame Nutzung von Speicherbereichen (shared memory). Es endet mit einer Einführung in konkrete Implementierungen der virtuellen Speichertechnik in den Betriebssystemen Unix, Linux und Windows. Konkrete Optimierungsmöglichkeiten und Verwaltungstechniken werden ebenfalls aufgezeigt.

Zielsetzung des Kapitels

Der Studierende soll die Probleme der Speicherverwaltung verstehen und erläutern können. Vor allem sollte er die grundlegenden Konzepte der virtuellen Speichertechnik und ihrer Optimierungsmöglichkeiten erklären können und die verschiedenen Strategien der Seitenersetzung beschreiben und miteinander vergleichen können.

Wichtige Begriffe

Adressraum, Seiten (Pages) und Seitenrahmen (Frames), MMU, Lokalitätsprinzip, Overlay-Technik, Swapping, Demand Paging und Prepaging; Virtueller Speicher,

Seitentabelle (einstufig, zweistufig, mehrstufig), virtuelle Speicheradresse, reale (physikalische) Adresse, Seitenersetzungsverfahren bzw. Verdrängungsverfahren NRU, FIFO, Second Chance, Clock Page, LRU, NFU, Adressumsetzpuffer (TLB), invertierte Seitentabelle, Vergabestrategien, Buddy-Technik, Working Set.

7.1 Grundlegende Betrachtungen

7.1.1 Speicherhierarchien

Im Idealfall wäre der Zugriff auf den Speicher genau so schnell oder sogar schneller als die CPU-Geschwindigkeit. Allerdings sind heute die CPUs meist immer noch schneller als der Speicher. Dies versucht man zu kompensieren, indem man eine Hierarchie von Speichersystemen, wie dies in Abbildung 7-1 dargestellt ist, verwendet.

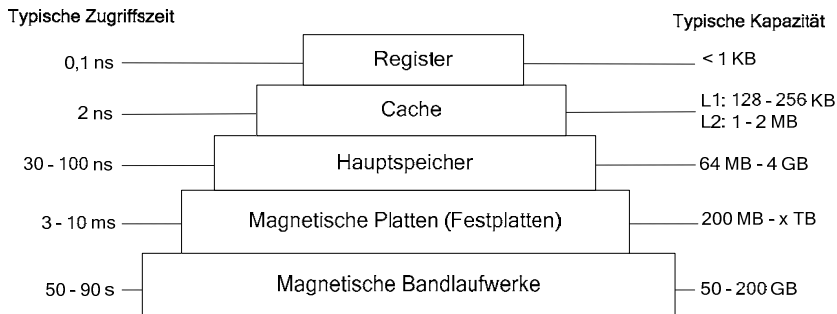


Abbildung 7-1: Typische Speicherhierarchie eines modernen Rechnersystems (Tanenbaum 2002)

Die schnellsten, aber auch teuersten Speicher sind die CPU-Register. Sie sind fast so schnell wie die CPU selbst und verursachen daher kaum Zugriffsverzögerungen. CPU-Register werden durch die Software kontrolliert. Danach kommen Cache-Speicher, die üblicherweise über die Hardware kontrolliert werden. Wie bereits in Kapitel 1 erläutert, ist ein L1-Cache näher an der CPU und verfügt über etwas schnellere Zugriffszeiten als ein L2-Cache. Es gibt verschiedene Organisationsmöglichkeiten für Caches, auf die hier nicht eingegangen werden soll. Weiterhin werden für Caches ähnliche Konzepte für die Verwaltung usw. eingesetzt, wie sie auch für die im Weiteren besprochenen Strategien der Hauptspeicherverwaltung gelten.

Die Hardware versucht, möglichst die Hauptspeicherbereiche im Cache zu halten, die in nächster Zukunft verwendet werden. Cache-Speicher ist aber auch sehr

teuer und daher meist nur in begrenztem Umfang vorhanden. Cache-Speicher enthalten Kopien aus dem gerade benötigten Hauptspeicherinhalt. Die Cache-Steuerung erfolgt heute – zumindest bei den hier gemeinten Hardware-Caches – vollständig mit in Hardware realisierter Logik.

Als nächstes kommt in der Speicherhierarchie der Hauptspeicher (Main Memory, Arbeitsspeicher), der die Hauptlast der Programmausführung trägt. Schließlich sind noch Festplatten und Bandlaufwerke zu nennen, die wesentlich langsamer sind.

7.1.2 Lokalität

Um einen schnellen Zugriff auf die benötigten Adressbereiche zu erreichen, nutzt man vor allem den *Lokalitätseffekt* sehr stark aus. Damit ist gemeint, dass die Wahrscheinlichkeit sehr hoch ist, dass Daten, die gerade benutzt werden, auch in naher Zukunft wieder benötigt werden. Man unterscheidet zeitliche und räumliche Lokalität:

- Unter der *zeitlichen Lokalität* versteht man, dass Adressbereiche, auf die zugegriffen wird, auch in naher Zukunft mit hoher Wahrscheinlichkeit wieder benutzt werden. Der Lokalitätseffekt wird aus Sicht des zeitlichen Ablaufs in Abbildung 7-2 dargestellt (siehe Glatz 2006). Wie man erkennen kann, erfolgt der Zugriff im zeitlichen Verlauf häufig auf die gleiche Speicheradresse. Diesen Effekt kann man in allen Hierarchieebenen ausnutzen, um Speicherbereiche möglichst schnell im Zugriff zu halten. Man kann dies erreichen, indem man versucht, Speicherbereiche auf die erst kürzlich zugegriffen wurde, in einem Cache-Speicher zu verwalten oder z.B. nicht aus dem Hauptspeicher zu entfernen.

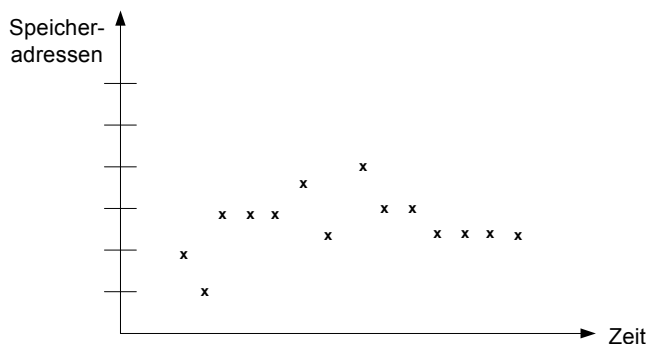


Abbildung 7-2: Zeitliche Lokalität

- Unter der *räumlichen Lokalität* versteht man, dass nach einem Zugriff auf einen Adressbereich der nächste Zugriff auf eine Adresse in der unmittelbaren Nachbarschaft mit hoher Wahrscheinlichkeit erfolgt. Wie man in Abbildung 7-3 sieht, werden im zeitlichen Verlauf immer wieder Speicheradressen angesprochen, die sehr nahe beieinander liegen, was sich im Programmcode etwa durch Schleifen (while, for,...) oder durch mehrmaligen Aufruf derselben Methode zeigt. Dies kann man ausnutzen, indem man bei einem Speicherzugriff auch gleich benachbarte Adressbereiche in die nächste Hierarchiestufe bringt.

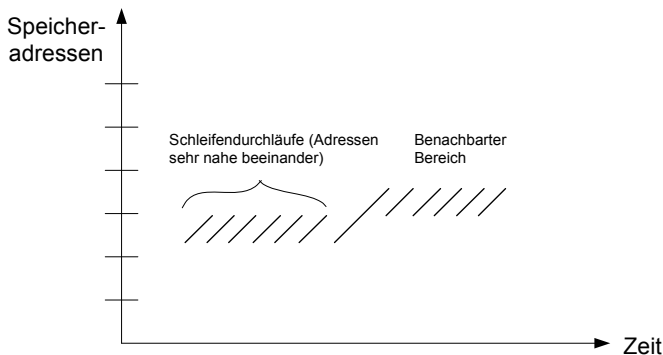


Abbildung 7-3: Räumliche Lokalität

Man spricht in diesem Zusammenhang auch vom *Arbeitsbereich* oder *Working Set* eines Prozesses. Dieser ändert sich im Laufe eines Programms nur sehr langsam, was schon in den 70er Jahren durch P. Denning festgestellt wurde. Der Effekt wird nicht nur für das Caching, sondern auch für die Verwaltung des Hauptspeichers ausgenutzt.

7.1.3 Adressen und Adressräume

Adressen. Wie in Kapitel 1 erwähnt, werden in einem Von-Neumann-Rechner der Programmcode und auch die zu verarbeitenden Daten im Hauptspeicher abgelegt. Ein Programm kann nur ablaufen, wenn es im Hauptspeicher ist. Der Hauptspeicher ist nach Speicherstellen geordnet. Üblicherweise ist ein Byte (8 Bit) die kleinste adressierbare Einheit. Jedes Byte im Speicher wird daher mit einer *Adresse* versehen. Adressen werden meist hexadezimal (z.B. 0x40000000) notiert. Wenn man im Prozessor 32 Adressbits zur Verfügung hat, kann man maximal 2^{32} Bytes, also vier GB, adressieren und der komplette Adressbereich reicht von 0x00000000 bis 0xFFFFFFFF.

Adressraum. Man spricht von einem Adressraum, wenn man die Menge aller möglichen Adressen meint. Bei einem 4-GB-Adressraum sind dies die Adressen $\{0, 1, 2, \dots, 2^{32}-1\}$. Die Adressen eines Adressraums werden also durchnummeriert, wobei 0 die kleinste Adresse darstellt. Dies ist in Abbildung 7-4 für einen 4-GB-Adressraum dargestellt.

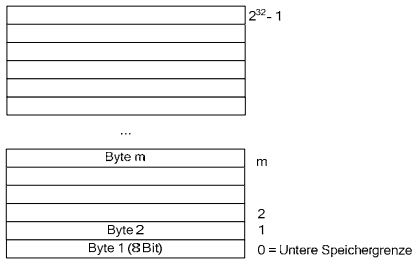


Abbildung 7-4: Nummerierung des Adressraums

Adressraumbelegung. Wie die Daten und der Programmcode in den Adressraum gelegt werden, wird in einem Belegungsplan festgelegt. Man spricht hier auch von einem Adressraumbelegungsplan. Die übergreifende Belegung ist im Betriebssystem festgelegt. Teile für die Anwendungsprogramme und deren Daten werden reserviert. Ein Compiler oder Interpreter entscheidet, wie die Daten der Programme abgelegt werden.

Damit auf die Daten im Hauptspeicher schnell zugegriffen werden kann, wird üblicherweise eine Ausrichtung nach sog. *Maschinenwörtern* durchgeführt. Die Länge eines Wortes richtet sich nach der Anzahl der Adressbits. Bei einer 32-Bit-Adresse wie beim Intel-Pentium-Prozessor ist die Wortlänge z.B. vier Bytes lang. Die Speicherung erfolgt also meist auf Wortgrenze. Ein Maschinenwort beginnt im Speicher auf einer ohne Rest durch vier teilbaren Adresse. Wenn keine Ausrichtung durchgeführt wird, dann ist der Zugriff trotzdem möglich, benötigt aber zwei Zugriffszyklen und ist damit langsamer.

Adressraumbelegung für Anwendungsprogramme. Die Belegung des Adressraumbereichs für Anwendungsprogramme legen der Übersetzer (Compiler, Interpreter) und dessen Laufzeitsystem fest. In der Regel werden verschiedene Speicherbereiche verwaltet (siehe Glatz 2005):

- Programmcode, Konstanten und initialisierte Daten (statische Variablen), die direkt aus einer ausführbaren Datei in den Speicher geladen werden.
- Nicht initialisierte Daten (auch statische Variablen).
- Dynamischer Datenbereich für Daten, die zur Laufzeit erzeugt werden. Der dafür zuständige Datenbereich wird als *Heap* bezeichnet.
- Dynamischer Datenbereiche für den *Stack*.

Stack und Heap wachsen und schrumpfen während des Programmablaufes. Damit es keine Kollisionen gibt, werden sie möglichst weit auseinandergelegt. Sie können aufeinander zuwachsen oder auch an entgegengesetzte Adressraumgrenzen gelegt werden (siehe Abbildung 7-5). In jeder Programmiersprache gibt es eigene Regeln zur Platzierung von Programmcode und Daten.

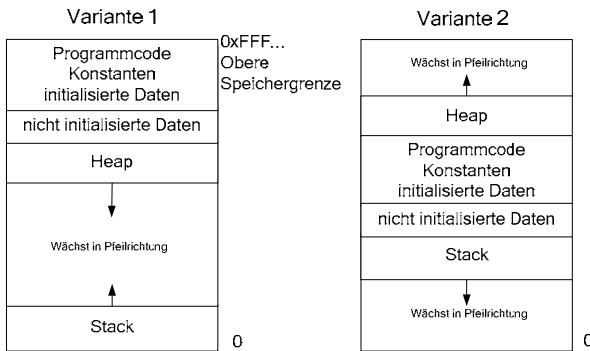


Abbildung 7-5: Adressraumbelegungen durch Übersetzer

7.1.4 Techniken der Speicherverwaltung

Dieses Kapitel befasst sich überwiegend mit dem Hauptspeicher und erläutert dessen Verwaltung durch das Betriebssystem. Betriebssystemprozesse müssen für den Ablauf mit Hauptspeicher (Betriebsmittel) versorgt werden. Die Aufgabe der Speicherverwaltung übernimmt das Betriebssystem und zwar eine spezielle Komponente des Betriebssystems namens Memory-Manager bzw. Speicherverwalter. Er verwaltet freie und belegte Speicherbereiche und ordnet sie bei Bedarf den Prozessen zu.

In den letzten Jahrzehnten wurden die Techniken zur Speicherverwaltung stark weiterentwickelt. Einige etwas ältere Techniken finden heute (Spezialsysteme und Realzeitsysteme ausgenommen) daher keine Anwendung mehr. Das Konzept des virtuellen Speichers, oft auch in Verbindung mit der Swapping-Technik, hat sich weitgehend durchgesetzt und wird in den meisten Mehrzweckbetriebssystemen eingesetzt. Dieses Konzept bildet auch den Schwerpunkt dieses Kapitels. Trotzdem soll auch ein kurzer Blick auf etwas ältere Techniken geworfen werden, um die Fortschritte in der Entwicklung der Speicherverwaltung einschätzen zu können.

Im Laufe der Zeit wurden verschiedene Mechanismen entwickelt und eingesetzt. Hierzu gehören:

- Speicherverwaltungstechniken für das Monoprogramming

- Speicherverwaltungstechniken für das Multiprogramming mit festen Partitionen
- Speicherverwaltungstechniken für das Multiprogramming mit Variablen Partitionen und Swapping
- Speicherverwaltungstechniken für das Multiprogramming mit virtuellem Speicher und Paging

Speicherverwaltung bei Monoprogramming. Im Monoprogramming wird der Speicher nur dem gerade aktiven Programm und dem Betriebssystem zugeteilt. Dies ist relativ einfach, wie dies Abbildung 7-6 für die Betriebssysteme MS-DOS und Palm OS zeigt. Der untere Teil des Speichers wird bei MS-DOS für das Betriebssystem, der obere Bereich (also der mit den höheren Adressen) für Gerätetreiber verwendet und in einem ROM (Read Only Memory) abgelegt. Dieser Bereich wird auch als BIOS (Basic Input Output System) bezeichnet. Dazwischen liegt das gerade aktive Anwendungsprogramm. Bei Palm OS wird z.B. der obere Speicherbereich für das Betriebssystem genutzt.

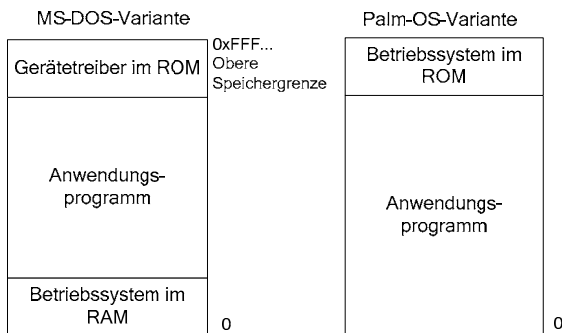


Abbildung 7-6: Speicherverwaltung bei Monoprogramming

Speicherverwaltung mit festen Partitionen. Beim Multiprogramming-Betrieb mit festen Partitionen wird der Speicher in feste Teile, sog. Partitionen zerlegt. Diese Technik wurde z.B. im Betriebssystem IBM OS/360 verwendet. Ankommende Jobs werden in eine Warteschlange eingereiht. Wenn eine Partition mit ausreichendem Speicher verfügbar ist, wird der Job abgearbeitet. In jeder Partition wird genau ein Job ausgeführt (Multiprogramming). Eine Variante dieses Konzepts ist die Zuteilung einer Warteschlange für jede Partition. In einer anderen Variante wird eine globale Queue verwaltet (siehe Abbildung 7-7), von der aus die Zuteilung erfolgt. Die Partitionen werden vom Operator fest konfiguriert und können nicht ohne weiteres verändert werden. Dieses Speichermodell ist relativ einfach zu implementieren, wird heutzutage aber kaum mehr benutzt.

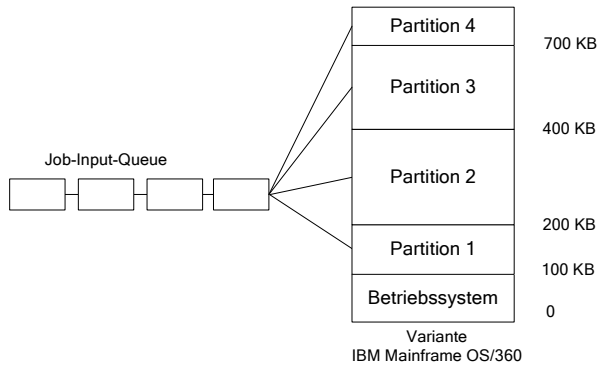


Abbildung 7-7: Speicherverwaltung bei Multiprogramming mit festen Partitionen (Tanenbaum 2002)

Speicherverwaltung mit variablen Partitionen und Swapping. Eine Weiterentwicklung des Partitionsmodells ist Multiprogramming mit variablen Partitionen und Swapping, das für Timesharing-Betriebssysteme entwickelt wurde. Hierbei geht man davon aus, dass in Timesharing-Betriebssystemen nicht immer alle aktiven Prozesse in den Hauptspeicher passen. Wenn ein Prozess aktiv ist, wird er im Ganzen in den Hauptspeicher geladen und nach einer gewissen Zeit unter Berücksichtigung einer passenden Strategie wieder auf einen Sekundärspeicher (in der Regel eine Festplatte) ausgelagert.

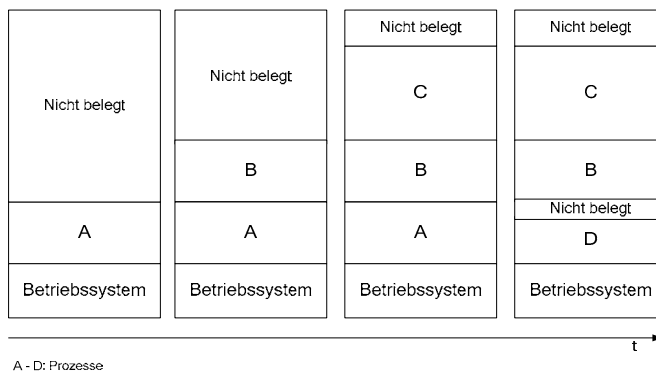


Abbildung 7-8: Speicherverwaltung bei Swapping (Tanenbaum 2002)

In Abbildung 7-8 ist ein Beispiel für einen zeitlichen Verlauf einer Speicherbelegung mit der Swapping-Technik dargestellt. Die Prozesse A, B und C können ihren Speicher problemlos erhalten. Um den Prozess D auszuführen, muss vorher A ausgelagert („geswapped“) werden. Durch das Swapping können ungenutzte Be-

reiche („Löcher“) im Hauptspeicher entstehen. Diese können eliminiert werden (Memory Compaction), was aber relativ aufwändig ist. Der wesentliche Unterschied zu festen Partitionen ist, dass die Anzahl der Prozesse, die Größe des Speicherplatzes und auch der Speicherort variieren können. Der Prozess wird immer dahin geladen, wo gerade Platz ist.

Die heute am meisten genutzte Speicherverwaltungstechnik, der virtuelle Speicher, soll im folgenden Abschnitt erläutert werden.

7.2 Virtueller Speicher

7.2.1 Grundbegriffe und Funktionsweise

Die virtuelle Speichertechnik ist heute für das Multiprogramming die gängige Form der Speicherverwaltung. Hinter diesem Konzept stehen die folgenden grundlegenden Gedanken:

- Ein Prozess sollte auch dann ablaufen können, wenn er nur teilweise im Hauptspeicher ist. Wichtig ist hierbei, dass die Teile des Prozesses (Daten und Code) im physikalischen Speicher sind, die gerade benötigt werden.
- Der Speicherbedarf eines Programms sollte größer als der physikalisch vorhandene Hauptspeicher sein können.
- Ein Programmierer sollte am besten nur einen kontinuierlichen (linearen) Speicherbereich, beginnend bei Adresse 0 sehen und sich nicht um die Zerstückelung (Fragmentierung) des Hauptspeichers auf mehrere Benutzer kümmern müssen.

Das Betriebssystem hält die gerade benutzten Speicherbereiche der Prozesse im Hauptspeicher und den Rest auf einem Sekundärspeicher (einer Festplatte). Dabei geht man davon aus, dass auch große Programme nicht den gesamten Speicher auf einmal benötigen, sondern sich zur Laufzeit oft mit recht hoher *Lokalität* in den gleichen Code- und Datenbereichen aufhalten.

Virtueller Adressraum. Der Speicherbereich, den ein Prozess zur Verfügung hat, wird als *virtueller Speicher*, oft auch als *virtueller Adressraum* bezeichnet, da er nicht physikalisch vorhanden ist, sondern vom Memory-Manager nur virtuell für jeden Prozess bereitgestellt wird. Der tatsächlich vorhandene Speicher wird auch *realer* bzw. *physischer Speicher* oder *realer Adressraum* genannt. Der virtuelle Speicher ist also ein logischer Speicher, den ein Prozess nutzt und der zur Laufzeit auf realen Speicher abgebildet wird.

Die Verwaltung und der optimale Einsatz der virtuellen Speichertechnik erfordert einige Arbeit. Das Betriebssystem bzw. der Memory-Manager muss mehrere Stra-

tegien implementieren. Hierzu gehört die Abrufstrategie (Fetch Policy, Varianten sind Demand Paging oder Prepaging), die Speicherzuteilungsstrategie (Placement Policy), die Austauschstrategie (Replacement Policy = Seitenersetzungs- bzw. Verdrängungsstrategie) und die Aufräumstrategie (Cleaning Policy).

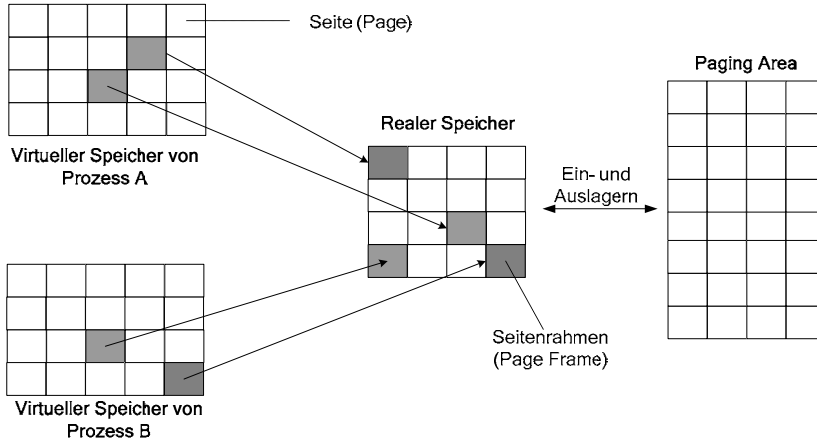


Abbildung 7-9: Grundprinzip der virtuellen Speichertechnik

Die Abrufstrategie regelt den Zeitpunkt des Einlesens von Speicherbereichen in den Hauptspeicher, die Speicherzuteilungsstrategie ermittelt, wohin neu eingele-sene Speicherbereiche im Hauptspeicher platziert werden, die Austauschstrategie kümmert sich um das Entfernen von Speicherbereichen, wenn eine Verdrängung notwendig ist, und die Aufräumstrategie kümmert sich darum, dass möglichst immer etwas Platz im Hauptspeicher ist. Auf die einzelnen Strategien und deren Varianten wird im Weiteren noch eingegangen.

Paging. Der Vorgang des Transports der Daten vom Hauptspeicher auf die Festplatte und umgekehrt wird als *Paging* bzw. Ein- und Auslagern bezeichnet. Genauer gesagt, spricht man in der heute üblichen Nutzung hier auch von *Demand Paging*, als Form der Abrufstrategie, da hier eine Einlagerung nur auf Anforderung, also wenn die Daten benötigt werden, durchgeführt wird. Demand Paging holt also nur die Seiten in den Hauptspeicher, die auch benötigt werden. Kennt man die Lokalität der Programme (siehe Lokalitätseffekt) bzw. kann diese gut einschätzen, kann man auch Prepaging als alternative Abrufstrategie nutzen. Im Gegensatz zu Demand Paging werden hier Seiten in den Hauptspeicher geholt, die noch nicht angefordert wurden. Auch Kombinationen von Demand Paging und Prepaging sind sinnvoll und praxisrelevant. Man kann zum Beispiel beim Holen einer ange-

forderten Seite gleich die benachbarten Seiten oder sonstige Seiten nach einem bestimmten Algorithmus in den Hauptspeicher laden.

Paging-Area. Der Teil der Festplatte, der für die ausgelagerten Seiten verwendet wird, heißt *Paging-Area* und wird auch als *Schattenspeicher* bezeichnet. Die Strategien, die zum Ein- und Auslagern benutzt werden, sind vielfältig, wobei grundsätzlich immer versucht wird, die Seiten im Hauptspeicher zu halten, die auch in naher Zukunft benötigt werden, um ein Paging möglichst selten auszuführen. Es sollte noch erwähnt werden, dass Paging für die Daten der Anwendungsprozesse sinnvoll ist, die umfangreichen und zum Teil auch sehr kleinen Datenstrukturen des Kernels sollten aber – wenn möglich – nicht oder nicht immer dem Paging unterworfen werden, sondern eher im Hauptspeicher verbleiben, um das System effizient zu halten. Welche Speicherbereiche ausgelagert werden dürfen, regelt das Betriebssystem.

Seiten und Seitenrahmen. Wie in Abbildung 7-9 deutlich wird, werden die Speicherbereiche des virtuellen und auch des realen Adressraums in Blöcke aufgeteilt, deren Größe von der Hardware abhängt. Die Blöcke des virtuellen Adressraums werden als *Seiten* (*Pages*) und die des realen Adressraums als *Seitenrahmen* (*Page Frames*, *Frames*, auch: Kacheln, oder kurz Rahmen genannt) bezeichnet. Eine ausgewogene Größe der Seiten und Seitenrahmen ist wichtig, da bei zu kleinen Seitenrahmen mehr Ein-/Ausgabeoperationen erforderlich sind und bei zu großen Seiten Speicherplatz verschwendet wird. In vielen 32-Bit-Prozessoren wurde eine Größe von 4 KB gewählt, aber auch Größen von 1, 8, 16 oder 64 KB sind vorzufinden.

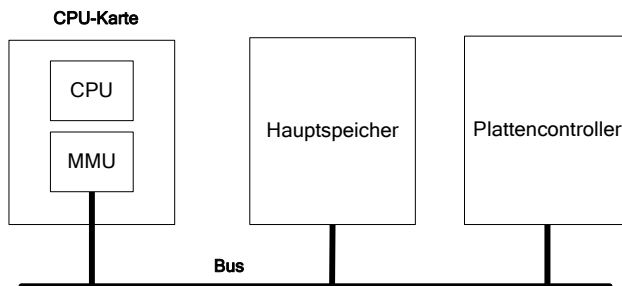


Abbildung 7-10: CPU und MMU im Zusammenspiel

In der virtuellen Speichertechnik muss der Memory-Manager also Strategien zur Verwaltung von virtuellem und realem Adressraum implementieren. Da Seiten ausgelagert werden müssen, um für andere wieder Platz zu machen, sind auch sog. Austausch- bzw. *Seitenersetzungs-* oder *Verdrängungsstrategien* notwendig. Zur Suche und Zuordnung geeigneter Frames im Hauptspeicher verwendet man verschiedene *Speicherzuteilungsstrategien*.

MMU. Die virtuelle Adressierung wird heute von den gängigen Prozessoren der Intel-Familie, der SPARC-Familie, usw. hardwaretechnisch unterstützt. Für diese Hardwareunterstützung ist die *Memory Management Unit* (MMU) zuständig. Die MMU ist heute meistens eine Funktionseinheit des Prozessors und dient dazu, die virtuellen Adressen auf die physikalischen Adressen abzubilden.

Die CPU sendet der MMU die (logische) virtuelle Adresse, die MMU erzeugt daraus nach einem vorgegebenen Algorithmus die physikalische Adresse im Hauptspeicher und sendet diese über den Adressbus an den Hauptspeicher weiter (vgl. hierzu Abbildung 7-10). Im Weiteren soll nun die Funktionsweise eines virtuellen Speichers näher erläutert werden.

Der Memory-Manager stellt im Multiprogramming-Betrieb jedem Prozess einen eigenen virtuellen Adressraum zur Verfügung. Weiterhin verwaltet der Memory-Manager die Paging-Area, deren Größe sich von der Anzahl der Prozesse und von gewissen Erfahrungswerten ableitet und ebenfalls im System konfiguriert wird.

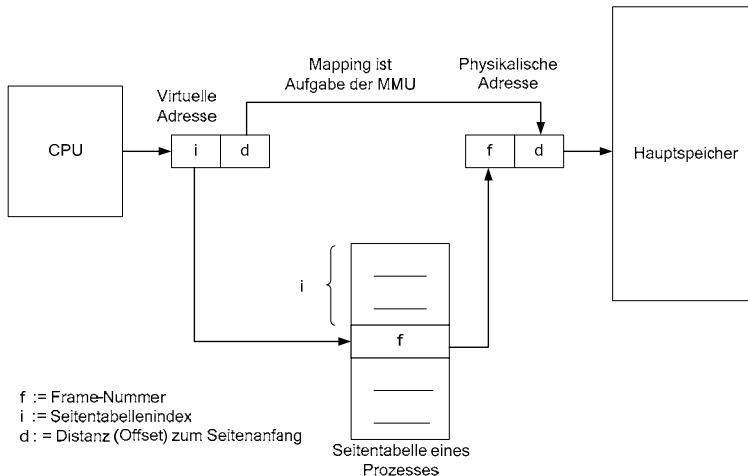


Abbildung 7-11: Einfache Adressumsetzung bei virtueller Adressierung

Seitentabellen. Die virtuellen Adressräume werden im Memory-Manager als sog. *Seitentabellen* (Page Tables) verwaltet, und zwar je eine Seitentabelle je Prozess. Dies sind einfache Tabellen mit Information darüber, wo die Frames tatsächlich im Hauptspeicher zu finden sind. Eine virtuelle Adresse wird in einen Seitentabellenindex und in eine Distanz zerlegt. Die Distanz gibt die genaue Byteadresse innerhalb einer Seite an. Mit Hilfe des Seitentabellenindex wird in der zugehörigen Seitentabelle der Eintrag ermittelt, in dem der Verweis auf den adressierten Frame (Frame-Nummer) steht. Die Frame-Nummer wird zur physikalischen Adresse

ergänzt. Gemeinsam mit der Distanz erhält man die physikalische Adresse (siehe Abbildung 7-11).

Für den Programmierer (bzw. bei höheren Programmiersprachen den Compiler, der die Maschinenbefehle einschließlich Adressierung der Operanden erzeugt) ist der virtuelle Adressraum sichtbar, so als ob er der reale wäre. Die Umsetzung auf die realen Adressen übernimmt der Memory-Manager meist mit Hilfe der MMU oder in Eigenregie, sofern keine MMU vorhanden ist.

In Abbildung 7-12 ist ein virtueller Adressraum von 64 KB dargestellt, wobei die Seitengröße 4 KB ist. Als realer Speicher sind nur 32 KB mit Frames der Größe 4 KB vorhanden. Generell ist es zwingend, die Seitengröße mit der Seitenrahmengröße gleichzusetzen. Die Abbildung zeigt, dass im virtuellen Adressraum einige Seiten belegt sind. Die Pfeile deuten an, in welchen Frames die Seiten im realen Speicher eingelagert sind. Wichtig ist auch, dass die im virtuellen Adressraum belegten Seiten auf beliebige, nicht zwangsläufig in der gleichen Reihenfolge vorliegende Frames abgebildet werden.

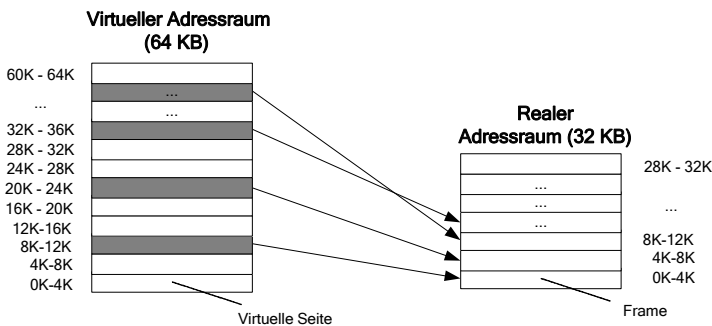


Abbildung 7-12: Grundprinzip der Seitentabelle (Tanenbaum 2002)

Es sei nochmals darauf hingewiesen, dass in den Seitentabellen, die der Memory-Manager verwaltet, nur Verweise auf die Frames im Hauptspeicher sind und (da virtuell) keine Daten verwaltet werden. Die Daten und der Programmcode liegen im realen Speicher. Während der Prozess läuft, müssen alle benötigten Daten und der gerade ausgeführte Code im realen Speicher sein. Die Seitentabellen liegen üblicherweise in einem Speicherbereich, den der Kernel verwaltet.

Adressumsetzung. Für den tatsächlichen Zugriff auf den Hauptspeicher muss der Memory-Manager für jede virtuelle Adresse, die vom Programm aus angesprochen wird, eine Adressumsetzung über folgende Funktion durchführen:

$f(\text{virtuelle Adresse}) \rightarrow \text{reale Adresse}$

Die virtuelle Adresse, die umgesetzt werden soll, ist Ausgangspunkt der Berechnung. Die Umsetzung ist in Abbildung 7-13 nochmals grafisch dargestellt. Über eine Seitentabelle wird letztendlich das Mapping der Page-Nummer (p) auf die Frame-Adresse durchgeführt (f). Die Distanz d innerhalb der Seite bzw. des Frames wird bei der Abbildung unverändert aus der virtuellen Adresse in die physikalische Adresse übernommen. Die Größe der Page entspricht damit der Größe des Frames (z.B. 4096 Byte).

Anhand des Assemblerbefehls `MOVE R1, 8196` soll die Adressumsetzung erläutert werden: R1 ist hier ein CPU-Register, in das der Wert geladen wird, der auf Adresse 8196 des virtuellen Adressraums liegt. In dem Beispiel liegt die angesprochene Adresse bei einer Seitengröße von 4 KB in der dritten Seite des virtuellen Adressraums (0 ist die erste). Die Seite beginnt bei der Adresse 8192 (dezimal). Diese Seite befindet sich im realen Speicher im ersten Rahmen (mit 0 beginnend), und daher bildet die MMU diese Adresse auf die reale Adresse 4 ab.

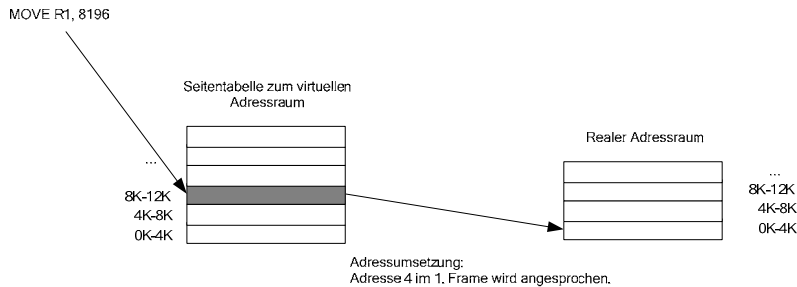


Abbildung 7-13: Beispiel zur Adressumsetzung

Page Fault. Damit ein Befehl ausgeführt werden kann, müssen die Seiten, in welcher der Befehl liegt, und alle Seiten, in denen die verwendeten Daten liegen, auch tatsächlich im realen Speicher sein. Ist dies zum Zeitpunkt der Befehlsausführung nicht so, müssen alle erforderlichen Seiten zunächst in den Hauptspeicher eingelefen werden. Um dies zu bewerkstelligen, wird von der MMU ein besonderer Interrupt, der sog. *page fault* oder *Seitenfehler* erzeugt, und damit wird der aktuell laufende Prozess zunächst einmal unterbrochen. Ein *page fault* (Seitenfehler) ist also ein Trap, den die MMU erzeugt, wenn von einem Prozess eine virtuelle Adresse angesprochen wird, die nicht im Hauptspeicher geladen ist. Das Betriebssystem springt in den Kernelmodus und zwar auf eine spezielle Interruptroutine zur Bearbeitung des Seitenfehlers und versucht unter Beachtung der Seitenersatzstrategie und der Vergabestrategie die Seite in einen Frame zu laden. Anschließend erhält der Prozess je nach Zuteilungsstrategie wieder den Prozessor.

Einstufige Seitentabellen

Man unterscheidet je nach Verwaltungsstrategie unterschiedliche Typen von Seitenumsetzungstabellen bzw. kurz Seitentabellen. Bei einer einstufigen Seitentabelle besteht die virtuelle Adresse aus zwei Teilen:

- Eine Seitennummer i als Index auf die Seitentabelle
- Eine Distanz (Offset) d als Verweis auf die exakte Adresse innerhalb des Frames

Über den Index wird der Anfang des Eintrags in der Seitentabelle adressiert. Falls dieser Seite aktuell ein Frame zugeordnet ist, steht im adressierten Seitentabelleneintrag die Adresse des Frames, in dem die eigentlichen Informationen liegen. In jedem Seitentabelleneintrag gibt ein Present/Absent-Kennzeichen (Flag) an, ob die Seite im Hauptspeicher liegt oder nicht.

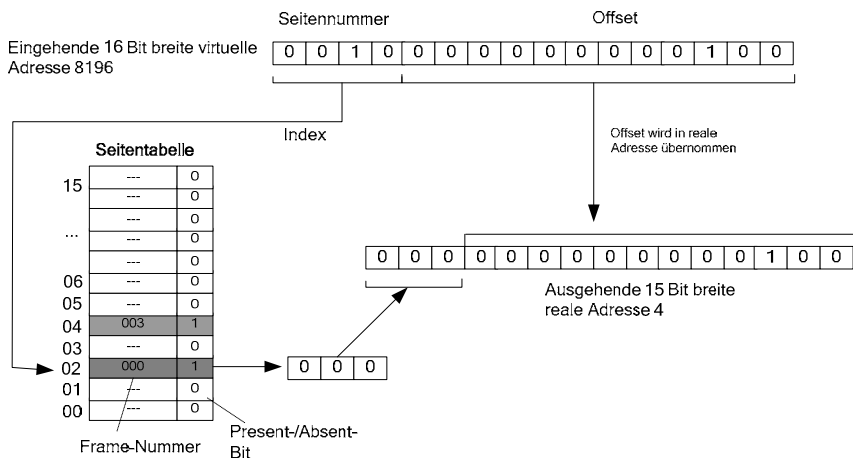


Abbildung 7-14: Beispiel für Adressumsetzung mit einstufiger Seitentabelle (Tanenbaum 2002)

Abbildung 7-14 zeigt nochmals die Adressumsetzung der virtuellen 16 Bit langen Adresse 8196 mit einer einstufigen Seitentabelle, die über 16 Einträge verfügt. 16 Einträge ergeben einen virtuellen Adressraum von 64 KB ($16 \text{ KB} \cdot 4 = 64 \text{ KB}$). Als realer Adressraum sind im Beispiel über 15-Bit-Adressen genau 32 KB adressierbar. Die Adresse wird bei der Adressumsetzung in die Seitennummer 0b0010 und das Offset 0b0000000000100 zerlegt. Mit der Seitennummer wird die Seitentabelle indiziert. Im entsprechenden Eintrag steht der Verweis 0b000 auf den Frame. Frame-Adresse und Offset werden zur 15 Bit langen realen Adresse zusammengefügt ($2^{15} = 32 \text{ KB}$).

Mit 4 Bit, die für die Seitennummer zur Verfügung stehen, kann man also 16 Einträge adressieren. Bei 10 Bit sind es schon 1024 Einträge (2^{10}) und damit ergibt sich bei einer Seitengröße von 4 KB ein virtueller Adressraum von $1024 * 4 \text{ KB} = 4 \text{ MB}$.

Für große Adressräume sind auch große Seitentabellen zu verwalten. Bei einer 32-Bit-Adresse und einer Seitengröße von 4 KB sind es immerhin schon eine Million Einträge (2^{20}) in der Seitentabelle, und dies für jeden einzelnen Prozess, der gerade im System ist. Da die Adressumsetzung sehr häufig ausgeführt wird, muss sie sehr schnell sein. Hinzu kommt, dass die Seitentabellen für die meisten Prozesse viel zu groß sind, man verschenkt also durch die Verwaltung großer Seitentabellen ggf. viel Speicher.

Mehrstufige Seitentabellen

Zu Optimierungszwecken setzt man heute meist mehrstufige Seitentabellen ein. Die IA32-Architektur unterstützt beispielsweise eine zweistufige Seitentabelle mit 32 Bit breiten virtuellen Adressen, die je 10 Bit für die zwei Seitentabellenindizes und ein Offset von 12 Bit enthalten. In der Prozessorarchitektur des *AMD Opteron* bzw. *Athlon* (64-Bit-Prozessor, x64-Architektur) wird sogar eine vierstufige Seitentabelle unterstützt. Die virtuellen Adressen sind 48 Bit breit, die vier Seitentabellenindizes zu je 9 Bit und ein 12 Bit langes Offset enthält. Wie ein Betriebssystem die Hardwaregegebenheiten nutzt, ist den Systementwicklern überlassen. Linux verwaltet standardmäßig z.B. eine dreistufige Seitentabelle. Sowohl bei der IA32- als auch bei der x64-Architektur ist also ein Mapping der Hardwaremechanismen auf die Betriebssystemmechanismen erforderlich.

Der allgemeine mehrstufige Mechanismus soll nun am Beispiel der zweistufigen Seitentabelle näher betrachtet werden (siehe Abbildung 7-15).

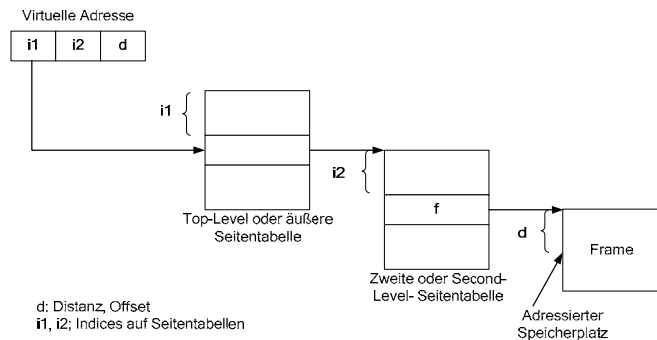


Abbildung 7-15: Zweistufige Seitentabelle

Die virtuelle Adresse setzt sich aus drei Bestandteilen, im Bild mit i_1 , i_2 und d bezeichnet, zusammen. i_1 ist ein Index und verweist innerhalb der sog. äußeren Seitentabelle (Top-Level- oder First-Level-Seitentabelle) auf einen Eintrag, der wiederum auf den Anfang der zweiten Seitentabelle verweist. i_2 ist ein weiterer Index, der innerhalb der zweiten Seitentabelle auf den adressierten Eintrag verweist. In diesem Eintrag steht dann letztendlich der Verweis f auf den konkreten Seitenrahmen. Die Distanz d wird wie bei der einstufigen Seitentabelle genutzt.

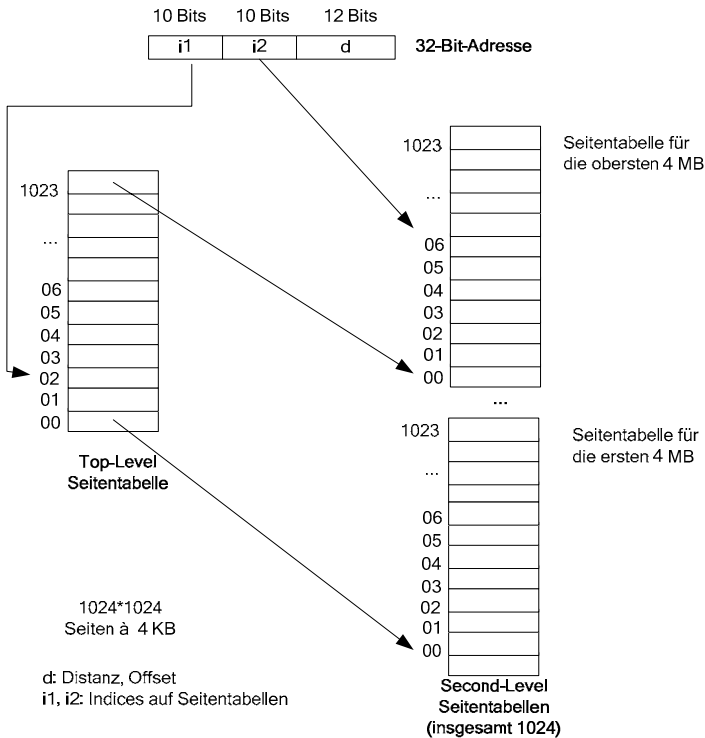


Abbildung 7-16: Zweistufige Seitentabelle am Beispiel

In Abbildung 7-16 ist z.B. eine konkrete zweistufige Seitentabelle dargestellt, wobei 32-Bit-Adressen unterstützt werden. Der 10-Bit-Index i_1 verweist auf die Top-Level-Seitentabelle, der 10-Bit-Index i_2 auf eine der Second-Level-Seitentabellen und 12 Bit sind für das Offset vorgesehen, was einer Seiten- und Framegröße von 4 KB (2^{12}) entspricht. Der gesamte virtuelle Adressraum ist 2^{32} Byte, also 4 GB groß. Wie die Abbildung zeigt, sind vom Memory-Manager je Prozess maximal 1024 Second-Level-Seitentabellen zu verwalten, allerdings sind nur die angelegt, die auch tatsächlich benötigt werden. Bei voller Belegung des Adressraums müssen

natürlich genau so viele Second-Level-Seitentableneinträge verwaltet werden wie bei einer einstufigen Seitentabelle, also eine Million Einträge à 4 KB. Hinzu kommt der Speicherbedarf für die Top-Level-Tabelle.

Zu der speicherintensiven Verwaltung der Seitentabellen kommt noch der Aufwand hinzu, dass eine nicht im Hauptspeicher eingelagerte Seite bei einem Seitenfehler nachgeladen und der Hauptspeicher möglichst immer von unnötigen Ballastseiten befreit werden muss. Trotz des immensen Aufwands ist die virtuelle Speichertechnik mit Demand Paging heute das am meisten verwendete Verfahren zur Speicherverwaltung.

Es sei hier erwähnt, dass die Speicherbereiche für die Seitentabellen je nach Betriebssystem meist auch dem Paging unterliegen und daher bei Bedarf ausgelagert werden. Ein Prozess kann allerdings nur aktiv werden, wenn er Zugriff auf die Seitentabelle im Hauptspeicher hat.

Abbildung 7-17 zeigt einen möglichen Eintrag einer Seitentabelle mit typischen Informationen. Das C-Bit gibt an, ob die Seite in einem Betriebssystempuffer (Cache) abgelegt werden darf oder nicht. Je nach Betriebssystem kann der Zugriffsschutz (Protection) auf einzelne Seiten angewendet werden oder nicht. Die Bedeutung der weiteren Felder oder Kennzeichen ergibt sich aus der Abbildung. Die Informationen zu den Seitentableneinträgen sind im Betriebssystemkernel gespeichert.

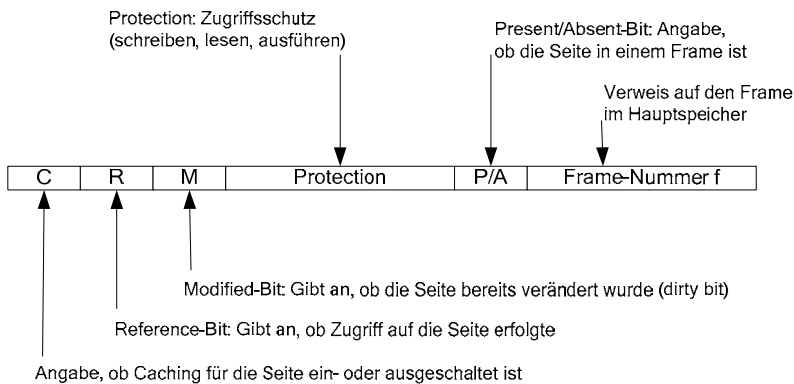


Abbildung 7-17: Typischer Seitentableneintrag

7.2.2 Optimierung der Speicherverwaltung

Die Umsetzung einer virtuellen Adresse auf eine reale Adresse ist im Vergleich zur klassischen Adressierung mit zusätzlichen Hauptspeicherzugriffen auf die Seiten-

tabellen verbunden und verschlechtert damit die Leistungsfähigkeit. Anstatt eines Hauptspeicherzugriffs auf die adressierten Daten sind bei zweistufiger Adressierung drei Hauptspeicherzugriffe erforderlich, da neben den adressierten Daten auch die Top-Level- und die Second-Level-Seitentabelleneinträge aufgesucht werden müssen. Hinzu kommt, dass mit zunehmender Adressraumgröße auch der Speicherplatz für die Verwaltung der Seitentabellen enorm wächst.

Bei 64-Bit-Prozessoren können mehrstufige Seitentabellen zwar grundsätzlich angewendet werden, aufgrund der adressierbaren Speichergröße reichen zwei oder auch drei Stufen aber nicht aus. Man muss dann bis zu sechs Stufen verwenden, was den Aufwand beim Speicherzugriff nochmals deutlich erhöht.

Optimierungskonzepte sind also notwendig. Zur Optimierung der Speicherzugriffe hat man daher weitere Mechanismen entwickelt, die direkt von der Hardware (in der MMU) unterstützt werden. Im Weiteren soll auf *Adressumsetzpuffer* (TLB) und *invertierte Seitentabellen* eingegangen werden. Eine andere Möglichkeit ist es, größere Seiten (sog. large Pages) zu verwenden, was insbesondere bei 64-Bit-Prozessoren sinnvoll ist und auch in Betriebssystemen wie Windows unterstützt wird.

Adressumsetzpuffer (TLB)

Ein *Adressumsetzpuffer* (*Translation Lookaside Buffer*) ist ein eigener, schneller Speicher (Cache), der eine Tabelle darstellt, in der eine Zuordnung von virtueller auf reale Adressen verwaltet wird. Ein TLB hat je nach Implementierung schnellere Zugriffszeiten (z.B. um den Faktor 5) als der Hauptspeicher (vgl. hierzu Herrmann 2002).

Es werden die aktuell am häufigsten benutzten Seitennummern in die Tabelle eingetragen. Bei jeder Adressumsetzung wird zunächst geprüft, ob die aktuelle virtuelle Adresse im Adressumsetzpuffer liegt. Wenn ja (TLB hit), muss kein Zugriff auf die Seitentabellen erfolgen, da die reale Adresse des gesuchten Frames schon ermittelt worden ist und man sich somit weitere Hauptspeicherzugriffe erspart. Im anderen Fall (TLB miss) muss wie bisher über die klassische Adressumsetzung über die Seitentabelle(n) ein Mapping auf die physikalische Adresse vorgenommen werden (Abbildung 5-16).

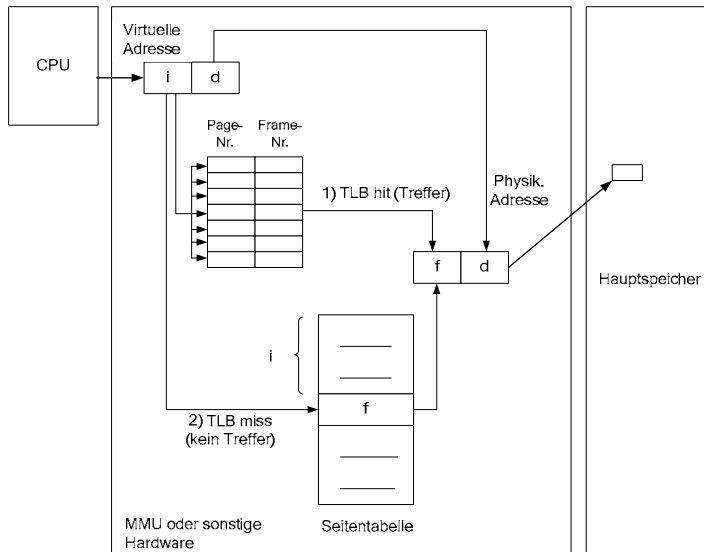


Abbildung 7-18: Arbeitsweise eines TLB

Die Anzahl der Einträge in einem Adressumsetzpuffer ist abhängig von der Rechnerarchitektur und liegt häufig zwischen 8 und 128. IA64-Prozessoren verfügen beispielsweise über 128 Einträge im TLB. Interessant ist, dass aufgrund der hohen Lokalität vieler Programme bereits bei acht Einträgen eine beträchtliche Leistungs-optimierung möglich ist. Bei genügend großen Adressumsetzpuffern kann man sogar auf die Hardwareunterstützung bei der Adressumrechnung verzichten.

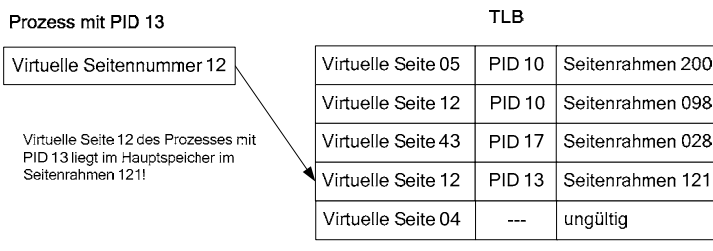


Abbildung 7-19: Aufbau eines TLB

Der Inhalt eines TLB ist einfach und kann der Abbildung 7-19 entnommen werden. Ein TLB-Eintrag enthält neben der virtuellen Adresse und dem Verweis auf den Seitenrahmen im Hauptspeicher auch die Prozess-Identifikation (PID), da die virtuelle Adresse allein im Betriebssystem nicht ausreicht, um eine Seite einem konk-

reten Prozess zuzuordnen. Jeder Prozess-Adressraum verfügt schließlich über die gleichen Seitennummern.

Invertierte Seitentabellen

Bei 64-Bit-Prozessoren wie etwa dem Intel-Itanium-Prozessor, der über einen virtuellen Speicher von 16 Exabyte (2^{64} Adressen) verfügt, steigt aufgrund der breiten Adressen die Größe der Seitentabellen unverhältnismäßig an. Die benötigten Datenmengen nur für die Ablage der Seitentabellen eines Prozesses im Hauptspeicher wären nicht mehr vertretbar.

Um Speicher zu sparen, verwendet man sog. *invertierte Seitentabellen*. Man nutzt hier die Tatsache, dass der physikalische Speicher viel kleiner ist als der virtuelle. Man geht so vor, dass man eine Tabelle anlegt, mit der man die physikalischen auf die virtuellen Adressen abbildet.

Der Vorteil der Vorgehensweise ist, dass wesentlich weniger Tabelleneinträge benötigt werden. Man benötigt nur noch so viele Seitentableneinträge wie man Frames im Hauptspeicher zur Verfügung hat. Nachteilig ist, dass es keine Ordnung nach virtuellen Adressen mehr gibt und die Suche daher etwas aufwändiger ist. Es kann nicht mehr über Seitentabellenindices positioniert werden.

Man unterscheidet verschiedene Varianten von invertierten Seitentabellen, die sich vorwiegend in der Suche unterscheiden. Neben einfachen invertierten Seitentabellen, die sequentiell durchsucht werden, verwendet man auch invertierte Seitentabellen kombiniert mit Hashverfahren. Bei Seitentabellen kombiniert mit Hashverfahren (*hashed page table*) wird für die Suche nach einem Seitenrahmen aus der virtuellen Adresse ein Hashwert gebildet, der als Index auf eine Hashtabelle dient, in der zu den virtuellen Adressen die Adressen der Seitenrahmen gespeichert sind. Invertierte Seitentabellen kombiniert mit Hashverfahren werden z.B. im Power PC, in der IBM AS/400 und im Intel-Itanium-Prozessor genutzt. Die Suche kann über Hashverfahren wesentlich beschleunigt werden.

In die Einträge der invertierten Seitentabellen speichert man die Prozess-Identifikation (PID), die Seitennummer der virtuellen Seite und die Frame-Nummer, die im Hauptspeicher für die Seite verwendet wird. Für die Hashwertbildung wird dann neben der virtuellen Adresse auch die PID des gerade betroffenen Prozesses verwendet. Die PID ist bei der invertierten Seitentabelle wichtig, da die Seitennummer für sich nicht eindeutig ist, es sei denn die invertierte Seitentabelle wird beim Prozesswechsel immer nur für den aktiven Prozess gefüllt.

Die Zusammenhänge sind in Abbildung 7-20 dargestellt. Wie die Abbildung zeigt, wird aus der PID des Prozesses, für den eine Adressumsetzung erfolgen soll, und

aus einen Teil der virtuellen Adresse ein verkürzter Hashwert (im Beispiel 12 Bit) gebildet und damit in der invertierten Seitentabelle nach einem Eintrag gesucht. Wird ein Eintrag gefunden, so kann man aus diesem die Nummer des gesuchten Seitenrahmens direkt auslesen.

Da durch das Hashverfahren eine Doppel- oder Mehrfachbelegung eines Eintrags der Seitentabelle (Kollisionen können auftreten, siehe Anhang), möglich ist, wird zu einem Seitentabelleneintrag bei Bedarf eine verkettete Kollisionsliste ergänzt. Um bei einer Kollision schnell zu ermitteln, welcher Eintrag der verketteten Liste der richtige ist, werden die Einträge der Liste sequentiell abgearbeitet und dabei mit der gesuchten Adresse verglichen. Statistische Untersuchungen bei Prozessoren wie IBM RS/6000 haben gezeigt, dass im Durchschnitt zwei Zugriffe auf die invertierte Seitentabelle notwendig sind, um den richtigen Eintrag zu finden.

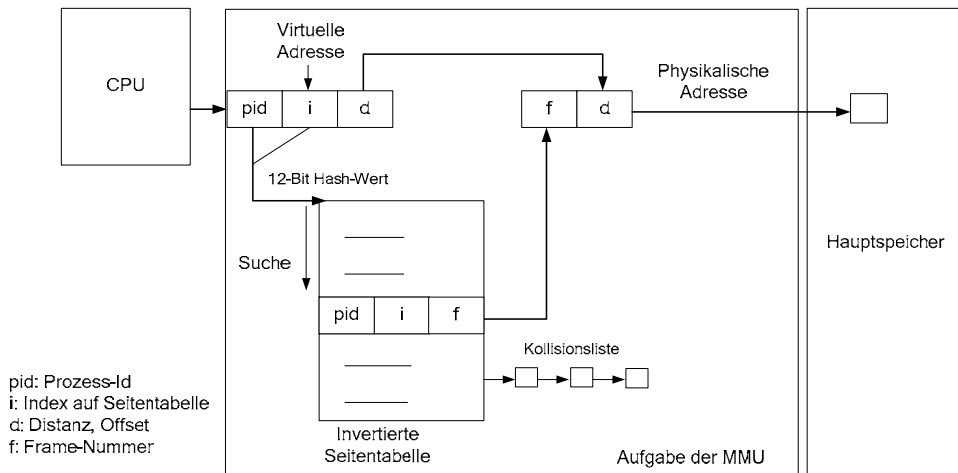


Abbildung 7-20: Arbeitsweise der invertierten Seitentabelle

Der Zugriff auf die Seitentabelle mit Hilfe eines Hashverfahrens (Hashing der virtuellen Adresse) kann durch die Hardware unterstützt werden. Prozessorarchitekturen wie IBM RS/6000, HP PRECISION, IBM AS/400, Intel Itanium usw. bieten dies beispielsweise an.

Die invertierte Seitentabelle wird in der Regel komplett im Hauptspeicher gehalten und muss nicht, wie bei großen „herkömmlichen“ Seitentabellen, ausgelagert werden.

Zur weiteren Verbesserung der Leistungsfähigkeit verwendet man zusätzlich die oben beschriebenen Adressumsetzpuffer, in denen die Seiten- mit den zugehörigen Frame-Adressen aufbewahrt werden, die sehr häufig benutzt werden. Eine Suche

in der invertierten Seitentabelle muss nur dann erfolgen, wenn die adressierte Seite nicht im TLB ist.

7.2.3 Seitenersetzung und Verdrängung (Replacement)

Bei einem Seitenfehler wählt der Memory-Manager, sofern nicht noch Seitenrahmen frei sind, eine Seite aus, die verdrängt, also in die Paging-Area ausgelagert werden muss, um Platz für die neue Seite zu schaffen. Die Strategie, die verwendet wird, um eine zu verdrängende Seite auszuwählen, wird als *Seitenersetzungsstrategie* (Replacement-Strategie) bezeichnet.

Optimal wäre, wenn alle zukünftigen Seitenzugriffe vorher bestimmt werden könnten. Dies ist aber nicht zu realisieren. Bedarfsgerechte Strategien, auch als *Demand-Paging* bezeichnet, reagieren erst, wenn eine Anforderung zur Seiteneinlagerung da ist und kein freier Seitenrahmen mehr verfügbar ist.

Man unterscheidet prozesslokale und globale Strategien zur Seitenersetzung. Bei ersteren wird die Seitenersetzung für jeden Prozess isoliert durchgeführt, bei letzteren für das ganze System. Globale Strategien bieten mehr Möglichkeiten der Optimierung.

Wir werden im Folgenden einige Algorithmen betrachten, die für die Replacement-Strategie in Frage kommen und beginnen mit der optimalen, aber rein theoretischen Variante.

Der optimale Algorithmus. Wie bereits angedeutet, wäre es am besten, wenn die Seitenersetzung immer die zukünftigen Seitenzugriffe aller Prozesse berücksichtigen könnte, weil dann wenige Seitenfehler auftreten würden. Ein optimaler Algorithmus würde die Seitenrahmen für eine Ersetzung auswählen, die am spätesten von allen belegten Seitenrahmen wieder benötigt würden. In Abbildung 7-21 wäre das bei der vierten Referenz auf die Seite mit der Nummer 2 der Seitenrahmen, in dem sich die Seite 7 befindet (siehe Beispiel aus Silberschatz 1994). Diese optimale Strategie fand Belady bereits im Jahre 1966.

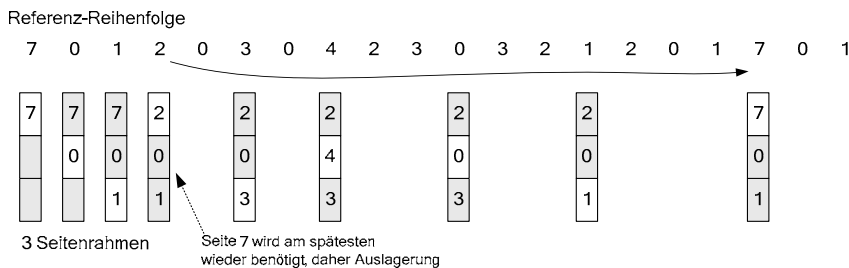


Abbildung 7-21: Optimale Seitenersetzung nach Belady

Der Grund für die schwierige bzw. unmögliche Realisierung ist, dass es nicht praktikabel ist, das Verhalten eines Prozesses hinsichtlich seiner Speicherzugriffe vorher sicher zu bestimmen. Das Betriebssystem kann zum Zeitpunkt des Seitenfehlers nicht wissen, wann welche Seite als nächstes verwendet wird. Das Verfahren würde nur für deterministische Programme, bei denen alle Seitenanforderungen schon von vornherein bekannt sind, funktionieren. Ein ähnliches Problem hatten wir bereits beim Prozess-Scheduling mit den Shortest-Job/Process-First-Verfahren. Auch hier kann das Betriebssystem zum Zeitpunkt der Scheduling-Entscheidung nicht wissen, wie lange ein Prozess noch dauert. Der Algorithmus von Belady wird heute gerne als Referenz-Algorithmus benutzt

Im Laufe der letzten Jahrzehnte wurden verschiedene Seitenersetzungs- oder Replacement-Algorithmen entwickelt, von denen im Folgenden einige kurz besprochen werden sollen. Hierzu gehören NRU, FIFO, Second Chance, Clock Page, LRU und NFU. Es gibt natürlich noch weitere Strategien, die erforscht wurden. Hierfür sei auf die Literatur verwiesen.

NRU. Seiten, die in letzter Zeit nicht genutzt wurden, sind bei *NRU* (Not Recently Used) die Kandidaten für die Verdrängung. Dieser Algorithmus ist einfach zu implementieren, da nur die R/M-Bits (Referenced- und Modified-Bit) zu verwalten sind (siehe Abbildung 7-22). Wie bereits erläutert, werden diese Bits in den Seitentabelleneinträgen der virtuellen Adressräume verwaltet.

...	R	M	...	Frame-Nummer

Abbildung 7-22: Seitentabelleneintrag für NRU

Das M-Bit ist zu setzen, wenn ein Seitenrahmen (und damit die logische Seite) sich verändert, das R-Bit muss gesetzt werden, wenn auf den Seitenrahmen – lesend oder schreibend – zugegriffen wird. Das Setzen der Bits erfolgt dabei meist durch die Hardware, das Zurücksetzen durch die Software (Kernel). In bestimmten Zeitabständen wird das R-Bit gelöscht, so dass nur bei den Seiten das R-Bit gesetzt ist, die in der letzten Zeit auch benutzt wurden. Man unterscheidet vier Klassen von Seiten:

- 1) $R = 0, M = 0$ (Seiten werden als erstes ausgelagert)
- 2) $R = 0, M = 1$
- 3) $R = 1, M = 0$
- 4) $R = 1, M = 1$ (Seiten werden als letztes ausgelagert)

Die Seitenersetzung könnte bei NRU wie folgt durchgeführt werden:

- Wird eine Seitenersetzung fällig, wählt der Memory-Manager, falls vorhanden, eine Seite, die weder modifiziert noch referenziert wurde (Klasse 1).
- Ist keine derartige Seite verfügbar, wählt er eine nicht referenzierte, aber bereits modifizierte Seite aus (Klasse 2).
- Erst wenn es auch keine Seite diesen Typs gibt, wird eine referenzierte, nicht modifizierte Seite ausgewählt (Klasse 3).
- Gibt es auch keine verfügbare Seite der Klasse 3, wird eine Seite der Klasse 4 gewählt.

Eine ausgewählte Seite mit gesetztem M-Bit muss natürlich vor der Verdrängung noch auf die Paging-Area geschrieben werden. Damit stellt man Seiten, die vor längerer Zeit verändert wurden, schlechter als Seiten, die erst vor kurzem referenziert wurden. Man impliziert dabei, dass letztere noch benötigt werden. Bei Seiten, die vor längerer Zeit verändert wurden, nimmt man an, dass sie mit höherer Wahrscheinlichkeit in Zukunft nicht mehr benötigt werden. Die Lösung verspricht zwar nur eine durchschnittliche Leistung, allerdings ist der Algorithmus mit relativ wenig Aufwand zu implementieren.

FIFO. Bei *FIFO* (First In First Out) wird der älteste belegte Seitenrahmen als nächstes ersetzt. Man benötigt also im Seitentableneintrag einen Zeitstempel, der mit dem Zeitpunkt der letzten Einlagerung versehen wird. Das R- und das M-Bit werden für das reine FIFO nicht benötigt (Abbildung 5-21).

...	Frame-Nummer

Abbildung 7-23: Eintrag in der Seitentabelle für FIFO

Das Betriebssystem muss für die Unterstützung des *FIFO-Algorithmus* bei prozess-lokaler Strategie für jeden Prozess eine Liste verwalten, in der die Seitenrahmen absteigend nach der Zeit der letzten Einlagerung verkettet werden. Bei einer globalen Strategie ist nur eine Liste im System zu verwalten. Ganz vorne wird immer der älteste Seitenrahmen platziert. Bei einem Seitenfehler wird der erste Seitenrahmen in der Liste zur Verdrängung ausgewählt, und die neue Seite wird hinten an die Liste gehängt. Bei einer Einlagerung wird die entsprechende Seite also an das Ende der Liste gehängt.

Nachteil von *FIFO* ist, dass evtl. wichtige Seiten aus dem Speicher geworfen werden, obwohl sie noch benötigt würden. Es wird keine Rücksicht auf die aktuelle Nutzung der Speicherseiten genommen.

In Abbildung 7-24 ist der Aufbau einer typischen linearen Liste, die eine FIFO-Organisation repräsentiert, dargestellt. Der Listenanker verweist auf die Seite, die schon am längsten im Hauptspeicher ist. Die Seite mit der Nummer 1 ist also im Beispiel der erste Kandidat für eine Ersetzung, sofern keine Frames mehr frei sind.

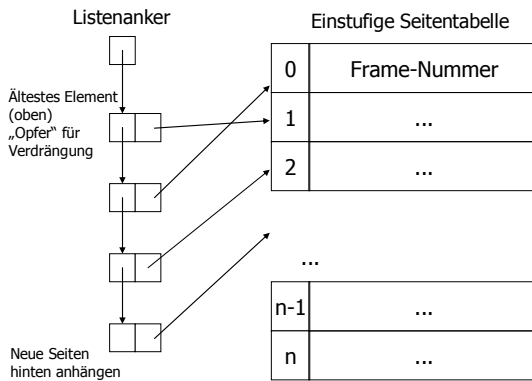


Abbildung 7-24: Beispiel einer FIFO-Organisation

Der Algorithmus wird in der Praxis eingesetzt und ist mit vergleichsweise wenig Aufwand und geringem Overhead implementierbar.

Second Chance. Der *Second-Chance-Algorithmus* ist eine Verbesserung von *FIFO* dahingehend, dass vor einer Verdrängungsentscheidung auch das R-Bit abgefragt wird. Es wird geprüft, ob eine zur Verdrängung anstehende Seite in letzter Zeit benutzt wurde. Wurde die älteste Seite schon genutzt, so wird sie nicht ausgelagert und stattdessen an das Ende der *FIFO*-Liste gehängt. Erst wenn alle Seiten schon referenziert wurden, wird nach *FIFO* ausgewählt. Der vereinfachte Seitentableneintrag für Second Chance ist in Abbildung 7-25 dargestellt.

...	R	...	Frame-Nummer

Abbildung 7-25: Seitentableneintrag für Second Chance

Clock-Page. Der *Clock-Page*- oder *Clock-Algorithmus* ist nur eine Implementierungsverbesserung von *Second-Chance* (vgl. Abbildung 7-26). Die Seiten werden bei *Clock-Page* in zirkulierenden Listen wie eine Uhr verwaltet. Bei einem Seitenfehler wird immer die Seite untersucht, auf die der Zeiger gerade verweist.

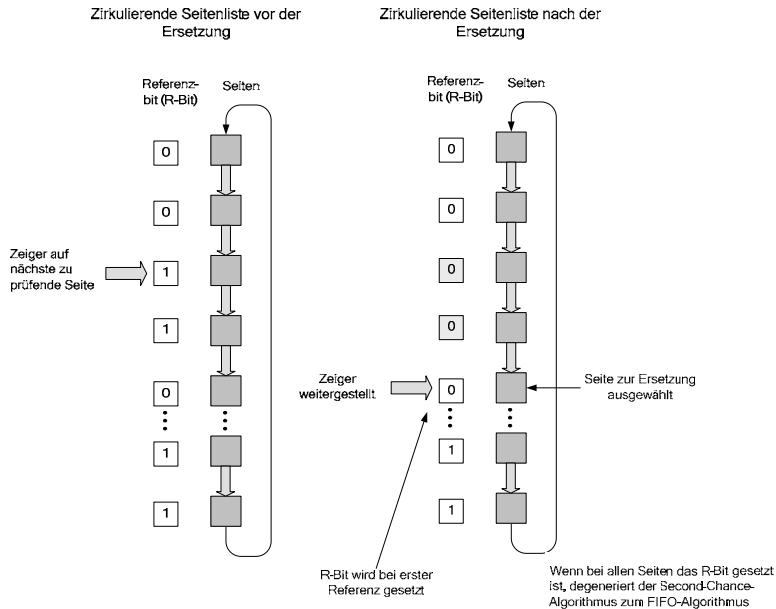


Abbildung 7-26: Clock-Page Algorithmus zur Seitenersetzung (nach Silberschatz 1994)

Der Seitentableneintrag wird nicht umgehängt, und man spart sich einige Listenoperationen, da die Liste nicht so oft verändert werden muss. Man nutzt beim Clock-Page-Algorithmus auch wieder das oben genannte R-Bit. Der Algorithmus funktioniert prinzipiell wie folgt:

- Die Seite, auf die der Zeiger verweist, wird nur ausgelagert, wenn das R-Bit nicht gesetzt ist (R-Bit = 0), die Seite also nicht genutzt wurde.
- Ist das R-Bit = 1, so wird der Zeiger weitergestellt und das R-Bit der aktuell in Prüfung befindlichen Seite auf 0 gesetzt.
- Das geht so lange weiter, bis eine Seite mit R-Bit = 0 gefunden wird. Spätestens nach einer Runde hat man also einen Verdrängungskandidaten.

Wie im Beispiel aus Abbildung 7-26 zu erkennen ist, werden die ersten zwei Seiten, der zirkulierenden Liste nicht für eine Verdrängung verwendet, da das R-Bit dieser beiden Seiten gesetzt ist. Erst die dritte Seite wird als „Opfer“ ausgewählt. Bei der nächsten Suche nach einer zu verdrängenden Seite wird zunächst die der zuletzt verdrängten Seite folgende Seite getestet.

LRU. LRU steht für *Least recently used*. Bei LRU wird diejenige Seite ausgelagert, deren letzte Nutzung (Referenzierung) zeitlich auch am weitesten zurückliegt. Dem LRU-Algorithmus liegt folgende Annahme zugrunde: Seiten, die in der jüngsten Vergangenheit verwendet wurden, werden auch in Zukunft häufig genutzt,

und Seiten, die lange nicht mehr verwendet wurden, werden auch in Zukunft nicht genutzt. Also wird die Seite ersetzt, die am längsten nicht genutzt wurde.

Im Seitentableneintrag benötigt man für die Implementierung von LRU den Zeitstempel der letzten Nutzung (siehe Abbildung 7-27).

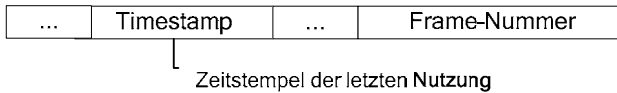


Abbildung 7-27: Seitentableneintrag für LRU

LRU kommt dem optimalen Algorithmus recht nahe, ist allerdings recht aufwändig zu realisieren. Hier wäre eine eigene Hardware zur Berechnung sinnvoll, was aber meist nicht gegeben ist.

Die Problematik soll kurz skizziert werden. Damit man die am längsten unbenutzte Seite schnell im Zugriff hat, muss man einigen Aufwand bei jedem Zugriff leisten. Eine Möglichkeit ist die Verwaltung einer nach der zeitlichen Nutzung sortierten linearen Liste, bei der die am längsten nicht benutzte Seite im ganz obersten Element liegt. Der hohe Aufwand ergibt sich durch das Umhängen der Elemente bei jedem Zugriff. Die aktuell genutzte Seite muss nämlich bei jedem Zugriff an das Ende der Liste gehängt werden. Das Umhängen erfordert in unserem Fall zwei Operationen.

In Abbildung 7-28 ist ein Zugriff auf die Seite 0 eines beliebigen Prozesses skizziert, was dazu führt, dass das entsprechende Listenelement der *LRU*-Liste mit zwei Operationen umgehängt werden muss. Der *next*-Zeiger des Vorgängerelements muss nun auf das Folgeelement (Operation 1) und der *next*-Zeiger des letzten Elements der Liste auf das betroffene Element gestellt werden (Operation 2).

Aufgrund des hohen Aufwands implementiert man heute meistens sog. *Pseudo-LRU-Algorithmen*, wobei auch das R- und das M-Bit benutzt werden. Zwei dieser Algorithmen sind die bereits erläuterten Algorithmen Clock-Page- und Second-Chance.

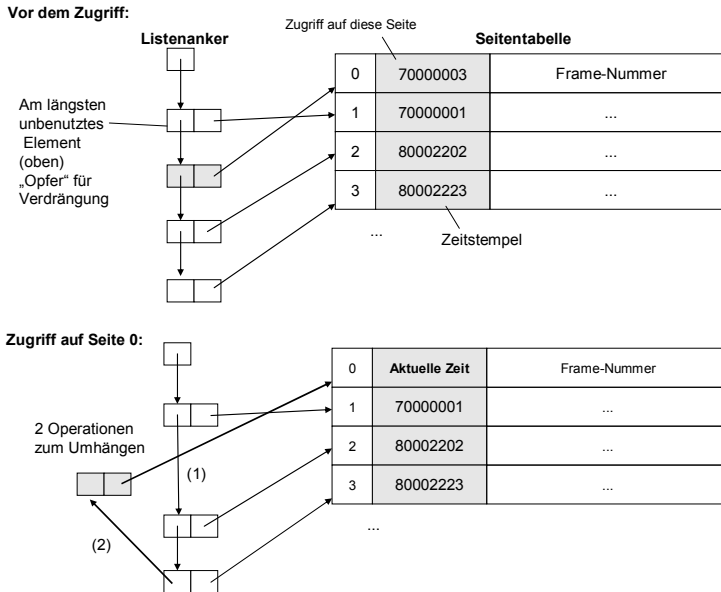


Abbildung 7-28: Listenverwaltung für LRU-Strategie, Umhängen

NFU. Eine gute Alternative zum *LRU*-Algorithmus ist auch der *NFU*-Algorithmus (Not-Frequently Used) in Verbindung mit einem passenden Aging-Mechanismus. Es werden die Seiten ersetzt, die in der letzten Zeit innerhalb eines Zeitintervalls selten benutzt wurden. Ein Eintrag in der Seitentabelle hat zusätzlich einen Zugriffszähler, der mit 0 initialisiert wird. Der Zähler wird bei Benutzung der Seite erhöht. Zur schnellen Ermittlung der bei der nächsten Anforderung zu verdrängenden Seite („Opfer“) kann z.B. auch eine Liste verwaltet werden (siehe Abbildung 7-29). Bei einem Seitenfehler wird die Seite zur Verdrängung ausgewählt, die den kleinsten Zählerwert hat. Im Beispiel wäre die als nächstes zu verdrängende Seite die mit dem Zugriffszählerwert 1. Sie ist in der Liste ganz oben eingehängt. Ein Umhängen innerhalb der Liste kommt bei NFU nicht so oft wie bei LRU vor.

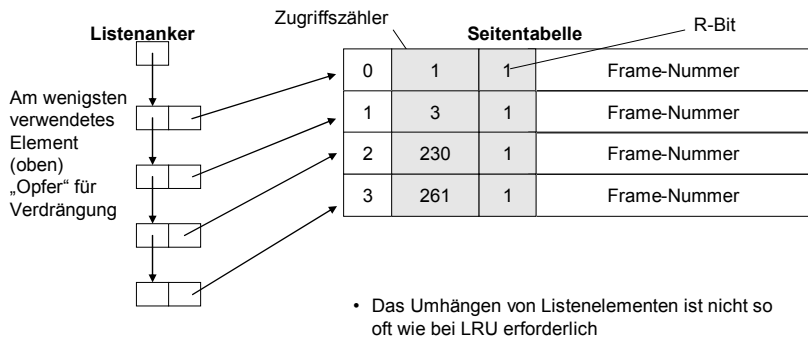


Abbildung 7-29: Listenverwaltung für NFU-Strategie ohne Aging

Das Problem bei NFU ist, dass auch Seiten, auf die vor längerer Zeit häufig zugegriffen wurde, nicht ausgelagert werden. Eine Lösung hierfür bietet die Berücksichtigung des letzten Zugriffs auf eine Seite (Aging). Seiten, die vor längerer Zeit im Zugriff waren und in der letzten Zeit nicht mehr, sind Auslagerungskandidaten. Bei jeder Nutzung einer Seite muss bei diesem Verfahren der aktuelle Zeitstempel in den jeweiligen Seitentableneintrag eingetragen werden. Der Seitentableneintrag zur Unterstützung von NFU mit Aging enthält dann vor allem folgende Felder (Abbildung 7-30):

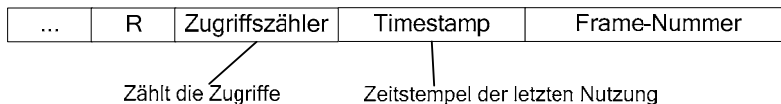


Abbildung 7-30: Seitentableneintrag NFU mit Aging

Neben dem Zugriffszähler wird bei der Aging-Erweiterung auch der Zeitpunkt der letzten Nutzung berücksichtigt. NFU in Verbindung mit *Aging* kommt dem klassischen LRU-Algorithmus schon sehr nahe.

Working-Set-Modell. Bereits in den 70er Jahren wurde von Denning (Baumgarten 2006) noch ein anderes Modell zur Bestimmung der Seiten, die im Hauptspeicher gehalten werden sollen, entwickelt. Dieses Modell wird als *Working-Set-Modell* bezeichnet. Die aktuell benötigte Seitenmenge eines Prozesses bzw. die Menge an Seiten, die innerhalb einer überschaubaren Zeit in der Vergangenheit benutzt wurde, wird hier als *Working-Set* des Prozesses bezeichnet. Ist der Working-Set immer im Hauptspeicher, gibt es theoretisch eigentlich keinen Seitenfehler. Working-Set-Algorithmen gehen von folgenden Annahmen aus:

- Der Seitenbedarf ändert sich aufgrund der Lokalität nur langsam

- Die als nächstes benötigten Seiten sind mit hoher Wahrscheinlichkeit im Working-Set.

Betrachtet man das Beispiel in Abbildung 7-31 so erkennt man anhand der Referenzreihenfolge, dass zum Zeitpunkt t_1 der Working-Set des betrachteten Prozesses die Seiten 1, 2, 5, 6 und 7 enthält. In diesem Beispiel werden zur Bildung des Working-Set immer die 10 letzten Referenzen als sog. Distanz betrachtet. Zum Zeitpunkt t_2 besteht der Working-Set sogar nur noch aus zwei Seiten (Seite 3 und 4). Man schreibt hier – etwas vereinfacht – formal

$$WS(t_2) = \{3,4\}.$$

Die Distanz ist ein wichtiger Parameter für dieses Verfahren und hängt von der Lokalität der Prozesse ab. Da aber die Lokalität erfahrungsgemäß hoch ist, kann die Distanz auch verhältnismäßig klein sein.

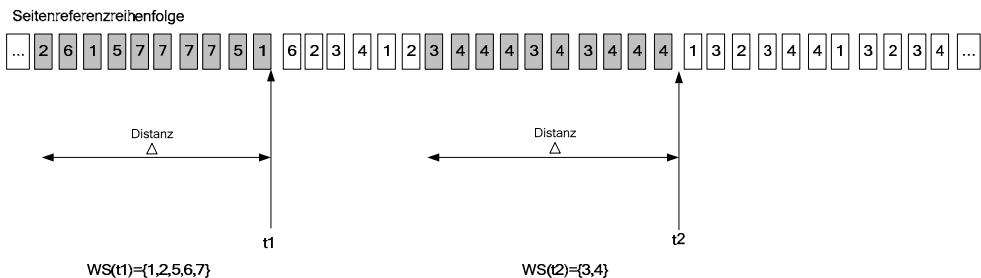


Abbildung 7-31: Working-Set am Beispiel (Silberschatz 1994)

Bei einer Seitenverdrängungsentscheidung wird nun der Working-Set des betroffenen Prozesses (lokale Entscheidung) oder auch die Working-Sets aller Prozesse (globale Entscheidung) betrachtet. Die Seiten, die sich nicht im Working-Set befinden, sind Kandidaten für eine Verdrängung.

Dieses Verfahren kann durch ein präventives Einlagern von Seiten im *Prepaging* unterstützt werden. Prepaging kann auch für Prozesse durchgeführt werden, die gerade nicht aktiv sind. Wenn eine neue Working-Set-Ermittlung durchgeführt wird, und eine Seite des Working-Sets eines Prozesses nicht im Hauptspeicher liegt, könnte diese, wenn möglich, eingelagert werden.

Das Working-Set-Modell ist in realen Betriebssystemen in Reinform kaum vorzufinden. Allerdings sind Varianten des Modells durchaus implementiert. Wie wir noch weiter unten sehen werden, nutzt z.B. Windows eine Variante des Working-Set-Modells.

7.2.4 Vergleich von Seitenersetzungsverfahren

In diesem Abschnitt sollen einige Seitenersetzungsalgorithmen anhand eines einfachen Beispiels verglichen werden. Hierzu wird folgende Referenzreihenfolge (zeitliche Aufeinanderfolge von Seitenzugriffen) verwendet:

2-3-2-1-5-2-4-5-3-2-5-2

Die folgenden Tabellen stellen die Seitenersetzungen gemäß der angegebenen Referenzreihenfolge dar. Die mit Klammern (z.B. (3)) gekennzeichneten Seitenzugriffe führen jeweils zu einer Verdrängung. Je nach Strategie wird dann eine von drei verfügbaren Hauptspeicherseiten verdrängt. In der Paging-Area können in unserem Beispiel vier Seiten untergebracht werden.

Optimale Strategie. Zunächst wird der optimale Algorithmus nach Belady betrachtet. Die Referenzreihenfolge ergibt insgesamt drei Seitenersetzungen. Eine Seitenersetzung erfolgt beim Zugriff auf die Seiten 5, 4 und 2. Beim Zugriff auf die Seite 5 wird z.B. die Seite 1 aus dem Hauptspeicher in die Paging-Area verdrängt.

Zugriff	2	3	2	1	5	2	4	5	3	2	5	2
Hauptspeicher	2	2	2	2	2	2	(4)	4	4	(2)	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	(5)	5	5	5	5	5	5	5
Paging-Area					1	1	1	1	1	1	1	1
							2	2	2	4	4	4

FIFO. Bei Nutzung von FIFO sind sechs Seitenersetzungen für die gleiche Referenzreihenfolge notwendig. Ab dem Einlagern der Seite 5 ist fast bei jedem Zugriff eine Seitenersetzung erforderlich. Die Leistung des Systems ist damit für diese konkrete Referenzreihenfolge mit der reinen FIFO-Strategie sehr schlecht.

Zugriff	2	3	2	1	5	2	4	5	3	2	5	2
Hauptspeicher	2	2	2	2	(5)	5	5	5	(3)	3	3	3
		3	3	3	3	(2)	2	2	2	2	(5)	5
				1	1	1	(4)	4	4	4	4	(2)
Paging-Area					2	3	3	3	5	5	2	4
							1	1	1	1	1	1

LRU. Bei Nutzung der LRU-Strategie müssen in diesem Beispiel vier Seitenersetzungen durchgeführt werden. In diesem Fall ergibt sich also eine deutliche Verbesserung zu FIFO.

Zugriff	2	3	2	1	5	2	4	5	3	2	5	2
Hauptspeicher	2	2	2	2	2	2	2	2	(3)	3	3	3
		3	3	3	(5)	5	5	5	5	5	5	5
				1	1	1	(4)	4	4	(2)	2	2
Paging-Area					3	3	3	3	2	4	4	4
							1	1	1	1	1	1

NFU. Bei Verwendung der NFU-Strategie kommt man auf nur drei Seitenersetzungen und hat damit bei dieser Zugriffsreihenfolge ein besseres Ergebnis als bei LRU. Um die Nachvollziehbarkeit zu erleichtern, ist der Zugriffszähler bei jeder Seitennummer als tiefgestellte Zahl beigefügt. 2_1 bedeutet z.B., dass auf die Seite 2 im Hauptspeicher einmal lesend oder schreibend zugegriffen wurde. Die Ersteinlagerung wird mitgezählt, da hier auch bereits ein Zugriff erfolgt.

Zugriff	2	3	2	1	5	2	4	5	3	2	5	2
Hauptspeicher	2_1	2_1	2_2	2_2	2_2	2_3	2_3	2_3	2_3	2_4	2_4	2_5
		3_1	3_1	3_1	(5_1)	5_1	5_1	5_2	5_2	5_2	5_3	5_3
				1_1	1_1	1_1	(4_1)	4_1	(3_1)	3_1	3_1	3_1
Paging-Area					3	3	3	3	4	4	4	4
							1	1	1	1	1	1

Der Vergleich mit der angegebenen Referenzreihenfolge ist zwar nicht repräsentativ, aber er zeigt doch die prinzipielle Arbeitsweise der einzelnen Strategien auf. Der Leser möge dies an anderen Beispielen und auch mit anderen Strategien weiter nachvollziehen und vertiefen. Interessant ist auch, dass bei gewissen Referenzreihenfolgen eine Erhöhung der Anzahl an Frames sogar zu einer Verschlechterung der Leistung führt. Dies hat auch Belady herausgefunden, und daher wird das Verhalten als *Belady's Anomaly* bezeichnet.

7.2.5 Speicherbelegungs- und Vergabestrategien (Placement)

Über die Laufzeit eines Systems kann es vorkommen, dass die freien Seitenrahmen (Frames) auf den ganzen physikalischen Adressraum verteilt sind. Zusammenhängende Speicherbereiche sind dann für neue Speicheranforderungen fast nicht mehr

vorhanden. Dieses Problem wird als Fragmentierung bezeichnet und muss aus Leistungsgründen möglichst vermieden werden.

Fragmentierung. Man unterscheidet die *interne* und *externe Fragmentierung*. Interne Fragmentierung entsteht, wenn einem Prozess ein größerer Speicherbereich zugewiesen wird, als dieser benötigt. Teile des zugewiesenen Speichers bleiben also ungenutzt. Externe Fragmentierung entsteht, wenn die verfügbaren Speicherbereiche immer kleiner werden. Dadurch werden Speicherzuweisungen immer aufwändiger, da für eine Speicheranforderung mehrere Einzelanforderungen für kleine Speicherbereiche erforderlich sind.

Der Hauptspeicher muss so verwaltet werden, dass man bei der Einlagerung möglichst schnell einen freien Seitenrahmen bzw. eine Menge von zusammenhängenden Seitenrahmen finden kann und auch möglichst wenig Speicher verschwendet wird. Dies bedeutet, dass sowohl interne als auch externe Fragmentierung so gut wie möglich zu vermeiden ist.

Für das schnelle Auffinden von freien Speicherbereichen ist eine Speicherbelegungsstrategie (Placement Policy) notwendig. Vom Memory-Manager wird meist eine einfache Bit-Map verwaltet, in der jedem Seitenrahmen ein Bit zugeordnet wird. Ist das Bit auf 1 gesetzt, so ist der Seitenrahmen gerade belegt. Freie Hauptspeicherbereiche erkennt man an aufeinanderfolgenden Nullen (Abbildung 7-32).

Die Größe der Bit-Map hängt mit der Größe des Hauptspeichers zusammen. Hat der Hauptspeicher z.B. 4 GB bei einer Seitengröße von 4 KB, so gibt es eine Million Seitenrahmen. Die Bit-Map hat demnach eine Größe von ca. 125.000 Byte.

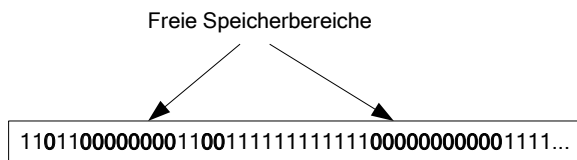


Abbildung 7-32: Bit-Map zur Verwaltung der Speicherbelegung

Welche Seitenrahmen nun für die Einlagerung neuer Seiten verwendet werden, entscheidet eine Vergabe- oder Placement-Strategie, die Fragmentierung vermeiden sollte. Mögliche Strategien hierfür sind:

- Sequentielle Suche des nächsten passenden Bereichs (*First Fit*). Diese Technik ist nicht besonders effizient und verhindert eine Fragmentierung nicht.
- Optimale Suche nach dem passendsten Bereich (*Best Fit*)., um Fragmentierung möglichst zu vermeiden. Die Suche ist hier sehr aufwändig.

- Buddy-Technik bzw. Halbierungsverfahren, d.h. schrittweise Halbierung des Speichers und dabei Suche nach dem kleinsten geeigneten Bereich. Die Suche ist beendet, wenn ein Bereich gefunden wird, in den die neuen Seiten gerade noch hineinpassen. Bei der Speicherfreigabe werden Seitenrahmen wieder zusammengefasst. Der freigegebene Bereich wird also mit benachbarten freien Bereichen zu einem größeren freien Bereich verbunden.

Buddy-Technik. Da die Buddy-Technik häufig verwendet wird, soll sie etwas ausführlicher diskutiert werden. Die Technik verwaltet freie Speicherbereiche nach aufsteigender Größe in Listen. Nebeneinanderliegende, freie Speicherbereiche gleicher Größe werden immer zu zwei Partnern zusammengefasst, die man als Buddies (deutsch: Kumpel) bezeichnet. Beide Partner können unabhängig voneinander vergeben werden. Wenn also eine Speicheranforderung befriedigt werden muss, die nur die Hälfte eines freien Speicherbereichs benötigt, wird der Bereich zerlegt. Ein Teil wird vergeben und der zweite wird in eine Liste mit entsprechend kleineren freien Bereichen eingehängt. Sofern die beiden Bereiche wieder frei sind, fasst die Speicherverwaltung sie zu einem großen Paar zusammen, das gewissermaßen als Buddy der nächsten Stufe dient.

Alle Buddies einer Größe werden also im Kernel in einer eigenen Liste verwaltet, wie dies in Abbildung 7-33 zu sehen ist. Wie das Bild zeigt, sind die Größen der Speicherbereiche Potenzen von 2. Das Bild ist von oben nach unten zu interpretieren und zeigt drei Zustände des Buddy-Systems an.

In Szenario 1 ist z.B. nur ein Speicherbereich mit 16 Seitenrahmen (Größe 2^4) dargestellt. Allgemein werden also Seitenrahmen der Größe 2^i verwaltet, wobei i gleichzeitig der Index auf eine Tabelle ist, in der Verweise auf die Listen der Speicherbereiche verwaltet werden. Im Bild wird der Einfachheit halber jeweils nur ein Listenelement dargestellt. Eine Anwendung fordert zunächst einen Speicherbereich mit acht Seitenrahmen an. Der vorhandene Speicherbereich der Größe 2^4 (= 16 Seitenrahmen) wird in zwei gleich große „Buddies“ zerlegt. Einer der Buddies wird der Anwendung zugeordnet und der andere wird in die Liste der 8-seitigen Speicherblöcke gestellt (Listenbezeichnung 2^3). Anschließend benötigt die Anwendung einen Speicherblock mit zwei Seitenrahmen. Daher wird der Block aus acht Speicherrahmen in zwei Buddies zu je vier Seitenrahmen aufgeteilt. Nun wird ein 2-seitiger Block an die Anwendung übergeben und die restlichen werden in die entsprechenden Listen des Buddy-Systems eingetragen (Listenbezeichnungen 2^1 und 2^2). Im Beispiel bleiben somit ein freier 4-seitiger und ein 2-seitiger Speicherblock in den entsprechenden Listen im System übrig. Umgekehrt, also wenn die Anwendungen den Speicher nicht mehr benötigen, werden die freigegebenen Speicherblöcke als Buddies wieder in die passenden Listen eingetragen.

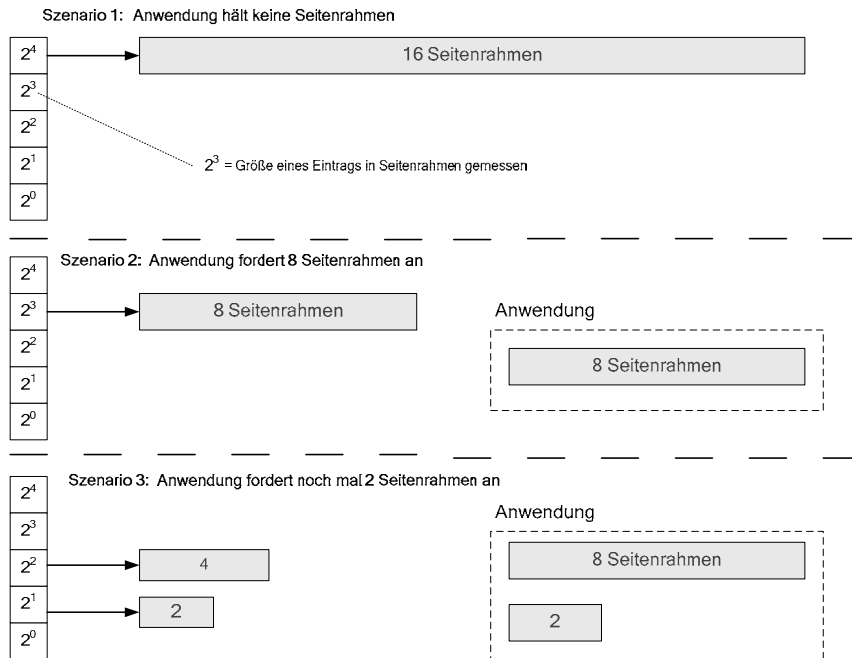


Abbildung 7-33: Beispielbelegung bei der Buddy-Technik (Mauer 2004)

Mit dieser Technik steigt die Wahrscheinlichkeit, dass bei einer Speicheranforderung die richtige Speichergröße zur Verfügung steht. Zudem wirkt die Buddy-Technik einer externen Fragmentierung entgegen. Dies geht aber auf Kosten einer verstärkten internen Fragmentierung.

Der Kernel selbst benötigt meist kleinere Speicherbereiche als die Anwendungen. Die Speicherbelegung sollte hier sehr schnell gehen, weshalb in konkreten Betriebssystemen auch noch verfeinerte Techniken verwendet werden.

7.2.6 Entladestrategie (Cleaning)

Eine weitere, schon mehrmals angesprochene Aufgabe bei der Speicherverwaltung für die virtuelle Speichertechnik ist das „Sauberhalten des Hauptspeichers“, das über eine Entladestrategie (Cleaning Policy) realisiert wird. Diese legt fest, zu welchem Zeitpunkt eine im Hauptspeicher modifizierte Seite auf die Paging-Area geschrieben wird. Hier gibt es mehrere Varianten.

Demand-Cleaning. Eine einfache Möglichkeit ist das sog. *Demand-Cleaning*. In diesem Fall wird das Entladen einer Seite bei Bedarf durchgeführt, also wenn der belegte Seitenrahmen benötigt wird. Demand-Cleaning hat zwar den Vorteil, dass

eine Seite so lange wie möglich im Hauptspeicher verbleibt. Nachteilig ist aber, dass einer Seitenverdrängung immer unmittelbar vorher ein Zurückschreiben vorausgeht, wodurch die Wartezeit eines Prozesses, der eine Seite benötigt, unnötig verlängert wird.

Precleaning. Eine weitere Strategie sieht ein präventives Zurückschreiben veränderter (dirty) Seiten vor. Diese Strategie wird als *Precleaning* bezeichnet. Damit ist bei einer Seitenrahmenanforderung in der Regel sofort ein Seitenrahmen verfügbar. Allerdings kann es vorkommen, dass noch benötigte Seiten entladen werden.

Page-Buffering. Wieder eine andere Möglichkeit, das sog. *Page-Buffering*, puffert zur Verdrängung ausgewählte Seiten zunächst, bevor eine echte Entladung durchgeführt wird. Meist werden hierfür mehrere Listen verwaltet. Eine sog. *Modified List* enthält Seitenrahmen, die freigegeben werden können und verändert wurden. Eine *Unmodified List* enthält nicht modifizierte Seiten. Die modifizierten Seiten werden zunächst zwischengepuffert und dann zyklisch gleich in ganzen Gruppen auf die Platte übertragen. Danach werden sie der *Unmodified List* zugeordnet und können noch verwendet werden. Wird nun eine dieser Seiten erneut benötigt, kann sie einfach aus der *Unmodified List* entnommen werden, was lediglich mit einer Eingliederung in eine andere Liste verbunden ist („schnelle Verdrängung“). Dies ist wesentlich weniger zeitaufwändig, als wenn eine Seite eingelagert werden müsste. Nur wenn ein Seitenrahmen benötigt wird und kein anderer freier Seitenrahmen mehr verfügbar ist, werden Seitenrahmen aus der *Modified List* zugeordnet. *Page Buffering* wird heute in modernen Betriebssystemen üblicherweise verwendet.

7.2.7 Eine weitere Technik: Segmentadressierung

Die Einteilung des Hauptspeichers in Segmente (Segmentierung) ist neben der virtuellen Adressierung ein zusätzlicher Mechanismus der Speicherverwaltung. Bei der Segmentadressierung wird der Prozessadressraum in mehrere Segmente unterteilt. Segmente sind üblicherweise größer als Seiten. Man spricht auch von Segment-Seitenadressierung, wenn man Segment- und Seitenadressierung kombiniert (Baumgarten 2006). Jedes Segment wird dabei wiederum in gleich große Seiten aufgeteilt und man nutzt dadurch die Vorteile beider Verfahren. Die Adressangabe enthält die Segmentnummer, die Seitennummer und die relative Byte-Adresse innerhalb der Seite.

Die Hardware unterstützt die Segmentierung üblicherweise durch die Bereitstellung spezieller Register, sog. Segmentregister, deren Inhalte als *Deskriptoren* bezeichnet werden. Das Programmiermodell wird entsprechend komplizierter, und der Programmierer muss sich dieser Technik bewusst sein. Segmentierung wird

von gängigen Rechnerarchitekturen wie IBM RS/6000, IBM ESA/370, ESA/390, HP PRECISION usw. unterstützt, allerdings von Betriebssystemen wie Linux und Windows nicht genutzt.

Ein etwas anderes *Segmentierungsmodell* unterstützen die Intel-Prozessoren (Intel 80x86-Prozessoren, Pentium, Pentium II, Pentium III). Wie in Abbildung 7-34 ersichtlich ist, unterstützt der Intel-80386-Prozessor z.B. sechs Segmentregister. Die Inhalte der Segmentregister werden nicht Deskriptoren, sondern *Selektoren* genannt. Selektoren verweisen wiederum auf eine *Deskriptor-Tabelle*. Insgesamt gibt es zwei Deskriptor-Tabellen, deren Anfangsadressen in zwei weiteren CPU-Registern, dem *LDTR* (Local Descriptor Table Register) und dem *GDTR* (Global Descriptor Table Register) liegen. Der gesuchte Deskriptor wird in einer Deskriptor-Tabelle gefunden, und zwar wird hierzu ein Selektorregister-Inhalt als Index verwendet.

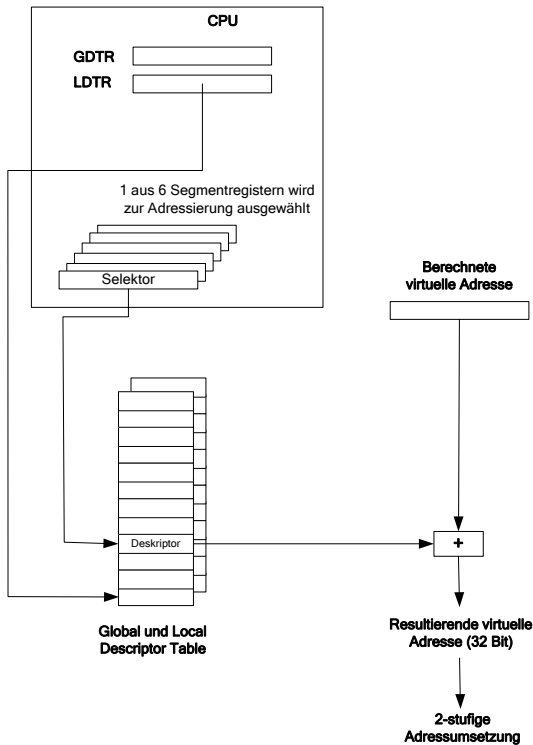


Abbildung 7-34: Segmentierung am Beispiel der Intel-Prozessoren (Herrmann 2002)

Mit dem Inhalt eines Eintrags aus der Deskriptor-Tabelle wird die virtuelle Adresse ermittelt, die 32 Bit breit ist. Diese wird dann in der bekannten Weise über eine zweistufige Seitentabelle auf eine reale Adresse abgebildet.

Für jeden Prozess wird eine Local Descriptor Table (LDT) verwaltet. Globale Segmente können in der globalen, für alle Prozesse gemeinsamen Tabelle, liegen. Diese Segmente sind zum Beispiel dem Kernel zugeordnet und/oder werden als Shared Memory genutzt.

Auch dieses Segmentierungsmodell wird heute von modernen Betriebssystemen praktisch nicht benutzt, da diese von einem flachen, linearen Adressraummodell ausgehen.

7.2.8 Shared Memory

Das virtuelle Speichermodell eröffnet auch die Möglichkeit, gemeinsam genutzte Speicherbereiche nur einmal für mehrere Prozesse in den Hauptspeicher zu laden. Auf die gemeinsam genutzten Speicherbereiche verweisen die Seitentableneinträge verschiedener Prozesse. Diese Speicherbereiche bezeichnet man als Shared Memory. Ein Shared Memory ist also ein von mehreren Prozessen gemeinsam genutzter Speicherbereich, der nur einmal in den Hauptspeicher geladen wird.

Ziel eines Shared Memory ist es, Hauptspeicher zu sparen oder einen prozessübergreifenden Kommunikationsmechanismus zum Datenaustausch bereitzustellen. Hauptspeicher kann eingespart werden, wenn z.B. Codeteile, die mehrere Prozesse benötigen, nur einmal im Hauptspeicher liegen.

Diese Codeteile werden auch als Shared Libraries oder DLLs bezeichnet. Shared Libraries sind gemeinsam genutzte Bibliotheken aus der Unix-Welt, die nur einmal in den Hauptspeicher geladen werden und vielen Prozessen zur Verfügung stehen (siehe Abbildung 7-35). DLLs (Dynamic Link Libraries) sind etwas Gleichwertiges im Windows-Umfeld.

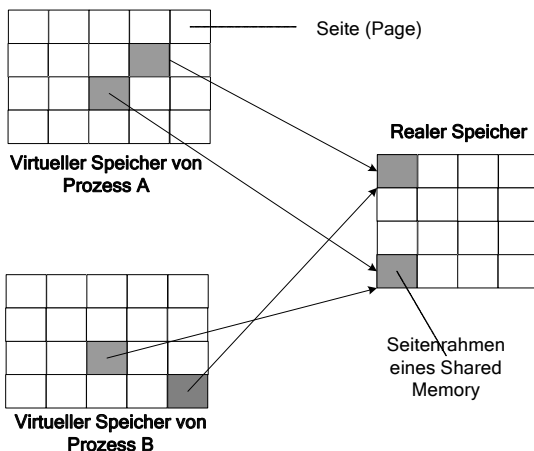


Abbildung 7-35: Virtueller Speicher und Shared Memory

Globale Datenbereiche (Shared Memory) dienen dem Austausch von Informationen zwischen Prozessen. Beispiele für die Nutzung eines Shared Memory sind die Caches von Datenbankmanagementsystemen, auf die mehrere Datenbank-Benutzerprozesse nebenläufig zugreifen.

Beim Zugriff auf gemeinsame Datenbereiche sind Synchronisationsmechanismen wie Semaphore oder Monitore erforderlich, um den Zugriff zu serialisieren und um damit Konflikte und Anomalien zu vermeiden. Gemeinsam genutzte Codeteile müssen wiedereintrittsfähig (reentrant, threadsafe) sein, also so programmiert werden, dass ein Prozess nicht einen anderen, der denselben Code nutzt, stört.

7.3 Speicherverwaltung in ausgewählten Systemen

In diesem Abschnitt betrachten wir bei einigen ausgewählten Betriebssystemen die Implementierungskonzepte der Speicherverwaltung. Wir beginnen mit Linux und fahren anschließend mit Windows (Windows 2000 und folgende) fort. Betriebssysteme nutzen die Möglichkeiten, die ihnen die Hardware zur Verfügung stellt. Die Prozessorarchitektur bestimmt daher auch die Speicherverwaltung im Betriebssystem. Generell ist anzumerken, dass die verschiedenen Speicherstrategien ineinander verzahnt arbeiten und daher nicht isoliert betrachtet werden können.

7.3.1 Linux-Speicherverwaltung

Die ersten Unix-Systeme nutzten ausschließlich Swapping und kein Paging. Hierfür war ein Swapper-Prozess (mit PID 1) vorhanden, der bei Bedarf das Swapping, also das Auslagern eines kompletten Prozesses mit allen seinen Frames und auch eine zyklische Überprüfung der Hauptspeicherauslastung vornahm. Ab BSD 3 wurde Demand Paging ergänzt, alle anderen Unix-Derivate (System V) haben es übernommen. Heute eingesetzte Unix-Systeme nutzen vorwiegend Demand Paging. Der Begriff Swapping wird unter Unix und Linux oft mit Paging gleichgesetzt. Man meint damit dann in der Regel Paging. Swapping wird kaum mehr verwendet.

Virtueller Adressraum und Adressierung:

Der Adressraum von Linux liegt für 32-Bit-Betriebssysteme bei vier GB (32-Bit-Adressen) mit Seitengrößen von 4 KB. Davon sind drei GB dem User- und ein GB dem Systemadressraum zugeordnet.

Bei 64-Bit-Linux-Derivaten werden z.B. je nach Prozessor bis zu 48 Bit breite virtuelle Adressen verwendet, was einen maximalen Adressraum von 2^{48} (2^{40} ent-

spricht einem Terabyte) ermöglicht. Dies ist weit mehr als der tatsächlich physikalisch vorhandene Hauptspeicher.

Abbildung 7-36 skizziert den virtuellen Adressraum unter Linux für eine 32-Bit-Architektur (wie etwa die IA32-Architektur). Der Benutzeradressraum beginnt bei der virtuellen Adresse 0 und endet im Bild bei 3 GB. Der Kerneladressraum ist ein GB groß. Alle Prozesse nutzen den gleichen Adressbereich für den Kernel, der auch physikalisch an einer Stelle im Hauptspeicher liegt. Der Benutzeradressraum wird auf beliebige freie Seitenrahmen im Hauptspeicher gemappt. Durch die architekturenspezifische Konstante `TASK_SIZE` wird die Größe des Benutzeradressraums je nach Prozessor-Architektur konfiguriert. Bei IA32-Prozessoren ist die Konstante `TASK_SIZE` auf drei GB eingestellt, bei 64-Bit-Architekturen ist der Wert natürlich viel größer. Bei der Kernel-Konfigurierung sind weitere Einstellungen möglich. Es können z.B. mehrere Auslagerungsdateien (Paging-Areas) im Verbund konfiguriert werden (Glatz 2006).

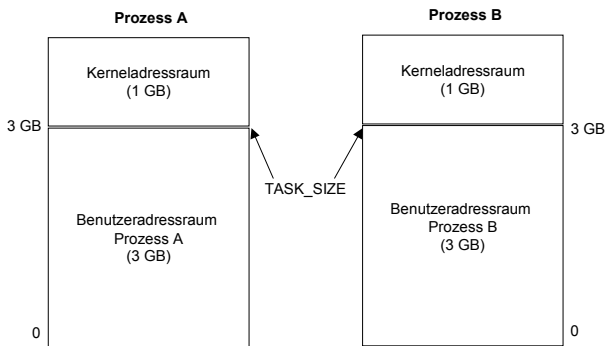


Abbildung 7-36: Belegung des Adressraums bei einer 32-Bit-Architektur unter Linux

In der Regel werden bei Unix-Derivaten mehrstufige, in Linux sogar dreistufige und ab Linux-Version 2.6.11 sogar vierstufige Seitentabellen verwendet, die aber nicht auf jedem Prozessor implementierbar sind. So unterstützen Intel-IA32-Prozessoren z.B. nur zweistufige Seitentabellen, weshalb eine Abbildung des Softwarekonzepts auf das Hardwarekonzept erforderlich ist. Diese Abbildung übernimmt der Kernel. Ähnliches gilt für Prozessoren, die mehr als drei Stufen unterstützen (siehe z.B. *AMD Opteron* mit einer vierstufigen Seitentabelle). Spezielle Segmentierungsmechanismen der unterlagerten Prozessoren werden in der Regel gar nicht genutzt.

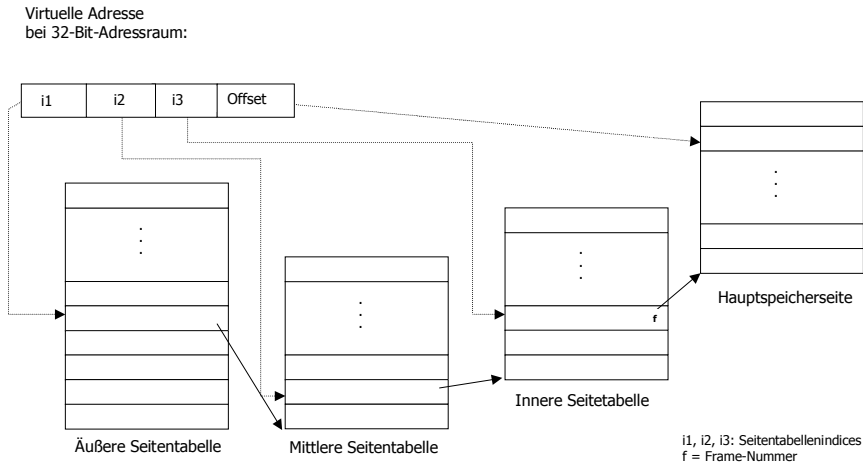


Abbildung 7-37: Dreistufige Seitentabelle unter Linux (Tanenbaum 2002)

Abbildung 7-37 zeigt beispielsweise eine dreistufige Seitentabelle des Linux-Betriebssystems (bis Version 2.6.10) für einen 32-Bit-Adressraum mit einer äußeren, einer mittleren und einer inneren Seitentabelle. Die virtuelle Adresse ist in vier Teile gegliedert:

- Drei Indices i1, i2 und i3, die auf Seitentabellen verweisen.
- Ein Offset (Distanz) zur Adressierung des gewünschten Byte innerhalb der adressierten Seite.

Die Größen der einzelnen Indices und des Offsets hängen vom Prozessor ab (Bovet 2005). Ab Linux-Version 2.6.11 kommt noch ein vierter Index hinzu.

Eine konkrete Abbildung der dreistufigen Seitentabelle unter Linux auf Intel-x86-Prozessoren (IA32-Architektur) wird in Abbildung 7-38 skizziert. Wie man sieht wird der Index i2 auf die mittlere Seitentabelle vom Betriebssystem mit 0 Bit belegt und ist damit nicht vorhanden.

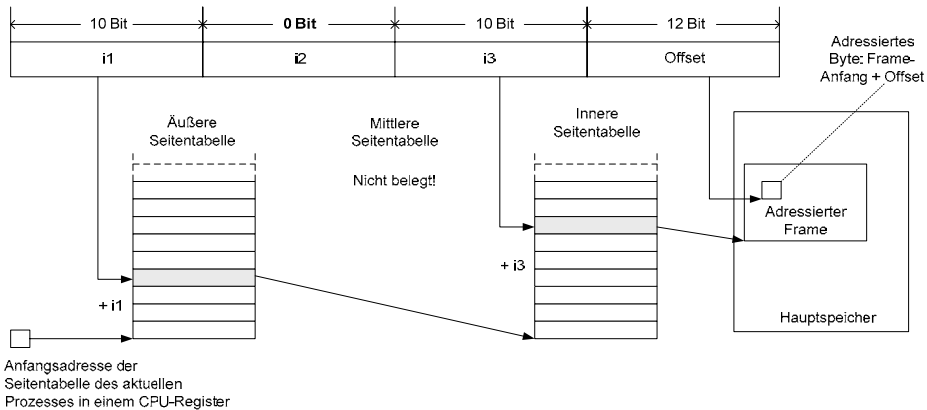


Abbildung 7-38: Mapping der dreistufigen Linux-Seitentabelle auf x86-Adressen

Verwendete Strategien:

Fetch Policy. Als Fetch Policy wird prinzipiell das Demand-Paging-Verfahren angewendet. Seiten werden also erst auf Anforderung der Prozesse in den Hauptspeicher eingelesen.

Replacement und Cleaning Policy. Zur Seitenersetzung wird in Linux-Systemen eine Art Clock-Page-Algorithmus verwendet. Je Seite wird ein Reference- und ein Modified-Bit verwaltet. Im Kernel werden mehrere Listen geführt, die saubere (clean), schmutzige (dirty) und gesperrte (locked) Seitenrahmen verwalten. Bei jedem Seitenzugriff wird ein Referenzzähler erhöht. Spezielle Kernel-Threads ohne eigenen Adressraum wie *kswapd* prüfen auch präventiv, welche Seiten nicht mehr verwendet werden. Es wird mit einer Kombination aus lokaler und globaler Ersetzung gearbeitet.

Der Kernel-Thread *kswapd* setzt periodisch das R-Bit jeder allokierten Seite (Reference-Bit) in der Seitentabelle auf 0 und wartet dann eine bestimmte Zeit. Nach dieser Zeit hängt er die Seiten, bei denen das R-Bit immer noch auf 0 ist, in eine andere Liste um. Dies sind dann Kandidaten für eine Verdrängung. Nicht veränderte Seitenrahmen werden in die Liste mit den sauberen Seiten umgehängt. Veränderte Seitenrahmen in eine Modified-Page-Liste.

Der Kernel-Thread *pdflush* (früher *bdflush*) schreibt veränderte („dirty“) Seitenrahmen periodisch auf die Paging-Area, allerdings auch nur dann, wenn eine bestimmte Speicherauslastung erreicht ist..

Die *Entladestrategie* (Cleaning) kann prinzipiell als *Page Buffering* bezeichnet werden.

Placement Policy. Zur Verwaltung der belegten Frames wird in Linux die Buddy-Technik eingesetzt. Für die Verwaltung von Kernel-Speicherbereichen verwendet der Linux-Kernel einen Cache, den sog. *Slab-Cache*, der auf der Buddy-Technik aufsetzt und die Speicherblöcke nochmals unterteilt, um kleinere Speicherobjekte schnell bereitstellen zu können.

7.3.2 Windows-Speicherverwaltung

Die Speicherverwaltung von Windows richtet sich nach der Prozessorarchitektur. Es gibt daher Unterschiede bei den Windows-Derivaten. Ein Windows, das auf Basis eines 32-Bit-Prozessors läuft (IA32-Architektur, kurz: 32-Bit- oder x86-Windows), differiert in einigen Aspekten zu einem Windows, das eine IA64- und x64-Architektur (kurz: IA64- oder x64-Windows) nutzt.

Die Paging-Area wird unter Windows als *Auslagerungsdatei* bezeichnet. Es werden mehrere Auslagerungsdateien unterstützt. Der Sitzungs-Manager-Prozess *smss.exe* öffnet die Auslagerungsdateien und holt sich hierzu die Informationen aus der Windows-Registry. Die Auslagerungsdateien und deren Größe kann verändert werden.¹

Legt man keine Auslagerungsdateien im System explizit an, werden abhängig vom jeweiligen Windows-Derivat und vom verfügbaren Hauptspeicher Standardgrößen verwendet. Bei Windows Server 2003 und auch bei Windows XP wird z.B. die Paging-Area im Minimum über die Formel $1 * \text{Hauptspeichergröße}$ und im Maximum über die Formel $3 * \text{Hauptspeichergröße}$ berechnet. Diese Formeln gelten bei Hauptspeichergrößen über einem Gigabyte (Solomon 2005).

In der Windows-Speicherverwaltung gibt es noch eine Fülle von interessanten Mechanismen, wie z.B. der Cache-Manager, die aus Platzgründen hier nicht alle behandelt werden können. Einige Mechanismen sind in (Solomon 2005) ausführlicher beschrieben. Wichtige Aspekte der Windows-Speicherverwaltung werden im Folgenden kurz skizziert.

Virtueller Adressraum und Adressierung:

Zur Verwaltung der virtuellen Adressräume sind je nach Prozessor-Architektur *zwei- oder mehrstufige Seitentabellen* vorgesehen. Bei x86-Architekturen ist die Ad-

¹ Zur Veränderung der Auslagerungsdatei nutzt man als Systemadministrator die Systemsteuerung. Im Ordner System unter dem Register Erweitert können Systemleistungsoptionen verändert werden. Hier kann man auch die Auslagerungsdatei verändern. Eine Veränderung wirkt sich auf den zugesicherten Speicher, konkret auf den Grenzwert aus, der im Taskmanager im Register Systemleistung angezeigt wird.

ressierung zweistufig, bei x64-Windows sogar vierstufig. Die Adresstransformation wird bei x64-Windows über eine 48 Bit breite virtuelle Adresse mit 4 Indices zu je 9 Bit (je Seitentabellen-Level) und 12 Bit für die Distanz innerhalb des Frames durchgeführt. Auf die Nutzung von Segmentierungsmechanismen, die z.B. von Intel/AMD-Prozessoren angeboten werden, wird verzichtet. Im 32-Bit-Windows werden 2 GB für den Benutzeradressraum eines 32-Bit-Prozesses bereitgestellt. Andere 32-Bit-Windows-Produkte unterstützen auch noch größere virtuelle Adressräume. 32-Bit-Prozesse unter 64-Bit-Architekturen sind auch möglich (Kompatibilitätsmodus). In diesem Fall kann der Prozessadressraum bis zu 4 GB groß sein. Die Seitengröße beträgt beim 32-Bit-Windows maximal 64 KB, beim Intel Pentium 4 KB und beim Intel Itanium 8 bzw. 16 KB. Auch sog. *Large Pages* zur Minimierung der Seitentabellen mit einer Größe von 4 MB (x86) bis 16 MB (IA64) werden unterstützt. Die Tabelle 7-1 zeigt die Seitengrößen nach Prozessorarchitekturen. Allerdings muss für die Nutzung von Large Pages genügend Hauptspeicher verfügbar sein (bei Windows XP z.B. mehr als 255 MB).

Tabelle 7-1: Seitengrößen unter Windows in Abhängigkeit der Prozessorarchitektur nach Solomon 2005

Prozessorarchitektur	Größe der Small Page	Größe der Large Page
X86	4 KB	4 MB
x64 (AMD)	4 KB	2 MB
IA64 (Intel)	8 KB	16 MB

Die Belegung des virtuellen Adressraums differiert, je nach Windows-Variante. In Abbildung 7-39 ist die Belegung des virtuellen Adressraums für das 32-Bit-Windows dargestellt. Im Adressraum unter 2 GB sind im Wesentlichen die privaten Daten und der private Code eines Prozesses abgelegt. Dieser ist auch vor dem Zugriff durch andere Prozesse geschützt. Der Adressraum über 2 GB ist für den Kernelmodus reserviert und in einer vorgegebenen Struktur aufgeteilt.

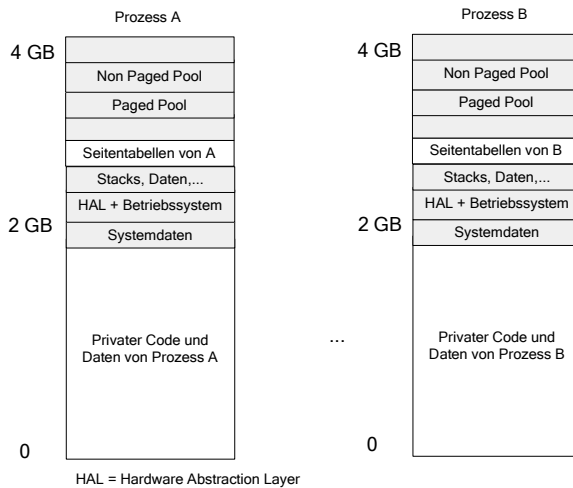


Abbildung 7-39: Belegung des Adressraums bei einer 32-Bit-Architektur unter Windows nach Tanenbaum 2002

Beispiele für definierte Speicheradressen (siehe Solomon 2005) sind:

- Ab Adresse 0x80000000 beginnt der Betriebssystemcode (NTOSKRNL.DLL, Gerätetreiber, Code der HAL, HAL.DLL, usw.)
- Ab Adresse 0xC0000000 bis 0xC03FFFFFF sind in 4 MB die prozessspezifischen Seitentabellen abgelegt. Für jeden Prozess ist dieser Bereich anders belegt.
- Ab Adresse 0xFFC00000 ist Speicherplatz für die HAL-Nutzung reserviert.
- Im sog. Non-Paged Pool liegen Speicherseiten des Betriebssystems, die nicht ausgelagert werden dürfen.
- Im Paged Pool liegen Seiten des Betriebssystems, die auch auf die Paging-Area ausgelagert werden dürfen.

Die genaue Speicherbelegung bei x64- und IA64-Windows ist der Literatur zu entnehmen (Solomon 2005). Theoretisch kann man mit einer 64-Bit-Adresse 16 Exabyte adressieren. Dieser immense Speicherbereich wird unter x64- und IA64-Windows verwaltungstechnisch in mehrere Bereiche (Regionen) eingeteilt. Neben der Region für den Benutzeradressraum gibt es noch den System-Cache, den Paged-Pool, den Non-Paged-Pool usw. Die Größe der Regionen liegt für beide Prozessorarchitekturen jeweils bei 128 GB. Eine Ausnahme bildet die Region des Benutzeradressraums, die bei 7152 GB (IA64) bzw. 8192 GB (x64) liegt. Die Größe verschiedener Speicherbereiche ist in Windows auch konfigurierbar.

Verwendete Strategien:

Fetch Policy. Als *Einlagerungsstrategie* wird das *Demand-Paging*-Verfahren unterstützt. Ab Windows XP bzw. 2003 wurde zusätzlich auch ein *Prefetch*-Verfahren eingeführt, mit dem versucht wird, präventiv, also vor dem eigentlichen Zugriff, die benötigten Seiten einzulagern. Meist wird zusammen mit einer angeforderten Seite auch gleich ein Prefetching nachfolgender Seiten ausgeführt.

Replacement und Cleaning Policy. Als *Auslagerungs- und Ersetzungsstrategie* wird unter Windows eine Kombination aus lokaler und globaler Ersetzung realisiert, die auf dem Working-Set-Verfahren aufsetzt. Lokale Ersetzung bedeutet in diesem Zusammenhang, dass eine Seite des anfordernden Prozesses entfernt wird, bevor eine Seite eines anderen Prozesses ausgewählt wird. Eine globale Ersetzung bedient sich auch der Seiten anderer Prozesse. In Multiprozessormaschinen wird zur Auswahl der Auslagerungskandidaten eine *lokale FIFO-Strategie* verwendet. Wenn also eine Seite ausgelagert werden muss, so wird die ausgewählt, die schon am längsten im Hauptspeicher war. Bei Einprozessormaschinen wird ein *Pseudo-LRU-Algorithmus* verwendet, der auf dem *Clock-Page-Verfahren* basiert. Es wird also versucht, Seiten für die Auslagerung zu finden, die am längsten nicht mehr im Zugriff waren.

Ein weiterer Mechanismus der Windows-Speicherverwaltung ist ein spezielles Working-Set-Verfahren. Jeder Prozess erhält einen sog. *Working-Set*, also eine Arbeitsmenge an Frames, die im Rahmen einer Working-Set-Strategie zugeordnet werden. Jeder Prozess verfügt über einen Working-Set von mindestens 50 und maximal 345 Seiten je nach verfügbarem Hauptspeicher. Diese Angabe bezieht sich auf x86-Windows, bei 64-Bit-Windows sind die Werte anders. Stark „pagende“ Prozesse können den zugewiesenen Working-Set auch erhöhen, aber nur maximal auf die Anzahl aller aktuell verfügbaren Seiten - 512 (siehe Solomon 2005). Wenn ein Working-Set eines Prozesses an seine Obergrenze kommt, entfernt die Speicherverwaltung zunächst Seiten von diesem Prozess und danach erst, falls noch nötig, Seiten anderer Prozesse. Es wird zyklisch überprüft, ob die Working-Sets der Prozesse noch zur gesamten Hauptspeicherauslastung passen. Wenn noch genügend Hauptspeicher vorhanden ist, wird bei einem Seitenfehler etwas mehr geladen. Ist der Hauptspeicher knapp, werden Seitenrahmen präventiv entzogen, aber noch in Listen verwaltet, bis sie tatsächlich von anderen Prozessen benötigt werden. Als *Entladestrategie* wird also prinzipiell *Page Buffering* verwendet. Die Windows-Hauptspeicherverwaltung führt einige Listen, sog. *Auslagerungslisten* zur Optimierung der Seitenersetzung und der Frame-Auswahl, die im Folgenden kurz beschrieben werden sollen:

- Die *Modified-Page-List* enthält Seiten, die bereits für eine Seitenersetzung ausgewählt, aber noch nicht ausgelagert wurden. Diese Seiten sind noch einem Prozess zugeordnet.
- Die *Standby-Page-List* enthält Frames, die bereits sauber gemacht wurden, also eine Kopie auf der Paging-Area haben und noch einem Prozess zugeordnet sind.
- Die *Free-Page-List* enthält saubere Frames, die keine Zuordnung zu einem Prozess haben.
- Die *Zeroed-Page-List* enthält Frames, die bereits mit Nullen initialisiert wurden.

In Abbildung 7-40 sind die Auslagerungslisten der Windows-Speicherverwaltung sowie die Listenbewegungen skizziert.

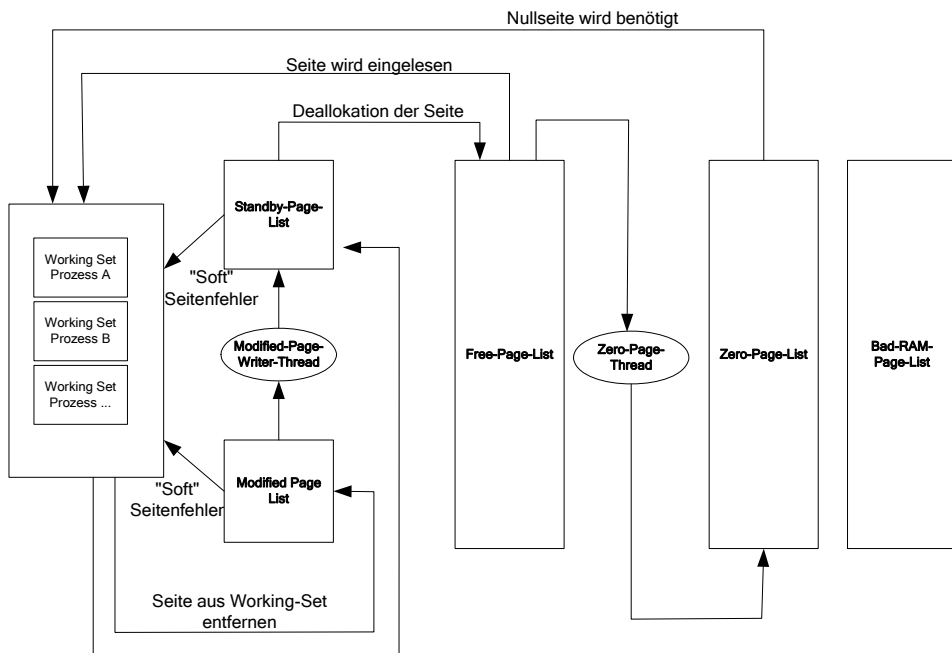


Abbildung 7-40: Auslagerungslisten der Windows-Speicherverwaltung

Es gibt einige Systemthreads, die unter dem Prozess mit dem symbolischen Namen *System*² ablaufen und für die Hauptspeicherverwaltung gewisse Aufgaben erledigen. Sie versuchen u.a. nach bestimmten Verfahren freien Speicher zu schaf-

² Siehe Task-Manager im Windows 2000 und folgende.

fen und sind für die Verwaltung der dargestellten Listen zuständig. U.a. sind dies folgende Threads (vgl. auch Abbildung 7-40):

- Ein zyklisch arbeitender *Working-Set-Manager-Thread* wird einmal pro Sekunde oder bei Bedarf aufgerufen, um den Arbeitsspeicher frei zu machen.
- Der *Swapper-Thread* läuft alle paar Sekunden, sucht nach Threads, die schon länger nichts tun und legt deren Frames in die Modified- oder Standby-Page-List.
- Der *Modified-Page-Writer-Thread* schreibt die Frames aus der Modified-Page-List in die Paging-Area, wenn diese Liste verkleinert werden muss.
- Der *Zero-Page-Thread* befüllt die Frames der Free-Page-List mit Nullen und hängt sie in die Zero-Page-List um.

Die Seitenrahmen des Hauptspeichers werden in der sog. *PFN-Datenbank* (Page Frame Number) verwaltet, wobei der Begriff „Datenbank“ hier etwas übertrieben ist. Es handelt sich um eine transiente Datenstruktur mit einer Tabelle (siehe Abbildung 5-41), die einen Eintrag je Seitenrahmen enthält. In der PFN-Datenbank werden die einzelnen Listen der Seitenersetzungsstrategie als verkettete lineare Listen verwaltet. Zu jeder Liste existiert ein Anker.

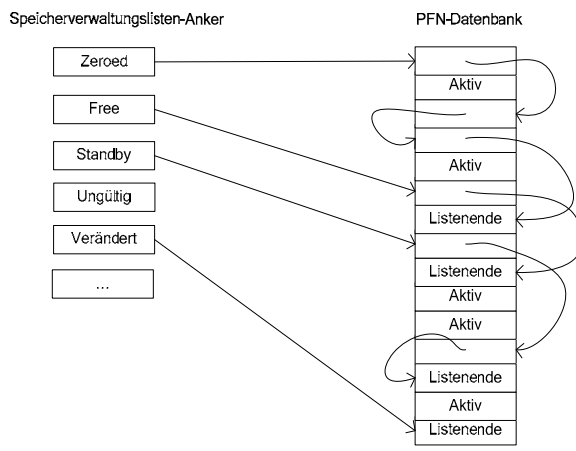


Abbildung 5-41: Aufbau der PFN-Datenbank (Solomon 2005)

Zu jedem Seitenrahmen wird ein Statusfeld verwaltet, wobei acht Zustände möglich sind. Hierzu gehören die Stati *Active* (valid), *Standby*, *Modified*, *Zeroed*, *Free*, *Bad* und *Modified-No-Write* und *Transition*. Der Zustand *Zeroed* gibt z.B. an, dass ein Speicherrahmen mit lauter Nullen „bereinigt“ ist und in der *Zero-Page-List* eingegliedert ist. Der Zustand *Free* deutet z.B. an, dass der Rahmen frei ist und in der

Free-Page-List liegt. In den PFN-Einträgen werden Verkettungsinformationen verwaltet, die die Zugehörigkeit zu den oben angegebenen Listen regeln.

Placement Policy. Für die Speicherbelegungsstrategie dienen Windows-eigene, nicht weiter beschriebene Verfahren. Die Buddy-Technik wird nicht explizit verwendet.

7.4 Übungsaufgaben

1. Welche grundlegenden Gedanken stecken hinter dem Konzept des virtuellen Speichers?
2. Welche Aufgabe hat eine Verdrängungsstrategie und was ist ein Seitenfehler?
3. Was bedeutet Demand-Paging und welche Alternative wäre denkbar?
4. Wie viele Seitentabellen müssen bei virtueller Adressierung vom Betriebssystem verwaltet werden, wenn mit einer einstufigen Seitentabelle gearbeitet wird und gerade 10 Prozesse aktiv sind?
5. Ein virtueller Adressraum wird mit 32 Bit langen virtuellen Adressen adressiert. Eine virtuelle Adresse enthält jeweils 10 Bit für den Index in der Top-Level-Seitentabelle und 10 Bit für den Index in der Second-Level Seitentabelle.
 - a) Wie viele Second-Level Seitentabellen gibt es maximal je Prozess?
 - b) Wie groß sind die Seitenrahmen im Hauptspeicher?
 - c) Wie groß ist der gesamte virtuelle Adressraum eines Prozesses?
6. Welcher Seitenersetzungsalgorithmus wäre theoretisch optimal und warum ist dieser praktisch nur schwer zu realisieren?
7. Warum ist der LRU-Algorithmus zur Seitenersetzung bei der virtuellen Adressierung schwer zu realisieren? Welcher Algorithmus ist eine gute Alternative hierzu?
8. Welche Probleme ergeben sich bei sehr großen virtuellen Adressräumen in Bezug auf die Verwaltung der Seitentabellen?
9. Welche Möglichkeiten gibt es, die hohe Belastung der Seitentabellenverwaltung insbesondere bei großen Adressräumen zu optimieren?
10. Erläutern Sie das Working-Set-Verfahren, das Windows einsetzt!
11. Erläutern Sie die Grundprinzipien des TLB und der invertierten Seitentabelle mit Hashing!
12. Was versteht man unter einem Shared Memory? Nennen Sie Einsatzmöglichkeiten!
13. Welche grundsätzlichen Speicherverwaltungsmechanismen sind in Unix-Systemen vorzufinden?
14. Das Betriebssystem Windows verwendet eine recht aufwändige Logik zur Verwaltung der Frames in der Speicherverwaltung. Frames werden in sog. Auslagerungslisten geführt.
 - a) Welche Auslagerungslisten gibt es und welche Typen von Frames werden dort verwaltet?

- b) Erläutern Sie, welche System-Threads bei der Verwaltung der Auslagerungslisten mitwirken!
15. Wozu dient die Buddy-Technik in der Speicherverwaltung und wie funktioniert sie?
16. Es seien 3 RAM-Seitenrahmen im Hauptspeicher verfügbar, der Rest der Seiten wird auf einer Paging-Area abgelegt. Folgende Referenzreihenfolge, in der die Seiten mit den Nummern 0 bis 6 angesprochen werden, sei gegeben:
0-1-2-3-4-0-1-5-6-0-1-2
- a) Skizzieren Sie den optimalen Algorithmus (Belady) für die Seitenersetzung und zählen Sie die Anzahl der Seitenersetzungen.
- b) Skizzieren Sie die Seitenersetzungen unter Nutzung des FIFO-Verfahrens und ermitteln Sie die Anzahl der Seitenersetzungen.
17. Es seien 3 RAM-Seitenrahmen im Hauptspeicher verfügbar, der Rest der Seiten wird auf einer Paging-Area abgelegt. Folgende Referenzreihenfolge, in der die Seiten mit den Nummern 1 bis 7 angesprochen werden, sei gegeben:
2-5-6-2-1-2-5-3-2-7-2
- Zeigen Sie, dass bei FIFO-Seitenersetzung 6, bei LRU 3 und bei der optimalen Strategie nach Belady 3 Seitenersetzungen notwendig sind, um die Referenzreihenfolge abzuarbeiten.

8 Geräte- und Dateiverwaltung

Dieses Kapitel befasst sich mit der Geräteverwaltung und mit der Dateiverwaltung sowie mit heute üblichen Storage-Systemen. Zunächst wird auf die Geräteverwaltung eingegangen. Es wird dargestellt, wie die Geräteansteuerung ins Betriebssystem eingebettet ist. Geräte sind heute üblicherweise über Bussysteme an die CPU angeschlossen. Daher wird auch ein kurzer Blick auf heutige Bussysteme geworfen. Ein klassischer Gerätetyp, die Festplatte, wird etwas intensiver betrachtet. Die Einbettung der Geräteansteuerung unter Unix wird kurz skizziert, und es wird rudimentär erläutert, was unter DMA und Memory-Mapped-Ein-/Ausgabe zu verstehen ist.

Anschließend wird auf die Dateiverwaltung in heutigen Betriebssystemen eingegangen. Dabei wird erläutert, wie Dateisysteme grundsätzlich aufgebaut sind. Anhand von Beispielen aus den Betriebssystemen Unix und Windows werden spezielle Dateisysteme wie UFS (ursprüngliches Unix-Dateisystem), NTFS (neueres Windows-Dateisystem) und FAT (älteres MS-DOS- und Windows-Dateisystem) skizziert.

Weiterhin wird in diesem Kapitel auf Storage-Systeme eingegangen. Die wichtigsten *RAID*-Varianten werden diskutiert, und es wird der Unterschied zwischen *Network Attached Storage* (NAS) und *Storage Attached Networks* (SAN) aufgezeigt.

Zielsetzung des Kapitels

Der Studierende sollte verstehen, wie die Geräteverwaltung und die Dateiverwaltung in Betriebssystemen grundsätzlich funktionieren und dies auch an Beispielen erläutern können. Anhand der Festplatte sollte er die Funktionsweise eines Geräts und dessen Anbindung an die CPU erläutern können. Spezielle Storage-Systeme sollte er benennen und prinzipiell erklären können.

Wichtige Begriffe

Gerätetreiber, Logischer Sektor, Zylinder, Spur, Dateisystem, i-Node, FAT, NTFS, UFS, RAID-Level, RAID-0, RAID-1, RAID-5, RAID-6, RAID 1+0, SAN, NAS.

8.1 Aufgaben und Überblick

8.1.1 Grundlegendes

Die Geräteverwaltung, auch Ein- und Ausgabeverwaltung oder E-/A-Management genannt, dient als Teil des Betriebssystems der optimierten Verwaltung von externen Geräten zur Ein- und Ausgabe. Zu den Geräten gehören u.a. Festplatten, Floppy-Disks, CD-ROM- und DVD-Laufwerke, die Tastatur, der Bildschirm und die Netzwerkanbindungen.

Die Leistungsfähigkeit eines Computersystems hängt nicht nur vom Prozessor ab, sondern auch von der Geschwindigkeit der Datenübertragung zwischen dem Prozessor, dem Hauptspeicher und den Ein- und Ausgabegeräten (Abbildung 8-1).

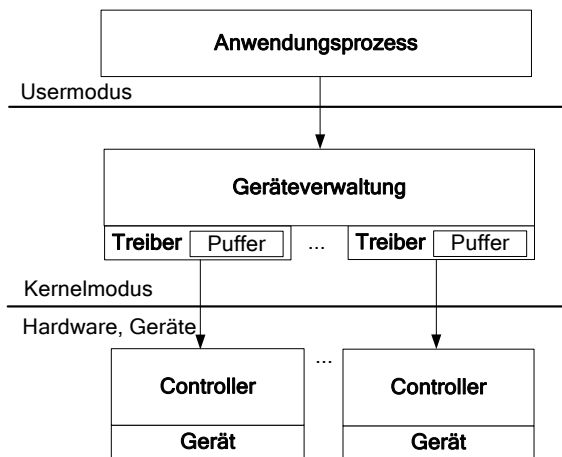


Abbildung 8-1: Schichtung der Ein-/Ausgabeverarbeitung

Die Geräteverwaltung stellt die Schnittstelle zwischen den physikalischen Geräten und dem Rest des Betriebssystems dar und hat die Aufgabe, die Geräte anzusteuern. Hierzu gehört die Ausgabe von Kommandos an externe Geräte, die Interruptbearbeitung und eine entsprechende Fehlerbehandlung. Die einzelnen Geräte sind von ihrer Funktionalität und ihrem Aufbau her sehr unterschiedlich.

Gerätespezifische Programmteile werden in eigenen Modulen bzw. Softwarekomponenten realisiert. Diese werden als *Gerätetreiber* bezeichnet und stellen gewissermaßen *virtuelle Maschinen* dar, die der Initialisierung und der Ausführung der Datenübertragung von und zu einem Gerät dienen. Treiber laufen in Multiuser-Betriebssystemen typischerweise im Kernelmodus ab. Sie verfügen über eine Treiberschnittstelle, deren Funktionalität vom Betriebssystem abhängt. Die Geräte-

verwaltung nutzt die Treiberschnittstellen aller Treiber des Betriebssystems für den Zugriff auf die entsprechenden Geräte-Controller.

Ein Treiber hat u.a. folgende Aufgaben zu erfüllen:

- Er definiert ein Gerät und macht es dem Betriebssystem bekannt, initialisiert also die Geräte-Controller beim Systemstart.
- Er stellt ein logisches Programmiermodell bereit und übersetzt es in geräte-spezifische Anforderungen.
- Er dient der Pufferung von Daten auf dem Weg vom Gerät zum Hauptspeicher und umgekehrt.
- Er übernimmt die Unterbrechungsbearbeitung für ein Gerät. In der Regel liegt im Treiber auch die passende Interrupt Service Routine (ISR) für das Gerät.
- Er dient der Koordination der nebenläufigen Zugriffe auf ein Gerät.

Es hängt vom jeweiligen Betriebssystem ab, ob Treiber statisch zum Systemstartzeitpunkt bekannt sein müssen oder während des Betriebs dynamisch installiert und gestartet bzw. angehalten werden können.

Die Anwendungsprogramme greifen üblicherweise über eine vordefinierte und standardisierte Schnittstelle mit Hilfe von Systemcalls auf die Geräteverwaltung zu. Typische Operationen sind *open*, *close*, *read* und *write*. Windows stellt z.B. eine einheitliche, generische Treiberschnittstelle namens WDM (Windows Driver Model) zur Verfügung. Über das WDF (Windows Driver Foundation) wird die Treiberprogrammierung unterstützt, u.a. auch die *Plug-and-Play*-Programmierung.

Die Schichtenarchitektur der Geräteverwaltung ist in Abbildung 8-1 dargestellt. Die Auftragsverwaltung kümmert sich um die anstehenden Zugriffsaufträge und läuft zusammen mit den Treibern im Kernelmodus ab. Das Gerät selbst verfügt meist über einen Controller, der als Kommunikationspartner eines Treibers zur Verfügung steht. Im Treiber wird meist ein Pufferspeicher (Cache) für lesende und schreibende Zugriffe verwaltet, um die Zugriffe zu beschleunigen. Ähnlich wie bei der Hauptspeicherverwaltung versucht man hier, Daten, die benötigt werden, im Puffer zu halten, und benutzt dazu verschiedene Strategien wie LRU usw.

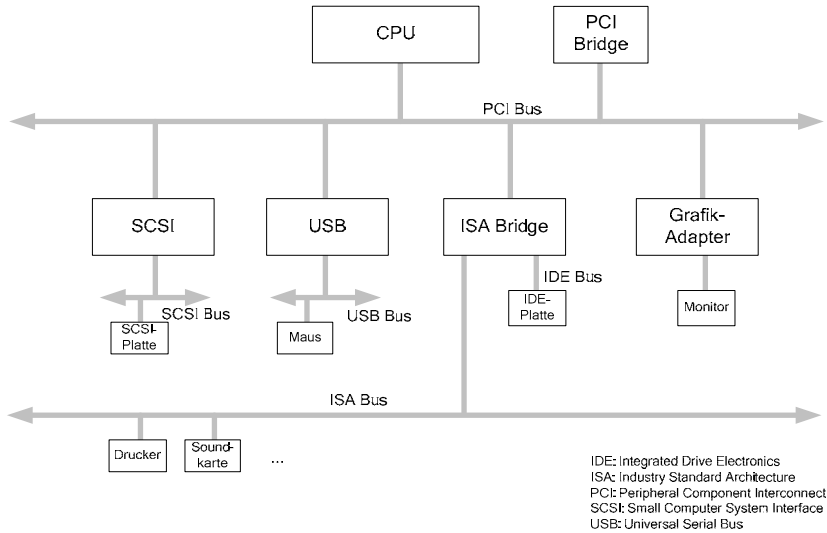


Abbildung 8-2: Typische PC-Bussysteme (Tanenbaum 2002)

Wie Abbildung 8-2 zeigt, werden Ein-/Ausgabe-Geräte bei gängigen PC-Systemen über verschiedene Bussysteme an die CPU angeschlossen. Typische Datenraten für die Übertragung zwischen CPU und Gerät bzw. Bussystem sind der Tabelle 8-1 zu entnehmen. PC-Systeme nutzen heute u.a. folgende Bussysteme:

- Der *PCI-Bus* bzw. dessen Vorgänger und der *ISA-Bus* für die Anbindung externer Geräte wie z.B. Festplatten. Der *IDE-Bus* dient zum Anschluss von sog. IDE-Plattengeräten bzw. CD-ROM-Laufwerken. Er hat seinen Ursprung im Platten-Controller-Interface des IBM PC/AT.
- Der *SCSI-Bus* dient zum Anschluss von schnellen SCSI-Platten und sonstigen schnellen SCSI-kompatiblen Geräten. SCSI steht für *Small Computer System Interface*. Bei SCSI unterscheidet man historisch SCSI-1, SCSI-2, Wide-SCSI, Fast-SCSI, Ultra-SCSI, Ultra-Wide-SCSI,..., Ultra-Wide-2-SCSI, usw. Um an einen Computer SCSI-Geräte anschließen zu können, wird ein SCSI-Hostadapter benötigt, der den Datentransfer auf dem SCSI-Bus kontrolliert. Das anzuschließende SCSI-Gerät besitzt einen SCSI-Controller, um die Daten über den Bus zu übertragen und mit dem Hostadapter zu kommunizieren.
- An den *USB¹* (*Universal Serial Bus*) werden serielle Geräte angeschlossen und dies ohne die Notwendigkeit, einen zusätzlichen Treiber zu installieren. USB

¹ USB ist eine sehr gute Unterstützung für das Plug&Play über das sog. „Hot Plugging“. Man kann, ohne zusätzliche Interrupts oder E/A-Adressen zu reservieren, neue Geräte im

ist der Nachfolger der sog. seriellen Schnittstelle bei PCs, allerdings wesentlich schneller und flexibler. In einem PC ist heute üblicherweise ein USB-Hostcontroller, der über standardisierte USB-Anschlüsse mehrere USB-Schnittstellen (bis zu 127) unterstützt, über die USB-Geräte angeschlossen werden können. An eine USB-Schnittstelle können wiederum USB-Hubs angeschlossen werden, an die sich mehrere USB-Geräte anschließen lassen. Heute werden alle möglichen Geräte über USB angeschlossen (USB-Festplatten, USB-Sticks, Digitalkameras, PDAs usw.).

Tabelle 8-1: Typische Übertragungsgeschwindigkeiten von Geräten und Bussystemen

Gerät	Transferrate [Bit/s] bzw. Datenrate [Byte/s]
Keyboard	ca. 10 Byte/s
Maus	ca. 100 Byte/s
Laserdrucker	ca. 100 KByte/s
Grafikkarte	2 GBit/s
USB1 (Universal Serial Bus)	1,5 (low speed) bis 12 (full speed) Mbit/s
USB2	480 Mbit/s (high speed)
USB3	5 GBit/s (super speed)
ISA-Bus	16 MByte/s
Fast Ethernet (100 Mbit/s)	12,5 MByte/s
XGA Monitor	ca. 60 MByte/s
SCSI-Bus	ca. 320 MByte/s
Ultrium Tape	ca. 320 MByte/s
PCI Bus	ca. 528 MByte/s
IDE-Festplatten mit ATA ²	ca. 130 Mbyte/s
Festplatten mit S-ATA 3.0 ³	ca. 6 GBit/s

laufenden Betrieb anstecken. Diese melden sich dann selbstständig bei der Geräteverwaltung an.

² ATA = *Advanced Technology Attachment with Packet Interface* ist ein Protokoll für die Anbindung von Massenspeicher in Rechnersystemen. ATA wird für die parallele Übertragung auf Basis der IDE-Schnittstelle in PCs eingesetzt (Integrated Device Electronics).

Man spricht bei PCs auf von einer *Northbridge* und einer *Southbridge*. Dies sind zwei Chips auf der Hauptplatine (Motherboard), welche die verschiedenen Komponenten miteinander verbinden. Die Northbridge ist näher an der CPU und verbindet diese auf Basis einer hohen Datenrate mit dem Hauptspeicher. Auch die Grafikkarte ist meist aufgrund der hohen Anforderungen an die Datenrate an die Northbridge angebunden. Die Southbridge dient der Anbindung von Geräten über die verschiedenen Bussysteme. South- und Northbridge sind meist über den PCI-Bus miteinander verbunden. Beide Chips bilden den *Chipsatz*.

8.1.2 Gerätearten

Bei Geräten (Devices) unterscheidet man *blockorientierte*, *zeichenorientierte* und *sonstige* Geräte. Zeichenorientierte Geräte sind z.B. die Maus, eine Netzwerkkarte oder ein Drucker. Die Information zwischen der CPU und dem Gerät wird in einzelnen Zeichen ausgetauscht und einzelne Zeichen sind nicht adressierbar. Vielmehr erfolgt das Lesen und Schreiben in Zeichenströmen (Streams). Bei blockorientierten Geräten, wie Festplatten oder Bandlaufwerken, kann direkt auf Blöcke (Größe z.B. 512 – 32.768 Byte) positioniert bzw. zugegriffen werden. Blöcke sind also direkt adressierbar. Zu den sonstigen Geräten gehören z.B. Clocks (Systemuhren). Exemplarisch soll als Geräteart die Magnet- oder Festplatte näher betrachtet werden.

Festplatten. Magnet- oder Festplatten sind heute das gängige Speichermedium. Sie bestehen aus Kunststoff- oder Aluminiumscheiben, die mit einer hauchdünnen Magnetschicht überzogen sind, und meistens sind auch mehrere Schichten übereinander angeordnet. Eine beidseitige Plattenbeschichtung ist heute der Normalfall. Die Schreib-/Leseköpfe sind auf einem mechanischen Träger befestigt und mit kleinen Elektromagneten ausgestattet. Je Oberfläche ist ein Schreib-/Lesekopf notwendig. Meistens ist zu einer Zeit nur ein Schreib-/Lesekopf aktiv. Auf den Plattenoberflächen sind Spuren angebracht (siehe Abbildung 8-3). Die Platten rotieren mit einer konstanten Drehzahl von 7000 bis 15000 Umdrehungen/min um eine senkrechte Drehachse.

³ S-ATA = *Serial ATA*, Nachfolger von ATA auf Basis serieller Übertragung, weniger Störanfällig und schneller.

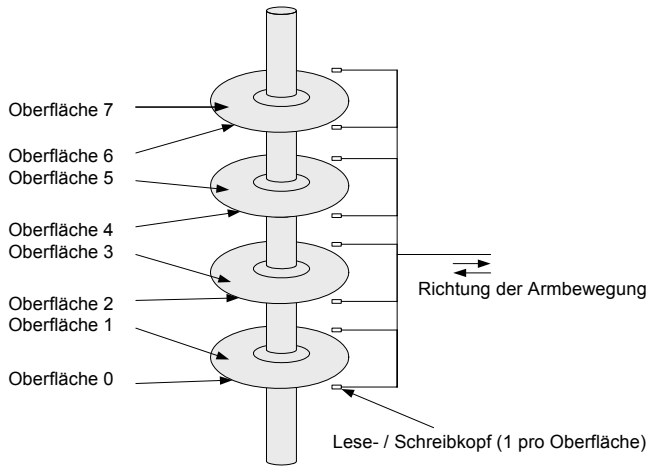


Abbildung 8-3: Aufbau einer Festplatte

Physikalische und logische Sektoren. Die Daten sind als Blöcke fester Größe in sog. Sektoren gespeichert. Die Plattenoberfläche ist physikalisch in kleine Kuchenstücke, sog. Kreissektoren aufgeteilt. Die Spurenanteile eines Kreissektors werden als logische Sektoren bezeichnet. Alle Spuren untereinander nennt man Zylinder. Zylinder sind über eine Position der Schreib-/Leseköpfe ansprechbar. Der übliche Plattendurchmesser liegt heute meist bei 3,25 Zoll, 2,5 Zoll oder kleiner. Es werden zwischen zwei und acht Platten übereinander gestapelt. Eine typische Platte verfügt weit über 135.000 Spuren pro Zoll.

Die Lokalisierung von Sektoren erfolgt über vier Koordinaten, die über ein Quadrupel der Form (g,c,h,s) adressiert werden, wobei g das Laufwerk, c die Zylindernummer, h die Nummer des Schreib-/Lesekopfes und s die Sektornummer ist. Vom Betriebssystem werden nur die logischen Sektoren in durchnummerierter Reihenfolge adressiert. Das Mapping auf die Koordinaten wird durch den Geräte-Controller durchgeführt. Die logischen Sektoren werden auch als *LSN* bezeichnet (Logical Sector Number).

Die Kapazität hängt von der Anzahl der Plattenoberflächen, der Anzahl an Spuren und der Größe der Spuren ab. Bei 39 Sektoren, 761 Zylindern, 8 Köpfen und 512 Byte großen Sektoren ergibt sich beispielsweise eine Kapazität von $39 \times 761 \times 8 \times 512 = 121.565.184$ Byte.

Abbildung 8-4 zeigt den Sektorenaufbau am Beispiel zweier Sektoren auf einer Festplattenspur. Neben einer Präambel mit Verwaltungsinformation enthält ein Sektor Nutzdaten (hier 4096 Bit) und einen ECC (Error Correction Code). Die For-

matierung der Spuren, d.h. die Belegung mit einzelnen Sektoren und der entsprechenden Verwaltungsinformationen wird über den Platten-Controller vorgenommen.

Das klassische konzentrische Aufzeichnungsverfahren hat den Nachteil, dass die innerste Spur mit der kleinsten Aufzeichnungslänge das Maß für den Sektor vorgibt. Die außen angeordneten Spuren verschwenden unnötig viel Platz. Neuere Verfahren benutzen daher zur Lösung des Problems eine unterschiedliche Anzahl von Kreissektoren. Es werden Gruppen benachbarter Ringe gebildet, die als Zonen bezeichnet werden.

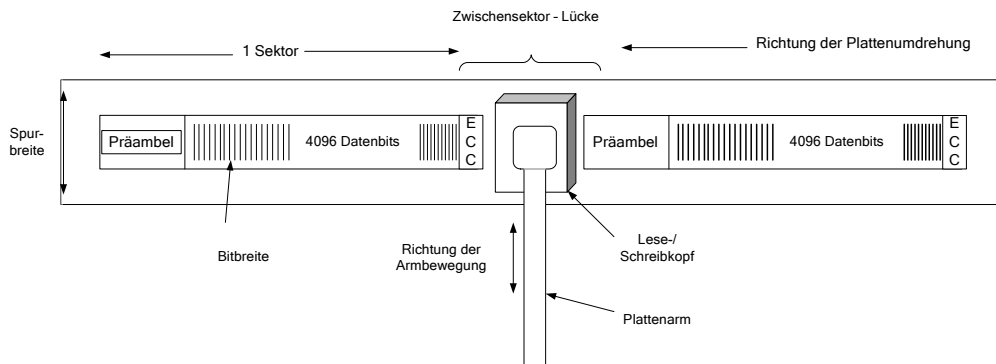


Abbildung 8-4: Aufbau der Information auf einer Festplattenspur (Tanenbaum 2002)

Die Durchnummerierung der logischen Sektoren erfolgt, wie in Abbildung 8-5 zu sehen ist, nach einem einfachen Schema entlang der Spur und nicht – wie man annehmen könnte – entlang der Zylinder. Der Grund dafür ist darin zu sehen, dass eine Verschiebung entlang der Spur elektronisch abgewickelt wird und daher schneller vonstatten geht als der mechanische Wechsel zum nächsten Zylinder. Die Abbildung zeigt zwei aufeinander folgende Plattenseitenoberflächen. Im Sektor 1 der Seite 0, also im Zylinder 0, der äußeren Spur liegt der erste logische Sektor mit der Nummer 0. Die Folgeseite 1 enthält dann erst den Sektor 8.

Bei optischen Plattenspeichern wie CD-ROM und DVD verwendet man eine andere Aufzeichnungsgeometrie. Auf der Oberfläche ist nämlich eine Aufzeichnungsspirale, und es sind keine einzelnen Spuren vorhanden.

Der Zugriff auf einen Sektor dauert im Vergleich zur CPU-Geschwindigkeit sehr lange und ergibt sich aus einer evtl. Wartezeit im Betriebssystem, der Suchzeit, der Drehwartezeit und der Transferzeit zur Übertragung der Daten in den Hauptspeicher bzw. vom Hauptspeicher auf die Platte. Die Suchzeit ist die Zeit, bis die Spur

gefunden wird. Die Drehwartezeit ist die Positionierzeit innerhalb der Spur und ist im Mittel die Zeit für eine halbe Plattenumdrehung.

Für den Zugriff auf die Festplatte verwendet man verschiedene Strategien. Ein Plattentreiber muss die Lese- und Schreibaufträge möglichst optimal und fair bearbeiten. Mit der Strategie *FCFS* (First Come First Serve) werden z.B. alle Schreib- und Leseaufträge in der Reihenfolge ausgeführt, in der sie ankommen, was recht einfach zu implementieren und auch fair ist. *SSTF* (Shortest Seek Time First) versucht dagegen, Aufträge in näherer Umgebung der aktuellen Schreib-/Lesekopf-Position auszuführen, um Kopfbewegungen zu minimieren. Wie bei der Prozessverwaltung mit SJFS besteht auch hier das Problem des Verhungerns von Aufträgen.

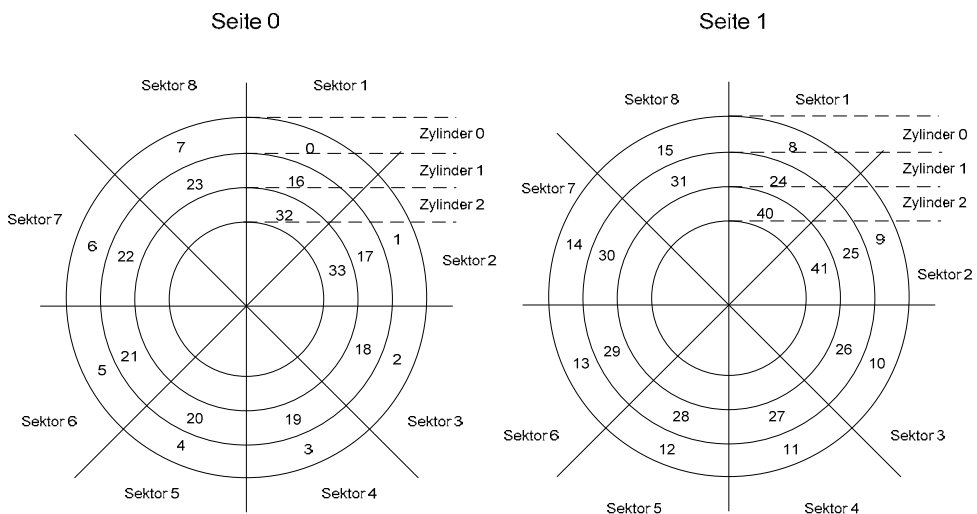


Abbildung 8-5: Logische Blocknummerierung auf einer Festplatte (Glatz 2006)

8.1.3 Geräteanbindung unter Unix

Geräte (Drucker, Terminals, Maus etc.) werden unter Unix logisch wie Dateien, als sog. *Special Files* behandelt. Hierfür gibt es spezielle Systemverzeichnisse wie `/dev`. Genauere Bezeichnungen sind vom jeweiligen Unix-Derivat abhängig. Jeder Gerätetreiber bietet eine Dienstschnittstelle mit Operationen wie *open*, *read*, *write* und *close* (Systemcalls) an.

Bei den Special Files unterscheidet man sog. *Block Special Files*, wie z.B. `/dev/hd1` (Festplatte) und *Character Special Files*, z.B. `/dev/tty` (Bildschirm, Tastatur).

Man kann die Special Files auch in Kommandos nutzen. Beispielsweise kann man mit dem Kommando `cp file /dev/lp` eine Datei namens *file* direkt auf den Drucker senden.

In Abbildung 8-6 ist anhand eines Character- und eines Block-Device (Gerätes) in Unix System V dargestellt, wie die Einbettung im Betriebssystem erfolgt. Man bezeichnet die Architektur auch als Streams-orientierte Architektur, wobei ein Stream eine Abstraktion eines Datenflusses ist. Die Ein- und Ausgabe wird in einem Datenstrom (Stream) abgewickelt. Treiberschichten (Filter) können in den Stream eingebaut werden.

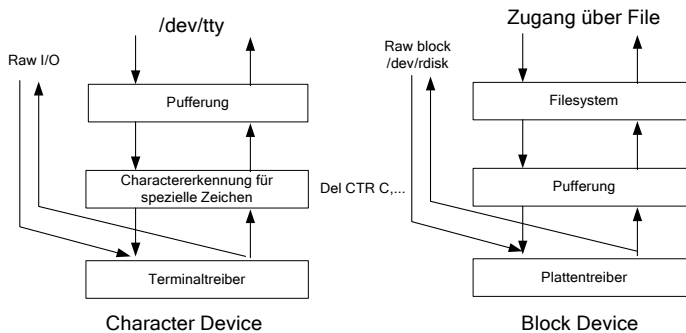


Abbildung 8-6: Geräteeinbindung unter Unix (Brause 2001)

Wie in Abbildung 8-6 zu sehen ist, ist im Stream des Character-Device ein spezieller Filter zur Interpretation und Eliminierung von Steuerzeichen wie CTRL-C eingebaut. Beim Stream des dargestellten Block-Device gibt es einen Puffer-Filter, der vor das Dateisystem geschaltet ist. Der Zugriff eines Anwendungsprogramms erfolgt normalerweise am oberen Ende des Streams, aber man kann auch direkt auf das „Raw-Device“ zugreifen und nutzt dafür einen eigenen Dateinamen aus dem Verzeichnis `/dev`. Diese Option nutzen z.B. Datenbanksysteme, um die Pufferung zu umgehen. Auch Netzwerkverbindungen kann man auf diese Weise bearbeiten, wofür es eine weitere Schnittstelle gibt, die hier nicht behandelt werden soll (Socket-Interface).

8.1.4 Memory-Mapped Ein-/Ausgabe und DMA

Die Kommunikation der CPU mit den Geräten wird bei heutigen Systembus-orientierten Rechnerarchitekturen über „Memory-Mapped-Ein-/Ausgabe“ durchgeführt. Man bildet dabei die Kontrollregister sowie die Ein-/Ausgabe-Pufferbereiche der Geräte-Controller (E/A-Controller) auf Adressen im virtuellen

Adressraum ab. Diese Adressen werden als E/A-Adressen bezeichnet. Jedem Gerät wird eine bestimmte E/A-Adresse zugewiesen.

Die E/A-Adressen der einzelnen Geräte dürfen nicht miteinander kollidieren, sonst gibt es Konfliktsituationen, die zum Absturz des Systems führen können. Daher werden die Adressen heute z.B. unter Windows automatisch vergeben, und es erfolgt eine Überwachung auf Adresskonflikte durch das System.

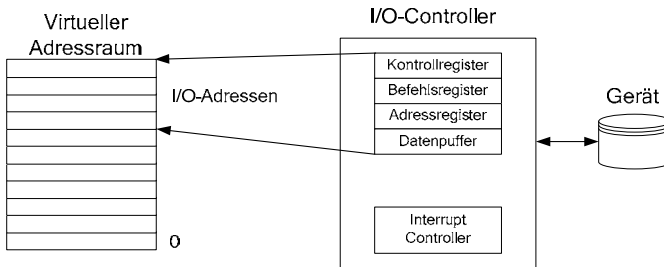


Abbildung 8-7: Memory-Mapping (Brause 2001)

In Abbildung 8-7 ist das Grundprinzip des Memory-Mapping skizziert. Der Teil des Adressraums, auf den die E/A-Adressen abgebildet werden, liegt üblicherweise im geschützten Kernel-Adressraum. Im Intel-Pentium-Prozessor werden z.B. die Puffer auf den Adressbereich zwischen 640 KByte und 1 MByte abgebildet, während die E/A-Ports für die Kontrollregister zwischen 0 und 64 KByte angesiedelt sind.

Die Adressierung der Geräteadressen funktioniert nun wie folgt: Wenn die CPU ein Kontrollregister oder einen Datenpuffer eines Gerätes adressieren möchte, legt sie die Adresse wie eine normale Hauptspeicheradresse auf den Adressbus. Über eine zusätzliche Signalleitung wird angegeben, ob es sich um eine Hauptspeicheradresse oder um eine Geräteadresse handelt. Im zweiten Fall wird die Anfrage von dem zuständigen Gerät beantwortet.

Unabhängig von der Memory-Mapped-E/A-Technik wird für den schnellen Zugriff der Geräte auf den Hauptspeicher der sog. *Direct Memory Access (DMA)* verwendet. Mit DMA werden ganze Datenblöcke zwischen dem Geräte-Controller und dem Hauptspeicher transportiert. Der Zugriff erfolgt über sog. *DMA-Controller* mit speziellen *DMA-Chips* (Hardwareunterstützung). DMA-Zugriffe auf den Geräte-Controller können parallel zur CPU-Bearbeitung ausgeführt werden. Die CPU stößt die Übertragung an und kann dann eine andere Aufgabe erledigen. Der DMA-Controller erzeugt bei Ende der Übertragung eine Unterbrechung an die

CPU. Es gibt DMA-Controller mit unterschiedlicher Funktionalität. Die einfachen können nur einen Zugriff zu einer Zeit ausführen, die intelligenteren führen dagegen mehrere Zugriffe parallel aus.

8.2 Dateiverwaltung

8.2.1 Allgemeines

Persistent, also dauerhaft zu verwaltende Daten werden üblicherweise in Dateien (Files) gespeichert, die auf externen Medien, meist auf Magnetplatten liegen. Eine Datei ist ein abstrakter Mechanismus zur Speicherung und zum Finden von Informationen. Der Teil des Betriebssystems, der sich mit Dateien und deren Organisation befasst, wird als Dateiverwaltung bezeichnet. Die Dateiverwaltung liegt von der Schichtung her über der Geräteverwaltung und nutzt die Dienste der Geräteverwaltung zum Zugriff auf die Hardware. Abbildung 8-8 zeigt die Einbindung der Dateiverwaltung in das Betriebssystem. Wie man sieht, ist die Dateiverwaltung als Schicht über der Geräteverwaltung eingebunden. Sie stellt die Dienste zur Verwaltung von Dateien zur Verfügung und nutzt intern die Dienste der Geräteverwaltung.

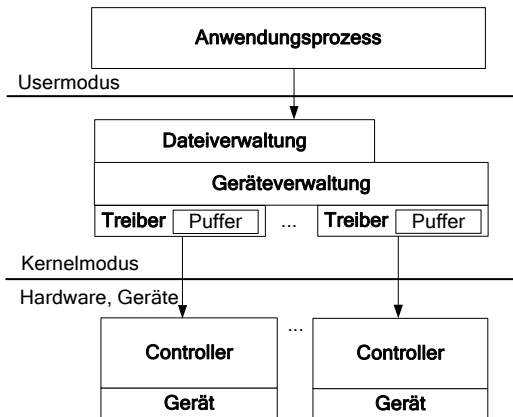


Abbildung 8-8: Dateiverwaltung als eigene Schicht

Zu jeder Datei sind bestimmte Informationen notwendig, die in speziellen Datenstrukturen verwaltet werden. Dateien werden in sog. *Dateisystemen* (*file system*) verwaltet. Dateisysteme werden heute hierarchisch gemäß Abbildung 8-9 organisiert, wobei Gruppen von Dateien in Dateiverzeichnissen bzw. Katalogen (*directory*) zusammengefasst werden können. Zu jedem Verzeichnis wird Verwaltungsinformation gepflegt. Durch die hierarchische Anordnung entsteht ein (umgedreh-

ter) Baum. Jedes Verzeichnis kann wieder beliebig viele Unterverzeichnisse (subdirectories) oder als Endknoten die Dateien selbst enthalten.

Unix verfügt z.B. über ein systemweites Dateisystem, während im Betriebssystem MS-DOS für jedes Gerät (Floppy-Laufwerk, Festplatte) ein unabhängiges Directory verwaltet wurde. Hierarchische Verzeichnissysteme beliebiger Tiefe sind heute „State of the Art“.

Dateien haben einen Dateinamen und einen Inhalt. Jede Datei wird in einer Datenstruktur mit folgenden Informationen verwaltet: Name der Datei, Typ der Datei (Normal- oder Katalogdatei), Länge der Datei in Byte, Blöcke, aus denen die Datei besteht, Zugriffsrechte zu dieser Datei, Passwörter zum Schutz der Datei, Statistikdaten, wie z.B. das Datum der letzten Änderung usw.

Für die Bearbeitung von Dateien und Verzeichnissen stellen Betriebssysteme bestimmte Systemdienste über Systemcalls zur Verfügung. Hierzu gehören:

- Erzeugen eines Katalogeintrags (directory)
- Löschen eines Katalogeintrags (inkl. des Dateiinhalts)
- Kopieren von Dateien
- Löschen einer Datei
- Ändern des Dateinamens oder eines Katalogs
- Übertragen eines Katalogeintrags in einen anderen Katalog

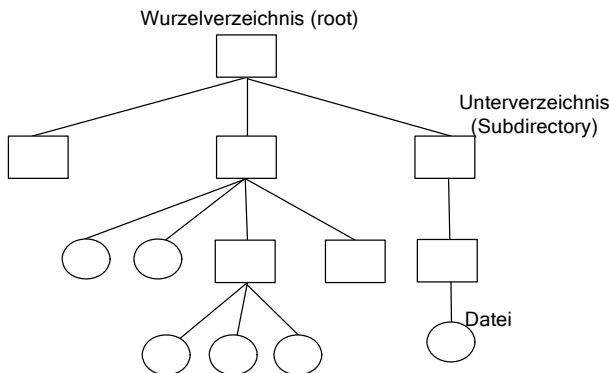


Abbildung 8-9: Hierarchisch organisiertes Dateisystem

Die Bearbeitung von Dateien ist grundsätzlich block- oder zeichenweise möglich. Für jede gefüllte Datei wird eine Liste mit von ihr belegten Blöcken auf dem externen Medium geführt. Der Katalog enthält Hinweise auf die Blöcke, aus denen die Dateien bestehen. Außerdem verwaltet das Dateisystem zudem eine *Pseudodatei* bestehend aus allen freien Blöcken jedes Datenträgers und eine Pseudodatei mit einer Liste aller unzuverlässigen Blöcke.

8.2.2 Fallbeispiel: Dateisysteme unter Unix

Im klassischen Unix erfolgt die Dateiverwaltung in einem beliebig tiefen, hierarchischen Dateisystem. Das klassische Unix-Dateisystem wird auch als UFS-Dateisystem) bezeichnet (Unix File System). Wie in Abbildung 8-10 dargestellt, heißt die Wurzel des Baums im Betriebssystem Unix „*root*“ oder das *Root-Directory* und wird mit „/“ abgekürzt. Für die Organisation des Unix-Baumes haben sich gewisse Konventionen eingebürgert. Unter dem *root*-Verzeichnis befinden sich üblicherweise die *Home-Directories* der einzelnen User z.B. unter */usr*, Programme unter */bin*, Konfigurationsdateien unter */etc* sowie Libraries unter */lib*. Darunter befinden sich dann weitere Subdirectories oder Dateien.

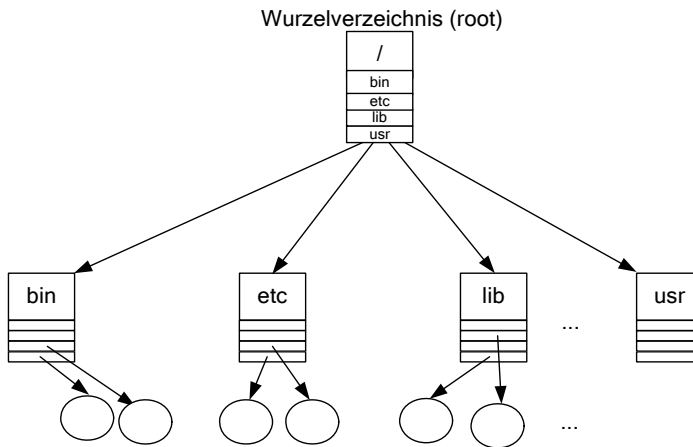


Abbildung 8-10 Dateisystem-Layout unter Unix

Die Namen der Verzeichnisse und Dateien stehen neben weiteren Informationen in den Verzeichnissen. Im Dateisystem kann über das Kommando *cd* (change directory) traversiert werden. Mit dem Kommando *mkdir* (make directory) wird ein neues Verzeichnis angelegt, und mit *rmdir* (remove directory) wird es gelöscht. Mit dem Kommando *pwd* (print working directory) kann das aktuelle Verzeichnis erfragt werden.

Externe Datenträger wie CDROM- oder Floppy-Laufwerke können unabhängige Dateisysteme enthalten. Diese kann man mit dem Unix-Befehl *mount* an beliebiger Stelle in den Unix-Dateibaum einklinken. Danach gibt es keine logische Unterscheidung mehr zwischen Dateien auf der Festplatte und anderen Datenträgern. Mit dem Befehl *umount* wird ein „gemountetes“ Verzeichnis wieder entfernt.

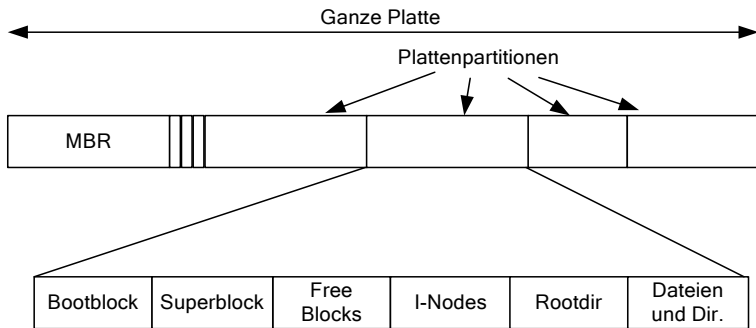


Abbildung 8-11: Dateisystem-Layout unter Unix

Dateisystem-Layout. Ein Dateisystem hat unter Unix ein vorgegebenes Layout (vgl. hierzu Abbildung 8-11 nach Tanenbaum 2002), das die folgenden Bestandteile enthält:

- *MBR* = Master Boot Record auf Sektor 0 zum Booten des Rechnersystems.
- Am Ende des MBR liegt die *Partition Table*: Eine Platte kann i.d.R. in mehrere Partitionen eingeteilt werden.
- *Bootblock* wird beim Hochfahren gelesen und ausgeführt.
- *Superblock* enthält Verwaltungsinformationen zum Dateisystem (Anzahl der Blöcke,...).
- *Free Blocks* (z.B. Bitmap) gibt die freien Blöcke des Dateisystems an.
- *Rootdir* enthält den Inhalt des Dateisystems (nach der Wurzel).

Heute gibt es natürlich in den verschiedenen Unix-Derivaten schon weiterentwickelte Dateisysteme, die beispielsweise bei einem Ausfall eines Rechnersystems nach dem Neustart ein Recovery durchführen können (siehe hierzu z.B. das Dynamic File System für Solaris 10 (DFS) von Sun).

i-Nodes. *i-Nodes* sind Einträge im Inhaltsverzeichnis des Dateisystems. Die sog. *i-Nodes*-Struktur wird je Datei verwaltet und ermöglicht eine flexible Zuordnung von Dateispeicher (Blöcken) zu Dateien. Hierüber sind alle Blöcke, die eine Datei hat, im Zugriff. Abbildung 8-12 zeigt die *i-Node*-Struktur eines typischen Unix-Systems.

Im *i-Node* einer Datei sind max. 13 Verweise verfügbar. Die ersten 10 sind direkte Verweise, die Verweise 11 bis 13 sind indirekt und können wiederum eine oder mehrere Verweisebenen bilden. Diese indirekten Verweise zeigen also auf Listen von Verweisen und diese evtl. wieder auf Verweislisten usw. bis die letzte Ebene schließlich auf die eigentlichen Blöcke zeigt. Kleinere Dateien benötigen also evtl. nur die ersten 10 Verweise, während große Dateien mehrstufige Verweishierarchien nutzen.

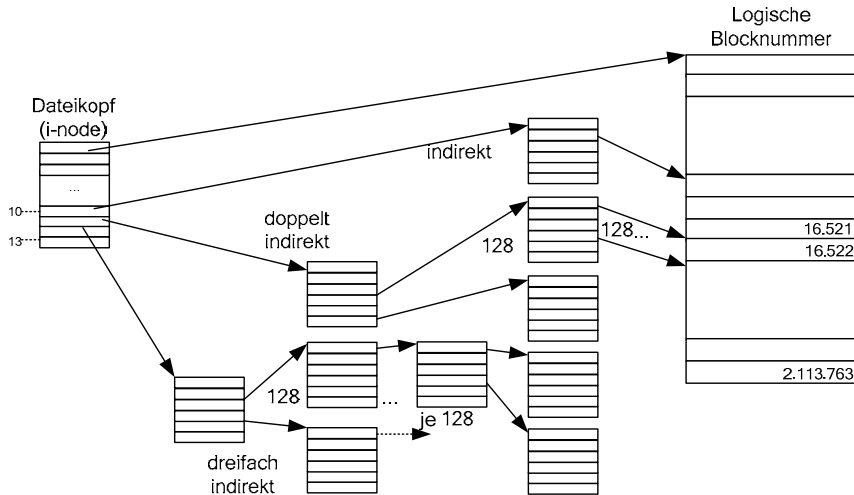


Abbildung 8-12: i-Node-Struktur unter Unix

Netzwerkdateisysteme. Ebenso existieren seit langem netzwerkfähige Dateisysteme, bei denen es möglich ist, transparent für den Anwender ein logisch zusammengehöriges Dateisystem auf mehrere Rechner zu verteilen. Das bekannteste Dateisystem dieser Art ist das Network File System (NFS). Über einen speziellen „Mount-Befehl“ können in einem lokalen Verzeichnis entfernte Verzeichnisse eingehängt werden. Beim Traversieren durch den Dateisystem-Baum erkennt man dann nicht, wo die Dateien eigentlich liegen. Verteilte Dateisysteme nutzt man heute bei Datei-Servern, welche gemeinsam genutzte Dateien verwalten.

8.2.3 Fallbeispiel: Dateisysteme unter Windows

Windows unterstützt eine Reihe von Dateisystem-Typen, was zum Teil historisch begründet ist. Hierzu gehören:

- CDFS (CD-ROM File System) ist ein einfaches Read-Only-Format für CD-ROMs. Windows implementiert den ISO-9660-Standard. Die Länge von Verzeichnis- und Dateinamen ist auf 31 Zeichen begrenzt. Es wird nicht zwischen Groß- und Kleinschreibung bei Datei- und Verzeichnisnamen unterschieden. Die Hierarchie der Verzeichnisse ist auf acht Ebenen limitiert. Dieses Format ist aber schon veraltet. Es hat sich das UDF-Format durchgesetzt.
- UDF (Universal Disk Format) ist ISO-13346-konform. Es ersetzt CDFS und ist in der DVD-Spezifikation enthalten. Es unterstützt maximale Dateinamenlängen von 255 Zeichen, eine maximale Pfadlänge von 1023 Zeichen und Groß- und Kleinschreibung bei Datei- und Verzeichnisnamen.

- FAT-16 (FAT = File Allocation Table) ist das alte MS-DOS-Dateisystem und nutzt 16-Bit-Plattenadressen.
- FAT-32 ist ebenfalls ein altes Dateisystem und nutzt 32-Bit-Plattenadressen.
- NTFS (NT Dateisystem) ist ein modernes, hierarchisches Dateisystem. NTFS nutzt 64-Bit-Plattenadressen.

Das x bei FAT-x gibt die Anzahl der Bits an, die das Dateisystem für die Identifikation eines Dateisystemblocks, *Cluster* genannt, verwendet. Ein Cluster entspricht einem oder mehreren logischen Sektoren auf dem Datenträger. In Abbildung 8-13 ist z.B. ein Plattenstapel mit einem Cluster skizziert, der sich über vier Sektoren erstreckt.

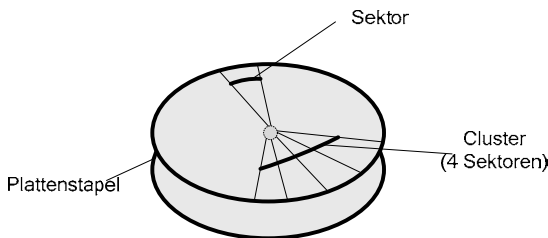


Abbildung 8-13: Sektoren und Cluster einer Festplatte

Welche Clustergröße bei den FAT-x-Formaten verwendet werden, hängt von der Formatierung des Datenträgers ab. Bei FAT-16 werden z.B. Datenträger mit einer Größe bis zu 32 MB standardmäßig mit einer Clustergröße von 512 Byte formatiert, Datenträger zwischen 33 und 64 MB mit einer Clustergröße von 1 KB usw. Im Formatierungsbefehl *format* kann der Standardwert verändert werden.

Die einzelnen Cluster, die einer Datei zugeordnet sind, sind über sog. *Dateizuordnungsketten* miteinander verkettet. Im Dateiverzeichniseintrag steht der Anker für die jeweilige Datei.

FAT-12. In FAT-12 können demnach nur maximal 2^{12} , also 4.096 Cluster adressiert werden. Als Clustergrößen werden in FAT-12 zwischen 512 Byte und 8 KB (dies entspricht auch der Sektorgröße auf einer Festplatte) unterstützt, was eine maximale Dateisystemgröße von 32 MB zur Folge hat.

Berechnung der maximalen Dateisystemgröße bei FAT-12:

$$8.196 * 4.096 = 33.554.432 \text{ Byte} = 32 \text{ MB}$$

FAT-12 wird vorwiegend für Disketten-Dateisysteme eingesetzt.

FAT-16. FAT-16 unterstützt schon maximal 2^{16} , also 65.536 Cluster mit einer Größe zwischen 512 Byte und 64 KB. Damit lassen sich Dateisysteme mit einer Maximalgröße von 4 GB (genau 4.294.960.000 oder $4,29496 * 10^9$ Byte) verwalten.

Berechnung der maximalen Dateisystemgröße bei FAT-16:

$$65.536 * 65.536 = 2^{16} * 2^{16} = 2^{32} = 4.294.960.000 = 4,29496 * 10^9 \text{ Byte} = 4 \text{ GB}$$

FAT-32. FAT-32 verwendet 32-Bit lange Cluster-Identifikatoren, nutzt aber tatsächlich nur 28 Bit aus. Die maximale Clustergröße bei FAT-32 ist 32 KB, was eine maximale Dateisystemgröße von 8 TB ermöglicht. Allerdings wird die Größe auf 32 GB begrenzt.

Berechnung der maximalen Dateisystemgröße bei FAT-32:

$$2^{15} * 2^{28} = 2^{43} = 8,79609 * 10^{12} = 8.796.090.000.000 \text{ Byte} = 8 \text{ TB}$$

Unter Windows kann man auf einer Festplatte auch unterschiedliche Dateisystem-Typen (FAT und NTFS) konfigurieren. FAT-Dateisysteme werden nur noch aus Kompatibilitätsgründen zu älteren Windows-Derivaten unterstützt. NTFS ist das eigentliche Dateisystemformat und soll daher auch etwas ausführlicher betrachtet werden.

NTFS. Bei NTFS werden 64 Bit für den Clusterindex bereitgestellt. Damit lassen sich maximal bis zu 16 Exabyte = 16 Milliarden GB an Clustern adressieren. Allerdings wird auch hier eine Begrenzung auf 128 TB festgelegt. Auch bei NTFS gibt es Standardwerte für die Vergabe der Clustergröße. Bis zu 512 MB große Datenträger erhalten eine Clustergröße von 512 Byte. Datenträger, die größer als 2048 MB sind erhalten eine Clustergröße von 4 KB.

NTFS unterstützt zudem u.a. folgende Mechanismen:

- Wiederherstellbarkeit von Dateisystemen bei Systemausfall
- Redundante Datenspeicherung

Wiederherstellbarkeit von Dateisystemen bedeutet, dass bei Systemausfall die Verzeichnisstruktur eines NTFS-Dateisystems konsistent wieder hergestellt werden kann. NTFS nutzt hierzu sog. *atomare Transaktionen* für alle Operationen, die Verzeichnisstrukturen verändern. Im Falle eines Systemabsturzes wird, ähnlich wie bei Datenbanksystemen, ein Rollback auf den letzten konsistenten Stand durchgeführt. Nur Operationen, die vor dem Ausfall bereits vollständig abgearbeitet wurden, sind wirksam, alle anderen werden zurückgesetzt.

Die redundante Datenspeicherung für wichtige Dateisysteminformationen dient ebenfalls der Ausfallsicherheit. Wird ein Sektor auf dem Datenträger beschädigt, kann man bei NTFS trotzdem auf die kritischen Systemdateien zugreifen. Dies geschieht über die Unterstützung von *RAID* (Erläuterungen zu RAID kommen noch weiter unten) direkt im E/A-Manager (Softwarelösung). Bei Auftreten eines Fehlers (bad sector) werden die Daten von einer Spiegelplatte automatisch rekonstruiert.

Die logische Struktur (logische Partitionierung) eines NTFS-Dateisystems wird beim *Formatieren* erzeugt. Eine Festplatte kann also einen oder mehrere Datenträger enthalten, wobei, wie oben bereits erwähnt, neben NTFS- auch FAT-Datenträger möglich sind. Beispielkonfigurationen sind in Abbildung 8-14 dargestellt.

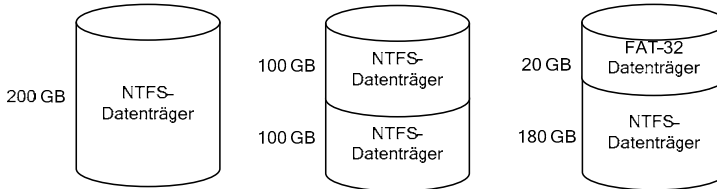


Abbildung 8-14: NTFS-Partitionierungsbeispiele

Die Größe eines Clusters für ein NTFS-Dateisystem wird auch als Clusterfaktor bezeichnet und beim *Formatieren* festgelegt. Er ist immer eine 2er-Potenz und eine Anzahl von physischen Sektoren. Der Clusterfaktor entspricht der Anzahl an Byte, die ein Cluster enthält (512 Byte, 1024 Byte, ...). Zudem wird jedem Cluster eine logische Clusternummer zugeordnet, die als *LCN* (Logical Cluster Number) bezeichnet wird. Die LCN nummeriert die Cluster eines Datenträgers vom Anfang bis zum Ende durch. Über die LCN werden physische Orte auf dem Datenträger referenziert. Die Abbildung der LCN auf den physischen Ort rechnet NTFS einfach über den Clusterfaktor aus:

$$\text{Physische Adresse} = \text{LCN} * \text{Clusterfaktor}$$

Innerhalb einer Datei wird ebenfalls eine Cluster-Nummerierung vorgenommen, wobei die sog. VCN (Virtual Cluster Number) verwendet wird. Über die VCN wird für jede Datei eine Durchnummerierung der zugeordneten Cluster durchgeführt. VCNs werden auf LCNs abgebildet.

Jedes Volume (Bezeichnung für logische Festplatte) wird in NTFS durch eine *Master File Table (MFT)*⁴ beschrieben. Die MFT liegt auch in einer Datei. Jeder Eintrag in der MFT ist 1 KB groß, enthält einen Datensatz (*Record*) und beschreibt eine Datei oder ein Directory mit Attributen (Name, Zeitstempel für den letzten Zugriff, Liste von Plattenadressen der Datenblöcke,...). Bei großen Dateien werden auch mehrere MFT-Einträge genutzt, der erste ist dann der Basisdatensatz.

⁴ Es gibt Dienstprogramme, mit denen man sich die MFT anschauen kann. Beispiel hierfür ist das Tool *nfi*, das auf der Microsoft-Support-Website zu finden ist.

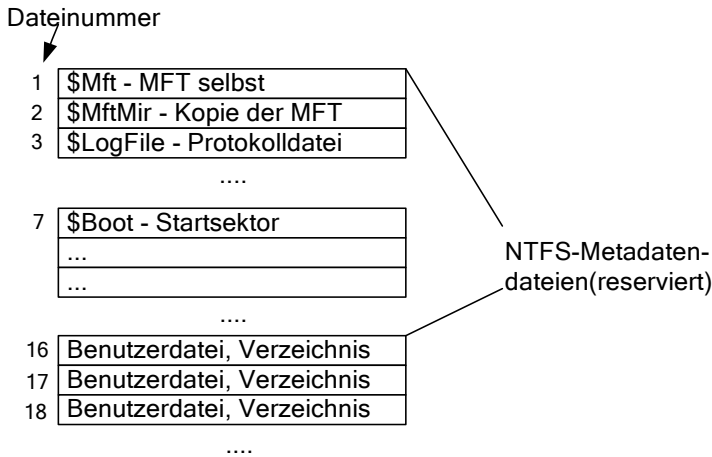


Abbildung 8-15: Master File Table unter Windows

Die ersten MFT-Dateien sind für den Benutzer nicht sichtbar. Deren Namen beginnen mit „\$“ und enthalten Metainformationen. Die erste Datei *\$Mft* beschreibt die MFT selbst. Eine weitere, wichtige Datei ist *\$Boot*, die als Startdatei dient. Hier wird der Bootstrapcode gespeichert. Der Aufbau der MFT ist in Abbildung 8-15 skizziert. Ab dem Record 16 liegen die Beschreibungen von Benutzerdateien und Verzeichnissen.

Neben den dargestellten NTFS-Mechanismen gibt es noch eine Fülle von fortgeschrittenen Funktionen in NTFS, wie z.B. Unicode-Namen (16-Bit-Zeichensatz, mit dem man alle wichtigen Schriften darstellen kann), Komprimierung, Verschlüsselung und Defragmentierung. Zudem ist über den Volume-Manager eine logische Plattenverwaltung und eine Neupartitionierung der Dateisysteme dynamisch ohne einen Neustart des Systems möglich.

Die folgende Tabelle 8-2 fasst die wichtigsten Eigenschaften der FAT- und des NTFS-Dateisystems nochmals zusammen:

Tabelle 8-2: Parameter der Dateisysteme unter Windows

Dateisystemtyp	Anzahl Bit für Clusterindex	Anzahl Cluster	Unterstützte Clustergrößen	Max. Dateisystemgröße
FAT-12	12 Bit	$2^{12} = 4096$	512 Byte – 8 KByte	32 MB
FAT-16	16 Bit	$2^{16} = 65536$	512 Byte – 64 KByte	4 GB
FAT-32	32 Bit, aber nur 28 genutzt	$2^{28} = 268435456$	512 Byte – 32 KByte	8 TB, begrenzt auf 32 GB
NTFS	64 Bit	$2^{64} = 1,84467 \cdot 10^{19}$	512 Byte – 64 Kbyte	16 Exabyte, begrenzt auf 256 TB

8.3 Storage-Systeme

8.3.1 RAID-Plattensysteme

Multiple Plattenspeicher, sog. *RAIDs* sind heute sehr verbreitet. RAID ist eine Abkürzung für *Redundant Array of Inexpensive Disks*. Mehrere kleine Platten werden hier als große virtuelle Platte verwaltet. „Inexpensive“ wurde später durch das Attribut „Independent“ ersetzt. Der Betrieb eines RAID-Systems erfordert mindestens zwei Festplatten. Die Festplatten bilden einen Verbund, der aus Sicht des Betriebssystems als eine einzige logische Festplatte betrachtet wird. Die Daten werden je nach RAID-Verfahren über die einzelnen physikalischen Festplatten verteilt. Redundante Information wird zu den Nutzdaten ergänzt und ebenfalls auf die physikalischen Platten verteilt, um im Fehlerfall die Wiederherstellung einer Platte zu unterstützen.

Man verwendet RAID-Systeme zur Verbesserung der Leistung und zur Erhöhung der Ausfallsicherheit, wobei verschiedene Varianten unterschieden werden. Man unterscheidet Hardware- und Software-RAID. Im ersteren Fall wird der Plattenzugriff über einen eigenen RAID-Controller organisiert. Von Software-RAID spricht man, wenn das Zusammenwirken der Festplatten über Software geregelt wird. Die meisten modernen Betriebssysteme unterstützen heute auch Software-RAID. Vorteil dieser Variante ist, dass keine eigene Controller-Hardware erforderlich ist, jedoch ist Software-RAID in der Regel langsamer als Hardware-RAID.

Es gibt heute viele Varianten von RAID-Systemen (RAID-0,-1,-2,-3,-4,-5,-6,-7 und verschiedene Kombinationen wie RAID 0+1 oder RAID 1+0). Man spricht auch von sog. *RAID-Levels*. Sie unterscheiden sich hauptsächlich hinsichtlich der Verteilung der Daten und der Ergänzung von redundanter Information. Einige RAID-Levels sollen hier kurz skizziert werden.

Bei den meisten RAID-Systemen wird der gesamte logische Plattenbereich in Streifen (Stripes) eingeteilt (siehe Abbildung 8-16), wobei ein Streifen mehrere Blöcke enthält. Man spricht in diesem Zusammenhang von *Striping*.

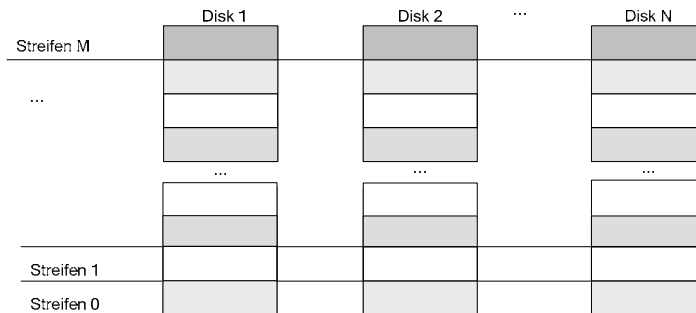


Abbildung 8-16: Striping

RAID-0. In sog. *RAID-0*-Systemen (Abbildung 8-17) werden die Stripen über mehrere Platten eines Arrays verteilt. Die Größe der Stripen beeinflusst die Leistung erheblich. Üblich sind hier Stripe-Größen von 32, 64 und 128 KB. Man spricht hier auch von *Striping-Granularität*. Die Verteilung übernimmt entweder das Betriebssystem oder ein eigener RAID-Controller (Hardware- oder Software-RAID). RAID-0-Systeme weisen z.B. einen hohen E/A-Durchsatz auf, sind also schnell, aber nicht ausfallsicher. Der hohe Durchsatz ergibt sich durch die parallele Abwicklung der Festplattenzugriffe. RAID-0 ist im eigentlichen Sinne kein echtes RAID-System, da keine Redundanz gespeichert wird.

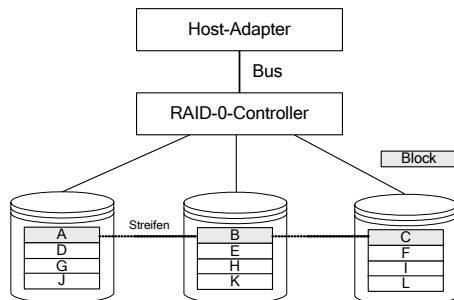


Abbildung 8-17: RAID-0-Systeme

RAID-1. In *RAID-1*-Systemen (Abbildung 8-18) wird eine Vollspiegelung der Daten durchgeführt, was die Ausfallsicherheit verbessert, aber im Allgemeinen langsamer ist. RAID-1 bietet also die volle Redundanz der gespeicherten Daten, wobei die Kapazität des Systems höchstens so groß ist, wie der kleinste Spiegel. Fällt ein Spiegel aus, können die Platten des anderen Spiegels weiterhin genutzt werden. In RAID-1-Systemen wird bei schreibendem Zugriff immer auf beide Festplattenspiegel zugegriffen, da eine Kopie erstellt werden muss.

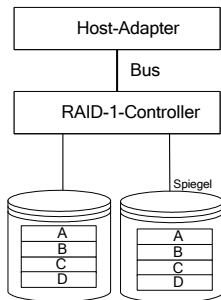


Abbildung 8-18: RAID-1-Systeme

RAID-2 (Abbildung 8-19) wurde früher hauptsächlich bei Großrechnern verwendet. Die Daten werden bei RAID-2 in Bitfolgen mit fester Größe zerlegt und mit Hilfe eines Hamming-Codes auf größere Bitfolgen abgebildet. Jedes Byte oder Wort wird um ein Paritätsbit ergänzt. Der ermittelte Hamming-Code dient der Erkennung von 2-Bitfehlern und der Behebung von 1-Bit-Fehlern. Die Bits des Codeworts werden über einzelne Platten aufgeteilt, was einen hohen Durchsatz ermöglicht. Bei Ausfall einer Platte ist eine Rekonstruktion möglich.

RAID-2. RAID-2 benötigt im Vergleich zu RAID-1 weniger Festplatten und wäre dann sinnvoll, wenn die Festplatten eine hohe Fehlerrate aufweisen, was heute aber nicht mehr der Fall ist. Daher wird das Verfahren auch so gut wie gar nicht eingesetzt.

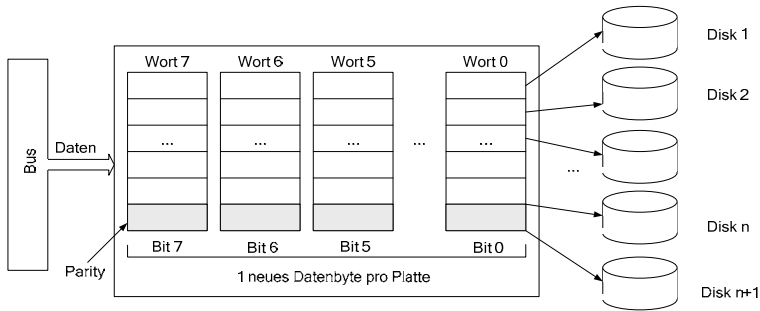


Abbildung 8-19: RAID-2-Systeme (nach Brause 2001)

RAID-1+0. RAID-1+0 (Abbildung 8-20) vereint als Verbund von RAID-1 und RAID-0 die Vorteile beider Verfahren. Bei einem RAID-1+0-Verbund (auch RAID-10 genannt) wird zunächst ein Striping vorgenommen und danach eine Spiegelung durchgeführt. Für RAID-10 benötigt man mindestens vier Festplatten. RAID-10-Systeme sind schnell sowie ausfallsicher und werden heute häufig verwendet.

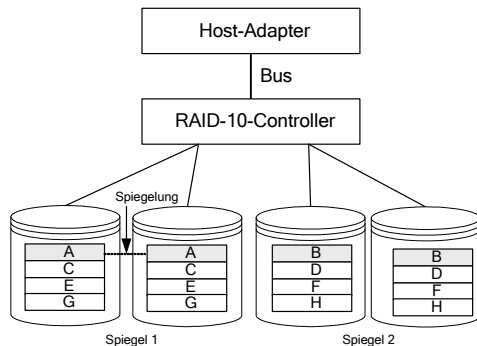


Abbildung 8-20: RAID-10-Systeme

RAID-3. RAID-3-Systeme sind ein Vorgänger der RAID-5-Systeme, bei der Prüfdaten zusammengefasst und auf einer einzigen redundanten Platte gespeichert werden. Es wird auch kein Hamming-Code als Fehlerkorrekturcode, sondern nur ein einfaches Paritätsbit verwendet. RAID-3 dient damit der Reduzierung von Kosten. Der Ausfall einer Platte führt nicht zu einer Ausfallzeit, der Ausfall von zwei Platten schon. Das Verfahren ist in der Praxis auch nicht verbreitet und weitgehend durch RAID-5 abgelöst.

RAID-4. Bei RAID-4 werden die Datenbits wie bei RAID-0 zu Streifen zusammengefasst. Pro Streifen wird eine Prüfsumme gebildet und auf einer eigenen Platte gespeichert. Die Ausfallsicherheit ist ähnlich zu RAID-3, der Speicheraufwand

weniger als bei RAID-1. Nachteilig bei RAID-4 ist der Leistungsverlust beim Schreiben. Die dedizierte Platte mit Paritätsinformation kann zu einem Leistungsengpass werden und aufgrund des häufigen Zugriffs auch leichter ausfallen.

RAID-5. In RAID-5-Systemen (Abbildung 8-21) sind die Paritätsabschnitte auf alle Platten verteilt, um eine gleichmäßige Plattenauslastung zu erreichen. Ansonsten ist RAID-5 sehr ähnlich zu RAID-4. Dies führt zu guter Leistung beim Lesen, aber schlechterer Leistung beim Schreiben. Durch die Paritätsinformation geht natürlich ein Teil der Gesamtkapazität der eingesetzten Festplatten für die Nettokapazität (Speicherplatz für die Nutzdaten) verloren. Die Nettokapazität errechnet sich wie folgt:

Nettokapazität des RAID-5-Systems =
 Kapazität der kleinsten Platte * (Anzahl der eingesetzten Platten - 1).

Bei Einsatz von 4 Festplatten zu je 200 GB ergibt sich eine Nettokapazität von 600 GB für die Nutzdaten, 200 GB wird für Redundanz benutzt. Der Verlust einer Platte hat keine Auswirkungen, da die Platte wiederhergestellt werden kann. Der Verlust von zwei Platten bedeutet den Ausfall des gesamten Systems. RAID-5 ist nicht so geeignet bei hoher Transaktionslast.

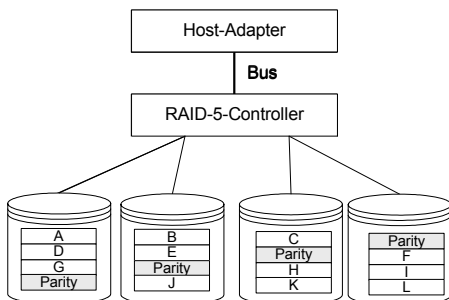


Abbildung 8-21: RAID-5-Systeme

RAID-6. RAID-6-Systeme sind wie RAID-5 aufgebaut, speichern aber doppelt so viele Paritätsinformationen, so dass sogar der Ausfall von zwei Platten ohne Auswirkung bleibt. Wenn n Festplatten für die Nutzdaten benötigt werden, ist ein RAID-6-System aus n+2 Platten aufgebaut. RAID-6 zeichnet sich durch gute Leistung beim Lesen aus, die Schreibleistung ist allerdings schlechter als bei RAID-5. RAID-6 ist für sehr hohe Ausfallsicherheitsanforderungen gut geeignet.

Weitere RAID-Varianten und eine weiterführende Beschreibung von RAID-Systemen sind u.a. in (Stallings 2003), (Herrmann 2002) und (Brause 2001) zu finden.

8.3.2 NAS und SAN

NAS (Network Attached Storage) und SAN (Storage Area Networks) sind Storage-Systeme für typische Client-/Server-Landschaften.

Network Attached Storage. Als NAS bezeichnet man Massenspeichereinheiten, die an ein lokales Netzwerk (LAN) angeschlossen sind (siehe Abbildung 8-22). NAS werden eingesetzt, um die hohen Aufwände bei der Installation und Administration eines dedizierten Dateiservers zu vermeiden. Ein Problem der Network-Attached-Storage-Systeme ist, dass diese zusätzlich das vorhandene Netzwerk mit den Zugriffen auf die Datenträger belasten. Bei Network-Attached-Storage-Systemen ist eine effiziente und flexible Nutzung der Speicherkapazität daher nur eingeschränkt möglich.

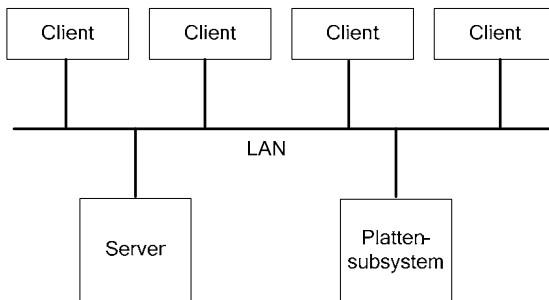


Abbildung 8-22: NAS-System

Storage Area Network. SAN (Storage Area Network) wurde entwickelt, um diesem Problem entgegenzuwirken. Ein Storage Area Network unterscheidet sich von einem gewöhnlichen LAN, indem es ein dediziertes Netzwerk zwischen Servern und von den Servern genutzten Speicherressourcen nutzt. Der Datenverkehr in einem SAN besteht hauptsächlich in der Übertragung von blockbasierten Daten. Ein SAN ist also auch ein Netzwerk, in dem große Datenmengen gespeichert und bewegt werden. Im SAN wird der gesamte Speicher, unabhängig vom Standort und Betriebssystem, zentral verwaltet und zu virtuellen Einheiten zusammengefasst. Der Speicher muss dazu nicht an ein und demselben Ort untergebracht sein. Abbildung 8-23 zeigt ein typisches SAN mit vier Platten, auf die von zwei Servern zugegriffen wird.

Ein SAN repräsentiert einen Massenspeicher für die Serversysteme und ist für die schnelle Datenübertragung zwischen den einzelnen Speichern und Servern optimiert. So ist es z.B. möglich, die Datensicherung im SAN während des laufenden Betriebs zu machen, ohne dass es zu Überlastungen und Verzögerungen im Netz-

werk kommen kann. Die Server sehen das SAN als eine Art Datenpool, der in voneinander getrennte logische Einheiten aufgeteilt ist. Mehrere redundante Wege zwischen dem Anwender und den Daten können auch vor möglichen Ausfällen schützen.

Ein großer Vorteil, den das SAN bringt, ist die Virtualisierung der vorhandenen Plattensubsysteme. Der verteilt vorhandene Massenspeicher kann virtuell wie eine einzige Festplatte behandelt werden. Ein weiterer Vorteil von SANs ist ihre Fehler-toleranz. So können alle wichtigen Elemente mehrfach redundant vorhanden sein. In einem typischen Storage Area Network wäre es denkbar, dass sich an zwei möglichst weit auseinanderliegenden Orten auf dem Betriebsgelände jeweils ein baugleiches Plattensubsystem befindet. Jedes Plattensubsystem kann über ein eigenes Netz angebunden werden.

SANs werden heute meistens über Glasfaserkabelnetze gebildet. Ein einfaches SAN besteht aus einem sog. Fibre-Channel-Switch, mindestens mehreren Plattensubsystemen und den Servern, die über so genannte Host-Bus-Adapter, mit dem Fibre-Channel-Switch verbunden werden. In diesem Umfeld werden Bandbreiten im Bereich von 1 GBit/s bis 4 GBit/s erreicht.

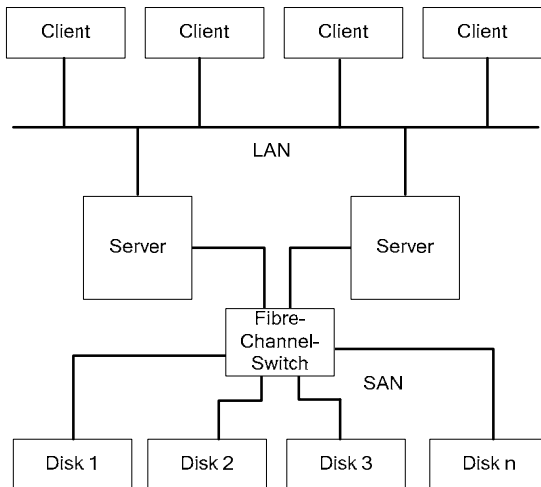


Abbildung 8-23: Prinzipieller Aufbau eines Storage Area Network

Diese Architektur wird auch als Fibre-Channel-Switched-Fabric bezeichnet und ist sehr leistungsfähig und ausfallsicher (siehe Abbildung 8-23). Im Mittelpunkt steht ein Fibre-Channel-Switch, der zwischen den Servern und den Geräten eine direkte Punkt-zu-Punkt-Verbindung schalten kann. SANs verwenden ein Optimierungskonzept, das als *Multi-Pathing* bezeichnet wird. Wenn ein zugreifender Rechner

über mehrere Host-Bus-Adapter ein Plattensubsystem erreichen kann, so wird der Datentransfer zwischen beiden Systemen auf beide Datenwege verteilt. Dies führt zu einer Steigerung der Übertragungsgeschwindigkeit.

Obwohl Fibre-Channel sich als Übertragungsmedium für SAN durchgesetzt hat, kommen auch andere Techniken in Frage. Neben Fibre-Channel gibt es als weitere Technologien für SANs *iSCSI* (Internet Small Computer System Interface over IP) und das klassische Ethernet⁵.

iSCSI ist ein Verfahren für Speichernetzwerke auf Basis der Internet-Protokolle IP und TCP. Bei diesem Verfahren werden die Informationen zwischen den Servern und den Festplattengeräten als SCSI-Daten in TCP/IP-Nachrichten verpackt und über IP-Netze transportiert. Hier können vorhandene Netzwerkkomponenten eingesetzt werden. Neue Hardware wie ein Fibre-Channel-Switch ist nicht nötig.

Bei der direkten Nutzung der Ethernet-Technologie ist ebenfalls keine eigene Netzwerk-Hardware nötig. Die Datenübertragung erfolgt direkt über diese Technologie.

8.4 Übungsaufgaben

1. Welche Aufgaben hat ein Gerätetreiber?
2. Erläutern Sie den prinzipiellen Aufbau einer Festplatte!
3. Was bedeutet Formatieren einer Festplatte?
4. Wie verwaltet Unix seine Dateien und wie werden Ein-/Ausgabegeräte behandelt?
5. Wozu ist der DMA-Mechanismus sinnvoll und wie funktioniert er prinzipiell?
6. Was bedeutet RAID?
7. Erläutern Sie die Arbeitsweise von RAID-1!
8. Was ist der Unterschied zwischen RAID-4 und RAID-5?
9. Welchen wesentlichen Vorteil bietet RAID-6 im Vergleich zu RAID-5 und welcher Nachteil wird dadurch erkauft?
10. Welche Vorteile bietet RAID-10?
11. Welchen Vorteil hat Software-RAID gegenüber Hardware-RAID?
12. Zu welchen Problemen kann Caching von Plattenspeicherinhalten in Treiberpuffern bei Ausfall eines Systems führen?
13. Was sind i-Nodes unter Unix und wozu dient die i-Node-Struktur?
14. Welche Dateisysteme unter Windows kennen Sie und wie unterscheiden sie sich?
15. Was bezeichnet man als SAN?

⁵ Ethernet ist ein weit verbreiteter Standard mit vielen Varianten für lokale Netze.

9 Schlussbemerkung

Dieses Buch sollte eine Einführung in die Grundlagen moderner Betriebssysteme geben. Es wurden vor allem die Betriebsmittel Prozess, Thread, Speicher, Datei und Gerät sowie Maßnahmen für ihre effiziente Verwaltung erläutert. Weiterhin wurde auf die Synchronisation nebenläufiger Prozesse bzw. Threads, die um Betriebsmittel konkurrieren, sowie auf die Kommunikation dieser eingegangen. Betriebsmittel sind zwar heute nicht mehr so knapp wie noch vor 20 Jahren, jedoch benötigen heutige Anwendungssysteme wesentlich mehr davon (vor allem externen und internen Speicher und CPU-Zeit), weshalb man auch heute noch auf eine vernünftige Nutzung der verfügbaren Ressourcen achten muss. Im Fokus stand dabei die Sichtweise eines Informatikers, der betriebliche Steuerungs- und Informationssysteme realisiert und auf Betriebssystemen zum Ablauf bringt. Weniger betrachtet wurden realzeitnahe Anwendungen.

Aus Sicht eines anwendungsorientierten Informatikers oder Wirtschaftsinformatikers werden die meisten Algorithmen, die in Betriebssystemen implementiert sind, in der praktischen Arbeit relativ selten benötigt. Trotzdem sollte man insbesondere bei Systemtests sowie bei der Beurteilung und beim Deployment von Anwendungssystemen wissen, wo die Grenzen eines genutzten Betriebssystems liegen können bzw. worauf man achten muss, um ein Anwendungssystem entsprechend in einer Betriebssystem- und Netzwerkumgebung effizient zum Ablauf zu bringen. Deshalb sind die grundlegenden Kenntnisse der Ressourcenverwaltung, wie sie in diesem Lehrbuch erläutert wurden, auch für die heutige Praxis von großer Bedeutung.

Eine weiterführende Beschäftigung mit Verfahren und Algorithmen aus der Welt der Betriebssysteme kann über die angegebene Literatur erfolgen. Das Thema bleibt weiterhin interessant, insbesondere wenn man Konzepte für verteilte und mobile Systeme, Grid-Computing und Virtualisierung beobachtet. Aber auch die heute gängigen Betriebssysteme müssen sich neuen Herausforderungen stellen. Zukünftige Betriebssysteme müssen bald auch noch größere Adressräume und externe Speichermedien unterstützen. Betriebssysteme müssen also weiterentwickelt werden, damit sie auch in Zukunft eine vernünftige Basis für neue Anwendungssysteme bereitstellen.

10 Lösungen zu den Übungsaufgaben

10.1 Einführung

1. *Nennen Sie fünf Betriebssystemkategorien!*

- Mainframes (High-End-Systeme wie IBM OS/390 (heute IBM zSeries) und Siemens Fujitsu BS 2000/OSD)
- Serverbetriebssysteme (Unix, Windows NT/2000/2003/XP)
- Echtzeitbetriebssysteme (VxWorks, QNX)
- PC-Betriebssysteme (Windows-Derivate, Linux)
- Embedded Systems (Steuerungssysteme)
- Handheld-Computer (Palm OS, Windows CE, Symbian)
- Smartcard-Betriebssysteme

2. *Was ist ein Von-Neumann-Rechner und wie unterscheidet er sich von einem Harvard-Rechner?*

Ein Von-Neumann-Rechner besteht aus den vier Funktionseinheiten Leitwerk (Control Unit, CU), Rechenwerk (Processing Unit, PU), Speicher (Memory) sowie Ein-/Ausgabe (Input/Output, E/A). Das Leitwerk holt die Maschinenbefehle nacheinander in den Speicher und führt sie aus. Damit stellt das Leitwerk den „Befehlsprozessor“ dar. Das Rechenwerk, auch Arithmetisch-Logische Einheit (ALU) genannt, stellt den „Datenprozessor“ dar und führt logische und arithmetische Operationen aus. Im Speicher liegen die Maschinenbefehle und die zu verarbeitenden Daten. Maschinenbefehle und Daten sind also in einem gemeinsamen Speicher. Rechenwerk und Leitwerk werden heute in der Regel in einem Prozessor, der als Zentraleinheit (CPU, Central Processing Unit) bezeichnet wird, zusammengefasst.

Der Harvard-Rechner, der nach der Struktur des Mark-I-Rechners benannt ist, hat im Unterschied zum Von-Neumann-Rechner zwei getrennte Speicher, einen für die Daten und einen für die Maschinenbefehle. Beide Speicher werden auch über einen getrennten Bus mit der CPU verbunden.

3. *Wozu braucht man in Computern CPU-Register?*

Diese schnellen Speicher werden benötigt, um die Maschinenbefehle auszuführen. Je nach Maschinenbefehl werden eines oder mehrere Register benötigt. Die Maschinenbefehle schreiben ihre Operanden oft in Register oder lesen

Operanden aus Registern. Man nennt die Register je nach Typ Integerregister, Universalregister, Gleitkommaregister, Datenregister, Segmentregister usw.

Ein weiteres, spezielles Register ist der Program Counter (PC, Programmzähler, Befehlszähler, Befehlsregister), der immer die Hauptspeicheradresse des nächsten auszuführenden Befehls enthält.

4. *Was ist ein Program Status Word (PSW) und wozu wird es verwendet?*

Ein PSW (= Program Status Word) ist ein Statusregister, das für Vergleichsoperationen benutzt wird und weitere Kontrollbits wie etwa den aktuellen Modus (Benutzermodus, Kernelmodus), in dem sich die Rechneranlage befindet, enthält.

5. *Was ist ein Mehrzweck- oder Universalbetriebssystem?*

Ein Mehrzweck- oder Universalbetriebssystem ist ein Betriebssystem, das für verschiedenste Anwendungen eingesetzt werden kann, aber meist nicht über Realtime-Eigenschaften verfügt. Es wird überwiegend für betriebliche und verwaltungstechnische Aufgabenstellungen, weniger für Realzeitprobleme eingesetzt wird.

10.2 Betriebssystemarchitekturen und Betriebsarten

1. *Was versteht man unter einem Mikrokern?*

Im Gegensatz zu monolithischen Architekturen wird eine Betriebssystemarchitektur, die einen leichtgewichtigen Kernel enthält, als Mikrokern-Architektur bezeichnet. Der Kernel wird bei dieser Architekturvariante dadurch entlastet, dass Funktionalität in Anwendungsprozesse, sog. Serverprozesse, ausgelagert wird. Der Kernel übernimmt hier im Wesentlichen die Abwicklung der Kommunikation zwischen Client- und Serverprozessen. Clientprozesse greifen bei dieser Betriebssystemarchitektur auf den Mikrokern über eine Service-Request-Schnittstelle zu. Der Mikrokern leitet die Requests an die entsprechenden Serverprozesse, die in Anwendungsprozessen ablaufen, weiter und stellt umgekehrt den Clientprozessen die Ergebnisse zu.

2. *Nennen Sie ein Beispiel für ein Betriebssystem mit Mikrokern-Architektur!*

- Mach von der Carnegie Mellon University (CMU)
- MAC OS X
- Hurd, L4
- Amoeba
- Chorus.

3. Was versteht man unter einem „echt“ verteilten Betriebssystem im Gegensatz zu den heutigen Client-/Server-Systemen?

In einem echt verteilten Betriebssystem ist der Kernel auf mehrere Rechner im Netzwerk verteilt.

4. Was versteht man unter Kommunikations-Middleware?

Kommunikations-Middleware wie CORBA (Common Object Request Broker Architecture), Java RMI (Java Remote Method Invocation) oder Java EJB (Enterprise Java Beans) setzt man ein, um verteilte Anwendungssysteme zu realisieren. Kommunikations-Middleware wird in der Regel im Benutzermodus betrieben und stellt dem Anwendungsprogramm komfortable Dienste zur Kommunikation bereit.

5. Nennen Sie vier Betriebsmittel, welche das Betriebssystem verwaltet! Welche davon sind hardware- und welche softwaretechnische Betriebsmittel?

Betriebsmittel sind u.a.:

- Speicher (Hardware)
- Prozessor (Hardware)
- Gerät (Hardware)
- Datei (Software)
- Nachricht (Software)
- Prozess (Software)

6. Erläutern Sie den Unterschied zwischen Teilnehmer- und Teilhaberbetrieb! In welcher Betriebsart wird üblicherweise ein Transaktionsmonitor eingesetzt?

Im Teilnehmerbetrieb erhält jeder Anwender seinen eigenen Benutzerprozess sowie weitere Betriebsmittel vom Betriebssystem zugeteilt. Der Benutzer meldet sich über einen Login-Dialog beim System an und bekommt die Betriebsmittel dediziert zugeordnet.

Im Teilhaberbetrieb werden Prozesse und Betriebsmittel über einen Transaktionsmonitor zugeteilt. Im Gegensatz zum Teilnehmerbetrieb kommt der Teilhaberbetrieb mit wenigen Prozessen aus, um viele Dialoganwender zu unterstützen. Die wenigen Prozesse werden vom Transaktionsmonitor verwaltet und effizient an die Benutzer verteilt und zwar so, als ob jeder Benutzer einen eigenen Prozess zugeordnet hätte. Die Betriebsart Teilhaberbetrieb ist ideal für dialogorientierte Programme mit vielen parallel arbeitenden Anwendern, die meistens kurze und schnelle Transaktionen ausführen, wie dies etwa bei einem Buchungssystem für Flugbuchungen der Fall ist.

7. *Welche Aufgaben erfüllt ein Transaktionsmonitor, welche ein Application-Server?*

Ein Transaktionsmonitor ist ein Dienstprogramm, das oberhalb des Kernels angesiedelt ist und Aufgaben wie die Zugangskontrolle, die Verteilung der Anfragen auf bereitgestellte Prozesse und die Optimierung der Zugriffe sowie die Verwaltung der Ressourcen und auch die Zuordnung von Datenbankverbindungen übernimmt. Transaktionsmonitore werden im Teilhaberbetrieb eingesetzt und dienen dem Ablauf von Transaktionsprogrammen. Ein Application-Server erfüllt ähnliche Aufgaben wie der Transaktionsmonitor. Anstelle von Transaktionsprogrammen laufen im Application-Server Komponenten verteilter Anwendungen ab.

8. *Nennen Sie einen Vorteil der Schichtenarchitektur bei Betriebssystemen!*

Die Abhängigkeiten von der Hardware sind in einer Schicht gekapselt, was eine Portierung auf eine andere Hardwareplattform erleichtert.

9. *Was versteht man unter Multiprogramming?*

Im Gegensatz zum Einprogrammbetrieb eröffnet Multiprogramming die Möglichkeit der „gleichzeitigen“ oder aber auch „quasi-gleichzeitigen“ Ausführung von Programmen in einem Betriebssystem.

10. *Benötigt man für Multiprogramming mehrere CPUs?*

Nein, nicht unbedingt. Multiprogramming kann auch über eine CPU realisiert werden, die nach einer vorgegebenen Strategie des Betriebssystems (Scheduling und Dispatching) auf die nebenläufigen Programme (Prozesse) aufgeteilt wird.

11. *Verfügen die Betriebssysteme Unix und Windows 2000/2003/XP über Mikrokern-Architekturen und wenn nein, wie sind sie konzipiert?*

Es sind alles keine echten Mikrokern-Architekturen. Unix verfügt meist über eine schichtenorientierte Architektur, Windows ebenso. Windows nutzt Konzepte des Mikrokerns, Subsysteme und Systemservices sind in den Benutzermodus ausgelagert. Nach der reinen Lehre ist es aber kein Mikrokern.

12. *Was bezeichnet man als Timesharing?*

Die Zuordnung des Prozessors nach Zeitintervallen an die nebenläufigen Programme wird als Timesharing bezeichnet.

13. *Wozu verwendet man Embedded Systems? Nennen Sie ein Beispiel!*

Als Embedded System bezeichnet man ein Rechner- bzw. Steuerungssystem, das in Geräten (Telefone, DVD-Player, Waschmaschinen, Fernseher), Robotern oder Fahrzeugen (Flugzeuge, Autos) eingebaut ist und dort seine Aufgaben meist unsichtbar verrichtet. Embedded Systems kann man weitgehend als geschlossene Systeme betrachten, die eine dedizierte Aufgabe übernehmen. Das Betriebssystem ist klein ausgelegt, da wenige Ressourcen verfügbar sind. Oft verfügen Embedded Systems aufgrund der knappen Ressourcen nicht über ein Betriebssystem. Die Software bedient dann direkt die Hardwareschnittstellen. Embedded Systems (eingebettete Systeme) sind oft auch Echtzeitsysteme.

14. *Wozu dient ein Terminalserver?*

Terminalserver bedienen „dumme“ Clientrechner (sog. Thin Clients), wobei die Anwendungsprogramme vollständig in den Servern ablaufen und die Clientrechner nur noch für Zwecke der Präsentation eingesetzt werden. Die Idee hinter Terminalsdiensten ist die Zentralisierung von Betriebsmitteln, um die beteiligten Systeme leichter administrieren zu können. Der Anwender arbeitet mit der gewohnten Benutzeroberfläche, die Anwendungen laufen aber komplett im Server ab, ohne dass dies der Benutzer bemerken soll. Lediglich Bildschirmänderungen werden im Client dargestellt. Ein Server bedient je nach Leistungsfähigkeit mehrere oder viele Benutzer.

10.3 Interruptverarbeitung

1. *Was ist der Unterschied zwischen Polling und interruptgesteuerter Verarbeitung?*

Unter Polling versteht man das zyklische Abfragen von einer Ereignisquelle bzw. mehreren Ereignisquellen (z.B. E/A-Geräte), um deren Kommunikationsbereitschaft festzustellen bzw. um anliegende Ereignisse oder Kommunikationswünsche der Ereignisquelle abzufragen. Polling hat den Nachteil, dass die CPU ständig arbeiten muss und damit die Effizienz eines Systems beeinträchtigt ist. Die meiste Zeit wird umsonst nachgefragt. Allerdings ist das Verfahren relativ leicht zu implementieren.

Im Gegensatz zu Polling sind Interrupts (Unterbrechungen) sog. Betriebssystembedingungen oder asynchrone Ereignisse, die den Prozessor veranlassen, einen vordefinierten Code auszuführen, der außerhalb des normalen Programmflusses liegt. Überwachte Ereignisquellen müssen nicht ständig abgefragt werden, sondern die Ereignisquellen melden sich beim Auftreten eines Ereignisses, das behandelt werden muss.

2. *Was ist der Unterschied zwischen den Exception-Typen Fault und Trap? Nennen Sie jeweils ein Beispiel!*

Faults sind Unterbrechungen vor der Befehlsausführung wie z.B. Seitenfehler.

Traps sind Unterbrechungen nach der Befehlsausführung, die z.B. bei einer Division durch 0 auftreten.

3. *Wozu dient ein Systemcall und wie wird er üblicherweise von einem Betriebssystem wie Windows oder Unix ausgeführt?*

Ein Systemcall ist ein synchroner, also ein vorhersehbarer und reproduzierbarer Interrupt und wird auch als Software-Interrupt bezeichnet.

Ein Systemcall ist ein Dienstauftrag an das Betriebssystem, bei dessen Ausführung in den Kernelmodus gewechselt wird. Der Kontrollfluss wird dabei meist von einem Anwendungsprogramm an den Kernel übergeben.

4. *Was bedeutet „Maskierung“ von Unterbrechungsanforderungen?*

Eine Maskierung ist das explizite Ausschalten eines Interrupts für ein bestimmtes Gerät. Die Maskierung erfolgt für gewöhnlich über ein Maskenregister (Interrupt Mask Register). Für jede Interrupt-Quelle wird in dem Register ein Maskierungsbit verwaltet. Wird das Bit auf 1 gesetzt, ist der Interrupt ausgeschaltet.

5. *Wie erkennt die CPU, dass eine Unterbrechungsanforderung ansteht?*

Die Prüfung, ob eine Unterbrechung ansteht, ist Teil des Befehlszyklus. Nach Ausführung eines Maschinenbefehls wird überprüft, ob ein Interrupt-Request anliegt. Ist dies der Fall, wird in ein spezielles Unterprogramm, die ISR, oder bei größeren Betriebssystemen in eine entsprechend davor geschaltete Verteilungsroutine, verzweigt.

6. *Was versteht man unter einer Interrupt-Vektor-Tabelle?*

Dies ist eine Tabelle, welche Interrupt-Vektoren, also die eigentlichen Einsprungsadressen der Interrupt-Service-Routinen enthält und an einer vordefinierten Stelle im Kernelspeicher liegt. Der Index zur Adressierung innerhalb der Tabelle wird der CPU implizit durch den Interrupt-Controller anhand der belegten Adressleitungen übermittelt. Jeder Interrupt-Quelle wird ein fester Index auf diese Tabelle zugeordnet.

7. *Was ist eine Interrupt-Service-Routine und wann wird sie aufgerufen?*

Das Programmstück, das den Interrupt bearbeitet, wird als Interrupt-Service-Routine (ISR, Interrupt-Bearbeitungsroutine) bezeichnet. Für jeden Interrupt-Typen gibt es eine ISR. Eine ISR kann aber auch mehreren Interrupt-Typen zugeordnet werden. Das Betriebssystem stellt für alle Interrupts eine passende ISR zur Verfügung.

8. *Was bedeutet Interrupt-Sharing?*

Beim Interrupt-Sharing teilen sich mehrere Geräte eine Interrupt-Nummer. Bei Auftreten eines Interrupts muss dann ermittelt werden, welches Gerät den Interrupt tatsächlich ausgelöst hat. Diese Aufgabe erledigt meist das Betriebssystem.

9. *Was versteckt der Trap-Mechanismus zum Aufruf eines Systemdienstes vor dem Anwendungsprogramm?*

Der Trap-Mechanismus dient dazu, dass der Aufruf eines Systemcalls von einem Anwendungsprogramm aus ermöglicht wird, ohne dass die tatsächliche Adresse der Systemroutine bekannt sein muss. Dieser Mechanismus versteckt also die Adresse der Systemroutine vor dem Anwendungsprogramm.

10. *Erläutern Sie die Abwicklung eines Hardware-Interrupts unter Linux. Gehen sie dabei auf Tasklets ein!*

In Linux-Systemen wird ebenfalls eine Tabelle mit Referenzen auf die Interrupt-Handler (Interrupt-Service-Routinen, ISR) verwaltet. Jedem Interrupt wird über einen Index auf die Tabelle ein Interrupt-Handler zugeordnet. Bei Auftreten eines Interrupts wird zunächst in den Kernelmodus gewechselt, sofern nicht schon geschehen, und anschließend werden die Register gesichert.

Alles was nicht unbedingt sofort erledigt werden muss, wird zunächst notiert und später abgewickelt. Dieser Mechanismus wird in Linux als Tasklet bezeichnet. Hierzu wird eine Datenstruktur namens `tasklet_struct` mit entsprechenden Informationen zur Verarbeitung der später auszuführenden Aufgabe erzeugt, und schließlich wird eine Registrierung des Tasklets über eine im Kernel vorhandene Scheduler-Funktion (`tasklet_schedule`) vorgenommen. Vom Tasklet-Scheduler wird eine Liste aller anstehenden Tasklets verwaltet und deren Abarbeitung vorgenommen.

11. *Erläutern Sie die Abwicklung eines Traps (Systemcalls) bei einem seriellen und bei einem nebenläufigen Kernel!*

Traps werden durch einen speziellen Maschinenbefehl des Prozessors, den sog. Supervisor-Call oder SVC unterstützt. Bei Ausführung des Systemcalls über den Software-Interrupt wird wie folgt verfahren:

- Der aktuelle Kontext des laufenden Programms, also die Information, welche den aktuellen Status eines Prozesses beschreibt, wird gesichert.
- Der Program Counter wird mit der Adresse der passenden Systemroutine belegt.
- Vom Benutzermodus wird in den Kernelmodus geschaltet.
- Die adressierte Systemroutine wird durchlaufen.
- Anschließend wird wieder der alte Kontext des Prozesses hergestellt und der Program Counter mit der Adresse des Befehls nach dem Systemcall belegt.

Bei einer seriellen Kernel-Implementierung verarbeitet der Kernel zu einem Zeitpunkt höchstens einen Systemcall und ist dabei nicht unterbrechbar. Im Gegensatz dazu ist ein nebenläufiger Kernel unterbrechbar und verarbeitet mehrere Systemcalls quasi gleichzeitig.

12. *Nennen Sie den Unterschied zwischen einem synchronen und asynchronen Interrupt!*

Synchrone Interrupts treten bei synchronen Ereignissen auf. Dies sind Ereignisse, die bei identischen Randbedingungen (Programmausführungen mit gleichen Daten) immer an der gleichen Programmstelle auftreten. Synchrone Interrupts werden auch als Ausnahmen (Exceptions) bezeichnet. Ausnahmen werden von der CPU selbst ausgelöst und sind für das laufende Programm bestimmt.

Asynchrone Interrupts sind die klassischen Interrupt-Typen, die nicht an ein laufendes Programm gebunden sind. Sie treten unabhängig davon auf, was das System gerade macht. Typische Beispiele für asynchrone Interrupts sind die Ankunft einer Nachricht an einem Netzwerkadapter oder die Zustellung eines Plattenspeicherblocks an die CPU. Beide Ereignisse unterbrechen in der Regel für kurze Zeit den Ablauf des laufenden Programms. Asynchrone Interrupts sind nicht vorhersehbar und können auch nicht ohne weiteres reproduziert werden.

13. *Warum verwendet Windows den DPC-Mechanismus?*

Der DPC-Mechanismus (Deferred Procedure Call) wird eingesetzt, um die Bearbeitungszeiten in der ISR möglichst kurz zu halten und damit das System

nicht unnötig zu blockieren. Die adressierte DPC-Routine wird dann später mit niedrigerer Priorität auf IRQ-Level 2 aufgerufen und abgearbeitet.

14. *Welche Aufgabe hat ein Interrupt-Controller?*

Ein Interrupt-Controller erzeugt aus einem IRQ (einer Interruptanforderung) eine Unterbrechung der CPU, die mit Hilfe eines passenden Programmstücks (einer ISR) bearbeitet werden muss.

10.4 Prozesse und Threads

1. *Was ist in der Prozessverwaltung ein PCB, wozu dient er und welche Inhalte hat er? Nennen Sie dabei drei wichtige Informationen, die im PCB verwaltet werden!*

Ein Eintrag in der Prozesstabelle heißt Process Control Block (PCB). Je nach Betriebssystem gibt es deutliche Unterschiede im Aufbau. Einige Informationen sind aber prinzipiell sehr ähnlich. Hierzu gehört u.a die Information zur Identifikation des Prozesses, die Information zum aktuellen Prozesszustand sowie Informationen zu sonstigen Ressourcen, die dem Prozess zugeordnet sind (Dateien, offene Netzwerkverbindungen). Weitere Informationen, die für einen Prozess im PCB verwaltet werden, sind der Programmzähler, der Prozesszustand, die Priorität und die Prozessnummer sowie die aktuellen Registerinhalte.

2. *Threads werden heute von den meisten Betriebssystemen unterstützt. Was versteht man unter einem Thread?*

Ein Thread stellt eine nebenläufige Ausführungseinheit innerhalb eines Prozesses dar. Threads werden im Gegensatz zu den traditionellen (schwergewichtigen, heavy-weight) Prozessen als leichtgewichtige (light-weight) Prozesse oder kurz LWP bezeichnet.

3. *Wie verhalten sich Threads zu Prozessen im Hinblick auf die Nutzung des Prozessadressraums?*

Threads erhalten den vollen Zugriff auf den zugehörigen Prozessadressraum.

4. *Beschreiben Sie den groben Ablauf eines Prozess-Kontextwechsels und erläutern Sie, warum ein Thread-Kontextwechsel schneller sein kann, als ein Prozess-Kontextwechsel!*

Bei einem Prozess-Kontextwechsel wird ein aktiver Prozess unterbrochen. Die gesamte Information des unterbrochenen Prozesses, also der Prozesskontext,

wird gesichert und der Prozesskontext des neu aktivierten Prozesses in die Ablaufumgebung (CPU-Register, MMU-Register,...) geladen.

Ein Thread-Kontextwechsel innerhalb des laufenden Prozesses kann schneller sein als ein Prozess-Kontextwechsel, weil meist Speicherbereiche des gleichen Prozesses verwendet werden. Ein Austausch von Speicherbereichen ist daher oft nicht erforderlich, was den Betriebssystem-Overhead reduziert.

5. *Was versteht man unter User-Level-Threads im Vergleich zu Kernel-Level-Threads und welche Beziehungen zwischen beiden sind möglich?*

User-Level-Threads laufen auf der Benutzerebene (Benutzermodus), Kernel-Level-Threads dagegen auf der Kernelebene (Kernelmodus, Systemmodus) ab.

6. *Was bedeutet eine 1:n-Beziehung zwischen den Betriebsmitteln Prozess und Thread?*

Ein Prozess kann mehrere Threads beherbergen. 1:n-Beziehungen, bei denen ein Prozess nahezu beliebig viele Threads enthalten kann, sind heute in Betriebssystemen üblich.

7. *Erläutern Sie die beiden Möglichkeiten in Java, eigene Threads zu definieren und zu nutzen?*

Es besteht die Möglichkeit, eine eigene Threadklasse zu implementieren, indem man diese von der Basisklasse Thread ableitet. Die Methode run muss überschrieben werden, da dies die Methode ist, die beim Starten eines Threads aufgerufen wird. Die andere Variante, einen Thread zu erzeugen, ist die Definition einer Klasse, die das Interface Runnable implementiert. In der Klasse wird ebenfalls die Methode run implementiert. Als Runnable-Objekt wird die Klasse angegeben, welche das Interface Runnable implementiert. Beide Varianten sind semantisch vollkommen identisch.

8. *In welcher Methode wird die eigentliche Arbeit eines Java-Threads ausgeführt?*

In der Methode run, die über die Methode start aufgerufen wird.

9. *Was passiert beim Aufruf des Systemcalls fork unter Unix?*

Mit dem Systemcall fork wird ein neuer Prozess als Kindprozess des erzeugenden Prozesses erzeugt.

10. *Welche Aufgabe hat ein Thread unter Windows?*

Threads stellen die eigentliche Scheduling-Einheit des Windows-Betriebssystems dar.

11. *Wie kann man in C# eigene Threads definieren und nutzen? Welche Bedeutung hat in diesem Zusammenhang die sog. Start-Methode?*

Im Gegensatz zu Java ist in C# nicht unbedingt eine eigene Klasse zu definieren, um einen Thread zu erzeugen. Es genügt auch die Definition einer Startmethode, die dann bei der Thread-Erzeugung angegeben wird. Der Name der Startmethode ist frei vergebbar.

Ein C#-Thread wird durch Aufruf der Methode *Start* erzeugt. Die angegebene Startmethode wird dann in einem neuen Thread ausgeführt. Wenn die Startmethode sich beendet, ist auch der Thread zu Ende.

12. *Kann es unter Windows sein, dass ein Thread mehreren Prozessen zugeordnet ist? Begründen Sie Ihre Entscheidung!*

Nein, ein Thread wird innerhalb eines Prozesses erzeugt und ist diesem bis zur Terminierung zugeordnet.

13. *Warum ist der Einsatz von Threads sinnvoll?*

Threads sind leichtgewichtiger als Prozesse. Ein Thread-Kontextwechsel innerhalb des laufenden Prozesses kann schneller sein als ein Prozess-Kontextwechsel, weil meist Speicherbereiche des gleichen Prozesses verwendet werden. Ein Austausch von Speicherbereichen ist daher oft nicht erforderlich, was den Betriebssystem-Overhead reduziert.

14. *Welche zwei grundsätzlichen Implementierungsmöglichkeiten für Threads gibt es und welche Vor- bzw. Nachteile haben diese jeweils?*

Threads können auf der Benutzerebene implementiert werden. Vorteil dieser Implementierungsvariante ist die hohe Effizienz, da beim Thread-Kontextwechsel kein Umschalten in den Kernelmodus notwendig ist. Nachteilig ist, dass alle Threads eines Prozesses blockieren, wenn ein Systemaufruf innerhalb eines einzelnen Threads blockiert.

Threads können auch auf der Kernebene implementiert werden. Bei Kernel-Threads werden die Threads im Kernelmodus verwaltet. Die Implementierung von Threads auf Kernebene hat Vorteile, aber auch Nachteile. Von Vorteil ist beispielsweise, dass das Betriebssystem in der Lage ist, die Zuteilung der Rechenzeit über die Threads zu gestalten und so einen Prozess nicht unnötig zu blockieren. Mit dieser Implementierungsvariante kann man auch Multiprozessorsysteme besser unterstützen, da das Betriebssystem ablaufbereite Threads selbstständig auf die verfügbaren CPUs verteilen kann. Ein weiterer Vorteil für Kernel-Threads ist, dass ein Prozess nicht blockiert ist, wenn ein Thread inner-

halb des Prozesses blockiert ist. Ein anderer Thread des Prozesses kann weiterarbeiten. Nachteilig ist beispielsweise, dass im Kernel implementierte Threads nicht so effizient sind, da sie sich bei jedem Thread-Kontextwechsel an den Kernel wenden müssen (Software-Interrupt). Weiterhin ist die größere Systemabhängigkeit von Nachteil.

15. *Beschreiben Sie einen einfachen Zustandsautomaten eines Prozesses!*

Ein einfacher Zustandsautomat für einen Prozess hat vier Zustände. Im Zustand „bereit“ ist der Prozess zur Bearbeitung vorbereitet, im aktiven Zustand hat er eine CPU und im Zustand „blockiert“ wartet er auf Ressourcen, um weitermachen zu können. Im Zustand „beendet“ ist der Prozess dann schon nicht mehr im System vorhanden. Die Zustandsübergänge lassen sich wie folgt beschreiben:

- bereit → aktiv: Das Betriebssystem wählt den Prozess aus (Aktivieren)
- aktiv → bereit: Das Betriebssystem wählt einen anderen Prozess aus (Deaktivieren, Preemption, Vorrangunterbrechung)
- aktiv → blockiert: Der Prozess wird blockiert (z.B. wegen Warten auf Input, Betriebsmittel wird angefordert)
- blockiert → bereit: Der Blockierungsgrund wird aufgehoben (Betriebsmittel verfügbar)
- aktiv → beendet: Prozessbeendigung oder schwerwiegender Fehler (Terminieren des Prozesses)

16. *Erläutern Sie die Prozesshierarchie unter Unix!*

Unix organisiert die Prozesse in einer baumartigen Prozessstruktur, wobei der sog. init-Prozess der Urvater aller zeitlich folgenden Prozesse ist. Der Prozess init hat die PID 1, ein ausgezeichneter Speicherverwaltungs-Prozess namens swapper erhält die PID 0 und ein anderer Speicherverwaltungsprozess page die PID 2.

Bei der Erzeugung eines Prozesses erbt der „Kindprozess“ vom „Elternprozess“ die gesamte Umgebung (inkl. Umgebungsvariablen), alle offenen Dateien und Netzwerkverbindungen sowie den Adressraum mit den Daten- und Codebereichen als Kopie. Er kann dann das gleiche Programm ausführen oder lädt sich bei Bedarf durch Aufruf des Systemcalls `execve` ein neues Programm und überlädt somit das Programm des Elternprozesses.

17. Was ist ein „Zombie-Prozess“ unter Unix?

Ein Zombie ist ein Prozess in einem speziellen Zustand. In diesen Zustand gelangt ein Prozess, der terminieren will. Er verweilt solange in diesem Zustand, bis der Elternprozess eine Nachricht über das Ableben des Kindprozesses erhalten hat und terminiert erst dann.

18. Skizzieren Sie die möglichen Zustandsübergänge eines Windows-Threads (ab Windows 2003) in den Zustand *Runnig*!

Folgende Zustandsübergänge in den Zustand *Running* sind definiert:

- *Ready* --> *Running*: Direkte Zuordnung eines Threads zu einem Prozessor.
- *Deferred Ready* --> *Running*: Zuordnung des ersten Threads zu einem bereits festgelegten Prozessor aus einer speziellen Warteschlange.
- *Standby* --> *Running*: Dieser Zustandsübergang wird für einen Thread durchgeführt, der schon einem Prozessor zugeordnet ist.
- *Waiting* --> *Running*: Dieser Zustandsübergang kommt vor, wenn ein Thread die CPU abgegeben hat, um auf ein Ereignis zu warten, z.B. weil ein Seitenfehler aufgetreten ist. Nach Eintreffen des Ereignisses kann der Thread sofort wieder auf *Running* gesetzt werden.

10.5 CPU-Scheduling

1. Welche Scheduling-Algorithmen sind für Echtzeitbetriebssysteme (Realtime-System) sinnvoll und warum?

Folgende Scheduling-Algorithmen für Realtime-Systeme sind u.a. bekannt:

- Minimal Deadline First
- Polled Loop
- Interrupt-gesteuert

Bei „Minimal Deadline First“ wird der Prozess mit der kleinsten nächsten Zeitschranke (deadline) als erstes ausgewählt. Bei „Polled Loop“ werden alle Geräte (Ereignisquellen) zyklisch nach einem anstehenden Ereignis abgefragt und dieses wird dann gleich bearbeitet. Interrupt-gesteuerte Systeme warten z.B. in einer Warteschleife auf Interrupts von Ereignisquellen und führen dann die geeignete Interrupt-Service-Routine (ISR) aus.

Echtzeitsysteme erfordern andere Strategien bei der Auswahl des nächsten Jobs als batch- und dialogorientierte Systeme. Hier ist vor allem eine schnelle und berechenbare Reaktion auf anstehende Ereignisse wichtig. Es wird zwischen *hard real time* und *soft real time* unterschieden. Erstere müssen schnell reagieren, bei letzteren ist eine gewisse Verzögerung zumutbar.

2. *Welche Aufgaben haben im Prozess-Management der Dispatcher und der Scheduler?*

Die Komponente im Prozessmanager, die für die Planung der Betriebsmittel-zuteilung zuständig ist, heißt Scheduler. Die Komponente, die dann einen tatsächlichen Prozesswechsel ausführt, wird als Dispatcher bezeichnet.

3. *Nennen Sie jeweils zwei geeignete Scheduling-Verfahren für Batch- und Dialog-Systeme und erläutern Sie diese kurz!*

Batchsysteme:

- First Come First Serve (FCFS): FCFS bearbeitet die im System ankommenden Aufträge in der Reihenfolge ihres Eintreffens.
- Shortest Job First (SJF): SJF sucht sich dagegen immer den Job bzw. Prozess aus, von dem es die kürzeste Bedienzeit erwartet.

Dialogsysteme:

- Round Robin (RR): RR ist im Prinzip FCFS (siehe Batch-Strategien) in Verbindung mit einer Zeitscheibe (Quantum genannt). RR geht davon aus, dass alle Prozesse gleich wichtig sind. Ein Prozess erhält ein bestimmtes Quantum, und wenn es abgelaufen ist, wird der Prozess unterbrochen und ein anderer Prozess erhält die CPU. Der unterbrochene Prozess wird hinten in die Warteschlange eingetragen und kommt erst dann wieder an die Reihe, wenn die anderen Prozesse ihr Quantum verbraucht haben oder aus einem anderen Grund unterbrochen wurden. Die Frage nach der Länge der Zeitscheibe ist von großer Bedeutung für die Leistung des Systems.
- Priority Scheduling (PS): PS wählt immer den Prozess mit der höchsten Priorität aus. Dies setzt natürlich die Verwaltung von Prioritäten voraus.

4. *Erläutern Sie den Unterschied zwischen preemptive und non-preemptive Scheduling und nennen Sie jeweils zwei Scheduling-Strategien, die in diese Kategorien passen.*

Im non-preemptive, auch „run-to-completion“-Verfahren genannt, darf ein Prozess nicht unterbrochen werden, bis er seine Aufgaben vollständig erledigt hat. Beispiele: First Come First Serve (FCFS) und Shortest Job First (SJF).

Im Gegensatz dazu darf im preemptive Scheduling eine Unterbrechung stattfinden. Rechenbereite Prozesse können somit suspendiert werden. Dies setzt natürlich eine Strategie zur Vergabe der CPU voraus, die vom Betriebssystem unterstützt werden muss und in der Regel auf der Zeitscheibentechnik basiert. Dieses Verfahren ist für die Unterstützung konkurrierender Benutzer geeignet. Beispiele: Round Robin (RR) und Priority Scheduling (PS).

5. Ermitteln Sie für folgende fünf Jobs die gesamte Verweilzeit und die durchschnittliche Verweilzeit unter Berücksichtigung folgender Scheduling-Strategien (Angaben in ms).

Job	A	B	C	D	E
Ablaufzeit	8	12	20	16	5
Priorität	5	4	3	2	1

- a) Reines Priority Scheduling (höchste Priorität ist 5).
 b) FCFS (First Come First Server) unter Berücksichtigung der Reihenfolge-Annahme: A, B, D, C, E
 c) SJF (Shortest Job First)
 d) RR (Round Robin) ohne Prioritäten bei einem Quantum von 2 ms, wobei die Reihenfolge der Abarbeitung A, B, C, D, E sein soll.

Die Jobs treffen ungefähr gleichzeitig im System ein. Die reine Prozesswechselzeit wird für die Berechnung vernachlässigt und die Aufträge werden nacheinander ausgeführt. Eine Verdrängung (Preemption) wird nur im Fall d) ausgeführt.

zu a) Reines Priority Scheduling:

Job	A	B	C	D	E
Verweilzeit	8	20	40	56	61

Die Summe über alle Verweilzeiten ist $V_{\text{all}} = 8 + 20 + 40 + 56 + 61 = 185$ ms.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 37$ ms.

zu b) FCFS:

Job	A	B	D	C	E
Verweilzeit	8	20	36	56	61

Die Summe über alle Verweilzeiten ist $V_{\text{all}} = 8 + 20 + 36 + 56 + 61 = 181$ ms.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 36,2$ ms.

zu c) SJF:

Job	E	A	B	D	C
Verweilzeit	5	13	25	41	61

Die Summe über alle Verweilzeiten ist $V_{\text{all}} = 5 + 13 + 25 + 41 + 61 = 145$ ms.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 29$ ms.

zu d) RR:

Scheduling-Reihenfolge:

A-B-C-D-E-A-B-C-D-E-A-B-C-D-B-C-D-B-C-D-C-D-C-D-C-C

Addition der Verweilzeiten:

A: $2+2+2+2+2+2+2+2+2+2+2+2+1 = 31$ ms

B: $31 + 14 = 45$ ms

C: $57 + 4 = 61$ ms

D: $45 + 12 = 57$ ms

E: $2+2+2+2+2+2+2+2+2+2+2+2+1 = 29$ ms

Job	A	B	C	D	E
Verweilzeit	31	45	61	57	29

Die Summe über alle Verweilzeiten ist $V_{\text{all}} = 31 + 45 + 57 + 61 + 29 = 223$ ms.

Die durchschnittliche Verweilzeit ist $V_{\text{avg}} = V_{\text{all}} / 5 = 44,6$ ms.

6. *Beweisen Sie, dass die SJF-Strategie für das Scheduling die optimale Strategie darstellt!*

Wir betrachten für den Beweis fünf beliebige Jobs mit den Bezeichnungen A, B, C, D und E und benennen die erwarteten Ausführungszeiten der Einfachheit halber für die folgende Berechnung mit a, b, c, d und e.

Die gesamte Verweilzeit aller Jobs im System ergibt sich dann wie folgt:

$$V_{\text{all}} = a + (a+b) + (a+b+c) + (a+b+c+d) + (a+b+c+d+e) = 5a + 4b + 3c + 2d + e$$

Die durchschnittliche Verweilzeit ergibt sich dann aus

$$V_{\text{avg}} = (5a + 4b + 3c + 2d + e) / 5$$

Der Job A trägt also am meisten zur durchschnittlichen Verweilzeit bei, der Job B steht an zweiter Stelle usw. Will man die durchschnittliche Verweilzeit minimieren, sollte zunächst der Job mit der kürzesten Verweilzeit ausgeführt werden, da die Ausführungszeit des ersten gestarteten Jobs für die Berechnung mit 5 multipliziert wird und damit am meisten Gewicht hat. Danach sollte der zweitkürzeste Job folgen usw.

7. *Nennen Sie Vergleichskriterien, nach denen Scheduling-Algorithmen verglichen werden können und erläutern Sie diese!*

- Wartezeit: Zeit, die ein Prozess auf die Ausführung warten muss, also die Summe aller Zeiträume, in denen ein Prozess warten muss.

- Bedienzeit (Servicezeit): Zeit, in der ein Prozess die CPU hält und arbeiten kann.
- Antwortzeit: Zeit, in der ein Anwender auf die Bearbeitung seines Auftrags warten muss.
- Durchsatz: Anzahl an Prozessen, die ein System in einer bestimmten Zeit bearbeiten kann.
- CPU-Auslastung: Auslastung der CPU während der Bearbeitung von Prozessen in % der Gesamtkapazität.
- Durchlaufzeit = Verweilzeit: Gesamte Zeit, in der sich ein Prozess im System befindet (Servicezeiten + Wartezeiten).

8. Betrachten Sie folgende Aufträge an ein Betriebssystem mit ihren Ausführungszeiten (in ms):

Job	A	B	C	D	E
Ablaufzeit	8	27	1	5	10

Berechnen Sie für die folgenden Scheduling-Algorithmen die durchschnittliche Wartezeit für die Ausführung:

- FCFS in der Reihenfolge A, B, D, C, E
- SJF (kürzester Prozess ist C). Wie das System dies ermittelt ist nicht von Belang für diese Aufgabe.
- RR mit einem Quantum von 2 ms in der Reihenfolge A, B, D, C, E.

Der Overhead für den Prozesswechsel soll nicht betrachtet werden. Die Aufträge werden nacheinander ausgeführt.

zu a) FCFS:

Job	A	B	D	C	E
Wartezeit	0	8	35	40	41

Die Summe über alle Wartezeiten ist $W_{\text{all}} = 0 + 8 + 35 + 40 + 41 = 124$ ms.

Die durchschnittliche Wartezeit ist $W_{\text{avg}} = W_{\text{all}} / 5 = 24,8$ ms.

zu b) SJF:

Job	C	D	A	E	B
Wartezeit	0	1	6	14	24

Die Summe über alle Wartezeiten ist $W_{\text{all}} = 0 + 1 + 6 + 14 + 24 = 45$ ms.

Die durchschnittliche Wartezeit ist $W_{\text{avg}} = W_{\text{all}} / 5 = 9$ ms.

zu c) RR:

Scheduling-Reihenfolge:

A-B-D-C-E-A-B-D-E-A-B-D-E-A-B-E-A-B-E-B-E-B-B-B-B-B-B

Addition der Wartezeiten:

A: $0+2+2+1+2+0+2+2+2+2+1+2 = 18$ ms

B: $2+0+2+1+2+2+0+2+2+2+0+1+2+2+0+2+0+2 = 24$ ms

D: $2+2+0+1+2+2+2+0+2+2+2+0 = 17$ ms

C: $2+2+2+0 = 6$ ms

E: $2+2+2+1+0+2+2+2+0+2+2+1+0+2+2+0+2 = 24$ ms

Job	A	B	D	C	E
Wartezeit	18	24	17	6	24

Die Summe über alle Wartezeiten ist $W_{\text{all}} = 18 + 24 + 17 + 6 + 24 = 89$ ms.

Die durchschnittliche Wartezeit ist $W_{\text{avg}} = W_{\text{all}} / 5 = 17,8$ ms.

9. *Warum sind Windows und Linux keine echten Realtime-Systeme, obwohl sie Realtime-Threads unterstützen?*

Es sind keine echten Realzeitsysteme, da sie keine harten Realzeitbedingungen unterstützen können. Es gibt zwar Prozesse vom Typ „Realtime“, aber ein Kontextwechsel ist auch für diese Prozesstypen nicht deterministisch.

10. *Wie wird unter Windows verhindert, dass Threads mit niedriger Priorität verhungern?*

Unter Windows könnten Threads mit niedrigerer Priorität verhungern, da rechenintensive Threads höherer Priorität immer bevorzugt werden. Daher ist unter Windows noch ein Mechanismus implementiert, der einmal pro Sekunde prüft, ob ein Thread schon 300 ms oder länger nicht mehr die CPU hatte, obwohl er im Zustand „bereit“ ist. Ist dies der Fall, wird seine Priorität auf 15 angehoben und sein Quantum wird verdoppelt. Nachdem er die CPU erhalten hat, wird er wieder auf den alten Zustand gesetzt. Ein Verhungern von Prozessen wird damit also vermieden.

11. *Wie funktioniert der RR-Scheduling-Algorithmus?*

RR (Round Robin) ist im Prinzip FCFS (First Come First Serve) in Verbindung mit einer Zeitscheibe. RR geht davon aus, dass alle Prozesse gleich wichtig sind. Ein Prozess erhält ein bestimmtes Quantum (auch sog. Zeitscheibe oder engl. „time slice“ genannt) und wenn es abgelaufen ist, wird der Prozess un-

terbrochen und ein anderer Prozess erhält die CPU. Der unterbrochene Prozess wird hinten in die Warteschlange eingetragen und kommt erst dann wieder an die Reihe, wenn die anderen Prozesse ihr Quantum verbraucht oder aus einem anderen Grund unterbrochen wurden. Die Frage nach der Länge der Zeitscheibe ist von großer Bedeutung für die Leistung des Systems.

12. *Warum ist der Scheduling-Algorithmus Shortest Remaining Time First (SRTF) kaum zu realisieren?*

SRTF wählt den Prozess mit der kürzesten noch verbleibenden Zeit als nächstes aus und wäre theoretisch für Dialogprozesse optimal. Er ist kaum zu realisieren, da in Betriebssystemen keine Informationen über die verbleibende Prozesszeit vorliegen.

13. *Erläutern Sie den Aufbau und die Nutzung der Datenstruktur Run-Queue unter Linux!*

Die zentrale Datenstruktur des Schedulers wird auch in Linux als Run-Queue bezeichnet. Für jede CPU wird unter Linux eine eigene Run-Queue verwaltet. Ein Loadbalancing-Mechanismus verteilt die Prozesse/Threads auf die CPUs. Die Datenstruktur enthält im Wesentlichen Referenzen (Zeiger) auf die eigentlichen Prozess-Queues. Diese werden repräsentiert durch die Datenstruktur *prio_array* und werden auch als Priority-Array bezeichnet.

Linux unterscheidet eine Prozess-Queue für aktive (*active*) und eine für abgelaufene (*expired*) Prozesse. In der *active Queue* sind alle Prozesse mit einem Quantum > 0 enthalten. In der *expired Queue* werden alle Prozesse, deren Quantum bereits abgelaufen ist, verwaltet. Sobald ein Prozess sein Quantum abgearbeitet hat, wird er in die *expired Queue* umgehängt. Wenn alle Prozesse der *active Queue* ihr Quantum abgearbeitet haben, wird das Quantum für alle Prozesse der *expired Queue* neu ermittelt, und es werden einfach die entsprechenden Zeiger auf die *active* und die *expired Queue*, die in der *Run-Queue* verwaltet werden, ausgetauscht.

14. *Welche Bedeutung haben die statische und effektive (dynamische) Priorität, der Bonus und das Quantum für das Linux-Scheduling?*

Die statische Priorität wird beim Prozessstart vergeben. Die effektive Priorität wird zur Laufzeit ermittelt. Neben der Priorität verwaltet der Kernel für jeden Prozess einen Bonuswert, der zur statischen Priorität addiert wird. Der Bonus beeinflusst die Auswahl des nächsten Prozesses unmittelbar. Der Wertebereich für den Bonus liegt zwischen -5 und $+5$. Ein negativer Wert bedeutet eine Ver-

besserung der Priorität, da 0 die höchste Priorität ist, ein positiver Wert eine Verschlechterung der Priorität. Einen Bonus erhalten Prozesse, die viel „schlafen“, also auf Ein-/Ausgabe warten. Je mehr ein Prozess warten muss, desto höher wird sein Bonus. Mit dem Bonus werden also vor allem interaktive Prozesse begünstigt, die oft auf Eingaben des Benutzers warten. Das Quantum (Länge der Zeitscheibe) wird anhand der statischen Priorität ermittelt. Je höher die Priorität ist, umso höher ist auch das Quantum.

15. *Wie ermittelt der Linux-Scheduler den nächsten zu aktivierenden Prozess?*

Linux verwaltet in der *Run-Queue* für jede Priorität eine Liste von Prozessen im RR-Verfahren. Wenn der aktive Prozess sein Quantum verbraucht hat oder vorher unterbrochen wurde, wird der nächste Prozess aus der RR-Queue mit der höchsten Priorität aktiviert. Erst wenn alle Prozesse einer Prioritäts-Queue abgearbeitet wurden, wird die Queue mit der nächst niedrigen Priorität abgearbeitet. Wird wieder ein Prozess mit höherer Priorität ablaufbereit, wird der laufende Prozess unterbrochen.

16. *Was ist ein Priority Boost unter Windows?*

Wenn ein Thread unter Windows auf Ein-/Ausgabe wartet, wird die Priorität angehoben, damit er nach der Wartezeit schnell wieder die CPU zugeteilt bekommt. Die Anhebung wird als *priority boost* bezeichnet. Es wird maximal auf die Priorität 15 angehoben. Wie stark die Anhebung tatsächlich ausfällt, hängt davon ab, auf was gewartet wird. Letztendlich entscheidet dies der Gerätetreiber des Gerätes, an dem gewartet wird.

Die Priorität wird aber anschließend wieder Zug um Zug herabgesetzt. Je abgelaufenes Quantum wird die Priorität des Threads um 1 reduziert, bis wieder die Basispriorität erreicht ist. Bei der Reduktion ist die Unterbrechung eines Zeitquantums durch höher priorisierte Threads jederzeit möglich. Das Zeitquantum wird dann zunächst aufgebraucht, bevor die Priorität um 1 vermindert wird. Ein erneutes Warten auf Ein-/Ausgabe kann dabei erneut zu einem *Priority Boost* führen.

10.6 Synchronisation und Kommunikation

1. *Was bezeichnet man in der Prozessverwaltung als Blockieren, Verklemmen und Verhungern?*

Blockieren: Ein Prozess P1 belegt ein Betriebsmittel, ein zweiter Prozess P2 benötigt dasselbe Betriebsmittel ebenfalls und wird daher blockiert, bis es P1 freigegeben hat.

Verhungern (Starvation): Ein Prozess erhält trotz Rechenbereitschaft keine CPU-Zeit zugeteilt, z.B. weil ihm immer wieder Prozesse mit höherer Priorität vorgezogen werden.

Verklemmung: Zwei oder mehrere Prozesse halten jeder für sich ein oder mehrere Betriebsmittel belegt und versuchen ein weiteres zu belegen, das aber von einem anderen Prozess belegt ist. Es liegt ein Zyklus von Abhängigkeiten vor. Kein Prozess gibt seine Betriebsmittel frei und alle Prozesse warten daher ewig. Dieser Zustand wird auch als Deadlock bezeichnet.

2. *Bei Betriebssystemen, bei systemnahen Programmen, aber auch bei Anwendungssoftware muss man sich als Entwickler von nebenläufig auszuführenden Aktionen mit kritischen Abschnitten befassen. Was versteht man unter einem kritischen Abschnitt?*

Ein kritischer Abschnitt ist ein Codeabschnitt, der zu einer Zeit nur durch einen Prozess bzw. Thread durchlaufen und in dieser Zeit nicht durch andere nebenläufige Prozesse bzw. Threads betreten werden darf. Ein Prozess bzw. Thread, der einen kritischen Abschnitt betritt, darf nicht unterbrochen werden. Sofern das Betriebssystem in dieser Zeit aufgrund einer Scheduling-Entscheidung eine Unterbrechung zulässt, darf der Prozess bzw. Thread, der den kritischen Abschnitt belegt, durch andere Prozesse, die die CPU erhalten, nicht beeinflusst werden.

3. *Welche Maßnahmen sind zu treffen, damit es beim Durchlaufen eines kritischen Abschnitts nicht zu Inkonsistenzen kommt? Gehen Sie dabei auf den Begriff des gegenseitigen Ausschlusses (mutual exclusion) ein!*

Um Inkonsistenzen zu vermeiden, muss ein kritischer Abschnitt geschützt werden. Dies kann durch gegenseitigen (oder wechselseitigen) Ausschluss (engl.: mutual exclusion) erreicht werden. Prozesse, die einen kritischen Abschnitt ausführen wollen, müssen warten, bis dieser frei ist. Mit einem wechselseitigen Ausschluss wird also die Illusion einer atomaren Anweisungsfolge geschaffen, denn echt atomar wird sie natürlich nicht ausgeführt. Es kann ja

immer noch vorkommen, dass ein nebenläufiger Prozess zwischendurch die CPU erhält.

4. *Was sind Semaphore? Gehen Sie dabei kurz auf die Semaphoreoperationen P() und V() ein!*

Das Semaphore-Konzept ist ein Konzept zur Lösung des Mutual-Exclusion-Problems auf Basis von Sperren. Ein Semaphore verwaltet intern eine Warteschlange für die Prozesse bzw. Threads, die gerade am Eingang eines kritischen Abschnitts warten müssen, und einen Semaphorezähler. Es kommt auf die Initialisierung des Semaphorezählers an, wie viele Prozesse in den kritischen Abschnitt dürfen. Für den Eintritt in den bzw. Austritt aus dem kritischen Abschnitt gibt es zwei Operationen:

- P() wird beim Eintritt in den kritischen Abschnitt aufgerufen. Der Semaphorezähler wird um 1 reduziert, sofern er größer als 0 ist. Wenn er gerade auf 0 steht, wird der Eintritt verwehrt, der Prozess/Thread wird in die Warteschlange eingereiht und suspendiert.
- V() wird beim Verlassen des kritischen Abschnitts aufgerufen. Der Semaphorezähler wird wieder um 1 erhöht, so dass ein weiterer Prozess/Thread in den kritischen Abschnitt darf.

5. *Warum müssen die Semaphore-Operationen P() und V() selbst wieder ununterbrechbar sein?*

Semaphore-Operationen müssen selbst wieder ununterbrechbar, also atomar sein, weil eine Unterbrechung zu Inkonsistenzen in der Warteschlangenbearbeitung oder im Semaphorezähler führen kann. Die korrekte Implementierung eines Semaphors benötigt daher am besten auch eine unteilbare Hardware-Operation wie etwa TSL.

6. *Welche Auswirkung hat die folgende, falsche Nutzung eines binären Semaphors?*

```
V(); ... kritischer Abschnitt ...; P();
```

Diese Anweisungsfolge führt dazu, dass alle Prozesse nur noch im kritischen Abschnitt zugelassen sind. Sie kommen alle problemlos in den kritischen Abschnitt hinein.

7. *Welche Auswirkung hat die folgende, falsche Nutzung eines binären Semaphors?*

```
P(); ... kritischer Abschnitt ...; P();
```

Diese Anweisungsfolge führt dazu, dass nach kurzer Zeit kein Prozess mehr in den kritischen Abschnitt darf und die ersten Prozesse möglicherweise auch im kritischen Abschnitt verbleiben.

8. Welche vier Kriterien sind nach Dijkstra für die Behandlung kritischer Abschnitte zu beachten?

Nach Dijkstra ist bei kritischen Abschnitten folgendes zu beachten:

- Mutual exclusion: Zwei oder mehr Prozesse dürfen sich nicht gleichzeitig im gleichen kritischen Abschnitt befinden.
- Es dürfen keine Annahmen über die Abarbeitungsgeschwindigkeit und die Anzahl der Prozesse bzw. Prozessoren gemacht werden. Der kritische Abschnitt muss unabhängig davon geschützt werden.
- Kein Prozess außerhalb eines kritischen Abschnitts darf einen anderen nebenläufigen Prozess blockieren.
- Fairness Condition: Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen (kein ewiges Warten).

9. Der Ablauf zweier konkurrierender Prozesse wird in Kapitel 6 als Petrinetz dargestellt. Beantworten Sie hierzu folgende Fragen:

a) Über welche Stelle wird der kritische Abschnitt realisiert und warum?

Die Stellen S12 und S22 stellen kritische Abschnitte dar, da nur ein Prozess zu einer Zeit eine der Stellen belegen kann.

b) Welche Transitionen sind Ein- bzw. Ausgänge des kritischen Abschnitts?

Die Eingänge in den kritischen Abschnitt sind die Transitionen t11 und t21, die Ausgänge sind t12 und t22.

c) Welche Stelle bewirkt den gegenseitigen Ausschluss?

Die Stelle S0 realisiert den gegenseitigen Ausschluss, da die Transitionen t11 und t21 nur alternativ, also einander ausschließend schalten können.

d) Welche Stellen bezeichnen einen unkritischen Abschnitt?

Die Stellen S11 und S21 stellen unkritische Abschnitte dar.

10. Was ist ein Mutex im Sinne der Synchronisation konkurrierender Prozesse?

Vereinfachte binäre Semaphore werden auch als Mutex bezeichnet. Ein Mutex verfügt praktisch über eine Variable, die nur zwei Zustände, nämlich locked und unlocked annehmen kann, und verwaltet keine Warteschlange für wartende Prozesse.

11. Semaphore zur Kontrolle des Eintritts in einen kritischen Abschnitt müssen effizient implementiert werden. Nehmen Sie zu dieser Aussage Stellung und betrachten Sie dabei die Implementierungsvariante mit Polling (Busy Waiting).

Semaphor-Operationen werden meist von nebenläufigen Prozessen häufig durchlaufen und können somit bei ineffizienter Implementierung Auswirkungen auf die Leistung haben. Würde die P()-Operation mit Polling realisiert, so würde ein Prozess, der gerade am Eingang des kritischen Abschnitts wartet, den Rechnerkern nicht freigeben, bis sein Quantum abgelaufen ist. Daher ist Polling eine schlechte Implementierungsvariante.

12. Ein TSL-Befehl (Test and Set Lock) wird verwendet, um atomare Aktionen zu implementieren.

a) Welche Operanden hat ein TSL-Befehl?

Ein Register (z.B. R1) und eine Speicherzelle z.B. mit dem Namen LOCK (Befehl: TSL R1, LOCK).

b) Welche Werte können die Operanden annehmen und was bedeuten sie?

Das Register und die Hauptspeicherzelle können jeweils die Werte 0 und 1 annehmen. Es gelten folgende Regeln: Der Inhalt der Speicherzelle wird mit dem TSL-Befehl in das Register geladen, wobei der Wert der Speicherzelle auf 1 verändert wird. Das Register enthält nach der Operationsausführung den alten Wert der Speicherzelle, und dieser Wert kann anschließend dahingehend überprüft werden, ob der Lock schon vorher gesetzt war oder nicht. Wenn das Register nun den Wert 1 enthält, dann ist die Sperre schon gesetzt gewesen, 0 bedeutet, der laufende Prozess erhält die Sperre.

13. Zeigen Sie, wie man den TSL-Befehl für die Implementierung einer Sperre mit den Operationen lock und unlock verwenden kann. Skizzieren Sie dabei grob die Algorithmen in einem Assembler-ähnlichen Pseudocode!

MyLock_lock:

```
TSL R1, LOCK           // Lies LOCK und setze Wert von LOCK auf 1
CMP R1, #0             // Vergleiche Registerinhalt von R1 mit 0
JNE MyLock_lock        // Erneut versuchen, falls Lock nicht gesetzt
                        // werden konnte
RET                    // Kritischer Abschnitt kann betreten werden
```

MyLock_unlock:

```
MOVE LOCK, #0          // LOCK auf 0 setzen (freigeben)
RET                    // Kritischer Abschnitt kann von anderem
                        // Prozess betreten werden
```

14. Nennen Sie die vier notwendigen und hinreichenden Bedingungen, die zu einem Deadlock führen!

Ein Deadlock kann nur eintreten, wenn folgende vier Bedingungen eintreffen:

- Mutual Exclusion für die benötigten Betriebsmittel.
- Prozesse belegen Betriebsmittel und fordern weitere an.
- Kein Entzug eines Betriebsmittels ist möglich.
- Zwei oder mehrere Prozesse warten in einer verketteten Liste (in einer Schleife, circular waiting) auf weitere Betriebsmittel.

15. Warum lassen sich Deadlocks bei nebenläufigen Prozessen schwer ganz vermeiden?

Deadlock-Vermeidung wird durch die Aufweichung mindestens eines der Kriterien erreicht: Deadlocks lassen sich a priori aber nur schwer vermeiden, weil sonst nur jeder Prozess ohne Gefahr eines Deadlocks zum Ablauf kommen dürfte. Alle Betriebsmittel müssten im Vorfeld reserviert werden.

16. Welche Alternative zur Behandlung von Deadlocks kennen Sie?

Eine in der Praxis häufig eingesetzte Technik ist das Erkennen und Beseitigen von Deadlocks zur Laufzeit, wobei man hierzu sog. Betriebsmittelbelegungsgraphen einsetzt. Dies sind Graphen, die als Knoten Ressourcen und Prozesse/Threads enthalten, und Kanten, welche die Belegung der Ressourcen durch Prozesse/Threads aufzeigen. Als Maßnahmen zur Beseitigung eines Deadlocks sind das Abbrechen eines Prozesses/Threads oder das Entziehen eines Betriebsmittels möglich.

17. Erläutern Sie die Arbeitsweise der Java-Synchronisationsprimitive *synchronized*!

Die Synchronisationsprimitive *synchronized* ermöglicht eine Zugriffsserialisierung, die auf Klassen- oder Objektebene erfolgt, je nachdem, ob man sie auf statische Methoden oder Instanzmethoden anwendet. Im Folgenden ist der Rumpf einer synchronisierten Methode dargestellt.

```
public synchronized void method1()
{
    ...
}
```

Auch die Nutzung von *synchronized* in Anwendungsblöcken ist möglich, wobei hier ein Objekt angegeben wird, für das der Zugriff zu serialisieren ist. Verwendet man als Objekt *this*, gilt auch hier die Serialisierung für das ganze Objekt:

```
synchronized (object1)
{
```

```
}  
...
```

Die Aufrufe einer synchronisierten Methode oder der Zugriff auf das mit `synchronized` gekennzeichnete Objekt werden strikt serialisiert und verzögert, wenn ein anderer Thread gerade im synchronisierten Code ist. Damit lässt sich ein kritischer Abschnitt deklarativ implementieren.

18. Nennen Sie Kritikpunkte der Java-Umsetzung des Monitor-Konzepts von Hoare oder Brinch Hansen. Wo liegen die Schwachstellen und wie kann man sie umgehen?

Es ist keine reine Umsetzung des Monitorkonzepts nach Hoare oder Brinch Hansen, da nicht alle `public`-Methoden zwingend als `synchronized` deklariert sein müssen und auch nicht nur `private` Variablen zugelassen sind. Weiterhin gibt es auch die Möglichkeit, öffentliche Variablen zu deklarieren, auf die ein beliebiger Zugriff von außen möglich ist. Brinch Hansen kritisiert dies, da der Compiler hier keine Möglichkeit der Überprüfung hat.

Man kann die Probleme durch disziplinierte Programmierung umgehen (keine Variable als `public` und alle Methoden mit `synchronized` deklarieren).

19. Was sind Race Conditions?

Race Conditions sind Situationen, bei denen zwei oder mehr Prozesse gemeinsame Betriebsmittel nutzen und die Endergebnisse der Nutzung von der zeitlichen Reihenfolge der Operationen abhängen.

20. Nehmen wir an, zwei Prozesse P1 und P2 greifen auf zwei gemeinsame globale Variablen `a` und `b` lesend und verändernd in der folgenden Form zu.

```
// Globale Variablen  
int a = 10;  
int b = 0;
```

Prozess P1:

```
(1.1)  b = 5;  
(1.2)  a = a + b;
```

Prozess P2:

```
(2.1) b = -5;  
(2.2) a = a + b;
```

Zeigen Sie zwei nebenläufige Ablaufsequenzen auf, die jeweils zu einem anderen Wert der Variable `a` führen! Wie könnte man diese Situation vermeiden?

Ablaufsequenz 1:

```
P1: b = 5;  
P1: a = a + b; (a = 15)  
P2: b = -5; (b = -5)
```

P2: $a = a + b$; ($a = 10$)

Ergebnis: $b = -5$, $a = 10$

Ablaufsequenz 2:

P1: $b = 5$;

P2: $b = -5$;

P1: $a = a + b$; ($a = 5$)

P2: $a = a + b$; ($a = 0$)

Ergebnis: $b = -5$, $a = 0$

Die Situation lässt sich durch die Nutzung eines Synchronisationsmechanismus vermeiden, über den man einen kritischen Abschnitt definiert, in dem die Variablen a und b bearbeitet werden dürfen. Durch diesen kritischen Abschnitt müssen beide Prozesse. Die Realisierung kann z.B. mit einem einfachen Lock, einem Mutex, einem Semaphor oder einem Monitor erfolgen.

21. *Vergleichen Sie die Funktionsweise des Java-Schlüsselwortes `synchronized` mit der Funktionsweise eines C#-Monitors! Gehen Sie dabei darauf ein, wie die Sperrmechanismen greifen!*

Auch bei den C#-Monitoren handelt es sich nicht um reine Monitore nach Hoare oder Brinch Hansen. Hier gelten im Wesentlichen die gleichen Argumente wie bei dem Java-Schlüsselwort `synchronized`. Beispielsweise sind nicht alle öffentlichen Methoden einer Klasse in die Synchronisationskontrolle eingeschlossen und man kann in C# auch nicht nur private Variablen zulassen. Der Compiler übernimmt keine weitere Prüfung in dieser Hinsicht.

Gesperrt wird bei Monitoren auch auf Objektebene, d.h. in der Enter-Methode wird ein Objekt angegeben, für das eine Sperre gesetzt werden soll. Das Sperren und Entsperren wird atomar durchgeführt.

22. *Was ist der Unterschied zwischen synchroner und asynchroner Kommunikation?*

Synchron bedeutet in diesem Zusammenhang, dass Sender und/oder Empfänger blockieren, um sich abzustimmen. Ein Sender sendet z.B. eine Nachricht über den Aufruf eines Dienstes (`send`) ab und wartet so lange, bis der Empfänger die Nachricht über einen weiteren Dienst (`receive`) gelesen hat. Umgekehrt kann es so organisiert sein, dass ein Empfänger zunächst einen `receive`-Dienst aufruft und damit blockiert, bis ein Sender die Nachricht an ihn sendet.

Bei der asynchronen Kommunikation wird im Kommunikationssystem ein Puffer bzw. eine Nachrichtenwarteschlange verwaltet, die es dem Sender ermöglicht, die Nachricht dort abzulegen und mit anderen Aufgaben ohne Blo-

ckieren weiterzumachen. Der Empfänger kann bei Gelegenheit die Nachricht aus der Nachrichtenwarteschlange lesen und verarbeiten.

23. *Was ist Multicast-Kommunikation im Unterschied zu Broadcast und Anycast?*

- Multicast: Hier adressiert der Sender eine definierte Gruppe von Empfängern. Eine Nachricht wird also an die gesamte Gruppe gesendet (Beziehung 1:n, wobei n die Anzahl der Gruppenmitglieder ist).
- Broadcast: Diese Adressierung sendet Nachrichten an alle möglichen Empfänger, z.B. alle Partner in einem Netzwerk (remote) oder auf einen bestimmten Rechner (lokal), (Beziehung 1:m, wobei m die Anzahl aller möglichen Partner ist).
- Anycast: Diese Art der Empfängeradressierung wendet sich auch an eine Gruppe, allerdings nimmt hier mindestens ein Empfänger, aber nicht zwangsweise alle Empfänger die Nachrichten entgegen (Beziehung 1:1...n).

24. *Was versteht man unter einem Halbduplex- im Gegensatz zu einem Vollduplex-Betrieb?*

Bei der nachrichtenbasierten Kommunikation unterscheidet man, ob beide Partner gleichzeitig senden können (bidirektional), oder nur einer nach dem anderen (unidirektional). Man spricht im ersten Fall von Vollduplex- und im zweiten Fall von Halbduplex-Betrieb.

25. *Erläutern Sie kurz die Funktionsweise von Pipes!*

Wie der Name andeutet handelt es sich bei einer Pipe um einen Datenstrom von einem Prozess zu einem anderen. Daten, die ein Prozess in ein Ende der Pipe schreibt, können auf der anderen Seite von einem anderen Prozess gelesen werden. Eine Pipe stellt damit einen unidirektionalen Datenstrom bereit. Die Daten werden von einem Prozess ohne Nachrichtenbegrenzer in die Schreibseite eingetragen, die von einem lesenden Prozess aus dem anderen Ende (Leseseite) entnommen werden können. Eine bidirektionale Kommunikation zweier Prozesse kann über die Einrichtung von zwei Pipes erreicht werden. Eine Pipe stellt einen Pufferbereich fester Länge dar. Das Auslesen erfolgt in der Reihenfolge, in der die Daten eingetragen werden (FIFO-Puffer).

10.7 Hauptspeicherverwaltung

1. *Welche grundlegenden Gedanken stecken hinter dem Konzept des virtuellen Speichers?*

Folgende Gedanken sind grundlegend für die virtuelle Speichertechnik:

- Ein Prozess sollte auch dann ablaufen können, wenn er nur teilweise im Hauptspeicher ist. Wichtig ist hierbei, dass die Teile des Prozesses (Daten und Code) im physikalischen Speicher sind, die gerade benötigt werden.
- Der Speicherbedarf eines Programms sollte größer als der physikalisch vorhandene Hauptspeicher sein können.
- Ein Programmierer sollte am besten nur einen kontinuierlichen (linearen) Speicherbereich beginnend bei Adresse 0 sehen und sich nicht um die Zerstückelung (Fragmentierung) des Hauptspeichers auf mehrere Benutzer kümmern.

2. *Welche Aufgabe hat eine Verdrängungsstrategie und was ist ein Seitenfehler?*

Die Strategie, die verwendet wird, um eine zu verdrängende Seite auszuwählen, wird als Seitenersetzungsstrategie (Replacement-Strategie) bezeichnet.

Ein page fault (Seitenfehler) ist ein Trap, den die MMU erzeugt, wenn von einem Prozess eine physikalische Adresse angesprochen wird, die nicht im Hauptspeicher geladen ist. Das Betriebssystem springt in den Kernelmodus und zwar auf eine spezielle Interruptroutine zur Bearbeitung des Seitenfehlers und versucht unter Beachtung der Seitenersetzungsstrategie und der Vergabestrategie die Seite in einen Frame zu laden.

3. *Was bedeutet Demand-Paging und welche Alternative wäre denkbar?*

Demand Paging ist eine Abrufstrategie, bei der die Einlagerung nur auf Anforderung, also wenn die Daten benötigt werden, durchgeführt wird.

Eine alternative Strategie ist Prepaging. Im Gegensatz zu Demand Paging werden hier Seiten in den Hauptspeicher geholt, die noch nicht angefordert wurden.

4. *Wie viele Seitentabellen müssen bei virtueller Adressierung vom Betriebssystem verwaltet werden, wenn mit einer einstufigen Seitentabelle gearbeitet wird und gerade 10 Prozesse aktiv sind?*

Es müssen 10 Seitentabellen verwaltet werden, eine je Prozess.

5. *Ein virtueller Adressraum wird mit 32 Bit langen virtuellen Adressen adressiert. Eine virtuelle Adresse enthält jeweils 10 Bit für den Index in der Top-Level-Seitentabelle und 10 Bit für den Index in der Second-Level-Seitentabelle.*

a) *Wie viele Second-Level-Seitentabellen gibt es maximal je Prozess?*

$2^{10} = 1024$ je Prozess, da 10 Bit für den Index der Top-Level-Tabelle zur Verfügung stehen. Die 1024 Einträge verweisen auf 1024 Second-Level-Tabellen.

b) *Wie groß sind die Seitenrahmen im Hauptspeicher?*

2^{12} Byte = 4096 Byte, da 12 Bit für die Größe einer virtuellen Seite zur Verfügung stehen und die Größe einer physikalischen Seite (Frame) der Größe der virtuellen Seite entspricht.

c) Wie groß ist der gesamte virtuelle Adressraum eines Prozesses?

2^{32} Byte = 4 GB, da insgesamt 32 Bit für die Adressierung zur Verfügung stehen.

6. Welcher Seitenersetzungsalgorithmus wäre theoretisch optimal und warum ist dieser praktisch nur schwer zu realisieren?

Ein optimaler Algorithmus würde die Seitenrahmen für eine Ersetzung auswählen, die am spätesten von allen belegten Seitenrahmen wieder benötigt würden. Er ist als „Algorithmus von Belady“ bekannt.

Der Grund für die schwierige bzw. unmögliche Realisierung ist, dass es nicht praktikabel ist, das Verhalten eines Prozesses hinsichtlich seiner Speicherzugriffe sicher vorher zu bestimmen. Das Betriebssystem kann zum Zeitpunkt des Seitenfehlers nicht wissen, wann welche Seite als nächstes verwendet wird.

7. Warum ist der LRU-Algorithmus zur Seitenersetzung bei der virtuellen Adressierung schwer zu realisieren? Welcher Algorithmus ist eine gute Alternative hierzu?

Damit man die am längsten unbenutzte Seite schnell im Zugriff hat, muss man einigen Aufwand bei jedem Zugriff leisten. Eine Möglichkeit ist die Verwaltung einer nach der zeitlichen Nutzung sortierten linearen Liste, bei der die am längsten nicht benutzte Seite im ganz obersten Element liegt. Der hohe Aufwand ergibt sich durch das Umhängen der Elemente bei jedem Zugriff. Die aktuell genutzte Seite muss nämlich bei jedem Zugriff an das Ende der Liste gehängt werden. Aufgrund des hohen Aufwands implementiert man heute meistens sog. Pseudo-LRU-Algorithmen, wobei auch das R- und das M-Bit benutzt werden. Zwei dieser Algorithmen sind Clock-Page- und Second-Chance.

Eine gute Alternative zum LRU-Algorithmus ist der NFU-Algorithmus (Not-Frequently Used) in Verbindung mit einem passenden Aging-Mechanismus.

8. Welche Probleme ergeben sich bei sehr großen virtuellen Adressräumen in Bezug auf die Verwaltung der Seitentabellen?

Die Umsetzung einer virtuellen Adresse auf eine reale Adresse ist im Vergleich zur klassischen Adressierung mit zusätzlichen Hauptspeicherzugriffen auf die Seitentabellen verbunden und verschlechtert damit die Leistungsfähigkeit. Anstatt eines Hauptspeicherzugriffs auf die adressierten Daten sind bei zweistufiger Adressierung drei Hauptspeicherzugriffe erforderlich, da neben den

adressierten Daten auch die Top-Level- und die Second-Level-Seitentabelleneinträge aufgesucht werden müssen. Hinzu kommt, dass mit zunehmender Adressraumgröße auch der Speicherplatz für die Verwaltung der Seitentabellen enorm wächst.

9. *Welche Möglichkeiten gibt es, die hohe Belastung der Seitentabellenverwaltung insbesondere bei großen Adressräumen zu optimieren?*

Zur Optimierung der Speicherzugriffe gibt es Adressumsetzpuffer (Translation Lookaside Buffers, kurz: TLB) und die invertierten Seitentabellen.

10. *Erläutern Sie das Working-Set-Verfahren, das Windows einsetzt!*

Jeder Prozess erhält ein sog. Working-Set, also eine Arbeitsmenge an Frames, die im Rahmen einer Working-Set-Strategie zugeordnet werden. Jeder Prozess verfügt über einen Working-Set von mindestens 20 bis 50 Seiten und maximal 40 bis 345 Seiten je nach verfügbarem Hauptspeicher. Stark „pagende“ Prozesse können ihren Working-Set auch erhöhen, aber nur bis maximal 512 Seiten.

11. *Erläutern Sie die Grundprinzipien des TLB und der invertierten Seitentabelle mit Hashing!*

Ein Adressumsetzpuffer ist ein eigener, schneller Speicher, der eine Tabelle darstellt, in der eine Zuordnung von virtuellen auf reale Adressen verwaltet wird. Es werden die aktuell am häufigsten benutzten Seitennummern in die Tabelle eingetragen. Bei jeder Adressumsetzung wird zunächst geprüft, ob die aktuelle virtuelle Adresse im Adressumsetzpuffer liegt. Wenn ja (TLB hit), muss kein Zugriff auf die Seitentabellen erfolgen, da die reale Adresse des gesuchten Frames schon ermittelt worden ist und man sich somit weitere Hauptspeicherzugriffe erspart. Im anderen Fall (TLB miss) muss wie bisher über die klassische Adressumsetzung über die Seitentabelle(n) ein Mapping auf die physikalische Adresse vorgenommen werden.

Bei invertierten Seitentabellen mit Hashing erfolgt die Suche nach einer Seite nicht über die klassische Seitentabellenindizierung, sondern über eine Hash-Tabelle, in der die virtuelle Seitennummer als Suchkriterium dient. Die invertierte Seitentabelle wird dann i.d.R. komplett im Hauptspeicher gehalten und muss nicht, wie bei großen „herkömmlichen“ Seitentabellen, ausgelagert werden.

12. *Was versteht man unter einem Shared Memory? Nennen Sie Einsatzmöglichkeiten!*

Ein Shared Memory ist ein von mehreren Prozessen gemeinsam genutzter Speicherbereich, der nur einmal in den Hauptspeicher geladen wird. Auf gemeinsam genutzte Speicherbereiche verweisen mehrere Seitentabelleneinträge verschiedener Prozesse.

Shared Memory kann genutzt werden, um Codeteile, die mehrere Prozesse benötigen, nur einmal in den Hauptspeicher zu laden. Hier spricht man von Shared Libraries. Weiterhin kann man Shared Memory verwenden, um mehreren Prozessen globale Datenbereiche zur Verfügung zu stellen. Mit Hilfe dieser Speicherbereiche kann dann eine Art Prozesskommunikation stattfinden (Beispiel: Caches von Datenbankmanagementsystemen, auf die mehrere Datenbank-Benutzerprozesse nebenläufig zugreifen).

13. *Welche grundsätzlichen Speicherverwaltungsmechanismen sind in Unix-Systemen vorzufinden?*

Grundsätzlich wird die virtuelle Speichertechnik mit Demand Paging als Strategie für das Einlesen von Daten (Abrufstrategie) verwendet. Meist werden mehrstufige Seitentabellen verwaltet. Zur Seitenersetzung wird in Unix-Systemen oft eine Art Clock-Page-Algorithmus verwendet.

14. *Das Betriebssystem Windows verwendet eine recht aufwändige Logik zur Verwaltung der Frames in der Speicherverwaltung. Frames werden in sog. Auslagerungslisten geführt.*

(a) *Welche Auslagerungslisten gibt es und welche Typen von Frames werden dort verwaltet?*

Folgende Auslagerungslisten werden verwaltet:

- Die *Modified-Page-List* enthält Seiten, die bereits für eine Seitenersetzung ausgewählt, aber noch nicht ausgelagert wurden. Diese Seiten sind noch einem Prozess zugeordnet.
- Die *Standby-Page-List* enthält Frames, die bereits sauber gemacht wurden, also eine Kopie auf der Paging-Area haben und noch einem Prozess zugeordnet sind.
- Die *Free-Page-List* enthält saubere Frames, die keine Zuordnung zu einem Prozess haben.
- Die *Zeroed-Page-List* enthält Frames, die bereits mit Nullen initialisiert wurden.

(b) *Erläutern Sie, welche System-Threads bei der Verwaltung der Auslagerungslisten mitwirken!*

Es gibt einige Systemthreads, die unter dem Prozess mit dem symbolischen Namen *System* ablaufen und für die Hauptspeicherverwaltung gewisse Aufgaben erledigen. Sie versuchen u.a. nach bestimmten Verfahren, freien Speicher zu schaffen und sind für die Verwaltung der dargestellten Listen zuständig. Vor allem sind dies folgende Threads:

- Ein zyklisch arbeitender Working-Set-Manager-Thread wird einmal pro Sekunde oder bei Bedarf aufgerufen, um den Arbeitsspeicher frei zu machen. Geänderte Frames werden auf die Paging-Area geschrieben.
- Der *Swapper-Thread* läuft alle paar Sekunden, sucht nach Threads, die schon länger nichts tun und legt deren Frames in die Modified- oder Standby-Page-List.
- Der *Modified-Page-Writer-Thread* schreibt die Frames aus der Modified-Page-List in die Paging-Area, wenn diese Liste verkleinert werden muss.
- Der *Zero-Page-Thread* befüllt die Frames der Free-Page-List mit Nullen und hängt sie in die Zero-Page-List um.

15. *Wozu dient die Buddy-Technik in der Speicherverwaltung und wie funktioniert sie?*

Die Buddy-Technik ist eine Placement-Strategie und dient der Verwaltung der freien Seitenrahmen. Bei dieser Technik werden die freien Speicherbereiche immer zu zwei Partnern zusammengefasst, die man als Buddies bezeichnet. Beide Partner können unabhängig voneinander vergeben werden. Sofern beide frei sind, fasst die Speicherverwaltung sie zu einem großen Paar zusammen, das gewissermaßen als Buddy der nächsten Stufen dient.

Mit dieser Technik steigt die Wahrscheinlichkeit, dass bei einer Speicheranforderung die richtigen Speichergrößen zur Verfügung stehen, und es wird das Fragmentierungsproblem reduziert.

16. *Es seien 3 RAM-Seitenrahmen im Hauptspeicher verfügbar, der Rest der Seiten wird auf einer Paging-Area abgelegt. Folgende Referenzreihenfolge, in der die Seiten mit den Nummern 0 bis 6 angesprochen werden, sei gegeben:*

0-1-2-3-4-0-1-5-6-0-1-2

Skizzieren Sie den optimalen Algorithmus (Belady) für die Seitenersetzung und zählen Sie die Anzahl der Seiterersetzungen.

10 Lösungen zu den Übungsaufgaben

Referenz	0	1	2	3	4	0	1	5	6	0	1	2
Hauptspeicher	0	0	0	0	0	0	0	0	0	0	0	0
	-	1	1	1	1	1	1	1	1	1	1	1
	-	-	2	(3)	(4)	4	4	(5)	(6)	6	6	(2)
Paging-Area				2	2	2	2	2	2	2	2	6
					3	3	3	3	3	3	3	3
								4	4	4	4	4
									5	5	5	5

Anzahl der Seitenersetzungen: 5

- a) Skizzieren Sie die Seitenersetzungen unter Nutzung des FIFO-Verfahrens und ermitteln Sie die Anzahl der Seitenersetzungen.

Referenz	0	1	2	3	4	0	1	5	6	0	1	2
Hauptspeicher	0	0	0	(3)	3	3	(1)	1	1	(0)	0	0
	-	1	1	1	(4)	4	4	(5)	5	5	(1)	1
	-	-	2	2	2	(0)	0	0	(6)	6	6	(2)
Paging-Area				0	0	2	2	2	2	2	2	6
					1	1	3	3	3	3	3	3
								4	4	4	4	4
									0	1	5	5

Anzahl der Seitenersetzungen: 9

17. Es seien 3 RAM-Seitenrahmen im Hauptspeicher verfügbar, der Rest der Seiten wird auf einer Paging-Area abgelegt. Folgende Referenzreihenfolge, in der die Seiten mit den Nummern 1 bis 7 angesprochen werden, sei gegeben:

2-5-6-2-1-2-5-3-2-7-2

Zeigen Sie, dass bei FIFO-Seitenersetzung 6, bei LRU 4 und bei der optimalen Strategie nach Belady 3 Seitenersetzungen notwendig sind, um die Referenzreihenfolge abzuarbeiten.

FIFO:

Referenz	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	2	2	2	(1)	1	1	(3)	3	3	3
	-	5	5	5	5	(2)	2	2	2	(7)	7
	-	-	6	6	6	6	(5)	5	5	5	(2)
Paging-Area					2	5	6	6	6	6	6
								1	1	1	1
										2	5

Anzahl der Seitenersetzungen: 6

LRU:

Referenz	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	2	2	2	2	2	2	2	2	2	2
	-	5	5	5	(1)	1	1	(3)	3	3	3
	-	-	6	6	6	6	(5)	5	5	(7)	7
Paging-Area					5	5	6	6	6	6	6
								1	1	1	1
										5	5

Anzahl der Seitenersetzungen: 4

Belady:

Referenz	2	5	6	2	1	2	5	3	2	7	2
Hauptspeicher	2	2	2	2	2	2	2	2	2	2	2
	-	5	5	5	5	5	5	(3)	3	3	3
	-	-	6	6	(1)	1	1	1	1	(7)	7
Paging-Area					6	6	6	6	6	6	6
								5	5	5	5
										1	1

Anzahl der Seitenersetzungen: 3

10.8 Geräte- und Dateiverwaltung

1. Welche Aufgaben hat ein Gerätetreiber?

Ein Gerätetreiber hat u.a. folgende Aufgaben zu erfüllen:

- Er definiert ein Gerät und macht es dem Betriebssystem bekannt, initialisiert also die Geräte-Controller beim Systemstart.
- Er stellt ein logisches Programmiermodell bereit und übersetzt es in gerätespezifische Anforderungen.
- Er dient der Pufferung von Daten auf dem Weg vom Gerät zum Hauptspeicher und umgekehrt.
- Er übernimmt die Unterbrechungsbearbeitung für ein Gerät. In der Regel liegt im Treiber auch die passende Interrupt Service Routine (ISR) für das Gerät.
- Er dient der Koordination der nebenläufigen Zugriffe auf ein Gerät.

2. Erläutern Sie den prinzipiellen Aufbau einer Festplatte!

Magnet- oder Festplatten bestehen aus Kunststoff- oder Aluminiumscheiben, die mit einer hauchdünnen Magnetschicht überzogen sind, und meistens sind auch mehrere Schichten übereinander angeordnet. Eine beidseitige Plattenbeschichtung ist heute der Normalfall. Die Schreib-/Leseköpfe sind auf einem mechanischen Träger befestigt und mit kleinen Elektromagneten ausgestattet. Je Oberfläche ist ein Schreib-/Lesekopf notwendig. Meistens ist zu einer Zeit nur ein Schreib-/Lesekopf aktiv. Auf den Plattenoberflächen sind Spuren an-

gebracht. Die Platten rotieren mit einer konstanten Drehzahl von 7000 bis 15000 U/min um eine senkrechte Drehachse.

Die Daten sind als Blöcke fester Größe in sog. Sektoren gespeichert. Die Plattenoberfläche ist physikalisch in kleine Kuchenstücke, sog. Kreissektoren aufgeteilt. Die Spurenanteile eines Kreissektors werden als logische Sektoren bezeichnet. Alle Spuren untereinander nennt man Zylinder. Zylinder sind über eine Position der Schreib-/Leseköpfe ansprechbar. Der übliche Plattendurchmesser liegt heute meist bei 3,25 oder 2,5 Zoll. Es werden zwischen zwei und acht Platten übereinander gestapelt. Eine herkömmliche Platte verfügt über 135.000 Spuren/Zoll.

3. *Was bedeutet Formatieren einer Festplatte?*

Beim Formatieren wird die logische Struktur (logische Partitionierung) eines Dateisystems erzeugt.

4. *Wie verwaltet Unix seine Dateien und wie werden Ein-/Ausgabegeräte behandelt?*

Im klassischen Unix erfolgt die Dateiverwaltung in einem beliebig tiefen, hierarchischen Dateisystem. E/A-Geräte werden wie Dateien angesprochen.

5. *Wozu ist der DMA-Mechanismus sinnvoll und wie funktioniert er prinzipiell?*

Mit DMA (Direct Memory Access) werden ganze Datenblöcke zwischen dem Geräte-Controller und dem Hauptspeicher transportiert. Der Zugriff erfolgt über sog. DMA-Controller mit speziellen DMA-Chips (Hardwareunterstützung). DMA-Zugriffe auf den Geräte-Controller können parallel zur CPU-Bearbeitung ausgeführt werden. Die CPU stößt die Übertragung an und kann dann eine andere Aufgabe erledigen. Der DMA-Controller erzeugt bei Ende der Übertragung eine Unterbrechung an die CPU.

6. *Was bedeutet RAID?*

RAID steht für Redundant Array of Inexpensive (heute Independent) Disks. Mehrere kleine Platten werden hier als große virtuelle Platte verwaltet. Der Betrieb eines RAID-Systems erfordert mindestens zwei Festplatten. Die Festplatten bilden einen Verbund, der aus Sicht des Betriebssystems als eine einzige logische Festplatte betrachtet wird. Die Daten werden je nach RAID-Verfahren über die einzelnen physikalischen Festplatten verteilt. Redundante Information wird bei den meisten RAID-Varianten zu den Nutzdaten ergänzt und ebenfalls auf die physikalischen Platten verteilt, um im Fehlerfall die Wiederhers-

tellung einer Platte zu unterstützen. Es gibt mehrere RAID-Varianten (RAID-0, RAID-1,...) und auch Kombinationen (RAID-1+0).

7. *Erläutern Sie die Arbeitsweise von RAID-1!*

In RAID-1-Systemen wird eine Vollspiegelung der Daten durchgeführt. RAID-1 bietet die volle Redundanz der gespeicherten Daten, wobei die Kapazität des Systems höchstens so groß ist, wie der kleinste Spiegel. Fällt ein Spiegel aus, können die Platten des anderen Spiegels weiterhin genutzt werden. Bei RAID-1 wird bei schreibendem Zugriff immer auf beide Festplatten-spiegel zugegriffen, da eine Kopie erstellt werden muss.

8. *Was ist der Unterschied zwischen RAID-4 und RAID-5?*

Bei RAID-4 werden die Datenbits wie bei RAID-0 zu Streifen zusammengefasst. Pro Streifen wird eine Prüfsumme gebildet und auf einer eigenen Platte gespeichert. In RAID-5-Systemen sind die Paritätsabschnitte auf alle Platten verteilt, um eine gleichmäßige Plattenauslastung zu erreichen. Ansonsten ist RAID-5 sehr ähnlich zu RAID-4.

9. *Welchen wesentlichen Vorteil bietet RAID-6 im Vergleich zu RAID-5 und welcher Nachteil wird dadurch erkauf?*

RAID-6-Systeme sind wie RAID-5 aufgebaut, speichern aber mehr redundante Prüfdaten, so dass sogar der Ausfall von zwei Platten ohne Auswirkung bleibt. Wenn n Festplatten für die Nutzdaten benötigt werden, ist ein RAID-6-System aus $n+2$ Platten aufgebaut.

RAID-6 zeichnet sich durch gute Leistung beim Lesen aus, die Schreibleistung ist allerdings schlechter als bei RAID-5.

10. *Welche Vorteile bietet RAID-10?*

RAID-10 vereint als Kombination von RAID-0 und RAID-1 die Vorteile beider Verfahren. Es wird Striping und anschließend eine Spiegelung durchgeführt, d.h. RAID-10-Systeme sind schnell und ausfallsicher.

11. *Welchen Vorteil hat Software-RAID gegenüber Hardware-RAID?*

Bei Software-RAID ist keine eigene Controller-Hardware erforderlich, da die Verwaltung der RAID-Platten im Betriebssystem erfolgt.

12. *Zu welchen Problemen kann Caching von Plattenspeicherinhalten in Treiberpuffern bei Ausfall eines Systems führen?*

Caching kann zu Inkonsistenzen führen, wenn das System ausfällt. Es kann nämlich sein, dass zum Ausfallzeitpunkt noch nicht alle Daten aus dem Cache auf einen persistenten Speicher geschrieben wurden.

13. *Was sind i-Nodes unter Unix und wozu dient die i-Node-Struktur?*

i-Nodes sind unter Unix spezielle Datenstrukturen, in denen die Dateiverwaltung je Datei festhält, welche physikalischen Blöcke dieser zugeordnet sind. Über i-Nodes sind alle Blöcke, die eine Datei hat, im Zugriff.

14. *Welche Dateisysteme unter Windows kennen Sie und wie unterscheiden sie sich?*

Windows kennt vor allem folgende Dateisysteme:

- CDFS (CD-ROM File System) ist ein einfaches Read-Only-Format für CD-ROMs. Windows implementiert den ISO-9660-Standard. Die Länge von Verzeichnis- und Dateinamen ist auf 31 Zeichen begrenzt. Es wird nicht zwischen Groß- und Kleinschreibung bei Datei- und Verzeichnisnamen unterschieden. Die Hierarchie der Verzeichnisse ist auf acht Ebenen limitiert. Dieses Format ist aber schon veraltet.
- UDF (Universal Disk Format) ist ISO-13346-konform. Es ersetzt CDFS und ist in der DVD-Spezifikation enthalten. Es unterstützt maximale Dateinamenlängen von 255 Zeichen, eine maximale Pfadlänge von 1023 Zeichen und Groß- und Kleinschreibung bei Datei- und Verzeichnisnamen.
- FAT-16 (FAT = File Allocation Table) ist das alte MS-DOS-Dateisystem und nutzt 16-Bit-Plattenadressen.
- FAT-32 ist ebenfalls ein altes Dateisystem und nutzt 32-Bit-Plattenadressen.
- NTFS (NT Filesystem) ist ein modernes, hierarchisches Dateisystem. NTFS nutzt 64-Bit-Plattenadressen.

15. *Was bezeichnet man als SAN?*

Ein SAN ist ein Netzwerk, in dem große Datenmengen gespeichert und bewegt werden. Im SAN wird der gesamte Speicher, unabhängig von Standort und Betriebssystem, zentral verwaltet und zu virtuellen Einheiten zusammengefasst. Der Speicher muss dazu nicht an ein und demselben Ort untergebracht sein.

Die Server sehen das SAN als eine Art Datenpool, der in voneinander getrennte logische Einheiten aufgeteilt ist. Mehrere redundante Wege zwischen dem Anwender und den Daten können auch vor möglichen Ausfällen schützen.

Anhang

A1 Zahlennamen

Die großen Zahlen ab Billion sind meist nicht so geläufig. Deshalb soll die folgende Tabelle diese zusammenfassen. Die Vorsilben zu „-illion“ und „-illiarde“ leiten sich aus dem Lateinischen ab.

Tabelle A-1: Zahlennamen für große Zahlen

Zahl	Zahlenname
10 ³	Tausend
10 ⁶	Million
10 ⁹	Milliarde
10 ¹²	Billion
10 ¹⁵	Billiarde
10 ¹⁸	Trillion
10 ²¹	Trilliarde
10 ²⁴	Quadrillion
10 ²⁷	Quadrilliarde
10 ³⁰	Quintillion
10 ³³	Quintilliarde
	...
10 ⁶⁰	Dezillion
10 ⁶³	Dezilliarde
10 ⁶⁶	Undezillion
10 ⁶⁹	Undezilliarde
	...

Es sei noch erwähnt, dass das gezeigte Namenssystem als System der *langen Leiter* bezeichnet wird. Es ist das heute anerkannte Referenzsystem für Zahlennamen. Trotzdem verwendet die USA und auch Brasilien und weitere Länder das System der kurzen Leiter, bei dem die Billion nur das 1000-fache einer Million ist. Der eu-

ropäischen Billion entspricht das US-amerikanische Trillion usw. Eine Billion ist also im US-amerikanischen Sprachraum 1000^3 , eine Trillion 1000^4 , eine Quadrillion 1000^5 usw. Auch das britische Englisch nutzt zunehmend den US-amerikanischen Sprachgebrauch.

A2 Metrische Grundeinheiten

Es sind weltweit sowohl die sog. metrischen als auch die angelsächsischen Einheiten in Gebrauch. Die metrischen Grundeinheiten sind in den folgenden Tabellen nachzulesen. Die Groß- und Kleinschreibung ist auch hier nicht einheitlich geregelt. Beide Schreibweisen sind in Gebrauch.

In der Informatik werden als Sonderfall für die Angaben von Speicherplatzgrößen Zweierpotenzen verwendet. Dies ist aber nicht aus den Maßeinheiten ersichtlich.

Beispiele hierzu sind:

Kbyte, KByte oder KB = 2^{10} = 1.024 Byte (Kilobyte)

Mbyte, MByte oder MB = 2^{20} = $1.024 * 1.024$ Byte = 1.048.576 (Megabyte)

Gbyte, GByte oder GB = 2^{30} = $1.024 * 1.024 * 1.024$ Byte = 1.073.741.824 (Gigabyte)

Tbyte, TByte oder TB = 2^{40} (Terabyte) ...

Bei Angaben zu Datenübertragungsraten werden aber keine Zweierpotenzen verwendet, z.B. ist ein Kbit/s = 1.000 Bit pro Sekunde und ein Mbit/s = 1.000.000 Bit pro Sekunde.

Tabelle A-2-1: Metrische Grundeinheiten 1

Exponentendarstellung	Ausgeschriebene Zahl	Präfix
10^{-3}	0.001	Milli
10^{-6}	0.000001	Micro
10^{-9}	0.000000001	Nano
10^{-12}	0.000000000001	Pico
10^{-15}	0.000000000000001	Femto
10^{-18}	0.000000000000000001	Atto
10^{-21}	0.000000000000000000001	Zepto
10^{-24}	0.000000000000000000000001	Yocto

Tabelle A-2-2: Metrische Grundeinheiten 2

Exponentendarstellung	Ausgeschriebene Zahl	Präfix
10^3	1.000	Kilo
10^6	1.000.000	Mega
10^9	1.000.000.000	Giga
10^{12}	1.000.000.000.000	Tera
10^{15}	1.000.000.000.000.000	Peta
10^{18}	1.000.000.000.000.000.000	Exa
10^{21}	1.000.000.000.000.000.000.000	Zetta
10^{24}	1.000.000.000.000.000.000.000.000	Yotta

A3 Wichtige Datenstrukturen für Betriebssysteme

Betriebssysteme verwalten verschiedene Daten bzw. Objekte.¹ Der Zugriff auf diese muss oft sehr effizient sein. Hierzu werden einige grundlegende Datenorganisationsformen bzw. Datenstrukturen verwendet. Dieser Anhang gibt, ergänzend zu den Hauptkapiteln, eine kurze Einführung in einige Datenstrukturen, die in Betriebssystemen häufig eingesetzt werden. Weitere Informationen zu Datenstrukturen sind in (Güting 2003) zu finden.

Datenstrukturen werden in der Informatik in Form ihrer Datenelemente sowie der darauf möglichen Operationen definiert. Datenstrukturen können über die Notationen von Programmiersprachen, über XML oder auch verbal beschrieben werden. Je nach Programmiersprache beschreibt man eine Datenstruktur als Struktur (COBOL, C) oder als Objektklasse (C++, Java, C#). Wird der Schwerpunkt der Beschreibung auf die Operationen gelegt, so kann man für deren formale Beschreibung eine algebraische Spezifikation bzw. sog. Abstrakte Datentypen (ADTs) verwenden.

Wir betrachten die folgenden in Betriebssystemen recht häufig verwendeten Datenstrukturen:

- Zeiger- und Referenztypen
- Sequenzen bzw. diverse Listen
- Stapelspeicher (auch Kellerspeicher oder Stack)

¹ Je nach Betrachtungsweise, daten- oder objektorientiert.

- Warteschlangen (Queues)
- Hashtabellen

Andere Datenstrukturen wie Bäume und sonstige Graphen würden den Rahmen dieser Einführung sprengen. Bäume werden beispielsweise häufig für die Datenorganisation in Dateisystemen und in Datenbanksystemen verwendet.

Zeiger- und Referenztypen

Ein Zeiger (auch Pointer) ist ein Verweis auf ein Objekt oder auf Daten im Hauptspeicher, also im Prinzip ein Verweis auf eine Speicheradresse, die in einer Programmiersprache über einen symbolischen Namen angesprochen werden kann. Zeiger und Referenzen werden oft nicht richtig verstanden. Offensichtlich fällt die Vorstellung, dass eine Variable als Wert eine Adresse enthält, die man auch noch zur Laufzeit verändern kann, nicht leicht. In der Assemblerprogrammierung spricht man hier einfach von Adressen. In den Sprachen C und C++ (vor allem in C) werden Zeiger häufig genutzt. Man kann Zeiger zur Laufzeit dynamisch mit Werten belegen und auch gewisse Operationen mit ihnen ausführen.

Der Ausdruck $p++$ für eine Zeigervariable mit dem symbolischen Namen p bedeutet z.B. in der Sprache C, dass sich der Wert des Zeigers verändert. Es wird genau die Länge des Typs von p auf den Wert von p addiert. Ist p also ein Zeiger auf eine Integervariable, so wird – je nach Rechnerarchitektur – 2 oder 4 zu p addiert. Dies entspricht der Länge eines Integerwertes in Byte, die dieser im Speicher benötigt. Ist p vom Typ *Kunde*, so wird die Länge in Byte, die eine Datenstruktur vom Typ *Kunde* benötigt, zu p addiert.

Von Java behauptet man oft, dass es eine Sprache ohne Zeiger sei, man spricht eher von Referenzen. Dies führt oft zur Verwirrung. Man kann dies aber auch anders sehen: In Java gibt es schon Zeiger, allerdings kann der Programmierer im Gegensatz zu C diese nicht direkt beeinflussen (Güting 2003).

Zeiger bzw. Referenzen sind dynamische Typen, die man hauptsächlich benötigt, wenn man zur Laufzeit Daten bzw. Objekte erzeugt und zu gegebener Zeit wieder freigibt. Die Daten/Objekte werden durch das Laufzeitsystem der Sprache im sog. Heap² verwaltet. Während man sich in den Sprachen C und C++ selbst um die Bereinigung des Heaps kümmern muss, gibt es in Sprachen wie Java und C# in der Speicherverwaltung innerhalb des Laufzeitsystems einen sog. Garbage Collector, der sich darum kümmert.

² Ein Speicherbereich für dynamisch erzeugte Datenbereiche.

Man definiert einen Zeigertyp auf einen bestimmten Objekttyp in C bzw. C++ üblicherweise wie folgt:

```
int *p1;
```

In diesem Fall wird eine Zeigervariable mit dem Namen `p1` auf einen Integerwert deklariert.

Man kann auch Zeiger auf eigene Datentypen verwenden. Der Bezeichner *typedef* führt einen neuen Datentyp ein. Dies sieht etwas komplizierter aus, wie das folgende Beispiel andeuten soll:

```
typedef struct { int i, double d} MyType;  
MyType *Pointer_auf_MyType
```

Das Zeichen „*“ bedeutet, dass der nachfolgende Bezeichner der Name eines Zeigers ist. Der Typ, auf den der Zeiger verweisen kann, ist im obigen Beispiel *MyType*. Dies ist im Prinzip eine Datenstruktur bestehend aus einem Integer- und einem Doublewert.

Die Deklaration eines Zeigers (hier mit dem Namen `p2`) vom Typ *Pointer_auf_MyType* sieht dann wie folgt aus:

```
Pointer_auf_MyType *p2;
```

In Java ist jede Variable, die eine Instanz einer bestimmten Objektklasse repräsentiert, eine Referenz, die wie folgt deklariert werden kann:

```
// Klasse MyType wird definiert  
class MyType { ... }  
  
...  
MyType p1; // Referenz p1 wird erzeugt  
MyType p2; // Referenz p2 wird erzeugt
```

Die Variablen `p1` und `p2` sind in diesem Fall Referenzen auf Objekte der Klasse *MyType*, die allerdings erst einen richtigen Wert (eine Adresse) erhalten, wenn eine Instanz von *MyType* zugewiesen und damit einer der Konstruktoren von *MyType* durchlaufen wird. Dies kann wie folgt aussehen:

```
// Objekt vom Typ MyType wird instanziiert und Standardkonstruktor  
// wird durchlaufen  
p1 = new MyType;
```

In Abbildung A3-1 sind die beiden Variablen skizziert. `p1` verweist bereits auf eine konkrete Instanz, hat also als Wert eine definierte Adresse, `p2` hat dagegen einen undefinierten Wert, da noch keine konkrete Referenz auf eine Instanz zugewiesen wurde.

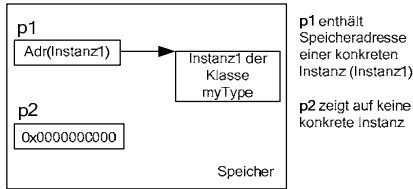


Abbildung A3-1: Java-Referenzen

Wie viel Speicherplatz der Compiler oder der Interpreter (je nach Sprache) bzw. das zugehörige Laufzeitsystem für die Referenzen belegt, hängt von der Größe des Adressraums und dem Aufbau der Adressen ab. Bei einem Adressraum von 4 GB benötigt eine Referenz 32 Bit.

Wie bereits angedeutet, benötigt man Zeiger oder Referenzen insbesondere für die Erzeugung und Manipulation dynamischer Datenstrukturen. Die Elemente von Listen oder Warteschlangen werden häufig mit Zeigern implementiert, falls die Anzahl der zu verwaltenden Elemente von vornherein nicht bekannt ist und diese sich während der Programmlaufzeit stark verändern kann.

Sequenzen, Listen

Datentypen mit einem Wertebereich, der aus einer endlichen Folge (einer Sequenz) von Objekten des gleichen Typs besteht, bezeichnet man als Sequenzen. Allgemein ist eine Sequenz $\langle a_1, \dots, a_n \rangle$ eine Folge aus n Elementen des gleichen Grundtyps. Als Grundtypen kommen wiederum alle möglichen Datentypen in Frage. Es kann sich hier z.B. um Objekte des Typs *Prozess* (Name frei vergeben) handeln, der wie folgt definiert werden kann (hier in Java):

```

class Process {
    int type;
    int status;
    Date creationDate;
    ...
};
    
```

Ohne Bezug auf eine konkrete Implementierung zu nehmen, werden Sequenzen meistens auch als Listen bezeichnet, von denen es verschiedene Ausprägungen und Implementierungsvarianten gibt.

Eine konkrete Ausprägung einer Liste mit der Bezeichnung *MyList* soll hier beispielhaft etwas formaler angelehnt an (Güting 2003) beschrieben werden:

```

algebra MyList
    sorts list, elem, bool
    
```

```

ops
empty :  $\rightarrow$  list
first : list  $\rightarrow$  elem
rest : list  $\rightarrow$  list
append : list x elem  $\rightarrow$  list
concat : list x list  $\rightarrow$  list
isempty : list  $\rightarrow$  bool
sets list = {<a1..., an> | n >= 0, ai  $\in$  elem}

functions
empty = 0
first (a1...an) = a1, falls n > 0; sonst undefiniert
rest (a1...an) = a2...an, falls n > 0; sonst undefiniert
append (a1...an, x) = x a1...an
concat (a1...an, b1...bm) = a1...an * b1...bm
isempty (a1...an) = (n = 0)

end MyList

```

In diesem Beispiel wird eine Liste mit Elementen vom Typ *elem* spezifiziert, die eine Reihe von Operationen (*ops*) mit den Bezeichnungen *first*, *rest*, *empty*, *append*, *concat* und *isempty* bereitstellt. Die Operation *first* liefert z.B. das erste Element der Liste, die Operation *concat* ermöglicht das Verbinden zweier Listen, und die Operation mit dem Namen *append* dient dem Anhängen eines neuen Elements vorne an die Liste. Je nach Listenausprägung können noch weitere Operationen, wie *insert* für das Einfügen eines Elements, an einer bestimmten Stelle hinzukommen.

Die Liste wird als Menge von Elementen dargestellt, wobei eine Liste auch leer sein kann (siehe *sets*). In *functions* wird die Funktionsweise der Operationen näher spezifiziert. Hier wird z.B. genau gezeigt, wo ein neues Element durch die Operation *append* abgelegt wird. Das neue Element mit der Bezeichnung *x* wird in diesem Fall ganz vorne an die Liste angefügt.

Listen sind üblicherweise dynamische Datenstrukturen, in denen eine beliebige, anfangs noch nicht feststehende Anzahl von Objekten gespeichert werden kann. Man kann nun Listen auf vielfältige Weise implementieren. Mögliche Varianten sind einfache Arrays, einfach und doppelt verkettete Listen mit Hilfe von Zeigern bzw. Referenzen und einfach oder doppelt verkettete Listen in Arrays. Bei verketteten Listen verweist jedes Element immer auf das Folgeelement und je nach Organisation evtl. auch auf das vorhergehende Element. Man spricht von einer Verkettung der Elemente.

Implementierung in einfachen Arrays. Arrays (auch als Felder bezeichnet) sind einfache Implementierungen von Sequenzen bzw. Listen zur Speicherung von mehreren Elementen des gleichen Typs. Die Elemente werden hintereinander im Speicher abgelegt. Der Zugriff auf ein Element im Array erfolgt über einen numerischen Index.

Arrays sind meist in Programmiersprachen verfügbar oder als erweiterte Arrays in Bibliotheken (C, C++), Packages (Java) oder Namespaces³ (C#). Ein Beispiel für etwas komfortablere Arrays ist in Java und C# unter dem Namen *ArrayList* zu finden.

Arrays können meist beliebig viele Dimensionen annehmen. Eindimensionale Arrays heißen auch Vektoren, und zweidimensionale Arrays heißen Tabellen oder Matrizen. In Abbildung A3-2 ist ein eindimensionales Array mit Integer-Elementen dargestellt.

Index	
0	1333
1	1444
2	1555
3	132
4	1400
5	2
6	15
7	4

Abbildung A3-2: Eindimensionales Array mit Integerwerten

Die Adressierung eines Elements in einem eindimensionalen Array kann wie folgt aussehen, wobei in der Regel das erste Element mit dem Index 0 adressiert wird:

Adresse eines Elements i = Anfangsadresse des Arrays + (i * Länge des Elements)

Einfache sequenzielle Implementierungen von Listen in Arrays sind recht unflexibel. Das Einfügen und Löschen von Elementen ist aufwändig und erfordert meist ein Verschieben der restlichen Elemente. Effizient sind Arrays, wenn die Anzahl der Elemente von Anfang an bekannt ist und Elemente selten wieder entfernt werden.

³ Unter einem Namespace (Namensraum) versteht man in C# eine logische Gruppierung von Typen, der vergleichbar mit Java-Packages ist. Namensräume können hierarchisch organisiert sein.

Für das Suchen und Löschen muss eine Vergleichsoperation vorhanden sein. In den Daten muss es also eine Schlüsselinformation geben, um die entsprechende Stelle in der Liste finden zu können. Wenn eine Sortierung der Elemente nach einem Index möglich ist, kann ein Element auch schnell aufgefunden werden. Die Suche nach anderen Kriterien muss aber sequentiell erfolgen, da das Suchkriterium jeweils mit dem entsprechenden Wert jedes Elements verglichen werden muss.

Implementierung in verketteten Listen. Sequenzen sollten in dynamischen Datenstrukturen verwaltet werden, wenn die Anzahl der Elemente stark von der Laufzeit abhängig und auch nicht vorhersehbar ist. Je nach Problemstellung können diese einfach verkettet oder auch doppelt verkettet sein. Bei einfach verketteten Listen hat jedes Element neben den Nutzdaten (data) einen Zeiger (next), der auf das Folgeelement verweist (siehe Abbildung A3-3). Darüber hinaus ist ein Anfangszeiger (*Anker*) und meist auch ein Zeiger auf das Ende der Liste (im Bild mit *Last* bezeichnet) notwendig. Auf letzteren könnte allerdings auch verzichtet werden, da man beim Traversieren der Liste auch bis zum letzten Element kommt.

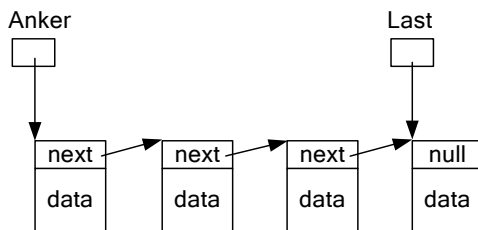


Abbildung A3-3: Lineare, einfach verkettete Liste

Die oben beschriebene Operation *append* zum Anhängen eines Listenelements an den Anfang der Liste könnte dann wie in Abbildung A3-4 dargestellt, realisiert werden. Es sind zwei Aktionen notwendig. Zunächst muss der next-Zeiger des neuen Elements auf das ursprünglich erste Element gestellt werden. Im zweiten Schritt muss der Anker so verändert werden, dass er auf das neue Element verweist. Die anderen Listenelemente sind von der Operation nicht betroffen. Wie wir in späteren Kapiteln noch sehen werden, müssen diese beiden Aktionen ganz oder sie dürfen gar nicht ausgeführt werden. Würde die Operation nach der ersten Aktion beendet, hätte man einen inkonsistenten Zustand in der Liste. Darüber wird noch im Zusammenhang mit parallelen Zugriffen auf von mehreren Prozessen gemeinsam genutzte Ressourcen die Rede sein (Kapitel 6).

Etwas komplizierter wäre eine Operation *insert*, welche die Aufgabe hätte, ein Listenelement nach einem bestimmten anderen Listenelement, also nicht zwangsläufig am Anfang der Liste einzufügen. Für diese Aufgabe ist ein Vergleichskriterium

in den Nutzdaten notwendig, und die Liste muss von vorne bis zu dem Element traversiert werden, welches das Kriterium erfüllt. Das Einfügen als erstes Element stellt dann einen Spezialfall dar.

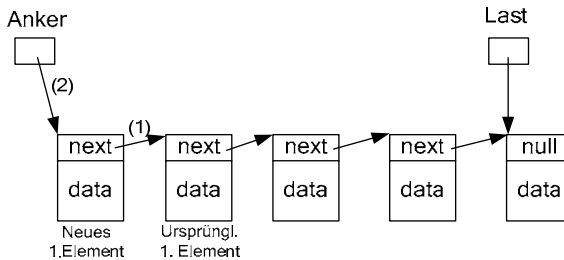


Abbildung A3-4: Anhängen eines Listenelements an eine einfach verkettete Liste

Wie in Abbildung A3-5 dargestellt, wird bei der Operation *insert* zunächst über einen Hilfszeiger bis zu dem Element traversiert, welches das Kriterium erfüllt. Anschließend wird der next-Zeiger des einzufügenden Elements auf das Folgeelement des gefundenen Elements gestellt. Schließlich wird in einer dritten Aktion der next-Zeiger des gefundenen Elements auf das neue Element gestellt. Wenn das Element an das Ende angehängt wird, ist zusätzlich noch eine Veränderung des Last-Zeigers erforderlich. Das Gleiche gilt für den Anker beim Einfügen des Elements an den Anfang der Liste. Der Leser möge sich nun selbst überlegen, wie das Löschen eines Elements an einer bestimmten Stelle (delete-Operation) aussehen könnte.

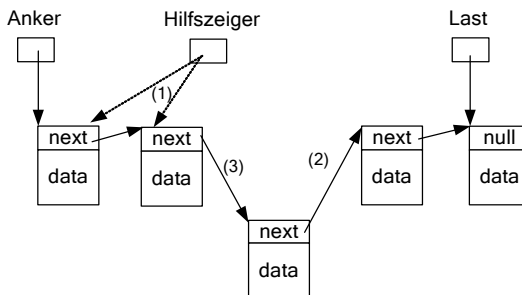


Abbildung A3-5: Einfügen eines Listenelements in eine einfach verkettete Liste

Einfach verkettete Listen haben den Nachteil, dass sie nur in eine Richtung traversiert werden können. Doppelt verkettete Listen vermeiden diesen Nachteil, indem Sie zu jedem Listenelement neben einem next-Zeiger auch noch einen Vorgänger-Zeiger (pred) verwalten. Dies kostet natürlich beim Einfügen und Entfernen von

Elementen mehr Aufwand. Ein Beispiel über Zeiger einer doppelt verketteten Liste ist in Abbildung A3-6 skizziert.

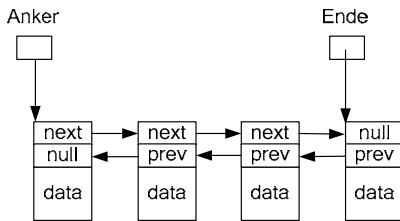


Abbildung A3-6: Lineare, doppelt verkettete Liste

Die Operationen zum Anhängen, Einfügen und Löschen sollen hier nicht weiter vertieft werden.

Implementierung von verketteten Listen in Arrays. Verkettete Listen können auch in statischen Arrays verwaltet werden. Dies ist sinnvoll, wenn die Anzahl der Elemente von vorne herein bekannt ist. Bei einer Veränderung der maximalen Anzahl allerdings muss eine aufwändige Reorganisation durchgeführt werden.

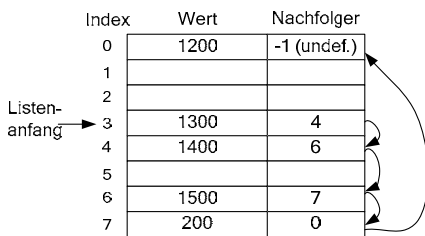


Abbildung A3-7: Lineare, einfach verkettete Liste im Array

Die Indizes auf die Elemente geben hier die Position innerhalb der Liste an. Abbildung A3-7 zeigt ein Beispiel einer einfach verketteten Liste mit acht möglichen Elementen, von denen fünf aktuelle belegt sind. Die Liste beginnt in dem Element mit dem Index 3 und ist über ein Nachfolgerfeld, in dem der Index des nächsten Elements steht, verlinkt.

Auch mehrere Listen können in einem Array verwaltet werden, was in Betriebssystemen insbesondere bei der Verwaltung von Speicherbereichen häufig angewendet wird (mehr dazu in Kapitel 7).

Die gezeigten Operationen stellen nur einige sinnvolle Beispiele dar. Je nach Nutzung der Liste sind weitere, spezielle Operationen denkbar. Listen werden natürlich nicht nur zur Verwaltung von Objekten in Betriebssystemen benötigt. In nahezu jedem komplexeren Anwendungssystem wird diese Art der Datenorganisation

in der einen oder anderen Form verwendet. Allerdings programmiert man die abstrakten Datentypen heute nicht mehr selbst, sondern nutzt weitgehend Basisdatentypen oder Basisklassen aus dem Fundus vorhandener Bibliotheken bzw. Paketen. Man spricht in der objektorientierten Programmierung hier auch von sog. Collection-Klassen (siehe z.B. in Java), deren Implementierung nicht mehr nach außen sichtbar ist. Diese Klassen sind heute so generisch konzipiert, dass sie alle möglichen Basisdatentypen aufnehmen können.

Stapelspeicher, Stack

Ein Stapelspeicher (Kellerspeicher, engl. Stack) dient dazu, eine beliebige Anzahl von meist gleich großen „Objekten“ zu speichern. Die Objekte werden wie in einem Stapel immer oben drauf gelegt und können auch nur von oben, also in umgekehrter Reihenfolge, wieder entnommen (gelesen) werden. Dies entspricht von der Organisationsform dem *LIFO*-Prinzip (Last-In-First-Out, deutsch: zuletzt hinein - zuerst heraus). Für die Definition und damit die Spezifikation des Stapelspeichers ist es unerheblich, welche Objekte darin verwaltet werden. Ein Stack ist im Prinzip ein Spezialfall einer Liste. Für die Bearbeitung der Datenstruktur werden die folgenden, speziellen Operationen benötigt, deren Namen historisch festgelegt sind:

- *push*, um ein Objekt im Stapelspeicher zu speichern (ganz oben).
- *pop*, um das zuletzt gespeicherte Objekt wieder zu lesen und damit auch vom Stapel zu entfernen.
- Optional gibt es noch die Operation *top* bzw. *peek*, um das oberste Objekt ausschließlich zu lesen und nicht zu löschen.

Die *top*-Operation ist nicht zwingend vorgeschrieben, wird aber oft implementiert, um *pop/push* zu ersetzen, da es oft interessant ist, das oberste Element zu "testen". Die Schnittstellensignatur für einen Stack kann formal etwas vereinfacht wie folgt angegeben werden (Güting 2003):

```
algebra MyStack
  sorts stack, elem, bool
  ops
    // Anlegen eines Stack
    empty : → stack
    // Einfügen eines Elements
    push : stack x elem → stack
    // Abholen eines Elements mit Löschen
    pop : stack → stack
    // Auslesen eines Elements ohne Löschen
    top : stack → elem
    // Abfrage, ob Stack leer ist
```

```

isempty : stack → bool
...
end MyStack

```

Die Implementierung eines Stack kann vielfältig gestaltet sein. Man kann einen Stack in einer verketteten, dynamischen Liste, in einer verketteten Liste innerhalb eines Arrays oder auch in einem einfachen Array bzw. Vektor realisieren. Ein Zeiger verweist üblicherweise immer auf das oberste Element (siehe Abbildung A3-8).

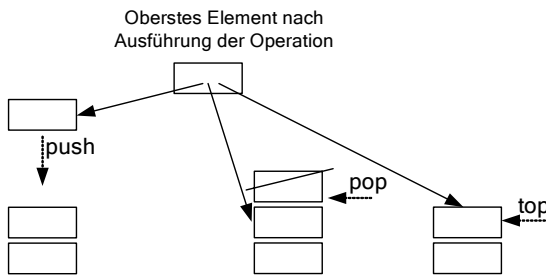


Abbildung A3-8: Stapelspeicher-Operationen

Stacks werden heute vielfältig verwendet. Beispielsweise nutzt man sie in vielen Programmiersprachen⁴ zur Parameterübergabe sowie zur Rückgabe der Ergebnisse bei Prozeduraufrufen. Lokale Variablen von aufgerufenen Prozeduren (Unterprogrammen) werden ebenfalls auf dem Stapel gespeichert.

Weiterhin werden sie in Betriebssystemen z.B. zur Speicherung des aktuellen Zustands bei einem Wechsel von einem Prozess zu einem anderen (Kontextwechsel, siehe Kapitel 5) verwendet. Bei Mikroprozessoren werden sie meist von der Hardware direkt verwaltet, was enorme Leistungsvorteile bringt. Es gibt meistens ein spezielles Hardwareregister, das als Stackpointer oder Stapelzeiger bezeichnet wird. Wie wir noch sehen werden, gibt es in sog. Multitasking-Betriebssystemen für jedes nebenläufige Programm einen eigenen Stapelspeicher. Beim Umschalten zwischen den Programmen, die in Prozessen oder Threads ablaufen, wird der dazugehörige Stapelspeicher durch das direkte Belegen des Stapelzeigers in einem

⁴ Die Verwendung eines Stapelspeichers zur Übersetzung von Programmiersprachen wurde 1957 Friedrich Ludwig Bauer und Klaus Samelson unter der Bezeichnung „Kellerprinzip“ patentiert. Siehe hierzu: Deutsches Patentamt, Auslegeschrift 1094019, B441221X/42m. Verfahren zur automatischen Verarbeitung von kodierten Daten und Rechenmaschine zur Ausübung des Verfahrens. Anmeldetag: 30. März 1957. Bekanntmachung der Anmeldung und Ausgabe der Auslegeschrift: 1. Dezember 1960.

speziellen Register initialisiert. Auch im Java-Interpreter werden Stacks verwendet. Die Stackpointer werden hier meist in der Software realisiert.

In der theoretischen Informatik nutzt man den Kellerspeicher in sog. Kellerautomaten als formales Beschreibungsverfahren, um bestimmte Sprachen näher zu betrachten. Kellerautomaten sind nichtdeterministische endliche Automaten, die um einen Kellerspeicher erweitert werden. Sie werden von Compilern zur Syntaxanalyse verwendet (Eirund 2000).

Warteschlangen, Queues

Warteschlangen sind ebenfalls spezielle Listen, bei denen Elemente nur vorne eingefügt und hinten, also am anderen Ende entnommen werden können. Man bezeichnet die Organisationsform daher als FIFO (First-In-First-Out). Die Elemente, welche als erstes eingefügt wurden, werden als erstes auch wieder entnommen. Eine Warteschlange kann eine beliebige Menge von Objekten aufnehmen und gibt diese in der Reihenfolge ihres Einfügens wieder zurück. Für Queues sind folgende Hauptoperationen sinnvoll:

- *enqueue* zum Hinzufügen eines Objekts
- *dequeue* zum Auslesen und Entfernen eines Objektes

Die Schnittstellensignatur für eine Warteschlange kann formal etwas vereinfacht wie folgt angegeben werden (Güting 2003):

```
algebra MyQueue
  sorts queue, elem, bool
  ops
    // Anlegen einer Queue
    empty : → queue
    // Erstes Element
    front: queue → elem
    // Einfügen eines Elements
    enqueue: queue x elem → queue
    // Abholen eines Elements
    dequeue: queue → queue
    // Abfrage, ob Queue leer ist
    isempty : queue → bool
    ...
end MyQueue
```

Mit der Operation *dequeue* wird immer das Objekt aus der Warteschlange gelesen, welches als erstes mit *enqueue* hineingelegt wurde. Warteschlangen sind im praktischen Leben sehr häufig anzutreffen. Man findet sie z.B. vor jeder Kasse im Kino oder in einem Geschäft. Die Person, die vorne am nächsten zur Kasse steht, wird als erstes bedient und dann aus der Warteschlange „entfernt“.

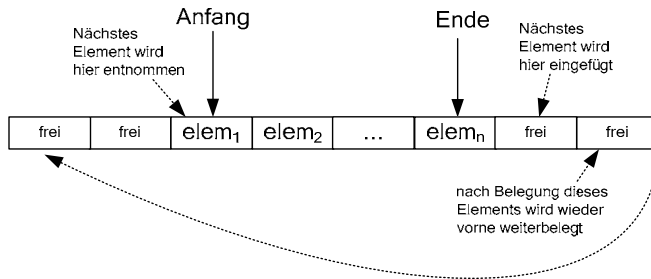


Abbildung A3-9: Queue-Implementierung in einem Array

Man kann nun Warteschlangen über viele Varianten implementieren. Eine einfache Implementierung kann in einem statischen Array erfolgen. Allerdings ist es sinnvoll, das Array zyklisch zu verwalten. Das letzte Element verweist hier wieder an den Anfang. Bei dieser Organisationsform kann bei richtiger Dimensionierung der Array-Größe erreicht werden, dass der Speicherplatz nicht ausgeht. Nach dem letzten Platz im Array wird bei einer *enqueue*-Operation wieder der erste Array-Platz mit Index 0 belegt, sofern dort Platz ist. Man nennt diese Art der Implementierung auch Ringpuffer. Abbildung A3-9 zeigt eine mögliche Implementierung einer zyklischen Queue innerhalb eines Arrays als Ringpuffer. Wenn der Puffer voll ist, wird im Extremfall das älteste Element überschrieben. Eine saubere Implementierung muss aber einen Pufferüberlauf vermeiden, indem im schlimmsten Fall eine Reorganisation durch zusätzliche Speicheranforderung erfolgt, um Datenverluste zu vermeiden. Der Anfangszeiger verweist auf das Element, welches als nächstes entnommen wird, der Ende-Zeiger auf das zuletzt eingefügte Element. Nach Belegung des letzten, physikalisch verfügbaren Platzes im Array wird wieder der erste Platz im Array belegt, falls dieser frei ist. Ist er nicht frei, gibt es einen Pufferüberlauf.

Eine andere Implementierungsvariante zeigt Abbildung A3-10. Hier wird die Queue als einfach verkettete, dynamische Liste organisiert. Der Anfangszeiger zeigt auf das Element, das als nächstes über eine *dequeue*-Operation entnommen wird. Der Ende-Zeiger zeigt auf das letzte Element der Queue. Eine *enqueue*-Operation würde den Ende-Zeiger verändern. Das aktuell letzte Element würde dann das vorletzte Element werden.

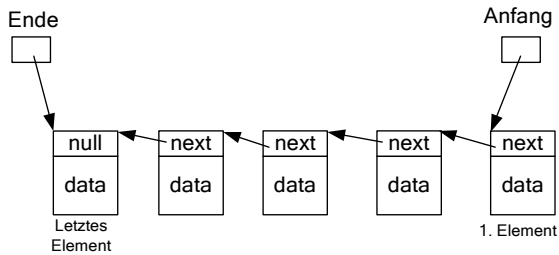


Abbildung A3-10: Queue-Implementierung als einfach verkettete Liste

Eine besondere Form der Queue ist die sog. Vorrang- bzw. Prioritätswarteschlange, bei der vom FIFO-Prinzip abgewichen wird. Bei der *enqueue*-Operation wird hier zusätzlich nach einer Priorität sortiert. Die *dequeue*-Operation liefert das Element mit der höchsten Priorität. Queues werden in Betriebssystemen vielfältig, z.B. für die Verwaltung von Aufträgen (Jobs), Druckaufträgen, eingehenden Nachrichten usw. verwendet. Prioritätswarteschlangen findet man auch häufig bei Algorithmen, die sich mit der Zuordnung von Prozessen zu Prozessoren befassen (siehe Kapitel 5).

Hashtabellen

Eine Hashtabelle ist eine spezielle Indexstruktur, die sich dazu eignet, Elemente aus großen Datenmengen schnell aufzufinden. Beim Einsatz einer Hashtabelle für die Suche in Datenmengen nutzt man ein Hashverfahren zur Auffindung der gesuchten Elemente. Das Hashverfahren wird über eine Hashfunktion realisiert, die aus einem Element-Schlüssel einen sog. Hash-Wert ermittelt, der den Index des gesuchten Elements innerhalb der Tabelle darstellt. Die Hashfunktion findet im Idealfall genau einen Hash-Wert, der den Index auf ein Element repräsentiert, die Hashfunktion kann aber auch zu einem Hash-Wert mehr als ein Element finden. Diesen Fall nennt man Kollision. Hashfunktionen sind also in der Regel nicht eindeutig.

Als Zugriffsoperationen auf Hashtabellen benötigt man mindestens:

- Suchen nach einem Element über einen Element-Schlüssel
- Einfügen eines Elements inkl. des Element-Schlüssels
- Löschen eines Elements über einen Element-Schlüssel

Hashverfahren benötigt man vor allem dann, wenn die Menge der möglichen Schlüssel sehr groß, die Menge der tatsächlich vorhandenen Schlüssel aber relativ klein und von vornherein nicht bekannt ist.

Ein praktisches und anschauliches Beispiel für ein Hashverfahren ist die Suche nach Telefonnummern in einem Telefonbuch. Im Telefonbuch repräsentieren die Namen (Personen, usw.) die Schlüssel. Man sucht über einen Namen eine Telefonnummer. Auch hier kann es Kollisionen geben, wenn z.B. der Name „Mandl“ gesucht wird und dieser mehrfach vorkommt.

Die Implementierung einer Hashtabelle kann in einem einfachen Array erfolgen. Wie bereits angedeutet, ist eine Hashfunktion im Allgemeinen nicht unbedingt linkseindeutig (injektiv), was bedeutet, dass zwei verschiedene Schlüssel zum selben Hash-Wert führen können, und damit wird im Array der gleiche Index angesprochen. Daher speichert man im Array meist nicht die Elemente, sondern wiederum Verweise (Zeiger) auf Container, die dann jeweils für einen Index alle passenden Elemente enthalten. Dieser Container muss dann sequentiell nach dem tatsächlich gewünschten Element durchsucht werden. Es gibt aber auch andere Implementierungsmöglichkeiten, die an dieser Stelle nicht diskutiert werden sollen.

Eine Hashtabelle T ist also ein Array von Zeigern auf die eigentlichen Elemente. Eine Hashtabelle T der Größe m hat die Indizes $0, 1, \dots, m - 1$. Die zugehörige Hashfunktion h ist eine Funktion von der Schlüsselmenge in eine Teilmenge der natürlichen Zahlen und liefert für einen gegebenen Schlüssel s eine Adresse $h(s)$ in der Hashtabelle. Damit ergibt sich:

$h: U \rightarrow \{0, 1, \dots, m-1\}$, wobei U die Menge aller möglichen Schlüssel ist.

Durch die Hashfunktion wird eine gegebene Schlüsselmenge S als Teilmenge von U in die Hashtabelle eingetragen. Die Schlüsselmenge U sollte durch die Hashfunktion möglichst gleichmäßig über den Adressenbereich verteilt werden, was einer ausgewogenen Verteilung entspricht. Werden zwei Schlüssel s und s' auf den gleichen Index und damit die gleiche Adresse abgebildet, ist also $h(s) = h(s')$, dann ergibt sich eine Kollision. Sind die Elemente in der Hashtabelle ausgewogen verteilt, befinden sich in den Containern nur wenige Elemente. Damit ist der letzte Suchschritt relativ schnell möglich. Natürlich kann es auch zu sehr ungleichmäßigen Verteilungen im Array kommen, wenn der Schlüssel nicht richtig gewählt ist. Im schlimmsten Fall kann durch die Hashfunktion h die ganze Menge S auf eine Adresse abgebildet werden. Ein Beispiel für eine typische Hashfunktion ist $h(s) = s \bmod m$. Um bei dieser Modulo-Funktion eine gleichmäßige Verteilung der Schlüssel zu erhalten, sollte m eine Primzahl sein. Es lässt sich nachweisen, dass bei einer Primzahl Kollisionen reduziert werden können.

Vorteil des Hashverfahrens ist im Allgemeinen, dass die Suche in der Regel sehr schnell vonstatten geht. Ein weiterer Vorteil ergibt sich durch den geringen Speicherbedarf im Vergleich zu Arrays, bei denen man gewöhnlich einen Index und

damit ein Array-Feld für jedes Element reserviert. Als nachteilig an Hashtabellen kann festgehalten werden, dass aufgrund der fehlenden Injektivität eine Suche vom Hashwert zum Schlüssel, also genau umgekehrt, einen wesentlich höheren Aufwand erfordert. In diesem Fall muss man im Durchschnitt die Hälfte der Elemente durchsuchen, bis man den Schlüssel zu einem Hashwert findet.

Es gibt viele Anwendungsfälle für Hashtabellen. Man spricht hier auch von assoziativen Arrays, wozu Wörterbücher, Dictionaries usw. gehören. Wichtig sind Hashtabellen auch für Datenbanksysteme. Hier werden sie als Index für Tabellen verwendet. Ein sogenannter Hashindex kann unter günstigen Bedingungen zu idealen Zugriffszeiten führen. Weiterhin findet man Hashverfahren bei der Implementierung von Cache-Speichern und auch bei Compilern. In Betriebssystemen benutzt man diese Datenstruktur u.a. zum schnellen Auffinden von Adressen bei der Speicherverwaltung.

A4 Java-Implementierung des Dining-Philosophers-Problems

Die im Kapitel 6 eingeführte Semaphorklasse soll nun in einer Java-Lösung des Dining-Philosophers-Problems zur Vertiefung diskutiert werden. Hierzu wird eine leicht abgewandelte Version aus (Kredel 1999) verwendet.

Zur Erinnerung: Dieses anschauliche Beispiel zeigt eine Lösung für das grundlegende Problem auf, dass mehrere Threads jeweils mehrere Betriebsmittel (hier zwei Gabeln) zur Ausführung ihrer Arbeit benötigen. Die Reservierung der Betriebsmittel muss also in einer atomaren Aktion erfolgen, sonst kann es zu einem Deadlock kommen. Der Programmcode beginnt mit der Klasse *Phil*. In dieser Klasse werden in einer *main*-Methode die fünf Philosophen als eigene Threads erzeugt. Sie essen ein Menü mit fünf Gängen und haben fünf Gabeln zur Verfügung.

Die Klasse *Forks* implementiert die eigentliche Synchronisation. Es werden binäre Semaphore (Mutexe) je Gabel genutzt. Eine Gabel darf zu einer Zeit nur einmal belegt sein. Die Methode *take* realisiert das Nehmen einer Gabel, *place* implementiert das Zurücklegen. Die Semaphore *table* wird genutzt, um zu erzwingen, dass maximal vier Philosophen an dem Tisch sitzen können. Sie wird hier im Beispiel als Semaphor mit einer Obergrenze von 4 erzeugt.

Diese Lösung ist Deadlock-frei, denn auch wenn vier Philosophen jeweils eine Gabel halten, ist noch eine Gabel frei. Damit kann mindestens ein Philosoph in jedem Fall essen. Es muss auch kein Philosoph verhungern, denn wenn ein Philosoph *i* einmal seine linke Gabel hat, gibt der Philosoph *i-1* zu seiner Linken seine rechte Gabel irgendwann einmal frei und der Philosoph *i* kann essen. In der Anla-

ge sind die Ausgaben von zwei Durchläufen des Programms aufgezeichnet. Interessant ist auch hier, dass die Reihenfolge, in der die Philosophen ihre Gänge einnehmen, und das Ende des Menüs vom aktuellen Scheduling-Szenario abhängen und bei den einzelnen Durchläufen variieren.

```
import java.io.*;
public class Phil {
    public static void main(String[] args) {
        new Phil().work(new PrintWriter(System.out,true));
    }
    void work(PrintWriter out) {
        int philosophers = 5;
        int dishes = 5; // 5 Gänge!
        Forks forks = new Forks(philosophers);
        Thread pt[] = new Thread[philosophers];
        for (int i=0; i<philosophers; i++) {
            pt[i] = new Thread(new Philosopher(forks, i, dishes, out));
            pt[i].start();
        }
        // Warten auf das Ende der Threads
        for (int i=0; i<philosophers; i++)
            try { pt[i].join(); } catch (InterruptedException e) {}
        out.println("terminated");
    }
}

class Forks
{
    int num; // Anzahl der Gabeln
    MySemaphor[] forkSemaphore; // Semaphore für die Nutzung der Gabeln
    MySemaphor tableSemaphore; // Semaphore für den Tisch

    /**
     * Konstruktor
     */
    public Forks(int n)
    {
        num = n;
        tableSemaphore = new MySemaphor(num-1);
        forkSemaphore = new MySemaphor[num];

        for (int j=0; j<num; j++)
        {
            forkSemaphore[j] = new MySemaphor(1);
        }
    }

    /**
     * Zwei Gabeln werden genommen und zwar die links und rechts
     * angeordneten.
     */
    public void take(int i)
```

```
{
    tableSemaphore.P(); // Prüfen, ob am Tisch noch Platz ist
    forkSemaphore[i].P(); // Linke Gabel nehmen
    forkSemaphore[(i+1)%num].P(); // Rechte Gabel nehmen
}

/**
 * Beide Gabeln zurücklegen und Tisch verlassen.
 */
public void place(int i)
{
    forkSemaphore[i].V(); // Linke Gabel zurücklegen
    forkSemaphore[(i+1)%num].V(); // Rechte Gabel zurücklegen
    tableSemaphore.V(); // Tisch verlassen
}
}
```

Die Klasse `Philosopher` implementiert einen Philosophen als Thread. Ein Philosoph tut folgendes in seiner `run`-Methode:

- 1) Er denkt nach (*doSomeWork*).
- 2) Dann tritt er an den Tisch heran und nimmt genau zwei Gabeln auf (siehe *forks.take(id)*). Sind seine beiden Gabeln nicht frei, so nimmt er gar keine und wartet, bis er beide bekommt.
- 3) Dann isst er.
- 4) Schließlich legt er die Gabeln wieder zurück und geht nach 1) (denkt also wieder). Das Zurücklegen erfolgt durch den Aufruf der Methode *forks.place(id)*.

```
class Philosopher implements Runnable {
    Forks forks;
    int id;
    int max;
    PrintWriter out;

    // Erzeugen eines Philosophen
    public Philosopher(Forks f, int i, int m, PrintWriter o) {
        forks = f;
        id = i;
        max = m;
        out = o;
    }

    public void run() {
        for (int i=0; i<max; i++) {
            doSomeWork(2); // denken
            forks.take(id); // Gabeln aufnehmen
            out.println("Philosopher "+id+" eating");
            doSomeWork(2); // denken
            forks.place(id); // Gabeln zurücklegen
        }
        out.println("Philosopher "+id+" done");
    }
}
```

```
}  
  
// Denken und Essen wird hier simuliert  
static void doSomeWork(double t) {  
    // Zufällig langes Warten  
    try {  
        Thread.currentThread().sleep((long)(t*1000.0*Math.random()));  
    }  
    catch (InterruptedException e) {  
    }  
}  
}
```

Die folgenden Beispielabläufe sollen zeigen, wie sich das Programm verhält. Es zeigt sich, dass jeder Philosoph letztendlich zum Essen seiner fünf Gänge kommt:

Lauf 1:

```
Philosopher 4 eating  
Philosopher 2 eating  
Philosopher 1 eating  
Philosopher 3 eating  
Philosopher 0 eating  
Philosopher 2 eating  
Philosopher 4 eating  
Philosopher 1 eating  
Philosopher 3 eating  
Philosopher 0 eating  
Philosopher 2 eating  
Philosopher 4 eating  
Philosopher 1 eating  
Philosopher 1 eating  
Philosopher 3 eating  
Philosopher 0 eating  
Philosopher 2 eating  
Philosopher 4 eating  
Philosopher 3 eating  
Philosopher 1 eating  
Philosopher 1 done  
Philosopher 0 eating  
Philosopher 2 eating  
Philosopher 4 eating  
Philosopher 4 done  
Philosopher 0 eating  
Philosopher 3 eating  
Philosopher 2 done  
Philosopher 0 done  
Philosopher 3 done  
terminated
```

Lauf 2:

```
Philosopher 3 eating
Philosopher 2 eating
Philosopher 4 eating
Philosopher 1 eating
Philosopher 3 eating
Philosopher 0 eating
Philosopher 2 eating
Philosopher 4 eating
Philosopher 3 eating
Philosopher 2 eating
Philosopher 1 eating
Philosopher 0 eating
Philosopher 4 eating
Philosopher 3 eating
Philosopher 2 eating
Philosopher 4 eating
Philosopher 1 eating
Philosopher 0 eating
Philosopher 4 eating
Philosopher 4 done
Philosopher 3 eating
Philosopher 3 done
Philosopher 2 eating
Philosopher 1 eating
Philosopher 2 done
Philosopher 0 eating
Philosopher 1 eating
Philosopher 1 done
Philosopher 0 eating
Philosopher 0 done
terminated
```

A5 C#-Implementierung des Dining-Philosophers-Problems

Wir erläutern die Anwendung von *Wait/Pulse* wie oben auch an einem konkreten Beispiel und zwar an dem bekannten Problem der speisenden Philosophen. Die Basis-Semaphorklasse heißt *Sema*. Die Klasse *Philosopher* implementiert wie in der Java-Lösung in Kapitel 6 einen Philosophen als Thread. Das C#-Programm ist semantisch weitgehend identisch zur Java-Lösung. Die Klasse *Forks* implementiert die eigentliche Synchronisation. Die Semaphore *table* wird genutzt, um zu erzwingen, dass maximal vier Philosophen an dem Tisch sitzen können. Es werden C#-Monitore für die Synchronisation und für das Warten und Signalisieren benutzt. Eine Gabel darf zu einer Zeit nur einmal belegt sein. Die Methode *take* realisiert das Nehmen einer Gabel, *place* implementiert das Zurücklegen.

```
using System;
using System.Threading;
namespace CsharpSynchronisation
```

```

{
    public class Sema
    {
        // Anzahl der maximal möglichen Threads im kritischen
        // Abschnitt
        private int init;
        // Anzahl der verfügbaren Plätze im kritischen Abschnitt
        private int s;
        // Anzahl der in P-Operation wartenden Threads
        private int del;
        /// Konstruiere eine Semaphore mit einer Obergrenze
        /// <param name="i">Obergrenze</param>
        public Sema(int i)
        {
            if (i >= 0) { init = i; }
            else { init = 0; }
            s = init;
            del = 0;
        }
        /// Standardkonstruktor
        public Sema() : this(0)
        { }
        /// Destruktor
        protected void Finalize()
        {
            if (init != s)
            {
                int x = init - s;
                Console.WriteLine
                    ("sema: " + x + " pending operations.");
            }
        }
        /// P-Operation
        public void P()
        {
            Monitor.Enter(this);
            while (s <= 0)
            {
                del++;
                try {
                    Monitor.Wait(this);
                }
                catch (Exception) {}
                del--;
            }
            s--;
            Monitor.Exit(this);
        }
        /// V-Operation
        public void V()
        {
            Monitor.Enter(this);
            s++;
        }
    }
}

```



```

        if (del > 0)
        {
            Monitor.Pulse(this);
        }
        Monitor.Exit(this);
    }
}

public class Phil
{
    int philosophers = 5; // 5 Philosophen
    int dishes = 5; // 5 Gänge!
    /// Einstieg
    public static void Main()
    {
        new Phil().work();
        // Nur damit der Bildschirm stehen bleibt
        Console.ReadLine();
    }
    /// Initialisierung der Philosophen-Threads
    void work()
    {
        Forks forks = new Forks(philosophers);
        Philosopher[] pc = new Philosopher[5];
        Thread[] pt = new Thread[philosophers];
        for (int i=0; i<philosophers; i++)
        {
            // Einen Thread je Philosophen starten
            pc[i] = new Philosopher(forks, i, dishes);
            ThreadStart startMethod
                = new ThreadStart(pc[i].run);
            pt[i] = new Thread(startMethod);
            pt[i].Name = ("Phil" + i).ToString();
            pt[i].Start();
        }
        // Warten auf das Ende der Threads
        for (int i=0; i<philosophers; i++)
        {
            try {
                pt[i].Join();
            }
            catch (Exception ) { }
        }
        Console.WriteLine("terminated");
    }
}

public class Philosopher
{
    int id; // Nummer des Philosophen
    int max; // Maximale Anzahl an Gängen
    Forks forks; // Verfügbare Gabeln
    /// Konstruktor
    /// <param name="f">Anzahl Gabeln</param>
    /// <param name="i">Nummer des Philosophen</param>

```

```

/// <param name="m">Anzahl der Gängen</param>
public Philosopher(Forks f, int i, int m)
{
    forks = f;
    id = i;
    max = m;
}
/// Aktivitäten aller Philosophen
public void run()
{
    for (int i=0; i<max; i++)
    {
        doSomeWork(); // denken
        forks.take(id); // Gabeln aufnehmen
        Console.WriteLine("Philosopher " + id + " eating");
        doSomeWork(); // essen
        forks.place(id); // Gabeln zurücklegen
    }
    Console.WriteLine("Philosopher " + id + " done");
}
/// Denken und Essen wird hier simuliert
/// <param name="t"></param>
static void doSomeWork()
{
    // Zufällig lange Zeit in diesem Zustand verharren
    try {
        Random random = new Random();
        int r = random.Next(1,1000);
        Console.WriteLine(
            "Thread " + Thread.CurrentThread.Name +
            " wartet " + r + " ms");
        Thread.Sleep(r);
    }
    catch (Exception) {
        Console.WriteLine("Exception in sleep");
    }
}
}
public class Forks
{
    int num;
    Sema[] fa; // Liste mit Semaphoren für die Nutzung der Gabeln
    Sema table; // Semaphore für die Tischbelegung
    public Forks(int n)
    {
        num = n;
        table = new Sema(num-1);
        fa = new Sema[num];
        for (int j=0; j<num; j++) fa[j] = new Sema(1);
    }
    /// Zwei Gabeln aufnehmen
    /// <param name="i">Index auf Philosophen in der
    /// Liste</param>

```

```
        public void take(int i)
        {
            table.P();
            fa[i].P();
            fa[(i+1)%num].P();
        }
        /// Zwei Gabeln zurücklegen
        /// <param name="i">Index auf Philosophen in der
        /// Liste</param>
        public void place(int i)
        {
            fa[i].V();
            fa[(i+1)%num].V();
            table.V();
        }
    }
}
```

Im Folgenden wird ein typischer Beispiellauf aufgezeigt. Aus dem Ablauf geht hervor, dass jeder Philosoph schließlich zum Essen seiner fünf Gänge kommt und das Menü abschließen kann:

```
Philosopher 0 eating
Philosopher 2 eating
Philosopher 1 eating
Philosopher 4 eating
Philosopher 3 eating
Philosopher 0 eating
Philosopher 2 eating
Philosopher 4 eating
Philosopher 1 eating
Philosopher 3 eating
Philosopher 0 eating
Philosopher 2 eating
Philosopher 4 eating
Philosopher 1 eating
Philosopher 3 eating
Philosopher 0 eating
Philosopher 4 eating
Philosopher 2 eating
Philosopher 1 eating
Philosopher 3 eating
Philosopher 0 eating
Philosopher 2 eating
Philosopher 0 done
Philosopher 2 done
Philosopher 4 eating
Philosopher 1 eating
Philosopher 4 done
Philosopher 1 done
Philosopher 3 eating
Philosopher 3 done
Terminated
```

Literaturhinweise

- [Baumgarten 2006] *Baumgarten, U.; Siegert H.-J.*: Betriebssysteme, 6. Auflage, Oldenbourg-Verlag 2006
- [Bengel 2002] *Bengel, G.*: Grundkurs Verteilte Systeme, 3. Auflage, Vieweg-Teubner-Verlag, 2002
- [Birell 2003] *Birell, Andrew, D.*: An Introduction to Programming with C# Threads, Microsoft Corporation, 2003 (sehr interessant, weiterführend)
- [Bovet 2005] *Bovet, D. P.; Cesati, M.*: Understanding the Linux Kernel, O'Reilly, 2005
- [Brause 2001] *Brause, R.*: Betriebssysteme Grundlagen und Konzepte, Springer Verlag, 2001
- [Brinch Hansen 1973] *Brinch Hansen, P.*: Operating System Principles, Section 7.2 Class Concepts, Juli 1973, S. 226-232
- [Brinch Hansen 1999] *Brinch Hansen, P.*: „Java's Insecure Parallelism“, SIGPLAN Notices 34, 4, 1999
- [Ehse 2005] *Ehse, E.; Köhler, L.; Riemer, P.; Stenzel, H. Victor, F.*: Betriebssysteme: Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/Linux, Pearson Studium, 2005 (zu empfehlen)
- [Eirund 2000] *Eirund, H.; Müller, B.; Schreiber, G.*: Formale Beschreibungstechniken der Informatik, Teubner, 2000
- [Englewood 1995] *Englewood Cliffs N.J.; Brinch Hansen, P.*: The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, 1, Juni 1995, S. 199-207 (sehr interessant, Klassiker)
- [Glatz 2006] *Glatz, E.*: Betriebssysteme Grundlagen, Konzepte, Systemprogrammierung, dpunkt.verlag, 2006 (sehr zu empfehlen)
- [Güting 2003]: *Güting, H. R.; Dieker, S.*: Datenstrukturen und Algorithmen, 2. Auflage, Teubner Verlag, 2003
- [Herrmann 2002] *Herrmann, P.*: Rechnerarchitektur, 3. Auflage, Vieweg Verlag, 2002
- [Hoare 1974]: *Hoare, C.A.R.*: Monitors: An Operating System Structuring Concept, Communications of the ACM, Vol. 17, No. 10, Oktober 1974 (Klassiker)
- [Kredel 1999] *Kredel, H.; Yoshida, A.*: Thread- und Netzwerk-Programmierung mit Java, dpunkt.verlag, 1999

- [Mauerer 2004] *Mauerer, W.*: Linux Kernelarchitektur, Hanser Verlag, 2004
- [Prosise 2002] *Prosise, J.*: Microsoft .NET Entwicklerhandbuch, Microsoft Press, 2002
- [Richter 1985] *Richter, L.*: Betriebssysteme, Teubner Verlag, 1985 (Klassiker)
- [Schwichtenberg 2007] *Schichtenberg, H.*: Microsoft .net 3.0 Crashkurs, Microsoft Press, 2007 (gute Einführung in die .NET-Technologien)
- [Silberschatz 1994] *Silberschatz, G.*: Operating System Concepts, Fourth Edition, Addison-Wesley, 1994 (Klassiker, gut zu lesen, umfangreich)
- [Solomon 2000] *Solomon, D. A.; Russinovich, M.*: Inside Microsoft Windows 2000, Microsoft Press, 2000
- [Solomon 2005] *Solomon, D. A.; Russinovich, M.*: Microsoft Windows Internals, Microsoft Press, 2005 (sehr detaillierte Einführung in Windows-Internas)
- [Solymosi 2001] *Solymosi, A.; Solymosi, P.*: Effizient Programmieren mit C# und .NET, Vieweg Verlag, 2001
- [Stärk 2003] *Stärk, R.; Börger, F.*: An ASM specification of C# threads and .NET memory model, Computer Science Department, ETH Zürich, www.inf.ethz.ch, 2003
- [Stallings 2003] *Stallings, W.*: Betriebssysteme Prinzipien und Umsetzung, Prentice Hall, 2003 (Klassiker, umfangreich)
- [Sun 2005] *Sun Microsystems*: Java2 Platform Standard Edition 1.5
- [Tanenbaum 2001] *Tanenbaum, A. S.*: Modern Operating Systems, Prentice Hall, 2001
- [Tanenbaum 2002] *Tanenbaum, A. S.*: Moderne Betriebssysteme, 2. überarbeitete Auflage, Pearson Studium, 2002
- [Tanenbaum 2003] *Tanenbaum, A. S.*: Computernetze, 4. revidierte Auflage, Prentice Hall, 2003 (Klassiker zu Computernetzen, sehr ausholend)
- [Tanenbaum 2009] *Tanenbaum, A. S.*: Moderne Betriebssysteme, 3. aktualisierte Auflage, Pearson Studium, 2009 (Klassiker, sehr umfangreich)

Sachwortverzeichnis

- 8-Bit-Betriebssystem 17
- Abrufstrategie 210
- active Queue 120
- Adressumsetzpuffer 219
 - TLB hit 219
 - TLB miss 219
- Affinity-Maske 122
- AMD Athlon 11
- AMD Opteron 11
- AMD64-Architektur *Siehe* x64-Architektur
- Anycast 188
- APC 54
- APIC 50
- Application-Server 37
- Arbeitsspeicher *Siehe* Hauptspeicher
- Array 330
- Assoziative Arrays 340
- Atomare Aktion 134
- AtomicBoolean 166
- AtomicInteger 166
- AtomicLong 166
- Auftrag 103
- Ausnahme 45
- Austauschstrategie 211
- Basic Input Output System 207
- Batchprocessing *Siehe* Stapelbetrieb
- Batchverarbeitung *Siehe* Stapelbetrieb
- Befehlsregister *Siehe* Befehlszähler
- Befehlszähler 6
- Belady 224
- Belady's Anomaly 233
- Benutzermodus 25
- Betriebsart 32
- Betriebsmittel 3, 206
- Betriebsmittelbelegungsgraph 140
- Betriebssystem-Virtualisierung 28
- BIOS *Siehe* Basic Input Output System
- Blockieren 140
- Broadcast 188
- Buddy-Technik *Siehe* Halbierungsverfahren
- Bussystem 5
- Busy Waiting 137
- C#-Interlocked-Klasse 181
- C#-Lese- und Schreibsperrern 180
- C#-Monitor 169
 - Enter 169
 - Exit 169
 - Pulse 169
 - TryEnter 177
 - Wait 169
- C#-Mutex 178
 - Close 179
 - ReleaseMutex 179
 - WaitOne 179
- Cache 5
 - Level-1 8
 - Level-2 8
 - Level-n-Cache 5
- Chipsatz 258
- Cleaning Policy *Siehe* Entladestrategie
- clone 92, 115
- CLR *Siehe* Common Language Runtime
- Clusterfaktor 271
- Clusternummer 271
- Common Level Runtime 80
- Computersysteme 2
- Container 37
- CPU *Siehe* Zentraleinheit
- CPU-Register 202, 214, 238
- Dämon-Prozess 93
- Dateisystem 264
- Dateiverwaltung 264
- Dateiverzeichnis 264
 - Unterverzeichnis 264
- Deadlock *Siehe* Verklemmung
- Deskriptor 237
- Dining-Philosophers-Problem 147
- Directory *Siehe* Dateiverzeichnis
- Distanz 215
- DLL 239
- DPC 56

- Dreistufige Seitentabelle *Siehe* Mehrstufige Seitentabelle
- ECC *Siehe* Error Correction Code
- Echtzeitbetriebssystem 14
- Eingebettetes System *Siehe* Embedded System
- Einprozessorsystem 33
- Einstellung der
 - Unterbrechungsanforderung 53
- Einstufige Seitentabelle 215
- Einwegekommunikation 185
- Embedded System 15
- Entladestrategie 236
 - Demand-Cleaning 236
 - Page-Buffering 237
 - Precleaning 237
- Error Correction Code 259
- exec 92
- execve 92
- expired Queue 120
- Fairness 101
 - strong fair 101
 - weak fair 101
- Festplatten *Siehe* Magnetplatten
- Festplattenzugriffsstrategie 261
 - FCFS 261
 - SSTF 261
- Fiber 91
- Fibre-Channel 279
- Fibre-Channel-Switched-Fabric 279
- Filedescriptor 195
- fork 92
- format 269
- Formatieren 271
- Fragmentierung 234
 - externe 234
 - interne 234
- Frame *Siehe* Seitenrahmen
- GDT *Siehe* Global Descriptor Table
- Gegenseitiger Ausschluss *Siehe* Wechselseitiger Ausschluss
- Geräte 258
 - blockorientierte 258
 - zeichenorientierte 258
- Gerätetreiber 254
- Gerätverwaltung 254
- Global Descriptor Table 238
- HAL *Siehe* Hardware Abstraction Layer
- Halbierungsverfahren 235
- Handheld-Computer 15
- hard real time 105
- Hardware Abstraction Layer 30
- Hardware-Kontext 67
- Harvard-Rechner 4
- Hashing 338
 - Hashfunktion 338
 - Hashtabelle 338
 - Hashverfahren 338
 - Hash-Wert 338
- Hauptspeicher 203
- Hilfsregister 6
- Hyperthreading 13
- IA32-Architektur 11, 241, 242, 244
- IA64-Architektur 11
- Idle-Thread 85
- IDT *Siehe* Interrupt-Dispatcher-Tabelle
- init-Process 92
- Instruction Pointer *Siehe* Befehlszähler
- int 62
- Interrupt 214, *Siehe* Unterbrechung
- Interrupt-Dispatcher-Tabelle 54
- Interrupt-Level 47
- Interrupt-Request-Level *Siehe* Einstellung der Unterbrechungsanforderung
- Interrupt-Service-Routine 46
- Interrupt-Vektor 47
- Interrupt-Vektor-Tabelle 47
- Invertierte Seitentabellen 221
- IP-Adresse 192
- IRQL *Siehe* Einstellung der Unterbrechungsanforderung
- iSCSI 280
- ISR Interrupt-Service-Routine
- Itanium-Prozessor 9
- Java-Monitor 157
 - notify 164
 - notifyAll 164
 - wait 163
- Java-Thread-Scheduling 129
- Jiffy *Siehe* Tick

- Job *Siehe* Auftrag
- Kachel *Siehe* Seitenrahmen
- Kernel 213, 224, 235, 236, 239, 241, 243
- Kernel-Level-Thread 74
- Kernelmodus 25
- Kernel-Thread 243
- Kommunikation
 - asynchron 184
 - bidirektional 188
 - halbduplex 188
 - Kommunikationsphasen 187
 - simplex 188
 - synchron 184
 - unidirektional 188
 - Verbindungskontext 187
 - verbindungslos 187
 - verbindungsorientiert 187
 - voll duplex 188
- Kommunikations-Middleware 27
- Kommunikationssystem 185
- Kontextwechsel 67
- Kritischer Abschnitt 137
- kswapd 243
- Last-In-First-Out 334
- LCN *Siehe* Clusternummer
- LDT *Siehe* Local Descriptor Table
- Lebendigkeit 140
- Leichtgewichtiger Prozess *Siehe* Thread
- LIFO *Siehe* Last-In-First-Out
- Liste 328, 329
 - doppelt verkettete Liste 329
 - verkettete Liste 329
- Local Descriptor Table 239
- Lock *Siehe* Sperre
- Lokalität 203
 - Räumliche Lokalität 204
 - Zeitliche Lokalität 203
- Lokalitätseffekt 203
- Loopback-Adresse 192
- Lost Update 136
- LWP *Siehe* Thread
- Magnetplatten 258
- Main Memory *Siehe* Hauptspeicher
- Mainframe 14
- Maschinensprache 2
- Maskenregister 45
- Master File Table 271
- Mehrkernprozessor *Siehe* Multicore-Prozessor
- Mehrprogrammbetrieb *Siehe* Multiprogramming
- Mehrprozessorsystem 33
- Mehrstufige Seitentabelle 216
- Mehrzweckbetriebssystem 15
- Memory Management Unit 212
- Memory-Manager 212
- Memory-Mapped-Ein-/Ausgabe 262
- Message-Passing 186
- MFT *Siehe* Master File Table
- Mikrokern-Architektur 25, *Siehe* Mikrokern-Architektur
- Mikroprogramm 2
- mkdir 266
- mknod 197
- MMU *Siehe* Memory Management Unit
- Monitor 150
 - Bedingungsvariable 150
 - wait und signal 150
- Monolithischer Betriebssystemkern 23
- Monoprogramming 207
- mount 266
- MS-DOS 207
- Multicast 188
- Multicore-Prozessor 12
 - Dualcore-Prozessor 12
 - Quadcore-Prozessor 12
 - Singlecore 12
- Multi-Level-Feedback-Scheduling 107
- Multi-Level-Scheduling 106
- Multiprogramming 33, 201, 207, 212
- Multitasking 34
- Mutual Exclusion *Siehe* Wechselseitiger Ausschluss
- Nachrichtenbasierte Kommunikation 184
- NAS *Siehe* Network Attached Storage
- Nebenläufigkeit 134
- Network Attached Storage 278
- Network File System 268
- NFS *Siehe* Network File System
- nice-Systemdienst 116

- Nice-Wert 116
- NMI *Siehe* Non Maskable Interrupt
- Non Maskable Interrupt 46
- non-preemptive *Siehe* nicht verdrängend
- non-preemptive 103
- Northbridge 258
- Offset *Siehe* Distanz
- Page *Siehe* Seite
- Page Fault 214
- Page Frame *Siehe* Seitenrahmen
- Page Table *Siehe* Seitentabelle
- Paging 210
- Paging-Area 211
- Partition 207
- PCI-Bus 256
- pdflush 243
- Pentium-Prozessor* 8
- Physischer Speicher *Siehe* Realer Adressraum
- PIC 50
- PID *Siehe* Process Identification
- Pipe
 - Pipe-Symbol 194
- Pipes 193
 - Named Pipes 197
 - Unnamed Pipes 197
- Pointer *Siehe* Zeiger
- Point-to-Point 187
- Polling 44
- Portnummer 192
- POSIX 93
- preemptive *Siehe* verdrängend
- Prioritätsklassen
 - Relativprioritäten 123
 - Threadprioritäten 123
 - Windows-Prioritätsebenen 123
- Prioritätswarteschlange 338
- Priority Boost 126
- Privilegierter Modus *Siehe* Kernelmodus
- Process Control Block 68
- Process Identification 70
- Producer-Consumer-Problem 147
- Programmiermodell 6
- Programmzähler *Siehe* Befehlszähler
- Protokoll 183
- Prozess 66
- Prozess-Kontext 67
- Prozessmanager 100
- Prozessoraffinität 116, 122
 - explizite Prozessoraffinität
 - implizite Prozessoraffinität
- Prozesstabelle 68
- PSW *Siehe* Statusregister
- Publish-Subscribe 187
- pwd 266
- Quantum Boost 126
- Quantum Stretching 126
- Race Condition 137
- RAID 273
 - Hardware-RAID 273
 - RAID-0 274
 - RAID-1 275
 - RAID-1+0 276
 - RAID-2 275
 - RAID-3 276
 - RAID-4 276
 - RAID-5 277
 - RAID-6 277
 - RAID-Controller 273
 - RAID-Level 274
 - Software-RAID 273
- RAID-10 *Siehe* RAID-1+0
- Read Only Memory 207
- Reader-Writer-Problem 147
- Realer Adressraum 209
- Rechnerkern *Siehe* CPU
- Rechnersysteme *Siehe* Computersysteme
- reentrant *Siehe* threadsafe
- Referenz 326
- Registersatz 6
- Register-Stack-Engine 68
- Replacement Policy *Siehe* Austauschstrategie
- rmdir 266
- ROM *Siehe* Read Only Memory
- Run-Queue 114
- SAN *Siehe* Storage Area Network
- Schattenspeicher 211
- Scheduling-Verfahren 104
 - FCFS 104

- Garantiertes Scheduling 105
- Lottery-Scheduling 105
- Minimal Deadline First 106
- Polled Loop 106
- PS 105
- RR 105
- SJF 104
- SPF 104
- SPN 104
- SRPT 105
- SRTF 105
- SRTN 104
- SCSI-Bus 256
- Segmentierung 237, 238
- Seite 211
- Seitenersetzungsstrategie 223, *Siehe*
 - Austauschstrategie
 - Clock-Page 226
 - FIFO 225
 - LRU 227
 - NFU 229
 - NFU mit Aging 230
 - NRU 224
 - Second-Chance 226
- Seitenfehler 214, 218, 223, 225, 226, 230, 247, *Siehe* Page Fault
- Seitenrahmen 211
- Seitentabelle 212, *Siehe* Seitenumsetztabelle
- Seitenumsetztabelle 215
- Sektor 259
 - Kreis Sektor physikalisch 259
 - logisch 259
 - LSN 259
- Semaphor 144
 - Binäres Semaphor 146
 - Counting Semaphor 146
 - Down-Operation 145
 - Mutex 146
 - P-Operation 144
 - Up-Operation 145
 - V-Operation 144
- Semaphorzähler 144
- Sequenz 328
- Serverbetriebssystem 14
- Shared Library 239
- Shell 29
- Sicherheit 140
- Signal-and-Continue-Konzept 151
- Signal-and-Wait-Konzept 151
- Singletasking 34
- Slab-Cache 244
- Smartcard-Betriebssystem 15
- Sockets 192
 - TCP-Sockets 192
 - UDP-Sockets 192
- soft real time 105
- Software-Interrupt 44
- Southbridge 258
- Speicherbasierte Kommunikation 184
- Speicherbelegungsstrategie 234
- Speicherzuteilungsstrategie 211
- Sperre 141
- Spinlock *Siehe* Sperre
- Spur 259
- Stack 8, *Siehe* Stapelspeicher
- Stapelbetrieb 33
- Stapelspeicher 334
- Starvation *Siehe* Verhungern
- Statusregister 6
- Storage Area Network 278
- Stripes 274
- Striping 274
- Striping-Granularität 274
- Supervisor-Call 61
- SVC *Siehe* Supervisor-Call
- Swapping 208
- Synchronisationsprimitive synchronized 156
- Synchronisationsvariable 150
- Systemcall 44, 60
- Systemsoftware 2
- Task-Manager *Siehe* Windows-Task-Manager
- TCP 192
- Teilhaberbetrieb 34
- Teilnehmerbetrieb 34
- Terminalserver 39
- Thread 71
- Thread Control Block 72

- Thread-Kontext 71
- threadsafe 74
- Tick 117
- Timesharing 34, 208
- TLB *Siehe* Adressumsetzpuffer
- Transaktionsmonitor 35
- Translation Lookaside Buffer *Siehe* Adressumsetzpuffer
- Trap 44, 61, 214
- TSL 142
- UDP 192
- UFS 266
- UltraSparc III 11
- Unicast 188
- Universalbetriebssystem *Siehe* Mehrzweckbetriebssystem
- Unix 201, 239, 240, 241
 - Block Special File 261
 - Character Special File 261
 - Special File 261
- Unix-Dateisystem *Siehe* UFS-Dateisystem
 - Bootblock 267
 - Free Blocks 267
 - i-Node 267
 - MBR 267
 - Superblock 267
- Unterbrechung 44
 - asynchron 44
 - synchron 44
- USB 256
- User-Level-Thread 74
- Usermodus *Siehe* Benutzermodus
- VCN 271
- verdrängend 103
- Verdrängungsstrategie *Siehe* Austauschstrategie
- Verhungern 104, 140
- Verklemmung 140
- vierstufige Seitentabelle *Siehe* Mehrstufige Seitentabelle
- Virtualisierung 28, 279
- Virtuelle Adresse *Siehe* Virtuelle Speichertechnik
- Virtuelle Speichertechnik 209
 - Virtueller Adressraum 209
 - Virtueller Prozessor 66
 - Virtueller Speicher *Siehe* Virtueller Adressraum
 - Volume 271
 - Von-Neumann-Rechner 3
 - Vorrangwarteschlange *Siehe* Prioritätswarteschlange
 - Warteschlange 207
 - WDM *Siehe* Windows Driver Model
 - Wechselseitiger Ausschluss 137
 - wiedereintrittsfähig *Siehe* threadsafe
 - Windows 201, 231, 240, 244, 245, 247, 248
 - Windows Driver Model 255
 - Windows-Dateisystem 268
 - CDFS 268
 - Cluster 269
 - FAT-12 269
 - FAT-16 269
 - FAT-32 269
 - NTFS 269
 - UDF 268
 - Windows-Prioritätsebenen 123
 - Basispriorität 123
 - Windows-Seitenersetzungsstrategie 247
 - Free-Page-List 248
 - Modified-Page-List 248
 - PFN-Datenbank 249
 - Standby-Page-List 248
 - Windows-Task-Manager 87
 - Working-Set-Modell 230
 - x64-Architektur 12, 216, 244
 - x86-Architektur 244
 - Zeiger 326
 - Zeitkritischer Ablauf *Siehe* Race Condition
 - Zeitscheibenverfahren *Siehe* Timesharing
 - Zentraleinheit 4
 - zombie 94
 - Zone 260
 - Zweistufige Seitentabelle *Siehe* Mehrstufige Seitentabelle
 - Zweiwegekommunikation 185
 - Zylinder 259