hochschule
hof

Fakultät Informatik

Vorlesung Informatik im 1. Semester

„Betriebssysteme I“

Prof. Dr. Horst Heineck
Alfons-Goppel-Platz 1
95028 Hof/Saale

Raum: **B 134, Anmeldung über B 132**
Tel.: **09281/409-4440**
Fax: **09281/409-55-4440**
Email: **Horst.Heineck@hof-university.de**

Sprechstunde **Wintersemester 2014/15**
Mittwoch: **10^{00} Uhr – 11^{00} Uhr**

Literaturverzeichnis

/Gates-95/ Gates, B.:
Der Weg nach vorn, William H. Gates III. und Hoffmann und Campe Verlag Hamburg, 1995

/Gulbins-95/ Gulbins, J., Obermayr, K.:
Unix System V.4. Begriffe, Konzepte, Kommandos, Schnittstellen, Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 1995

/Horn-95/ Horn, Ch., Kerner, I. O.:
Lehr- und Übungsbuch INFORMATIK, Band 1: Grundlagen und Überblick, Fachbuchverlag Leipzig GmbH, 1995

/Werner-95/ Werner, D.:
Taschenbuch der INFORMATIK, Fachbuchverlag Leipzig GmbH, 1995

/Born-98/ Born, G.:
Inside the Microsoft Windows 98 Registry, Microsoft Press, Redmond, 1998

/Hein-98/ Hein, J.:
Linux Systemadministration, Einrichten, Wartung, Software-Updates, Addision-Wesley, 1998

/Herold-99/ Herold. H.:
Linux Unix Shells, Addision-Wesley, 1999

/Hipson-00/ Hipson, P. D.:
Mastering Windows 2000 Registry, Sybex, Alameda, 2000

/Witzak-00/ Witzak, M.P.:
Echtzeitbetriebssysteme, Eine Einführung in Architektur und Programmierung, Franzis Verlag GmbH, 2000

/Hansen-01/ Hansen, H.R.:
Wirtschaftsinformatik I, 8.Auflage, Lucius&Lucius-Verlag Stuttgart, 2001

/Sterling-01/ Sterling, T.:
Beowulf Cluster Computing with Linux, MIT Press, Cambridge, Massachusetts, 2001

/Torvalds-01/ Torvalds, L., Diamond, D.:
Just for Fun, Wie ein Freak die Computerwelt revolutionierte, Hanser Verlag München Wien, 2001

/Mayer-01/ Mayer, A.:
Shellprogrammierung in Unix, Computer & Literatur Verlag GmbH, 2001

/Vogt-01/ Vogt. C.:
Betriebssysteme, Spektrum Akademischer Verlag Heidelberg, Berlin 2001

/Hansen-02/ Hansen, H.R.:
Arbeitsbuch Wirtschaftsinformatik I, 6.Auflage, Lucius&Lucius-Verlag Stuttgart, 2002

/Sterling-02/ Sterling, T.:
Beowulf Cluster Computing with Windows, MIT Press, Cambridge, Massachusetts, 2002

/Tanenbaum-02/ Tanenbaum, A.S.:
Moderne Betriebssysteme, 2. überarbeitete Auflage, Pearson Studium, 2002

/Weltner-02/ Weltner, T.:
Microsoft Windows XP Professional, Das Handbuch, Microsoft Press, 2002

/Berman-03/ Berman, F., Fox, G., Hey, T.:
Grid Computing, Making the global Infrastructure a Reality, John Wiley & Sons Ltd., 2003

/Gulbins-03/ Gulbins, J., Obermayr, K., Snoopy:
Linux, Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 2003

/Harris-03/ Harris, J.A.:
Betriebssysteme, 330 praxisnahe Übungen mit Lösungen, mitp-Verlag Bonn, 2003

/Morrison-03/ Morrison, R.S.:
Cluster Computing, Architectures, Operating Systems, Parallel Processing & Programming Language, Richard S. Morrison, 2003

/Stallings-03/ Stallings, W.:
Betriebssysteme, Prinzipien und Umsetzung, 4. Überarbeitete Auflage, Pearson-Studium, 2003

/Brause-04/ Brause, R.:
Betriebssysteme. Grundlagen und Konzepte, 3. überarbeitete Auflage, Springer Verlag Berlin, Heidelberg, New York, London, Paris, Tokio, 2004

/Burtch-04/ Burtch, K.O.:
Linux Shell Scripting witch Batch, Sams Publishing, Indianapolis, Indiana, 2004

/Zilm-04/ Zilm, T., Günther, K.:
Bash, ge-packt, mitp-Verlag Bonn, 2004

/Ehses-05/ Ehses, E., Köhler, L., Riemer, P., Stenzel, H., Victor, F.:
Betriebssysteme, Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/LINUX, Pearson-Studium, 2005

/Hammerschall-05/ Hammerschall, U.:
Verteilte Systeme und Anwendungen, Architekturkonzepte, Standards und Middleware-Technologien, Pearson-Studium, 2005

/Lucke-05/ Lucke, R.W.:
Building Clustered Linux Systems, Prentice Hall PTR, 2005

/Tanenbaum-05/ Tanenbaum, A., van Steen, M.:
Verteilte Systeme, Grundlagen und Paradigmen, Pearson-Studium, 2005

/Wörn-05/ Wörn, H., Brinkschulte, U.:
Echtzeitbetriebssysteme, Springer-Verlag, 2005

/Achilles-06/ Achilles, A.:
Betriebssysteme, Springer-Verlag Berlin, Heidelberg, 2006

/Adelstein-07/ Adelstein, T., Lubanovic, B.:
Linux, Schnellkurs für Administratoren, O'Reilly Verlag GmbH & Co. KG, 2007-12-04

/Herold-07/ Herold, H., Lurz, B., Wohlrab, J.:
Grundlagen der Informatik, Person-Studium, 2007

/Willemer-08/ UNIX, Das umfassende Handbuch, Galileo Press, Bonn, 2008

/Zimmer-10/ Zimmer, D., Wöhrmann, B., Schäfer, C., Wischer, S., Kügow, O.:
WMware vSphere 4 – Das umfassende Handbuch, Galileo Press, Bonn, 2010

/Isaacson-11/ Isaacson, W.:
Steve Jobs, C. Bertelsmann Verlag, München, 2011

/Bartmann-12/ Bartmann, E.:
Durchstarten mit Raspberry Pi, O'Reilly, 2012

/Glatz-12/ Glatz, E.:
Differenzanalyse – Linux, Mac OS X, Solaris, Windows: Gemeinsamkeiten und Unterschiede, IX 11/2012

/Kofler-12/ Kofler, M.:
Linux 2012, Installation, Konfiguration, Anwendung, 11. überarbeitete und erweiterte Auflage, Addison-Wesley, 2012

/Krimmer-12/ Krimmer, M., Ochsenkühn, A., Szierbeck, J.:
Mein iPhone & ich, amac-buch Verlag, 2012

/Rehberg-12/ Rehberg, A. I.:
Das inoffizielle Android Systemhandbuch, Franzis Verlag GmbH, 2012

/Tayler-12/ Tayler, D.:
Learning Unix for OS X Mountain Lion, O'Railly, 2012

/Kersken-13/ Kersken, S.:
IT-Handbuch für Fachinformatiker, Galileo Press, 2013

/Greenwald-14/ Greenwald, G.:
Die globale Überwachung, Der Fall Snowden, Droemer Verlag, 2014

/Quade-14/ Quade, J.:
Embedded Linux, lernen mit dem Raspberry Pi, dpunkt.verlag 2014

Ein herzlicher Dank geht an den Verlag Pearson Studium, der freundlicherweise die Bilder aus, z.B. /Tanenbaum-02/ für Dozenten in elektronischer Form zur Verfügung stellt.

Inhaltsverzeichnis

Literaturverzeichnis 1

Inhaltsverzeichnis 7

1. Betriebssysteme 15

1.1. Zitat von Microsoftgründer Bill Gates und die Reaktion von General Motors 15

1.2. Was ist ein Betriebssystem 16

1.3. Geschichte der Betriebssysteme 20
- 1.3.1. Die erste Generation (1945 – 1955) 20
- 1.3.2. Die zweite Generation (1955 – 1965) 21
- 1.3.3. Die dritte Generation (1965 – 1980) 22
- 1.3.4. Die vierte Generation (1980 bis heute) 23

1.4. Arten von Betriebssystemen 23
- 1.4.1. Mainframe-Betriebssysteme 24
- 1.4.2. Server-Betriebssysteme 24
- 1.4.3. Multiprozessor-Betriebssysteme 25
- 1.4.4. Betriebssysteme für den Personal Computer 26
- 1.4.5. Echtzeit-Betriebssysteme 26
- 1.4.6. Betriebssysteme für Smartphones und Tablets 27
  - 1.4.6.1. Android 28
  - 1.4.6.2. Bada 28
  - 1.4.6.3. Blackberry 29
  - 1.4.6.4. iOS 29
  - 1.4.6.5. Symbian 30

1.4.6.6. Windows Phone 30
1.4.7. Betriebssysteme für Chipkarten 31

1.5. Typische Hardware eines Personal Computers 32
1.5.1. Prozessoren 33
1.5.2. Hauptspeicher 35
1.5.3. Ein- / Ausgabegeräte 36
1.5.4. Bussysteme 37

1.6. Beispiele für preiswerter Linux- und Android-Rechner 40
1.6.1. Raspberry Pi 40
1.6.2. Smart PC Stick 2.0 41

1.7. Klassifizierung von Betriebssystemen 43
1.7.1. Klassifikation nach dem Anwendungsgebiet 43
1.7.1.1. Betriebssysteme für allgemeine Anwendungen 43
1.7.1.2. Echtzeitbetriebssysteme 44
1.7.2. Klassifikation nach der vorhandenen Anzahl von Prozessoren 45
1.7.2.1. Ein-Prozessor-Betriebssystem 46
1.7.2.2. Mehr-Prozessor-Betriebssystem 46
1.7.3. Klassifikation nach der Anzahl parallel ablaufender Programme 47
1.7.3.1. Single-Task-Betriebssystem 48
1.7.3.2. Multi-Task-Betriebssystem 48
1.7.4. Klassifikation nach der Anzahl gleichzeitig aktiver Nutzer 49
1.7.4.1. Single-User-Betriebssystem 50
1.7.4.2. Multi-User-Betriebssystem 51

2. Der Betriebssystemkern 52

2.1. Grundlegende Funktionen des Betriebssystemkerns 52

2.2. Das Prozesssystem 53
2.2.1. Programme, Prozeduren, Prozesse und Instanzen 53
2.2.2. Prozesserzeugung 58
2.2.3. Prozessbeendigung....60
2.2.4. Prozesszustände....61
2.2.5. Threads....63
2.2.6. Interprozesskommunikation....65
2.2.6.1. Signale 66
2.2.6.2. Events 67
2.2.6.3. Semaphore 67
2.2.6.4. Binäre Semaphoren - Mutex....68
2.2.6.5. Message 68
2.3. Das Ein- / Ausgabe-System 69
2.3.1. Ein- / Ausgabegeräte....69
2.3.2. Steuereinheiten 70
2.3.3. Direct Memory Access 71
2.3.4. Interrupts 72
2.4. Dateien und Dateisystem 74
2.4.1. Dateinamen....74
2.4.2. Dateizugriff 76
2.4.3. Dateiattribute....77
2.4.4. Dateioperationen....79
2.4.5. Verzeichnisse....80
2.4.6. Realisierung von Dateien 81
2.5. Das Speicherverwaltungssystem 84
2.5.1. Grundlagen der Speicherverwaltung 86

2.5.2. Swapping 89
2.5.3. virtueller Speicher 92
2.5.4. Paging 93
3. Verfügbarkeit und Hochverfügbarkeit 94
3.1. Verfügbarkeit 94
3.2. Hochverfügbarkeit 94
3.3. Verfügbarkeitsklassen 95
3.3.1. Verfügbarkeitsklasse 2 96
3.3.2. Verfügbarkeitsklasse 3 96
3.3.3. Verfügbarkeitsklasse 4 96
3.3.4. Verfügbarkeitsklasse 5 96
3.3.5. Verfügbarkeitsklasse 6 96
3.4. Availability Environment Classification 97
3.5. Vereinbarter Zeitraum der Verfügbarkeit 97
4. Virtualisierung 99
4.1. Vorteile der Virtualisierung 100
4.2. Softwarevirtualisierung 101
4.3. Hardwarevirtualisierung 104
4.4. Netzwerkvirtualisierung 104
4.5. Virtualisierungslösungen 105
5. Das Betriebssystem Unix 106
5.1. Historische Entwicklung 106

5.2. Die Entwicklungsgeschichte von Linux 109
5.3. Schalenmodell von Unix 110
5.4. Logisches Dateisystem 113
5.4.1. Aufbau des Unix-Dateisystems 113
5.4.2. Bewegen im Dateibaum 115
5.4.3. Dateiattribute, Benutzerklassen und Zugriffsrechte 116
5.5. Physisches Dateisystem 118
5.5.1. Aufbau des physischen Dateisystems 118
5.5.2. Aufbau eines Inodes 120
5.5.3. Aufbau eines Verzeichnisses 123
5.5.4. Dateiverwaltungskommandos 123
5.5.5. Gerätedateien 124
5.6. Prozesssystem 125
5.6.1. Prozesshierarchie 125
5.6.2. Hintergrundprozesse 126
5.6.3. Prozesspriorität 127
5.6.4. Prozesszustände 129
5.6.5. Kommandos zum Prozesssystem 130
5.7. Die Unix-Kommandos als Filter 131
5.7.1. Die Bourne-Shell als Kommandointerpreter 131
5.7.2. Ein- /Ausgabeumlenkung 133
5.7.3. Kommandoverkettung mit der Pipe 134
5.7.4. weitere wichtige Kommandos 134
6. Betriebssysteme für Smartphones und Tablets (Wikipedia) 135
6.1. Android 135

6.1.1. Die Geschichte von Android 136
6.1.2. Die Architektur von Android 136
6.1.3. Bezug von Software (Apps) für Android 139
6.2. Apple iOS 140
6.2.1. Die Geschichte von Apple iOS 141
6.2.2. Das Bedienkonzept 143
7. Shell-Programmierung 145
7.1. Kommandosyntax 145
7.2. Variable und Parameter 151
7.2.1. Umgang mit Shell-Variablen 151
7.2.2. Spezielle Variable 155
7.2.3. Positionsparameter 156
7.3. Ersetzungsmechanismen 162
7.4. Kommandoersetzung 166
7.5. Verarbeitungsstrukturen 168
7.5.1. Bedingte und einfache Fallunterscheidung 168
7.5.2. Kommando test 170
7.5.2.1. Eigenschaften von Dateien 171
7.5.2.2. Eigenschaften und Vergleiche von Zeichenketten 172
7.5.2.3. Algebraische Vergleiche ganzer Zahlen 173
7.5.2.4. Logische Verknüpfung von Bedingungen 175
7.5.3. Mehrfache Fallunterscheidung 177
7.5.4. Abweisende und nicht abweisende Schleife 179
7.5.4.1. Abweisende Schleife 179
7.5.4.2. Nichtabweisende Schleife 179

7.5.5. Zählschleife ... 181
7.5.6. Das Kommando expr ... 184
7.5.6.1. Vergleichsoperatoren ... 185
7.5.6.2. Arithmetische Operatoren ... 185
7.5.6.3. Spezielle Operatoren ... 186

7.6. Funktionen ... 187

8. Das Betriebssystem Windows ... 194

8.1. Die Geschichte von MS-DOS und Windows ... 194

8.2. Windows-Programmierung ... 197
8.2.1. Die Programmierschnittstelle Win32 ... 197
8.2.2. Die Registrierung ... 198

8.3. Die Struktur des Betriebssystems ... 200
8.3.1. Die Hardware Abstraction Layer ... 200
8.3.2. Der Betriebssystemkern ... 202
8.3.3. Realisierung von Objekten ... 203

8.4. Prozesse und Threads ... 206
8.4.1. Grundlegende Konzepte ... 206
8.4.2. Scheduling ... 210
8.4.3. Starten von Windows ... 212

8.5. Speicherverwaltung ... 214
8.5.1. Konzepte ... 214
8.5.2. Systemcalls zur Speicherverwaltung ... 218

8.6. Ein-/Ausgabe ... 219

8.7. Das Dateisystem ... 223

8.7.1. MS-DOS ... 223
8.7.2. Die FAT (File Allocation Table) ... 224
8.7.3. Das NTFS (NT File System) ... 225

9. Echtzeitbetriebssysteme ... 228

9.1. Einleitung ... 228

9.2. Grundlagen ... 229
9.2.1. Rechtzeitigkeit ... 230
9.2.2. Gleichzeitigkeit ... 231
9.2.3. Determiniertheit ... 231
9.2.3.1. harte Echtzeitsysteme ... 232
9.2.3.2. weiche Echtzeitsysteme ... 232

9.3. Elemente von Echtzeitsystemen ... 233
9.3.1. Taskmodell ... 235
9.3.2. Semaphore ... 236
9.3.3. Messages ... 237
9.3.4. Spezialisierte Warteschlangen (Pipes) ... 238
9.3.5. Unterbrechungen und Signale ... 239
9.3.6. Systemuhren und Zeitgeber ... 240

Tabellenverzeichnis ... 241

Abbildungsverzeichnis ... 243

Definitionsverzeichnis ... 247

Index ... 248

1. Betriebssysteme

1.1. Zitat von Microsoftgründer Bill Gates und die Reaktion von General Motors

Auf einer Computermesse hat **Bill Gates** die Computer-Industrie mit der Auto-Industrie verglichen und das folgende Statement gemacht:

"Wenn General Motors (GM) mit der Technologie so mitgehalten hätte wie die Computer-Industrie, dann würden wir heute alle 25-Dollar-Autos fahren, die 1000 Meilen pro Gallone Sprit fahren würden".

Als Antwort darauf veröffentlichte General Motors (von Mr. Welch selbst) eine Presseerklärung mit folgendem Inhalt:

Wenn General Motors eine Technologie wie Microsoft entwickelt hätte, dann würden wir alle heute Autos mit folgenden Eigenschaften fahren:

1. Ihr Auto würde ohne erkennbaren Grund zweimal am Tag einen Unfall haben.
2. Jedes Mal, wenn die Linien auf der Straße neu gezeichnet würden, müsste man ein neues Auto kaufen.
3. Gelegentlich würde ein Auto ohne erkennbaren Grund auf der Autobahn einfach ausgehen, und man würde das akzeptieren, neu starten und weiterfahren.
4. Wenn man bestimmte Manöver durchführen würde, wie z. B. eine Linkskurve, würde das Auto einfach ausgehen und sich weigern, neu zu starten. Man müsste dann den Motor erneut installieren.
5. Man könnte nur alleine in dem Auto sitzen, es sei denn, man würde "Car95" oder "CarNT" kaufen. Aber dann müsste man jeden Sitz einzeln bezahlen.

6. Macintosh würde Autos herstellen, die mit Sonnenenergie fahren, zuverlässig laufen, fünfmal so schnell und zweimal so leicht zu fahren wären, aber sie würden nur auf 5% der Straßen laufen.
7. Die Kontrollleuchte und die Warnlampe für Temperatur und Batterie würde durch eine "Genereller Auto-Fehler"-Warnlampe ersetzt werden.
8. Neue Sitze würden erfordern, dass alle dieselbe Gesäßgröße haben.
9. Das Airbag-System würde "Sind Sie sicher?" fragen, bevor es auslöst.
10. Gelegentlich würde das Auto Sie ohne jeden erkennbaren Grund aussperren. Sie könnten nur wieder mit einem Trick aufschließen, und zwar müsste man gleichzeitig den Türgriff ziehen, den Schlüssel drehen und mit einer Hand an die Radioantenne fassen.
11. General Motors würde Sie zwingen, mit jedem Auto einen De-Luxe-Kartensatz der Firma Rand Mc-Nally (seit neustem eine GM-Tochter) mit zu kaufen, auch wenn Sie diesen Kartensatz nicht brauchen oder möchten. Wenn Sie diese Option nicht wahrnehmen, würde das Auto sofort 50% langsamer werden (oder schlimmer). Darüber hinaus würde GM deswegen ein Ziel von Untersuchungen der Justiz werden.
12. Immer dann, wenn ein neues Auto von GM vorgestellt werden würde, müssten alle Autofahrer das Autofahren neu erlernen, weil keiner der Bedienhebel genau so funktionieren würde, wie in den alten Autos.
13. Man müsste den "Start-Knopf" drücken, um den Motor auszuschalten.

1.2. Was ist ein Betriebssystem

Um ein Rechnersystem für bestimmte Aufgaben benutzen zu können, benötigt man ein **vollständiges Programm**. Vollständig bedeutet dabei, dass das Programm alle Anweisungen enthält, um die zu verarbeitenden Daten einzulesen (von Tastatur, Diskette, Scanner, usw.), Algorithmen auf sie anzuwenden und die Ergebnisse geeignet darzustellen (Bildschirm, Drucker,

Festplatte usw.). Die Anweisungen müssen dazu im Maschinenbefehlssatz des verwendeten Prozessors vorliegen.

Ein vollständiges Programm zu einer bestimmten Aufgabe (z.B. Verwaltung der Konten einer Bank, Informationsserver im Internet, Steuerung einer CNC-Werkzeugmaschine u.v.m.) ist damit sehr komplex und aufwendig herzustellen.

Bedenkt man, dass heutige Rechnersysteme sich selbst innerhalb einer Rechnerfamilie vielfältig in Speicherausstattung, Art und Umfang der angeschlossenen Geräte unterscheiden, so wird klar, dass die Erstellung vollständiger Programme im obigen Sinne für jede mögliche Rechnerkonstellation ein praktisch undurchführbares Unternehmen ist.

Die Lösung dieses Problems heißt auch hier wie in anderen Ingenieurbereichen: **Modularisierung**.

Programme werden in **Module** zerlegt, die zueinander über definierte **Schnittstellen** in Beziehung stehen. Somit ist es möglich, innerhalb eines Programms einen Modul durch einen anderen mit gleicher Schnittstelle zu ersetzen, um das Programm an eine andere Rechnerkonstellation anzupassen.

Definition 1 - Betriebssystem:

> **~ (operating system) heißt das vollständige Programm, das zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bildet und insbesondere die Abwicklung von Programmen steuert und überwacht.**

In einem vollständigen Programm gibt es aufgabenspezifische Module, bezogen auf das Einsatzgebiet und Module, die von der konkreten Aufgabe relativ unabhängig und für andere Aufgaben in der gleichen Weise einsetzbar sind.

Die Auswahl und Zusammenstellung der allgemeingültigen Module wird bestimmt durch die eingesetzte Hardware und die Art der Programme, die durch diese Module unterstützt werden sollen.
Sie ist für viele Programme, die auf einem Rechner abgearbeitet werden sollen, gleich und unterscheiden sich wiederum etwas von Rechner zu Rechner.

Es bietet sich daher an, diese allgemeingültigen Module jeweils für einen Rechner zu einem System zusammenzufassen, dem **Betriebssystemkern**.

Die Zusammenfassung der aufgabenspezifischen Module einer Anwendung bildet das **Anwendungsprogramm**, welches über die definierten Schnittstellen auf die Funktionen des Betriebssystemkerns zugreift.

Um eine möglichst bequeme Nutzung von Betriebssystemkern und Anwendungsprogramm zu ermöglichen, erhält der Anwender zum Betriebssystem eine Reihe von nützlichen und unbedingt notwendigen Anwendungsprogrammen, so genannten **Dienstprogrammen** dazu.

Betriebssystemkern und Dienstprogramme bilden zusammen das **Betriebssystem**. Die einzelnen Betriebssysteme unterscheiden sich im konkreten Funktionsumfang des Betriebssystemkerns und seiner Programmschnittstellen sowie der Dienstprogramme. Dennoch findet man viele Grundprinzipien in allen Betriebssystemen wieder. In Abbildung 1. ist der Zusammenhang zwischen den Komponenten eines Rechnersystems in dem Schalen- bzw. Schichtenmodell zusammengefasst. Dienstprogramme und das Anwendungsprogramm sind einer gemeinsamen Schale bzw. Schicht zugeordnet.

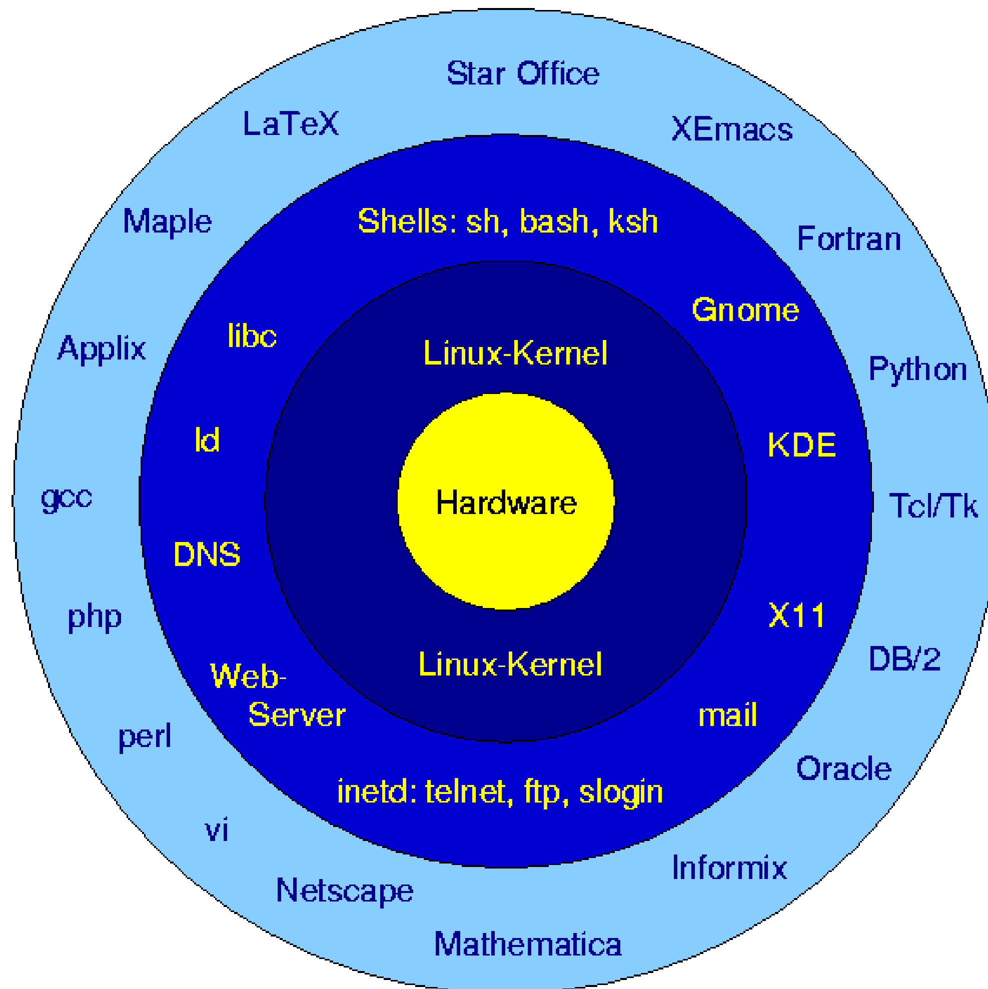


Abbildung 1.: Schalen- bzw. Schichtenmodell eines Rechners

1.3. Geschichte der Betriebssysteme

Die geschichtliche Unterteilung von Betriebssystemen lässt sich in vier Generationen untergliedern. Als Beginn wird die Entwicklung des englischen Mathematikers **Charles Babbage** (1792 – 1871) gewertet. Seine Arbeiten beschäftigten sich mit der Entwicklung eines (wahren) Digitalrechners. Leider ist es ihm nie gelungen seine „analytische Maschine“ zu bauen, da er nur auf rein mechanische Elemente (Räder, Gestänge, Zahnräder, u.s.w.) zurückgreifen konnte. Diesen fehlte die notwendige Präzision.

Bereits Babbage erkannte die Notwendigkeit von Software für seine Maschine. Dafür stellte er eine Dame namens **Ada Lovelace** ein. Die **Programmiersprache ADA** ist nach ihr benannt.

1.3.1. Die erste Generation (1945 – 1955)

Nach den erfolglosen Bestrebungen von Babbage mit mechanischen Bauteilen wurden in der Mitte der 40er Jahre mit Elektronenröhren und Relais Rechenmaschinen konstruiert. Als bekannteste Entwickler sind zu nennen:

- **Howard Aiken** an der Havard Universität,
- **John von Neumann** am Institut für Advanced Study in Princton,
- **J.P. Eckert** und **William Mauchley** an der Universität Pennsylvania,
- **Konrad Zuse** in Deutschland.

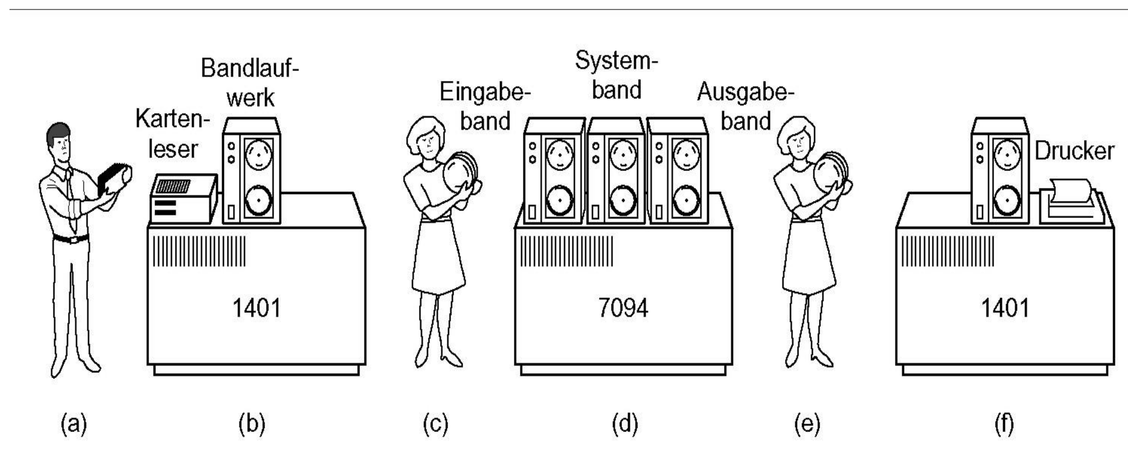
Abbildung 2.: Konrad Zuse

Die Programmierung wurde in Maschinensprache durchgeführt sowie häufig durch Verdrahtung der Steckkarten. Programmiersprachen selbst Assemblersprachen und Betriebssysteme waren unbekannt.

In den frühen 50er Jahren wurden die Arbeiten durch die Verwendung von Lochkarten in gewissem Rahmen verbessert.

1.3.2. Die zweite Generation (1955 – 1965)

Die Entwicklung des Transistors hat in der Mitte der 50er Jahre bedeutet eine radikale Veränderung. Die Rechner wurden zuverlässiger und konnten an Kunden verkauft werden, um sinnvolle Arbeit zu leisten. Diese Rechner werden seitdem **Mainframe** genannt. Eine typische Bearbeitung von Jobs ist in Abbildung 3. zu sehen:



(a) Die Programmierer bringen die Lochkartenstapel zur **IBM 1401**.
(b) Die **IBM 1401** liest den Stapel von Jobs auf ein Band.
(c) Ein Operator trägt das Eingabeband zur **IBM 7094**.
(d) Die **IBM 7094** führt die Berechnungen aus.
(e) Ein Operator trägt das Ausgabeband zur **IBM 1401**.
(f) Die **IBM 1401** druckt die Ergebnisse aus.

Abbildung 3.: Job-Verarbeitung in einem Mainframe

Die Rechner der zweiten Generation wurden meistens für wissenschaftliche oder technische Berechnungen eingesetzt, wie das Lösen partieller Differenzialgleichungen. Sie wurden überwiegend in der **Programmiersprache FORTRAN** oder **Assembler-Sprache** programmiert.

1.3.3. Die dritte Generation (1965 – 1980)

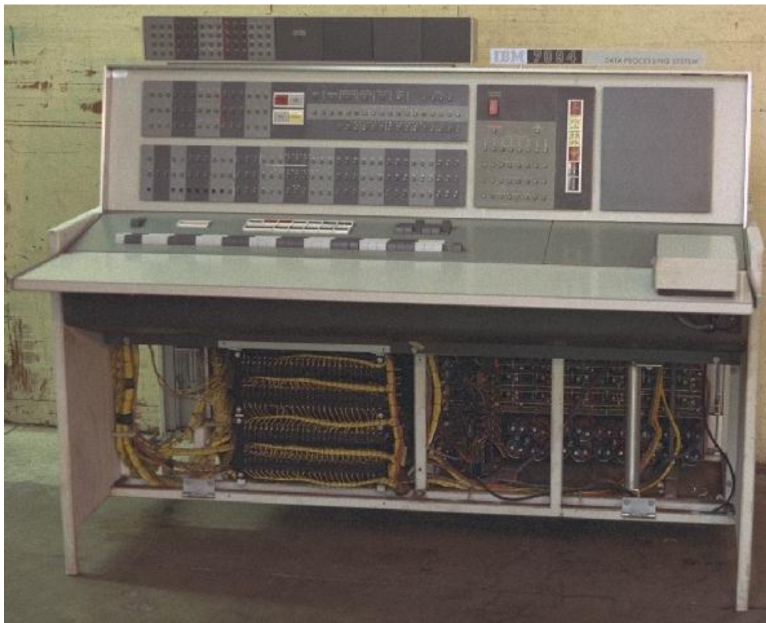


Abbildung 4.: IBM 7094

Zu Beginn der 60er Jahre verfolgten die meisten Computerhersteller zwei verschiedene, zueinander völlig inkompatible Produktlinien:

- Auf der einen Seite standen die wortorientierten, großen wissenschaftlichen Rechner, wie die **IBM 7094**, die für numerische Berechnungen in Wissenschaft und Technik verwendet wurden.
- Auf der anderen Seite standen die zeichenorientierten, kommerziellen Rechner, wie die **IBM 1401**, die im Wesentlichen für das Sortieren und Ausdrucken von Bändern in Banken und Versicherungen eingesetzt wurden.

IBM versuchte dieses Problem mit der Entwicklung des Systems /360 zu lösen. Es handelte sich dabei um eine Serie softwarekompatibler Rechner, die sich lediglich im Preis und in der Leistungsfähigkeit unterschieden.

Die größte Stärke dieser Rechner war auch zugleich ihre größte Schwäche. Die gesamte Software, also auch das Betriebssystem, musste auf allen Modellen der Rechnerserie laufen. Das Betriebssystem OS/360 war ein großes und außergewöhnlich komplexes Betriebssystem. Es bestand aus Millionen Zeilen Assemblercode und wurde von Tausenden von Programmierern entwickelt und enthielt Tausende von Fehlern.

Die reine Job-Bearbeitung hatte viele Nachteile, so waren die Programmierer immer räumlich von den Rechnern getrennt und die Bearbeitung der einzelnen Jobs erfolgte rein sequenziell. Folgende Entwicklungen verfolgten das Ziel, die Antwortzeiten zu verkürzen und mehrere Jobs gleichzeitig zu bearbeiten. Das Prinzip nennt man Timesharing. Es entstanden diese Systeme:

- **CTSS**, **C**ompatible **T**ime **S**haring **S**ystem,
- **MULTICS**, **MULT**iplexed **I**nformation and **C**omputing **S**ystem,
- **Unix**, **System V** von AT&T und **BSD Unix** (Berkeley Software Distribution)

1.3.4. Die vierte Generation (1980 bis heute)

Durch die Entwicklung von **LSI**-Schaltungen (**L**arge **S**cale **I**ntegration), das sind hochintegrierte Schaltkreise mit damals Tausenden Transistoren pro Quadratzentimeter Silizium begann die Entwicklung des Personal Computerss auch Mikrocomputer genannt.

Die ersten Betriebssysteme waren:

1. für den **Intel 8080** , **CP/M** (**C**ontrol **P**rogram for **M**icrocomputers) der Firma Digital Research,
2. für den **IBM PC**, **Intel 8086**, **DOS** (**D**isc **O**perating **S**ystem) der Firma Seattle Computer Products,
3. für nachfolgende Intel-Prozessoren, **MS-DOS** (**M**icro**s**oft **D**isc **O**perating **S**ystem) der Firma Microsoft

1.4. Arten von Betriebssystemen

Die ganze Geschichte und Entwicklung hat eine Vielzahl von unterschiedlichen Betriebssystemen hervorgebracht, die selten in ihrer Gesamtheit wahrgenommen wird. Überblicksweise sollen diese hier kurz vorgestellt und angesprochen werden. Auf einzelne Arten wird in nachfolgenden Abschnitten wieder eingegangen.

Diese Unterteilung ist nicht gleichbedeutend mit der Klassifizierung von Betriebssystemen, wie sie im Abschnitt 1.6. vorgenommen wird.

1.4.1. Mainframe-Betriebssysteme

Im Highend-Bereich aller Systeme sind Betriebssysteme für Mainframes. Diese raumgroßen Geräte sind auch heute noch in großen Rechenzentren von Firmen zu finden. Sie zeichnen sich durch eine sehr hohe Ein- / Ausgabebandbreite gegenüber allen anderen Systemen aus. Ausstattungen mit 1000 Festplatten und mehreren Terabyte sind in diesem Bereich nicht ungewöhnlich.

In der heutigen Zeit werden solche Systeme als große Webserver, Server für E-Commerce oder Server für Business-to-Business-Anwendungen eingesetzt.

Das Betriebssystem muss eine Vielzahl von Prozessen mit einem hohen Bedarf an schneller Ein- / Ausgabe verwalten. Bezüglich der Prozessverwaltung sind alle Möglichkeiten in diesem Bereich zu finden:

- Batchbearbeitung,
- Transaktionssteuerung und
- Zeitaufteilungsverfahren

Als typischer Vertreter dieser Betriebssystemart ist das Mainframe-Betriebssystem **OS/390** der **Firma IBM** als Nachfolgesystem des bereits erwähnten **OS/360**.

1.4.2. Server-Betriebssysteme

Eine Ebene unter den Mainframe-Betriebssystemen sind Server-Betriebssysteme angesiedelt, die auf sehr großen Personal Computern, Workstations oder sogar Mainframes laufen.

Sie bedienen eine Vielzahl von Benutzern, die über das Netzwerk zur gleichen Zeit auf verteilte Hardware- und Softwareressourcen zugreifen. Dazu können sie Druckdienste, Dateifreigaben und Webdienste anbieten.

Typische Vertreter der Serverbetriebssysteme sind Unix, Linux und Windows-Server 2008.

Ein Beispiel für einen Dateiserver ist in Abbildung 5 dargestellt.

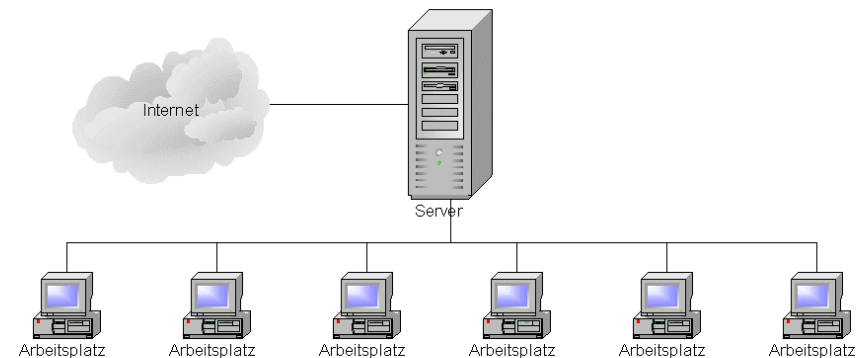


Abbildung 5.: Beispiel für einen Dateiserver

1.4.3. Multiprozessor-Betriebssysteme

Eine neuere Entwicklung erhöht die Rechenleistung durch Zusammenschalten mehrerer Prozessoren in einem oder mehreren Rechnern. In Abhängigkeit der Verschaltung untereinander nennt man diese Systeme:

- Parallelcomputer,
- Multicomputer oder
- Multiprozessorcomputer

Die neusten Entwicklungen gehen dabei in Richtung von
- Computernetzwerken bzw. –Clustern oder
- Grid-Computing.

Als Beispiele für diese Betriebssystemart sind die meisten Unix- oder Linuxdistributionen oder Windows-Serversysteme zu nennen.

1.4.4. Betriebssysteme für den Personal Computer

Den kommerziell am meisten verbreiteten Bereich der Betriebssysteme stellen solche für den Personal Computer dar. Mit ihnen können die Benutzer im professionellen und privaten Umfeld die verschiedensten Aufgaben von der Textbearbeitung, der Bearbeitung von Filmen, oder alle Aufgaben rund um das Internet bewältigen.

Die Verbreitung dieser Systeme ist so gigantisch, dass es wohl nur wenige Menschen gibt, die keinen Umgang mit solchen Systemen haben. Mache davon wissen teilweise überhaupt nicht, dass es noch andere Systeme gibt.

Als Beispiele für diese Betriebssystemart sind die Microsoft-Produkte Windows, das Betriebssystem für den Macintosh der Firma Apple oder mit steigender Bedeutung das Linux-Betriebssystem zu nennen.

Abbildung 6.: Betriebssysteme für PCs

1.4.5. Echtzeit-Betriebssysteme

Die Art der Echtzeitbetriebssysteme zeichnet sich dadurch aus, dass die Zeit ein sehr wichtiger Parameter bei der Ressourcenvergabe ist. Meistens gibt es sehr wichtige Zeitpunkte, die unbedingt eingehalten werden müssen. Einsatzgebiete sind in vielen Bereichen der industriellen Produktion, zum Beispiel der Fertigungs- oder der Robotersteuerung.

Aus der Vielzahl an möglichen und sehr spezifischen Systemen sind zu nennen:

- OS9 der Firma Microware Systems Corp.,
- LynxOS der Firma Lynx Real Time Systems oder
- QNX der Firma QNX Software Systems.

1.4.6. Betriebssysteme für Smartphones und Tablets

Der Markt für Smartphones und Tablets ist in den letzten Jahren sprunghaft angestiegen. Ausgangspunkt war die Vorstellung der ersten iPhone-Generation am 9. Januar 2007 auf der Macworld Conference & Expo in San Francisco durch Steve Jobs, damaliger Apple-CEO.

Ein **Smartphone** ist ein Mobiltelefon, das mehr Computerfunktionalität und -konnektivität als ein herkömmliches fortschrittliches Mobiltelefon zur Verfügung stellt. Erste Smartphones vereinigten die Funktionen eines PDA bzw. Tablet-Computers mit der Funktionalität eines Mobiltelefons. Später wurde dem kompakten Gerät auch noch die Funktion eines transportablen Medienabspielgerätes, einer Digital- und Videokamera und eines GPS-Navigationsgeräts hinzugefügt.

Viele moderne Smartphones sind mit einem hoch auflösenden berührungsempfindlichen Bildschirm ausgestattet. Dieser kann sowohl Standard-Webseiten als auch mobil optimierte Webseiten darstellen. Eine schnelle Internet-Anbindung erfolgt mittels mobilem Breitband und WLAN.

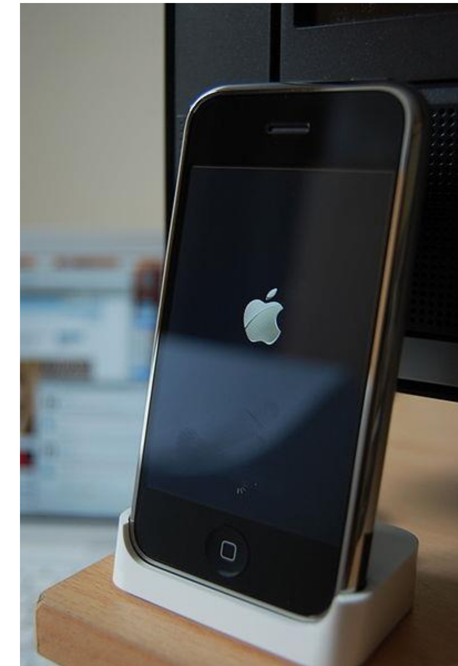


Abbildung 7.: Apple – iPhone der ersten Generation

Folgende Systeme mit den Betriebssystemen in alphabetischer Reihenfolge sind derzeit auf dem Mark:

1.4.6.1. Android

Android wird unter Federführung von Google entwickelt, der Programmcode ist frei verfügbar. Handyanbieter passen ihn auf ihre Geräte an und sind auch dafür zuständig, Software-Aktualisierungen für ihre Handys zu liefern. Das führt zuweilen zu Frust bei den Nutzern, wenn Updates auf die neueste Android-Version länger auf sich warten lassen. Das System gilt als flexibel und offen. Googles „**Play Store**“ (vormals „Android Market“) bietet eine sehr große Zahl an Zusatzprogrammen. Doch auch Apps aus anderen Quellen lassen sich installieren. Viele Android-Handys haben einen Steckplatz für Speicherkarten. In der Regel lassen sie sich zum einfachen Austausch von Musik und anderen Dateien als USB-Laufwerk oder zumindest über das Media Transfer Protocol (MTP) an den Computer anschließen. Die Synchronisierung von Adressen und Kalender erfolgt standardmäßig übers Internet. Manche Handyanbieter bieten für ihre Geräte PC-Software für eine direkte Synchronisation per USB.

1.4.6.2. Bada

Bada ist eine Entwicklung von Samsung und läuft nur auf Geräten dieses Anbieters. Die offizielle Quelle für Bada-Apps ist Samsungs Online-Laden „Samsung Apps“. Für die direkte Synchronisation von Adressbuch und Kalender stellt Samsung das PC-Programm Kies zum kostenlosen Herunterladen bereit. Für sonstigen Datenaustausch funktionieren Bada-Handys am Computer als USB-Laufwerk.

1.4.6.3. Blackberry

Blackberry-OS ist das System des kanadischen Anbieters Blackberry (vormals Research in Motion, RIM). Blackberry-Geräte sind traditionell vor allem für Unternehmenskunden gedacht und bieten spezielle Synchronisations- und E-Mail-Funktionen, die eine entsprechende Netzwerkrechner-Infrastruktur voraussetzen. Außerdem war lange Zeit ein spezieller Handytarif nötig. Erst seit der Blackberry-OS-Version 10 brauchen Blackberry-Nutzer keinen speziellen Mobilfunkvertrag mehr und können neben den eigenen Diensten von Blackberry uneingeschränkt auch andere E-Mail-, Kalender- und Adressbuch-Dienste nutzen. Blackberry liefert ein Programm zur direkten Synchronisation mit dem PC mit. Blackberry-Nutzer können Apps aus der „Blackberry App World“ oder aus anderen Quellen installieren.

1.4.6.4. iOS

iOS von Apple läuft neben dem iPhone auch auf dem iPod Touch und auf dem iPad. Das System ist beliebt wegen seiner einfachen Steuerung und dem enormen Angebot an Apps, wird aber für seine Geschlossenheit kritisiert. Apps kommen nur aus Apples „App Store“, für den ein Nutzerkonto nötig ist. Der interne Speicher von iOS-Geräten lässt sich nicht mit Speicherkarten erweitern. Der Zugriff auf den Speicher ist stark beschränkt. Der Datenaustausch mit dem PC läuft über das Programm iTunes. Das Programm muss aus dem Internet geladen werden. Bis zur iOS-Version 4 war es auch zur Aktivierung der Geräte notwendig. Erst seit iOS 5, das zusammen mit dem iPhone 4S veröffentlicht wurde, ist es möglich, ein neues iPhone auch ohne PC mit iTunes in Betrieb zu nehmen.

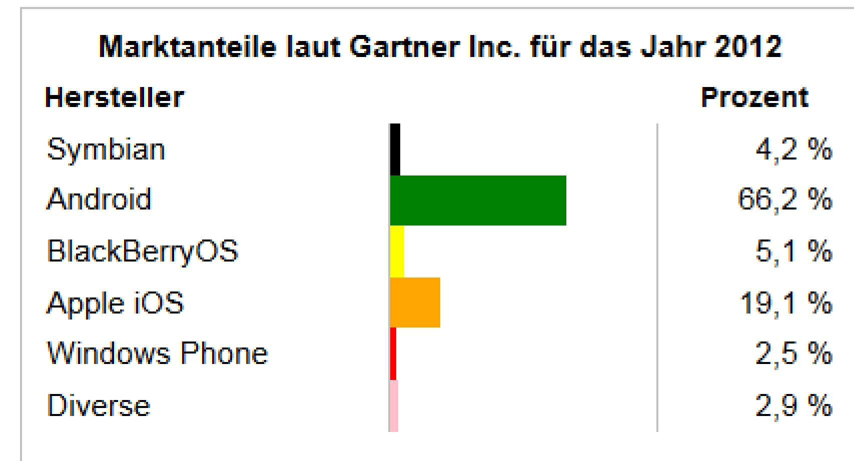
1.4.6.5. Symbian

Symbian ist ein Betriebssystem, das inzwischen nur noch Nokia einsetzt. Das System gilt als offen und vielseitig, aber kompliziert. Apps kommen von Nokias „Ovi Store“ oder aus anderen Quellen. Zum Datenaustausch melden sich Symbian-Handys am PC als USB-Laufwerk an. Bei der Synchronisation von Adressen und Kalender setzt das System seit der Version Symbian^3 wie andere Systeme verstärkt auf Internetdienste. Die PC-Software Ovi Suite hilft aber zum Beispiel beim Übertragen von Navigations-Kartenmaterial auf das Handy. In Zukunft will Nokia seine Smartphones mit Windows Phone 7 betreiben. Es gibt aber bisher auch immer noch neue Geräte mit Symbian-System. Die neuesten Versionen des Systems tragen klingende Namen wie „Anna“ und „Belle“.

1.4.6.6. Windows Phone

Windows Phone von Microsoft wird wie Android von verschiedenen Handy-Anbietern genutzt, die Microsoft dafür aber Lizenzgebühren zahlen. Microsoft macht enge Vorgaben für die Handy-Hardware, kümmert sich dafür aber auch um die Systemaktualisierungen. Das System ist erheblich besser zu bedienen, aber auch geschlossener als der Vorgänger Windows Mobile 6. Apps können nur von Microsofts „Windows Phone Store“ installiert werden, es ist ein Nutzerkonto nötig. Die Synchronisation von Adressbuch und Kalender läuft primär übers Internet. Musik und Fotos kann der Nutzer seit der neuesten Version Windows Phone 8 auch ohne Zusatzsoftware zwischen PC und Handy austauschen - bei älteren Versionen ist dafür das PC-Programm Zune nötig. Weitere Neuerung: Windows-Phone-8-Handys können nun auch einen Speicherkartensteckplatz haben.

Der Markt für Smartphones und Tablets ist sehr umkämpft, da sehr lukrativ. Ständig finden Veränderungen der Marktanteile statt. Für das Jahr 2012 hat Gartner Inc. folgende Anteile der einzelnen Systeme veröffentlicht.



Marktanteile laut Gartner Inc. für das Jahr 2012

Hersteller	Prozent
Symbian	4,2 %
Android	66,2 %
BlackBerryOS	5,1 %
Apple iOS	19,1 %
Windows Phone	2,5 %
Diverse	2,9 %

Tabelle 1.: Marktanteile im Smartphone-Geschäft

1.4.7. Betriebssysteme für Chipkarten

Abbildung 8.: Gesundheitskarte ab 2011

Die kleinsten Betriebssysteme laufen auf Smart Cards, diese haben die Größe einer Kreditkarte und besitzen einen eigenen Prozessor. Dabei sind sehr harte Bedingungen an die Rechenleistung, den Stromverbrauch und die Robustheit zu stellen.

Viele dieser Systeme sind javaorientiert und besitzen eine **JVM** (**J**ava **V**irtual **M**aschine). So genannte Java-Applets werden auf die Smart Card geladen und können durch die JVM sofort bearbeitet werden.

1.5. Typische Hardware eines Personal Computers

Ein Betriebssystem ist sehr eng mit der Hardware eines Computers verbunden und erweitert den Befehlssatz des Rechners und verwaltet dessen Ressourcen. Die Zusammenarbeit zwischen Hardware und Betriebssystem entscheidet über die Leistungsfähigkeit des Gesamtsystems.

Eine sehr einfache Struktur ist in Abbildung 9 zur sehen.

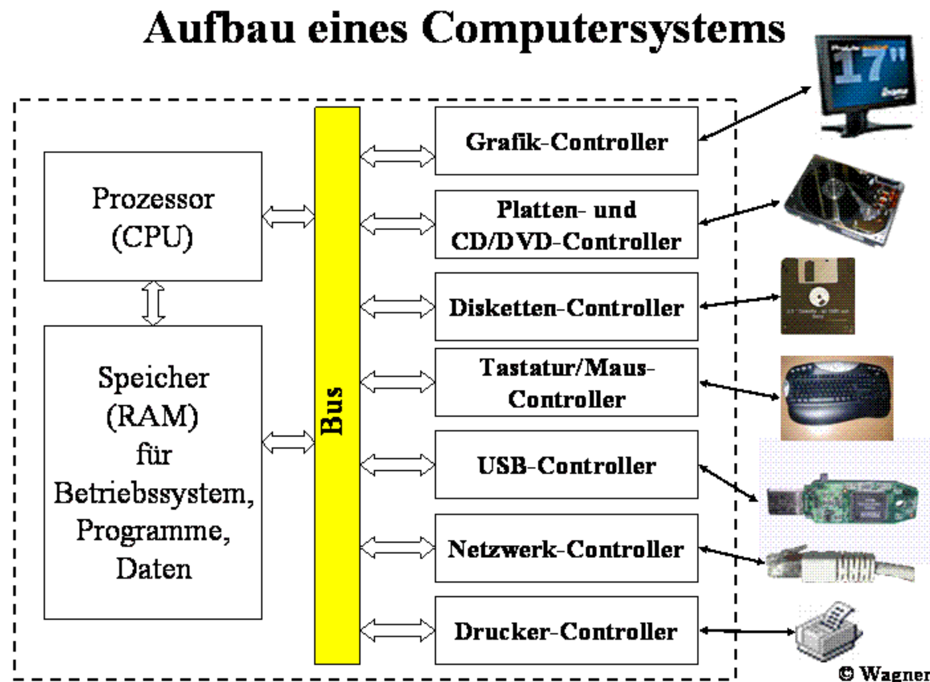


Abbildung 9.: einfache Struktur eines Personal Computers

Grundsätzlich bauen die Strukturen auch heutiger Computer prinzipiell auf der John von-Neumann-Architektur auf, siehe Abbildung 10.

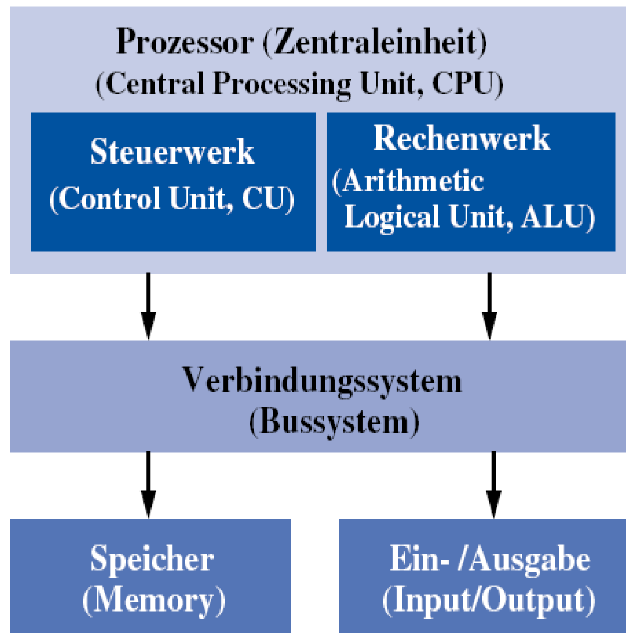


Abbildung 10.: von-Neumann-Architektur

Im Folgenden sollen einige Hardwarekomponenten kurz vorgestellt werden.

1.5.1. Prozessoren

Das Herzstück eines jeden Computers ist der Prozessor, als **CPU** (**C**entral **P**rocessor **U**nit) bezeichnet. Hier gibt es unterschiedliche Ansätze in der Entwicklung. Jede CPU verfügt über einen Befehlssatz, d.h. eine Anzahl von

Anweisungen, die sie ausführen kann. Auf Grund der Verschiedenartigkeit dieser Befehlssätze ist verständlich, warum beispielsweise ein **Intel-Pentium-Prozessor** nicht die Programme eines **IBM-Power-Prozessors** oder eines **SUN-Sparc-Prozessors** verarbeiten kann und umgekehrt der **IBM-Power-Prozessor** nicht die Programme eines Intel-Pentium-Prozessors, genauso, wie auch der Sparc-Prozessor diese nicht verarbeiten kann.

Abbildung 11.: Prozessoren von AMD, Intel, Oracle-Sun und IBM, sowie für Smartphones

Bezüglich der Anzahl verfügbarer Befehle haben sich zwei grundsätzliche Strukturen entwickelt:

- **CISC** (***c***omplex ***i***nstruction ***s***et ***c***omputer = CPU mit komplexen Befehlssatz) und
- **RISC** (***r***educed ***i***nstruction ***s***et ***c***omputer = CPU mit reduziertem Befehlssatz).

Die **RISC** brauchen weniger Transistorfunktionen, sind auf kleinerem Platz auf dem **Chip** unterzubringen und damit schneller und aufgrund der geringeren Komplexität zuverlässiger.

Auch wenn die fehlenden Befehle durch mehrere einfache Befehle ersetzt werden müssen, stellt sich letztlich heraus, das **RISC** wesentlich schneller sind als vergleichbare **CISC**. Der IBM-Forscher **John Cocke** bewies 1974, das ganze 20 Prozent aller möglichen Rechenoperationen eines Prozessors 80 Prozent der Rechenarbeit leisten. Daraufhin wurde die RISC-Technologie entwickelt.

Im Bereich der Personal Computer (CISC) haben sich zwei Konkurrenten in der Prozessorentwicklung herausgebildet, die sich ständig durch neue Entwicklungen und Marketingstrategien neu Marktanteile sichern wollen.

1.5.2. Hauptspeicher

Die zweitwichtigste Komponente in einem Computer ist der Speicher. Idealerweise sollte dieser gleich schnell wie die CPU sein, da er sonst diese „ausbremst". Weiterhin sollte er auch sehr groß und sehr billig sein. Da diese Kriterien sehr schwer zu verwirklichen sind, muss man für diesen Bereich den Kompromiss zwischen der Speichergröße, -geschwindigkeit und dem Preis eingehen.

Die schnellste Speicherstruktur ist in der CPU selbst untergebracht und wird als Registerstruktur bzw. Registersatz bezeichnen.

Etwas langsamer ist der Cache, der teilweise ebenfalls auf dem Chip oder in unmittelbarer Nähe zu diesem angeordnet ist. Die Kosten für diese Speicherart sind vergleichbar extrem hoch, da sie sehr schnell sind.
Um einen sehr großen, aber trotzdem preiswerten Speicherbereich zu haben, wird in den Computern Hauptspeicher eingesetzt.

Heute erreicht man bei „modernen" Personal Computern schon Größenordnungen bis 32 Gbyte Speicherraum. Da dies nicht komplett ausreichend ist, werden weitere Speichermedien extern, als Ein- / Ausgabegeräte eingesetzt.

Abbildung 12.: Hauptspeicher für Laptop

1.5.3. Ein- / Ausgabegeräte

Die klassischen externen Speichermedien haben eine magnetische Schicht zur Speicherung der Daten. Die langsamste und kapazitiv kleinste Möglichkeit sind Disketten mit 1,44 MByte bzw. 2,88 Mbyte, oder erweitert bis zu 200 Mbyte Speicherkapazität. Abbildung 13 zeigt ein Standarddiskettenlaufwerk.

Abbildung 13.: Diskettenlaufwerke im Wandel der Zeit 8“, 5¼“ und 3½“

Werden sehr große Speicherbereiche für relativ kleinen Preis benötigt, sind z.Z. noch Festplattenlaufwerke eingesetzt. Immer wieder kann man in der Fachpresse von neuen Kapazitätsgrenzen lesen, in die die Hersteller vorgestoßen sind.

In Abbildung 14 sind zwei Vertreter in heutigen Computern, links für Standard-Personal-Computer von Western Digital 3,5“ und rechts für Laptops von Toshiba 2,5“ zu sehen.

Abbildung 14.: Festplattenlaufwerke

Als relativ neue Technik werden SSD-Festplatten die bisherigen Festplatten mit mechanisch beweglichen Teilen ablösen. Es ist nur eine Frage der Zeit, wann diese in Preis und Kapazität die Festplatten, siehe Abbildung 14 ablösen.

Abbildung 15 zeigt eine SSD-Festplatte.

Abbildung 15.: moderne SSD-Festplatte

In der heutigen Zeit sind zum Austausch von Daten und Informationen zwischen einzelnen Computersystemen die USB-Sticks nicht mehr wegzudenken, die heute schon in Kapazitäten von mehreren Gbyte vordringen.

Abbildung 16.: USB-Stick 16 GByte

In dieser Aufzählung sind bewusst nur externe Speichermedien benannt, deren Liste sich natürlich auch noch fortsetzen lässt. Andere Ein- / Ausgabegeräte wie Bildschirm, Tastatur und Maus sind natürlich für den Computer sehr notwendig, sollen aber an dieser Stelle nicht weiter betrachtet werden.

1.5.4. Bussysteme

Bei den Betrachtungen der einzelnen Komponenten wird deutlich, dass ihre Anforderungen an die Geschwindigkeit der Datenübertragungen sehr unterschiedlich sind und sich noch weiter auseinander entwickeln werden. Dabei ist eine Struktur, wie sie Abbildung 9 zeigt nicht mehr zufriedenstellend.

Die Entwicklung ist weg gegangen von der Ein-BUS-Struckturtruktur. In Abbildung 17 sieht man die Struktur eines Pentium-Systems mit mehreren verschiedenen Bus-Arten, die sich in Funktion, Taktrate, Übertragungsbreite und damit in der Übertragungsrate stark unterscheiden:

- **ISA**-Bus (**I**ndustry **S**tandard **A**rchitecture),
- **PCI**-Bus (**P**eripheral **C**omponent **I**nterconnect),
- Cache-Bus,
- Lokaler Bus,
- Speicher-Bus
- **IDE**-Bus (**I**ntegrated **D**rive **E**lectronics),
- **SCSI**-Bus (**S**mall **C**omputer **S**ystem **I**nterface),
- **SATA**-Bus (**S**erial **A**dvanced **T**echnology **A**ttachment),
- **USB**-Bus (**U**niversal **S**erial **B**us),
- Firewire-Bus (IEEE 1394)

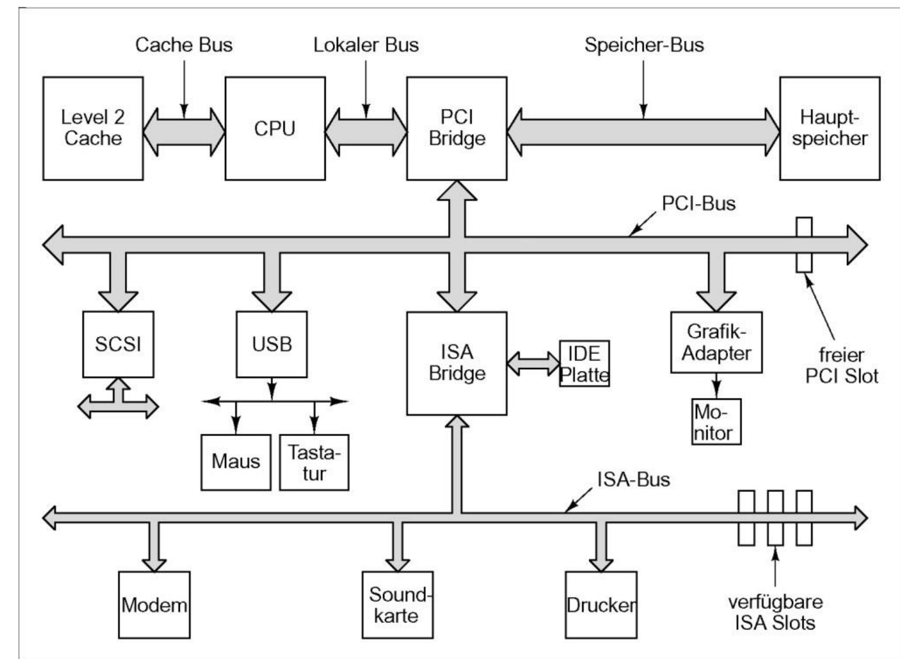
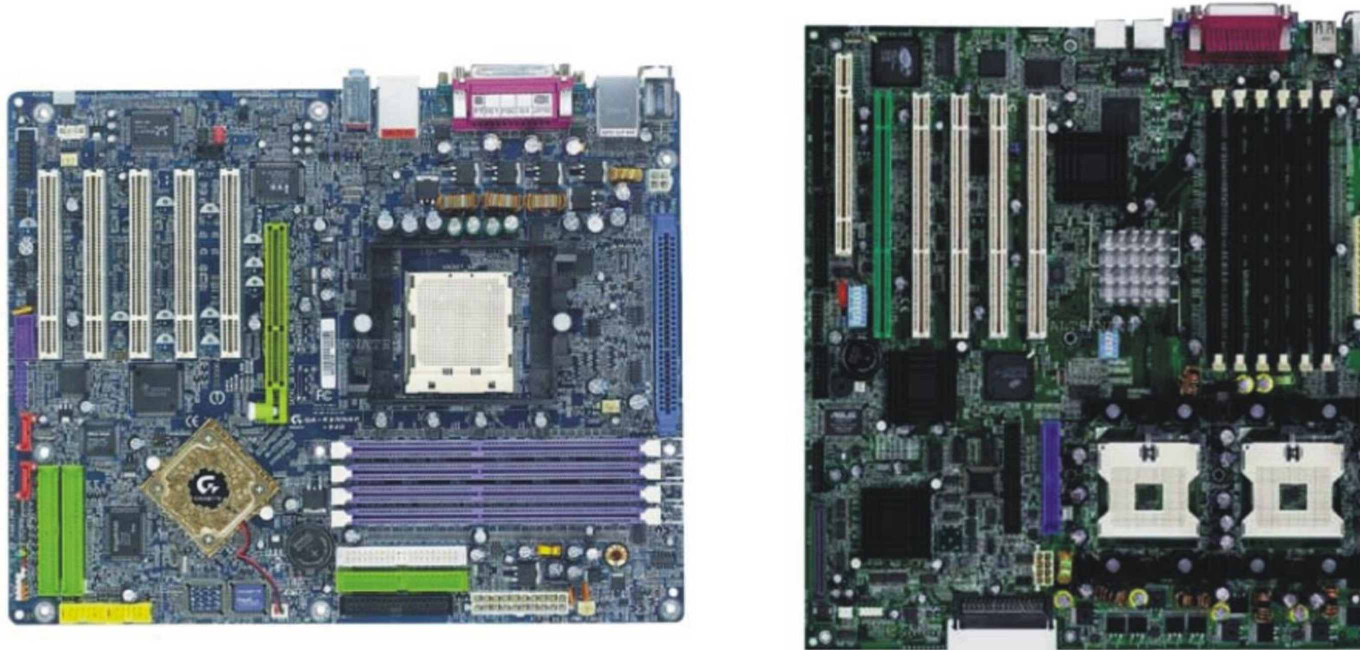


Abbildung 17.: typische Bus-Struktur eines Personal Computers

Aus Kosten- und Geschwindigkeitsgründen werden die meisten Baugruppen zusammen auf einer Platine untergebracht, den so genannten Mainboards.



In Abbildung 18 sind Mainboards für die beiden typischen Prozessoren, die in Personal Computern zum Einsatz kommen abgebildet. Links das Mainboard Gigabyte für AMD-64-Bit-Prozessoren und rechts das Mainboard Asus für 2 Intel-Xeon-Prozessoren. Damit sind jedoch Leistungen erreichbar, wie sie typischerweise in Servern gebraucht werden.

Abbildung 18.: Mainboards für den Personal Computer

1.6. Beispiele für preiswerter Linux- und Android-Rechner

Seit dem Jahr 2012 gibt es eine sehr preiswerte Alternative für Hobbybastler, die auf der Suche nach einem Linux bzw. Android-Rechner sind.

1.6.1. Raspberry Pi

Mit dem **Raspberry Pi** wird ein kreditkartengroßer Einplatinen-Computer, entwickelt von der ***Raspberry Pi Foundation*** angeboten. Als Basis dient das Einchipsystem BCM 2835, dessen Herzstück der 700-MHz Hauptprozessor ARM1176JZF-S ist. Als Arbeitsspeicher stehen 512 Mbyte RAM zur Verfügung. Für die Speicherung des Betriebssystems ist eine mindestens 2 Gbyte große SD-Karte vorgesehen.

Abbildung 19.: Raspberry Pi, Version B

Bautechnisch sind die Anschlüsse für die notwendigen peripheren Geräte im Vergleich zum Rechner sehr groß. Zur Stromversorgung von ca. 5 Watt kommt ein, meistens bei Smartphones verwendetes Netzteil mit microUSB Stecker zum Einsatz. Die integrierte Grafikkarte lässt die maximale Auflösung von Full HD mit 1920 x 1080 Bildpunkten über einen HDMI-Anschluss zu. Darüber wird auch die Audio-Ausgabe realisiert. Alternativ ist ein RCA Video-Out und eine 3,5 mm-Stereo-Buchse vorhanden.

Für den Anschluss einer Tastatur und der Maus gibt es zwei USB-Buchsen. Der Netzwerkanschluss erfolgt über eine RJ 45-Buchse.
Betriebssystemtechnisch sind verschiedene Linux-Derivate für den ARM-Prozessor bereits über das Internet ladbar. So stehen die Versionen GNU/Linux Debian, Fedora, Arch Linux und ein RISC OS zur Verfügung.

Preislich bewegt sich dieser vollständige Computer zwischen ca. 25 und 50 Euro und ist somit eine sehr attraktive Alternative für Privatanwender.

1.6.2. Smart PC Stick 2.0

Der Smart PC Stick 2.0 der Fima joy-it stellt die vollständige Hardware eines modernen Shartphones zur Verfügung.

Hardware:
- Prozessor: Cortex-A9 (2x 1,4 GHz)
- Arbeitsspeicher: 1024 MB
- Grafikkarte: Mali400MP4
- Netzwerkkarte: Wireless LAN 802.11 b/g/n
- Anschlüsse: Rückseite 2x USB 2.0, Front 1x HDMI
- Bluetooth integriert
- Festplatte 4 GB
- MicroSD Kartenslot mit bis zu 32 GB erweiterbar

Abbildung 20.: Smart PC Stick 2.0

Als Betriebssystem kommt Android 4.X, in der neusten Version 4.1.1 mit der Sprachunterstützung für Deutsch, Englisch, und andere EU Länder zum Einsatz.

Einsatzmöglichkeiten:

- E-book: Lesezeichen, Hintergrundmusik und Foto Anzeige
- Foto: JPG, BMP, GIF
- Video: AVI, RM, RMVB, MKV, MOV (Bis zu 1920-1080 Auflösung / HD Movie bis zu 1080P Wiedergabe)
- USB 2.0 Anschluss: 2x z.B. für drahtloses Tastatur/Maus-Set
- Mini USB Port: Für USB Stromversorgung
- HDMI Anschluss: Für Anschluss am TV
- Wireless LAN: 802.11 b/g/n
- Bluetooth: integriert
- Zubehör: Netzteil, HDMI-Kabel, USB-Kabel, Anleitung

Abbildung 21.: Anschlußmöglichkeiten des Smart PC Stick 2.0

1.7. Klassifizierung von Betriebssystemen

Die Klassifizierung von Betriebssystemen gibt Auskunft darüber, wie bestimmte Funktionen in dem Betriebssystemkern realisiert werden bzw. werden müssen. Oder anders ausgedrückt, sind für bestimmte Klassifikationsmerkmale bestimmte Funktionen, die der Betriebssystemkern realisiert, unbedingt notwendig.

Diese angesprochenen Funktionen liegen vor allem in dem Prozesssystem des Betriebssystemkerns bzw. des betrachteten Betriebssystems.

Jedoch müssen alle Funktionsgruppen des Betriebssystemkernes, unabhängig von der Klassifikation und dem Funktionsumfang, enthalten sein.

1.7.1. Klassifikation nach dem Anwendungsgebiet

1.7.1.1. Betriebssysteme für allgemeine Anwendungen

- Programmentwicklung,
- betriebswirtschaftliche Aufgaben,
- wissenschaftlich-technische Aufgaben,
- Multimediaanwendungen,
- **CAD** - **C**omputer **A**ided **D**esign u.s.w.

1.7.1.2. Echtzeitbetriebssysteme

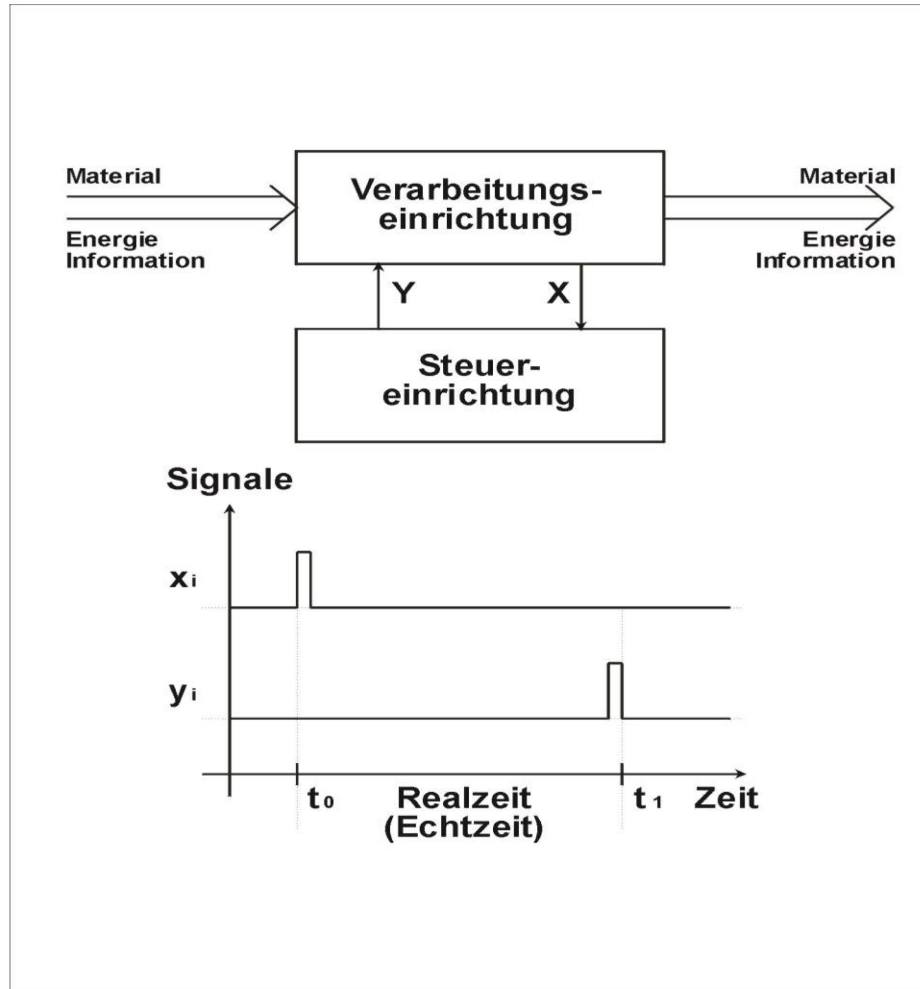


Abbildung 22.: Echtzeit

- Prozess-Steuerungen,
- Labor- und Geräteautomation,
- **CAM** - **C**omputer **A**ided **M**anufacturing

Wesentliches Kriterium für ein Echtzeitbetriebssystem ist, dass eine minimale (möglichst kleine) Reaktionszeit angebbar ist, nach der auf ein äußeres Ereignis maximal reagiert werden kann. In Abschnitt 9. Echtzeitbetriebssysteme wird auf diese Art von Betriebssystemen genauer eingegangen.

Prozess	**Antwortzeiten**
Maschinenreglung	1 … 10 ms
Prozessreglung	10 … 100 ms
Prozessführung	0,1 … 1 s
Prozessüberwachung / Auskunftssysteme	1 … 10 s

Tabelle 2.: Echtzeitbedingungen

1.7.2. Klassifikation nach der vorhandenen Anzahl von Prozessoren

Wie bereits eingeführt, hat John von Neumann einen Vorschlag für den Aufbau von Digitalrechnern vorgelegt, nach dessen Vorbild bisher fast alle Rechner aufgebaut sind. Von Neumann geht dabei von einer Verarbeitungseinheit aus, da es zur Zeit der Entwicklung seiner Idee noch nicht die Möglichkeiten gab, mehrere Verarbeitungseinheiten in einem Rechner zusammenzufassen.

In heutigen Systemen werden für fast alle Geräteansteuerungen eigene Prozessoren verwendet. So z.B. für

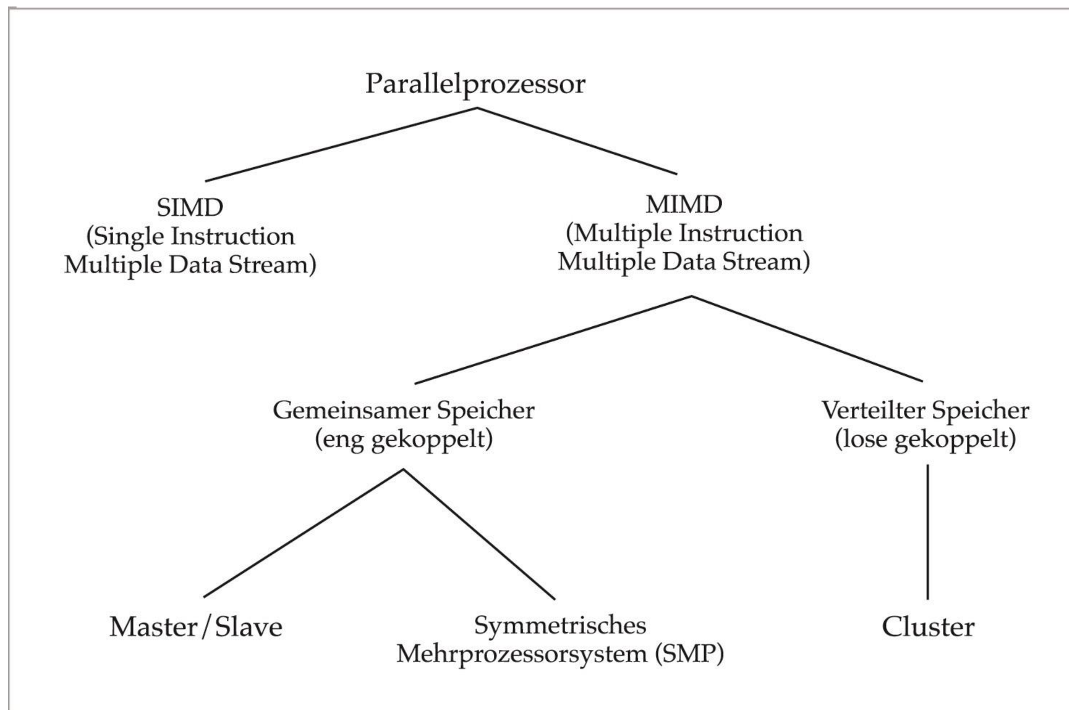
- Grafikprozessoren für CAD-Anwendungen,
- Prozessoren in den Ansteuerkarten für Festplatten, CD-ROM-Laufwerke usw.
- Prozessoren für den Netzzugang usw.

Bei der genannten Klassifizierungsart geht es jedoch nicht darum, wie viel Prozessoren allgemein in einem Rechner verwendet werden, sondern wie viel **Universalprozessoren** für die Verarbeitung der Daten zur Verfügung stehen. Somit ergibt sich folgende Unterscheidung:

1.7.2.1. Ein-Prozessor-Betriebssystem

Die meisten Rechner, die auf der von-Neumann-Architektur aufgebaut sind, verfügen über **nur** einen Universalprozessor. Aus diesem Grund unterstützen auch die meisten Betriebssysteme für diesen Anwendungsbereich nur einen Prozessor. Die Verwendung von Mehr-Core-Prozessoren ist erst in moderneren Betriebssystemen möglich.

1.7.2.2. Mehr-Prozessor-Betriebssystem



Für diese Klassifizierung der Betriebssysteme, entsprechend der Rechnerhardware, ist noch keine Aussage über die Kopplung der einzelnen Prozessoren getroffen worden. Auch gibt es keinen quantitativen Hinweis über die Anzahl der Prozessoren, nur, dass es mehr als ein Prozessor ist.

Mehrprozessorsysteme werden für spezielle Anwendungsgebiete geschaffen:

Abbildung 23.: verschiedene Multiprozessorstrukturen

Für die Realisierung der Betriebssysteme für die Mehrprozessorsysteme gibt es zwei Herangehensweisen:

1. Jedem Prozessor wird durch das Betriebssystem eine eigene Aufgabe zugeteilt. D.h., es können zu jedem Zeitpunkt nur so viel Aufgaben bearbeitet werden, wie Prozessoren zur Verfügung stehen. Somit entstehen Koordinierungsprobleme, wenn die Anzahl der Aufgaben nicht gleich der Anzahl verfügbarer Prozessoren ist.

2. Jede Aufgabe kann prinzipiell jedem Prozessor zugeordnet werden. D.h., die Verteilung der Aufgaben zu den Prozessoren ist nicht an die Bedingung gebunden, dass die Anzahl der Aufgaben gleich der Anzahl Prozessoren ist. Sind mehr Aufgaben zu bearbeiten, als Prozessoren vorhanden sind, so bearbeitet ein Prozessor mehrere Ausgaben **quasi-parallel**. Sind mehr Prozessoren als Aufgaben vorhanden, dann bearbeiten mehrere Prozessoren eine Aufgabe. Das Betriebssystem kann dabei seinerseits auch auf mehrere Prozessoren verteilt sein. Man spricht dann von **verteilten Betriebssystemen**.

1.7.3. Klassifikation nach der Anzahl parallel ablaufender Programme

Für einen Anwender, der mittels Betriebssystem seine Aufgaben bearbeitet haben möchte, ist von großem Interesse, wie viel Aufgaben kann er parallel bearbeiten bzw. bearbeiten lassen. Dabei spielt die Klassifikation nach 1.2.2. keine Rolle.

In dieser Klassifikation kommt der Begriff **Task** vor. Alternativ kann der deutsche Begriff **Prozess** Verwendung finden. Eine genauere Definition folgt später. Aus Anwendersicht kann an dieser Stelle auch der Begriff **Aufgabe** bzw. **Auftrag** verwendet werden.

Man unterscheidet:

1.7.3.1. Single-Task-Betriebssystem

Auch wenn für den Anwender die Anzahl vorhandener Prozessoren bei dieser Klassifikation keine Rolle spielt, sind typische **Single-Task-Betriebssysteme** auch gleichzeitig **Ein-Prozessor-Betriebssysteme**.

Kennzeichnend ist, dass der Anwender immer nur eine Anwendung starten kann. Erst wenn er diese beendet hat, kann die nächste Aufgabe bearbeitet werden.

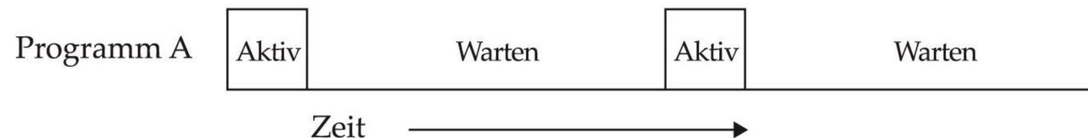


Abbildung 24.: Single-Task-Betrieb

1.7.3.2. Multi-Task-Betriebssystem

Bei **Multi-Task-Betriebssystemen** kann der Anwender zu jedem beliebigen Zeitpunkt mehrere Aufgaben durch das Betriebssystem bearbeiten lassen. Dabei wird im Allgemeinen nur eine Aufgabe direkt zu jedem Zeitpunkt **aktiv** bearbeitet.

Die anderen Aufgaben oder Aufträge können nach anfänglicher Eingabe von Parametern u.ä. durch das Betriebssystem selbstständig bearbeitet werden. Beziehungsweise der Anwender führt gerade keinen **Dialog** mit den Anwendungen.

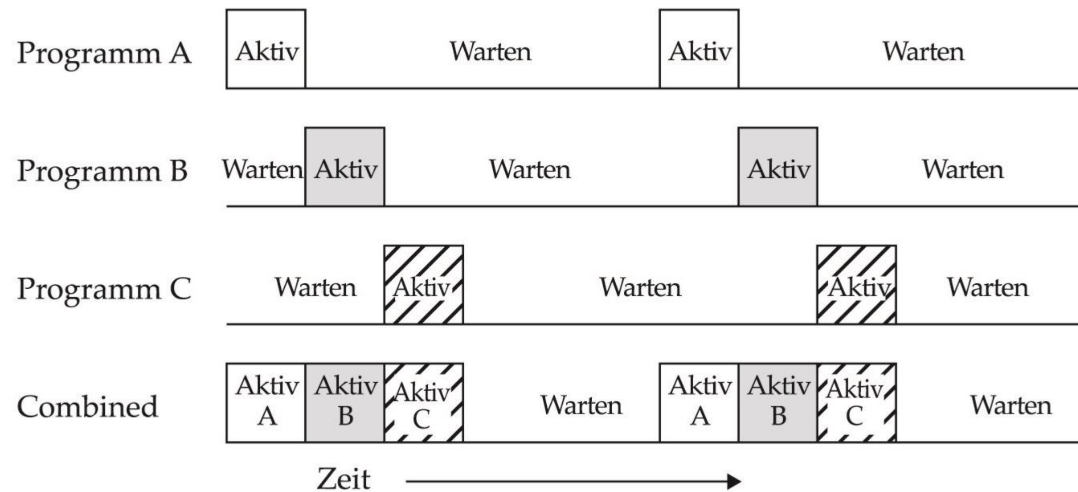


Abbildung 25.: Multi-Task-Betrieb

1.7.4. Klassifikation nach der Anzahl gleichzeitig aktiver Nutzer

Wenn man die geschichtliche Entwicklung der Rechentechnik betrachtet, waren die verfügbaren Rechner in den Anfängen recht unhandlich und teuer. Deshalb konnte man es sich nicht leisten, einen Rechner für einen Anwender anzuschaffen. Mit der Einführung des **P**ersonal **C**omputers durch die Firma **IBM 1984** änderte sich das schlagartig. Die Entwicklung war soweit fortgeschritten, dass dem einzelnen Mitarbeiter ein persönlich zugeordneter Rechner angeschafft werden konnte.

Abbildung 26.: IBM-PC

Mit der zunehmenden Vernetzung und Verflechtung der Rechner in einem Unternehmen und darüber hinaus weltweit, machte sich wieder ein umgekehrter Trend bemerkbar, dass moderne Betriebssysteme wieder die Nutzung durch mehrere Anwender zulassen.
Man unterscheidet:

1.7.4.1. Single-User-Betriebssystem

Entsprechend der Zielsetzung und des Einsatzgebietes dieser Art von Betriebssystemen bzw. denen der Rechner, kann zu einem beliebigen Zeitpunkt nur ein Anwender mit dem Betriebssystem arbeiten und Aufgaben lösen.

Dabei kann nicht ausgeschlossen werden, dass zu einem anderen Zeitpunkt ein anderer Anwender mit dem gleichen System arbeitet. **Single-User-Betriebssysteme** zeichnen sich durch einen geringen oder keinen Schutz des Systems vor dem Zugriff anderer Anwender oder Personen aus.

Der Benutzer zu einem Zeitpunkt hat Zugriff auf alle Ressourcen des Systems.

1.7.4.2. Multi-User-Betriebssystem

Arbeiten mehrere Anwender gleichzeitig an einem Rechner, so sind Leistungen durch das Betriebssystem zu erbringen, die in einem **Single-User-Betriebssystem** nicht oder nur in geringem Umfang vorhanden sein müssen:

1. Die Leistungen des Rechners sind allen Anwendern gerecht zur Verfügung zu stellen.
2. Der Anwender muss annehmen, dass er ganz allein an dem Rechner arbeitet.
3. Es sind besondere Schutzmechanismen zum Schutz der Daten der einzelnen Anwender notwendig. Dabei sind folgende Möglichkeiten getrennt zu realisieren:

    - Schutz **gegen den Rest der Welt**
    - Schutz untereinander.

2. Der Betriebssystemkern

2.1. Grundlegende Funktionen des Betriebssystemkerns

Definition 2 - Betriebssystemkern:

> **Unter einem ~ versteht man eine Sammlung von Programmen, die wohldefinierte Funktionen über eine definierte Schnittstelle zur Verfügung stellen. Sie greifen steuernd auf die Hardware des Rechnersystems zurück.**

Für alle Typen von Betriebssystemen hat der Betriebssystemkern gleiche Funktionen, die sich jedoch erheblich im Umfang der Realisierung der einzelnen Funktionen in Abhängigkeit von dem Einsatzgebiet des Betriebssystems unterscheiden.

Folgende vier Funktionen bzw. Funktionsgruppen werden unterschieden:

- Prozesssystem,
- E/A-System,
- Dateisystem und
- Speicherverwaltungssystem.

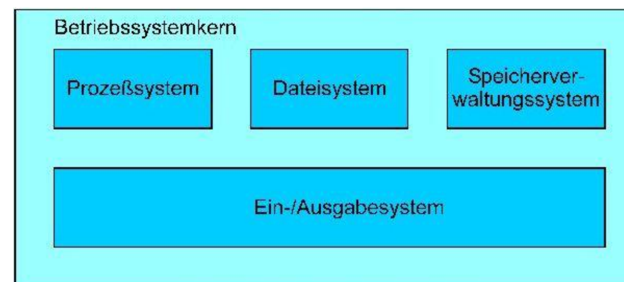


Abbildung 27.: Funktionen des Betriebssystemkerns

2.2. Das Prozesssystem

2.2.1. Programme, Prozeduren, Prozesse und Instanzen

Definition 3 - Programm:

Die Lösung einer Programmieraufgabe (=Algorithmus) wird in Form eines Programms realisiert. Teillösungen werden dabei als Prozeduren (Unterprogramme) formuliert, welche nach Beendigung ihrer Arbeit zum aufrufenden übergeordneten Programm zurückkehren.

Damit die Leistungen des Betriebssystemkerns problemlos in Anwenderlösungen eingebunden werden können, sind sie ebenfalls als Prozeduren realisiert.

Ein Programm (Prozedur, Unterprogramm) besteht aus

1. Befehlen (**Codebereich**, Textbereich)
2. Programmdaten (**Datenbereich**)

beide Komponenten sind **problemorientiert**.

Definition 4 - Prozess:

Wird ein Programm (Prozedur) unter der Kontrolle eines Betriebssystems (genauer gesagt unter der Kontrolle eines Betriebssystemkerns) ausgeführt, so wird dieser Ablauf als Prozess (engl. Task) bezeichnet.

Diese Betrachtungsweise macht es möglich, dass mehrere Programme gleichzeitig als Prozesse **parallel** auf einem **sequenziell arbeitenden Rechnersystem** (unabhängig von der realen Anzahl Prozessoren) ablaufen können.
Als Sonderfall gilt die Ausführung mehrerer Prozesse auf einem Prozessor.

(ACHTUNG: Die Funktionalität des Betriebssystemkerns bezüglich der Verwaltung von Prozessen ist bei der Ausführung mehrerer Prozesse **n** auf einem Prozessor **1** äquivalent der Verwaltung auf mehreren Prozessoren **m**. Dabei kann das Verhältnis **m : n** sowohl statisch als auch dynamisch änderbar sein.)

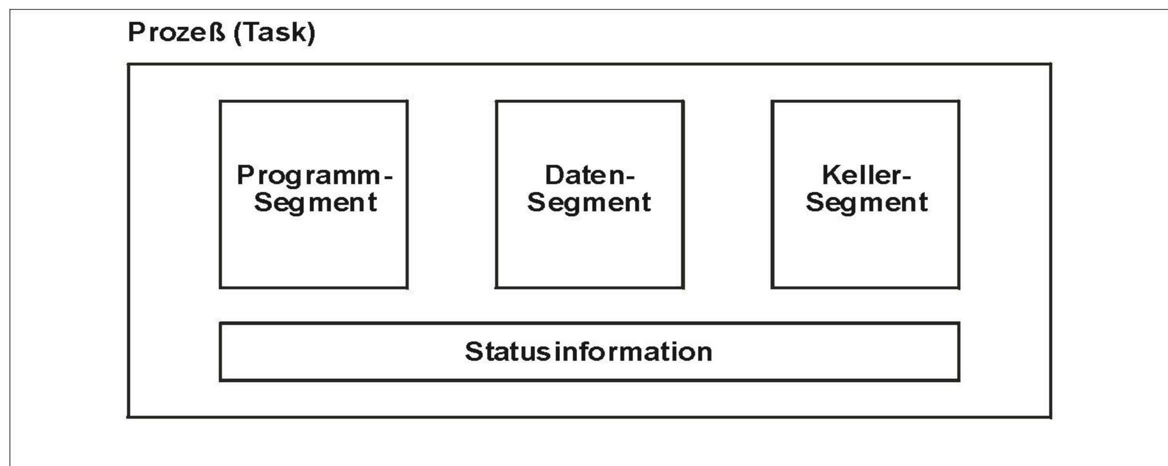


Abbildung 28.: Komponenten eines Prozesses

Bei der Ausführung von Prozessen entstehen Daten, die durch den Betriebssystemkern verwaltet werden. Diese werden **Statusinformationen** genannt und sind **systemabhängig**.

Als weitere Komponente wird beim Ablauf eines Programms ein Kellerspeicher (**Stack**) aufgebaut. Somit lässt sich ein Prozess modellhaft wie in Abbildung 28. darstellen.

In Tabelle 3 sind typische Informationen enthalten, die typischerweise in den Statusinformationen, in einer Prozesstabelle hinterlegt sind.

Prozessmanagement	Speichermanagement	Dateiverwaltung
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Kellersegment	Dateideskriptor
Kellerzeiger		Benutzer-ID
Prozesszustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozess-ID		
Elternprozess		
Prozessfamilie		
Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
CPU-Zeit des Kindes		
Zeitpunkt des nächsten Alarms		

Tabelle 3.: Prozesstabelle

Alle vier Komponenten, die bei der Ausführung eines Programms (einer Prozedur) beteiligt sind, werden als **Instanz** zusammengefasst.

Definition 5 - Instanz:

Eine Instanz umfasst das Tupel (P, D, K, S)

P: Programmsegment → problemorientiert
D: Datensegment → problemorientiert
K: Kellersegment → system- / problemorientiert
S: Statusinformationen → systemorientiert

Die physische Anordnung dieser Komponenten im Arbeitsspeicher eines Rechners kann in unterschiedlichen Betriebssystemen verschieden sein.

In Abbildung 29 ist die Problematik der Zuordnung von mehreren Prozessen zu einem Prozessor dargestellt.

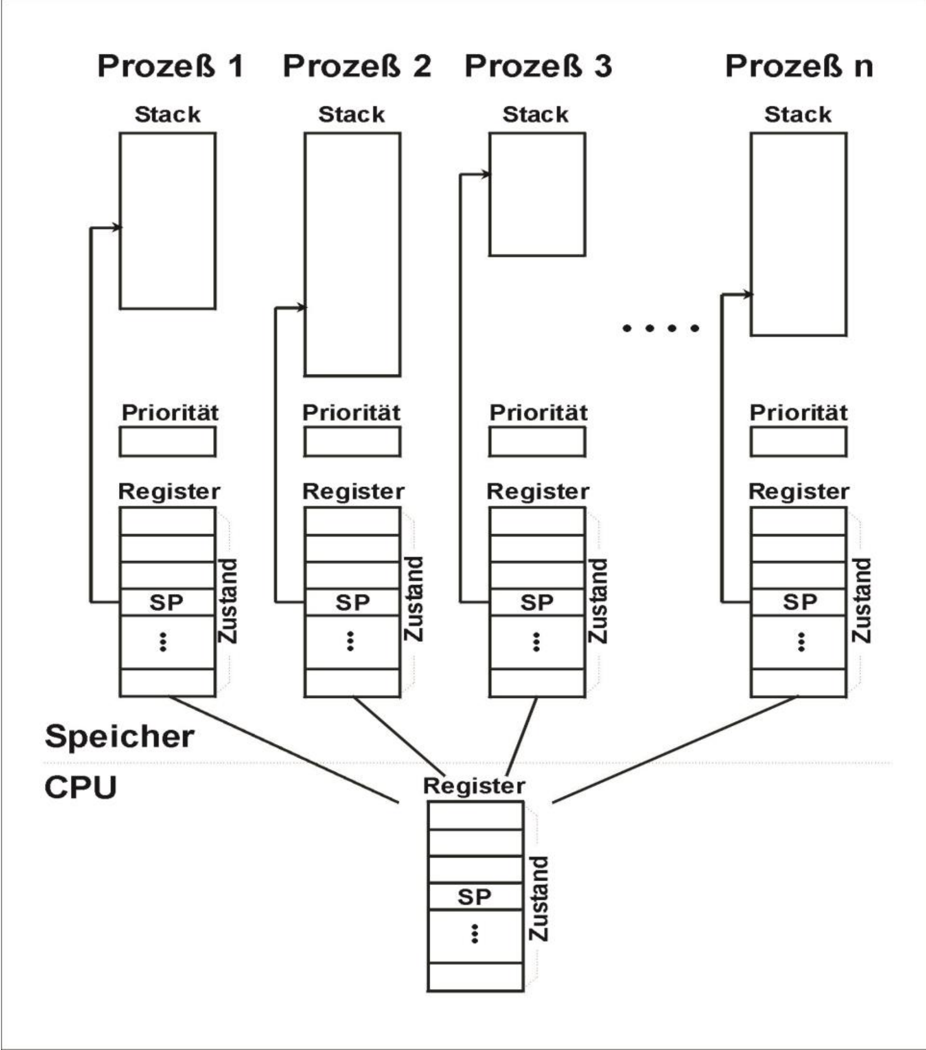


Abbildung 29.: Verwaltung von n Prozessen auf 1 Prozessor

2.2.2. Prozesserzeugung

Betriebssysteme benötigen ein Verfahren, um sicherstellen zu können, dass alle notwendigen Prozesse existieren. Prinzipiell gibt es dazu zwei Vorgehensweisen:

1. Alle Prozesse werden beim Systemstart erzeugt und
2. Es existieren Verfahren, die Prozesse im laufenden System je nach Bedarf zu erzeugen und zu beenden.

Dabei ist von Interesse, welche Ereignisse zur Erzeugung eines Prozesses dienen können:

1. Initialisierung des Systems
2. Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
3. Benutzeranfrage, einen neuen Prozess zu erzeugen
4. Initiierung einer Stapelverarbeitung (**Batch-Job**)

Beim Starten des Betriebssystems wird in der Regel eine Reihe von Prozessen bereits gestartet. Nach ihrer Arbeitsweise unterscheidet man zwischen:

1. Vordergrundprozesse – Prozesse, die im Dialog zum Anwender stehen und Aufgaben für diese erledigen.
2. Hintergrundprozesse – Prozesse für bestimmte Aufgaben, z.B. Verwalten der Warteschlange für den Drucker, die keine Dialogeingabe seitens eines Anwenders benötigen und die meiste Zeit nichts tun. Diese Prozesse werden auch als **Dämons** bezeichnet.

In interaktiven Systemen können Benutzer Programme durch Eingabe eines Kommandos oder das (Doppel-) Klicken eines **Icons** starten. Jede dieser Aktionen startet einen neuen Prozess und lässt das gewählte Programm darin ablaufen.

Technisch gesehen wird in jedem Fall ein neuer Prozess erzeugt, indem ein bestehender Prozess einen Systemaufruf zur Erzeugung eines neuen Prozesses ausführt, siehe Abbildung 30.

```
#define TRUE 1

while (TRUE) {                              /* Endlosschleife */
      type_prompt( )                        /* Prompt ausgeben */
      read_command(command,parameters);     /* Befehl einlesen */

      if (fork () != 0)                     /* Kindprozess erzeugen */
          /* Elternprozess */
          waitpid(–1, &status, 0)           /* Auf Ende von Kind warten */
      } else {
          /* Kindprozess */
          execve(command,parameters,0)      /* Befehl ausführen */
      }
}
```

Abbildung 30.: Erzeugung neuer Prozesse

In **Unix** existiert beispielsweise der Systemcall zur Erzeugung eines neuen Prozesses **fork**. Dieser Aufruf erzeugt eine exakte Kopie des aufrufenden Prozesses. Nach dem **fork** haben die beiden Prozesse, also Vater und Kind (oder Sohn), das gleiche Speicherabbild, die gleichen Umgebungsvariablen und die gleichen geöffneten Dateien. Üblicherweise führt der Kindprozess anschließend **execve** oder einen ähnlichen Systemcall aus, um sein Speicherabbild zu wechseln und ein neues Programm abzuarbeiten.

Im Gegensatz dazu wickelt unter **Windows** ein einziger Win32-Funktionsaufruf, nämlich **CreateProcess**, sowohl die Erzeugung des Prozesses als auch das Laden des richtigen Programms in den Prozess ab, wozu dieser Funktionsaufruf eine Reihe von Parametern erhält.

Sowohl in **Unix** als auch in **Windows** haben Vater und Kind nach der Prozesserzeugung je einen eigenen getrennten Adressraum. Wenn einer von beiden ein Speicherwort im Adressraum ändert, so ist diese Änderung für den anderen Prozess nicht sichtbar.

2.2.3. Prozessbeendigung

Jeder der einmal erzeugten Prozesse wird früher oder später terminiert. Dies wird üblicherweise aufgrund einer der folgenden Bedingungen erfolgen:

1. Normales Beenden (freiwillig)
2. Beenden aufgrund eines Fehlers (freiwillig)
3. Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
4. Beenden durch einen anderen Prozess (unfreiwillig)

Die meisten Prozesse terminieren, nachdem sie ihre Aufgabe erledigt haben. Dazu wird ein Systemcall ausgeführt, der dem Betriebssystem mitteilt, dass die Arbeit erledigt ist. Diese Systemcall lautet unter **Unix exit** und unter **Windows exitProcess**.

Es ist denkbar, dass durch die Terminierung eines Prozesses gleichzeitig alle von diesem Prozess erzeugten Kindprozesse sofort terminiert werden.

2.2.4. Prozesszustände

Obwohl jeder Prozess eine unabhängige Einheit darstellt, müssen Prozesse häufig mit anderen Prozessen kommunizieren. Folgendes Beispiel einer Kommandozeile unter **Unix** soll dies verdeutlichen:

who | wc -l

In diesem Beispiel werden sofort zwei Prozesse mit den Programmen **who** und **wc** gestartet. Das Programm **who** listet alle z.Z. am System angemeldeten Benutzer auf. Das Programm **wc** mit der Option **-l** zählt die Anzahl Zeilen in der Ausgabe von **who**. Beide Prozesse sind durch eine **Pipe** verbunden, die der Kommunikation zwischen den Prozessen dient. Es wird deutlich, dass der Prozess des Programms **wc** erst dann arbeiten kann, wenn der andere Prozess mit dem Programm **who** Daten geliefert hat. Dieser Prozess befindet sich in dem Prozesszustand blockiert. In Abschnitt 2.2.6. Interprozesskommunikation wird näher auf dieses Beispiel eingegangen.

Prinzipiell sind im einfachsten Fall die drei Zustände für einen Prozess möglich, die in Abbildung 31 dargestellt sind, wenn man davon ausgeht, dass nicht für jeden Prozess ein eigener Prozessor zur Verfügung steht.

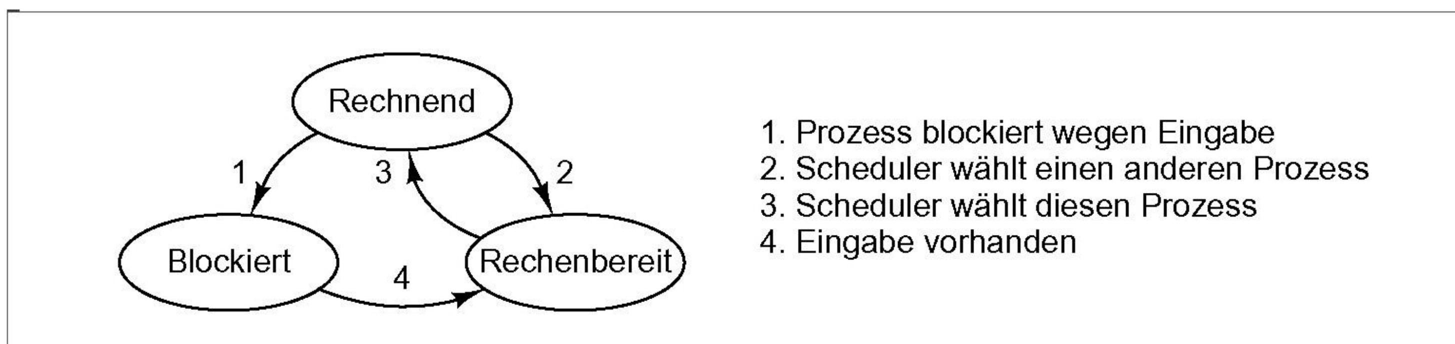


Abbildung 31.: Taskmodell mit 3 Zuständen

Folgende Zustände können dabei minimal unterschieden werden:

1. **rechnend** (die Befehle werden in diesem Moment auf der CPU ausgeführt)
2. **rechenbereit** (kurzzeitig gestoppt, um einen anderen Prozess rechnen zu lassen)
3. **blockiert** (nicht lauffähig bis ein bestimmtes externes Ereignis eintritt)

Sicherlich ist es möglich, die Art der Blockade von Prozessen weiter zu spezifizieren. So sind Prozesszustandsmodelle mit weit mehr als drei Zuständen denkbar. Als Beispiele können die Abbildung 32 und Abbildung 62 dienen.

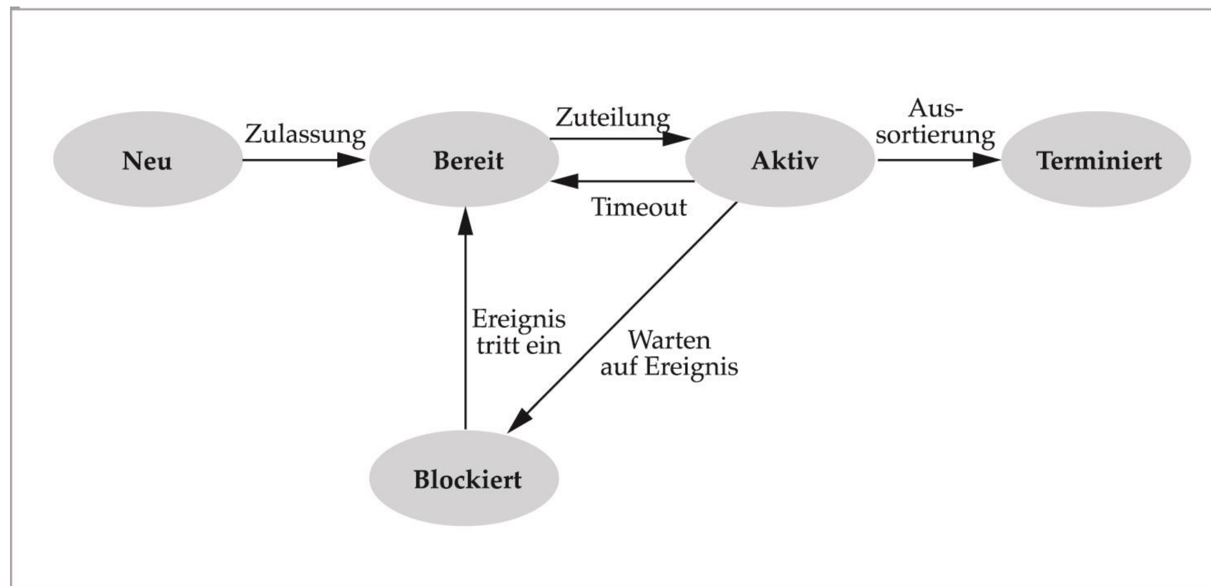


Abbildung 32.: Taskmodell mit 5 Zuständen

2.2.5. Threads

In traditionellen Betriebssystemen hat jeder Prozess einen eigenen Adressraum und einen einzigen Ausführungsweg. Jedoch gibt es viele Situationen, wo es wünschenswert ist, mehrere Ausführungswege in ein und demselben Adressraum quasiparallel ablaufen zu lassen, als ob es einzelne Prozesse wären, abgesehen von dem gleichen Adressraum.

Threads erweitern das Prozessmodel um die Möglichkeit, mehrere Ausführungswege, die sich in hohem Grade unabhängig voneinander verhalten, in derselben Prozessumgebung ablaufen zu lassen. Abbildung 33 dokumentiert die Unterschiede zwischen dem klassischen Prozess und der Aufteilung in mehrere Threads.

Ganz klar ist bei dieser Idee, dass Prozesse Parallelarbeit in verschiedenen Adressräumen während Threads Parallelarbeit im gleichen Adressraum durchführen.

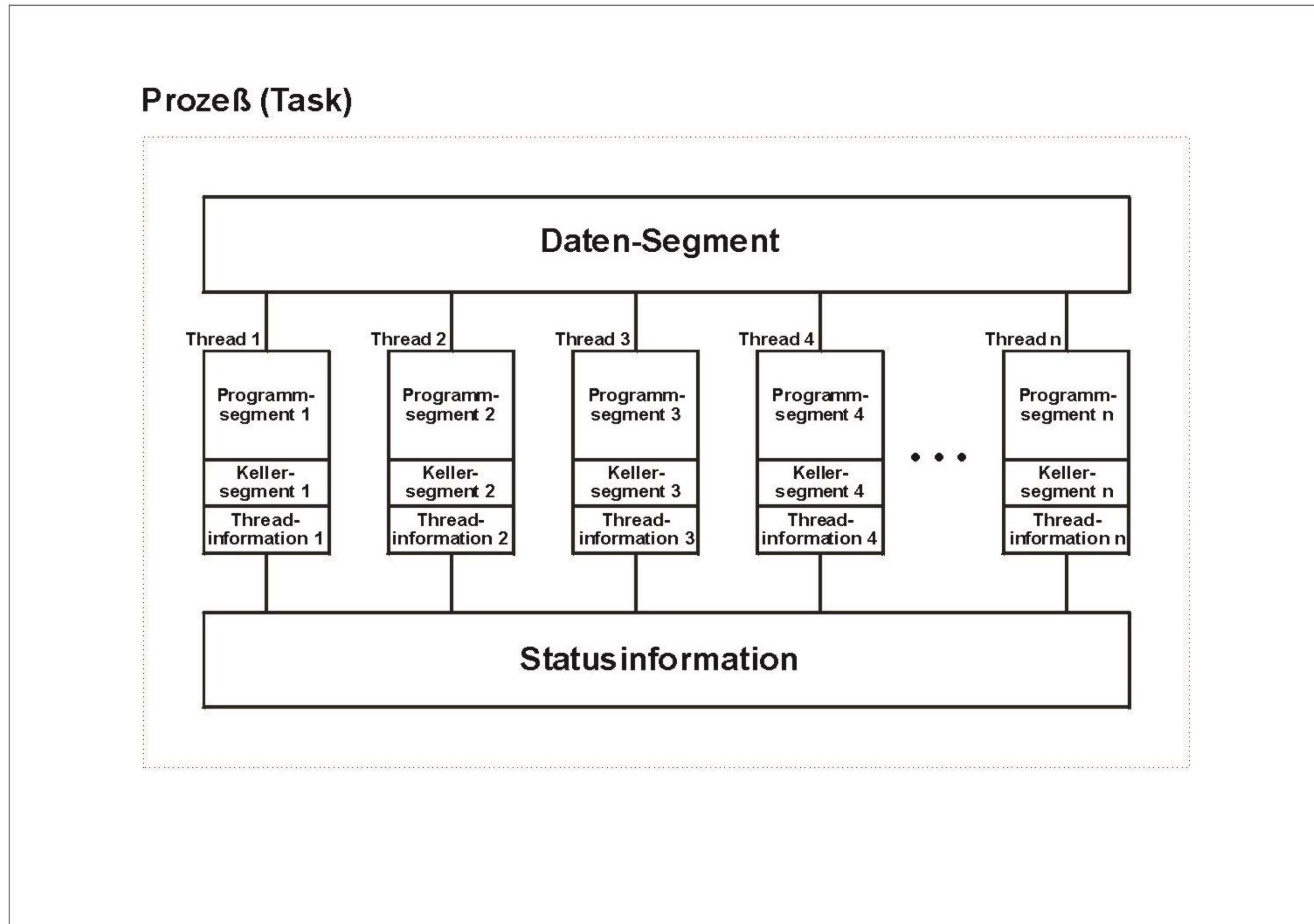


Abbildung 33.: Beziehung zwischen Prozess und Thread

2.2.6. Interprozesskommunikation

Prozesse müssen ständig mit anderen Prozessen kommunizieren. In einer Shell-Pipe, unter Unix zum Beispiel muss die Ausgabe des ersten Prozesses an den zweiten Prozess weitergereicht werden. Folgendes Beispiel aus Abschnitt 2.2.4. Prozesszustände, auf das im Abschnitt 5.7.3. Kommandoverkettung mit der Pipe näher eingegangen werden soll, demonstriert diesen Sachverhalt:

who | wc -l

In diesem Beispiel wird deutlich, dass dabei sowohl eine **Datenkommunikation** als auch eine **Prozesssynchronisation** stattfindet.

Das who-Kommando listet alle momentan am System angemeldeten Benutzer auf und gibt sie auf der Standardausgabe aus. Es werden damit Daten **erzeugt**. Im Kommando wc mit der Option -l werden alle Daten auf der Standardeingabe geprüft und die Anzahl Zeilen gezählt und am Ende ausgegeben. Durch die Pipe werden die erzeugten Daten von Prozess 1 (who) im Prozess 2 (wc –l) **verbraucht**. Es handelt sich um ein so genanntes **Erzeuger-Verbraucher-Problem**.

Gleichzeitig findet auch eine Synchronisation zwischen den Prozessen statt. Der Prozess 1 (who) kann nur endlich viele Daten in die Pipe schicken, wenn man sich diese als Puffer im Hauptspeicher vorstellt. Ist der Puffer voll, wird der Prozess 1 angehalten, man sagt, er **schläft**. Andererseits kann der Prozess 2 (wc -l) ohne Daten seine Aufgaben nicht erledigen. Ist dies der Fall, dann schläft er ebenfalls.

Das Beispiel führt nur dann zum Ziel, wenn alle Daten, die Prozess 1 erzeugt, durch Prozess 2 verbraucht werden. Ist Prozess 1 fertig, sendet er ein **Signal** (**EOF** – **E**nd **o**f **F**ile) an Prozess 2 und beendet sich. Nachdem Prozess 2 dieses Signal empfangen hat, ist dies für ihn das Zeichen, die ermittelte Zeilenanzahl auszugeben und sich dann ebenfalls zu beenden.

2.2.6.1. Signale

Eine Möglichkeit der Prozesssynchronisation stellen Signale dar. Ein Signal ist ein asynchrones Ereignis und bewirkt eine Unterbrechung auf der Prozessebene. Signale können entweder von außen durch den Benutzer an den Terminal oder durch das Auftreten von Programmfehlern (Adressfehler, Division durch Null usw.) erzeugt oder durch externe Unterbrechung hervorgerufen werden. Unter Unix kann auch ein anderer Prozess mittels Systemcall kill (SIGNAL, PID) ein Signal senden. In Tabelle 4 sind die Signale, wie sie in Unix definiert sind, aufgelistet.

Signalname	Signalnummer	Bedeutung
SIGHUP	1	Abbruch einer Terminalleitung (analog zum Aufhängen beim Telefon
SIGINT	2	Unterbrechung von dem Terminal
SIGQUIT	3	Abbruch vom Terminal
SIGILL	4	Ausführung eines ungültigen Befehls
SIGTRAP	5	Trace trap Unterbrechung (Einzelschrittausführung)
SIGFPE	8	Ausnahmesituation bei einer Gleitkommaoperation
SIGKILL	9	Kill-Signal (kill -9 PID)
SIGBUS	10	Fehler auf dem Systembus
SIGSEGV	11	Speicherzugriff mit unerlaubtem Segmentzugriff
SIGSYS	12	Ungültiges Argument beim Systemaufruf
SIGPIPE	13	Es wurde in eine Pipe geschrieben, von der nicht gelesen wird
SIGALARM	14	Ein Zeitintervall ist abgelaufen
SIGTERM	15	Signal zur Programmbeendigung
SIGCLD	18	Signalisiert die Beendigung des Sohnprozesses
SIGPWR	19	Signalisiert einen Spannungsausfall

Tabelle 4.: Signale in Unix

2.2.6.2. Events

Events sind Ereignisse, auf die Prozesse warten und damit aus dem schlafenden Zustand zu erwachen. Prozesse können andererseits Events schicken um genau diesen Effekt bei anderen Prozessen zu erreichen. Z.B. werden im Unix Events durch Signale abgebildet. Das Kommando

```
kill -9 %1
```

sendet das Signal 9 (SIGKILL) an den Prozess %1. Dieser kann das Signal nicht ignorieren und muss sich beenden.

2.2.6.3. Semaphore

Semaphore sind typische Elemente von Echtzeitsystemen zur Steuerung des Systemverhaltens. Sie sind systemweit bekannte Objekte, die zur Synchronisation der im System vorhandenen Prozesse verwendet werden. Synchronisation bedeutet in diesem Zusammenhang:

1. dass ein Prozess auf Daten eines anderen Prozesses wartet,
2. dass sich zwei Prozesse ein Betriebsmittel teilen müssen oder
3. dass eine Ereignissteuerung im Gegensatz zum Polling realisiert werden soll.

Der Niederländer **E.W.Dijkstra** schlug 1965 vor, eine ganzzahlige Variable einzuführen, die er **Semaphore** nannte. Ein Semaphor könnte den Wert 0 besitzen, wenn kein Event vorliegt. Mit irgendeinem positiven Wert wird die Anzahl der Events, von verschiedenen Prozessen kommend angezeigt. Zur Bearbeitung sollten zwei Operationen dienen, **down** und **up**. Die **down-Operation** eines Semaphors prüft, ob der Wert größer 0 ist. Falls dem so ist, erniedrigt sie den Wert um eins (z.B. um ein Event zu bearbeiten) und macht einfach weiter. Falls der Wert 0 ist, wird der Prozess sofort schlafen gelegt, ohne **down** vollständig auszuführen.

Die **up-Operation** erhöht den Wert, der von dem Semaphor adressiert wird, um eins. Falls ein oder mehrere Prozesse wegen eines Semaphors schlafen sollten, unfähig eine frühere down-Operation abzuschließen, wird vom System per Zufall ein Prozess gewählt, der sein down vervollständigen kann. Somit bleibt zwar nach dem up an einem Semaphore, auf den schlafende Prozesse warten, der Wert des Semaphors immer noch auf dem Wert 0, aber es gibt einen Prozess weniger, der wegen des Semaphors schläft.

2.2.6.4. Binäre Semaphoren - Mutex

Benötigt man die Möglichkeit eines Semaphors zu zählen nicht, wird manchmal eine vereinfachte Version eines Semaphors, **Mutex** genannt, verwendet. Mutexe dienen nur der Verwaltung des gegenseitigen Ausschlusses von irgendeiner gemeinsam genutzten Ressource.

Ein Mutex ist eine Variable, die zwei Zustände annehmen kann: nicht gesperrt oder gesperrt. Folglich wird nur 1 Bit benötigt.

2.2.6.5. Message

Eine Message ist eine Methode der Interprozesskommunikation und benutzt zwei Operationen send und receive:

- send(destination, &message);
- receive(source, &message);

2.3. Das Ein- / Ausgabe-System

Gerät	Übertragungsrate
Tastatur	10 Byte/s
Maus	100 Byte/s
56K Modem	7 KB/s
Telefonkanal	8 KB/s
Zwei ISDN-Leitungen	16 KB/s
Laserdrucker	100 KB/s
Scanner	400 KB/s
Klassisches Ethernet	1,25 MB/s
USB (Universal Serial Bus)	1,5 MB/s
Digitale Kamera	4 MB/s
IDE Platte	5 MB/s
40x CD-ROM	6 MB/s
Fast Ethernet	12,5 MB/s
ISA Bus	16,7 MB/s
EIDE (ATA-2) Platte	16,7 MB/s
FireWire (IEEE1394)	50 MB/s
XGA Monitor	60 MB/s
SONET OC-12 Netzwerk	78 MB/s
SCSI Ultra 2 Platte	80 MB/s
Gigabit Ethernet	125 MB/s
Ultrium Bandlaufwerk	320 MB/s
PCI Bus	528 MB/s
Sun GigaplaneXB Backplane	20 GB/s

Tabelle 5.: typische Ein- / Ausgabegeräte und die Übertragungsraten

Eine der Hauptaufgaben eines Betriebssystems ist es, alle Ein- / Ausgabegeräte eines Computers zu überwachen und zu steuern. Es müssen Kommandos an die Geräte weitergeleitet, Unterbrechungen (**Interrupts**) abgefangen und etwaige Fehler behandelt werden.
Außerdem sollte eine Schnittstelle zwischen den Geräten und dem Rest des Systems zur Verfügung stehen, die einfach und leicht zu benutzen ist. Soweit wie möglich ist, sollte die Schnittstelle für alle Geräte gleich sein (**Geräteunabhängigkeit**).

2.3.1. Ein- / Ausgabegeräte

Im Sinne der Betriebssystembetrachtungen ist die Schnittstelle der Geräte zur Software wichtig – die Kommandos, die die Hardware kennt, die Funktionen, die sie ausführt und mögliche Fehlermeldungen.

Eine Reihe von Geräten (wie Tastatur, Bildschirm, Drucker u.ä.) zeichnen sich dadurch aus, dass der Zugriff auf diese Geräte **zeichenweise sequenziell** (**zeichenorientierte Geräte**) erfolgt. D.h., einzelne Dateien bzw. Zeichen werden in einer bestimmten, wohldefinierten Reihenfolge von diesen Geräten **gelesen** bzw. auf diese Geräte **geschrieben**.

Funktionen zur zeichenorientierten Ein- und Ausgabe nutzen diesen Umstand aus und ermöglichen dem Anwendungsprogramm auf relativ einheitliche Weise, ein breites Spektrum an **Peripheriegeräten** zu bedienen.

Ein Teil dieser Geräte (z.B. Drucker) verarbeiten die Zeichen zur Ein- bzw. Ausgabe wesentlich langsamer, als sie der Rechner zur Verfügung stellt bzw. verarbeiten kann. In diesem Fall muss eine Synchronisation zwischen der CPU des Rechners und dem Gerät herbeigeführt werden. Aus diesem Grund kann zu jedem Zeitpunkt der Zustand des peripheren Gerätes durch den Rechner abgefragt werden.

Weitere Geräte (wie Disketten-, Festplatten- oder CD-ROM-Laufwerke) ermöglichen die direkte Adressierung der Daten. Diese Geräte verarbeiten die Daten nicht typischerweise zeichenorientiert, sondern **blockweise** (**blockorientierte Geräte**). Ein Zugriff auf die Daten ist nur in Einheiten von Blöcken bzw. Sektoren möglich, wobei die Sektoren entweder durch eine fortlaufende Nummer adressiert werden oder durch Angabe der physischen Speicherzelle auf dem Datenträger (Laufwerk, Kopfnummer, Zylindernummer, Sektornummer). Die physische Ansteuerung dieser Geräte wird in Funktionen zur blockorientierten Ein- und Ausgabe zusammengefasst.

2.3.2. Steuereinheiten

Abbildung 34.: Einsteckkarten

Die Ein- / Ausgabeeinheiten bestehen typischerweise aus einer **mechanischen** und einer **elektronischen Komponente**. Meistens ist es möglich, diese beiden Komponenten voneinander zu trennen, so dass man einen modularen und allgemeinen Entwurf erhält. Die elektronische Komponente wird als **Controller** oder **Adapter** bezeichnet. Bei Personal Computern ist es häufig als Einsteckkarte realisiert. Diese Struktur wird in Abbildung 9 dargestellt.

In Abbildung 34 sind zwei Beispiele für verschiedene Einsteckkarten dargestellt. Links eine **Grafikkarte Asus A9800 Pro** und rechts eine **Belkin Wireless Desktop Network Card**.

Jeder Controller besitzt einige Register, die für die Kommunikation mit dem Prozessor über den standardisierten Bus vorhanden sind. Damit können die Befehle an den Controller übermittelt werden. Weiterhin besitzen die meisten Controller zusätzlich Datenpuffer, die durch das Betriebssystem gelesen und geschrieben werden können.

Für den Zugriff auf diese Register existieren zwei Möglichkeiten:

1. Jedem Register wird eine sogenannte **Ein- / Ausgabe-Port-Nummer** zugewiesen, die ein 8-Bit- oder 16-Bit-Integer-Wert sein kann.
2. Jedem Register wird eine Hauptspeicheradresse zugewiesen, an der sich kein Hauptspeicher befindet. Dieses wird **Memory-Mapped Input Output** bezeichnet.

2.3.3. Direct Memory Access

Die Aufgabe für den Prozessor besteht darin, den Controller zu adressieren, um Daten mit diesen auszutauschen. Die CPU könnte die Daten vom Controller zeichenweise holen, aber das würde sehr viel Rechenzeit verschwenden, weshalb oft ein anderes Verfahren eingesetzt wird – **DMA** (Direct Memory Access).

Das Betriebssystem kann DMA nur dann verwenden, wenn die Hardware mindestens einen DMA-Controller zur Verfügung stellt. Dieser ist in den meisten Rechnern vorhanden und kann

- entweder direkt am Gerät an der Steuereinheit oder
- zentral auf dem Mainboard vorhanden sein.

Der DMA-Controller hat unabhängig von der CPU immer Zugriff auf den Systembus, wie in Abbildung 35 zu sehen ist.

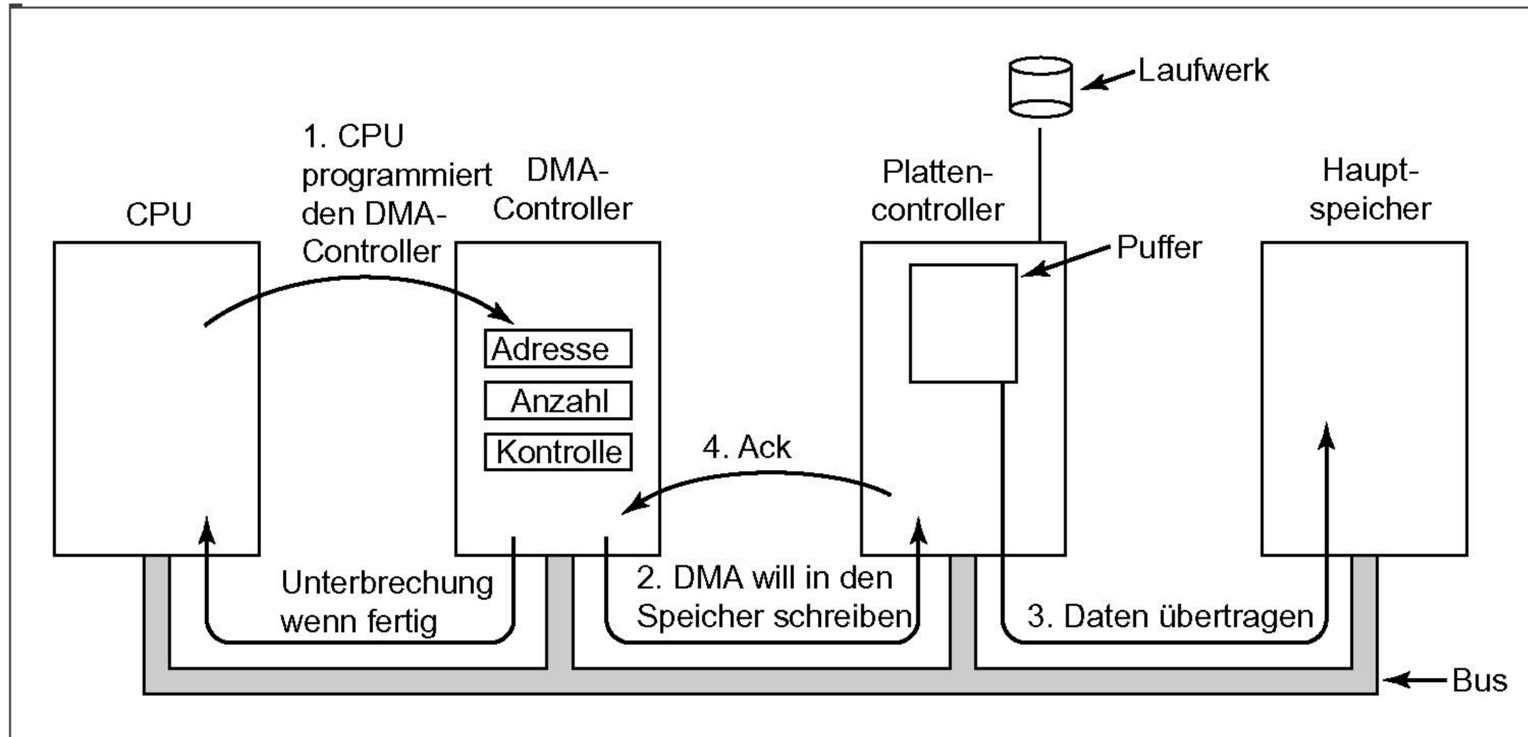


Abbildung 35.: DMA-Transfer

2.3.4. Interrupts

Aus den Angaben in Tabelle 5 ist ersichtlich, dass die einzelnen Geräte mit sehr unterschiedlicher Geschwindigkeit die Daten übertragen können. Es muss eine Synchronisation zwischen den Geräten und der CPU kommen. In der Theorie sind dazu prinzipiell zwei Verfahren möglich:

1. Polling – Die CPU überträgt ein Zeichen auf das oder von dem Gerät und wartet dann in einer Schleife bis das Gerät für den nächsten Transfer bereit ist. In dieser Zeit kann die CPU nichts anderes tun.

2. Interrupts – Die CPU beauftragt den Controller mit der Aufgabe der Übertragung und arbeitet normal weiter. Ist das Gerät fertig, unterbricht es die laufende Arbeit der CPU und diese kann jetzt den nächsten Transfer anstoßen.

Die prinzipielle Arbeitsweise bei Interrupts ist in Abbildung 36 zu sehen.

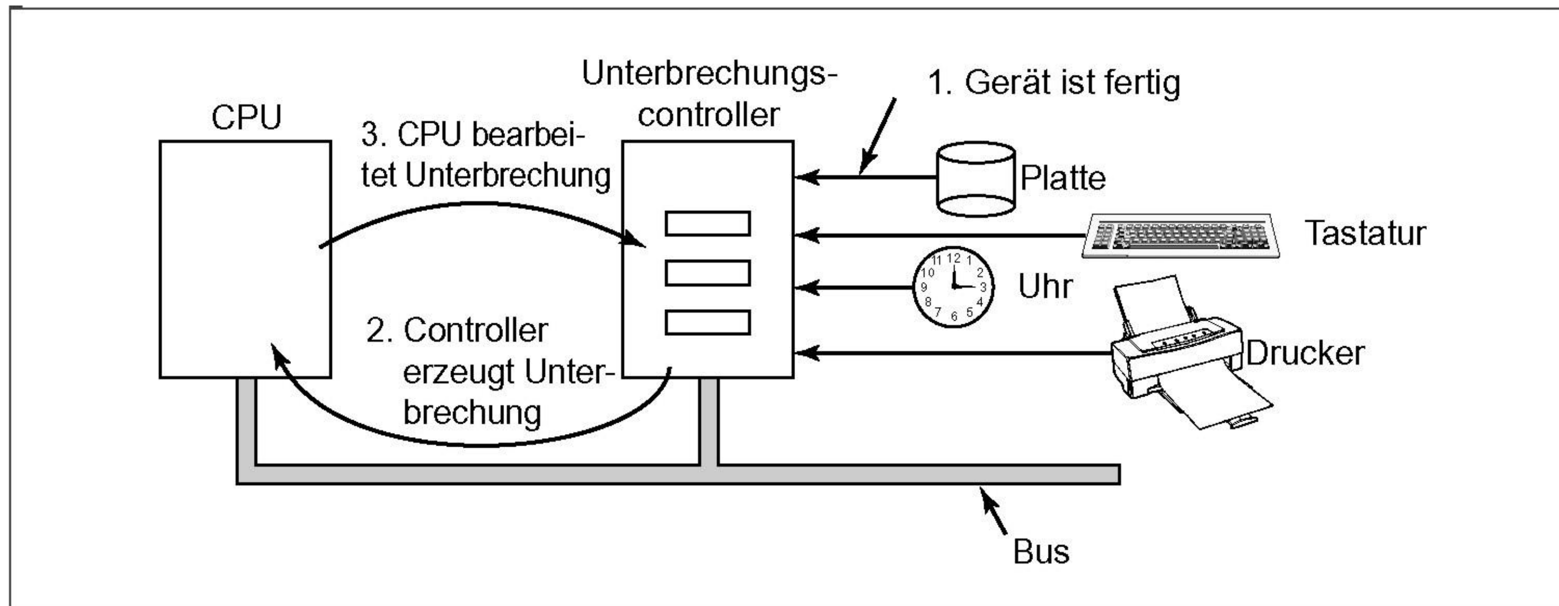


Abbildung 36.: Ablauf einer Unterbrechung (Interrupt)

2.4. Dateien und Dateisystem

Alle Anwendungsprogramme hantieren mit Mengen logisch zusammenhängender Daten, den Dateien. Das muss derart geschehen, dass dem Benutzer die Details über das **Wo** und **Wie** der Speicherung verborgen bleiben.

Die wahrscheinlich wichtigste Eigenschaft jeder Abstraktion ist die Art und Weise, wie die zu verwaltenden Objekte benannt werden.

2.4.1. Dateinamen

Die Speicherung dieser Dateien erfolgt meistens auf Geräte, den so genannten externen Speichermedien (extern, da sie außerhalb des Prozessors liegen). Und die Verwaltung, d.h. die Aufzeichnung, welche Dateien gespeichert sind, wo die Daten stehen, wann die Dateien erzeugt oder geändert wurden, welche Zugriffsrechte vergeben sind usw., übernimmt das **Dateisystem** (Filesystem) des Betriebssystemkerns.

> **Definition 6 - Datei:**
> **Eine ~ ist eine Menge an Daten (unstrukturiert oder strukturiert), die auf einem Speichermedium unter einem Namen abgelegt sind.**

Eine Anwendung (später als Prozess bezeichnet) erzeugt eine Datei, indem sie Daten unter einem wohlbestimmten Namen auf einem Speichermedium speichert. In der Regel existiert diese Datei auch nach Beenden der Anwendung und eine andere Anwendung kann ihrerseits auf diese Daten über den ihr bekannten Namen zugreifen.

Für den Anwender und damit die Anwendungen ist somit die Regel für die Vergabe von Dateinamen von großer Bedeutung und diese unterscheiden sich bei den verschiedenen Betriebssystemen nach folgenden Kriterien:

- Maximale Anzahl Zeichen,
- zugelassene Zeichen,
- Schreibweise (Unterscheidung zwischen Groß- und Kleinschreibung oder keine Unterscheidung),
- Anzahl Teile (maximal zwei), die zusammen den Dateinamen bilden (d.h. mit oder ohne Dateierweiterung) und
- Kennzeichnung für besondere Dateien (z.B. Systemdateien).

In Tabelle 6 sind einige typische Dateierweiterungen dargestellt, die ihren Ursprung im Betriebssystem MS-DOS haben, aber wegen ihrer weiten Verbreitung auch in anderen Betriebssystemen äquivalent verwendet werden, die keine Dateierweiterung verwenden. Mit einer solchen Standardisierung der Dateierweiterungen kann man diesen bewusst eine Anwendung zuordnen, mit der diese Art von Dateien immer bearbeitet werden kann.

Erweiterung	Bedeutung
Name.bak	Backup-Datei
Name.c	C-Quelltextdatei
Name.gif	Compuserve Graphical Interchange File Format
Name.hlp	Hilfedatei
Name.html	Hypertext-Markup-Language-Datei für das WWW
Name.jpg	Bilddatei nach dem JPEG-Standard
Name.mp3	Musik nach MPEG Layer 3 kodiert
Name.mpg	Film nach MPEG-Standard kodiert
Name.o	Objektdatei (übersetzt, aber nicht gebunden)
Name.pdf	Portable-Document-Format-Datei
Name.ps	PostScript-Datei
Name.tex	Eingabedatei für TeX
Name.txt	Allgemeine Textdatei
Name.zip	Komprimiertes Archiv

Tabelle 6.: typische Dateierweiterungen

2.4.2. Dateizugriff

Ein weiteres Kriterium ist die Art und Weise, wie ein Betriebssystem die einzelnen Dateien auf dem Speichermedium strukturiert ablegt. Dabei sind folgende Forderungen zu erfüllen:

- Die Dateigröße sollte nicht begrenzt sein (nur die physikalische Grenze des Speichermediums darf begrenzend wirken.
- Die einzelnen Bereiche der Datei sollten möglichst schnell auffindbar und für die Anwendung in den Hauptspeicher ladbar sein.

Auch in dieser Strukturierung unterscheiden sich die einzelnen Betriebssysteme wesentlich.

Sehr stark von der gewählten Art der Strukturierung der Dateien sind die Möglichkeiten des Zugriffs auf die Dateien abhängig. Bei bestimmter Strukturierung kann nur **sequenziell** zugegriffen werden. Eine Anwendung kann alle Datei in ihrer Reihenfolge als Bytes oder Records nacheinander beim Anfang beginnend einlesen. Überspringen oder Zugriffe außerhalb der Reihenfolge sind nicht möglich. Diese Strukturierung ist dann sehr komfortabel, wenn auch die Strukturierung des Speichermediums sequenziell ist, z.B. Magnetbänder u.s.w.

Voraussetzung für viele Anwendungen ist jedoch ein **wahlfreier Zugriff** auf die Bytes oder Records innerhalb der Datei, Random-Access-Dateien. Dabei treten die o.g. Forderungen sehr stark in den Vordergrund. Welche Möglichkeiten dabei entwickelt wurden, wird später beleuchtet.

2.4.3. Dateiattribute

Jede Datei hat einen Namen und ihre Daten. Darüber hinaus ist er ratsam, noch weitere Informationen über eine Datei abzuspeichern, z.B. Erstellungsdatum, Eigentümer oder Länge der Datei, um nur wenige zu nennen. Diese Zusatzinformationen werden auch **Dateiattribute** bezeichnet.

In Tabelle 7 sind mögliche Dateiattribute mit ihrer Bedeutung zusammengefasst, die denkbar sind. Die einzelnen Betriebssysteme benutzen jeweils immer nur einen Teil dieser Attribute.

Attribut	Bedeutung
Schutzinfos	Wer kann wie auf die Datei zugreifen
Passwort	Passwort für den Zugriff auf die Datei
Ersteller	ID des Erzeugers der Datei
Eigentümer	Aktueller Eigentümer
Read-only Flag	0: Lesen/Schreiben; 1: nur Lesen
Hidden Flag	0: normal; 1: in Listen nicht sichtbar
System Flag	0: normale Datei; 1: Systemdatei
Archive Flag	0: wurde gesichert; 1: muss noch gesichert werden
ASCII/binary Flag	0: ASCII-Datei; 1: Binärdatei
Random access Flag	0: sequentielle Datei; 1: wahlfreier Zugriff
Temporary Flag	0: normal; 1: Datei bei Prozessende löschen
Lock Flags	0: nicht gesperrt; nicht 0: Datei gesperrt
Record-Länge	Anzahl der Bytes in einem Record
Schlüsselposition	Offset des Schlüssels in einem Record
Schlüssellänge	Anzahl der Bytes in einem Schlüssel
Erstellungszeit	Datum und Zeit der Dateierstellung
Zeit des letzten Zugriffs	Datum und Zeit des letzten Zugriffs
Zeit der letzten Änderung	Datum und Zeit der letzten Dateiänderung
Aktuelle Größe	Anzahl der Bytes in einer Datei
Maximale Größe	Anzahl der Bytes für maximale Größe einer Datei

Tabelle 7.: mögliche Dateiattribute

2.4.4. Dateioperationen

Im Sinne des objektorientierten Paradigmas sind Dateien Objekten mit den dazu definierten Methoden, die im Falle von Dateien Dateioperationen genannt werden. Die folgenden Dateioperationen (Systemaufrufe) sind denkbar, aber stark mit der Dateistruktur verbunden und von dieser abhängig:

Create Die Datei wird ohne Daten erzeugt. Der Sinn und Zweck dieses Aufrufes besteht darin, für die kommenden Daten den Speicherraum vorzubereiten. Gleichzeitung werden die Dateiattribute auf Defaultwerte gesetzt.

Delete Wird die Datei nicht länger benötigt, wird sie gelöscht, um den von ihr belegten Platz auf dem Speichermedium wieder freizugeben.

Open Vor der Nutzung einer Datei muss die Anwendung diese öffnen. Damit verbunden werden durch das Betriebssystem die Dateiattribute in den Hauptspeicher geladen um bei folgenden Zugriffen auf die Datei performanter zugreifen zu können.

Close Sind alle Zugriffe auf die Datei beendet, werden die Dateiattribute nicht mehr länger benötigt und können gelöscht werden, um Hauptspeicher freizugeben. Änderungen in den Dateiattributen und der Datei, z.B. der letzte Block werden vorher auf das Speichermedium geschrieben.

Read Die Daten werden aus der Datei gelesen. Gewöhnlich werden die Daten von der aktuellen Position beginnend gelesen. Dazu muss dieser Operation angegeben werden, wie viele Daten (Bytes oder Records) gelesen werden sollen und in welchen Pufferbereich im Hauptspeicher die Daten abgelegt werden sollen.

Write Die Daten werden in die Datei geschrieben. Gewöhnlich an der aktuellen Position. Ist die aktuelle Position am Dateiende, so erhöht sich die Dateigröße. Ansonsten werden die Daten in die Datei geschrieben, wobei andere Daten an dieser Stelle überschrieben werden und verloren gehen.

Append Eingeschränkte Write-Operation. Es wird immer am Dateiende geschrieben.

Seek	Für Dateien mit wahlfreiem Zugriff wird diese Operation benötigt um die aktuelle Position für Read oder Write definiert zu verändern.
Get Attributes	Die aktuelle Belegung der Dateiattribute wird erneut in den Hauptspeicher gelesen.
Set Attributes	Die aktuelle Belegung der Dateiattribute im Hauptspeicher wird auf das Speichermedium geschrieben.
Rename	Der Dateiname wird verändert, ohne Datei die Daten zu kopieren oder sonst wie zu verändern.

2.4.5. Verzeichnisse

Damit der ständig steigenden Speicherkapazität der Speichermedien nicht nur die mögliche Größe einer Datei ständig anwächst, sondern auch die Anzahl von Dateien pro Speichermedium, bedarf es einer gewissen Ordnung. Diese wird durch Verzeichnisse oder Ordner erreicht. Die einfachste Form der Organisation sieht pro Speichermedium nur ein Verzeichnis vor, in dem alle Dateien des Datenträgers mit ihren Dateinamen und –attributen verzeichnet sind. Meistens ist in diesen Fällen die Anzahl Einträge begrenzt.

Eine weitaus flexiblere Organisation liegt bei einer Baumstruktur der Verzeichnisse vor. Dabei kann organisiert werden, dass pro Verzeichnis beliebig viele Einträge vorgenommen werden können und es auch beliebig viele Verzeichnisse auf dem Speichermedium geben kann. Da es bezüglich der Lage der Verzeichnisse zueinander dabei eine strenge Ordnung gibt spricht man in diesem Fall von einem hierarchischen Verzeichnissystem bzw. einem hierarchischen Dateisystem. In Abbildung 55 ist ein Beispiel für eine solche Baumstruktur dargestellt. Im Abschnitt 5.4. Logisches Dateisystem werden die Operationen auf dieser Struktur am Beispiel von Unix dargestellt.

2.4.6. Realisierung von Dateien

Der wichtigste Aspekt bei der Realisierung eines Dateisystems ist die Organisationsform einer Datei auf dem Speichermedium. D.h., welche Plattenblöcke werden durch eine Datei belegt. Dabei sind die Forderungen aus Abschnitt 2.4.2. Dateizugriff in besonderer Weise zu beachten.

Die wohl einfachste Form der Abspeicherung einer Datei auf einem Speichermedium ist die zusammenhängende Belegung, wie sie in Abbildung 37 dargestellt wird. Dabei ist als Beispiel die Plattenbelegung von sieben Dateien zu sehen (a). In (b) sind die Dateien D und F gelöscht worden und hinterlassen freie Bereiche auf dem Speichermedium.

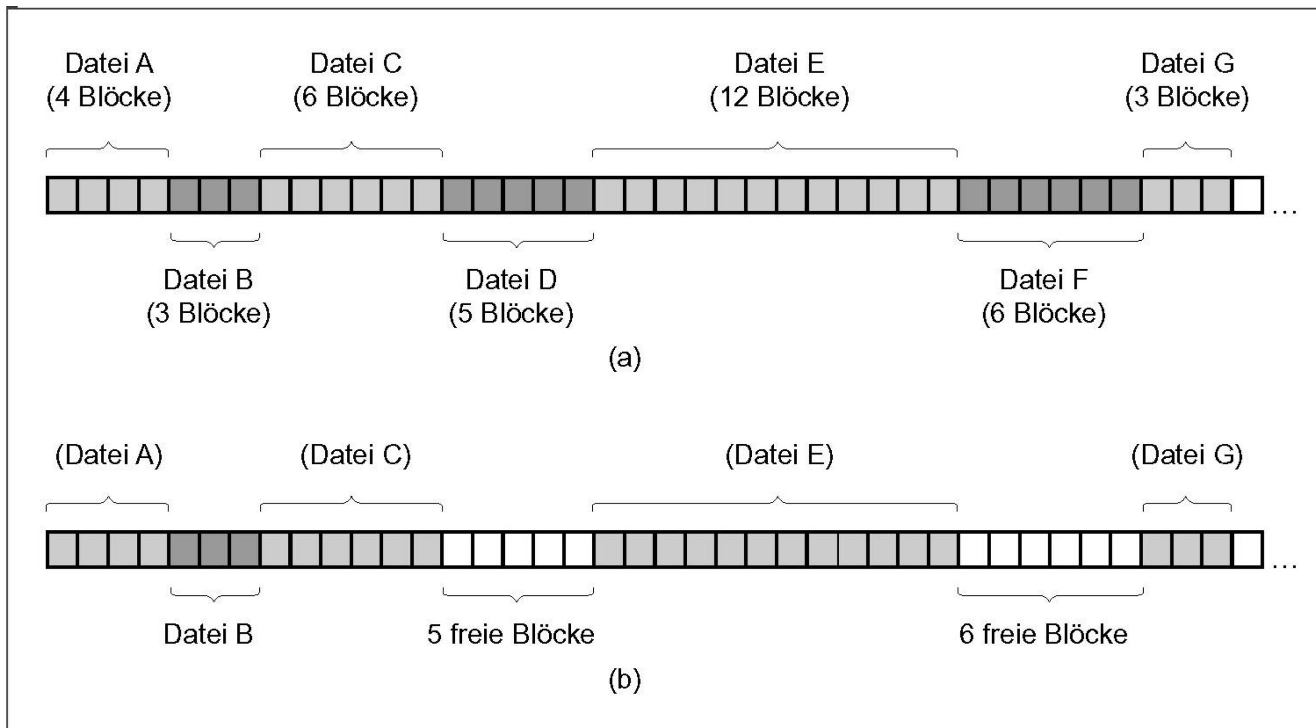


Abbildung 37.: Zusammenhängende Belegung

Die Problematik der Arbeitsweise liegt bei dieser Speicherform auf der Hand. Die Verwaltung der freien Bereiche und deren Wiederverwendung für neue Dateien ist entweder durch Verschieben der noch auf dem Speichermedium befindlichen Dateien, hier Datei E und G, was in der Praxis undenkbar ist. Oder es müssen die freien Bereiche in einer Datenstruktur verwaltet werden, um neue Dateien an die passende Stelle zu platzieren, das wiederum voraussetzt, dass deren Dateigröße vorher bekannt sein muss.

Eine weitere Möglichkeit stellt die Belegung durch verkettete Listen dar. Dabei wird im ersten Fall in jedem Plattenblock an dessen Beginn ein Verweis auf den folgenden Block abgelegt, wodurch sich der restliche Platz im Plattenblock um 3, besser 4 Byte verringert und damit keine Potenz von Zwei ist. Die meisten Anwendungen lesen und schreiben aber Daten in Potenzen von Zwei, was bei dieser Speicherbelegung zusätzlichen Aufwand mit sich bringt.

Im Beispiel in Abbildung 38 belegt die Datei A die Plattenblöcke 4, 7, 2, 10 und 12 und die Datei B die Plattenblöcke 6, 3, 11 und 14.

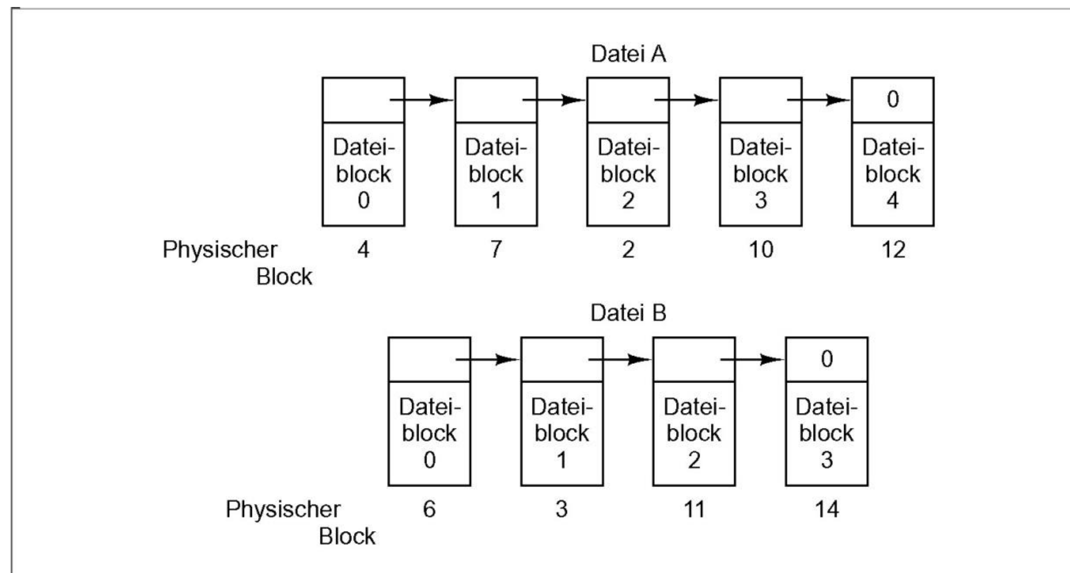


Abbildung 38.: verkettete Listen

Auch diese Form der Dateiabspeicherung auf einem Speichermedium ist nicht praktikabel und wurde weiterentwickelt. Die Form der verketteten Liste bleibt bestehen aber diese wird nicht zusammen mit den Daten in den Plattenblöcken abgelegt, sondern in einer Tabelle im Hauptspeicher bzw. im Verzeichnis. Diese Tabelle wird allgemein bekannt als **FAT** (**F**ile **A**llocation **T**able) bezeichnet. Diese Organisationsform wurde erstmals durch **Bill Gates** in MS-DOS eingebracht. Spricht man dabei von der **FAT**, so ist die ursprüngliche Form als **FAT-16** gemein. 16 bedeutet dabei die Anzahl Bits (16) zur Adressierung der einzelnen Plattenblöcke bzw. Cluster.

Damit ist relativ leicht die Begrenzung der Speichergröße eines Speichermediums durch die **FAT-16** auf 65536 Plattenblöcke mit jeweils 512 Byte = 32 MByte bzw. 256 MByte bei 4 KByte Clustergröße erkennbar. Für die Speichermedien der ersten Personal Computer war das sicherlich ausreichend, musste aber im Laufe der Zeit mehrfach verändert werden. Die heutigen Windows-Betriebssysteme verwenden als ein Dateisystem die **FAT-32**.

In Abbildung 39 sind die beiden Dateien A und B mit den gleichen Plattenblöcken enthalten wie in Abbildung 38.

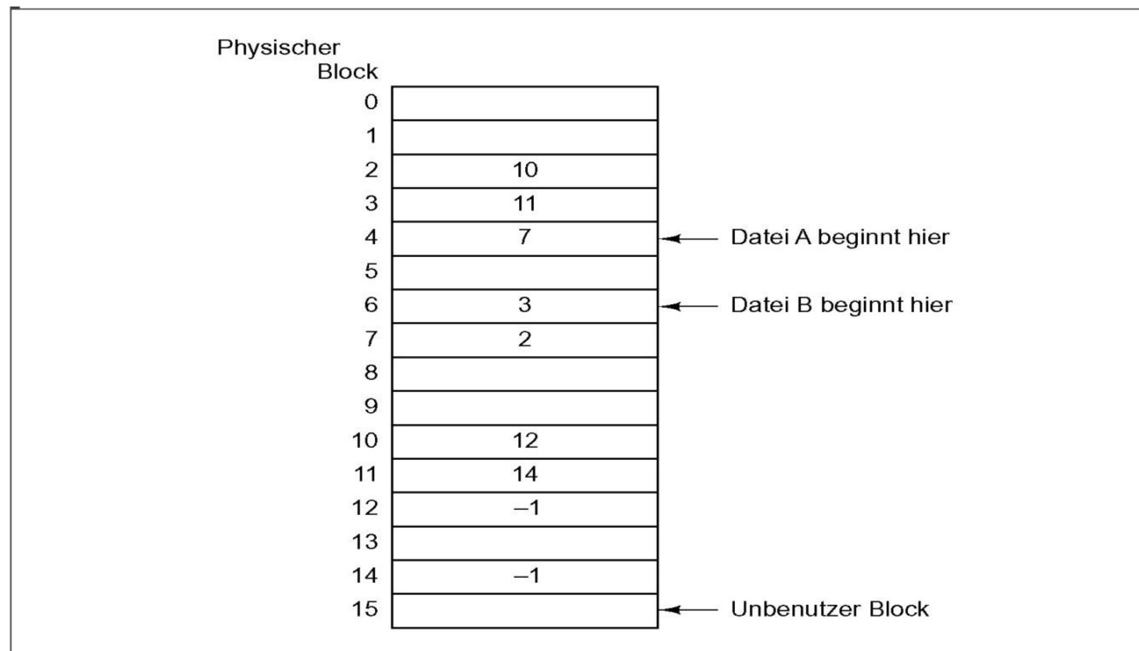


Abbildung 39.: FAT - File Allocation Table

Eine deutlich ältere, aber wesentlich intelligentere Speicherorganisation der Speichermedien stellt das Unix-Betriebssystem mit der Inode-Struktur bereit, wie es im Abschnitt 5.5. Physisches Dateisystem erläutert wird.

Anwendungsprogramme nutzen ausschließlich die Routinen des Dateisystems, um Dateien anzulegen, zu lesen, zu schreiben, zu löschen oder um Zugriffsrechte zu verändern. Nur so ist es möglich, dass Dateien, die von einem Anwendungsprogramm erzeugt wurden, von einem anderen weiterverarbeitet werden können. Das Dateisystem verwaltet die Namen der Dateien in so genannten Verzeichnissen (Directory).

Attribute der Dateien (Länge, Erstellungsdatum, Zugriffsrechte usw.), können in den Verzeichniseinträgen oder aber in separaten Listen abgelegt sein. Jedes Dateisystem verwaltet den verfügbaren Speicherbereich zur Ablage von Dateien (Freispeicherliste, Freiblockliste) ebenso, wie Informationen und die Größe und Lage des Wurzelverzeichnisses (root directory).

Informationen zur Freispeicherverwaltung und weitere Informationen werden im Kopf des Dateisystems abgelegt.

2.5. Das Speicherverwaltungssystem

Speicher ist eine wichtige Ressource, die sorgfältig verwaltet werden muss. Jeder Programmierer hätte am liebsten einen unendlich großen, unendlich schnellen Speicher, der auch noch nicht flüchtig (nonvolatile) ist, dessen Inhalt also nicht verloren geht, falls die Stromversorgung ausfällt.

> Von dem Mitgründer der Firma Microsoft **Bill Gates** stammt der Ausdruck ***„... ich kann mir nicht vorstellen, dass eine Anwendung mehr als 640 KByte Hauptspeicher benötigt. ...“***

Leider funktionieren reale Speicher so nicht. Aus diesem Grund haben die meisten Computer deshalb eine Speicherhierarchie, siehe 1.5.2. Hauptspeicher.

An der Spitze stehen die Register der CPU und ein sehr kleiner, sehr schneller, teurer und flüchtiger Cache-Speicher. Die nächsten Stufen bilden der einige hundert bis tausende Mbyte große mittelschnelle Hauptspeicher (**RAM**) und ein langsamer, billiger und nichtflüchtiger Plattenspeicher, der einige hundert Gbyte groß sein kann. Das Betriebssystem hat die Aufgabe, die Nutzung dieser verschiedenen Speicher zu koordinieren:

- Register,
- Cache-Speicher,
- Hauptspeicher (RAM) und
- Externe Speichermedien.

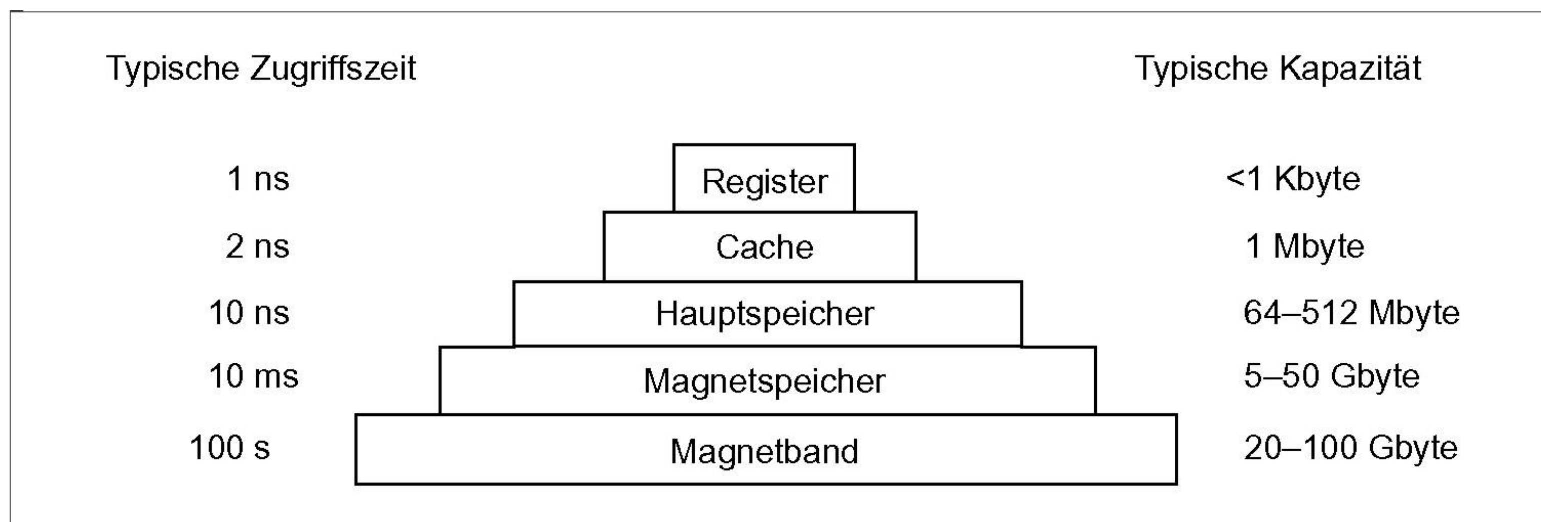


Abbildung 40.: Speicherhierarchie

Die in Abbildung 40 enthaltenen Werte sind sehr grobe Schätzungen und werden sich durch die stetige Entwicklung auch ständig zu größeren Speicherkapazitäten und geringeren Zugriffszeiten verschieben.

2.5.1. Grundlagen der Speicherverwaltung

Es gibt zwei Klassen von Speicherverwaltungssystemen:

- Speicherverwaltung nur im Hauptspeicher und
- Speicherverwaltung des Hauptspeichers mit Zuhilfenahme von Speichermedien (**Swapping** und **Paging**)

Die einfachste Strategie ist, nur ein Programm laufen zu lassen. Verschiedene Anwendungen werden nacheinander, sequenziell bearbeitet. Verschiedene Varianten der Aufteilung des Hauptspeichers zwischen dem Betriebssystem und der Anwendung sind denkbar, siehe Abbildung 41.

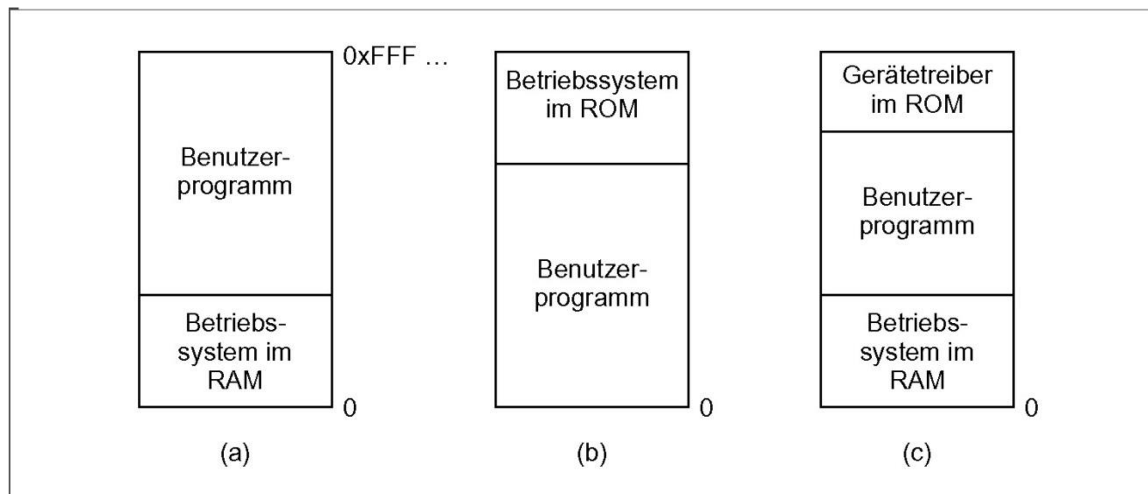


Abbildung 41.: Aufteilung des Hauptspeichers zwischen Betriebssystem und Anwendung

Die Variante **(a)** wurde früher in Mainframes und Minicomputern verwendet und wird heute nicht mehr eingesetzt. Variante **(b)** wird in einigen Palmtops und eingebetteten Systemen benutzt.

Die Variante **(c)** wurde von frühen Personal Computern (z.B. unter MS-DOS) verwendet. Der Teil des Systems im ROM wird **BIOS** (Basic Input Output System) genannt.

In einem so strukturierten System kann immer nur eine Anwendung laufen. Sobald der Benutzer einen Befehl eingegeben hat, lädt das Betriebssystem das entsprechende Programm vom Speichermedium in den Hauptspeicher und startet es.

Die meisten modernen Betriebssysteme können mehrere Prozesse gleichzeitig ausführen. Dabei kann ein Prozess bearbeitet werden, wenn ein anderer z.B. auf eine Ein- / Ausgabeoperation wartet. Die einfachste Art dieser Multiprogrammierung ist, den Hauptspeicher einfach in n (möglicherweise verschieden große) Bereiche aufzuteilen. Diese Einteilung kann beim Systemstart in Abhängigkeit von der zur Verfügung stehenden Hauptspeichergröße erfolgen.

Wenn ein Prozess gestartet wird, kann er an eine Warteschlange für den kleinsten Hauptspeicherbereich abgehängt werden, der für diesen Prozess groß genug ist. In Abbildung 42 sind die Verwaltungsmöglichkeiten der festen Hauptspeicherbereiche dargestellt. In Variante **(a)** existiert für jeden Speicherbereich eine eigene Warteschlange, während in Variante **(b)** alle Speicherbereiche über eine Warteschlange verwaltet werden.

Dieses System mit festen Hauptspeicherbereichen wurde viele Jahre lang von **IBM OS/360** auf den großen Mainframes verwendet und heißt **MFT** (**M**ultiprogramming with a **f**ixed number of **T**asks).

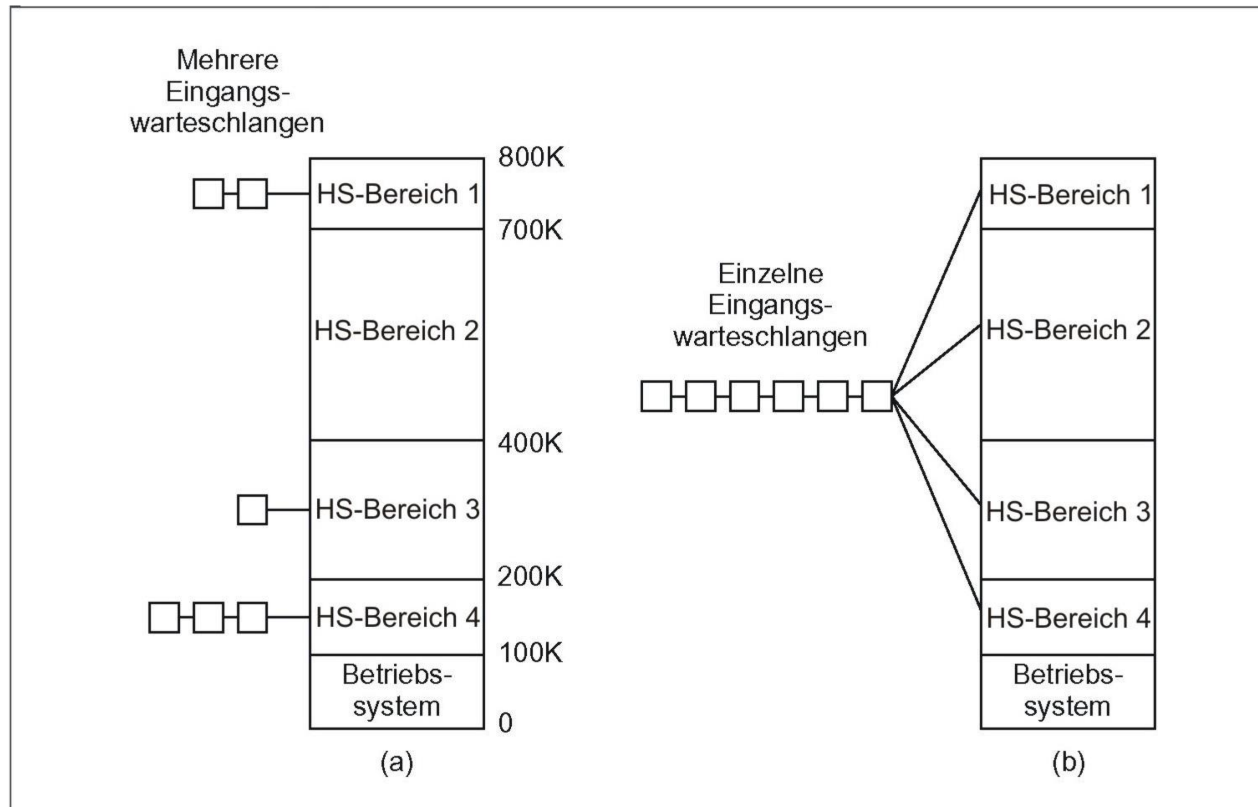


Abbildung 42.: feste Hauptspeicherbereiche mit verschiedenen Warteschlangenverwaltungen

2.5.2. Swapping

Für Timesharing-Systeme oder Personal Computer mit grafischer Benutzeroberfläche reicht die Speicherverwaltung mit festen Hauptspeicherbereichen nicht aus. Es kann vorkommen, dass der vorhandene Hauptspeicher nicht für alle aktiven Prozesse ausreicht. Deshalb müssen einige der Prozesse auf ein Speichermedium ausgelagert werden und bei Bedarf dynamisch wieder in den Hauptspeicher zurückgeholt werden.

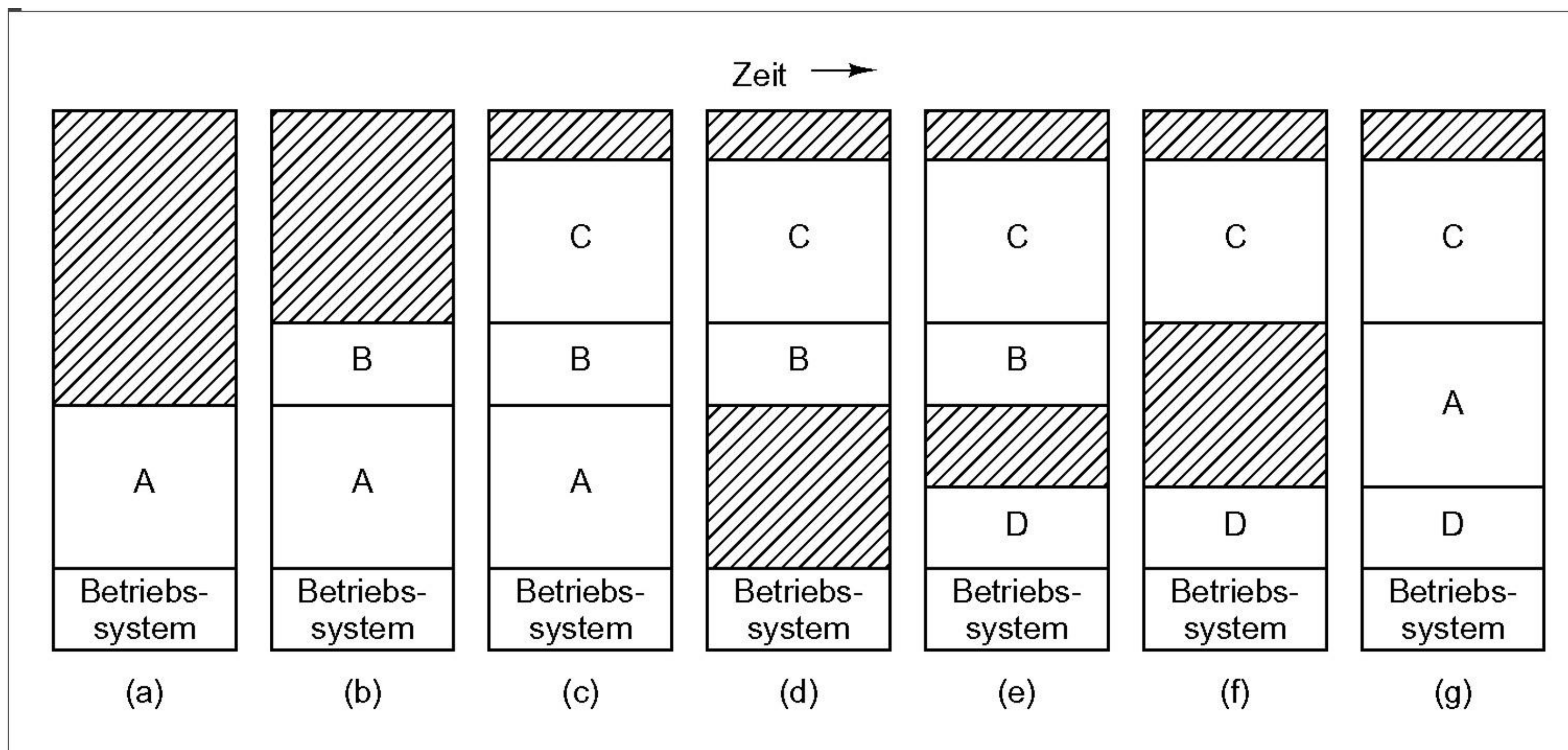


Abbildung 43.: Swapping

Für diese Art von Speicherverwaltung gibt es zwei grundlegende Ansätze. Für welche man sich entscheidet, hängt teilweise von der verfügbaren Hardware ab. Die einfachste Strategie ist das sogenannte Swapping, siehe Abbildung 43, bei dem jeder Prozess komplett in den Hauptspeicher geladen wird, eine gewisse Zeit laufen darf und anschließend wieder auf das Speichermedium ausgelagert wird.

Bei der anderen Strategie, dem virtuellen Speicher, können auch dann Programme laufen, wenn sich nur ein Teil von ihnen im Hauptspeicher befindet, siehe Abschnitt 2.5.4. Paging.

Der Hauptunterschied zwischen den festen Hauptspeicherbereichen aus Abbildung 42 und den variablen Hauptspeicherbereichen aus Abbildung 43 besteht darin, dass Anzahl, Größe und Ort der Speicherbereiche hier nicht feststehen, sondern sich dynamisch ändern können. Diese Flexibilität verbessert die Speicherausnutzung, weil feste Speicherbereiche zu klein oder zu groß sein können, aber sie machen auch die Zuteilung, Freigabe und Verwaltung des Speichers komplizierter.

Eine wichtige Frage ist auch, wie viel Speicher für einen Prozess reserviert werden soll, wenn er gestartet oder wieder eingelagert wird. Bei fester Speichergröße ist diese Verwaltung trivial.

Wenn man jedoch davon ausgeht, dass die meisten Prozesse während ihrer Laufzeit wachsen, ist es sinnvoll immer etwas mehr Hauptspeicher zu reservieren, wenn ein Prozess eingelagert wird. Dadurch lässt sich der zusätzliche Aufwand vermeiden, wenn ein Prozess nicht mehr in seinen Speicherbereich passt. Wenn ein Prozess ausgelagert wird, sollte natürlich nur der wirklich benutzte Hauptspeicher auf das Speichermedium geschrieben werden.

In Abbildung 44 (a) ist eine Speicherkonfiguration zu sehen, in der für zwei Prozesse Platz reserviert wurde.

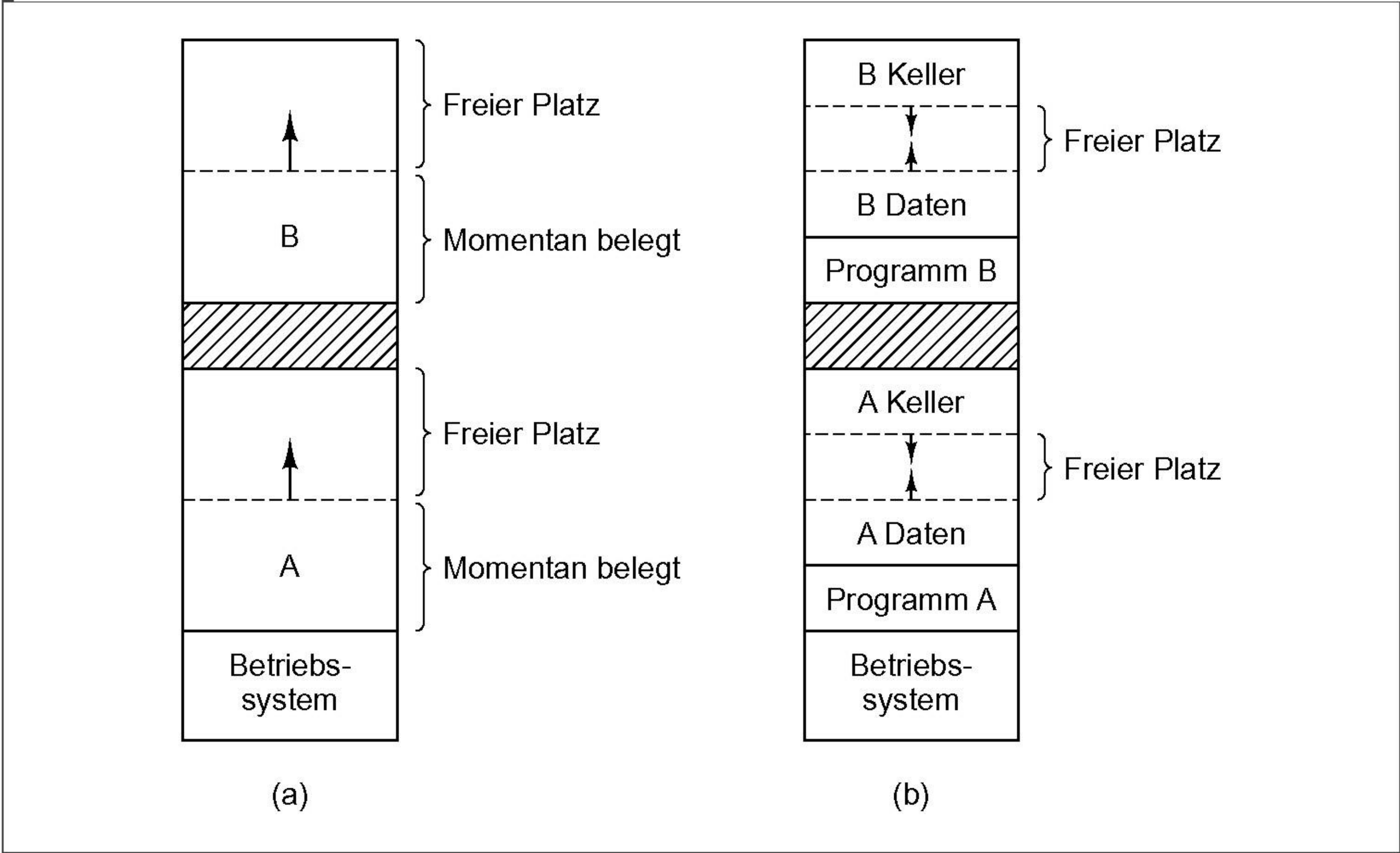


Abbildung 44.: Reservierung von Hauptspeicherplatz

Wenn ein Prozess zwei wachsende Segmente haben kann, zum Beispiel ein Daten- und ein Kellersegment, dann liegt es nahe, die Anordnung aus Abbildung 44 (b) zu verwenden. Der freie Speicher in der Mitte kann für jedes der beiden Segmente genutzt werden. Wenn er nicht ausreicht, müssen die Prozesse in ein größeres Loch verschoben, ausgelagert oder abgebrochen werden.

2.5.3. virtueller Speicher

Vor langer Zeit stießen Programmierer zum ersten Mal auf das Problem, dass die Programme zu groß für den verfügbaren Speicher wurden. Normalerweise lösten sie dieses Problem, indem sie die Programme in mehrere Teile aufspalteten, so genannte **Overlays**. Zunächst wurde Overlay 0 ausgeführt, das dann, sobald es fertig war, ein anderes Overlay aufrief.

Obwohl die Overlays vom Betriebssystem verwaltet wurden, musste der Programmierer das Programm selbst aufteilen. Große Programme in kleine, modulare Teile aufzuspalten, war eine zeitaufwendige und langweilige Arbeit. Deshalb wurde diese Aufgabe bald durch den Computer erledigt.

Diese Methode wurde als **virtueller Speicher** bekannt. So kann man beispielsweise ein 16 Mbyte großes Programm auf einer Maschine mit 4 Mbyte Hauptspeicher ausführen, indem man die 4 Mbyte, die im Hauptspeicher liegen, zu jedem Zeitpunkt sorgfältig auswählt und Teile des Programms nach Bedarf ein- und auslagert.

Virtueller Speicher funktioniert auch für Systeme mit Multiprogrammierung. Dabei können dann Teile von mehreren verschiedenen Programmen gleichzeitig im Hauptspeicher liegen.

2.5.4. Paging

Die meisten Systeme mit virtuellem Speicher verwenden eine Technik namens **Paging**. Für jeden Computer gibt es eine Menge von Speicheradressen, die Programme erzeugen können. Adressen können u.a. mit Hilfe von Indizierung, Basisregistern und Segmentregistern generiert werden.

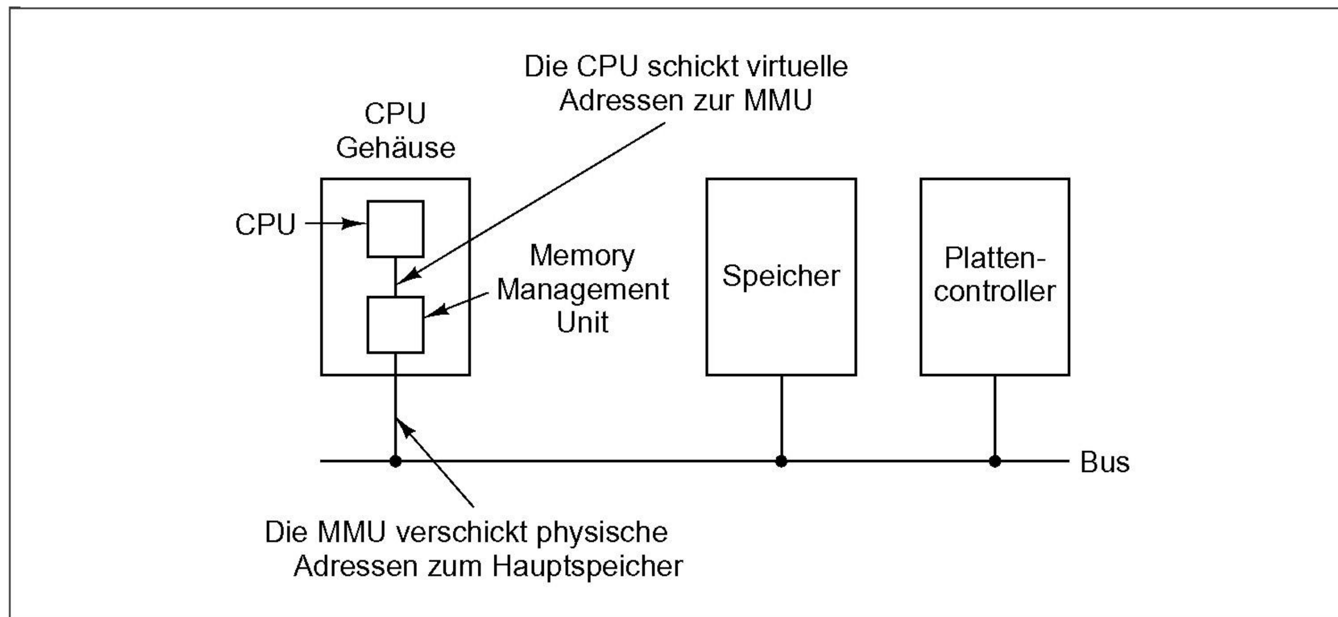


Abbildung 45.: Funktion der MMU

Diese vom Programm generierten Adressen heißen virtuelle Adressen und bilden den virtuellen Adressraum. Bei Rechnern mit virtuellem Speicher gehen die Adressen nicht direkt an den Hauptspeicher, sondern an die MMU (Memory Management Unit), die die virtuelle Adresse auf eine physische Adresse abbildet, Abbildung 45.

3. Verfügbarkeit und Hochverfügbarkeit

3.1. Verfügbarkeit

Ein System wird als verfügbar bezeichnet, wenn es in der Lage ist, die Aufgaben zu erfüllen, für die es vorgesehen ist.

Definition 7 - Verfügbarkeit:

Als Verfügbarkeit wird die Wahrscheinlichkeit bezeichnet, dass ein System innerhalb eines spezifizierten Zeitraums funktionstüchtig (verfügbar) ist. Die Verfügbarkeit wird als Verhältnis aus ungeplanter (fehlerbedingter) Stillstandszeit (= Ausfallzeit) und gesamter Produktionszeit eines Systems bemessen:

$$\text{Verfügbarkeit(in Prozent)} = \left(1 - \frac{\text{Ausfallzeit}}{\text{Produktionszeit} + \text{Ausfallzeit}}\right) \cdot 100$$

oder auch:

$$\text{Verfügbarkeit(in Prozent)} = \left(\frac{\text{Produktionszeit(uptime)}}{\text{Produktionszeit(uptime)} + \text{Ausfallzeit(downtime)}}\right) \cdot 100$$

3.2. Hochverfügbarkeit

Die genaue Definition von Hochverfügbarkeit kann variieren. Das Institute of Electrical and Electronics Engineers (IEEE) gibt folgende Definition:

> "High Availability (HA for short) refers to the availability of resources in a computer system, in the wake of component failures in the system."

Eine andere Definition der Hochverfügbarkeit lautet:

Definition 8 - Hochverfügbarkeit:

> **Ein System gilt als hochverfügbar, wenn eine Anwendung auch im Fehlerfall weiterhin verfügbar ist und ohne unmittelbaren menschlichen Eingriff weiter genutzt werden kann. In der Konsequenz heißt dies, dass der Anwender keine oder nur eine kurze Unterbrechung wahrnimmt. Hochverfügbarkeit (abgekürzt auch *HA*, abgeleitet von engl. *high availability*) bezeichnet also die Fähigkeit eines Systems, bei Ausfall einer seiner Komponenten einen uneingeschränkten Betrieb zu gewährleisten.**

3.3. Verfügbarkeitsklassen

Die Frage, ab welcher Verfügbarkeitsklasse ein System als hochverfügbar einzustufen ist, wird je nach Definition der Verfügbarkeit unterschiedlich beantwortet.

Eine Verfügbarkeit von 99 % definiert im Allgemeinen keine Hochverfügbarkeit, sie wird allgemein heutzutage als grundlegend oder normal angesehen, zumindest bei qualitativ hochwertigen EDV-Geräten. Folglich wird von Hochverfügbarkeit erst ab 99,9 % oder höher gesprochen. Ob aber bereits 3*9 ausreichen oder erst 4*9 oder 5*9 ein System zum Hochverfügbaren System machen, ist quellen- und herstellerabhängig sowie unter dem jeweiligen Einsatzszenario zu bewerten.

Im Allgemeinen kann ein System als hochverfügbar eingestuft werden, wenn seine jährliche Ausfallzeit im Bereich weniger Minuten (~99,999 % bzw. AEC-2) oder darunter liegt.

Berechnet man mit der obigen Formel die Verfügbarkeit im Zeitraum eines Jahres, so entspricht eine Verfügbarkeit von 99,99 % beispielsweise einer Stillstandszeit von 52,6 Minuten. Man benutzt nun üblicherweise die Anzahl der Neunen in der Prozentangabe, um die Verfügbarkeitsklasse zu kennzeichnen: so bedeutet das obige Beispiel mit 99,99 % die Verfügbarkeitsklasse 4.

Bei einer gegebenen maximalen Ausfallzeit folgt eine Übersicht der relevanten Klassen 2 bis 6, wobei ein Jahr mit durchschnittlich 365,25 Tagen, der Monat als 1/12 Jahr gerechnet wird:

3.3.1. Verfügbarkeitsklasse 2

99 % ≡ 438 Minuten/Monat bzw. 7:18:18 Stunden/Monat = 87,7 Stunden/Jahr, d. h. 3 Tage und 15:39:36 h

3.3.2. Verfügbarkeitsklasse 3

99,9 % ≡ 43:48 min/Monat oder 8:45:58 Stunden/Jahr

3.3.3. Verfügbarkeitsklasse 4

99,99 % ≡ 4:23 Minuten/Monat oder 52:36 Minuten/Jahr

3.3.4. Verfügbarkeitsklasse 5

99,999 % ≡ 26,3 Sekunden/Monat oder 5:16 Minuten/Jahr

3.3.5. Verfügbarkeitsklasse 6

99,9999 % ≡ 2,63 Sekunden/Monat oder 31,6 Sekunden/Jahr

3.4. Availability Environment Classification

Die Harvard Research Group (HRG) teilt Hochverfügbarkeit in ihrer *Availability Environment Classification* (AEC) in sechs Klassen ein.

HRG-Klasse	Bezeichnung	Erklärung
AEC-0	*Conventional*	Funktion kann unterbrochen werden, Datenintegrität ist nicht essentiell
AEC-1	*Highly Reliable*	Funktion kann unterbrochen werden, Datenintegrität muss jedoch gewährleistet sein
AEC-2	*High Availability*	Funktion darf nur innerhalb festgelegter Zeiten oder zur Hauptbetriebszeit minimal unterbrochen werden
AEC-3	*Fault Resilient*	Funktion muss innerhalb festgelegter Zeiten oder während der Hauptbetriebszeit ununterbrochen aufrechterhalten werden
AEC-4	*Fault Tolerant*	Funktion muss ununterbrochen aufrechterhalten werden, 24/7-Betrieb (24 Stunden, 7 Tage die Woche) muss gewährleistet sein
AEC-5	*Disaster Tolerant*	Funktion muss unter allen Umständen verfügbar sein

Tabelle 8.: Availability Environment Classification

3.5. Vereinbarter Zeitraum der Verfügbarkeit

Die Hochverfügbarkeit wird in Unternehmen häufig im Rahmen von Service Level Agreements (SLA) definiert, und stellt ein wesentliches Bewertungskriterium für IT-Services dar.

Viele hochverfügbare Systeme müssen 24 Stunden * 7 Tage online sein, also das ganze Jahr „rund um die Uhr".

Manche dieser Systeme müssen die Eigenschaft der Hochverfügbarkeit jedoch nur für einen bestimmten Zeitausschnitt haben: Handelssysteme der Deutschen Börse etwa brauchen nachts und an börsenfreien Tagen nicht hochverfügbar zu sein. Die Hochverfügbarkeit bezieht sich bei diesen Systemen damit nur auf die Tageszeit und/oder die Arbeitstage, an denen es benötigt wird.

4. Virtualisierung

In der Informatik ist die eindeutige Definition des Begriffs **Virtualisierung** nicht möglich, da der Begriff in vielen unterschiedlichen Anwendungsfällen anders ausgeprägt ist. Es gibt viele Konzepte und Technologien im Bereich der Hardware und Software, die diesen Begriff verwenden. Ein sehr offener Definitionsversuch lautet wie folgt:

Definition 9 - Virtualisierung:

Virtualisierung bezeichnet in der Informatik die Erzeugung von virtuellen (d. h. nicht physikalischen) Dingen wie einer emulierten Hardware, eines Betriebssystems, Datenspeichers oder Netzwerkressource. Dies erlaubt es etwa, Ressourcen von Computern (insbesondere im Server-Bereich) transparent zusammenzufassen oder aufzuteilen, oder ein Betriebssystem innerhalb eines anderen auszuführen.

Primäres Ziel ist, dem Benutzer eine Abstraktionsschicht zur Verfügung zu stellen, die ihn von der eigentlichen Hardware – Rechenleistung und Speicherplatz – isoliert. Eine logische Schicht wird zwischen Anwender und Ressource eingeführt, um die physischen Gegebenheiten der Hardware zu verstecken.

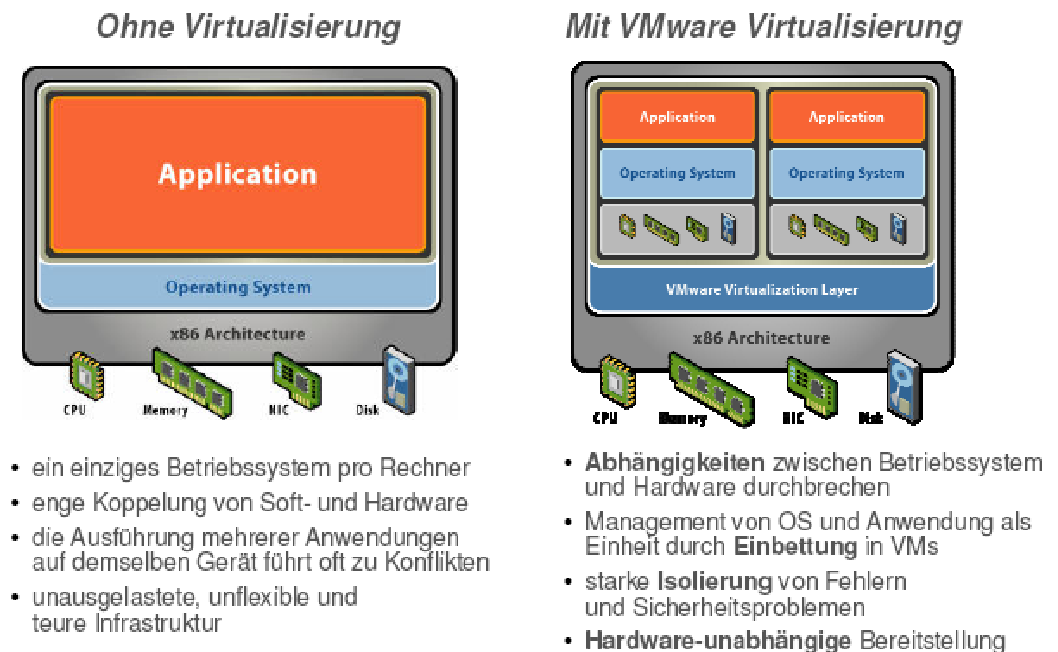


Abbildung 46.: Gründe für die Virtualisierung

Dabei wird jedem Anwender (so gut es geht) vorgemacht, dass er (a) der alleinige Nutzer einer Ressource sei, bzw. (b) werden mehrere (heterogene) Hardwareressourcen zu einer homogenen Umgebung zusammengefügt. Die für den Anwender unsichtbare bzw. transparente Verwaltung der Ressource ist dabei in der Regel die Aufgabe des Betriebssystems.

4.1. Vorteile der Virtualisierung

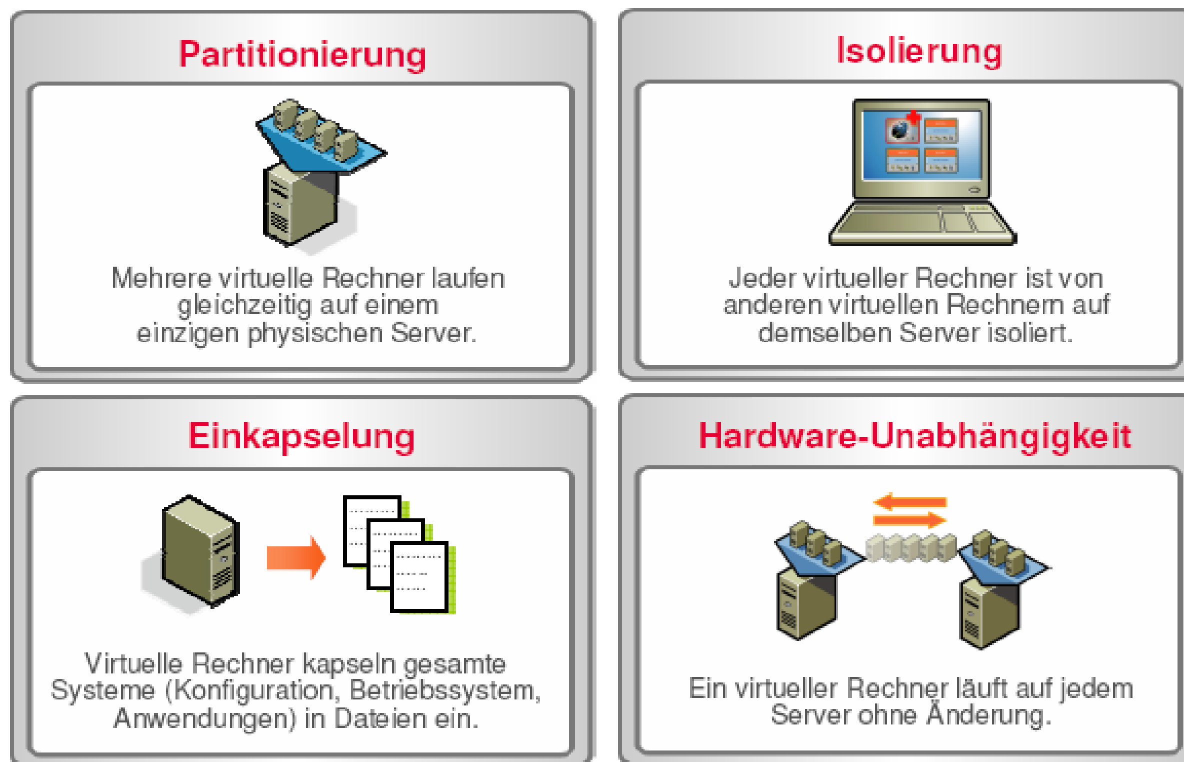
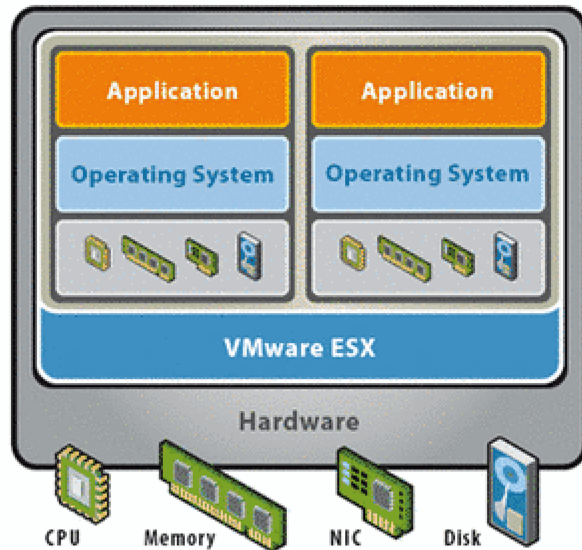


Abbildung 47.: Vorteile der Virtualisierung

4.2. Softwarevirtualisierung



Die Softwarevirtualisierung kann für mehrere Zwecke eingesetzt werden, z. B. um ein Betriebssystem oder nur eine Anwendung zu simulieren.

Bei Virtualisierung auf Betriebssystemebene wird anderen Computerprogrammen eine komplette Laufzeitumgebung virtuell innerhalb eines geschlossenen Containers oder „jails“ zur Verfügung gestellt, es wird kein zusätzliches Betriebssystem gestartet, was zur Folge hat, dass es nicht möglich ist ein anderes OS als das Hostsystem zu betreiben. Die OS-Container stellen eine Teilmenge des Wirtbetriebssystems dar. Vorteil dieses Konzepts liegt in der guten Integration der Container in das Gastbetriebssystem. Der Nachteil dieses Konzepts liegt in den Containern. Aus den Containern heraus können keine Treiber geladen bzw. andere Kernel geladen werden.

Abbildung 48.: Softwarevirtualisierung mit VMware

Bei der OS-Virtualisierung läuft immer nur ein Host-Kernel, wobei UML eine gewisse Sonderrolle zukommt, da dort ein spezieller User-Mode-Kernel unter der Kontrolle des Host-Kernels läuft. Beispiele: OpenSolaris, Zoning, BSD jails, Mac-on-Linux, OpenVZ, Virtuozzo, Linux-VServer, User Mode Linux und Xenserver von Citrix.

Bei Virtualisierung mittels eines Virtual Machine Monitors („virtuelle Maschine") werden die bereitstehenden nativen (= real physisch verfügbaren) Ressourcen intelligent verteilt. Dies kann durch Hardware-Emulation, Hardware-Virtualisierung oder Virtualisierung mittels Hypervisors stattfinden.

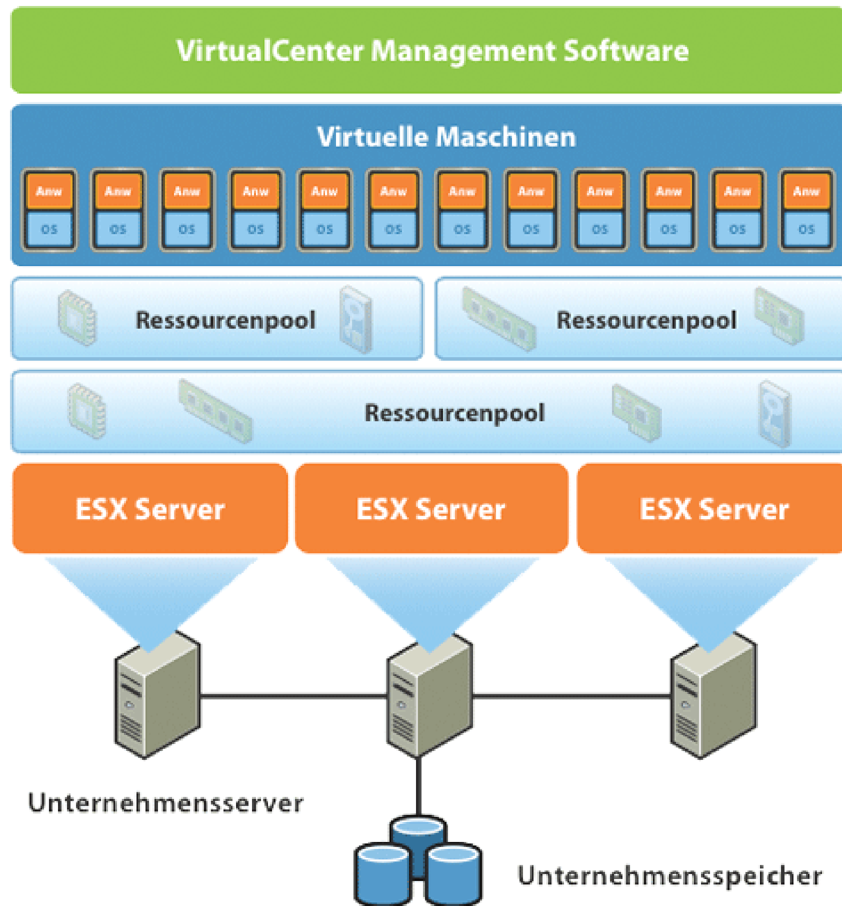


Abbildung 49.: Verbund mehrerer ESX-Server

Den einzelnen Gastsystemen wird dabei jeweils ein eigener kompletter Rechner mit allen Hardware-Elementen (Prozessor, Laufwerke, Arbeitsspeicher, usw.) vorgespiegelt. Der Vorteil ist, dass an den Betriebssystemen selbst (fast) keine Änderungen erforderlich sind und die Gastsysteme alle ihren eigenen Kernel laufen haben, was eine gewisse Flexibilität im Gegensatz zur Betriebssystemvirtualisierung mit sich bringt.

Wenn weder diese Hardware-Elemente noch die Betriebssysteme der Gastsysteme diese Form der Virtualisierung unterstützen, muss die Virtualisierungssoftware eine Emulationsschicht benutzen, um jedem Gastsystem vorzuspiegeln, es hätte die Hardware für sich allein. Diese Emulation ist oft weniger effizient als direkter Zugriff auf die Hardware, was dann zu einer verringerten Geschwindigkeit führen kann.

Beispiele: VMware Workstation, Microsoft Virtual PC, VirtualBox, Parallels Workstation.

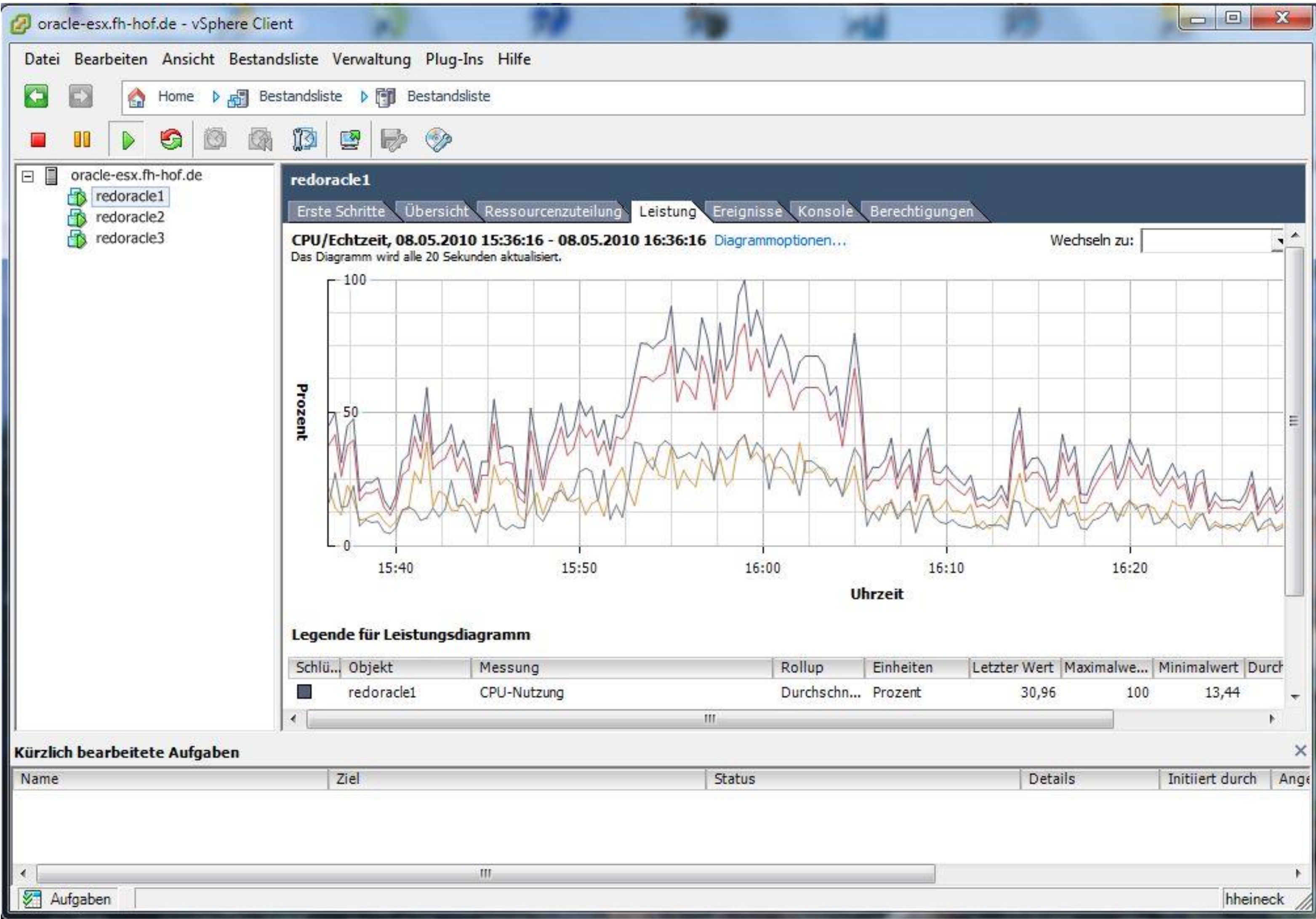


Abbildung 50.: RedCluster auf einem ESXi-Server

4.3. Hardwarevirtualisierung

Hardware-Emulation (irreführend auch Full Virtualisation genannt):

Die virtuelle Maschine simuliert die komplette Hardware und ermöglicht einem nichtmodifizierten Betriebssystem, das für eine andere CPU ausgelegt ist, den Betrieb. Beispiele: Bochs (hier anstatt Emulation Simulation), PPC-Version von Microsoft Virtual PC.

Hardware-Virtualisierung (native Virtualisation, full Virtualisation):

Die virtuelle Maschine stellt dem Gastbetriebssystem nur Teilbereiche der physischen Hardware in Form von virtueller Hardware zur Verfügung. Diese reicht jedoch aus, um ein unverändertes Betriebssystem darauf in einer isolierten Umgebung laufen zu lassen. Das Gastsystem muss hierbei für den gleichen CPU-Typ ausgelegt sein. Beispiele: VMware, x86-Version von Microsoft Virtual PC, Xen 3.0 auf Prozessoren mit Hardware-Virtualisierungstechnologien: Intel VT-x oder AMD Pacifica.

Hierfür können entweder das ganze System (Partitioning mit LPAR, Domaining) oder einzelne seiner Komponenten wie z. B. CPU (Intels Vanderpool oder AMDs Pacifica) virtualisiert werden.

4.4. Netzwerkvirtualisierung

Durch Virtual Local Area Networks werden Geräte in einem lokalen Netzwerk in Gruppen aufgeteilt, zwischen denen Verbindungen grundsätzlich unterbunden sind, aber gezielt ermöglicht werden können. Ein Virtual Private Network bildet ein nach außen abgeschirmtes Netzwerk über fremde oder nicht vertrauenswürdige Netze. Software für den gleichzeitigen Betrieb mehrerer virtueller Betriebssysteme auf einem Computer kann ein virtuelles Netzwerk bereitstellen, über das diese kommunizieren. Es können auch mehrere Netze simuliert werden, über die beispielsweise zur Erprobung wiederum ein *Virtual Private Network* aufgebaut wird.

4.5. Virtualisierungslösungen

Kommerzielle Virtualisierungslösungen

- VMware
- Citrix XenServer
- Virtuozzo
- vAdmin

Freie Virtualisierungslösungen

- Xen
- lguest
- Oracle VM
- VirtualBox
- KVM
- OpenVZ
- Linux-VServer
- SandBoxIE
- Microsoft Hyper-V

5. Das Betriebssystem Unix

Das Betriebssystem Unix blickt auf eine lange Vergangenheit zurück. Ursprünglich wurde es für PDP-Rechner der Firma **DEC** (**D**igital **E**quipment **C**orporation) entwickelt und sollte hauptsächlich die Bedürfnisse von professionellen Softwareentwicklern befriedigen.

Heute realisiert das Betriebssystem **Unix** seine 98'er Spezifikation. Damit ist es auf Workstations mit **64-Bit-Prozessoren** zugeschnitten. Zu den Neuerungen gehört die Einführung von **Threads**, mit denen ein Programm mehrere Funktionen gleichzeitig ausführen kann. Ebenfalls neu sind **Echtzeitfunktionen** mit denen Programmlaufzeiten vorhersagbar sind. Insbesondere für **Datenbanken** ist die Möglichkeit, sehr große Dateien verarbeiten zu können, von nicht unerheblicher Bedeutung.

Ein Schritt in Richtung einer völligen Unabhängigkeit von der zugrunde liegenden Architektur ist die Normierung ganzer Zahlen mit einer Länge von mehr als 64-Bit. Ebenfalls neu ist die Einbeziehung der Benutzeroberfläche in die Spezifikation.

Damit hat das Betriebssystem **Unix** eine mehr als 30 jährige Entwicklung hinter sich und belegt heute Marktanteile, die ihm andere Betriebssysteme streitig machen wollen.

5.1. Historische Entwicklung

Als **Ken Thompson 1969** bei **Bell Laboratories**, einer gemeinsamen Tochter der Firmen **AT&T** und **Western Electric**, die Entwicklung eines neuen Betriebssystems begann, waren die meisten der vorhandenen Systeme ausgesprochene **Closed Shop - Batch Systems.** D.h., der **Programmierer** gab seine Lochkarten oder Lochstreifen beim **Operator** ab, diese wurden in den Rechner eingelesen und ein Rechenauftrag nach dem anderen abgearbeitet, siehe Abbildung 3.

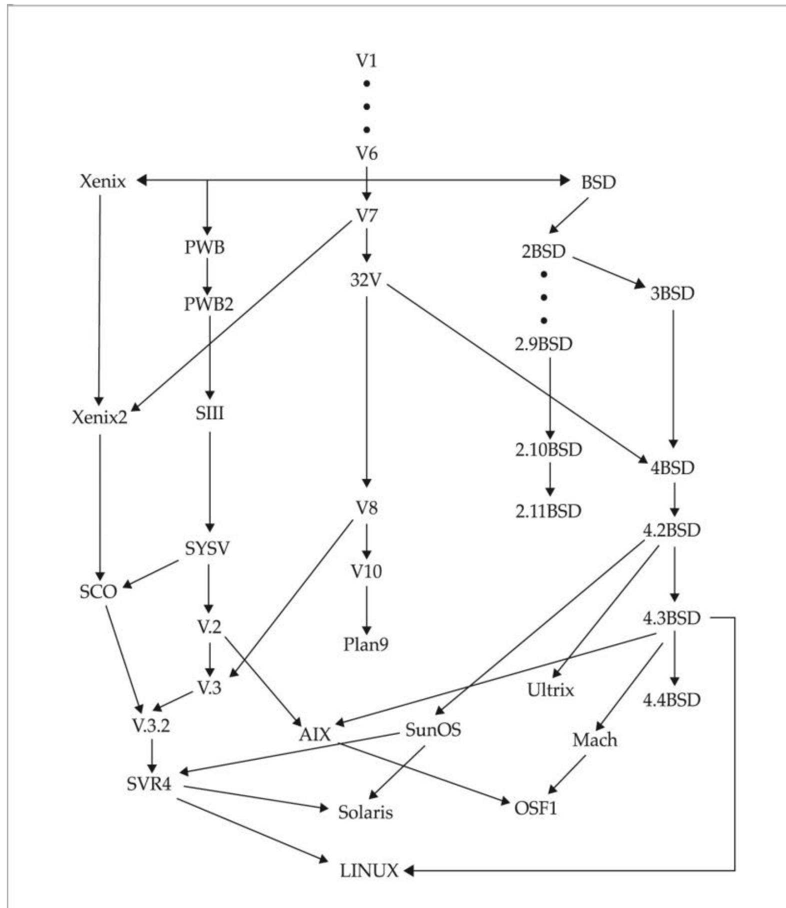
Der Programmierer konnte dann nach einiger (in der Regel längeren) Zeit seine Ergebnisse abholen.
Ziel von Ken Thompsons Entwicklung war es deshalb, ein System zu schaffen, auf welchem mehrere Programmierer im Team und im Dialog mit dem Rechner arbeiten, Programme entwickeln, korrigieren und dokumentieren konnten, ohne von einem Großrechner mit allen seinen Restriktionen abhängig zu sein. Dabei standen Funktionalität, strukturelle Einfachheit und Transparenz sowie leichte Bedienbarkeit im Vordergrund der Entwicklung.

> **Das Betriebssystem Unix ist aus diesem Grunde ein System von Programmierern für Programmierer, die wissen was sie tun.**

Dieses erste System mit dem Namen **Unix** lief auf einer **PDP-7**, einem Kleinrechner der Firma **DEC**. Die erste Version von **Unix** war dabei in der Assemblersprache der **PDP-7** geschrieben.

Um bei künftigen Projekten die Maschinenabhängigkeit durch eine maschinennahe Sprache zu umgehen, entwarf Thompson die Programmiersprache **B**, aus der dann **Dennis Ritchie** die Sprache **C** entwickelte.

Abbildung 51.: Ken Thompson und Dennis Ritchie



Unix wurde **1971** in der Programmiersprache **C** ungeschrieben und auf die **PDP-11** übertragen. Von nun an erfolgte die Weiterentwicklung des Systemkerns sowie der meisten Dienstprogramme in dieser Sprache. Die Kompaktheit und strukturelle Einfachheit des Systems ermunterte viele Benutzer zur eigenen Aktivität und Weiterentwicklung des Systems, so dass **Unix** recht schnell einen relativ hohen Reifegrad erreichte.

Dies ist deshalb bemerkenswert, da kein Entwicklungsauftrag hinter diesem Prozess stand und die starke Verbreitung von **Unix** nicht auf den Vertrieb oder die Werbung eines Herstellers, sondern primär auf das Benutzerinteresse zurückzuführen ist. Hilfreich hierbei war sicherlich jedoch auch, dass für Hochschulen und Universitäten die **Unix-Quellcodelizenz** bisher praktisch für die Kopier- und Dokumentationskosten von **Bell Laboratories** abgegeben wurde.

Abbildung 52.: historische Entwicklung

Dass die ursprünglich angestrebten Ziele trotz zahlreicher funktionaler und qualitativer Erweiterungen lange Zeit erhalten geblieben sind, zeigt sich darin, dass der Kern des Unix-Systems nur aus etwa 20.000 Zeilen Programm bestand, von denen rund 2.000 Zeilen in Assembler geschrieben worden sind. Maschinenabhängige Assemblerteile werden dabei nur dort verwendet, wo hohe Effizienz oder spezielle Maschineneigenschaften dies notwendig machen.

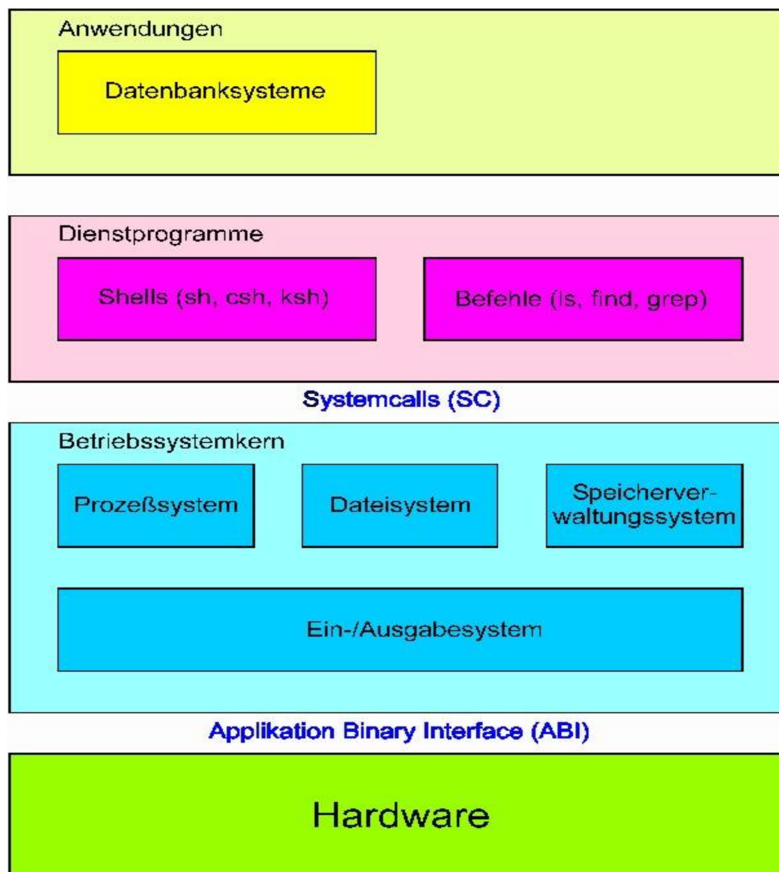
5.2. Die Entwicklungsgeschichte von Linux

Professor **Andrew S. Tanenbaum** implementierte 1987 ein zu Unix Version 7 kompatibles System und nannte es **Minix**. Es diente ihm als Lehrsystem für seine Studenten und wurde im Quellcode für ein gewisses Entgelt verfügbar gemacht. Größtes Manko von Minix waren die durch den Autor beschränkten Erweiterungsmöglichkeiten im Kernel, so dass z.B. das **X-System** niemals unter Minix laufen konnte.

1991 kaufte sich **Linus Torvalds**, ein finnischer Student einen 386er Personal Computer und installierte darauf das Betriebssystem Minix. Er schrieb einige Programme in Assemblersprache für den Intel 80386, z.B. ein Terminalprogramm, um sich in den Uni-Unix-Rechner einloggen zu können.

Weitere Nachteile von Minix und die damals sehr leistungsfähigen Möglichkeiten des Prozessors, z.B. der **Real Mode** und der viel leistungsfähigere **Protected Mode** brachten Linus dazu, sein eigenes Betriebssystem zu schreiben. Im September 1991 stellte er die **Version 0.01** seines Betriebssystems **pubOS/Linux** ins Internet, nachdem er schon im Vorfeld mit der Gemeinde der Minix Nutzer über sein Anliegen kommuniziert hatte.

Durch Hinweise und Anregungen zu seiner noch sehr fehleranfälligen Version wurde Torvalds angestachelt weiter an seinem Betriebssystem zu arbeiten. Nach dem Portieren einer Shell folgte der wichtigste Bestandteil, der **gcc**-Compiler. Im Oktober 1991 veröffentlichte er die Version 0.02 und nach mehreren Fehlerkorrekturen und einigen zusätzlichen Programmen im November die Version 0.03.
Von Anfang an stellte Torvalds seine Sourcen unter die Verantwortung der **GPL** (**G**NU **P**ublic **L**icence), so dass diese frei kopiert werden konnten und jedem Interessenten zur Verfügung standen. Gleichzeitig war er auf Konformität zum **POSIX-Standard** bedacht, wodurch **Linux** ohne großen Aufwand auf andere Plattformen portierbar wurde.



5.3. Schalenmodell von Unix

Zur Verdeutlichung der Zusammenarbeit verschiedener Komponenten in einem Rechnersystem wird allgemein das Schalenmodell, auch als Schichtenmodell bezeichnet, verwendet. Wie der Name aussagt, werden dabei einzelne Komponenten in Form von Schalen oder Schichten dargestellt. Die Grenzen zwischen den einzelnen Schalen werden dabei als **Schnittstellen** bezeichnet.

In Abbildung 53 wird dieses Modell beispielhaft auf Unix angewendet.

Abbildung 53.: Schalenmodell von Unix

Die Funktionen des Betriebssystemkerns sind dabei speziell hervorgehoben. Die Dialogschnittstelle zur Kommunikation mit dem Benutzer (zeichenorientiert) wird dabei als **Shell** bezeichnen.

Diese Shells unter Unix erfüllen zwei Funktionen, sie werden

- als Kommandointerpreter und
- als Programmiersprache

verwendet.

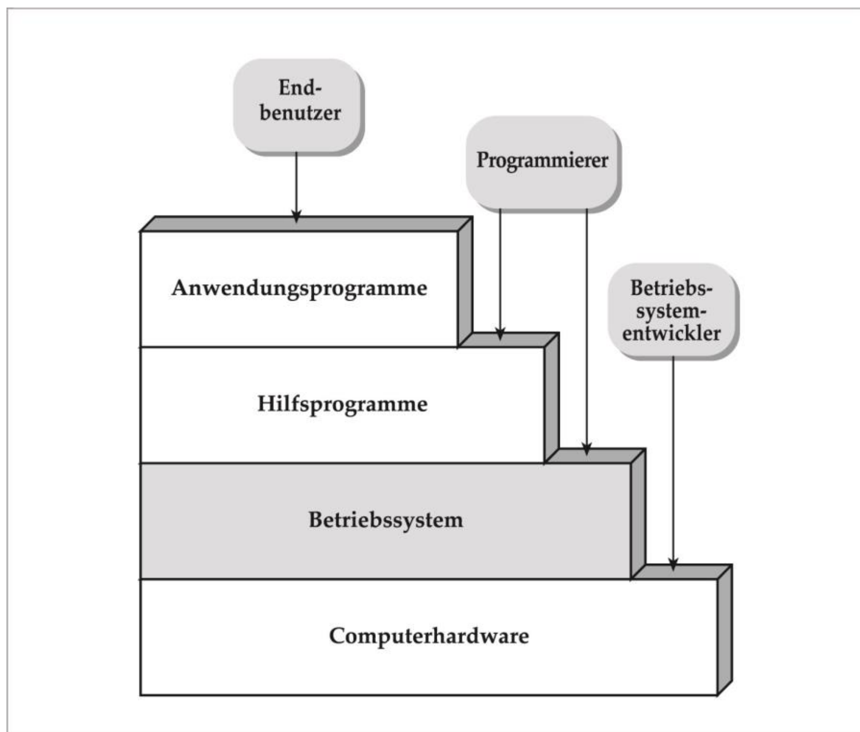


Abbildung 54.: Nutzerzugriffe

Die zweite Funktion ist unter Unix deshalb so bedeutend, da die gesamte Verwaltung (Administration) des Betriebssystems mit Skripten in dieser „Programmiersprache" erfolgt. Diese Möglichkeit wird im Abschnitt 7. Shell-Programmierung behandelt.

Der Zugriff der einzelnen Nutzer auf die einzelnen Schichten wird in Abbildung 54 dargestellt.

Prozessmanagement

Aufruf	Beschreibung
pid = fork( )	Erzeugen eines neuen Kindprozesses vom Vater
pid = waitpid(pid, &statloc, options)	Warten auf Beendigung des Kindes
s = execve(name, argv, environp)	Speicherabbild eines Prozesses ersetzen
exit(status)	Prozess beenden und Status zurückliefern

Dateimanagement

Aufruf	Beschreibung
fd = open(file, how,...)	Datei zum Lesen, Schreiben öffnen
s = close(fd)	Offene Datei schließen
n = read(fd,buffer, nbytes)	Daten aus Datei in Puffer lesen
n = write(fd, buffer, nbytes)	Daten vom Puffer in Datei schreiben
position = lseek(fd, offset, whence)	Dateilesezeiger bewegen
s = stat(name, &buf)	Status einer Datei ermitteln

Verzeichnis- und Dateimanagement

Aufruf	Beschreibung
s = mkdir(name, mode)	Erzeugen eines neuen Verzeichnisses
s = rmdir(name)	Löschen eines leeren Verzeichnisses
s = link(name1, name2)	Neuer Eintrag name2 zeigt auf name1
s = unlink(name)	Verzeichniseintrag löschen
s = mount(special,name, ag)	Dateisystem einhängen
s = umount(special)	Eingehängtes Dateisystem entfernen

Verschiedenes

Aufruf	Beschreibung
s = chdir(dirname)	Wechsel des aktuellen Verzeichnisses
s = chmod(name, mode)	Änderung der Dateirechte
s = kill(pid, signal)	Signal an einen Prozess schicken
seconds= time(&seconds)	Zeit seit 1. Januar 1970 erfragen

Tabelle 9.: Unix-Systemcalls

Die Unix-Systemcalls sind seit Jahren standardisiert und im so genannten **System V Interface Definition Guide** beschrieben.

Eine Auswahl der dort beschrieben Systemcalls ist in Tabelle 9 enthalten.

5.4. Logisches Dateisystem

5.4.1. Aufbau des Unix-Dateisystems

Aus der Sicht des Anwenders (logische Sicht) besitzt das Dateisystem einen baumähnlichen Aufbau siehe Abbildung 55.

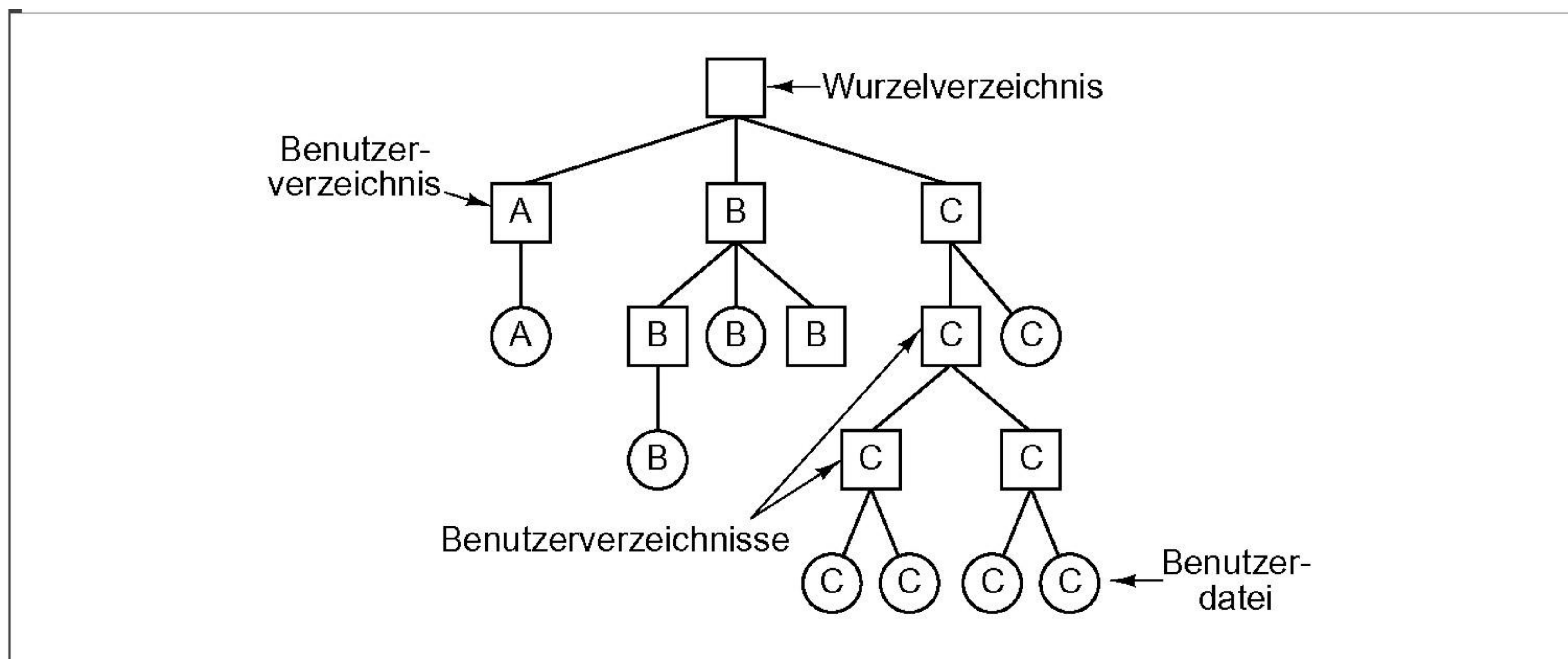


Abbildung 55.: hierarchisches Unix-Dateisystem

Diese hierarchische Struktur, auch Dateibaum genannt, besteht aus fünf unterschiedlichen Komponententypen

1. **normale Dateien** (ordinary file) entsprechen den üblichen Dateien in anderen Betriebssystemen,
2. **Gerätedateien** (device file) enthalten Verweise zu Peripheriegeräten der Systemarchitektur (Terminals, Drucker, usw.). Sie stellen aus Anwendersicht die Geräte dar und dienen als Schnittstelle zu diesen,
3. **Verzeichnisse** (directory) kann man als "strukturierende" Dateien ansehen, in denen Einträge von "Dateien" enthalten sind,
4. **benannte Pipes** (FIFO) erweitern die Möglichkeiten unter Unix im Umgang mit **Pipes,**
5. **symbolische Links** (symbolic link) werden verwendet, um auf andere Komponenten zuzugreifen, die zeitweise nicht verfügbar sind bzw. Um eine Dateistruktur zu erzeugen, die nicht der Realität entspricht

Die Wurzel des Dateibaums stellt das **Wurzelverzeichnis** dar (**/** bzw. **root**) dar. Jede Komponente des Dateibaums wird durch einen Namen gekennzeichnet, die Länge des Dateinamens kann 14 bzw. **255** Zeichen beinhalten. Dabei wird von Unix zwischen Groß- bzw. Kleinschreibweise unterschieden. Innerhalb eines Verzeichnisses müssen die Dateinamen eindeutig sein.

Der Dateibaum kann sich aus mehreren Teilbäumen (**physischen Dateisystemen**) zusammensetzen. Die Lage dieser Teilbäume, innerhalb oder außerhalb des Rechnersystems, ist dabei von untergeordneter Bedeutung. Laufwerksnamen, die in anderen Betriebssystemen geläufig sind, gibt es in Unix nicht.

Das Zusammenbinden mehrerer Teilbäume zu einen Dateibaum nennt man montieren (**mount**) und ist in Abbildung 56 dargestellt.

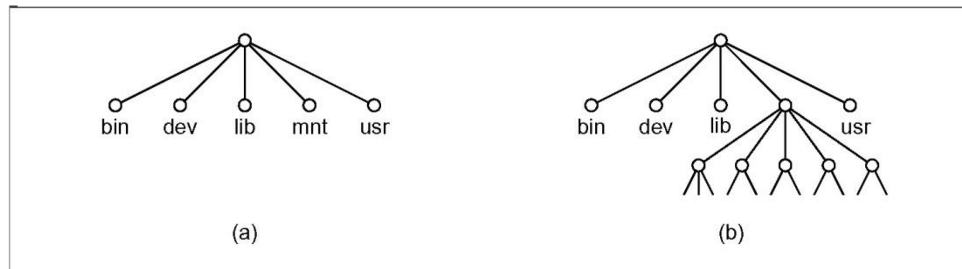


Abbildung 56.: Montieren von physischen Dateisystemen

5.4.2. Bewegen im Dateibaum

Nachdem die logische Struktur des Dateibaums vorgestellt wurde, stellt sich die Frage nach der Positionierung im Dateibaum. Dazu müssen folgende Begriffe geklärt werden:

- Aufbau eines Dateiverzeichnisses
- Anmeldeverzeichnis (**home directory**) und
- Arbeitsverzeichnis (**working directory**).

Ausgehend von diesen Erkenntnissen lassen sich zwei Formen von Pfadnamen unterscheiden:

- **absoluter Pfadname** und
- **relativer Pfadname**.

Die Positionierung im Dateibaum (bewegen des working directory) wird durch folgende Befehle übernommen:

- **cd** change directory und
- **pwd** print working directory.

5.4.3. Dateiattribute, Benutzerklassen und Zugriffsrechte

Für jede Komponente des Dateibaums verwaltet Unix intern eine Reihe von Attributen wie **Eigentümer**, **Zugriffsrechte**, **Modifikationsdaten**, usw.

Das Betriebssystem Unix kennt nur einen Benutzer mit besonderen Privilegien, den Systemverwalter mit dem Benutzernamen **root**, dem die Verwaltung des gesamten Rechnerbetriebs obliegt. Alle anderen Benutzer haben generell denselben Status.

Der Systemverwalter fasst die einzelnen Anwender zusätzlich zu Gruppen zusammen. Dazu erhält jeder Unix-Anwender eine eindeutige Benutzeridentifikation zugeteilt. Diese besteht aus einer Benutzernummer (**UID**) und Gruppennummer (**GID**).

Bezüglich aller Komponenten im Dateibaum verfährt Unix nach folgenden Regeln:

1. Jede Komponente des Dateibaums gehört zu einem Benutzer und zu einer Gruppe von Benutzern.
2. Jeder, der berechtigt ist, sich beim System anzumelden, erhält eine Benutzeridentifikation (**UID**, **GID**).
3. Wird eine Komponente innerhalb des Dateibaums erzeugt, wird diese mit der **UID** und **GID** des Erzeugers markiert.

Betrachtet man weiterhin alle Komponenten im Dateibaum als **Objekte** und die Anwender des Systems als **Subjekte**, dann erfolgt bei jedem Zugriff im Betriebssystemkern ein **Subjekt-Objekt-Abgleich**. Dabei unterscheidet Unix folgende **Benutzerklassen**:

Benutzer → **user**
Gruppenmitglied → **group**
Andere → **other**

Weiterhin verwaltet das Betriebssystem Unix für jede Komponente je Benutzerklasse drei Zugriffsrechte:

lesen → **read**
schreiben → **write**
ausführen → **execute**

Somit ergeben sich folgende **9** Zugriffsrechte, die das Betriebssystem Unix für jede Komponente des Dateibaums verwaltet.

Zur Verwaltung der Benutzerklassen und Zugriffsrechte stehen dem Anwender folgende Befehle zur Verfügung:

- **ls** list
- **chmod** change mode
- **umask** update mask
- **chown** change owner
- **chgrp** change group

5.5. Physisches Dateisystem

Aus Anwendersicht erscheint der Unix-Dateibaum als eine homogene Struktur. Auf physischer Ebene setzt er sich jedoch aus mehreren Teilbereichen zusammen, den so genannten physischen Dateisystemen.

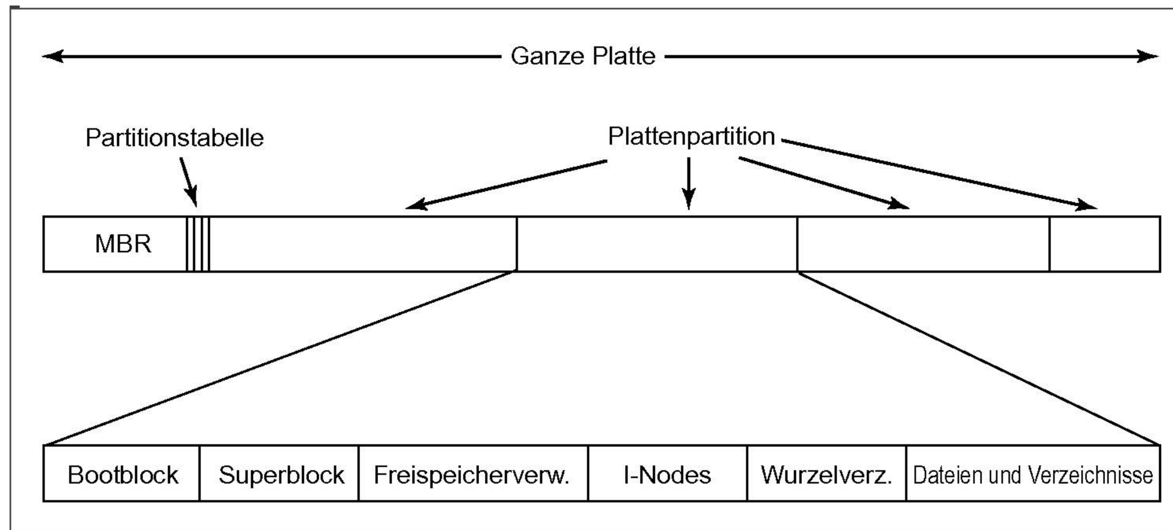


Abbildung 57.: Unterteilung einer Festplatte in Partitionen

5.5.1. Aufbau des physischen Dateisystems

Ein physisches Dateisystem ist eine dateiorientierte Struktur auf einem logischen Datenträger (**Partition**, **Slice**). Unix bietet die Möglichkeit, mehrere dieser logischen Datenträger auf einem physischen Datenträger (**Festplattenlaufwerk**) zu verwalten. Eine mögliche Unterteilung ist in Abbildung 57 dargestellt.

Des Weiteren können neben den resistenten physischen Dateisystemen (Dateisysteme auf dem Systemlaufwerk) auch nicht resistente Dateisysteme auf montierbaren Datenträgern (Festplattenlaufwerke anderer Rechnersysteme, Disketten usw.) in den Dateibaum eingehängt bzw. entfernt werden. Dadurch wird ein hoher Grad an Flexibilität in der Systemarchitektur erreicht, der dem Anwender weitestgehend verborgen bleibt.

Schaut man sich ein physisches Dateisystem genauer an, so erkennt man den **Betriebssystemblock** als kleinste Einheit (im Bereich von **512 Byte** bis **16 KByte**). Das Betriebssystem legt nun über diese (durchnummerierte) Folge von **Blöcken** ein höheres Ordnungsschema, wodurch ein physisches Dateisystem in **vier Bereiche** mit unterschiedlicher Funktionalität aufgespaltet wird:

1. **Boot-Block**
   Er enthielt im **root-Dateisystem** einen Urlader-Programm, das das eigentliche Unix-System (Betriebssystemkern) in den Arbeitsspeicher lädt. Dieser Bereich ist bei heutigen Systemen leer.

2. **Super-Block**
   Er beinhaltet alle relevanten Verwaltungsinformationen zu einem physischen Dateisystem:
   - Name und Größe des physischen Dateisystems
   - Größe der nachfolgenden Bereiche des physischen Dateisystems (Inodeliste, Nutzdatenbereich)
   - Verweise auf die Freiblocklisten (Liste der freien Datenblöcke und Liste der freien Inodes)
   - Datum der letzten Sicherung und Modifikation
   - und weitere Angaben.

3. **Inodeliste**
   Sie stellt ein Inhaltsverzeichnis aller in dem Dateisystem existierenden Dateien dar. Sie besteht aus einer Folge von Inodes (Dateiköpfen), die die Verwaltungsdaten zu jeder Datei enthalten.

4. **Nutzdatenbereich**
   Hier befinden sich die freien Blöcke, Datenblöcke (Inhalte von Dateien und Verzeichnissen) und Referenzblöcke, die zur Adressierung der Datenblöcke eingesetzt werden.

Die Größe der einzelnen Bereiche wird bei der Initialisierung eines physischen Dateisystems spezifiziert und kann im Nachhinein nicht mehr dynamisch verändert werden (außer im Unix-System **AIX** von **IBM**).

Das dazu notwendige Kommando des Systemadministrators ist **mkfs** make file **s**ystem.

Die Bereiche eines physischen Dateisystems sind auf die Kapazität eines logisch Datenträgers und damit maximal auf die Gesamtkapazität eines Festplattenlaufwerks beschränkt, d.h. festplattenübergreifende physische Dateisysteme sind nicht möglich.

5.5.2. Aufbau eines Inodes

Die Inodeliste besteht aus einer Folge von Inodes, die durch einen eindeutigen Index, der Knotennummer, gekennzeichnet werden.

Da alle Inodes die identische Länge von 128 Byte haben, erübrigt sich die Abspeicherung der Knotennummer innerhalb des Inodes.

Ein Inode enthält die relevanten Verwaltungsattribute einer Datei:

1. Dateityp und Zugriffsrechte
2. Referenzzähler (**Linkcounter**)
3. Benutzernummer (**UID**) und Gruppennummer (**GID**)
4. Dateigröße in Byte
5. Zeitstempel
    - Erstellungsdatum
    - Datum der letzten Modifikation
    - Datum des letzten Zugriffs
6. Blockadressen der ersten zehn Datenblöcke
7. Blockadressen der Indirektionsblöcke

Zu beachten ist, dass der Name der Datei nicht aufgeführt wird.

Diese Systemarchitektur ermöglicht es, unter mehreren (verschiedenen) Namen als Einträge von Verzeichnissen die gleiche Datei (genauer gesagt Inode und Dateiinhalt) anzusprechen.

Der Inode fungiert somit als Bindeglied zwischen dem Namen und dem Inhalt einer Datei. Für die Adressierung der Inhalte einer Datei ergibt sich eine bestimmte Verweisstruktur.

Unter der Voraussetzung, dass eine Blockgröße von 512 Byte betrachtet wird, ergibt sich eine mögliche Dateistruktur und -größe, wie sie in Abbildung 58 dargestellt ist und liegt bei 2.113.673 Blöcken.

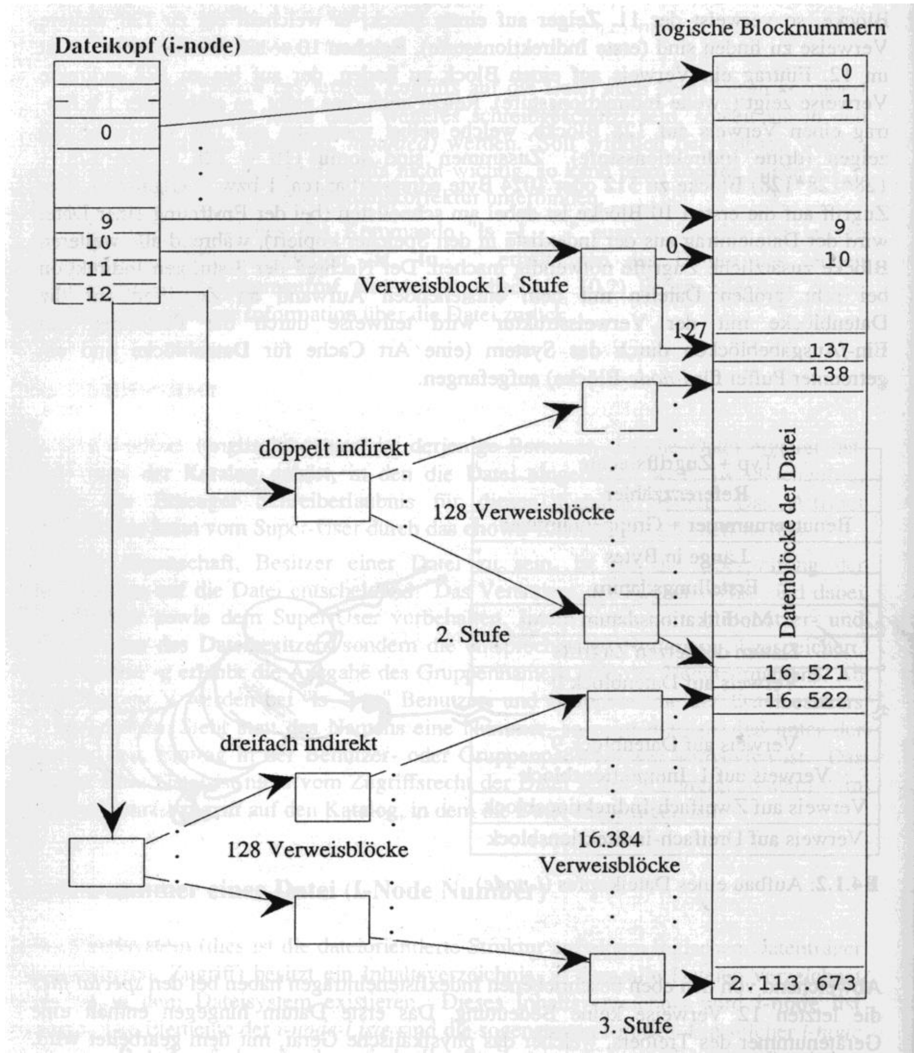
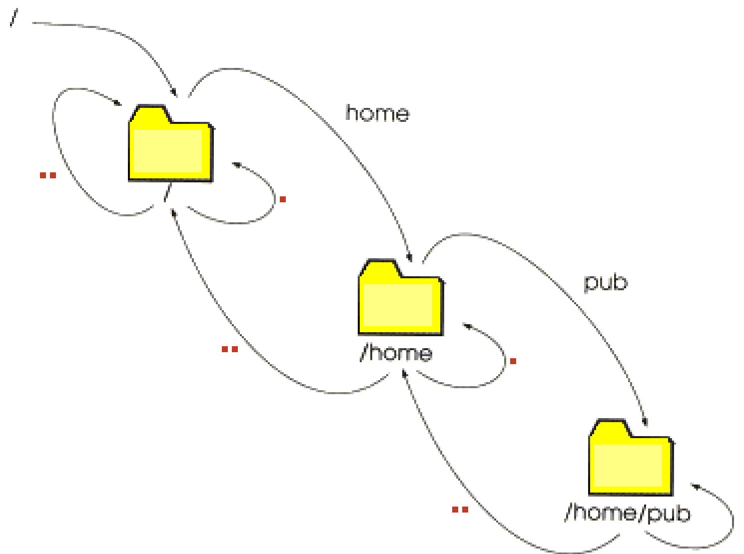


Abbildung 58.: Inode-Verweisstruktur



5.5.3. Aufbau eines Verzeichnisses

Jedes Verzeichnis, das wie eine andere Dateiart einen eindeutigen Inode besitzt, hat zwei obligatorische Einträge. Einen Eintrag (**.**) als Verweis auf den eigenen Eintrag und einen weiteren Eintrag (**..**) auf den Inode des Vorgängerverzeichnisses.

Zusätzlich können weitere Einträge (Namen von Dateien und zugehörige Inodenummern) als Verzeichnisinhalt aufgeführt sein.

5.5.4. Dateiverwaltungskommandos

Mittels folgender Kommandos kann auf die Dateistruktur zugegriffen werden:

- **cp** copy
- **mv** move
- **ln** link
- **rm** remove
- **mkdir** make directory
- **rmdir** remove directory

5.5.5. Gerätedateien

Gerätedateien dienen zur Abwicklung des Datenverkehrs zwischen den Programmen und der Peripherie. Da sie, wie die abstrakten Komponenten (normale Dateien, Verzeichnisse, usw.) einheitlich in den Systembaum integriert werden, sind auch für sie die Zugriffsschutzmechanismen und Ein- /Ausgabeumleitung gültig.

Die Geräte werden durch logische Namen angesprochen, die die Gerätetreiber der Peripheriegeräte bezeichnen. Intern wird jedes Gerät durch eine Treiber-Nummer (**major device number**) und eine Geräte-Nummer (**minor device number**) markiert. Diese Nummern sind im **ls -l** Eintrag in dem Feld enthalten, das die Dateigröße angibt. Die Gerätedateien müssen im Geräteverzeichnis **/dev** abgelegt sein.

Gerätedateien arbeiten entweder blockorientiert (**blockdevice - b**) mit einem Betriebssystemkern-Puffer oder zeichenorientiert (**characterdevice - c**) ohne Betriebssystemkern-Puffer im so genannten "rohen" Modus (**raw mode - raw device**). Manche Geräte (Festplatten, Disketten usw.) lassen beide Arbeitsweisen zu. Den logischen Namen der Geräte wird in der ungepufferten Notationsform der Präfix **r** vorangestellt:

- **/dev/fd0135ds18** – blockorientiert
- **/dev/rfd0135ds18** - zeichenorientiert

Aufbau des logischen Namens: [r]<**treibername**><**lw**><**tpi**><**seiten**><**spt**>
mit : <**treibername**> **fd**, **hd**, ...
<**lw**> 0, 1, ...
<**tpi**> tracks per inch
<**seiten**> ss, ds
<**spt**> sectors per track

als Beispiel für die Microsoft-Notation für Diskettenlaufwerke.

Die Vergabe von logischen Namen für die Geräte ist nicht einheitlich und wird von jedem Betriebssystemanbieter in eigener Regie vorgenommen. Jedoch sind mehrere Namen (durch ***Links***) möglich.

Das Pseudogerät **/dev/null** fungiert als "unendlicher Abfalleimer".

5.6. Prozesssystem

5.6.1. Prozesshierarchie

Jeder Prozess, der im Betriebssystemkern aktiviert wird, erhält eine eindeutige Kennzeichnung, die so genannte Prozessnummer (**PID**). Über diese kann jeder Prozess eindeutig identifiziert werden.

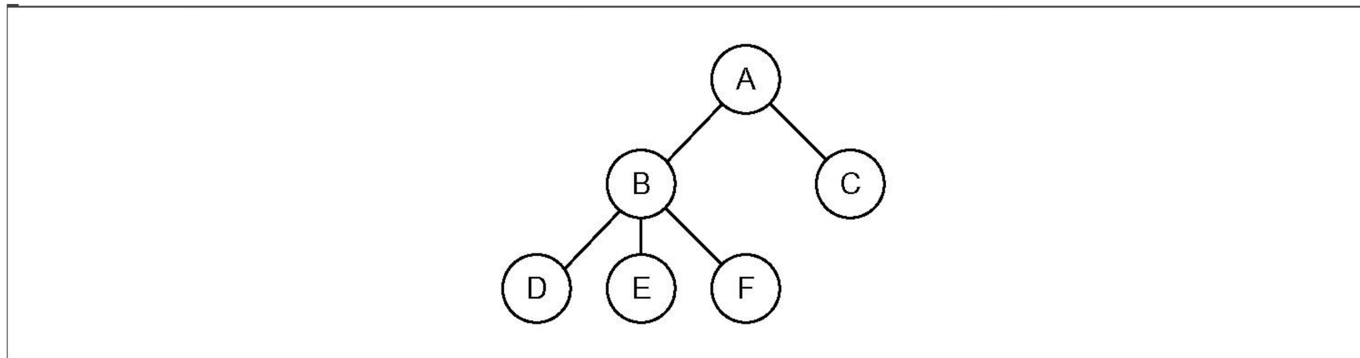


Abbildung 59.: Prozesshierarchie

Ein neuer Prozess kann nur von einem bereits laufenden Prozess erzeugt werden, d.h. innerhalb des laufenden Programms wird ein Kommando abgearbeitet, das die Erzeugung eines neuen Prozesses verursacht.

Dadurch werden, ähnlich wie beim Dateibaum die einzelnen Prozesse im Betriebssystemkern in einer baumartigen Struktur verwaltet. D.h., jeder **Sohn-Prozess** ist genau einem **Vater-Prozess** untergeordnet. Die Wurzel der Prozessstruktur wird durch den Systemstart geschaffen und als **init-Prozeß** (**PID = 1, /sbin/init**) bezeichnet.

5.6.2. Hintergrundprozesse

In der Regel wartet der Vater-Prozess auf die Beendigung seiner Sohn-Prozesse. Diese Art der Prozesssynchronisation wird als **synchrone Ausführung** bezeichnet, der Sohn-Prozess wird als **Vordergrundprozess** ausgeführt.

Diese Arbeitsweise wird in der Regel angewandt. Bezogen auf einen Benutzer ist die Shell (**Login-Shell**) der Vater-Prozess. Alle Kommandos, die der Benutzer startet, sind Sohn-Prozesse. Während diese abgearbeitet werden ist der Vater-Prozess terminiert.

Als **asynchroner Prozess** oder **Hintergrundprozess** werden solche Prozesse bezeichnet, bei denen der erzeugende Vater-Prozess nicht auf das Ende seines Sohn-Prozesses wartet, sondern parallel (**quasiparallel** auf einer Ein-Prozessor-Maschine) asynchron weiterläuft.

Auf der Shell-Ebene kann jeder Prozess durch Anfügen von **&** (**Ampersand-Zeichen**) in der Kommandozeile als Hintergrundprozess gestartet werden:

<kommando> - synchrone Ausführung
<kommando> & - asynchrone Ausführung

5.6.3. Prozesspriorität

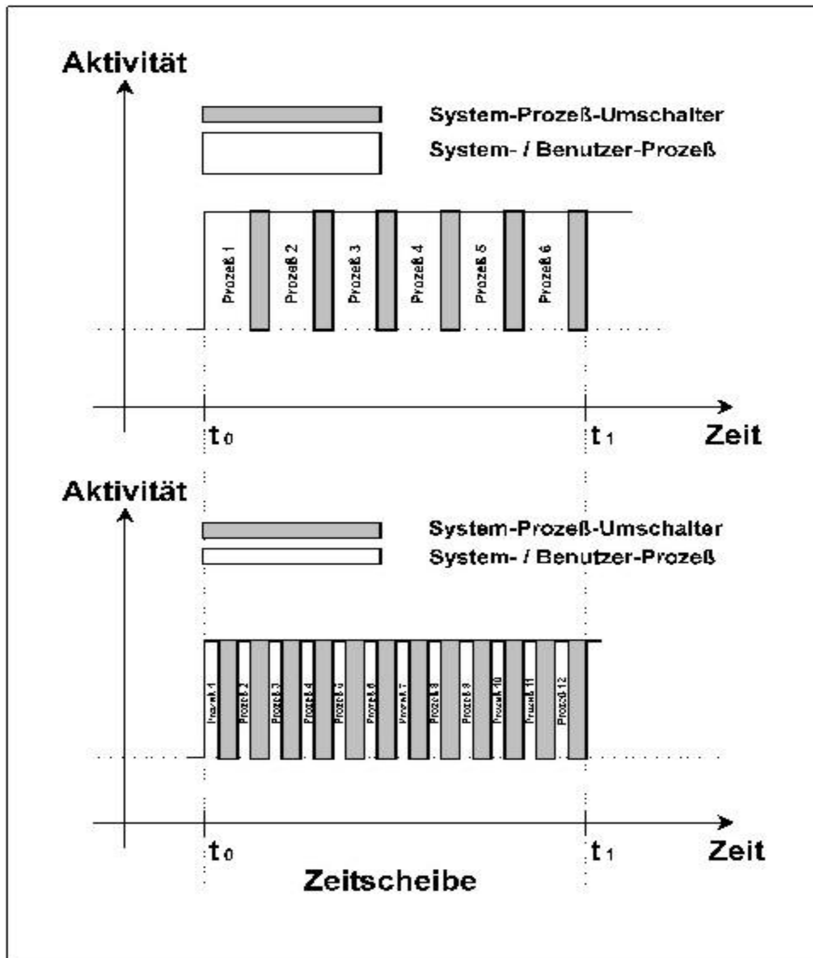


Abbildung 60.: Unix-Timesharing-Prinzip

Unix ist ein Multi-Task-Betriebssystem, d.h. mehrere Prozesse eines oder mehrerer Benutzer konkurrieren um die Vergabe der Rechenzeit des Prozessors. Wie viele andere Systeme auch, arbeitet Unix nach dem **Timesharing-Prinzip**, siehe Abbildung 60.

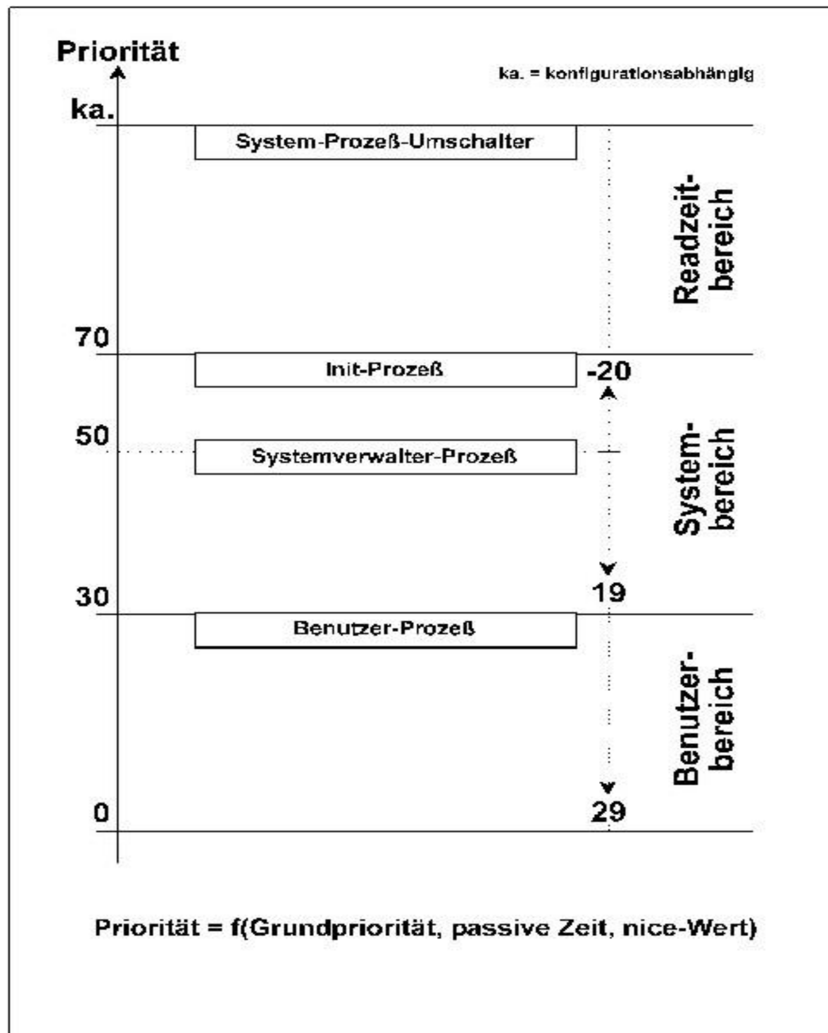


Abbildung 61.: Prozesspriorität

Über einen **Schedulingalgorithmus** zur Berechnung der Priorität erhält jeder einzelne Prozess einen bestimmten Teil der Rechenzeit zugewiesen. D.h. der Prozess mit der zurzeit höchsten Priorität erhält die CPU, wird nach einen Zeitintervall suspendiert und, falls noch nicht beendet, zu einem späteren Zeitpunkt wieder reaktiviert.

Die aktuelle Priorität eines Prozesses setzt sich aus dem Produkt des CPU-Faktors und der Grundpriorität zusammen. Die Einteilung der einzelnen Prozesse in Grundprioritäten zeigt Abbildung 61.

5.6.4. Prozesszustände

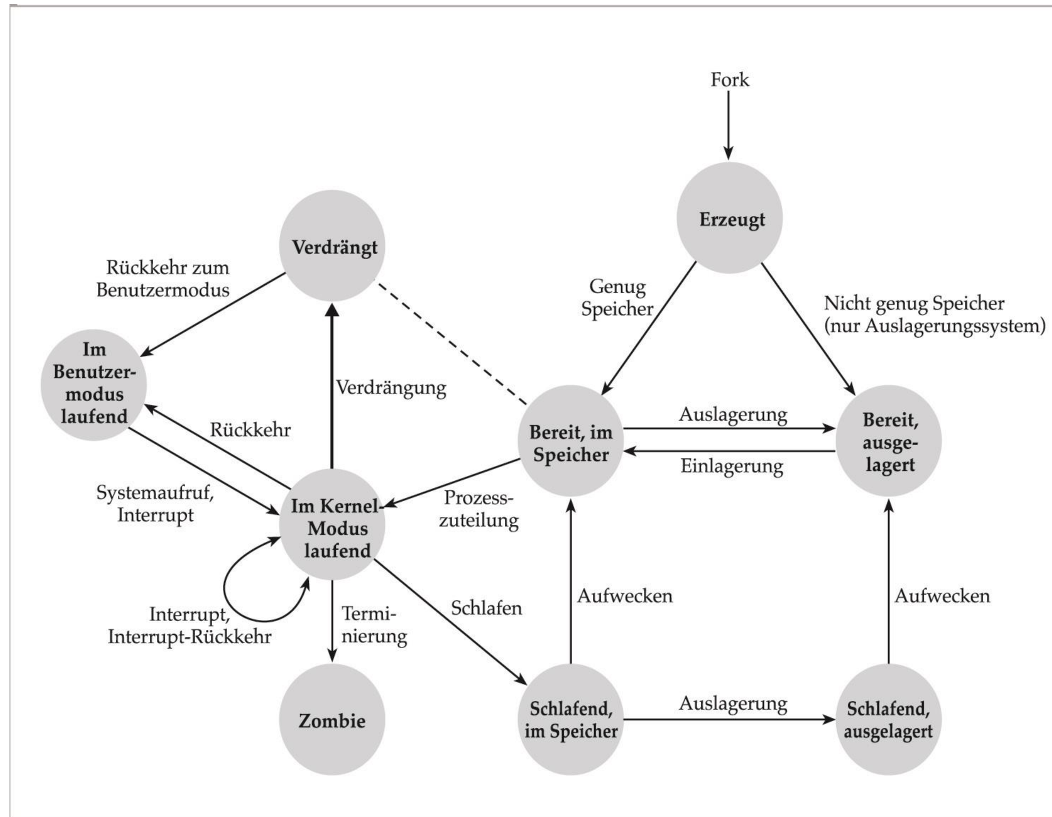


Abbildung 62.: Unix-Taskmodell

Obwohl jeder Prozess eine unabhängige Einheit mit eigenem Befehlszähler und innerem Zustand darstellt, müssen Prozesse häufig mit anderen Prozessen kommunizieren. In Abschnitt 5.7.3. Kommandoverkettung mit der Pipe ist dafür ein klassisches Beispiel angegeben. In einem solchen Fall ist nicht nur die Prozesspriorität für die Bearbeitung entscheidend, sondern auch die Beziehung zwischen den Prozessen. Grundlage für die gesamte Betrachtung ist das so genannte Taskmodell, das hinter der Shedulingalgoritmus steht.

In Abbildung 62 ist dazu das Taskmodell von Unix abgebildet.

5.6.5. Kommandos zum Prozesssystem

Folgende Kommandos beziehen sich auf das Prozesssystem:

- **ps** process status
- **kill**
- **jobs**
- **nice**
- **nohup**
- **sleep**

5.7. Die Unix-Kommandos als Filter

5.7.1. Die Bourne-Shell als Kommandointerpreter

Die Schnittstelle zwischen den Benutzern und dem Betriebssystem stellt die Bourne-Shell (genannt nach ihrem Entwickler Stephen R. Bourne) dar. Dieses ist ein interaktiver Kommandointerpreter und wird in der Regel nach dem Anmeldevorgang als so genanntes Login-Programm gestartet.

Andere Login-Programme können die **C-Shell csh**, die **Korn-Shell ksh** oder die **GNU Bourne Again Shell bash** sein.

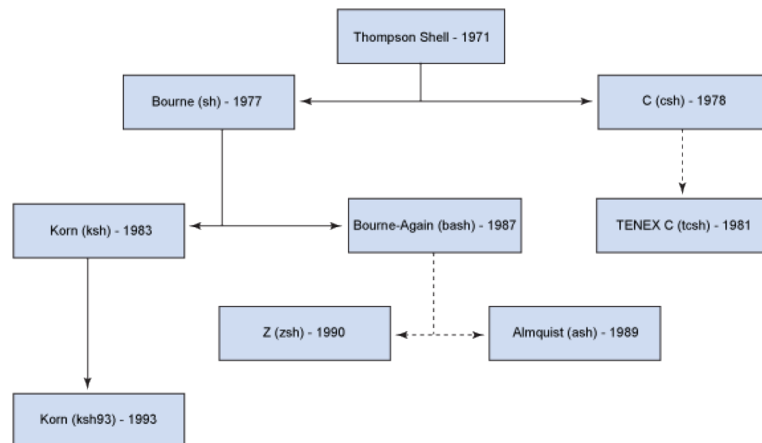


Abbildung 63.: Historie der Unix-Shells

Wie sieht nun der Anmeldevorgang im Einzelnen aus?

Nach Überprüfung des Benutzernamens und dessen Passworts werden für alle Benutzer, die mit der Bourne-Shell, oder der Korn-Shell arbeiten, die Kommandos der Datei **/etc/profile** ausgeführt, die in der Regel Initialisierungen und weitere Verwaltungsaufgaben beinhalten. Diese Datei wird vom Systemadministrator verwaltet und kann z.B. folgende Aktivitäten beinhalten:

- Ausgabe des Einschaltreports **/etc/motd**
- Starten des Postsystems **mail**
- u.s.w.

Das Betriebssystem verzweigt nun in das HOME-Verzeichnis und führt, falls vorhanden, die benutzereigene Initialisierungsdatei **.profile** aus.
Die für den Anmeldevorgang nötigen Verwaltungsdaten werden zentral für alle Benutzer in der Systemverwaltungsdatei **/etc/passwd** abgelegt. Jede Informationszeile in dieser Datei enthält 7 Einträge, die durch **Doppelpunkte** getrennt sind:

- Benutzername (max. 8 Zeichen)
- verschlüsseltes Passwort (bis System V Version 3.1)
- Benutzernummer (UID)
- Gruppennummer (GID)
- Kommentarfeld (oft leer)
- Login-Verzeichnis (HOME-Verzeichnis)
- Login-Programm (falls leer => Bourne-Shell)

In der Datei **/etc/group** sind zusätzlich die einzelnen Gruppen (GID) mit den jeweiligen Gruppenmitgliedern aufgelistet.

Ab System V Version 3.2 existiert die Datei **/etc/shadow**, die die Passwörter enthält und nur für den Systemadministrator lesbar ist. Die Datei **/etc/passwd** enthält dann stets den Buchstaben **x** als Eintrag 2.

Die Bereitschaft zur Kommandoeingabe wird durch die Zeichen **$** für normale Benutzer und **#** für Systemadministratoren dargestellt.

Die nach dem Anmelden in der Regel laufende Shell (damit auch die Rechnersitzung) wird mit dem Dateiendezeichen (**EOF**) **CTRL-d** oder dem Kommando **exit** beendet.

5.7.2. Ein- /Ausgabeumlenkung

Das Unix-Betriebssystem ist als Dialogsystem konzipiert, d.h., die Kommandos werden über die Tastatur eingegeben und erscheinen, wie auch die Kommandoausgaben auf dem Bildschirm.

Intern erfolgt der Datenfluss immer über Dateien in Verbindung mit so genannten Dateidiskriptoren.

ls -l > listing
ls -l /bin >> listing
wc -l < listing (**< listing wc -l**)
cat /etc/passwd >passwd 2>/dev/null
sort Datei_1 >Datei_1

5.7.3. Kommandoverkettung mit der Pipe

Oft tritt der Fall ein, dass ein Kommando seine Ausgaben in eine Datei umgelenkt bekommt, und ein zweites Programm benutzt diese Angaben, gelesen aus der Datei, für seine weitere Arbeit. Diese etwas umständliche Situation lässt sich auf eine elegante und einfache Art durch eine **Pipe** (Leitung, Röhre) lösen.

Verbindet man beide Kommandos mit dem Zeichen **|**, so wird die Standardausgabe des ersten Kommandos auf die Standardeingabe des zweiten Kommandos umgeleitet.

```
who > /tmp/temporary
wc -l < /tmp/temporary
rm /tmp/temporary
who | wc -l
```

Intern wird die Pipe durch eine **FIFO** (**f**irst **i**n **f**irst **o**ut) bzw. einen Ringpuffer organisiert.

5.7.4. weitere wichtige Kommandos

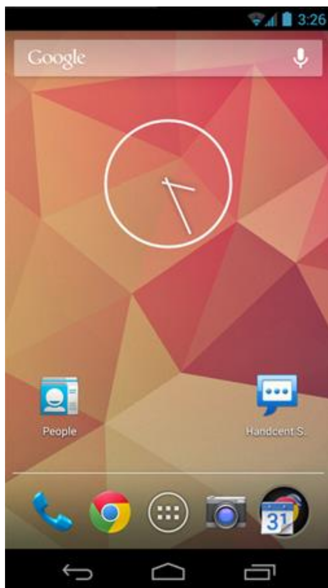
Weitere Kommandos, die in diesem Zusammenhang von Bedeutung sein können:

man manual
more
lp line printer
find.

6. Betriebssysteme für Smartphones und Tablets (Wikipedia)

In Kapitel 1.4.6. Betriebssysteme für Smartphones und Tablets wurde bereits die Bedeutung von diesen Geräten, sowie die dazu entwickelten Betriebssysteme kurz vorgestellt. Anhand der in Tabelle 1 dargestellten Marktanteile ist es z.Z. ratsam, sich mit den beiden Marktführern in diesem Gerätesegment zu beschäftigen. Vorausbetrachtend soll darauf verwiesen werden, dass es sich bei den beiden Betriebssystemen Android und iOS um Systeme handelt, deren Basis eine Linux-Distibution darstellt. Grundlegende Informationen zu Unix bzw. Linux wurden ja bereits in Kapitel 5. Das Betriebssystem Unix behandelt.

6.1. Android



Android (von englisch *android* / von griechisch *androïdes* menschenähnlich) ist sowohl ein Betriebssystem als auch eine Software-Plattform für mobile Geräte wie Smartphones, Mobiltelefone, Netbooks und Tablet-Computer, die von der Open Handset Alliance (Hauptmitglied: Google Inc.) entwickelt wird. Basis ist der Linux-Kernel. Bei Android handelt es sich um freie Software, die quelloffen entwickelt wird.

Android hatte als Smartphone-Betriebssystem im zweiten Quartal 2013 einen weltweiten Marktanteil von 79,3 Prozent nach 68,1 Prozent im zweiten Quartal 2012, 52,5 Prozent im dritten Quartal 2011 und 25,5 Prozent im dritten Quartal 2010.

Im April 2013 verkündete Eric Schmidt (**Eric Emerson Schmidt** (* 27. April 1955 in Washington, D.C.) ist ein US-amerikanischer Informatiker und Manager), dass pro Tag 1,5 Millionen neue Android-Geräte dazu kommen und im September 2013 waren über einen Milliarde Android-Geräte weltweit aktiviert.

Abbildung 64.: Android 4.3 (Jelly Bean) auf dem Samsung Galaxy Nexus

6.1.1. Die Geschichte von Android

Im Sommer 2005 kaufte Google das im Herbst 2003 von Andy Rubin gegründete Unternehmen Android, von dem nur wenig mehr bekannt war, als dass es Software für Mobiltelefone entwickelte und standortbezogene Dienste favorisierte. Ursprünglich war Android ausschließlich für Digitalkameras gedacht. Am 5. November 2007 gab Google bekannt, gemeinsam mit 33 anderen Mitgliedern der Open Handset Alliance ein Mobiltelefon-Betriebssystem namens Android zu entwickeln. Seit dem 21. Oktober 2008 ist Android offiziell verfügbar.

Als erstes Gerät mit Android als Betriebssystem kam am 22. Oktober 2008 das HTC Dream unter dem Namen *T-Mobile G1* in den Vereinigten Staaten auf den Markt. Dass bereits dieses erste Gerät auf das Global Positioning System (GPS) zugreifen konnte und mit Bewegungssensoren ausgestattet war, gehörte zum Konzept von Android. Inzwischen gibt es eine große Anzahl unterschiedlicher Geräte von diversen Herstellern, auf denen Android vorinstalliert ist. Seit Januar 2010 bringt Google mit der Nexus-Produktereihe auch selbst Android-Geräte auf den Markt.

6.1.2. Die Architektur von Android

Die Architektur von Android baute anfangs auf dem Linux-Kernel 2.6 auf, ab Android 4.x wird ein Kernel der 3.x-Serie verwendet. Er ist für die Speicherverwaltung und Prozessverwaltung zuständig und stellt die Schnittstelle zum Abspielen von Multimedia und der Netzwerkkommunikation dar. Außerdem bildet er die Hardwareabstraktionsschicht für den Rest der Software und stellt die Gerätetreiber für das System.

Weitere wichtige Bausteine sind die auf der von Sun Microsystems entwickelten Java-Technik basierende virtuelle Maschine Dalvik und die dazugehörigen Android-Java-Klassenbibliotheken. Die Inhalte der Klassenbibliothek orientieren sich stark an der Funktionalität der Java-Standard-Edition.

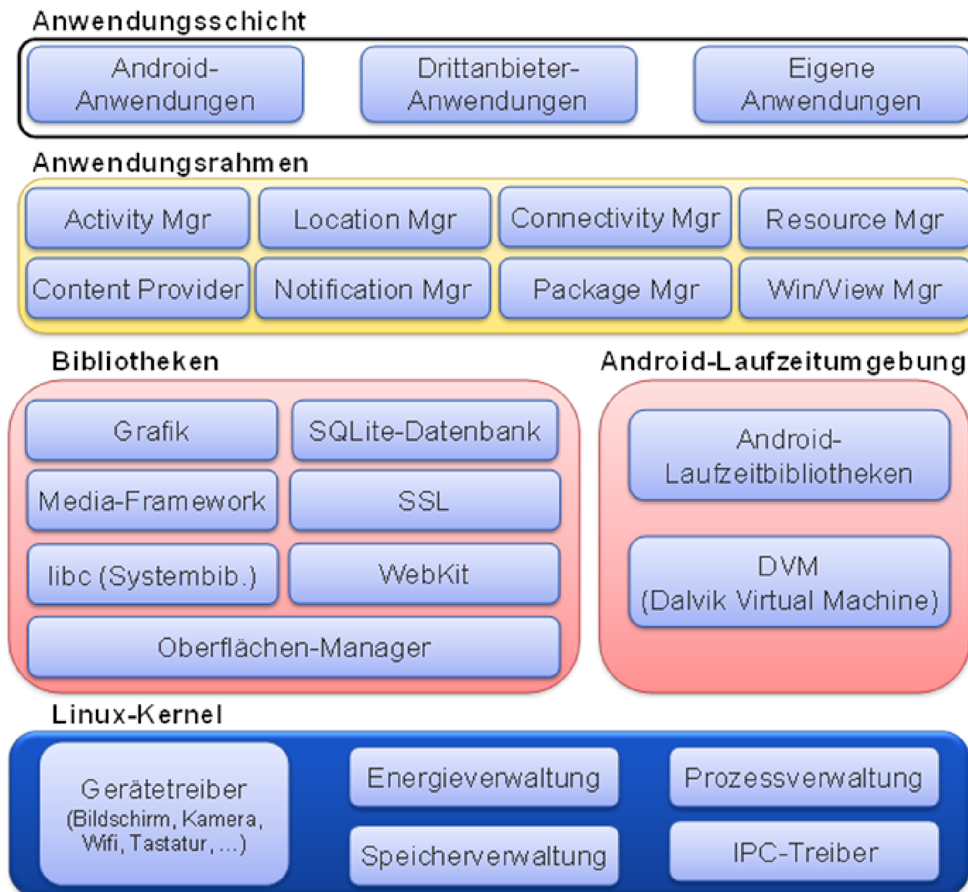


Abbildung 65.: Die grundsätzliche Architektur von Android

Dabei wurde als Grundlage die freie Reimplementierung der Java-Standard-Edition Apache Harmony verwendet.

Die Laufzeitumgebung von Android basiert auf der Dalvik Virtual Machine, einer von Google-Mitarbeiter Dan Bornstein entwickelten virtuellen Maschine. Die Dalvik-VM ähnelt funktional der normalen Java-VM, beide führen sogenannten Byte-Code aus.

Einer der wesentlichen Unterschiede ist die zugrundeliegende virtuelle Prozessorarchitektur. Die Java-VM basiert auf einem Kellerautomaten; Dalvik-VM hingegen ist eine Registermaschine. Durch die sich unterscheidende Prozessorarchitektur sind die Kompilate normaler Java-Compiler nicht für die Dalvik-VM geeignet, dennoch konnte Google auf die bestehenden Java-Entwicklungswerkzeuge zurückgreifen.

Die meisten modernen Compiler generieren als Zwischencode Kellerautomatencode. Dieser Zwischencode erlaubt es, von der Prozessorarchitektur der Zielplattform zu abstrahieren, der programmiersprachliche Teil wird

von der konkreten Prozessorarchitektur getrennt. Da das Prozessormodell des Kellerautomaten besonders einfach ist, wird es üblicherweise für die Übersetzerzwischensprache verwendet.

Die meisten realen Prozessoren sind heute aber Registermaschinen, so zum Beispiel die 80x86- und die ARM-Prozessoren. Registerarchitekturen sind oft effizienter, da bei ihr die CPU über eigene besonders schnell zugreifbare Speicherzellen – die Register – verfügt. Dalvik nimmt die Umwandlung des Kellerautomatencodes in die Registermaschinencodes schon zur Übersetzungszeit vorweg. Dafür wird das Werkzeug dx verwendet, *„dx“* steht für Dalvik Cross-Assembler.

Anwendungen für die Androidplattform werden in der Regel in Java geschrieben, jedoch greifen diese in geschwindigkeitskritischen Bereichen auf zahlreiche in C oder C++ geschriebene native Bibliotheken zu. Darunter befinden sich neben Codecs für die Medienwiedergabe auch ein Webbrowser auf der Basis von WebKit, eine Datenbank (SQLite) und eine auf OpenGL basierende 3D-Grafikbibliothek.
Um eigene Programme für Android zu entwickeln, benötigt man ein aktuelles Java-SDK und zusätzlich das Android-SDK. Zuerst wird der in Java geschriebene Quelltext mit einem normalen Java-Compiler übersetzt und dann von einem Cross-Assembler für die Dalvik-VM angepasst. Aus diesem Grund können Programme prinzipiell mit jeder Java-Entwicklungsumgebung erstellt werden.

Die fertige Anwendung muss in ein .apk-Paket („Android Package“) verpackt werden, anschließend kann sie über einen Anwendungs-Shop oder direkt auf dem Gerät installiert werden, via Paketmanager.

Das Framework setzt auf starke Modularität. So sind alle Komponenten des Systems generell gleichberechtigt (ausgenommen die virtuelle Maschine und das unterliegende Kernsystem) und können jederzeit ausgetauscht werden. Es ist also beispielsweise möglich, eine eigene Anwendung zum Erstellen von Kurznachrichten oder zum Wählen von Rufnummern zu erstellen und die bisherige Anwendung damit zu ersetzen.

Einen weiteren Anwendungsentwicklungs- und Portierungsweg jenseits von Java bietet die SDL-Bibliothek für SDL- und nativen C-Code an. Über einen kleinen java-basierten Wrappercodeanteil wird über das Java Native Interface (JNI) das Verwenden von nativem Code möglich gemacht.

Da Android jedoch weder ein natives X Window System bietet noch den vollen Umfang der GNU-Bibliotheken standardmäßig umfasst, ist eine Portierung vorhandener (Desktop-) Linuxanwendungen oder Bibliotheken schwierig.

6.1.3. Bezug von Software (Apps) für Android

Im Google Play Store (ehemals *Android Market*) gibt es 700.000 Anwendungen (Stand: November 2012). Damit verfügt Googles Play Store mit Stand 1. November 2012 über die gleiche App-Anzahl wie der bislang marktführende App-Anbieter Apple. Im SDK werden zusätzlich eine Reihe von Anwendungen, darunter ein Webbrowser, die Kartenanwendung Google Maps, eine SMS-, E-Mail- und Adressbuchverwaltung, ein Musikprogramm, eine Kamera- und Galerieapplikation, sowie ein Satz von API-Demoanwendungen mitgeliefert.

Erstellte Software kann von den Entwicklern bei Google Play angeboten werden. Verkaufen kann man sie dort allerdings nur, wenn man in bestimmten Staaten ansässig ist, aufgeführt sind unter anderem Deutschland, Österreich sowie die Schweiz. Kostenfreie Software macht etwa 69 Prozent aus.

Neben Google Play stehen Entwicklern und Endanwendern auch noch eine Reihe anderer Märkte und Plattformen für Android-Software offen; manche vermeintlich eigenständige verweisen jedoch wiederum auf Google Play. Google behält eine gewisse Kontrolle über Android Software. Nur lizenzierte Android-Distributionen dürfen die Google-eigenen (closed-source) Anwendungen wie Google Mail oder Google Maps verwenden sowie auf den Google Play Store für weitere Applikationen zugreifen.

Verschiedene Tablets verwenden ein unlizenziertes Android 4 und haben keine Berechtigung für den Zugriff auf den Google Play Store. Auch ist es nicht möglich, Applikationen aus dem Google Play Store auf einem Nicht-Android-System herunterzuladen (z.B. einem normalen PC), um sie dann über USB auf einem Android-Gerät zu installieren. Einige Software-Hersteller bieten jedoch ihre Applikationen auch in alternativen App-Stores oder direkt als Installationsdatei an; diese lässt sich dann auf beliebige Weise herunterladen und auf dem Android-Gerät installieren.

6.2. Apple iOS

In einem Interview erzählte Steve Jobs, damaliger Apple-CEO, dass er bereits Anfang der 2000er Jahre die Idee hatte, einen Multi-Touch-Bildschirm zu entwickeln, auf dem man wie auf einer Computertastatur tippen konnte. Dieser Bildschirm war ursprünglich für ein Tablet gedacht, Jobs entschied jedoch, dass zuerst ein Telefon gebaut werden sollte. 2004 begann die Entwicklung des späteren iPhones unter dem Codenamen "Project Purple". Dabei waren einige Bezeichnungen für das Smartphone im Gespräch. So hätte es "TelePod", eine Anspielung auf das Wort "Telephone" und der iPod-Produktlinie, oder "Mobi", die Kurzform von "mobile", heißen können. Außerdem war der Vorschlag "TriPod" im Gespräch, der die drei Hauptfunktionen des Handys (iPod, Telephone und Internet) beschreiben sollte. "iPad" war eine weitere Namensidee, die 2010 für das Tablet von Apple verwendet wurde.

6.2.1. Die Geschichte von Apple iOS

Im September 2005 stellte Apple das Motorola ROKR E1 vor, das als erstes Mobiltelefon mit iTunes synchronisiert werden konnte. Schon kurz darauf sickerte jedoch durch, dass Steve Jobs mit dem ROKR unzufrieden sei, weil es als Fremdprodukt nicht in die Designlinie der Apple-Produktpalette passte. Diese Einschätzung wurde im September 2006 von Apple durch den Entzug der ROKR-Unterstützung bei iTunes bestätigt. Stattdessen wurde eine weitere iTunes-Aktualisierung mit Unterstützung für ein noch unbekanntes Mobiltelefon veröffentlicht, das offensichtlich nicht nur Audio-, sondern auch Video- und Bilddateien wiedergeben können sollte. Dies führte in verschiedenen Medien zu Spekulationen über ein zu erwartendes Apple-Mobiltelefon, die bis zum Jahresende 2006 immer konkreter wurden. Das ursprüngliche Betriebssystem, damals von Steve Jobs als *OS X* bezeichnet, wurde am 9. Januar 2007 zusammen mit dem iPhone vorgestellt. Es ist ein Derivat von Mac OS X, das an das iPhone und seinen ARM-Prozessor angepasst wurde. Mit der Einführung der Version 2.0 im Jahr 2008 wurde aus dem Betriebssystem das iPhone OS.

Nachdem dieses Betriebssystem auf immer mehr Geräte übertragen wurde, und nicht mehr nur alleiniges Betriebssystem des iPhones war, wurde es von Apple am 7. Juni 2010 in *iOS* umbenannt. Für die Umbenennung hat Apple den entsprechenden Markennamen von Cisco Systems lizenziert.

Anfang Juni 2007 wurde der Verkaufsbeginn in den Vereinigten Staaten am 29. Juni durch die Ausstrahlung eines Fernsehwerbespots angekündigt. Vorerst waren die Geräte nur in den Apple Retail Stores und Verkaufsstellen von AT&T erhältlich. In weiten Teilen Europas wurde das Gerät ab dem 9. November 2007 angeboten. Zu diesem Zeitpunkt begann der Vertrieb in Deutschland ausschließlich über T-Mobile zum Preis von 399 Euro, gekoppelt mit einem auf das T-Mobile-Netz beschränkten SIM-Lock, im Gegenzug beteiligte T-Mobile Apple an den monatlichen Umsätzen.

Der Verkauf der Nachfolgegeneration, das iPhone 3G, startete am 11. Juli 2008 parallel in 21 Ländern, darunter Deutschland, Österreich und die Schweiz. Vorgestellt wurde die zweite Geräte-Generation am 9. Juni 2008 im Rahmen der Worldwide Developers Conference **(**WWDC 2008). Neu waren unter anderem die Unterstützung neuer Mobilfunkstandards sowie die Rückseite aus schwarzem Kunststoff.

Die dritte iPhone-Generation, das iPhone 3GS wurde etwa ein Jahr später, am 8. Juni 2009 vorgestellt. Neuerungen waren vor allem die bessere Kamera, der digitale Kompass sowie der schnellere Prozessor. Das Design des Geräts blieb unverändert. Der Verkauf in Deutschland wurde am 19. Juni gestartet. Das *S* in der Modellbezeichnung steht für „Speed“ (*engl. für Geschwindigkeit*).
Mit der Veröffentlichung des iPhone 4 erlangte die iPhone-Reihe die bis dato größte Bekanntheit. An den ersten drei Verkaufstagen ging das Smartphone laut Apple 1,7 Mio. Mal über die Ladentheke. Das Aussehen wurde von Grund auf verändert. Vorder- und Rückseite bestehen aus Glas. Neu ist zudem das Retina-Display. Außerdem wurde eine Frontkamera für Videotelefonate verbaut. In die Kritik geriet das iPhone 4 wegen seiner Empfangsprobleme, die bei zu festem Umklammern des Gehäuses auftraten. Vorgestellt wurde das Gerät auf der WWDC am 7. Juni 2010.

Das iPhone 4S kam im Oktober 2011 auf den Markt. Neu war vor allem der Spracherkennungsassistent Siri, der in diesem Gerät erstmals zur Verfügung steht. Vom Design her unterscheidet sich das Gerät nur geringfügig vom Vorgänger. Das *S* in der Modellbezeichnung steht für Siri.

Das iPhone 5 wurde am 12. September 2012 vorgestellt. Der Bildschirm wurde vergrößert, trotzdem ist es dünner und leichter als der Vorgänger. Außerdem wurde ein leistungsfähigerer A6-Prozessor eingebaut.

Abbildung 66.: iPhone 5

6.2.2. Das Bedienkonzept

Das Bedienkonzept von iOS soll möglichst einfach gehalten sein. Somit beschränkt es sich fast ausschließlich auf den Home-Bildschirm, auch *Springboard* genannt, und die Synchronisierung mit der iCloud bzw. iTunes. iOS wird fast ausschließlich über den Multitouchbildschirm gesteuert, nur das Sperren des Geräts wird mit dem *Lockbutton* ausgelöst, und das Beenden von Anwendungen (genannt Apps) mit dem *Homebutton*. Dieser kann das Gerät ebenso wie der Lockbutton aus dem Standby-Modus aufwecken. iOS ist darauf ausgelegt mit allen anderen Apple-Produkten zusammenzuarbeiten. Es unterstützt Multi-Touch mit bis zu fünf Fingern. Multitouch wird teilweise zur Gestensteuerung verwendet, so lassen sich beispielsweise beim iPad Apps durch Gesten schließen oder wechseln.

Der Home-Bildschirm stellt die eigentliche Benutzeroberfläche von iOS dar. Kennzeichnend für diesen sind die auf einzelnen Seiten als Icons dargestellten Apps, von denen vier im *Dock* abgelegt werden können, die Statusleiste am oberen Bildschirmrand mit der Uhrzeit, dem Akkuladestand und gegebenenfalls Signalstärken, sowie der Lockscreen mit dem *Entriegeln*-Slider und einer Digitaluhr.

Erst im Laufe der Zeit war es möglich, Apps zu verschieben und nach Belieben zu ordnen oder zu löschen, mehrere Seiten mit Apps zu erstellen oder weitere Apps aus dem App Store zu installieren. Mit iPhone OS 3.0 kam die aus OS X bekannte Spotlight-Suche hinzu. Mit der Spotlight-Suche lässt sich nach Inhalten auf dem iOS-Gerät suchen. Ebenso kamen mit iPhone OS 3.0 *Push-Nachrichten* hinzu. Push-Nachrichten werden von Apps an das iOS-Gerät geschickt. Es handelt sich dabei um Textnachrichten, die z. B. von Instant-Messaging-Programmen oder Nachrichtenapps stammen können. Mit iOS 4.0 war es erstmals möglich, Apps in Ordnern zu ordnen. Außerdem konnte man den bis dahin nicht individualisierbaren schwarzen Hintergrund des Home-Bildschirms mit einem eigenen Hintergrundbild versehen. Zudem wurde mit iOS 4 eine Taskleiste eingeführt, die seitdem mit einem Doppelklick auf den Homebutton aufrufbar ist. In dieser Leiste werden die zuletzt verwendeten Apps angezeigt und können direkt von dort aufgerufen werden. Seit iOS 4 werden Apps nicht mehr durch den Homebutton beendet, sondern pausiert, sodass sie schneller geladen werden können. Bei Bedarf

werden Apps beendet um Arbeitsspeicher freizugeben. Mit iOS 5 kamen das *Notification-Center* und Siri hinzu. Das Notification-Center zeigt die letzten Push-Nachrichten an und lässt sich mit einem Wisch von der Statusbar nach unten aufrufen. Siri ist ein Assistent, der auf Sprachbefehle reagiert und diverse Aufgaben, wie das Erstellen und Absenden von SMS, ausführen kann.

7. Shell-Programmierung

Im Betriebssystem **Unix** ist die Verwendung von Shells als **Kommandointerpreter** Standard. In der Geschichte von Unix sind eine Reihe von solchen Kommandointerpretern entwickelt und vertrieben worden.

Der erste Kommandointerpreter von Unix und damit auf jedem Derivat verfügbar ist die **Bourne-Shell**, benannt nach ihrem Entwickler S. R. Bourne als Mitarbeiter der **Bell Laboratories von AT&T**.

Als Weiterentwicklung aus dem Berkeley-Unix-Derivaten BSD wurde an der University of Califonia in Berkeley die **C-Shell** entwickelt, die heute ebenfalls auf den meisten Unix-Derivaten verfügbar ist. Diese Shell orientiert sich sehr stark an der Syntax der Programmiersprache C und ist somit als Programmiersprache nicht kompatible mit der Bourne-Shell, verfügte aber über sinnvolle Erweiterungen, z.B. den Historymechanismus.

Daraufhin wurde von David G. Korn an den Bell Laboratories von AT&T die **Korn-Shell** als Auftragswerk entwickelt. Diese Shell vereinigt die Syntax der Bourne-Shell mit den Erweiterungen der C-Shell und verfügt darüber hinaus über weitere sinnvolle Erweiterungen.

Alle Unix-Derivate, die auf der Unix-Version **SVR4** basieren, verfügen über alle drei Shells, die der Anwender auswählen und verwenden kann.

Im Rahmen der Vorlesung sollen nur die Funktionen und Abläufe behandelt werden, die für die Bourne- und die Korn-Shell gelten.

7.1. Kommandosyntax

Eine Shell im Betriebssystem Unix umfasst jeweils zwei Funktionen:

- Kommandointerpreter
- Programmiersprache

Im Abschnitt 7.5. Verarbeitungsstrukturen soll die Shell als Programmiersprache behandelt werden. Zunächst soll jedoch der weiteren Betrachtung die Kommandosyntax des Betriebssystems, repräsentiert durch einen Kommandointerpreter, vorangestellt werden.

Die erste syntaktische Einheit in der Kommandozeile wird als Kommando, d.h. als ein ausführbares Programm, ein shellinternes Kommando oder ein Schlüsselwort der Programmiersprache interpretiert.

Beispiele für **kommandoname**:

- find ausführbares Programm (/usr/bin/find),
- ls ausführbares Programm (/bin/ls),
- cd shellinternes Kommando,
- pwd shellinternes Kommando,
- while Schlüsselwort der Programmiersprache und
- for Schlüsselwort der Programmiersprache.

In Abhängigkeit von **kommandoname** werden die **argumente** definiert. Dabei lässt sich allgemein angeben, dass es sich in der Regel bei dem ersten Argument um **Optionen** bzw. **Schalter** handelt, die im Allgemeinen mit **-** beginnen. Danach können weitere **Parameter** folgen.
Das Syntaxdiagramm ist in Abbildung 67. angegeben:

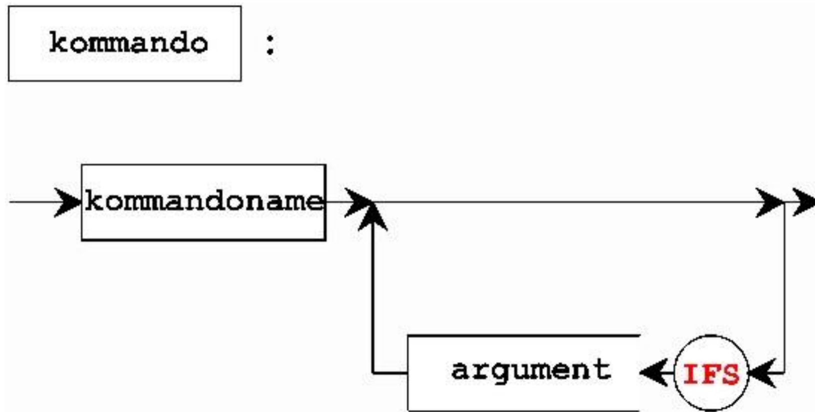


Abbildung 67.: Syntaxdiagramm der Unix-Kommandos

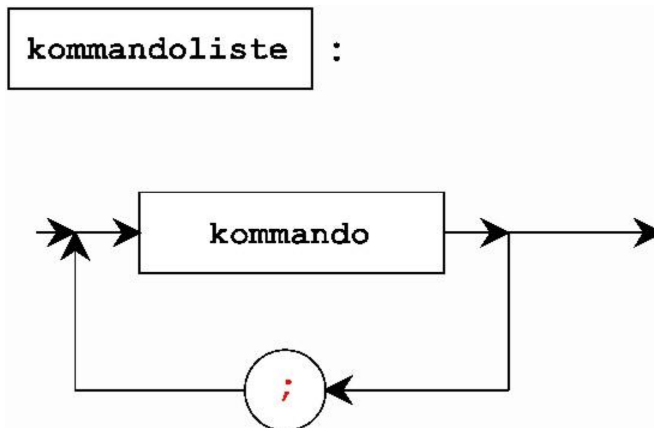


Abbildung 68.: Syntaxdiagramm der Kommandoliste

Wie in anderen Betriebssystemen auch kann man in der Kommandozeile der Shell mehrere Kommandos angeben, die dann sequenziell abgearbeitet werden. Das Sonderzeichen, das als Trenner zwischen den Kommandos fungiert ist das **Semikolon**.

Das Syntaxdiagramm ist in Abbildung 68. angegeben.

Eine weitere Verknüpfung von Kommandos stellt das **Pipekonzept** von Unix dar. Dabei wird der **Standardausgabekanal** des Kommandos **i** mit dem Standardeingabekanal des Kommandos **i+1** verbunden.

Bezüglich der Betrachtung von Prozessen liegt somit eine einfache Form der

- **Prozesskommunikation und**
- **Prozesssynchronisation**

vor. Als Sonderzeichen zur Verbindung von Standardausgabekanal mit Standardeingabekanal fungiert das **Pipez**eichen.

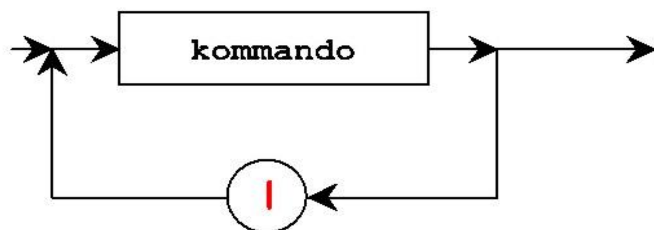


Abbildung 69.: Syntaxdiagramm des Unix-Pipekonzeptes

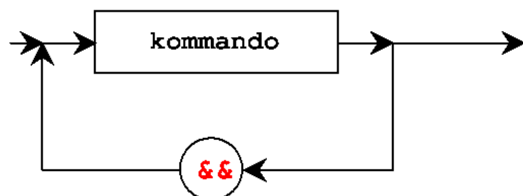
Die Shell von Unix stellt weiterhin noch die Möglichkeit des logischen Verknüpfens von Kommandos zur Verfügung. Dabei sind die beiden Varianten **AND (UND)** und **OR (ODER)** vorhanden, die durch die Zeichenketten **&&** und **||** in der Kommandosyntax dargestellt werden.

Kommandofolgen, die mit dem logischen UND verknüpft sind, werden von links beginnend abgearbeitet, solange die Kommandos den **Exitstatus 0** liefern.

Bei Kommandofolgen mit dem logischen ODER werden die Kommandos solange von links beginnend bearbeitet, bis ein Kommando den **Exitstatus 0** liefert.

Die Syntaxdiagramme für beide Formen der logischen Verknüpfung von Unix-Kommandos sind im Abbildung 70. dargestellt:

logisches UND :



logisches ODER :

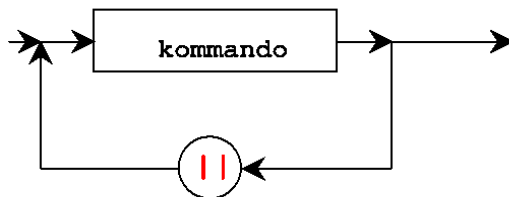


Abbildung 70.: Syntaxdiagramme für UND und ODER

Beispiel für eine UND-Verknüpfung von Unix-Kommandos:

```
$ who | grep -s "^paul" && write paul <nachricht
```

Für die Bewertung der Abarbeitung wird folgende Wahrheitstabelle verwendet:

Exitstatus Kommando 1	**Exitstatus Kommando 2**	**UND-Verknüpfung**
Exitstatus = 0 → ok	Exitstatus = 0 → ok	Exitstatus = 0 → ok
Exitstatus = 0 → ok	Exitstatus ≠ 0 → nicht ok	Exitstatus ≠ 0 → nicht ok
Exitstatus ≠ 0 → nicht ok	**wird nicht gestartet!**	Exitstatus ≠ 0 → nicht ok
Exitstatus ≠ 0 → nicht ok	**wird nicht gestartet!**	Exitstatus ≠ 0 → nicht ok

Tabelle 10.: Wahrheitstabelle für UND-Verknüpfung

Beachtet werden muss, dass bei fehlerhafter Ausführung des Kommandos 1 das Kommando 2 nicht mehr gestartet wird. Die Abarbeitung wird abgebrochen.

Beispiel für eine ODER-Verknüpfung von Unix-Kommandos:

```
$ (who | grep -s "^paul" && write paul <nachricht ) || > mail paul < nachricht
```

Auch hier muss die Bewertung der Abarbeitung nach folgender Wahrheitstabelle erfolgen:

Exitstatus Kommando 1	Exitstatus Kommando 2	ODER-Verknüpfung
Exitstatus = 0 → ok	**wird nicht gestartet!**	Exitstatus = 0 → ok
Exitstatus = 0 → ok	**wird nicht gestartet!**	Exitstatus = 0 → ok
Exitstatus ≠ 0 → nicht ok	Exitstatus = 0 → ok	Exitstatus = 0 → ok
Exitstatus ≠ 0 → nicht ok	Exitstatus ≠ 0 → nicht ok	Exitstatus ≠ 0 → nicht ok

Tabelle 11.: Wahrheitstabelle für ODER-Verknüpfung

7.2. Variable und Parameter

Die Shells von Unix kennen nur einen Datentyp ihrer Variablen und das ist der Datentype **string.**

In der Korn-Shell wurden darüber hinausgehend noch die Datentypen **integer** und **Feld (array)** realisiert. Damit lassen sich bestimmte Aufgaben in Shell-Skripten einfacher, übersichtlicher und im Ablauf schneller realisieren. Dennoch sind auch die einfachen Mechanismen, aufwärts kompatible zur Bourne-Shell vorhanden.

7.2.1. Umgang mit Shell-Variablen

Im Umgang mit Variablen der Shell sind doch einige Besonderheiten gegenüber anderen Programmiersprachen zu beachten.

Im Umgang mit Variablen lassen sich grundlegend drei Formen unterscheiden:

- **Variablendeklaration,**
- **Wertzuweisung und**
- **Wertreferenzierung (Zugriff auf den Wert einer Variablen)**

Zur Verdeutlichung des Sachverhaltes sollen wenige Beispiele dienen:

Form	**Beispiel**	**Bildausgabe**
Deklaration	**$ paul=**	
Wertzuweisung	**$ paul=otto**	
Wertreferenzierung	**$ echo $paul**	**otto**

Tabelle 12.: Umgang mit Shell-Variablen

Im Allgemeinen werden Variablen in der Shell nicht deklariert. In der Wertzuweisung ist die Variablendeklaration implizit mit enthalten.

Wird eine Variable dennoch ohne Wertzuweisung deklariert, so wird bei der Wertreferenzierung die leere Zeichenkette zurückgegeben. Es gibt aber in der Shell Möglichkeiten, diesen Fall auszutesten und gegebenenfalls bestimmte Aktionen daraufhin auszuführen (siehe 7.3. Ersetzungsmechanismen.).

Weiterhin interessant ist die Gültigkeitsdauer von Shell-Variablen. Dazu soll eine kurze Kommandosequenz zur Verdeutlichung helfen:

$ lage=oben	# Variable *lage* wird deklariert mit dem Wert oben
$ echo $lage	# der Wert der Variablen wird referenziert
oben	# Wert der Variablen auf dem Bildschirm
$ ksh	# Aufruf einer Sub-Shell (Unterprogramm)
$ echo $lage	# erneut wird der Wert der Variablen referenziert die Variable *lage* ist in der Sub- # Shell nicht bekannt
$ lage=unten	# der Variablen *lage* wird ein neuer Wert zugewiesen
$ echo $lage	# der Wert der Variablen wird wieder referenziert
unten	# neuer Wert der Variablen auf dem Bildschirm
$ ^D	# Abmelden von der Sub-Shell, damit ist die alte Shell wieder aktiv
$ echo $lage	# erneut wird der Wert der Variablen referenziert
oben	# der erste Wert der Variablen ist erhalten geblieben

Tabelle 13.: Kommandosequenz ohne export

$ lage=oben	#	Variable *lage* wird deklariert mit dem Wert oben
$ echo $lage	#	der Wert der Variablen wird referenziert
oben	#	Wert der Variablen auf dem Bildschirm
$ export lage	**#**	**die Variable *lage* wird exportiert**
$ ksh	#	Aufruf einer Sub-Shell (Unterprogramm)
$ echo $lage	#	erneut wird der Wert der Variablen referenziert
oben	**#**	**durch den Export der Variablen wird sie an die Sub-Shell übergeben**
$ lage=unten	#	der Variablen *lage* wird ein neuer Wert zugewiesen
$ echo $lage	#	der Wert der Variablen wird wieder referenziert
unten	#	neuer Wert der Variablen auf dem Bildschirm
$ ^D	#	Abmelden von der Sub-Shell, damit ist die alte Shell wieder aktiv
$ echo $lage	#	erneut wird der Wert der Variablen referenziert
oben	#	der erste Wert der Variablen ist erhalten geblieben

Tabelle 14.: Kommandosequenz mit export

Grundsätzlich ist zu verzeichnen, dass Shell-Variable nur lokal in einer Shell, und damit in einem Shell-Skript Gültigkeit haben. Sie können durch den Befehl **export** global für weitere Ebenen (Unterprogramme) gemacht werden.

Es ist jedoch nicht möglich, den Wert einer Variablen aus einem Unterprogramm in die aufrufende Ebene zu übergeben.

7.2.2. Spezielle Variable

Die Arbeitsweise der Shell lässt sich sehr stark an individuelle Belange anpassen. Dazu werden Variable, so genannte Umgebungsvariable verwendet. Diese Variablen werden beim Starten der **Login-Shell** mit entsprechenden Werten deklariert.

Die Anpassung dieser Umgebungsvariablen kann durch die Abarbeitung der Shell-Skripte

- **/etc/profile**
- **$HOME/.profile**

durch den Systemadministrator und den Benutzer selbst vorgenommen werden.

Beispiele für Umgebungsvariable:

Variable	Bedeutung	typischerer Wert
HOME	Home-Directory des Benutzers	/home/otto
PATH	Anzahl von Directories zur Kommandosuche	/bin:/usr/bin:/etc
PS1	1. Promptzeichen **$** oder **#**	"$\PWD> "
PS2	2. Promptzeichen **>**	"weiter> "
MAIL	Pfad zum elektronischen Briefkasten	/usr/spool/mail/otto
IFS	Separatorzeichen für Kommandozeile	TAB, NL, Leerzeichen
TZ	Zeitzone	MEZ-1MESZ,3.5.0,9.5.0

Tabelle 15.: Beispiele für Umgebungsvariable

Über die Umgebungsvariablen hinausgehend verwendet die Shell weitere spezielle Variablen, deren Werte zwar in einem Shell-Skript referenziert werden können. Das Verändern der Werte ist jedoch nicht möglich.

Diese speziellen Variablen werden durch die Shell selbst gesetzt und beinhalten Informationen über den Zustand der Shell.

Folgende spezielle Variablen sind definiert:

Variable	Bedeutung	Kommando
-	gesetzte Shell-Optionen	set -xv
$	PID (Prozessnummer) der Shell	kill -9 $$
!	PID des letzten Hintergrundprozes-ses	kill -9 $!
?	Exitstatus des letzten Kommandos	cat lalala ; echo $?

7.2.3. Positionsparameter

Sehr häufig wird man Shell-Skripte in der gleichen Art und Weise verwenden wollen, wie die "normalen" Unix-Kommandos. D.h., dem ablaufenden Shell-Skript müssen beim Aufruf eine Reihe von **argumenten** übergeben werden. Zu diesem Zweck gibt es die Positionsparameter. Dieser Begriff ist frei gewählt.

Die Shell analysiert die Kommandozeile und übergibt diese Eingabe in aufbereiteter Form an das Shell-Skript. Interessant dabei ist, dass dieser Mechanismus auch verschachtelt funktioniert. Somit kann man die Positionsparameter **lokal** für einen Programmaufruf betrachten. Für die Parameterübergabe in die nächste Ebene werden dann dieselben Positionsparameter verwendet, allerdings mit anderen Werten.

Folgende Positionsparameter gibt es in der Umgebung des aufgerufenen Shell-Skriptes:

Positionsparameter	Bedeutung
#	Anzahl der Argumente **ohne kommandoname**
0	Name des Kommandos (***kommandoname***)
1	1. Argument **nach dem kommandoname**
:	:
9	9. Argument **nach dem kommandoname**
@	alle Argumente **ohne kommandoname**
*	alle Argumente **ohne kommandoname**

Zur Verdeutlichung soll ein kleines Beispiel-Shell-Skript dienen:

Shell_Proz:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "Alle Parameter zusammen:
$*"
echo "Meine Prozessnummer PID = $$"
```

Nachdem dieses Shell-Skript mit einem Editor erstellt wurde, muss es noch ausführbar gemacht werden. Dazu wird folgender Befehl eingegeben:

```
$ chmod u+x Shell_Proz
```

Daran anschließend wird das Shell-Skript wie folgt gestartet und erzeugt die entsprechenden Ausgaben auf dem Bildschirm:

```
$ Shell_Proz emil erna paul fred otto
Mein Name ist ./Shell_ProzMir wurden 5 Parameter übergeben
1. Parameter = emil
2. Parameter = erna
3. Parameter = paul
Alle Parameter zusammen:
emil erna paul fred otto
Meine Prozessnummer PID = 4711
$
```

Für ein sinnvolles Handeln der Positionsparameter 0 .. 9 bietet die Shell die Möglichkeit der **Linksverschiebung** mit dem Kommando **shift <zahl>.**

Dieses kleine Shell-Skript kann nun noch erweitert werden. Um das Prozesssystem andeutungsweise beobachten zu können, werden die speziellen Variablen verwendet. Dazu wird das Shellskript wie folgt ergänzt:

Shell_Proz:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "Alle Parameter zusammen:
$*"
echo "Meine Prozessnummer PID = $$"
# Ergänzungen für das Prozesssystem
UNTERPROGRAMM="Shell_Up"
$UNTERPROGRAMM &
echo "Das Unterprogramm $UNTERPROGRAMM mit der PID = $! wurde gestartet"
date
echo "Das Unterprogramm wird jetzt terminiert"
kill -9 $!
```

Bei den Ergänzungen fällt sofort auf, dass ein Unterprogramm **Shell_Up** im Hintergrund, als asynchron zu **Shell_Proz** gestartet werden soll. Dieses Unterprogramm muss mit einem Editor erstellt werden:

Shell_Up:

```
# !/bin/ksh
# copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK, 10. November
1997
#
echo "--> Mein Name ist $0"
echo "--> Meine PID = $$"
ps -f
```

Danach muss das Shell-Skript **Shell_Up** noch ausführbar gemacht werden. Dazu wird folgender Befehl eingegeben:

```
$ chmod u+x Shell_Up
```

Daran anschließend wird das Shell-Skript **Shell_Proz,** wie oben gezeigt, gestartet und erzeugt die entsprechenden Ausgaben auf dem Bildschirm:

```
$ Shell_Proz emil erna paul fred otto
Mein Name ist ./Shell_Proz
Mir wurden 5 Parameter übergeben
1. Parameter = emil
2. Parameter = erna
3. Parameter = paul
Alle Parameter zusammen:
emil erna paul fred otto
Meine Prozessnummer PID = 611
Das Unterprogramm Shell_Up
mit der PID=612 wurde gestartet
--> Mein Name ist ./Shell_Up
--> Meine PID = 612
Mo, 10 Nov 1994 19:09:50 MEZ
Das Unterprogramm wird jetzt terminiert
UID PID PPID C STIME TTY TIME COMMAND
hheineck 456 1 13 18:50:59 04 0:01 -ksh
hheineck 614 1 7 19:09:50 04 0:00 ps -f
$
```

In der Ausgabe sieht man sehr deutlich die Mischung der beiden Ausgaben von **Shell_Proz** und **Shell_Up**. Daran ist ablesbar, in welcher Reihenfolge die beiden Shell-Skripts durch das Prozesssystem gestartet und abgearbeitet werden.

Interessant ist auch die Tatsache, dass im abschließenden Beobachten der Prozesstabelle, mittels Kommando **ps -f** sowohl das Shell-Skript **Shell_Proz** mit der **PID = 611** als auch das Unterprogramm **Shell_Up** mit der **PID = 612** zum Zeitpunkt der Ausführung des Kommandos nicht mehr enthalten sind.
Ebenfalls erkennbar ist die Tatsache, dass ein Prozess, dessen **Vaterprozess** selbst terminiert, als Vater-PID **(PPID)** die Prozessnummer des so genannten **Init-Prozess, PID = 1** erhält.

7.3. Ersetzungsmechanismen

Die Shell als Kommandointerpreter kennt so genannte **Metazeichen**, z.B. zur Dateinamen-Expandierung. Wie aus anderen Betriebssystemumgebungen sind die Zeichen * und **?**, und darüber hinaus in der Shell die Intervallangaben **[...]** bzw. **[!...]** bekannt. Die Sonderbehandlung dieser Metazeichen lässt sich auch sperren. Dazu verwendet man

- für ein Zeichen den **Backslash **
- für eine Zeichenkette
  a) "..."
  b) '...'

Nun gibt es in der Shell als Programmiersprache auch das Metazeichen zur Referenzierung von Shellvaribalen **$**.

Folgende Befehlssequenz soll den Unterschied zwischen den o.g. Möglichkeiten **a)** und **b)** demonstrieren:

```
$ A="Inhalt von A"
$ X="Zugriff auf $A"
$ echo $X
Zugriff auf Inhalt von A

$ X='Zugriff auf $A'
$ echo $X
Zugriff auf $A
$
```

Während die Maskierung des Metazeichens **$** mittels der Möglichkeit **b) '...'** gelingt, hat die Variante **a) "..."** bezüglich der Variablenreferenzierung keinen Effekt.

"$variable"	**Die Shell wertet den Inhalt von variable aus!**
'$variable'	**Die Shell wertet den Inhalt von variable nicht aus!**

Die Shell kennt darüber hinaus noch eine Möglichkeit der **doppelten Wertreferenzierung.**
Diese Möglichkeit basiert auf der Verwendung des Befehls **eval** und soll an einem Beispiel demonstriert werden:

```
$ text="Hallo Freunde"
$ zeiger="\$text"
$ echo $text
Hallo Freunde
$ echo $zeiger
$text
$ eval echo $zeiger
Hallo Freunde
$
```

Einen ähnlichen Mechanismus, wie das Kommando **eval** verwendet die Korn-Shell beim Aktualisieren der Umgebungsvariablen. Durch das Kommando

```
PS1="\$PWD> "
```

wird in Abhängigkeit von der Veränderung der Umgebungsvariablen **PWD** durch das Kommando **cd**, auch die Umgebungsvariable **PS1** mitgeändert.

Sollen in einem Shell-Skript in der Dialogausgabe der Text zusammengesetzt werden, so bietet die Shell eine weitere Möglichkeit der Notation der Variablenreferenzierung an.

```
$ a="Variable"
$ echo "$a A"
Variable A
$ echo "$anwert = A"
$
$ echo "${a}nwert = A"
Variablenwert = A
$
```

Bezüglich der Möglichkeit, den Namen einer Variablen durch die Zeichen **{}** kenntlich zu machen, sind weitere spezielle Variablenzugriffsmechanismen in der Shell implementiert worden:

${variable:-wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.
b) Die Variable **variable** hat keinen Wert, dann wird bei der Referenzierung der Wert **wort** eingesetzt.

${variable:=wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.

b) Die Variable **variable** hat keinen Wert, dann wird der Variablen **variable** der Wert **wort** zugewiesen und bei der Referenzierung der Wert **wort** eingesetzt.

${variable:?wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird auf diesen Wert referenziert.
b) Die Variable **variable** hat keinen Wert, dann wird die Fehlermeldung **wort** ausgegeben und die Shell-Prozedure abgebrochen.

${variable:+wort}

Die Variable **variable** ist deklariert:

a) Die Variable **variable** hat einen Wert, dann wird der Wert **wort** referenziert.
b) Die Variable **variable** hat keinen Wert, dann bleibt dieser Zustand erhalten.

7.4. Kommandoersetzung

In der gleichen Art und Weise, wie Kommandos ihre Ausgabe auf den Bildschirm tätigen, sollte es natürlich möglich sein, dies in einer Variablen zu speichern.

Die folgende Form der Eingabe führt jedoch nicht zum Ziel

```
$ VERZEICHNIS=pwd
```

denn damit wird der **Variablen** VERZEICHNIS nur der Wert **pwd** zugewiesen, wie die Kontrolle deutlich macht:

```
$ echo $VERZEICHNIS
pwd
$
```

Für die Kommandosyntax wurde ein weiteres Metazeichen definiert, dessen Verwendung die folgende Befehlssequenz demonstriert:

```
$ VERZEICHNIS=`pwd`
$ echo $VERZEICHNIS
/home/hheineck/shell
$
```

Somit ist es möglich, im Laufe eines Programmablaufes auf Zwischenwerte in Kommandos zurückzugreifen, in der Form:

```
$ cd $VERZEICHNIS
$
```

Somit ist eine gängige Möglichkeit beschrieben, den Zustand beim Eintreten in ein Shell-Skript in Variablen zwischenzuspeichern und beim Austritt wieder einzustellen, ohne dass Zwischenzustände nach außen für den Benutzer sichtbar werden.

7.5. Verarbeitungsstrukturen

Zu einer Programmiersprache in der 3. Generation gehört auch ein prozeduraler Aspekt. Dieser wird durch die Verarbeitungsstrukturen in der Shell-Programmierung realisiert.

Begonnen werden soll mit der bedingten und einfachen Fallunterscheidung.

7.5.1. Bedingte und einfache Fallunterscheidung

Da die Syntaxdiagramme für die Verarbeitungsstrukturen von anderen Programmiersprachen bekannt sind, soll an dieser Stelle darauf verzichtet werden und nur die reine Syntax in ihrer Schreibweise verwendet werden.

Dabei werden die Schlüsselwörter der Sprache **blau** dargestellt. Dabei ist festzustellen, dass diese Schlüsselwörter in dieser Sprache die Begrenzer für die **rot** dargestellten syntaktischen Einheiten sind.

```
if kommando_1 ; kommando_2 ; ...
   then kommando_a ; kommando_b ; ...
   else kommando_A ; kommando_B ;...
fi
```

Das Kennzeichnende an den Verarbeitungsstrukturen in der Shell ist es, dass nach dem Schlüsselwort **if** keine Bedingung formuliert wird.

Vielmehr werden die Kommandos **kommando_1 ; kommando_2 ; ...** abgearbeitet und der Exitstatus des letzten Kommandos entscheidet, welcher Zweig danach durchlaufen wird.
Dabei gelten folgende Festlegungen:

a) bezüglich des Exitstatus eines Programms:

1. Programm ist fehlerfrei abgelaufen → Exitstatus = 0
2. Programmablauf war fehlerbehaftet → Exitstatus $\neq 0$

b) bezüglich der zu durchlaufenden Programmzweige:

1. Der Exitstatus des letzten Kommandos aus **kommando_1 ; kommando_2 ; ...** ist = 0
   → Kommandos **kommando_a ; kommando_b ; ...** werden abgearbeitet.
2. Der Exitstatus des letzten Kommandos aus **kommando_1 ; kommando_2 ; ...** ist $\neq 0$
   → Kommandos **kommando_A ; kommando_B ; ...** werden abgearbeitet.

Für die Verwendung von verschachtelten Bedingten und Einfachen Fallunterscheidungen gibt es noch ein weiteres Schlüsselwort **elif**, das eine Reduktion des Schreibaufwandes bedingt:

```
 if kommandoliste
    then kommandoliste
    elif kommandoliste
        then kommandoliste
        else kommandoliste
 fi
```

Ein kleines Beispiel-Shell-Skript soll die Verwendung dieser Verarbeitungsstruktur verdeutlichen:

```
#!/bin/ksh
# copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
PROG=mail
DIR=`pwd`
if cd /bin ; ls | grep $PROG >/dev/null
   then echo "Das Programm $PROG ist im Verzeichnis /bin!"
   elif cd /usr/bin ; ls | grep $PROG >/dev/null
        then echo "Das Programm $PROG ist im Verzeichnis /usr/bin!"
        else echo "Kann $PROG nicht finden!"
fi
cd $DIR
```

7.5.2. Kommando test

Das Kommando **test** ist ein shellinternes Kommando und wird in Shell-Skripten sehr häufig verwendet, um den Test auf eine Bedingung zu formulieren. Dazu wird der Exitstatus des Kommandos entsprechend der Auswertung der Bedingung gesetzt:

a) Die Bedingung ist erfüllt → Exitstatus = 0
b) Die Bedingung ist nicht erfüllt → Exitstatus ≠ 0

Für die Verwendung des Kommandos gibt es zwei Notationsformen, die äquivalent verwendet werden können:

```
test [!] ausdruck
[ [!] ausdruck ]
```

Welche Art von Bedingungen können mittels Kommando **test** geprüft werden:

1. Eigenschaften von Dateien,
2. Eigenschaften und Vergleiche von Zeichenketten und
3. algebraische Vergleiche ganzer Zahlen

7.5.2.1. Eigenschaften von Dateien

ausdruck	Bedeutung
-r <datei>	datei **existiert und Leserecht**
-w <datei>	datei **existiert und Schreibrecht**
-x <datei>	datei **existiert und Ausführungsrecht**
-f <datei>	datei **existiert und ist einfache Datei**
-d <datei>	datei **existiert und ist Verzeichnis**
-h <datei>	datei **existiert und ist symbolische Link**
-c <datei>	datei **existiert und ist zeichenorientiertes Gerät**

`-b <datei>`	**datei existiert und ist blockorientiertes Gerät**
`-p <datei>`	**datei existiert und ist benannte Pipe**
`-u <datei>`	**datei existiert und für Eigentümer s-Bit gesetzt**
`-g <datei>`	**datei existiert und für Gruppe s-Bit gesetzt**
`-k <datei>`	**datei existiert und t- oder sticky-Bit gesetzt**
`-s <datei>`	**datei existiert und ist nicht leer**
`-t <dateikennzahl>`	**dateikennzahl ist einem Terminal zugeordnet**

Tabelle 16.: test – Eigenschaften von Dateien

Beispiel-Shell-Skript zur Verdeutlichung:

```
if [ -r /etc/passwd ]
   then more /etc/passwd
   else echo "Kann Datei nicht lesen oder existiert nicht"
fi
```

7.5.2.2. Eigenschaften und Vergleiche von Zeichenketten

Zeichenketten in der Shell sind beliebige Folgen von Zeichen. Dabei ist auch die leere Menge möglich. Diese sollte jedoch unbedingt in Anführungszeichen `""` eingeschlossen sein. Dadurch wird verhindert, dass das Kommando test keinen Parameter an einer beliebigen Stelle übergeben bekommt und dies durch den Exitstatus 1 und Abbruch quittieren wird.

Man sollte sich deshalb generell das Einschließen von Zeichenketten in den Anführungszeichen angewöhnen.

ausdruck	Bedeutung
-n <zeichenkette>	wahr, wenn zeichenkette nicht leer
-z <zeichenkette>	wahr, wenn zeichenkette leer ist
<zeichenkette1>= <zeichenkette2>	wahr, wenn Zeichenketten gleich sind
<zeichenkette1>!= <zeichenkette2>	wahr, wenn Zeichenketten verschieden sind
<zeichenkette>	wahr, wenn zeichenkette nicht leer

Tabelle 17.: test – Eigenschaften von Zeichenketten

Beispiel-Shell-Skript zur Verdeutlichung:

```
if [ "$USER" = "otto" ]
   then echo "Guten Morgen Otto!"
        exit
fi
```

7.5.2.3. Algebraische Vergleiche ganzer Zahlen

In Variablen der Shell können bekanntlich nur Zeichenketten gespeichert werden. Dennoch kann die Shell eine Menge von Ziffern in beliebiger Größe als ganze Zahl behandeln und verarbeiten.

Die Syntax des Kommandos test bei dieser Möglichkeit reduziert sich auf folgende Notationsform:

```
ausdruck: <zahl1> <operator> <zahl2>
```

Die Syntax zeigt, dass es für die möglichen Vergleiche unterschiedliche Operatoren **operator** gibt:

operator	Bedeutung
-eq	**equal - gleich**
-ne	**not equal - ungleich**
-ge	**greater than or equal - größer gleich**
-gt	**greater than - größer**
-le	**less than or equal- kleiner gleich**
-lt	**less than - kleiner**

Tabelle 18.: test – Vergleichsoperatoren

Beispiel-Shell-Skript zur Verdeutlichung:

```
#!/bin/ksh
# copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
# Möglichkeit der Verwendung eines algebraischen Vergleiches
# Test auf die Übergabe der richtigen Anzahl Parameter
# und Test auf gültigen 2. Parameter
#
if [ "$#" -ne "2" ]
then echo "usage: $0 -1Amn <file>"
```

```
        exit
fi
if [ ! -r "$2" ]
then echo "Cannot read file \'$2\': No such file"
        exit
fi
# gleiche Funktionalität, jedoch zeitlich günstiger, da keine
# Zahlenkonvertierungen notwendig sind!
if [ "$#" != "2" ]
then echo "usage: $0 -1Amn <file>"
        exit
fi
if [ ! -r "$2" ]
then echo "Cannot read file \'$2\': No such file"
        exit
fi
```

7.5.2.4. Logische Verknüpfung von Bedingungen

Das Kommando test lässt ebenfalls zu, dass mehrere Bedingungen zu einem Ausdruck ausdruck logisch verknüpft werden können.

Bei den logischen Verknüpfungen von Bedingungen sind die Möglichkeiten der booleschen Algebra mit folgender Notation verwendbar:

UND:	<bedingung1>	-a <bedingung2>
ODER:	<bedingung1>	-o <bedingung2>
Klammern:	\(<ausdruck> \)	
Negation:	! <ausdruck>	

Tabelle 19.: logische Verknüpfungen

Shell-Skript:

```
#!/bin/ksh
#copyrigth by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
OLDDIR=`pwd`
BASE=`basename $1`
DIR=`dirname $1`
DATEI=`cd $DIR;pwd`/$BASE
if [ -s "$DATEI" -a -f "$DATEI" -a -r "$DATEI" ]
   then if (file "$DATEI" | grep "c program text" || \
            file "$DATEI" | grep "commands text" || \
            file "$DATEI" | grep "assembler program text" || \
            file "$DATEI" | grep "English text" || \
            file "$DATEI" | grep "ascii text") >/dev/null 2>&1
           then (pr -o10 -l64 -h "$DATEI" "$DATEI" | \
                 lp ) >/dev/null 2>&1
```

```
                else echo "Datei $DATEI hat nicht druckbaren Typ!"
            fi
        else echo "Datei $DATEI existiert nicht oder nicht zugreifbar"
    fi
    cd $OLDDIR
```

7.5.3. Mehrfache Fallunterscheidung

Eine mehrfache Fallunterscheidung in abgeänderter Bedeutung kann mittels case realisiert werden:

```
case wort in
    muster_1 ) kommandoliste_1;;
    muster_2 ) kommandoliste_2;;
    ...
esac
```

Für die einzelnen Fälle ist es nicht möglich, einzelne Bedingungen zu formulieren, sondern es wird mit Zeichenmustern die Unterscheidung realisiert. Die Zeichenkette **wort** wird dabei in der Reihenfolge der Anweisungen mit den vorgegebenen Mustern **muster_1**, **muster_2** usw. verglichen und bei Übereinstimmung wird die nachfolgende Kommandoliste ausgeführt. Die Kommandoliste mus mit einem doppelten Semikolon (**;;**) enden.

Folgendes Beispiel verdeutlicht die Verwendung der mehrfachen Fallunterscheidung:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK, 10. November
1997
#
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
WEITER=nein
while [ "$WEITER" = "nein" ]
do
   echo ${N} "Bitte geben Sie J oder N ein: $C"
   read ANTWORT
   case $ANTWORT in
        y | Y | j | J ) echo "Sie haben J eingegeben"
                        WEITER=ja ;;
        n | N )         echo "Sie haben N eingegeben"
                        WEITER=ja ;;
        * )             echo "Sie können nicht lesen"
                        ;;
   esac
done
```

7.5.4. Abweisende und nicht abweisende Schleife

In der gleichen Art und Weise, wie bei der einfachen und bedingten Fallunterscheidung, steht auch bei den Schleifen keine Bedingung nach den Schlüsselwörtern **while** und **until**, sondern eine Kommandoliste.

Die Verwendung der beiden Schleifen ist analog zur Notation der Fallunterscheidung und sieht wie folgt aus:

7.5.4.1. Abweisende Schleife

```
while kommandoliste_1
do
   kommandoliste_2
done
```

7.5.4.2. Nichtabweisende Schleife

```
until kommandoliste_1
do
   kommandoliste_2
done
```

Ein kleines Beispiel soll die Verwendung in Shell-Skripten demonstrieren:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
# Unterschiedliche Möglichkeiten der Unterdrückung
# von Newline
#
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
#
# Beispiel für die abweisende Schleife
# Gestaltung eines sicheren Bedienungsdialoges
#
WEITER="j"
while [ "$WEITER" = "j" ]
do
   echo ${N} "Bitte die Gruppe eingeben: $C"
   read GRUPPE
   if grep "$GRUPPE" /etc/group >/dev/null 2>&1
      then WEITER="n"
      else echo "--> Fehleingabe!"
           echo
   fi
done
```

7.5.5. Zählschleife

Die Notation und Verwendung der Zählschleife in Shell-Skripten unterscheidet sich sehr stark von ihrer Verwendung in anderen Programmiersprachen. Dabei ist zu beachten, dass keine Laufvariable in bestimmten Grenzen hoch- bzw. heruntergezählt wird.

Vielmehr wird in der Shell der Zählschleife eine Menge von Werten übergeben. Bei jedem Durchlauf wird einer Variablen der nächste Wert aus der Menge von Werten übergeben.

Die Anzahl von Durchläufen ist an die Menge von übergebenen Werten gebunden:

```
for name in wert_1 wert_2 wert_3 wert_4
do
   kommandoliste
done
```

Für viele Anwendungen ist es notwendig, für die in den Positionsparametern übergebenen Parameter in einer Schleife die gleichen Kommandos auszuführen.

D.h., es wäre schön, wenn anstelle der **wert_1 wert_2 wert_3 wert_4** ein Positionsparameter übergeben werden kann.

Dazu gibt es folgende Vereinbarung:

Wenn der Positionsparameter **$*** verwendet werden soll, wird die Zählschleife wie folgt notiert:

```
for name
do
   kommandoliste
done
```

Damit sind folgende Zugriffsmöglichkeiten, z.B. auf Dateien in einem bestimmten Verzeichnis wie folgt möglich:

```
set - bsp*
for i
do
   if [ -s "$i" -a -r "$i" -a -f "$i" ]
      then more $i
   fi
done
```

Ein etwas umfangreicheres Beispiel soll an dieser Stelle die bisher eingeführten Möglichkeiten der Shell vereinigen und demonstrieren. Als zentrale Ablaufsteuerung wird dabei die Rekursion verwendet, bei der sich ein Programm selbst wieder aufruft:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
OLDDIR=`pwd`
BASE=`basename $0`
DIR=`dirname $0`
PROGRAMM=`cd $DIR;pwd`/$BASE
if [ "$#" != "3" ]
   then echo "usage : $PROGRAMM <directory> <UID> <GID>"
        exit 1
fi
if [ ! \( -d "$1" -a -r "$1" \) ]
   then echo "Cannot read directory \`$1\`: No such directory"
        exit 2
fi
if grep "^$2" /etc/passwd >/dev/null 2>&1
   then :
   else echo "Cannot read UID \`$2\`: No such User"
        exit 3
fi
if grep "^$3.*$2" /etc/group >/dev/null 2>&1
   then :
   else echo "Cannot read GID \`$3\`: No such Group \
              or User \`$2\` not member"
        exit 4
fi
USER="$2"
GROUP="$3"
```

```
BASE=`basename $1`
DIR=`dirname $1`
HOMEDIR=`cd $DIR;pwd`/$BASE
echo "program change $HOMEDIR"
cd $HOMEDIR
set - *
(chown $USER *;chgrp $GROUP *) >/dev/null 2>&1
for i
do
   if [ -d $i -a -r $i -a "$i" != "*" ]
      then $PROGRAMM $i $USER $GROUP
   fi
done
cd $OLDDIR
exit 0
```

7.5.6. Das Kommando expr

Da es innerhalb der Shell direkt nicht möglich ist, werden zu diesem Zweck mehrere „Rechenprogramme“ unter Unix zur Verfügung gestellt. Zu den gebräuchlichsten gehört das Kommando expr, das die übergebenen Parameter als Ausdruck interpretiert und diesen auswertet.

Damit sind arithmetische Operationen wie Addition, Subtraktion, Multiplikation, Division und Restwertberechnung durchführbar und reguläre Ausdrücke können ausgewertet werden.

Die Notationsform kann wie folgt geschrieben werden:

```
expr ausdruck_1 operator ausdruck_2
variable=`expr ausdruck_1 operator ausdruck_2`
```

Dabei sind folgende Operatoren möglich:

1. Vergleichsoperatoren `<, <=, >, >=, =, !=`
2. arithmetische Operatoren `+, -, *, /, %`
3. spezielle Operatoren `|, &, :`

7.5.6.1. Vergleichsoperatoren

Beispiele für Vergleichsoperationen:

```
$ a=5
$ b=6
$ expr "$a" \> "$b"
0
$ expr "$a" \< "$b"
1
```

7.5.6.2. Arithmetische Operatoren

Beispiele für arithmetische Operationen:

```
$ n=`expr "$n" + 1`
$ echo $n
25761
```

```
$ x=`expr "$#" \* 2`
$ echo $x
4
```

Bei den meisten Operatoren ist zu beachten, dass sie innerhalb der Shell meistens eine Sonderbedeutung haben und als Metazeichen vor der Shell versteckt (maskiert) werden müssen.

7.5.6.3. Spezielle Operatoren

a) `expr ausdruck_1 | ausdruck_2`

Wenn ausdruck_1 weder die leere Zeichenkette noch 0 ist, ist das Ergebnis ausdruck_1 sonst ausdruck_2.

b) `expr ausdruck_1 & ausdruck_2`

Wenn weder ausdruck_1 noch ausdruck_2 die leere Zeichenkette oder 0 ist, ist das Ergebnis ausdruck_1 sonst 0.

c) `expr ausdruck_1 : ausdruck_2`

Die Zeichenketten ausdruck_1 und ausdruck_2 werden miteinander verglichen, beginnend mit dem ersten Zeichen in jeder Zeichenkette und endet mit dem letzten Zeichen in ausdruck_2. ausdruck_2 kann in der Form von regulären Ausdrücken angegeben werden.

Beispiele für c):

```
$ expr "Hallo Freunde" : "Hallo"
5
$ expr "Hallo" : "allo"
0
$ expr "$PATH" : ".*"
65
$ LANG=`expr "$VARIABLE" : ".*"`
```

Innerhalb der regulären Ausdrücke stehen dabei:

.	ein beliebiges Zeichen und
*	eine beliebig häufige Wiederholung des vorangestellten Zeichens
^	das folgende Textmuster steht am Zeilenanfang

7.6. Funktionen

Seit dem Unix System V Release 2 können in der Bourne-Shell auch Funktionen definiert und verarbeitet werden. Diese Funktionen werden im Datenbereich der Shell gespeichert und direkt von dieser abgearbeitet. Damit sind folgende Vorteile gegeben:

- kein Zugriff auf das Dateisystem um ein Shell-Skript zu laden, die Funktion ist im Speicher der Shell vorhanden.
- es wird kein neuer Prozess gestartet

```
funktionsname()
    {kommando_1;kommando_2;...}
```

Ein kleines Beispiel soll die Möglichkeiten verdeutlichen, die mittels Funktionsdefinitionen realisiert werden können:

Das Kommando cp muss mit mindestens zwei Parametern aufgerufen werden. Der 1. Parameter ist im Allgemeinen die Quelldatei, der 2. Parameter ist die Zieldatei, bei mehr als zwei übergebenen Parametern das Zielverzeichnis. Werden jedoch weniger als zwei Parameter übergeben, so bricht das Kommando cp mit dem Fehler ab:

```
cp: insufficient arguments (0)
usage: cp f1 f2
       cp f1 ... fn d1
```

Die Aufgabe besteht nun darin, eine Funktion zu schreiben, die die fehlenden Parameter im Dialog abfragt, damit der Fehler nicht auftritt:

```
cp()
    {
    case "$#" in
         0)  echo "Bitte Quelldatei eingeben: \c"
             read QUELLE
             cp $QUELLE;;
         1)  echo "Bitte Zieldatei eingeben: \c"
             read ZIEL
             cp $1 $ZIEL;;
         *)  /bin/cp $*
    esac
    }
```

Fakultätsberechnung mittels Funktion unter Verwendung des expr-Kommandos:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
fak()
   {
   if [ "$#" -eq 1 ]
      then if [ "$1" -ge 1 ]
              a=1
              then n=$1
                   while [ "$n" -gt 1 ]
                   do
                      a=`expr "$a" \* "$n"`
                      n=`expr "$n" - 1`
                   done
           fi
           echo $a
      else echo "usage fak:  fak <integer>"
   fi
   return 0
   }
```

Abschlussbeispiel für Shellprogrammierung:

```
#!/bin/ksh
#copyright by hheineck, Hochschule Hof, FAKULTÄT INFORMATIK
#
HOMEDIR=/home/seminare/1110
DEFDATEI=/home/hheineck/profile/Datei
DIR=`pwd`
clear
echo "Kopieren beliebiger Dateien für Nutzer"
echo "======================================"
echo
if echo '\c' | grep c > /dev/null 2>&1
   then N='-n'
   else C='\c'
fi
WEITER="j"
while [ "$WEITER" = "j" ]
  do
    echo ${N} "Bitte die Gruppe eingeben: $C"
    read GRUPPE
    if grep "$GRUPPE" /etc/group >/dev/null 2>&1
       then WEITER="n"
       else echo "--> Fehleingabe!"
            echo
    fi
  done
echo ${N} "Bitte Namen für Quelldatei (default=$DEFDATEI) angeben: $C"
```

```
read QUELLDATEI
if [ "$QUELLDATEI" = "" ]
   then QUELLDATEI=$DEFDATEI
   else BASE=`basename $QUELLDATEI`
        QDIR=`dirname $QUELLDATEI`
        QUELLDATEI=`cd $QDIR;pwd`/$BASE
fi
echo ${N} "Bitte Namen für Zieldatei angeben: $C"
read ZIELDATEI
if [ "$ZIELDATEI" = "" ]
   then ZIELDATEI="  "
fi
cd $HOMEDIR
set - *
for i
  do
    if ls -ld $i |grep "^d.*$i.*$GRUPPE.*$i" >/dev/null
       then echo ${N} "Soll für $i Datei nach \"$ZIELDATEI\" kopiert
werden (j/n): $C"
            read antwort
            case $antwort in
                 j|J|y|Y) cp $QUELLDATEI $HOMEDIR/$i/"$ZIELDATEI"
                          chown $i $HOMEDIR/$i/"$ZIELDATEI"
                          chgrp $GRUPPE $HOMEDIR/$i/"$ZIELDATEI";;
                 n|N    ) ;;
                 *      ) echo
                          exit 1;;
            esac
```

```
                echo
        fi
      done
cd $DIR
echo
echo "--> fertig!"
echo
exit 0
```

8. Das Betriebssystem Windows

Windows ist ein Betriebssystem, das seinen Siegeszug der Entwicklung und der Produktion des Personal Computers und somit der massenhaften Verbreitung dieser Systeme verdankt. Es hat sich dabei vom reinen DOS-orientierten Betriebssystem zu einem leistungsfähigen Betriebssystem für High-End-Desktops und Server entwickelt. Die **Firma Microsoft** hat es verstanden, Entwicklungen anderer Firmen im Betriebssystemumfeld rasch in das eigene Betriebssystem zu übertragen und als eigene Leistung zu vermarkten. Die dabei verwendeten Marketingstrategien sind sehr aggressiv und damit angreifbar, wie bestimmte Verfahren gegen den Konzern beweisen. Diese beiden Gründe haben dazu geführt, dass die Firma Microsoft heute der weltweit an Nummer 1 stehende unabhängige Softwareanbieter ist.

Gleichzeitig spalten das Betriebssystem und die Marketingstrategien von Microsoft die Anwender und Nutzer in zwei Lager. Man kann es nur lieben oder hassen.

8.1. Die Geschichte von MS-DOS und Windows

Die Geschichte der Betriebssysteme der **Firma Microsoft** sind untrennbar mit der Personal-Computer-Entwicklung von **IBM** verbunden. Als im Jahr 1981 IBM ihren ersten Personal Computer auf den Markt bringen wollte, fehlte dafür ein Betriebssystem. So kann Bill Gates ins Spiel, der einen BASIC-Interpreter zu dieser Zeit entwickelt hatte. Über diesen Kontakt wurde ein Computerhersteller ausfindig gemacht, die Firma Seattle Computer Products, die über ein passendes System **DOS** (Disc Operating System) verfügte. Bill Gates kaufte das System und bot es IBM im Zusammenhang mit seinem BASIC-Interpreter an. Wegen Änderungen am System, die IBM haben wollte, stellte er den Entwickler von DOS, **Tim Paterson** in seine junge Firma Microsoft ein. Es entstand das Betriebssystem **MS-DOS**, das den IBM-Personal-Computer-Markt beherrschte.

In erster Näherung ist MS-DOS eine größere und bessere Version von CP/M. Es läuft nur auf Intel-Plattformen und unterstützt keine Multiprogrammierung (Multitasking). Der Kommandozeileninterpreter hatte mehr Funktionen und es gab mehr Systemaufrufe, doch bleibt die grundlegende Funktion des Betriebssystems auf dem Niveau von CP/M stehen.

Als Microsoft dann entschied, einen Nachfolger für MS-DOS zu entwickeln, waren sie sehr vom Erfolg des Macintosh beeinflusst. Sie entwickelten eine **GUI**-(**G**raphical **U**ser **I**nterface)-basiertes System namens **Windows**, welches zunächst auf MS-DOS aufsetzte (das bedeutet, dass es mehr eine Shell war als ein Betriebssystem). Von 1985 bis 1995 blieb es dabei, dass Windows nur eine grafische Umgebung zur MS-DOS war.

Erst 1995 wurde eine eigenständige Version von Windows verkauft, **Windows 95**. Dieses System beinhaltete viele Betriebssystemelemente und MS-DOS wurde nur noch zum Hochfahren und als Laufzeitumgebung für ältere MS-DOS-Programme genutzt. 1998 stand mit **Windows 98** eine leicht veränderte Version dieses Systems zur Verfügung. Trotzdem, sowohl Windows 95 wie Windows 98 besitzen immer noch einen großen Anteil von **16-Bit-Intel-Maschinencode**. Eine deutlich stabilere Version dieses Betriebssystems stellte die Version **Windows 98 SE** (**S**econd **E**dition), die eine gewisse Verbreitung erfahren hat.

Fähigkeit	Windows 95/98	Windows NT
Reines 32-Bit-System	Nein	Ja
Security?	Nein	Ja
Sichere Dateiverwaltung?	Nein	Ja
Eigener Adressraum pro MS-DOS-Programm?	Nein	Ja
Unicode-fähig?	Nein	Ja
Läuft auf	Intel 80x86	80x86, Alpha, MIPS, …
Multiprozessorunterstützung?	Nein	Ja
Re-entranter Code im Kern?	Nein	Ja
Plug-and-Play?	Ja	Nein
Power Management?	Ja	Nein
FAT-32-Dateisystem?	Ja	Optional
NTFS-Dateisystem?	Nein	Ja
Win32 API?	Ja	Ja
Laufen alte MS-DOS-Programme?	Ja	Nein
Kritische Bereiche vom Benutzer ungeschützt?	Ja	Nein

Ein anderes Betriebssystem von Microsoft ist **Windows NT** (**N**ew **T**echnilogy). Es ist bis zu einem gewissen Grad kompatibel zu Windows 95, trotzdem aber ein komplett neu geschriebenes, reines System mit durchgeht **32-Bit-Intel-Maschinencode**.

Tabelle 20.: Unterschiede zwischen Windows 98 und Windows NT

Obwohl es ein moderneres Betriebssystem war, erfolgte erst mit **Windows NT 4.0** der Durchbruch, vor allem in Hinblick auf Unternehmensnetzwerke.

Windows NT 5 wurde zu Beginn des Jahres 1999 in **Windows 2000** umbenannt. Es war der Nachfolger von Windows 98 und Windows NT. Aber auch das gelang nicht vollständig, deshalb brachte Microsoft eine neue Version von Windows 98 heraus, die **Windows Me** (Millenium edition) genannt wurde.

Version	Max. Speicher	CPUs	Max. Clients	Clustergröße	Optimiert für
Professional	4 GB	2	10	0	Antwortzeit
Server	4 GB	4	Unbegrenzt	0	Durchsatz
Advanced Server	8 GB	8	Unbegrenzt	2	Durchsatz
Datacenter Server	64 GB	32	Unbegrenzt	4	Durchsatz

Tabelle 21.: verschiedene Versionen von Windows 2000

8.2. Windows-Programmierung

Vor der Betrachtung der Realisierung der einzelnen Funktionen des Betriebssystems sollen zunächst die Programmierschnittstelle und die Registrierung, eine kleine, im Hauptspeicher befindliche Datenbank betrachtet werden.

8.2.1. Die Programmierschnittstelle Win32

Wie in anderen Betriebssystemen üblich, verfügt auch Windows über eine Anzahl von Systemcalls, die aber von Microsoft nicht öffentlich bekannt gemacht werden, sich aber von Version zu Version ständig ändern. Stattdessen hat man eine vollständig beschriebene Schnittstelle mit dem Namen **Win32 API** (**Win32 A**pplication **P**rogramming **I**nterface) geschaffen, siehe Abbildung 71.

Binärprogramme für den Intel 80x86, die sich genau an diese Schnittstelle halten laufen somit ohne Änderung auf allen Windows-Versionen seit **Windows 95**. Lediglich die alten Windows-Versionen benötigten eine spezielle Bibliothek für die Abbildung auf die 16-Bit-Betriebssysteme (**Win32s**). Die Definition dieser Win32 API ändert sich nicht, nur werden ständig neue Funktionen hinzugefügt, die dann in alten Versionen nicht vorhanden sind.

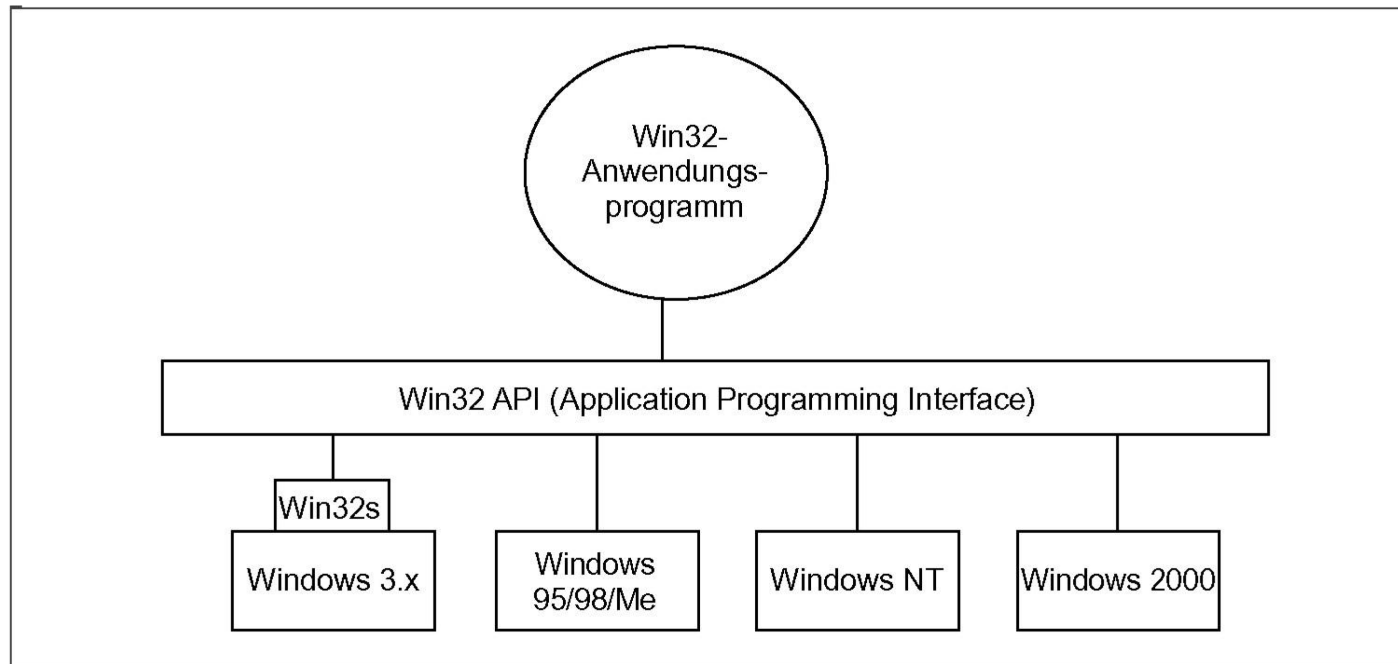


Abbildung 71.: Die Win32 API-Schnittstelle

8.2.2. Die Registrierung

Unter Windows-Betriebssystemen gibt eine Vielzahl von verschiedenartiger Hardware, die verwendet werden kann, siehe Abschnitt 1.5. Typische Hardware eines . Deshalb muss sich das Betriebssystem eine Vielzahl von Informationen darüber „merken". Unter **Windows 3.x** wurden die Informationen in Hunderten **.ini-Dateien** gespeichert, die überall auf der Festplatte verstreut waren. Seit **Windows 95** wurden alle Informationen in einer großen zentralen Datenbank, der Registrierung (**Registry**) zusammengefasst.

Dabei muss erwähnt werden, dass viele Teile des Windows-Betriebssystems kompliziert und verwirrend sind, die Registrierung jedoch einer der schlimmsten ist. Die sehr kryptische Nomenklatur macht das Ganze nicht besser.

Schlüssel	Beschreibung
HKEY_LOCAL_MACHINE	Eigenschaften von Hard- und Software
HARDWARE	Gerätebeschreibungen und Zuordnung von Hardware zu Treibern
SAM	Security und Zugangsinformationen für Benutzer
SECURITY	Systemweite Sicherheitsrichtlinien
SOFTWARE	Informationen über installierte Anwendungsprogramme
SYSTEM	Informationen zum Systemstart
HKEY_USERS	Informationen über jeden Benutzer; ein Schlüssel pro Benutzer
USER-AST-ID	Einstellungen für Benutzer AST
AppEvents	Geräusche bei Ereignissen (neue Mail, Fehler, ...)
Console	Einstellungen für die Kommandozeile (Farbe, Schrift, ...)
Control Panel	Desktop-Erscheinungsbild, Bildschirmschoner, Mauseinstellungen, ...
Environment	Umgebungsvariablen
KeyboardLayout	Welche Tastatur: 102-Tasten US, QWERTZ, Dvorak etc.
Printers	Informationen über installierte Drucker
Software	Benutzereinstellungen für Microsoft und andere Software
HKEY_PERFORMANCE_DATA	Hunderte von Performancezählern des Systems
HKEY_CLASSES_ROOT	Link zu HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
HKEY_CURRENT_CONFIG	Link zum aktuellen Hardwareprofil
HKEY_CURRENT_USER	Link zum aktuellen Benutzerprofil

Tabelle 22.: Registrierung

In Tabelle 22 ist der Aufbau der Registrierung, bestehend aus 6 Hauptschlüsseln und ihren Inhalten kurz dargestellt. Im Rahmen dieser Vorlesung wird nicht weiter auf diese eingegangen. Eine Vielzahl von Literatur beschreibt jedoch dieses Microsoft-Ungetüm, das mit den Kommandos **regedit** oder **regedit32** bearbeitet werden kann.

8.3. Die Struktur des Betriebssystems

Windows besteht aus zwei großen Teilen, dem Betriebssystem selbst, das im so genannten Kernmodus läuft und den Umgebungssubsystemen, die im Benutzermodus laufen. Der Betriebssystemkern ist ein traditioneller Kern in dem Sinne, dass er sich um die Prozessverwaltung, die Speicherverwaltung, das bzw. die Dateisysteme usw. kümmert.

Eine der zahlreichen Verbesserungen, die **Windows NT** gegenüber **Windows 3.x** aufwies, war seine modulare Struktur. Es besteht aus einem relativ kleinen Kern und zusätzlich aus einigen Serverprozessen. Benutzerprozesse interagieren mit den Serverprozessen über ein **Client-Server-Modell**.

Wie in Abbildung 72 ersichtlich, ist Windows in verschiedene Ebenen aufgeteilt, wobei jede Ebene die Dienste der darunter liegenden Ebene benutzt. Eine einzelne Ebene ist horizontal in viele Module aufgeteilt. Jedes Modul hat seine eigenen Funktionen und eine wohl definierte Schnittstelle zu den anderen Modulen.

8.3.1. Die Hardware Abstraction Layer

Auf der Hardware aufsetzend, sollte die **HAL** (**H**ardware **A**bstraction **L**ayer) die Portabilität des Betriebssystems auf verschiedenen Plattformen realisieren. Diese HAL ist dafür ausgelegt, die Unterschiede zwischen zwei Platinen verschiedener Hersteller zu verstecken, aber nicht die Unterschiede zwischen verschiedenen Prozessortypen, z.B. einem Pentium und einem Alphaprozessor. Wie in der Realität bewiesen, sind Windows-Betriebssysteme damit nur sehr schwer portierbar.

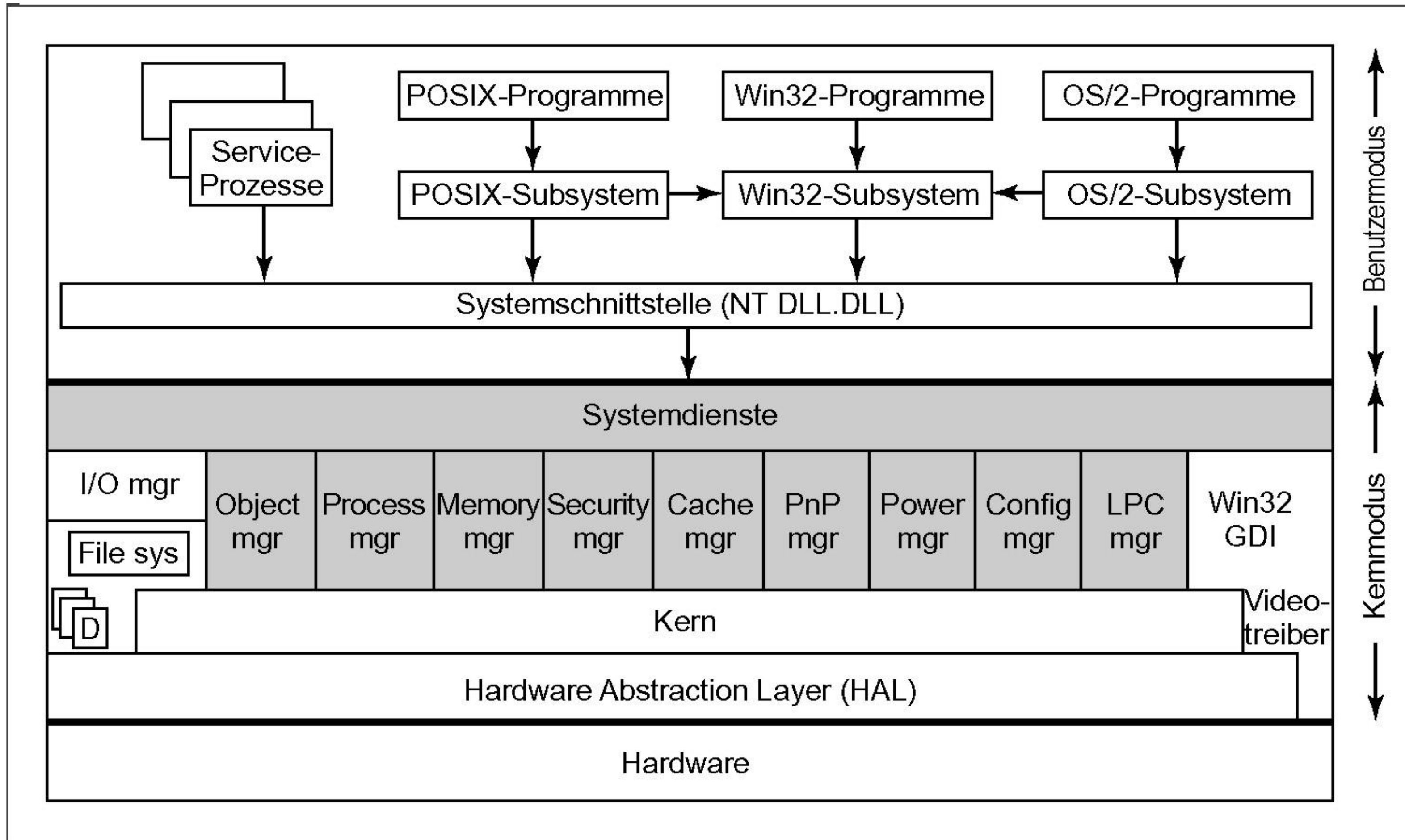


Abbildung 72.: Windows 2000 Systemstruktur

8.3.2. Der Betriebssystemkern

Oberhalb der HAL ist eine Schicht, die Microsoft Kernel bzw. Mikrokern nennt, die Gerätetreiber. Weiterhin liegen aber auch Systemkomponenten, wie die Speicherverwaltung, das Dateisystem u.s.w. in diesem Kernel.

Typ	Beschreibung
Prozess	Benutzerprozess
Thread	Thread innerhalb eines Prozesses
Semaphor	Zählsemaphor für Interprozesskommunikation
Mutex	Binäres Semaphor für kritische Bereiche
Ereignis	Synchronisationsobjekt mit persistentem Zustand
Port	Mechanismus für Prozessnachrichtenaustausch
Uhr	Objekt, damit Threads für eine gewisse Zeit schlafen können
Queue	Objekt für Benachrichtigung bei asynchroner Kommunikation
Offene Datei	Objekt für eine offene Datei
Access Token	Security-Bezeichner für ein Objekt
Profil	Datenstruktur für Messung der CPU-Belastung
Sektion	Struktur für das Einblenden von Dateien in den virtuellen Speicher
Schlüssel	Schlüssel in der Registry
Objektverzeichnis	Verzeichnis zur Gruppierung im Objektmanager
Symbolischer Link	Zeiger auf ein anderes Objekt durch den Namen
Gerät	Ein-/Ausgabe-Geräteobjekt
Gerätetreiber	Jeder geladene Gerätetreiber besitzt sein eigenes Objekt

Tabelle 23.: Allgemeine Objekttypen

Die in Abbildung 72 schraffiert dargestellten Komponenten sind die sogenannte Executive. Eine Vielzahl verschiedenen Manager, die vielmehr eine Sammlung von Prozeduren bereitstellen, die von Prozessen und Threads ausgeführt werden können. Die Gerätetreiber sind die mit **D** bezeichneten Rechtecke in der o.g. Abbildung.

8.3.3. Realisierung von Objekten

Objekte sind das vielleicht bedeutendste Konzept, sie bieten eine einheitliche und konsistente Schnittstelle zu allen Systemressourcen und Datenstrukturen an. Ihre Verwaltung wird vordringlich im **Objekt-Manager** betrieben. Einige Beispiele von Objekttypen sind in Tabelle 23 zusammengefasst.

Diese Einheitlichkeit bringt verschiedene Vorteile:

1. Alle Objekte werden auf die gleiche Weise benannt und es wird in der gleichen Weise über **Objekt-Handles** darauf zugegriffen.
2. Alle Zugriffe von Objekten laufen über den **Objekt-Manager**. Damit ist es möglich dort alle Sicherheitsüberprüfungen zu konzentrieren und dafür zu sorgen, dass kein Prozess sie umgehen kann.
3. Die gemeinsame Nutzung von Objekten wird ebenfalls einheitlich gehandhabt.
4. Da jedes Öffnen und Schließen eines Objektes über den **Objekt-Manager** läuft, ist es einfach zu verfolgen, welche Objekte noch benutzt werden und welche bedenkenlos gelöscht werden können.
5. Dieses einheitliche Objektverwaltungsmodell ermöglicht die unkomplizierte Verwaltung von Ressourcenkontigenten.

Ein Objekt ist eine Datenstruktur im RAM, die die in Abbildung 73 gezeigte Struktur hat. Da Objekte nur **Kerndatenstrukturen** sind, bedeutet das ihren Verlust beim Neustart (oder beim Absturz). Wenn das System hochfährt sind keine Objekte vorhanden (außer für die Leerlauf- und Systemprozesse, deren Objekte fest in ntoskrnl.exe eingebaut sind). Alle Objekte werden im laufenden Betrieb erzeugt, wenn das System hochfährt oder eines der zahlreichen Initialisierungs- (und später Benutzer-) Programme abläuft.

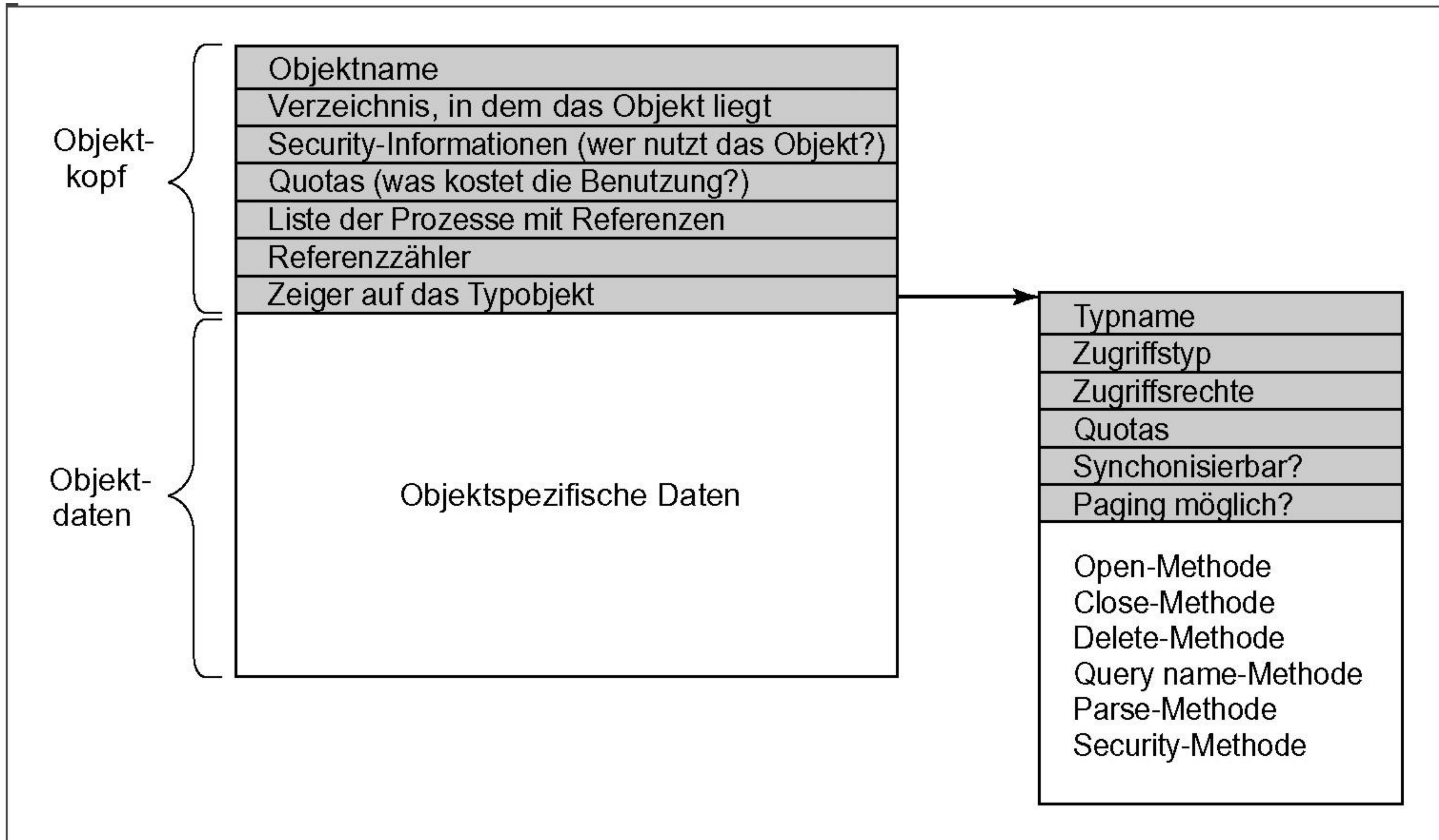
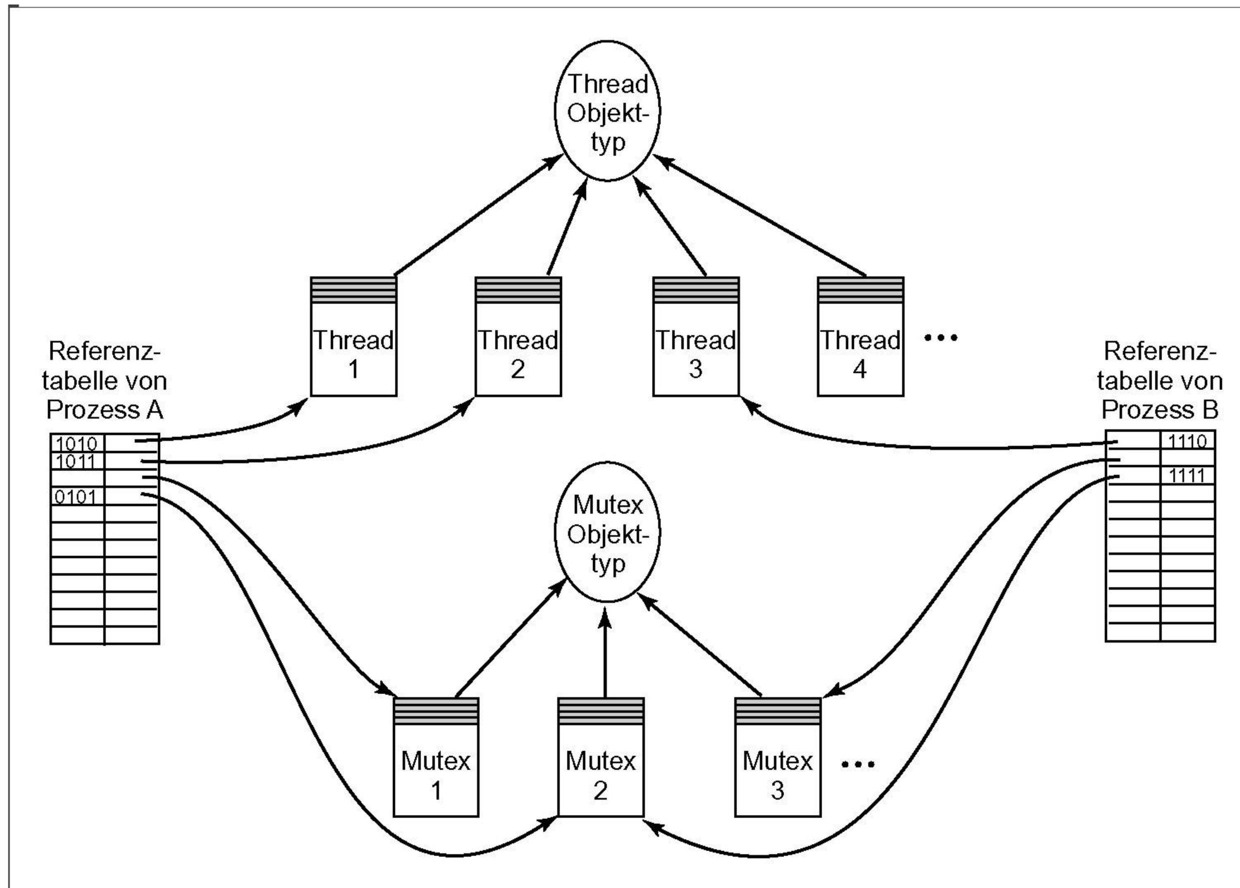


Abbildung 73.: Die Struktur eines Objektes



Die Handle-Tabellen für zwei Prozesse und ihre Beziehung zu einigen Objekten ist in Abbildung 75 dargestellt. Die zugehörigen Einträge in den Handle-Tabellen beinhalten die Rechte für alle diese Objekte. Mutex 2 wird von beiden Prozessen benutzt und dient somit der Synchronisation zwischen den Prozessen.

Abbildung 74.: Die Beziehung zwischen Handle-Tabellen, Objekten und Objekttypen

8.4. Prozesse und Threads
8.4.1. Grundlegende Konzepte

Windows stellt traditionelle Prozesse, die miteinander kommunizieren und sich untereinander synchronisieren können, bereit, wie sie aus **Unix** bekannt sind. Jeder Prozess enthält mindestens einen **Thread**, der wiederum mindesten einen **Fiber** enthält. Mehrere Prozesse können zu bestimmten Verwaltungszwecken in **Jobs** gebündelt werden.

Name	Beschreibung
Job	Gruppe von Prozessen, die Quotas und Limits teilen
Prozess	Container zur Speicherung von Ressourcen
Thread	Einheit, die vom Kern scheduled wird
Fiber	Leichtgewichtiger Thread, der vom Benutzer verwaltet wird

Tabelle 24.: Basiskonzepte für CPU- und Ressourcenverwaltung

Ein Job ist in Windows eine Sammlung von einem oder mehreren Prozessen, die als Einheit verwaltet werden sollen. Eine Besonderheit sind Kontingente und Ressourcenbeschränkungen, die zu einem Job gehören und in dem entsprechenden Jobobjekt gespeichert werden. Kontingente enthalten Informationen, wie z.B.:

- maximale Anzahl von Prozessen,
- die gesamte CPU-Zeit, für jeden Prozess individuell oder für alle Prozesse zusammen,
- maximale Speichernutzung, für jeden Prozess individuell oder für alle Prozesse zusammen,
- Sicherheitseinschränkungen.

Prozesse sind interessanter als Jobs und auch wichtiger. Wie auch in Unix sind Prozesse Container für Ressourcen. Jeder Prozess hat einen 4 GByte Adressraum, wobei der Benutzer die unteren 2 GByte belegt und das Betriebssystem den Rest, siehe Abbildung 75.

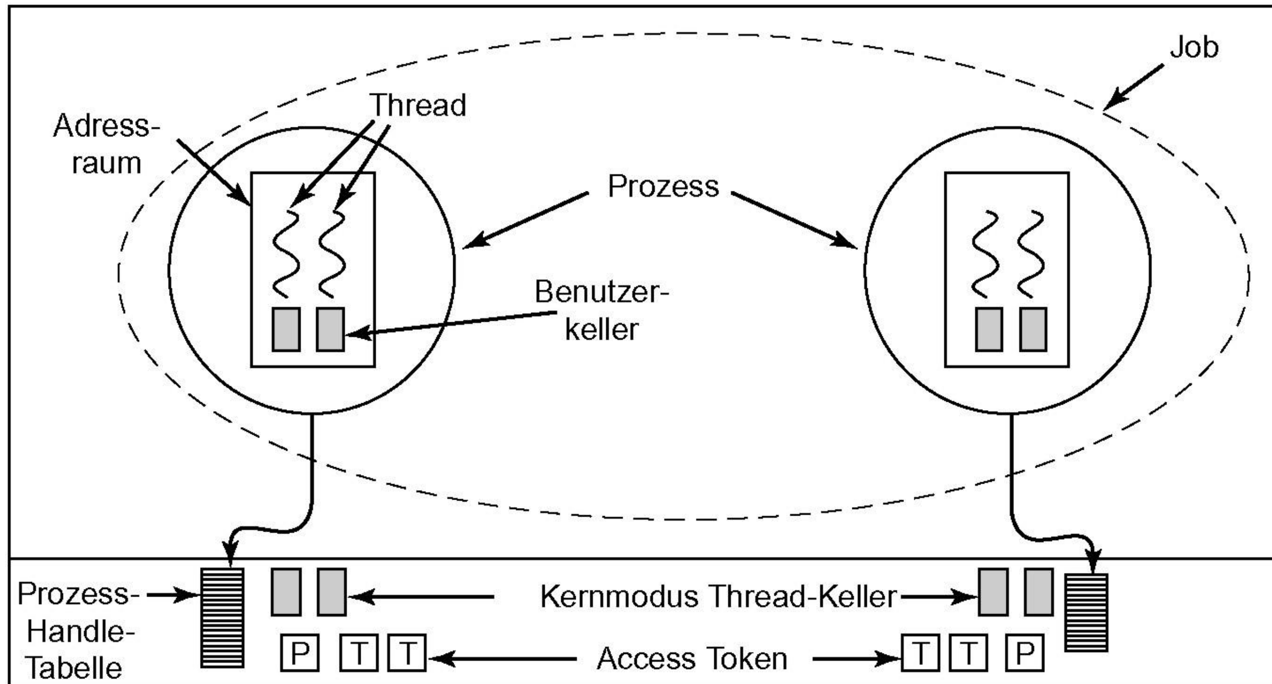


Abbildung 75.: Beziehungen zwischen Jobs, Prozessen und Threads

Ein Prozess hat eine Prozess-ID, mindestens einen Thread, eine Liste von Handles (die im Kernmodus verwaltet wird) und ein **Access-Token**, das die Sicherheitsinformationen beinhaltet. Prozesse werden durch den Win32-Aufruf CreateProcess erzeugt, der als Parameter den Namen einer ausführbaren Datei enthält, die den anfänglichen Inhalt des Adressraums angibt und den ersten Thread anlegt.

Jeder Prozess startet mit einem Thread, aber neue können dynamisch erzeugt werden. Threads bilden die Grundlage des Scheduling, weil das Betriebssystem immer Threads zur Ausführung wählt, nicht Prozesse. Folglich hat jeder Thread im Gegensatz zu Prozessen einen Zustand:

- bereit,
- laufend und
- blockiert.

Threads werden durch den Win32-Aufruf **CreateThread** erzeugt und laufen normalerweise im Benutzermodus. Wenn er aber einen Systemaufruf ausführt, wechselt er in den Kernmodus und läuft mit denselben Eigenschaften und Einschränkungen weiter. Jeder Thread hat zwei Stacks, einen für den Benutzer- und einen für den Kernmodus.

Es ist wichtig zu verstehen, dass Threads ein Scheduling-Konzept sind und kein Konzept für den Besitz von Ressourcen. Jeder Thread kann auf alle Objekte zugreifen, die zu seinem Prozess gehören. Es gibt keine Beschränkungen für einen Thread, dass er auf ein Objekt nicht zugreifen kann, nur weil ein anderer Thread dieses Objekt erzeugt oder geöffnet hat.

Der Wechsel zwischen Threads ist in Windows relativ aufwendig, da er immer mit den Ein- und späteren Austritt des Kernmodus bedeutet. Um einen **leichtgewichtigen Preudoparallelismus** zu bieten, stellt Windows **Fiber** bereit. Diese ähneln Threads, werden aber im Benutzerraum durch ein Programm (oder dessen Laufzeitsystem), das sie erzeugt, zeitlich eingeplant.

Jeder Thread kann mehrere Fiber haben, so wie ein Prozess mehrere Threads haben kann. Jedoch besteht die Ausnahme, dass ein Fiber, wenn es logisch blockiert wird, sich selbst in eine Warteschlange blockierter Fiber stellt und einen anderen Fiber auswählt, der dann im Kontext seines Threads laufen darf.

Win32-API-Funktion	Beschreibung
CreateProcess	Erzeuge neuen Prozess
CreateThread	Erzeuge neuen Thread im bestehenden Prozess
CreateFiber	Erzeuge neuen Fiber
ExitProcess	Beende Prozess und alle Threads
ExitThread	Beende diesen Thread
ExitFiber	Beende diesen Fiber
SetPriorityClass	Setze Prioritätenklasse für Prozess
SetThreadPriority	Setze Priorität für einen Thread
CreateSemaphore	Erzeuge neues Semaphor
CreateMutex	Erzeuge neuen Mutex
OpenSemaphore	Öffne bestehendes Semaphor
OpenMutex	Öffne bestehenden Mutex
WaitForSingleObject	Warte auf Semaphor, Mutex etc.
WaitForMultipleObjects	Warte auf eine Menge von Objekten mit geg. Handle
PulseEvent	Setze Ereignis auf signalisiert und dann auf nicht sig.
ReleaseMutex	Gib Mutex frei, damit anderer Thread zugreifen kann
ReleaseSemaphore	Erhöhe Semaphor um 1
EnterCriticalSection	Blockiere Sperre für einen kritischen Bereich
LeaveCriticalSection	Gebe Sperre für kritischen Bereich frei

Tabelle 25.: Win32-Aufrufe zum Prozesssystem von Windows

8.4.2. Scheduling

Windows besitzt keinen zentralen Thread für das **Scheduling**. Stattdessen wechselt ein Thread, wenn er nicht mehr weiterlaufen kann, in den Kernmodus und ruft selbst den Scheduler auf, um zum nächsten Thread zu wechseln. Die folgenden Bedingungen zwingen den aktuell laufenden Thread, den Scheduler aufzurufen:

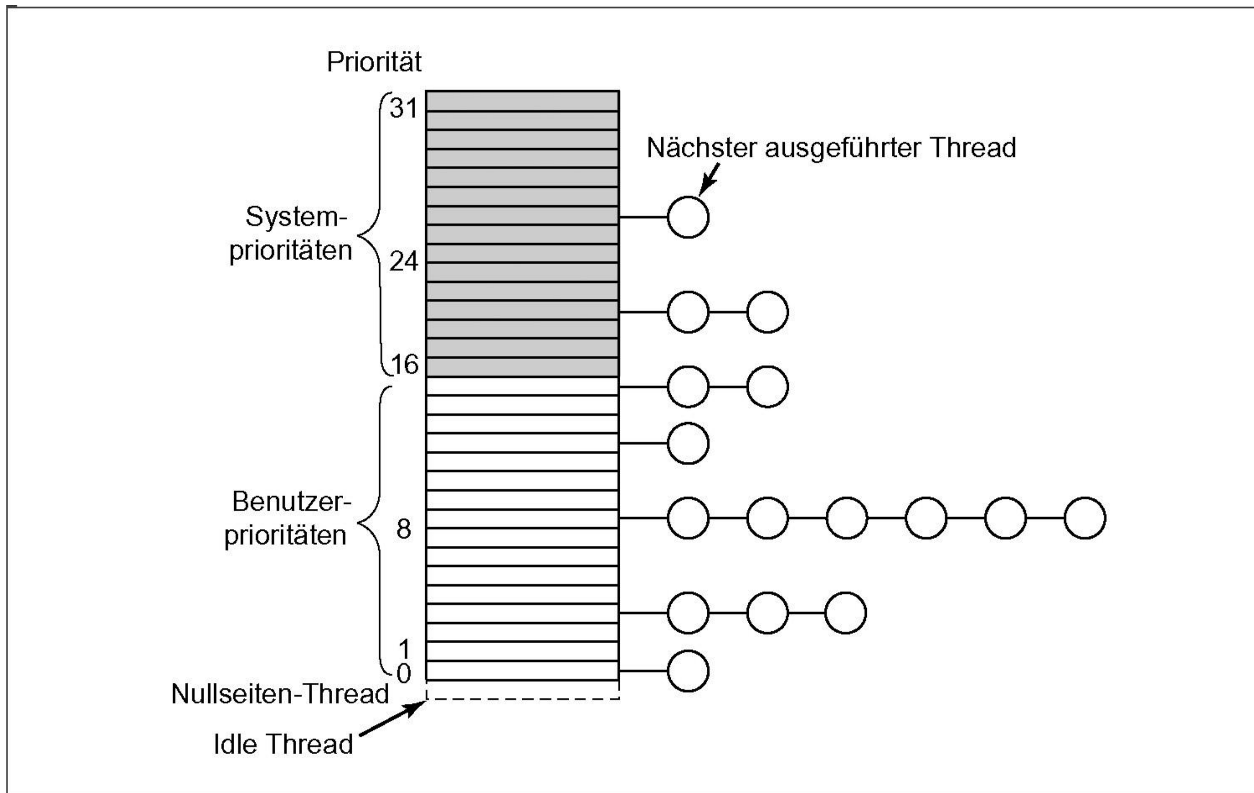
1. Der Thread wird durch ein Semaphore, einen Mutex, ein Event, eine Ein-/Ausgabe etc. blockiert.
2. Der Thread schickt ein Signal an ein Objekt (z.B. ein Up-Signal an einen Semaphor).
3. Die Zeitscheibe des laufenden Threads ist abgelaufen.

In den ersten beiden Fällen läuft der Thread bereits im Kernmodus. Jedoch ist es ihm im 1. Fall, im Gegensatz zum Fall 2 unmöglich weiterzulaufen und der Scheduler muss aufgerufen werden, um den nachfolgenden thread auszuwählen und zu starten. Im 2. Fall muss der Scheduler ebenfalls aufgerufen werden, da es sein kann, dass durch das Signal ein anderer Thread mit höherer Priorität fortgesetzt werden kann.

Im 3. Fall erfolgt durch den Scheduleraufruf unmittelbar der Wechsel in den Kernmodus. Jedoch ist es in Abhängigkeit von den wartenden Threads auch möglich, dass derselbe Thread eine neue Zeitscheibe erhält und fortgesetzt wird. Andernfalls erfolgt ein Threadwechsel.

Der Scheduler wird auch noch unter zwei weiteren Bedingungen aufgerufen:

1. Eine Ein-/Ausgabe wird beendet.
2. Eine bestimmte Wartezeit läuft ab.



Der Scheduler unter Windows hat 32 Prioritäten, die von 0 bis 31 durchgezählt werden, siehe Abbildung 76. Um die Prioritäten für das Scheduling zu benutzen, stellt das System eine Liste mit 32 Einträgen zur Verfügung. Jeder Eintrag in dieser Liste verweist auf den Kopf einer weiteren Liste mit allen wartenden Threads der entsprechenden Priorität. Der Grundalgorithmus besteht darin, die Liste von Priorität 31 aus abwärts zu durchsuchen. Sobald ein nicht leerer Eintrag gefunden wird, wird der Thread, der ganz oben auf dieser Liste steht, ausgewählt und darf für eine Zeitscheibe laufen. Nach Ablauf der Zeitscheibe wird er ans Ende der Liste auf seiner Prioritätsstufe geschrieben und der Thread ganz vorne wird ausgewählt.

Das Scheduling geschieht durch Auswählen eines Threads, ohne Rücksicht darauf, zu welchem Prozess er gehört.

Abbildung 76.: Windows unterstützt 32 Prioritäten

8.4.3. Starten von Windows

Bevor Windows gestartet werden kann, muss es gebootet werden. Der Boot-Prozess erzeugt den Initialprozess, der das System hochfährt. Der **Hardware-Boot-Prozess** beinhaltet das Lesen und Ausführen des Boot-Programms des ersten Sektors der ersten Festplatte (**der Master-Boot-Sektor**).
Dieses kurze Maschinenprogramm liest die Partitionstabelle, um daraus zu ermitteln, welche Partition das bootfähige Betriebssystem beinhaltet. Wurde die Partition mit dem Betriebssystem gefunden, wird der erste Sektor dieser Partition, Boot-Sektor genannt, gelesen und das darin befindliche Programm gestartet.

Das Programm im Boot-Sektor liest das Wurzelverzeichnis der Partition auf der Suche nach der Datei **ntldr**. Wird diese Datei gefunden, wird sie in den Hauptspeicher geladen und ausgeführt. **ntldr** lädt Windows. Für den Boot-Sektor gibt es verschiedene Versionen in Abhängigkeit vom verwendeten Dateisystem. Die korrekte Version des Master-Boot-Sektors und des Boot-Sektors wird bei der Installation von Windows auf die Festplatte geschrieben.

ntldr liest anschließend eine Datei boot.ini, die die einzelnen Konfigurationsinformationen beinhaltet, die nicht in der Registry gespeichert sind. Sie listet alle Versionen von **hal.dll** und **ntoskrnl.exe** auf, die zum Booten in dieser Partition zur Verfügung stehen. Anschließend werden diese gefundenen Dateien geladen, ebenso **bootvid.dll**, den Standard-Videotreiber, um während des Boot-Prozesses auf die Ausgabe schreiben zu können.

In der Registry werden alle Treiber verschiedenster Geräte die zum Abschluss des Boot-Prozesses benötigt werden, gesucht und ebenfalls geladen. Am Ende wird die Steuerung an **ntoskrnl.exe** übergeben.

Nach dem Start des Betriebssystems wird dieses allgemeine Initialisierungsroutinen ausführen und Komponenten aufrufen, die ihrerseits weitere Initialisierungen durchführen. Der letzte Schritt ist die Erzeugung des ersten richtigen Benutzerprozesses, des **Session-Managers**, **smss.exe**. Damit ist der Boot-Prozess abgeschlossen.

Bis zu diesem Zeitpunkt sind alle Prozesse reine Windows-Prozesse, die noch nicht auf die Win32-Umgebung zugreifen können, da es diese zu diesem Zeitpunkt noch nicht gibt. Erst durch **csrss.exe** wird diese Umgebung gestartet und zur Verfügung gestellt. Für die Anmeldung von Benutzern muss noch der **Anmeldedienst winlogon.exe** gestartet werden. Damit ist das Betriebssystem hochgefahren und gestartet. Typische Prozesse, die beim Hochfahren gestartet werden, sind in Tabelle 26 dargestellt. Die Prozesse über der Linie werden immer gestartet, die Prozesse unterhalb sind optional.

Prozess	Beschreibung
idle	Kein echter Prozess, sondern der Idle-Prozess
system	Erzeugt smms.exe und Auslagerungsdatei; lädt DLLs; liest Reg.
smss.exe	Erster Prozess; Initialisierung; erzeugt csrss und winlogon
csrss.exe	Win32-Prozesssubsystem
winlogon.exe	Login-Hintergrundprozess
lsass.exe	Authentifikationsdienst
services.exe	Durchsucht Registry und startet Dienste
Druckserver	Verschiedene Prozesse können den Drucker benutzen
Dateiserver	Behandelt entfernte Anfragen nach lokalen Dateien
Telnet Dämon	Erlaubt entfernten Login
Mail Server	Empfängt und speichert eingehende E-Mail
Fax Server	Empfängt und druckt eingehende Faxe
DNS Server	Internet Domain Name System
Ereignisprotokoll	Protokolliert verschiedene Systemereignisse
Plug-and-play-Manager	Überwacht die Hardware und reagiert auf Veränderungen

Tabelle 26.: Prozesse, die während des Hochfahrens gestartet werden

8.5. Speicherverwaltung

Windows besitzt ein sehr ausgeklügeltes **virtuelles Speichersystem**. Um es zu benutzen, bietet Windows eine Reihe von Win32-Funktionen an. Außerdem sind ein Teil der Executive und sechs **dedizierte Kernthreads** für die Verwaltung zuständig.

8.5.1. Konzepte

In Windows hat jeder Prozess seinen eigenen **virtuellen Adressraum**. Virtuelle Adressen sind 32 Bit lang, so dass jeder Prozess einen 4 Gbyte virtuellen Adressraum hat. Die unteren 2 Gbyte abzüglich ungefähr 256 Mbyte sind für den Programmcode und die Daten des Prozesses, die oberen 2 Gbyte bilden den Speicher des Betriebssystemkerns auf eine geschützte Art ab. Dieser virtuelle Adressraum wird bei Bedarf eingelagert mit einer festen Seitengröße (4 KByte auf einem Intel Pentium).

Das Aussehen des virtuellen Adressraums für drei Benutzerprozesse ist in einer leicht vereinfachten Form in Abbildung 77 gezeigt.

Die obersten und untersten 64 KByte jedes virtuellen Adressraums eines Prozesses werden nicht benutzt. Diese Entscheidung wurde absichtlich getroffen, um das Abfangen von Programmierfehlern zu erleichtern. Ungültige Zeiger sind oftmals alle Bits auf 0 oder 1.

Die wissen Bereiche in der Abbildung sind für jeden Prozess privat. Die schattierten Bereiche werden von allen Prozessen geteilt.

Jede virtuelle Speicherseite kann in einem von drei Zuständen sein:
- frei,
- reserviert oder
- belegt.

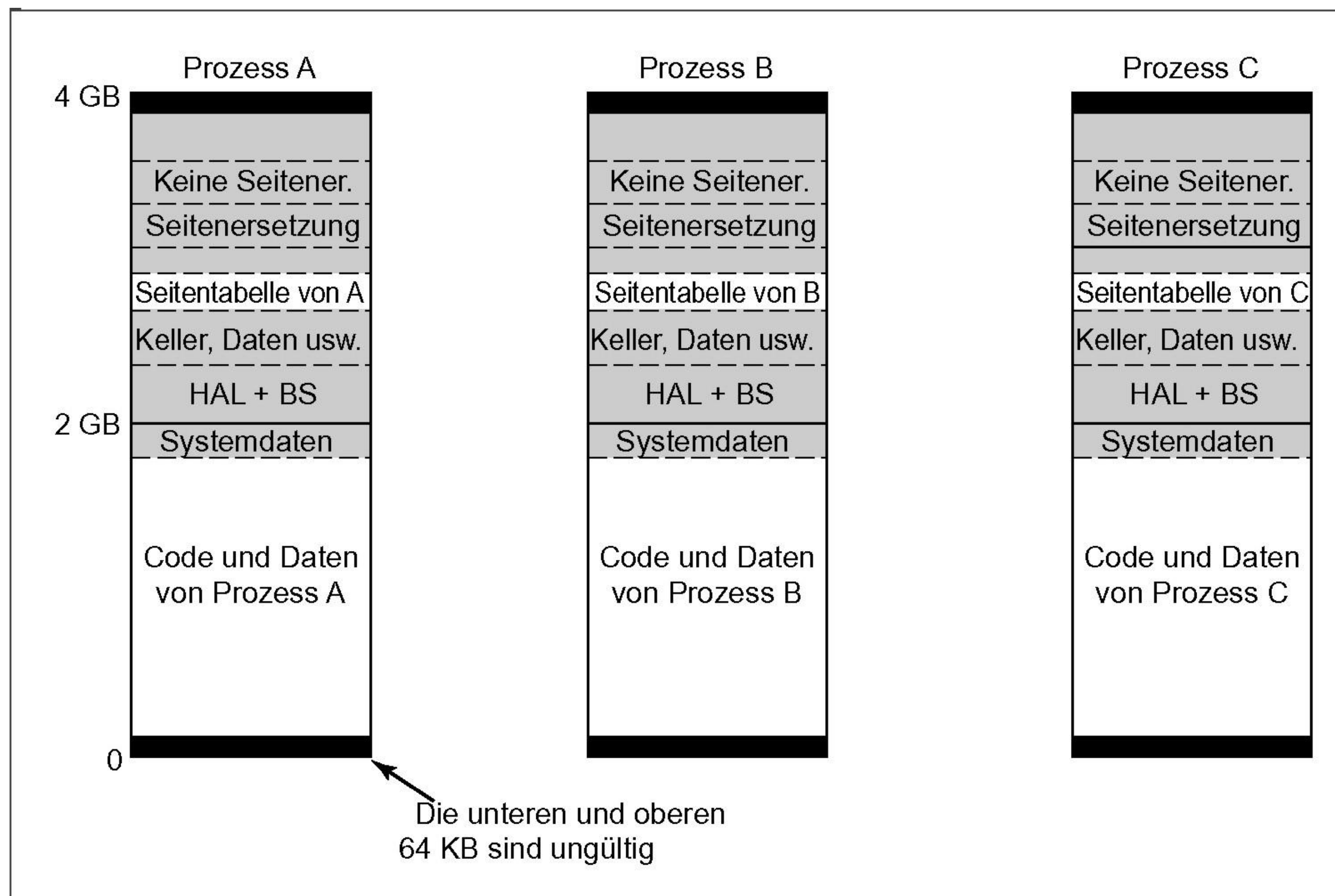


Abbildung 77.: Der virtuelle Adressraum für drei Prozesse

Eine **freie Seite** ist derzeit nicht in Gebrauch und ein Verweis darauf würde einen **Seitenfehler** verursachen. Beim Start eines Prozesses sind all seine Seiten so lange in diesem Zustand, bis das Programm und initiale Daten in seinen Adressraum abgebildet werden.

Sind erst einmal Programmcode oder Daten in eine Seite eingelagert, sagt man, die Seite ist **belegt**. Ein Verweis auf eine belegte Seite ist erfolgreich, falls sich die Seite im Hauptspeicher befindet.
Falls die Seite nicht im Hauptspeicher ist, tritt ein Seitenfehler auf und das Betriebssystem sucht die Seite und lädt sie von der Festplatte in den Hauptspeicher.

Eine virtuelle Seite kann sich auch im **reservierten** Zustand befinden, d.h., sie steht für Einlagerungen so lange nicht zur Verfügung, bis die Reservierung explizit aufgehoben ist.

Das Auslagern von Seiten erfolgt in Auslagerungsdateien pagefile.sys. Die Abbilder auf der Festplatte sind in einer oder mehreren solcher Auslagerungsdateien angeordnet. Es kann bis zu 16 Auslagerungsdateien geben, die für eine bessere Ein- / Ausgabe-Bandbreite über bis zu 16 verschiedenen Festplatten verteilt sein kann. Jede hat eine Anfangsgröße und eine maximale Größe, bis zu der sie später bei Bedarf wachsen kann.

Windows erlaubt es, Dateien direkt in Bereichen des virtuellen Adressraums abzubilden (d.h. Folgen von aufeinander folgenden Seiten). Wenn eine Datei erstmals auf den virtuellen Adressraum abgebildet ist, kann sie mit gewöhnlichen Speicherreferenzen gelesen und geschrieben werden. Im Speicher abgebildete Dateien sind auf die gleiche Art implementiert wie andere Seiten, nur dass sich die Abbilder in einer Benutzerdatei und nicht in der Auslagerungsdatei befinden.

Windows sieht es ausdrücklich vor, dass zwei oder mehrere Prozesse zur selben Zeit denselben Teil einer Datei einlagern, möglicherweise an unterschiedlichen virtuellen Adressen, wie in Abbildung 78 gezeigt.

Durch das Lesen und Schreiben von Speicherwörtern können die Prozesse so miteinander kommunizieren und Daten mit einer hohen Bandbreite austauschen, da kein Kopieren nötig ist. Dabei können verschiedene Prozesse unterschiedliche Zugriffsbeschränkungen haben. Da alle Prozesse, die eine eingelagerte Datei benutzen, sich dieselben Seiten teilen, werden Änderungen, die von einem Prozess vorgenommen werden, sofort für alle anderen Prozesse sichtbar, auch wenn die Datei auf der Festplatte noch nicht aktualisiert wurde.

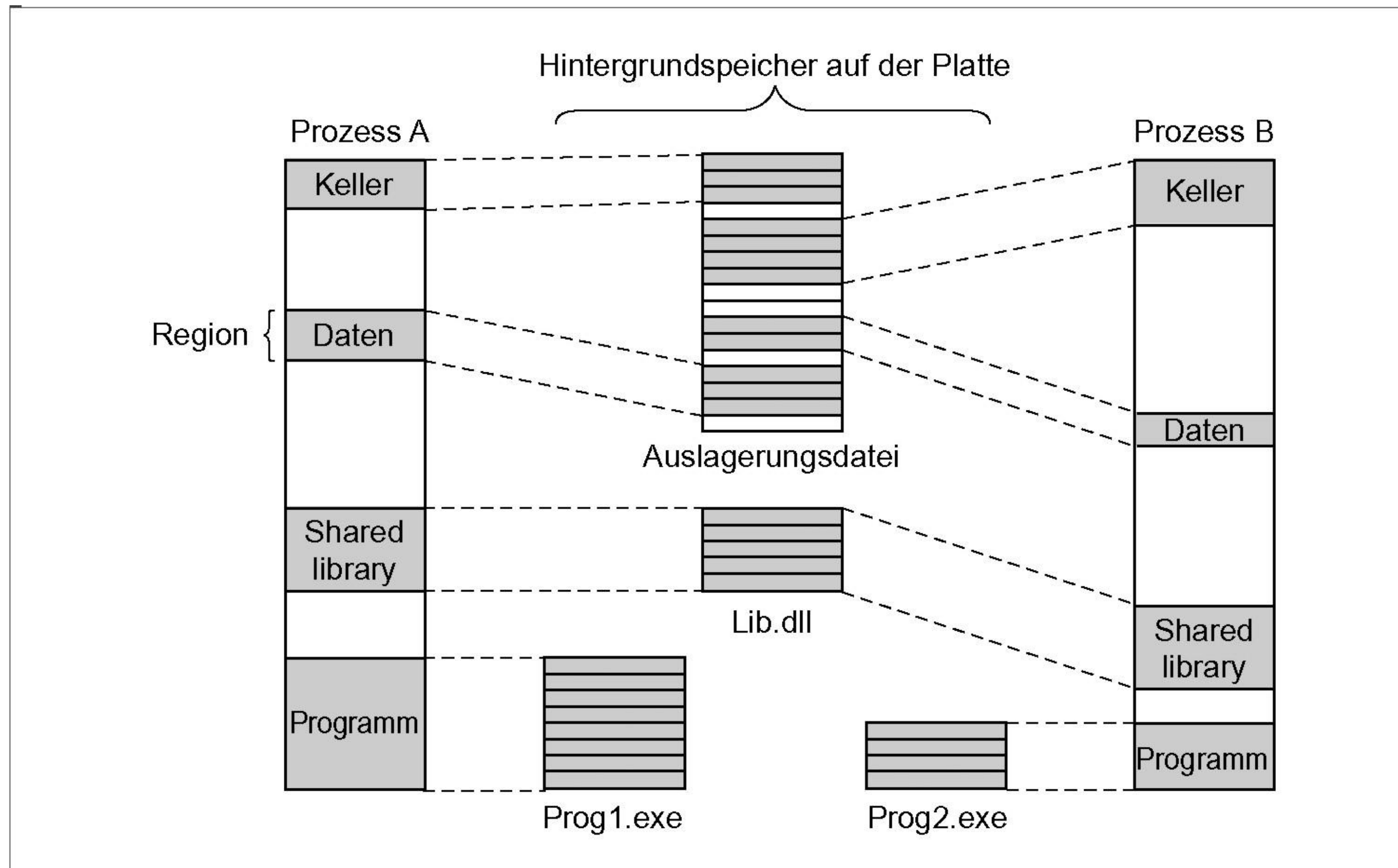


Abbildung 78.: Eingelagerte Bereiche mit ihren Abbildungen auf der Festplatte

8.5.2. Systemcalls zur Speicherverwaltung

Die Win32-API enthält eine Vielzahl von Funktionen, die es einem Prozess erlauben, seinen virtuellen Speicher explizit zu verwalten. Die aller wichtigsten dieser Funktionen sind in Tabelle 27 aufgelistet. Alle diese Funktionen arbeiten auf einem Bereich, der entweder eine einzelne Seite (4 KByte) oder eine Folge von zwei oder mehreren direkt aufeinanderfolgenden Seiten im virtuellen Adressraum umfasst.

Win32-API-Funktion	Beschreibung
VirtualAlloc	Reservieren oder Anbinden einer Region
VirtualFree	Freigeben oder Entbinden einer Region
VirtualProtect	Ändern der Lesen/Schreiben/Ausführen-Bits einer Region
VirtualQuery	Status einer Region abfragen
VirtualLock	Sperren einer Region, d.h. keine Auslagerung erlauben
VirtualUnlock	Auslagerung einer Region zulassen
CreateFileMapping	Dateiabbildung erzeugen und evtl. einen Namen zuweisen
MapViewOfFile	Datei (oder Teile) in den Adressraum abbilden
UnmapViewOfFile	Entfernen einer Dateiabbildung im Adressraum
OpenFileMapping	Öffnen einer bereits erzeugten Dateiabbildung

Tabelle 27.: Win32 API-Funktionen zur virtuellen Speicherverwaltung

8.6. Ein-/Ausgabe

Das Ziel des Windows Ein- / Ausgabesystems ist es, einen Rahmen bereitzustellen für die effiziente Behandlung eines sehr breiten Spektrums an E/A-Geräten. Die momentan bekannten und in Zukunft neu entwickelten Geräte müssen leicht in das System eingepasst werden können.

Der **Ein- / Ausgabe-Manager** ist eng verbunden mit dem **Plug-and-Play-Manager**. Die Grundidee hinter **Plug and Play** ist die eines aufzählbaren Busses. Es wurden viele Busse so entwickelt, siehe Abschnitt 1.5.4, damit der **Plug-and-Play-Manager** eine Anfrage an jeden Steckplatz schicken und das dort befindliche Gerät anweisen kann, sich zu identifizieren. Wenn er in Erfahrung gebracht hat, was dort los ist, kann der **Plug-and-Play-Manager** Hardwareressourcen, wie Unterbrechungsstufen, zuteilen, den passenden Treiber suchen und ihn in den Speicher laden. Für jeden geladenen Treiber wird ein Treiberobjekt erzeugt.

Einige Busse zählen nur beim Booten, wie der **SCSI-Bus**, andere Busse hingegen, wie **USB** oder **IEEE 1394**, können zu jeder Zeit zählen. Das erfordert einen sehr engen Kontakt zwischen dem **Plug-and-Play-Manager**, dem Bus-Treiber (der eigentlich die Aufzählung durchführt) und dem **Ein- / Ausgabe-Manager**.

Der **Ein- / Ausgabe-Manager** ist ebenfalls verbunden mit dem **Power-Manager**, der wiederum den Computer in einen der folgenden Zustände versetzen kann:

1. Voll funktionsfähig.
2. Sleep-1: CPU-Leistung reduziert, RAM und Cache arbeiten → sofortiges Aufwachen möglich.
3. Sleep-2: CPU und RAM arbeiten, CPU-Cache ist aus → Fortfahren vom momentanen Zustand.
4. Sleep-3: CPU und Cache aus, RAM arbeitet → Neustart von einer festgelegten Adresse aus.
5. Hibernate: CPU, RAM und Cache aus → Neustart aus einer auf der Platte gesicherten Datei.
6. Off: Alles aus → völliger Neustart ist nötig.

E/A-Geräte können ebenso verschiedene Zustände annehmen. Das Ein- und Ausschalten wird vom **Power-Manager** und dem **Ein- / Ausgabe-Manager** zusammen geregelt.

Man muss beachten, dass die oben benannten Zustände 2. bis 6. nur benutzt werden, wenn sich die CPU für eine kurze oder längere Zeit im Leerlauf befindet.

API Gruppe	Beschreibung
Fensterverwaltung	Erzeugen, Zerstören und Ändern von Fenstern
Menüs	Erzeugen, Zerstören und Ergänzen von Menüs
Dialogfenster	Dialogbox öffnen und Informationen sammeln
Zeichnen und Malen	Anzeigen von Punkten, Linien und geometrischen Figuren
Text	Anzeigen von Text in spezieller Größe, Art und Farbe
Bitmaps und Icons	Verwalten von Bitmaps und Icons am Bildschirm
Farben und Paletten	Verwaltung der möglichen Farben
Zwischenablage	Informationsaustausch zwischen Anwendungen
Eingabe	Eingabe von der Maus und der Tastatur

Tabelle 28.: Win32 API-Gruppen für die Ein- / Ausgabe

Windows hat über 100 verschiedene APIs für das breite Spektrum der E/A-Geräte. Das wahrscheinlich bedeutendste ist das Grafiksystem, für das es Tausende von Win32-API-Aufrufe gibt. Eine kurze Zusammenfassung der Aufrufe ist in Tabelle 28 dargestellt.

Um sicherzustellen, dass Gerätetreiber gut mit dem Rest von Windows zusammenarbeiten, hat **Microsoft** das **Windows-Driver-Model** definiert, mit dem Gerätetreiber konform sein müssen. Darüber hinaus hat **Microsoft** außerdem eine Sammlung von Werkzeugen entwickelt, das den Entwicklern von Treibern dabei helfen soll, diese Konformität zu wahren.

Treiber, die als konform gelten sollen, müssen alle der folgenden Voraussetzungen erfüllen:

1. Behandlung von ankommenden Ein- / Ausgabe-Anfragen, die in einem Standardformat vorliegen.
2. Gleiche Objektbasiertheit, wie der Rest von Windows.
3. Plug-and-Play-Geräte müssen dynamisch hinzugefügt und entfernt werden können.
4. Zulassung des Power-Managements, wo es möglich ist.
5. Konfigurierbarkeit in Bezug auf den Verbrauch von Ressourcen.
6. Wiedereintrittsfähigkeit für den Einsatz auf Multiprozessorplattformen.
7. Portierbarkeit zwischen Windows 98 und Windows 2000.

Ein- / Ausgabe-Anfragen werden an die Treiber in Form eines standardisierten Pakets namens **IRP** (**I**/O **R**equest **P**acket) weitergereicht. Konforme Treiber müssen in der Lage sein, sie zu behandeln.

Ein Treiber darf in Windows die ganze Arbeit allein machen, wie es beispielsweise der Druckertreiber in Abbildung 79 macht. Auf der anderen Seite dürfen Treiber aber auch geschachtelt sein. Dass bedeutet, dass eine Anfrage eine Folge von Treibern durchläuft, wobei jeder einen Teil der Arbeit verrichtet. Zwei geschachtelte sind ebenfalls in Abbildung 79 dargestellt.

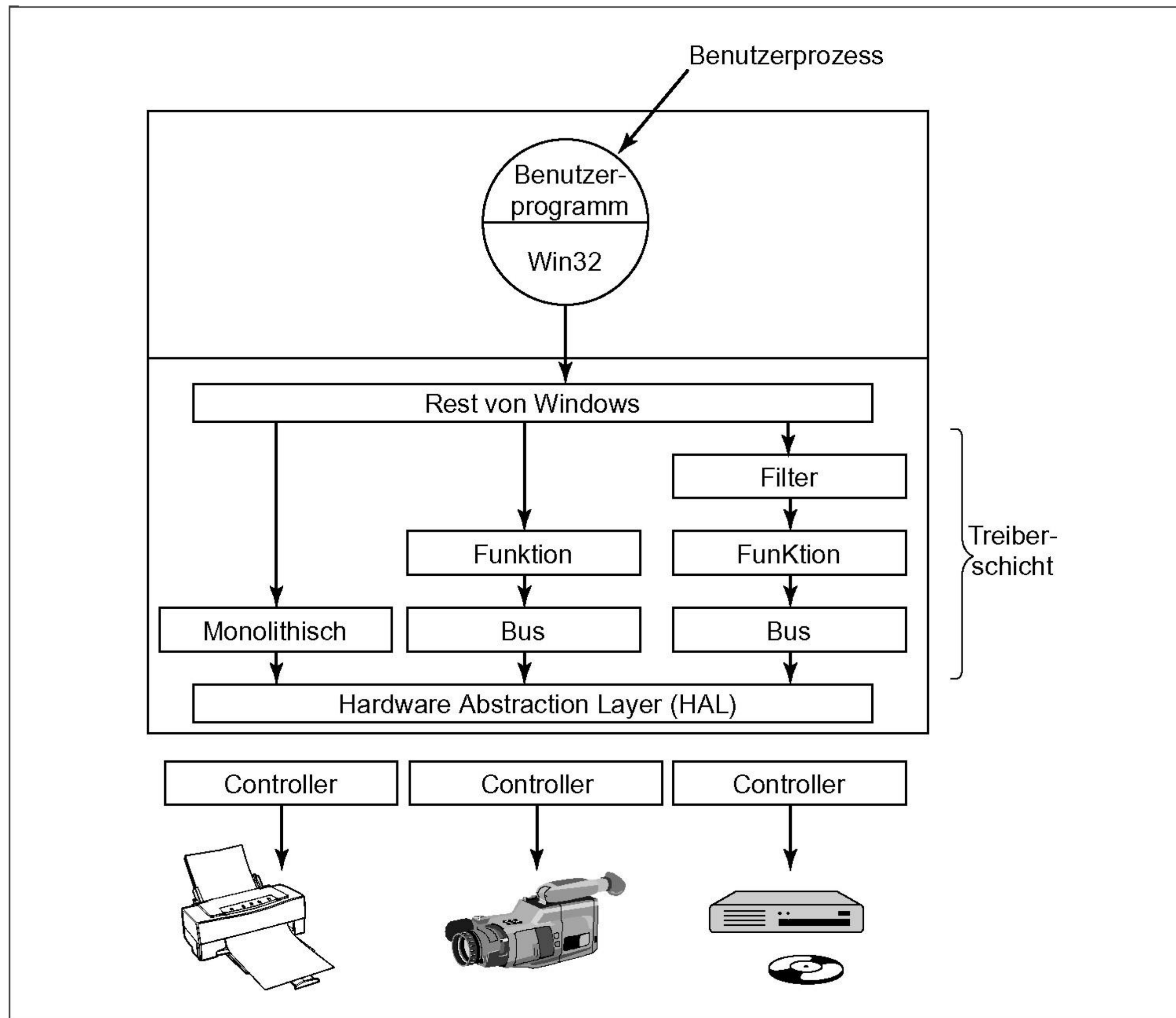


Abbildung 79.: Gerätetreiber über mehrere Schichten

8.7. Das Dateisystem

Windows unterstützt mehrere Dateisysteme, die wichtigsten sind **FAT-16**, **FAT-32** und **NTFS**.

8.7.1. MS-DOS

Das MS-DOS Dateisystem orientierte sich sehr stark an dem CP/M Dateisystem. Dies schließt die Verwendung der 8+3 (großbuchstabigen) Dateinamen mit ein. Die erste Version (MS-DOS 1.0) war sogar auf nur ein Verzeichnis, wie bei CP/M, beschränkt.

Mit MS-DOS 2.0 wurde die Funktionalität des Dateisystems stark erweitert. Die größte Verbesserung stellte ein Hierarchisches Dateisystem dar, welches Unix schon Jahre zuvor eingeführt hatte. Damit konnte das Wurzelverzeichnis (welches immer noch eine feste Größe hatte) Unterverzeichnisse haben.

Obwohl MS-DOS-Verzeichniseinträge, wie die von CP/M, von variabler Länge sind, benutzen sie eine feste Länge von 32 Byte. Das Format eines MS-DOS-Verzeichniseintrages ist in Abbildung 80 abgebildet. Ein überraschend großer Teil des Eintrages (10 Byte) ist ungenutzt, obwohl es Einträge gibt, die zwangsläufig zu Problemen führen:

- 2 Byte = 16 Bit für die Zeit = 65536, ein Tag hat aber 86400 Sekunden → kleinste Zeiteinheit = 2 Sekunden und
- 7 Bit für Jahreszahl, beginnend bei 1980 ist damit bei 2107 Schluss der Jahresangaben.

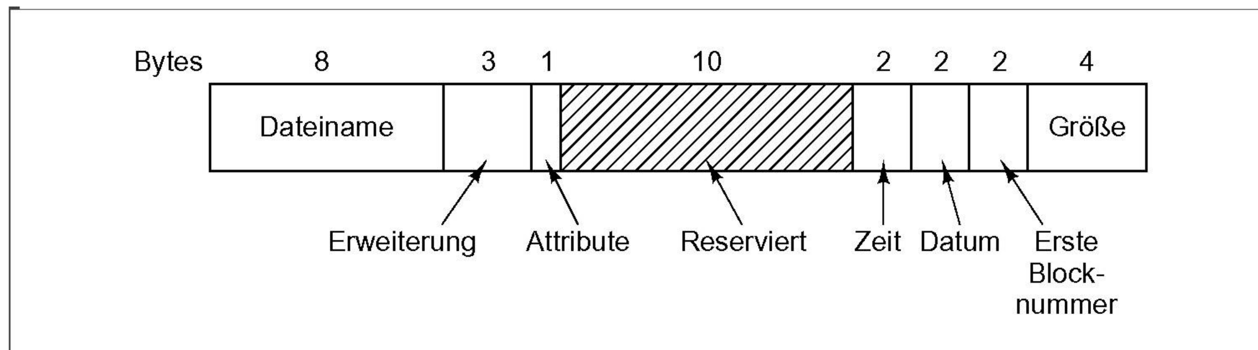


Abbildung 80.: MS-DOS-Verzeichniseintrag

8.7.2. Die FAT (File Allocation Table)

MS-DOS unterscheidet sich dennoch gravierend von CP/M, denn es speichert die Adressen der Plattenblöcke der Datei nicht in seinen Verzeichniseinträgen. Stattdessen hält **MS-DOS** die Informationen über die Plattenblöcke im Hauptspeicher in der **FAT** siehe 2.4.6. Realisierung von Dateien, Abbildung 39.

Das FAT-Dateisystem gibt es in drei Versionen für MS-DOS: **FAT-12**, **FAT-16** und **FAT-32**, abhängig von der Anzahl Bits, die für die Adresse eines Plattenblocks verwendet werden. In der Tat ist FAT-32 eine Fehlbenennung, da nur die niederwertigen 28 Bit benutzt werden.

Für alle FAT-Dateisysteme können die Plattenblöcke auf ein Vielfaches von 512 Byte gesetzt werden, mit einer unterschiedlichen Menge von erlaubten Blockgrößen (bei Microsoft Cluster genannt). So benutzte die erste Version von MS-DOS bei der FAT-12 512-Byte-Böcke und erlaubte eine maximale Partitionsgröße von 212 * 512 Byte = 2 Mbyte. Damit konnten maximal Disketten als Speichermedium verwendet werden. Aus diesem Grund waren Blockgrößen (Cluster) von 1 KByte, 2 KByte und 4 KByte zugelassen und damit wuchs die maximal Partitionsgröße auf 16 Mbyte.

Blockgröße	FAT-12	FAT-16	FAT-32
0,5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Abbildung 81.: Zusammenhang zwischen Blockgröße, FAT und maximaler Partitionsgröße

8.7.3. Das NTFS (NT File System)

Die einzelnen Dateinamen im NTFS sind auf eine Länge von maximal 255 Zeichen beschränkt, vollständige Pfadangaben sind auf 32767 Zeichen begrenzt. Die Dateinamen werden im Unicode angegeben, dadurch sind auch Dateinamen gültig, die nicht auf dem lateinischen Alphabet basieren. NTFS unterstützt auch die Unterscheidung zwischen Groß- und Kleinschreibung. Leider unterstützt aber die Win32-API diese Unterscheidung für Dateinamen nicht vollständig und schon gar nicht für Verzeichnisnamen, so dass dieser Vorteil verloren geht.

Eine NTFS-Datei ist nicht nur eine lineare Sequenz von Bytes. Stattdessen besteht eine Datei aus mehreren Attributen, von denen jedes durch einen Bytestrom repräsentiert wird. Die meisten Dateien haben nur kurze Byteströme, wie den Dateinamen und die 64-Bit-Objekt-ID, und einen langen unbenannten Strom, der die Daten enthält. Eine Datei kann aber auch mehrere lange Datenströme enthalten. Jeder Strom hat einen Namen, der sich folgendermaßen zusammensetzt: Dateiname, Doppelpunkt, Stromname (z.B. <dateiname>:<stromname>). Jeder Strom hat seine eigene Größe und kann unabhängig von den anderen Strömen gesperrt werden.

Die Idee von mehreren Strömen in einer Datei wurde von Apple Macintosh abgeschaut. Dieses Konzept wurde in NTFS eingebaut, so dass ein NTFS-Server in der Lage ist, Macintosh-Clients zu bedienen.

Neben der Kompatibilität zu Macintosh ist die Verwendung von mehreren Datenströmen für viele Anwendungen von Interesse, z.B. bei der Bildverarbeitung, oder der Textverarbeitung u.s.w.

Die maximale Datenstromlänge beträgt 2^{64} Byte. Dateizeiger werden verwendet, um zu verfolgen, an welcher Stelle in einem Strom ein Prozess gerade ist. Diese Dateizeiger sind 64 Bit lang, um die maximale Stromlänge von etwa 18,4 Exabyte adressieren zu können.

Jedes NTFS-Volumen (Partition auf der Festplatte) enthält Dateien, Verzeichnisse, Bitmaps und andere Datenstrukturen. Dabei ist jedes Volumen als eine lineare Sequenz von Blöcken organisiert, wobei die Blockgröße für jedes Volumen festgelegt ist. Je nach Größe des Volumens reicht die Blockgröße von 512 Byte bis zu 64 KByte. Die meisten NTFS-Festplatten benutzen 4 KByte Blöcke als Kompromiss. Blöcke werden durch eine 64-Bit-Adresse, einen Offset vom Beginn des Volumens an, referenziert.

Die wichtigste Datenstruktur in jedem Volumen ist die MFT (Master File Table), die eine lineare Sequenz von in der Größe festgelegten 1-Kbyte-Einträgen ist. Jeder MFT-Eintrag beschreibt eine Datei oder ein Verzeichnis. Er enthält die Attribute der Datei und eine Liste von Adressen, die angeben, wo auf der Festplatte die zugehörigen Blöcke stehen. Falls eine Datei extrem groß ist, kann es manchmal nötig sein, zwei oder mehrere MFT-Einträge zu verwenden, um die Liste von Blöcken verwalten zu können.

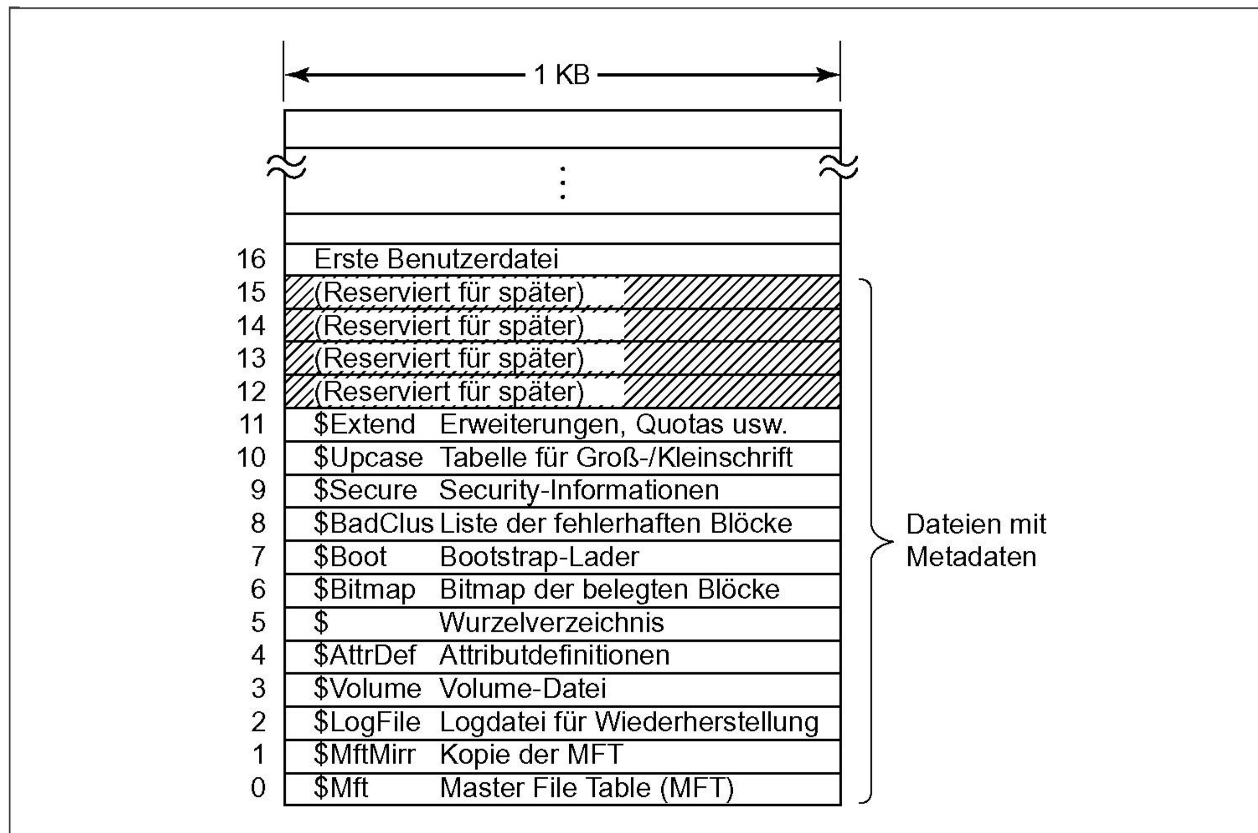


Abbildung 82.: Die NTFS-Master-File-Table

Die MFT ist in Abbildung 82 dargestellt. Jeder Eintrag besteht aus einer Folge von (Attributkopf, Wert) Paaren.

9. Echtzeitbetriebssysteme

Bereits in den Abschnitten 1.4.5. Echtzeit-Betriebssysteme oder 1.7.1.2. Echtzeitbetriebssysteme wurde auf grundlegende Eigenschaften von Echtzeitbetriebssystemen eingegangen.

9.1. Einleitung

Die Grenzen zwischen reinen Echtzeitsystemen und Informationssystemen ohne zeitliche Randbedingungen sind in der Regel fließend, siehe Tabelle 29. Manche Systeme scheinen auch bewusst in ihrer Verarbeitung verzögert zu arbeiten. So sollte z.B. eine Überweisung auf üblichem Bankenweg mit heutiger Rechenleistung nicht Tage dauern, sondern in Sekunden oder höchstens Minuten ablaufen. Für den Kunden wirkt sich dies so aus, dass er nach erfolgter Überweisung von seiner Bank auf eine Fremdbank oder umgekehrt nach etwa ein bis zwei Tagen mit der Abbuchung oder Gutschrift rechnen darf. Interessanterweise kommt die äußerst mangelhafte Echtzeitfähigkeit von Bankensoftware nahezu ausnahmslos den Banken zu Gute.

Informationssysteme	**Echtzeitsysteme**
datengesteuert	Ereignisgesteuert
komplexe Datenstrukturen	einfache Datenstrukturen
große Mengen an Eingangsdaten	meist kleine Mengen an Eingangsdaten
Ein- / Ausgabeintensiv	Rechenintensiv
maschinenunabhängig	Auf eine Hardwarearchitektur zugeschnitten

Tabelle 29.: Systemvergleich konventionell - Echtzeit

Die wichtigste Definition ist die allgemeine Definition von Echtzeitsystemen.

Definition 10 - Echtzeitsysteme:

~ erlauben die Verarbeitung von Daten innerhalb festgelegter und reproduzierbarer zeitlicher Toleranzen. Das bedeutet, ein ~ arbeitet in seinem zeitlichen Verhalten in vorherbestimmten Grenzen, die meist durch das technische System gegeben sind, deterministisch.

Ein Echtzeitsystem besteht gemeinhin aus der Hardware, einem Echtzeitbetriebssystem mit den für die Anwendung angepassten Bestandteilen und der Applikationssoftware. In der Applikation werden die vom Anwender gewünschten Aufgaben erfüllt.

Diese drei Komponenten müssen im Zusammenspiel das beschriebenen Antwortverhalten, d.h. den zeitlichen Determinismus garantieren. Hier gilt, je härter die zeitlichen Anforderungen sind, desto schneller muss die eingesetzte Hardware sein desto simpler muss das eingesetzte Betriebssystem bzw. die eingesetzte Software sein.

9.2. Grundlagen

Entsprechend den technischen Anforderungen besteht somit die Aufgabe darin, die vom technischen Prozess gegebenen Echtzeitanforderungen zu analysieren und ein passendes Echtzeitbetriebssystem in Kombination mit einem geeigneten Rechner auszuwählen.

Definition 11 - Echtzeitdatenverarbeitung:

~ muss von der Datenaufnahme (Input), über die Datenverarbeitung bis hin zur Datenausgabe (Output) stets die zeitlichen Anforderungen, die von realen Ereignissen bestimmt werden, erfüllen.

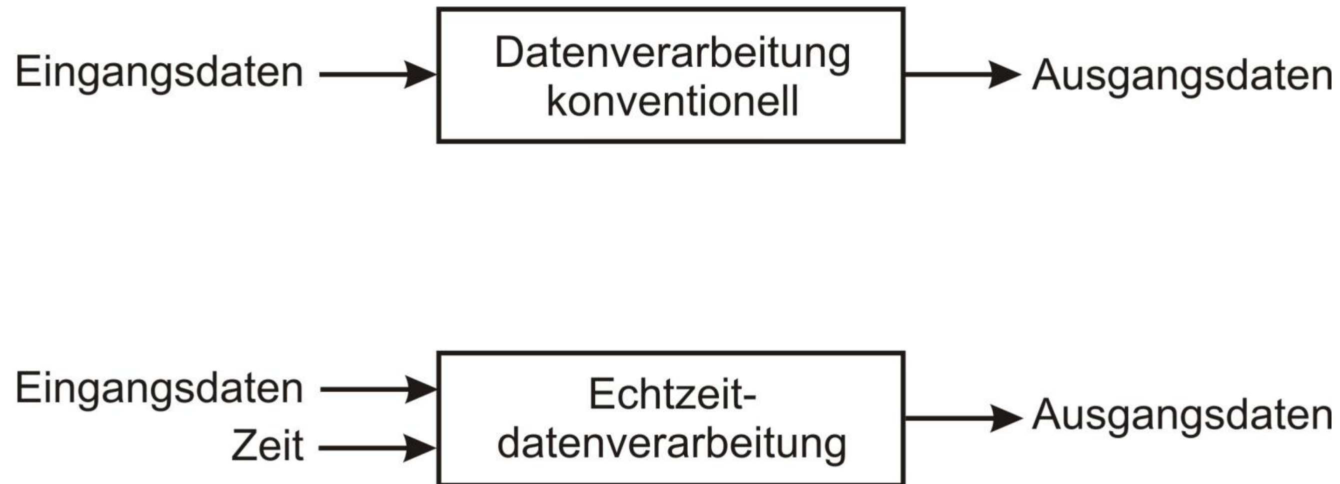


Abbildung 83.: Zeit als zusätzliche Eingangsgröße

Die Hardware, die in Echtzeitsystemen eingesetzt wird, zeichnet sich durch hohen Integrationsgrad und Kompaktheit aus. Zum Teil wird die Hardware bereits in das (technische) Prozesssystem eingebaut. Hierfür hat sich der Begriff **embedded systems** eingebürgert.

9.2.1. Rechtzeitigkeit

Der Begriff Rechtzeitigkeit auf die Datenverarbeitung angewandt bedeutet, dass Eingangsdaten rechtzeitig im Sinne von nicht zu spät zur Verfügung stehen müssen. Die Berechnung der Ausgabedaten muss ebenso rechtzeitig zu verwertbaren Ausgangsdaten führen. Treten im Verlauf der Bearbeitungskette zwischen Aus- und Eingangsdaten, bei einem technischen Prozess, Verzögerungen auf, so kann daraus ein Fehlverhalten resultieren. Dies ist darin begründet, dass die eingelesenen Daten zum Zeitpunkt der Ausgabe des Ergebnisses keine Gültigkeit mehr besitzen. Je nach den zeitlichen Anforderungen des technischen Prozesses oder allgemein der re-

alen Ergebnisse verlieren die Eingangsdaten unterschiedlich schnell ihre Gültigkeit. Das heißt die Eingangsdaten beschreiben den aktuellen Systemzustand nicht mehr korrekt.

Misst man zum Zeitpunkt t_0 die Temperatur eines Raumes zum Zweck der Reglung, so sollte zwischen dem Stellbefehl an das Reglerventil (Datenausgabe) zum Zeitpunkt t_1 und der Messung (Dateneingabe) t_0 nicht zwei Stunden liegen. Zwischenzeitlich könnte jemand das Fenster geöffnet haben, so dass die ursprüngliche gemessene Temperatur nicht mehr stimmt. Die Folge wäre ein falscher Stellwert und somit eine Fehlregelung.

9.2.2. Gleichzeitigkeit

Typischerweise müssen Echtzeitsysteme für technische Prozesse in der Regel mehrere Eingangsgrößen parallel auswerten. Der Grund liegt darin, dass reale Ereignisse sich nur mit einer Vielzahl von Daten beschreiben lassen.

Betrachtet man eine Kraftwerksregelung so laufen dort pro Sekunde mehrere tausend Daten ein, die zum Teil in Zusammenhang stehen. Ein Datenverarbeitungssystem muss dort in der Lage sein, mittels geeigneter Mechanismen die ankommenden Daten gleichzeitig zu verarbeiten. Die Erfüllung dieser Aufgabe verlangt nach Möglichkeiten der Parallelisierung von Berechnungsaufgaben und nach Unterscheidungsmöglichkeiten in verschiedene Wichtungen der Aufgaben. Die Hilfsmittel zur Lösung dieser Aufgaben sind typische Bestandteile und Eigenschaften von Echtzeitbetriebssystemen.

9.2.3. Determiniertheit

Die Rechtzeitigkeit und Gleichzeitigkeit bedingen, dass Berechnungsaufgaben zum Teil parallel, nach Wichtigkeit geordnet und im Normalfall unterbrechbar erledigt werden. Unterbrechbar bedeutet, dass Echtzeitsysteme sogenannte asynchrone Ereignisse erzeugen, auf die das Steuerungssystem auch asynchron reagieren muss.

Ein entsprechendes Echtzeitbetriebssystem muss über entsprechende Betriebsmittel verfügen. Die Parallelität, die Gewichtung der Aufgaben und die asynchrone Unterbrechungsmöglichkeit dürfen nicht dazu führen, dass die Rechtzeitigkeit verletzt wird. Da die Rechtzeitigkeit meist eine relative Größe ist, d.h. auf Zeitpunkte bezogen ist, wird eine Verallgemeinerung eingeführt, die besagt, dass ein Echtzeitsystem in vorgebbaren zeitlichen Schranken deterministisch arbeiten muss.

Definition 12 - Determiniertheit:

> **Ein Echtzeitsystem arbeitet zeitlich determiniert, wenn zu jeder Kombination von Eingangsgrößen die Reaktionszeit des Echtzeitsystems in festen zeitlichen Grenzen vorhersagbar ist.**

Die Gültigkeitsdauer von Eingangsdaten bzw. Änderungsgeschwindigkeiten von externen Ereignissen bestimmen die zeitlichen Deadlines der Echtzeitsysteme. Man unterscheidet:

9.2.3.1. harte Echtzeitsysteme

Hiermit sind Systeme wie z.B. Flugzeuge, Kraftwerke oder Werkzeugmaschinen gemeint, die harte zeitliche Schranken vorgeben. Die entsprechende Echtzeitdatenverarbeitung muss in jedem Fall sicherstellen, dass fest vorgegebene deadlines eingehalten werden, da es sonst zu schwerwiegenden Systemausfällen kommen kann. Hier können Verletzungen von Zeitvorgaben nicht toleriert werden. Das Datenverarbeitungssystem muss höchste Anforderungen an die Rechtzeitigkeit bei unter Umständen sehr hohem Datendurchsatz erfüllen können. Derartige Echtzeitsysteme müssen in der Regel 100% deterministisch arbeiten.

9.2.3.2. weiche Echtzeitsysteme

Hierunter fallen Systeme, die prinzipiell als Echtzeitsysteme zu betrachten sind, die aber sehr dehnbare Zeitlimits besitzen. Ein typisches Beispiel sind Bankterminals.

Dort steht der Datenverarbeitung der Mensch als technischer Prozess gegenüber. Da der Mensch aufgrund seiner Sinneswahrnehmung generell sehr langsam reagiert und Verzögerungen selbst im Sekundenbereich gut verkraften kann, sind die Anforderungen an das Datenverarbeitungssystem sehr gering. Die Erfüllung der Rechtzeitigkeit ist meist mit der unmittelbaren Kundenakzeptanz verbunden.

Das entsprechende Echtzeitsystem muss aber trotzdem deterministisch arbeiten, wobei die vorhersehbaren zeitlichen Grenzen sehr große Toleranzen aufweisen.

9.3. Elemente von Echtzeitsystemen

Der Betriebssystemkern steuert die Prozesse des Rechnersystems. Er realisiert das Ablaufen der Prozesse. Bei Single-Prozessor-Systemen bringt der Betriebssystemkern die einzelnen Prozesse so zum Ablaufen auf der CPU, dass der Eindruck entsteht, die Prozesse liefen quasi parallel ab. Innerhalb von Multi-Prozessor-Systemen verteilt der Betriebssystemkern die Prozesse auf die einzelnen Prozessoren.

Der Teil des Betriebssystemkerns, der die verschiedenen Prozesse steuert, nennt man Scheduler. Dieser Scheduler arbeitet mit einem so genannten Scheduler Algorithmus. Folgende gängige Scheduler Verfahren sind möglich:

1. Prioritätsbasiertes Scheduling
   Diese Strategie basiert auf den benutzerdefinierten Prozessprioritäten. Die Priorität repräsentiert dabei die Wichtigkeit der jeweiligen Aufgabe, die mit dem Prozess gelöst wird. Grundsätzlich kommt bei prioritätsbasierten Systemen immer der Prozess zur Ausführung, der zum Zeitpunkt der Einplanung (Scheduling) die aktuell höchste Priorität besitzt. Der Aufruf des Schedulers kommt entweder per Systemcall oder nach Interrupts vor.

2. Round-Robin Scheduling.
   Bei dieser Strategie ist das Ziel, die Leistung des Prozessors möglichst gleichmäßig auf alle Prozesse zu verteilen. Man spricht auch von so genannten gerechten Verfahren. Alle Prozesse besitzen die gleiche Priorität. Grundlage der Round Robin Technologie ist ein Zeitscheibensystem. Dabei wird jedem Prozess ein gleicher Teil der Zeitscheibe zugeteilt. Erst wenn alle Prozesse ihren Anteil an der Zeitscheibe an Bearbeitungszeit erhalten haben, wird der erste Prozess wieder gestartet.

In Betriebssystemen wie Unix oder Windows besitzt jeder Prozess einen eigenen Speicherbereich, der exklusiv reserviert wird. Dort erhält jeder Prozess seinen eigenen Datenbereich. Der Programmcode kann von mehreren Prozessen genutzt werden, siehe 2.2.1. Programme, Prozeduren, Prozesse und Instanzen.

In Echtzeitbetriebssystemen realisieren alle Tasks in demselben Speicherbereich. Damit wird die Prozessinterkommunikation sehr stark vereinfacht und der Kontextwechsel verläuft im Prinzip ohne besonderen Adressierungsoverhead. Der Nachteil liegt im fehlenden Schutz der Prozesse untereinander.

Die gemeinsame Nutzung von dem Programmcode durch mehrere Prozesse setzt eine wichtige Eigenschaft voraus. Der gemeinsame Code muss reentrant (wiedereintrittsfähig) sein. Das bedeutet, der gemeinsame Code muss über Mechanismen verfügen, die erlauben, zwar den Programmcode gemeinsam zu benutzen, den Zugriff auf Variablen und Daten jedoch für jeden Prozess getrennt zu verwalten. Folgende Vorkehrungen machen den Programmcode reentrant:

1. Die ausschließliche Verwendung von dynamischen Stackvariablen. Das bedeutet. Die aufrufende Taskfunktion übergibt die Variable als Zeiger auf den taskeigenen Speicher. Damit ist sichergestellt, dass im gemeinsam benutzten Programmcode keine Variablenkonflikte auftreten.
2. Schutz des gemeinsamen Programmcode durch Semaphore.

9.3.1. Taskmodell

Die Scheduling Algorithmen setzen voraus, dass sich die Prozesse bzw. Task in verschiedenen Zuständen befinden. Das Schema, das die Beziehungen zwischen den einzelnen Zuständen und die Überführungen zwischen den Zuständen beschreibt nennt man das Taskmodell, siehe Abbildung 31, Abbildung 32 oder Abbildung 84.

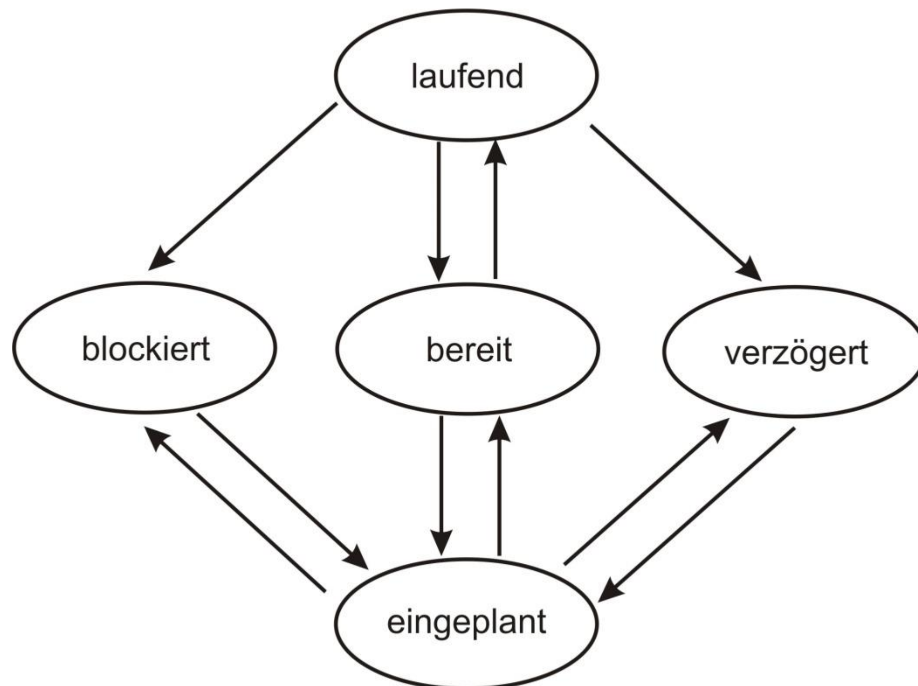


Abbildung 84.: Taskzustände im Taskmodell

Die einzelnen Zustände haben dabei folgende Bedeutung:
laufend Der Prozess ist dem Prozessor zugeordnet und wird aktuell bearbeitet.

bereit Der Prozess ist ausführbereit, alle Bedingungen sind erfüllt. Derjenige Prozess, der beim nächsten Scheduleraufruf die höchste Priorität besitzt und gleichzeitig im Zustand bereit ist, bekommt den Prozessor zugeteilt.

blockiert Der Prozess ist blockiert, er wartet auf ein Ereignis. Der blockierte Zustand wird dann angenommen, wenn zwei Prozesse synchronisiert werden sollen oder mehrere Prozesse auf ein Betriebsmittel zugreifen wollen. Man bezeichnet dies als Warte- oder Schlafzustand.

verzögert Der Prozess ist verzögert, er wartet eine bestimmte Zeit. Hierbei handelt es sich ebenfalls um einen Warte- oder Schlafzustand, der allerdings nur zeitliche Randbedingungen beinhaltet.

eingeplant Der Prozess ist eingeplant oder wurde neu erzeugt. Damit ist der Prozess dem System bekannt. Meistens kann der Prozess nicht unmittelbar in den laufenden Zustand überführt werden.

9.3.2. Semaphore

Semaphore wurden bereits im Abschnitt 2.2.6.3. Semaphore behandelt. Typischerweise werden sie auf Grund der Besonderheit von Echtzeitbetriebssystemen, dass Prozesse im gleichen Adressraum ausgeführt werden, zu deren Schutz verwendet. Im Allgemeinen unterscheidet man drei verschiedene Typen von Semaphoren:

1. Binäre Semaphore,
2. Ausschlusssemaphore (Mutexe) und
3. Zählsemaphore.

Semaphore können als systemglobale Variable angesehen werden. Die Erzeugung wird in der Regel durch einen Erzeugungsbefehl vorgenommen. Wichtig ist, dass die Semaphore als volle oder leere Semaphoren erzeugt werden können. Dieser Erzeugungszustand ist sehr wichtig, da die Wirkung und der Synchronisationsablauf ganz wesentlich vom Initialzustand der Semaphore bestimmt werden.

Grundsätzlich existiert eine Reihe von Funktionen zur Manipulation der Semaphore. Abhängig vom Zustand sind gewisse Operationen erlaubt. Wird eine Operation auf eine Semaphore ausgeführt, die sich aufgrund des Zustandes der Semaphore verbietet, führt dies automatisch zur Blockierung des Programmablaufs. Da in Echtzeitbetriebssystemen der Programmablauf stets in einem Prozesskontext stattfindet, wird der entsprechende Prozess ebenfalls blockiert. Werden mehrere Prozesse durch gegenseitig blockierte Semaphore ebenfalls blockiert, spricht man von einem **Dead Lock**.

9.3.3. Messages

Während Semaphore dazu dienen die Synchronisation der einzelnen Prozesse zu übernehmen, braucht es zusätzlich komplexere Mechanismen zur Kommunikation der einzelnen Prozesse untereinander. In Echtzeitbetriebssystemen dienen sogenannte **Message Queues** zur Interprozesskommunikation. Hierbei unterscheidet man zwischen der Kommunikation unter den Prozessen, die ausschließlich auf einem Prozessor ablaufen und der Interprozesskommunikation in Multiprozessorsystemen. Letzteres bedarf je nach System zusätzliche Hilfsmittel, die meist das Speicherverwaltungssystem betreffen.

Message Queues erlauben das Verschicken von einer beliebigen Anzahl von Mitteilungen mit beliebiger Länge. Dem Mechanismus liegt meist eine **FIFO** (**F**irst **I**n **F**irst **O**ut) Algorithmus zu Grunde. D.h. Mitteilungen, die zuerst geschickt werden kommen auch zuerst an. Die nachfolgenden Mitteilungen werden in eine Warteschlange eingetragen.

Grundsätzlich können mehrere Prozesse Mitteilungen über eine Message Queue verschicken bzw. mehrere Prozesse können über dieselbe Message Queue Mitteilungen empfangen. Zur Realisierung eines **full-duplex** Modus braucht es allerdings in der Regel zwei Message Queues, jeweils für eine Richtung. Die Abbildung 85 zeigt eine **full-duplex** Kommunikation über zwei Message Queues.

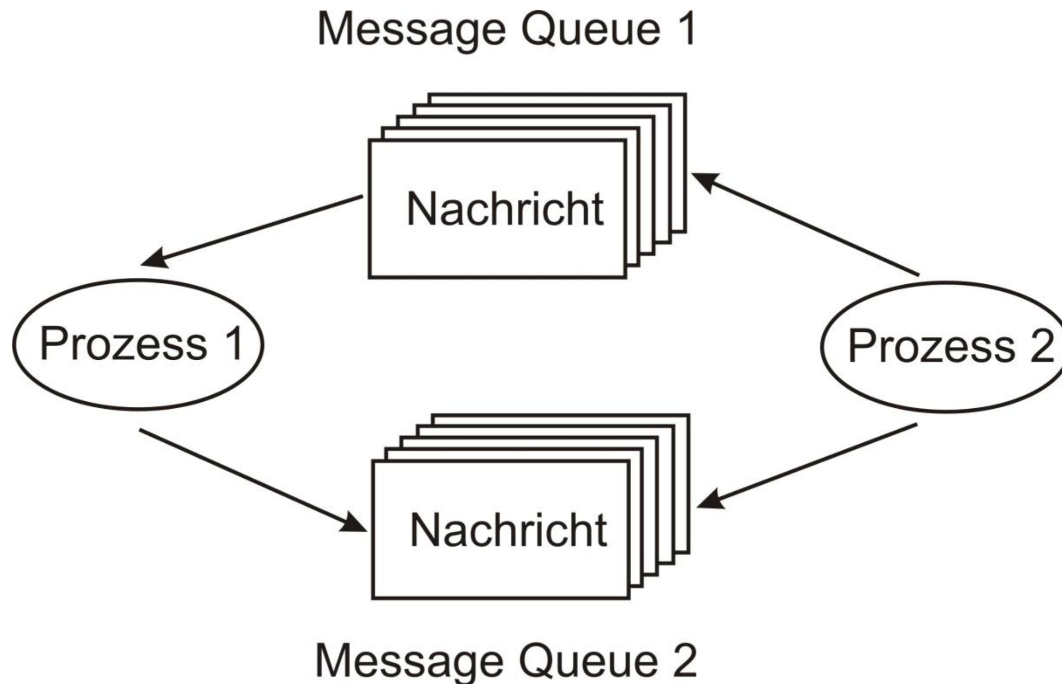


Abbildung 85.: Full Duplex Interprozesskommunikation mit Message Queues

9.3.4. Spezialisierte Warteschlangen (Pipes)

Sogenannte Pipes können als spezialisiertes Interface zu Message Queues verstanden werden. Pipes existieren im Betriebssystem Unix, wie auch in Echtzeitbetriebssystemen. Pipes sind dort sogenannte virtuelle Geräte.

Pipes werden analog zu Message Queues erzeugt, ausgelesen, beschrieben und bei Bedarf wieder gelöscht. Die Verwaltung geschieht durch das Betriebssystem. In Tabelle 30 sind die Unterschiede zwischen Message Queues und Pipes dargestellt.

Message Queues	Pipes
Möglichkeiten das Blockierungsverhalten zu beeinflussen	keine Beeinflussungsmöglichkeiten
Möglichkeiten die Mitteilungen zu priorisieren	keine Beeinflussungsmöglichkeiten
weniger Kommunikationsoverhead, deswegen schneller	langsamer
können bei Bedarf während der Laufzeit dynamisch entfernt werden	in der Regel ist die Entfernung erst nach Systemneustart möglich
Anzeigeroutinen sind meistens vorhanden	keine Anzeige
freier Zugriff, weniger formalisiert	nutzt die Standard E/A Schnittstelle des Betriebssystems
keine Möglichkeit	kann für das Umlenken des Standard E/A genutzt werden

Tabelle 30.: Unterschiede zwischen Message Queues und Pipes

9.3.5. Unterbrechungen und Signale

Der reguläre Betrieb des Echtzeitsystems kann von externen (**Interrupts**) oder internen (**Signalen**) Ereignissen unterbrochen werden. Interrupts sind im Prinzip elektrische Signale, die direkt oder über so genannte Interrupt Controller an die CPU gemeldet werden. Setzt man voraus, dass Echtzeitsysteme für kürzeste Reaktionszeiten konstruiert sind, so muss jeder externe Interrupt in der Lage sein, den aktuell ablaufenden Prozess zu unterbrechen. Die Zeit, die vom Anlegen des Interupts bis zum Start der entsprechenden **Interruptserviceroutine** vergeht, nennt man **Interruptlatenzzeit**. Sie wird oftmals zum Vergleich der verschiedenen Echtzeitsysteme herangezogen. Typische Interruptlatenzzeiten liegen bei modernen Echtzeitbetriebssystemen im Bereich um die 10 Mikrosekunden.

Signale sind interne Programmunterbrechungen, die grundsätzlich ebenfalls Prozesse unterbrechen. Damit Signale Prozesse unterbrechen können, muss innerhalb des Prozesses eine Art **Signalserviceroutine** (**Signal Handler**) installiert sein, die beim Eintreffen des passenden Signals aktiviert wird.

Signale sind im Prinzip **Softwareinterrupts**, die ein asynchrones Ereignis anzeigen. Der Signal Handler wird beim Auftreten des Signals ausgeführt. Nachdem die Signalserviceroutine abgelaufen ist, wird der Prozess an der Stelle der Unterbrechung fortgesetzt. Grundsätzlich stellen Signale die Möglichkeit dar, wichtige Operationen asynchron mit hoher Priorität zur Ausführung zu bringen.

9.3.6. Systemuhren und Zeitgeber

Spezifisch für Echtzeitsysteme sind die Einbindung von **Zeitgebern** in Anwendungsprogramme. Ferner sind Mechanismen vorhanden, um den Programmablauf zeitlich zu steuern. Wie im Abschnitt 9.2.1. Rechtzeitigkeit bereits dargestellt, hängt die Richtigkeit des Ergebnisses bei der Echtzeitdatenverarbeitung neben der logischen Richtigkeit ganz wesentlich von der Rechtzeitigkeit des Ergebnisses ab.

Bezüglich der Systemuhren und Zeitgeber kann man folgendes unterscheiden:

1. Systemuhr oder Systemzeitgeber (**System Clock**)
2. Hilfsuhr (**Auxiliary Clock**)
3. Alarmzeitgeber (**Watchdog Timer**)

Je nach Hardware findet man tatsächlich drei verschiedene Zeitgeber (Timer) oder man bedient zwei oder mehrere Zeitgeber mit einem **Timerbaustein**. Ein Timerbaustein besteht aus einem Quarz, der mit einer definierten Frequenz schwingt.

Tabellenverzeichnis

Tabelle 1.: Marktanteile im Smartphone-Geschäft 31
Tabelle 2.: Echtzeitbedingungen 45
Tabelle 3.: Prozesstabelle 55
Tabelle 4.: Signale in Unix 66
Tabelle 5.: typische Ein- / Ausgabegeräte und die Übertragungsraten 69
Tabelle 6.: typische Dateierweiterungen 76
Tabelle 7.: mögliche Dateiattribute 78
Tabelle 8.: Availability Environment Classification 97
Tabelle 9.: Unix-Systemcalls 112
Tabelle 10.: Wahrheitstabelle für UND-Verknüpfung 150
Tabelle 11.: Wahrheitstabelle für ODER-Verknüpfung 151
Tabelle 12.: Umgang mit Shell-Variablen 152
Tabelle 13.: Kommandosequenz ohne export 153
Tabelle 14.: Kommandosequenz mit export 154
Tabelle 15.: Beispiele für Umgebungsvariable 155
Tabelle 16.: test – Eigenschaften von Dateien 172
Tabelle 17.: test – Eigenschaften von Zeichenketten 173
Tabelle 18.: test – Vergleichsoperatoren 174
Tabelle 19.: logische Verknüpfungen 176
Tabelle 20.: Unterschiede zwischen Windows 98 und Windows NT 196
Tabelle 21.: verschiedene Versionen von Windows 2000 197
Tabelle 22.: Registrierung 199
Tabelle 23.: Allgemeine Objekttypen 202
Tabelle 24.: Basiskonzepte für CPU- und Ressourcenverwaltung 206
Tabelle 25.: Win32-Aufrufe zum Prozesssystem von Windows 209

Tabelle 26.: Prozesse, die während des Hochfahrens gestartet werden 213
Tabelle 27.: Win32 API-Funktionen zur virtuellen Speicherverwaltung 218
Tabelle 28.: Win32 API-Gruppen für die Ein- / Ausgabe 220
Tabelle 29.: Systemvergleich konventionell - Echtzeit 228
Tabelle 30.: Unterschiede zwischen Message Queues und Pipes 239

Abbildungsverzeichnis

Abbildung 1.: Schalen- bzw. Schichtenmodell eines Rechners 19
Abbildung 2.: Konrad Zuse 20
Abbildung 3.: Job-Verarbeitung in einem Mainframe 21
Abbildung 4.: IBM 7094 22
Abbildung 5.: Beispiel für einen Dateiserver 25
Abbildung 6.: Betriebssysteme für PCs 26
Abbildung 7.: Apple – iPhone der ersten Generation 27
Abbildung 8.: Gesundheitskarte ab 2011 31
Abbildung 9.: einfache Struktur eines Personal Computers 32
Abbildung 10.: von-Neumann-Architektur 33
Abbildung 11.: Prozessoren von AMD, Intel, Oracle-Sun und IBM, sowie für Smartphones 34
Abbildung 12.: Hauptspeicher für Laptop 35
Abbildung 13.: Diskettenlaufwerke im Wandel der Zeit 8“, 5¼“ und 3½“ 36
Abbildung 14.: Festplattenlaufwerke 36
Abbildung 15.: moderne SSD-Festplatte 37
Abbildung 16.: USB-Stick 16 GByte 37
Abbildung 17.: typische Bus-Struktur eines Personal Computers 38
Abbildung 18.: Mainboards für den Personal Computer 39
Abbildung 19.: Raspberry Pi, Version B 40
Abbildung 20.: Smart PC Stick 2.0 41
Abbildung 21.: Anschlußmöglichkeiten des Smart PC Stick 2.0 42
Abbildung 22.: Echtzeit 44
Abbildung 23.: verschiedene Multiprozessorstrukturen 46
Abbildung 24.: Single-Task-Betrieb 48
Abbildung 25.: Multi-Task-Betrieb 49

Abbildung 26.: IBM-PC....50
Abbildung 27.: Funktionen des Betriebssystemkerns....52
Abbildung 28.: Komponenten eines Prozesses....54
Abbildung 29.: Verwaltung von n Prozessen auf 1 Prozessor....57
Abbildung 30.: Erzeugung neuer Prozesse....59
Abbildung 31.: Taskmodell mit 3 Zuständen....61
Abbildung 32.: Taskmodell mit 5 Zuständen....62
Abbildung 33.: Beziehung zwischen Prozess und Thread....64
Abbildung 34.: Einsteckkarten....70
Abbildung 35.: DMA-Transfer....72
Abbildung 36.: Ablauf einer Unterbrechung (Interrupt)....73
Abbildung 37.: Zusammenhängende Belegung....81
Abbildung 38.: verkettete Listen....82
Abbildung 39.: FAT - File Allocation Table....83
Abbildung 40.: Speicherhierarchie....85
Abbildung 41.: Aufteilung des Hauptspeichers zwischen Betriebssystem und Anwendung....86
Abbildung 42.: feste Hauptspeicherbereiche mit verschiedenen Warteschlangenverwaltungen....88
Abbildung 43.: Swapping....89
Abbildung 44.: Reservierung von Hauptspeicherplatz....91
Abbildung 45.: Funktion der MMU....93
Abbildung 46.: Gründe für die Virtualisierung....99
Abbildung 47.: Vorteile der Virtualisierung....100
Abbildung 48.: Softwarevirtualisierung mit VMware....101
Abbildung 49.: Verbund mehrerer ESX-Server....102
Abbildung 50.: RedCluster auf einem ESXi-Server....103
Abbildung 51.: Ken Thompson und Dennis Ritchie....107
Abbildung 52.: historische Entwicklung....108

Abbildung 53.: Schalenmodell von Unix 110
Abbildung 54.: Nutzerzugriffe 111
Abbildung 55.: hierarchisches Unix-Dateisystem 113
Abbildung 56.: Montieren von physischen Dateisystemen 115
Abbildung 57.: Unterteilung einer Festplatte in Partitionen 118
Abbildung 58.: Inode-Verweisstruktur 122
Abbildung 59.: Prozesshierarchie 125
Abbildung 60.: Unix-Timesharing-Prinzip 127
Abbildung 61.: Prozesspriorität 128
Abbildung 62.: Unix-Taskmodell 129
Abbildung 63.: Historie der Unix-Shells 131
Abbildung 64.: Android 4.3 (Jelly Bean) auf dem Samsung Galaxy Nexus 135
Abbildung 65.: Die grundsätzliche Architektur von Android 137
Abbildung 66.: iPhone 5 142
Abbildung 67.: Syntaxdiagramm der Unix-Kommandos 147
Abbildung 68.: Syntaxdiagramm der Kommandoliste 147
Abbildung 69.: Syntaxdiagramm des Unix-Pipekonzeptes 148
Abbildung 70.: Syntaxdiagramme für UND und ODER 149
Abbildung 71.: Die Win32 API-Schnittstelle 198
Abbildung 72.: Windows 2000 Systemstruktur 201
Abbildung 73.: Die Struktur eines Objektes 204
Abbildung 74.: Die Beziehung zwischen Handle-Tabellen, Objekten und Objekttypen 205
Abbildung 75.: Beziehungen zwischen Jobs, Prozessen und Threads 207
Abbildung 76.: Windows unterstützt 32 Prioritäten 211
Abbildung 77.: Der virtuelle Adressraum für drei Prozesse 215
Abbildung 78.: Eingelagerte Bereiche mit ihren Abbildungen auf der Festplatte 217
Abbildung 79.: Gerätetreiber über mehrere Schichten 222

Abbildung 80.: MS-DOS-Verzeichniseintrag....224
Abbildung 81.: Zusammenhang zwischen Blockgröße, FAT und maximaler Partitionsgröße 225
Abbildung 82.: Die NTFS-Master-File-Table....227
Abbildung 83.: Zeit als zusätzliche Eingangsgröße 230
Abbildung 84.: Taskzustände im Taskmodell 235
Abbildung 85.: Full Duplex Interprozesskommunikation mit Message Queues....238

Definitionsverzeichnis

Definition 1 - Betriebssystem: 17
Definition 2 - Betriebssystemkern: 52
Definition 3 - Programm: 53
Definition 4 - Prozess: 53
Definition 5 - Instanz: 56
Definition 6 - Datei: 74
Definition 7 - Verfügbarkeit: 94
Definition 8 - Hochverfügbarkeit: 95
Definition 9 - Virtualisierung: 99
Definition 10 - Echtzeitsysteme: 229
Definition 11 - Echtzeitdatenverarbeitung: 229
Definition 12 - Determiniertheit: 232

Index

.profile 132
/dev/null 125
/etc/group 132
/etc/motd 131
/etc/passwd 132
/etc/profile 131
/etc/shadow 132
16-Bit-Intel-Maschinencode 195
32-Bit-Intel-Maschinencode 196
Access-Token 207
Ada Lovelace 20
Adapter 70
Andrew S. Tanenbaum 109
Android 28, 136
Anmeldedienst 213
Antwortverhalten 229
Anwendungsprogramm 18
apk-Paket 138
App 139, 143
Append 79
Apple 26, 29, 139
ARM-Prozessor 41
Assembler-Sprache 21
Asus 39
AT&T 106
Ausschlusssemaphore 236
Auxiliary Clock 240
Bada 28
bash 131
Batchbearbeitung 24
Batch-Job 58
Befehlssatz 33
Bell Laboratories 106
benannte Pipe 114
bereit 208, 236
Betriebssystem 17, 18, 20, 76
Betriebssystemkern 18, 52, 53, 74
Bill Gates 15, 83, 84, 194
Binäre Semaphore 236
BIOS 87
Blackberry 29
blockdevice 124
blockiert 62, 208, 236
blockorientierte Geräte 70
bootvid.dll 212
Cache-Bus 38
Cache-Speicher 85
CAD 43, 45
CAM 44
cd. 116
characterdevice 124
Charles Babbage 20
chgrp 117
chmod 117
chown 117
CISC 34

Client-Server-Modell 200
Close 79
Controller 70, 71
cp 123
CP/M 23, 195
CPU 33, 35, 128
Create 79
CreateProcess 60
CreateThread 208
csh 131
csrss.exe 213
CTSS 23
Daemon 58
Datei 74, 75, 76, 77, 79, 81
Dateiattribut 77
Dateioperation 79
Dateisystem 52, 74, 81
Datenkommunikation 65
Dead Lock 237
DEC 106
DEC-Alpha-Prozessor 34
dedizierte Kernthreads 214
Delete 79
Dennis Ritchie 107
Determiniertheit 232
Dienstprogramm 18
Digital Research 23
Digitalrechner 20
DMA 71
DOS 23, 194
down 67
down-Operation 67
E.W.Dijkstra 67
E/A-System 52
Echtzeitanforderung 229
Echtzeitbetriebssystem 44, 228, 229
Echtzeitfähigkeit 228
Echtzeitsystem 229
Ein- / Ausgabe-Manager 219
Ein- / Ausgabe-Port-Nummer 71
eingeplant 236
Ein-Prozessor-Betriebssystem 48
Elektronenröhre 20
embedded systems 230
EOF 65
Erzeuger-Verbraucher-Problem 65
erzeugt 65
execve 59
exit 60
exitProzess 60
FAT 83, 224
FAT-12 224
FAT-16 ..... 83, 223, 224
FAT-32 ..... 83, 223, 224
Fiber 206, 208
FIFO 237
Firewire-Bus 38
fork 59
full-duplex 237
Gerätedatei 114
Geräteunabhängigkeit 69
gerechtes Verfahren 234
Get Attributes 80
GID 116, 121, 132

GigaByte 39
Gleichzeitigkeit 231
Global Positioning System 136
GNU 109
Google 28, 136
Google Maps 139
Google Play 139
GPL 109
GUI 195
hal.dll 212
Hardware 198
Hardware-Boot-Prozess 212
Harvard Research Group 97
High Availability 95
Hochverfügbarkeit 94
Howard Aiken 20
IBM 22, 24, 194
IBM OS/360 87
IBM PC 23
IBM Systems /360 22
iCloud 143
Icon 58, 143
IDE-Bus 38
IEEE 94
IEEE 1394 .... 38, 219
Inode 84, 119
Intel 8080 23
Intel 8086 23
Intel-Pentium-Prozessor 34
Interrupt 69, 239
Interruptlatenzzeit 239
Interruptserviceroutine 239
iOS 29, 135
iPhone 140
iPhone OS 141
IRP 221
ISA-Bus 38
iTune 141, 143
Jelly Bean 135
Job 206
Jobbearbeitung 23
John Cocke 34
John von Neumann 20, 33, 45
JVM 31
Ken Thompson 106, 107
Kerndatenstruktur 203
kill 130
Konrad Zuse 20
ksh 131
laufend 208, 235
leichtgewichtiger Preudoparallelismus 208
Linkcounter 121
Linus Torvalds 109
Linux 25, 26, 109, 135
ln. 123
lokaler Bus 38
ls. 117
LSI-Schaltungen 23
Lynx Real Time Systems 27
LynxOS 27
Mac OS X 141
Macintosh 26
Mainboard 38
Mainframe 21, 24

Mainframe-Betriebssystem 24
major device number 124
Master-Boot-Sektor 212
Memory-Mapped Inpot Output 71
Message Queue 237
MFT 87
Microsoft 23, 194, 221
Microware Systems Corp. 27
Mikrocomputer 23
Minix 109
minor device number 124
mkdir 123
mkfs 120
Mobiltelefon 135
Modul 17
Modularisierung 17
MS-DOS 23, 75, 194, 224
MS-DOS 1.0 223
MS-DOS 2.0 223
MULTICS 23
Multi-Prozessor-System 233
Multi-Task-Betriebssystem 48
Multitasking 195
Multitouchbildschirm 143
Multi-Touch-Bildschirm 140
Multi-User-Betriebssystem 51
Mutex 68
mv 123
Netbooks 135
nice 130
nohup 130
normale Datei 114
NTFS 223
ntldr 212
ntoskrnl.exe 212
Objekt-Handles 203
Objekt-Manager 203
Open 79
OpenGL 138
operating system 17
OS/390 24
OS9 27
Overlay 92
Paging 86, 93
Partition 118
PCI-Bus 38
Peripheriegerät 70
PID 125
Pipe 61
Plug and Play 219
Plug-and-Play-Manager 219
POSIX 109
Power-Manager 219
Prioritätsbasiertes Scheduling 233
Programm 53
Programmiersprache ADA 20
Programmiersprache C 108
Programmiersprache FORTRAN 21
Prozedur 53
Prozess 47, 53, 129
Prozesssynchronisation 65
Prozesssystem 43, 52
ps 130
pwd 116

QNX ... 27
QNX Software Systems ... 27
quasiparallel ... 126
quasi-parallel ... 47
RAM ... 85
Random-Access-Datei ... 77
Raspberry Pi ... 40
raw device ... 124
Read ... 79
rechenbereit ... 62
rechnend ... 62
Rechnersystem ... 16
Rechtzeitigkeit ... 230, 240
reentrant ... 234
regedit ... 200
regedit32 ... 200
Register ... 85
Registry ... 198
Relais ... 20
Rename ... 80
Retina-Display ... 142
RISC ... 34
rm ... 123
rmdir ... 123
ROM ... 87
root directory ... 84
Round-Robin Scheduling ... 234
Scheduler ... 233
Scheduler Algorithmus ... 233
Scheduling ... 210
schläft ... 65
SCSI-Bus ... 38, 219
Seattle Computer Products ... 23
Seek ... 80
Seitenfehler ... 215
Semaphore ... 67, 236
sequenziell ... 77
Session-Manager ... 212
Set Attributes ... 80
Signal ... 65
Signal Handler ... 240
Signale ... 239
Signalserviceroutine ... 240
Single-Prozessor-System ... 233
Single-Task-Betriebssystem ... 48
Single-User-Betriebssystem ... 51
Siri ... 142
sleep ... 130
Slice ... 118
Smart PC Stick 2.0 ... 41
Smartphone ... 27
SMS ... 139
smss.exe ... 212
Softwareinterrupt ... 240
Softwarevirtualisierung ... 101
Sohn-Prozess ... 126
Speicher-Bus ... 38
Speicherverwaltungssystem ... 52, 86
SQLite ... 138
SSD-Festplatte ... 37
Steve Jobs ... 140
SUN-Sparc-Prozessor ... 34
Super-Block ... 119
Swapping ... 86

Symbian ... 30
symbolischer Link ... 114
System Clock ... 240
System V Interface Definition Guide ... 112
Tablet ... 27
Task ... 47
Taskmodell ... 129, 235
Thread ... 63, 106, 206
Tim Paterson ... 194
Timerbaustein ... 240
Timesharing ... 23, 127
T-Mobile ... 141
Toshiba ... 36
Transaktionssteuerung ... 24
Transistor ... 21
UID ... 116, 121, 132
umask ... 117
Unix ... 23, 25, 59, 60, 61, 84, 106, 135, 206
Unterprogramm ... 53
up ... 67
up-Operation ... 68
USB ... 219
USB-Bus ... 38
USB-Stick ... 37
Vater-Prozess ... 126
verbraucht ... 65
Verfügbarkeit ... 94
Verfügbarkeitsklassen ... 95
verteiltes Betriebssystem. ... 47
Verzeichnis ... 114
verzögert ... 236
Virtual Machine Monitor ... 102
Virtualisierung ... 99
virtuelle Maschine ... 102
virtueller Adressraum ... 214
virtueller Speicher ... 92
virtuelles Speichersystem ... 214
vollständiges Programm ... 16, 17
wahlfreier Zugriff ... 77
Warte- oder Schlafzustand ... 236
Watchdog Timer ... 240
wc -l ... 61
Western Digital ... 36
Western Electric ... 106
who ... 61
Win32 API ... 197
Win32s ... 197
Windows ... 26, 60, 194, 195
Windows 2000 ... 196
Windows 3.x ... 198, 200
Windows 95 ... 195, 197, 198
Windows 98 ... 195
Windows Me ... 196
Windows NT ... 196, 200
Windows NT 4.0 ... 196
Windows NT 5 ... 196
Windows Phone ... 30
Windows-Driver-Model ... 221
Windows-Server ... 25
winlogon.exe ... 213
Write ... 79
Wurzelverzeichnis ... 114
WWDC ... 142
X Window System ... 139

Zählsemaphore.. 236
zeichenorientierte Geräte 69
Zeitaufteilungsverfahren..24
Zeitgeber...240