Zur Verfügung gestellt von ~Creepy~Mind~

Dieses Dokument untersteht dem Copyright
von
Prof. Dr. Karim Roger Kremer

Vorlesungsscript FH Friedberg *(Hessen)*

Skript zu Betriebssysteme

Prof. Dr. Karim Roger Kremer

21. März 2005

Inhaltsverzeichnis

1 Einführung 5
1.1 Aufgaben und Qualitätskriterien von Betriebssystemen 5
1.2 Aufbau von Betriebssystemen 7
1.3 Anwendungsbereiche von Betriebssystemen 9

2 Prozess-Verwaltung 13
2.1 Prozess-Modell 13
2.2 Aufgaben der Prozessverwaltung 14
2.3 Arbeitsmodi und Systemaufrufe 15
2.4 Prozess-Scheduling 16
2.5 Fallbeispiel: Unix-Prozessverwaltung 18
2.5.1 Prozess-Scheduling 18
2.5.2 Prozesserzeugung und Prozessbeendigung 20
2.5.3 Standardeingabe, -ausgabe, Pipes, Filter 21

3 Nebenläufigkeit in Betriebssystemen 25
3.1 Nebenläufigkeit 25
3.2 Nichtdeterminismus und Verklemmungen 25
3.3 Multithreading 26
3.4 Synchronisation nebenläufiger Prozesse 28
3.4.1 Gemeinsame Variablen 28
3.4.2 Schlossvariable 30
3.4.3 Semaphor 32
3.4.4 Monitor 34
3.4.5 Bedingungsvariablen 36
3.4.6 Barriere 38
3.4.7 Synchronisation über Nachrichten 38
3.5 Behandlung von Verklemmungen 40
3.5.1 Erkennen von Verklemmungen 41
3.5.2 Vermeiden von Verklemmungen 41
3.5.3 Verhindern von Verklemmungen 42

4 Dateisysteme 47
4.1 Strukturierung und Typisierung von Dateien 47
4.2 Dateizugriff und Dateioperationen 48
4.3 Implementierungs-Möglichkeiten eines Dateisystems 49
4.3.1 Konsekutive Blöcke 49
4.3.2 Verkettete Liste von Blöcken 50

4.3.3 Index einer verketteten Liste von Blöcke 51
4.3.4 Index-Knoten (I-Node) 51
4.4 Fallbeispiel: Unix-Dateisystem 52
4.4.1 Verzeichnisbaum und Pfad wichtiger Dateien 52
4.4.2 Pfadangaben, Heimatverzeichnis und elementare Kommandos 55
4.4.3 Masken 56
4.4.4 Zugriffsschutz von Dateiobjekten 57
4.4.5 Realisierung des Zugriffsschutzes 59
4.5 Fallbeispiel: Windows NT-Dateisystem (NTFS) 62

5 Speicher-Organisation und -Verwaltung 65
5.1 Anforderungen und Eigenschaften von Speichern 65
5.2 Speicherhierarchie und virtueller Speicher 67
5.3 Halbleiterspeicher 68
5.4 Arbeitsspeicher-Organisation 68
5.5 Arbeitsspeicher-Verwaltung 72
5.6 Cache-Organisation und -Verwaltung 78
5.7 Cache-Konsistenz und -Kohärenz 81
5.8 Magnetische Festplatten 85
5.9 RAID-Technik 87

6 Ein-/Ausgabesysteme 91
6.1 Aufgaben von Ein-/Ausgabesystemen 91
6.2 Treiber für Ein-/Ausgabegeräte 92
6.3 Einbindung von Gerätetreibern in das Betriebssystem 92
6.4 Ablauf einer Ein-/Ausgabe-Operation 93
6.5 Fallbeispiel: Ein-/Ausgabe bei IBM-kompatiblen PC's 93

Kapitel 1

Einführung

1.1 Aufgaben und Qualitätskriterien von Betriebssystemen

Allgemein versteht man unter einem Betriebssystem (Operating System) eine Menge von Programmen zur Steuerung der Arbeit eines Rechners. Das Betriebssystem erfüllt dazu folgende Teilaufgaben:

- Verwaltung der Betriebsmittel (Hardware, Dateisystem, Prozesse)
- Schnittstelle zwischen Anwendern/Anwendungen und Rechner
- Ablauf und Steuerung von Anwendungen

DIN 44300 definiert die Aufgaben eines Betriebssystems wie folgt:

Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Zugegeben: Diese Definition ist etwas lang, aber es lassen sich i.w. drei Aufgabenbereiche eines Betriebssystems ablesen:

Anpassung der Rechner an die Benutzerwelt: Hiermit ist die Abbildung der logischen Benutzerwelt auf die physikalische Rechnerwelt gemeint, die für die Benutzer transparent verläuft.

Organisation des Betriebsablaufs: Hiermit ist die gleichzeitige oder zumindest zeitlich abwechselnde Arbeit an mehreren Aufgaben gemeint, die im Sinne hoher Systemauslastung und kurzer Bearbeitungszeiten von Programmen erforderlich ist. Dazu müssen die Betriebsmittel verwaltet und zum Teil virtualisiert werden (virtueller Prozessor, virtueller Arbeitsspeicher) und die um die Betriebsmittel konkurrierenden Prozesse koordiniert werden.

Steuerung der Bearbeitung von Aufgaben durch Benutzer: Hiermit ist die Möglichkeit des Benutzers gemeint, seine Aufgabe in einer geeigneten

Sprache zu beschreiben (z.B. Skript-Sprache oder Job Control-Sprache). Neben dem Programm werden hier vor allem bei Großrechnern weitere Parameter wie Rechenzeit, Speicherbedarf und Dateien für die Ein- und Ausgabe angegeben. Dieser Teil hängt stark mit der verwendeten Betriebsart Dialog- oder Stapel-Betrieb zusammen und wird i.d.R. von Interpretern bearbeitet.

Die Qualität eines Betriebssystems wird in folgenden allgemeinen Kriterien gemessen:

Robustheit: Hierunter wird eine angemessene Reaktion auf Fehler in Programmen und in der Hardware verstanden.

Datensicherheit: Daten dürfen auch bei Fehlern in der Hardware nicht verfälscht werden oder nicht verloren gehen.

Datenschutz: Daten verschiedener Anwender und des Betriebssystems selber müssen auf unterschiedlichen Betriebsmitteln vor unberechtigtem Zugriff geschützt werden.

Effizienz: Der Eigenbedarf des Betriebssystems und seine Fähigkeiten müssen in einem vernünftigen Verhältnis zueinander stehen. Die Leistungsfähigkeit eines Rechners wird hauptsächlich durch die zeitliche und örtliche Verteilung der Arbeiten im Rechner durch das Betriebssystem bestimmt. Maßeinheiten für die Effizienz sind u.a. der Durchsatz, d.h die Anzahl bearbeiteter Aufträge pro Zeiteinheit, und die Antwortzeit, d.h. die Zeit zwischen Erteilung und Ende eines Auftrags.

Erweiterbarkeit: Hierunter wird die Eigenschaft verstanden mit veränderter Hardware und Software umgehen zu können, u.U. müssen neue Teile des Betriebssystems wie Hardware-Treiber hinzugefügt werden.

Skalierbarkeit: Dieses Kriterium ist bei Multiprozessorsystemen entscheidend. Hier soll die Leistungsfähigkeit des Gesamtsystems mit der Summe der Leistungsfähigkeit seiner Einzelkomponenten wachsen.

Übertragbarkeit: Ein Betriebssystem soll mit möglichst wenig Programmieraufwand auf neue Hardware, z.B. eine geänderte Prozessorarchitektur, angepasst werden können.

Im klassischen Sinne werden folgende Arbeitsbereiche eines Betriebssystems unterschieden:

Systemverwaltung: Die Systemverwaltung hat als Aufgaben die Zulassung und Verwaltung der Benutzer sowie die Installation und Wartung von Hardware und Software.

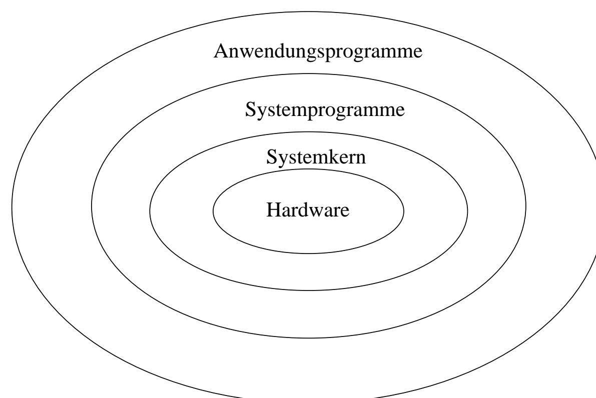
Auftragsverwaltung und Ablaufsteuerung: Durch diese Komponenten werden Benutzeraufträge entgegengenommen und ausgeführt. Realisiert wird dies mit Hilfe von Prozessen, die erzeugt, verwaltet und vernichtet werden. Ggfs. wird eine Kontingentierung und Abrechnung der verwendeten Betriebsmittel durchgeführt.

Betriebsmittelverwaltung: Das Betriebssystem verwaltet die Betriebsmittel Prozessor, Arbeitsspeicher, Sekundärspeicher durch verschiedene Techniken, damit sie für den Benutzer transparent verwendet werden können. Z.B. kann durch ein sogenanntes Time-Sharing-Verfahren der Prozessor an mehrere Prozesse abwechselnd vergeben werden, so dass ein Multi-Tasking-Betrieb möglich wird. Eine weitere Aufgabe der Betriebsmittelverwaltung betrifft das Vermeiden von Systemverklemmungen, die im Zusammenhang mit exklusiven Betriebsmitteln auftreten könnnen. Exklusive Betriebsmittel sind z.B. der Prozessor oder ein Drucker, der zu einer Zeit nur an einen Prozess vergeben werden darf.

Zugriffsüberwachung: Mit der Zugriffsüberwachung wird der Datenschutz realisiert. Bei unerlaubten Zugriffen, z.B. auf Arbeitsspeicherbereiche anderer Prozesse oder auf Dateien anderer Benutzer, wird eine Ausnahmebehandlung für den zugreifenden Prozess eingeleitet.

1.2 Aufbau von Betriebssystemen

Da Betriebssysteme aus komplexer Software bestehen, werden sie in einem hierarchischen Schichten- oder Schalenmodell aufgebaut. Dabei ist das Ziel, das Zusammenwirken der einzelnen Komponenten übersichtlich zu gestalten. Eine Komponente nimmt dabei nur Dienste von Komponenten der unmittelbar darunter liegenden Schicht in Anspruch. Auf der anderen Seite verkapseln die Komponenten einer Schicht die interne Realisation, d.h. um diese Komponenten nutzen zu können, muss der Aufrufer nur ihre Schnittstellen kennen. Ein Beispiel für ein solches Schalenmodell zeigt das folgende Bild:



Zum Systemkern gehört die Auftagsverwaltung und Ablaufsteuerung sowie die Betriebsmittelverwaltung und die Zugriffsüberwachung. Da die Teile des Systemkerns für den Betrieb eines Rechners notwendig sind, ist der Systemkern zur Ausführung im Arbeitsspeicher resident geladen.

Systemprogramme werden bei Bedarf nachgeladen; dies sind z.B. Standardbibliotheken bzw. Programme zur Benutzer- und Systemverwaltung sein.

Die Anzahl der Schichten im Schalenmodell kann nach Komplexität des Betriebssystems variieren, die innerste Schale bildet aber immer die Hardware die

äußerste Schale immer die Anwendungsprogramme, die beide nicht Teil des Betriebssystems sind.

Betrachtet man das Schichtenmodell genauer, so erkennt man eine Reihe von hierarchischen Schichten. Jede Schicht stellt der übergeordneten Schicht Dienste zur Verfügung; sie stellt eine Abstraktion der ihr untergeordneten Schicht dar. Implementiert wird eine Schicht auschließlich durch Aufrufe der unmittelbar untergeordenten Schicht; ein Zugriff auf tiefere Schichten ist verboten. Typischerweise sieht das Schichtenmodell, z.B. in Unix oder Windows NT wie folgt aus:

- Die Anwendung, z.B. ein C-Programm nutzt Funktionen der Standardbiblioteken beispielsweise für die Ein-/Ausgabe oder mathematische Dienste. Sie ruft Funktionen, wie printf(), scanf(), fopen(), fprintf(), fscanf(), fclose(), sin(), exp() usw., auf. Die Standardbibliothek sieht bei verschiedenen Betriebssystemen gleich aus. Dies ermöglicht die Portierung von Programmen durch einfache Neuübersetung.

- Die Funktionen der Standardbibliothek rufen ihrerseits Betriebssystemfunktionen über sogenannte Systemaufrufe auf. Systemaufrufe sind betriebssystemspezifisch; sie sind zwischen verschiedenen Betriebssystemen i.d.R. unterschiedlich. Für die Ein-/Ausgabe und Dateiverarbeitung sind das beispielsweise in Unix write(), read(), open() und close(). Neben den Funktionen der Standardbibliotheken benutzen Anwendungen häufig auch Systemaufrufe z.B. zur Prozessverwaltung in Unix fork(), exec() und wait(). Sie sind können dann nicht mehr durch Neuübersetzung portiert werden.

- Die Systemaufrufe bedienen sich der Basisfunktionen des Betriebssystems. Dazu gehören das Prozess-Steuerungssystem mit dem Scheduler, der Speicherverwaltung und der Realisation der Interprozesskommunikation sowie das Dateisystem.

- Unterhalb der Basisfunktionen liegt die Hardware-Abstraktions-Schicht. Sie kapselt Hardware-spezifische Merkmale wie den Prozessor-Typ, Cache, Bussysteme und die Peripherie. Der Vorteil des Einsatzes der Hardware-Abstraktions-Schicht ist, dass bei einer Änderung oder Anpassung des Betriebssystems auf eine andere Hardware lediglich die Teile dieser Schicht modifiziert werden müssen. Dies verbessert die Portabilität des Betriebssystems auf verschiedene Hardware. Als Schnittstelle zur Peripherie werden sogenannte Hardware-Treiber eingesetzt. Sie betreiben z.B. eine bestimmte Grafik-Karte oder eine bestimmte Festplatte und werden je nach installierter Hardware modular zum Betriebssystem hinzugefügt. Für die Basisfunktionen des Betriebssystems stellen sie eine einheitliche Schnittstelle verschiedener Hardware, z.B. unterschiedlicher Festplatten-Typen, zur Verfügung.

- Auf der untersten Schicht befindet sich die Hardware, die die geforderten Aktionen der Hardware-Abstraktions-Schicht umsetzt.

Anwendungsprogramme
Standardbibliotheken
Systemaufrufe
Basisfunktionen z.B. Prozess–Steuerung und Dateisystem
Hardware–Abstrakktions–Schicht z.B. Hardware–Treiber
Hardware

Der Betriebssystemkern kann als eine monolithische Sammlung von Funktionen oder als modulare Architektur implementiert sein. Bei der modularen Architektur werden die Dienste des Betriebssystems auf verschiedene Prozesse verteilt, die bei Anforderung aktiviert werden. Diese Prozesse nennt man Server-Prozesse. Anwendungen sind in diesem Sinne Client-Prozesse, die solche Server-Prozesse über Interprozess-Kommunikation (Inter Process Communication, IPC) aufrufen. Der Betriebssystemkern solcher modularer Betriebssysteme, wie z.B. Windows NT und Mach, enthält nur wenige unverzichtbare Funktionen, wie die Prozessor- und Speicherverwaltung. Andere Dienste, wie das Dateisystem und grafische Oberflächen werden durch Serverprozesse realisiert. Hierdurch ist der Systemkern weniger komplex als bei einem monolithischen Betriebssystem. Man spricht daher häufig von einem Mikrokern (Micro Kernel).

Die Vorteile der Mikrokern-Architektur liegen im klareren Design und in flexibleren Nutzungsmöglichkeiten gegenüber einem monolithischen Kern. Einen Nachteil stellt der zusätzliche Aufwand für die Kommunikation der Serverprozesse mit dem Mikrokern dar, der bei einer monolithischen Architektur entfällt. Dieser Zusatzaufwand kostet geringfügig Performance.

1.3 Anwendungsbereiche von Betriebssystemen

Nach ihrer Betriebsart können Einzel- und Mehrbenutzer-Betriebssysteme und Betriebssysteme für Einzel- oder Mehrprogramm-Betrieb unterschieden werden. Einige Repräsentanten solcher Betriebssysteme sind in der folgenden Tabelle zusammengestellt:

	Single User	Multiple User
Single Programming	MS-DOS, Windows 3.x	-
Multiple Programming	OS/2, Windows 9x	Unix/Linux, Windows NT

Diese Betriebssysteme sind sämtlich auf Personal Computern verfügbar und daher mehr oder weniger bekannt. Neben diesen Standard-Betriebssystemen gibt es eine ganze Reihe von Betriebssystemen für speziellere Hardware, z.B.

Großrechner, Vektorrechner, Parallelrechner, Echtzeitrechner für Prozesssteuerungen usw.

Bezüglich der Betriebsart unterscheidet man weiterhin Betriebssysteme mit:

Stapelverarbeitung (Batch Processing): Diese Betriebsart wird bei Großrechnern auch heute noch häufig angewendet. Der Benutzer spezifiziert in seinem Auftrag, welchen Betriebsmittelbedarf er hat z.B. Rechenzeit und Platz im Arbeitsspeicher. Dann übergibt der Benutzer seinen Auftrag (Job) dem System (früher in Form eines Lochkartenstapels -daher der Name-). Das Betriebssystem führt den Auftrag zu einem späteren Zeitpunkt aus. Durch die zeitliche Disponierbarkeit der Aufträge können die Betriebsmittel meist sehr gut ausgelastet werden.

Dialogverarbeitung (Time Sharing): Bei der Dialogverarbeitung kann der Benutzer den Rechenlauf interaktiv überwachen und steuern. Er bestimmt hier die Zeitpunkte zu denen bestimmte Aktionen durchgeführt werden. Dabei muss das System in einer angemessen kurzen Verzögerungszeit antworten. Die Auslastung des Systems steht als Optimierungskriterium weniger im Vordergrund als die Einhaltung kurzer Anwortzeiten.

Echtzeitverarbeitung (Real Time Processing): Die Automatisierung von technischen Einrichtungen, z.B. zur Produktion, erfordert eine möglichst schnelle Reaktion auf extern auftretende Ereignisse. Dabei werden enge Grenzen für die Antwortzeiten des Rechners gesetzt. Die Zeitdauer kann von bis zu 10 Sekunden, z.B. bei Auskunftssystemen, bis zu wenigen Millisekunden, z.B. bei elektrischen Antrieben, betragen. Je nachdem, ob der Prozess vom Rechner unmittelbar geregelt wird, oder ob ein Bediener das Ergebnis einer Rechnung zur Entscheidung für einen Eingriff benutzt, spricht man von closed loop- bzw. open loop-Steuerung.

Alle modernen Betriebssysteme, wie z.B. Windows NT, Windows 2000 und Linux sind mit Netzwerkfähigkeiten für den Bereich der lokalen Netze und auch für den Weitverkehrsbetrieb ausgestattet. Im Weitverkehrsnetz hat sich als allgemeiner Standard das Internet-Protokoll TCP/IP durchgesetzt. Auch in lokalen Netzen schreitet die Verbreitung von TCP/IP gegenüber herstellerspezifischen Protokollen, wie NetBEUI im Windows-LAN, voran. Der Hauptvorteil ist dabei die Verfügbarkeit von TCP/IP als einheitliches Protokoll für verschiedene Hardware und Betriebssysteme.

Durch die Vernetzung der Rechner hat sich ein weiteres Konzept stark verbreitet. Es gibt Dienst-erbringende und Dienst-fordernde Rechner und Prozesse, die man als Server und Client bezeichnet. Im strengen Sinn ist ein Server kein Rechner sondern ein Prozess, der auf einem Rechner arbeitet und Dienste für andere Client-Prozesse im Netzwerk erbringt. Typische Beispiele für Client und Server sind Browser und Webserver oder SQL-Client und SQL-Datenbankserver.

Die Vorteile der Vernetzung sind vielfältig:

- Kostensenkung durch bessere Nutzung von Hard- und Software, z.B. Druckerserver.
- Schnellere Bearbeitung von Daten, da sie dort bearbeitet werden können, wo sie anfallen. Anstatt die Daten zu den Programmen zu transferieren

können in einigen Fällen auch entfernt gespeicherte Programme verwendet werden.

- Breiteres Dienstleistungsangebot, z.B. Internet.
- Geringere Auswirkung von Systemausfällen durch redundante Rechner.

Die Vernetzung von Rechnern bringt jedoch auch Nachteile mit sich, z.B.:

- Kosten für den Kauf und Betrieb des Netzes und seiner Komponenten.
- Sicherheitsprobleme.
- Kommunikationsnetz als zusätzliche Fehlerquelle.
- Überlastung einzelner Server-Rechner.

Betriebssyteme sind i.d.R. für bestimmte Hardware-Plattformen realisiert und hierfür optimiert. Beispiele dafür sind Windows 98 und ME für Büro- oder Heim-PC's und Windows NT und XP für Client-PC's mit hoher Leistung und Server-PC-Systeme. Betriebssysteme aus der Unix-Familie können hingegen meist vom PC bis zum Großrechner eingesetzt werden, wobei natürlich der interne Aufbau des Betriebssystems an die Hardware-Architektur angepasst ist.

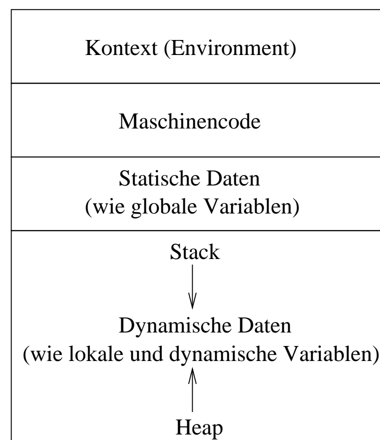
Kapitel 2

Prozess-Verwaltung

2.1 Prozess-Modell

Als Prozess wird ein Programm in seiner Ausführung bezeichnet. Wird das gleiche Programm von verschiedenen Benutzern ausgeführt, so sind dies verschiedene Prozesse.

Das logische Arbeitsspeicherbild eines Prozesses enthält die Maschinenbefehle, die durch Compiler, Assembler und Linker erzeugt wurden, sowie Bereiche für statische Daten, die von Beginn bis zum Ende des Prozesses vorhanden sind (z.B. globale Variablen und Konstanten), und für dynamische Daten, die während der Ausführung angelegt und ggf. wieder gelöscht werden (z.B. lokale Variablen und Rücksprungadressen von Unterprogrammen und dynamische Variablen). Während dynamische Variablen in einem sogenannten Heap untergebracht werden, sind die Übergabeparameter und Rücksprungadressen von Unterprogrammen in einem sogenannten Stack realisiert. Häufig werden bei der Prozesserzeugung noch Umgebungsinformationen, wie z.B. der Suchpfad für Programmdateien und eine Argumentliste übergeben. Dieser Prozesskontext wird in Unix und Windows als sogenanntes Environment bezeichnet.



Zur Ausführung benötigt ein Prozess verschiedene Betriebsmittel, wie z.B. einen Teil des Arbeitsspeichers, einen Prozessor, Dateien und Ein-/Ausgabe-Geräte wie Tastatur und Bildschirm. Die Zuordnung dieser Betriebsmittel übernimmt der Betriebssystemkern. Die hierfür notwendigen Daten befinden sich im sogenannten Prozessskontrollblock. Dazu gehören u.a.:

- Prozessidentifiator PID
- Priorität
- Prozesszustand
- Rechtedeskriptor z.B. für Dateien, Arbeitsspeicher usw.
- Dateideskriptor z.B. für die aktuelle Lese- oder Schreibposition und den Öffnungsmodus
- Arbeitsspeicherdeskriptor z.B. für die Zuordnung von Speichersegmenten zum Prozess
- Prozessorzustand z.B. Inhalt der Register, des Programmzählers und des Prozessorstatusworts
- Betriebsmittelnutzung für Abrechnungszwecke

Das Betriebssystem legt diesen Prozesskontrollblock bei der Prozesserzeugung an, es pflegt ihn, solange der Prozess aktiv ist, und es löscht ihn nach dem Prozessende. Die Menge aller Prozesskontrollblöche wird auch häufig als Prozesstabelle bezeichnet. Informationen über die Prozesstabelle können in Unix z.B. mit dem Kommando ps gewonnen werden.

2.2 Aufgaben der Prozessverwaltung

Prozesse sind aktive Programme, die vom Betriebssystem bearbeitet werden. D.h. jedoch nicht dass sie zu jeder Zeit im Prozessor rechnen. Mehrere Prozesse können auch bei einer Prozessor gleichzeitig aktiv sein, aber natürlich kann nur einer zu einer Zeit rechnen. Prozesse befinden sich also in zeitabhängig in verschiedenen Zuständen. Sie können z.B. rechnen, auf ein externes Ereignis wie eine Tastatureingabe oder einen Festplattenzugriff warten, oder einfach auf die Freigabe der Prozessor durch einen anderen rechnenden Prozess warten.

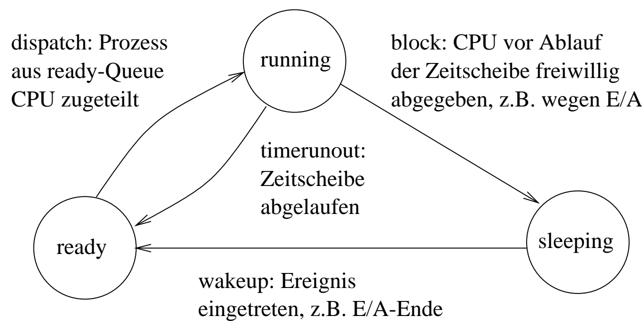
Ein Prozess kann sich im wesentlichen (technisch gibt es noch eine feinere Unterscheidung von Zuständen) in einem der folgenden drei Zustände befinden:

running (aktiv): Der Prozess wird gerade von der Prozessor ausgeführt.

sleeping (blockiert): Der Prozess wartet auf das Eintreten eines externen Ereignisses, z.B. E/A-Ende.

ready (bereit): Der Prozess kann die Prozessor nutzen, sie wird aber momentan durch einen anderen Prozess belegt.

Bei einer Prozessor kann nur ein Prozess zu einer Zeit aktiv sein, aber es gibt Warteschlangen bereiter und blockierter Prozesse. Der Übergang zwischen den wichtigsten Zuständen eines Prozesses wird durch folgendes Bild veranschaulicht. Es gibt noch weitere Prozesszustände, die aber für das Grundverständnis zunächst nicht wichtig sind.



Die Aufgabe des Betriebssystems ist, die Prozesse in ihren verschiedenen Zuständen zu überwachen und zu steuern. Hierbei werden nacheinander verschiedene Prozesse dem Prozessor zugeteilt. Man spricht von Prozesswechsel (Dispatching). Zentraler exekutiver Bestandteil für den Prozess-Wechsel ist das Unterbrechungssystem (Interrupt System). Von der Hardware werden Signale an das Betriebssystem geschickt, z.B. ein Zeit-Signal, das anzeigt, das die Zeitscheibe eines rechnenden Prozesses abgelaufen ist. Daraufhin suspendiert das Betriebssystem den Prozess, indem es den Prozesskontrollblock aktualisiert, und startet einen anderen rechenbereiten Prozess, indem es die Register und den Programmzähler des Prozessors aus dem zugeordneten Prozesskontrollblock lädt. Die Frage welcher Prozess aus mehreren rechenbereiten Prozessen ausgewählt wird betrifft die zentrale strategische Komponente der Prozess-Verwaltung, das Scheduling (Zeitplanung).

Eine weitere Aufgabe der Prozessverwaltung betrifft die Interprozesskommunikation. Hierbei werden Mechanismen geschaffen, damit zwei oder mehr Prozesse miteinander Daten austauschen können. Hierbei treten häufig Zeitprobleme auf, die durch geeignete Schutz- und Synchronisations-Mechanismen des Betriebssystems behandelt werden müssen.

2.3 Arbeitsmodi und Systemaufrufe

Der Zugriff auf die Hardwarekomponenten darf bei einem Multitasking- bzw. Multiuser-Betriebssystem nur durch das Betriebssystem und nicht durch die Anwendungsprogramme geschehen. Deswegen sind moderne Prozessoren mit (mindestens) zwei Arbeitsmodi ausgestattet. Im Benuter-Modus (User Mode) ist nur der Zugriff innerhalb des Adressraums eines Prozesses möglich. Hingegen sind im System-Modus (Supervisor Mode) auch privilegierte Befehle erlaubt, die z.B. notwendig sind um auf EA-Geräte zuzugreifen oder einen Prozesswechsel zu vollführen.

Der Aufruf von Programmteilen, die privilegierte Maschinenbefehle enthalten, geschieht vom Anwendungsprogramm aus über Systemaufrufe. Ein Systemaufruf ist also eine Funktion, die von einem Benutzerprozess aufgerufen wird,

um einen Dienst des Betriebssystemkerns zu aktivieren. Der Systemaufruf prüft die übergebenen Parameter und bildet hieraus eine Datenstruktur, die an den Systemkern weitergereicht wird. Danach wird ein spezieller Maschinenbefehl (Instruktion) ausgeführt, ein sogenannter Software Interrupt oder Trap, der dafür sorgt, dass der gewünschte Systemdienst aufgerufen wird. Bei der Ausführung des Systemdienstes wird der Zustand des Benutzerprozesses gerettet und es findet ein Wechsel in den System-Modus des Prozessors (Betriebssystems) statt, indem dann auch privilegierte Maschinenbefehle ausgeführt werden können, die auf die Hardware zugreifen können.

2.4 Prozess-Scheduling

Das Hauptproblem des Schedulings ist, dass jeder Prozess in seinem Verhalten nicht vorhersagbar ist, d.h. wie er den Prozessor und das I/O-System nutzt entscheidet sich erst zur Laufzeit. Damit können zum Verfolgen der Scheduling-Ziele i.a. keine exakten sondern nur heuristische Verfahren, die auf Erfahrungswerten beruhen, genutzt werden. Einige der Scheduling-Ziele sind:

Fairness: Jeder Prozess erhält einen gerechten Anteil der Betriebsmittel (Prozessor, I/O-System)

Effizienz: Die Betriebsmittel sind möglichst gut ausgelastet.

Antwortzeit: Die Wartezeit interaktiver Aufträge wird minimiert.

Verweilzeit: Die Wartezeit von Stapel-Aufträgen wird minimiert.

Durchsatz: Die Anzahl Aufträge pro Zeitintervall wird maximiert.

Diese Zielsetzungen schließen sich teilweise gegenseitig aus. Z.B. ist es im Sinne der Effizienz am besten einen Prozess möglichst wenig in seinem rechnenden Ablauf zu unterbrechen, dies widerspricht aber dem Ziel der Fairness.

Man unterscheidet Scheduling-Strategien mit langen Zuteilungsphasen des Prozessors (Stapelverarbeitung), sogenanntes Long Term Scheduling und Strategien kurzfristiger Zuteilung (interaktive Verarbeitung), sogenanntes Short Term Scheduling.

Vor allem Stapel-Betriebssysteme unterbrechen den Rechenlauf aus Effizienzgründen häufig nicht. Man nennt diese Systeme non preemptive (nicht unterbrechend). Auch Windows 3.x enthielt einen non preemptive Scheduler.

Eine einfache Strategie ist hierbei First-Come First-Served (FCFS), die mit Hilfe einer First In First Out (FIFO) Warteschlange realisiert werden kann. Das Problem von FCFS ist das die mittlere Wartezeit der Prozesse relativ hoch ist. Es folgt ein Beispiel für FCFS:

- Dauer der Berechnung: Prozess 1: 24s, Prozess 2: 3s, Prozess 3: 3s.
- 1. Schedule Prozess 1, Prozess 2, Prozess 3: mittlere Wartezeit $((0+24+27)/3)s = 17s$
- 2. Schedule Prozess 2, Prozess 3, Prozess 1: mittlere Wartezeit $((0+3+6)/3)s = 3s$

Eine Strategie, die die mittleren Wartezeiten minimiert, ist Shortest Job First (SJF). Diese Strategie ist auch bei Teilen von Prozessen anwendbar, wenn z.B. die Zeit für die nächste Rechenphase bekannt ist. In Stapel-Betriebssystemen werden die Rechenzeiten daher häufig bei der Spezifikation der Aufträge von den Benutzern abgeschätzt und dem System angegeben. Es folgt ein Beispiel für SJF:

- Dauer der Berechnung: Prozess 1: 6s, Prozess 2: 8s, Prozess 3: 7s, Prozess 4: 3s.
- SJF-Schedule Prozess 4, Prozess 1, Prozess 3, Prozess 2: mittlere Wartezeit $((0+3+9+16)/4)s = 7s$

Dialog-Betriebssysteme sind i.a. preemptive, da mehrere Prozesse aus Nutzersicht quasi gleichzeitig ablaufen müssen. Für das Wechseln der Prozesse ist aber Verwaltungsaufwand (Overhead) notwendig. Erst ab Windows 95 enthalten die Windows-Betriebssysteme einen unterbrechenden Scheduler.

Das Scheduling kann im einfachsten Fall nach der Round-Robbin-Methode geschehen. Hierbei bekommen alle rechenbereiten Prozesse reihum die Prozessor maximal für eine vorgegebene Zeitscheibe zugeteilt. Dabei können die Prozesse natürlich die Prozessor vor Beendigung der Zeitscheibe freiwillig z.B. wegen I/O abgeben. Die Dauer der Zeitscheibe beeinflusst entscheidend den Overhead durch den Prozesswechsel und die interaktiven Antwortzeiten. Kurze Zeitscheiben erzeugen mehr Overhead; lange Zeitscheiben erhöhen die Antwortzeiten. Es muss also ein Kompromiss für die Dauer der Zeitscheibe gefunden werden.

Die Round-Robbin-Methode geht davon aus, dass alle Prozesse mit dem gleichen Prozessor-Anteil versorgt werden müssen. Beim Scheduling mit Prioritäten gibt es Prozesse, die schneller ausgeführt werden sollen als andere; sie haben dann höhere Priorität.

Prioritäten können statisch von Anfang bis Ende eines Prozesses zugewiesen werden. Das Problem statischer Prioritäten ist das Verhungern von Prozessen mit niedriger Priorität, wenn immer wieder neue Prozesse mit hoher Priorität eintreffen oder lange hochpriore Prozesse vorhanden sind.

Daher werden praktisch meist dynamische Prioritäten vergeben, die bei wartenden Prozessen steigen. Moderne Betriebssysteme wie z.B. Unix (Linux) und Windows NT arbeiten mit einer Kombination aus Round Robbin-Zeitscheiben-Verfahren und dynamischen Prioritäts-Scheduling. Hierbei kann ein Prozess hoher Priorität, der seine Zeitscheibe komplett aufbraucht, in eine niedrigere Prioritätsklasse kommen, damit er den Prozessor nicht für sich monopolisiert. Wartende Prozesse steigen in ihrer Priorität und werden dann dem Prozessor zugeteilt.

Bisher haben wir einstufiges Scheduling von Prozessen, deren Adressraum im Arbeitsspeicher liegt, betrachtet. Der Arbeitsspeicher reicht aber häufig nicht aus, um alle Prozesse unterzubringen. Daher werden Teilmengen der Prozesse auf Sekundärspeicher (Festplatte) ausgelagert. Man spricht bei dieser Auslagerung von Prozessen von Swapping. Das Scheduling mit Swapping ist zweistufig. In der ersten Stufe werden die im Arbeitsspeicher befindlichen Prozesse dem Prozessor zugeteilt, während in der zweiten Stufe des Schedulings darüber entschieden wird, welche Prozesse von der Festplatte in den Arbeitsspeicher eingelagert (Swap In) und welche Prozesse vom Arbeitsspeicher auf die Festplatte ausgelagert (Swap Out) werden.

Das Swapping bestimmt den Grad des Multiprogrammings, d.h. die Anzahl der Prozesse, die um den Prozessor konkurrieren. Da die Ein- und Auslagerung zeitintensive Operationen sind, werden sie im Vergleich zum Prozessor-Scheduling selten durchgeführt. Beim Swapping wird versucht eine Mischung aus Ein-/Ausgabe-intensiven (interaktiven Prozessen) und rechenintensiven (Stapel-Prozessen) im Arbeitsspeicher zu halten, wodurch der Rechner insgesamt gut ausgelastet ist und verschiedene Prozessprofile gerecht bedient werden.

2.5 Fallbeispiel: Unix-Prozessverwaltung

2.5.1 Prozess-Scheduling

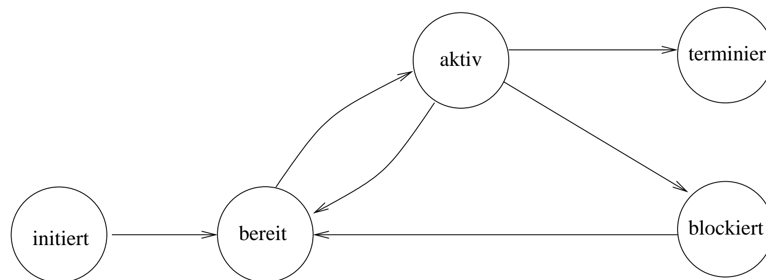
Bei Unix-Betriebssystemen wird die Prozessor nach einem Zeitscheibenverfahren an die Prozesse je nach System abwechselnd z.B. im Bereich von 10 bis 20 ms vergeben, so dass es für die Benutzer so aussieht als ob ihre Prozesse kontinuierlich in Arbeit sind. Wenn ein Prozess initiiert wird gelangt er zunächst in die ready-Warteschlange, von der ihm nach einer Wartezeit die Prozessor zugeteilt wird (Zustand running). Wenn der Prozess die zugeteilte Zeitscheibe in der Prozessor nun vollständig braucht, so wird ihm die Prozessor entzogen und er reiht sich wieder am Ende der ready-Warteschlange ein. Hat der Prozess die Prozessor vor Ablauf der zugeteilten Zeitscheibe abgegeben, z.B. weil er auf eine Tastatureingabe wartet, so kommt er in den Zustand sleeping. Von sleeping kann er durch ein externes Ereignis wieder in den Zustand ready, d.h. in die ready-Warteschlange überführt werden.

Zu jedem Prozess legt das System deen Prozesskontrollblock zur Verwaltung des Prozesses an. Dieser enthält folgende Informationen:

- Momentaner Prozesszustand
- Reale und effektive ID's des Prozess-Erzeugers
- Eindeutige Prozesskennzahl (PID)
- Prozesskennzahl des erzeugenden Elternprozesses (PPID für Parent PID)
- Priorität
- Reservierte Ressourcen (Dateien, Arbeitsspeicher etc.)

Diese Informationen werden u.a. für den Prozesswechsel der Prozessor benötigt. Nachfolgend wird der BSD-Scheduler beschrieben. Scheduler der System V Familie arbeiten ähnlich.

Die Abbildung zeigt die Zustände, die ein Prozess während seiner Ausführung annehmen kann. Mögliche Zustandsübergänge sind durch Pfeile gekennzeichnet. Die Übergänge zwischen den Zuständen bereit, aktiv und blockiert sind Aufgabe des Schedulers. Der Übergang aktiv nach blockiert wird als freiwilliger Kontextwechsel bezeichnet; Ursache ist z.B. eine Ein-/Ausgabeoperation. Der Übergang aktiv nach bereit wird erzwungener Kontextwechsel genannt; Ursache ist z.B. der Ablauf der Zeitscheibe eines rechnenden Prozesses.



Der Scheduler arbeitet nach dem Prinzip der Multilevel Feidback Queues, d.h. die Prioritäten werden fortlaufend neu berechnet. Anhand von Prioritäten von 0 bis 127 werden rechenbereite Prozesse in eine der Warteschlangen (Run Queues) des Schedulers eingereiht. Jede der 32 Run Queues deckt 4 Prioritätswerte ab, wobei hohe Werte niedrige Priorität bedeuten. Die Prozessor wird nur Prozessen aus der Run Queue mit der höchsten Priorität zugeteilt, die mit mindestens einem Prozess belegt ist. Die Prozesse innerhalb einer Run Queue sind gleichbereichtigt, d.h. sie werden nach Round Robbin-Verfahren nacheinander in gleichen Zeitscheiben dem Prozessor zugeteilt.

Die Dauer einer Zeitscheibe beträgt 6 clock ticks, wobei ein clock tick die Länge 16,6 ms beträgt und eine Sekunde in 60 clock ticks zerfällt. Diese Werte können abhängig vom Rechnertyp differieren. Der clock tick ist die kleinste im Rechner planbare Zeiteinheit. Ein erzwungener Kontextwechsel, z.B. durch Bereitwerden eines höherprioren Prozesses, kann nur nach Ablauf eines clock ticks durchgeführt werden. Freiwillige Kontextwechsel können hingegen jederzeit durchgeführt werden. Die Verteilung der Prozesse auf die Run Queues, basierend auf den Prioritätswerten, erfolgt einmal je Sekunde.

Die Berechnung der Prioritätswerte der Prozesse geschieht vor allen Dingen abhängig von der Prozessor-Nutzung der Prozesse mehrstufig:

- Parameter über die mittlere Anzahl von Prozessen in den Run Queues: Dieser Wert wird nach 5 Sekunden nach folgender Formel jeweils neu ermittelt $anzproz_{t+5} = 0.92 * anzproz_t + 0.08 * nrun$, wobei $nrun$ die Anzahl der Prozesse ist, die sich aktuell in den Run Queues befinden. Durch diese Formel werden innerhalb einer Minute 63% des alten Durchschnitts vergessen und durch die 12 neuen Messwerte ersetzt.

- Parameter über die aktuelle Nutzung der Prozessor durch einen bestimmten Prozess: Dieser Wert wird wie folgt berechnet.

  - Während der Ausführung wird der Wert nach 10 ms um 1 inkrementiert: $cpu - nutzung_{t+10ms} = cpu - nutzung_t + 1$.
  - Der Wert bereiter Prozesse wird einmal pro Sekunde wie folgt verringert: $cpu - nutzung_{t+1s} = \frac{2*anzproz_t}{2*anzproz_t+1} * cpu - nutzung_t$. Diese Formel bewirkt, dass der Wert $cpu - nutzung$ während der Wartezeit kontinuierlich sinkt; z.B. sind nach $5 * anzproz_t$ Sekunden 90 % der alten Nutzung vergessen. Je geringer die mittlere Anzahl von Prozessen in den Run Queues $anzproz_t$ ist, desto schneller ist die Vergessensfunktion.

- Parameter Prozesspriorität: Die Prozesspriorität wird im Wesentlichen durch die $cpu - nutzung$ bestimmt, d.h. je geringer dieser Wert ist, desto

höher ist die Priorität. Die Vergessensfunktion in $cpu - nutzung$ bei den wartenden Prozessen sorgt dafür, dass die Priorität dieser Prozesse mit längerem Warten steigt. Zusätzlich werden eine Basispriorität $PUSER$ für Benutzerprozesse (geringere Priorität als Systemprozesse) und ein gewichteter *nice*-Wert, den der Nutzer freiwillig angeben kann, um die Priorität zu verringern addiert: $Prioritt = PUSER + \frac{cpu-nutzung}{4} + 2 * nice$.

Insgesamt wird durch dieses Verfahren also erreicht, dass rechenintensive Prozesse eine niedrigere Priorität erhalten und dadurch in eine geringer priore Run Queue gelangen. Nach entsprechender Wartezeit steigt ihre Priorität und sie gelangen in eine hohe Run Queue und werden irgendwann wieder ausgeführt. Hierduch wird erreicht, dass rechenintensive (Stapelaufträge) und Ein-/Ausgabe-intensive (interaktive) Prozesse einen gerechten Anteil an der Prozessor erhalten. Bei vielen Prozessen im System, d.h. hohem Wert *anzproz*, verbessert sich die Priorität rechenintensiver Prozesse weniger schnell, da dann in den Run Queues und damit im Arbeitsspeicher bereits viele Prozesse liegen. Bei hoher Last werden interaktive Prozesse gegenüber den rechenintensiven Prozessen stärker bevorzugt. Weiterhin wäre bei hoher Last eine schnelle Änderung des Prioritätswertes für den Durchsatz kontraproduktiv, da dann häufige Prozesswechsel mit Nachladevorgängen vom Sekundärspeicher die Folge wären. Insgesamt ist das BSD-Scheduling also ein typischer Vertreter für interaktive Systeme.

2.5.2 Prozesserzeugung und Prozessbeendigung

Ein Prozess kann im Unix-Prozesskonzept weitere Prozesse starten, ohne selbst zu enden. Man spricht in diesem Zusammenhang von Eltern- und Kind-Prozessen (Parent-, Child-Process). Es ergibt sich also eine Prozesshierarchie. Allerdings soll ein Eltern-Prozess im Gegensatz zur Natur nicht beendet werden (terminieren), bevor alle seine Nachkommen beendet sind. Der Eltern-Prozess wartet normalerweise auf das Ende aller seiner Nachkommen und führt ggfs. Abschlußarbeiten anhand von deren Rückgabewerten durch.

Die folgenden Kommandos sind im Zusammenhang mit Prozessen wichtig:

Mit ps können Informationen über Prozesse abgerufen werden. Es gibt eine Reihe von Optionen: ps l liefert z.B. ein long format von ps.

Möchte man einen Prozess vorzeitig beenden, z.B. wegen einer Endlosschleife, so kann man dafür das Kommando kill verwenden. Hierbei wird asynchron ein Signal an den Prozess geschickt, das der Prozess (bis auf das Signal 9) interpretieren und bearbeiten kann. Die Signalnummer kann bei kill optional übergeben werden z.B.:

```
kill -3 4711
```

oder

```
kill -SIGQUIT 4711
```

Für den einfachen Abbruch sollte kill ohne Signalnummer genügen (default ist TERM=15). Den Abbruch kill -9 oder kill -SIGKILL sollte man wirklich nur im äußersten Notfall benutzen, denn der Programmierer hat normalerweise bei den anderen Abbrüchen eine reguläre Terminierung mit Schließen offener Dateien,

löschen temporärer Dateien usw. vorgesehen, die bei kill -9 nicht durchgeführt wird.

Das & hinter einem Kommando führt den Prozess im Hintergrund aus, d.h. man kann weiter Tastatureingaben im Fenster machen, z.B.:

```
xterm &.
```

Hat man das & beim Aufruf vergessen, so kann man den Vordergrund-Prozess mit bg (in engl. background) anschließend in den Hintergrund schicken.

Einen Hintergrund-Prozess kann man durch das Kommando fg (in engl. foreground) wieder in den Vordergrund holen.

Das Kommando jobs zeigt eine Liste aller Hintergrund-Prozesse mit eigenen JobID's, die nicht gleich den PID's sind.

Einen Hintergrund-Prozess kann man mit kill sowohl mit der PID als auch mit der JobID beenden, dabei stellt man der Zahl ein %-Zeichen für die JobID voran, z.B.:

```
kill %3.
```

Das Kommando nice dient zur Änderung der Prozesspriorität. Dabei kann man als nicht-Superuser, wie der Name schon sagt, nur nett sein und seine Priorität reduzieren. Einzig root darf die Priorität von Prozessen erhöhen.

Erwähnenswert ist noch der Befehl nohup (in engl. no hangup), der dafür sorgt, dass ein Prozess nach Abmelden vom System weiterlaufen kann.

2.5.3 Standardeingabe, -ausgabe, Pipes, Filter

Kommandos sind in Unix in ihrer Funktionalität meist übersichtlich. Aufgaben verschiedener Kommandos überlappen selten. Die Flexibilität bei der Anwendung von Kommandos wird dadurch erreicht, dass sie vielseitig miteinander kombiniert werden können. Die Unix-Prozesse, die die Kommandos ausführen, sind miteinander koppelbar.

Hierzu verfügen alle Unix-Prozesse über Standard-Dateideskriptoren. Dateideskriptoren sind positive ganze Zahlen, mit denen Dateien innerhalb des Prozesses angesprochen werden können. Die drei Standard-Dateideskriptoren eines Prozesses sind:

0: Standardeingabe (stdin in C)

1: Standardausgabe (stdout in C)

2: Standardfehlerausgabe (stderr in C)

Vom Betriebssystemkern wird ein Prozess so erzeugt, dass die Standardeingabe mit der Tastatur und die Standardausgabe und -fehlerausgabe mit dem Bildschirm verbunden ist.

Die Ein- und Ausgabe kann jedoch durch Umlenkung (Redirection) auf beliebige Dateien oder Geräte geleitet werden. In allen gebräuchlichen Shells gibt es für die Umlenkung folgende Operatoren:

<: Umlenkung der Standardeingabe aus einer Datei.

Im Beispiel wird der Inhalt der Datei hunger an das mail-Programm übergeben:

```
mail otto < hunger
```

$>$:Umlenkung der Standardausgabe mit Neuerzeugung einer Ausgabedatei.

Im Beispiel wird die Ausgabe des ls-Befehls in die Datei ls_inhalt geschrieben:

```
ls -al > ls_inhalt
```

$>>$: Umlenkung der Standardausgabe, wobei die Ausgabe bei einer existierenden Ausgabedatei an deren Ende gehangen wird.

Im Beispiel wird der Inhalt der Datei durst hinter die Datei hunger gehängt:

```
cat durst >> hunger
```

2$>$ und 2$>>$: Umlenkung der Standardfehlerausgabe mit gleicher Funktionalität wie oben beschrieben.

Im Beispiel werden die Fehlerausgaben auf die Datei error_list geschrieben:

```
prog 2>error_list
```

Durch die Umleitung von Standardausgabe und/oder Standardfehlerausgabe in eine Datei können Fehlermeldungen von normalen Ausgaben getrennt werden. Dies kann sehr nützlich sein. Voraussetzung dazu ist allerdings, dass Programmierer ihre Fehlermeldungen auf die Standardfehlerausgabe schreiben, was nur eine Konvention ist.

Um die zeitliche Reihenfolge von Ausgaben und Fehlerausgaben zu erhalten, kann man die Standardausgabe und -fehlerausgabe in die gleiche Datei leiten. Dazu wird der Deskriptor 2 auf den Deskriptor 1 umgeleitet. Die Syntax hierfür zeigt das nächste Beispiel:

```
prog > out_error_list 2>&1
```

Mit Hilfe der Umlenkung in Dateien kann man nun Kommandos miteinander koppeln. Eine naheliegende Möglichkeit ist die Kopplung über Hilfsdateien. Im Beispiel wird die Ausgabe von ls in eine Zwischendatei geschrieben, die dann als Eingabe für das Kommando wc (word count) eingesetzt wird. wc -w zählt hierbei die Anzahl der Worte in der Zwischendatei und gibt somit die Anzahl der Dateien des aktuellen Verzeichnisses aus:

```
ls > hilfsdatei
wc -w < hilfsdatei
rm hilfsdatei
```

Ein eleganteres und schnelleres Verfahren koppelt zwei Kommandos über einen Puffer im Arbeitsspeicher: Beim Pipelining wird die Standardausgabe eines Kommandos direkt mit der Standardeingabe eines anderen Kommandos verbunden. Dabei arbeiten beide Prozesse parallel und die Synchronisation der Prozesse findet implizit, ohne Benutzereingriff, statt. Das Symbol für die Pipe auf der Shell-Ebene ist der senkrechte Strich |:

```
ls | wc -w
```

Man kann natürlich auch mehr als 2 Kommandos über Pipes miteinander koppeln. Dabei wandeln die mittleren Kommandos in dieser Kette die Daten der Standardeingabe in Daten der Standardausgabe um. Solche Programme heißen Filter. Es gibt eine ganze Reihe von Standard-Filterprogrammen.

Das Filterprogramm sort kann die Zeilen von Textdateien nach Feldern numerisch oder lexikographisch aufsteigend oder absteigend sortieren. Das Beispiel zeigt den einfachsten Fall. Hier wird sort ohne Parameter aufgerufen, um die Ausgabezeilen von ps nach dem ersten Feld lexikographisch aufsteigend zu ordnen. less zeigt die Ausgabe seitenweise an und ermöglicht im Gegensatz zu more auch das Rückwärtsblättern:

```
ps aux | sort | less
```

Mit dem Filter grep können Suchen in Textdateien vorgenommen werden Dazu werden reguläre Ausdrücke verwendet (general regular expression print). grep kann sowohl als Filter (grep <Ausdruck>) als auch direkt auf Dateien (grep <Ausdruck> <Datei(en)>) angewendet werden. Das folgende Beispiel ermittelt die Anzahl von virtuellen Terminals des Benutzers kremer:

```
who | grep kremer | wc -l
```

Das Filterprogramm tee liest von der Standardeingabe und verzweigt die Ausgabe auf die Standardausgabe und Datei. Im Beispiel wird die Ausgabe eines Compiler-Laufs sowohl am Bildschirm angezeigt als auch in die Datei prog.list geschrieben:

```
gcc -o prog prog.c 2>1& | tee prog.list
```

Kapitel 3

Nebenläufigkeit in Betriebssystemen

3.1 Nebenläufigkeit

In Betriebssystemen werden häufig mehrere Vorgänge parallel, d.h. gleichzeitig, oder zeitlich verzahnt (Interleaving) ausgeführt. Der Unterschied zwischen zeitlich verzahnten und parallelen Vorgängen sei an einem Beispiel verdeutlicht:

- Zeitlich verzahnte Vorgänge: Mehrere rechenbereite Prozesse wechseln sich bei der Belegung des Prozessors ab.

- Parallele Vorgänge: Während ein Prozess im Prozessor bearbeitet wird, bearbeitet der Ein-/Ausgabe-Prozessor einen Druckauftrag oder mehrere Prozesse bearbeiten auf mehreren Prozessoren gemeinsam einen Auftrag.

Zusammenfassend spricht man bei solchen Vorgängen von nebenläufigen Prozessen. Nebenläufige Prozesse können miteinander kooperieren und/oder um exklusive Betriebsmittel, wie den Prozessor, konkurrieren.

3.2 Nichtdeterminismus und Verklemmungen

Die Nebenläufigkeit birgt zwei Problemkreise in sich: den Nichtdeterminismus und die Gefahr von sogenannten Verklemmungen.

Nichtdeterminismus liegt immer dann vor, wenn nebenläufige Prozesse bei gleichen Anfangsbedingungen abhängig vom zeitlichen Ablauf unterschiedliche Ergebnisse liefern. Hierzu ein Beispiel:

Zwei Prozesse greifen auf eine gemeinsame Variable im Arbeitsspeicher zu und verändern diese. Je nachdem, ob Prozess 1 vor Prozess 2 oder umgekehrt oder ob und wie sie verschachtelt ineinander ausgeführt werden, ergeben sich verschiedene Ergebnisse. Es kann also nicht vorhergesehen werden, welches Ergebnis sich im Mehrprozessbetrieb ergibt. Man spricht daher von Nichtdeterminismus.

Prozess 1	Prozess 2
x=x+5	x=x*10
x=x*2	x=x+3

Ausgangssituation x=2

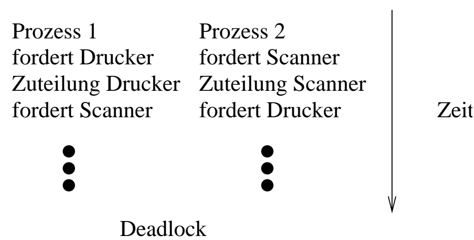
Prozess 1 vor Prozess 2:
x=2+5=7; x=7*2=14
x=14*10=140; x=140+3=143
Ergebnis x=143

Prozess 2 vor Prozess 1:
x=2*10=20; x=20+3=23
x=23+5=28; x=28*2=56
Ergebnis x=56

Prozess 2 in Prozess 1 verschachtelt:
x=2+5=7
x=7*10=70; x=70+3=73
x=73+2=146
Ergebnis x=146

Nichtdeterminismus ist i.a. unerwünscht. Um ihn zu vermeiden, d.h., um deterministische Systeme zu erhalten müssen die Vorgänge synchronisiert werden.

Ein weiteres Problem nebenläufiger Prozesse sind Verklemmungen (Deadlocks). Verklemmungen können entstehen, wenn mehrere (mindestens zwei) Prozesse mehrere (mindestens zwei) exklusive Betriebsmittel belegen oder belegen wollen. Die exklusiv nutzbaren Betriebsmittel können hier Hardware, z.B. ein Drucker und ein Scanner, oder auch Software, z.B. Verzeichniseinträge oder Datensatz-Sperren, sein.



Die an einer Verklemmung beteiligten Prozesse können in ihrer Arbeit also nicht fortfahren, weil sie auf ein exklusives Betriebsmittel eines anderen Prozesses warten, das dieser aber nicht freigibt.

3.3 Multithreading

Ein Thread oder Thread of Control (wörtlich Faden oder Kontrollfaden) ist eine sequenziell ausgeführte Folge von Befehlen innerhalb eines Programms. Häufig ist es z.B. aus Design- oder Effizienzgründen sinnvoll, mehrere Threads zeitlich parallel innerhalb eines Programms auszuführen. Man spricht hierbei von Multithreading. Ein Anwendungsbeispiel kann z.B. ein Multiprozessorsystem sein, in dem ein Programm m Threads auf $n \leq m$ Prozessoren gleichzeitig nutzt.

Die Implementierung von Threads ist entscheidend für ihre Performance und Anwendbarkeit. So können Threads als Prozesse über Multitasking oder über

Thread-Bibliotheken auf Benutzerebene (User Threads) oder als leichtgewichtige Prozess auf Betriebssystemebene (Kernel Threads) realisiert sein.

Multithreading auf Prozessebene:

- Prozesse sind durch separate Adressräume und Zugriffskontrollen auf Betriebsmittel gegeneinander abgeschottet.
- Threads sollen jedoch auf gemeinsamen Daten arbeiten.
- Der Prozesswechsel und die Interprozesskommunikation zur Realisation von Threads über Prozesse ist sehr zeitaufwändig.
- Die Prozessebene ist daher eine schlechte Wahl für die Realisation von Multithreading.

Multithreading auf Benutzerebene

- Die Verwaltung des Multithreading erfolgt ohne Einwirkung des Betriebssystems mit Hilfe von Bibliotheksaufrufen.
- Im Gegensatz zum Multithreading auf Prozessebene erfolgt das Multithreading ohne Zutun des Betriebssystemkerns und ist daher effizienter.
- Mehrere kooporierende Threads sind immer in einem einzigen Prozess realisiert.
- Die Threads arbeiten innerhalb des Prozesses abwechselnd. Daraus folgt das Parallelverarbeitung mehrerer kooporierender Threads nicht möglich ist.
- Blockiert ein Thread innerhalb des Prozesses z.B. wegen einer Ein-/Ausgabe, so sind auch alle anderen Threads blockiert.
- Der Hauptvorzug von Benutzer-Threads liegt darin, dass hierbei Multithreading auf allen Betriebssystemen durchführbar ist.

Multithreading auf Betriebssystemebene

- In Betriebssystemen, wie Solaris von Sun und Windows NT, werden zur Realisierung von Threads sogenannte leichtgewichtige Prozesse (Lightweight Processes oder Kernel-Threads) verwendet.
- Gegenüber einem herkömmlichen Prozess besteht das Modell leichtgewichtiger Prozesse aus:
  - Programm-bezogenen Informationen, wie dem gemeinsamen Adressraum, den Zugriffsrechten und Dateideskriptoren mehrerer Threads.
  - Thread-bezogenen Informationen, wie Zustand, Maschinenstatus, Priorität und Stack für Unterprogrammaufrufe.
- Rechenbereite Threads innerhalb eines Prozesses konkurrieren um die Prozessoren. Der Prozess ist ein statischer Informationsrahmen für mehrere dynamisch ausgeführte Threads.

- Multithreading auf Betriebssystemebene ist i.a. am flexibelsten in der Prozessorzuteilung und insgesamt am schnellsten.

Fast alle modernen Betriebssysteme unterstützen Multithreading zumindest auf der Benutzerebene meist aber auf der Betriebssystemebene. Allerdings sind die Thread-Bibliotheken in Syntax und Semantik unterschiedlich. Eine standardisierte Schnittstelle, POSIX-Thread (Pthread), wird neben der herstellereigenen in letzter Zeit zunehmend zur Verfügung gestellt.

3.4 Synchronisation nebenläufiger Prozesse

Sowohl Prozesse als auch Threads können nebenläufig sein. In diesem Abschnitt ist durchgehend von Prozessen die Rede. Die Überlegungen sind aber genauso für Threads gültig.

Wenn Prozesse in einer bestimmten Reihenfolge ausgeführt werden müssen, sind sie voneinander abhängig. Diese Abhängigkeit kann zwei Gründe haben: Entweder die Prozesse kooperieren, d.h. sie lösen Teilaufgaben einer Gesamtaufgabe, oder sie konkurrieren, d.h. die Aktivitäten der Prozesse können einander behindern. Allerdings schließen Kooperation und Konkurrenz einander nicht aus. Kooperierende Prozesse zur Lösung von Teilaufgaben konkurrieren oft um Betriebsmittel wie Prozessoren oder Speicher.

Die Koordination von Kooperation und Konkurrenz von Prozessen nennt man Synchronisation. Eine Synchronisation bringt die Aktivitäten von Prozessen in eine Reihenfolge, d.h. die Unabhängigkeit der Ausführungsreihenfolge der Aktivitäten von Prozessen wird eingeschränkt.

Bei der mehrseitigen Synchronisation wird verhindert, dass Aktivitäten eines Prozesses gleichzeitig mit bestimmten Aktivitäten anderer Prozesse stattfinden; die Reihenfolge der Aktivitäten spielt hierbei keine Rolle. Bei der einseitigen Synchronisation wird dafür gesorgt, dass bestimmte Aktivitäten eines Prozesses erst nach bestimmten Aktivitäten anderer Prozesse durchgeführt werden. In beiden Fällen müssen Aktivitäten u.U. verzögert werden.

Prozesse versorgen sich gegenseitig mit Informationen. Diesen Vorgang nennt man Kommunikation. Die Kommunikation bedingt eine Synchronisation der Prozesse, denn Informationsabgabe und Informationsaufnahme müssen zeitlich aufeinander abgestimmt sein. Andererseits kann eine Synchronisation technisch auch durch einen Informationsaustausch realisiert sein.

3.4.1 Gemeinsame Variablen

Eine Möglichkeit der Kommunikation von Prozessen sind gemeinsame Variablen (Shared Variables), auf die schreibend und lesend zugegriffen werden kann. Die Folge von Lese- und Schreibzugriffen auf gemeinsame Variablen wird ohne Synchronisation dadurch bestimmt, wie schnell es den Prozessoren gelingt, Zugriffe durchzuführen. Die Prozessoren liefern sich ein Wettrennen (Race). Das Ergebnis eines solchen Wettrennens kann zufällig, d.h. nicht deterministisch, sein. Programmsysteme sollen aber i.d.R. deterministische Ergebnisse erzielen. Deshalb muss eine Synchronisation die Ergebnisse unabhängig von den Umständen des Wettrennens machen.

Durch Synchronisationsmaßnahmen werden Prozesse also u.U. verzögert. Fällt der Grund für die Verzögerung nicht weg, d.h. Prozesse warten auf Ereignisse, die nicht mehr eintreten können, so tritt eine Verklemmung (Deadlock) ein. Synchronisation birgt also die Gefahr der gegenseitigen Blockierung von Prozessen in sich.

Bei einer sogenannten mehrseitigen (multilateralen) Synchronisation ist die zeitliche Reihenfolge von Aktivitäten unerheblich für das berechnete Ergebnis, aber es darf nur eine Aktivität zu einer Zeit ausgeführt werden. Anweisungen verschiedener Prozesse, deren Ausführung einen gegenseitigen Ausschluss (Mutual Exclusion) fordert, nennt man kritische Abschnitte. Kritische Abschnitte dürfen also nicht parallel ausgeführt werden. Jeweils einer der beteiligten Prozesse muss warten, wenn ein anderer Prozess sich im kritischen Abschnitt befindet. Welcher Prozess warten muss, liegt nicht von vorneherein fest, daher spricht man von mehrseitiger Synchronisation.

Einseitige Synchronisation bedeutet, dass zwei Aktivitäten in einer vorgegebenen Reihenfolge stehen müssen, d.h. das Ende von Aktivität 1 ist Voraussetzung für den Beginn von Aktivität 2. Aktivität 2 wird u.U. solange verzögert, bis Aktivität 1 beendet worden ist. Da sich die Synchronisation nur auf Aktivität 2 und nicht auf Aktivität 1 auswirkt, ist sie einseitig (unilateral).

Es folgt ein Beispiel zur mehrseitigen Synchronisation: Ein Programm soll näherungsweise die Zahl π mit Hilfe des folgenden Algorithmus bestimmen. Es wird eine Reihe zweistelliger Vektoren von Zufallszahlen $z \in [0,1] \otimes [0,1]$ generiert. Für jeden Vektor z_i wird überprüft, ob er im ersten Quadranten des Einheitskreises liegt oder nicht, d.h. ob $|z_i| = \sqrt{x_i^2 + y_i^2} \leq 1$ bzw. $x_i^2 + y_i^2 \leq 1$ ist. Die Näherung für $\pi/4$ ergibt sich dann aus dem Quotienten der Trefferanzahl und der Gesamtanzahl.

Das Programm soll nach dem Master-Worker-Prinzip arbeiten, wobei die Arbeiter jeweils einen bestimmten Anteil von Zufallszahlen generieren und überprüfen. Die Gesamtzahl der Treffer aller Arbeiter ist in einer gemeinsamen Variablen von Meister und Arbeitern untergebracht. Bei Beendigung eines Arbeiters erhöht dieser die Gesamtanzahl der Treffer um seine ermittelten Treffer. Die Aufrufe für das Erzeugen (starte) und das Warten auf das Ende von Worker-Prozessen (warte) im gezeigten Programm haben lediglich demonstrativen Charakter; sie entsprechen keiner realen Syntax.

```
Meister-Programm:

/* Globale Variablen */
long int treffer = 0, anzahl = 0;

void main (void) {

  double pi=0.0;
  int i;

  puts ("Wieviele Zufallszahlen sollen
   berechnet werden 1000-100000 ?\n");
  scanf ("%ld", &anzahl);
  /* Start der Arbeiterprozesse */
  for (i=0; i < MAX_WORKER; i++)
```

```
    starte(worker(i));
  /* Warten auf das Ende der Arbeiterprozesse */
  for (i=0; i < MAX_WORKER; i++)
    warte(worker(i));
  pi = (double) 4 * treffer / ( (double) anzahl);
  printf ("Näherungslösung für PI: %f \n", pi);
}

Arbeiter-Programm:

void worker (int i) {

  int j, imkreis = 0;
  double pos_x, pos_y, tmp;

  /* Zufallszahlengenerator initialisieren */
  srand ((int) time(NULL));
  for ( j = 0; j < anzahl / MAX_WORKER; j++) {
    pos_x = (double) rand () / (double) RAND_MAX;
    pos_y = (double) rand () / (double) RAND_MAX;
    /* Im Einheitskreis? */
    tmp = (double) (pos_x * pos_x + pos_y * pos_y);
    if (tmp <= 1) imkreis++;
  }
  /* Nicht synchronisierter Zugriff auf
     gemeinsame Variable treffer */
  treffer = treffer + imkreis;
  return NULL;
}
```

Durch den nicht synchronisierten Zugriff auf die gemeinsame Variable treffer entsteht nun ein Problem. Werden zwei Prozesse ungefähr gleichzeitig mit ihren Rechnungen fertig, so greifen beide auch ungefähr gleichzeitig lesend auf die gemeinsame Variable treffer zu, um diese zu erhöhen.

Der Wert von treffer, z.B. 42, wird auf der rechten Seite der Gleichung treffer = treffer + imkreis in beiden Prozessen in eine lokale Variable eingesetzt: treffer = 42 + imkreis. Bis hierhin scheint noch alles in Ordnung. Nun hat Prozess A imkreis = 28 lokale Treffer ermittelt; Prozess B hat imkreis = 30 Treffer kurz nach Prozess A ermittelt. Prozess A schreibt also zunächst den Wert 70 in die gemeinsame Variable. Anschließend schreibt Prozess B den Wert 72 in die gemeinsame Variable. Dies liegt daran, dass der Wert von treffer (42) als lokale Kopie in Prozess B nicht mehr aktuell und deswegen falsch ist. Anstatt 100 ist das Gesamtergebnis von treffer 72.

3.4.2 Schlossvariable

Wie kann man dieses Problem lösen? Der Fehler wäre nicht passiert, wenn jeweils höchstens ein Prozess auf die gemeinsame Variable zugegriffen hätte, wobei Lesen und Schreiben von treffer durch die Anweisung unteilbar von einem Prozess durchgeführt werden müsste.

Technisch wird dies z.B. mit Hilfe einer sogenannten Schlossvariablen (Mutex: Kurzform von Mutual Exclusion) realisiert. Um den Zugang mehrerer Prozesse zu kritischen Abschnitten zu regeln, wird eine Schlossvariable (lock variable) (mutex von mutual exclusion) eingeführt. Die Variable wird tatsächlich als Türschloss für den Zugang zu einem kritischen Abschnitt folgendermaßen verwendet:

Will ein Prozess in einen kritischen Abschnitt eintreten, so wartet er ggf., bis das zugehörige Schloss offen ist. Dann betritt der Prozess den kritischen Abschnitt und verschließt das Schloss sozusagen von innen (lock), damit kein anderer Prozess in den kritischen Abschnitt eintreten kann. Hat der Prozess den kritischen Abschnitt beendet, so schließt er wieder auf (unlock). Wichtig ist, dass eine unlock-Operation zu einem kritischen Abschnitt nur von dem Prozess ausgeführt werden kann, der vorher die lock-Operation ausgeführt hat.

Die Implementierung der lock-Operation muss als test_and_set Maschineninstruktion nicht unterbrechbar sein, sonst könnte ein Thread während der Überprüfung auf den offenen Mutex vom Betriebssystem unterbrochen werden. Anschließend könnte ein anderer Thread aktiviert werden und ebenfalls den Mutex als offen vorfinden. Beide Threads könnten dann den kritischen Abschnitt betreten.

Eine Schlossvariable ist ein abstrakter Datentyp, der aus einer booleschen Variablen und mindestens aus den Operationen lock und unlock besteht. In den POSIX-Threads hat die Schlossvariable den Typ pthread_mutex_t und folgende Funktionen sind auf diesen Variablen definiert:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);
```

Diese Funktion inititialisiert eine Schlossvariable mutex mit den in attr angegebenen Werten. Mit diesen Attributen kann ein Mutex z.B. auf einen Prozess begrenzt werden oder über Prozessgrenzen hinaus wirken.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Diese Funktion löscht den Mutex.

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Diese Funktion schließt einen offenen Mutex. Ist der Mutex verschlossen, so blockiert der aufrufende Thread solange bis der Mutex offen ist. Anschließend verschließt er den Mutex.

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Diese Funktion schließt einen offenen Mutex. Ist der Mutex verschlossen (return-Wert), so fährt der aufrufende Thread fort, ohne in den kritischen Abschnitt einzutreten.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Diese Funktion öffnet den Mutex. Die Scheduling-Strategie des Betriebssystems entscheidet dann darüber, welcher der bereits blockierten Threads wiederaufgenommen wird. Der wiederaufgenommene Thread wiederholt die lock-Operation. Allerdings kann in der Zeit zwischen der Wiederaufnahme und der lock-Operation ein anderer nicht blockierter Thread mit seiner lock-Operation zuvorkommen.

3.4.3 Semaphor

Ein Semaphor (engl. semaphore) ist ein abstrakter Datentyp aus einer nicht-negativen Integer-Variablen (Semaphorzähler) und zwei Operationen dem P (Passieren) und dem V (Verlassen). Der Semaphorzähler wird dabei über die Operationen P und V folgendermaßen verändert:

P(S):

```
Wenn S > 0
dann S := S-1
sonst wird der aufrufende Prozess suspendiert (Wartezustand)
```

V(S):

```
S := S + 1;
```

P und V umschließen den kritischen Abschnitt wie lock und unlock bei der Schlossvariablen. Die Operationen P und V sind ebenfalls als nicht unterbrechbar implementiert. Falls ein wartender Prozess existiert, wird er nach einer V-Operation auf denselben Semaphor wieder erweckt. Sind mehrere Prozesse im Wartezustand, so ist es wieder abhängig vom Betriebssystem in welcher Reihenfolge die wartenden Prozesse geweckt werden.

Binäre Semaphore, die nur die Werte 0 oder 1 annehmen können, sind in ihrer Funktion zu den Schlossvariablen ähnlich. Allerdings müssen die P- und V-Operationen nicht wie bei Schlossvariablen die lock- und unlock-Operationen vom selben Prozess ausgeführt werden.

Allgemeine Semaphore können beliebige Werte des Semaphorzählers annehmen. Im Fall einer mehrseitigen Synchronisation gibt der Anfangswert des Semaphorzählers die maximale Anzahl von Prozessen an, die gleichzeitig im kritischen Abschnitt arbeiten dürfen.

In einem Programm mit n-Prozessen kann der gegenseitige Ausschluss auf folgende Weise programmiert werden.

```
program gegenseitigerausschluss;
  var s: semaphor;
  procedure process (i: integer);
  begin
    repeat
      P(s);
      kritischerbereich;
      V(s);
      rest;
    until bedingung_erfüllt;
  end;

  begin (* Hauptprogramm *)
    s := 1;
    cobegin
    process(1)
    process(2)
    process(3)
```

```
    coend;
  end.
```

Werden in diesem Beispiel mehr als zwei Prozesse gleichzeitig aufgerufen, so kann der dritte Prozess ausgesperrt werden. Wird z.B. immer der wartende Prozess mit der kleinsten Prozessnummer wiedererweckt, so wechseln sich Prozess 1 und 2 gegenseitig ab und Prozess 3 wartet (unendlich) lange auf seine Aktivierung. Es ist also eine Frage der Implementation der Semaphore, ob solche Ausschlüsse vorkommen.

Eine Anwendung für einen allgemeinen Semaphor ist die Pufferverwaltung beim Erzeuger-Verbraucher-Problem: Mehrere Erzeuger-Prozesse erzeugen Daten und legen sie in einem Pufferspeicher ab. Weitere Verbraucher-Prozesse lesen Daten aus dem Pufferspeicher und bearbeiten sie. Der Puffer soll hierbei beliebig dynamisch wachsen können.

Wenn zwei oder mehr Prozesse unsynchronisiert gleichzeitig Daten entnehmen oder einfügen wollen, oder ein Prozess Daten entnehmen und ein anderer gleichzeitig Daten einfügen will, kommt es zum Wettrennen. Entnahme und Einfügen von Daten sind kritische Abschnitte auf dem Pufferspeicher.

Diese kritischen Abschnitte werden über einen binären Semaphor realisiert. Die Anzahl der Elemente im Pufferspeicher werden über einen allgemeinen Semaphor realisiert. Wenn der allgemeine Semaphor 0 ist, bedeutet das das der Pufferspeicher leer ist. Dann muss der Verbraucher-Prozess auf ein neues Datenelement von einem Erzeuger-Prozess warten.

Der Pufferspeicher wird über nicht unterbrechbare Operationen legeab und entnehme gesteuert werden. Der allgemeine Semaphor ist die Differenz zwischen der Anzahl von V-Signalen zur Anzahl von P-Signalen. Nach der Operation legeab wird eine V-Operation und vor der Operation entnehme eine P-Operation durchgeführt. Die beiden Operationen legeab und entnehme sind kritische Bereiche mit gegenseitigem Aussschluss und werden durch einen binären Semaphor gegeneinander geschützt.

```
program erzeugerverbraucher;
var n: semaphore; (* allgemeiner Semaphor *)
    s: semaphor;  (* binärer Semaphor *)
procedure erzeuger;
begin
  repeat
    erzeuge;
    P(s);
    legeab;
    V(s);
    V(n);
  forever;
end;
procedur verbraucher;
begin
  repeat
    P(n);
    P(s);
    entnehme;
```

```
    V(s);
    verbrauche;
  forever;
end;
begin (* Hauptprogramm *)
  n:=0;
  s:=1;
  cobegin
    erzeuger;
    verbraucher;
  coend
end.
```

Das Vertauschen der P-Operationen von binärem und allgemeinem Semaphor kann zu einer Verklemmung führen. Der Verbraucher führt erfolgreich P(s) aus, da anfänglich s=1 gilt und wird dann durch P(n) blockiert. Der Erzeuger kann aber in den kritischen Abschnitt für legeab nicht mehr eintreten, da s=0 ist. Das System ist blockiert.

3.4.4 Monitor

Semaphore lösen Synchronisationsprobleme auf einer logisch niedrigen Ebene. Wird nur eine Semaphoroperation falsch programmiert, so kann es zu Systemzusammenbrüchen kommen. Das lässt sich weitgehend verhindern, wenn Synchronisationswerkzeuge auf höherer logischer Ebene, wie Monitore, verwendet werden.

Monitore sind abstrakte Datenstrukturen aus Variablen, Prozeduren und Funktionen, über die auf den Monitor zugegriffen wird, und einem Monitorkörper der nach dem Start des Programms den Monitor initialisiert.

Die Realisierung des gegenseitigen Ausschluss und die Implementierung des Monitors bleiben dem Benutzer verborgen. Zugriffsoperationen auf einen Monitor werden immer im gegenseitigen Ausschluss durchgeführt, die Monitorvariablen sind also exklusive Betriebsmittel für Prozesse. Falls ein zweiter Prozess den Monitor betreten will, wird er suspendiert und wartet von außen auf die Freigabe des Monitors.

Innerhalb des Monitors werden die primitiven Operationen zur Synchronisation in Prozeduren oder Funktionen gekapselt. Da Zählen in integer-Variablen im geschützten Bereich des Monitors möglich ist, genügen eine Blockier- und Aufweckoperation, mit denen eine Bedingungsvariable angesprochen wird. Auf Bedingungsvariablen kann nur mit den Operationen wait() und signal() innerhalb des Monitors zugegriffen werden:

```
wait(c):
```

Der aufrufende Prozess wird blockiert und in eine interne Warteschlange von Prozessen eingeordnet, die ebenfalls wegen Bedingung c blockiert sind. Danach wird der Monitor verlassen und damit freigegeben.

```
signal(c):
```

Falls die Warteschlange zu c nicht leer ist, wird der Prozess am Kopf der internen Warteschlange zu c geweckt.

Da signal einen Prozess erwecken kann und sich nur ein Prozess zu einer Zeit im Monitor aufhalten darf, darf jede Monitorprozedur höchstens einen signal-Aufruf am Ende der Prozedur enthalten. Mehrere signal-Aufrufe sind sinnlos, da nicht mehrere Prozesse gleichzeitig im Monitor arbeiten dürfen. Ein signal-Aufruf vor dem Ende der Monitorprozedur ist ebenfalls sinnlos, da der Monitor noch durch den aktuellen Prozess belegt ist.

Weiterhin ist nach einem signal der Prozess am Kopf der betreffenden Warteschlange der nächste auszuführende Prozess. Insbesondere haben Prozesse der internen Warteschlange Vorrang vor Prozessen, die von außen den Monitor aufrufen. Die Scheduling-Strategie ist also nicht wie bei Schlossvariablen und Semaphoren von der Implementierung des Betriebssystems abhängig.

Als Programmbeispiel folgt das Erzeuger-Verbraucher-Problem unter Annahme eines beschränkten Pufferspeichers. Beim Start des Programms wird zunächst der Monitorkörper ausgeführt. Die Monitorprozeduren legeab und entnehme bleiben passiv, bis sie von einem Prozess aufgerufen werden.

```
program erzeugerverbraucher;
const puffergrösse=1000;
monitor beschränkterpuffer;
var b:array[0..puffergrösse] of integer;
    in, out: integer; (* Pufferzeiger *)
    n: integer; (* Anzahl Zeichen im Puffer *)
    nichtleer, nichtvoll: condition;
procedure legeab (v: integer);
begin
  if n = puffergrösse+1 then wait(nichtvoll); (* Puffer voll *)
  b[in] := v; in := in + 1; n := n + 1;
  if in = puffergrösse + 1 then in := 0;
  signal(nichtleer);
end;
procedure entnehme (var v: integer);
begin
  if n = 0 then wait(nichtleer); (* Puffer leer *)
  v := b[out]; out := out + 1; n := n - 1;
  if out = puffergrösse + 1 then out := 0;
  signal(nichtvoll);
end;
begin (* Monitorkörper *)
  in := 0; out := 0; n := 0;
end;
procedure erzeuger;
var v: integer;
begin
  repeat
    erzeuge(v); legeab(v);
  forever;
end;
procedure verbraucher;
```

```
var v: integer;
begin
  repeat
    entnehme(v); verbrauche(v);
  forever;
end;
begin
  cobegin
    erzeuger; verbraucher;
  coend;
end.
```

Monitore sind idealerweise in die Programmiersprache integriert, z.B. in Concurrent Pascal oder Modula-2. Der Compiler erzeugt dann Code, der den Eintritt eines Prozesses in den Monitor durch eine Schlossvariable oder einen Semaphor regelt.

Bei den heutigen mehrfädigen Betriebssystemen wie Solaris, IBM AIX wird statt des Monitorkonzepts eine Kombination aus Schloss- und Bedingungsvariable angewendet.

In der objektorientierten Programmiersprache Java wurde das Monitorkonzept jedoch wiederbelebt, wobei die Moduleigenschaft des Monitors durch Klassen realisiert wird.

3.4.5 Bedingungsvariablen

Bedingungsvariablen werden bei allen Thread-Konzepten und bei Monitoren angewendet. Thread-Bedingungsvariablen werden immer in Verbindung mit Mutex-Variablen eingesetzt, wobei das Sperren den exklusiven Zugriff auf die Bedingungsvariable und auf gemeinsame Daten zur Auswertung der Bedingung sichert. Die Progammierung von Mutex-Variablen geschieht explizit zusätzlich zur Programmierung der Bedingungsvariablen. Wichtigste Operationen auf Bedingungsvariablen sind signal, broadcast und wait.

Für POSIX-Threads sind Bedingungsvariable vom Typ pthread_cond_t und folgende Operationen definiert:

```
int pthread_cond_init (pthread_cond_t *cond,
  const pthread_condattr_t *attr);
```

Diese Funktion initialisiert eine Bedingungsvariable cond mit den Attributen unter attr (Default: NULL). Bedingungsvariablen müssen vor Gebrauch initialisiert werden.

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

Diese Funktion löscht die Bedingungsvariable cond.

```
int pthread_cond_wait (pthread_cond_t *cond,
  pthread_mutex_t *mutex);
```

Diese Funktion gibt die Schlossvariable mutex frei und versetzt den aufrufenden Thread in einen Wartezustand bezüglich der Bedingungsvariablen cond. Durch

die Freigabe des mutex können andere Threads in den kritischen Bereich eintreten. Durch eine signal- oder broadcast-Operation wird ein wartender Thread wieder fortgeführt.

```
int pthread_cond_timedwait (pthread_cond_t *cond,
  pthread_mutex_t *mutex, const struct timespec *abstime);
```

Diese Funktion arbeitet wie die vorherige, zusätzlich wird der wartende Thread automatisch spätestens ab der Systemzeit abstime wieder aufgerufen.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Die signal-Operation weckt einen Thread auf, der für cond in einen Wartezustand versetzt wurde. Welcher Thread aufgeweckt wird, ist Sache des Betriebssystems. Der aufgeweckte Thread versucht die Mutex-Variable zu erhalten, damit er in den kritischen Bereich eintreten kann.

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Die broadcast-Operation weckt alle Threads auf, die für die Bedingungsvariable cond in den Wartezustand versetzt wurden. Jeder der aufgeweckten Threads bemüht sich automatisch erst um den Erhalt der Mutex-Variable, bevor er in den kritischen Abschnitt eintritt. Der Einsatz einer Broadcast-Operation ist sinnvoll, wenn die Auswertung der Bedingung in den wartenden Threads unterschiedliche Ergebnisse liefert, so dass nicht von vornherein klar ist, dass jeder Thread erweckt werden kann.

Das nachfolgende Programmstück zeigt ein Beispiel für das Zusammenspiel von Mutex- und Bedingungsvariablen:

Programmstück aus dem Thread mit Bedingungsauswertung:

```
pthread_mutex_lock(&mutex); /* exklusiven Zugriff sichern */
  ... /* Bedingung auswerten */
  if (bedingung == FALSE) {
    pthread_cond_wait (&cond, &mutex); exit(0);
  }
  ... /* Aktivitäten des Thread */
pthread_mutex_unlock(&mutex); /*exklusiven Zugriff freigeben */
```

Programmstück aus dem Thread mit Bedingungsänderung:

```
pthread_mutex_lock(&mutex); /* exklusiven Zugriff sichern */
  modify_bedingung /* Bedingung ändern */
  pthread_cond_signal(&cond);
  ... /* Aktivitäten des Thread */
pthread_mutex_unlock(&mutex); /*exklusiven Zugriff freigeben */
```

Beim Konzept der POSIX-Bedingungsvariablen werden Threads im Gegensatz zur FIFO-Reihenfolge bei Monitoren in beliebiger Reihenfolge aufgeweckt. Weiterhin muss signal oder broadcast nicht vor dem Verlassen einer Programmeinheit stehen. Das Verlassen eines kritischen Bereichs muss der Thread explizit z.B. mit pthread_mutex_unlock(&mutex) anzeigen.

3.4.6 Barriere

Eine Barriere ist ein Synchronisationspunkt für mehrere Prozesse. Hierbei warten alle Prozesse an der Barriere so lange, bis der letzte Prozess einer Gruppe an der Barriere angekommen ist.

Eine Barriere muss vor dem ersten Benutzen initialisiert werden, dabei weist eine init_barrier-Anweisung der Barriere-Variable die Zahl der Prozesse zu, auf die an der Barriere gewartet werden soll. Prozesse, die an der Barriere ankommen, führen eine wait_barrier-Anweisung aus. Sofern die Prozessanzahl größer Null ist, wird sie dekrementiert und der Prozess suspendiert. Wenn der letzte der Prozesse (Prozessanzahl gleich Null) an der Barriere ankommt, werden alle wartenden Prozesse wieder aufgenommen und die Barriere auf den Initialwert zurückgesetzt.

Ziel von Barrieren ist es, Prozesse, die nach Berechnungen Daten austauschen müssen, auf einen gleichen Bearbeitungsstand zu bringen.

In den meisten Thread-Paketen und in den POSIX-Threads gibt es keine Barrieren. Sie können dort aber über Mutex- und Bedingungsvariablen nachgebildet werden. Barrieren sind z.B. bei Parallelrechnern der Firma Cray implementiert.

3.4.7 Synchronisation über Nachrichten

Bei mehreren Rechnern mit verteiltem Speicher scheidet die Möglichkeit der Kommunikation und Synchronisation über gemeinsame Variablen aus. An die Stelle des Schreibens und Lesens von gemeinsamen Variablen tritt das Senden und Empfangen von Nachrichten. Dieser Nachrichtenaustausch wird aber auch häufig für die Kommunikation von Prozessen auf einem Rechner verwendet. Hierbei besteht eine implizite Sequentialität zwischen Senden und Empfangen: Eine Nachricht muss gesendet werden, bevor sie empfangen werden kann. Deshalb führt der Nachrichtenaustausch neben der Kommunikation zu einer impliziten Synchronisation der beteiligten Prozesse. Eine Nachricht entsteht durch eine Sendeoperation eines Prozesses mit folgenden Angaben:

- Ziel der Nachricht (Prozess, Prozessor, Kommunikationskanal)
- Nachrichtennummer zur Identifikation der Nachricht
- Sendepuffer: Speicherbereich, dessen Inhalt verschickt werden soll
- Signatur: Anzahl und Typ der Datenelemente im Sendepuffer, Sender und Empfänger müssen in dieser Signatur übereinstimmen

Blockiert die Sendeoperation bis die korrespondierende Empfangsoperation durchgeführt wurde, so nennt man die Sendeoperation synchron. Arbeitet der Sendeprozess unabhängig vom Empfang der Nachricht weiter, so ist der Sendeprozess asynchron.

Wird der Inhalt aus dem Sendepuffer des Anwenderprogramms direkt auf das Verbindungsnetz geschickt, so spricht man von einer ungepufferten Sendeoperation. Wenn der Sendepuffer in einen Systempuffer kopiert wird, bevor die Sendung ausgeführt wird, nennt man dies eine gepufferte Sendeoperation.

Die ungepufferte Sendeoperation ist i.a. schneller als die gepufferte, die eine Kopieroperation zusätzlich beinhaltet. Bei heterogenen Workstation-Clustern

ist eine gepufferte Sendung oft notwendig, weil bei der Übertragung in den Sendepuffer Datenkonvertierungen in das Datenformat des Empfängers stattfinden.

Wenn die Sendeoperation im Sendeprozess blockiert, bis der Sendepuffer auf das Netz oder in den Systempuffer übertragen worden ist, so spricht man von einer blockierenden Sendeoperation. Im anderen Fall, spricht man von einer nicht-blockierenden Sendeoperation. Bei einer nicht-blockierenden Sendeoperation besteht allerdings die Gefahr, dass der Sendepuffer durch nachfolgende Befehle nach der Sendeoperation unabsichtlich verändert wird. Hieraus ergeben sich folgende mögliche Kombinationen:

- synchrone Sendung ist immer auch blockierend;
- asynchron, blockierende Sendung heißt nach Ende der Sendung vor Empfang der Sendung kann im Sender mit den darauffolgenden Befehlen weitergearbeitet werden;
- asynchron, nicht-blockierende Sendung heißt vor Ende der Sendung kann im Sender mit den darauffolgenden Befehlen weitergearbeitet werden.

Im Empfänger kann eine Nachricht konsumierend (zerstörend) oder konservierend gelesen werden. Bei konservierendem Empfang kann eine Nachricht mehrfach gelesen werden.

Normalerweise ist eine Empfangsoperation synchron, d.h. sie versetzt den Prozess in einen Wartezustand, bis die Nachricht eingetroffen ist. Bei einer asynchronen Empfangsoperation erhält der Prozess entweder die Antwort, dass keine Nachricht vorliegt oder er erhält die Nachricht. In beiden Fällen setzt der Prozess seine Arbeit mit den der Empfangsoperation folgenden Befehlen fort.

Betrachtet man Sende- und Empfangsoperationen gemeinsam, so spricht man von synchronem Nachrichtenaustausch, wenn Sende- und Empfangsoperation synchron ausgeführt werden. Beim asynchronen Nachrichtenaustausch verwendet mindestens der Sender oder der Empfänger eine asynchrone Operation. Meistens ist dies der Sender. An der asynchron operierenden Seite muss ein Sende- oder Empfangspuffer eingerichtet werden.

Ein sendender Prozess muss angeben, wohin seine Nachricht gesendet werden soll. Ein empfangender Prozess muss i.d.R. angeben, woher er eine Nachricht erhalten will. Es ist also eine Adresse für Ziel oder Ursprung einer Nachricht erforderlich. Man unterscheidet folgende Adressierungsarten:

Direkte Benennung (direct naming): Hierbei besitzt jeder Prozess eine Prozessidentifikation, die bereits zur Entwicklungszeit in die Kommunikationsanweisung eingebaut werden kann. Solche Prozessidentifikationen können auch relativ zu einem Prozessorknoten oder einer logischen Prozessgruppe definiert sein.

Briefkasten (mailbox): Dies ist ein globaler Speicherplatz in den mehrere Prozesse Nachrichten durch Senden ablegen oder durch Empfangen herausnehmen.

Pforte (port): Eine Pforte ist an einen Prozess gebunden, wobei der Prozess je nach Vereinbarung die Pforte ausschließlich sendend oder empfangend benutzen darf.

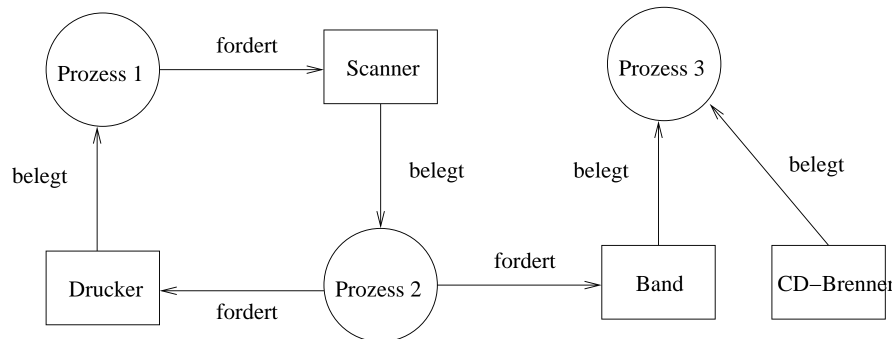
Verbindung oder Kanal (connection, channel): Zu den Pforten gehört das Einrichten von Verbindungen. Bevor Verbindungen eingerichtet werden können, müssen lokal zu Prozessen vereinbarte Pforten existieren. Unabhängig von den zu Pforten gehörenden Prozessen verbindet man die Pforten mit connect-Anweisungen. Die Sendepforte eines Prozesses wird an die Empfangspforte eines anderen verbunden.

Die bisher behandelten Kommunikationen sind datenorientiert, denn Daten werden von einem Prozess an einen anderen gesendet. Steht bei der Kommunikation, die bei einem anderen Prozess beantragte Aktivität im Vordergrund, wie beim Client-Server-Modell, so spricht man von einer aktionsorientierten Kommunikation. Diese aktionsorientierte Client-Server-Kommunikation erfordert zwei datenorientierte Kommunikationen.

Wenn sich der Prozess, der entfernte Prozeduren anbietet, aussuchen kann, wann er die aufgerufene Prozedur ausführt, so spricht man von einem Rendezvous der beteiligten Prozesse. Wenn beide Prozesse bereit sind, treffen sie sich zur Ausführung einer Prozedur. Danach arbeiten sie wieder unabhängig voneinander weiter.

3.5 Behandlung von Verklemmungen

Die Belegung exklusiver Betriebsmittel kann durch einen Belegungs-/Anforderungs-Graf veranschaulicht werden. Die Knoten sind die Prozesse und Betriebsmittel; Kanten spiegeln Belegungen und Anforderungen wider.



Folgende Bedingungen sind notwendig und hinreichend dafür, dass Verklemmungen auftreten können (Bedingungen für Deadlock-gefährdete Systeme):

1. Es gibt eine zyklische Kette von Prozessen im Belegungs-/Anforderungs-Graf, von denen jeder Prozess ein Betriebsmittel belegt, das der nächste Prozess in der Kette benötigt.

2. Die gemeinsam benutzten Betriebsmittel können nicht parallel durch mehrere Prozesse sondern nur exklusiv durch einen Prozess genutzt werden. Ein Prozessor oder ein Drucker sind z.B. exklusive Betriebsmittel.

3. Die belegten Betriebsmittel können nicht entzogen werden, d.h. die Nutzung ist ununterbrechbar. Beispiel für ein nicht entziehbares Betriebsmittel ist ein Drucker, ein Prozessor ist hingegen entziehbar.

4. Prozesse belegen, während sie exklusive Betriebsmittel anfordern und auf deren Zuteilung warten, bereits exklusive Betriebsmittel, die sie nicht freigeben. D.h. sie benötigen mehrere exklusive Betriebsmittel gleichzeitig.

Es gibt drei unterschiedliche Verfahrensarten, die das Verklemmungsproblem behandeln:

- Erkennung und Beseitigung von Verklemmungen: Man stellt Verklemmungen fest und beseitigt sie. Die Beseitigung von Verklemmungen ist i.d.R. aufwändig und auch nicht immer möglich.
- Vermeidung von Verklemmungen (Prevention): Vermeidungsverfahren beruhen darauf, dass durch die Festlegung von Regeln dafür gesorgt wird, dass mindestens eine der für das Auftreten einer Verklemmung notwendigen Bedingungen nicht erfüllt ist. Da sich solche Regeln nicht für jedes Verklemmungsproblem finden lassen, ist ein allgemeinerer Algorithmus gesucht: das Verhinderungsverfahren.
- Verhinderung von Verklemmungen (Avoidance): Das Verfahren beruht auf der Idee, zukünftige Betriebsmittelanforderungen zu analysieren und nur solche Zustände zu verhindern, die zu Verklemmungen führen können.

3.5.1 Erkennen von Verklemmungen

Ist nach einer gewissen Zeit ein angefordertes Betriebsmittel immer noch nicht zugewiesen, so wird ein Erkennungs-Algorithmus gestartet, der i.d.R. nach einem der folgenden Ansätze arbeitet.

Ansatz 1: Suche nach Zyklen im Belegungs-/Anforderungsgraph

Ansatz 2: Prüfe, ob es eine Ausfürungsreihenfolge der Prozesse gibt, so dass alle Prozesse beendet werden können.

Algorithmus zu Ansatz 2:

1. Starte mit Prozessmenge P, die alle Prozesse enthält.
2. Suche Prozess $p \in P$, dessen noch ausstehende Anforderungen beim aktuellen Belegungszustand erfüllbar sind. Falls kein solcher Prozess existiert liegt eine Verklemmung vor und der Algorithmus terminiert.
3. Nehme an, der Prozess aus 2. wird zu Ende durchgeführt und gibt seine belegten Betriebsmittel wieder frei.
4. Entferne p aus P. Gehe zu 2, falls P nicht leer ist. Ist P leer, dann liegt keine Verklemmung vor und der Algorithmus terminiert.

3.5.2 Vermeiden von Verklemmungen

Hierfür können stellvertretend zwei Ansätze genannt werden:

1. Ansatz für die Verklemmungs-Vermeidung: Ein Prozess darf nur dann ein exklusives Betriebsmittel anfordern, wenn er kein anderes exklusives Betriebsmittel mehr belegt. Damit ist die 4. Bedingung für eine Verklemmung nicht erfüllt.

2. Ansatz für die Verklemmungs-Vermeidung: Um einen Zyklus im Belegungs-/Anforderungs-Graph (Bedingung 1) zu vermeiden, wird eine lineare Ordnung über den exklusiven Betriebsmitteln definiert: $BM_1 < BM_2 < ... < BM_m$. Jeder Prozess darf exklusive Betriebsmittel nur in dieser global festgelegten Reihenfolge anfordern. D.h. belegt ein Prozess ein exklusives Betriebsmittel BM_i, so darf er nur exklusive Betriebsmittel BM_j anfordern, für die $j > i$ gilt.

Beweisidee: Prozess 1 belegt das exklusive Betriebsmittel BM_l und Prozess 2 belegt das exklusive Betriebsmittel BM_k. Nun kann ein Zyklus nur dann eintreten, wenn Prozess 1 BM_k anfordert und! Prozess 2 BM_l anfordert. Gilt $l < k$, so darf aber Prozess 2 BM_l nicht anfordern. Gilt $k < l$, so darf aber Prozess 1 BM_k nicht anfordern.

Für jeden Prozess gilt, dass BM_i i.a. länger reserviert wird als BM_j, wenn $i < j$ gilt. Betriebsmittel, die gut ausgelastet werden sollen, dürfen einem Prozess nicht zu lange zugeordnet werden, d.h. sie müssen in der Ordnung weiter hinten stehen.

Das Hauptproblem dieses Verfahrens besteht darin, eine akzeptable Ordnung der Betriebsmittel zu finden. Da auch z.B. Prozess-Tabellen und gesperrte Datenbanksätze usw. exklusive Betriebsmittel darstellen, ist in der Realität die Anzahl der exklusiven Betriebsmittel und möglichen sinnvollen Anforderungsreihenfolgen so groß, dass keine einheitliche Nummerierung, für alle diese Möglichkeiten gefunden werden kann.

3.5.3 Verhindern von Verklemmungen

Für die Verhinderung von Verklemmungen ist der Bankiers-Algorithmus von Dijkstra grundlegend:

Szenario: Ein Bankier hat ein bestimmtes Kapital zur Verfügung, dass er im Rahmen von Krediten seinen Kunden zur Verfügung stellen kann. Ein Kredit ist eine Geldmenge, die der Bankier einem Kunden für ein Geschäft zur Verfügung stellt, und die der Kunde dem Bankier irgendwann, spätestens aber nach dem Ende seines Geschäfts, komplett zurückzahlt. Jeder Kunde vereinbart für ein Geschäft einen festen Kreditrahmen, der kleiner als das Kapital des Bankiers ist und der zu keiner Zeit vom Kunden überschritten wird. Weiterhin hat jeder Kunde einen aktuellen Kredit (evtl. auch 0).

Aufgabe des Bankiers: Die Aufgabe des Bankiers ist es, zu gewährleisten dass jeder Kunde seine geforderten Kredite für Geschäfte irgendwann, d.h. evtl. nach einer Wartezeit auf den angeforderten Kredit, wahrnehmen kann. Natürlich sollen dabei mehrere Geschäfte von Kunden parallel bedient werden. (Die sequentielle Kreditaufnahme und Kreditrückzahlung durch jeden Kunden ist trivialerweise immer möglich.)

Analogie: In dieser Analogie entspricht das Geschäft eines Kunden einem Prozess mit einem maximalen Betriebsmittelbedarf (Kreditrahmen) innerhalb einer Betriebsmittelklasse. Eine Betriebsmittelklasse besteht aus mehreren gleichen Betriebsmitteln, z.B. CD-Laufwerken oder gleichartigen Druckern. Dem Bankier entspricht das Betriebssystem, das die Betriebsmittel der

Klasse zuteilt (Kredite vergibt) und nach der Freigabe (Kredite zurückgezahlt) wieder in die Klasse der freien Betriebsmittel aufnimmt. Der maximale Betriebsmittelbedarf wird nach dem Prozessstart dem Betriebssystem angegeben. Der Prozess fordert nun zu verschiedenen Zeiten Betriebsmittel der Klasse an (Kredit beantragt) und gibt sie auch (teilweise) wieder zurück (Kredite zurückzahlen). Spätestens bei Prozessende werden alle Betriebsmittel zurückgegeben.

Idee des Bankiers-Algorithmus: Bei jeder Anforderung wird überprüft, ob deren Zuteilung zu einem Zustand führt, ab dem in mindestens einer Reihenfolge der Zuteilungen alle weiteren potentiellen Anforderungen von Prozessen ggf. nach Wartezeiten erfüllt werden können. Nur wenn dies der Fall ist, wird die Zuteilung durchgeführt; ansonsten wird die Anforderung zurückgestellt.

Genauer: Es wird überprüft, ob es nach der potentiellen Zuteilung bei der Anzahl noch freier Betriebsmittel eine verklemmungsfreie Reihenfolge von weiteren Zuteilungen (ggf. mit Wartezeiten) für Anforderungen aller Prozesse jeweils (pessimal) bis zu deren Kreditrahmen gibt. Dann können alle Prozesse ohne Verklemmung bis zum Ende bedient werden.

Beispiel:

Mehrere Radiostationen benutzen gemeinsam einen Rechner als Informationsquelle für Nachrichten und zur Ausstrahlung von Musik mit Hilfe von MP3-Dateien von der Festplatte. Ein CD-Wechsler mit maximal 100 CD's steht den Musikredakteuren zur Verfügung, um für ihre Sendungen CD's mit Hilfe eines vorgefertigten Prozesses in MP3-Dateien umzuwandeln. Vor Beginn der jeweiligen Konvertierung wird der CD-Wechsler mit den notwendigen CD's von einem Robotersystem automatisch geladen. Der Redakteur gibt zu Beginn des gesamten Konvertierungsprozesses den Maximalbedarf von notwendigen CD's an. Während des Prozesses kann der Musikredakteur CD's zur Konvertierung dynamisch anfordern und wieder freigegeben. Der Rechner (das Betriebssystem) muss dafür sorgen, dass alle Anforderungen von CD's nach eventuellen Wartezeiten befriedigt werden. Zu Beginn werden folgende Maximalbedarfe von CD's von vier Konvertierungsprozessen angegeben:

Prozess	Aktuelle Belegung	Maximale Belegung
A	0	60
B	0	50
C	0	40
D	0	70

Dieser Zustand ist verklemmungssicher, da alle Maximalanforderungen nacheinander erfüllt werden können. Nach einer gewissen Zeit ist folgender Zustand von zugeteilten Plätzen für CD's erreicht:

Prozess	Aktuelle Belegung	Maximale Belegung
A	10	60
B	10	50
C	20	40
D	40	70

Zu diesem Zeitpunkt sind noch 20 Plätze im CD-Wechsler frei. Die potentiellen maximalen Anforderungen sind in folgender Tabelle gezeigt:

Prozess	Aktuelle Belegung	Maximale Anforderung	Maximale Belegung
A	10	50	60
B	10	40	50
C	20	20	40
D	40	30	70

Der Zustand ist verklemmungssicher, da es eine Reihenfolge gibt, in der alle Prozesse bei maximalen Anforderungen zu Ende geführt werden können. Mit dem Rest von 20 CD-Plätzen kann zunächst Prozess C fertig gestellt werden.

Prozess	Aktuelle Belegung	Maximale Anforderung	Maximale Belegung
A	10	50	60
B	10	40	50
C	-	-	-
D	40	30	70

Nach dessen Ende sind 40 CD-Plätze frei. Hiermit kann Prozess B oder D zu Ende geführt werden. Nehmen wir an Prozess D wird als nächstes bedient. Dann sind nach Zuteilung von 30 CD-Plätzen noch 10 CD-Plätze frei.

Prozess	Aktuelle Belegung	Maximale Anforderung	Maximale Belegung
A	10	50	60
B	10	40	50
C	-	-	-
D	-	-	-

Nach dem Ende von Prozess D sind 10+70 also insgesamt 80 CD-Plätze frei. Hiernach kann Prozess A mit 50 CD-Plätzen Anforderung bedient werden. Es bleiben 30 freie CD-Plätze.

Prozess	Aktuelle Belegung	Maximale Anforderung	Maximale Belegung
A	-	-	-
B	10	40	50
C	-	-	-
D	-	-	-

Nach dem Ende von Prozess A werden 60 CD-Plätze frei. Insgesamt stehen 30+60 also insgesamt 90 CD-Plätze zur Verfügung mit denen Prozess B zu Ende geführt werden kann.

Gehen wir nun von dem verklemmungssicheren Zustand wieder aus und nehmen wir an, dass Prozess B weitere 10 CD-Plätze anfordert. Ist der dann entstehende Zustand auch verklemmungssicher?

Prozess	Aktuelle Belegung	Maximale Anforderung	Maximale Belegung
A	10	50	60
B	20	30	50
C	20	20	40
D	40	30	70

Es bleiben nach der Zuteilung noch 10 freie CD-Plätze. Alle Prozesse, die nun mehr als 10 CD-Plätze anfordern können nicht bedient werden. Da aber potentiell alle Prozesse mehr als 10 CD-Plätze anfordern können, könnte dann keiner mehr bedient werden. Eine Verklemmung wäre entstanden. Der entstehende Folgezustand ist also nicht verklemmungssicher und die Anforderung von 10 weiteren CD-Plätzen an Prozess B wird zurückgestellt.

Prozess B muss auf die Zuteilung seiner Anforderung warten, bis ein anderer Prozess zumindesten 10 CD-Plätze freigibt. Z.B. nach dem Ende von Prozess A (bzw. z.B. wenn Prozess D während des Laufs 20 CD-Plätze freigibt), können Prozess B 10 CD-Plätze zugeteilt werden, da dann mindestens 20 CD-Plätze (bzw. 30 CD-Plätze) frei sind und zumindest Prozess B oder C (bzw. B oder C oder D) zu Ende geführt werden kann. D.h. es ist dann wieder ein verklemmungssicherer Folgezustand nach der Zuteilung von 10 CD-Plätzen an B erreichbar.

Bankiers-Algorithmus mit mehreren Betriebsmittelklassen:

I.a. fordern Prozesse mehrere exklusive Betriebsmittel unterschiedlicher Klassen an. Dieses Szenario kann wie folgt modelliert werden. Die verfügbaren Betriebsmittel pro Klasse werden durch folgende Tabelle beschrieben:

Klasse 1	Klasse 2	Klasse 3	Klasse 4
6	3	4	2

Zunächst geben die Prozesse die maximalen Belegungen pro Betriebsmittelklasse an:

Prozess	Klasse 1	Klasse 2	Klasse 3	Klasse 4
A	4	1	1	1
B	0	2	1	2
C	4	2	1	0
D	1	1	1	1
E	2	1	1	0

Nehmen wir an, dass sich nach einer gewissen Zeit folgende Belegung eingestellt hat:

Prozess	Klasse 1	Klasse 2	Klasse 3	Klasse 4
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Dann sind noch folgende Betriebsmittel frei:

Klasse 1	Klasse 2	Klasse 3	Klasse 4
1	0	2	0

Die maximalen weiteren Anforderungen sind:

Prozess	Klasse 1	Klasse 2	Klasse 3	Klasse 4
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Nun kann wie folgt untersucht werden, ob ein Belegungszustand verklemmungssicher ist:

Schritt 1: Suche eine Zeile in der Anforderungsmatrix, in der alle Anforderungen kleiner oder gleich den noch freien Betriebsmitteln sind. Existiert eine solche Zeile nicht, so ist der Zustand nicht verklemmungssicher. Der Entscheidungs-Algorithmus ist zu Ende. (Die vorangegangene Anforderung wird zurückgestellt.) Existiert eine solche Zeile, so fahre bei 2. fort.

Schritt 2: Führe den ausgewählten Prozess zu Ende durch und gebe die Betriebsmittel zurück. Im konkreten Fall ist einzig Prozess D als nächster sicher zu Ende abwickelbar. Die freien Betriebsmittel sind dann:

Klasse 1	Klasse 2	Klasse 3	Klasse 4
2	1	3	1

Schritt 3: Wiederhole nun Schritte 1 und 2, bis entweder alle Prozesse beendet sind, dann ist der Zustand verklemmungssicher oder eine Verklemmung in Schritt 1 auftritt.

Der Bankiers-Algorithmus ist theoretisch einwandfrei; aber er ist in der Praxis wenig relevant, da er davon ausgeht, dass die Prozesse zu Beginn bereits ihre maximalen Bedarfe an Betriebsmitteln abschätzen können, was aber höchstens bei der Stapelverarbeitung der Fall ist.

Kapitel 4

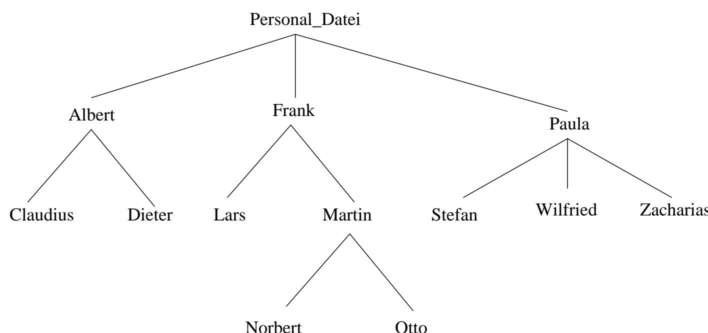
Dateisysteme

Da die Lebensdauer eines Prozesses begrenzt ist, können Informationen langfristig nicht in seinem Adressraum untergebracht werden. Es werden Dateien benötigt, in denen Informationen unabhängig von der Lebenszeit der Prozesse und sogar nach Systemabschaltung und Neustart des Betriebssystems wiedergefunden werden können. Dateien werden statt im Arbeitsspeicher meist auf einer Festplatte (ggf. auch CD, DVD, Magnetband o.ä.) gehalten.

Das Dateisystem ist der Teil des Betriebssystems, der Dateien verwaltet. Bei der Verwaltung ist z.B. zu berücksichtigen, wie Dateien benannt und identifiziert werden, wie sie in einem Mehrprozess- oder Mehrbenutzersystem koordiniert werden (Kontrolle des Mehrfachzugriffs, Schutz vor unerwünschten Zugriffen) und wie sie zur guten Auffindbarkeit untergebracht werden (z.B. in einem hierarchischen Verzeichnisbaum).

4.1 Strukturierung und Typisierung von Dateien

Bie vielen älteren Betriebssystemen (z.B. CP/M) bestanden Dateien aus einer Folge von Sätzen (Records) fester Länge. Diese Strukturierung rührte daher, dass die Ein-/Ausgabemedien z.B. Lochkartenleser mit 80 Zeichen oder Drucker mit 132 Zeichen pro Zeile waren. Eine weitere Organisationsform ist eine Datei als einen Baum mit Sätzen unterschiedlicher Länge darzustellen. Jeder Satz hat hierbei einen Schlüssel, der seine Position im geordneten Baum eindeutig festlegt. Dies ist schneller als bei der sequentiellen Organisation, wenn auf einen Satz mit einem bestimmten Schlüssel zugegriffen werden soll. Auch das Einfügen und Löschen von Sätzen mit einem bestimmten Schlüssel ist unproblematisch.



Heute weit verbreitete Betriebssysteme, wie Unix- und Windows-Versionen, organisieren Dateien lediglich als eine Folge von Bytes. Die Strukturierung der Dateien wird komplett den Anwendungsprogrammen überlassen. Dies ermöglicht die höchste Flexibilität der Software.

Damit Anwendungen und Betriebssystem feststellen können, was mit einer Datei gemacht werden soll und darf, werden verschiedene Dateitypen eingeführt. Unix- und Windows-Betriebssysteme unterscheiden z.B. gewöhnliche Dateien, die ASCII-Daten oder Binärdaten enthalten, Verzeichnisse zur Strukturierung des Dateisystems, zeichenorientierte Dateien zur Abbildung des Ein-/Ausgabesystems auf das Dateisystem und blockorientierte Dateien zur Modellierung der Sekundärspeicher (wie Festplatten, Band-, CD- und DVD-Laufwerke).

Der Dateityp darf nicht mit den Dateinamenserweiterungen (Extensions) verwechselt werden. Solche Erweiterungen erlauben Windows-Betriebssysteme (z.B. .c, .java, .exe, .com, .doc) und Unix-Betriebssysteme (z.B. .c, .cpp, .java, .gz). Hierbei wird festgelegt, was Anwendungsprogramme mit Dateien machen können (z.B. .java: der Java-Compiler darf diesen Quellcode übersetzen). Natürlich haben Dateien mit einer bestimmten Erweiterung auch einen bestimmten Typ (z.B. ist eine Java-Quelle eine gewöhnliche ASCII-Datei). Die Umkehrung gilt aber nicht.

4.2 Dateizugriff und Dateioperationen

In älteren Betriebssystemen gibt es nur sequentiellen Zugriff. Das bedeutet alle Bytes einer Datei auch bei Suchoperationen sequentiell nacheinander gelesen werden müssen. Der Grund dafür liegt in dem damals verwendeten Speichermedium, dem Magnetband. Neuere Betriebssysteme kennen häufig diese Zugriffsart nicht mehr. Vielmehr kann auf bestimmte Sätze einer Datei (z.B. über einen Schlüssel) direkt zugegriffen werden. Dieser Direktzugriff (Direct Access) wurde durch den Einsatz von Festplatten-Laufwerken möglich.

Zum Erzeugen, Löschen, Umbenennen von Dateien und Arbeiten in Dateien werden meist folgende Operationen zur Verfügung gestellt:

- Erzeugen (Create): Die Datei wird benannt und einige Attribute zur Verwaltung initialisiert. Sie enthält keine Daten.
- Löschen (Delete): Die Datei wird aus dem Dateisystem entfernt.
- Umbenennen (Rename): Die Datei wird umbenannt.

- Öffnen (Open): Bevor mit einer Datei gearbeitet werden kann, muss sie geöffnet werden. Hierbei werden die Dateiattribute überprüft (z.B. Zugriffsberechtigung) und Teile der Datei werden in den Arbeitsspeicher transferiert, um schneller darauf zugreifen zu können.

- Schließen (Close): Die Daten der Datei werden vom Arbeitsspeicher entfernt und ggf. auf die Datei zurückgeschrieben. Der Platz für die Dateiattribute im Arbeitsspeicher wird ebenfalls freigegeben. Arbeiten im Dateiinhalt sind nun wieder erst nach dem Öffnen der Datei möglich.

- Lesen (Read): Eine bestimmte Anzahl von Daten werden ab der aktuellen Position in der Datei in einen Puffer im Arbeitsspeicher gelesen. Den Puffer muss das aufrufende Programm zur Verfügung stellen.

- Schreiben (Write): Daten werden ab der aktuellen Position geschrieben. Ist die aktuelle Position am Dateiende, so wird die Datei vergrößert, sonst werden vorhandene Daten überschrieben. Das Schreiben geschieht meistens ebenfalls gepuffert.

- Anfügen (Append): Hier werden Daten am Dateiende angefügt.

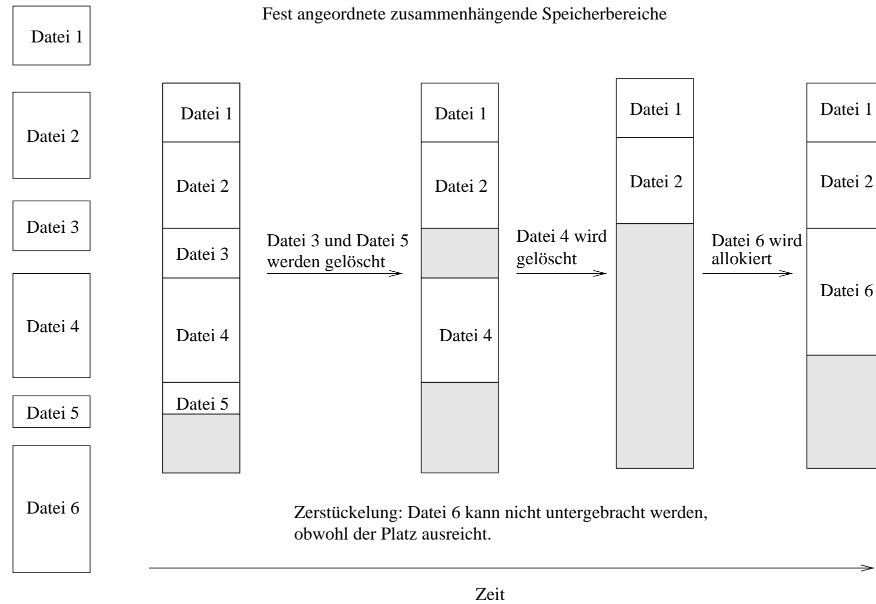
- Suchen (Seek): Der Seek-Befehl sucht eine bestimmte Position in der Datei, auf die die aktuelle Position der Datei zum Lesen und Schreiben gesetzt wird.

4.3 Implementierungs-Möglichkeiten eines Dateisystems

Daten aus Dateien und Verzeichnissen werden auf Blöcken einer Festplatte untergebracht. Daher ist es Aufgabe des Dateisystems, die Zuordnung zwischen diesen Blöcken und den Dateien festzuhalten.

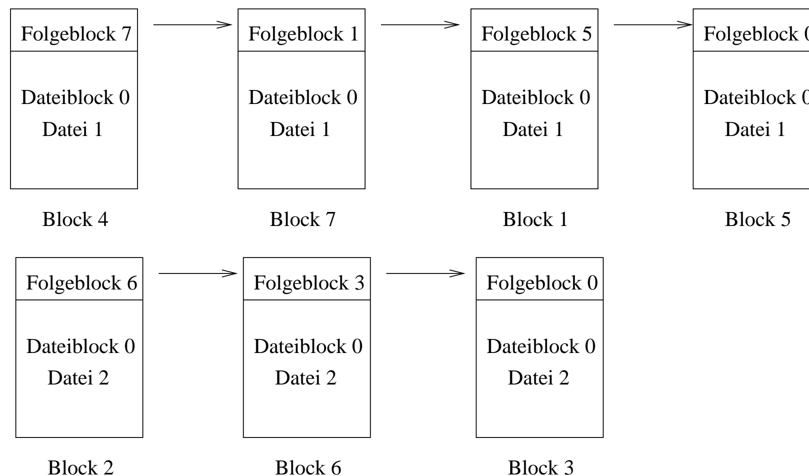
4.3.1 Konsekutive Blöcke

Die einfachste Methode ist, die Daten der Datei in aufeinanderfolgenden Blöcken der Festplatte zu speichern. Die Verwaltung ist einfach: Zu jeder Datei braucht nur die Anfangsblocknummer und die Anzahl der Blöcke in einer Tabelle ermerkt werden. Weiterhin ist das Laden der Datei in den Arbeitsspeicher in einer Operation möglich und daher schnell. Problematisch ist das Verfahren, da die Länge der Datei bei der Allokation bereits feststehen muss; dynamisch wachsende Dateien sind nicht berücksichtigt. Weiterhin wird der Plattenplatz mit der Zeit in kleine Stücke fragmentiert, so dass die Auslastung des Speichers schlecht wird. (Nur zusammenhängende Blöcke sind für die Allokation verwendbar.)



4.3.2 Verkettete Liste von Blöcken

Um dem Problem der Fragmentierung zu begegnen, kann z.B. eine verkettete Liste von Blöcken verwendet werden. Hierbei enthält jeder Block neben Daten aus der Datei die Nummer des nachfolgenden Blocks. Der letzte Block einer Datei enthält eine ungültige Folgeblocknummer als Endekennung. Zur Verwaltung benötigt das Dateisystem lediglich eine Liste mit den Nummern freier Blöcke und eine Tabelle mit Dateien und den Blocknummern der ersten Blöcke. Der Hauptnachteil ist, dass der wahlfreie Zugriff auf die Datei sequentielles Lesen aller vorherigen Blöcke erfordert, um die Blocknummer des Zielblocks zu ermitteln. Hierdurch wird der wahlfreie (random) Zugriff langsam.



4.3.3 Index einer verketteten Liste von Blöcke

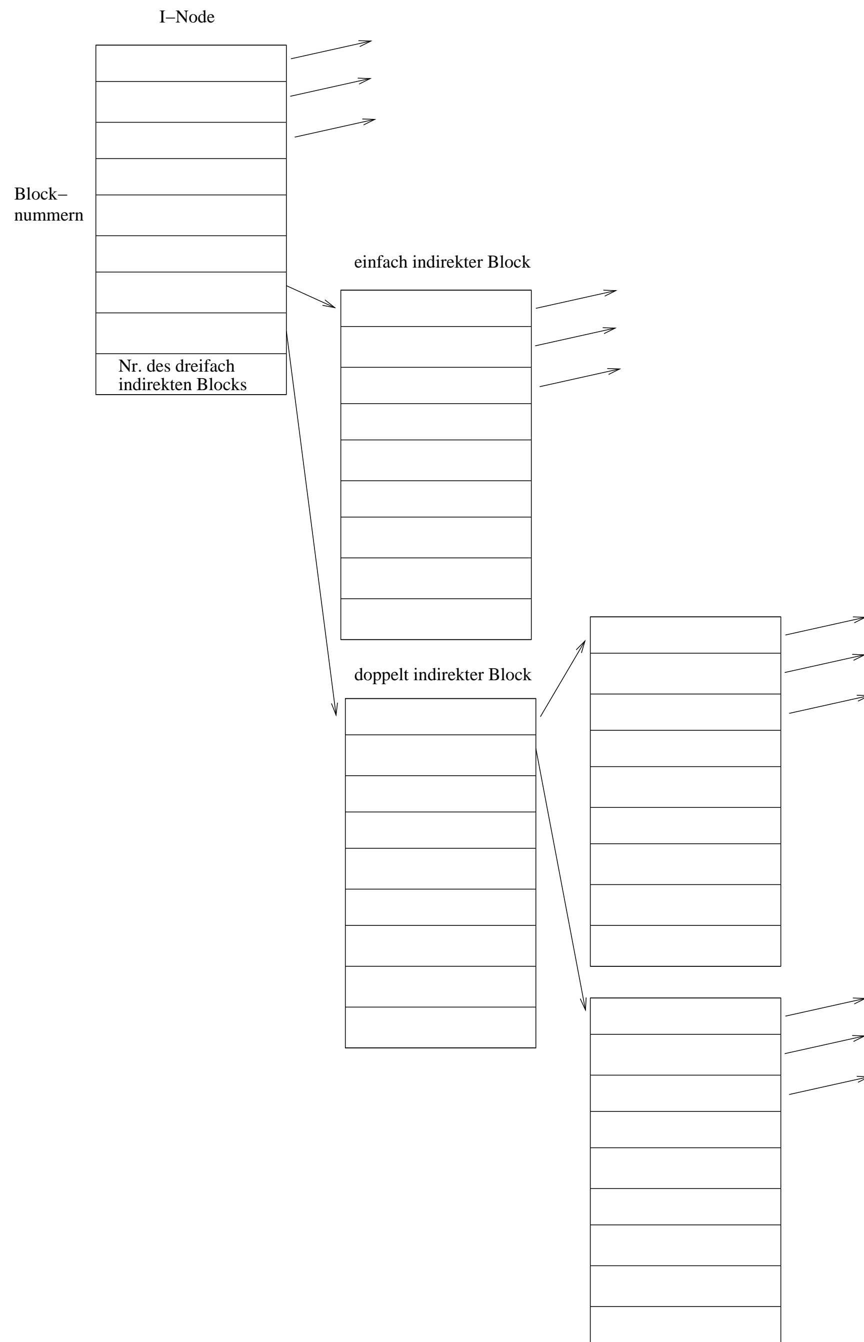
Um den Suchaufwand der physikalischen Blocknummer zu einem logischen Offset vom Dateianfang aus zu reduzieren, kann die Liste der verketteten Blöcke seperat im Arbeitsspeicher gehalten werden. Die Blöcke selbst enthalten dann keine Folgeblocknummern mehr, sondern nur noch Daten.

Physikalischer Block		
1	5	
2	6	← Datei 2
3	0	
4	7	← Datei 1
5	0	
6	3	
7	1	
8	frei	
9	frei	
10	frei	
	⋮	

Das Hauptproblem stellt hierbei der notwendige Platz für die Index-Tabelle dar. Z.B. bei einer Blockgröße von 1 K Byte gibt es bei einer 1 G Byte großen Festplatte 1 M Blöcke (2^{20}). Eine Blocknummer wird also durch 3 Byte (24 Bit) repräsentiert. Zur Darstellung des Index werden also 3 M Byte Arbeitsspeicher benötigt, unabhängig davon welche Dateien gerade benötigt werden.

4.3.4 Index-Knoten (I-Node)

Durch die Index-Knoten Technik wird das Platzproblem gelöst. Hierbei gibt es für jede Datei eine kleine Tabelle fester Länge mit Blockadressen. Für den Fall, dass die zur Verfügung stehenden Tabelleneinträge für die Datei nicht ausreichen wird folgende Technik angewendet: ein Eintrag in der Tabelle zeigt auf einen Block, der seinerseits Blockadressen enthält (einfach indirekt). Jeder Tabelleneintrag bezeichnet also in der Ausgangstabelle nicht einen Block sondern viele (z.B. einige hundert Blöcke). Sollte der Platz immer noch nicht reichen, um die Datei unterzubringen, kann diese Technik iteriert werden ein Block enthält Adressen von Blöcken, die dann die eigentlichen Blockadressen für die Datei enthalten (doppelt indirekt) u.s.w.



4.4 Fallbeispiel: Unix-Dateisystem

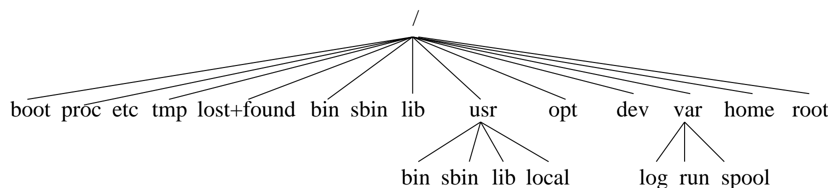
4.4.1 Verzeichnisbaum und Pfad wichtiger Dateien

Das Dateisystem von Unix ist hierarchisch aufgebaut, d.h. die Dateien sind in Form eines Baums geordnet. Als Baum wird eine Struktur bezeichnet, die

aufgezeichnet wie ein umgedrehter stilisierter Baum aussieht. Von der Wurzel verzweigt sich der Baum über Äste immer mehr, bis von den Blättern keine weiteren Verzweigungen mehr ausgehen. Im Unix-Dateisystemen spricht man bei den Verzweigungspunkten von Verzeichnissen oder Directories.

Die Wurzel des Verzeichnisbaums heißt root und wird mit einem / im engl. slash bezeichnet. Die darunterliegenden Verzeichnisse tragen Namen, die - wie auch die von Dateien - im Prinzip frei wählbar sind. Sie dürfen auch mehr als 8 Zeichen lang sein. Innerhalb eines Verzeichnisses müssen Namen eindeutig sein. Zwischen Groß- und Kleinschreibung wird bei Namen, wie generell sonst auch in Unix, unterschieden.

Nachfolgend möchte ich auf das Dateisystem von Linux nach dem Filesystem Hierarchy Standard eingehen. Einen typischen Verzeichnisbaum zeigt die folgende Abbildung. Das Bild ist nicht vollständig, aber es zeigt einige der wichtigsten Verzeichnisse.



Der Betriebssystemkern befindet sich im Verzeichnis /boot und heißt meist unix (bei Linux: vmlinuz).

Das Verzeichnis /proc beherbergt das Prozessdateisystem. Dieses ist ein Pseudodateisystem, das viele interessante Kernelinformationen enthält, die direkt oder mit speziellen Programmen aufbereitet angesehen werden können.

Das Unterverzeichnis /etc enthält wichtige Dateien zur Administration des Systems, wie passwd, group, profile, fstab, exports, hosts, route.conf, inetd.conf, conf.modules usw.

Im Verzeichnis /tmp werden temporäre Dateien abgelegt, die normalerweise spätestens bei einem Neustart des Systems gelöscht werden können.

In sehr seltenen Fällen kann der Verzeichniseintrag einer Datei gelöscht werden, ohne dass die dazugehörende Datei entfernt wird. Solche verlorenen Dateien werden vom File-System-Check (fsck) gefunden und in lost+found wieder zugänglich gemacht.

In /bin stehen elementare Benutzerkommandos, wie login, ls, cp, mv usw. Das Verzeichnis /sbin enthält Programmdateien für die wichtigsten Aufgaben der Systemverwaltung, deren Ausführung dem Superuser (root) vorbehalten ist, z.B. init, fsck und dump. Unterhalb des Verzeichnisses /usr gibt es noch einmal Verzeichnisse /usr/bin und /usr/sbin. In /usr/bin befindet sich das Gros der Programmdateien einer Distribution, z.B. gcc, make, du, lpr, man und mtools. /usr/sbin enthält alle Superuser-Programme, die nicht zum Boot oder für die Arbeit mit Dateisystemen benötigt werden, z.B. die Internet-Dämone für ftp, telnet, rlogin, den Dämon für den Drucker lpd und Programme zur Verwaltung von Benutzern useradd, userdel, usermod.

Das Verzeichnis /lib enthält Systembibliotheken als sogenannte Shared Libraries. Shared Libraries werden von den meisten Kommandos des Betriebssystems dynamisch zur Laufzeit dazugebunden. Der dynamische Linker ld.so befin-

det sich genauso wie die dynamische Standardbibliothek für C-Programme, z.B. libc.so.6, in diesem Verzeichnis . Normalerweise haben Shared Libraries Namen der Form libxyz.so.2.4.33. Der dynamische Linker öffnet in diesem Fall eine Datei mit dem Namen libxyz.so.2. Das Systemprogramm ldconfig sorgt dafür, dass der Namensverweis (hier libxyz.so.2) (symbolischer Link) stets auf die aktuellste Version zeigt (hier libxyz.so.2.4.33). Weitere Shared Libraries befinden sich in den Unterverzeichnissen /usr/lib und /usr/lib/X11R6. In /usr/lib stehen beispielsweise Bibliotheken für C++, Tcl/Tk, JPEG, MPEG usw. /usr/X11R6/lib enthält Bibliotheken für das X Window System.

Bibliotheken, die zu den Programmen statisch hinzugebunden werden, befinden sich ebenfalls in /usr/lib. Ihre Namen sind von der Form lib*.a, z.B. libc.a und libm.a. Im Verzeichnis /usr/lib/gcc-lib befinden sich die vom gcc-Compiler abhängigen Dateien.

In /usr gibt es noch eine Reihe von Unterverzeichnissen für den Benutzerbetrieb. Dazu gehören /usr/man für die Hilfe (Manual-Pages), /usr/include für C-Header-Dateien und /usr/src für Programmquellen, wie die Quelldateien des Linux-Betriebssystems. Um eine deutliche Trennung der originalen Distribution und lokalen Erweiterungen zu erreichen, gibt es hinter dem Verzeichnis /usr/local noch eine Hierarchie, die im Prinzip der /usr-Hierarchie entspricht. Der FH-Standard verlangt, dass dieses Verzeichnis leer bleibt. Damit kann das gesamte /usr-Verzeichnis von einem Server an alle Clients im Netz (ausschließlich mit Leserecht) exportiert werden.

Das Unterverzeichnis /var enthält Daten die sich im Gegensatz zu Daten im /usr Verzeichnis während des Betriebs des Rechners ändern können oder müssen. Im Unterverzeichnis /var/log werden Systemmeldungen gespeichert, z.B. alle Login-Vorgänge in der Datei wtmp und Systemlogdateien wie messages und warnings vom Dämon syslogd. Das Unterverzeichnis /var/run enthält Informationen zum laufenden System, z.B. die Prozessnummer vieler Dämone und die aktuell arbeitenden Benutzer in utmp. Das Unterverzeichnis /var/spool enthält Daten von Mail (/var/spool/mail) und News (/var/spool/news) sowie Druckdateien (/var/spool/lpd).

Das /dev-Verzeichnis beinhaltet Gerätedateien (Devices), die dazu dienen, die Benutzerebene vollständig von der Hardware abzuschirmen. Wann immer ein Benutzerprozess an eine Hardwarekomponente Daten senden oder von dort lesen will, muss er erst eine zu dieser Device gehörende Gerätedatei öffnen. Der Systemverwalter hat die gleichen Möglichkeiten, Zugriffsrechte auf Gerätedateien zu erlauben, wie bei anderen Dateien. Indem nur bestimmten Benutzern, beispielsweise Pseudo-Benutzern ohne Login-Möglichkeit, über ihre Dämonprozesse der Zugriff auf die Gerätedatei erlaubt wird, lassen sich viele Sicherheitsprobleme von vornherein ausschließen. Die folgende Tabelle zeigt Namen wichtiger Gerätedateien und ihre zugehörige Hardware:

Namensbeispiele	Hardware
hda,hdb,hdc,hdd	IDE-Festplatten, ATAPI-Laufwerke
sda, sdb,..., sdp	SCSI-Laufwerke
fd0	Floppy-Disk
md0,..., md4	RAID-Laufwerke
tty, tty0,...	virtuelles Terminal
ptyp0, ttyp0,...	Pseudo Master-/Slave-Terminal für X
cua0,cua1	serielle Schnittstellen
psaux, inportbm	spezielle Mäuse
lp1	Drucker an der parallelen Schnittstelle
mem, kmem	physikalischer, virtueller Arbeitsspeicher
ram0	Ram-Disk
rmt0	QIC-Streamer
st0	SCSI-Streamer
null	Papierkorb

Für kommerzielle Erweiterungen, die nicht Linux-Bestandteile sind, dient das Verzeichnis /opt.

Das /home-Verzeichnis enthält als Unterverzeichnisse die Heimatverzeichnisse der Benutzer, z.B. /home/kremer.

/root ist das Heimatverzeichnis des Superusers.

4.4.2 Pfadangaben, Heimatverzeichnis und elementare Kommandos

Zur eindeutigen Adressierung einer einzelnen Datei benutzt man sogenannte Pfadangaben im engl. Path. Sie werden wie folgt gebildet: Von der Wurzel aus folgt man dem eindeutig bestimmten Pfad zur gesuchten Datei. Dabei werden die Namen der Verzeichnisse durch / (nicht \ wie bei DOS) voneinander getrennt. Pfadangaben, die mit einem / für das root-Verzeichnis beginnen, werden absolute Pfadangaben genannt.

Nach dem login befindet man sich (normalerweise) in einem Verzeichnis, das der eigenen Benutzerkennung entspricht. Daher nennt man dieses Verzeichnis auch Heimatverzeichnis oder Home Directory. Dass man sich dort befindet bedeutet, dass alle Befehle, die auf Dateinamen ohne Pfadangaben wirken, nur Dateien in diesem sogenannten aktuellen Verzeichnis betreffen. Man kann sich z.B. mit dem Befehl ls (im engl. list) ein Inhaltsverzeichnis der existierenden Dateien in dem aktuellen Verzeichnis anzeigen lassen.

In seinem Home Directory kann jeder Benutzer nach seinem Bedarf Dateien und weitere Unterverzeichnisse anlegen, ohne andere Benutzer zu stören. Die Home Directories tragen üblicherweise den login-Namen des Benutzers und sind im Verzeichnis /home des Verzeichnisbaums angesiedelt. Das Home Directory des Benutzers mit dem login-Namen kremer wäre z.B. /home/kremer. Durch Navigation durch das Dateisystem kann man das aktuelle Verzeichnis ändern. Hierzu dient der Befehl cd eine Abkürzung für change directory. Als Parameter erhält cd eine Pfadangabe z.B.: cd /home/kremer/vorlesung. Mit dem Parameter cd .. gelangt man in das (es gibt nur eins) jeweilig übergeordnete Verzeichnis. Man befindet sich dann im Verzeichnis /home/kremer. Will man in das Verzeichnis /home/kremer/vorlesung/linux wechseln, so braucht man nun nicht

den gesamten absoluten Pfad angeben, sondern es reicht die sogenannte relative Pfadangabe vom aktuellen Verzeichnis /home/kremer aus: cd vorlesung/linux. Merken Sie sich, dass der relative Pfad niemals mit einem / beginnt, dies ist nämlich Kennzeichen für einen absoluten Pfad.

Da relative Pfadangaben nur nützlich sind, wenn man auch weiß, in welchem aktuellen Verzeichnis man sich befindet, gibt es den Befehl pwd (im engl. print working directory), der den absoluten Pfad des aktuellen Verzeichnisses ausgibt. Das aktuelle Verzeichnis hat auch einen besonderen Namen nämlich . (einen Punkt).

Der Befehl mkdir (im engl. make directory) dient zum Erstellen eines neuen Verzeichnisses. Ein leeres Verzeichnis kann mit rmdir (im engl. remove directory) gelöscht werden. Enthält das Verzeichnis noch Dateien oder Unterverzeichnisse, so müssen diese erst mit dem Befehl rm (im engl. remove) oder rmdir gelöscht werden. Achtung: Unix behandelt Benutzer wie mündige Erwachsene und löscht Dateien ohne vorherige Rückfrage. Wem das nicht behagt, der kann die Option rm -i für Löschen mit Rückfrage verwenden. Nun ist es zugegebenermaßen etwas lästig immer rm -i einzugeben. Dafür bieten die meisten Unix-Shells einen sogenannten Alias-Mechanismus, mit dem man Kommandos durch eine Textersetzung umdefinieren kann, z.B.:

```
alias del='rm -i'
```

Das Kopieren bzw. Verschieben von Dateien oder auch ganzen Verzeichnissen inklusive des darunterliebenden Teilbaums erledigen cp (im engl. copy) bzw. mv (im engl. move). mv wird auch zum Umbenennen von Dateien eingesetzt, da Unix dafür kein eigenes Kommando kennt.

Beide Kommandos verlangen als ersten Parameter den Namen der Quelle und als zweiten Parameter den Namen des Ziels. Beispielsweise kopiert der Befehl

```
cp /home/kremer/vorlesung/linux/teil1 .
```

die Datei teil1 in das aktuelle Verzeichnis (ein Punkt).

Mit mv kann man neben Dateien auch komplette Unterverzeichnisbäume in andere Verzeichnisse verschieben, z.B. :

```
mv /home/kremer/vorlesung/ /home/vorlesung
```

Im Gegensatz dazu bearbeitet cp ganze Teilbäume nur mit der Option -r, z.B.:

```
cp -r /home/kremer/vorlesung /opt/vorlesung
```

Den Inhalt einer Datei kann man mit dem Kommando cat (im engl. für catalog) ansehen. Ist die Datei länger als ein Fenster, so sollte man eines der Kommandos more oder less verwenden, die die Datei seitenweise anzeigen.

4.4.3 Masken

Anstelle von Dateinamen besteht in allen gängigen Unix-Shells die Möglichkeit sogenannte Masken zu verwenden. Diese sehen wie Dateinamen aus, enthalten aber an einer oder mehreren Stellen sogenannte Metazeichen oder Wildcards. Im Kartenspielen sind Wildcards Karten, deren Wert der Ausspielende nach

Belieben festlegen kann. Am häufigsten werden * für beliebig viele (auch kein) Zeichen und ? für ein beliebiges Zeichen verwendet. So benennt man z.B. alle Dateien, deren Namen mit f beginnen und mit r aufhören durch f*r. Will man nur die Namen mit vier Zeichen, so schreibt man f??r. Der Befehl cp /home/test/* . kopiert alle Dateien des Verzeichnisses /home/test ins aktuelle Verzeichnis (nicht die Unterverzeichnisse von /home/test).

Nützlich sind auch die eckigen Klammern [ ], mit denen man einen bestimmten Bereich von Zeichen angeben kann. Dies wirkt wie ein eingeschränktes ?: Etwa steht [abc] für entweder a oder b oder c. [A-Z] steht für einen beliebigen Großbuchstaben. Schreibt man [!A-Z], so meint das alle Zeichen, die nicht Großbuchstaben sind.

Metazeichen lassen sich natürlich auch kombinieren, so trifft [Kk]*l?er* beispielsweise auf die Namen Kabelfernsehen, kapitulieren, Kerzenhalter und Koelner zu nicht aber auf Kerzenleuchter zu. Eine Beschreibung des Ersetzungsmechanismus finden Sie u.a. in der man-page zur bash (bourne again shell) im Abschnitt Pathname Expansion. Soll verhindert werden, dass die Shell ein Metazeichen interpretiert, so kann man es durch Voranstellen eines \ vor das Metazeichen ausmaskieren oder man setzt die ganze Zeichenfolge in einfache Anführungszeichen. Dies bewirkt, dass kein Metazeichen ersetzt wird, z.B. wird durch '[klm]*' echt das Wort [klm]* getroffen.

4.4.4 Zugriffsschutz von Dateiobjekten

Zum Schutz von Objekten wie Dateien und Verzeichnissen vor dem Zugriff durch unberechtigte Benutzer verwaltet Unix Zugriffsrechte. Neben Verzeichnissen und Dateien gibt es unter Unix weitere Objekte, wie Gerätetreiber oder devices und Kommunikationsobjekte, wie named pipes oder sockets. Zunächst vermerkt Unix zu jedem Objekt seinen Besitzer oder Owner oder User. Der Owner ist zunächst der Erzeuger eines Objektes. Er kann aber durch den Befehl chown (im engl. change owner) geändert werden, d.h. der Eigentümer darf sein Objekt an einen anderen Benutzer verschenken.

Unix unterscheidet beim Zugriffsschutz drei verschiedene Benutzerklassen. Es werden Rechte des Eigentümers des Objekts, bzw. von Benutzern, die in derselben sogenannten Gruppe oder group wie der Eigentümer des Objekts sind und von anderen Benutzern außerhalb dieser Gruppe (others) verwaltet. Für jedes Objekt werden bzgl. dieser drei Klassen drei Typen von Rechten überwacht, und zwar das Recht auf ein Objekt lesend (r für read), schreibend (w für write) und zur Ausführung (x für execute) zuzugreifen.

Der Befehl ls -l mit der Option -l für long format liefert beispielsweise folgende Ausgabe:

```
total 24
drwx------   5 root     root         1024 Jul 16 11:18 Desktop
drwx------   2 root     root         1024 Jul 16 15:40 Mail
drwxr-x---   2 root     root         1024 Jul 23 17:30 bin
drwxr-x---   2 root     root         1024 Aug 20 10:55 c++src
drwxr-x---   2 root     root         1024 Jul 16 21:43 csrc
-rw-r--r--   1 root     root            0 Sep 28 09:30 dirlist
drwxr-x---   9 root     root         1024 Sep 27 17:02 dirrechte
drwxr-x---   2 root     root         1024 Aug 19 16:31 javascript
```

```
drwxr-x---   4 root     root         1024 Aug  3 11:48 javasrc
drwxr-x---  16 root     root         1024 Aug 20 11:17 kshbuch
drwxr-x---   2 root     root         1024 Jul 29 11:01 kshsrc
drwxr-x---   3 root     root         1024 Jul 16 11:54 loadlin
-rw-------   1 root     root         4726 Jul 16 16:11 mbox
drwxr-x---  15 root     root         1024 Jul 23 16:07 mpich
drwx------   2 root     root         1024 Jul 29 17:52 nsmail
drwxr-x---   3 root     root         1024 Aug 19 16:25 perlsrc
drwxr-x---  16 root     root         1024 Mar 19  1998 ppmpi_c
drwx------  19 root     root         1024 Jul 23 08:38 pvm3
drwxr-x---   2 root     root         1024 Jul 23 08:23 tarfiles
drwxr-x---   6 root     root         1024 Jun 24  1994 unify
drwxr-x---   2 root     root         1024 Aug  3 10:23 util
```

Hierbei sind zunächst die ersten zehn Zeichen jeder Zeile für uns interessant. Das erste Zeichen gibt den Dateityp an, z.B. steht d für directory und - für eine normale Datei. Die nächsten neun Zeichen bezeichnen den Inhalt von neun Flags, die die Zugriffsrechte in der Reihenfolge user, group und others definieren.

Das Ausführrecht ist für Programmdateien gedacht; es besagt, dass unter dem Dateinamen keine Daten sondern ein ausführbares Programm gespeichert ist. Durch Aufruf des Programmnamens kann dann das Programm zur Ausführung gebracht werden.

Rechte können mit dem Befehl chmod geändert werden, der als Parameter die gewünschte Änderung und den Dateinamen enthält. Die Änderung besteht aus der Angabe der betroffenen Benutzer u für user, g für group, o für others und a für all. Danach folgt ein + für das Hinzufügen, ein - für den Entzug von Rechten und ein = für das Setzen von Rechten. Schließlich folgt eine Kombination von r, w und x für die betroffenen Rechte. Folgende Beispiele geben einen Überblick über den chmod-Gebrauch:

```
chmod ug+x programm
chmod a-rw programm
chmod u+w,g-wx,o-rwx datei
chmod u=rw,g=r,o=- datei
chmod u+rw,g=r,o= datei
```

Eine andere gebräuchliche Parameterübergabe für chmod ist die Oktaldarstellung der Zugriffbits, z.B. bedeutet chmod 764 datei folgende Setzung:

```
Owner: rwx Group: rw- Others: r--
```

Dies entspricht im Dualzahlensystem:

```
111 110 100
```

oder im Oktalzahlensystem:

```
764
```

Bei Verzeichnissen haben die Rechte eine andere Bedeutung. Hierbei bedeutet:

r: Der Befehl ls darf ausgeführt werden.

w: Dateien können im Verzeichnis neu erzeugt und gelöscht werden, falls die Rechte der Dateien dies erlauben.

x: Mit cd kann in das Verzeichnis gewechselt werden.

Für den Zugriff auf ein Verzeichnis sollte man i.d.R. mindestens die Rechte r und x. haben. Mit dem Schreibrecht sollte man vorsichtig umgehen, was group und others angeht, denn bei gesetztem Schreibrecht w kann ein anderer Benutzer leicht ein trojanisches Pferd in das eigene Verzeichnis einschmuggeln.

Dies geschieht z.B. auf folgende Weise: Der Angreifer kopiert ein Skript mit dem Namen eines Systembefehls, wie ls, in das offene Verzeichnis. Neben ls werden aber eine Reihe für den Angreifer unerlaubter Befehle ausgeführt, die der Angegriffene unbemerkt ausführt, wenn er das Skript ls anstatt des Systembefehls ls aufruft. Dies kann z.B. geschehen, wenn in diesem Verzeichnis noch andere ausführbare Dateien des Angegriffenen stehen und er deshalb im Suchpfad für Programmdateien PATH das Verzeichnis vor das Systemverzeichnis /bin gesetzt hat. Hierdurch kann der Angreifer z.B. Zugriff auf Dateien erhalten, für die er keine Leseberechtigung hat. Das Skript ls sieht z.B. folgendermaßen aus:

```
cp nichtlesbare_datei /home/gangster/datei
chown gangster /home/gangster/nichtlesbare_datei
/bin/ls
rm ls
```

An diesem einfachen Beispiel sieht man, dass die Schreibrechte eines Verzeichnisses nur in seltenen Ausnahmefällen für group oder others gesetzt sein sollten. Will man Dateien zwischen Benutzern austauschen, so braucht man auch nicht das Schreibrecht auf ein Verzeichnis einem anderen Benutzer geben. Vielmehr sollte einem der andere Benutzer Leserecht auf die Dateien geben, die man dann selber in sein Verzeichnis kopieren kann.

Zum weiteren Schutz vor unberechtigten Zugriffen sollte man den Befehl umask (user file creation mask) vor jeder Sitzung ausführen. Dies kann z.B. in einem login-Skript der Shell geschehen. umask legt die Rechte neu angelegter Dateien fest. Dabei wird jedem Recht r, w und x von user, group und others ein Bit zugeordnet, das, wenn es gesetzt ist, das Recht entzieht. Der Parameter für umask wird dann betreffend user, group und others dezimal codiert. Der Befehl umask 022 bewirkt z.B., dass das Schreibrecht für group und others zurückgesetzt ist.

umask wirkt allerdings nicht bei den Befehlen cp und mv, bei denen die Originalrechte der Dateien übernommen werden.

4.4.5 Realisierung des Zugriffsschutzes

In der Datei /etc/passwd sind für jeden Benutzer seine individuelle Charakteristika aufgeführt. Dazu ist für jeden Benutzer in einer separaten Zeile der Benutzername oder login-Name, das verschlüsselte Passwort, eine Benutzerkennzahl oder UserID, eine (primäre) Gruppenkennzahl oder GroupID, ein Kommentar, der Pfad des Heimatverzeichnisses und die login-Shell verzeichnet. Nachfolgend ist der Inhalt der Datei /etc/passwd eines Linux-Beispielsystems gezeigt:

```
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:daemon:/sbin:/bin/bash
lp:x:4:7:lp daemon:/var/spool/lpd:/bin/bash
news:x:9:13::/var/lib/news:/bin/bash
uucp:x:10:14::/var/lib/uucp/taylor_config:/bin/bash
games:x:12:100::/tmp:/bin/bash
man:x:13:2::/var/catman:/bin/bash
at:x:25:25::/var/spool/atjobs:/bin/bash
postgres:x:26:2:Postgres Database Admin:/var/lib/pgsql:/bin/bash
lnx:x:27:27:LNX Database Admin:/usr/lib/lnx:/bin/bash
mdom:x:28:28:Mailing list agent:/usr/lib/majordomo:/bin/bash
yard:x:29:29:YARD Database Admin:/usr/lib/YARD:/bin/bash
wwwrun:x:30:-2:Daemon user for apache:/tmp:/bin/bash
squid:x:31:-2:WWW proxy squid:/var/squid:/bin/bash
fax:x:33:14:Facsimile Agent:/var/spool/fax:/bin/bash
gnats:x:34:-2:Gnats Gnu Backtracking System:/usr/lib/gnats:/bin/bash
empress:x:35:100:Empress Database Admin:/usr/empress:/bin/bash
adabas:x:36:100:Adabas-D Database Admin:/usr/lib/adabas:/bin/bash
amanda:x:37:6:Amanda Admin:/var/lib/amanda:/bin/bash
ixess:x:38:29:IXware Admin:/usr/lib/ixware:/bin/bash
ftp:x:40:2:ftp account:/usr/local/ftp:/bin/bash
nobody:x:-2:-2:nobody:/tmp:/bin/bash
kremer:x:100:100::/home/kremer:/bin/ksh
```

Der Name einer Gruppe sowie weitere sekundäre Gruppen zu denen der Benutzer gehört ist in der Datei /etc/group registriert.

```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:
tty:x:5:
disk:x:6:
lp:x:7:
wwwadmin:x:8:
kmem:x:9:
wheel:x:10:
mail:x:12:
news:x:13:news
uucp:x:14:uucp,fax,root
shadow:x:15:root
dialout:x:16:root
at:x:25:at
lnx:x:27:
mdom:x:28:
yard:x:29:
dosemu:x:30:
game:x:40:
users:x:100:
nogroup:x:-2:root
```

Die durch Doppelpunkt getrennten Felder haben von links nach rechts folgende Bedeutung: Gruppenname, Passwort, Gruppennummer und Liste der login-Namen der Gruppenmitglieder.

Das Kommando id liefert einem Benutzer seine (effektive) UserID und GroupID sowie alle Gruppen, denen er angehört. Wenn root id aufruft erscheint z.B.:

```
uid=0(root)gid=0(root)groups=0(root),1(bin),14(uucp),
15(shadow),16(dialout),65534(nogroup)
```

Der sogenannte Superuser mit dem Namen root hat die UserID 0; er darf - unabhängig von der Rechtevergabe durch die Besitzer der Objekte - auf alle Objekte zugreifen und die Rechte und die Besitzer der Objekte ändern.

Der Zugriffsschutz ist nun über die UserID und GroupID eines Benutzers wie folgt realisiert. Nach dem erfolgreichen Anmelden am System, wird für den Benutzer das in /etc/passwd angegebene Programm gestartet. Dieses Programm ist normalerweise eine Shell. Es ensteht ein Shell-Prozess, der u.a. als Kennzeichen - wie alle von diesem Benutzer gestarteten Prozesse - dessen UserID und GroupID trägt. Auf der anderen Seite trägt auch jedes Objekt als Kennzeichen die UserID und GroupID seines Eigentümers. Will der Prozess auf ein Objekt zugreifen, so vergleicht das Betriebssystem die UserID und GroupID des Prozesses mit der UserID und GroupID des Objekts. Stimmt die UserID von Prozess und Objekt überein, so gelten die Owner-Rechte. Stimmt die GroupID von Prozess und Objekt überein, so gelten die Group-Rechte. Stimmen weder UserID noch GroupID von Prozess und Objekt überein, so gelten die Others-Rechte.

An diesem Zugriffsprinzip gibt es nun einen Schwachpunkt. Man kann nämlich anderen Benutzern nur uneingeschränkt den Zugriff auf Objekte geben. Das bedeutet z.B., wenn man eine Datei hat in der bestimmte Sätze durch fremde Benutzer gelesen werden dürfen und andere nicht, ist dies mit den bisherigen Mitteln nicht möglich. Man könnte sich aber vorstellen, dass fremde Benutzer keinen direkten Zugriff auf das Objekt haben aber über ein Programm zugreifen dürfen. Dieses Programm müßte also einen kontrollierten Zugriff auf ein ansonsten geschütztes Objekt erlauben. Dies ist in Unix ebenfalls möglich, und zwar mit dem sogenannten SetID-Mechanismus. Aber wie funktioniert das?

Bei der Schilderung des Zugriffsprinzips haben wir - der besseren Übersicht wegen - eine Feinheit weggelassen. Ein Prozess trägt nämlich im Unterschied zu einem Objekt zwei UserID's und GroupID's. Diese werden reale und effektive UserID und GroupID genannt. Die Aufgaben realer und effektiver ID's sind nun verschieden. Während die realen ID's immer eindeutig den Erzeuger eines Prozesses kennzeichnen, sind die effektiven ID's für den Zugriff auf Objekte nach dem oben geschilderten Verfahren zuständig.

Normalerweise, d.h. z.B. nach dem login, sind reale und effektive ID's gleich. Während die realen ID's also für jeden erzeugten Prozess gleich den beim login aus /etc/passwd ermittelten ID's sind, können die effektiven ID's auch die Werte eines anderen Benutzers annehmen.

Startet ein fremder Benutzer nun ein "normales" Programm, auf das er also Group- oder Others-Ausführrechte besitzt, so hat der entstehende neue Prozess die realen und effektiven ID's des aufrufenden Benutzers. Er kann also auch über dieses Programm auf ein geschütztes Objekt keinesfalls zugreifen.

Will man dem fremden Benutzer nun über das Programm den Zugriff auf ein ansonsten geschütztes Objekt ermöglichen, so muss mindestens eine der effektiven ID's auf die ID des Eigentümers der Programmdatei gesetzt werden. Genau

dieses leistet der SetID-Mechanismus. Mit dem Befehl chmod u+s programm wird die effektive UserID jedes Benutzers, der das Programm ausführen kann zur UserID des Eigentümers des Programms. Mit chmod u+g programm gilt das entsprechende für die effektive GroupID.

Die Überwachung des Zugriffs kann nun im Programm durch die Abfrage der realen UserID und GroupID des Prozesses mit einem Systemaufruf erfolgen.

Eine Anwendung des SetID-Mechanismus ist folgende: Systemprogramme, die für nicht-root-Benutzer ausführbar sind, wie z.B. passwd, erhalten über diesen Mechanismus Zugriffsrechte auf ansonsten geschützte Systemdateien, wie z.B. /etc/passwd.

Der Befehl ls -l /etc/passwd ergibt:

```
-rw-r--r-- 1 root root   1158   Aug 2 /etc/passwd
```

Der Befehl ls -l /usr/bin/passwd ergibt:

```
-rwsr-xr-x 1 root shadow 181964 Mar 4 /usr/bin/passwd
```

An dem Kennzeichen s statt x für den Owner sieht man, dass das SetUserid-Bit gesetzt ist. Für das SetGroupID-Bit stünde ein s anstatt des x bei den Gruppenrechten.

Im Zusammenhang mit dem Zugriffsschutz sind u.a. noch folgende Kommandos interessant.

Das Kommando chown dient dazu die Eigentümerkennung (UserID) und ggfs. den Eigentümer-Gruppennamen (GroupID) eines Objekts zu ändern. Natürlich werden gesetzte SetID-Bits bei Schenkungen von Programmdateien zurückgesetzt, sonst könnte man sich ja auf diese Weise unerlaubten Zugriff auf Objekte fremder Eigentümer verschaffen.

Mit dem Kommando chgrp kann man die Gruppenzugehörigkeit eines Objekts ändern.

Mit dem Kommando newgrp kann ein Benutzer seine reale GroupID ändern, z.B. kann root das Kommando newgrp bin ausführen, um in die sekundäre Gruppe bin zu wechseln. Mit dem Kommando exit kehrt er wieder zur alten Gruppe zurück.

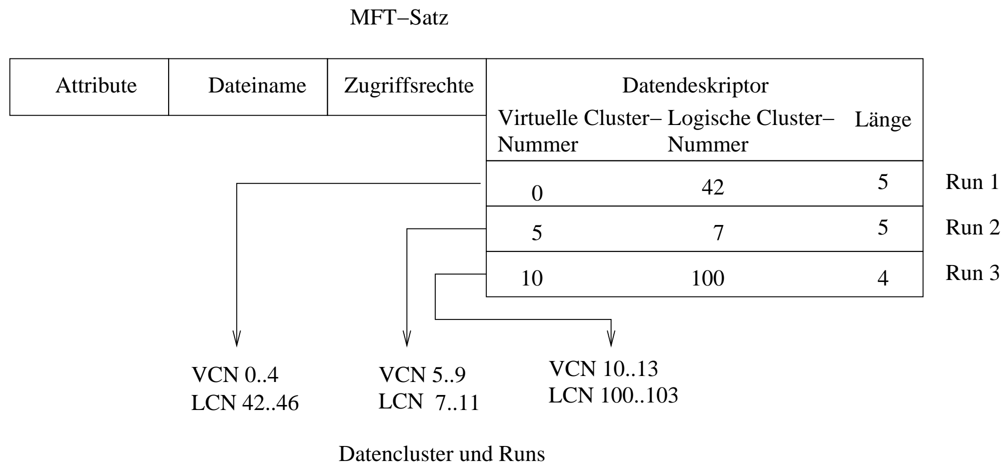
4.5 Fallbeispiel: Windows NT-Dateisystem (NTFS)

Das Dateisystem von Windows NT (NTFS: NT File System) kennt genau wie Unix und DOS nur unstrukturierte Byteströme als Dateiinhalte.

Die Tabelle der Dateideskriptoren nennt sich in NTFS Master File Table (MFT). Ein Dateideskriptor ist also (mindestens) ein Satz in der MFT. Die Anzahl der Dateiattribute innerhalb eines Deskriptors ist variabel. Daher kann sich ein Dateideskriptor auch auf mehrere MFT-Sätze erstrecken.

Wie beim Unix-Dateisystem werden bei NTFS nur kleine Dateien mit allen Attributen in einem MFT-Satz untergebracht. Größere Dateien werden in sogenannten Clustern physikalisch gestreut gespeichert. Dabei versucht NTFS-Cluster möglichst physikalisch konsekutiv zu speichern. In diesem Zusammenhang spricht man von sogenannten Runs: Ein Run ist ein zusammenhängender Bereich mehrerer Cluster.

Im besten Fall besteht den Datenstrom aus einem Run mit allen Clustern, im schlechtesten Fall ist jeder Run nur ein Cluster lang. Ein MFT-Satz sieht wie folgt aus:



Datencluster und Runs

Windows NT verwaltet Objekte, z.B. Speichersegmente oder Dateien, und schützt diese vor unberechtigtem Lesen, Ändern und Löschen. Hierbei werden Zugriffsdomänen definiert, die zu einem Prozess gehören. Dabei besteht eine Zugriffsdomäne aus Paaren der Form: Objekt - erlaubte Operation auf dem Objekt. Ein Objekt gehört i.d.R. zu mehreren Zugriffsdomänen.

Ein Prozess trägt in Windows NT analog zu Unix eine Benuteridentifikation und eine Gruppenidentifikation. Diese Identifikation nennt sich Access Token. Bei einem Systemaufruf werden erweiterte Rechte auf Objekte eingeräumt.

Die Rechte an Objekten bzgl. Domänen können in einer Matrix z.B. wie folgt eingetragen sein:

Domäne	Datei 1	Datei 2	Scanner	Ethernet
1	read,write, execute		read	read, write
2		read	read	
3	read, execute	read,write		

Eine solche Matrix zu speichern, erweist sich in der Praxis als ineffizient, da die meisten Einträge leer sind. Deshalb werden zeilenweise oder spaltenweise nur Einträge gespeichert, die nicht leer sind.

Bei einer zeilenweisen Speicherung spricht man von sogenannten Capabilities (bzw. Capability-Listen). Für jede Zugriffsdomäne werden die Objekten mit all ihren Rechten gespeichert.

Bei einer spaltenweisen Speicherung spricht man von sogenannten Access Control Lists (ACL's). Windows NT verwendet ACL's. Jeder Eintrag in der Liste besteht aus der Identifikation (Benutzer, Gruppe) einer Reihe von Rechten und einem Flag, das anzeigt, ob die Rechte gewehrt oder verwehrt werden. Ein Beispie könnte wie folgt aussehen:

Flag	Identifikation	Rechte
erlaube	mitarbeiter	lesen
verbiete	Schmidt	*
erlaube	*	ausführen
erlaube	Franke	lesen, schreiben, ausführen

In dem Sinne der ACL's verwaltet Unix eine start vereinfachte ACL, die jeweils nur drei Identifikationen kennt: den Eigentümer des Objekts, die Benutzer in der Gruppe des Eigentümers des Objekts und alle anderen Benutzer.

Kapitel 5

Speicher-Organisation und -Verwaltung

5.1 Anforderungen und Eigenschaften von Speichern

In allen Rechnern werden heute digitale Speicher eingesetzt, die abhängig von einem externen Signal zwei verschiedene stabile Zustände annehmen können. Zur Auswahl eines Speicherelements ist ein Identifikator, die Adresse, erforderlich. I.a. wird aber nicht jedem Speicherelement eine Adresse zugeordnet, sondern einer Gruppe von Speicherelementen. Die kleinste adressierbare Einheit eines Speichers nennt man Speicherzelle. Bei PC's ist das z.B. immer eine Gruppe von 8 Bit oder einem Byte. Man spricht daher auch von byteorganisierten Speichern. Mehrere Speicherzellen oder Bytes können logisch zu einem Wort oder Doppelwort zusammengefasst und in wortadressierten Speichern gemeinsam adressiert werden.

Unter Speicherorganisation soll der physikalische und architektonische Aufbau von Speichern verstanden werden. Die Speicherverwaltung bestimmt gemeinsam mit der Speicherorganisation die Eigenschaften des Speichers. Die Speicherverwaltung ist durch die Kombination von spezieller Speicher-Hardware und Software des Betriebssystems realisiert. Zu ihren Aufgaben gehören:

Bestimmung des Speicherortes: Die Bestimmung des Speicherortes soll für den Benutzer transparent und datenspezifisch erfolgen. Häufig verwendete Daten werden automatisch in schnellen Speichern gehalten; selten verwendete Daten können auf langsamere Medien ausgelagert werden. Logische Programmadressen werden für den Benutzer transparent auf physikalische Speicherplatzadressen abgebildet.

Daten- und Programmschutz: Daten und Programme müssen vor Lesen, Verfälschung und Zerstörung durch unberechtigte Prozesse geschützt werden.

Datensicherheit: Ausfälle von Teilen oder ganzen Speichern müssen erkannt und möglichst automatisch korrigiert werden.

Aufgaben von Speichern sind die Datenhaltung und die Bereitstellung von Daten für die Weiterverarbeitung. Die Anforderungen an Speicher und auch die Eigenschaften von Speichermedien sind aber sehr unterschiedlich. Folgende Anforderungsparameter sind für die Architektur eines Speichers entscheidend:

Dauer der Datenhaltung: Sie kann kurzfristig, z.B. zur Speicherung von Zwischenergebnissen bis langfristig z.B. zur Archivierung von Daten sein.

Zeit zur Datenbereitstellung: Sie ist oft entscheidend für die Gesamtleistung des Rechners.

Eigenschaften der Daten: Oft bestimmen die Eigenschaften der Daten, wie Art und Anzahl, ein Speichermedium oder eine bestimmte Art der Speicherung.

Einige Eigenschaften von Speichern sind:

Flüchtige oder permanente Speicherung

Änderbarkeit des Speicherinhalts: Es kann sich um reine Lesespeicher (read only) oder um Schreib-und Lesespeicher (read/write) handeln. Die Anzahl der möglichen Schreib- oder Lesevorgänge kann dabei begrenzt sein.

Art der Adressierung: Die Adressierung kann orts- oder informationsbezogen erfolgen.

Art des Speicherzugriffs: Es kann beliebig oder wahlfrei auf einen Speicher zugegriffen werden, z.B. beim Arbeitsspeicher (RAM: Random Access Memory) oder sequentiell, z.B. bei Bandlaufwerken. Mischformen sind auch üblich, z.B. indexsequentiell, wobei zuerst aus einer Liste eine ungefähre Position der Daten entnommen wird, ab der dann sequentiell weitergesucht wird. Typisches Beispiel für indexsequentiellen Zugriff sind Disketten und Plattenlaufwerke, bei denen über eine Spurnummer die Positionierung des Schreib-/Lesekopfes erfolgt und dann die Information sequentiell aufgesucht werden.

Größe der adressierbaren Dateneinheiten: Hiermit hängt unmittelbar die Datenübertragungsrate zusammen.

Die wichtigsten Leistungskenndaten von Speichern sind:

Kapazität: Anzahl speicherbarer Daten.

Zugriffszeit: Die Zugriffszeit ist die Zeit von der Anforderung bis zur Bereitstellung der Daten.

Zykluszeit: Die Zeit nach der ein Lese- oder Schreibvorgang im Speicher wiederholt werden kann.

Datenübertragungsrate: Die Datenübertragungsrate gibt die Anzahl übertragener Daten z.B. zwischen Speicher und Prozessor pro Zeiteinheit an.

Integrationsdichte und Energieverbrauch: Mit dem Energieverbrauch hängt häufig die Verlustwärme und damit die mögliche Integrationsdichte der Speicher zusammen.

Preis pro Datenmenge

5.2 Speicherhierarchie und virtueller Speicher

Die folgende Tabelle gibt größenordnungsmäßige Leistungskenndaten und relative Kosten von Speichern an:

Speichereinheit	Zugriffszeit	Übertragungsrate	Relative Kosten
Register	$10^{-9}s$		10^7
Cache (SRAM)	$10^{-8}s$	$500\, MByte/s$	10^6
Arbeitsspeicher (DRAM)	$10^{-7}s$	$250\, MByte/s$	10^5
Sekundärspeicher	$10^{-2}s$	$10\, MByte/s$	10^3
Archivspeicher	$10^2 s$	$5\, MByte/s$	10^0

Moderne superskalare Prozessoren sind in der Lage durch Pipelining pro Takt mehrere Operationen gleichzeitig auszuführen. Das bedeutet, dass für einen Prozessor mit einer Taktrate von 500 MHz alle 2 ns mehr als eine Operation mit Daten versorgt werden muss.

Hier wird eine Problematik deutlich: Die Zugriffszeit der meisten Speicher beträgt ein Vielfaches der Ausführungezeit eines Maschnienbefehls eines modernen Prozessors. Mit langsameren und billigeren Speichern kann der Prozessor also nicht schnell genug mit Daten und Befehlen versorgt werden. Der Speicher bremst den Prozessor aus. Auf der anderen Seite ist es finanziell nicht vertretbar, ausschließlich Speicher mit kurzen Zugriffszeiten und angemessen hoher Kapazität zu verwenden.

Heutige Rechner besitzen daher sowohl Speicher mit geringerer Zugriffszeit und Kapazität als auch höherer Zugriffszeit und Kapazität. Ziel ist es hierbei, den Prozessor möglichst mit den notwendigen Daten zu versorgen. Diese verschiedenartigen Speicher werden als eine homogene Einheit verwaltet; die Technik dazu nennt man einen virtuellen Speicher.

Beim virtuellen Speicher werden Daten aus langsameren Speichermedien für die Benutzer transparent auf schnellere Speichermedien abgebildet. Grundlage dieses Prinzips ist die Lokalität des adressierten Speicherbereichs beim Ablauf von Programmen. In empirischen Untersuchungen wurde festgestellt, dass 90 % aller Operationen eines Programms mit 10 % der Daten und Befehle des gesamten Programmadressraums stattfinden. Also reicht ein kleinerer schneller Speicherbereich, in dem die aktuell vom Prozessor benötigten Daten und Befehle gehalten werden, um den Prozessor schnell genug mit Daten zu versorgen.

Für wiederholte Zugriffe auf Daten oder Befehle hat man auf verschiedenen Ebenen der Speicherhierarchie aus Registern, Cache, Arbeitsspeicher und Sekundärspeicher das Prinzip der Pufferung entwickelt. Die aktuell benutzten Daten oder Befehle werden dabei in einem Puffer der nächst schnelleren Hierarchiestufe gehalten. Am Beispiel der PC's sieht das wie folgt aus:

- Auf dem Prozessor-Chip wird in einem First-Level-Cache von 8 bis 32 KByte, der gerade benutzte Teil des Arbeitsspeichers zwischengepuffert.

- Auf dem Mainboard befindet sich ein externer Second-Level-Cache von bis zu 1 MByte, der mit schnellen Speicherbausteinen ausgerüstet einen größeren Teil des Arbeitsspeichers zwischenpuffert.

- Im Arbeitsspeicher werden Speicherbereiche von der Festplatte oder einer CD-ROM zur schnelleren Verarbeitung gepuffert.
- Auf den Festplatten-Controllern werden die Daten beim Lese- bzw. Schreibvorgang in einem Geräte-Cache gepuffert.

5.3 Halbleiterspeicher

Für Arbeitsspeicher, Grafikkarten oder Festplatten-Cache werden heute Halbleiterspeicher mit wahlfreiem Zugriff (RAM: Random Access Memory) verwendet. Hierbei unterscheidet man zwischen dynamischem RAM (DRAM) und statischem RAM (SRAM).

Dynamische RAM's lassen sich preiswerter als statische herstellen. Eine Speicherzelle wird dabei nur durch Ladungsspeicherung in einem Feldeffekttransistor realisiert. Wie ein Kondensator ist auch die DRAM Speicherzelle einem Entladevorgang unterworfen. Daher muss die Ladung in bestimmten Zeitabständen erneuert werden. Man nennt dieses auch Refresh. Meist wird das Refresh-Signal unabhängig von der Prozessor durch einen DMA-Controller realisiert. Allerdings sind während der Refresh-Zeit keine Speicherzugriffe möglich.

Video RAM's (VRAM) sind spezielle dynamische RAM's, auf denen man auch während der Refresh-Zeit lesen und schreiben kann (Dual Ported RAM).

Statische RAM's (SRAM) werden z.B. für Register und Cache verwendet. Sie bestehen aus mehreren Transistoren, die meistens in bipolarer Technik hergestellt werden. SRAM's sind wesentlich teuerer als DRAM's.

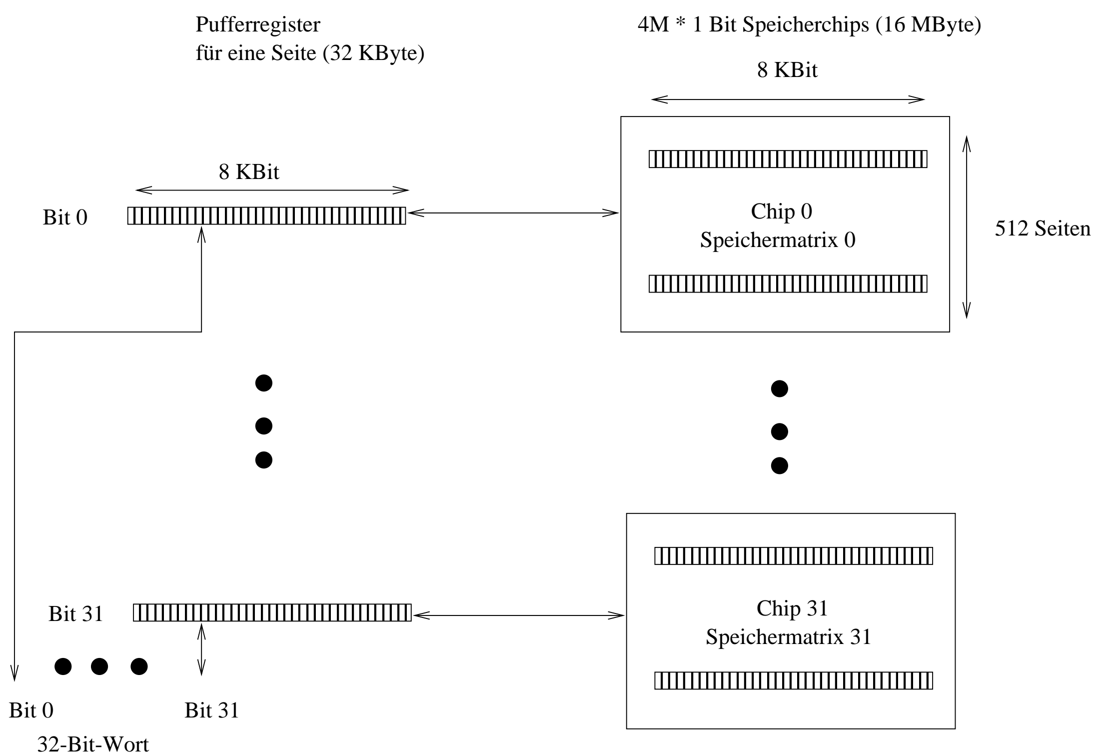
ROM's (Read Only Memory) werden über Masken vom Hersteller fest mit Speicherinhalt versehen. Zunehmend setzen sich allerdings programmierbare ROM's durch, obwohl sie teurer sind. Man unterscheidet hierbei PROM's und EPROM's (Erasable PROM). Die Programmierung eines PROM erfolgt durch anlegen von hohen Spannungen, die bestimmte Leiterbahnen des PROM unwiderruflich zerstören. Bei einem EPROM-Baustein können die Speicher hingegen durch ultraviolette Stahlung gelöscht werden.

5.4 Arbeitsspeicher-Organisation

Die zwischen Arbeitsspeicher und Prozessor transferierten Dateneinheiten werden als Speicherworte bezeichnet. Worte sind typischerweise 8, 16, 32 oder 64 Bit lang. Ein wortbreiter Speicher ist aus mehreren Speicherchips aufgebaut. Dabei kann jedes Chip genau 1 Bit oder auch mehrere Bit eines Wortes speichern. Bei der 1-Bit-Organisation werden also genauso viele Chips verwendet, wie die Wortbreite in Bit angibt. Bei 4 MBit-Chips können also 4 M Worte, d.h. bei einer Wortlänge von 32 Bit genau 16 MByte gespeichert werden. Die Kapazität in Worten ist also gleich der Kapazität eines Chips in Bit. Für kleinere Kapazitäten kann man pro Chip auch mehrere Bits speichern und so bei gleicher Wortlänge weniger Chips einsetzen. Sind die 4 MBit-Chips so organisiert, dass 4 Bit pro Chip gleichzeitig ausgelesen werden können, so werden statt 4 M nur 1 M Worte gespeichert werden.

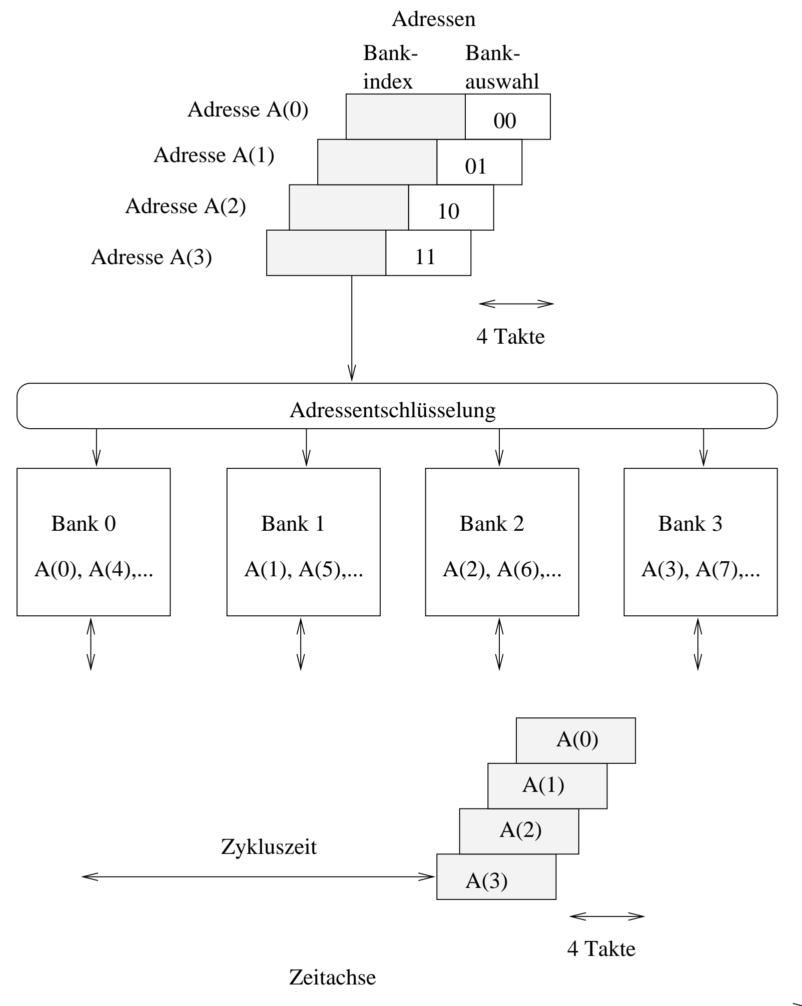
Die Speicherchips sind beim Arbeitsspeicher (DRAM) in Form von Matrizen aufgebaut, wobei die Matrixzeilen mit gleichem Zeilenindex aus allen Chips des Speichers eine sogenannte Seite bilden. Der Arbeitsspeicher verfügt über

Pufferregister, in der eine Seite des Speichers zwischengespeichert werden kann. Wird ein Speicherwort über eine bestimmte Spalte adressiert, dass sich bereits in diesen Pufferregistern befindet, so muss nicht auf den Speicher zugegriffen werden. Erst wenn sich ein Wort auf einer anderen Seite des Arbeitsspeichers, d.h. Zeile der Chips, befindet, müssen veränderte Pufferregister in den Speicher zurückgeschrieben und neu aus dem Arbeitsspeicher gelesen werden. Dies nennt man einen Seitenwechsel. Ein Zugriff mit Seitenwechsel ist i.d.R. 4 bis 5 mal langsamer als ein Zugriff ohne Seitenwechsel. Bei einem 200 MHz Prozessor mit einer Taktdauer von 5 ns benötigt man z.B. bei einer Zykluszeit mit Seitenwechsel von 120 ns für einen Speicherzugriff 24 Takte. Dies ist viel zu lange, weshalb man architektonische Maßnahmen zur Verbesserung der Situation ergreift.



Eine architektonische Maßnahme zur Beschleunigung der Speicherzugriffe auf DRAM's ist die sogenannte Speicherverschränkung. Hierbei wird der Arbeitsspeicher in mehrere getrennt adressierbare wortbreite Speicherbänke aufgeteilt. Daten, wie z.B. Vektoren, werden auf diese verschiedenen Speicherbänke verteilt. Hierzu ein Beispiel: Ist die Zykluszeit des Arbeitsspeichers 4 mal so lang wie die Taktzeit des Prozessors, so wird ein Speicher mit 4 Bänken, d.h. ein 4-fach-verschränkter Speicher verwendet. Die Adressierung der Bänke erfolgt z.B. anhand der letzten beiden Bit der Speicherplatzadresse mit Hilfe einer einfachen Entschlüsselungslogik. Zum Lesen eines Vektors der zyklisch auf die Speicherbänke 0 bis 3 verteilt ist, kann nun pro Takt eine Adresse an den Arbeitsspeicher angelegt werden. Einen Speicherzyklus oder 4 Takte nach Anlegen der ersten Adresse wird nun pro Takt und nicht pro Zyklus ein Wort aus dem Speicher

bereitgestellt. Wir haben es also wieder mit dem Prinzip der Fließbandverarbeitung oder des Pipelining zu tun: Zwar wird die Zykluszeit des Speichers nicht beschleunigt, aber es können mehr Daten pro Zeit von und zum Speicher übertragen werden. Allerdings muss auf jeden Fall mindestens eine Zykluszeit auf das erste Speicherwort eines Vektors gewartet werden. Je länger der gespeicherte Vektor ist, desto weniger fällt diese Startup-Zeit aber relativ gesehen ins Gewicht.



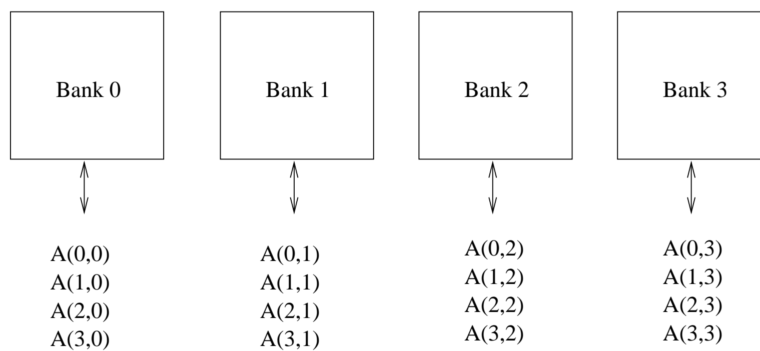
Bei der Speicherverschränkung ist die Verteilung der Daten auf die Speicherbänke allerdings entscheidend. Wenn aufeinander folgende Daten in derselben Speicherbank gehalten werden, muss jedes mal pessimal eine volle Zykluszeit gewartet werden, bis der nächste Datenwert zur Verfügung gestellt werden kann. Wird z.B. eine Matrix auf einen verschränkten Speicher abgebildet, so sollten

die Matrixelemente so verteilt werden, dass alle im Programm benötigten Zugriffsmuster, wie Zeilen, Spalten, Diagonalen usw. konfliktfrei, parallel aus den Bänken gelesen werden können.

Hierzu ein Beispiel: Wird die Matrix wie im oberen Bild spaltenweise auf die Speicherbänke verteilt, so kann auf Zeilen, Diagonalen und Gegendiagonalen parallel zugegriffen werden. Beim spaltenweisen Zugriff auf die Matrix entsteht allerdings ein Speicherbankkonflikt, da aus einer Bank nicht mehrere Speicherworte gleichzeitig bearbeitet werden können. Solch ein Speicherbankkonflikt erzwingt also einen sequentiellen Zugriff auf die betroffene Speicherbank.

Es werden also Abbildungsvorschriften gesucht, bei denen möglichst viele oder bestimmte Zugriffsmuster parallel bearbeitet werden können. Hierzu werden sogenannte Skewing Schemata oder schiefe Organisationen von Matrizen verwendet.

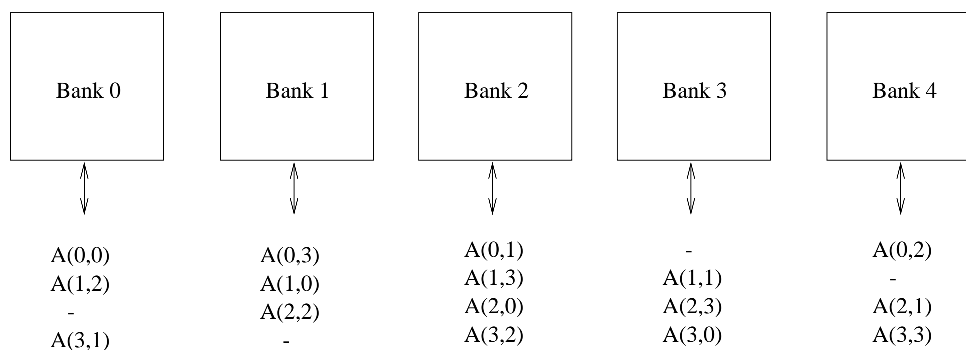
Zugriffskonflikt beim Spaltenzugriff



Kein Zugriffskonflikt beim Spaltenzugriff (parallele Arbeitsweise)

Skewing-Schema (schiefe Organisation) Bank(A(i, j)) = (d1*i + d2*j) mod m

Hier: Bank(A(i, j)) = (i + 2j) mod 5



Nachfolgend wird ein Satz für den konfliktfreien Zugriff auf Matrizen mit Hilfe von Skewing-Schemata hergeleitet. Dazu zunächst eine Definition:

Gegeben sei ein verschränkter Speicher mit m Bänken. Der n-elementige Vektor $a = (a_0, a_1, ..., a_{n-1})$ heißt d-geordneter n-Vektor modulo m, wenn das Element a_k des Vektors auf die Speicherbank $\mu(a_k) = d * k \, mod \, m$ abgebildet wird.

Für zweidimensionale Matrizen wird das Schema als $\mu(A(i,j)) = (d_1 * i + d_2 * j) \, mod \, m$ dargestellt. Eine Spalte ist also ein d_1 -geordneter Vektor, eine Zeile ein d_2 -geordneter Vektor, eine Diagonale ein $d_1 + d_2$ -geordneter Vektor und eine Gegendiagonale ein $d_2 - d_1$ - geordneter Vektor.

Der Satz über den konfliktfreien Zugriff lautet nun: Auf einen d-geordneten n-Vektor modulo m kann dann und nur dann konfliktfrei zugegriffen werden, wenn $\frac{m}{ggT(d,m)} > n$ ist.

Der Beweis dieses Satzes verläuft wie folgt: Ein Zugriffskonflikt zwischen dem k-ten und dem k+i-ten Element entsteht genau dann, wenn i multipliziert mit der Abgriffsdistanz d ein Vielfaches der Bankzahl m ist. Der erste Zugriffskonflikt entsteht also für die kleinsten Faktoren i und j für die $i * d = j * m$ ist, also für die Faktoren des kleinsten gemeinsamen Vielfachen von d und m : $kgV(d,m)$. Da $kgV(d,m) = \frac{d*m}{ggT(d,m)}$ gilt, bestimmt sich der Mindestabstand i zwischen den Indizes zweier Konfliktelemente zu $\frac{m}{ggT(d,m)}$. Ist der Mindestabstand größer als die Anzahl n der Vektorelemente, so herrscht Konfliktfreiheit: $\frac{m}{ggT(d,m)} > n$.

Hierzu nun ein Beispiel: Im Skewing-Schema der Matrix des vorigen Bildes ist die Vektorlänge $n = 4$, die Anzahl der Bänke $m = 5$ und die Abgriffsdistanzen $d_1 = 1$ und $d_2 = 2$. Da alle Zugriffsdistanzen $d = d_1$, $d = d_2$, $d = d_1 + d_2$, $d = d_2 - d_1$ Primzahlen kleiner als die Primzahl $m = 5$ sind, ist $ggT(d,5) = 1$und somit herrscht für $m > n$Konfliktfreiheit.

Damit $ggT(d,m)$ möglichst gleich 1 ist, wählt man also für die Anzahl der Bänke meist Primzahlen.

5.5 Arbeitsspeicher-Verwaltung

Bisher wurden Aspekte der Arbeitsspeicher-Organisation besprochen. Um den Arbeitsspeicher in einer Mehrprozess- bzw. Mehrbenutzerumgebung effektiv einsetzen zu können, müssen Verwaltungsverfahren verwendet werden, die folgende Mindestanforderungen erfüllen:

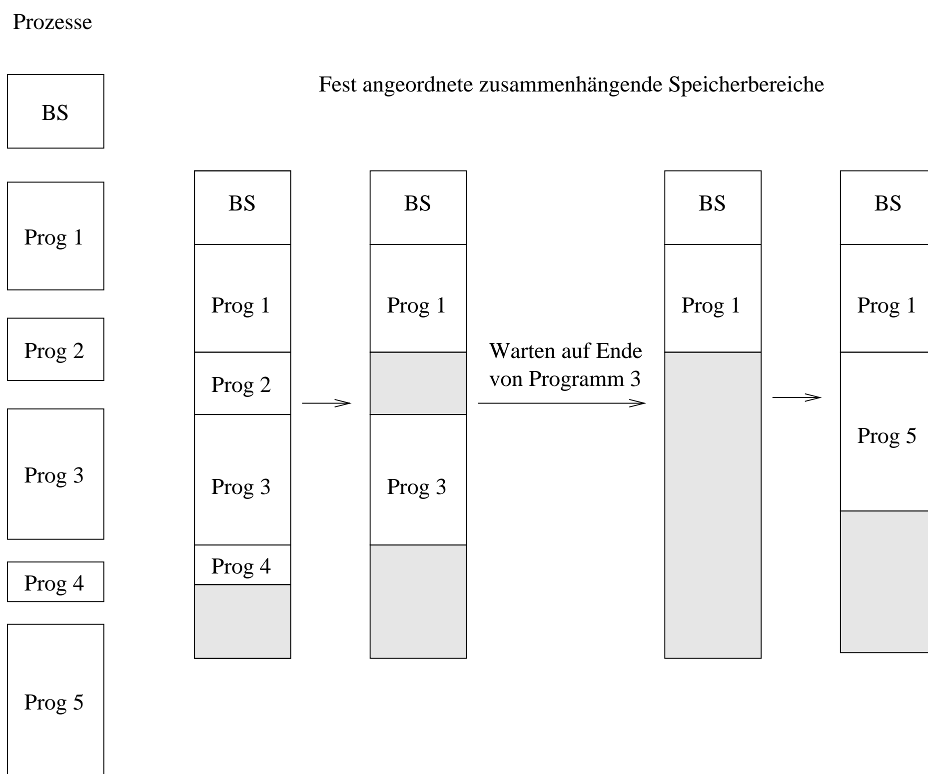
- Beim Mehrprozess-Betrieb müssen Programmteile, die häufig aktiviert werden, im Arbeitsspeicher resident gehalten werden.
- Die Adressbereiche der Benutzer- und Betriebssystem-Prozesse müssen untereinander gegen Adressverletzungen geschützt werden.
- Da die Größe des Arbeitsspeichers zur residenten Haltung aller Programm- und Daten-Anteile aller aktivierten Prozesse meist nicht ausreicht, muss der Sekundärspeicher für die Benutzer transparent mit einbezogen werden.

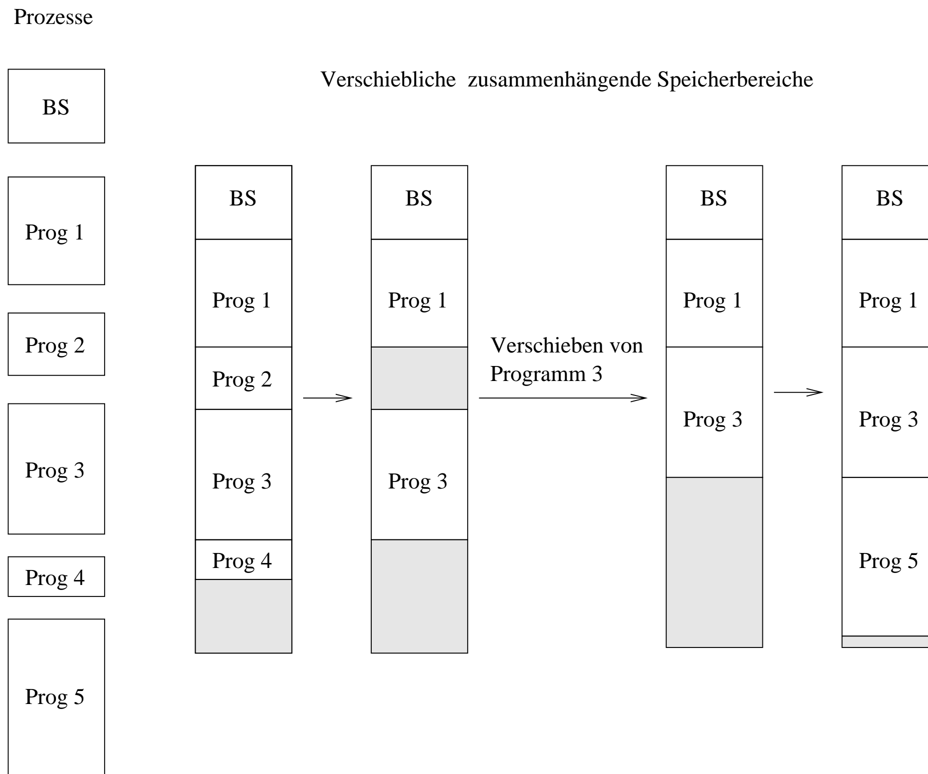
Einfache Arbeitsspeicher-Verwaltungen werden in den folgenden Bildern dargestellt:

Bei fest zusammenhängenden Adressbereichen entsteht nach dem Ende der Programme 2 und 4 ein Speicherverschnitt-Problem: Da der freie Speicherbereich nicht zusammenhängend sondern zerstückelt ist, kann Programm 5 nicht

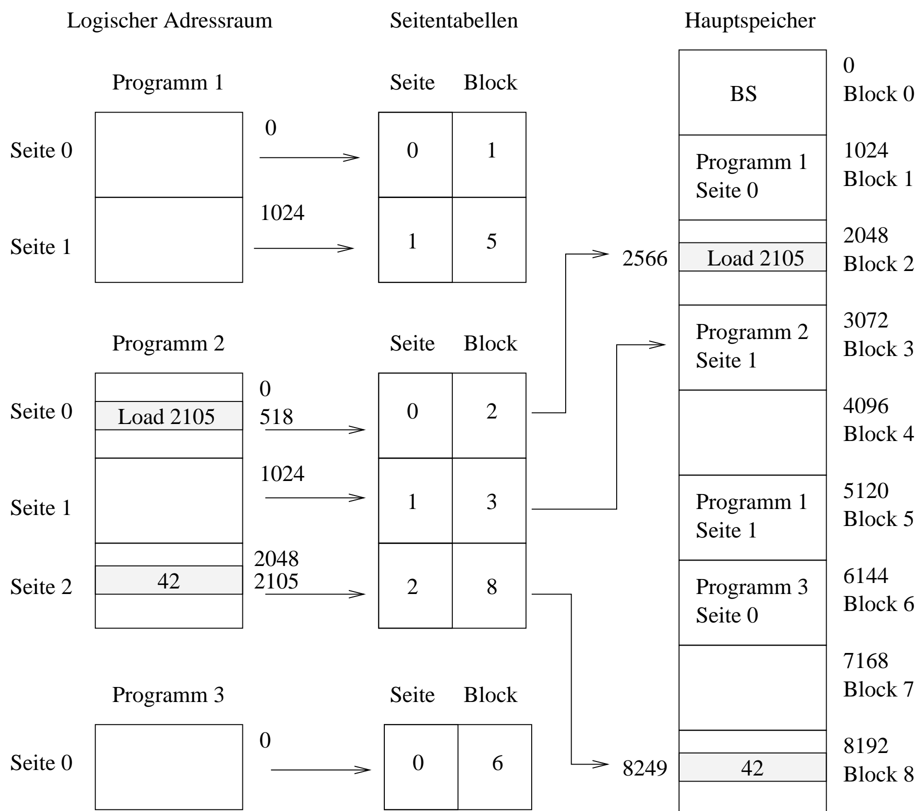
geladen werden, obwohl der Speicherplatz für dieses Programm ausreichen würde.

Bei der zusammenhängenden Speicherbereichs-Verwaltung mit der Möglichkeit der Verschiebung von Speicherbereichen kann nach dem Ende von Programm 2 und 4 durch Verschiebung von Programm 3 das Programm 5 nachgeladen werden. Anstelle des Speicherverschnitts tritt hier jedoch der Aufwand für die Verschiebung von Speicherbereichen. Technisch kann diese Verschiebe-Verwaltung durch ein Verschiebe-Register und eine Feldlängen-Register pro Prozess realisiert werden. Das Verschiebe-Register enthält die Anfangsadresse des Prozesses im Arbeitsspeicher. Der Inhalt des Verschiebe-Registers wird vor jedem Speicherzugriff auf die logische Speicherplatzadresse des Programms addiert. Das Feldlängen-Register enthält die Länge des Speicherbereichs des Prozesses. Es wird zur Überwachung der Grenzen des Adressraums eines Prozesses verwendet.

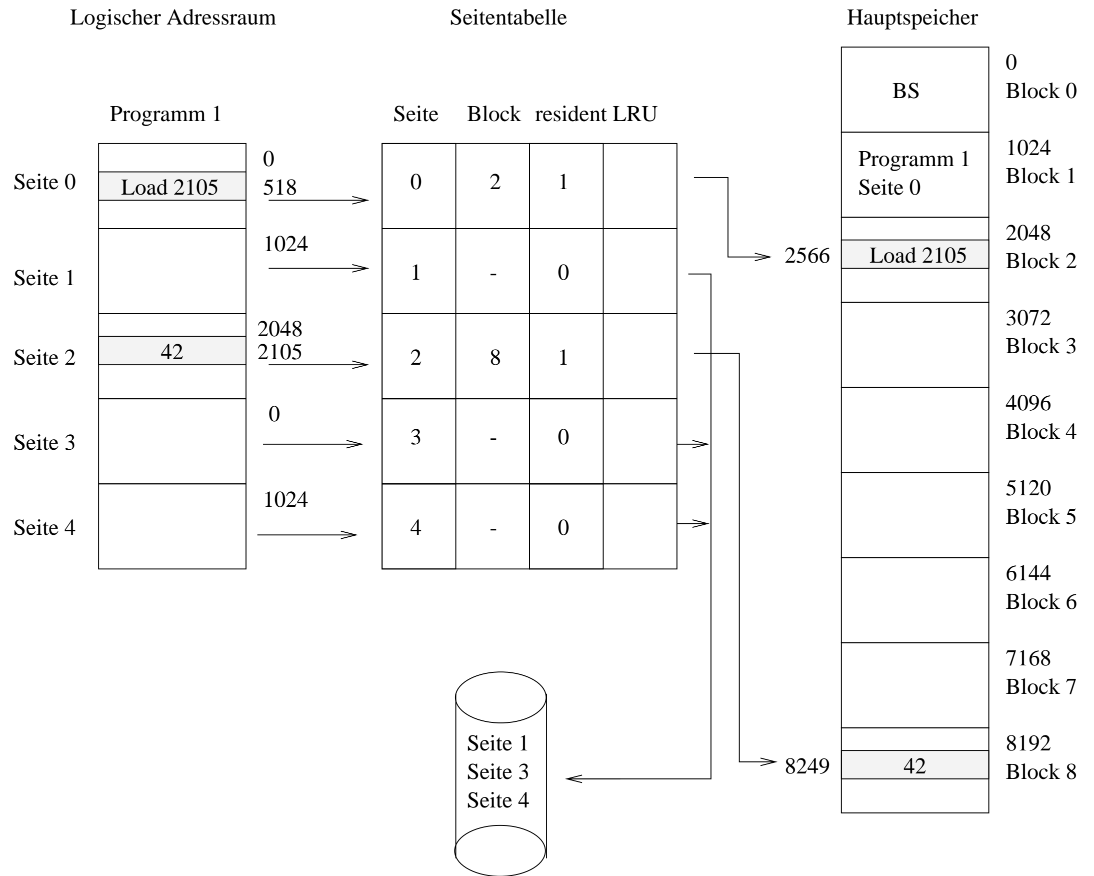




Ein Verfahren, das bei reduziertem Speicherverschnitt ohne Verschiebungen von Speicherbereichen auskommt, ist das Seitenwechsel- oder Paging-Verfahren. Hierbei wird jeder Programmadressraum in mehrere Seiten fester Größe aufgeteilt, die in Blöcke des Arbeitsspeichers abgebildet werden können. Eine Seitentabelle beschreibt diese Abbildung. Ein geringer Speicherverschnitt kann innerhalb der letzten Seite des Programms auftreten. Er beträgt im Mittel die Hälfte der Seitengröße. Bei kleinen Programmen oder zu hoher Seitengröße ist der Speicherverschnitt beim Paging höher als bei großen Programmen. Die Seitengröße muss also zum Speicherplatz-Profil der Programme passen.



Beim Paging-Verfahren müssen alle Seiten eines ausgeführten Prozesses im Arbeitsspeicher resident sein. Es müssen aber nicht alle Seiten eines Programms gleichzeitig im Arbeitsspeicher geladen sein, sondern nur die Seiten, die innerhalb eines kurzen Zeitintervalls benötigt werden. Beim Demand-Paging oder Seitenwechsel auf Abruf befinden sich nur Teile des Programms im Arbeitsspeicher. Seltener benutzte Seiten eines Programms werden bei Bedarf (on demand) vom Sekundärspeicher in den Arbeitsspeicher nachgeladen. Das Demand-Paging realisiert das Konzept des virtuellen Speichers, denn die Größe aller Prozesse kann die Größe des Arbeitsspeichers übertreffen. Allerdings müssen Seiten aus dem Arbeitsspeicher zum Sekundärspeicher hin verdrängt werden, wenn eine neue Seite geladen werden soll und keine freien Blöcke mehr vorhanden sind. Zur Entscheidung welche Seite verdrängt wird, werden unterschiedliche Strategien benutzt. Häufig wird die Least-Recently-Used-Strategie (LRU) angewendet, bei der die Seite verdrängt wird, auf die am längsten nicht zugegriffen wurde.



Beim Segmentierungs-Verfahren wird der Arbeitsspeicher als eine Sammlung von getrennten Bereichen, den Segmenten, verwaltet. Segmente können Hauptprogramme, Unterprogramme (insbesondere Bibliotheksroutinen) und Daten enthalten. Jedem Segment ist ein Name und eine feste Segmentlänge zugeordnet. Die Adressierung der Speicherzellen erfolgt zweidimensional über den Segmentnamen, der in eine Segmentnummer umgesetzt wird, und einen Offset innerhalb des Segments. Ein Programm verfügt also i.d.R. über mehrere eindimensionale Adressräume oder Segmente.

Die Segmentierung hat gegenüber den bisherigen Verfahren folgende Vorteile:

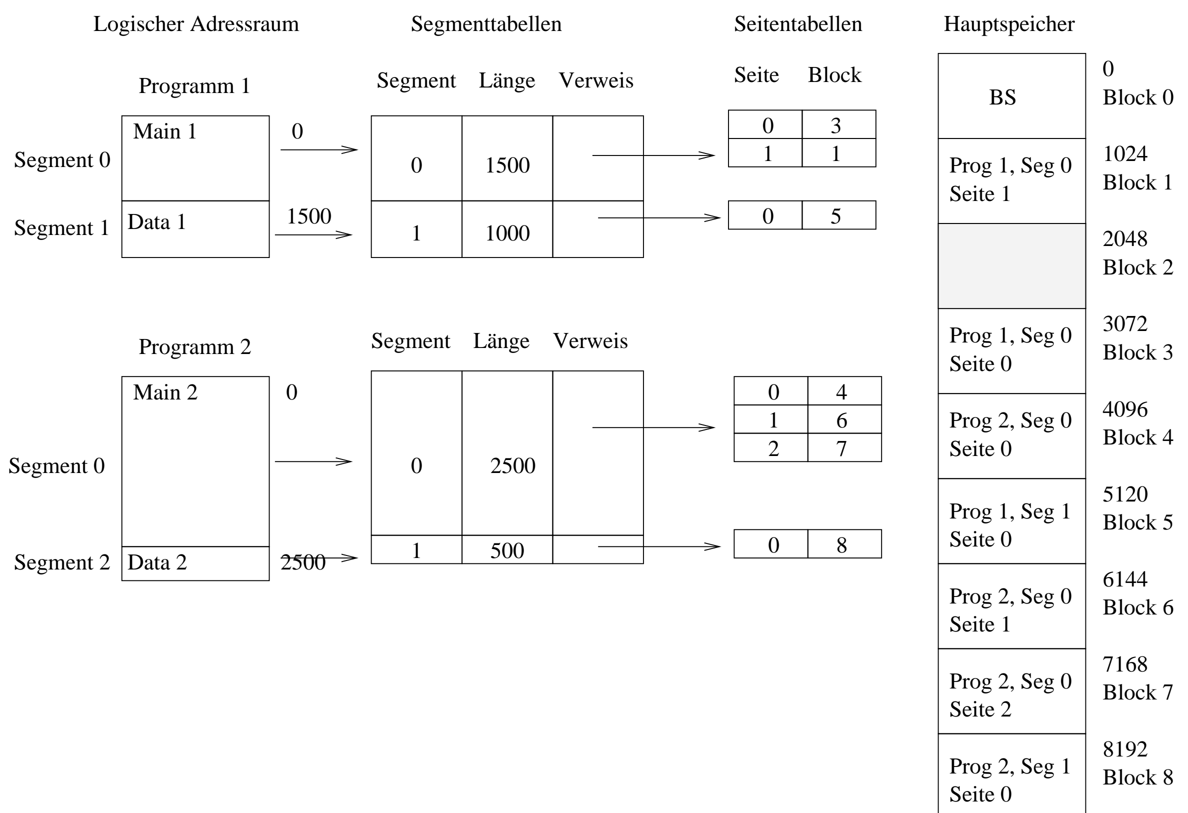
- Durch den zweidimensionalen Adressraum entfällt das Binden von Haupt- und Unterprogrammen vor der Programmausführung (dynamisches Binden).

- Reentrant Unterprogramme oder Daten, die von verschiedenen Programmen verwendet werden, können in einem einzigen Segment gespeichert werden. Dies spart Speicherplatz.

- Segmente können vor unberechtigtem Lesen, Schreiben oder Ausführen mit entsprechenden Attributen geschützt werden.

- Bereichsüberschreitungen außerhalb des Segments können durch Vergleich des Offset mit der Segmentlänge erkannt werden.

Damit der Speicherverschnitt gering bleibt, wird die Segmentierung meist mit Paging oder Demand-Paging kombiniert. Im Bild enthält die Segmenttabelle die Kenndaten der Segmente eines Programms, z.B. die Segmentnummer, die Segmentlänge und einen Speicherplatz-Verweis auf eine Seitentabelle pro Segment. Den Seiten werden über Blockadressen physikalische Blöcke im Arbeitsspeicher zugeordnet.



Bei der Adresstransformation von der logischen zur physikalischen Adresse sind durch den zusätzlichen Zugriff auf die Segment- und Seitentabellen, die ebenfalls im Arbeitsspeicher liegen, pro Adressierung maximal drei Speicherzyklen notwendig. Daher liegt es nahe, die Adresstransformation durch einen schnellen Speicher (Cache) für die Segment- und Seitentabellen zu beschleunigen. Allerdings ist die Anzahl der möglichen Tabelleneinträge hoch: Bei 32-Bit-Adressen und 4 KByte Seitengröße (2^{12}) liegt sie bei 2^{20} Tabelleneinträgen also im Bereich von einigen MByte Cache. Da sich aber auch diese Tabelleneinträge wie die Programme adressenlokal verhalten genügen auch wesentlich kleinere Cache-Speicher (10 % der potentiellen Größe) für die Segment- und Seitentabellen.

Eine weitere Aufgabe der Arbeitsspeicher-Verwaltung ist der Schutz von Daten und Programmen vor Zugriffen (Lesen, Schreiben) durch unberechtigte Prozesse. Dazu werden die Adressbereiche der Prozesse voneinander getrennt verwaltet und die Prozesse mit Hilfe von Zugriffsrechten kontrolliert.

Hierzu wird häufig die Segmentierung angewendet: Die Adressräume von Programmcode und Daten verschiedener Benutzern werden in unterschiedlichen Segmenten getrennt untergebracht. Der Prozessor bearbeitet Benutzerprozesse und Betriebssystemprozesse in zwei unterschiedlichen Zuständen: dem User-Mode und dem Supervisor-Mode. Tatsächlich gibt es Maschinenbefehle, die den Prozessor vom User-Mode in den Supervisor-Mode überführen. Im Supervisor-Mode wird dann ein Betriebssystemprozess ausgeführt. Will ein Prozess beispielsweise neuen Speicher anfordern, so geht das nur, indem er einen Betriebssystemprozess aufruft. Während das Betriebssystem alle Zugriffsrechte auf die Segmente hat, darf ein Benutzerprozess nur innerhalb seiner Segmente adressieren.

5.6 Cache-Organisation und -Verwaltung

Da beim Cache Organisation und Verwaltung untrennbar miteinander verbunden sind, werden die Verwaltungsverfahren der Caches zusammen mit ihren Organisationsformen besprochen.

Ziel eines Cache-Speichers ist es, die mittlere Zugriffszeit auf den Gesamtspeicher (Cache, Arbeitsspeicher, Sekundärspeicher) soweit zu reduzieren, dass sie möglichst nahe an der Operationszeit des Prozessors liegt. Um dies zu erreichen, muss die Zugriffszeit auf den Gesamtspeicher in der Nähe der Cache-Zugriffszeit sein, d.h. möglichst viele Zugriffe müssen im Cache-Speicher stattfinden.

Bzgl. der Organisation unterscheidet man Cache-Formen die mehr oder weniger assoziative Suchen erlauben. Bei der assoziativen Suche werden Arbeitsspeicheradressen der Daten im Cache-Speicher gehalten und mit einer angelegten Adresse parallel verglichen, um auf einen Datenwert lesend oder schreibend zuzugreifen. Mit steigender Anzahl parallel vergleichbarer Adressen wächst natürlich der Hardware-Aufwand für die Adressen-Vergleichslogik des Cache.

Aufgabe der Cache-Verwaltung ist es, Maschinenbefehle und Daten für den Benutzer transparent aus dem Arbeitsspeicher in den Cache zu laden und wenn keine freien Plätze mehr im Cache vorhanden sind, Cache-Zellen wieder in den Arbeitsspeicher zurückzuschreiben, um die neuen Arbeitsspeicher-Zellen in den Cache laden zu können.

Für eine hohe Beschleunigung der Speicherzugriffe durch den Cache sollten dabei verschiedene Forderungen erfüllt werden.

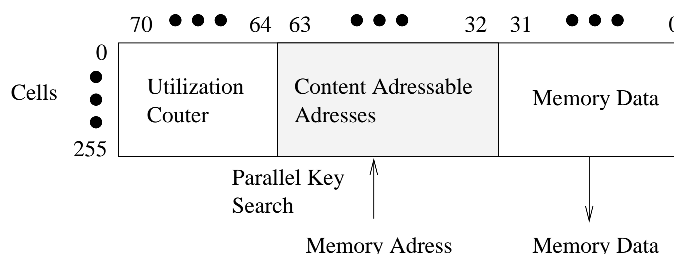
- Die Anzahl der Cache-Zugriffe muss maximiert werden. Dabei muss die Größe des Cache zu der Größe der Speicherzellen auf die häufig zugegriffen wird (Working Set der zu bearbeitenden Programme) passen.
- Da Prozesswechsel das Neuladen des Caches aus dem Arbeitsspeicher erfordern, sollten sie nicht zu häufig durchgeführt werden, d.h. die Anzahl der Prozesswechsel pro Zeiteinheit darf nicht groß sein.
- Wenn Zugriffe auf den Arbeitsspeicher notwendig sind, sollten sie möglichst mit anderen Operationen überlappt ausgeführt werden, damit der

Prozessor nicht untätig wird. Dies kann durch eine intelligente Anordnung der Maschinenbefehle durch den Compiler erreicht werden. Auserdem können mehrere aufeinanderfolgende Arbeitsspeicherworte in einem geladen werden, da es wahrscheinlich ist, dass sie auch adressiert werden.

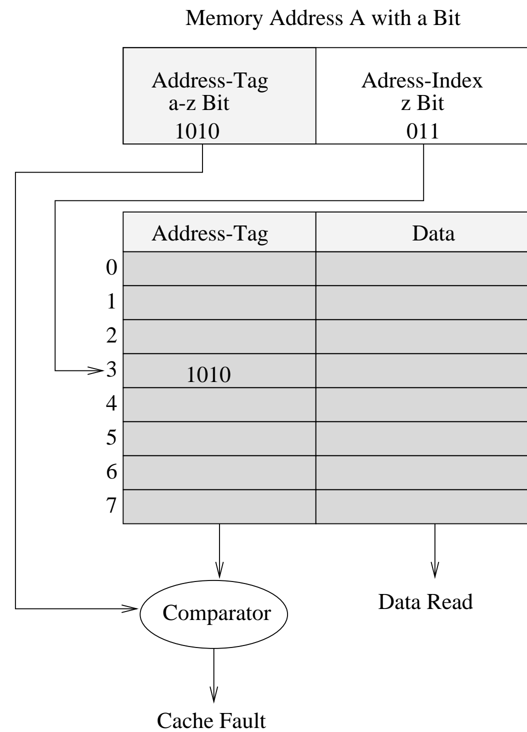
- Der Arbeitsspeicher-Durchsatz muss zum Cache-Durchsatz passen, d.h. die Bandbreite von Arbeitsspeicher, Bus und Cache müssen möglichst gleich sein. Für den Arbeitsspeicher werden dazu z.B. Zellen mit mehrfacher Cache-Zellen-Gröse und/oder ein verschränkter Arbeitsspeicher verwendet.

Der vollassoziative Cache hat maximalen Hardware-Aufwand, da für jede Cache-Zelle eine Vergleichslogik für die angelegte Arbeitsspeicheradresse vorhanden ist. Beim Nachladen des Cache aus dem Arbeitsspeicher entsteht bei voll belegtem Cache die Frage nach der Verdrängungsstrategie. D.h. welche Zelle soll aus dem Cache in den Arbeitsspeicher verdrängt und anschließend durch den Inhalt einer anderen Arbeitsspeicher-Zelle überschrieben werden. Eine häufig verwendete Strategie ist Least-Recently-Used, d.h. eine der am längsten nicht benutzten Zellen wird verdrängt.

Das LRU-Verfahren kann über einen Zugriffszähler (Utilization Counter) der Cache-Zellen realisiert werden. Bei jeder Referenzierung einer Zelle wird der Zugriffszähler auf einen festen Wert gesetzt, während die Zugriffszähler aller anderen Zellen minimal bis zur 0 dekrementiert werden. Eine der Zellen mit dem niedrigsten Wert des Zugriffszählers wird bei vollem Cache verdrängt. Wird der Wert des Zugriffszählers bei jedem Zugriff auf Z/2 gesetzt, wenn Z die Anzahl der Cache-Zellen ist, so sind zu jeder Zeit mindestens Z/2 Zugriffszähler 0. Die zugehörigen Zellen können verdrängt werden. Dies gilt, da das Maximum für alle Zugriffszähler erreicht wird, wenn jeder Zugriff auf eine andere Zelle erfolgt. Nach Z/2 Zugriffen sind die Zugriffszähler nicht betroffener Zellen 0 geworden.



Der Direct-Mapping-Cache hat den geringsten Hardwareaufwand. Dies wird durch eine feste Zuordnung von Arbeitsspeicher- und Cache-Adressen erreicht. Es gibt genau eine Vergleichslogik für alle Cache-Zellen. Der Direct-Mapping-Cache arbeitet wie folgt: Die Arbeitsspeicheradressen werden in zwei Teile aufgeteilt: die Adress-Kennung oder Adress-Tag und den Adress-Index. Mit Hilfe des Adress-Index wird eine Cache-Zelle adressiert. Die Adress-Kennung jeder geladenen Arbeitsspeicherzelle wird im Cache gespeichert. Die Arbeitsspeicherzelle liegt nun genau dann im Cache, wenn die Adress-Kennung der angelegten Arbeitspeicheradresse mit der Adress-Kennung im Cache übereinstimmt.



Wenn 2^z die Größe des Cache ist, wird der Arbeitsspeicher vom Cache aus gesehen in 2^{a-z} Seiten der Größe 2^z aufgeteilt. Eine Seite kann also vollständig in den Cache abgebildet werden. Allerdings können keine Arbeitspeicher-Inhalte aus verschiedenen Seiten mit gleichem Adress-Index im Cache gemeinsam gespeichert werden. In dieser Situation findet eine Verdrängung aus dem Cache in den Arbeitsspeicher statt. Die Verdrängungsstrategie ist also fest.

Zwischen dem Direct-Mapping-Cache und dem vollassoziativen Cache stellt der Set-Associative-Cache einen Kompromiss in Bezug auf den Hardware-Aufwand und die Flexibilität der Verdrängungsstrategie dar. Der Set-Associative-Cache besteht aus mehreren Blöcken mit je einer Vergleichslogik, wobei jeder einzelne Block wie ein Direct-Mapping-Cache arbeitet. Zur Abbildung einer Arbeitsspeicherzelle stehen nun 2^k verschiedene Blöcke zur Verfügung, so dass eine Verdrängung z.B. nach dem LRU-Verfahren aus dem Block mit der am längsten nicht referenzierten Cache-Zelle stattfinden kann. Ein Cache-Block ist üblicherweise eine Arbeitsspeicherseite lang, so dass maximal Zellen mit gleichem Index aus 2^k verschiedenen Seiten gehalten im Cache gleichzeitig gespeichert werden können.

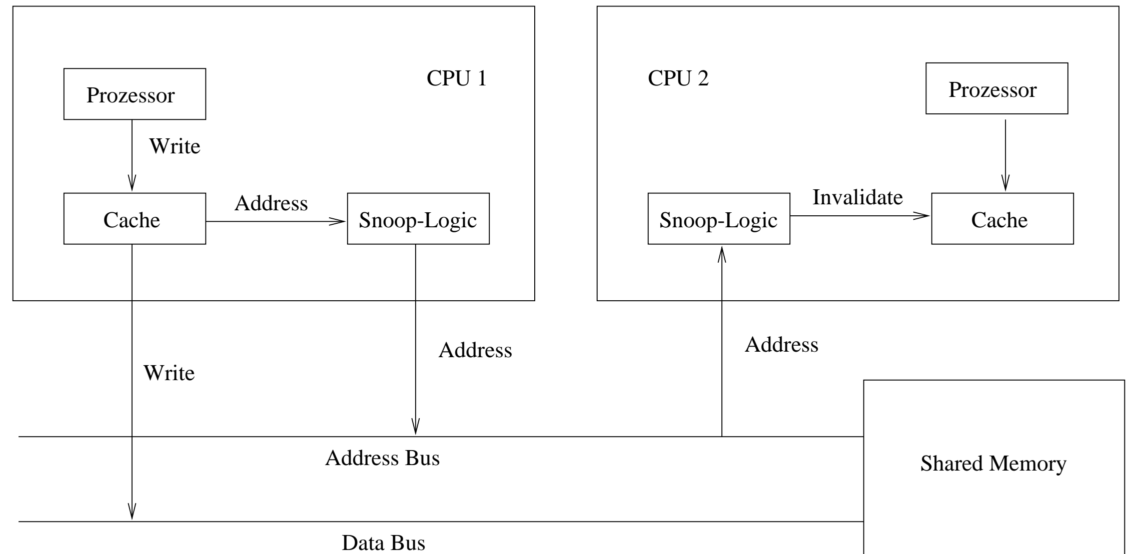
5.7 Cache-Konsistenz und -Kohärenz

Bei Systemen mit mehreren Prozessoren mit lokalem Cache-Speicher, die auf einen gemeinsamen Arbeitsspeicher zugreifen, können verschiedene Ausprägungen der Daten in den lokalen Caches und im Arbeitsspeicher bzgl. einer gemeinsamen Arbeitsspeicheradresse entstehen. Dies ist z.B. der Fall, wenn ein Prozessor einen Datenwert in seinem lokalen Cache ändert, den ein anderer Prozessor ebenfalls in seinem Cache gespeichert hat. Dann entstehen unterschiedliche Ausprägungen des Datenwertes im Arbeitsspeicher und in den lokalen Caches der Prozessoren. Damit ein Prozessor also nicht mit falschen, d.h. nicht mehr aktuellen, Datenwerten arbeitet, kann man fordern, dass keine unterschiedlichen Datenwerte in allen Caches und im Arbeitsspeicher bzgl. einer gemeinsamen Adresse existieren dürfen. Diese Forderung nennt man Konsistenz.

Das sogenannte Write-Through-Protokoll erfüllt die Konsistenzforderung: Hierbei wird bei einem Schreibzugriff unabhängig davon, ob der Datenwert im

Cache liegt oder nicht, der Arbeitsspeicher aktualisiert. Die Cache-Controller aller Prozessoren sind mit sogenannter Snoop-Logic (snoop heißt schnüffeln) ausgestattet, d.h. sie hören die Adressen auf dem gemeinsamen Adressbus zum Arbeitsspeicher ab. Bei jedem schreibenden Zugriff eines Prozessors auf den Arbeitsspeicher überprüfen alle anderen Prozessoren mit Hilfe ihrer Snoop-Logic, ob sie die Arbeitsspeicheradresse in ihrem Cache geladen haben. Ist dies der Fall, so wird die betreffende Cache-Zelle als ungültig markiert, d.h. sie muss beim nächsten Lesezugriff aus dem Arbeitsspeicher in den Cache transferiert werden.

Cache-Zustand	Aktiver Prozessor	Übrige Prozessoren
Lesefehlschlag	Wort in Cache holen	
Lesetreffer	Daten aus Cache nutzen	
Schreibfehlschlag	Arbeitsspeicher aktualisieren	
Schreibtreffer	Cache, Arbeitsspeicher aktualisieren	Cache invalidieren



Da jede Schreiboperation über den gemeinsamen Bus zum Arbeitsspeicher läuft, ist das Write-Through-Protokoll nur für wenige Prozessor's geeignet. Um den Datenverkehr zum gemeinsamen Arbeitsspeicher zu begrenzen, wurden Protokolle entwickelt, bei denen nicht alle Schreiboperationen auf den Arbeitsspeicher führen.

Ein solches Protokoll ist MESI, das z.B. ab dem Pentium II in der 80x86-Reihe von Intel eingesetzt wird. Zur Realisierung des MESI-Protokolls sind die Snoop-Logic's der Prozessor's mit zusätzlichen Steuersignalleitungen für Treffer in den Caches (HIT und HITModified) miteinander verbunden.

Bei MESI wird eine schwächere Bedingung als die Konsistenz nämlich die Kohärenz gefordert. Sie besagt, dass jeder Prozessor bei einem Lesevorgang den

aktuellen Wert im Cache erhält. Jeder Cache-Eintrag kann beim MESI-Protokoll einen von vier Zuständen einnehmen:

MESI-Zustände	veränderter Wert	unveränderter Wert
exklusiv in einem Cache	M: Exclusive Modified	E: Exclusive Unmodified
gemeinsam in mehreren Caches	I: Invalid	S: Shared Unmodified

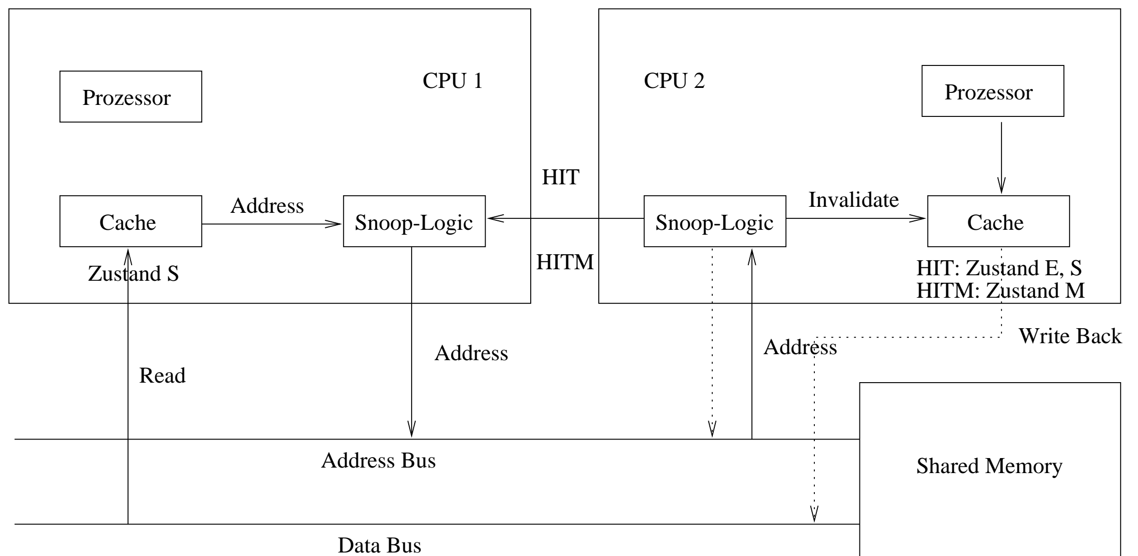
Beim Hochfahren des Rechners werden alle Cache-Einträge als ungültig gekennzeichnet. Im Betrieb können zwei Fälle unterschieden werden:

- In Prozessor 1 wird aus dem Arbeitsspeicher gelesen, d.h. die Cache-Zelle befindet sich im Zustand Invalid. Prozessor 1 sie legt eine Adresse auf den Adressbus. In diesem Fall können folgende Situationen entstehen:

  1. Die Snoop-Logic's aller anderen Prozessor's erkennen, dass sie diese Adresse in ihren Caches nicht geladen haben. Der Zustand der Cache Zelle wird Exclusive Unmodified gesetzt.

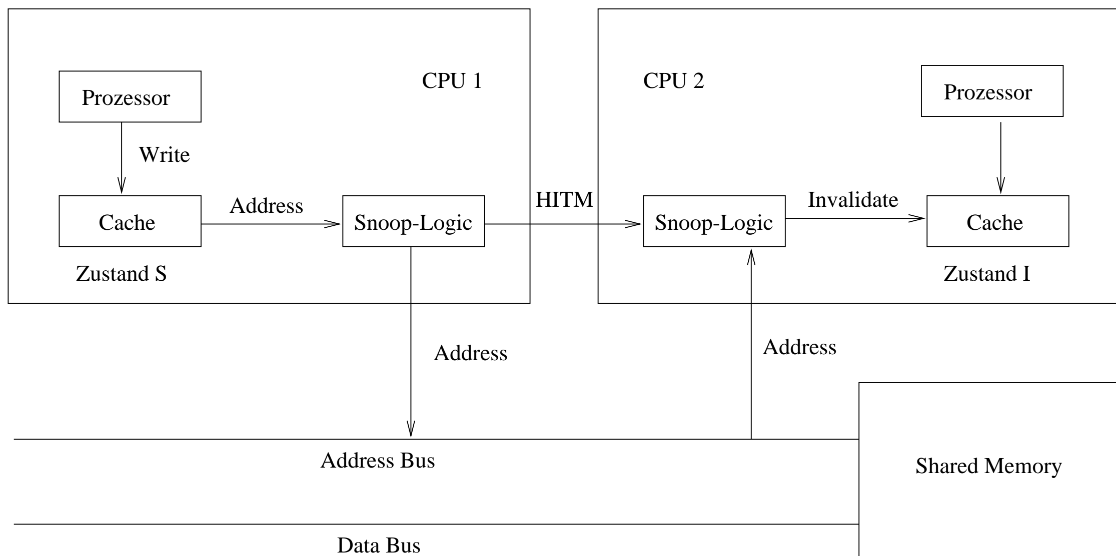
  2. Die Snoop-Logic mindestens einer anderen Prozessor, z.B. Prozessor 2, erkennt, dass sie die Adresse im Cache geladen hat, aber den Wert nicht verändert hat (Exclusive Unmodified oder Shared Unmodified). Prozessor 2 sendet das Signal HIT, legt die Adresse auf den Adressbus und überführt die Cache-Zelle in den Zustand Shared Unmodified. Prozessor 1 erkennt das HIT-Signal und setzt die Cache-Zelle zu dieser Adresse ebenfalls auf Shared Unmodified.

  3. Die Snoop-Logic genau einer anderen Prozessor, z.B. Prozessor 2, erkennt, dass sie die Adresse im Cache geladen und den Wert verändert hat (Exclusive Modified). Prozessor 2 startet das Rückschreiben in den Arbeitsspeicher und veranlasst Prozessor 1 zu warten, bis es vollendet ist. Prozessor 2 sendet das Signal HITModified, legt die Adresse auf den Adressbus und überführt die Cache-Zelle in den Zustand Shared Unmodified. Prozessor 1 liest den Wert aus dem Arbeitsspeicher und erkennt das HITModified-Signal und setzt die Cache-Zelle zu dieser Adresse ebenfalls auf Shared Unmodified



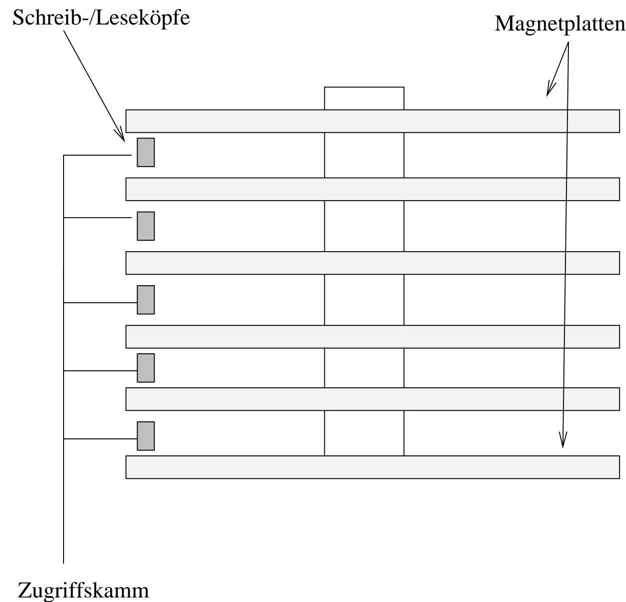
- In den Cache der Prozessor 1 wird geschrieben. In diesem Fall können folgende Situationen entstehen:

  1. Der Zustand der Cache-Zelle in Prozessor 1 ist Exclusive Unmodified: Der Wert wird in den Cache geschrieben und der Zustand in Exclusive Modified überführt.
  2. Der Zustand der Cache-Zelle in Prozessor 1 ist Exclusive Modified: Der Wert wird in den Cache geschrieben und der Zustand bleibt.
  3. Der Zustand der Cache-Zelle in Prozessor 1 ist Shared Unmodified: Prozessor 1 sendet HITModified, legt die Adresse auf den Adressbus und überführt die Cache-Zelle in den Zustand Exclusive Modified. Die Snoop-Logic's der anderen Prozessor's erkennen das HITModified-Signal und überführen geladene Cache-Zellen zur anliegenden Arbeitsspeicheradresse in den Zustand Invalid.
  4. Der Zustand der Cache-Zelle in Prozessor 1 ist Invalid. Hier gibt es zwei unterschiedliche Strategien:
     (a) Bei Write-Allocate wird die Cache-Zelle verändert und auf den Zustand Exclusive Modified gesetzt. Alle anderen Prozessor's werden über HITModified davon unterrichtet, dass sie ihre Cache-Zellen zur anliegenden Adresse auf Invalid setzen.
     (b) Ohne Write-Allocate wird der Arbeitsspeicher direkt beschrieben. In den Cache wird nichts geladen.



5.8 Magnetische Festplatten

Als Sekundärspeicher werden heute überwiegend Festplattenlaufwerke eingesetzt. In einem Festplattenlaufwerk liegen meistens mehrere Magnetscheiben übereinander, die genau wie in einer Diskette in konzentrische Spuren und Sektoren aufgeteilt sind. Für jede Magnetscheibe oder Platte stehen zwei Lese- und Schreibköpfe zur Verfügung, die auf die Ober- und Unterseite der Platte wirken. Alle Schreib-/Leseköpfe sind auf einem gemeinsamen Arm untergebracht, so dass der Spurwechsel für alle Platten gleichzeitig vollzogen wird. Die Zusammenfassung aller Spuren auf dem Plattenstapel mit gleicher Spurnummer bezeichnet man als Zylinder. Man verwendet zum Aufbau eines Festplattenlaufwerks verwindungssteifes Material, das eine hohe Drehzahl von ca. 3600 bis 5400 U/min und eine genaue Positionierung der Schreib-/Leseköpfe ermöglicht. Um die Abnutzung der magnetischen Trägerschicht und die Wärmeentwicklung durch Reibung der Schreib-/Leseköpfe zu minimieren, gleiten die Schreib-/Leseköpfe auf einem dünnen Luftpolster.



Als Standard für PC's haben sich sogenannte IDE-Platten durchgesetzt (Integrated Device Electronics). Wie der Name schon sagt, ist der Festplattencontroller bei IDE-Platten im Laufwerk mit untergebracht. Auf der Erweiterungskarte ist zum Anschluss einer IDE-Platte nur noch ein einfacher Adapter notwendig. Eine IDE-Platte ist also fast direkt am Systembus angeschlossen.

Der Enhaced-IDE-Standard erlaubt 255 Sektoren pro Spur und 65536 Zylinder. Dies bedeutet eine potentielle Kapazität von 127,5 GByte. Leider sind durch das BIOS eines PC's maximal 1024 Zylinder möglich, wodurch die Kapazität eingeschränkt wird. Die Datenbus-Bandbreite von Enhanced-IDE liegt bei 16,6 MByte/s. Diese Größen schaffen Spielraum für die Entwicklung der Festplatten der nächsten Jahre. Heutige Festplatten arbeiten meist mit einer Übertragungsrate von ca. 5 MByte/s und einer Kapazität von 1 bis 4 GByte.

SCSI-Platten (Small Computer Systems Interface) werden vor allen Dingen in Workstations aber auch zunehmend in PC's eingesetzt. Der SCSI-Standard definiert ein Bussystem, an das bis zu acht verschiedene Einheiten Festplatten, Bandlaufwerke, optische Platten aber auch Scanner, Drucker und Kommunikationsgeräte angeschlossen werden können. Da SCSI ein eigenes Bussystem benutzt, wird der Systembus gegenüber IDE weniger stark belastet. Dabei ist der SCSI-Bus mit dem Systembus durch einen Host-Adapter gekoppelt.

Weil nicht nur die Leitungsbelegung des SCSI-Bus sondern auch die Kommandos, die an Geräte gesendet werden, standardisiert sind, können SCSI-Geräte zwischen unterschiedlichen Systemen beliebig getauscht werden. Wenn ein Host-Adapter in einem Rechner eingebaut ist, lässt sich ein SCSI-Gerät einfach einfügen, indem eine ID-Nummer von 0 bis 6 am Gerät eingestellt wird. Die ID-Nummer 7 ist für den Host-Adapter reserviert. Ferner muss noch darauf geachtet werden, dass am physischen Ende des SCSI-Busses ein Abschlusswiderstand angebracht ist (meist wird er gesteckt).

Der Standard SCSI-II erlaubt in der Variante Fast-SCSI eine maximale Transferrate von 10 MByte/s bei einer Wortbreite von 8 Bit und einer Taktfre-

quenz von 10 MHz. Wide-SCSI wird überwiegend mit 16-Bit-Wortbreite seltener auch mit 32-Bit-Wortbreite betrieben. Wide-SCSI-Geräte können eine Übertragungsrate von 20 bis 40 MByte/s erreichen, wobei normale und Wide-SCSI-Geräte an ein und demselben SCSI-Bus betrieben werden können. Ultra SCSI verwendet zusätzlich zu Wide-SCSI eine Taktfrequenz von 20 MHz auf dem Bus, wodurch 40 bis 80 MByte/s erreicht werden können.

5.9 RAID-Technik

Der Abstand zwischen der Leistungsfähigkeit der Festplaten und der Prozessor wurde mit der Zeit immer größer. In den 70'er Jahren betrug die Suchzeit von Minicomputer-Platten etwa 50 bis 100 ms; heute beträgt sie immer noch 10 ms. Hingegen hat sich die Prozessor-Geschwindigkeit mit einer Verdopplung alle 18 Monate exponentiell gesteigert.

Zur Steigerung der Leistung von Festplatten und zur Verbesserung der Zuverlässigkeit kam man auf die Idee mehrere Festplatten parallel einzusetzen. Die Technik dazu nennt man RAID (Redundant Array of Inexpensive Discs). Konzept von RAID ist es, den Festplatten-Controller durch einen RAID-Controller zu ersetzen, der die Abbildung der Daten auf den Festplattenstapel übernimmt. RAID erscheint dem Betriebssystem also wie eine einzige Festplatte mit verbesserter Leistungsfähigkeit und Zuverlässigkeit.

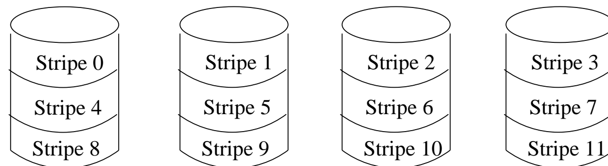
Da SCSI-Plattenlaufwerke sich durch hohe Leistung und niedrigen Preis auszeichnen und außerdem bis zu sieben Laufwerke an einem Controller möglich sind (bei Wide SCSI 15), bestehen die meisten RAID's aus einem SCSI-Controller und SCSI-Plattenlaufwerken.

Es gibt verschiedene Möglichkeiten der parallelen Ein- und Ausgabe von Daten in RAID's, die durch unterschiedliche RAID-Levels bezeichnet werden. Die Bezeichnung Level ist hier missverständlich, da es sich nicht um eine Hierarchie handelt.

RAID Level 0 legt Daten in aufeinanderfolgenden Streifen (Stripes), die aus mehreren Sektoren bestehen, auf mehrere Festplatten. Wenn das Betriebssystem den Befehl gibt einen Datenblock zu lesen oder zu schreiben, der aus mehreren Streifen besteht, teilt der RAID-Controller den Plattenlaufwerken separate Ein- oder Ausgabebefehle zu. Die Ein-/Ausgabe arbeitet parallel; Konflikte gibt es nur wenn auf Streifen gleicher Festplatten zugegriffen wird.

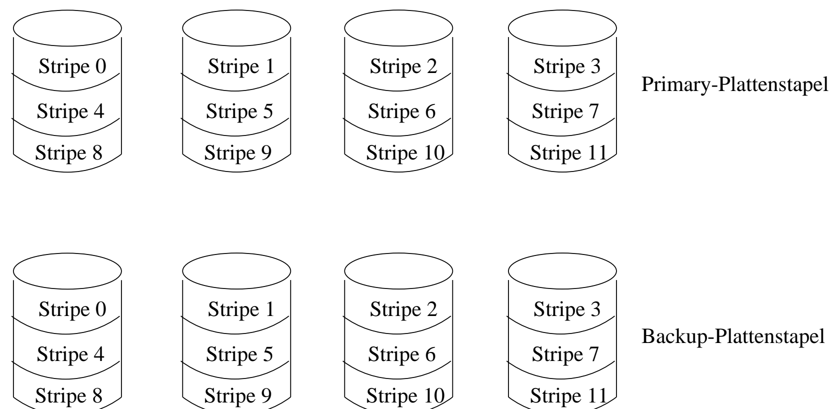
RAID Level 0 arbeitet schnell beim Zugriff auf große Datenblöcke, denn dann kommt die parallele Ein-/Ausgabe zur Wirkung. Dementsprechend langsamer ist die Ein-/Ausgabe bei Betriebssystemen, die standardmäßig nur auf einen einzigen Datensektor zugreifen. Ein weiterer Nachteil ist die geringere Zuverlässigkeit durch den Betrieb mehrerer Festplatten. Es wurde bei RAID Level 0 auf Redundanz zur Erkennung und Korrektur von Fehlern verzichtet, weshalb das R in RAID hier eigentlich irreführend ist.

RAID Level 0



RAID Level 1 arbeitet mit Redundanz, um die Zuverlässigkeit gegenüber RAID Level 0 zu erhöhen. Hier wird ebenfalls Striping angewendet; allerdings gibt es zwei Plattensätze: einen Primär-und einen Backup-Plattensatz. Bei einem Schreibvorgang wird jeder Streifen parallel zweimal geschrieben. Bei Lesevorgängen können beide Plattensätze benutzt werden, so dass die Leseleistung maximal doppelt so groß wie bei RAID Level 0 werden kann. Die Fehlertoleranz von RAID Level 1 ist sehr gut: Bei Versagen eines Primär-Laufwerks wird das zugehörige Backup-Laufwerk weiterbenutzt. Beim Recovery wird eine Kopie des Backup-Laufwerks erstellt. Die Redundanz von RAID Level 1 beträgt allerdings 50 %. Die Redundanz ist dabei wie folgt definiert: $R = \frac{n-i}{n}$, wobei n die Gesamtzahl der Bits und i die Anzahl informationstragender Bits ist.

RAID Level 1



Die Redundanz des RAID Level 2 ist beim Einsatz von mindestens 7 Plattenlaufwerken geringer als 50 %. Sie fällt mit der Anzahl der eingesetzten Plattenlaufwerke: Bei 7 Plattenlaufwerken beträgt die Redundanz ca. 43 %; bei 38 Plattenlaufwerken beträgt die Redundanz ca. 16 %.

Bei RAID Level 2 wird jedes Wort bitweise auf die Plattenlaufwerke verteilt. Dazu ein Beispiel: Jedes Byte kann in 2 Nibbles zu je 4 Bit aufgeteilt werden. Um einen Hamming-Code zu realisieren werden zu jedem Nibble 3 Paritätsbits hinzugefügt. Mit dem Hamming-Code können dann einfache Fehler korrigiert werden. Wenn also eins der 7 Festplattenlaufwerke ausfällt, kann das RAID-System trotzdem weiterbetrieben werden.

Bei den Hamming-Codes (Blockcodierung) wird eine Parität zur Überprüfung von Fehlern benutzt. Dabei wird ein Vektor $(x_{k+1}, ..., x_{k+p})$ aus mehreren

Paritätsbits durch Multiplikation eines Vektors der Informationsbits $(x_1, ..., x_k)$ mit einer Paritätsmatrix vor dem Schreiben im RAID-Controller berechnet:

$$(x_{k+1}, ..., x_{k+p}) = (x_1, ..., x_k) * \begin{pmatrix} g_{11} & ... & g_{1p} \\ ... & ... & ... \\ g_{k1} & ... & g_{kp} \end{pmatrix}$$

Nach dem Lesen eines Wortes werden im RAID-Controller alle Bits mit einer um eine $p * p$-Einheitsmatrix vergrößerten Paritätsmatrix multipliziert. Dabei werden die übertragenen Paritätsbits auf die errechneten addiert. Es ergibt sich ein Resultatvektor, bei dem bei fehlerfreier Übertragung sämtliche Stellen 0 sind. Wie man leicht sieht, ist der Resultatvektor bei einem Fehler an der Stelle x_i gleich den Werten der $i-$ten Zeile $(g_{i1}, ..., g_{ip})$ der Paritätsmatrix. Wenn alle Zeilen der Paritätsmatrix verschieden sind, kann ein solcher Fehler also korrigiert werden. Ist die Paritätsmatrix linear unabhängig, so können sogar alle Mehrfachfehler erkannt werden:

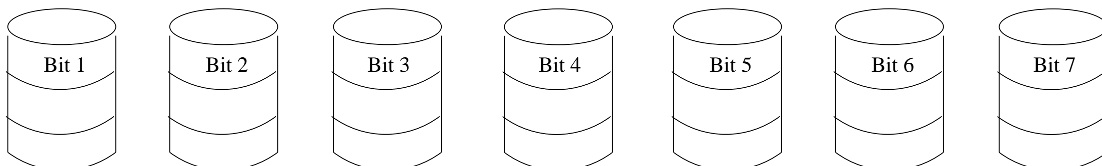
$$(x_1, ..., x_k, x_{k+1}, ..., x_{k+p}) * \begin{pmatrix} g_{11} & ... & g_{1p} \\ ... & ... & ... \\ g_{k1} & ... & g_{kp} \\ 1 & 0 & ... \\ 0 & 1 & \\ ... & ... & ... \\ 0 & ... & 1 \end{pmatrix} =$$

$$= \begin{cases} 0 \, kein \, Fehler \\ (g_{i1}, ..., g_{ip}) \, Fehler \, an \, Stelle \ \ i \\ a \neq 0 \, Fehler \end{cases}$$

Die minimale Anzahl Paritätsbits, um einfache Fehler korrigieren zu können, ergibt sich aus der Bedingung $k + p \leq 2^p$, da es 2^p verschiedene Paritätszeilen gibt. Für $k = 4$ ergibt sich $p = 3$ als niedrigste Anzahl Paritätsbits für die einfache Fehlerkorrektur.

Nach diesem Schema arbeitet z.B. der Parallelrechner Thinking-Machines CM-2. Hier werden zu 32-Bit-Datenworten 6 Paritätsbits und ein zusätzliches Bit zur Überprüfung der Wortparität hinzgefügt. Die Beschleunigung der parallelen Ein-/Ausgabe ist gewaltig und die Fehlertoleranz sehr gut. Mit ca. 19 % Redundanz hält sich auch der Mehraufwand für die Fehlertoleranz in Grenzen. Andererseits erfordert RAID Level 2 die Synchronisation aller Laufwerke und der RAID-Controller muss den Hamming-Code schnell genug bearbeiten können.

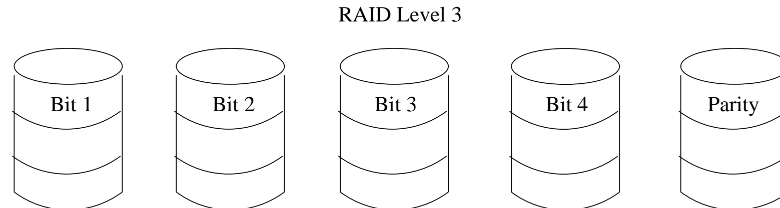
RAID Level 2



Der RAID Level 3 ist eine vereinfachte Version von Level 2. Dabei wird ein einzelnes Paritätsbit berechnet, so dass das exklusive Oder aller Bitstellen 0 (gerade Parität) oder 1 (ungerade Parität) ergibt. Zunächst erscheint es so, als ob

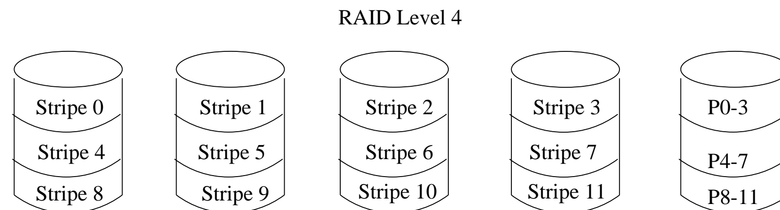
mit einem Paritätsbit nur 1-Bit-Fehler erkannt aber nicht korrigiert werden können. Da jedoch die Position des aufgefallenen Laufwerks dem RAID-Controller bekannt ist, kann ein erkannter Fehler natürlich auch korrigiert werden, d.h. bei einem Paritätsfehler war die Bitstelle 1.

RAID Level 3



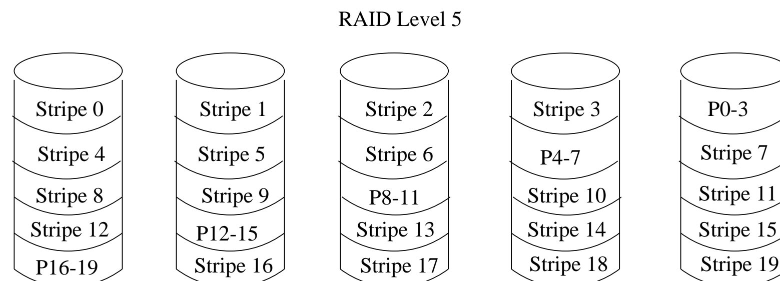
RAID Level 4 und 5 arbeiten wieder mit Streifen samt Parität und erfordern keine synchronisierten Laufwerke. RAID Level 4 entspricht RAID Level 0, wobei die Paritäts-Streifen wie bei RAID Level 3 auf einem zusätzlichen Laufwerk untergebracht sind. Dieses Verfahren bietet Schutz bei Defekt eines Plattenlaufwerks. Allerdings wird die Leistungsfähigkeit eines RAID Level 4-Systems bei vielen Lesezugriffen mit kleinen Datenmengen reduziert, da Streifen aus allen Laufwerke zur Überprüfung der Parität ausgelesen werden müssen und andere echte parallele Zugriffe während dieser Zeit nicht möglich sind. Auch Schreibzugriffe z.B. auf einen einzigen Sektor oder Streifen, erfordern zumindesten das Lesen und Zurückschreiben des neu berechneten Paritätsstreifens. Deshalb kann das Paritätslaufwerk bei RAID Level 4 zu einem echten Flaschenhals werden.

RAID Level 4



Um den Flaschenhals des Paritätslaufwerks zu vermeiden, verteilt RAID Level 5 die Paritätsstreifen gleichmäßig über die Plattenlaufwerke. Die Rekonstruktion der Daten bei einem Laufwerksfehler ist bei RAID Level 5 allerdings ein komplizierter Vorgang.

RAID Level 5



Kapitel 6

Ein-/Ausgabesysteme

6.1 Aufgaben von Ein-/Ausgabesystemen

Ein-/Ausgabesysteme dienen zur Kommunikation mit der Außenwelt. Dies können Benutzer an konkreten oder virtuellen Terminals oder Peripheriegeräte wie Festplatten, CD-Laufwerke, digitale Kameras usw. sein.

Ein-/Ausgabesysteme mussen für Programmierer und Anwender eine adäquate Abstraktion zur Verfügung stellen. Logische Ein-/Ausgabekanäle werde auf konkrete Geräte abgebildet. Programme werden hierdurch unabhängig von konkreten Geräten. Dasselbe Programm kann ohne Änderung beispielsweise auf eine Datei oder auf ein Terminal schreiben oder von einer Datei oder von einer Tastatur lesen. Es muss dabei nicht berücksichtigen welches Gerät sich hinter dem logischen Ein-/Ausgabekanal verbirgt.

In Unix-Systemen können Ein-/Ausgabekanäle verknüpft sein mit:

- Dateien die sich auf Festplatten aller Art befinden und durch unterschiedliche Dateisysteme verwaltet werden;
- physikalischen oder logischen Terminals (TTY oder PTTY)
- Kommunikationskanälen wie Pipes für die lokalen und Sockets für die Netzwerk-weiten Verbindungen zwischen Prozessen.

Geräte werden in Unix-Systemen in drei Klassen eingeteilt:

- Zeichenorientierte Geräte (Character Devices) können beliebige Zeichensequenzen übertragen, z.B. Terminals, Modems, Mäuse, Drucker und Scanner
- Blockorientierte Geräte (Block Devices) können nur Datenblöcke fester Größe übertragen, z.B. Festplatten, CD's und Bandgeräte.
- Sonstige Geräte wie z.B. Grafikkarten und Netzwerkkarten sind meist bitorientiert.

6.2 Treiber für Ein-/Ausgabegeräte

Ein-/Ausgabegeräte haben sehr unterschiedliche Arbeitseigenschaften, z.B. hinsichtlich der Übertragungsgeschwindigkeit und der Art, wie die Daten auf dem Ein-/Ausgabegerät verwaltet werden, z.B. zur Ein- und Ausgabe auf einem sequentiell, indexsequentiell oder wahlfrei organisierten Speicher, nur zur Ausgabe über eine Grafikkarte auf einem Bildschirm oder nur zur Eingabe von einer Maus.

Für Benutzer wäre es lästig, mit diesen Eigenschaften konfrontiert zu werden. Daher werden geeignete Betriebssystem-Dienste aufgerufen, die Ein-/Ausgabe-Vorgänge ausführen. Alle gerätespezifischen Programme, sogenannte Treiber, werden vom Betriebssystem verwaltet. Hierdurch wird das Benutzerprogramm unabhängig von den verwendeten Typen der Ein-/Ausgabegeräte. Die für die Arbeit mit den Treibern benötigten Initialisierungsschritte und Datentransfermechanismen werden durch das Betriebssystem vor den Benutzerprogrammen verborgen.

Ein Treiber stellt dem Betriebssystem eine geräteunabhängige Schnittstelle zur Verfügung, wobei hier von dem konkreten Gerät eine abstrakte Darstellung angeboten wird. Auf der anderen Seite arbeitet der Treiber mit dem Controller des Ein-/Ausgabegeräts zusammen, um die notwendigen Steuersignale für das spezielle Gerät zu produzieren. Ein Controller stellt den elektronischen Teil eines Ein-/Ausgabegeräts dar, das z.B. im Falle einer Festplatte den mechanischen Teil z.B. den Zugriffskamm ansteuert.

6.3 Einbindung von Gerätetreibern in das Betriebssystem

Da es sehr viele unterschiedliche Ein-/Ausgabegeräte auch eines Typs, z.B. von Grafikkarten, Festplatten oder Mäusen, gibt und durch Neuentwicklungen geben wird, ist es nicht möglich, ein Betriebssystem von vorne herein oder für die Zukunft mit allen notwendigen Treibern auszustatten. Treiber müssen also in ein fertiges Betriebssystem eingebunden werden. Das kann prinzipiell auf drei unterschiedliche Arten geschehen:

- Der Code des Treibers wird fest in den Betriebssystemcode eingebunden. Bei jedem neuen Peripheriegerät muss das Betriebssystem hier neu übersetzt und gebunden werden. Dazu muss der Quellcode des Betriebssystems vorliegen. Dies ist z.B. bei Linux möglich. Der Treiber wird Teil des Betriebssystemkerns. Die Arbeit mit dem Treiber ist daher schnell.

- Der Name des neuen Treibers und dessen Speicherort werden in eine Tabelle eingetragen und das Betriebssystem bindet den Treiber beim nächsten Neustart automatisch ein. Diese Vorgehensweise findet z.B. bei Windows-Betriebssystemen häufig statt.

- Der Code des Treibers wird dynamisch zur Laufzeit des Betriebssystems eingebaut, z.B. bei vielen USB-Geräten in Windows. Diese Technik nennt sich "Plug-&-Play". Sie funktioniert leider nicht immer reibungslos.

6.4 Ablauf einer Ein-/Ausgabe-Operation

Eine Ein-/Ausgabe-Operation läuft prinzipiell wie folgt ab:

1. Der anfordernde Benutzerprozess ruft mit Hilfe eines Systemaufrufs über den Betriebssystemkern den Gerätetreiber auf. Häufig über einen sogenannten Software-Interrupt s.u.

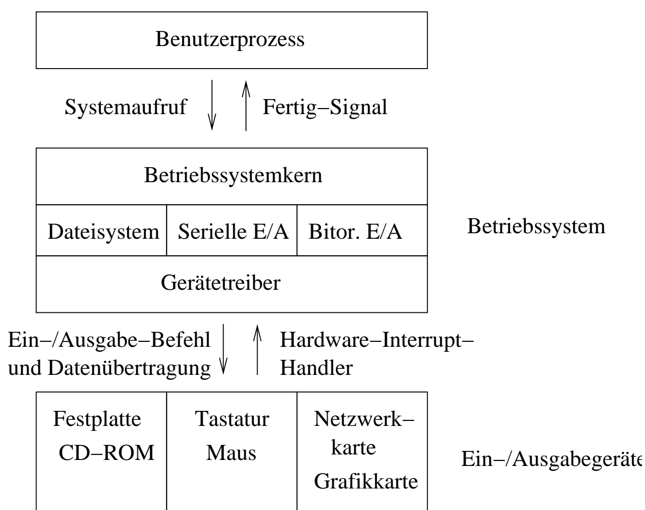
2. Der Betriebssystemkern, z.B. das Dateisystem, überprüft, ob die geforderte Operation für den Benutzerprozess erlaubt ist. Falls die Erlaubnis gegeben wird, geht der Benutzerprozess in den Zustand blockiert über.

3. Der Gerätetreiber startet die Ein-/Ausgabe-Operation, indem er Parameter und Steuerinformation an das Geräteprogramm überträgt. Das Geräteprogramm des Treibers führt die Ein-/Ausgabe-Operation durch.

4. Nach Abschluss der Ein-/Ausgabe-Operation erzeugt das Gerät ein Unterbrechungssignal, einen sogenannten Hardware-Interrupt. Dieser veranlasst den Prozessor einen sogenannten Interrupt-Handler auszuführen, d.h. ein Programm, das die Unterbrechung behandelt.

5. Der Interrupt-Handler identifiziert die Herkunft der Unterbrechung, d.h. welches Gerät die Unterbrechung ausgelöst hat und aktiviert den zugehörigen Gerätetreiber.

6. Der Gerätetreiber ermittelt den Prozess, der die Übertragung initiiert hat und erzeugt ein Signal mit dem der anfordernde Benutzerprozess vom Zustand blockiert wieder in den rechenbereiten Zustand überführt wird.



6.5 Fallbeispiel: Ein-/Ausgabe bei IBM-kompatiblen PC's

Interrupts bilden also den Mechanismus zur Kommunikation zwischen Programmen oder Hardware mit dem Betriebssystem. Durch einen Interrupt wird

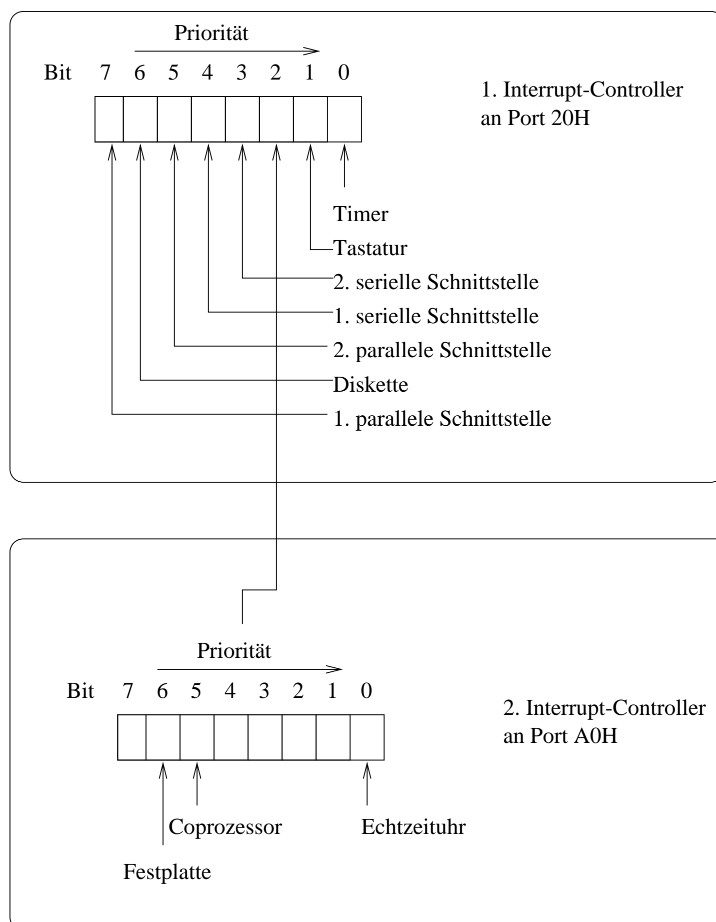
der Prozessor zur kurzzeitigen Unterbrechung des aktuell ausgeführten Prozesses und zur Ausführung einer Behandlungsroutine des Betriebssystems für den Interrupt, den Interrupt-Handler, veranlasst. Nach der Abarbeitung dieses Betriebssystem-Prozesses wird i.d.R. der unterbrochene Benutzerprozess fortgeführt. Allerdings kann während der Ausführung eines Interrupt asynchron ein weiterer Interrupt ausgelöst werden. In diesem Fall muss mit einer Priorisierung anhand der Dringlichkeit des neuen Interrupts entschieden werden, ob der gerade ausgeführte Interrupt weiter fortgesetzt oder ebenfalls unterbrochen wird.

Realisiert werden Interrupts mit Hilfe einer Tabelle, die die Anfangsadressen der Interrupt-Handler enthält. Diese Interrupt-Vektor-Tabelle ist z.B. bei IBM-kompatiblen PC's im Adressbereich 0000:0000 bis 0000:03FF (Segment : Offset) im Arbeitsspeicher niedergelegt. Jeder Eintrag des Interrupt-Vektors besteht aus 4 Byte, und zwar 2 Byte für das Segment und 2 Byte für den Offset der Anfangsadresse des jeweiligen Interrupt-Handlers. Beim Aufruf eines Interrupt wird nun eine Interrupt-Nummer übergeben, die als Index dieses Interrupt-Vektors interpretiert wird. Die CPU arbeitet dann den Interrupt, an der angegebenen Arbeitsspeicheradresse ab. Da die Interrupt-Vektor-Tabelle im Arbeitsspeicher steht, kann ihr Inhalt geändert werden. Das nutzen einige Dienstprogramme und Treiber, um sich im System zu installieren.

Adresse	Arbeits-speicher	Interrupt-Nummer	Belegung
0000:03FE	CS	255	Frei
0000:03FC	IP		
0000:000E	CS	3	Abbruch
0000:000C	IP		
0000:000A	CS	2	NMI
0000:0008	IP		
0000:0006	CS	1	Einzelschritt
0000:0004	IP		
0000:0002	CS	0	Division durch 0
0000:0000	IP		

Für die Bearbeitung eines Interrupts ist es wichtig, ob er von der Hardware oder von einem Programm stammt. Während Hardware-Interrupts völlig asynchron zur Programmausführung sind, werden Software-Interrupts z.B. bei PC's über den Maschinenbefehl INT synchron zum Programmlauf analog zu einem Unterprogramm aufgerufen. Ein weiterer Unterschied zwischen Hardware- und Software-Interrupts besteht darin, dass die Interrupt-Nummer beim Software-Interrupt von einem Programm und beim Hardware-Interrupt von einem Interrupt-Controller bereitgestellt wird.

Im PC kommt für die Kommunikation mit der Ein-/Ausgabe-Peripherie das Hardware-Interrupt-Verfahren zum Einsatz. Die Peripherie meldet über eine von 15 Steuerleitungen (8 Steuerleitungen beim XT), wenn Handlungsbedarf für die CPU besteht. Auf den Erweiterungskarten wird der Interrupt, ähnlich wie die Portadresse, eingestellt. Beim IBM XT besitzt der Port mit der Interrupt-Leitung 0 die höchste und der mit der Interrupt-Leitung 7 die niedrigste Priorität. Ab dem IBM AT können die Interrupt-Controller 15 Interrupt-Quellen verwalten. Dabei werden zwei Interrupt-Controller mit je 8 Eingängen verwendet. Wird eine Interrupt-Anforderung an den zweiten Controller gestellt, so simuliert dieser einen Interrupt an Leitung 2 (IRQ2) des ersten Controllers. Dadurch haben alle Interrupts des zweiten Controllers höhere Priorität als die Leitungen 3 bis 7 (IRQ3 -IRQ7) des ersten Controllers. Falls nun ein Interrupt vorliegt wird der CPU dies über eine Steuerleitung mitgeteilt. Danach liest die CPU über einen Ein-/Ausgabe-Befehl die Bitkette des ersten Controllers ein, in der die anfragende Interrupt-Leitung gespeichert ist. Daraus kann die CPU erkennen, welcher Interrupt-Handler zu laden ist. Wird der IRQ2 entdeckt, so kommt der Interrupt-Handler mit der Nummer 10 zur Ausführung. Dieser Handler liest das Register des zweiten Controllers ein und ermittelt hieraus den eigentlich auszuführenden Interrupt-Handler mit einer Nummer zwischen 0x70 und 0x77.



Wird ein Interrupt durch eine externe Ein-/Ausgabe-Hardware ausgelöst, so laufen i.w. folgende Schritte ab:

- Eine der Anforderungsleitungen des Interrupt-Controllers wird durch die externe Hardware auf einen hohen Pegel gesetzt.
- Der Interrupt-Controller leitet ein Interrupt-Signal an den Prozessor.
- Der Prozessor empfängt das Signal und reagiert mit einem Quittungssignal an den Interrupt-Controller, falls sie den Interrupt annimmt (maskierbare Interrupts).
- Der Prozessor veranlasst den Interrupt-Controller mit einem zweiten Quittungssignal einen 8 Bit Wert auf den Datenbus auszugeben, der die Nummer des aufzurufenden Interrupts ist.
- Der Prozessor liest die Anfangsadresse des Interrupt-Handlers aus der Interrupt-Vektortabelle und führt den Interrupt-Handler aus.
- Nach dem Ende des Interrupt-Handler-Laufs wird ein anderer niedriger priorisierter Interrupt oder ein anderes rechenbereites Anwendungsprogramm ausgeführt.

Vom Prozessor aus wird beim PC eine Erweiterungskarte oder ein anderes Ein-/Ausgabe-Modul, das sich auf dem Motherboard befindet, über eine Portadresse angesprochen. Die Portadressen werden beim PC nicht durch das System verteilt, sondern auf den Erweiterungskarten entweder mittels DIP-Schaltern oder über Software (Plug&Play) eingestellt. Die freie Vergabe der 16 Bit langen Portadressen bei peripheren Geräten führt zu häufigen Adresskonflikten. Deshalb haben Standardschnittstellen im Industrie-PC-Standard feste Portadressen, wie für den IBM AT in der Tabelle angegeben.

Baustein	Portadressen
DMA-Controller	0x000-0x01F
Interrupt-Controller	0x020-0x03F
Zeitgeber	0x040-0x05F
Tastatur	0x060-0x06F
Echtzeituhr	0x070-0x07F
DMA-Seitenregister	0x080-0x09F
Interrupt-Controller 2	0x0A0-0x0BF
DMA-Controller 2	0x0C0-0x0DF
Coprozessor	0x0F0-0x0F1
Coprozessor	0x0F8-0x0FF
Festplatten-Controller	0x1F0-0x1F8
Joystick	0x200-0x207
2. Paralleldrucker	0x278-0x27F
2. serieller Anschluss	0x2F8-0x2FF
Prototypkarte	0x300-0x31F
Netzwerkkarte	0x360-0x36F
1. Paralleldrucker	0x378-0x37F
Monochrom-Grafikkarte	0x3B0-0x3BF
Farbgrafikkarte	0x3D0-0x3DF
Disketten-Controller	0x3F0-0x3F7
1. serieller Anschluss	0x3F8-0x3FF

Intel-Prozessoren verfügen über die Befehle IN und OUT, um Ein-/Ausgabe-Operationen durchzuführen. Diese Befehle senden über den Steuerbus ein Signal, das alle Ports empfangsbereit macht, um die nachfolgende Adresse auszuwerten. Der Hauptspeicher wird in diesem Fall abgeschaltet. Anschließend erfolgt der Transport eines Bytes oder Wortes zum oder vom Port.

Literaturverzeichnis

[1] Tanenbaum, Andrew S.; Moderne Betriebssysteme; 2. Auflage; Pearson-Studium; 2002

[2] Stallings, William; Betriebssysteme - Prinzipien und Umsetzung; 4. Auflage; Pearson-Studium; 2002

[3] Schaffrath, Wilhelm; Grundkurs UNIX/Linux; Vieweg-Verlag; 2003

[4] Pils, Helmut; Das Linux-Tutorial - Ihr Weg zum LPI-Zertifikat; Vieweg-Verlag; 2004

[5] Kofler, Michael; Linux - Installation, Konfiguration, Anwendung; 7. Auflage; Addison Wesley; 2004

[6] Brause, R.; Betriebssysteme - Grundlagen und Konzepte; 3 Auflage; Springer; 2004

[7] Goll, Joachim; Bröckl, Ulrich; Dausmann, Manfred; C als erste Programmiersprache; 4. Auflage; Teubner; 2003