Einführung in die Systemprogrammierung unter UNIX

Fachhochschule für Technik Esslingen

Inhaltsverzeichnis

1 Das Betriebssystem eines Rechners

Die Software eines Computers kann grob gesprochen in 2 Klassen eingeteilt werden:

?? **Systemsoftware**
?? und **Anwendungs-Software**

Zur **Systemsoftware** werden in der Regel gezählt:

?? Betriebssystem
?? Netzwerk-SW
?? Datenbanken
?? Werkzeuge wie Compiler, Editoren etc.

Die Anwendungs-Software sind Programme, die die Aufgaben der Nutzer unterstützen, wie z.B. eine Buchhaltungs-Software für eine kaufmännische Abteilung.

Das **Betriebssystem** ist das wichtigste Systemprogramm. Es kontrolliert die Ressourcen des Computers und ist die Basis, auf der die anderen System-Programme und die Anwendungs-Programme aufsetzen.

Ein moderner Rechner besteht aus einem oder mehreren Prozessoren, dem Hauptspeicher, Massenspeichern, Netzwerkkomponenten, Datensichtstationen und anderen Ein-/Ausgabeeinheiten.

1.1 Grundsätzliche Aufgaben eines Betriebssystems

Ein Betriebssystem hat **zwei grundsätzliche Aufgaben**:

? es **verwaltet die Betriebsmittel des Rechners**,
? es hat den **Nutzer zu unterstützen**.

Dies sehen Sie am besten an dem folgenden Schichtenmodell.:

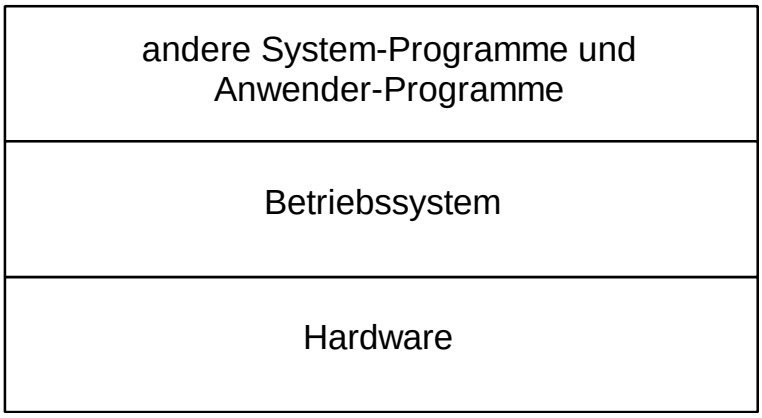


Bild 1.1 - 1 Schichtenstruktur eines Rechners. Zu den "anderen Systemprogrammen" gehören u.a. Compiler, Editoren.

Das Betriebssystem hat zum einen Schnittstellen zur Hardware, zum anderen Schnittstellen zu den Programmen des Anwenders. Beide Schnittstellen muß es „bedienen“.

In der Regel gibt es ganze Rechnerfamilien, deren Mitglieder einen unterschiedlichen HW-Aufbau haben, die aber dennoch unter ein und demselben Betriebssystem laufen. Bei Mikroprozessor-Betriebssystemen finden Sie auch Betriebssysteme, die auf Prozessoren verschiedener Hersteller z.B. Intel und Motorola laufen.

Das Betriebssystem verbirgt also die Hardware gegenüber den anderen System- und Anwendungsprogrammen des Nutzers. Es zeigt sich dem Benutzer gegenüber als **virtuelle Maschine**, die leichter zu programmieren ist als die zugrundeliegende Hardware.

Ziel eines Betriebssystems ist es damit,

?? die Ressourcen eines Rechners gerecht und fehlerfrei den Anwendern zur Verfügung zu stellen, d.h. Prozessoren, Speicher und I/O-Geräte den konkurrierenden Programmen gerecht und geordnet zuzuteilen.

?

?? und es einem Programmierer zu erlauben, mit vernünftigem Aufwand Programme schreiben zu können, ohne daß der Programmierer jeweils wissen müßte, wie die jeweilige Hardware funktioniert.

Das Betriebssystem stellt dabei die Verbindung zwischen den Anwendungs-Programmen und anderen System-Programmen - wie etwa Datenbanken - zur Hardware des Rechners her und stellt seine Leistungen diesen Programmen zur Verfügung. Das Betriebssystem stellt damit eine Software-Schicht dar, die über der Hardware liegt.

1.2 Verwalten der Betriebsmittel durch ein Betriebssystem

[Quelle: Vorlesungsmanuskript Wettstein: Stand und Trends von Betriebs-systemkonzepten,
Herrtwich, Hommel: Kooperation und Konkurrenz]

Für den Konstrukteur eines Betriebssystems ist zunächst eine Menge von technischen Apparaturen gegeben, die es zu betreiben gilt. Diese physikalischen Geräte faßt man im weiteren mit sich ergebenden logischen Objekten unter dem zentralen Begriff **Betriebsmittel** zusammen.

Ein **Betriebssystem** kann dann erklärt werden als

"eine Sammlung von Programmen zu geregelter Verwaltung und Benutzung von Betriebsmitteln verschiedener Art"

Dabei entstehen in Hinblick auf die Betriebsmittel zwei grundlegende Aufgabenbereiche eines Betriebssystems:

- **Zuordnung der Betriebsmittel zu den Interessenten**

  Da in der Regel mehr Interessenten als Betriebsmittel vorhanden sind, entsteht ein Wettbewerb. Dessen Regelung erfordert das Belegen und Freigeben der Betriebsmittel so, daß Information konsistent erhalten bleibt.

- Steuerung der Betriebsmittel

  Die Betriebsmittel sollen letztlich sicher gesteuert, gut ausgenutzt und die Kosten fair auf die Interessenten verteilt werden.

1.2.1 Prozesse als virtuelle Objekte zur Strukturierung nebenläufiger Programmsysteme

Bereits in der Vorlesung Datenverarbeitung 3 wurden die Betriebsmittel in **drei Kategorien**

- **Aktive Betriebsmittel, zeitlich aufteilbar**
- **Passive Betriebsmittel, exklusiv benutzt**
- **Passive Betriebsmittel, räumlich aufteilbar**

eingeteilt, und definiert, was man unter einem **entziehbaren (preemptive) Betriebsmittel** versteht:

Ein Betriebsmittel ist entziehbar (preemptive), wenn ein Prozeß es nach späterer Wiederzuweisung so verwenden kann, als hätte er es nie abgegeben.

Dabei war ein Prozeß definiert als "ein Programm in Ausführung oder als Programm, das laufen möchte", wobei bei dieser Definition vorausgesetzt ist, daß dieses Programm selbst nur sequentielle Anweisungen enthält. Ein Prozeß entsteht durch Ausführung von Anweisungen auf einem Prozessor (Prozeß = ablauffähige Programmeinheit oder 'schedulable Entity', die auf dem Prozessor laufen möchte oder dort läuft).

Ein Prozeß besteht aus

- einem Programmtext,
- den Programmdaten,
- evtl. dem Heap,
- dem Stack,
- seinem Programmzähler,
- dem Stackpointer

?? und anderen Registerinhalten
?? sowie weiteren Informationen, die zur Ausführung des Programms benötigt werden.

Dies wird später noch im Detail behandelt und ist nicht Sinn der jetzigen Betrachtung.

Hat man mehrere Prozessoren, so kann man mehrere Prozesse parallel bearbeiten. Hat man nur einen Prozessor, so können verschiedene Prozesse nur nacheinander auf dem Prozessor ablaufen.

Prozesse sind in erster Linie virtuelle Objekte zur Strukturierung von nebenläufigen Programmsystemen. Dabei reicht es aus, sich für jeden Prozeß einen **virtuellen Prozessor** zu denken. **Nebenläufig (concurrent)** bedeutet, daß Anweisungen unabhängig voneinander in nichtdeterministischer Weise ausgeführt werden können.

Sie können also

?? **parallel** von mehreren Prozessoren ausgeführt werden
?? oder in beliebiger Folge **sequentiell** von einem Prozessor **(quasi-parallel)** ausgeführt werden

Ferner wurden die Betriebsmittel und die Zustände eines Prozesses eingeführt und Zustandsübergänge von Prozessen besprochen. Auf diese Begriffe wird in dieser Vorlesung zurückgegriffen.

Wie in Datenverarbeitung 3 besprochen, wird üblicherweise für jeden Prozeß ein **Process Control Block (PCB)** eingerichtet. Alle PCBs der Prozesse im Zustand **ready-to-run** werden zu einer 'Bereit-Liste' verkettet. Wird der Prozessor frei, so erhält der Prozeß, der am Beginn der 'Bereit-Liste' steht, den Prozessor.

Sollte diese Liste leer sein, so kann der Prozessor keine sinnvolle Arbeit verrichten, solange kein blockierter Prozeß fortgesetzt oder ein neuer Prozeß gestartet wird. Dies wird oft dadurch vermieden, daß ein **Leerlauf-Prozeß (idle process)** existiert, der immer zugewiesen werden kann.

1.2.2 Zusammenstellung konkrete und virtuelle Betriebsmittel

Im folgenden werden die **konkreten Betriebsmittel** und die **virtuellen Betriebsmittel** einander gegenübergestellt:

Konkretes Betriebsmittel	**Virtuelles Betriebsmittel**	**Vorteil**
Prozessor	**Prozeß oder Task**	**Multiplexen der CPU**
Ein/Ausgabegeräte wie Tastatur, Bildschirm, Drucker, Externspeicher	**Datenströme** (Standard-Eingabe, Standard-Ausgabe)	**Einheitliche Behandlung aller Peripheriegeräte** (Ein/Ausgabegeräte) **als Datei**. Hardwareunabhängigkeit Flexibilität der Umlenkung auf anderes Peripheriegerät
Externspeicher	**Datei**	**Multiplexen des Externspeichers**
Arbeitsspeicher	**Adreßräume z.B. von Prozessen**	**Multiplexen des Arbeitsspeichers**

Die wesentlichen **Objekte**, die das Betriebssystem verwalten muß, sind:

?? Benutzer
?? Ein/Ausgabegeräte und die ihnen entsprechenden Datenströme
?? Externspeicher und (abstrahiert) Dateien
?? Prozessoren und als abstrakte Äquivalente Prozesse oder Tasks
?? Arbeitsspeicher und Adreßräume als die zugeordneten abstrakten Objekte

1.3 Erforderliche Betriebssystemkenntnisse für verschiedene Bedienerklassen

Zu unterscheiden sind die folgenden Bediener von Rechnersystemen:

?? Nutzer (nutzt das System)
?? Systemverwalter (verwaltet das System)
?? Systemprogrammierer (programmiert das System)

Jede dieser Bedienerklassen muß unterschiedliche Kenntnisse haben.

Für ein System kann man noch unterscheiden, ob es für die Nutzer im

?? Teilhaberbetrieb

oder

?? Teilnehmerbetrieb

läuft.

Nutzer im Teilhaberbetrieb

Bei einem Teilhaberbetrieb arbeiten alle Anwender mit demselben Programm und denselben Datenbeständen. Diese Form des Betriebs trifft man beispielsweise bei Platzbuchungs- oder sonstigen Auskunftsystemen an. Hier kommt der Anwender in der Regel überhaupt nicht auf die Betriebssystemebene (sieht den prompt des Kommandointerpreters nicht), sondern arbeitet mit vorgefertigten Bildschirmmasken, zwischen denen er zwar umschalten kann, jedoch kann er aus der ihm zugebilligten Anwendung nicht ausbrechen.

Nutzer im Teilnehmerbetrieb

Hier kann jeder Nutzer mit Steuerungsbefehlen im Rechner gespeicherte Programme aktivieren und Daten eingeben. Jeder Nutzer kann unabhängig von einem anderen Programme aufrufen, d.h. es muß nicht jeder Nutzer an derselben Anwendung arbeiten.

Auch hierbei kann man den Nutzer über Auswahlmenüs steuern, man kann ihm jedoch auch Zugriff auf die Betriebssystemebene geben. Dann kann er im Rahmen der ihm zugebilligten Rechte durch Befehle an den Kommandointerpreter des Betriebssystems Dateien lesen, schreiben, ausführen und löschen.

Erlaubt ein System dem Nutzer den Zugriff auf die Betriebssystemebene, dann muß sich dieser mit dem Kommandointerpreter des Betriebssystems unterhalten können, d.h. dessen Sprache sprechen (siehe hierzu Kap. 5).

Systemverwalter

Der Systemverwalter muß das System generieren, starten und stoppen, sowie überwachen können. Er muß ferner Nutzer für das System einrichten und auch wieder austragen können. Hierzu benötigt er über die Kommandos eines Nutzers hinaus **privilegierte Kommandos**, die einem Nutzer nicht zur Verfügung gestellt werden.

Systemprogrammierer

Ein Systemprogrammierer benötigt über die Kenntnisse der Kommandos hinaus Kenntnisse der Mechanismen des Betriebssystems für das Starten, Stoppen und Überwachen von Anwendungsprozessen, sowie die Möglichkeiten der Interprozeßkommunikation zwischen diesen Prozessen (siehe hierzu Kap. 6).

1.4 Einrechner-Betriebssysteme, Netzwerk-Betriebssysteme, Verteilte Betriebssysteme

Ein Betriebssystem klassischer Art verwaltet nur die Betriebsmittel eines Rechners mit einem Prozessor. Ein solches Betriebssystem bezeichnet man als **Einprozessor-Betriebssystem**. Da es auch Multiprozesssor-Betriebssysteme für Stand-alone Rechner gibt, verwenden wir als Oberbegriff für diese beiden Klassen den Begriff **Einrechner-Betriebssystem.**

Im Rahmen verteilter Systeme gibt es ferner die Betriebssystemklassen:

?? **Netzwerk-Betriebssysteme**
?? **Verteilte Betriebssysteme**

Ein **Netzwerk-Betriebssystem** ist ein Betriebssystem, welches Dienste in einem Netzwerk von Rechnern zur Verfügung stellt, wobei jedoch der Benutzer zwischen fernem und lokalem Zugriff auf Betriebsmittel oder Operationen unterscheiden muß. So kann sich der Nutzer explizit auf einem anderen Rechner einloggen. Ein anderes Beispiel ist ein Drucker-Service, der allen Rechnern im Netz zugänglich ist.

Verteilte Betriebssysteme hingegen verbergen die Einzelheiten der Verteilung vor dem Nutzer, indem transparente Zugriffe auf verteilte Komponenten gestattet werden. Transparenter Zugriff auf ein Objekt bedeutet, daß man auf das Objekt über den Objektnamen zugreift, ohne zu wissen, wo es physikalisch liegt.

1.5 Schichtenmodell für Verteilte Systeme

Im allgemeinen lassen sich in einem Verteilten System mindestens drei unterschiedliche Schichten identifizieren, wobei jede einzelne ihrerseits in mehrere Teilschichten aufgegliedert sein kann (Quelle: Herrtwich, Hommel: Kooperation und Konkurrenz):

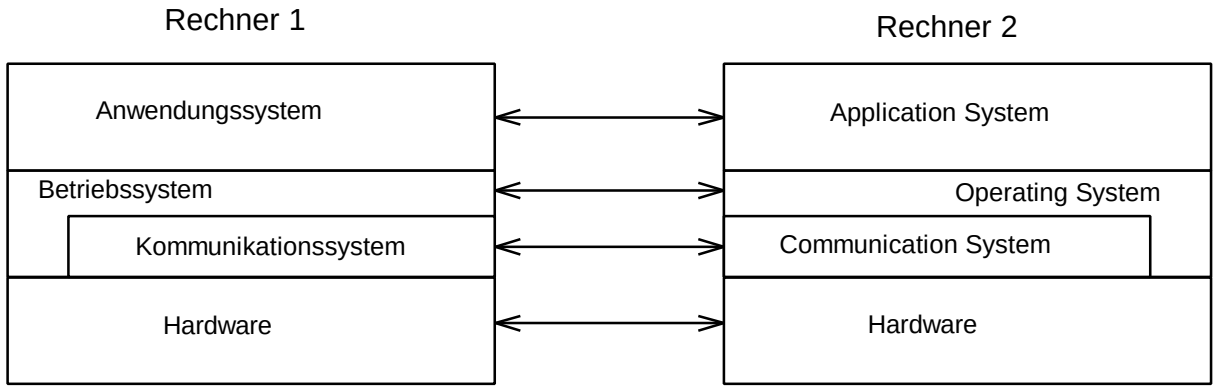


Bild 1.4 - 1: Schichten eines Verteilten Systems

Das **Anwendungssystem (application system)** ist jene Software, mit der man bei der Lösung eines bestimmten Anwendungsproblems direkt in Kontakt kommt wie z.B. ein Textsystem oder ein Flugbuchungssystem.

Das **Betriebssystem (operating system)** eines Rechners sorgt für die Ausführung der Prozesse und stellt auch die Mechanismen für die Interprozeßkommunikation der Betriebssystem-Prozesse auf diesem Rechner bereit.

Das **Kommunikationssystem (communication system)** schließlich ist für den rechnerübergreifenden Austausch von Nachrichten verantwortlich.

Man kann alle Schichten zusammengefaßt als ein Verteiltes System betrachten. Eigentlich bildet aber bereits jede einzelne Schicht für sich ein Verteiltes System, in dem **Prozesse einer Schicht auf einem Rechner mit denen der gleichen Schicht auf einem anderen Rechner anhand eines gemeinsamen Protokolls zusammenarbeiten**, was in Bild 1.4 - 1 durch die **horizontalen Pfeile** ausgedrückt werden soll.

Aber beachten Sie, daß zwei Anwendungssysteme oder zwei Betriebssysteme miteinander nur dann kommunizieren können, wenn beide Rechner ein Kommunikationssystem haben. Dann kann etwa eine Anwendung auf einem Rechner einer anderen Anwendung auf einem anderen Rechner eine Nachricht schicken, oder man kann dem Betriebssystem eines Rechners einen 'remote login'-Betriebssystem-Befehl zum Anmelden beim Betriebsystem eines anderen Rechners erteilen.

Probleme mit der Schichtenbildung:

Die Grenze zwischen den einzelnen Schichten lassen sich oft nicht klar ziehen. Insbesondere wird das Kommunikationssystem oft als Bestandteil des Betriebssystems aufgefaßt.

Im folgenden machen wir ein eigenes Schichtenmodell für Rechner, die in Verteilten Systemen kommunizieren. Wir stellen zuerst die Forderungen für dieses Modell auf.

Anforderungen an ein Modell:

Forderung 100:
Jede Schicht soll nur auf einen Dienst der direkt darunterliegenden Schicht zugreifen können.

Forderung 200:
Das Kommunikationssystem soll nicht als Bestandteil des Betriebssystems, sondern als eine eigenständige Komponente des Rechners aufgefaßt werden.

Forderung 300:
Das Kommunikationssystem soll als Software oder als Hardware oder als Hardware/Software-Kombination realisiert werden können.

Forderung 400:
Das Anwendungssystem soll direkt auf das Betriebssystem zugreifen können
(eine Anwendung hat vielleicht auch etwas anderes zu tun, als nur zu kommunizieren)

Forderung 500:
Das Anwendungssystem soll direkt auf das Kommunikationssystem zugreifen können, wenn es kommunizieren will

Forderung 600:
Das Kommunikationssystem soll auf das Betriebssystem zugreifen können

Forderung 700:
Das Betriebssystem soll auf das Kommunikationssystem zugreifen können

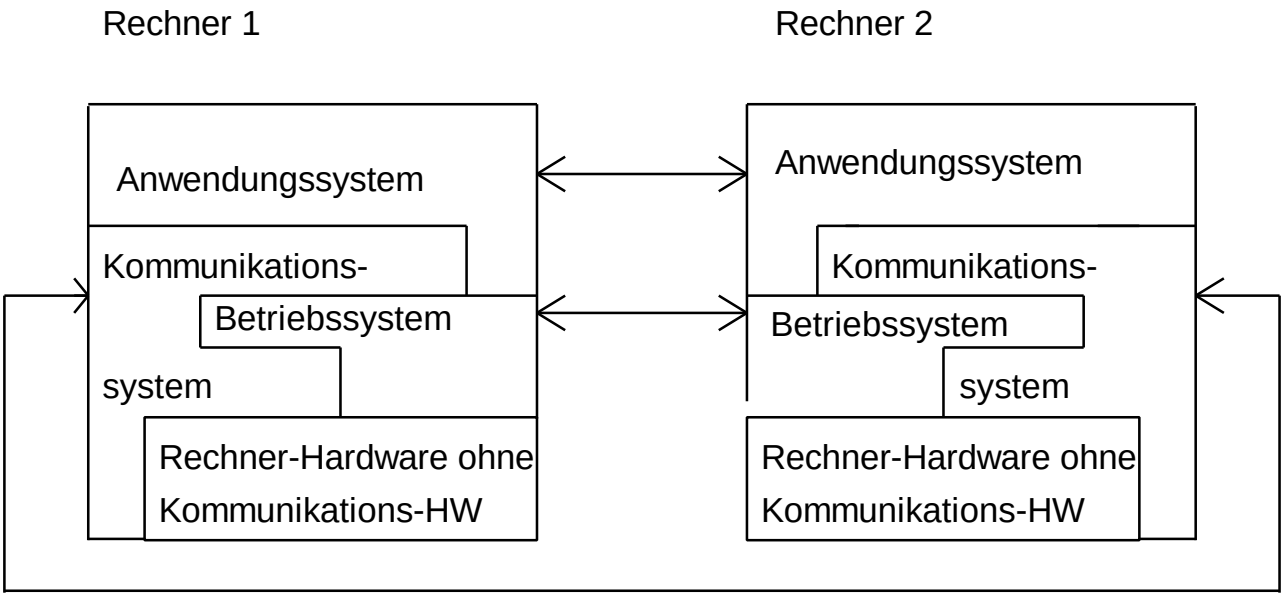


Bild 1.4 - 2: Schichten eines Verteilten Systems

Dieses Schichtenmodell ist zwar verzwickt, aber es erlaubt, die Anforderungen zu erfüllen.

ISO/OSI und TCP/IP

Während früher für das Kommunikationssystem nur herstellerspezifische Produkte verfügbar waren, gibt es heute Produkte der Internet-Protokoll-Familie (basierend auf **TCP/IP**) und gemäß **ISO/OSI**, die einen weltweit verbreiteten de facto-Standard (Internet-Familie) bzw. einen internationalen Standard (ISO/OSI) darstellen.

Die **Internet-Protokoll-Familie** strukturiert das Kommunikationssystem in 4 Schichten:

?? Anwendungsschicht
?? Transportschicht
?? Internetschicht
?? Schnittstellenschicht

Das **ISO/OSI Basic Reference Model** strukturiert das Kommunikationssystem in die 7 Schichten:

?? Anwendungsschicht (Application Layer)
?? Darstellungsschicht (Presentation Layer)
?? Kommunikationssteuerungsschicht (Session Layer)
?? Transportschicht (Transport Layer)
?? Vermittlungsschicht (Network Layer)
?? Sicherungsschicht (Data Link Layer)
?? Bitübertragungsschicht (Physical Layer)

1.6 Aufgaben zu Kap. 1

Aufgabe 1:
Ordnen Sie den folgenden konkreten Betriebsmitteln virtuelle Betriebsmittel zu und geben Sie die Vorteile an:

konkretes Betriebsmittel	virtuelles Betriebsmittel	Vorteile des virtuellen Betriebsmittels
Prozessor		
Externspeicher		
Arbeitsspeicher		

Tragen Sie die Lösung in die Tabelle ein!

2 Die Geschichte von UNIX

[Quelle: Bach: UNIX - Wie funktioniert das Betriebssystem ?
Gulbins: UNIX
Leffler, McKusick, Karels, Quaterman: Das 4.3 BSD UNIX Betriebssystem]

2.1 Entwicklung und Verbreitung von UNIX

Im Jahre **1965** begann ein Projekt von Bell Labs (Bell Laboratorien - sie gehören zu AT&T) mit General Electric und dem MIT, um ein neues Betriebssystem mit Namen MULTICS (**MULT**iplexed **I**nformation and **C**omputing **S**ystem) zu entwickeln, welches Multi-User-Eigenschaften mit privaten und gemeinsam genutzten Daten haben sollte. Das Betriebssystem lief nicht gut (**1969**), und die Bell Labs stiegen aus dem Projekt aus.

Mit dem Ende der Beteiligung am MULTICS-Projekt blieben die Mitarbeiter des Computing Research Centers der Bell Laboratorien ohne die nötige Rechenleistung und wurden offenbar nicht sogleich für das nächste Projekt verplant. Ohne Projektauftrag zur Entwicklung eines neuen Betriebssystems entwarfen Ken Thompson und Dennis Ritchie zur Verbesserung ihrer Programmierumgebung ein Dateisystem, welches sie zusammen mit einem Prozeßsteuersystem und einigen Hilfsprogrammen auf der PDP-7 installierten. Dieses Betriebssystem erhielt den Namen UNIX, den ein anderer Mitarbeiter des Computing Research Centers, Brian Kernighan, für MULTICS geprägt hatte. Geschrieben wurde es in Assembler.

Den ersten praktischen Einsatz fand UNIX auf der PDP-11 im Jahre **1971** für ein Textverarbeitungssystem der Patentabteilung der Bell Laboratorien. Nach diesem ersten Erfolg entwarf und installierte Thompson die Programmiersprache B beeinflußt von BCPL. B war eine interpretative Sprache ohne Datentypen. Um diese Schwächen zu beseitigen, entwickelte Ritchie sie zu C weiter, einer Sprache mit Codegenerator und Datentypen. Im Jahre **1973** wurde UNIX dann neu in C geschrieben. Der Kern des UNIX-Betriebsystems besteht heute aus nur etwa 20 000 Zeilen Programm, von denen rund 1/10 in Assembler geschrieben sind. Maschinenabhängige Assemblerteile werden nur da verwendet, wo hohe Effizienz oder spezielle Maschineneigenschaften dies notwendig machen. Für Hochschulen und Universitäten wurde die UNIX-Quellcodelizenz praktisch für die Kopier- und Dokumentationskosten abgegeben. Zu dieser Zeit durfte AT&T selbst keine Computer verkaufen, da es ein diesbezügliches Abkommen mit der Regierung gab und AT&T demnach ein Telefon-Monopol hatte.

UNIX Version 6, allgemein bekannt als V6, war im Jahre 1976 die erste Version, die weit und breit außerhalb der Bell Labs erhältlich war. Das System unterschied sich von anderen Betriebssystemen in drei wesentlichen Punkten:

?? es war in einer höheren Programmiersprache geschrieben
?? es wurde im Quellcode verkauft
?? das System war auch auf kleinen Rechnern leistungsfähig

3% der Betriebssystem-Funktionen, für die es erforderlich war, z.B. das Context-Switching, wurde durch Assemblerteile erreicht.

Hinweis:

> Context-Switching ist die Unterbrechung des gerade laufenden Prozesses und Umschaltung auf einen anderen Prozeß, wobei sichergestellt wird, daß ein Prozeß bei Wiederanlauf seine Daten genauso vorfindet, wie er sie vor der Unterbrechung hatte.

1978 war das Jahr der ersten Portierung von UNIX auf einen anderen Rechner als eine PDP.

Mit der wachsenden Verbreitung von Mikroprozessoren portierten viele Hersteller UNIX auf die neuen Maschinen, wobei immer weitere Verbesserungen zu vielen Varianten führten.

Zwischen 1977 und 1982 vereinigten die Bell Laboratorien verschiedene AT&T-Varianten zu einem einzigen System, das unter **UNIX System III** bekannt wurde. Später wurde System III von den Bell Labs erweitert zu UNIX System V, und **AT&T** kündigte schließlich im **Januar 1983 offizielle Unterstützung** für das **System V** an. In der Zwischenzeit hatte die Berkeley Universität gesponsert durch DoD-Aufträge eine eigene Variante von UNIX, das BSD-UNIX, auf VAX-Rechnern mit interessanten Eigenschaften wie z.B. paging on demand (siehe später) entwickelt. Die Berkeley University entwickelte auch TCP/IP zur Vernetzung von UNIX-Rechnern.

Anfang **1984** gab es weltweit bereits etwa **100 000 UNIX-Installationen,** die auf einem breiten Spektrum von Rechnern liefen.

Mit UNIX wurde in der Vergangenheit in der Regel ein C-Compiler mit ausgeliefert, so daß die Verbreitung von C automatisch auch mit der Verbreitung von UNIX einher ging.

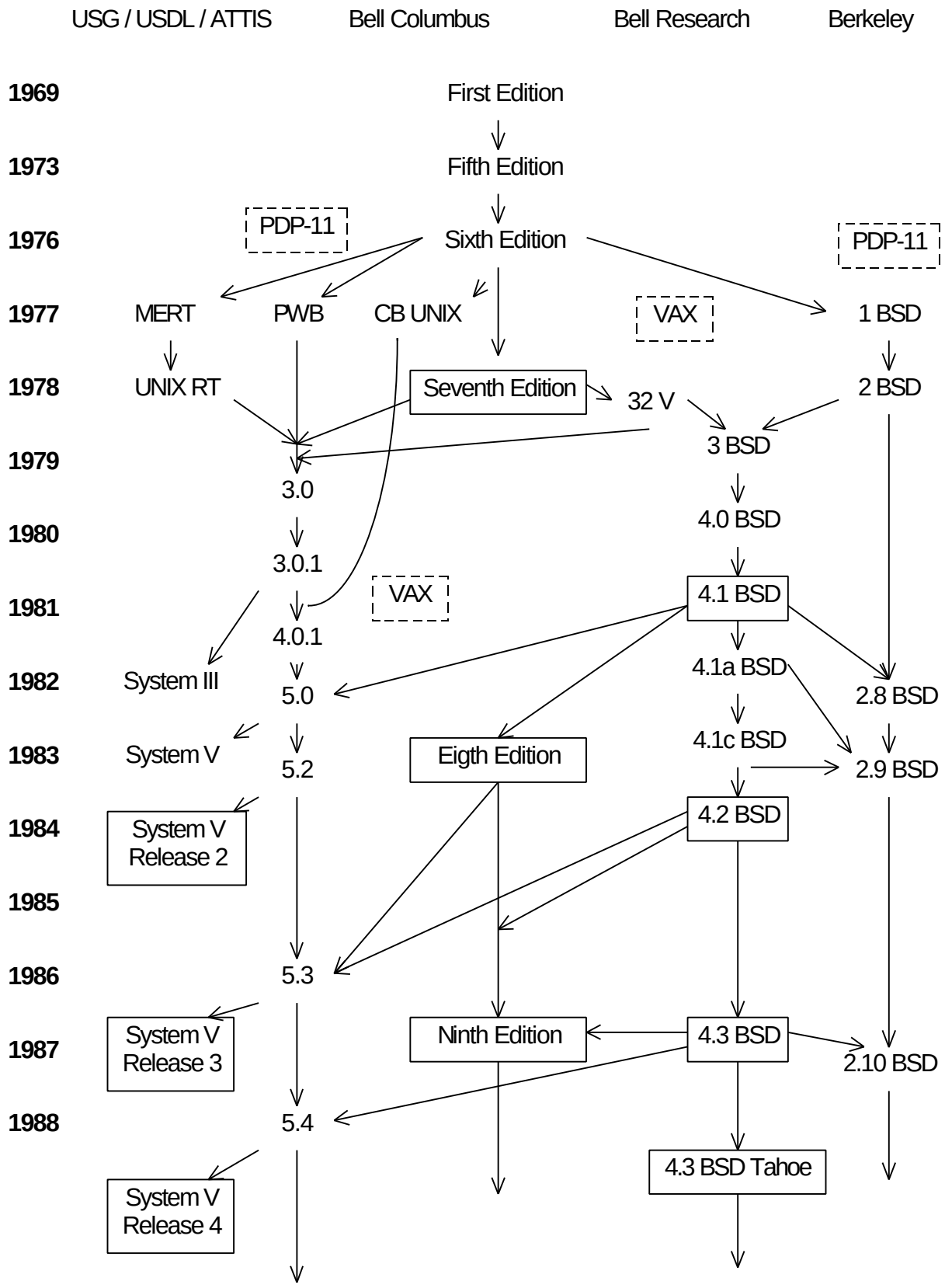


Bild 2.1 - 1 Die Entwicklungslinien von UNIX

Während das AT&T Unix zu Beginn zentrale Eigenschaften hatte und für den Fall von Mehrprozessor-Systemen auf einen Rechner mit mehreren Prozessoren setzte und dafür den Mechanismus des Shared Memorys für die Interprozeß-Kommunikation bereit stellte, setzte das BSD Unix bereits frühzeitig auf Verteilte Systeme. Im BSD-UNIX wurden die sogenannten Sockets geschaffen. Sockets sind Nachrichtenkanäle, die an die TCP/IP-Protokolle anflanschen und damit Rechnergrenzen in der Interprozeß-Kommunikation (Kommunikation von Prozeß zu Prozeß) überwinden.

Heutige Versionen von UNIX enthalten in der Regel Mechanismen aus der AT&T- und der BSD-Linie.

2.2 Standardisierung von UNIX

Im Rahmen der Standardisierung **offener Systeme (d.h. herstellerunabhängiger Systeme) im Sinne des X/Open Portability Guides bzw. von POSIX** werden Schnittstellen von Programmen zu UNIX-Betriebssystemen (nicht der Aufbau der Betriebssysteme der verschiedenen Hersteller) standardisiert. Die bevorzugte Sprache, für die als erste standardisierte Schnittstellen festgelegt werden, ist hierbei die Programmiersprache C. Der X/Open Portability Guide befaßt sich über die Standardisierung der Schnittstellen von Programmen zu UNIX-Betriebssystemen auch mit der Standardisierung von Utilities, Netzwerkdiensten, Window-Management, Daten-Management etc.

3 Relevante Schichtenmodelle für UNIX-Rechner

3.1 Schichtenstruktur von UNIX.

Das Betriebssystem UNIX besteht aus drei Schichten:

?? dem Systemkern (Kernel)
?? den Werkzeugen (Utilities)
?? dem Kommandointerpreter (Shell)

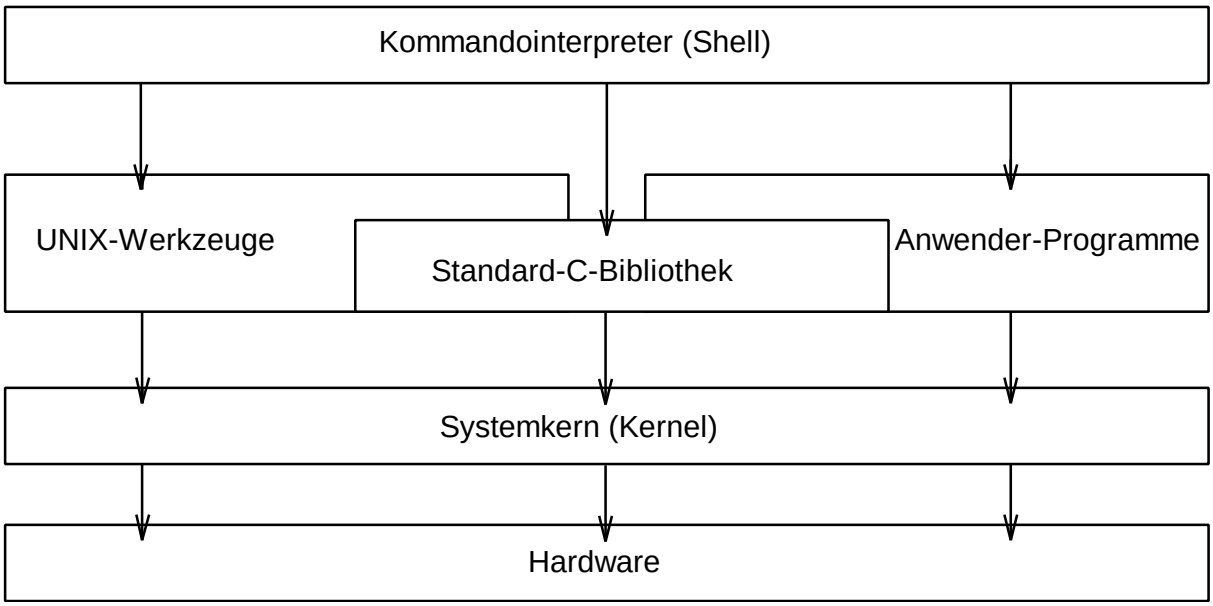


Bild 3.1 - 1 Schichtenstruktur von UNIX. Hinweis: ? bedeutet: ruft auf

Der **Kernel** hat die folgenden Aufgaben zu erfüllen:

?? Verwaltung des Speichers und der Prozesse für Systemsoftware (Betriebssystem, X Window, etc.) und Anwendungen
?? Verwaltung des File-Systems und Ansteuerung von peripheren Geräten wie Terminals, Drucker, Band- und Diskettenstationen etc.

Die **Shell** interpretiert die Kommandos der Anwender und führt sie aus. Die Shell (Shell-Scripts) ist eine vollwertige Programmiersprache zur Automatisierung von Abläufen auf der Kommando-Ebene und für Rapid Prototyping,

Für den Aufruf von **Utilities** sind weit über 200 Kommandos vorhanden. Die Art der Ausführung der meisten Kommandos ist über Optionen steuerbar.

Kommandointerpreter (Shell), Compiler, Editoren und ähnliche anwendungsunabhängige Systemprogramme sind kein Anteil des Systemkerns. Sie werden zwar typischerweise mit dem Betriebssystem vom Computerhersteller ausgeliefert, arbeiten aber wie normale Anwender-Programme im sogenannten **User-Modus**. Der **Systemkern** ist jener Anteil der System-Software, der im **Kernel-Modus** arbeitet. Dieser Modus ist in der Regel gegen Fehleingriffe der Benutzer durch die Hardware geschützt (Ältere Prozessoren verfügen nicht alle über diesen Schutz).

Systemaufrufe (system calls)

Dienstleistungen des Betriebssystems können durch **system calls** angefordert werden. Dies sind Funktionsaufrufe bzw. Prozeduraufrufe in Assembler oder in höheren Programmiersprachen wie C (Bibliotheksfunktionen). Eine solche Funktion schreibt die Parameter eines Systemaufrufs an angegebene Stellen, wie zum Beispiel Maschinenregister, und führt dann einen TRAP-Befehl aus, um das Betriebssystem zu starten. Mit den Bibliotheksfunktionen wird der Zweck verfolgt, die Details des TRAP-Befehls vor dem Nutzer zu verstecken und den system calls das Aussehen eines gewöhnlichen Funktionsaufrufs zu verleihen. Der TRAP-Befehl ist auch als **Kernaufruf** bekannt.

Bei einem Aufruf einer Betriebssystem-Routine (system call) wird also vom **user mode** in den **kernel mode** umgeschaltet.

Wenn das Betriebssystem nach einem TRAP die Kontrolle übernimmt, prüft es die Parameter auf Gültigkeit und führt, falls sie gültig sind, die angeforderte Arbeit aus. Wenn die Arbeit beendet ist, schreibt das Betriebssystem einen Statuscode in ein Register, um den Erfolg oder Mißerfolg der ausgeführten Aktion zu dokumentieren, und führt dann den RETURN FROM TRAP-Befehl aus, um die Kontrolle an die Bibliotheksfunktion zurückzugeben. Die Bibliotheksprozedur kehrt dann auf die übliche Art und Weise zum Aufrufer zurück und übergibt diesem den Statuscode in Form eines Funktionswertes. Manchmal werden zusätzliche Werte in den Parametern zurückgegeben.

Ein Systemaufruf erzeugt, zerstört oder benutzt verschiedene Software-Objekte, die durch das Betriebssystem verwaltet werden. Die wichtigsten Objekte sind

- Prozesse
- und Dateien

3.2 Komponenten des Kernels

In DV3 hatten wir gelernt, daß Betriebssysteme i.a. die folgenden wesentlichen Komponenten haben:

- ?
- ?? Prozeßverwaltung mit Scheduler, Memory Management und Interprozeß-Kommunikation
- ?? Dateisystem
- ?? Gerätetreiber für die HW-Schnittstellen
- ?? Kommandointerpreter
- ?

Dies sehen wir auch, wenn wir den Aufbau des Kernels betrachten (Anmerkung: der Kommandointerpreter (die Shell) liegt über dem Kernel auf der Ebene der Anwendung).

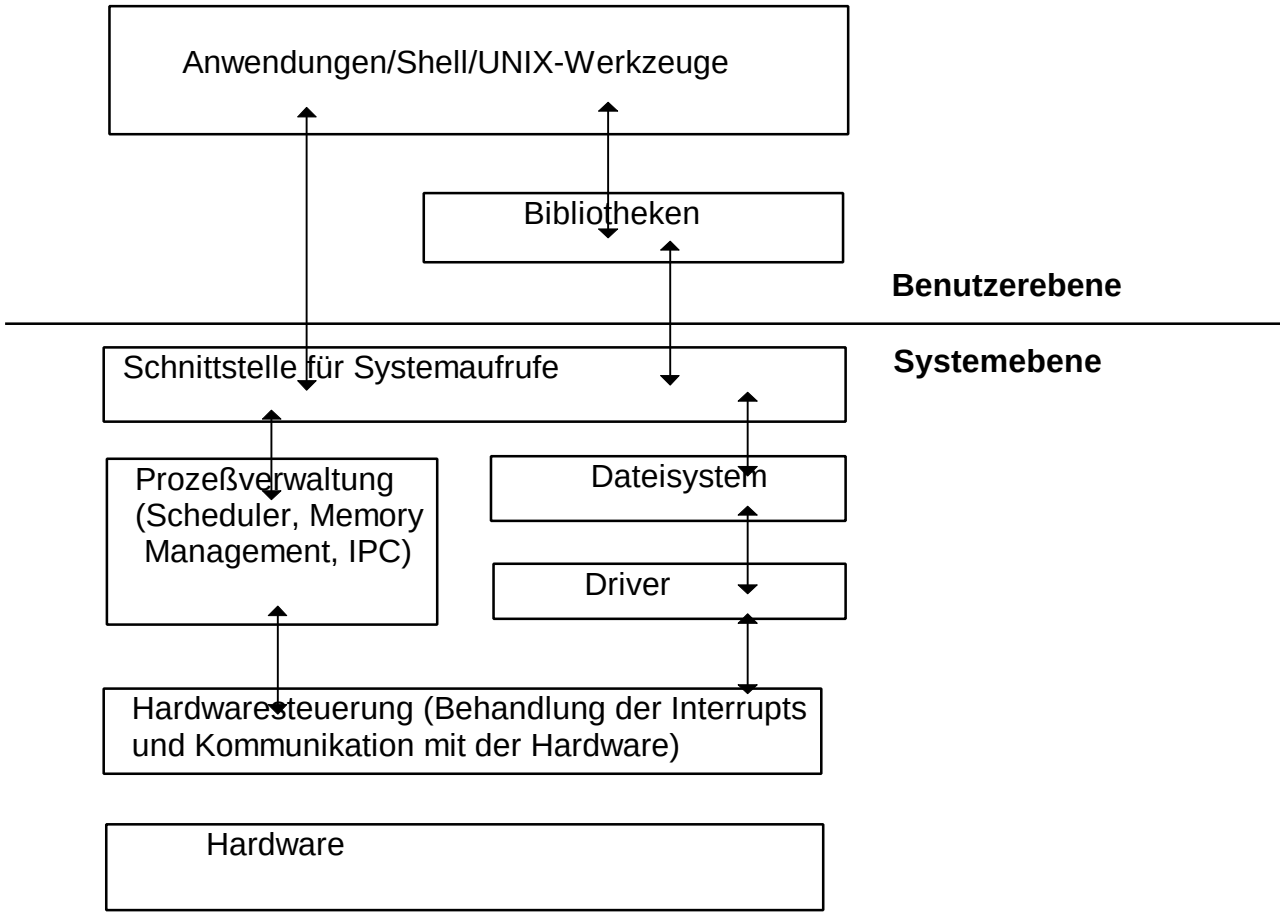


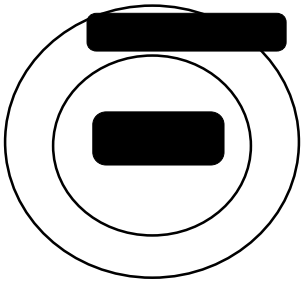
Bild 3.2 - 1 Architektur von UNIX [Quelle: Bach, UNIX - Wie funktioniert das Betriebssystem ?]

Dieses Bild zeigt auch - wie schon gesagt - daß Prozesse und Dateien die wichtigsten Objekte des Kernels sind.

Aus Zeitgründen werden in diesem Kurs Prozesse bevorzugt behandelt. Dateien sind Ihnen bereits von MS-DOS grundsätzlich bekannt und können aus Zeitgründen hier nur sehr stiefmütterlich behandelt werden.

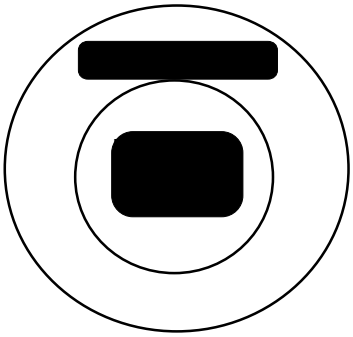
3.3 Schalenmodelle

Zeichnet man keine Schichten, sondern Schalen

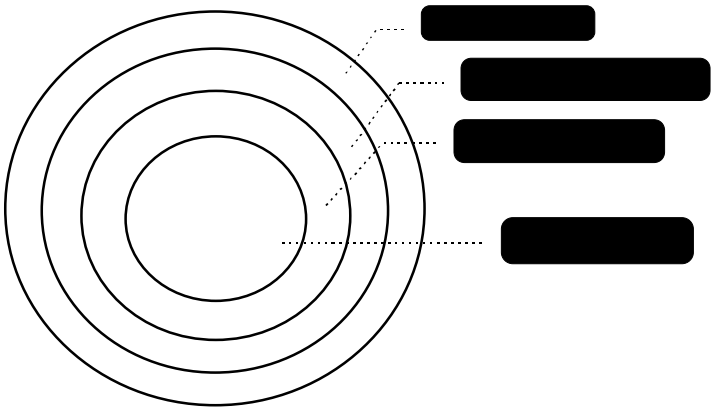


so schmiegt sich der Kommandointerpreter wie eine Schale (Shell) um den Kernel.

Ein entsprechendes Schalenmodell kann man auch für die beiden Modi Kernel Modus und User Modus machen:



Entsprechende Schalenmodelle gibt es auch bei anderen Betriebssystemen. VMS hat beispielsweise 4 Modi:



Der Grund für solche Schalenmodelle ist der Aufbau einer Schutzhierarchie. Ein fehlerhaftes Nutzerprogramm soll nicht die Möglichkeit haben, Speicherstellen des Betriebssystems zu überschreiben. Wenn beispielsweise ein Prozeß versucht, Speicherstellen des Betriebssystems zu referenzieren, wird er abgebrochen. Wenn ein Speicherbereich in einem gegebenen Mode zugänglich ist, ist er auch in einem höheren Mode zugänglich, aber nicht in einem niederigeren Mode.

Die äußerste Schale in obigem Bild umfaßt das gesamte Memory. Irgendein Punkt innerhalb dieses Kreises ist zugänglich von jeder Stelle eines inneren Kreises, aber nicht von einem äußeren Kreis.

Bei VMS laufen in den verschiendene Modi:

Kernel Mode	Swapper Verarbeitungsroutinen für Interrupts und Fehlerfälle (Exceptions) Routinen für Prozeß-Wechsel
Executive Mode	das Dateisystem
Supervisor Mode	der Kommando-Interpreter
User-Mode	Systemprogramme, die keinen privilegierten Zugriff benötigen wie Compiler oder aber Anwendungsprogramme

Design von Unix

[Quelle: P. Schnupp, Standard-Betriebssysteme]

Ob man den Kommandointerpreter und Dienstleistungsprogramme (Utilities) wie hier im Falle von UNIX ebenso lose an den Kern anhängt wie Anwendungsprogramme, ist eine Frage des Designs des Betriebssystems.

Im Falle von MS-DOS ist das Design anders: Hier sind der Kommandoprozessor und auch Dienstleistungen (interne und externe DOS-Befehle) in das Betriebssystem integriert, d.h. sie sind zu einem sehr großen Teil von System-Internas abhängig.

Betriebssystem als virtuelle Maschine

UNIX selbst ist eine **virtuelle Maschine**. Viele seiner Mechanismen haben genau die Aufgabe, ein konkretes Betriebsmittel in sein virtuelles Gegenstück zu verwandlen. Ein typisches Beispiel hierfür sind die **termcap-** und **terminfo**-Dateien, welche die Eigenschaften aller verwendeten, zeichenorientierten Bildschirmgeräte so verkoden, daß sowohl der Anwendungsprogrammierer als auch der UNIX-Benutzer von einem **virtuellen Terminal** ausgehen können. Das virtuelle Terminal kann auch bei einer Anlage auf unterschiedlichen Bildschirmgeräten verschiedener Hersteller realisiert werden. Hierzu muß nur jeweils die Datei **termcap** bzw **terminfo** angepaßt werden.

3.4 Einordnung eines Benutzeroberflächen-Toolkits in ein Schichtenmodell

[Quellen:
Niall Mansfield
The X Window System: A User's Guide
Addison-Wesley Publishing Company
September 1989

Benutzeroberflächen für UNIX-Workstations
Design & Elektronik
Ausgabe 1 vom 9. 1. 1990

Markus Stumpf
X Window System, Version 11, Release 4 - eine Übersicht
Design & Elektronik
Ausgabe 9 vom 2. 5. 1990

Bernhard Pfeiffer/ Franz X. Glas
Das X-Window-System - Grundlagen und Programmierbeispiele,
Design & Elektronik
Teil 1 - Ausgabe 11 vom 30. 5. 1989
Teil 2 - Ausgabe 18 vom 5. 9. 1989
Teil 3 - Ausgabe 25 vom 12. 12. 1989
Teil 4 - Ausgabe 1 vom 9. 1. 1990

Das Ende des Fenster-Chaos
Open Look - die System-V-4.0 Oberfläche
IX Multiuser-Multitasking-Magazin, Ausgabe 2/89

Motivationen. Die Architektur von Motif
IX Multiuser-Multitasking-Magazin, Ausgabe 2/90

Informatik-Spektrum, 14, (1991), S. 34]

Die aufwendige Einarbeitung, die in der Vergangenheit kennzeichnend für UNIX war, betraf nicht nur die Gewöhnung an das Betriebssystem selbst, sondern auch das Erlernen der verschiedenen Softwarepakete, die alle ihr eigenes **Look-and-feel**-Konzept, d.h. ihr eigenes Erscheinungsbild und ihre spezifischen Bedienungseigenschaften hatten. Steht eine **standardisierte Benutzeroberfläche** zur Verfügung, so können die Anwendungsentwickler, die sich an die Norm halten, Applikationsprogramme mit einer einheitlichen Benutzeroberfläche entwickeln. Damit können die selbst geschriebenen, anwendungsspezifischen Dialoge sowie die Dialoge mit kommerziell verfügbaren Produkten, die der Norm entsprechen, in einheitlicher Art und Weise geführt werden.

Zur Zeit gibt es unter UNIX immer noch zwei genormte Bedienoberflächen, nämlich **Open Look** und **OSF/Motif**. Allerdings wurde 1994 bekannt, daß AT&T/Sun Open Look aufgeben wird.

Bild 3.2 - 1 zeigt die Einordnung eines solchen Toolkits in ein Schichtenmodell. Um den Stellenwert eines solchen Toolkits richtig einordnen zu können, wird zunächst die Funktionalität der verschiedenen Schichten kurz vorgestellt.

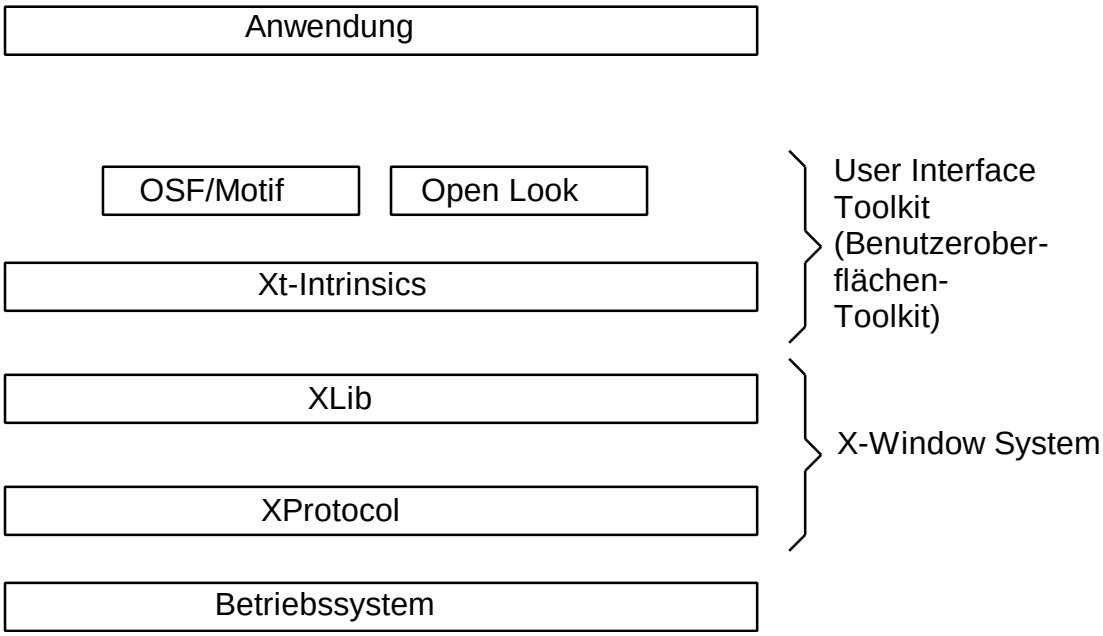


Bild 3.2 - 1 Schichtung des X Window Systems und des User Interface Toolkits (Anmerkung: In diesem Bild kann die Anwender-Software auch auf tiefere Schichten direkt, d.h. ohne die Vermittlung der dazwischen liegenden Schichten zugreifen)

3.4.1 Das X Window System (XLib und XProtocol)

Die Forderung nach graphisch-interaktiven Benutzerschnittstellen, die leicht zu bedienen sind und der zunehmende Einsatz von Netzwerken führte zu der Entwicklung von **X Window**. X Window ist ein **netzwerkfähiges graphisches Fenstersystem**. Es definiert ein **netzwerktransparentes, rechner- und herstellerunabhängiges Protokoll** zur **Kommunikation zwischen Anwendungsprogrammen und einem Arbeitsplatz.**

Das **X Window System** wurde am **MIT (Massachusetts Institute of Technology)** in Cambridge, MA (USA) entwickelt. Es wird vom **X-Consortium** (Zusammenschluß der Firmen und Institutionen, die die Verbreitung von X Window unterstützen) betreut. Es kann vom MIT als voll dokumentierter Source Code gegen eine geringe Gebühr erworben werden.

Systemmodell

Grundlage des X-Window-Systems ist ein **Client-Server-Modell**. Der **Server (X-Server)** verwaltet die Geräte genau **eines** Arbeitsplatzrechners, zu dem neben

?? der Tastatur und
?? einem Zeigegerät (üblicherweise einer Maus) auch
?? ein oder mehrere Bildschirme gehören.

Der X-Server verwaltet die graphischen Ressourcen wie

?? die Bildschirmfläche in Form von rechteckigen Bildschirmfenstern (Windows)
?? virtuelle Ausgabebereiche im Speicherbereich des X-Servers (Pixmaps)
?? Farbtabellen und Farbtabelleneinträge
?? Bildschirm-Schriftarten (Fonts)

Er empfängt und bearbeitetet Ausgabe- und Verwaltungsanforderungen (**Requests**) der Anwendungen und benachrichtigt die Anwendungen durch asynchrone Zustellung von Ereignissen (**Events**) vom Eintreffen von Benutzereingaben und von Änderungen in der Geometrie (Größe, Position, Sichtbarkeit) der ihnen zugeordneten Fenster.

Die **Programme** spielen die **Rolle der Clients** und richten ihre Anforderungen zur Nutzung der Bildschirm-Ressourcen an den Server und fordern ihn mit Hilfe eines eigens definierten **Protokolls** dazu auf, Texte und Grafiken darzustellen. Client-Programme können nicht nur auf der lokalen Workstation, sondern irgendwo im Netzwerk laufen. Für die Kommunikation können beliebige Transportmechanismen eingesetzt werden, solange sie nur einen zuverlässigen Byte-Strom realisieren.

Sämtliche Geräteabhängigkeiten werden durch den X-Server verborgen. Somit sind die Anwendungen durch die Benutzung des X-Protokolls in der Lage, ohne Änderungen oder Neuübersetzen auf den unterschiedlichsten Darstellungssystemen Ein- und Ausgaben durchzuführen.

Das X-Window System gliedert sich in zwei Teile:

?? **eine Funktionsbibliothek (Xlib)** und
?? **eine Protokollschicht (XProtocol)**.

XLib ist die niederste Programmierschnittstelle zum X-Window-System. Es handelt sich bei XLib um eine **niedere C-Bibliothek**, die ein unter X laufendes **Client-Applikationsprogramm** nutzen kann, um (mit Hilfe von X-Protocol) **mit einem beliebigen X-Server in Verbindung zu treten**. Die XLib verwaltet die grundlegenden Ressourcen des Systems (Bildschirmfenster, Farbmasken, Eingabegeräte) und stellt die bitmaskierten Darstellungsfunktionen zur Verfügung. Diese Bibliothek von Unterprogrammen enthält z.B. Routinen für die:

?? **Schnittstellen zum X-Protokoll**
?? **Verbindungsverwaltung Client-Server**
?? **Verwaltung von Window-Eigenschaften**
?? **Verarbeitung von Events**
?? **Durchführung der graphischen Ausgabe**

Oberhalb von XLib setzen **Toolkits oder Client-Anwendungen** auf. Je besser das Toolkit ist, umso kleiner wird der Aufwand für die Client-Anwendung.

3.4.2 Das Benutzeroberflächen-Toolkit

Die Benutzeroberfläche umfaßt alle Einrichtungen, die dem Dialog zwischen Bediener und System dienen. Sie baut auf dem Fenstersystem auf und bildet die eigentliche Implementierung des 'Look-and-feel-Standards' für ein einheitliches Erscheinungsbild und einheitliche Bedienungseigenschaften.

Beispiel:

> Wenn ein Programmierer beispielsweise eine scroll-fähige Text-Box erstellt, legt der Look-and-feel-Standard fest, auf welche Weise der Text gescrollt und editiert werden kann, welche Befehle notwendig sind, um an das Ende oder den Anfang des Textes zu gelangen, und auf welche Weise die Text-Box wieder verlassen werden kann.

Prinzipiell können Applikationsprogramme ausschließlich aus direkten Aufrufen an die niedere XLib-Schnittstelle geschrieben werden, jedoch erstellt man sie in der Regel mit einer höheren Programmbibliothek, dem Benutzeroberflächen-Toolkit.

Das Toolkit verkörpert die physische Implementierung des Look-and-feel-Standards für die Bildschirmfenster-Umgebung des Computers. Es vereinfacht die für den Aufbau einer solchen Umgebung notwendige Programmierarbeit, indem es **Toolkit-Bausteine** wie

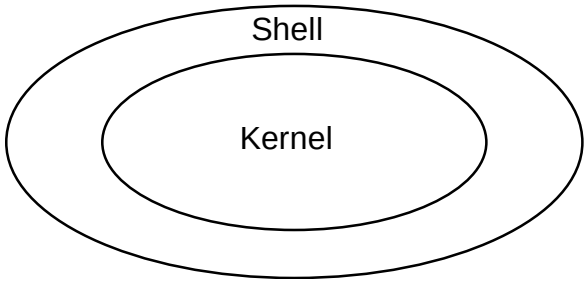
?? **Scroll-Balken**
?? **Editierfenster**
?? **Befehlstasten**

etc. zur Verfügung steht. Das Programmieren mit solchen Toolkit-Bausteinen ist bedeutend einfacher als die völlige Neuentwicklung jeder einzelnen Funktion.

4 Kommandointerpreter (Shell)

[Quelle: Gulbins, UNIX]

Kommandosprachen erlauben es einem Bediener, an das Betriebssystem Befehle zu übergeben. Auf den prompt des Kommandointerpreters kann der Bediener seinen Befehl abgeben und mit <RETURN> abschließen. Der Kommandointerpreter in UNIX heißt Shell. Die Shell umschließt den Kern des Betriebssystems



Die Shell interpretiert die Kommandos und setzt sie in Betriebssystemaufrufe um, überprüft die Rückmeldungen des Systems und setzt sie im Fehlerfall in Fehlermeldungen um.

Gängige Shells

Als Kommandointerpreter kann man über dem Betriebssystem verschiedene Shells einhängen, beispielsweise eine der kommerziell bedeutenden Shells

- die **Bourne-Shell** (stammt aus dem System V AT&T Unix)
- die **C-Shell** (stammt aus dem BSD UNIX)
- oder die **Korn-Shell**

Man könnte sich aber auch eine Shell selbst schreiben, da die Architektur von UNIX die Shell nicht als ein Systemprogramm, sondern als ein Anwenderprogramm sieht.

Die älteste und heute nicht mehr so weit verbreitete Shell ist die Bourne-Shell. Die C-Shell ist heutzutage überaus verbreitet. Sie hat eine C-artige Syntax und mehr Funktionalität als die Bourne-Shell. So kann man beispielsweise mit Hilfe eines History-Mechanismus die eingegebenen Kommandos speichern und dem Benutzer wieder zur Verfügung stellen wie z.B. in DOS mit Hilfe von `doskey`.

Kommandos und Programme

Ein Programm wird durch Angabe seines Namens aufgerufen. Ein Programm entspricht also einem Kommando. In der Regel wird ein Kommando auch durch ein eigenes

Programm realisiert. Aber wie im Falle von MS-DOS, gibt es einige Kommandos, die direkt von der Shell abgearbeitet werden, wie zum Beispiel das Wechseln des Verzeichnisses durch **cd.**

Anmerkung:

MS-DOS hat einige wenige Merkmale von UNIX übernommen, hierzu gehört auch das Umleiten der Ausgabe oder Eingabe eines Programms.

Anmerkung:

In DOS gibt es **interne und externe DOS-Kommandos.** Dabei sind die sogenannten **internen Kommandos** in ihrer Gesamtheit im transienten Teil von COMMAND.COM enthalten, während hingegen **externe Kommandos (externe Befehle)** eigenständige kleine Hilfsprogramme in Form von ablauffähigen Dateien sind, die bei Aufruf des externen Kommandos in den Hauptspeicher geladen werden.

Kommandoprozeduren

Es ist möglich, nicht nur einzelne Befehle der Kommandosprache abzusetzen, sondern ganze Dateien von Befehlen (Batch jobs). Eine solche Kommandofolge nennt man in UNIX ein **Script**. Eine solche Kommandofolge entspricht einer .BAT-Datei unter DOS.

4.1 Grundfunktionen einer Shell am Beispiel der Bourne-Shell

4.1.1 Der Prompt der Shell

Die Shell benutzt drei Arten von **Prompts (Bereit-Zeichen)**:

$ im Modus eines normalen Benutzers, wenn sie bereit ist, das nächste Kommando zu verarbeiten

im Modus des **Super-Users (Systemverwalters)**, wenn sie bereit ist, das nächste Kommando zu verarbeiten

> wenn das eingegebene Kommando noch nicht vollständig ist und die Shell weitere Eingaben zur Vervollständigung des Kommandos benötigt. Gibt man etwa am Ende der Zeile ein <RETURN> ein, obwohl das Kommando noch nicht fertig ist, dann bringt die Shell in der neuen Zeile den Prompt >. Hinter diesem Prompt > schreibt man dann weiter.

Die **C-Shell** gibt als Prompt **%** anstelle von **$** aus.

4.1.2 Ein-/Ausgabeumlenkung

Die Standarddateien

?? **Standardausgabe**
?? **Standardfehlerausgabe**
?? **Standardeingabe**

zeigen normalerweise auf den Bildschirm bzw. die Tastatur, können aber - genau wie in MS-DOS auch - auf andere Dateien oder Geräte umgelenkt werden. Die Standard-dateien werden vom Laufzeitsystem zur Verfügung gestellt Die **Standardeingabe** hat unter UNIX die Nummer **0**, die **Standardausgabe** die Nummer **1**, die **Fehlerausgabe** die Nummer **2**.

Umlenken der Standardeingabe und der Standardausgabe

Die Symbole sind dieselben wie unter DOS, nämlich

> dateiname

für das Umlenken der Ausgabe in die Datei **dateiname**, und

< dateiname

für das Umlenken der Eingabe von der Tastatur.

Beispiele:

ls > goll.tmp

Das Inhaltsverzeichnis (erzeugt mit dem Befehl `ls`) wird in die Datei `goll.tmp` geschrieben. Derselbe Effekt wird erreicht durch

```
ls 1 > goll.tmp
```

(Anmerkung: funktioniert bei Bourne- und Korn-Shell, nicht bei C-Shell)

Mit dem Befehl

cat goll.tmp

kann man sich dann den Inhalt der Datei `goll.tmp` ansehen. **cat** entspricht also dem **type**-Befehl von MS-DOS.

Mit dem Befehl

wc < goll.tmp

wird die Datei goll.tmp als Input-Datei für das Programm **w**ord **c**ount (**wc**) verwendet. **wc** zählt die Anzahl der Zeilen, Worte und Zeichen von der Standardeingabe, bis die End-of-File-Marke erreicht ist (an der Tastatur eingegeben als <CTRL> + <D> im Gegensatz zu MS-DOS mit seinem <CTRL> + <Z>) und gibt ihre Zahl aus. Anmerkung: Man kann dasselbe Ergebnis erreichen durch den Aufruf:

wc goll.tmp

Enthält `goll.tmp` 3 Zeilen, 3 Worte, 23 Zeichen, so liefert der Befehl **`wc < goll.tmp`** beispielsweise das Ergebnis:

```
3 3 23
```

Dieses Ergebnis kann man auch in eine andere Datei wegschreiben, beispielsweise in die Datei `goll2.tmp`. Dann sieht der zugehörige Befehl folgendermaßen aus:

```
wc < goll.tmp > goll2.tmp
```

Die Angaben zur Umlenkung der Standardeingabe und der Standardausgabe werden dabei nicht an das Programm weitergeleitet, sondern die Umlenkung wird von der Shell durchgeführt.

Append-Operation

Zusätzlich gibt es noch den Operator >>**,**

Will man die Ausgabe eines Kommandos an eine vorhandene Datei anhängen, so erfolgt dies mit dem Operator **>>**. Beispiel:

```
ls >> goll.tmp
```

Das durch den Befehl **`ls`** erzeugte Verzeichnis wird an die Datei `goll.tmp` angehängt. Existiert die Ausgabedatei für den Operator **>>** noch nicht, so wird sie neu erzeugt.

Umlenkung der Fehlerausgabe

Die Standardfehlerausgabe kann mit "2 > datei" umgelenkt werden (Anmerkung: bei Bourne-Shell und Korn-Shell, nicht bei C-Shell)

Beispiel

```
wc < goll.tmp > goll2.tmp 2 > fehlerdatei
```

Piping

Wie unter MS-DOS auch, kann die Standardausgabe eines Programms als Standard-eingabe für das nächste Programm durch den Operator **|** umgeleitet werden**.** Den dabei verwendeten **Puffer**, in den das eine Programm hineinschreibt und das andere Programm herausliest, wird als **Pipe** bezeichnet. Der Mechanismus wird als **Fließbandverarbeitung** oder **pipelining** bezeichnet. Der Vorteil ist zum einen, daß hierbei kein Zugriff zur Platte erforderlich ist, sondern nur zum Arbeitsspeicher (es wird im Arbeitsspeicher quasi eine temporäre Datei angelegt, die wieder automatisch gelöscht wird) und zum anderen, daß ein paralleles Arbeiten ermöglicht wird. Hat das erste Programm in den Puffer der Pipe hineingeschrieben, bis dieser gefüllt ist, leert

das zweite Programm den Speicher. Während es den Speicher leert, kann das erste Programm bereits wieder hineinschreiben. Durch das Hintereinanderreihen von UNIX-Programmen mit Hilfe von Pipes kann man mächtige Funktionen erzeugen. Die einzelnen Programme können dabei ziemlich klein sein.

Beispiel:

```
ls | wc
```

Hier wird die Ausgabe des Programms **ls** als Eingabe für das Programm **wc** verwendet.

Ein Programm, welches Daten von der Standardeingabe liest, diese verarbeitet und sein Resultat auf die Standardausgabe schreibt, wird als **Filter** bezeichnet. **wc** ist also ein Filter. **Verschiedene Filter können also durch Pipes verbunden werden**.

Beispiel:

prog2 und **prog3** seien Filter. Dann lassen sie sich mit **prog1**, welches Ausgaben an die Standardausgabe erzeugt, mit Hilfe von pipes verbinden:

```
prog1 | prog2 | prog3
```

4.1.3 Hintergrundprozesse

Wird ein Programm aufgerufen, so startet die Shell das entsprechende Programm und wartet auf die Beendigung des Programms oder auf dessen Abbruch durch den Bediener (Eingabe von: <ctrl \>. Erst dann kann die Shell das nächste Kommando annehmen.

Startet man das Programm, indem man an das Kommando noch einen & anhängt (Beispiel: `kommando &`), so startet die Shell für dieses Programm einen eigenen Prozeß (**Hintergrundprozeß**), d.h. eine neue Shell, in der das Kommando läuft. Die alte Shell ist unmittelbar nach Programmstart wieder für die nächste Kommandoeingabe bereit. Hat die Shell einen Hintergrundprozeß gestartet, so gibt sie dessen Prozeßnummer **pid** aus und wartet auf die nächste Eingabe. Die Ausgabe der Prozeßnummer ist deshalb wichtig, damit man bei Bedarf diesen Prozeß mit Hilfe des Kommandos

```
kill pid
```

abbrechen kann.

Schickt man mehrere Prozesse in den Hintergrund, die Ausgaben erzeugen, so kann man auf dem Bildschirm einen bunten Ausgabemix erhalten, wenn man die Ausgaben der verschiedenen Hintergrundprozesse nicht umlenkt.

4.1.4 Gruppieren von Kommandos

In einer Zeile können mehrere Kommandos an die Shell gegeben werden. Die einzelnen Kommandos sind dann durch einen Strichpunkt zu trennen. Man hat dann eine **Kommandosequenz**. Der Strichpunkt ist die **Kommandoverkettung.**

Beispiel:

```
ls > goll.tmp; wc < goll.tmp
```

Dann wird ein Kommando nach dem anderen abgearbeitet. Der Bildschirm wird dadurch blockiert.

Möchte man

```
ls > goll.tmp; wc < goll.tmp
```

im Hintergrund ablaufen lassen, so muß die Kommandosequenz zuerst zu einer **Gruppe** zusammengefaßt werden (**Gruppierung von Kommandos**). Hierzu muß man die Kommandosequenz in runde Klammern einschließen. Durch das Anhängen eines **&**-Operators an die Gruppe wird eine neue Shell kreiert, in welcher die Kommandosequenz abläuft. Sofort nach der Eingabe des Kommandos ist die alte Shell wieder eingabebereit.

Beispiel:

```
(ls > goll.tmp; wc < goll.tmp) &
```

Hätte man

```
ls > goll.tmp; wc < goll.tmp &
```

geschrieben, so würde zuerst der Befehl

```
ls > goll.tmp
```

ausgeführt, auf dessen Beendigung gewartet, und danach durch

```
wc < goll.tmp &
```

das zweite Kommando als Hintergrundprozeß gestartet.

Beispiel für eine Gruppierung im Vordergrund

Alle Ausgaben einer Kommandosequenz **(program1; program2; program3)** sollen durch ein Filterprogramm **filter** gehen:

```
(program1; program2; program3) | filter
```

4.1.5 Allgemeine Form eines Kommandos

Ein Kommando besteht aus einem oder mehreren Wörtern. Ein Wort ist dabei eine Folge von Zeichen ohne Zwischenraum (Leerzeichen oder Tabulatorzeichen).

Die meisten UNIX-Kommandos sind mehr oder weniger einprägsame Abkürzungen der Kommandobedeutung in englischer Sprache.

Beispiele:

ls	kommt von **l**i**s**t	Erzeugen einer Liste der Dateien des aktuellen Verzeichnisses
pwd	kommt von **p**rint **w**orking **d**irectory	Anzeigen des Namens des aktuellen Verzeichnisses

Hinweis:

Für Kommandos, die Programme sind, d.h. durch Angabe des Dateinamens aufgerufen werden, kann man in einfacher Weise selbst mit Hilfe des link-Befehls **ln** neue Kommandonamen einführen, der den neuen Kommandonamen mit dem alten verbindet.

Kommandos und Programmaufrufe haben die folgende Syntax:

Kommandoname -optionen argument_1 ...argument_n

Optionen und Argumente werden auch als Parameter bezeichnet. Argumente werden oft als **normale Parameter** bezeichnet. Normale Parameter werden vom Kommando oft als Eingabedatei verarbeitet. **Optionen** sind **Zusatzangaben zur Verarbeitung**. Sie geben an, daß die Verarbeitung der normalen Parameter nicht standardmäßig ablaufen soll, sondern daß besondere Aktionen durchgeführt werden sollen. **Optionen** werden durch ein vorangestelltes - charakterisiert.

Beispiel:

```
ls -l /bin /etc
```

Hier wird das Programm **ls** aufgerufen und ihm die beiden Verzeichnisnamen **/bin** und **/etc** übergeben. Die Option **-l** besagt, daß das Kommando **ls** die Liste der Dateien in ausführlicher Form (**l** von **l**ong) ausgeben soll.

Optionale Teile eines Kommandos werden im folgenden durch geschweifte Klammern gekennzeichnet.

4.1.6 Parameterexpansion

Die Shell liest die Kommandozeile und zerlegt sie in ihre syntaktischen Bestandteile, wie z.B. den Kommandonamen, die Parameter und die Ein-/Ausgabeumlenkungs-angaben. Hierbei untersucht die Shell auch den Dateinamen auf Sonderzeichen, sogenannte **Metazeichen** oder **Dateinamen-Metazeichen**

Metazeichen werden verwendet, um Gruppen von Dateien zu spezifizieren (analog zu den Metazeichen * und ? in MS-DOS-Dateinamen).

Das Metazeichen *

Das Metazeichen * (wild card Symbol) steht in einem Dateinamen für null oder mehr beliebige Zeichen

Heißt der Dateiname beispielsweise **`f*`**, so setzt die Shell hierfür alle Dateien des aktuellen Verzeichnisses ein, die mit einem **`f`** beginnen. Das heißt, sie expandiert das Metazeichen *.

Achtung:

Steht das Zeichen * in einem Suchmuster an erster Stelle oder allein, so werden Dateinamen, die mit einem . beginnen, wie z.B. `.profile` ignoriert. Dies ist eine Ausnahme, verhindert aber den Ärger, wenn z.B. mit einem * gelöscht wird und plötzlich `.profile` fehlen würde.

Anmerkung:

Die Kommandoprozedur `.profile` liegt im Hauptkatalog des Nutzers und wird von der Shell beim Einloggen automatisch aufgerufen. In `.profile` stehen sinnvollerweise sitzungsbezogene Initialisierungskommandos wie z.B. die Definition spezieller Abkürzungen für Kommandos, die der Nutzer benützen möchte). In der C-Shell gibt es anstelle von `.profile` die Dateien `.cshrc` (C-shell recovery) und `.login`. Wurde die C-Shell durch `csh` explizit aufgerufen, so werden nur die Kommandos in `.cshrc` aufgerufen, beim Login werden `.login` und `.cshrc` ausgeführt.

Will man sich die expandierten Metazeichen eines Befehls `muster` anschauen, ohne daß der Befehl ausgeführt wird, so ruft man auf:

```
echo muster
```

Das Metazeichen ?

An der Stelle des ? darf ein beliebiges, aber nicht leeres Zeichen stehen.

Das Metazeichen [...]

Das Metazeichen [...] erlaubt es, in den eckigen Klammern mehrere zulässige Zeichen aufzuzählen. Jedes dieser Zeichen paßt dann bei einem Vergleich.

Beispiel

```
cat *[0123456789]
```

Hier werden alle Dateien ausgegeben, deren Namen als letztes Zeichen eine Ziffer hat.

Damit man nicht mühsam Buchstaben für Buchstaben eintippen muß, kann man in den eckigen Klammern auch ganze Bereiche angeben in der Form

```
[a-x]
```

Obiges Beispiel hätte man also einfacher als

```
cat *[0-9]
```

schreiben können.

Will man einen Zeichenbereich angeben, der nicht als Zeichen vorkommen soll, so erfolgt dies mit Hilfe des Ausrufezeichens in der Form

```
[!a-x]
```

Hier sind alle Zeichen im Bereich von a bis x nicht zulässig (verfügbar seit System V).

4.1.7 Quoting-Mechanismus

Zuweilen möchte man eines der Metazeichen

```
* ? [ ]
```

oder eines der Zeichen mit besonderer Bedeutung für die Shell

```
< > & ( ) | ; ^
```

< und **>**	werden von der Shell als Umleitoperatoren interpretiert.
&	wird als Hintergrundprozeß interpretiert
( )	wird als Gruppierung von Kommandos interpretiert
|	wird als Pipe interpretiert
;	wird als Sequenz von Kommandos interpretiert
^	wird beispielsweise vom Kommando `grep`, welches Stringverarbeitung macht, als Beginn einer Zeile interpretiert

an der Shell vorbeischmuggeln, da es nicht von dieser, sondern vom eigentlichen Programm interpretiert werden soll. Dies kann geschehen, indem man dem entsprechenden Zeichen einen Backslash **** voranstellt. Andere Möglichkeiten sind Apostrophzeichen und doppelte Hochkommas (siehe unten). ****, **"**, und **‘** sind sogenannte Quotes.

**Quoting mit **

Der vorangestellte Backslash bewirkt, daß das folgende Zeichen nicht von der Shell interpretiert wird.

Eine Kommandozeile wird in der Regel durch ein <RETURN> abgeschlossen. Will man ein Kommando über mehrere Zeilen schreiben, so muß man verhindern, daß die Shell das <RETURN> als Kommandoende interpretiert. Dies geschieht durch die Eingabe eines \ vor dem <RETURN>.

Beispiele:

Mit

```
rm a?
```

werden alle Dateien gelöscht, deren Name mit a beginnt und zwei Zeichen lang sind.

Mit

```
rm a\?
```

wird die Datei mit dem (unglücklichen) Namen a? gelöscht.

Quoting mit Apostrophen

Kommen in einem Namen zu viele Zeichen vor, welche die Shell nicht interpretieren soll, so ist es einfacher, den ganzen Namen in Apostrophzeichen zu setzen als die Zeichen einzeln zu quotieren.

Beispiel:

```
rm '*?*'
```

löscht die Datei mit dem (etwas merkwürdigen) Namen `*?*`.

Quoting mit doppelten Hochkommas

Durch das Anführungszeichen " werden alle Zeichen geschützt, falls aber eine Shell-Variable n in den doppelten Hochkommas eingeschlossen ist (in der Form $n, siehe Kap. 4.1.8) , so wird der Wert der Variablen n eingesetzt.

4.1.8 Shell Variable

Es gibt frei vereinbarte Shell-Variablen und vordefinierte Variablen mit fester Bedeutung. Zu den **vordefinierten Variablen mit fester Bedeutung** gehören:

$PATH

Suchpfad für Kommandos. Dies sind die Kataloge, in denen beim Aufruf eines Kommandos nach der Kommandodatei gesucht wird. Normalerweise ist dies zumindest der aktuelle Katalog des Benutzers und der Katalog `/bin` und `/usr/bin`. Die

einzelnen Kataloge sind in der Suchreihenfolge aufgeführt und durch ":" syntaktisch getrennt.

$HOME

Der Standardkatalog für das cd-Kommando. Dies ist normalerweise der Standardkatalog nach dem **login.** Wird **cd** ohne einen Parameter aufgerufen, so wird der in **$HOME** stehende Katalog zum **aktuellen Katalog**. Eine Reihe von UNIX-Dienstprogrammen suchen im **$HOME**-Katalog nach Initialisierungsdateien oder Angaben zur Standardeinstellung.

$PS1

(**P**rompt **S**tring **1**). Das erste Promptzeichen (Bereitzeichen der Shell). Der Standard ist **$**.

$PS2

(**P**rompt **S**tring **2).** Die Shell gibt dieses Prompt-Zeichen aus, wenn sie weitere Eingaben benötigt, z.B. weil das Kommando syntaktisch nicht vollständig eingegeben wurde.

$TERM

Gibt den Typ der Dialogstation an, an welcher sich der Benutzer angemeldet hat. Bildschirmorientierte Programme wie z.B. **vi**, **more** etc. benutzen den Wert dieser Variablen, um den Bildschirm mit den richtigen Steuersequenzen zu beschicken.

4.1.9 Die Datei .profile/.cshrc

Ist im Homedirectory eines Nutzers die Datei .profile vorhanden, so wird sie beim Starten jeder Shell ausgeführt, für den Fall, daß der Nutzer die Korn-Shell als Standard eingestellt hat. So können bestimmte Voreinstellungen hergestellt werden.

Für die C-Shell ist dies die Datei .cshrc.

5 Wichtige Unix-Kommandos

[Quelle: Gulbins, UNIX]

5.1 Kommandos für das File-System

5.1.1 Operationen auf Verzeichnis-Ebene

Verzeichnisse anlegen, wechseln. entfernen, Name des aktuellen Verzeichnisses anzeigen. Verzeichnisinhalte anzeigen

mkdir (make directory)

Kommando:

```
mkdir katalog
```

Bedeutung:

legt ein neues Unterverzeichnis an. Funktioniert wie der entsprechende MS-DOS-Befehl. Der Benutzer muß dabei Schreiberlaubnis auf den Dateikatalog haben, in welchem der neue Katalog eingetragen wird. Der Besitzer des neuen Katalogs ist derjenige, der ihn eingetragen hat. Deshalb muß der Super-User, wenn er ein Verzeichnis für einen neuen Benutzer eingetragen hat, ihm die Eigentümereigenschaft mit dem Kommando **chown (change owner)** übertragen**.**

Beispiel:

```
mkdir docu
```

Im aktuellen Verzeichnis wird das Unterverzeichnis *docu* angelegt.

cd (change directory)

Kommando:

```
cd katalog
```

Bedeutung:

Wechseln des aktuellen Verzeichnisses. Wird **cd** ohne Parameter aufgerufen, so wird in das home directory des Benutzers gewechselt. Das home directory hat man beim Einloggen in das System erhalten oder im Verlauf der Sitzung durch das Ändern der Shell-Variablen **$HOME** neu zugewiesen.

Mit **cd ..** wird eine Ebene höher im Katalog-Baum gestiegen. Mit **cd /** wird zur Wurzel des gesamten Dateibaumes gewechselt.

Beispiel:

Um von `/usr/users/goll/dv3` zu `/usr/users/goll/bri` zu wechseln, gibt man ein:

```
cd ../bri
```

Hinweis:

Man beachte den Leerschlag bei `cd ..` zwischen dem `cd` und den beiden Punkten

rmdir (remove directory)

Kommando:

rmdir katalog ...

Bedeutung:

re**m**ove **dir**ectory katalog. Löscht die angegebenen Kataloge. Die Kataloge müssen dazu leer sein.

Kataloge samt Inhalt können gelöscht werden durch:

rm -r katalog

pwd (print working directory)

Kommando:

pwd

Bedeutung:

der Pfad des aktuellen Dateiverzeichnisses wird ausgegeben

ls (list)

Kommando:

ls {optionen} {datei ...}

Bedeutung:

list contents of directories. Erzeugung des Inhaltsverzeichnisses des angegebenen Katalogs oder nur zu den spezifizierten Dateien. Fehlt diese Angabe, so wird das aktuelle Verzeichnis genommen. Die Optionen dürfen hintereinander geschrieben werden.

Einige wichtige Optionen zum ls-Kommando:

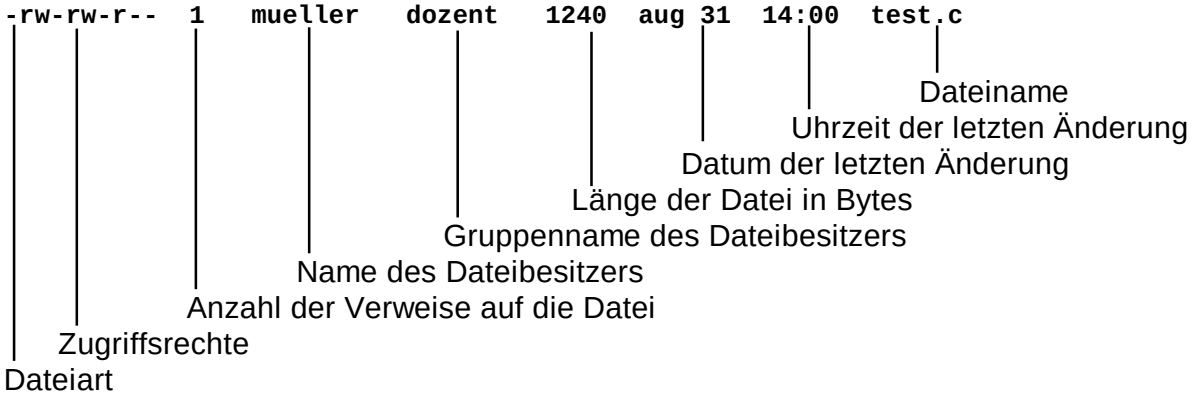
-a (**a**ll)	Es werden alle Dateien angezeigt, auch diejenigen, deren Namen mit einem **.** beginnt (das sind die sogenannten Dot-Files). Diese werden sonst nicht angezeigt
-d	Ist eine Datei ein Katalog, so soll dessen Name, jedoch nicht sein Inhaltsverzeichnis ausgegeben werden
-F	hinter dem Dateinamen wird ausgegeben - ein / bei Katalogen - ein * bei ausführbaren Dateien - ein @ bei symbolischen Links
-t (**t**ime)	die Liste wird statt nach dem alphabetischen Namen der Dateien nach dem Zeitstempel sortiert (Standard: die zuletzt modifizierte Datei zuerst)
-R (**r**ecursive)	der angegebene Katalog wird rekursiv durchsucht und sein Inhalt ausgegeben. Kommt im Katalog ein weiterer Katalog vor, so wird auch dieser durchsucht und sein Inhalt ausgegeben, usw. So kann der Inhalt eines ganzen Dateibaums ausgegeben werden
-l (**l**ong format)	Ausführliches Format mit: - Dateiart - Zugriffsrechten - Anzahl der Verweise auf die Datei - Name des Besitzers - Name der Gruppe - Größe der Datei in Bytes - Änderungsdatum und Uhrzeit (ist die Datei aus dem vorhergehenden Jahr oder älter, so wird statt der Uhrzeit das Jahr angegeben) zusätzlich zur Angabe des Dateinamens.

Weitere Optionen siehe in der Literatur

Beispiel:

das Kommando `ls - l`

erzeugt die folgende Ausgabe:



Dateiart und Zugriffsrechte zusammen heißen auch **Modus**.

Bei der **Dateiart** steht

- für eine normale Datei
d für ein Verzeichnis (**d**irectory)
b für ein block-orientiertes Gerät (**b**lock special file)
c für ein zeichen-orientiertes Gerät (**c**haracter special file)
p für einen `FIFO`-Puffer (named **p**ipe special file)
S für einen **S**ymbolic Link (bei SUN und DEC ULTRIX kein **S**, sondern ein **l**)

Bei den Zugriffsrechten steht in der Reihenfolge: Besitzer, Gruppe, Rest der Benutzer

r Leseerlaubnis (read)
w Schreibrecht (write)
x Erlaubnis, das Programm der Datei auszuführen bzw. in dem Katalog suchen zu dürfen (e**x**ecute)
s für das set-user-id-Bit oder das set-group-id-bit (siehe hierzu das ps-Kommando in Kap. 6)

Ein symbolischer Link wird hinter dem Dateinamen durch einen Verweis mit einem @ auf der SUN und auf den ULTRIX-Maschinen angezeigt.

weitere Möglichkeiten: siehe in der Literatur

5.1.2 Dateien kopieren, löschen, umbenennen. Dateien anzeigen.

cp (copy)

Kommando:

```
cp datei_1 datei_2
```

oder

```
cp datei_1 {datei_2...} katalog
```

Bedeutung:

```
cp datei_1 datei_2
```

Die Datei `datei_1` wird in eine neue Datei `datei_2` kopiert. Existiert die Datei `datei_2` bereits, so wird die alte Datei überschrieben. Nicht überschrieben wird jedoch der Modus (Dateiart und Zugriffsrechte, siehe Erläuterungen bei dem **ls**-Kommando) und der Besitzereintrag der Datei. Existiert die Datei *datei_2* jedoch noch nicht, so erhält sie die Attribute von *datei_1*.

```
cp datei_1 {datei_2...} katalog
```

Ist das letzte Argument ein Dateikatalog, so werden die davorstehenden Dateien unter dem gleichen Namen in diesen Katalog kopiert.

rm (remove)

Kommando:

```
rm {optionen} {datei ...}
```

Bedeutung:

re**m**ove the file(s). Die angegebenen Dateien werden gelöscht. Dateikataloge können mit **rmdir** gelöscht werden. Die Kataloge müssen dazu leer sein.

Kataloge samt Inhalt können gelöscht werden durch:

```
rm -r katalog
```

mv (move)

Kommando:

```
mv datei_alt datei_neu
```

oder

```
mv datei... katalog
```

Bedeutung:

```
mv datei_alt datei_neu
```

Die Datei `datei_alt` erhält den Namen `datei_neu`. Der alte Dateiname existiert danach nicht mehr. Ist eine Datei mit dem Namen `datei_neu` bereits vorhanden, so wird sie gelöscht. Liegen der Katalog, in dem der alte Namen eingetragen war, und der Katalog, in den der neue Namen eingetragen wird, auf unterschiedlichen logischen Geräten, so wird die Datei auf das Gerät des neuen Eintrags kopiert und die alte Datei gelöscht!!

Beispiele:

```
mv file datei
```

Die Datei `file` wird umbenannt in `datei`. Die alte Datei `file` wird gelöscht.

```
mv datei... katalog
```

Die Datei wird (die Dateien werden) mit ihrem alten Namen in den angegebenen Katalog eingetragen und die Referenz im alten Katalog gelöscht.

```
mv /usr/goll/*.c /usr/gruener
```

alle Dateien mit der Extension `.c` werden aus dem Katalog `/usr/goll` entfernt und in den Katalog `/usr/gruener` eingetragen.

Mit dem Befehl

```
mv verz1 verz2
```

wird `verz1` in `verz2` umbenannt. Dabei werden alle Inhalte von Verzeichnis `verz1` umgehängt in das neue Verzeichnis `verz2.`

ln (link)

Kommando:

```
ln alter_name neuer_name
```

Bedeutung:

link new **n**ame. Der Datei mit dem Namen `alter_name` wird ein weiterer Name `neuer_name` gegeben. Die Datei ist danach unter beiden Namen ansprechbar. Kataloge dürfen nur einen einzigen Namen besitzen.

Als Link wird der Eintrag in ein Verzeichnis bezeichnet, dessen Inhalt den Verweis auf die Datei *neuer_name* enthält.

Der Vorteil ist, daß Änderungen an Daten, die durch Links verschiedenen Benutzern zugänglich gemacht werden, sofort für alle zur Verfügung stehen.

Es gibt **hard-links** und **symbolic (symbolische) links.**

hard-links

?? können nur auf existierende Dateien eingetragen werden.
?? beide Katalogeinträge müssen auf dem gleichen logischen Datenträger liegen.

symbolic links

?? stellen eine spezielle Datei dar, die den Verweis enthält
?? können auf nicht existierende Dateien und Verzeichnisse eingetragen werden.
Hierbei können Filesystemgrenzen (logische Platten) überschritten werden

Wichtige Optionen

-s erzeuge einen symbolischen Link

Details zur Architektur von Soft- und Hard-Links siehe beispielsweise:Leffler, McKusick, Karels, Quaterman, Das 4.3 BSD UNIX Betriebssystem, Kap. 7, Das Dateisystem.

Beispiel

```
ln /usr/rm /usr/loesche
```

Gibt dem Kommando **rm** auch den Namen **loesche**. Nun ist das **rm-Kommando** auch mit **loesche** aufrufbar.

cat (catenate)

Kommando:

cat {optionen} {datei ...}

Bedeutung:

concatenate files. **cat** liest die angegebene(n) Datei(en) und schreibt sie auf die Standardausgabe hintereinander weg (concatenate = aneinanderhängen). Leitet man die Ausgabe durch das Anhängen von > ausgabedatei an das Kommando um, so werden die Dateien in die Datei ausgabedatei geschrieben.

Wichtige Optionen

-n	die Ausgabezeilen werden durchnumeriert
-v	nicht druckbare Zeichen werden sichtbar gemacht

more

Kommando:

more {optionen} {datei ...}

Bedeutung:

more erlaubt es, Dateien seitenweise auf dem Bildschirm auszugeben. **more** arbeitet auch als Filter, nimmt die Standardausgabe anderer Programme als seine Standard-eingabe, formatiert sie seitenweise und gibt sie auf der Standardausgabe aus. Jeweils nach einer Seite meldet sich **more** und erwartet eine Eingabe des Benutzers, um mehr (*more*) auszugeben. Die Größe einer Seite wird der **termcap**-Beschreibung der Dialogstation entnommen, kann jedoch geändert werden.

Einige wichtige Optionen

-n	n Zeilen auf einer Seite
+n	Ausgabe beginnt erst mit Zeile n der Datei
-f	überlange Zeilen werden am Bildschirmrand abgeschnitten
-s (squeeze**)**	mehrere Leerzeilen bei der Ausgabe werden zu einer Leerzeile komprimiert
-nf (n forwards)	n Bildschirmseiten nach vorne überspringen
-nb (n backwards)	n Bildschirmseiten nach hinten überspringen

weitere Möglichkeiten: siehe in der Literatur

Die Ausgabe der nächsten Information wird durch die Eingabe des Benutzers gesteuert::

<leertaste>	Ausgabe der nächsten Seite
q oder Q	Quit, d.h. **more** beenden
=	aktuelle Zeilennummer ausgeben

Die Kommandos werden sofort ausgeführt, d.h. sie müssen nicht durch <RETURN> abgeschlossen werden.

5.2 Protokoll einer Sitzung erstellen

script

Kommando:

`script` `{-a} {-q} {-s shell} { datei }`

Bedeutung :

make a **script** of the terminal session

Das `script`-Kommando bewirkt, daß alle Terminalaktionen (auch das Eingabe-Echo) protokolliert und in der angegebenen Datei gespeichert werden. Aktionen können nur protokolliert werden, wenn ihre Ausgabe nach `stdout` erfolgt. Wird kein Dateiname angegeben, so wird die Datei `typescript` erstellt und alle Aktionen dort abgespeichert.

Um sich den Inhalt dieser Datei ansehen zu können (z.B. mit `more` oder `cat`), muß `script` zuerst mit `exit` oder `CTRL D` beendet werden.

Jeder erneute Aufruf von `script` überschreibt die bestehende Protokoll-Datei.

Wichtige Optionen:

-a	Beim Kommandoaufruf wird die Protokollführung an die bestehende Datei `datei` oder `typescript` angefügt.
-q (**q**uiet)	unterdrückt die sonst üblichen Meldungen, daß `script` gestartet bzw. terminiert wurde
-s shell	erlaubt es, den Kommandointerpreter (Shell) anzugeben, der während der Sitzung benutzt werden soll, z.B. `-s /bin/sh` (Bourne-Shell)

Beispiel:

```
script goll.tmp
```

veranlaßt, daß der Text, der an der Dialogstation ein- und ausgegeben wird, in einem Protokoll in der Datei `goll.tmp` gespeichert wird.

5.3 Zugriffsrechte ändern

chmod (change mode)

Kommando:

chmod modus datei

Bedeutung:

change **mode** of file(s) to modus

ändert den Modus (d.h. die Zugriffsrechte) der angegebenen Dateien oder Kataloge. Der Modus kann entweder als Oktalzahl oder symbolisch angegeben werden.

Angabe als Oktalzahl

Die Oktalzahl ist die Addition folgender Werte:

4000	Setzt bei der Ausführung die Benutzernummer des Dateibesitzers als effektive Benutzernummer (Hinweis: siehe Kap. 6.1.4.4) ein
400	Lesezugriff für den Besitzer
200	Schreibzugriff für den Besitzer
100	Ausführungsrecht oder Katalogzugriff für den Besitzer
40	Lesezugriff für die Gruppe
20	Schreibzugriff für die Gruppe
10	Ausführungsrecht oder Katalogzugriff für die Gruppe
4	Lesezugriff für andere Benutzer
2	Schreibzugriff für andere Benutzer
1	Ausführungsrecht oder Katalogzugriff für andere Benutzer

Weitere Möglichkeiten siehe Literatur

Symbolische Angabe

{wer_hat_zugriff} zugriffs_recht {zugriffs_recht} {,...}

dabei steht für {wer_hat_zugriff}:

u	(**u**ser) für den Besitzer
g	(**g**roup) für die gleiche Gruppe
o	(**o**thers) für alle anderen oder
a	für alle = **ugo**

Fehlt die Angabe **wer_hat_zugriff**, so wird **u** (der Besitzer) angenommen. Das Zugriffsrecht wird angegeben durch **+** (füge neu hinzu) oder **-** (verbiete das Recht) oder **=** (lösche alle Rechte außer ...) gefolgt von der Art des Rechtes. Hierbei steht:

r (**r**ead) für das Recht zu lesen
w (**w**rite) für das Recht zu schreiben
x (**e**xecute) für das Recht, das Programm in der Datei ausführen bzw. in dem Katalog suchen zu dürfen
s (**s**et ID) Steht an Stelle des **x-Rechtes** beim Dateibesitzer oder Gruppe. Bei der Ausführung des Programms wird die Benutzer- oder Gruppennummer des Dateibesitzers benutzt, nicht die des Aufrufers (siehe Kap. 7.1.4.4)

Weitere Möglichkeiten siehe Literatur

Achtung: Nur der Besitzer einer Datei oder der Super-User darf den Modus ändern.

Wichtige Optionen:

-R Verzeichnisse werden rekursiv bearbeitet

Beispiele:

`chmod a+x pasc`	macht die Datei `pasc` für alle Benutzer ausführbar.
`chmod u=r geheim`	gibt nur dem Besitzer der Datei `geheim` das Leserecht. Alle anderen Benutzer können keinerlei Operationen auf der Datei ausführen (mit Ausnahme des Super-Users)
`chmod 777 oskar`	alle Benutzer dürfen die Datei `oskar` lesen, in sie schreiben und sie ausführen
`chmod a+x datei`	macht die Datei `datei` für alle Benutzer des Systems ausführbar. Ist `datei` eine Kommandoprozedur, so kann sie ohne ein vorangestelltes **sh** ebenso wie ein Programm aufgerufen werden.
`chmod ug+rw,o-rw nurwir`	setzt die Zugriffsrechte so, daß der Besitzer und die Mitglieder der gleichen Gruppe der Datei `nurwir` die Datei lesen und verändern können und alle anderen keine Zugriffsrechte auf die Datei haben.

umask (user mask)

Kommando:

umask {maske}

Bedeutung:

set **u**ser file creation **mask**. Es werden die Zugriffsrechte gesetzt, die defaultmäßig verwendet werden, wenn eine neue Datei erzeugt wird. Hinweis: Man kann diese Rechte dann später mit dem **chmod**-Befehl abändern. *maske* gibt dabei den Oktalcode der Zugriffsrechte an (siehe Befehl **chmod**). Alle in der Maske auf **1** gesetzten Bits besagen: "Dieses Recht soll **nicht** erteilt werden"

Fehlt die Angabe von *maske*, so wird der aktuell gesetzte Wert ausgegeben.

Beispiel:

`umask 007` Setzt die Zugriffsrechte (beim Anlegen neuer Dateien) so, daß andere Benutzer, die nicht zur Gruppe des Dateibesitzers gehören, diese weder lesen, noch modifizieren, noch ausführen dürfen.

4 **kein** Lesezugriff für andere Benutzer
2 **kein** Schreibzugriff für andere Benutzer
1 **kein** Ausführungsrecht oder Katalogzugriff für andere Benutzer

wc (word count)

Kommando:

```
wc {-optionen} {datei ...}
```

Bedeutung:

count **w**ords, lines and characters of a file. **wc** zählt die Anzahl der Zeilen, Worte und Zeichen von der Standardeingabe, bis die End-of-File-Marke erreicht ist (an der Tastatur eingegeben als <CTRL> + <D> im Gegensatz zu MS-DOS mit seinem <CTRL> + <Z>) und gibt ihre Zahl aus.

Wichtige Optionen
Ist keine Option angegeben, so werden alle drei Werte ausgegeben. Ist eine oder sind mehrere Optionen angegeben, so erscheinen nur die durch die Optionen angeforderten Werte. Die Optionen sind:

l (**l**ines). Es werden die Zeilen gezählt
w Es werden Worte gezählt.
c es werden Zeichen (**c**haracters) gezählt.

z.B. `who | wc -l`

zählt die Anzahl der aktuellen Sitzungen, da **who** für jeden Benutzer eine Zeile ausgibt.

Anmerkung: Das Kommando **who** gibt alle Benutzer (mit login-name, terminal-name und login-time) aus, die sich in diesem Augenblick im System befinden

diff (difference)

Kommando:

```
diff {option} datei _1 datei_2
```

Bedeutung:

differential file compare. **diff** vergleicht die beiden angegebenen Dateien und gibt auf die Standardausgabe aus, welche Zeilen wie geändert werden müssen, um mit Hilfe des **ed**-Editors datei_1 aus datei_2 zu erzeugen. Der **ed** ist ein zeilenorientierter Editor, der bei jedem UNIX vorhanden ist. Die Ausgabe hat etwa folgendes Format:

n1 **a** n3,n4 für einzufügende Zeilen
n1,n2 **d** n3 für zu löschende Zeilen
n1,n2 c n3,n4 für auszutauschende Zeilen

n1, n2, n3 sind dabei Zeilenangaben.

Die Zeilennummern vor den Kommandos beziehen sich auf die erste Datei, die nach den Kommandos auf die zweite. Auf die **ed-**Kommandos soll hier nicht eingegangen werden. Entscheidend ist, daß nach jedem dieser **ed**-Kommandos die Zeilen, die davon betroffen sind, aufgelistet werden. Vor Zeilen der ersten Datei steht <, vor denen der zweiten Datei >.

Wichtige Optionen

-b Tabulator- und Leerzeichen am Anfang der Zeile werden beim Vergleich ignoriert

BeispieL

```
diff -b prog.c.alt prog.c
```

vergleicht die Dateien `prog.c.alt` und `prog.c` und gibt in der oben beschriebenen Form die Abweichungen an. Die erzeugten Ausgaben zeigen an, welche Modifikationen in `prog.c.alt` gemacht werden müssen, damit daraus `prog.c` entsteht.

5.4 Weitere nützliche Kommandos

grep

Kommando:

grep {optionen} ausdruck {datei...}

Bedeutung:

Die **grep**-Programme **grep, egrep** und **fgrep** durchsuchen die angegebenen Dateien nach dem im Parameter `ausdruck` vorgegebenen Textmuster. Die Zeilen der Dateien, in denen das Textmuster gefunden wird, werden auf die Standardausgabe geschrieben. Wird mehr als eine Datei durchsucht, so wird der Dateiname ebenfalls angezeigt.

Bei **fgrep darf der Ausdruck nur aus mehreren durch <neue zeile> getrennten Zeichenketten bestehen. Bei** grep kann der Ausdruck sich auch aus den Metazeichen zusammensetzen, wie sie im Editor **ed** erlaubt sind. Da die Zeichen $ * [^ | ? ´ " ( ) und \ von der Shell interpretiert werden, müssen sie maskiert werden (\ oder "..." oder '...'). **egrep** akzepiert wie **grep** reguläre Ausdrücke mit folgenden Erweiterungen:

^ bedeutet: Suchmuster wird am Anfang der Zeile gesucht

Hinweis:

ein regulärer Ausdruck ist eine Folge von **normalen Zeichen** und **Metazeichen.** Ein regulärer Ausdruck ist ein Muster, mit dem die in Frage kommenden Objekte verglichen werden

Wichtige Optionen

-**c** es wird nur die Anzahl der passenden Zeilen gezählt (**c**ount)

weitere Optionen: siehe Literatur

Beispiel:

```
fgrep -c UNIX goll.txt
```

zählt, in wievielen Zeilen der Datei goll.txt das Wort UNIX vorkommt

6 Kernelfunktionen

6.1 Prozeß-Konzept

UNIX unterstützt ein Prozeßkonzept im Gegensatz zu Mach (einem verteilten UNIX), das Tasks (Prozesse) und threads unterstützt.

6.1.1 Scheduling-Mechanismen

Ein **Prozeß** ist eine selbständig ablauffähige Verwaltungseinheit des Betriebssystems, die ein sequentielles Programm - solange ihr der Prozessor zugeteilt ist - so ausführt, als würde ihr der Prozessor allein gehören.

Der Aufruf der verschiedenen Prozesse ist Sache des Schedulers (der Ablaufsteuerung) des Betriebssystems. UNIX hat zwar Prozeß-Prioritäten, dennoch ist UNIX ein **Time-Sharing Betriebssystem.** Es gibt jedoch Varianten von UNIX, die einen Realzeitbetrieb unterstützen. Bei solchen Varianten mußte jedoch der Betriebssystemkern neu entworfen werden.

Das UNIX-System verwendet beim Scheduling einen prioritätsgesteuerten Algorithmus. Es wird demjenigen rechenbereiten Prozeß als nächstem die CPU für eine Zeitscheibe zugeteilt, der die höchste Priorität besitzt. Dabei haben Prozesse, die sich im **Systemmodus** befinden, eine höhere Priorität als solche im Benutzermodus. Ein Prozeß befindet sich dann im Systemmodus, wenn er eine Systemfunktion aufgerufen hat und diese noch nicht beendet ist. Besitzen mehrere Prozesse gleichzeitig die höchste Priorität, so wählt der Kern über eine **round-robin**-Strategie denjenigen aus, der sich die längste Zeit im Zustand **ablauffähig** befand.

Um eine einseitige Vergabe der CPU-Zeit an Prozesse hoher Priorität zu verhindern - wie es bei Realzeitbetriebssystemen der Fall wäre - wird die Priorität eines Prozesses in gewissen Zeitintervallen (eine Sekunde) neu berechnet. In diese Berechnung gehen die im letzten Zeitintervall verbrauchte CPU-Zeit, die Größe des Prozesses und die Zeit, für die der Prozeß die CPU nicht erhielt, ein.

Die **aktuelle Priorität** ist die Priorität, die der Prozeß augenblicklich besitzt und die bei der nächsten CPU-Vergabe für das Scheduling verwendet wird.

Die **nice-Priorität** ist die Grundpriorität, die dem Prozeß beim Start mitgegeben wird. Mit Hilfe des **nice**-Befehls kann ein Prozeß seine Priorität verändern. Bei den als Priorität angegebenen Zahlen bedeutet ein **hoher Wert eine niedrige Priorität**. Nur der Superuser kann den **nice**-Wert verringern, um überdurchschnittlichen Service zu erlangen.

6.1.2 Speicher-Verwaltung (Memory-Management)

Ursprünglich unterstützte UNIX nur ein Swappen zur Speicher-Verwaltung (siehe Vorlesung DV3). Es wurde auf der PDP zum erstenmal implemementiert.

Bevor ein Prozeß die CPU bekommen kann, muß er im Hauptspeicher sein. Das Ein- und Auslagern vom Hauptspeicher auf den Hintergrundspeicher und zurück wird als **Swapping** bezeichnet. Der Bereich auf dem Hintergrundspeicher, auf den ausgelagert wird, heißt **Swapbereich (swap space),** das logische Gerät, auf das ausgelagert wird **swap device**. Das **swap device** ist als Geräteknoten im Katalog ***/dev*** unter dem Namen **/dev/swap** angelegt.

Das Auslagern von Prozessen geschieht durch einen eigenen Prozeß mit dem Namen **swapper** mit der Prozeßnummer **0.** Er wird beim Systemstart erzeugt und bleibt danach ständig aktiv.

Um zu verhindern, daß ein Prozeß ständig nur ein- und ausgelagert wird ohne ausreichend Zeit zu erhalten, ist der Auslagerungsmechanismus so aufgebaut, daß ein **Prozeß nur dann ausgelagert wird, wenn** er bereits eine **gewisse Zeit im Hauptspeicher war.**

Ob ein Prozeß ausgelagert ist oder sich im Hauptspeicher befindet, ist an dem Prozeßzustand (wird angezeigt beim ausführlichen **ps**-Kommando) zu erkennen. Ist ein Prozeß im Hauptspeicher, so gibt die Prozeßadresse seine Position im Hauptspeicher an, ansonsten steht hier die Adresse des Prozesses im **swap space**.

Die BSD-Version von UNIX, die für die VAX neu entworfen wurde, unterstützte als erste **on demand paging** (siehe unten). Bei einem **virtuellen System** (z.B. das BSD UNIX oder das AT&T VAX-UNIX System ab System V Version 2 Release 2) gibt es **Swappen** und **Pagen** (siehe unten).

6.1.2.1 Virtuelle (logische) und physikalische Adressen

Der **physikalische Speicher** eines Rechners setzt sich zusammen aus dem **Primär-(Haupt- bzw. Arbeits-)speicher** und dem **Sekundärspeicher (Platte, Band)**.

Ein Programm auf dem Rechner wird von der Speicherverwaltung im physikalischen Speicher (Hauptspeicher / Sekundärspeicher/ Haupt- und Sekundärspeicher) abgelegt. **Physikalische Adressen** zeigen entweder auf den Arbeitsspeicher oder auf die Platte.

Wie ein Programm im Arbeitsspeicher abgelegt wird, hängt entscheidend davon ab, ob es sich um ein 16-bit-Betriebssystem oder um ein 32-bit-Betriebssystem handelt.

Eine **logische (virtuelle) Adresse** eines Programms ist eine Adresse, die in der Maschinensprache gebildet bzw. angesprochen werden kann.

Bei einem 16-bit-Betriebssystem kann der Prozessor 2^{16} Adressen, bei einem 32-bit-Betriebssystem 2^{32} Adressen ansprechen. Wird der Arbeitsspeicher byteweise adressiert, so kann im Falle des 16-Bit-Betriebssystems ein Speicher von 2^{16} Bytes, im Falle des 32-Bit-Betriebssystems ein Speicher von 2^{32} Bytes vom Prozessor angesprochen werden.

Damit kann im Falle des 16-bit-Betriebssystems ein Prozeß (eine Task) ohne Kunstgriffe max. 2^{16} Bytes, im Falle eines 32-bit-Betriebssystems max. 2^{32} Bytes groß sein.

Da gängige Arbeitsspeicher in der Größenordnung 1-100 MB liegen, hat man als Ausgangssituation, daß im Falle eines 16-bit-Betriebssystems ein Prozeß i.a. kleiner als der Arbeitsspeicher, im Falle eines 32-Bit-Betriebssystems i.a. größer als der Arbeitsspeicher ist:

Zur Verwaltung des Arbeitsspeichers gibt es in beiden Fällen eine dafür zuständige Komponente des Betriebssystems, das Memory Management. Wenn ein Prozessor einen Prozeß abarbeitet, sorgt in beiden Fällen das Memory Management dafür, daß der Prozessor die entsprechenden physikalischen Speicherbereiche eines Prozesses anspricht, obwohl er selbst virtuelle Adressen aufruft. Dennoch ist das Memory Management bei 16-bit und echten 32-bit-Betriebssystemen grundsätzlich verschieden. Dies muß es auch sein, denn im Falle des 16-Bit-Betriebssystems paßt ein Prozeß problemlos in den Arbeitsspeicher. Er wird am Stück von der Platte geladen bzw. wieder aus dem Arbeitsspeicher auf die Platte ausgelagert (Swappen). Bei einem 32-bit-Betriebssystem kann ein Prozeß eventuell überhaupt nicht in den Arbeitsspeicher passen. Und damit ist das Problem da! Keine Angst, es wird elegant gelöst!

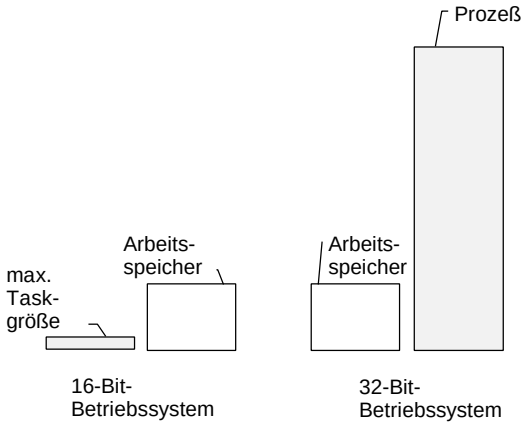


Bild 6.1.2.1 - 1 Typische Größenverhältnisse Prozeß/Arbeitsspeicher bei 16- und bei 32- Bit-Betriebssystemen

6.1.2.2 Memory Management bei einem 16-bit-Betriebssystem

Als Beispiel werde das Betriebssystem RSX-11M der PDP, das aus der Vorlesung DV3 bekannt ist, verwendet. Es macht keinen Sinn, UNIX als Beispiel zu nehmen, da UNIX als 16-Bit-Betriebssystem wie auch als 32-bit-Betriebssystem vorliegt.

Das erste UNIX lief ja gerade auf einer PDP, und das erste 32-bit-UNIX war das BSD-UNIX, welches auf einer VAX von Digital Equipment, auf der sonst das herstellerspezifische Betriebssystem VMS lief, implementiert wurde.

Betrachten wir zunächst, was auf der PDP unter RSX-11M mit einem Programm beim Compilieren und Linken passiert:

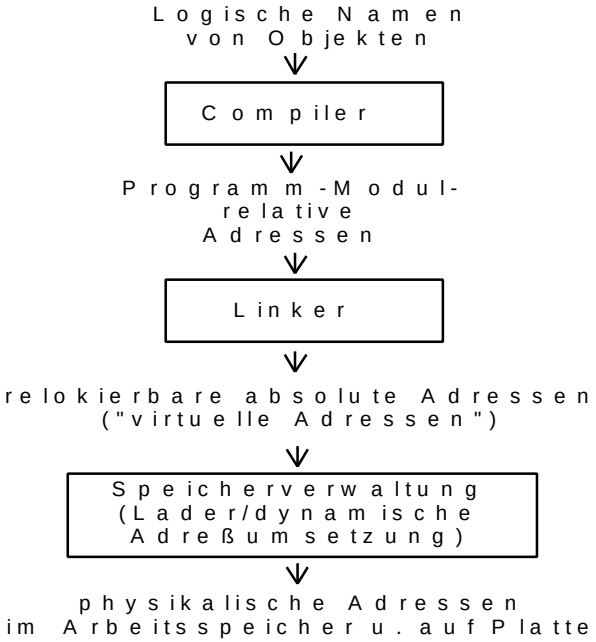


Bild 6.1.2.2 - 1: Virtuelle Adressen eines Programms. Physikalische Adressen eines Rechners

?? Ein Programm im Quellcode hat noch keine Adressen. Die Objekte einer Programmeinheit (Hauptprogramm, Unterprogramm) werden durch Namen bezeichnet.

?

?? Beim **Compilieren** werden die **Objekte an relativen Adressen innerhalb des Objekt-Files** abgelegt.

?

?? Die Bezüge zu anderen Programmeinheiten sind zunächst noch nicht gegeben. Sie werden durch den Linker hergestellt. Der **Linker fügt die einzelnen Adreßräume der Objekt-Files sowie erforderlicher Library-Files** so **zusammen**, daß sich die **Adressen nicht überlappen** und daß die Querbezüge gegeben sind. Hierzu stellt er eine Symbol Tabelle (siehe Linker Map) her, welche alle **Querbezüge (Adressen globaler Variablen, Einsprungadressen der Programmeinheiten)** enthält. Der Linker bindet die compilierten Objekt-Files, die aufgerufenen LIbrary-Files und das Laufzeitsystem des Compilers (z.B. Fehlerroutinen) zu einem **ablauffähigen Programm (executable image).** Durch den Linkvorgang wird ein **einheitlicher Adreßraum für das gesamte Programm** hergestellt.

Legt der Linker noch keine physikalische Adresse (siehe RSX-11M bei PDP), sondern nur eine **virtuelle Adresse** fest, so ist das Programm im Arbeitsspeicher verschiebbar.

?? Da das Programm auf Platte bzw. im Arbeitsspeicher liegt, ist **jeder virtuellen Adresse eines Programms** tatsächlich eine **physikalische Adresse zugeordnet**, sei es eine **Adresse im Arbeitsspeicher** oder ein **Block auf der Platte**.

Anmerkung:

Wird beim Linken bereits eine physikalische Adresse vergeben (ist bei der PDP nicht der Fall), so ist das Programm im Arbeitsspeicher nicht verschiebbar. Gewöhnliche Nutzerprogramme - nicht aber Shared Commons, die an fester Stelle des Arbeitsspeichers installiert werden, - sind bei der PDP unter RSX-11M verschiebbar.

Wird beispielsweise ein ablauffähiges Programm bei RSX-11M von der Platte in den Arbeitsspeicher geladen, so muß der Lader es an eine freie Stelle im Arbeitsspeicher legen. Dabei ist es für das Programm gleichgültig, an welcher Stelle es liegt (das **Programm ist relokierbar (verschiebbar, relocatable**), da alle Speicherplätze in seinem Programmbereich nur mit virtuellen Adressen angesprochen werden, nicht aber mit physikalischen Adressen.

Ein Programm kann auch vom Shuffler an eine andere Stelle im Speicher gelegt werden. Dies ist für seine Funktionalität unwesentlich. Dies bedeutet, daß die Programme unabhängig davon sind, an welcher Stelle im Hauptspeicher sie physikalisch liegen. Dies ist für ein effizientes multi-tasking erforderlich.

Memory Management bei der PDP

Eine virtuelle Adresse ist eine Adresse, die in der Maschinensprache gebildet bzw. angesprochen werden kann. Technisch wird ein virtuelle Adresse auf die Speicherhierarchie aus Primär- und Sekundärspeicher abgebildet (gemappt). Hierzu ist es notwendig, daß die virtuellen Adressen durch eine **Adreßübersetzung** automatisch in physikalische Adressen umgesetzt werden können.

Jeder Prozeß (Task) spannt einen eigenen virtuellen Adreßraum auf. Eine virtuelle Adresse dient dabei zur Adressierung innerhalb des Prozesses und spricht direkt keine physikalische Adresse im Arbeitsspeicher oder gar einen Block auf der Platte an.

Durch Zwischenschalten der MMU (Memory-Management-Unit) werden die 16 Adressierungsbits beispielsweise auf 18 oder 22 bits erweitert, was einem Adressierungsbereich von 256 KB bzw. 4 MB entspricht (der Speicher wird byteweise adressiert). Damit kann ein größerer physikalischer Speicher erschlossen werden. Man erreicht somit, daß im Arbeitsspeicher mehrere Tasks liegen können. Da der Speicher bei diesem Betriebssystem byteweise angesprochen wird, können 2^{16} B virtuell adressiert werden. Eine einzelne Task kann ohne Kunstgriffe (Kunstgriffe siehe unten!) dabei nur den virtuellen Adressierungsbereich von 2^{16} B ausnutzen.

also:
Adreßbereich für das Memory Management: 2^{18} bzw. 2^{22} B
virtueller Adreßbereich eines Prozesses: 2^{16} B

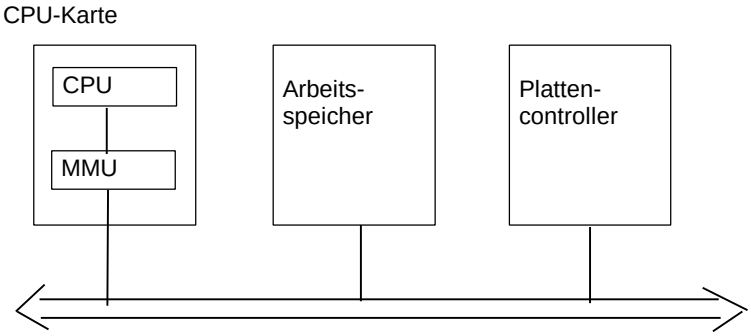


Bild 6.1.2.2 - 2: Die MMU mappt virtuelle auf physikalische Adressen.
Über den Bus gehen physikalische Adressen

Die programmgenerierten Adressen werden virtuelle Adressen genannt und bilden den virtuellen Adreßraum. Die Memory Management Unit (MMU) sendet physikalische Adressen an den Speicher. Sie macht die Zuordnung der virtuellen Adressen auf die physikalischen Adressen. Diese Zuordnung von virtuellen zu physikalischen Bereichen heißt **Mapping**

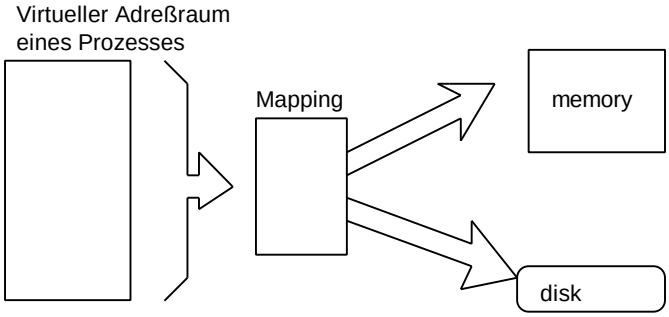


Bild 6.1.2.2 - 3: Mapping von virtuellen auf physikalische Adressen

Erinner Sie sich an das **Mapping beim IBM-PC**:

Liegt dort ein Programm im Memory, so erhält man durch das Addieren einer Start-Adresse durch die MMU auf die virtuelle Adresse eines Programms die entsprechende physikalische Adresse des Programms im Memory. Das heißt:

beim IBM-PC ist das **Mapping implementiert durch das Aufaddieren einer Start-Adresse**:

Physikalische Adresse = Start-Adresse + virtuelle Adresse (relative Adressierung)

Dabei wird eine Adresse, die auf den Beginn eines Segmentes zeigt - hier die Startadresse - Segmentbasisadresse genannt.

Bei der PDP wird das Mapping mit Hilfe einer Seitentabelle (page table) durchgeführt. Der virtuelle Speicher einer Task wird in Blöcke von 8 KB (4 kW) eingeteilt. Der physikalische Speicher wird auch in Blöcke von 4 kW eingeteilt.

Die page-table enthält für jede virtuelle Seite die zugehörige physikalische Seite:

virtuelle Seite	physik. Seite
0 KW - 4 KW	68 KW - 72 KW
4 KW - 8 KW	72 KW - 76 KW
8 KW - 12 KW	76 KW - 80 KW
12 KW - 16 KW	80 KW - 84 KW
16 KW - 20 KW	84 KW - 88 KW
20 KW - 24 KW	88 KW - 92 KW
24 KW - 28 KW	28 KW - 32 KW (Shared Common)
28 KW - 32 KW	124 KW - 128 KW (I/O-page)

Bild 6.1.2.2 - 4: Page-Table einer Task

Für eine jede Task gibt es eine solche page-table.

Die PDP hat für das Mappen wie ein 32-bit-Betriebssystem eine page-table. Allerdings gibt es bei der PDP den Mechanismus des Pagens zur Speicherverwaltung nicht. Die PDP kann nicht pagen, sie kann nur swappen.

Tricks eines 16-Betriebssystems, um die 64 KB-Grenze pro Task zu sprengen

Eine Task kann größer als 64 KB sein, wenn man

?? mit Overlays arbeitet
?? oder wenn der Benutzer die entsprechenden Direktiven der MMU benutzt, um an anderen Stellen im Arbeitsspeicher weiterzuarbeiten (Umschalten auf einen anderen 64 KB-Bereich)

Bei der PDP gibt es Overlays. Um eine Overlay-Struktur aufzubauen, muß der Benutzer die folgenden Schritte durchführen:

a) Modularisieren des Programms in Hauptprogramm und Unterprogramme, die nicht gleichzeitig im Arbeitsspeicher sein müssen
b) dem Linker eine Anweisung geben, welche Programm-Segmente gleichzeitig im Memory sein müssen und welche sich überlagern dürfen. Die einzelnen Abschnitte der Task, die sich im Rahmen einer Overlay-Struktur überlagern können, heißen Segmente.

Beispiel für eine Overlay-Struktur:

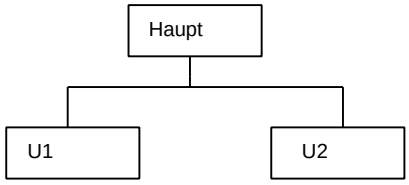


Bild 6.1.2.2 - 3: Overlay-Struktur

Diese Task besteht aus:

?? dem Root-Segment Haupt und
?? den Segmenten U1 und U2.

Dabei kann sich jedes Segment aus mehreren Object-Modulen zusammensetzen. Beispiel:

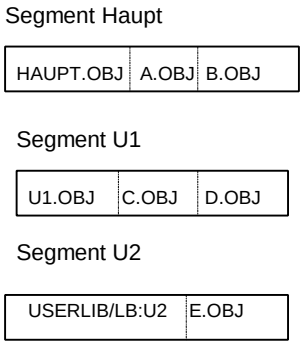


Bild 6.1.2.2 - 3: Aufbau von Segmenten

6.1.2.3 Memory Management bei einem echten 32-Bit-Betriebssystem

Als Musterbeispiel nehmen wir hier das Betriebssystem VAX/VMS.

Da ein Programm so groß sein kann, daß es nicht in den Hauptspeicher paßt, hat man ein Problem! Man löst es dadurch, daß man den virtuellen Adreßraum in virtuelle Seiten, den physikalischen Adreßraum in physikalische Seiten im Arbeitsspeicher und in Blöcke auf der Platte parzelliert, wobei eine virtuelle Seite gleich groß ist wie eine physikalische Seite oder ein Block auf Platte. Bei VMS ist eine Seite 512 B groß. Bei anderen 32-bit Betriebssystemen 512 B oder ein Mehrfaches von 512 B.

Bei VAX/VMS ist der virtuelle Adreßraum in **512-byte Abschnitte (pages)** eingeteilt. Eine **virtuelle Seite** entspricht damit genau einem **Block auf der Platte** und einem **Seitenrahmen (page frame)** im Hauptspeicher.

Das Programm wird also in **virtuelle pages (Seiten)** eingeteilt. In der Regel sind die Seiten (Code, Daten und Stack) aktuell im Hauptspeicher, mit denen das Programm gerade arbeitet. Der im Hauptspeicher direkt benutzbare Speicherplatz eines Prozesses, d.h. die Gesamtheit der Seiten, auf die der Prozeß direkt zugreifen kann, wird als sein **working set** bezeichnet.

Die Formulierung "direkt zugreifbar" deutet an, daß wir hier etwas genauer werden wollen. Es kann auch Seiten im Hauptspeicher geben, die nicht im working set sind. Hier gibt es nämlich Bereiche, die als Cache interpretiert werden können. Mit anderen Worten, eine Seite befindet sich entweder auf der Platte oder im working set oder in einem definierten Bereich des Hauptspeichers, der Seiten zwischenspeichert, ehe sie schließlich auf Platte ausgelagert werden.

Greift der Prozeß auf eine Seite zu, die nicht im Working Set steht, so entsteht ein Seitenfehler (page fault). Die Seite muß vom Betriebssystem aus dem Cache oder von der Platte beschafft werden **(Demand Paging).** Die page ist die Basiseinheit für eine Relokation von Platte ins Memory und zurück.

Eventuell wird dafür eine andere Seite auf die Platte ausgelagert**.** Der Bereich des Hintergrundspeichers, auf den die Seiten ausgelagert und bei Bedarf wieder eingelagert werden, heißt **paging area**.

Ein **page fault** ist eine Referenz auf eine Seite, die momentan nicht im working set des Prozesses ist.

Anders formuliert, kann man auch sagen, daß der **working set** eines Prozesses aus all den Seiten des virtuellen Memorys eines Prozesses besteht, die im Arbeitsspeicher sind und ohne einen **page fault** direkt zugreifbar sind (d.h. der Prozeß findet die Seite direkt).

Der working set ist eine **dynamische Größe** eines Prozesses, die bei VMS (bei anderen Betriebssystemen ist dies anders) eine untere (variable) und eine obere (feste) Schranke hat (minimale und maximale Größe). Vom System-Manager wird die obere Grenze eingestellt, die untere Grenze für einen Prozeß wird bei VMS vom Betriebssystem selbst zur Laufzeit dynamisch bestimmt. Hat es zu viele Prozesse im Speicher, so müssen die working sets kleiner werden. Dies bedeutet, daß ihre Prozesse oft auf Seiten zugreifen wollen, die nicht im working set sind. Das hat zur Konsequenz, daß zuviel gepaged wird. Wird zuviel gepaged, so lagert das Betriebssystem ganze working sets auf die Platte aus, d.h. es swappt. Damit haben die im Speicher belassenen working sets wieder die Chance, auf eine vernünftige Größe anzuwachsen bis zur ihrer oberen Grenze..

Die aktuelle Größe eines Prozesses liegt zwischen minimaler und maximaler Größe. Die aktuelle Größe bestimmt natürlich die Performance für Paging und Swapping.

Details von VAX/VMS

Die VAX (**V**IRTUAL **A**DDRESS E**X**TENSION) ist ein 32 bit-Rechner mit einem Betriebssystem (VMS=**V**IRTUAL **M**EMORY OPERATING **S**YSTEM), welches einen virtuellen Adressraum (virtuelle Adressierung) unterstützt. Der virtuelle Adreßraum beträgt 2^{32} Adressen. Die Adressierung des Speichers erfolgt byteweise, daher ist der virtuelle Speicher 4 GB groß. Der virtuelle Adreßraum wird aufgeteilt in 4 Bereiche von je 1 GB: P0, P1, S0 und S1 (siehe Bild 6.1.2.3 - 1).

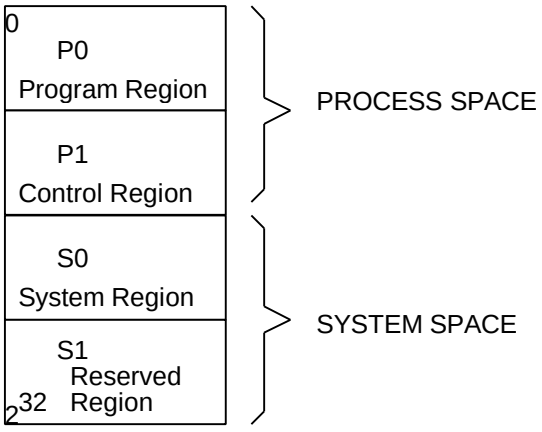


Bild 6.1.2.3 - 1 Virtueller Adressraum einer VAX

In P0 liegen die Anwender-Programme (Program Region), in P1 befinden sich Kontrollinformationen des Systems zu den Prozessen wie z.B. Stacks und die I/O-Daten der Prozesse (Control Region). In S0 liegt das Betriebsystem VMS und die Datenstrukturen, die erforderlich sind, um die Prozesse (z.B. Prozess Header) und die Seiten des virtuellen und physikalischen Speichers zu verwalten. Das bedeutet, daß der virtuelle Adreßraum eines Prozesses nicht nur das Programm umfaßt, sondern auch die Datenstrukturen zur Verwaltung des Prozesses. Solche Datenstrukturen gibt es genauso bei RSX-11M. Sie sind im Betriebssystem enthalten, welches nach dem Bootvorgang speicherresident im Memory liegt. Die Einteilung in diese 4 Bereiche rührt daher, daß die obersten 2 Bits einer virtuellen Adresse zur Adressierung dieser 4 Bereiche genommen werden

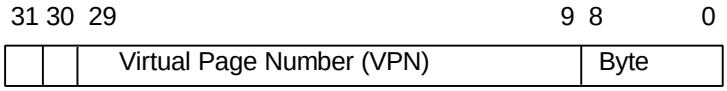


Bild 6.1.2.3 - 2: Aufbau einer virtuellen Adresse. Die obersten 2 Bits entscheiden über S0, P0 und P1

Damit umfaßt jeder dieser 4 Bereiche 1 GB Adressen. Die VPN bezeichnet eine bestimmte virtuelle Seite in diesem virtuellen Adreßraum und das Byte Offset selektiert ein Byte innerhalb dieser Seite. S1 ist ungenutzt und dient als Reserve.

Das **Mapping** wird **mit Hilfe von Seitentabellen (page tables) implementiert.** Dabei wird das Mapping für die Bereiche P0, P1 und S0 getrennt durchgeführt (eigene page tables).

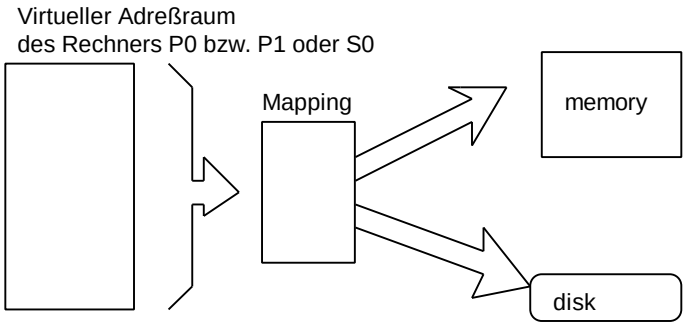


Bild 6.1.2.3 - 3: Mapping von virtuellen auf physikalische Adressen

Jede individuelle Seite hat in der Page table einen entsprechenden Eintrag. Die Funktion der Memory Management Einheit ist es, virtuelle Seiten in den physikalischen Adreßraum zu mappen und das Pagen der Seiten zwischen working set und anderen Bereichen des Hauptspeichers, die eine cache-Funktion haben, sowie dem Sekundärspeicher zu bewerkstelligen.

Die **Übersetzungstabelle für die Umsetzung virtuelle Seite <--> physikalische Seite ist die page table**. Wenn eine virtuelle Seite im Memory ist, enthält die page **table entry (PTE)** die **page frame number**, die benötigt wird, um die virtuelle Seite auf eine physikalische Seite zu mappen. Wenn sie nicht im Memory ist, enthält die page

table die Information, die erforderlich ist, um die Seite auf dem Sekundärspeicher zu lokalisieren.

Wenn ein Programm eine virtuelle Adresse referenziert, die im working set ist, dann kann unmittelbar der Zugriff erfolgen, ist die Seite nicht im working set, so resultiert ein **page fault**. Bei einem page fault muß die gesuchte Seite in den working set geladen und eine andere nicht benutzte Seite ausgelagert werden, falls die obere Grenze des Working Sets bereits erreicht ist. Ist die obere Grenze noch nicht erreicht, so können einfach die gewünschten zusätzlichen Seiten von der Platte geladen werden.

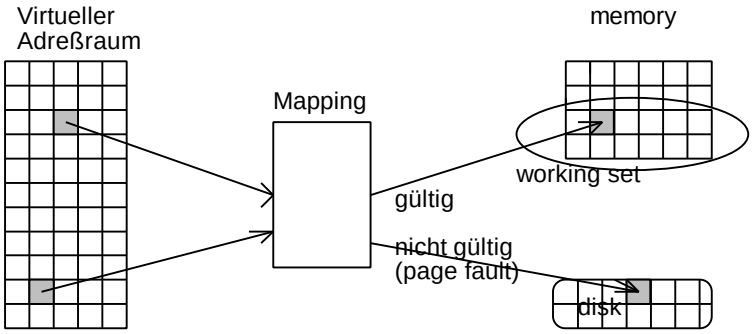


Bild 6.1.2.3 - 4: Gültige Adreßübersetzung und page fault

Eine virtuelle Adresse enthält die virtuelle Seitennummer und die Bytenummer (Byte Offset) innerhalb der Seite, entsprechend ist die physikalische Adresse aufgebaut. Der Byte Offset ist für virtuelle und physikalische Seite identisch.

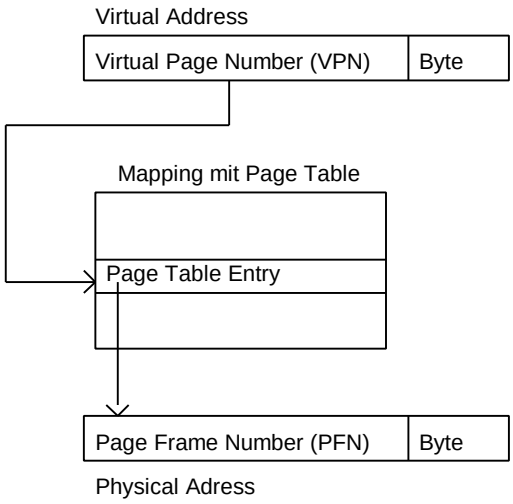


Bild 6.1.2.3 - 5: Übersetzung der VPN in die PFN bei gültiger Adreßübersetzung

Die **page table** ist eine lineare Struktur von Einträgen, geordnet nach virtuellen Seitennummern. Die Struktur ist als Array aufgebaut, so daß man mit der VPN als Index direkt auf den jeweiligen Eintrag zugreifen kann. Das Übersetzen einer virtuellen Adresse enthält also zwei Anteile:

?? den Byte Offset innerhalb der Seite
?? die virtuelle page number (VPN), die als Index in der Page Table benutzt wird, um die physikalische Seite zu lokalisieren.

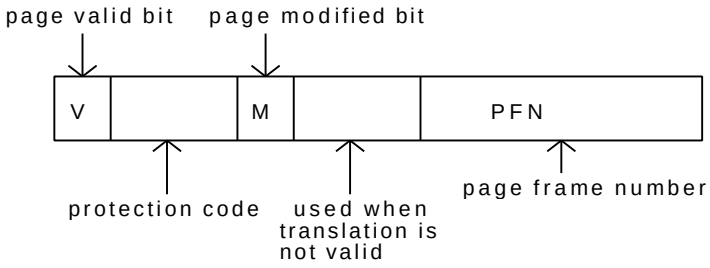


Bild 6.1.2.3 - 6: Page Table Entry

Wie Sie sehen, enthält eine Seite einen Protection Code. Die Seite ist also die Basiseinheit für den Zugriffsschutz.

Die Adreßübersetzung erfolgt durch die Rechner-HW. Tritt ein page fault bei der Adreßübersetzung auf, so muß mit Hilfe der Betriebssystem-SW ein Paging durchgeführt werden.

Zusammenfassung:

1. Virtueller Speicher ist eine Speichertechnik, die einem Prozeß die Illusion eines riesigen Arbeitsspeichers vorspiegelt, wobei in der Regel tatsächlich nur ein kleiner Teil des Prozesses im Arbeitsspeicher ist und der größere Teil auf dem Sekundärspeicher liegt.

2. Adressen werden in Seiten von 512 Bytes gruppiert. Es gibt:

   ?? virtuelle Seiten (virtual pages) im virtuellen Adreßraum
   ?? Seitenrahmen (page frames) im Memory
   ?? Blöcke auf dem Sekundärspeicher

3. Virtuelle Seiten werden auf Seitenrahmen oder Blöcke gemappt. Durch das Mapping weiß das Betriebssystem, wo eine virtuelle Seite ist.

4. Mapping beruht auf der Datenstruktur der Seitentabelle (page table). Jeder Eintrag in die page table (page table entry) beschreibt eine Seite des virtuellen Adreßraums.

5. Pages, die für den Prozeß im Memory verfügbar sind, sind im working set.

6. Das **page valid bit** in der PTE zeigt an, ob die Seite im working set ist. Ist dies nicht der Fall, so erfolgt ein page fault, wenn die Seite referenziert wird, d.h. wenn auf die Seite zugegriffen wird.

7. Die Adreßübersetzung erfolgt bei VAX/VMS hardwaremäßig. Wenn die Seite im working set ist, ist die Adreßübersetzung gültig.

8. Ist die Adresse nicht im working set, so resultiert ein **page fault**. Jetzt muß mit Hilfe eines SW-Mechanismus ein Paging durchgeführt werden, d.h. die

benötigte Seite in den working set gebracht und eine andere aus dem working set ausgelagert werden.

9. Das Format eines page table entrys ist:

   ?? **page valid bit (V):** sagt, ob die Seite im working set ist
   ?? **protection code:** sagt, wer wie auf die Seite zugreifen darf
   ?? **modified page (M):** bit sagt, ob die Seite abgeändert wurde
   ?? Ein Feld, welches benutzt wird, wenn die Seite nicht im working set ist
   ?? **page frame number (PFN):** sagt, welcher page frame die Seite enthält

10. Eine gültige Adress-Übersetzung übersetzt eine **virtual page number (VPN)** in eine **page frame number (PFN)**. Der Byte-Offset innerhalb einer Seite ist derselbe.

Größe der page table. Mehrstufige page tables

9 bits für das Byte Offset und 2 bits für die virtuellen Adreßräume P0, P1, S0 und S1 ergeben für die Darstellung der VPN 21 bits. Mit 21 bits lassen sich 2 Millionen Seiten adressieren. Bei VMS beträgt ein Eintrag für eine Seite in der page table 4 Bytes. Damit wird die page table 8 MB groß. Da VMS auch mit kleinen Speichern (wenige MB) laufen muß, ist klar, daß die page table nicht komplett im Arbeitsspeicher gehalten werden kann. Abhilfe schaffen mehrstufige Seitentabellen. VMS hat zweistufige Seitentabellen. Die Seitentabelle der ersten Stufe enthält Pointer auf die Seitentabellen der 2. Stufe. Alle nicht benötigten Seitentabellen werden nicht im Arbeitsspeicher gehalten.

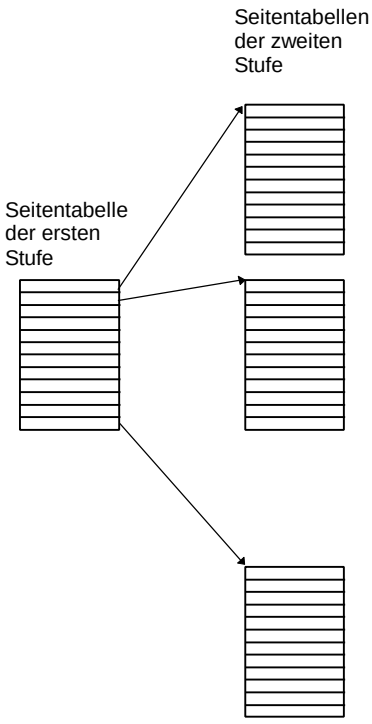


Bild 6.1.2.3 - 7 Zweistufige Seitentabellen bei VMS

Anzahl der Page Tables und der virtuellen Adreßräume

Bis jetzt waren wir vereinfachend davon ausgegangen, daß der virtuelle Adressraum von 4 G Adressen nur einmal für den Rechner vorhanden ist und daß es für jeden Bereich P0, P1, S0 und S1 jeweils 1 page table gibt. Tatsächlich ist der virtuelle Adreßraum P0, P1 pro Prozeß vorhanden. Diese Situation hatten wir ja bereits bei der PDP, wo auch jeder Prozeß seinen eigenen virtuellen Adreßraum aufspannte und wo es für jeden Prozeß eine eigene page table gab. Jedes Programm hat seinen eigene Program Region (Code und Daten) und Control Region (Stack) und sein Inhalt kann für jeden Prozeß verschieden sein. Der Adreßraum S0 und S1 ist im Adreßraum eines jeden Prozesses eingebunden und ist für alle Prozesse gleich. Jeder Prozeß hat für seinen eigenen P0- und P1-Bereich jeweils eine eigene page table.

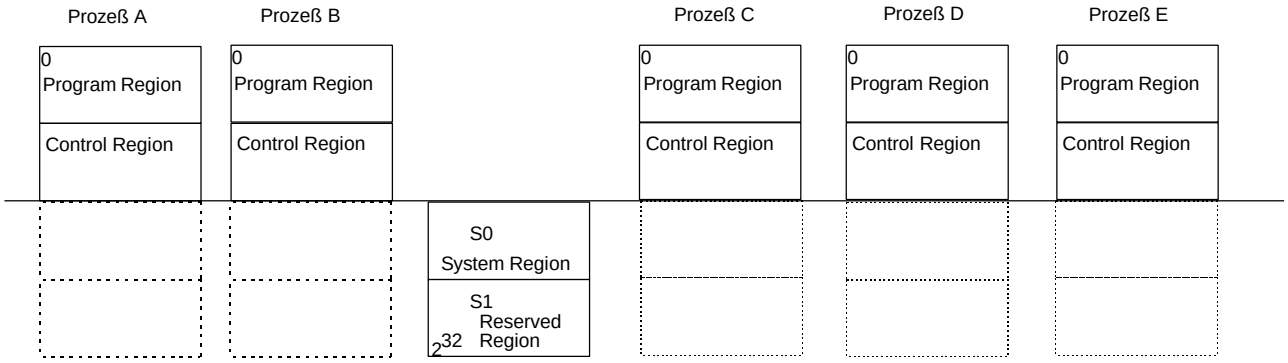


Bild 6.1.2.3 - 8 Virtuelle Adreßräume bei VMS. S0 enthält die Seitentabellen, die jedoch teilweise auch ausgelagert sein können.

6.1.3 Zustandsübergänge bei Prozessen

Zustandsübergänge von Prozessen wurden in Datenverarbeitung 3 prinzipiell besprochen. Die wichtigsten Zustände sind:

?? **running:** Der Prozeß hat die CPU
?? **suspended (= ready to run):** Der Prozeß hat alles außer der CPU
?? **waiting (= blocked):** Der Prozeß wartet auf ein Ereignis

Ein Prozeß verliert die CPU, wenn er sie freiwillig abgibt, um auf ein Ereignis zu warten oder wenn ein Ereignis eintritt, auf das ein Prozeß höherer Priorität gewartet hat.

Bild 6.1.4.1 - 1 zeigt ein Beispiel für UNIX.

6.1.4 Der Kontext eines Prozesses

[Quelle: Gulbins, UNIX
Bach, UNIX - wie funktioniert das Betriebssystem ?
Stevens, Programmieren von UNIX-Netzen]

Der Kontext eines Prozesses besteht aus dem Inhalt seines **(Benutzer-)Adreßraumes, (User Context)**, dem **Inhalt von Hardwareregistern (Register Context)** und **Datenstrukturen des Kernels (Kernel Kontext)**, die mit dem Prozeß in Beziehung stehen. D.h. zu einem Prozeß gehören das eigentliche Programm sowie die Prozeßumgebung mit

?? Speicherbelegung
?? Registerinhalte
?? geöffnete Dateien
?? aktueller Katalog
?? für den Prozeß sichtbare Umgebungsvariablen (wie z.B. $HOME und $PATH)

6.1.4.1 Adreßraum eines Prozesses

Der **virtuelle Adreßraum eines Prozesses** unterteilt sich in **Benutzeradreßraum (user context)** und **Systemadreßraum**. Der Benutzeradreßraum wird dem Prozeß vom Betriebssystem zugeteilt. Auf ihn kann der Prozeß im sogenannten **user mode** zugreifen.

Der eigene Adreßraum des Prozesses wird auch als Speicherabbild (core image, Kernspeicherbild) bezeichnet (im Andenken an den magnetischen Kernspeicher, der in längst vergangenen Tagen benutzt wurde).

Der user context besteht aus Text, Daten, Heap, Benutzerstack und gemeinsam benutztem Speicherbereich des Prozesses, die zusammen seinen virtuellen Adreßraum belegen. Auch die Teile des virtuellen Adreßraumes eines Prozesses, die

sich wegen Swapping oder Paging nicht im Hauptspeicher, sondern auf der Platte befinden, bilden einen Bestandteil des user contexts (Kontext auf Benutzerebene).

Bei einem Aufruf einer Betriebssystem-Routine (system call) wird vom **user mode** in den **kernel mode** umgeschaltet.

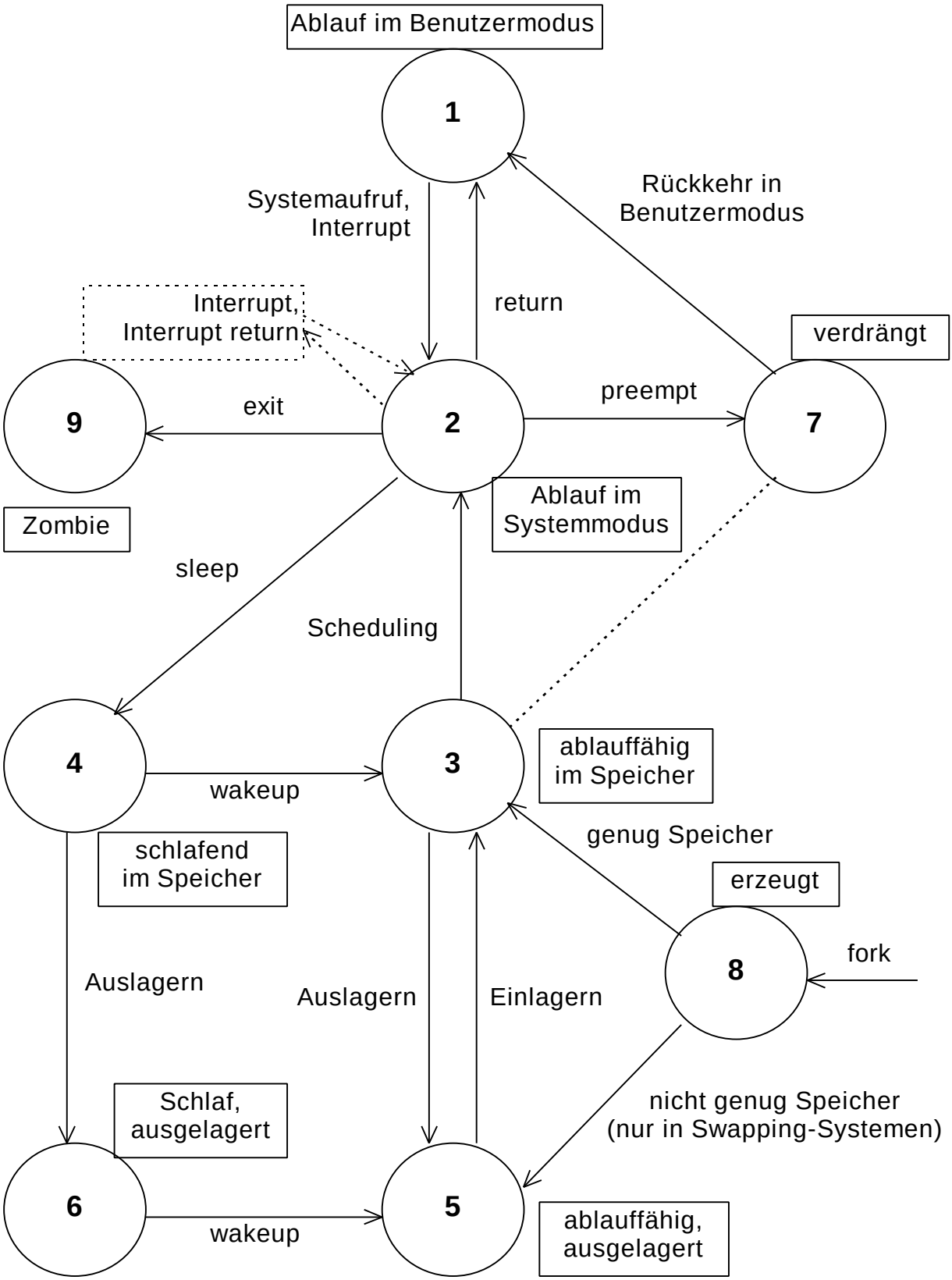


Bild 6.1.4.1 - 1: Zustandsübergänge für UNIX

Der **user context** besteht aus Segmenten.

Segmente sind Code, Daten, Heap, Stack, gemeinsamer Datenbereich:

?? **Code-Bereich (text portion).**
Enthält die von der HW auszuführenden Maschinenbefehle. Dieser Teil wird vom Betriebssystem mit dem Attribut read only versehen. Dadurch ist es möglich, daß mehrere Instanzen eines Programms mit einer einzelnen Kopie des Codes arbeiten **(shared text, reentrant).**

?? **Daten-Bereich (data portion)**
enthält die Daten des Programms. Man kann ihn in 3 Anteile aufteilen:
?? **Initialisierte, nur zu lesende Daten (initialized read-only data)**
Das sind Daten, die vom Programm mit Werten vorbelegt werden (z.B. Konstanten). Ein getrennter Bereich für die Initialisierungsdaten wird nicht von allen Betriebssystemen unterstützt.

?? **Initialisierte, zu lesende und zu beschreibende Daten (initialized read-write data)**
Daten, die vom Programm mit Werten vorbelegt werden, wobei sich während des Programmablaufs ihre Werte ändern können.

?? **Nicht initialisierte Daten**
Daten, die vom Programm nicht mit Werten vorbelegt werden. In UNIX-Systemen werden uninitialisierte Daten ***bss*** genannt. Der Vorteil dieser Datenklasse liegt darin, daß das System auf Platte keinen Speicherplatz zu reservieren braucht.

?? **Heap**
der Heap wird dazu verwendet, um sich im Programm dynamisch Speicherplatz beschaffen zu können.

?? **Stack**
der Stack wird ebenfalls während des Programmablaufs dynamisch verwendet. Auf ihm werden die **stack frames** abgelegt, die von vielen Programmiersprachen verwendet werden. Die stack frames enthalten die bei einem Funktions-/Prozeduraufruf geretteten Rücksprungadressen und lokale Daten der aufrufenden Programmeinheit, damit diese beim Ende der aufgerufenen Funktion/Prozedur wieder fortgesetzt werden kann.

Hinzu kommt noch gegebenenfalls ein

?? **gemeinsamer Datenbereich**
(falls vorhanden, in Bild 6.1.4.1 - 2 nicht eingezeichnet)

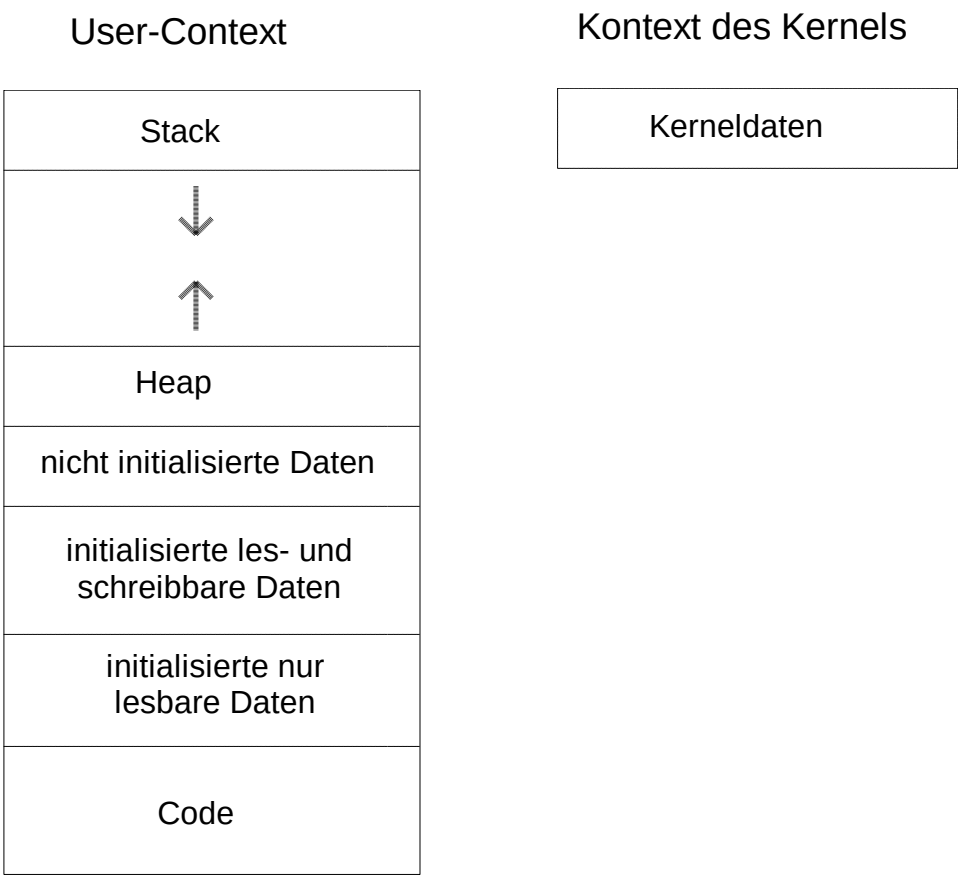


Bild 6.1.4.1 - 2: Typischer Aufbau eines Anwender-Prozesses

Zwischen Heap und Stack ist eine Lücke dargestellt. Bei vielen Betriebssystemen ist dies auch so, damit sich beide Bereiche während des Programmlaufs dynamisch vergrößern können. Dieser Bereich zwischen Heap und Stack ist genau dort, wo auch die Segmente des von mehreren Prozessen gemeinsam genutzten Speicherbereiches (**shared memory**) eingefügt werden.

6.1.4.2 Registerkontext

Der Registerkontext besteht aus den folgenden Komponenten:

Der **Programmzähler** bestimmt die Adresse der nächsten Anweisung für die Ausführung durch die CPU. Die Adresse ist eine virtuelle Adresse im Speicherbereich des Kerns oder des Benutzers.

Das **Prozessor-Statusregister (PSR)** gibt den Hardwarestatus des Rechners hinsichtlich des Prozesses an. Beispielsweise enthält das PSR Unterfelder zur Anzeige, ob ein Register übergelaufen und das Carrybit gesetzt ist, usw.

Der **Stackpointer** enthält die aktuelle Adresse des nächsten Eintrags im System- bzw. Benutzerstack, abhängig vom jeweiligen Ausführungsmodus (Hinweis: für jeden Ausführungsmodus gibt es einen eigenen Stack).

6.1.4.3 Kernel Kontext (Teil 1)

Der **Kontext des Kernels (kernel context)** bleibt während eines Prozesses unverändert. Auf ihn kann nur der Kernel zugreifen. Dieser Datenbereich enthält die Informationen, die der Kernel benötigt, um einen ablaufenden Prozeß zu überwachen und ihn zu stoppen, wenn andere Prozesse ausgeführt werden, und um ihn anschließend wieder zu starten. Typische Bestandteile dieses Datenbereichs sind Informationen über die vom Prozeß verwendeten CPU-Register und die Größe und die Anordnung der physikalischen und logischen Bereiche des Prozesses, usw. Auf diesen Bereich kann von einem ablaufenden Prozeß nicht zugegriffen werden. Ein Teil des Kernelkontextes wird in der **Prozeßtabelle (process table)** gespeichert. Diese Tabelle befindet sich im Kernel und enthält für jeden Prozeß eine Datenstruktur als Eintrag. Viele der Prozeßparameter (Prozeß-ID, User-ID usw.) werden in der Prozeßtabelle gespeichert.

Tritt ein Prozeß-Wechsel auf, so müssen die Register frei gemacht werden. Der Register-Kontext muß in den Kernel-Kontext gerettet werden. Wichtige Datenstrukturen des Kernel Kontextes sind ferner:

- **Prozeßtabelleneintrag**
  er enthält Steuerinformationen, die für den Kern immer verfügbar sein müssen
- **u-Bereich**
  er enthält Informationen, auf die nur im Kontext des Prozesses zugegriffen werden muß, d.h. die der Prozeß selbst braucht, wie z.B. welche Dateien er offen hat
- **P-Regionseinträge, Regionstabellen und Seitentabellen**
  bestimmen die Umsetzung von virtuellen in physikalische Adressen und definieren dadurch Text-, Daten-, Stack- und andere Regionen eines Prozesses (siehe 6.1.4.5).

6.1.4.4 Das ps-Kommando

Ein Teil der Kenndaten eines Prozesses, die in der Prozeßtabelle stehen, werden vom **ps-**Kommando im ausführlichen Format (Option "-l", also: ps -l) angezeigt. Hierzu gehören:

- die **Prozeßnummer des Prozesses und des Vaterprozesses** (der diesen Prozeß erzeugt hat)
- die **Benutzernummer** und **Gruppennummer**, unter der der Prozeß abläuft
- die **Priorität** des Prozesses,
- die **physikalische Adresse des Prozesses im Hauptspeicher** oder die **Blockadresse des Prozesses im Auslagerungsbereich**
- der **Prozeßzustand**
- die **Dialogstation, von der der Prozeß gestartet wurde**
- die vom Prozeß **verbrauchte Rechenzeit**

Prozeßnummer (PID)

Jeder Prozeß erhält eine eindeutige Prozeßnummer (**P**rocess **Id**entification = PID). Startet ein Benutzer einen Hintergrundprozeß durch ein angehängtes **"&"** hinter dem Kommando, so wird ihm beim erfolgreichen Start die PID des gestarteten Prozesses auf dem Bildschirm ausgegeben. Die Prozeßnummer wird benötigt, wenn man den Prozeß mit Hilfe des **kill**- Kommandos abbrechen möchte.

Prozeßnummer des Vaterprozesses (Parent **P**rocess **Id**entification = **PPID**)

Sie gibt an, von welchem Prozeß der jeweilige Prozeß gestartet wurde. Bei einem von der Dialogstation gestarteten Prozess z.B. ist dies die Prozeßnummer des auf dieser Dialogstation gestarteten Shell-Prozesses.

Benutzer- und Gruppennummer eines Prozesses

Benutzer- und Gruppennummern werden vom System dazu verwendet, um Prozesse einem Benutzer bzw. einer Benutzergruppe zuzuordnen, z.B. für Abbrechnungszwecke oder um Zugriffsrechte auf Dateien zu überprüfen. Ein Prozeß besitzt zwei Arten von Benutzer- und Gruppennummern:

?? die **effektive** Benutzer- und Gruppennummer
?? die **reale** Benutzer- und Gruppennummer

Die **reale** Benutzer- und Gruppennummer ist jeweils die Nummer, die der aufrufende Besitzer hat.

Nun gibt es beim aufgerufenen Programm zweierlei Möglichkeiten im Dateizugriffs-eintrag für das **Ausführungsrecht**:

?? "x"
?? oder "s"

Ist ein "x" eingetragen, so wird beim Programmstart die **effektive** und **reale** Benutzer- und Gruppennummer auf die jeweilige Nummer des aufrufenden Benutzers gesetzt.

Ist ein "s" (für **s**et user ID") eingetragen, so wird die Benutzernummer des Programmdateibesitzers als **effektive** Benutzernummer eingesetzt, während die **reale** Benutzernummer die des Aufrufers bleibt.

Dieser Mechanismus kommt zum Einsatz, wenn man einem unprivilegierten Benutzer für definierte Zwecke ein **privilegiertes** Programm zur Verfügung stellen möchte. So ist die Passwort-Datei geschützt (der Eigentümer ist der Systemverwalter). Der Nutzer soll aber sein Passwort ändern können. Das Passwort soll er ändern können durch das Programm **passwd.** Damit passwd die Passwort-Datei schreiben kann, muß im Programm ***/bin/passwd*** das Set-User-ID-Bit (Set-UID-Bit) und Set-Group-ID-Bit (Set-GID-Bit) gesetzt sein. Dann wird bei der Ausführung als **effektive** Nummern die Benutzer- und Gruppennummer des Super-Users eingesetzt und damit kann

schreibend auf die Datei ***/etc/passwd*** zugegriffen werden (der Super-User ist als Besitzer von ***bin/passwd*** eingetragen.

6.1.4.5 Kernel-Kontext (Teil 2): Datenstrukturen eines Prozesses im Kernel-Kontext

Bild 6.1.4.5 - 1 zeigt die wesentlichen Datenstrukturen eines Prozesses, die im Kernel Kontext geführt werden:

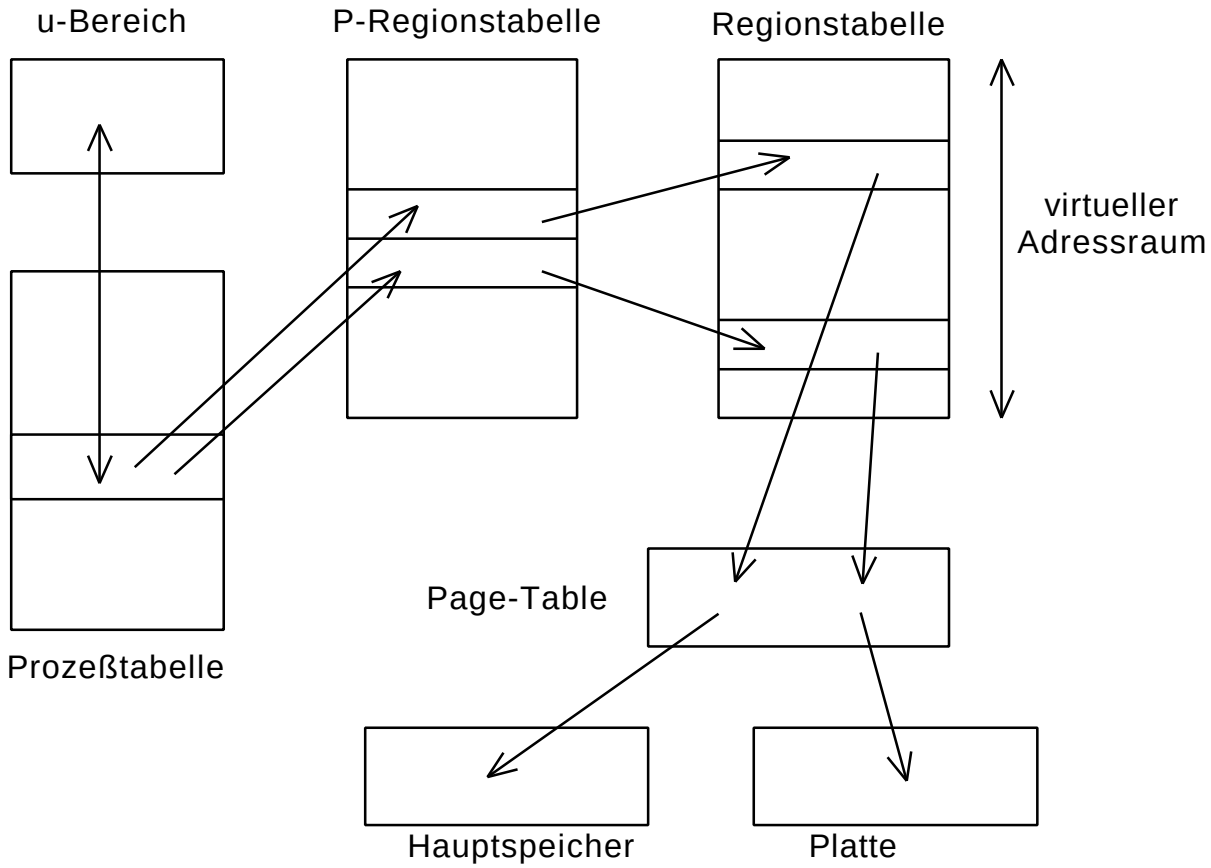


Bild 6.1.4.5 - 1: Datenstrukturen eines Prozesses

Für jeden Prozeß besteht ein Eintrag in der **Prozeßtabelle** des Systems sowie ein eigener **u-Datenbereich** (user-Datenbereich) für prozeßbezogene Daten, die nur vom System benutzt werden.

Die Prozeßtabelle zeigt auf eine **Prozeßregionstabelle (P-Regionstabelle)**. Jeder Prozeß hat eine eigene Prozeßregionstabelle. Diese besteht aus Zeigern auf die **Regionstabelle.** Der Kern unterhält eine Regionstabelle und ordnet jeder aktiven Region im System einen Eintrag in dieser Tabelle zu.

Eine Region ist ein zusammenhängender Bereich des virtuellen Adreßraumes eines Prozesses, der als Einheit gemeinsam bearbeitet oder geschützt werden kann (Regionen für Text, Daten oder Stack eines Prozesses). Die Einträge in der Regionstabelle beschreiben die Eigenschaften der Region, also ob sie Text oder Daten

enthält, ob sie privat oder gemeinsam genutzt werden kann, und wo sich die "Daten" der Region im Hauptspeicher wirklich befinden.

Das Konzept der Region ist unabhängig von der im Betriebssystem enthaltenen Speicherverwaltungsstrategie.
Aus Sicht eines Prozesses sind seine Seiten immer zusammenhängend und er adressiert auch diesen zusammenhängenden Speicherbereich (virtuellen Speicher). In Wirklichkeit können diese Seiten physikalisch im Primär- oder Sekundärspeicher liegen und müssen dort auch nicht zusammenhängend sein.

Die zusätzliche Stufe der Indirektion über die Prozeßregionstabelle zur Regionstabelle ermöglicht es, daß voneinander unabhängige Prozesse auf die gleichen Regionen zugreifen können. Beispielsweise können verschiedene Prozesse dasselbe Programm ausführen und dabei gemeinsam eine einzige Kopie der Textregion verwenden. In ähnlicher Weise können mehrere Prozesse auch bei der Verwendung eines gemeinsamen Datenbereichs zusammenarbeiten.

Macht ein Prozeß den Systemaufruf **exec**, so gibt der Prozeß seine bisher benutzten Regionen frei und erhält vom Betriebssystem neue Regionen für Text, Daten und Stack zugewiesen.

Macht ein Prozeß den Aufruf **fork**, so dupliziert das System den Adreßraum des alten Prozesses und gestattet dabei so weit wie möglich die gemeinsame Benutzung von Speicherseiten.

Macht ein Prozeß den Aufruf **exit**, so gibt das System alle vom Prozeß benutzten Regionen frei.

Prozeßtabelle

Die Prozeßtabelle enthält Steuerinformationen, die für den Kern immer verfügbar sind. Sie enthält u.a. die folgenden Felder:

?? das **Statusfeld** enthält den **Prozeßzustand**
?? Felder zum Lokalisieren des Prozesses und seines u-Bereichs (siehe Bild 6.1.4.5-1)
?? Feld mit der Größe des Prozesses (für die Zuteilung des Speichers)
?? PID
?? Ereignisdeskriptor, wenn der Prozeß sich im Zustand "schlafend" befindet
?? Ausführungszeit des Prozesses (für Scheduling-Algorithmus)
?? etc.

u-Datenbereich

Der u-Datenbereich enthält Informationen über einen Prozeß, die nur während dessen Ausführung notwendig sind. Hierzu gehören u.a.:

?? ein Pointer auf den Prozeßtabelleneintrag des gerade aktiven Prozesses
?? Parameter des aktuellen Systemaufrufs, Returncodes und Fehlercodes
?? aktueller Katalog, der zum Prozeß gehört

?? Filedeskriptoren für alle offene Dateien
?? etc.

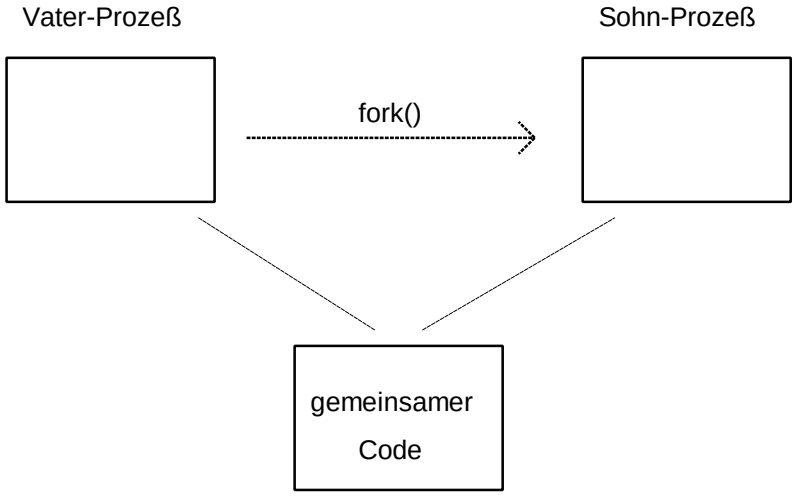
6.1.5 Steuerung von Prozessen

6.1.5.1 Kreieren von Prozessen

Die einzige Möglichkeit, unter UNIX einen neuen Prozeß zu generieren ist der ***fork()***-Aufruf (s.u.). Nach der Ausführung des ***fork()***-Aufrufs existieren zwei identische Prozesse, die vollständig unabhängig voneinander ablaufen, ein sogenannter Vater- und ein Sohn-Prozeß. Der Kernel erstellt einen Prozeß, indem er den Kontext eines anderen Prozesses dupliziert. D.h. im Prinzip: es werden alle Ressourcen des ursprünglichen Prozesses dupliziert und der Adreßraum des Prozesses kopiert (Anmerkung: es gibt ausgefeilte Techniken wie z.B. die copy-on-write-Technik, um ein Kopieren soweit wie möglich zu vermeiden, hierauf kann jedoch nicht eingegangen werden). Der neue Prozeß wird **Sohn-Prozeß** des ursprünglichen **Vater-Prozesses** genannt.

Der **fork**-Aufruf erscheint zweimal, einmal im Eltern-Prozeß, wo der Rückgabewert der Prozeß-Identifier des Sohnes ist, und zum anderen im Sohn-Prozeß, wo der Rückgabewert 0 ist. Vom Standpunkt des Anwenders aus ist der Sohn-Prozeß ein exaktes Duplikat des Eltern-Prozesses, abgesehen von zwei Werten, dem PID und dem Eltern-PID. Ein Aufruf von **fork** gibt den Sohn-PID an den Eltern- und 0 an den Sohn-Prozeß zurück. Durch Prüfen dieses Rückgabewerts kann folglich ein Programm identifizieren, ob es sich nach einem **fork** um einen Eltern- oder Sohn-Prozeß handelt.

Wenn ein neuer Prozeß erzeugt wird, geschieht dies normalerweise, um ein Programm auszuführen, das sich von dem unterscheidet, das der Eltern-Prozeß ausführt. Ein Prozeß kann mit einem neuen Programm mit Hilfe des Systemaufrufs **exec** (siehe unten) überlagert werden.



Der Systemaufruf fork()

Syntax

```
#include <unistd.h>

int fork( );
```

Der Rückgabewert der Funktion *fork()* ist
?? im Vater: die Prozeß-ID (PID) des Sohn-Prozesses
?? im Sohn: der Wert **0**
?? im Fehlerfall: der Wert **-1**

Im Fehlerfall wird die Variable **errno** (s.u.) gesetzt, auf die mit der Funktion **perror** (s.u.) zugegriffen werden kann.

Programmbeispiel

```
#include <unistd.h>

main()
{
   if(( pid = fork()) == 0 )
   {
      /* Sohn-Prozess */
   }
   else if( pid < 0 )
   {
      /* Fehlerfall */
   }
   else
   {
      /* Vater-Prozess */
   }
}
```

6.1.5.2 Weitere Systemaufrufe zur Prozeßsteuerung

Der Systemaufruf perror()

Syntax

```
void perror( char *s );
```

Die Funktion ***perror()*** gibt eine kurze Fehler-Nachricht bezugnehmend auf den letzten aufgetretenen Fehler einer Library-Routine aus. Die Fehler-Nachricht bezieht sich auf die Variable **errno** des Systems, die von den Library-Routinen im Fehlerfall gesetzt wird.
?? Wenn der Parameter **s** ein beliebiger String ist, wird dieser String gefolgt von einem Doppelpunkt und einem Blank und anschließend die Fehler-Nachricht ausgegeben.
?? Ist der Parameter **s** gleich NULL, wird nur die Fehler-Nachricht ausgegeben.

Programmbeispiel

```
#include <unistd.h>

main()
{
   int pid;

   if(( pid = fork()) == 0 )
   {
      /* Sohn-Prozess */
   }
   else if( pid < 0 )
   {
      perror("Fehler beim Kreieren eines Prozesses (fork)");
   }
}
```

Der Systemaufruf exit()

Syntax

```
#include <stdlib.h>

void exit( int status );
```

Ein Prozeß wird durch den Systemaufruf ***exit()*** beendet. Dadurch werden evt. noch geöffnete Dateien geschlossen, Shared-Memory Bereiche abgekoppelt usw.. Der ***exit()***-Befehl übergibt dem Vater-Prozeß die PID des Sohn-Prozesses und den der Funktion übergebenen Exit-Status. Dieser kann im Vater-Prozeß durch die Funktion ***wait()*** abgefragt werden (s.u.).

Im allgemeinen wird bei einem fehlerfreien Ende eines Prozesses der Aufruf ***exit(0)*** verwendet. Soll der Prozeß aber aufgrund eines Fehlers vorzeitig beendet werden, empfiehlt es sich, einen Exit-Code ungleich 0 zu verwenden (somit kann im Vaterprozeß gegebenenfalls eine entsprechende Fehlerbehandlung erfolgen).

Der Systemaufruf wait()

Syntax

```
#include <sys/wait.h>

int wait( int *status );
```

Mit der Funktion ***wait()*** wartet ein Vater-Prozeß auf das Ende eines Sohnprozesses. Der Rückgabewert der Funktion ist die PID des Sohn-Prozesses, der gerade beendet wurde. Ebenso wird in den Parameter ***status*** durch das Betriebssystem der Exit-Code des Sohn-Prozesses, der durch den Befehl ***exit()*** gegeben ist, eingetragen. Das

Statuswort (***status***) besteht aus 2 Byte: im oberen Byte steht der Exit-Code des Sohn-Prozesses und im unteren der Wert 0.
Im Fehlerfall (wenn kein Sohn-Prozeß vorhanden ist) gibt die Funktion den Rückgabewert **-1** zurück und die Variable **errno** wird gesetzt.

Programmbeispiel

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

main()
{
   int pid, sohn_pid;
   int status;

   if(( sohn_pid = fork()) == 0 )
   {
      /* Sohnprozess */
      exit( 0 );
   }

   /* Vater-Prozess */
   if(( pid = wait( &status )) < 0 )
   {
      perror("Fehler beim Warten auf Ende Sohn-Prozess");
      exit(-1);
   }
   printf("PID: %d, Status: %x\n", pid, status >> 8);
   exit(0)
}
```

Der Systemaufruf exec()

Die Funktion ***exec()*** lädt einen Programm-Code in einen laufenden Prozeß. Somit kann z.B. in einen durch ***fork()*** gestarteten Sohn-Prozeß mit Hilfe des Befehls ***exec()*** ein anderes Programm, vielmehr dessen Programm-Code, geladen und damit dieses Programm in dem Sohn-Prozeß gestartet werden.

Hierzu gibt es 5 verschiedene Routinen, die sich nur minimal in der Ausführung und den Übergabeparametern unterscheiden: ***execl, execle, execv, execlp*** und ***execvp***.

Nachfolgend wird nur die Funktion ***execlp()*** betrachtet, die die Besonderheit hat, daß die angegebenen Pfad-Angaben ($PATH) im Betriebssystem zur Suche nach dem zu startenden Programm benutzt werden.

Syntax)

```
#include <unistd.h>

int execlp( char *file, *arg0, *arg1, ..., *argn, (char *)0 );
```

Der Parameter file ist der Programmname mit zugehöriger Pfadangabe. Auf die Pfadangabe kann in den meisten Fällen verzichtet werden, da die Funktion **exec()** in dem Pfad des Environments sucht.
Das Argument **arg0** ist nach der UNIX- (C) Syntax gleich dem Programmnamen.

Wichtig:

Die Argumentenliste muß immer mit dem Wert (char *)0 oder NULL abgeschlossen werden.

Die Funktion kehrt nur im Fehlerfall zum aufrufenden Prozeß zurück. In diesem Fall ist der Rückgabewert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if( execlp( "xterm", "xterm", NULL ) < 0 )
{
   perror("Fehler beim Laden eines neuen Programms");
   exit(-1);
}
```

Abschließendes Programmbeispiel für die Prozeßsteuerung

Das folgende Programm erstellt innerhalb eines Sohnprozesses einen Xterm (ein Xterm ist ein Eingabefenster, innerhalb dessen man entweder Befehle durch Befehlseingabe ausführen kann, oder durch den Aufruf `xterm` mit dem Befehlsnamen als Parameter den Befehl sofort ausführt). Das Schließen des Xterm bedeutet gleichzeitig das Ende des Sohn-Prozesses. Der Vater-Prozeß wartet auf das Ende des Sohn-Prozesses und beendet sich automatisch.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

main( )
{
    int pid, ret_pid;
    int status;

    if(( pid = fork()) == 0 )
    {
        printf("Sohn: PID = %d\n", pid );

        if( execlp( "xterm", "xterm", NULL ) < 0 )
        {
            perror("Fehler beim Starten eines Xterm");
            exit( -1 );
        }
    }
    else
    {
        if( pid < 0 )
        {
          perror("Fehler beim Erstellen eines Sohn-Prozesses");
          exit( -1 );
        }
        else
        {
            printf("Vater: Sohn-PID = %d\n", pid );

            if(( ret_pid = wait( &status )) < 0 )
            {
                perror("Fehler beim Warten auf Ende Sohn-Prozess");
                exit( -1 );
            }

            printf("Prozess mit der PID %d beendet\n", ret_pid );
            printf("Status: %x\n", status >> 8 );
        }
    }

   exit( 0 );
}
```

6.2 Interprozeßkommunikation

6.2.1 Interprozeßkommunikation (IPC) mit unnamed und named Pipes

Zur Datenübertragung zwischen Prozessen stellt UNIX verschiedene Konzepte und Verfahren zur Verfügung, wie z.B Pipes, `FIFO`s (auch Named Pipes genannt), Nachrichtenwarteschlangen (message queues) und Gemeinschaftsspeicher (Shared Memory). Eine rechnerübergreifende Kommunikation von Prozessen wird mit der Bereitstellung von sockets unter Berkley-UNIX bzw UNIX V.4 möglich. In den folgenden Abschnitten werden die verschiedenen Verfahren mit den dazugehörigen Systemaufrufen vorgestellt.

6.2.1.1 Interprozeßkommunikation mit unnamed Pipes

Pipes (oder auch unnamed pipes) sind dabei unbenannte, speicherresidente Puffer, die nach dem "first in first out" - Prinzip behandelt werden. Das heißt, es kann nur in der Reihenfolge aus der Pipe gelesen werden, in der auch hineingeschrieben wurde. Die Systemaufrufe zum Lesen aus einer Pipe, bzw. zum Schreiben in eine Pipe sind die gleichen, die auch zum Lesen und Beschreiben normaler Dateien (read, write) verwendet werden.

Die Synchronisation zum Lesen und Beschreiben der Pipe wird dabei vom UNIX-Kern übernommen: Wird eine Pipe voll, so versetzt der Kern den schreibenden Prozeß solange in einen Wartezustand, bis ein lesender Prozeß die Pipe wieder entlastet hat. Umgekehrt wird jeder Prozeß, der aus einer leeren Pipe zu lesen versucht, solange in den Wartezustand versetzt, bis ein schreibender Prozeß die Pipe wieder mit Daten gefüllt hat.

Über den Systemaufruf:

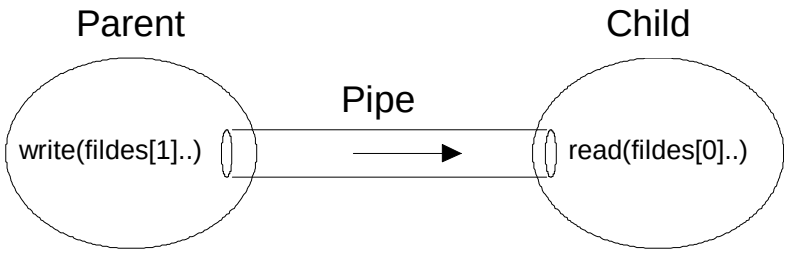
```
int pipe(int fildes[2]);
```

wird eine Pipe eingerichtet. Im Übergabeparameter `fildes` werden zwei Datei-Deskriptoren abgelegt. Der Datei-Deskriptor fildes[1] ist zu benutzen, wenn in die Pipe geschrieben wird. Zum Lesen von der Pipe ist der Deskriptor fildes[0] zu verwenden.

Da offene Pipes bei der Prozeßgenerierung mit fork() an die Child-Prozesse vererbt werden, läßt sich somit eine Verbindung zwischen Parent-Prozeß und Child-Prozeß über die Pipe realisieren. Zum Aufbau einer Einweg-Verbindung zwischen einem Parent- und einem Child-Prozeß sind zum Beispiel folgende Schritte notwendig:

- Der Parent-Prozeß erzeugt eine Pipe.
- Der Parent-Prozeß generiert mit fork() einen Child-Prozeß.
- Der Parent-Prozeß schließt die Leseseite der Pipe.
- Der Child-Prozeß schließt die Schreibseite der Pipe.
- Nun kann der Parent-Prozeß auf die Pipe schreiben und der Child-Prozeß aus der Pipe lesen.

Man erhält das folgende Bild:



Sollen die Rollen zwischen Parent- und Child-Prozeß vertauscht werden, so muß jeder Prozeß nur die jeweils andere Pipe-Seite schließen.

Die in einem Prozeß erzeugte Pipe ist jedoch nur den nachfolgenden Child-Prozessen bekannt, da diese die Pipe von ihrem Parent-Prozeß erben. Fremde Prozesse haben keinerlei Zugriff auf diese Pipe, so daß eine Interprozeßkommunikation zwischen nicht verwandten Prozessen mit Hilfe der Pipe nicht möglich ist.

Da in dem späteren Projekt die Daten-Kommunikation zwischen zwei voneinander unabhängigen Prozessen realisiert werden muß, wird an dieser Stelle nicht näher auf die Programmierung von "unnamed pipes" eingegangen. Eine Möglichkeit der Interprozeßkommunikation zwischen zwei fremden Prozessen über Pipes stellen FIFOs oder "named pipes" dar.

6.2.1.2 Interprozeßkommunikation mit FIFOs (Named Pipes)

FIFOs (oder auch Named pipes) sind ein Medium für die Kommunikation zwischen beliebigen Prozessen. In ihrem Verhalten sind sich Pipe und FIFO sehr ähnlich. Im Gegensatz zu Pipes besitzen FIFOs jedoch einen Namen. Damit ist nun auch die Kommunikation zwischen Prozessen möglich, die nicht miteinander verwandt sein müssen. Zwei beliebige Prozesse, die den FIFO-Namen kennen, können sich so miteinander unterhalten.

Erzeugt werden FIFOs mit dem Systemaufruf **mknod()** oder mit **mkfifo()**. Dadurch wird eine Datei im aktuellen Verzeichnis angelegt, die nach dem "First-In First-Out"-Prinzip (=FIFO) organisiert ist. Die beiden kooperierenden Prozesse öffnen das FIFO mit dem Systemaufruf **open()**. Mit den Systemaufrufen **read()**, **write()** können nun die beiden Prozesse entweder aus dem FIFO lesen oder in das FIFO schreiben.

Anstelle der Systemaufrufe (open, read, write) lassen sich auch die Ein/Ausgabe-Funktionen aus der Standard-C-Bibliothek, wie fopen, fread, fwrite, fprintf, fscanf ... verwenden.

Die Synchronisation der Schreib- und Lesezugriffe auf das FIFO übernimmt wie bei den Pipes wieder das Betriebssystem. Ist das FIFO leer, so werden aus dem FIFO lesende Prozesse angehalten, ist das FIFO voll, so werden entsprechend die in das FIFO schreibenden Prozesse angehalten.

Im folgenden werden die zur Kommunikation über FIFOs nötigen Systemaufrufe und Funktionen kurz beschrieben. Dabei sind nur die Befehle **mknod()** und **mkfifo()** zum Erzeugen des FIFOs wirklich neu. Alle anderen Befehle zum Öffnen, Schreiben, Lesen oder Schließen des FIFOs sind Befehle wie sie auch für normale Dateien angewendet werden.

Benötigte Systemaufrufe für FIFOs:

Erzeugen eines FIFOs:

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod( char *path, int mode, int dev );
```

Mit dem Systemaufruf **mknod()** lassen sich Verzeichnisse, Spezialdateien oder auch FIFOs kreieren. Im ersten Parameter wird der Name des FIFOs angegeben. Der Dateiname `path` kann dabei ein relativer oder auch absoluter Pfadname sein.

Der zweite Parameter `mode` beschreibt den Dateityp sowie die Zugriffsrechte der zu erzeugenden Datei. Für FIFOs muß hier der Oktal-Code: 010XXX (oder als Konstante S_IFIFO) eingetragen werden. Die unteren 9 Bits (XXX) bestimmen dabei die Zugriffsrechte des FIFO in der Reihenfolge User, Group und Other. Jeder dieser drei Stellen entspricht einer Oktalzahl, die in Binär-Schreibweise wiederum aus drei Bits besteht. Ist davon das erste Bit gesetzt, so darf die Datei gelesen (r) werden, ist das 2. Bit gesetzt, so darf die Datei beschrieben (w) werden und ist das dritte Bit gesetzt, so darf die Datei ausgeführt (x) werden.

Beispiel:

X = 100 (r- -) = 4 (oktal) : FIFO darf gelesen werden (read)
X = 010 (-w-) = 2 (oktal) : FIFO darf beschrieben werden (write)
X = 110 (rw-) = 6 (oktal) : FIFO darf gelesen und beschrieben werden.

Für alle drei Bereiche, User, Group und Others gilt dann zum Beispiel:

XXX = 111 110 100 = 7 6 4 :FIFO darf von allen nur gelesen werden, von der
U G O Gruppe gelesen und beschrieben und vom User
gelesen, beschrieben und ausgeführt werden.

Der dritte Parameter enthält die Gerätenummer. Diese Gerätenummer ist nur dann relevant, wenn eine Spezialdatei mit `mknod()` eingerichtet wird. Zum Kreieren eines FIFO wird hier der Wert 0 eingetragen.

Returnwert:
- 0: bei erfolgreichem Systemaufruf
- -1: im Fehlerfall; die Variable errno wird gesetzt

Beispiel:

```
if( mknod( "FIFO", S_IFIFO | 0666, 0 ) < 0 )
{
   perror("Fehler beim Erstellen eines FIFO");
   exit(-1);
}
```

Eine einfachere Möglichkeit zum Öffnen eines FIFOs ist die Funktion **mkfifo()**

Syntax:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo( char *path, mode_t mode );
```

Anmerkung: `mode_t` ist definiert als `unsigned short`

Mit dem Systemaufruf **mkfifo()** können ausschließlich FIFOs erstellt werden. Wie bei **mknod()** ist der Dateiname `path` (mit oder ohne Pfad) anzugeben. Als `mode` werden jetzt nur die Zugriffsrechte für die drei Bereiche User, Group und Other angegeben (siehe **mknod**).
Als Rückgabewert wird bei korrektem Ablauf **0** zurückgegeben; im Fehlerfall ist der Rückgabewert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if( mkfifo( "FIFO", 0666 ) < 0 )
{
   perror("Fehler beim Erstellen eines FIFO");
   exit(-1);
}
```

Öffnen einer Datei oder eines FIFOs:

Syntax:

```
#include <fcntl.h>

int open( char *path, int flag [, int mode] );
```

Als ersten Parameter `path` wird der Pfadname des FIFO übergeben. Dieser kann wieder ein absoluter oder relativer Pfadname enthalten. Der FIFOstatus wird in `flag` definiert. Die wichtigsten Flags sind hierbei:

flag	**Wirkung**	
O_RDONLY	"read only"	FIFO nur zum Lesen öffnen
O_WRONLY	"write only"	FIFO nur zum Schreiben öffnen

O_RDWR	"read write"	FIFO zum Lesen und Schreiben öffnen

Die einzelnen Flags können dabei über die ODER-Verknüpfung kombiniert werden.

Im letzten Parameter *mode* werden die Zugriffsrechte für User, Group und Other vergeben. Siehe auch **mknod()**. Dieser Parameter ist optional und wird im Falle eines FIFOs nicht benötigt, da die Zugriffsrechte durch den Befehl **mknod()**/**mkfifo()** bereits angegeben sind.

Returnwert : Der Systemaufruf **open()** gibt einen FIFO-Deskriptor zurück, der anstelle des fifonamens in den folgenden FIFO-Operationen verwendet wird. Im Fehlerfall wird der Wert **-1** zurückgegeben und die Variable **errno** gesetzt.

WICHTIG:

Der Befehl open() in Verbindung mit einem FIFO wird erst dann ausgeführt, wenn ein Partnerprozeß ebenfalls ein FIFO mit open() öffnet. D.h. der Prozeß wartet solange bei dem Befehl open(), bis ein anderer Prozeß diesen ebenfalls ausführt.

Beispiel:

```
if(( fd = open( "FIFO", O_RDWR )) < 0 )
{
   perror("Fehler beim Oeffnen der Pipe FIFO");
   exit(-1);
}
```

Schließen eines FIFO:

Syntax

```
#include <unistd.h>

int close( int fd );
```

Schließt das dem FIFO-Deskriptor `fd` zugeordnete FIFO. Der FIFO-Deskriptor ist dabei der Returnwert eines **open()**-Systemaufrufs.

Returnwert: -1 : Fehler, zum Beispiel unbekannter FIFO-Deskriptor; **errno** wird gesetzt
0 : Datei wurde geschlossen.

Beispiel :

```
if( close( fd ) < 0 )
{
   perror("Fehler beim Schliessen des FIFO");
   exit(-1);
}
```

Lesen aus einem FIFO:

Syntax

```
#include <unistd.h>

int read( int fd, char *buf, int nbyte );
```

Der FIFO-Deskriptor `fd` muß Returnwert eines zuvor erfolgreich ausgeführten **open()**-Aufrufs sein. Der Buffer ist die Anfangsadresse von `nbyte` zusammenhängenden Bytes. Der Wert `nbyte` ist dabei als Obergrenze zu verstehen, das heißt es werden maximal `nbyte` nach `buf` gelesen.

Beim Lesen von Pipes oder FIFOs sind folgende Punkte zu beachten:

- Wird von einer bereits geschlossenen Pipe/FIFO gelesen, so kehrt **read()** sofort mit dem Returnwert 0 zurück.
- Wird von einer leeren Pipe oder FIFO gelesen, so wird der Leseprozeß angehalten. Sobald wieder Daten zur Verfügung stehen, kann der Leseprozeß fortgesetzt werden.

Returnwert: Anzahl der tatsächlich in den Puffer eingelesenen Bytes
0 : Ende des FIFO wurde erreicht
-1 : Fehler beim Lesen aus dem FIFO und **errno** wird gesetzt

Beispiel:

```
char message[100];

if( read( fd, message, 100 ) < 0 )
{
   perror("Fehler beim Lesen aus dem FIFO");
   exit(-1);
}
printf("Nachricht: %s \n", message);
```

Schreiben in ein FIFO:

Syntax

```
#include <unistd.h>

int write( int fd, char *buf, int nbyte );
```

Der erste Parameter `fd` ist der FIFO-Deskriptor einer zuvor eröffneten Pipe. Der Buffer `buf` ist die Anfangsadresse von `nbyte` zusammenhängenden Bytes, die in das FIFO geschrieben werden.

Returnwert: Anzahl der in das FIFO geschriebenen Bytes
-1 : Fehlerfall; die Variable **errno** wird gesetzt

Beispiel:

```
char message[100];

strcpy(message,"Dies ist ein Beispielsatz");

if( write( fd, message, strlen(message)+1 ) < 0 )
{
   perror("Fehler beim Schreiben in das FIFO");
   exit(-1);
}
```

Löschen eines FIFO

Syntax

```
#include <unistd.h>

int unlink( char *path );
```

Allgemeines Löschen einer Datei.

Returnwert: 0 bei korrekter Ausführung
-1 im Fehlerfall; die Variable **errno** wird gesetzt

Beispiel:

```
if( unlink( "FIFO" ) < 0 )
{
   perror("Fehler beim Loeschen des FIFO");
   exit(-1);
}
```

Beispielprogramm für die Verwendung von FIFOs (named Pipes)

Zwei Programme: Ein Sende-Programm, welches in eine Pipe (FIFO) eine Nachricht einschreibt, und ein Empfänger-Programm, welches die Nachricht ausliest.

Wichtig (siehe Befehl open()):

Das Sende-Programm muß zuerst gestartet werden. Es bleibt aber beim Befehl open() stehen, bis das Empfänger-Programm gestartet ist.

```
/* Beispielprogramm PIPES (Sende-Programm) */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAXLEN   100

main()
{
   int fd;
   char buf[MAXLEN];

   /* Erstellen eines FIFO */
   if( mkfifo( "FIFO", 0666 ) < 0 )
   {
      perror("Fehler beim Erstellen eines FIFO");
      exit(-1);
   }

   /* Oeffnen der PIPE */
   if(( fd = open( "FIFO", O_WRONLY )) < 0 )
   {
      perror("Fehler beim Oeffnen des FIFO");
      exit(-1);
   }

   /* Einschreiben der Nachricht */
   strcpy( buf, "Versuche mit Pipes" );

   if( write( fd, buf, strlen( buf ) + 1 ) < 0 )
   {
      perror("Fehler beim Schreiben in das FIFO");
      exit(-1);
   }

   /* Schliessen des FIFO */
   if( close( fd ) < 0 )
   {
      perror("Fehler beim Schliessen des FIFO");
      exit(-1);
   }

   exit(0);
}
```

```
/* Beispielprogramm PIPES (Empfaenger-Programm) */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAXLEN   100

main()
{
   int fd;
   char buf[MAXLEN];

   /* Erstellen eines FIFO */
   if( mkfifo( "FIFO", 0666 ) < 0 )
   {
      perror("Fehler beim Erstellen eines FIFO");
      exit(-1);
   }

   /* Oeffnen der PIPE */
   if(( fd = open( "FIFO", O_RDONLY )) < 0 )
   {
      perror("Fehler beim Oeffnen des FIFO");
      exit(-1);
   }

   /* Lesen der Nachricht */
   if( read( fd, buf, MAXLEN ) < 0 )
   {
      perror("Fehler beim Lesen aus dem FIFO");
      exit(-1);
   }

   printf("Empfaenger-Programm: Nachricht aus Pipe - %s\n", buf );

   /* Schliessen des FIFO */
   if( close( fd ) < 0 )
   {
      perror("Fehler beim Schliessen des FIFO");
      exit(-1);
   }

   /* Warten, bis Sende-Prozeß beendet ist */
   sleep( 3 );

   /* Loeschen des FIFO */
   if( unlink( "FIFO" ) < 0 )
   {
      perror("Fehler beim Loeschen des FIFO");
      exit(-1);
   }

   exit(0);
}
```

6.2.2 Interprozeß-Kommunikation mit Shared Memory

Das Shared Memory ist ein Speicherbereich, auf den verschiedene Prozesse gleichzeitig Zugriff erhalten. Über ihn können sehr schnell große Datenmengen ausgetauscht werden. Man muß allerdings den konkurrierenden Zugriff der verschiedenen Prozesse koordinieren. Dazu dienen z.B. Semaphoren.

In einem Programm sind folgende Schritte notwendig, um ein Shared Memory einzurichten:

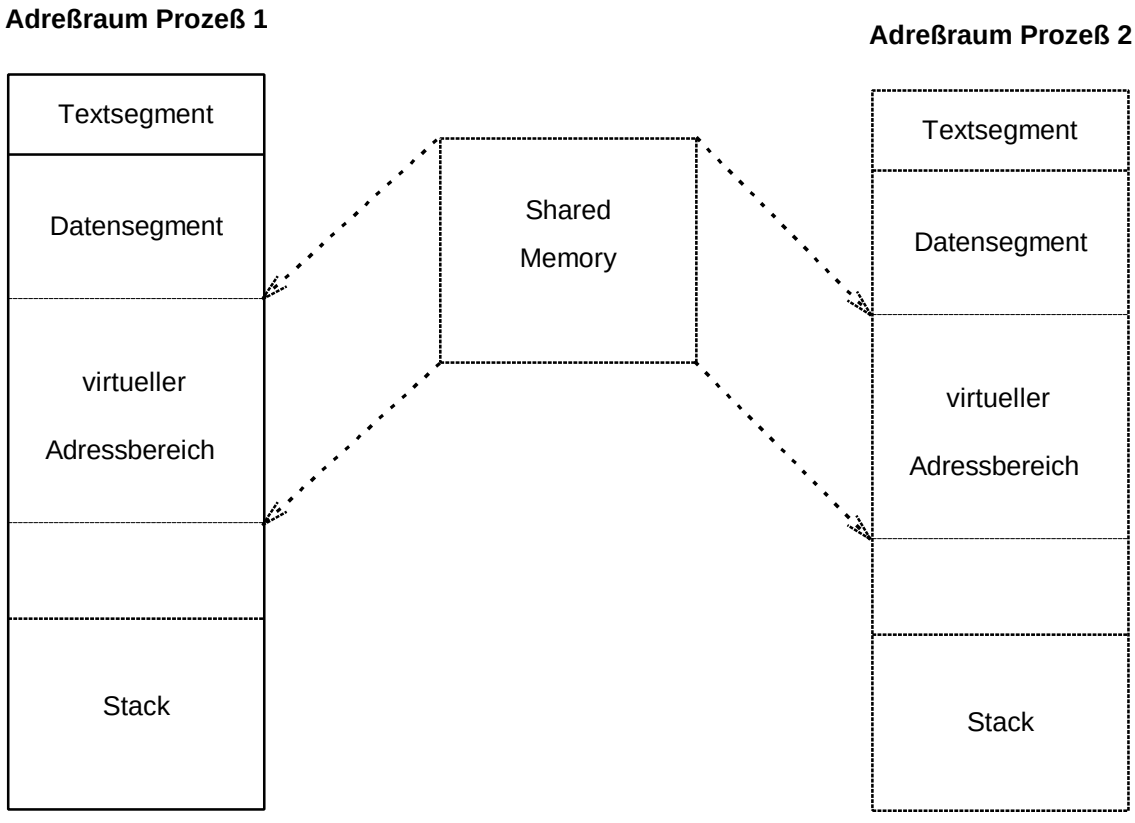
- Shared Memory vom Betriebssystem anfordern und dem Prozeß mit dem Systemaufruf **shmget()** zur Verfügung stellen

- Shared Memory mit dem Systemaufruf **shmat()** in den Adreßraum des aktuellen Prozesses einbinden (Pointer auf die Speicherfläche setzen)

- Mit verschiedenen Speicherbearbeitungsfunktionen kann dann der Datenbereich beschrieben bzw. gelesen werden (z.B. **memcpy()**)

- Mit dem Systemaufruf **shmdt()** wird das Shared Memory aus dem Adreßraum eines Prozesses wieder herausgelöst

- Mit dem Systemaufruf **shmctl()** kann ein Shared Memory Segment kontrolliert und gesteuert werden (z.B. das Abfragen des Status eines Shared Memory Segments im Rechner)



Benötigte Systemaufrufe für Shared Memory

Shared Memory anfordern und anlegen

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget( key_t key, int size, int shmflag );
```

Mit der Funktion **shmget()** wird ein Shared-Memory Segment angelegt und dem Prozeß zur Verfügung gestellt.

?? Der Parameter `key` ist der Schlüssel (Name) des Shared Memory Segments (der Typ `key_t` ist definert als `long`).

?? Mit den Parameter `size` wird die Größe des Segments in Byte bestimmt.

?? Die Zugriffsrechte auf das Shared-Memory-Segment werden mit `shmflag` angegeben (siehe hierzu auch **mknod()**). Weiterhin muß zum Erstellen eines Shared Memory Segments noch der Parameter **IPC_CREAT** angegeben werden (Oder-Verknüpfung mit den Zugriffsrechten in `shmflag`, z.B. **IPC_CREAT | 0666**). Bevor auf ein Shared-Memory Segment zugegriffen werden kann, muß es generell durch einen Prozeß kreiert werden. Der Aufruf **shmget()** mit der Option **IPC_CREAT** kann aber auch mehrmals erfolgen (durch alle Prozesse, die auf das SHM zugreifen wollen). Das Betriebssystem bemerkt dabei, daß das SHM-Segment schon angelegt ist und ignoriert die Option **IPC_CREAT**.

Bei korrektem Ablauf gibt die Funktion als Returnwert eine nicht negative ID des SHM-Segments zurück. Im Fehlerfall ist der Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if(( shmid = shmget( 5L, 200, IPC_CREAT | 0666 )) < 0 )
{
   perror("Fehler beim Anfordern eines SHM-Segments");
   exit(-1);
}
```

Ankopplung an den Adreßraum

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat( int shmid, char * shmaddr, int shmflag );
```

Mit der Funktion **shmat()** wird ein bestehendes SHM-Segment in den Adreßraum des Prozesses eingebunden (gemappt). Ab diesem Zeitpunkt kann mit Hilfe des Zeigers (Returnwert) auf das SHM-Segment zugegriffen werden.

?? Der Parameter `shmid` ist die ID des SHM-Segments aus dem Aufruf von **shmget()**.

?? Mit dem Parameter `shmaddr` kann entweder das SHM-Segment vom Betriebssystem frei zugewiesen (durch den Wert 0), oder es kann eine Adresse vorgeben werden.

?? Mit dem Parameter `shmflag` kann der Zugriff auf das SHM-Segment geregelt werden: 0 bedeutet, das auf das Segment sowohl lesend als auch schreibend zugegriffen werden kann.

Bei korrektem Ablauf gibt die Funktion als Returnwert einen Zeiger auf das SHM-Segment zurück. Im Fehlerfall ist der Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if(( shm_p = shmat( shmid, (char *)0, 0 )) < (char *)0 )
{
   perror("Fehler beim Ankoppeln des SHM-Segments");
   exit(-1);
}
```

Schreiben in das Shared Memory Segment

(als Beispiel die Funktion memcpy())

Syntax

```
#include <memory.h>

char * memcpy( char * s1, char * s2, int n );
```

Die Funktion **memcpy()** kopiert n Bytes vom Datenbereich `s2` in den Datenbereich `s1`. Als Returnwert wird `s1` zurückgegeben.

Beispiel:

```
memcpy( shm_p, text_pointer, strlen( text_pointer ) );
```

Abkopplung vom Adreßraum

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt( char *shmaddr );
```

Mit der Funktion **shmdt()** wird ein SHM-Segment aus dem Adreßraum des Prozesses abgekoppelt. D.h., der Prozeß kann nach diesem Aufruf nicht mehr auf das SHM-Segment zugreifen, da der Speicherbereich des SHM nicht mehr dem Prozeß zugeordnet wird (Programm bricht mit einer Meldung einer Segment-Verletzung ab und erzeug einen core-dump).

?? Der Parameter `shmaddr` ist der Zeiger auf das SHM-Segment aus der Funktion **shmat()**.

Bei korrektem Ablauf der Funktion ist der Returnwert **0**. Im Fehlerfall wird der Wert **-1** zurückgegeben und die Variable **errno** gesetzt.

Beispiel:

```
if( shmdt( shm_p ) < 0 )
{
   perror("Fehler beim Abkoppeln des SHM-Segments");
   exit(-1);
}
```

Shared Memory kontrollieren / steuern

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl( int shmid, int cmd, struct shmid_ds * buf );
```

Mit der Funktion **shmctl()** wird ein bestehendes SHM-Segment gesteuert.

?? Der Parameter `shmid` ist die ID des SHM-Segments aus dem Befehl **shmget()**.

?? Der Parameter `cmd` beschreibt die auszuführende Operation. Nachstehende Operationen sind u.a. möglich:

- ?? Status des Shared Memory abfragen - IPC_STAT. Die aktuellen Werte werden in die Struktur `shmid_ds` geschrieben.
- ?? Zugriffsrechte ändern - IPC_SET. Mit der Struktur `shmid_ds` können neue Zugriffsrechte zugeteilt werden.
- ?? Shared Memory löschen - IPC_RMID. Dabei ist `buf` = 0.

Die Funktion gibt bei korrektem Ablauf den Wert **0** zurück. Im Fehlerfall ist der Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if( shmctl( shmid, IPC_RMID, 0) < 0 )
{
   perror("Fehler beim Loeschen des SHM-Segments");
   exit(-1);
}
```

Programmbeispiel zu Shared Memory

```
/* Shared Memory Teil 1: Legt ein SHM an und schreibt eine Zeichenkette
(Aufrufuebergabeparameter) ein. */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY    5L
#define SHMSIZE   200

main( int argc, char *argv[] )
{
   int   shmid;
   char *shm_p;

   if( argc < 2 )
   {
      printf("Text beim Aufruf uebergeben\n");
      exit(-1);
   }

   if(( shmid = shmget( SHMKEY, SHMSIZE, IPC_CREAT | 0666 )) < 0 )
   {
      perror("Fehler beim Anlegen eines SHM-Segments");
      exit(-1);
   }

   if(( shm_p = shmat( shmid, (char *)0 , 0 )) < (char *)0 )
   {
      perror("Fehler beim Ankoppeln des SHM-Segments");
      exit(-1);
   }

   memcpy( shm_p, argv[1], strlen( argv[1] ) + 1 );

   if( shmdt( shm_p ) < 0 )
   {
      perror("Fehler beim Abkoppeln des SHM-Segments");
      exit(-1);
   }

   exit(0);
}
```

```
/* Shared Memory Teil 2: Anfordern eines SHM und Auslesen einer Zeichenkette
(anschließendes entfernen des SHM). */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 5L
#define SHMSIZE 200

main( )
{
   int   shmid;
   char *shm_p;

   if(( shmid = shmget( SHMKEY, SHMSIZE, 0444 )) < 0 )
   {
      perror("Fehler beim Anlegen eines SHM-Segments");
      exit(-1);
   }

   if(( shm_p = shmat( shmid, (char *)0 , 0 )) < (char *)0 )
   {
      perror("Fehler beim Ankoppeln des SHM-Segments");
      exit(-1);
   }

   printf("Inhalt des Shared Memory Segments: \n");
   printf("%s \n", shm_p );

   if( shmdt( shm_p ) < 0 )
   {
      perror("Fehler beim Abkoppeln des SHM-Segments");
      exit(-1);
   }

   if( shmctl( shmid, IPC_RMID, 0 ) < 0 )
   {
      perror("Fehler beim Loeschen des SHM-Segments");
      exit(-1);
   }

   exit(0);
}
```

6.2.3 Interprozeß-Kommunikation mit Semaphoren

Semaphoren werden dazu benützt, um Prozesse zu synchronisieren, die auf die gleichen Betriebsmittel zugreifen können. Betriebsmittel sind z.B. Dateien, Peripheriegeräte oder Shared Memory Bereiche. Semaphoren ermöglichen es, daß einem Prozeß der exklusive Zugriff auf ein Betriebsmittel gesichert wird und daß es nicht zu Kollisionen mit anderen Prozessen kommt (z.B. bei Schreib- / Lese-operationen).

Was sind Semaphoren ?

Eine Semaphore kann man sich einfach als eine globale Variable vorstellen, auf die zwei oder mehrere Prozesse zugreifen (d.h. lesen und schreiben) können. Vor dem Eintritt eines Prozesses in eine kritische Region (eine kritische Region ist ein Programmbereich, der einen Zugriff auf ein gemeinsam benutztes Betriebsmittel beinhaltet) muß durch das Programm geprüft werden, ob die Semaphore (globale Variable) einen Wert größer 0 hat (Wert > 0: Betriebsmittel frei zur Benutzung; Wert = 0: Betriebsmittel wird gerade benutzt). Ist der Wert der Semaphore > 0, so wird, um deutlich zu machen, daß das Betriebsmittel gesperrt ist, der Wert um 1 erniedrigt (meistens von 1 -> 0). Danach kann dann der Zugriff auf das Betriebsmittel erfolgen. Am Ende der kritischen Region wird dann der Wert der Semaphore wieder um 1 erhöht (z.B. von 0 -> 1), so daß der Zugriff auf das Betriebsmittel wieder für andere Prozesse freigegeben ist.

Trifft ein Prozeß vor dem Eintritt in eine kritische Region auf eine Semaphore mit dem Wert 0 (Betriebsmittel gesperrt), so muß dieser Prozeß solange warten, bis der Wert der Semaphore größer 0 wird, um im Programmablauf weiter fortzufahren.

Weiterhin muß beachtet werden, daß Semaphoren vor der Nutzung initialisiert werden, d.h. auf einen Anfangswert gesetzt werden müssen. Für den Fall, daß ein Betriebsmittel für einen Benutzer zur Verfügung steht, muß die Semaphore auf den Wert 1 gesetzt werden.

Um dies anhand von Beispielen deutlich zu machen, wird die Operation "Semaphore um 1 erniedrigen" **P(s)** und die Operation "Semaphore um 1 erhöhen" **V(s)** genannt (definiert laut Dijkstra: **P()** für **P**assieren und **V()** für Freigeben).

Prozeß 1
```
Anweisung
...
P(s);
/* kritische Region */
V(s);
...
end;
```

Prozeß 2
```
Anweisung;
...
P(s);
/* kritische Region */
V(s);
...
end;
```

Vor dem Eintritt in die kritsche Region rufen sowohl Prozeß 1 als auch Prozeß 2 die Funktion **P()** auf. Da die beiden Prozesse nicht gleichzeitig die kritische Region betreten können, führt nur ein Prozeß die kritische Region aus und der andere wartet bei **P()**.

Ebenso können mehrere Semaphoren zur Synchronisation benutzt werden.

Beispiel:

Der Zugriff auf einen Speicherbereich (z.B. Shared Memory) wird durch zwei Semaphoren (Write und Read) gesteuert.

Prozeß 1 (Speicher schreiben)

```
Anweisung;
...
P(write);
/* kritische Region */
/* schreiben in den Speicherbereich */
V(read);
...
end;
```

Prozeß 2 (Speicher lesen)

```
Anweisung;
...
P(read);
/* kritische Region */
/* auslesen des Speicherbereichs */
V(write);
...
end;
```

Zu Beginn müssen die Semaphoren **initialisiert** werden:

Semaphore **write** auf den Wert 1 (d.h. in den Speicherbereich darf geschrieben werden).

Semaphore **read** auf den Wert 0 (d.h. da der Speicherbereich zu Beginn leer ist , darf nicht gelesen werden).

Nun ist sichergestellt, daß zuerst Prozeß 1 durch eine Schreiboperation auf den Speicherbereich schreibt. Vor dem Schreiben (kritische Region) wird die Write-Semaphore auf 0 gesetzt. Nach dem Schreiben wird dann die Read-Semaphore auf 1 gesetzt. Somit ist ein weiteres Schreiben auf den Speicherbereich durch Prozeß 1 (oder einen anderen Prozeß verhindert). Als nächste Operation kann nur ein Auslesen des Speichers erfolgen (siehe Prozeß 2).

Semaphor-Systemaufrufe unter UNIX

Um unter dem Betriebssystem UNIX auf Semaphoren zugreifen zu können, sind folgende Schritte notwendig:

- Die Semaphore muß vom Betriebssystem angefordert und dem Prozeß verfügbar gemacht werden. Hierzu dient die Funktion ***semget()***.
- Die Initialisierung und Steuerung der Semaphore (Semaphoren) erfolgt mit dem Systemaufruf ***semctl()***
- Die Funktionen P() und V() werden durch den Systemaufruf ***semop()*** realisiert.

Anforderung einer Semaphore(n)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget( key_t key, int nsems, int semflag );
```

Mit der Funktion **semget()** wird eine Semaphore bzw. ein Semaphorensatz angefordert und ggf. erstellt.

?? Der Parameter `key` ist die Nummer (Name) des Semahorensatzes im System (der Type `key_t` ist definiert als `long`).

?? Mit dem Parameter `nsems` gibt man die Anzahl der Semaphoren im Semaphorensatz an.

?? Der Parameter `semflag` setzt die Zugriffsrechte des Semaphorensatzes. Die Zugriffsrechte auf die Semaphoren sind z.B. 0444 (lesen) oder 0666 (lesen und schreiben). Um eine Semaphore zu Erstellen muß zusätzlich (als ODER-Verknüpfung) der Parameter **IPC_CREAT** angegeben werden. Hierbei kann der Aufruf (siehe auch **shmget()**) mit **IPC_CREAT** mehrmals erfolgen. Existiert der Semaphorensatz bereits, ignoriert das Betriebssystem dieses Flag.

Bei korrektem Ablauf gibt die Funktion als Returnwert eine nicht negative ID des Semaphorensatzes zurück. Im Fehlerfall ist der Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if(( semid =semget( 5L, 1, IPC_CREAT | 0666 )) < 0 )
{
   perror("Fehler beim Anfordern einer Semaphore");
   exit(-1);
}
```

Steuerung von Semaphoren (z.B. Initialisierung)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl( int semid, int semnum, int cmd, union semun arg );

union semun
{
   int val;
   struct semid_ds *buf;
   ushort *array;
};
```

Mit der Funktion **semctl()** kann eine Semaphore bzw. ein Semaphoren-Satz gesteuert werden. Es stehen sowohl Kommandos zum **Auslesen** und **Setzen** des Semaphorenwertes als auch Kommandos zum **Auslesen des Status** und zum **Löschen** des Semaphorensatzes zur Verfügung.

?? Der Parameter `semid` ist die ID des Semaphoren-Satzes (vom Befehl **semget()**).

?? Mit dem Parameter `semnum` muß die Nummer der zu behandelnden Semaphore aus dem Semaphorensatz angegeben werden (die Nummerierung der Semaphoren eines Satzes beginnt immer mit 0).

?? Der Parameter `cmd` ist das anzuwendende Steuerkommando; folgende Kommandos sind u.a. möglich:

- ?? SETVAL — Initialisierung der Semaphore auf einen Anfangswert
- ?? GETVAL — liefert aktuellen Wert der Semaphore
- ?? GETPID — liefert PID des letzten Prozesses, der auf die Semaphore zugegriffen hat
- ?? SETALL — Initialisierung aller Semaphoren im Satz
- ?? GETALL — liefert Werte aller Semaphoren im Satz
- ?? IPC_STAT — liefert den Status einer Semaphore
- ?? IPC_SET — setzt den Status einer Semaphore
- ?? IPC_RMID — löscht den Semaphorensatz

In den nachfolgenden Beispielen werden nur die Kommandos SETVAL und IPC_RMID verwendet. Alle weiteren Kommandos werden für einfache Semaphor-Aktionen nicht benötigt.

?? Der Parameter `arg` ist eine Union, die von den einzelnen Kommandos benötigt wird. Diese Union ist bei den meisten Compilern nicht definiert, und muß deshalb in der obigen Form angegeben werden. Für das Kommando SETVAL wird z.B. die Variable `arg.val` benötigt, in der der Wert der Semaphore angegeben werden muß, auf die diese gesetzt werden soll.

Als Returnwert liefert die Funktion **semctl()** für die Aktionen GETVAL und GETPID z.B. als Returnwert das Ergebnis der Aktion zurück. Sonst ist der Rückgabewert bei korrektem Ablauf **0**. Im Fehlerfall liefert die Funktion den Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiele:

```
/* Initialisierung einer Semaphore */
union semun arg;

arg.val = 1;
if( semctl( semid, 0, SETVAL, arg ) < 0 )
{
   perror("Fehler bei der Initialiserung der Semaphore");
   exit(-1);
}

/* Loeschen eines Semaphorensatzes */
if( semctl( semid, 0, IPC_RMID, arg ) < 0 )
{
   perror("Fehler beim Loeschen der Semaphore");
   exit(-1);
}
```

Das Setzen und Rücksetzen der Semaphoren

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop( int semid, struct sembuf **sops, unsigned nsops );

struct sembuf
{
  short  sem_num;    /* Nummer des Semaphors im Satz */
  int    sem_op;     /* Operator */
  ushort sem_flg;    /* Flag */
};
```

Mit der Funktion **semop()** wird das Setzen und Rücksetzen der Semaphoren ermöglicht.

?? Der Parameter `semid` ist die ID des Semaphoren-Satzes (aus Befehl **semget()**).

?? Der Parameter `sops` ist ein Zeiger auf ein Array. Die einzelnen Teile der Struktur müssen je nach Aktion gesetzt werden (siehe Beispiel).

?? Mit dem Parameter `nsops` wird die Anzahl der Strukturen im Array `sops` angegeben, d.h., auf wieviel Semaphoren des Satzes eine Aktion angewendet werden soll.

Der Returnwert der Funktion ist bei korrektem Ablauf immer **0**. Im Fehlerfall gibt die Funktion den Wert **-1** zurück und die Variable **errno** wird gesetzt.

Um die P() und V()-Operationen ausführen zu können, muß das Array `sops` entsprechend vor dem Aufruf gesetzt werden.

P()-Operation: Die Variable `sem_op` muß auf den Wert -1 gesetzt werden.
V()-Operation: Die Variable `sem_op` muß auf den Wert +1 gesetzt werden.

Weiterhin muß die Nummer der entsprechenden Semaphore im Satz in `sem_num` angegeben werden.

Mit der Variablen `sem_flg` in `sops` kann evt. noch dem Betriebssystem mitgeteilt werden, ob die Funktion auf das Rücksetzen der Semaphore warten (`sem_flg` = 0) oder die Funktion abbrechen soll (Returnwert = -1), wenn die Semaphore 0 ist (`sem_flg` = IPC_NOWAIT).

Beispiele:

```
/* Setzen einer Semaphore; P-Operation */
struct sembuf ops[1];

ops[0].sem_num = 0;
ops[0].sem_flg = 0;
ops[0].sem_op  = -1;

if( semop( semid, ops, 1 ) < 0 )
{
   perror("Fehler beim Setzen der Semaphore");
   exit(-1);
}

/* Ruecksetzen einer Semaphore; V-Operation */
struct sembuf ops[1];

ops[0].sem_num = 0;
ops[0].sem_flg = 0;
ops[0].sem_op  = +1;

if( semop( semid, ops, 1 ) < 0 )
{
   perror("Fehler beim Ruecksetzen der Semaphore");
   exit(-1);
}
```

Beispielprogramm für die Verwendung von Semaphoren bei einem einfachen Dateizugriff

Als kritische Region wird ein Dateizugriff benutzt. Die Programme sollen nacheinander gestartet werden. Hierzu sind innerhalb der kritischen Region sleep-Befehle eingebaut, um auch optisch zu erkennen, daß die entsprechenden Dateizugriffe durch den anderen Prozeß verhindert wird.

Anmerkung:

Mit der Funktion time(0) wird die Zeit seit dem 1. Januar 1970 in Sekunden berechnet. Dadurch kann durch die Anweisung (time(0) - start_time) die Zeit in Sekunden seit Programmbeginn errechnet werden.

```
/* Testprogramm Semaphoren Teil 1 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY   5L

main()
{
   union semun
   {
      int val;
      struct semid_ds *buf;
      ushort *array;
   } arg;

   struct sembuf  ops[1];
   int            semid;
   FILE           *fp;
   time_t         start_time = time( 0 );

   printf("Semaphoren Programm 1 gestartet\n");

   /* Semaphore anfordern */
   if(( semid = semget( SEMKEY, 1, 0666 | IPC_CREAT )) < 0 )
   {
      perror("Fehler beim Anfordern der Semaphore");
      exit(-1);
   }

   /* Semaphore initialisieren */
   arg.val = 1;
   if( semctl( semid, 0, SETVAL, arg ) < 0 )
   {
      perror("Fehler beim Initialisieren der Semaphore");
      exit(-1);
   }

   ops[0].sem_num = 0;
   ops[0].sem_flg = 0;
   ops[0].sem_op  = -1;     /* P()-Operation vorbereiten */
```

```
   /* kritische Region beginnt */
   if( semop( semid, ops, 1 ) < 0 )
   {
      perror("Fehler bei der Operation P auf die Semaphore");
      exit(-1);
   }

      printf("Krit. Region Semaphoren Programm 1 beginnt %d\n",
              time(0)-start_time);

      fp = fopen( "daten", "w" );
      fprintf(fp, "Semaphoren Programm 1 war hier %d\n",
              time(0));

      sleep( 10 );  /* 10 Sekunden warten */

      fclose( fp );

   ops[0].sem_op = +1;     /* V()-Operation vorbereiten */

   if( semop( semid, ops, 1) < 0 )
   {
      perror("Fehler bei der Operation V auf die Semaphore");
      exit(-1);
   }
   printf("Krit. Region Semaphoren Programm 1 beendet %d\n",
          time(0)-start_time);

   exit(0);
}



/* Testprogramm Semaphoren Teil 2 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY   5L

main()
{
   union semun
   {
      int val;
      struct semid_ds *buf;
      ushort *array;
   } arg;

   struct sembuf  ops[1];
   int            semid;
   FILE           *fp;
   time_t         start_time = time( 0 );

   printf("\n Semaphoren Programm 2 gestartet\n");

   /* Semaphore verfuegbar machen */
   if(( semid = semget( SEMKEY, 1, 0666 | IPC_CREAT )) < 0 )
   {
      perror("Fehler beim Anfordern der Semaphore");
      exit(-1);
   }
```

```
   ops[0].sem_num = 0;
   ops[0].sem_flg = 0;
   ops[0].sem_op  = -1;     /* P()-Operation vorbereiten */

   /* kritische Region beginnt */
   if( semop( semid, ops, 1 ) < 0 )
   {
      perror("Fehler bei der Operation P auf die Semaphore");
      exit(-1);
   }
      printf("Krit. Region Semaphoren Programm 2 beginnt %d\n",
             time(0)-start_time);

      fp = fopen( "daten", "a" );
      fprintf( fp, "Semaphoren Programm 2 war auch hier %d\n",
               time(0));

      sleep( 3 );  /* 3 Sekunden warten */

      fclose( fp );

   ops[0].sem_op = +1;     /* V()-Operation vorbereiten */
   if( semop( semid, ops, 1 ) < 0 )
   {
      perror("Fehler bei der Operation V auf die Semaphore");
      exit(-1);
   }
   printf("Krit. Region Semaphoren Programm 2 beendet %d\n",
          time(0)-start_time);

   /* Semaphore loeschen */
   if( semctl( semid, 0, IPC_RMID, arg ) < 0 )
   {
      perror("Fehler beim Loeschen der Semaphore");
      exit(-1);
   }

   exit(0);
}
```

6.2.4 Interprozeß-Kommunikation mit Message-Queues

Mit dem System der Message-Queues kann, ähnlich dem System der Pipes, eine Nachricht (Nachrichtenpaket) an einen anderen Prozeß geschickt werden. Die wesentlichen Unterschiede zu Pipes sind:

?? "Messages" werden nicht direkt an den Empfänger geschickt, sondern zunächst in eine vom Betriebssystem verwaltete Warteschlange gehängt. Somit ist es nicht wie bei den Pipes nötig, daß beide Kommunikationsprozesse gleichzeitig (parallel) gestartet werden. Die Messages werden vielmehr vom Betriebssystem solange aufbewahrt, bis die Message-Queue ganz gelöscht wird.

?

?? Im Gegensatz zu Pipes kann nicht nur 1 Partner-Prozeß mit einem einzigen anderen Prozeß kommunizieren (**1:1-Beziehung**), vielmehr können hier m Prozesse Nachrichten an n Prozesse senden (**m:n-Beziehung**). Die Zuordnung einer Nachricht kann durch eine Message-Kennung erfolgen. Hierbei wird einer Nachricht beim Senden eine Message-Kennung neben der eigentlichen Nachricht beigefügt. Da mehrere Empfängerprozesse auf dieselbe Message-Queue zugreifen können, und diese keine Nachrichten empfangen wollen, die nicht für sie bestimmt sind, kann vom Empfänger-Prozeß nach der Message-Kennung selektiert werden.

?? Eine Pipe ist als FIFO organisiert. Eine Message-Queue ist für Messages mit der gleichen Message-Kennung ebenfalls als FIFO organisiert. Möchte man aber beispielsweise an einen Prozeß hochpriore und niedrpriore Nachrichten senden, so sendet man die hochprioren mit einer anderen Kennung als die niederprioren. Liest der Empfänger bevorzugt die hochprioren Meldungen aus, so können hochpriore Meldungen die niederprioren überholen.

Ein klassisches Beispiel für ein Warteschlangen-System bildet der Spooler eines Drucksystems. Ein Spooler ist eine Puffervorrichtung für zu langsame Geräte. Die Problemstellung ist, daß beispielsweise ein Drucker zur Ausgabe von Druckaufträgen sehr lange braucht und oftmals belegt ist, wenn bereits neue Druckaufträge eingehen. Wie Ihnen bekannt ist, ist ein Drucker ein passives, exklusiv nutzbares Betriebsmittel. Druckaufträge können nur am Stück abgearbeitet werden, bis sie fertig sind, dann kommt der nächste Druckauftrag dran. Die eingehenden Druckaufträge müssen also zwischengespeichert werden, damit sie nicht verlorengehen.

Ein solcher Spooler läßt sich in UNIX leicht implementieren. Dazu wird die Abarbeitung von Druckaufträgen in zwei Schritte unterteilt.

?? Zunächst werden die Druckaufträge durch das lp-Kommando in eine Warteschlange (Message -Queue) gehängt.

?

?? Ein anderer Prozeß "kümmert" sich um die Abarbeitung dieser Warteschlange und holt sich einen Druckauftrag nach dem anderen.

Beispiel:

In einer Message-Queue sind Nachrichten mit den Kennungen 1, 15, 2, 6, 7, 15 abgelegt. Ein Empfängerprozeß kann nun durch Angabe der Kennung Nr. 15 durch zweimaliges Auslesen alle Nachrichten mit der Kennung Nr. 15 auslesen.

Message-Queue Systemaufrufe unter UNIX

Folgende Schritte sind notwendig, einem Prozeß die Nutzung einer Message-Queue zu ermöglichen:

- Eine Warteschlange muß mit **msgget()** angelegt werden und dem Prozeß bekannt gemacht werden.
- Nachrichten werden mit **msgsnd()** in die Warteschlange geschrieben.
- Nachrichten werden mit **msgrcv()** aus der Warteschlange gelesen.
- Zu Verwaltung und Steuerung einer Warteschlange dient der Systemaufruf **msgctl()**

Anlegen und Anfordern einer Message-Queue

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget( key_t key, int msgflag );
```

Mit der Funktion **msgget()** wird eine Message-Queue vom Betriebssystem angefordert und ggf. angelegt.

- Der Parameter `key` ist der Schlüssel (Name) der Message-Queue im System (der Typ `key_t` ist definiert als `long`).
- Der Parameter `msgflag` setzt die Zugriffsrechte der Message-Queue. Die Zugriffsrechte auf die Message-Queue sind z.B. 0444 (lesen) oder 0666 (lesen und schreiben). Um eine Message-Queue zu Erstellen muß zusätzlich (als ODER-Verknüpfung) der Parameter **IPC_CREAT** angegeben werden. Hierbei kann der Aufruf (siehe auch **shmget()**) mit **IPC_CREAT** mehrmals erfolgen. Existiert die Message-Queue bereits, ignoriert das Betriebssystem dieses Flag.

Bei korrektem Ablauf gibt die Funktion als Returnwert eine nicht negative ID der Message-Queue zurück. Im Fehlerfall ist der Returnwert **-1** und die Variable **errno** wird gesetzt.

Beispiel:

```
if(( msgid = msgget( 20L, 0666 | IPC_CREAT )) < 0 )
{
   perror("Fehler beim Anfordern einer Message-Queue");
   exit(-1);
}
```

Einschreiben in die Message-Queue

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd( int msgid, struct msgbuf *msgp, int msgsz, int msgflag );
```

Mit der Funktion **msgsnd()** wird eine Nachricht in die Message-Queue eingeschrieben. Die Angabe des Typs der Nachricht wird innerhalb der Struktur `msgbuf` eingefügt.

?? Der Parameter `msgid` ist die ID der Message-Queue (vom Befehl **msgget()**).

?? Der Parameter `msgp` ist ein Zeiger auf eine Struktur der Nachricht (siehe unten).

?? Der Parameter `msgsz` ist die Größe der Nachricht (ohne mtype; siehe unten).

?? Der Parameter `msgflag` bestimmt die Art des Zugriffs auf die Queue; bei IPC_NOWAIT wartet die Funktion **msgsnd()** nicht, wenn die Message-Queue voll ist, sondern beendet sich mit dem Returnwert -1.

Der Returnwert der Funktion ist bei korrektem Ablauf **0**. Im Fehlerfall gibt die Funktion den Wert **-1** zurück und setzt die Variable **errno**.

Die Datenstruktur `msgbuf` ist in msg.h folgendermaßen definiert:

```
struct msgbuf
{
   long mtype;     /* Typ der Message */
   char mtext[1];  /* Text der Message */
};
```

Es können aber auch andere Datenstrukturen verwendet werden, um mit Hilfe der Message-Queue zu kommunizieren, z.B.:

```
struct my_msgbuf
{
   long mtype;          /* Typ der Message   */
   char nachricht[100]; /* Nachrichtenstring */
};
```

Bedingung hierbei ist aber immer, daß das erste Strukturelement eine Variable des Typs `long` ist, in der beim Einschreiben in die Message-Queue der Message-Typ (Message-Kennung) eingeschrieben ist, nach dem andere Prozesse selektieren können.

Im oben Beispiel von `my_msgbuf` ist die zu übertragende Größe in msgsz nicht die Größe der gesamten Struktur `my_msgbuf`, sondern nur die Länge des Strings, also z.B. `sizeof( nachricht )`.

Beispiel:

```
struct my_msgbuf buffer;

if( msgsnd( msgid, (struct msgbuf *)&buffer,
            sizeof( buffer.nachricht ), 0 ) < 0 )
{
   perror("Fehler beim Schreiben in die Message-Queue");
   exit(-1);
}
```

Auslesen aus der Message-Queue

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv( int msgid, struct msgbuf *msgp, int msgsz,
            long msgtyp, int msgflag );
```

Mit der Funktion msgrcv() wird eine Nachricht aus der Message-Queue ausgelesen.

?? Der Parameter `msgid` ist die ID der Message-Queue (vom Befehl **msgget()**).

?? Der Parameter `msgp` ist ein Zeiger auf eine Struktur der Nachricht (siehe **msgsnd()**).

?? Der Parameter `msgsz` ist die Größe der Nachricht, die maximal gelesen werden darf (ohne mtype; siehe **msgsnd()**).

?? Der Parameter `msgtyp` ist der Typ der Nachricht, die ausgelesen werden soll (siehe unten).

?? Der Parameter `msgflag` bestimmt die Art des Zugriffs auf die Queue; bei IPC_NOWAIT wartet die Funktion **msgrcv()** nicht, bis in der Message-Queue eine Nachricht ist, sondern beendet sich mit dem Returnwert -1.

Der Returnwert der Funktion ist bei korrektem Ablauf **0**. Im Fehlerfall gibt die Funktion den Wert **-1** zurück und setzt die Variable **errno**.

Die Selektion der Nachrichten kann auf folgende 3 Weisen geschehen:

1) Ist `msgtyp = 0`, so wird die erste Message in der Queue gelesen (FIFO-Prinzip).
2) Ist die Zahl positiv (`msgtyp > 0`), so wird sie als Typ der Message interpretiert und die älteste Message dieses Typs aus der Queue gelesen.
3) Ist die Zahl negativ (`msgtyp < 0`), so wird ihr Absolutwert genommen und die erste Message gelesen, deren Typ der kleinste und kleiner gleich dem Absolutwert ist. Ist also `msgtyp = -5`, so wird eine Message gelesen, deren Typ kleiner gleich 5 ist und die den kleinsten Wert für den Typ hat.

Beispiel:

```
struct my_msgbuf buffer;

/* Auslesen einer Nachricht mit dem Typ 5 */
if( msgrcv( msgid, (struct msgbuf *)&buffer,
            sizeof( buffer.nachricht ), 5L, 0 ) < 0 )
{
   perror("Fehler beim Lesen aus der Message-Queue");
   exit(-1);
}
```

Steuern einer Message-Queue

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl( int msgid, int cmd, struct msgid_ds *buf );
```

Die Funktion msgctl() dient zur Steuerung einer Message-Queue.

?? Der Parameter `msgid` ist die ID der Message-Queue (vom Befehl **msgget()**).

?? Der Parameter `cmd` ist das Steuerkommando; folgende Kommandos sind möglich:

- ?? **IPC_STAT:** Es kann der aktuelle Status der Message-Queue abgefragt werden. Das Ergebnis wird in `buf` geschrieben.
- ?? **IPC_SET:** Die Zugriffsrechte können geändert werden.
- ?? **IPC_RMID:** Die Message-Queue wird gelöscht.

?? Der Parameter `buf` ist eine Struktur vom Typ `struct msgid_ds` und wird für das Kommando IPC_STAT benötigt.

Der Returnwert der Funktion ist bei korrektem Ablauf **0**. Im Fehlerfall gibt die Funktion den Wert **-1** zurück und setzt die Variable **errno**.

Beispiel:

```
/* Loeschen einer Message-Queue */
if( msgctl( msgid, IPC_RMID, 0 ) < 0 )
{
   perror("Fehler beim Loeschen der Message-Queue");
   exit(-1);
}
```

Programmbeispiel zu Message-Queue

Das Sende-Programm schreibt eine Message in die Message-Queue und beendet sich. Das Empfangs-Programm liest die Message aus der Queue aus und löscht anschließend die Message-Queue im System.

```
/* Message Queue - Sende-Programm */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXLEN    256
#define MSGKEY    5L

struct my_msgbuf
{
   long mtype;
   char nachricht[MAXLEN];
};

main( int argc, char *argv[])
{
   struct my_msgbuf buffer;
   int   msgid;

   if( argc < 2 )
   {
      printf("Text beim Aufruf uebergeben\n");
      exit(-1);
   }

   /* Anlegen und Anfordern der Message-Queue */
   if(( msgid = msgget( MSGKEY, 0666 | IPC_CREAT )) < 0 )
   {
      perror("Fehler beim Anfordern der Message-Queue");
      exit(-1);
   }

   strcpy( buffer.nachricht, argv[1] );
   buffer.mtype = 1L;  /* Type der Nachricht wird 1 */

   /* Senden der Message */
   if( msgsnd( msgid, (struct msgbuf *)&buffer,
               siezof( buffer.nachricht ), 0) < 0 )
   {
      perror("Fehler beim Schreiben in die Message-Queue");
      exit(-1);
   }

   printf("Nachricht wurde gesendet\n");
   exit(0);
}
```

```
/* Message Queue - Empfangs-Programm */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXLEN    256
#define MSGKEY    5L

struct my_msgbuf
{
   long mtype;
   char nachricht[MAXLEN];
};

main()
{
   struct my_msgbuf buffer;
   int    msgid;

   /* Anlegen und Anfordern der Message-Queue */
   if(( msgid = msgget( MSGKEY, 0666 | IPC_CREAT )) < 0 )
   {
      perror("Fehler beim Anfordern der Message-Queue");
      exit(-1);
   }

   /* Empfangen der Message */
   /* Es soll nach dem FIFO-Prinzip ausgelesen werden */
   if( msgrcv( msgid, (struct msgbuf *)&buffer,
               sizeof( buffer.nachricht ), 0L, 0 )) < 0 )
   {
      perror("Fehler beim Lesen aus der Message-Queue");
      exit(-1);
   }

   printf("Nachricht wurde gelesen:\n");
   printf("MSG:  %s\n",  buffer.nachricht );
   printf("TYPE: %ld\n", buffer.mtype );

   /* Loeschen der Message-Queue im System */
   if( msgctl( msgid, IPC_RMID, 0 ) < 0 )
   {
      perror("Fehler beim Loeschen der Message-Queue");
      exit(-1);
   }

   exit(0);
}
```

6.2.5 Signale

[Quelle: A. S. Tanenbaum, Moderne Betriebssysteme].

Auf Signale soll hier nicht so ausführlich wie auf andere Mechanismen der Interprozeß-Kommunikation eingegegangen werden. Es soll jedoch kurz erwähnt werden, wozu sie gut sind.

Schickt man einem Prozeß ein Signal, so wird die normale Ausführung des Prozesses unterbrochen wie bei einem Hardware-Interrupt. Es wird der Signal-Handler des Prozesses angesprungen und abgearbeitet, d.h. im Signalhandler findet die Reaktion auf das Signal statt.. Nach Abarbeitung des Signalhandlers arbeitet der Prozeß an der Stelle weiter, an der er unterbrochen worden war.

Signale sind also das Software-Analogon zu Hardware-Unterbrechungen (Interrupts).

Viele Traps (Unterbrechungen), die von der Hardware erkannt werden wie die Ausführung eines illegalen Befehls oder die Benutzung einer illegalen Adresse werden vom Betriebssystem in ein Signal an den schuldigen Prozeß konvertiert.

Signale werden auch für die Kommunikation zwischen Prozessen verwendet, wenn ein Prozeß eilig einem anderen Prozeß etwas übermitteln möchte.

6.3 Aufgaben zu Kap. 6

1.1a
- Was versteht man bei einem virtuellen Betriebssystem wie VAX/VMS unter einer virtuellen und einer physikalischen Seite (page)? Wo kann eine physikalische Seite liegen ?
- Vergleichen Sie bei VAX/VMS die Größen von virtueller Seite, physikalischer Seite und Block auf Platte und geben Sie an, was daran günstig ist!

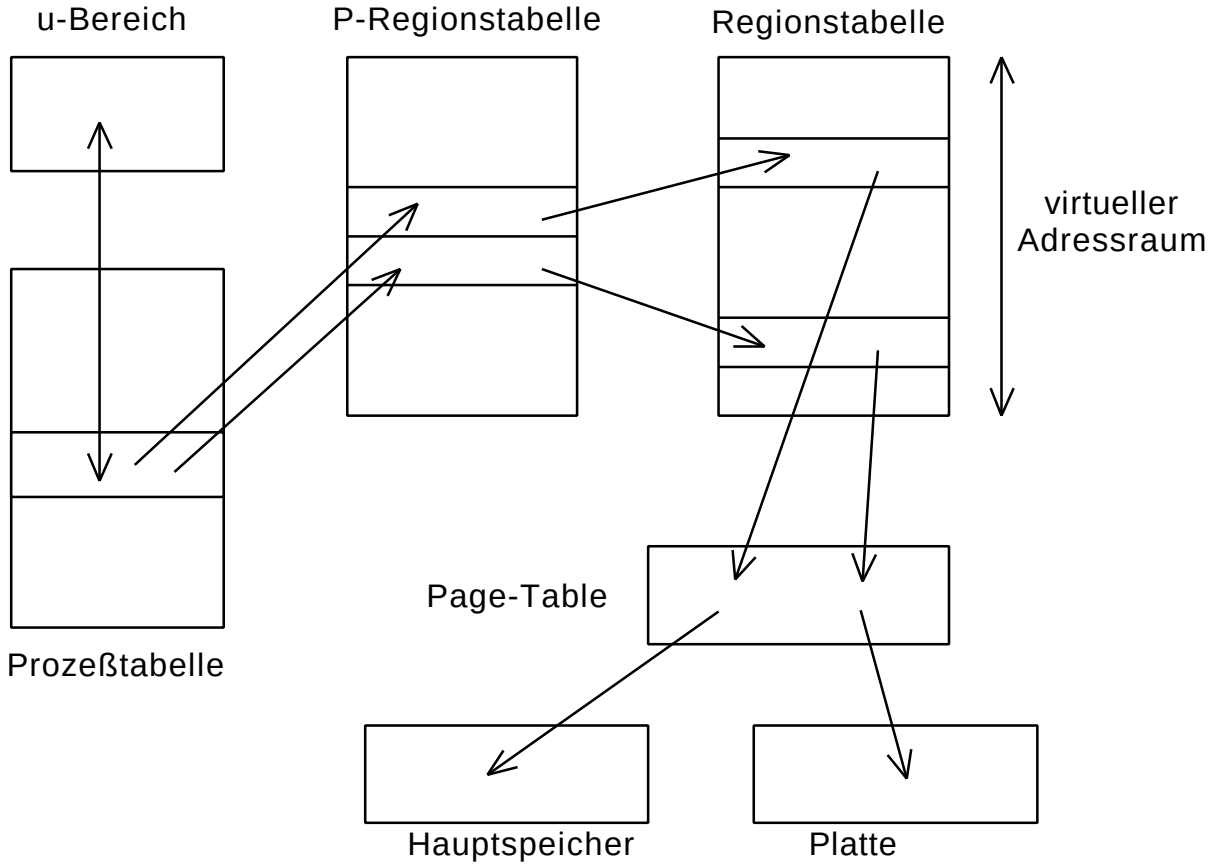
1.1b Wie wird die Zuordnung virtuelle Seite zu physikalischer Seite durchgeführt ?

1.2
- Was ist Pagen ?
- Was ist Swappen ?
- Gibt es bezüglich Pagen und Swappen Unterschiede zwischen VAX/VMS und RSX-11M (16-bit-Betriebssystem auf der PDP-11)?

1.3a Welche Regionen hat ein Betriebssystem-Prozeß ?

1.3b Wozu braucht man einen Stack ?

1.4 Shared Memory unter UNIX.
- Wieviele P-Regionstabellen gibt es auf einem Rechner ?
- Wieviele Regionstabellen gibt es auf einem Rechner ?
- Erweitern Sie die folgende Skizze, um zu zeigen, wie ein Shared Memory unter UNIX zwischen einem Prozeß A und einem Prozeß B realisiert wird. Erläutern Sie Ihre Skizze!



7 Dateisystem unter UNIX

[Thomas Natterer, Joachim Goll]

In Kapitel 7.1 wird zunächst die Einbettung des Dateisystems in die Architektur des UNIX-Betriebssystems behandelt. Weiter soll einen kurzer Überblick über das Dateisystem gegeben werden. Mit dem internen Aufbau des Dateisystems befaßt sich dann Kapitel 7.2.

7.1 Überblick über das Dateisystem

7.1.1 Architektur des UNIX-Betriebssystems

Um das Dateisystem besser verstehen zu können, wird hier die Architektur des UNIX-Betriebssystems nochmals dargestellt.

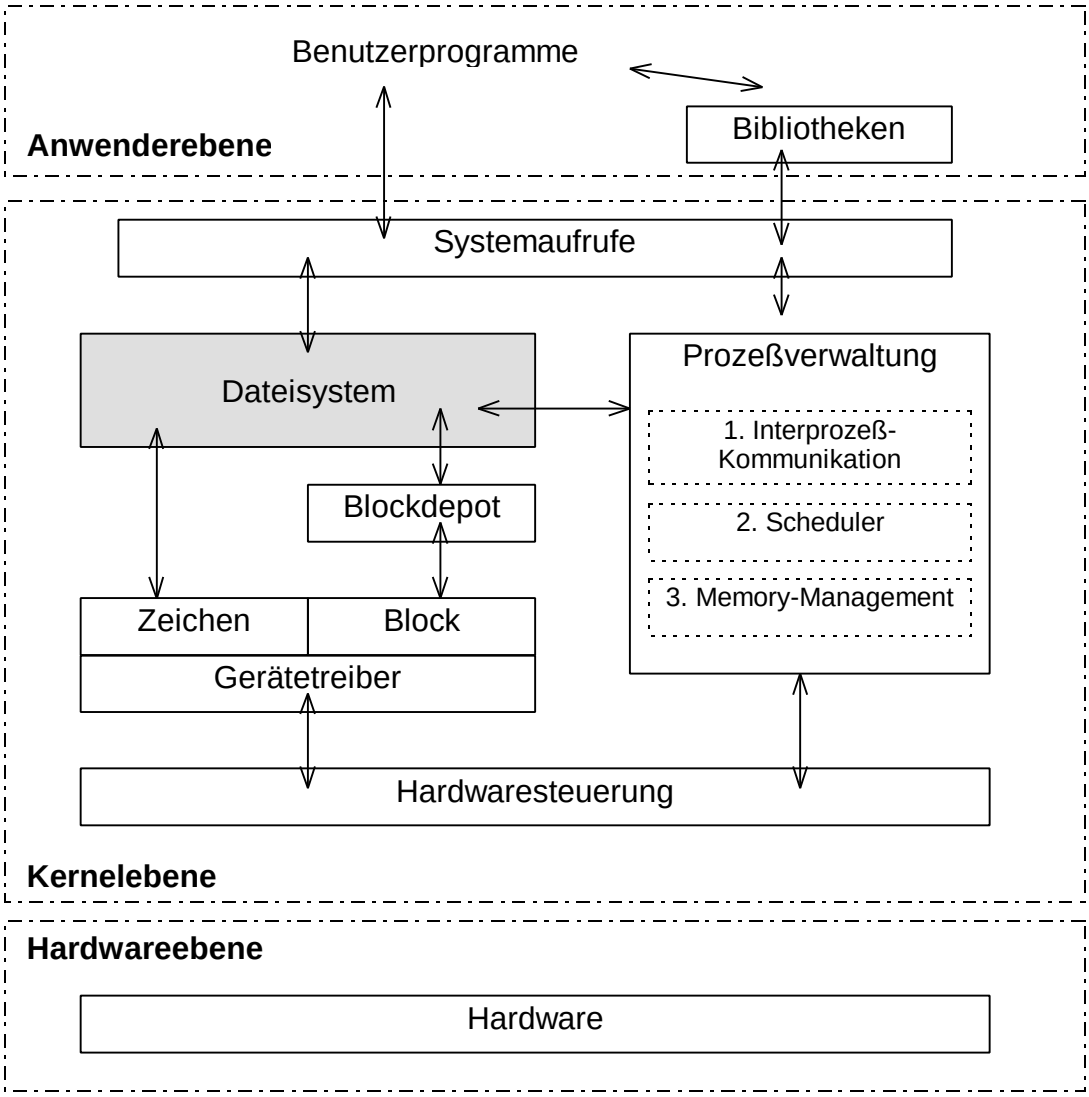


Bild 7.1.1-1 Blockdiagramm des Systemkerns

Das Betriebssystem läßt sich in drei Ebenen einteilen:
?? die Ebene der Anwendungen oder Benutzerprogramme
?? die Ebene des Kernels des Betriebssystems
?? die Hardwareebene

Neben dem Dateisystem ist die Prozeßverwaltung ein Hauptbestandteil des Systemkerns. Die **Prozeßverwaltung** umfaßt den **Scheduler**, das **Memory-Management** und die **Interprozeßkommunikation**. Systemaufrufe und Bibliotheksaufrufe bilden die Grenzen zwischen Anwendungen und dem Systemkern. Assemblerprogramme können Systemaufrufe direkt tätigen. Programme in höheren Programmiersprachen müssen hingegen einen Umweg über Bibliotheksfunktionen machen. Die Bibliothek ihrerseits führt die eigentlichen Aufrufe an das Betriebssystem durch.

Eine Verbindung gibt es auch zwischen Prozeßverwaltung und Dateisystem. Sie wird benötigt, wenn die Prozeßverwaltung eine Datei zur Ausführung in den Arbeitsspeicher lädt.

Die Systemaufrufe können in zwei Gruppen eingeteilt werden:
?? Systemaufrufe in Bezug auf das Dateisystem
`(close, read, write, chmod,...)`
?? Systemaufrufe in Bezug auf das Prozeßsteuersystem
`(fork, exit, exec, wait)`

Die Hardwaresteuerung bildet die Schnittstelle des Kernels zur Hardware. Die Hardwaresteuerung übernimmt die Interruptbehandlung und Kommunikation mit der Hardware des Rechners. Durch Interrupts können Geräte wie z.B. die Ein-/Ausgabegeräte oder der Zeitgeber des Rechners die CPU beim Bearbeiten eines Prozesses unterbrechen. Die Bearbeitung der Interrupts übernehmen dabei Funktionen im Systemkern.

7.1.2 Benutzersicht des Dateisystems

Zunächst noch etwas Detailinformation. Bevor eine Datei gelesen oder beschrieben werden kann, muß sie geöffnet werden, wobei die Zugriffsrechte überprüft werden. Wenn der Zugriff erlaubt ist, liefert das System eine kleine positive ganze Zahl, die **Dateideskriptor (file descriptor)** oder **Handle** genannt wird. Der Handle wird dann in den nachfolgenden Operationen wie Lesen, Schreiben, Positionieren oder Schließen der Datei verwendet. Wenn der Zugriff auf die Datei verboten ist, wird beim Öffnen ein Fehlercode zurückgegeben.

In UNIX - wie auch in MS-DOS, welches von UNIX abgespickt hat, aber auch in anderen Betriebssystemen - werden I/O-Geräte als Dateien abstrahiert. Es handelt sich dabei um sogenannte Spezialdateien (special files). Diese Spezialdateien wurden extra eingeführt, damit I/O-Geräte wie Dateien aussehen. Damit muß ein Programmschreiber - wenn der entsprechende Treiber zur Verfügung steht - nichts hinzulernen, sondern kann mit dem I/O-Device kommunizieren, wie wenn es eine Datei wäre.

Es gibt zwei Arten von Spezialdateien:

- **blockorientierte Spezialdateien (block special file)**
- **zeichenorientierte Spezialdateien (character special file)**

Blockorientierte Spezialdateien werden benutzt, um Geräte zu modellieren, die aus Blöcken bestehen, die frei adressierbar sind (random access devices wie z.B. Platten). Wird eine blockorientierte Spezialdatei geöffnet, so kann ein Block gelesen werden, ohne daß man sich um die Struktur des Dateisystems, das dies ermöglicht, kümmern zu müssen.

Zeichenorientierte Spezialdateien werden benutzt, um Geräte zu modellieren, die aus Zeichenströmen bestehen. Beispiele hierfür sind Terminals, Drucker, Netzschnittstellen. Ein Programm schreibt auf das entsprechende I/O-Gerät, indem es in die korrespondierende zeichenorientierte Spezialdatei schreibt. Analoges gilt für das Lesen.

Dateinamen

Während beispielsweise das Dateisystem von MS-DOS zwischen Groß- und Kleinbuchstaben in Dateinamen nicht unterscheidet, ist dies bei UNIX anders: `Messdaten` und `messdaten` sind für das UNIX-Dateisystem zwei verschiedene Dateien. Bei MS-DOS können Namen für Dateien auch zweiteilig sein, wobei zwischen dem eigentlichen Dateinamen und der Dateinamenserweiterung, die maximal 3 Stellen umfaßt, ein Punkt steht (Beispiel: ausgabe.c). Wenn bei einer UNIX-Version die Möglichkeit der Erweiterung besteht, so kann ein Name auch zwei oder mehr Erweiterungen haben, d.h. es können mehrere Punkte im Dateinamen vorkommen.

Das Dateisystem ist als Baumstruktur mit einem einzigen Wurzelknoten „root“ (geschrieben als „ / “) organisiert. Jeder Knoten, der nicht am Ende eines Zweiges steht, ist immer ein Inhaltsverzeichnis (Katalog). Kataloge stellen auch Dateien dar. Knoten am Ende eines Zweiges können entweder

- Kataloge,
- normale Dateien
- oder spezielle Gerätedateien (zeichenorientierte Spezialdateien, blockorientierte Spezialdateien

sein.

Zeichenorientierte Spezialdateien dienen in der Regel zur Ein/Ausgabe auf serielle Ein/Ausgabegeräte wie Terminals, Drucker oder Netzwerkschnittstellen, blockorientierte Spezialdateien zur Ein/Ausgabe auf Massenspeicher.

Hierarchische Verzeichnisstruktur

Das Dateisystem wird als Verzeichnisbaum organisiert. Dateien eines Nutzers sind topologisch gesehen Blätter im Verzeichnisbaum.

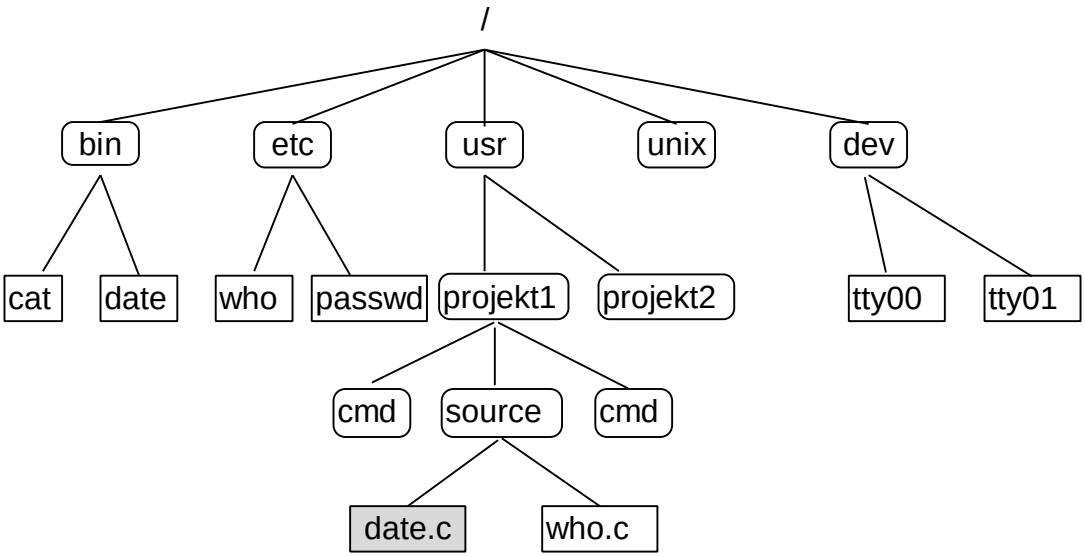


Bild 7.1.2 - 1 Strukturbaum des Dateisystems

Pfadnamen

Auf eine Datei wird in der Anwenderebene über ihren Pfadnamen zugegriffen. Der Pfadname beschreibt nicht nur den Weg zum Lokalisieren der Datei innerhalb des Dateisystems, sondern er enthält auch den Namen der Datei. Dabei setzt sich der Pfadname aus einer Folge von durch einen Separator getrennten Dateinamen zusammen. Wie Sie bereits wissen, ist im Falle von MS-DOS der Separator der \ (Backslash), im Falle von UNIX der / (Schrägstrich): Ein Dateiname ist dabei eine Folge von Zeichen zum eindeutigen Kennzeichnen einer Datei im Katalog.

Absoluter und relativer Pfadname

Wenn das erste Zeichen des Pfads der Separator ist, so ist der Pfad absolut, d.h. er beginnt bei der Wurzel des Dateisystems.

Ein Pfadname muß nicht unbedingt bei der Wurzel beginnen, sondern kann durch Weglassen des ersten Schrägstriches relativ zum aktuellen Katalog (Arbeitsverzeichnis) gebildet werden.

Ein Beispiel sollte das Zugreifen auf eine Datei aus Benutzersicht am besten erklären. So wird z.B. die Datei „date.c“ in Bild 2.1.2 - 1 durch folgende Ausdrücke bestimmt:

```
/usr/projekt1/source/date.c   (absoluter Pfadname)
projekt1/source/date.c        (relativer Pfadname, relativ zu
            aktuellem Katalog „usr “)
```

Die meisten Betriebssysteme, die ein hierarchisches Dateisystem unterstützen, haben zwei besondere Einträge in einem Verzeichnis, den Eintrag „.“ und den Eintrag „..“ (ausgesprochen als „Punkt“ und „Punkt-Punkt“). Der Eintrag „.“ bezeichnet das eigene

Verzeichnis, der Eintrag „..“ das Verzeichnis des Vaters. Das aktuelle Verzeichnis sei etwa

```
/usr/projekt1/source
```

Dann kann man mit dem Befehl

```
cp ../ ../projekt2/lib/drucken.c
```

die Datei drucken.c in das aktuelle Verzeichnis kopieren. Dabei geht das System zwei Hierarchiestufen höher bis zu /usr und dann wieder zwei Stufen tiefer über das Verzeichnis projekt2 zum Verzeichnis lib.

Dateiattribute

Eine Datei hat einen Namen und Daten, die in ihr abgespeichert sind. Zu einer Datei gehören aber auch zusätzliche Informationen, wie z.B.:

- wer wie auf die Datei zugreifen darf,
- wann die Datei erzeugt worden ist,
- wann die Datei zum letzten Mal verändert wurde,
- ob die Datei schon mit Backup gesichert worden ist oder nicht,
- wie groß die Datei ist,
- etc.

Diese zusätzlichen Dateinformationen werden als Dateiattribute bezeichnet. Welche dieser Attribute geführt werden, ist von Betriebssystem zu Betriebssystem verschieden.

Verzeichniseinträge

Ein Verzeichnis enthält pro Datei einen Eintrag. Beim Zugriff auf eine Datei sucht man die Datei mit Hilfe des Dateinamens, zusätzlich braucht man die Dateiattribute und die Plattenadresse, wo die Datei gespeichert ist.

Die einfachste Möglichkeit ist, daß ein Verzeichniseintrag auch die Dateiattribute und die Plattenadresse enthält. Eine andere Möglichkeit ist, die Dateiattribute nicht im Dateientrag, sondern in einer separaten Datenstruktur zu führen.

Attribute innerhalb des Verzeichniseintrags

Bei DOS werden die Attribute im Verzeichniseintrag eingetragen. Das folgende Bild zeigt die Struktur eines Verzeichnisses bei DOS.

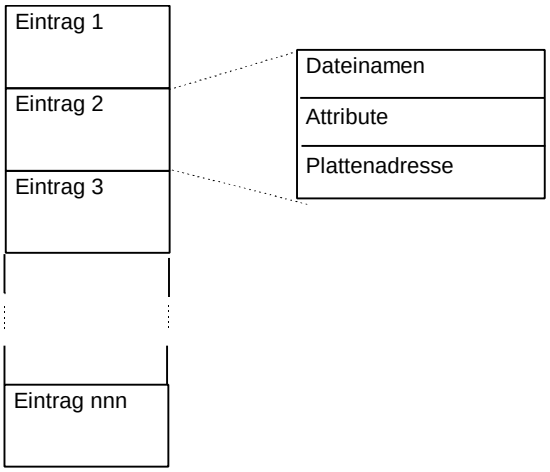


Bild 7.1.2 - 2 Verzeichnis bei DOS

Jeder Verzeichniseintrag enthält 32 Bytes:

- Dateiname
- Dateinamenserweiterung
- verschiedene Dateiattribute
- Zeiger auf ersten Block des Files

Die Gesamtkapazität eines Datenträgers wird bei DOS über die Dateizuordnungstabelle (FAT - File Allocation Table) und das Dateiverzeichnis verwaltet. Anhand von FAT und Dateiverzeichnis führt DOS Buch über den belegten und unbelegten Speicherplatz der Diskette. Dabei steht in der FAT, welche Dateien welche Blöcke belegt haben.

Attribute außerhalb des Verzeichniseintrags

Hier steht im Verzeichniseintrag ein Verweis auf eine separate Datenstruktur.

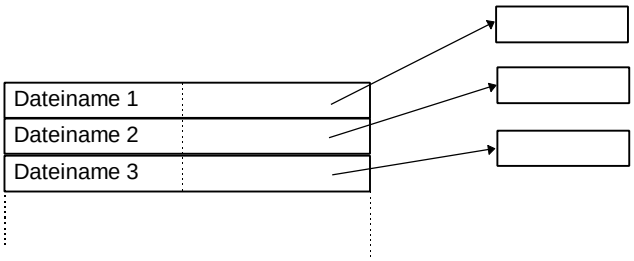


Bild 7.1.2 - 3 Attribute in separaten Datenstrukturen

7.1.3 Techniken zur Verwaltung von Plattenblöcken einer Datei

Im folgenden werden vier grundlegende Techniken betrachtet:

?? Zusammenhängende Speicherung einer Datei
?? Verkettete Liste von Plattenblöcken
?? Index-Tabelle von Zeigern und Verwendung der Startadressen
?? I-Nodes

7.1.3.1 Zusammenhängende Speicherung einer Datei

Am einfachsten ist es, Dateien am Stück auf Platte zu speichern. Da aber kleinere Lücken auf der Platte nicht für das Abspeichern einer großen Datei verwendet werden können, wird hierbei viel Platz verschwendet. Im Prinzip müßte die Platte laufend reorganisiert werden. Dies ist aber aus Performance-Gründen nicht möglich.

7.1.3.2 Verkettete Liste von Plattenblöcken

Auch hier muß man sich die Startadresse merken. Jeder Plattenblock enthält einen Zeiger auf den nächsten Plattenblock der Datei. Der letzte Plattenblock der Datei enthält den NULL-Zeiger, der das Ende des Files („end of file“) anzeigt.

Mit dieser Technik geht kein Platz auf der Platte verloren. Aus 2 Gründen ist die verkettete Liste von Blöcken jedoch unpraktisch. Zum einen kann die Größe eines Datenblocks keine Zweierpotenz mehr sein, da der Zeiger auch einige Bytes braucht. Dies ist für viele Programme unpraktisch. Zum anderen ist das Durchlaufen durch die verkettete Liste auf der Platte langsam.

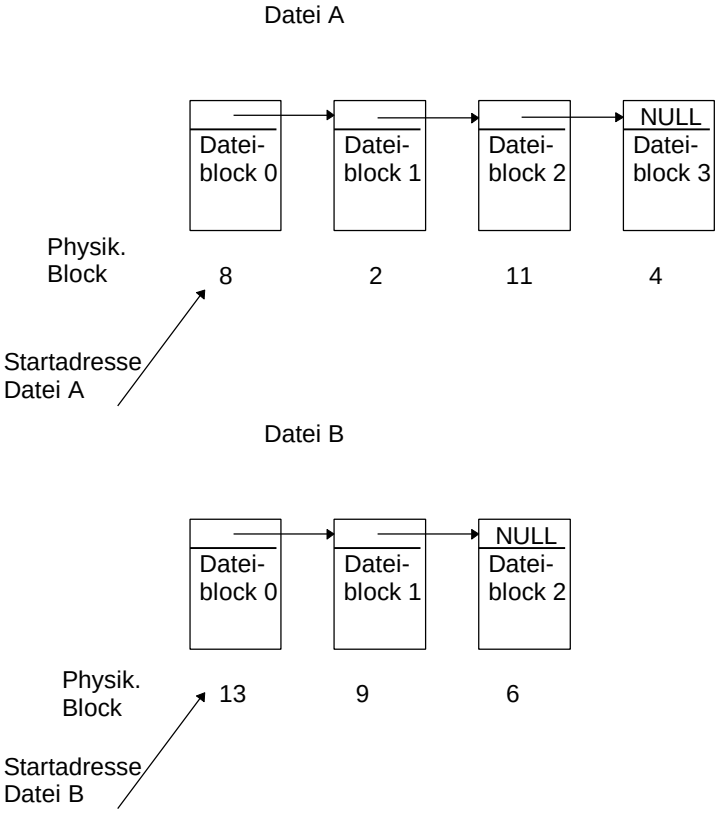


Bild 2.1.3.2 - 1 Verkettete Listen von Plattenblöcken

7.1.3.3 Index-Tabelle von Zeigern und Verwendung der Startadressen

Beide Nachteile der verketteten Liste von Plattenblöcken können beseitigt werden, wenn man im Hauptspeicher eine Tabelle (einen Index) aufbaut, in welcher die Verkettung der Plattenblöcke steht. Damit ist man zum einen schneller und zum anderen paßt in jeden Plattenblock eine Zweierpotenz von Bytes. Diese Tabelle wird natürlich auf der Platte gesichert. Jeder Block des Index enthält die Adresse des nächsten Blockes der Datei

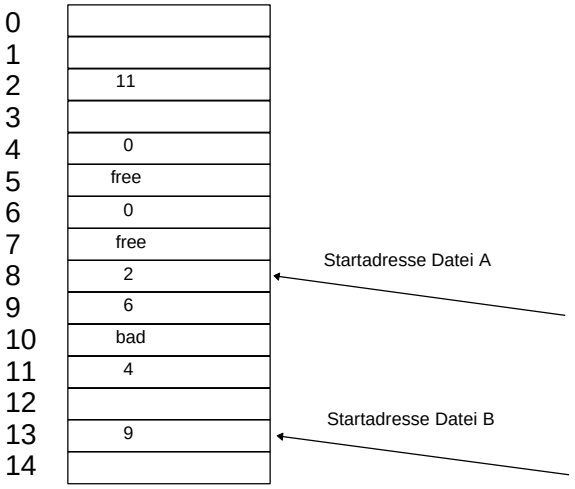


Bild 2.1.3.3 - 1 Index-Tabelle

Dies ist die prinzipielle Struktur der FAT von MS-DOS. Das Verzeichnis jeder Datei enthält den Startblock. Die FAT enthält den Rest der Kette.

7.1.3.4 I-Nodes (Inodes, Inoden)

Eine andere Technik ist, zu jeder Datei eine Tabelle bereitzustellen, die die Attribute und die Plattenadressen der Blöcke der Datei auflistet. Eine solche Tabelle wird Index-Node, kurz Inode genannt.

Um die Struktur eines Inode klein zu halten, um nicht eine Struktur variabler Länge für den Inode zu haben und dennoch große Dateien zuzulassen, sind Inoden wie in Bild 2.1.3.4 - 1 aufgebaut.

Die ersten Adressen von Plattenblöcken sind in dem I-Node gespeichert. Diese Adressen reichen nur für kleine Dateien.

Es gibt aber auch Adressen in dem I-Node, die die Adresse eines Plattenblockes enthalten, welcher weitere Plattenadressen enthält. Dieser Block wird „einfach indirekter Block“ genannt. Die Blöcke mit der Kennung **"einfach indirekt"** verweisen auf Blöcke, die ihrerseits eine Reihe von direkten Blocknummern enthalten. Beim Zugriff auf Daten über einen indirekten Block muß der Kern zuerst diesen indirekten Block lesen, den passenden direkten Blockeintrag ermitteln und dann diesen Block lesen.

Es gibt auch Adressen, die auf Blöcke zeigen, die die Adressen von einfach indirekten Blöcken enthalten. Solche Blöcke mit dem Kennzeichen **„doppelt indirekt"** enthalten eine Liste indirekter Blocknummern.

Blöcke mit dem Kennzeichen **„dreifach indirekt“** enthalten eine Liste von doppelt indirekten Blocknummern.

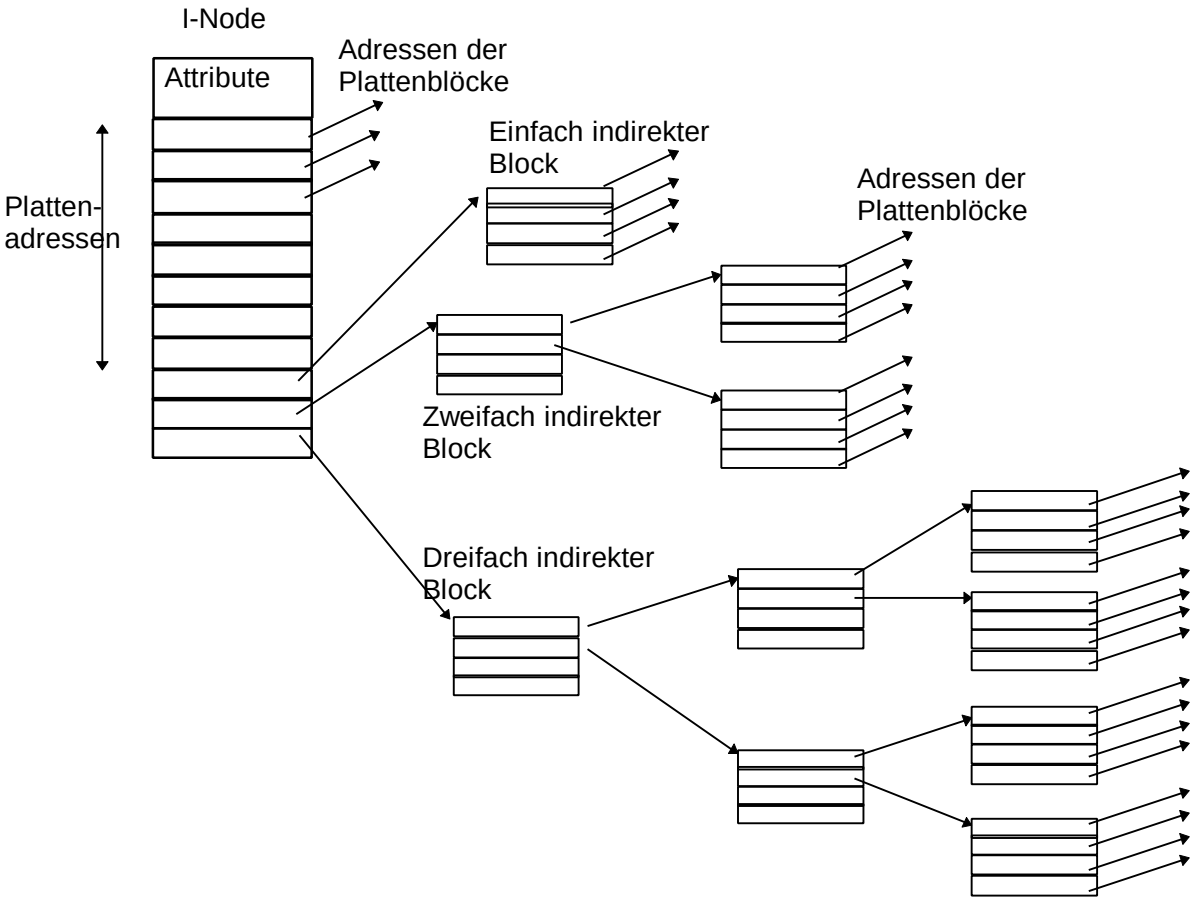


Bild 2.1.3.4 - 1 Struktur eines Inode

Im Prinzip könnte diese Methode auf "vierfach indirekt", "fünffach indirekt" usw. erweitert werden, doch hat sich die gezeigte Struktur in der Praxis als ausreichend erwiesen. Wird davon ausgegangen, daß ein logischer Block im Dateisystem 1K Byte enthält und die Blocknummer eine 32-Bit Integerzahl (4 Byte) darstellt, so kann ein Block bis zu 256 Blocknummern aufnehmen. Als maximale Größe einer Datei ergibt sich dann bei 10 direkten, 1 indirekten,1 doppelt indirekten und 1 dreifach indirekten Block in der Inode ein Bereich von 16 Gbyte (10 x 1K, 1 x 256 K, 1 x 256 x 256 K, 1 x 256 x 256 x 256 K). Da die Dateigrößenangabe in dem Inode auf 32 Bit beschränkt ist, ergibt sich jedoch eine tatsächliche Maximalgröße von 4 GigaByte (2^{32}) pro Datei.

7.1.4 Datenstruktur von Verzeichnissen unter UNIX

Verzeichnisse (Kataloge) sind die Dateien, die dem Dateisystem seine hierarchische Struktur verleihen. Sie spielen eine wichtige Rolle beim Umwandeln des Dateinamens in die Inodenummer.

Ein Katalog ist eine Datei mit einer Folge von **Einträgen**. Ein Verzeichniseintrag enthält nur die Inode-Nummer und den Dateinamen.

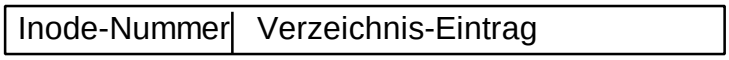


Bild 2.1.4 - 1 Verzeichnis-Eintrag unter UNIX

Die Attribute einer Datei werden in dem Inode gespeichert.

Der Kern speichert die Daten eines Katalogs ebenso wie die Daten einer gewöhnlichen Datei, also über die Inodestruktur und die verschiedenen Ebenen direkter und indirekter Blöcke. Prozesse können Kataloge genau wie normale Dateien lesen, doch der Kern reserviert sich das ausschließliche Recht, in Kataloge zu schreiben. Dadurch wird deren korrekte Struktur gewährleistet.

Die Zugriffsrechte für einen Katalog haben folgende Bedeutung:

- **Lesezugriff** gestattet einem Prozeß das Lesen eines Katalogs; Schreibzugriff gestattet einem Prozeß das Erzeugen neuer Katalogeinträge sowie das Entfernen alter Einträge (über die Systemaufrufe `creat`, `mknod`, `link` und `unlink`), wodurch der Inhalt des Katalogs verändert wird.
- **Das Ausführungsrecht** gestattet einem Prozeß das Durchsuchen des Katalogs nach einem bestimmten Dateinamen (es ist ja sinnlos, einen Katalog auszuführen).

Konvertierung eines Pfadnamens in einen Inode

Der erstmalige Zugriff auf eine Datei erfolgt über ihren Pfadnamen wie z. B. in den Systemaufrufen `open`, `chdir` (Wechseln des Katalogs = change directory) oder `link`. Da der Kern intern jedoch mit Inodes anstelle von Pfadnamen arbeitet, konvertiert er für den Dateizugriff diese Pfadnamen in Inodes. Der Algorithmus `namei` (eine Systemroutine) geht dazu den Pfadnamen schrittweise durch. Dabei wird jede einzelne Komponente über ihren Namen und den durchsuchten Katalog in einen Inode umgewandelt. Am Ende wird schließlich der Inode des Pfadnamens zurückgegeben.

Jeder Prozeß ist mit einem aktuellen Katalog verbunden. Der **u-Bereich** enthält dafür die aktuelle Kataloginode. Der aktuelle Katalog für den ersten Prozeß im System (Prozeß 0) ist der Wurzelkatalog. Der aktuelle Katalog jedes anderen Prozesses ist zumindest am Anfang der aktuelle Katalog des jeweiligen Vaterprozesses zum Zeitpunkt der Prozeßerzeugung. Prozesse können ihren aktuellen Katalog durch Ausführen des Systemaufrufs `chdir` wechseln. Alle Suchvorgänge nach Pfadnamen beginnen beim aktuellen Katalog eines Prozesses. Falls der Pfadname aber mit einem Schrägstrich beginnt, bewirkt dies eine Suche vom Wurzelkatalog aus. In beiden Fällen kann der Kern den Inode für den Start der Pfadnamensuche einfach ermitteln. Der

aktuelle Katalog steht im u-Bereich des Prozesses und die Wurzelinode ist in einer globalen Variablen abgelegt.

Es soll nun beispielsweise die Datei

`/usr/source/test.c`

geöffnet werden. Der Inode des Wurzelverzeichnisses steht an einer definierten Stelle auf der Platte. Der Inode enthält die Pointer auf die Plattenblöcke. Damit kann die Wurzeldatei bearbeitet werden und innerhalb der Wurzeldatei nach dem Eintrag `usr` gesucht werden. Bei diesem Eintrag steht der zugehörige Inode. Jeder Inode hat eine feste Position auf der Platte, deshalb ist ein Inode auf der Platte leicht zu finden. Mit Hilfe des Inodes kann auf die Datei `usr` zugegriffen werden. Innerhalb der Verzeichnisdatei wird nach dem Eintrag `source` gesucht. Dieser Eintrag enthält den zugehörigen Inode und damit kann innerhalb der Datei `source` nach dem Eintrag `test.c` gesucht werden. Ist der Eintrag gefunden, so hat man auch den zugehörige Inode und der Zugriff auf `test.c` ist möglich. Der Inode von `test.c` wird in den Arbeitsspeicher geladen und dort gehalten, bis durch `close` die Datei geschlossen wird.

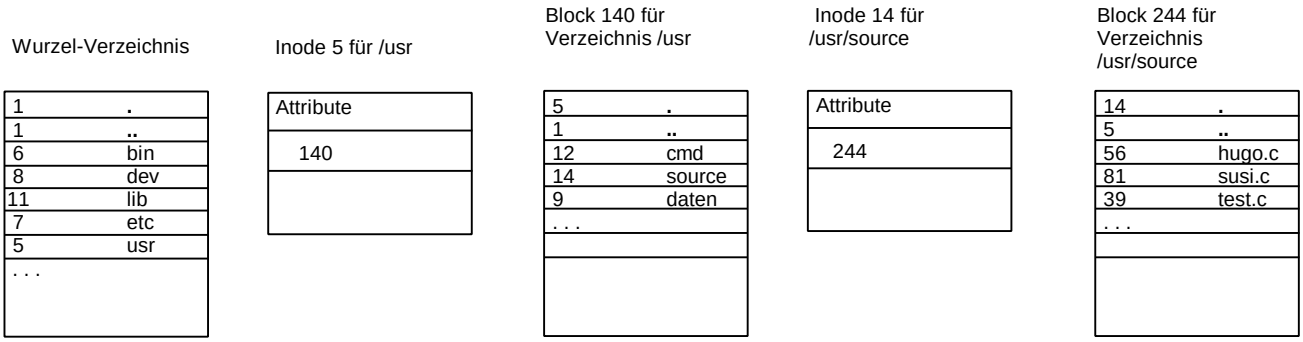


Bild 2.1.4 - 2 Suchvorgang beim Öffnen einer Datei

Anmerkung:

Das Programm **mkfs** initialisiert das Dateisystem so, daß „.“ und „..“ des Wurzelkatalogs die Nummer der Wurzelinode des Dateisystems erhalten.

7.1.5 Physikalischer Aufbau eines Dateisystems unter UNIX

UNIX speichert Dateien und Kataloge auf Platten, sogenannten Block-I/O-Geräten. Eine Platte kann dabei in mehrere logische Einheiten (Partitionen) aufgeteilt werden. Das Aufteilen einer Platte in mehrere Partitionen und damit in mehrere Dateisysteme erleichtert dem Systemverwalter die Pflege der dort gespeicherten Daten.

Die Ansteuerung der Plattencontroller wird durch den Plattentreiber übernommen. Für eine Platte eines bestimmten Typs ist ein Plattentreiber notwendig. Er übernimmt die Umsetzung von Dateisystemen in logische Einheiten (Partitionen) einer Platte.

Ein Dateisystem besteht - wie schon erwähnt - aus Dateien. Dateien wiederum bestehen aus mehreren logischen Datenblöcken. Je nach Implementierung besitzen diese eine Größe von 512, 1024, 2048 Byte oder jedes andere geeignete Vielfache von 512 Byte. Dabei ist zu beachten, daß die logischen Blöcke innerhalb eines Dateisystems die gleiche Größe haben.

Die Größe der Blöcke hat natürlich einen Einfluß auf die Datentransferrate eines Systems. So steigert die Verwendung großer Blöcke die effektive Transferrate zwischen Platte und Speicher. Der Systemkern kann mit weniger zeitraubenden Operationen mehr Daten bewegen. Bei zu großen Blöcken kann dagegen der Speicherplatz knapp werden. Denn jede Datei benötigt natürlich mindestens einen Block. Viele Dateien können aber nicht die Größe der von ihnen verwendeten Blöcke ganz ausnutzen.

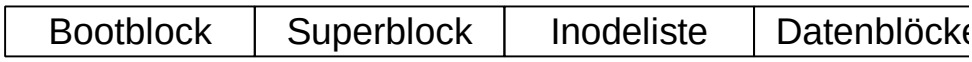


Bild 7.1.5-1 Physikalischer Aufbau eines Dateisystems

Am Anfang des Dateisystems steht der **Bootblock**. Physikalisch befindet sich der Bootblock meistens im ersten Sektor einer Partition. Er enthält den Bootstrap-Code, der beim Hochfahren eines UNIX-Rechners in den Speicher gelesen wird. Er lädt bzw. initialisiert das Betriebssystem.

Der Aufbau des Dateisystems wird im **Superblock** beschreiben. Wieviel Dateien das Dateisystem aufnehmen kann und wo noch freier Platz innerhalb des Dateisystems zu finden ist, ist im Superblock gespeichert (siehe auch Kapitel 7.2.1).

Die **Inodeliste** ist ein Inhaltsverzeichnis der Inodes im Dateisystem. Der Systemverwalter gibt beim Konfigurieren eines Dateisystems die Größe der Inodeliste an. Der Systemkern greift auf die Inodes über einen Index zur Inodeliste (engl.: indirection-node list) zu. In einem Dateisystem gibt es eine Wurzelinode. Über diese Wurzelinode ist nach dem Systemaufruf `mount` die Katalogstruktur des Dateisystems ansprechbar.

Die **Datenblöcke** beginnen nach der Inodeliste. Sie enthalten neben echten Daten auch Verwaltungsinformationen. Ein einmal zugeordneter Datenblock kann nur zu einer einzigen Datei gehören !

7.1.6 Tabellen des Dateisystems

Der Systemkern verfügt zur Verwaltung der Dateien über die folgende Datenstrukturen:
?? die Inodentabelle
?? die Dateitabelle
?? die Benutzer-Filedeskriptortabelle.

Erzeugt ein Prozeß eine neue Datei, so weist der Systemkern ihr eine freie Inode zu. Wie bereits erwähnt, werden Inodes innerhalb des Dateisystems in der Inodeliste gespeichert. Beim Bearbeiten einer Datei überträgt der Systemkern jedoch den Inode in eine im Speicher liegende **Inodentabelle**. Die Inodentabelle ist dabei nichts anderes als eine verkettete Liste von Inodes im Speicher.

Jedem Prozeß wird eine **Benutzer-Filedeskriptortabelle** zugeordnet. Im Gegensatz dazu ist die **Dateitabelle** eine globale Datenstruktur des Systemkerns. Beim Zugreifen eines Prozesses mit `open` oder `creat` auf eine Datei wird vom Systemkern in der Dateitabelle und der Benutzer-Filesdeskriptortabelle ein Eintrag erzeugt. Dieser verweist auf den Inode der Datei.

Die Tabellen beschreiben also den Zustand einer Datei und die darauf ausgeführten Zugriffe durch den Benutzer. Die **Dateitabelle** enthält:

?? den Offset in Byte zum Dateianfang für den nächsten `read`- bzw. `write`-Befehl des Benutzers

?? die Zugriffsberechtigungen für den Prozeß, der die Datei eröffnet hat.

Die **Benutzer-Filedeskriptorentabelle** identifiziert alle offenen Dateien eines Prozesses.

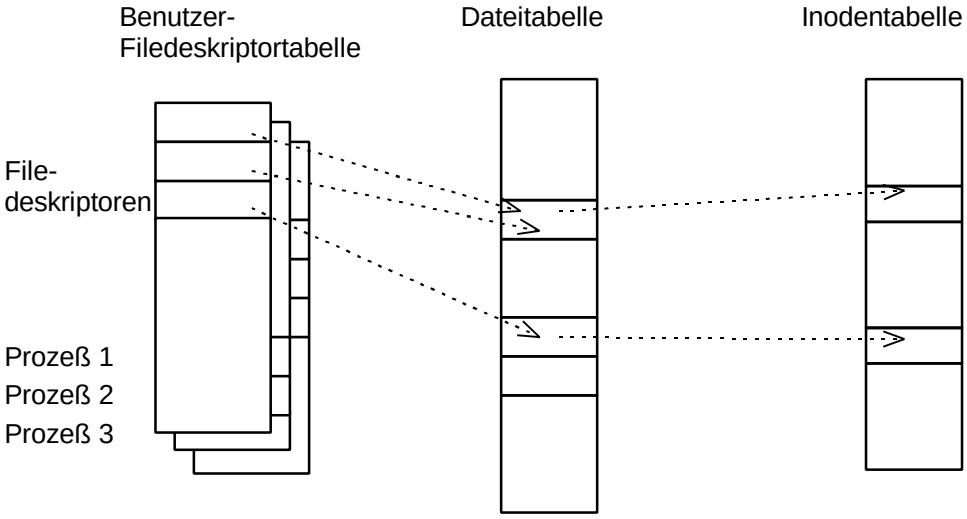


Bild 7.1.6-1 Benutzer-Filedeskriptortabelle, Dateitabelle und Inodentabelle

Bild 7.1.6-1 zeigt diese Tabellen und ihre Beziehung untereinander. Tätigen Prozesse die Aufrufe `open` und `creat`, so gibt der Systemkern einen **Filedeskriptor** (Indexwert) zurück. Dieser Indexwert ist nichts anderes als ein **Index für die Benutzer-Filedeskriptortabelle**. Bei den folgenden `read`- oder `write`-Aufrufen der Prozesse, wird der Filedeskriptor vom Systemkern zum Zugriff auf die Benutzer-Filedeskriptortabelle verwendet. Er verfolgt die Zeiger auf die Datei- und Inodentabelle. Mit dem Inode aus der Inodentabelle schließlich findet der Kern die Daten der Datei.

7.1.7 Das Blockdepot

Der Kern pflegt Dateien auf Massenspeichern und gestattet Prozessen, dort neue Informationen abzulegen bzw. gespeicherte Informationen wieder abzurufen. Will ein Prozeß auf den Inhalt einer Datei zugreifen, so überträgt der Kern die Daten in den Speicher. Dort kann der Prozeß die Dateien prüfen und ändern sowie die Rückspeicherung in das Dateisystem fordern.

Der Kern könnte nun für alle Zugriffe auf das Dateisystem direkt auf Platte lesen und schreiben. Durch den langsamen Plattenzugriff würden aber die Antwortzeit und der Durchsatz des Systems sehr niedrig sein. Der Kern versucht deshalb, die Anzahl der Plattenzugriffe möglichst niedrig zu halten. Zu diesem Zweck unterhält der Kern einen Pool interner Puffer im Hauptspeicher. Das sogenannte Blockdepot enthält die Daten der in letzter Zeit von der Platte gelesenen Datenblöcke.

7.2 Interner Aufbau des Dateisystems

So lange eine Datei existiert, ist einer Datei einem Inode und auch die Plattenblöcke mit den Daten festzugeordnet. In diesem Kapitel soll nun gezeigt werden, wie der Kern beim Erzeugen einer Datei einen Inode und Plattenblöcke dieser zuordnet und wieder freigibt.

7.2.1 Der Superblock

Der Superblock beschreibt den Aufbau des Dateisystems. Eine Kopie des Superblocks befindet sich ständig im Speicher. Der Kern schreibt den Superblock, falls er verändert wurde, in periodischen Abständen auf die Platte zurück, so daß er immer mit den aktuellen Daten im Dateisystem übereinstimmt. Der Superblock enthält folgende Felder:

- Größe des Dateisystems
- Anzahl der freien Blöcke im Dateisystem
- Liste der freien Blöcke im Dateisystem
- Index auf den nächsten freien Block in dieser Liste
- Größe der Inodeliste
- Anzahl der freien Inodes im Dateisystem
- Liste der freien Inodes im Dateisystem
- Index auf den nächsten freien Inode in dieser Liste
- Sperrkennzeichen für die Listen freier Blöcke und Inodes
- Flag für durchgeführte Änderungen im Superblock

7.2.2 Attribute in Inoden

Wie erwähnt, ist zu einer Datei in einem UNIX-System immer ein Inode zu geordnet. Häufig wird ein Inode auch als Dateikopf bezeichnet. Der Inode enthält alle nötigen Verwaltungsinformationen einer Datei, die ein Prozeß zum Zugriff auf diese benötigt.

Inodes werden im Dateisystem gespeichert und zur Bearbeitung in den Speicher geladen. Zu jedem Dateinamen gehört eine Inode. Zu einem Inode können aber mehrere Dateinamen gehören. So ist es möglich mit dem Systemaufruf `link` meherer Dateinamen für eine Datei bzw. Inode zu definieren. Diese beziehen sich dann natürlich alle auf denselben Inode. Da es eine Datei physikalisch nur einmal gibt, wird sie auch nur von einem Inode beschrieben.

Inodes bestehen in einer statischen Form auf der Platte, der Kern liest sie zur Bearbeitung in den Speicher. Platteninodes bestehen aus den folgenden Feldern:

- **Dateibesitzer**
  Die Eigentümerschaft liegt sowohl bei einzelnen Benutzern als auch bei Benutzergruppen. Sie definiert die Menge von Benutzern, die Zugriffsrecht auf eine Datei haben. Der Superuser hat das Zugriffsrecht auf alle Dateien im System.

- **Dateityp**
  Dateien können von folgenden Typen sein:
  - normale Dateien
  - Kataloge
  - Zeichen- oder Blockdateien
  - Pipes

?? **Dateizugriffsrecht**
Das System schützt Dateien nach drei Klassen:
- Eigentümer der Datei
- Gruppeneigentümer
- andere Benutzer

Jede Klasse kann das Recht zum Schreiben, Lesen und Ausführen einer Datei besitzen. Die Rechte können individuell vergeben werden. Da Kataloge nicht ausgeführt werden, bedeutet das Recht zum Ausführen eines Katalog die Bewilligung zur Suche nach einem bestimmten Dateinamen in diesem Katalog.

?? **Zeitstempel**
Er enthält den Zeitpunkt des letzten Zugriffs auf die Datei, ihrer letzten Änderung, sowie der letzten Änderung des Inodes.

?? **Anzahl der Verweise auf die Datei**
Sie ist gleichbedeutend mit der Anzahl der Namen der Datei.

?? **Pointern auf Plattenblöcke, welche die Datei benützt**
Der Benutzer sieht die Dateien als ein logischen Bytestrom an. Intern speichert der Kern sie aber in nichtzusammenhängenden Plattenblöcken. Der Inode muß also Pointer enthalten, welche auf die von der Datei belegten Plattenblöcke verweisen. Im UNIX System V enthält ein Inode zu diesem Zweck 13 Pointer.
Ein Inode einer Gerätedatei benötigt natürlich keine Pointer. Er besitzt anstatt der Pointer eine Major- und Minorgerätenummer.

?? **Dateigröße**
Jedes Byte ist innerhalb der Datei durch Abzählen der Byte erreichbar. Das erste Byte hat den Offset 0 das zweite Byte den Offset 1. Somit ist die größte Offsetangabe immer um 1 kleiner als die Dateigröße. Würde ein Benutzer z.B. eine Datei erzeugen und nur ein Datenbyte bei Offset 1000 in die Datei schreiben, so wäre die Dateigröße 1001 Byte !

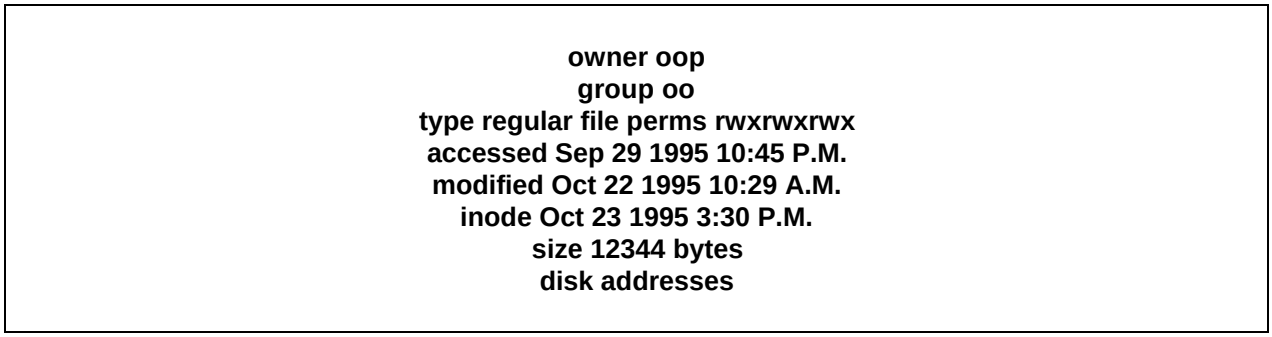


Bild 7.2.2-1 Platteninode einer Beispieldatei.

Zu beachten ist aber, daß ein Inode jedoch nicht die Pfadnamen, die auf die Dateien verweisen, enthält.

Der Inhalt einer Datei ändert sich nur beim Schreiben von Daten in diese. Dagegen wechselt der Inhalt des Inodes beim Verändern der Daten sowie beim Ändern von Eigentümer, Zugriffsrecht und Verweisen. Das Verändern des Dateiinhalts führt automatisch zum Ändern des Inodes. Eine Änderung des Inodeinhalts muß jedoch nicht notwendigerweise eine Änderung des Dateiinhalts zur Folge haben.

Zusätzlich zu den Feldern der Platteninode enthält das Speicherabbild des Inodes noch folgende Felder:

?? **Status des Inodes.** Er definiert den Umgang mit dem Inode. Er gibt an:
ob der Inode gesperrt ist.
ob ein Prozeß auf die Freigabe des Inodes wartet.
ob das Speicherabbild des Inodes vom Plattenabbild abweicht.

(Folge einer Änderung des Inodes)
ob das Speicherabbild der Datei vom Plattenabbild abweicht.
(Folge einer Änderung der Datei)
ob die Datei ein Montagepunkt ist

?? **Logische Gerätenummer** des Dateisystems, welches die Datei enthält.

?? **Inodenummer.** Da die Inodes auf der Platte in einem linearen Array abgelegt sind, ermittelt der Kern die Inodenumer aus der Position des Platteninodes in diesem Array. Der Platteninode selbst benötigt dieses Feld nicht.

?? **Pointer auf andere Speicherinodes**. Der Kern verknüpft Inodes im Speicher in zwei verketteten Listen, in einer sogenannten „Hashqueue“ und in einer „freien Liste“. Sucht der Kern nach einem bestimmten Inode, sucht er in der Hashqueue nach dieser. Benötigt der Kern dagegen einen freien Inode, so entnimmt er den ersten freien Inode aus der freien Liste. Eine Hashqueue ist dabei bestimmt durch die logische Gerätenummer der Inode sowie die Inodenummer. Der Kern kann immer nur eine Kopie eines Platteninode im Speicher halten.

?? **Referenzzähler**. Er gibt an, wie oft die Datei mit open im Augenblick geöffnet wurde.

7.2.3 Zuordnung von Plattenblöcken zu einer Datei

Schreibt ein Prozeß Daten in eine Datei, so muß der Kern Plattenblöcke aus dem Dateisystem für direkte und manchmal auch indirekte Blöcke zuteilen. Der Superblock des Dateisystems enthält ja eine Liste der freien Blöcke im Dateisystem.

Das Dienstprogramm **mkfs** (Erzeugen Dateisystem = make file system) organisiert die Datenblöcke eines Dateisystems in einer verketteten Liste. Dabei zeigt jeder Verweis der Liste im Superblock auf einen Plattenblock, welcher ein Array mit Nummern freier Plattenblöcke im Dateisystem enthält. Zusätzlich enthält der Plattenblock in seinem Array einen Eintrag, der die Nummer des nächsten Blocks in der verketteten Liste enthält. Dieser Block enthält natürlich wieder ein Array mit Nummern freier Plattenblöcke. Bild 7.2.3-1 zeigt die verkettete Liste, wobei der erste Block die freie Liste im Superblock darstellt und die anderen Blöcke in der verketteten Liste weitere freie Blocknummen enthalten.

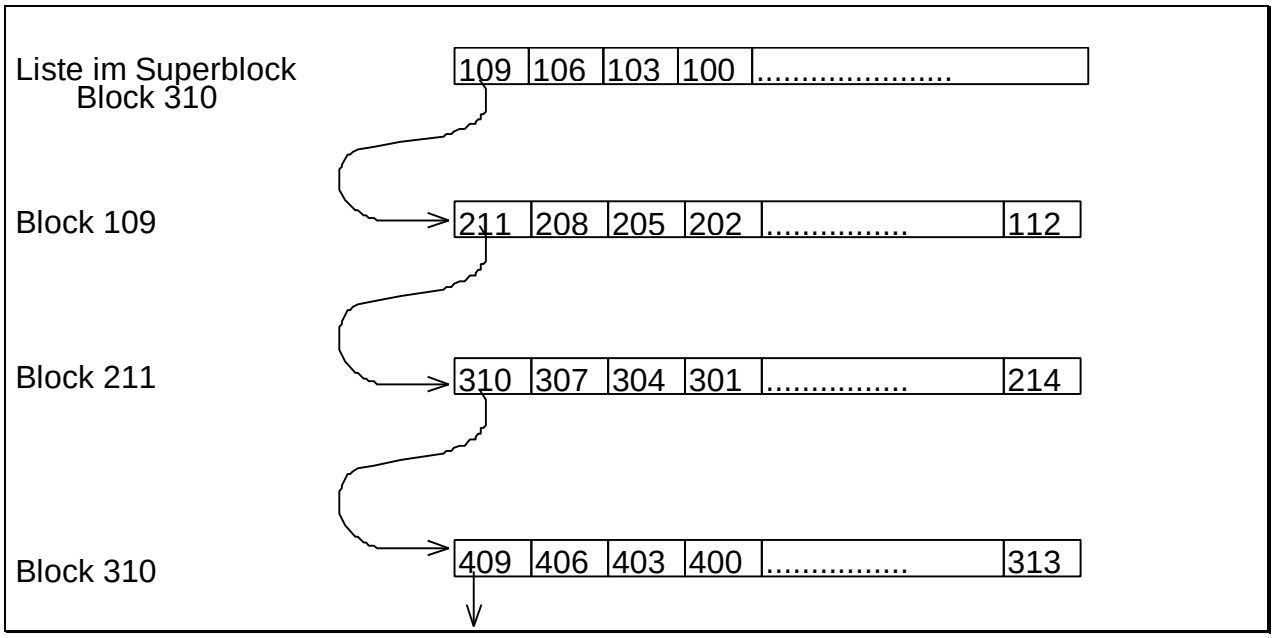


Bild 7.2.3-1 Verkettete Liste freier Plattenblocknummern

Will der Kern nun einen Block aus dem Dateisystem einer Datei zuteilen so nimmt er den ersten verfügbaren Block aus der Liste im Superblock. Einmal zugeteilt, kann dieser Block erst wieder neu zugeteilt werden, wenn er vorher freigegeben wurde.

Ist nun der zugeteilte Block der letzte verfügbare im Superblock, so behandelt ihn der Kern als Pointer auf einen weiteren Block mit einer Liste freier Blöcke. Er liest das Array diesen Blockes und füllt das Array im Superblock mit der neuen Liste von Blocknummern auf. Danach besitzt die Liste im Superblock wieder Nummern von Blöcken, welche frei und zu einer Datei zugeteilt werden können.

Der Kern fährt dann mit der Bearbeitung der ursprünglichen Blocknummer fort. Er teilt einen Puffer für den Block zu und löscht den Pufferinhalt. Jetzt ist der Plattenblock zugeordnet und der Kern hat einen Puffer, um damit zu arbeiten. Verfügt das Dateisystem über keine freien Blöcke mehr, so erhält der aufrufende Prozeß eine Fehlermeldung.

7.2.4 Andere Dateitypen

UNIX unterstützt noch zwei weitere Dateitypen: **Pipes und Spezialdateien**.

Eine **Pipe**, manchmal auch als fifo (first-in-first-out) bezeichnet, unterscheidet sich von normalen Dateien dadurch, daß ihre Daten flüchtig sind. Sind Daten einmal aus einer Pipe gelesen, können sie nicht noch einmal gelesen werden. Auch werden die Daten nur in der Reihenfolge gelesen, in der sie geschrieben wurden. Das System läßt keine Abweichung von dieser Regel zu. Der Kern speichert die Daten in einer Pipe auf die gleiche Weise, wie er sie in einer normalen Datei ablegt.

Der Unterschied zwischen einer gewöhnlichen Datei und einer Pipe besteht darin, daß die Pipe aus Effizienzgründen nur die direkten Blöcke eines I-Node verwendet. Der Kern baut aus den direkten Blöcken der I-Node einen Ringpuffer auf und führt einen Schreib- und Lesezeiger zum Aufrechterhalten der FIFO-Reihenfolge.

Der letzte Dateityp im UNIX-System sind schließlich **Spezialdateien**. Dazu gehören Blockgerätedateien und Zeichengerätedateien. Beide Dateiarten spezifizieren Geräte. Die Inodes der Dateien haben also keinen Bezug auf irgendwelche Daten. Stattdessen enthalten die Inodes zwei Zahlen, die als Major- und Minor-Gerätenummer bezeichnet werden. Die Major-Gerätenummer gibt die Numer des Geräts an.

7.3 Literatur zum Dateisystem

[Bach 91] UNIX-Wie funktioniert das Betriebssystem?; Maurice J. Bach; Hanser Verlag 1991.
[Gulbins 88] UNIX Version 7, bis System V.3; Jürgen Gulbins; Springer-Verlag 1988.
[Tanenbaum] Moderne Betriebssysteme, Hanser und Prentice Hall, 1995

8 Prinzipien der Ereignisbearbeitung

Was ist ein Ereignis? Ein Ereignis ist ein Signal an ein System, das eine bestimmte Behandlung erfordert. Zur Bearbeitung von Ereignissen in einem Computersystem kommen zwei Prinzipien zum Einsatz:

?? Polling (engl. = (Volks)Befragung)
?? Interrupt (engl. = Unterbrechung)

8.1 Polling-Prinzip

Zu bestimmten Zeiten wird überprüft, ob ein Ereignis eingetreten ist und bearbeitet werden muß. Programmtechnisch handelt es sich um Schleifen mit einer Zustandsabfrage. Diese Methode ist zwar sehr einfach, bringt aber auch wesentliche Nachteile mit sich:

- Die CPU ist durch das Warten auf das Ereignis belegt
- Falls in der Warteschleife mehrere Aktionen ausgeführt werden, so ist im ungünstigsten Fall die Ereignisbearbeitung die Aktion, die am spätesten ausgeführt wird. D. h. es vergeht eine bestimmte, nicht konstante Zeit, bis das Ereignis bearbeitet wird.
- Die Bearbeitungsreihenfolge aufeinanderfolgender Ereignisse ist durch die Programmstruktur festgelegt.

Das bedeutet, daß das Polling-Prinzip z.B. nicht eingesetzt werden kann, um Personen im Gefahrenbereich einer Werkzeugmaschine sinnvoll zu schützen oder eine zeitoptimierte Datenübertragung ohne Datenverlust durchzuführen. Deshalb wird dieses Verfahren nicht weiter betrachtet. Haupteinsatzgebiet für Polling ist die Meßwert-erfassung und -verarbeitung nach dem Produzenten-Verbraucher-Modell (siehe DV3).

8.2 Interrupt-Prinzip

Bei einem Interrupt handelt es sich um ein Ereignis, das asynchron auftritt (Ausnahme Timer-Interrupt = synchron), die aktuelle Bearbeitung unterbricht und eine spezielle Interrupt-Behandlung einleitet. Sobald die Interrupt-Behandlung beendet ist, wird die unterbrochene Bearbeitung fortgesetzt. In der Implementierung kann die Interruptbearbeitung mit einem Unterprogrammaufruf verglichen werden. "Unterprogramme", die die Bearbeitung eines Interrupts ausführen, werden als **Interrupt-Service-Routinen** oder **Interrupt-Handler** bezeichnet. Man unterscheidet zwei Arten von Interrupts:

?? Software-Interrupt (Ausgelöst durch Programmcode)
?? Hardware-Interrupt (Ausgelöst durch Peripheriegerät z.B RS-232 Schnittstelle)

Die Verwendung der Interrupt-Technik zur Ereignisbearbeitung benötigt zwar eine aufwendigere Hardware, bietet aber folgende Vorteile:

- Die CPU ist nicht durch das Warten auf das Ereignis belegt
- Das Ereignis kann sofort nach Eintreffen bearbeitet werden
- Die Bearbeitungsreihenfolge verschiedener Ereignisse kann nach ihrem Eintreffen oder bei Konflikten durch eine Prioritätssteuerung festgelegt werden.

9 Ablauf und Steuerung von Interrupts beim PC

Der Ablauf eines Interrupts kann in folgende Schritte unterteilt werden:

- Interrupt-Erkennung durch die Hardware
- Retten des Programmzählers auf den Stack (Segment:Offset)
- Retten des Prozessorstatus auf den Stack (Unterschied zu Unterprogrammaufruf)
- Ermitteln von Segment:Offset der zum Interrupt gehörigen Service-Routine
- Ausführen der Interrupt-Service-Routine
- Wiederherstellung des Ausgangszustandes vor der Unterbrechung durch Rückspeichern von Segment:Offset und Prozessorstatus vom Stack.

9.1 Die Interrupt-Erkennung durch die PC-Hardware

Zur Erkennung von Interrupts stehen an der CPU zwei Eingänge zur Verfügung:

- NMI (Non Maskable Interrupt)
- INTR (Interrupt Request)

Die CPU prüft jeweils am Ende der Ausführung eines Befehles, ob an diesen beiden Eingängen eine Unterbrechungsanforderung vorliegt (NMI = 1 oder INTR = 1). Dabei besitzen die Eingänge unterschiedliche Prioritätsstufen.

Ein **NMI**, also '**N**icht **M**askierbarer **I**nterrupt', ist wie ein Rettungswagen mit Martinshorn und Blaulicht. Aufgrund der Dringlichkeit hat er Vorrang vor anderen Verkehrsteilnehmern und darf nicht aufgehalten werden. Der Begriff Maskierung kennzeichnet in der Computertechnik die Möglichkeit, Signale durch sogenannte Bit-Masken von der Bearbeitung auszuschließen. Beim NMI besteht keine Möglichkeit der Maskierung. D.h., der Programmierer hat keinen Einfluß auf die Bearbeitung eines NON MASKABLE INTERRUPTS. NMIs sind in den meisten Fällen für das Betriebssystem reserviert, um fatale Hardwarefehler zu behandeln (z. B. Speicherfehler, Busfehler, Power-Fail der Festplatte). Treten während der Bearbeitung eines NMI-Interrupts weitere NMI-Interrupts auf, führen diese zu keiner Unterbrechung der laufenden Interrupt Service Routine. Stattdessen wird **ein** NMI zwischengespeichert und die anderen ignoriert. Nach dem Befehl **RTI** (Return from Interrupt), der in Bearbeitung befindlichen Interrupt Service Routine, wird der zwischengespeicherte NMI bearbeitetet.
Der NMI-Eingang der CPU ist für die Übungen nicht relevant und wird deshalb nicht detailierter beschrieben (Thema der Rechnertechnik-Vorlesung).

Ein **INTR** (Interrupt Request) besitzt eine niedrigere Priorität als ein NMI - er kann also immer von einem NMI unterbrochen werden. Wie der Name schon sagt, handelt es sich um eine Anfrage auf Bearbeitung durch eine Interrupt-Service-Routine. Die Bearbeitung wird unter bestimmten Bedingungen nicht gewährt:

?? Der entsprechende Interrupt ist maskiert, also durch eine Bitmaske gesperrt
?? Ein Interrupt höherer Priorität (z. B. NMI) wird gerade bearbeitet.

Zur Maskierung von INTR-Interrupts gibt es zwei Möglichkeiten. Zum einen die globale Freigabe / Sperrung mit dem Flag "**Interrupt Enable**" im Prozessor-Status-Wort (PSW) dr CPU. Mit diesem Flag besitzt der Programmierer z. B. die Möglichkeit, **alle** INTR-Interrupts während der Systeminitialisierung zu sperren. Zum anderen kann ein Interrupt in Peripheriebausteinen, wie z.B. Interrupt-Controller gezielt gesperrt werden.

Neben NMI und INTR existieren zwei weitere Möglichkeiten einen Interrupt auszulösen:

- **Software-Interrupt**
- **Exception**

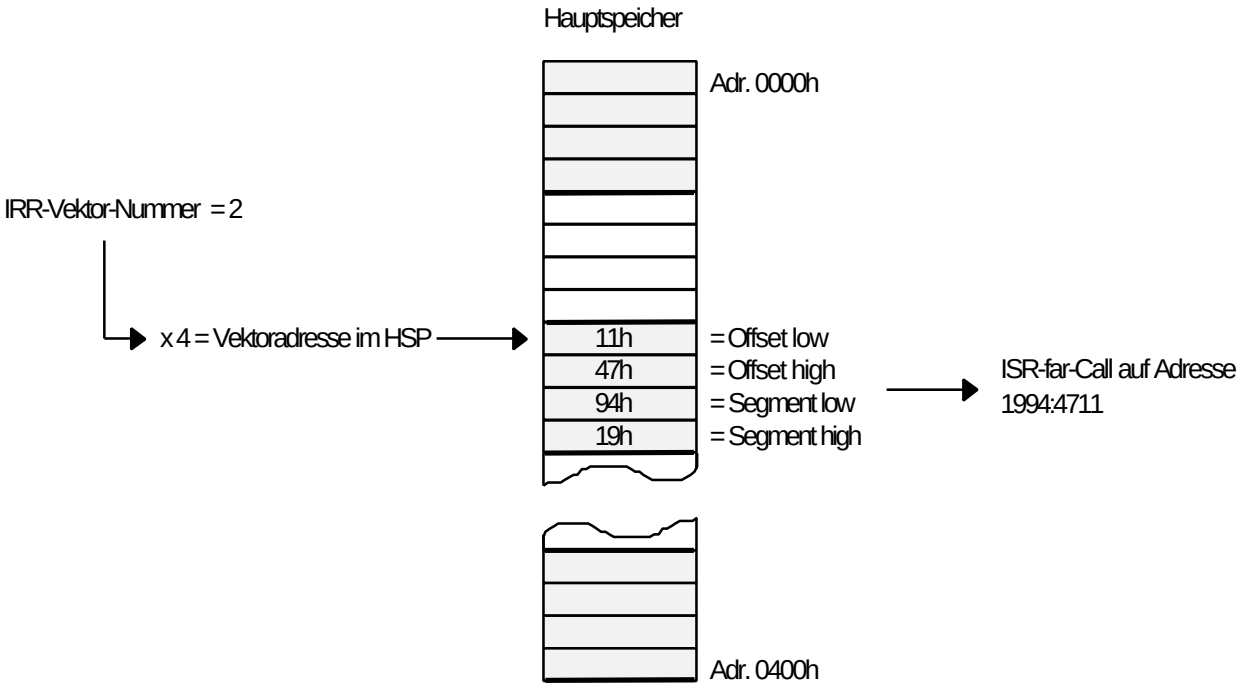
Software-Interrupts sind dadurch gekennzeichnet, daß sie fest mit einer Vektornummer verbunden sind, damit entfällt die Phase der Anforderung und Übertragung dieser Nummer zur CPU.

Exceptions sind Ausnahmesituationen, die während dem Programmablauf auftreten können und eine Sonderbehandlung notwendig machen. Beispiel für eine Exception ist die Division durch Null. Tritt dieser Befehl während der Ausführung eines DOS-Program-mes auf, so wird eine Meldung auf dem DOS-Bildschirm ausgegeben, und das System angehalten. Exceptions werden direkt von der CPU erkannt, die automatisch einen Interrupt mit der ebenfalls fest zugeordneten Vektornummer der Exception auslöst.

9.2 Ermitteln von Segment:Offset der Interrupt-Service-Routine

Nachdem ein Interrupt erkannt wurde, muß die Adresse der Interrupt-Service-Routine ermittelt werden, um die Interrupt-Bearbeitung durchzuführen. Diese Adresse besteht aus zwei Worten bzw. vier Byte (2 Byte Segment, 2 Byte Offset). Eine solche Adresse wird auch als **Interrupt-Vektor** bezeichnet, da es sich um einen Zeiger auf eine Interrupt-Service-Routine handelt. Weil in einem Computersystem mehrere Interrupts zum Einsatz kommen, bietet es sich an, alle Adressen der Interrupt-Service-Routinen nacheinander in einer Tabelle abzulegen. D.h., auf jedem vierten Platz der (Vektor)-Tabelle liegt die Adresse für die nächste Interrupt-Service-Routine. Wenn man nun die Interrupts von 0..n durchnummeriert, kann durch Multiplikation dieser Interrupt-Nummer mit vier, sofort der zugehörige Interrupt-Vektor ermittelt werden.
Wenn dann die Tabelle noch bei Adresse 0 im Hauptspeicher liegt, kann eine Adressierung ohne weitere Offsetaddition erfolgen.

Beispiel:


Hauptspeicher
Adr. 0000h
IRR-Vektor-Nummer = 2
x 4 = Vektoradresse im HSP
11h
47h
94h
19h
= Offset low
= Offset high
= Segment low
= Segment high
ISR-far-Call auf Adresse
1994:4711
Adr. 0400h

Für die Übungen hier eine

Kurzwiederholung zu Zeigern auf Funktionen in C:

Ein Zeiger auf eine Funktion kann ebenso wie eine normale Funktion aufgerufen werden. Es sei `f` ein Zeiger auf eine Funktion:

```
double (* f) (double); /* f zeigt auf eine Funktion mit einem    */
                       /* Rückgabewert vom Typ double und        */
                       /* einem Übergabeparameter vom Typ double */

f = sin                /* damit zeigt f auf die Funktion sin     */
                       /* (ein Funktionsname ist ein konstanter  */
                       /* Zeiger auf die entsprechende Funktion) */
```

Der Aufruf erfolgt durch Deferenzierung des Zeigers. Die Dereferenzierung muß in Klammern stehen, damit sie vor dem Aufruf der Funktion ausgeführt wird:

```
printf ("%f", (*f) (0.5));
```

Beispielprogramm:

```
/* Datei sin.c */

#include <stdio.h>
#include <math.h>

double (*f)(double);

void main (void)
{
  f = sin;
  printf ("%f\n", sin(3.1415));
  printf ("%f\n", (*f)(3.1415));
}
```

In dem Ausdruck `(*f)(3.1415)` wird zunächst der Verweis-Operator ausgewertet, dessen Ergebnis wegen der Zuweisung `f = sin` die Funktion `sin` ist. Dann wird die Funktion tatsächlich aufgerufen, indem der `()`-Operator auf sie angewendet wird. D.h. die Zeilen

```
  printf ("%f\n"; sin (3.1415));
```

und

```
  printf ("%f\n"; (*f) (3.1415));
```

bewirken dasselbe. Die Ausgabe des Programms ist:

```
0.000093
0.000093
```

Ende der Wiederholung

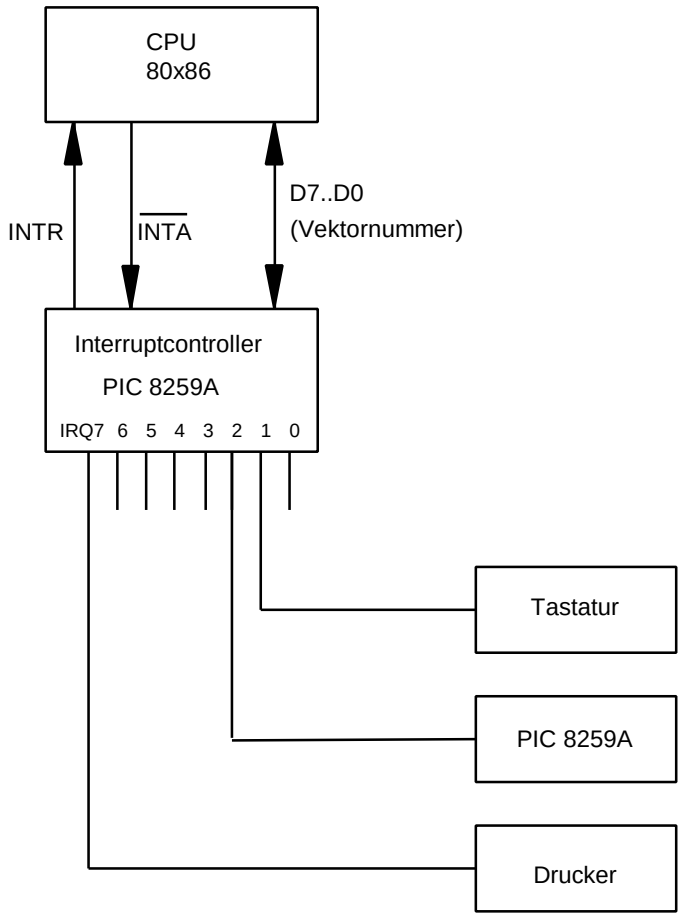
10 Interrupt-System eines Personal-Computers

10.1 Der IRR-Controller PIC 8259A

Die CPU 80x86 besitzt **nur einen** INTR-Eingang. Allerdings sollen mehrere Geräte mit Hilfe der Interrupt-Technik an die CPU gekoppelt werden. Zu diesem Zweck wird der Baustein **PIC 8259A** (**P**eripheral **I**nterrupt **C**ontroller) eingesetzt. Der PIC 8259A bietet folgende Funktionen zur Behandlung von Ereignissen:

?? Interrupt auslösen, sperren und freigeben
?? Prioritätssteuerung bei gleichzeitigen Ereignissen
?? Logische Verknüpfung mehrer Ereignisse mit der INTR-Leitung der CPU

Nachdem der Interruptcontroller auf der Leitung INTR der CPU angezeigt hat, daß ein Ereignis Bearbeitung wünscht, signalisiert die CPU ihre Bereitschaft zur Bearbeitung auf der Leitung INTA (Interrupt Acknowledged). Daraufhin legt der PIC 8259A die zum IRR gehörige Verktornummer auf den Datenbus D0..D7. Die Vektornummer wird dabei aus einem Basiswert (Default = 8) und der IRQ-Nummer durch Addition geblidet. D.h bei einem Basiswert von 8 würde IRQ0 einen Interrupt mit Vektornummer 8 und IRQ7 einen Interrupt mit Vektornummer 15 auslösen (Vgl. 3.2 IRR-Vektortabelle des PC's). Die folgende Abbildung zeigt die Einbindung des Controllers in das PC-System.



Im Standardfall sind die IRQ-Eingänge des IRR-Controllers folgendermaßen belegt:

IRQ	Belegung
0	Timer
1	Tastatur
2	Kaskadierter PIC 8952A
3	zweite serielle Schnittstelle (COM 2)
4	erste serielle Schnittstelle (COM 1)
5	zweiter parallele Drucker (LPT 2)
6	Diskettenlaufwerk
7	erster paralleler Drucker (LPT 1)

10.2 IRR-Vektortabelle des PC's

Die nachfolgend dargestellte Tabelle zeigt die Defaultzuornung von Vektornummern und den Ereignissen, die von der entsprechenden Interrupt Service Routine bearbeitet werden.

Vektor-nummer	Funktion
0	Division durch Null (Exception)
1	Single step interrupt
2	NMI
3	Breakpoint interrupt
4	INTO detected overflow (Exception)
5	Bound range exceeded (Exception)
6	Invalid Op-Code (Exception)
7	Processor extension not available
8	Int. Table limit too small (Exception)
9	Prozessor Extension segm. overrun (Exception)
10 11 12	Intel reserved - do not use Intel reserved - do not use (COM2-Interrupt) Intel reserved - do not use (COM1-Interrupt)
13	Segment overrun (Exception)
14-15	Intel reserved - do not use
16	Prozessor extension error
17-31	Intel reserved - do not use
32-255	Frei für Benutzer

So wird z. B. mittels Vektornummer 0 eine Interrupt Service Routine gestartet, die eine Division durch Null behandelt. Durch Veränderung der Adressen in der Vektortabelle (siehe auch 2.2 Beispiel) können bestehende Funktionen ersetzt oder den Vektornummern 32-255 neue Interrupt Service Funktionen zugeornet werden.

10.3 IRR-Prioritäten des PC's

Falls mehrere Interrupts gleichzeitig auftreten, entscheidet eine Prioritätslogik in der CPU, welcher bearbeitet wird. Dabei gilt folgende Rangfolge:

Priorität	Interrupt
1 = hoch	Instruction Exception
2	Single Step Interrupt
3	NMI
4	Processor Extension Error
5	INTR - Eingang
6 = nieder	Software-Interrupt "INT n"-Befehl