Betriebssysteme I

Sequentielle und enggekoppelte parallele Systeme

Vorlesungsskriptum
Wintersemester 2004/2005

Prof. Dr. Wolfgang Küchlin

Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
<kuechlin@informatik.uni-tuebingen.de>
http://www-sr.informatik.uni-tuebingen.de

Vorwort

Diese zweiteilige Vorlesung liefert eine Einführung in den Aufbau moderner Betriebssysteme und die zugrundeliegende Theorie. Der vorliegende Teil 1 befaßt sich mit sequentiellen und eng gekoppelten parallelen Systemen. Teil 2 hat verteilte Systeme und Netze zum Thema.

Das Standardbeispiel dieser Vorlesung ist UNIX, im parallelen Teil ist es MACH. Für Programmbeispiele und Programmierübungen wird die Sprache C benutzt.

Beim Programmieren von Betriebssystemen stellt sich die wesentliche neue Aufgabe, mehrere nebenläufige Prozesse zu synchronisieren. Die Vorlesung geht speziell auf dieses Thema des nebenläufigen Programmierens (*concurrent programming*) ein, da dies bei parallelen Rechnern auch vom Anwendungsprogrammierer verlangt wird.

Literatur zur Vorlesung

allgemein:

Andrew S. Tanenbaum: „Modern operating Systems“
Prentice Hall 2001

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne:
„Operating System Concepts with Java“
John Wiley and Sons, 2004

William Stallings: „Operating Systems“
Prentice Hall, 2004

UNIX:

Marshall K. McKusick, George V. Neville-Neil:
„The Design and Implementation of the FreeBSD Operating System“
Addison-Wesley 2004

Maurice J. Bach: „The Design of the Unix Operating System“
Prentice Hall, 1986

Realzeit:

Bill O. Gallmeister: „POSIX.4, Programming for the Real World“
O'Reilly & Associates 1995

Hardware:

Hans-Peter Messmer, Klaus Dembowski: „PC Hardwarebuch“
Addison-Wesley 2003

Hintergrund:

Wolfgang Küchlin, Andreas Weber: „Einführung in die Informatik“
Springer 2004

Inhaltsverzeichnis

Kapitel 1

Einführung

Das Betriebssystem ist die Vergabe- und Verwaltungsstelle für alle Betriebsmittel. Zentrale Aufgaben des Betriebssystems umfassen die Verwaltung von Prozessen, Hauptspeicher, Dateien, Ein/Ausgabe, Kommunikation und Rechenzeit. Das Betriebssystem liegt zwischen Hardware und Anwendersoftware und präsentiert dem Anwender eine logisch einfache und möglichst auch geräteunabhängige Schnittstelle zur darunterliegenden Hardware. Es liefert abstraktere, höhere Zugriffsroutinen als es die einzelnen Gerätefunktionen sein können und realisiert so eine (möglichst portable) *abstrakte Maschine* auf dem jeweiligen physischen Gerät, die einfacher und geräteunabhängig handhabbar ist.

Beispiel:
BS Aufruf: `read(`*file*`)`: liest 1 Block aus Datei „*file*“.

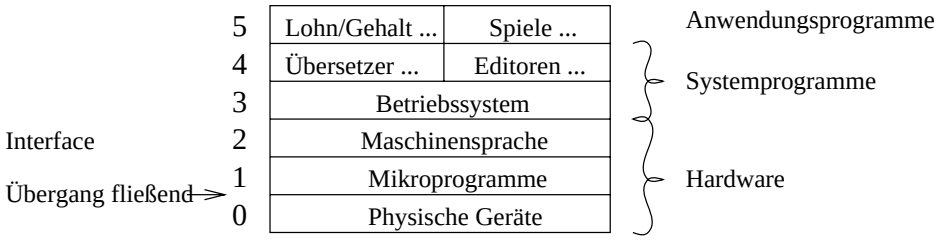


Abbildung 1.1: Schichten eines Rechnersystems

0: Hardware wie sie ist

1: Mikroprogramme setzen Befehle der Maschinensprache um in die Steuerung der einzelnen „gates“.

2: Software-Schnittstelle, die von Ebene 1 realisiert wird. Definiert Architektur unabhängig vom Gerät.

3: BS stellt Software-Architektur unabhängig von Hardware-Architektur dar (Filesystem, Prozesse, Virtuelle Speicher etc.)

4: Utilities (Dienstleistungsprogramme) stellen Software-Entwicklungs-Architektur (Software-Systemumgebung) unabhängig von Systemarchitektur dar.

5: Applikations-Architektur unabhängig von Entwicklungsarchitektur.

Beispiel zur Abstraktion:
Plattenzugriff: Auf Ebene 2:
Lade Geräteregister (spezielle Speicheradresse: wo?)
mit: Plattenadresse: Zylinder, Sektor, Offset (wo?)
Hauptspeicheradresse: (wo?)
Anzahl Bytes zu übertragen (woher?)
Lese/Schreibe

Maschinenebene: READ mit 13 Parametern in 9 Bytes gepackt. Blockadresse, Sektoren pro Zylinder, Schreibverfahren, Lückenlänge zwischen Sektoren, ...

BS-Aufruf: Auf Ebene 3
`read(file,buffer)` liest einen Block auf Datei *file* in buffer.

Bibliotheksaufruf: Auf Ebene 4
`fscanf(file,format)` liest Zeichenreihe gemäß Format (z.B. ein Integer und ein Real) von Datei *file* ein, egal ob kürzer oder länger als Blockgröße.

Auf Ebene 3 in UNIX (siehe „`man read`“)

```
ptroff -t -man /usr/man/man... |lpr - Postscriptprinter
```

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


/* Read() attempts to read nbyte bytes of data from the object
*  referenced by the descriptor fd into the buffer pointed to by buf.
*  The result is the number of bytes actually read.
*/
```

Beispiel:

```
#include stdio.h
          main()
{ char * buf = "Test.\n";
  read (0, buf, 4);                 /* 0 = stdin */
  printf ("%s",buf);
}
```

Hier noch nicht diskutiert: Fehler, Fehlermeldungen.

Kapitel 2

Aufbau und Funktionsweise eines Computers

The Analytical Engine consists of two parts:

1st. *The store in which all the variables to be operated upon, as well as all those quantities which have arisen from the result of other operations are placed.*

2nd. *The mill into which the quantities about to be operated upon are always brought.*

Charles Babbage (1864)

2.1 Einleitung und Überblick

Wir geben einen Einblick in die Prinzipien, nach denen ein heutiger Computer aufgebaut ist. Reale Maschinen können in den Details wesentlich komplexer sein. Das Thema berührt zwei Kerngebiete der Informatik, **Rechnerarchitektur** (*Computer Architecture*) und **Betriebssysteme** (*Operating Systems*). Die internationale Standardliteratur hierzu stammt u. a. von A. Tanenbaum und A. Silberschatz [TG01, Tan01, SG98]; siehe auch [Bra98].

Computersysteme bestehen aus **Hardware** und **Software**. Die Hardware ist fest gegeben, kann angefaßt werden und ist (bis auf den Austausch von Komponenten) unveränderlich. Die Software besteht aus den gespeicherten Programmen, die durch die Hardware ausgeführt werden. Die Software ist unsichtbar und sehr leicht zu speichern, zu ändern oder auszuführen, da sich dies nur in der Änderung von magnetischen (bei Festplatten) oder elektrischen (bei Speichern und Prozessoren) Zuständen der Hardware auswirkt, nicht aber in der Änderung fester Bestandteile.

Typische Hardwarekomponenten sind (neben dem Gehäuse) zunächst der zentrale **Prozessor** (**CPU**) (*central processing unit*) und der **Hauptspeicher** (*main memory*) umgeben von diverser **Peripherie** (*peripheral device*) wie **Festplatten** (*hard disk*), **Monitor**, **Maus**, **Tastatur** (*keyboard*), CD-ROM/DVD **Laufwerk** (*drive*), **Diskettenlaufwerk** (*floppy disk drive*), **Netzwerkkarten** (*network board*) usw. Die Daten werden zwischen den Komponenten über **Verbindungskanäle** (*channel*) übertragen. Wenn viele Komponenten einen einzigen Kanal

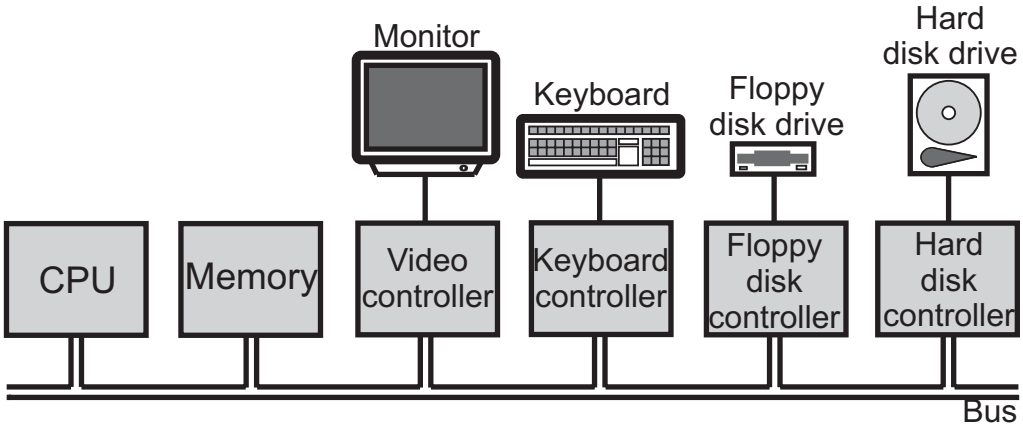


Abbildung 2.1: Architektur eines einfachen Computersystems mit Bus

benutzen bezeichnet man diesen als **Bus**. Jede Komponente außer der CPU ist grundsätzlich über eine elektronische **Steuereinheit** (*controller*) an den Kommunikationskanal angeschlossen. Die Steuereinheit akzeptiert Befehle in digitaler Form (als Zahlen) und setzt sie intern in die nötigen elektrischen Signale (analoge Ströme) um. Aus Sicht der Komponenten und der CPU ist auch der Bus jeweils über einen Bus-Controller angeschlossen. Bei PC-Systemen befinden sich CPU, Hauptspeicher, alle Busse und alle Controller auf der **Hauptplatine** (*motherboard*). Dort befinden sich auch **Steckplätze** (*slot*) für weitere Platinen (wie z. B. die Graphikkarte) oder für Buskabel, die zu Peripheriegeräten führen. Abb. 2.1 zeigt die Architektur eines ganz einfachen Rechnersystems mit Bus, Abb. 2.2 zeigt die zentralen Komponenten Prozessor (CPU) und Hauptspeicher, und Abb. 2.5 zeigt die Architektur eines Intel Pentium-4 Systems mit mehreren aktuellen Bussystemen.

Typische Softwarekomponenten sind die Programme der **Anwendersoftware** (*application software*) zur Lösung von Problemen der externen Welt der Anwender, sowie die Programme der **Systemsoftware** (*system software*) zur Lösung interner Aufgaben im Rechner. Anwendersoftware (z. B. Textverarbeitung, Tabellenkalkulation, Bildbearbeitung, Buchhaltung, Produktionsplanung, Lohn und Gehaltsabrechnung, Spiele) ist der Grund, weswegen der Anwender letztlich einen Rechner kauft; Systemsoftware hilft beim Betrieb des Rechners und bei der Konstruktion der Anwendersoftware. Systemsoftware umfaßt neben Datenbanksystemen, Übersetzern (*compiler*) etc. in jedem Fall das Betriebssystem.

Das **Betriebssystem** (*operating system*) isoliert die Anwendersoftware von der Hardware: Das Betriebssystem läuft auf der Hardware und die Anwendersoftware auf dem Betriebssystem. Das Betriebssystem verwaltet die Ressourcen der Hardware (wie z. B. Geräte, Speicher und Rechenzeit) und es stellt der Anwendersoftware eine abstrakte Schnittstelle (die Systemaufrufschnittstelle) zu deren Nutzung zur Verfügung. Dadurch vereinfacht es die Nutzung der Ressourcen und schützt vor Fehlbedienungen. Betriebssysteme, die es mit diesem Schutz nicht so genau nehmen, führen zu häufigen **Systemabstürzen** (*system crash*).

Anwender-Software
Betriebs-System
Hardware

Es gibt heute eine große Vielzahl von Rechnersystemen. **Eingebettete Systeme** (*embedded system*) verbergen sich in allerlei Geräten, wie z. B. Haushaltsgeräten oder Handys. In Autos ist die Elektronik heute schon für ca. 25% und zukünftig für bis ca. 40% ihres Wertes ver-

antwortlich; aus Sicht der Informatik sind sie rollende Rechnernetze. Übliche Computer kann man grob einteilen in die Klassen der **PC**s (*personal computer*), der **Arbeitsplatzrechner** (*workstation*), der **betrieblichen Großrechner** (*business mainframe*) und der **wissenschaftlichen Großrechner** (*supercomputer*). Bei PC's dominieren Intel Pentium und AMD Athlon Prozessoren und Windows oder Linux als Betriebssysteme, bei Workstations Prozessoren und Varianten des UNIX Betriebssystems von Firmen wie SUN, IBM und HP, bei Mainframes Prozessoren und Betriebssysteme von IBM. Für UNIX Systeme ist es nicht ungewöhnlich, daß sie monatelang ununterbrochen laufen, und bei einem Mainframe System kann die **Ausfallzeit** (*downtime*) auf wenige Minuten pro Jahr begrenzt werden.

2.2 Der Kern des Rechners: von Neumann-Architektur

> *[. . .] when any formula is required to be computed, a set of operation cards must be strung together, which contain the series of operations in the order in which they occur. Another set of cards must be strung together, to call in the variables into the mill, [in] the order in which they are required to be acted upon. Each operation card will require three other cards, two to represent the variables and constants and their numerical values upon which the previous operation card is to act, and one to indicate the variable on which the arithmetical result of this operation is to be placed.*
>
> *Charles Babbage (1864)*

Trotz aller Vielfalt bei den Rechnersystemen herrscht eine gewisse Grundordnung, denn alle Architekturen gehen auf das Prinzip von **Prozessoreinheit** (*processor*, *central processing unit – CPU*), **Speichereinheit** (*storage unit*, *store*, *memory*) und **Programmsteuerung** zurück, das bereits Babbage um 1834 für seine *Analytical Engine* erdacht hatte. **Programme** sind Sequenzen von Befehlen, die der Prozessor nacheinander abarbeitet und dabei auf Daten anwendet, die im Speicher stehen. Dadurch können insbesondere mathematische Funktionen berechnet werden.

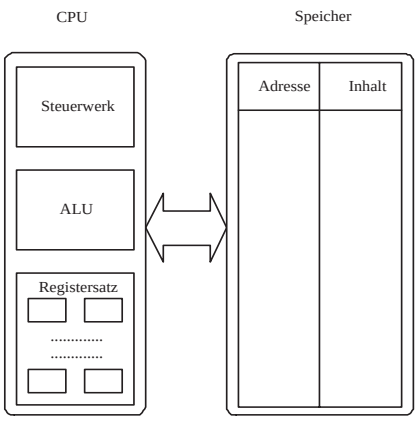


Abbildung 2.2: Von Neumann-Architektur

Für die moderne Welt der elektronischen Computer hat John von Neumann diese Architektur um 1950 verfeinert. Programme werden nun für die Ausführung wie die Daten in binär codierter Form im Speicher gehalten. (Bis weit in die 1970er Jahre wurden sie aber noch von Lochstreifen oder Lochkarten aus in den Speicher eingelesen.) Der Prozessor wurde weiter untergliedert in die **arithmetisch-logische Einheit** (ALU), die u. a. die Grundoperationen der Arithmetik ($+$, $-$, $\times$, $/$) und der Booleschen Algebra (AND, OR, NOT) ausführt, in den **Registersatz** (*register file*), wo die augenblicklich gebrauchten Daten lokal zwischengespeichert werden und in das **Steuerwerk**, das die Befehlsausführung organisiert.

2.2.1 Speicher

Bit 1	Bit 0	
0	0	Zustand 0
0	1	Zustand 1
1	0	Zustand 2
1	1	Zustand 3

Aus technischen Gründen kann die kleinste Speichereinheit (1 **Bit**) nur 2 Zustände speichern – 1 oder 0, z. B. je nachdem, ob in einem Schaltkreis Spannung anliegt oder nicht, wie die Magnetisierungsrichtung an einer Stelle einer Festplatte ist, oder ob auf einer Stelle einer CD eine Vertiefung ist oder nicht. Mit 2 Bits können dann 2×2 Zustände gespeichert werden und so weiter. Wir wollen die Bits immer von rechts her numerieren und dabei wie in der Informatik üblich mit 0 beginnen. Damit stimmt ihre Nummer mit ihrer Stelligkeit (Wertigkeit) bei der Repräsentation einer Dualzahl überein (vgl. Abschnitt 2.5.1 in [KW05]). Das am weitesten rechts stehende Bit 0 heißt deshalb auch das am **wenigsten signifikante Bit** (*least significant bit*), das am weitesten links stehende Bit $n-1$ heißt das **signifikanteste Bit** (*most significant bit*).

Der **Hauptspeicher** (*(main) memory*, *storage*, *store*) ist durch elektronische Bausteine realisiert, die dauernd Strom benötigen, um ihren Inhalt zu bewahren. Er ist logisch als eine Aneinanderreihung von **Speicherzellen** (*cell*) organisiert. Jede Zelle ist ein Paket aus mehreren Bits mit einer **Adresse** (*address*), über die sie zum Zweck des Auslesens ihres Inhalts oder Beschreibens mit einem neuen Inhalt angesprochen werden kann. Da jeder Zugriff unabhängig von der Adresse gleich lang dauert, sprich man von **wahlfreiem Zugriff** und von *Random Access Memory* (**RAM**). Heute sind Speicherzellen zu 8 Bits, genannt 1 **Byte**, allgemein üblich. Ein Byte ist damit auch die kleinste *adressierbare* Speichereinheit. Die Adressen eines Speichermoduls bezeichnen also fortlaufend Byte 0, Byte 1, etc. des Moduls. Größere Einheiten sind Kilobyte $= 2^{10}$ Bytes (1 KB), Megabyte $= 2^{20}$ Bytes (1 MB) und Gigabyte $= 2^{30}$ Bytes (1 GB); die entsprechenden Einheiten für Bits schreibt man Kb, Mb und Gb.

Weitere wichtige Einheiten sind 1 **Wort** (*word*) mit 4 Bytes, 1 Kurz- oder **Halbwort** (*short*) mit 2 Bytes und ein Lang- oder **Doppelwort** (*long*, *double*) mit 8 Bytes. Diese Bezeichnungen werden nicht immer ganz einheitlich gebraucht – Supercomputer rechnen z. B. in Worten zu 64 Bits. Heutige PCs sind noch als 32-bit Architektur (z. B. Intel IA-32) ausgeführt – der Prozessor kann Worte mit 32 Bits auf einmal verarbeiten und als Einheit vom und zum Speicher transferieren. Wir stehen aber am Übergang zu 64-bit Architekturen (z. B. Intel IA-64, Itanium-2 Prozessor), der im Bereich der UNIX/RISC Workstations z. T. bereits vollzogen ist.

Wenn ein Wort aus den Bytes mit den Adressen $n, n+1, n+2, n+3$ besteht, dann ist n die Adresse des Worts. Wir sprechen auch von einer **Speicherstelle** (*location*) für das Wort, die durch die (Anfangs-)Adresse identifiziert ist. In einem Speichermodul sind diejenigen Adressen n mit $n \equiv 0$ (modulo 4) die natürlichen Grenzen, auf denen 4-byte lange Worte beginnen. An solchen Stellen beginnende Worte sind an den **Wortgrenzen** (*word boundary*) **ausgerichtet** (*aligned*).

Ein Wort fängt immer mit dem am weitesten links stehenden signifikantesten Byte an, in dem die Bits der höchsten Stelligkeit stehen, und es endet mit dem am weitesten rechts stehenden am wenigsten signifikanten Byte mit den Bits der niedrigsten Stelligkeit. Die Frage ist nur, beginnt die Zählung $n, n+1, n+2, n+3$ der Bytes links oder rechts? Man macht sich das Problem am besten klar, wenn man sich die Speicherzellen vertikal von oben nach unten angeordnet und numeriert denkt, als Bytes $0, 1, 2, 3, \ldots$. Blickt man dann von links auf den Speicher, wenn man eine Zahl hineinschreibt, so bekommt das signifikanteste Byte die

kleinste Nummer n und das Byte rechts am Ende des Wortes die größte Nummer $n+3$; blickt man von rechts, dann bekommt das Byte links am Anfang eines Wortes die größte Nummer $n+3$ und das Byte rechts am Ende die kleinste Nummer n. Wenn das Byte mit der größten Nummer am Ende steht, heißt die Architektur *big endian*, wenn das kleinste Byte am Ende steht heißt sie *little endian*. (SUN SPARC und IBM Mainframes sind *big endian*, die Intel Familie ist *little endian*.) Dieser Unterschied macht (nur) dann große Probleme, wenn eine Zahl als Wort byteweise zwischen verschiedenen Computern übermittelt wird, wie es z. B. im Internet geschieht. Ein *Datenprotokoll* muß dann festlegen, ob die Zahl in little endian oder big endian Darstellung übertragen wird. C Programmierer können hierzu ein Protokoll wie XJR benutzen, Java hat ein solches Protokoll bereits eingebaut.

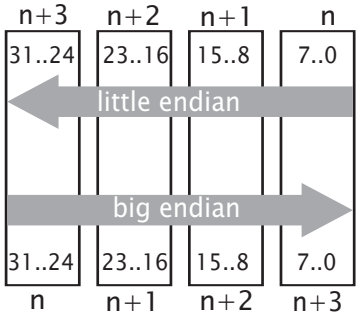


Abbildung 2.3: Unterschied zwischen little und big endian

Im Computer kann also alles immer nur nur in der Form von Bitmustern gespeichert werden. Eine Abbildung von gewöhnlichem Klartext in ein **Bitmuster** (*bit pattern*) nennt man einen **Binärcode** (*binary code*). Je nach dem Typ der Daten (Zahlen, Schriftzeichen, Befehle) benutzt man einen anderen Binärcode; innerhalb einer Programmiersprache ist jedem Typ ein eindeutiger Code zugeordnet. Bei Kenntnis des Typs kann man ein Bitmuster dekodieren und seinen Sinn erschließen. Verwechselt man den Typ, bekommt das Bitmuster eine ganz andere Bedeutung.

Beispiel: Im ASCII Code (siehe Abschn. 2.5.3 in [KW05]) für Schriftzeichen haben wir die Abbildung $0 \mapsto 011\,0000$; $1 \mapsto 011\,0001$; $2 \mapsto 011\,0010, \ldots$, sowie $A \mapsto 100\,0001$; $B \mapsto 100\,0010 \ldots$. Im Binärcode für ganze Zahlen hingegen, der in Java verwendet wird, haben wir die Abbildung $48 \mapsto 011\,0000, 49 \mapsto 011\,0001$; $50 \mapsto 011\,0010$ sowie $65 \mapsto 100\,0001$ und $66 \mapsto 100\,0010$.

Da wir Menschen Dinge gerne mit Namen benennen statt mit numerischen Adressen, kennt jede Programmiersprache das Konzept einer **Variable** (*variable*) als abstraktes Analogon zu einer Speicherstelle. Eine Variable hat einen symbolischen **Namen** (*name*), hinter dem eine Adresse verborgen ist, und der **Wert** (*value*) der Variable ist der Wert des dort gespeicherten Bitmusters. Um diesen erschließen zu können, hat die Variable einen **Typ** (*type*), der bei ihrer Vereinbarung angegeben werden muß. In jeder Programmiersprache gibt es einige fest eingebaute elementare (Daten-)Typen, wie etwa **char** (Schriftzeichen, *character*), **int** (endlich große ganze Zahlen, *integer*) oder **float** (endlich große Gleitkommazahlen, *floating point numbers*, wie wir sie vom Taschenrechner kennen). Jedem fundamentalen Typ entspricht ein Code, der jedem möglichen Wert des Typs ein Bitmuster einer festen Länge (z. B. int $\hat{=}$ 1 word) zuordnet (siehe Bsp. oben; mehr dazu in Abschn. 2.5 in [KW05]).

In Java ist z. B. `int i = 50;` die Vereinbarung einer „Integer“ Variablen mit Namen `i`, die sofort den Wert 50 erhält. Java sorgt selbst für den Speicherplatz, für die Zuordnung des Namens zur Adresse und dafür, daß dort das richtige Bitmuster gespeichert wird (siehe Bsp. oben und Kapitel 6.5 in [KW05]).

Auch Programme können als Daten aufgefaßt und wie solche gespeichert werden. Programme im **Quelltext** (*source code*) sind einfach Texte in einer Programmiersprache wie Java, sie bestehen also aus Schriftzeichen. Programme in **Objektcode** (*object code*) bestehen aus Befehlen, die in der spezifischen Sprache eines Prozessor-Typs, also seinem **Binärcode** (*binary code*, *binary*), geschrieben sind.

2.2.2 Prozessor und Programmausführung

Der Prozessor ist ein fester, endlicher Automat bestehend aus einem **Steuerwerk** (*control unit*) und einer **arithmetisch-logischen Einheit** (**ALU**) (*arithmetic logical unit*) mit einigen beigeordneten Speicherzellen, den **Registern** (*register*). Ein endlicher Automat besitzt eine endliche Menge von verschiedenen Zuständen, zwischen denen er gemäß festen Regeln, ggf. aufgrund von Eingaben, hin- und herwechselt. Die Zustandswechsel in einem Prozessor geschehen zu regelmäßigen Zeitpunkten, die durch einen **Takt** vorgegeben werden. Zu Beginn und am Ende eines Takts sind alle elektrischen Signalpegel im Prozessor in wohldefinierten stabilen Bereichen, so daß der Prozessor insgesamt einen wohldefinierten Zustand annimmt. Während des Zustandswechsels ändern sich dagegen die elektrischen Signale.

Die Zustandswechsel werden durch das Steuerwerk unter dem Einfluß einer Folge von externen Befehlen bewirkt, dem sog. **Programm** (amerik. *program*, engl. *programme*). Das Steuerwerk holt aus dem Speicher nacheinander jeden **Befehl** (**Instruktion**, *instruction*) eines Programms und und interpretiert ihn, d. h. es bringt ihn zur Ausführung. Hierzu setzt es den Befehl in elektrische Signale um, die die ALU und den Datenfluß steuern, wofür ein oder mehrere Takte benötigt werden. Ein Programm macht damit aus dem festen aber *universellen* Automaten jeweils einen speziellen Automaten, der eine ganz bestimmte Funktion auf den Daten ausführt. Man hätte diesen speziellen Automaten auch direkt in Elektronik realisieren können (und dieser hätte dann kein Programm mehr gebraucht), aber der Vorteil der Programmsteuerung besteht darin, daß Programme sehr leicht zu wechseln sind – bei heutigen Systemen kann dies 10 bis 100 Mal pro Sekunde geschehen.

Bei der **von Neumann-Architektur** (vgl. Abb. 2.2) werden Daten und Programme gemeinsam im Hauptspeicher gehalten und bei Bedarf vom und zum Prozessor transferiert. Alle Programme werden von der CPU in einem **fundamentalen Instruktionszyklus** (*basic instruction cycle*) abgearbeitet, auch *fetch-decode-execute cycle* genannt. Ein spezielles Register, der **Befehlszähler** (*instruction counter*), speichert die Adresse des jeweils nächsten Befehls; das **Instruktionsregister** (*instruction register*) speichert den gerade auszuführenden Befehl selbst.

Fundamentaler Instruktionszyklus einer CPU

1. **Fetch:** Hole den Befehl, dessen Adresse im Befehlszähler steht, aus dem Speicher in das Instruktionsregister.
2. **Increment:** Inkrementiere den Befehlszähler, damit er auf die nächste auszuführende Instruktion verweist.
3. **Decode:** Dekodiere die Instruktion, d. h. initiiere den entsprechenden vorgefertigten Ausführungszyklus der Elektronik. Bei Mikroprogrammsteuerung wird hier zur passenden Teilsequenz des Mikroprogramms verzweigt.
4. **Fetch Operands:** Falls nötig, hole die Operanden aus den im Befehl bezeichneten Stellen im Speicher.
5. **Execute:** Führe die Instruktion aus, ggf. durch die ALU. (Bei einem Sprung wird hier ein neuer Wert in den Befehlszähler geschrieben.)
6. **Loop:** Gehe zu Schritt 1 □

Ein Prozessor kann ein Programm nur dann unmittelbar ausführen, wenn es aus Befehlen in seinem speziellen Code, seiner **Maschinensprache** (*machine language*), besteht. Je nach Bauart des Automaten haben Prozessoren nämlich einen spezifischen Typ (z. B. Intel Pentium, Motorola 68k, SUN SPARC) und können nur solche Bitmuster interpretieren, die von Befehlen herrühren, die gemäß diesem Typ codiert wurden. Es gibt **CISC** (*complex instruction set computer*) Maschinensprachen mit komplexeren Befehlen, die oft mehrere Takte benötigen und **RISC** (*reduced instruction set computer*) Sprachen mit nur sehr einfachen Befehlen, die in der Regel in einem einzigen Takt ausgeführt werden können.

Einfache Befehle bestehen z. B. darin, Daten von bestimmten Speicherplätzen (an bestimmten Adressen) in ein Register zu **laden** (*load*); die Daten eines Registers an einer bestimmten Adresse **abzuspeichern** (*store*); Daten zweier Register zu einem Ergebnis zu **verknüpfen** (z. B. zu addieren) und das Ergebnis in einem Register abzulegen; einen **Sprung** (*jump*) auszuführen, d. h. mit einem anderen designierten Befehl fortzufahren (beim **bedingten Sprung** (*conditional jump*) nur dann, wenn der Wert eines Registers Null ist). Verknüpfungsbefehle bestimmen eine arithmetische oder logische **Operation**, die von der ALU ausgeführt wird. Komplexere Befehle (etwa zur Unterstützung einer Java Operation `z = x + y`) können sich direkt auf Adressen im Speicher beziehen, deren Werte vor der Berechnung zuerst herbeigeschafft werden müssen.

In jedem Takt kann eine einfache Aktion im Instruktionszyklus ausgeführt werden, z. B. eine Addition oder ein Umspeichern von Register zu Register; komplizierte Operationen wie eine Multiplikation oder ein Befehl mit externen Operanden dauern i. a. länger. Durch **Fließbandverarbeitung** (*pipelining*) im Instruktionszyklus kann man es erreichen, daß fast in jedem Takt ein Programmbefehl fertiggestellt wird – man dekodiert z. B. den nächsten Befehl bereits parallel zur Ausführung des momentanen Befehls. Bei Sprüngen muß man die Arbeit aber wiederholen, was zu einem Stopp des Fließbands führt (*pipeline stall*). Besonders problematisch ist es, wenn Daten von und zur CPU transferiert werden müssen. Der **von**

Neumann'sche Flaschenhals (*bottleneck*) besteht darin, daß der Prozessor u. U. warten muß, bis ein Transfer abgeschlossen ist. Daher gibt es auf der CPU viele Register und einen weiteren schnellen **Zwischenspeicher** (*cache*) für Instruktionen und Daten.

Eine CPU ist heute in **VLSI Technik** (*very large scale integration*) auf einem einzigen Scheibchen (*chip*) Silizium (*silicon*) etwa von der Größe eines Daumennagels realisiert, mit einigen -zigmillionen Transistoren und mit Grundstrukturen, die weniger als ein millionstel Meter (μm, *micron*) breit sind (zur Zeit ca. 0,1μm). Mit zunehmendem technischen Fortschritt lassen sich immer kleinere Strukturen erzeugen. Dadurch steigt zum einen die Anzahl der Schaltelemente und Leiterbahnen, die auf einen Chip passen und zum anderen wird die Verarbeitungsgeschwindigkeit höher, da sich kleinere Bauelemente auch schneller mit Elektronen füllen lassen.

Heutige Chips lassen sich schon mit ca. 4 GHz (Gigahertz) takten. Bei 1 GHz werden pro Sekunde 1 Milliarde Takte ausgeführt; jeder Takt dauert also 1 ns (Nanosekunde). Während dieser Zeit legt das Licht (und ein elektrischer Impuls) nur 30 cm zurück. Könnten wir den Zustand eines mit 1 GHz getakteten Zählers aus 3 m Entfernung beobachten, so würden wir in jedem Augenblick nur den Wert erkennen, den der Zähler 9–10 Takte vorher hatte.

Zur Zeit gilt noch *Moore's law*, das „Gesetz" (eigentlich eine Beobachtung) von Gordon Moore, einem der Gründer der Fa. Intel, nach dem sich die Anzahl der Transistoren auf einem Chip alle 18 Monate verdoppelt. Hatte der Pentium-2 Chip noch etwa 7 Millionen Transistoren, so hat der Itanium-2 Chip bereits 220 Millionen Transistoren. Allerdings sind der Schrumpfung der Strukturen physikalische Grenzen gesetzt, da man irgendwann (vielleicht um das Jahr 2020) zu Strukturen kommt, die nur noch wenige Atome breit sind.

2.3 System-Architektur der Hardware

Wir geben nun einen kurzen Überblick über die Hardware-Architektur von Rechnersystemen mit Schwerpunkt auf dem PC-Bereich. Wegen der explosionsartigen Entwicklung der Hardware aufgrund des Moore'schen Gesetzes kann dies nur eine Momentaufnahme darstellen. Trotzdem muß man sich ungefähr orientieren können, auch an typischen Zahlenwerten; jeweils aktuelle Werte findet man im Internet.

Die Hardware-Komponenten eines Computersystems werden durch Leitungen miteinander verbunden, damit sie kommunizieren können. Besonders bei kleineren Computersystemen sind diese Verbindungskanäle als Busse ausgeführt. Ein **Bus** (*bus*) ist ein Datenkanal „für alle" (lat. *omnibus*), an den mehrere Einheiten angeschlossen werden können, z. B. mehrere Prozessoren, mehrere Speichermodule oder mehrere **Ein-/Ausgabegeräte** (E/A-Geräte, *input/output devices*, *i/o devices*) wie Festplatten, Drucker etc.

Ein Bus hat mehrere parallele Adreß-, Daten- und Steuerleitungen. Der Prozessor legt z. B. eine Adresse auf die Adreßleitungen und signalisiert auf den Steuerleitungen einen Lese- oder Schreibwunsch. Bei einem Lesewunsch produziert das Gerät, zu dem die Adresse gehört, die entsprechenden Daten auf den Datenleitungen. Bei einem Schreibwunsch entnimmt das adressierte Gerät die Daten von den Datenleitungen und speichert sie ab. Die anderen angeschlossenen Komponenten ignorieren den Datenverkehr, der sie nichts angeht. Gegebenenfalls muß der Zugang zum gemeinsamen Bus von einem **Schiedsrichter** (*arbiter*) geregelt werden (*bus arbitration*), damit Komponenten hoher Priorität nicht warten müssen.

Geräte (inkl. Speichermodule und Busse) werden immer über eine elektronische **Steuerein-**

heit (*controller*) angeschlossen. Damit die Komplexität des Ganzen beherrschbar wird, realisiert ein Controller eine standardisierte **Schnittstelle** (*interface*), wie z. B. ATA/EIDE (*extended integrated drive electronics*) oder SCSI (*small computer system interface*) für Festplatten. Er nimmt relativ abstrakte Befehle entgegen und steuert das angeschlossene Gerät im Detail (siehe dazu auch Kap.5.5.1 in [KW05]). Der Controller erhält seine Aufträge dadurch, daß man über den Bus einen Wert in eines seiner Geräteregister schreibt. Er veranlaßt dann das Gerät zu der gewünschten Aktion und liefert seinerseits Resultatwerte über den Bus zurück. Dies geht am einfachsten dadurch, daß man den Geräteregistern der angeschlossenen Controller ebenfalls Adressen zuteilt, als lägen sie im Hauptspeicher. Wir sprechen von einer

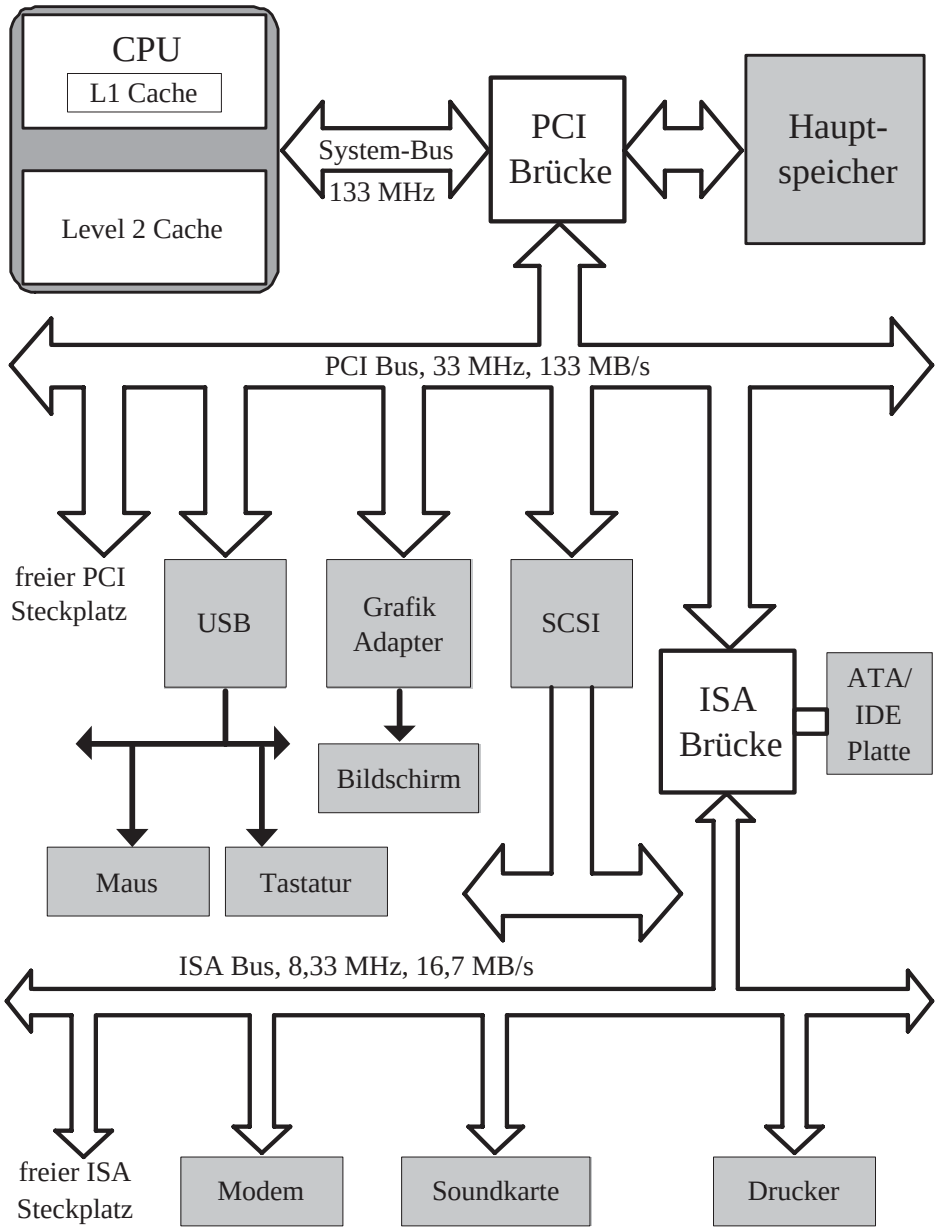


Abbildung 2.4: Architektur eines PC Systems mit mehreren Bussen an Brücken

Speicherabbildung (*memory mapping*) der Geräte.

Festplatten (*hard disk*) dienen der dauerhaften Speicherung großer Datenmengen (bis weit über 100 Gigabyte pro Platte). Dazu wird die Magnetisierung einer dünnen Oberflächenschicht auf einer rotierenden Scheibe durch Schreib-/Leseköpfe geändert bzw. abgetastet. Da die Köpfe und die Scheibe mechanisch bewegt werden müssen, ist die Wartezeit bis zur Datenübertragung vergleichsweise hoch (einige Millisekunden); man spricht hier von **Latenzzeit** (*latency*). Danach ist die Rate wichtig, mit der die Daten ausgelesen werden können; man spricht hier vom **Durchsatz** (*throughput*) der Daten. Er hängt von der Umdrehungsgeschwindigkeit (bis ca. 10.000 UpM) und der Speicherdichte ab und beträgt etwa 100 MB/s. Platten nehmen von der CPU nur einen Lesewunsch nach einem ganzen Datenblock entgegen und liefern diesen Block später im **DMA**-Verfahren (*direct memory access*) direkt im Hauptspeicher an der gewünschten Adresse ab; dabei erhalten sie mit Priorität Zugriff zum Speichermodul. Die meisten Festplatten sind heute mit ATA/ATAPI (auch als IDE bekannt) oder SCSI Schnittstellen versehen, die auch CD-ROM und Bandlaufwerke integrieren können.

Externe Speicher und die Kommunikationskanäle sind sehr viel langsamer als der Prozessor. Deshalb hält man sich schnelle **verborgene Zwischenspeicher** (*cache*) auf dem Chip selbst (*level 1 cache*) und ggf. auch gepackt mit dem Chip auf einem Prozessormodul (*level 2 cache*) oder auf der Hauptplatine neben dem Prozessormodul (*level 3 cache*) in ansteigender Größe und abnehmender Geschwindigkeit. Oft halten sich Programme nämlich eine längere Zeit in einem Speicherbereich auf und bearbeiten in einer Programmschleife immer wieder die gleichen Werte (siehe Kapitel 6 in [KW05]). Jedesmal, wenn ein Wert aus dem Hauptspeicher geholt werden muß, überträgt man **vorgreifend** (*prefetching*) gleich einen etwas größeren Speicherblock (**Cache-Zeile**, *cache line*), in dem der Wert enthalten ist, in der Erwartung, daß die Werte daneben wenig später auch gebraucht werden (die Werte können z. B. auch Instruktionen darstellen). Der Itanium-2 Prozessor hat schon 32 KB L1 Cache, 96 KB L2 und 3 MB L3 Cache *auf der Chipfläche*.

Es ist aus Kostengründen üblich, eine mehrstufige Bus-Architektur zu benutzen. Prozessor (ggf. mehrere), Caches und Hauptspeicher sind durch einen schnellen[1] Bus, den *memory bus* verbunden. (Bei PC Systemen spricht man auch vom System-Bus oder *front side bus*.) Festplatten und ältere Graphikkarten sind an einen langsameren (und billigeren) Bus, z. B. den PCI Bus (*peripheral component interconnect*) angeschlossen. Noch langsamere Geräte wie Drucker, Soundkarten und Modems hängen an einem noch langsameren (und wieder billigeren) Bus, früher z. B. an einem ISA Bus (*industry standard architecture*). Es gibt viele weitere Spezialbusse, wie USB (*universal serial bus*) mit 12 Mb/s zum einfachen Anschluß externer Geräte mit niedrigen Datenraten wie Maus und Tastatur, oder IEEE 1394 „FireWire“ mit garantierter Latenzzeit und Datenraten bis zu 400 Mb/s zum Anschluß von Geräten, die Video- oder Audio-Ströme übertragen müssen.

Die Busse können jeweils durch eine **Brücke** (*bridge*) mit einander verbunden werden. In der herkömmlichen Systemarchitektur der Fa. Intel, der sog. *northbridge / southbridge*-Architektur, sind zwei Brückenchips enthalten. Einer regelt den Verkehr zwischen Prozessor, Hauptspeicher und der Peripherie. Dafür enthält er Controller für Hauptspeichermodule und für einen PCI Bus. Der zweite stellt eine Brücke vom PCI Bus zum ISA Bus her nebst einer Anschlußmöglichkeit für ATA/IDE Festplatten.

[1] Die Datenrate eines Busses ist das Produkt aus seiner Taktfrequenz und der Anzahl der pro Takt übertragenen Bytes (oder Bits), der sog. Breite des Datenpfades. Durch die technische Entwicklung erhöht sich die Taktfrequenz und manchmal auch die Breite eines Busses.

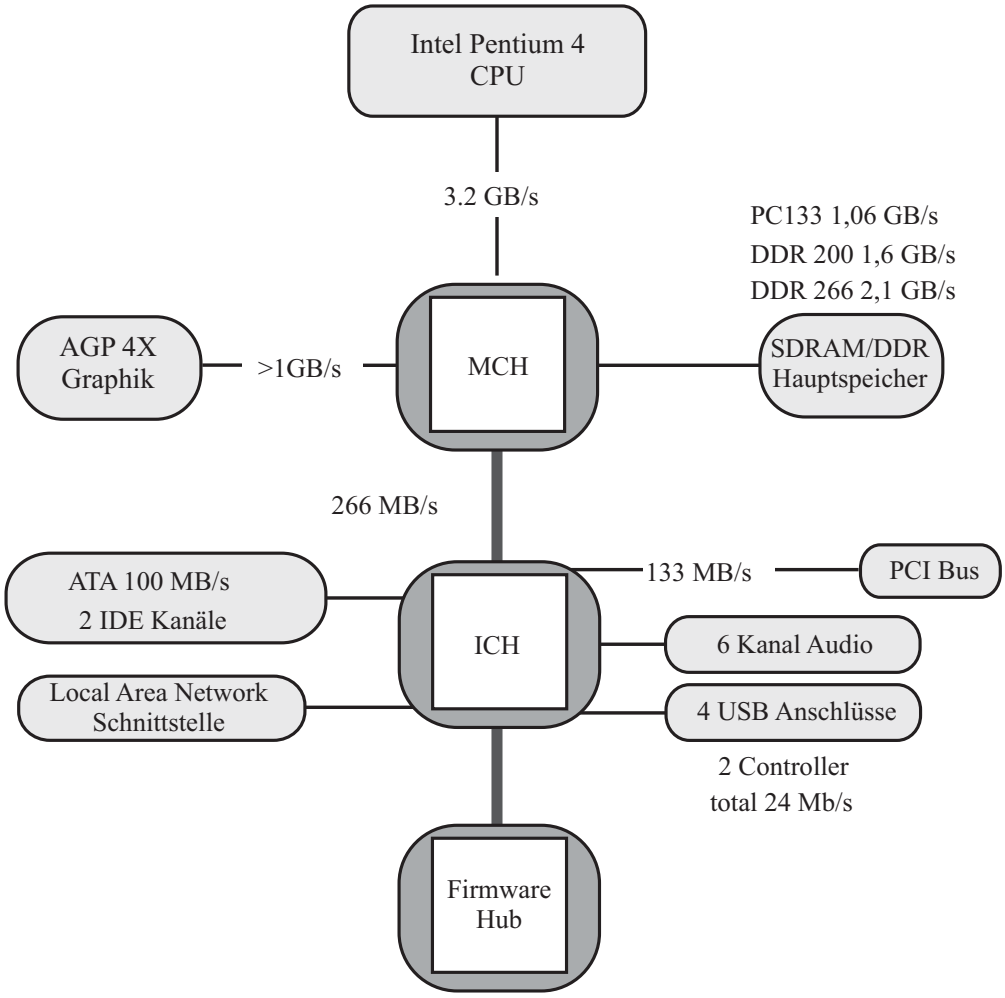


Abbildung 2.5: Architektur eines Pentium-4 PC Systems mit mehreren Bussen an Hubs

Heute kann man mehrere Bus-Controller auf einem einzigen Chip vereinigen und so einen zentralen **Verteilerknoten** (Nabe, *hub*) organisieren. Für die Pentium-4 Systeme mit AGP Anschluß für Hochleistungsgraphik bietet die Fa. Intel zwei Verteilerknoten an (vgl. Abb. 2.5): einen *input / output controller hub (ICH)* für die Verbindung der verschiedenen Peripheriebusse und einen *memory controller hub (MCH)* für die Verbindung des Prozessors mit dem Speicher (bei Itanium-2 bis 6,4 GB/s), der Hochleistungsgraphik und dem I/O hub. Als Hochleistungsverbindung wird hier zukünftig *PCI Express* zum Einsatz kommen. Ein *firmware hub* dient zur zentralen Speicherung der Firmware des Systems, wie z. B. des BIOS (*basic input output system*) mit elementarer Steuersoftware für Tastatur etc.; dort ist auch ein **Zufallszahlengenerator** (*random number generator*) untergebracht, der für Verschlüsselungszwecke gebraucht wird. Dessen Hardware nutzt thermisches Rauschen und folgt daher keinem systematischen Verfahren, das geknackt werden könnte.

2.4 System-Architektur der Software

2.4.1 Schichtenaufbau

Wir haben bereits in Abschnitt 2.1 gesehen, daß ein Computersystem grob in die Abstraktionsschichten *Hardware*, *Betriebssystem* und *Anwendersoftware* gegliedert werden kann. Der Aufbau in **Schichten** (*layer*) oder **Ebenen** (*level*) zunehmender Abstraktion mit definierten **Schnittstellen** (*interface*) zwischen den Schichten ist eine in der Informatik immer wieder angewandte Methode, um in hoch komplexen Systemen eine gewisse Ordnung zu schaffen.[2] Diese Schichtenaufteilung wollen wir jetzt genauer betrachten und weiter verfeinern; Abb. 2.6 gibt einen Überblick. Wir orientieren uns dabei an dem richtungweisenden Werk *Structured Computer Organization* von Andrew [Tan76], der diese Sicht popularisiert hat.

Jede Schicht besteht aus einer Maschine, die nach oben hin eine definierte Benutzerschnittstelle zur Verfügung stellt und ihrerseits die Schnittstelle(n) der darunter liegenden Maschine(n) benutzt. Die Schnittstelle besteht aus einer Ansammlung von Funktionalität, die man irgendwie aufrufen kann. Die Betriebssystemschnittstelle kann z. B. eine Funktion `write(c)` anbieten, die man aufruft, um ein Zeichen auf einen Ausgabekanal zu schreiben. Nur die unterste Maschine ist notwendigerweise in Hardware realisiert. Darüber liegen **virtuelle Maschinen** (*virtual machine*), die i. a. nur in Software existieren.[3] Wir bezeichnen sie auch als **abstrakte Maschinen** wegen ihrer von Details abstrahierenden Benutzerschnittstellen.

Zum Übergang zwischen den Schichten gibt es zwei fundamentale Techniken: **Interpretation** (*interpretation*) und **Übersetzung** (*compilation*). Sei ein **Instruktionsstrom** (Befehlsfolge, *instruction stream*) auf einer Schicht n gegeben. Bei der Interpretation existiert auf der darunterliegenden Schicht $n-1$ eine Maschine (als Hardware oder als lauffähiges Programm), für die die Befehle auf Schicht n lediglich Daten sind, die interpretiert werden und entsprechende Aktionen auslösen. Bei der Übersetzung wird die Befehlsfolge auf Schicht n durch einen **Übersetzer** (*compiler*) zu einer neuen aber äquivalenten Befehlsfolge auf Schicht $n-1$ konvertiert. Die neue Befehlsfolge besteht i. a. aus viel einfacheren Befehlen und ist deutlich länger, aber sie läuft nun direkt auf der Maschine, die die Schicht $n-1$ realisiert.

Die drei untersten Schichten in Abb. 2.6 gehören zum Mikroprozessor. Auf der untersten Schicht befindet sich die **digitale Logik**. Das sind in Silizium realisierte **Schaltkreise** (*circuits*), die in Form von **Logik-Gattern** (*gates*) die Operationen der **Schaltalgebra** (*switching algebra*) ausführen (gemeinhin auch als **Boolesche Algebra** (*Boolean Algebra*) bezeichnet, vgl. Kapitel 16.2 in [KW05]). Jedes Gatter besteht aus mehreren verschalteten Transistoren. Durch Schaltfunktionen, die in Boolescher Algebra dargestellt werden, können insbesondere arithmetische (wie $+$, $\times$, ...) und logische (UND, ODER, NICHT, ...) Operationen auf Zahlen und Bitmustern realisiert werden.

Durch die digitale Logik werden auch die **Mikroprogramme** (*micro programs*) in der darüberliegenden Ebene der **Mikroarchitektur** (*micro architecture*) ausgeführt. Umgekehrt gesprochen steuern die Mikroprogramme die darunterliegenden elementaren Schaltfunktionen individuell an und führen sie in einer bestimmten Reihenfolge aus. Dadurch können komplexere

[2] Die andere Methode, eine Gliederung in abgeschlossene Einheiten oder Module mit definierten Zugangsschnittstellen, haben wir im vorhergehenden Abschnitt 2.3 verfolgt. Beide Methoden werden uns im Verlauf des Buchs immer wieder begegnen.

[3] „Virtuell“ ist in der Informatik eine wörtliche, aber vielleicht etwas falsche Übersetzung des englischen *virtual*, das „im Effekt, aber nicht wirklich“ bedeutet. Eine virtuelle Maschine ist sehr real, sie ist nur nicht unbedingt eine konventionell in Hardware ausgeführte Maschine (kann aber auch dieses sein).

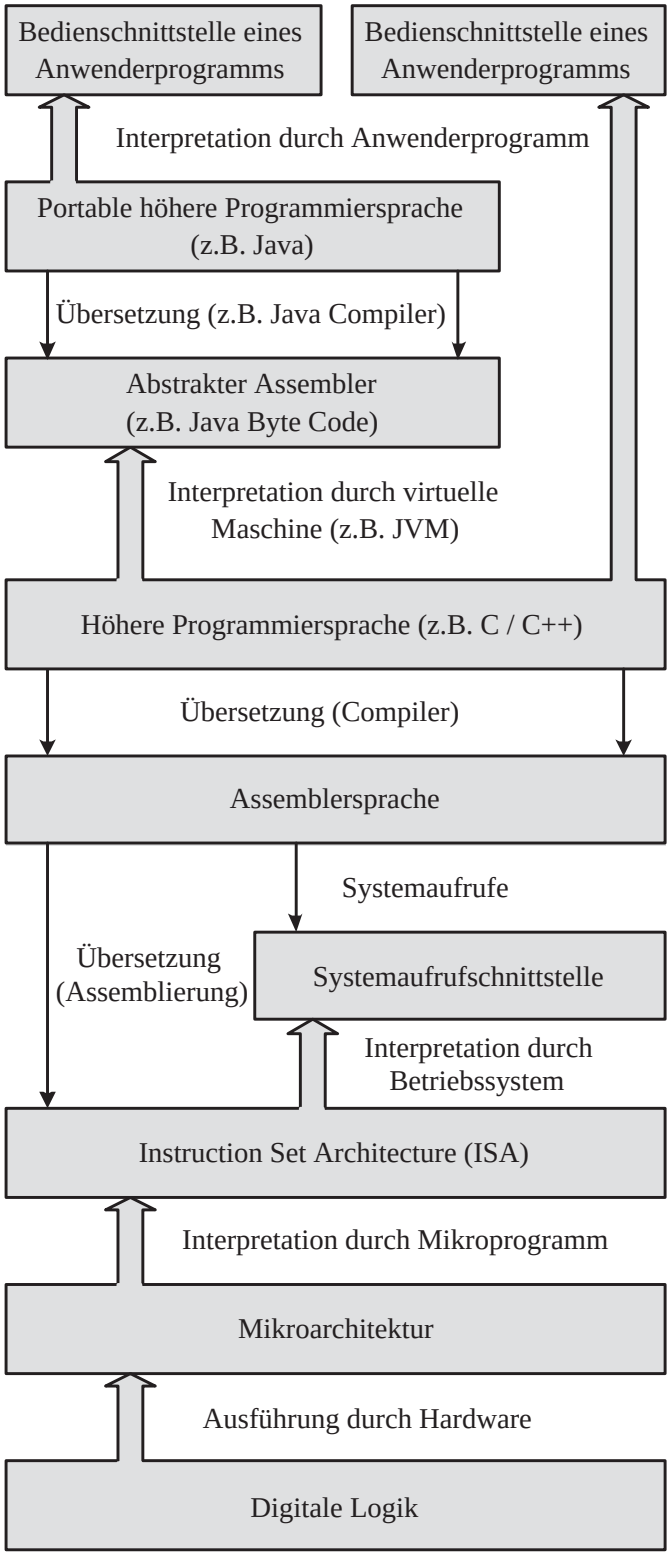


Abbildung 2.6: Schichtenaufbau der Software eines Rechnersystems

Operationen ausgeführt werden. So besteht etwa die Addition zweier Zahlen in der CPU aus 3 Einzelschritten: Bringe die Inhalte zweier Register in die ALU, führe eine Additionsoperation aus, bringe das Ergebnis in ein Register zurück. Die Mikroprogramme heißen auch *firmware*, weil sie nur vom CPU Hersteller selbst geschrieben und deshalb selten geändert werden.

Die darüber liegende Ebene der **Befehlsarchitektur** (*instruction set architecture*) stellt die **Maschinensprache** (*machine language*) zur Verfügung. Die Maschinensprache besteht aus elementaren Befehlen oder **Instruktionen** (*instruction*) im spezifischen Binärcode des Prozessors, wie wir sie in Abschnitt 2.2.2 diskutiert hatten; der oben beschriebene Additionsbefehl ist ein Beispiel.

Durch die Mikroprogrammtechnik lassen sich auf einfacher Hardware komplexe Instruktionssätze realisieren. Außerdem kann z. B. eine CPU neuester Technologie bei Bedarf auch leicht Instruktionen älterer Modelle interpretieren, indem man die passenden Mikroprogramme schreibt. Durch die zunehmende Miniaturisierung unterstützt man heute aber wieder mehr Instruktionen direkt in Hardware. Was bei einem modernen System durch Software und was durch Hardware realisiert wird ist auf den tieferen Schichten von außen schwer zu erkennen; das Konzept der abstrakten, virtuellen Maschinen lehrt uns, daß das im Endeffekt auch egal ist.

Maschinensprachen eignen sich nicht für den menschlichen Gebrauch, da sie zu wenig abstrakt sind. Die meisten Menschen wollen nicht in Binärcode denken, sie wollen Objekte mit Namen wie x und y ansprechen statt über ihre Speicheradressen, und sie wollen komplexe Dinge mit einem einzigen Befehl erreichen. Auf der Abstraktionsebene unmittelbar über den Maschinensprachen sind die jeweiligen **Assembler** angesiedelt. Mit diesem Sammelbegriff bezeichnet man eine etwas abstraktere und verständlichere Variante der jeweiligen Maschinensprache, die sich in beschränktem Umfang schon vom Menschen handhaben läßt (wenn es unbedingt sein muß). Assembler erlauben z. B. symbolische Namen für Daten oder für Sprungziele im Programm, machen aber immer noch den vollen Instruktionssatz der Maschine zugänglich. Ein Assemblerprogramm ist deshalb maschinenspezifisch und muß für jeden Prozessor neu geschrieben werden.

Höhere Programmiersprachen (*high level programming language*) wie ALGOL, FORTRAN, COBOL, C, C++ oder Java sind dagegen speziell für den menschlichen Gebrauch gemacht und betonen die Verständlichkeit der Programme für den Programmierer gegenüber der Effizienz bei der maschinellen Ausführung.[4] Programme in Assembler und allen höheren Sprachen müssen in gleichwertige Programme in Maschinensprache übersetzt werden, bevor eine CPU sie ausführen kann. Bei Assembler-Programmen ist dies ganz besonders einfach, weswegen man auch von **Assemblierung** (*assembly*) spricht. Bei höheren Sprachen muß der Compiler erheblich abstraktere Instruktionen in eine ganze Folge von Assembler-Befehlen übersetzen (er setzt den Effekt eines abstrakteren Befehls aus mehreren Assembler-Befehlen zusammen). Der Compiler ist wieder ein Programm, das bereits früher übersetzt wurde und schon lauffähig ist, vgl. [Wir95]. Kapitel 6 in [KW05] gibt eine Einführung in höhere Sprach-

[4] Die Namen von höheren Programmiersprachen sind häufig Akronyme: FORTRAN steht für *Formula Translator*, ALGOL für *Algorithmic Language*, LISP für *List Processing Language* und Prolog für *Programming in Logic*. Der Name der Sprache C erklärt sich aus der alphabetischen Anordnung der lateinischen Buchstaben, da C als Nachfolger der Sprache B entwickelt worden war. Im Namen von C++ spielte der Entwickler gar auf den Inkrement-Operator ++ von C an, um zu verdeutlichen, daß es sich um ein „Inkrement von C“ bzw. um den „Nachfolger von C“ handelt. Eine Zusammenfassung der diversen Überlegungen und historischen Zufälligkeiten, die Java zum jetzigen Namen verholfen haben, findet sich in der Zeitschrift *JavaWorld* vom Oktober 1996, siehe www.javaworld.com/javaworld/jw-10-1996/jw-10-javaname.html.

konzepte.

Die Sprache „C“ [KR88] war ein historischer Meilenstein, da sie schon eindeutig eine Hochsprache ist, aber noch in solch effiziente Maschinenprogramme übersetzt werden kann, daß sich C auch für die Programmierung von Systemsoftware (**Systemprogrammierung**) eignet und ein Programmieren in Assembler in den allermeisten Fällen unnötig macht. C fand mit dem Betriebssystem UNIX große Verbreitung, das bei der Entstehung zu ca. 90% in C geschrieben war und deshalb erstmals relativ leicht auf verschiedene Rechner portiert werden konnte. (Linux ist ein Derivat von UNIX und läuft ebenfalls auf Maschinen vom PC bis zum Großrechner.)

C++ [Str97] ist eine objektorientierte Erweiterung von C. [Str93] hat C++ als das bessere C propagiert, und wir werden oft C++ als eine Einheit ansehen. Man kann in C++ Code erzeugen, der so effizient ist wie bei C, hat aber bei Bedarf die Strukturierungsmöglichkeiten der Objektorientierung zusätzlich zur Verfügung. C++ unterstützt daher eine ungeheure Bandbreite an Programmiertechniken und -konzepten, wodurch es aber besonders für Anfänger vergleichsweise schwierig zu beherrschen ist.

Insbesondere die höheren Schichten eines Systems sind oft durch **Funktionsbibliotheken** (*library*) realisiert; dies ist auch ein Mittel, um große Softwaresysteme intern zu strukturieren. Eine Funktionsbibliothek ist eine Ansammlungen von Funktionen, die in einer Sprache einer Schicht n geschrieben und bereits vorcompiliert wurden. Ein Beispiel sind Bibliotheken zur Erzeugung von graphischen Oberflächen wie das AWT (siehe z. B. Kapitel 9 in [KW05]). Nun kann man auf Schicht n weitere Programme schreiben, die Funktionen dieser Bibliothek aufrufen; dadurch befindet sich die Bibliothek logisch in einer Zwischenschicht. Nach der Übersetzung in Code der Schicht $n-1$ wird der bereits übersetzte Code der benutzten Bibliotheksfunktionen von einem **Binder** (*linker*) zum Objektcode hinzugefügt und mit ihm zu einer lauffähigen Einheit verbunden.

2.4.2 Das Betriebssystem

Das Betriebssystem (*operating system*) verwaltet zum einen alle Ressourcen eines Rechners und bietet zum anderen allen Programmen eine (relativ) bequem aufrufbare Kollektion von Funktionen zum Zugriff auf diese Ressourcen an. Durch diese **Systemaufrufschnittstelle** ist die Hardware wesentlich einfacher und sicherer zu nutzen als durch direkte Bedienung der Controller-Schnittstellen. Ein Programm, das auf Funktionen des Betriebssystems zugreift, enthält sog. **Systemaufrufe** (*system call*). Anders als bei einem normalen Funktionsaufruf (vgl. Kap.6.9 in [KW05]) wird bei einem Systemaufruf das rufende Programm durch den Prozessor temporär blockiert und stattdessen das Betriebssystem an der gewünschten Stelle aktiviert. Bei der Abarbeitung eines Systemaufrufs werden also Instruktionen ausgeführt, die gar nicht Bestandteil des aufrufenden Programms sind, denn das Betriebssystem *interpretiert* den Systemaufruf, er wird nicht übersetzt. In diesem Sinn stellt auch das Betriebssystem eine virtuelle Maschine dar. Ein Programm hängt also auch von der Betriebssystemmaschine ab, die seine Systemaufrufe interpretiert. Deshalb läuft ein Windows-Programm nicht ohne weiteres auf Linux, auch wenn der Prozessor gleich ist.

Die wichtigste Aufgabe des Betriebssystems ist es, die Ausführung von Programmen zu ermöglichen. Es bündelt ausführbaren Programmcode mit den benötigten Ressourcen (wie Speicherplatz, Dateien und Kommunikationspfaden) zu einem **Prozeß** (*process*) und teilt ihm Rechenzeit zu. Weiter ermöglicht es u. a. die logisch gleichzeitige Ausführung mehrerer

Prozesse (*multiprogramming*) und die gleichzeitige Aktivität mehrerer Benutzer (*timesharing*). Da das Betriebssystem die Ausführung der Anwenderprogramme verwaltet, sagt man manchmal auch, eine Software laufe „unter“ einem Betriebssystem.

Das Betriebssystem enthält **Gerätetreiber** (*device drivers*) zur Bedienung der Controller und es verwaltet die gesamte I/O, d. h. nur das Betriebssystem transferiert Daten von und zu Ein-/Ausgabegeräten. Dazu organisiert es **Ströme** (*stream*) von Bytes zwischen dem Hauptspeicher und den Geräten. Es organisiert auf den Platten **Dateien** (*file*) zur Aufnahme von Daten und kann diese beschreiben oder lesen; es kann Netzverbindungen zu anderen Rechnern herstellen; es kann von Tastaturen lesen und über Graphikkarten auf Monitore schreiben. Es liefert jedem Prozeß auf Verlangen einen **Eingabestrom** (*input stream*) der Zeichen, die für ihn bestimmt sind, und es stellt einen **Ausgabestrom** bereit und leitet dessen Zeichenfolge an eine geeignete Stelle, meist in ein Fenster auf dem Monitor.

Der Schichtenaufbau des Rechnersystems geht innerhalb des Betriebssystems weiter: So läuft z.B. im Netzwerkteil das Verbindungsprotokoll TCP auf dem Internetprotokoll IP (weshalb man TCP/IP schreibt und *TCP over IP* sagt), und das Schnittstellenprogramm eines Betriebssystems (CMD-Tool, UNIX Shell) nimmt die Befehle des Benutzers entgegen und interpretiert sie durch Aufrufe der darunterliegenden Systemfunktionen.

2.4.3 Java und die Virtuelle Java-Maschine JVM

Bei immer größerer und komplexerer Software und sehr schnellen Prozessoren tritt heute die Verständlichkeit, Wartbarkeit und Portabilität von Programmen immer mehr gegenüber der Effizienz durch Maschinennähe in den Vordergrund. Durch die Verwendung höherer Programmiersprachen erreicht man schon eine gewisse Portabilität: Falls für zwei verschiedene CPU's jeweils ein Compiler für die selbe Sprache vorhanden ist, dann kann das Programm in zwei verschiedene Maschinenprogramme übersetzt werden und muß nicht von Hand umgeschrieben werden.

Java geht nun einen Schritt weiter und definiert auf einer höheren Abstraktionsebene die idealisierte **Virtuelle Java Maschine** JVM (*Java Virtual Machine*), siehe [LY96]. Ein Java-Programm wird nur noch in den relativ abstrakten Maschinencode der JVM, den Java **Byte-Code** übersetzt. Auf jedem Rechnertyp wird einmal die JVM mit den Systembibliotheken als Anwendersoftware (z. B. in C) entwickelt und installiert. Die JVM interpretiert dann den Byte-Code, sodaß Java Programme ohne Modifikation überall dort ablauffähig sind, wo schon die JVM installiert wurde. Statt die Installation (Portierung) für jedes einzelne Programm machen zu müssen, macht man sie in Java nur ein einziges Mal für die JVM, und dies ist schon für sehr viele reale Rechner und Betriebssysteme geschehen. Java Byte-Code kann deshalb auch sinnvoll über das Internet geladen und ausgeführt werden.

Durch die Zwischenschicht einer interpretierenden JVM verliert man natürlich Geschwindigkeit. Die JVM kann aber einen *just-in-time compiler* (JIT) benutzen, um den Byte-Code einer zu interpretierenden Funktion zuerst in Maschinencode zu übersetzen und mit größerer Effizienz **grundständig** (*native*) auszuführen. Wegen des Zusatzaufwandes beschränkt man die Übersetzung möglichst auf Funktionen, in denen viel Zeit verbracht wird (*hot spots*). Die JVM wurde aber auch in Hardware realisiert, z. B. als picoJava II Architektur. Diese ist besonders für den Bereich der eingebetteten Systeme interessant, für den Java ursprünglich entwickelt wurde.

Wie wir gesehen haben, sind Programme nicht nur von dem Prozessor abhängig, auf dem sie

laufen, sondern auch vom Betriebssystem. Wenn ein zu portierendes Programm Systemaufrufe enthält, so muß auch das Betriebssystem gleich sein (selbst unterschiedliche Varianten von UNIX unterscheiden sich auf subtile Weise). Der vielleicht größte Wert von Java liegt nun darin, daß es standardisierte Schnittstellen (sog. API – *application programming interface*) zum Betriebssystem hin bereitstellt, die hier für Einheitlichkeit sorgen. Für die Praxis sind diese Schnittstellen und ihre Implementierung in Standardpaketen (Systembibliotheken) von allergrößter Bedeutung, denn die Portierung auf ein anderes Betriebssystem kann viel schwieriger sein, als eine erneute Übersetzung. Java Programme enthalten üblicherweise keine direkten Systemaufrufe, sondern sie benutzen die Klassen der Systembibliotheken, die ihrerseits das Betriebssystem rufen.

Es ist also nicht nur die JVM alleine, sondern es ist die gesamte Java **Laufzeitumgebung** (*runtime*), die für die Portabilität sorgt. Wir nennen hier nur beispielhaft die Gebiete Eingabe / Ausgabe (`java.io`), Ausführung im Browser (`java.applet`), Parallelität (`java.lang.Thread`), Sicherheit (`java.security`), Verbindungen über das Internet (`java.net`, `java.rmi`), Zugriff auf relationale Datenbanken (`java.sql`) und graphische Benutzeroberflächen (`java.awt`).

Kapitel 3

UNIX

Die Schichten des UNIX-Systems:

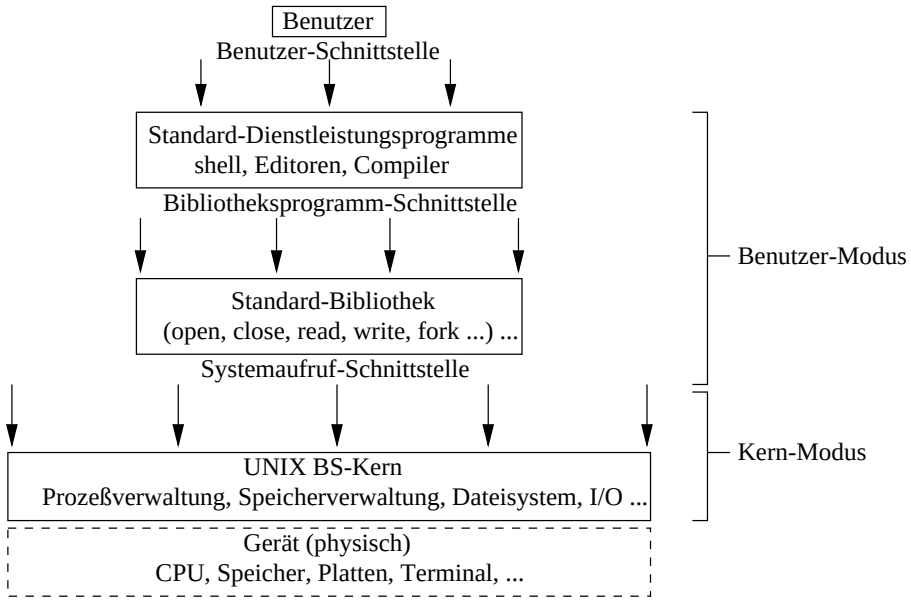


Abbildung 3.1: Der Schichtenaufbau von UNIX

- Aufrufe über die Bibliotheksprogramm-Schnittstelle per C-Prozeduraufruf (siehe Beispiel „read()“).
- Aufrufe über die Systemaufruf-Schnittstelle hinweg durch Aufruf von Assembler-Routinen (je eine per call). Jede as-Routine legt Parameter in Register oder auf Stack, und führt dann „trap“ Instruktion aus. Diese unterbricht das laufende Programm und fährt mit der angegebenen Routine des BS Kerns fort. Das BS verwendet normalerweise einen eigenen Stack zum Ausführen seiner Arbeiten.

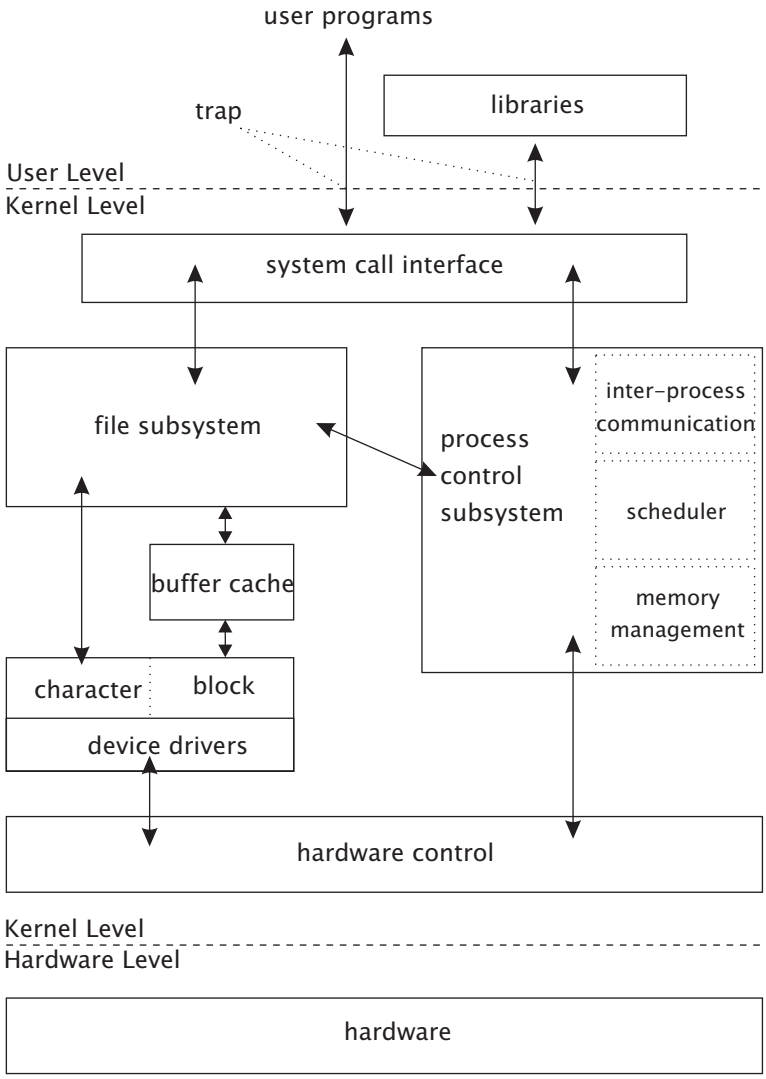


Abbildung 3.2: Block-Diagramm des UNIX System V Kerns

3.1 Die Bedienoberfläche

3.1.1 Shell

UNIX enthält ein „Schalen-Programm“, das mit dem Benutzer interagiert. Es akzeptiert seine Kommandos, führt die entsprechenden Programme aus (executable files entsprechenden Namens) und leitet deren Ausgaben weiter. Die Shell ist ein Benutzerprogramm, das vom Terminal lesen und auf ihn schreiben sowie Programme ausführen kann. sh, tcsh, csh, ksh. Eine Shell wird vom `login` Programm automatisch gestartet, weitere können aber jederzeit später auch gestartet werden.

Shells machen dem Benutzer die Arbeit leichter u.a. durch folgende Dienstleistungen:

- I/O Umleitung: `sort < in > out`
- Pipes: `sort < in | more`
- Kommando-Ergänzung: `ls *.c`
- Hintergrund-Prozesse: `a.out &`

Shell Eingabe ist selbst wieder eine Sprache:
Shell-Programme (*scripts*) sind Sequenzen in Shell-Sprache.

Beispiel:

```
writetape:
#!csh
foreach i (*.c  *.doc)
echo $i
dd if = $i  of = /dev/nrmt8  obs = 4000  cbs [1]= 80  conv = block
end
rewind /dev/nrmt8
```

3.1.2 Dateien und Dateikataloge/verzeichnisse in UNIX

Dateien sind Sequenzen von 0 oder mehr Bytes. Interpretation der Dateien ist Benutzersache.
Dateinamen sind bis 255 Zeichen lang.

Konvention:	`*.c`	C-Programme
	`*.s`	Assembler
	`*.f`	Fortran

Zugriffsrechte sind gruppiert:

Abbildung 3.3: Gruppierung von Zugriffsrechten

Beispiel:

```
chmod   o+r   *.c
```

Standardrechte: via `umask`

[1] conversion buffer size

Beispiel:
`umask 026` ⟶ `rwx.r-x.--x`

Verzeichnisse: selbst Datei mit Auflistung der gruppierten Dateien.

Beispiel:
`cat -v <dir>`

Erzeugen: `mkdir <name>`

Listen: `ls <name>`

Zugriffsrechte: Wie bisher, aber `x` Recht bezieht sich auf Zugriffsrecht auf bekannte Datei im Verzeichnis.

Verzeichnisse in Verzeichnissen:

```
/usr/bin
/usr/include/stdio.h
```

3.1.2.1 Beliebte Dateiverzeichnisse

`/bin`	System Binärdateien (ausführbar)
`/dev`	Dateien, die für Geräte stehen
`/etc`	Systemverwaltung
`/lib`	Bibliotheksfunktionen
`/tmp`	Temporäre Dateien (werden periodisch gelöscht)
`/usr`	Benutzerdateien
`/usr/adm`	Systemverwaltung
`/usr/name`	Datei für Benutzer mit `login` „*name*“
`/usr/bin`	Weitere System-Binärdateien
`/usr/include`	System Vorspann Dateien
`/usr/lib`	Compiler etc
`/usr/man`	On-line Manual
`/usr/spool`	Zwischenspeicher für Drucker etc
`/usr/src`	System Quellcode
`/usr/tmp`	Temporärdateien

3.1.3 UNIX Dienstleistungsprogramme (Utilities)

1	Datei und Katalogmanipulation	(`ls, chmod` ...)
2	Filter	(`tee` ...)
3	Compiler und Programmentwicklung	(`cc, rcs` ...)
4	Textverarbeitung	(`ed` ...)
5	Systemverwaltung	(`du` ...)
6	Verschiedene	(`sleep` ...)

(POSIX 1003.2)

3.2 Prozesse

Prozeß: elementare Ablauf- und Verwaltungseinheit im Rechner. Besteht aus Programmcode und der gesamten für dessen Ablauf nötigen Umgebung (z.B. Verzeichnis zugehörigen Hauptspeichers, Dateien, Programmzähler, Abrechnungsinformation, ...). Wir sprechen vom Hardware-Kontext, Software-Kontext und Memory-Kontext eines Prozesses.

UNIX ist ein Multiprozeß-BS. Es kann mehrere lauffähige Prozesse gleichzeitg verwalten und zwischen ihnen hin- und herschalten, so daß sie praktisch gleichzeitig ablaufen. (In Mehrprozessorerweiterungen geschieht dies auch tatsächlich.)

Aufgabe: Drucken Sie eine Liste aller momentan aktiven Prozesse. Welches Kommando haben Sie benutzt?

Systemprozesse im Hintergrund heißen Dämonen (*daemons*).

Beispiel:
„`cron`“. Wacht einmal pro Minute auf, sieht sich in Tabelle „`crontab`“ nach Arbeit um und führt diese dann aus und schläft wieder ein.

(Kind-)Prozesse können durch eine Gabelung „`fork`“ vom Vater erzeugt werden. `pid = fork();` Das Kind beginnt als exakte Kopie des Vaters (Programm + Daten). Der Kontrollfluß kann zwischen Vater und Kind verschiedene Wege gehen, da das Resultat `pid = fork()` im Kind $= 0$ ist, im Vater jedoch > 0 (die Speicherbereiche sind getrennt!).

Beispiel:

```
{ pid = fork();
  if (pid > 0) parent_code();
  else child_code();
}
```

Alles beginnt mit dem „`init`“ Prozeß, der nach dem Booten automatisch gestartet wird. Der liest eine Konfigurationstabelle und startet einen „`login`“ Prozeß für jeden Terminal. Diese warten auf input und führen bei erfolgreichem login eine shell aus. Diese setzt die Benutzerkommandos als Prozesse ab, z.B. `cp, cc, ....`

3.2.1 Prozeß-Synchronisation

1. *Pipes*
   Prozesse können zu „Leitungen“ (pipelines, Fließbändern ...) verbunden werden.
   Beispiel:
   `ls | more;`
   Produzent | *Konsument*;
   Produzent und Konsument kommunizieren in der Implementierung über einen endlichen Pufferspeicher. Sie senden sich gegenseitig *Signale*, um sich mitzuteilen, daß der Puffer geleert werden oder gefüllt werden kann.
   ⟶ System-Aufruf `pipe` ⟶ File-System

2. *Signale*
   Signale sind ein vom BS realisierter Unterbrechungs-Mechanismus, der analog zu Hardware-Interrupts gestaltet wurde, aber vollständig in Software abläuft.
   Ein Prozeß kann einem anderen durch den Systemaufruf `kill` ein *Signal* senden, sofern der Empfänger dieselbe effektive user-id hat. Ein Signal kann durch `killpg` an alle Prozesse einer Prozeßgruppe geschickt werden. Die Prozeßgruppen-ID kann durch `setgrp` gesetzt werden und wird an Nachfahren vererbt.
   Der BS-Aufruf zum Senden eines Signals ist
   ```
   kill (pid,signum)
   ```
   wobei `pid` die Prozeßnummer des Empfängers ist und `signum` das Signal.
   Im POSIX-Standard sind folgende Signale vorgesehen:

`SIGABRT`	Abort & core dump
`ALRM`	Hardware timer went off
`FPE`	FP error
`HUP`	Hung up (phone line)
`ILL`	Illegal instruction
`INT`	Interrupt (by user - DEL, ↑C)
`KILL`	Kill Process unconditionally
`PIPE`	wrote to pipe with no reader
`SEGV`	Segmentation Violation
`TERM`	Please terminate gracefully
`USR1`	User defined ♯1
`USR2`	User defined ♯2

   Ein Prozeß bereitet sich darauf vor, ein Signal zu verarbeiten, indem er dem BS mitteilt, welche Routine (signal handler) beim Eintreffen des Signals ausgeführt werden soll.
   BS-Aufruf `signal(SIGNAL, handler)`
   Bei Eintreffen eines Signals (BS führt `kill` eines anderen Prozesses aus) unterbricht das BS den signalisierten Prozeß, indem es statt dem normalen Program Counter des Prozesses den PC des handler lädt und statt dem Run-Time stack des Prozesses seinen signal stack. (Falls nichts vereinbart wurde, wird der Prozeß abgebrochen). Danach fährt der Prozeß an der alten Stelle weiter.

3.2.2 Implementierung von Prozessen in UNIX

Zunächst abstrakt: Alle Verwaltungsinformationen über einen Prozeß sind im *Prozeßleitblock* (process control block *pcb*) zusammengefaßt (z.B. Segmentgrenzen Text, Daten, Stack etc.).

In UNIX liegt die Verwaltungsinformation in 2 Strukturen:

1. Process structure
   Ein Eintrag (*process table entry*) in der HSP-residenten Tabelle (*process table*) aller Prozesse
2. Benutzer-Struktur (user structure, u-dot structure)
   wird mit dem Prozeß ausgelagert.

Informationen in der **Prozeßtabelle** (*PTE - process table entry*)
(was man braucht, auch wenn Prozeß ausgelagert ist)

1. Scheduling Parameters (Prozeßlaufplanung)
   Priorität, CPU-Zeit u. Blockiert-Zeit im letzten Zeitquantum.
2. Speicher-Abbild.
   Zeiger auf Text-, Daten-, Stack-Segmente oder deren Seitentabellen oder die externen Adressen, falls Prozeß ausgelagert.
3. Signale
   Welche Signale werden ignoriert, temporär ausgesperrt, von Verarbeitern behandelt oder gerade ausgeliefert.
4. Verschiedenes
   Prozeß-Status (blocked, running, ready ...)
   PID, PID des Vaters, User ID, group ID, Zeit bis zum nächsten Alarm, Ereignis (event), auf das gewartet wird.

Informationen in der **Benutzer–Struktur**

(Information, die nur benötigt wird, wenn Prozeß im HSP und lauffähig ist.)

1. Register (Prozessor u. Co-Prozessor)
   gehören zum Prozeßstatus, also zur Prozeßumgebung. Sie werden gesichert, wenn der Prozeß durch kernel trap unterbrochen wird, z.B. zur Auslagerung.
2. System-Aufruf-Status
   Parameter eines laufenden System-Aufrufs.
3. Dateibeschreibungstabelle (**file descriptor table**)
   Wird hierdurch für Systemaufrufe zugängig, d.h. `read(fd,buf,nbytes)` benutzt fd als Index in diese Tabelle, um die Geräteadresse herauszufinden, von der zu lesen ist.

4. Abrechnung (**accounting**)
   Zeiger auf Tabelle mit Abrechnungsinformation (⟶ `getrusage`) Zeit, Speicherplatz, ...
   auch Grenzen, die eingehalten werden müssen.

5. Kern-Stapel (**kernel stack**)
   Der Kern benutzt seinen eigenen Prozeß-spezifischen stack, um Systemaufrufe dieses Prozesses zu bearbeiten.

3.2.3 Realisierung von `fork()`

Nach kernel trap sucht das BS einen freien Eintrag in der Prozeßtabelle und kopiert fast den ganzen Eintrag des Vaterprozesses.
Wichtigste Ausnahme: Es werden neue, gleich große Segmente für Text, Daten, Stack und user structure angelegt und deren Zeiger gespeichert. Dann wird der Inhalt dieser Segmente vom Vater zum Kind kopiert.
Mögliche Optimierung (⟶ MACH): copy on write, d.h. zunächst wird auf das Segment des Vaters verwiesen und kopiert wird erst, falls das Segment verändert wird (durch Schreiben).

3.2.4 Scheduling

Die **Prozeß-Ablaufplanung** (*process scheduling*) legt fest, welcher Prozeß als nächstes den Prozessor bekommt und abläuft und welcher zunächst zurückgestellt wird. Hierzu bekommt jeder Prozeß eine **Priorität** (*priority*) zugeteilt.

Es gibt viele Möglichkeiten, die Priorität eines Prozesses zu definieren.
Z.B.:
hohe Priorität:

- schon lange nicht zum Zuge gekommen
- interaktiver Prozeß
- erst ganz kurz im HSP

niedrige Priorität:

- viel und lang gerechnet
- blockiert (wartet auf Ereignis z.B. I/0)
- Background Process (`nice()`)
- benutzt viel Speicher

Das traditionelle UNIX Scheduling wurde auf gute Leistung für interaktive Prozesse ausgelegt. Für reines Batch-Scheduling oder für Realzeitumgebungen ergeben sich z.T. abweichende Gesichtspunkte und Lösungen.

UNIX verfährt nach einem 2-Ebenen-Scheduling:

Obere Ebene (high level): swapper.
Lagert Prozesse in HSP ein und aus unter Benutzung sekundären Speichers (Platten).

Untere Ebene (low level):
Auswahl des nächsten vom Prozessor zu bearbeitenden Prozesses aus der Warteschlange lauffähiger Prozesse (run queue).
UNIX hat eine Mehrschichten-Warteschlange (multiple run queues).

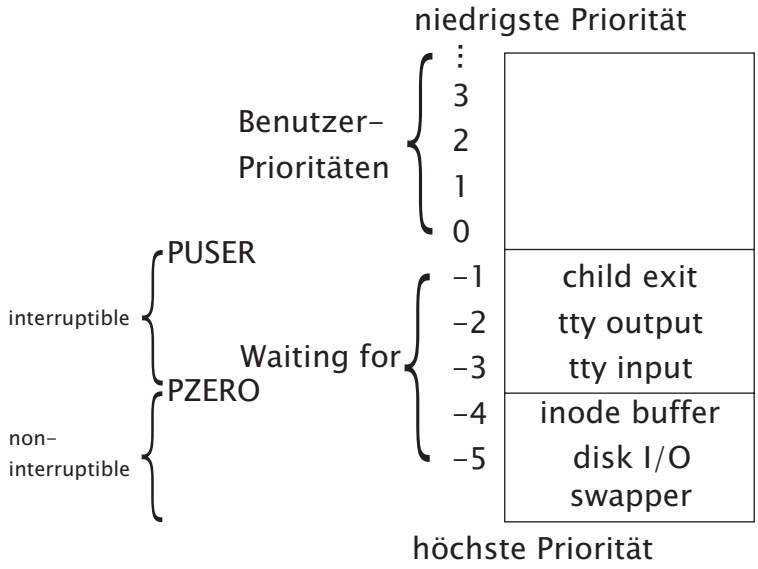


Abbildung 3.4: Prozeß–Prioritäten in UNIX

Der Scheduler wählt den ersten Prozeß der höchsten Prioritätsstufe. Prozeß läuft 1 Quantum (z.B. $5 \cdot \frac{1}{60}$sec), dann wird er wieder hinten in die Q angehängt. Einmal pro sec. werden die Prozeßprioritäten neu berechnet.

Benutzerprozesse können (temporär) eine höhere fixe Priorität zugewiesen bekommen, um Kern-Ressourcen schnell freigeben zu können. Muß ein Prozeß z.B. auf Disk–I/O warten, so wird ihm beim Schlafenlegen bereits eine hohe temporäre Priorität zugewiesen, mit der er die Daten dann einlesen (oder ausgeben kann). Bereits während des Wartens kann er keine Signale mehr empfangen, falls diese Priorität höher als PZERO ist. Nach Ende des Wartens läuft er mit dieser Priorität, bis er in den User-mode zurückkehrt, worauf seine Priorität neu bestimmt wird. Man will so erreichen, daß Prozesse schnell wieder aus dem Kern herauskommen. Würde der obige Prozeß z.B. durch ein Signal im Warten unterbrochen, so wäre wahrscheinlich die Zeit zur Übernahme der Daten von der Platte verpaßt, und es müßte neu gewartet werden. Das Warten auf Terminal I/O genießt hohe Priorität, da UNIX

für interaktive Benutzung optimiert ist. Solche Prozesse können aber durch Signale gestört werden, da Signalverarbeitung relativ zur Benutzer-Interaktion wenig Zeit verbraucht.

3.2.4.1 Berechnung der Priorität eines Prozesses

Die Priorität (im User-Mode) wird alle 4 Uhrtakte nach folgender Formel berechnet

$$\texttt{p\_usrpri} = \texttt{PUSER} + \left\lceil \frac{\texttt{p\_cpu}}{4} \right\rceil + 2 \cdot \texttt{p\_nice}$$

`PUSER` (= 50) ist die Basispriorität für den User-Mode. `p_cpu` reflektiert den Gebrauch der CPU. In jedem Uhrtakt wird `p_cpu` des laufenden Prozesses hochgezählt. Einmal jede Sekunde wird `p_cpu` gefiltert, um früheren CPU Gebrauch vergessen zu lassen.

$$\texttt{p\_cpu} = \frac{2 \cdot \texttt{load}}{2 \cdot \texttt{load} + 1} \cdot \texttt{p\_cpu} + \texttt{p\_nice}$$

`load` ist die durchschnittliche Länge der run-queue während der letzten Minute.

Bei 2 Prozessen im System wird dadurch nach 5 Sekunden etwa 90 % des vergangenen CPU Verbrauchs ausgefiltert (vergessen).

3.3 2 Ebenen Ablaufplanung

Sind sehr viele Prozesse vorhanden, sollen sehr lange inaktive Prozesse auf Disk ausgelagert werden (und dann sehr lange ausgelagerte auch wieder eingelagert werden). Dazu benutzt man gerne zwei scheduler; einen *dispatcher*, der nur innerhalb des HSP agiert, und einen high-level scheduler, der Teile der run-queue ein- und auslagert.

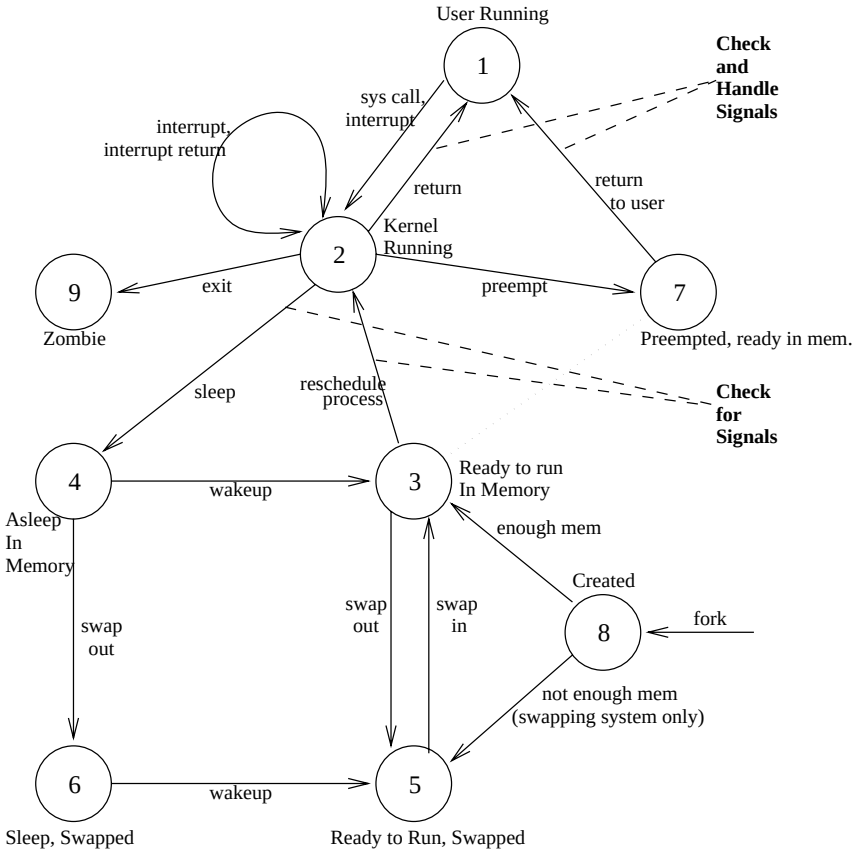


Abbildung 3.5: Prozeßzustandsdiagramm in UNIX SV

3.4 Leichte Prozesse (*threads of control*)

Gegeben ein Multiprozessor mit gemeinsamem Hauptspeicher

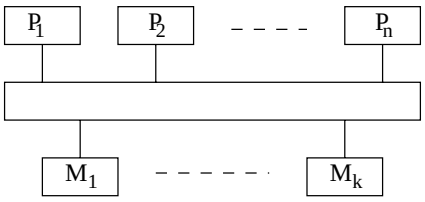


Abbildung 3.6: Shared Memory Multiprozessor

Ein einzelnes Anwenderprogramm kann nur dann die Maschine voll ausnützen, wenn es selbst parallel ist. Innerhalb eines UNIX Prozesses müssen mehrere unabhängige Instruktionssequenzen ausgeführt werden können (*threads of control*), z.B. mehrere Prozeduren gleichzeitig.

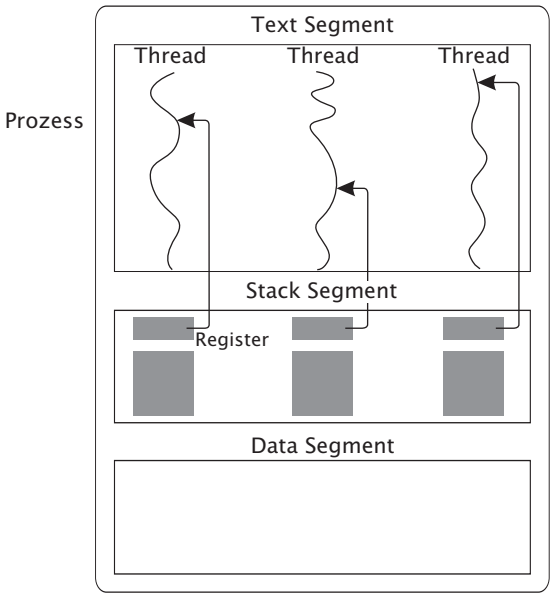


Abbildung 3.7: Speicherbild: Prozess mit Threads

Beispiel: `Sortiere (divide & conquer)`

```
            Sortiere (A[1...n],n)
/*Sortiere array A von Länge n
 */
{//(1) [Trivialfall.]
  if (n<=1) return;
 //(2) [Teile.] Errechne k und verteile A intern um,
 //     so daß  A[i]≤A[j] ∀ i,j, i≤k<j
 //(3) [Herrsche.]
                  Sortiere (A[1...k],k),
                  Sortiere (A[k+1,...n],n-k);
 //(4) [Ende.]
  return
}
```

Die beiden rekursiven Aufrufe in Schritt (3) können parallel ausgeführt werden. Würde für den 2. Aufruf ein neuer UNIX-Prozeß erzeugt werden, so würde dies zu erheblichem unnötigen Aufwand (overhead) führen: Es müßte der gesamte Prozeßleitblock kopiert werden, und neue Segmente (text, data, stack) samt Seitentabellen müßten angelegt werden. Dies ist unnötig, da Textsegmente und Datensegmente sowie Dateitabellen und Puffer gleich bleiben. Es reicht aus, wenn der neue Sortierprozeß seinen eigenen Stack erhält und einen reduzierten Prozeßleitblock, in dem der Registersatz bei einer Unterbrechung gespeichert werden kann und der zur Identifizierung des Prozesses dient. Wir nennen dies einen *leichten Prozeß*, da sein privater Kontext erheblich reduziert ist und er viel schneller gestartet werden kann (etwa eine Größenordnung schneller als ein UNIX-Prozeß). Wir bemerken, daß Threads keinen eigenen geschützten Datenbereich haben, nicht einmal ihr Stack-Segment ist geschützt. Das

Threads-Konzept ist bereits bei vielen modernen BSen vertreten (MACH, OSF/1, OS/2, Solaris 2, IRIX, ...). Threads wurden in POSIX (4.3) standardisiert (pthreads $\longrightarrow$ 12.1.6). Ein de-facto Standard sind ferner C-Threads, wie sie als Schnittstelle unter MACH (OSF(1)) angeboten werden. C-Threads sind weniger komfortabel, aber sehr portabel und sehr effizient.

C-Thread „System-Aufrufe“:

1. cthread_t cthread_fork (func,arg)
   any_t* func(), any_t* arg;
   Erzeugt neuen thread, der die (Haupt-)Funktion func(arg) ausführt. Das Resultat ist die ID des threads.

2. cthread_exit ()
   Terminiert den laufenden thread. Ein cthread_exit wird implizit am Ende der Hauptfunktion des threads ausgeführt.

3. any_t cthread_join (id)
   cthread_t id;
   Wartet auf das Ende des threads mit ID id. Das Resultat ist das Resultat der Hauptfunktion von thread id.

4. cthread_detach ()
   Deklariert, daß niemand auf das Ergebnis des laufenden thread warten wird.

5. cthread_t cthread_self ()
   Liefert die ID des laufenden thread.

6. cthread_yield ()
   Nimmt freiwillig den laufenden thread von der CPU herunter. (Benutzer Scheduling).

Beispiel: (multi-threaded): [Hausaufgabe]

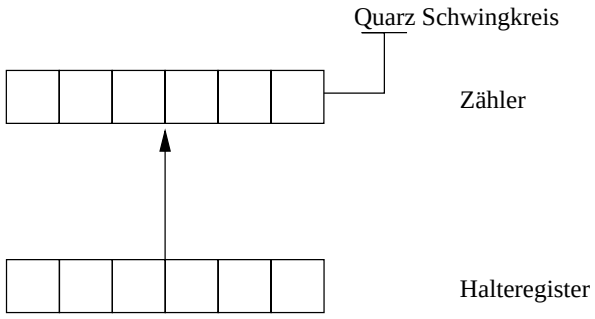
```
                Sortiere (A[1...n],l,r)

/*Sortiere Array A[1..n] im Intervall [l,r]
 */
{ //(1) [Trivialfall.] Sortiere A[1..n] im Intervall [l,r]
  if (r<=l) return;
  //(2) [Teile.] Errechne k, l≤k<r und verteile A intern um,
  //    so daß  A[i]≤A[j]  ∀ i,j,i≤k<j  und l≤k<r
  //(3) [Herrsche parallel.]
       { any_t a[3]; cthread_t id;
         a[0] = A[1..n]; a[1] = l; a[2]=k
         id = cthread_fork (Sortiere_jacket, a);
         Sortiere (A[1..n],k+1,n);
  //(4) [Ende.]
         cthread_join(id);
```

```
        }
}
          Sortiere_jacket(arg)
  any_t arg[3];
{         Sortiere (arg[0], arg[1], arg[2]); }
```

Bemerkung: Einzige Synchronisation über cthread_join, d.h. warten auf Fertigstellen der Kind-Prozedur, da die Probleme bereits in (2) vollständig geteilt werden.
Weitere Synchronisationsmittel werden im Abschnitt IPC behandelt.

3.5 Uhren (Timer)



Das Halteregister wird durch Software mit einem Wert geladen. Der Zähler wird bei jeder Schwingung des Quarzes dekrementiert. Ist Zähler = 0, so wird der Timer-Interrupt ausgelöst und das Halteregister in den Zähler kopiert. Die Periode zwischen Interrupts ist ein *Clock-Tick*.

Clock Software hat folgende Funktionen

- Uhrzeit nachführen (in Sekunden)
- Zeitscheibe für Prozesse bereitstellen, z.B. 100ms, 10ms.
- Accounting/Abrechnung
- ALARM Systemaufruf verwalten
- watchdog timers für das System bereitstellen.

Der Timer interrupt erlaubt es, nach jedem Clock tick die Clock Software anzustoßen. Es kann der Zeitzähler des laufenden (bzw. unterbrochenen) Prozesses inkrementiert werden, geprüft werden, ob die Zeitscheibe abgelaufen ist, ob ein SIGALRM geschickt werden muß etc.

3.6 Das UNIX-Speichermodell

3.6.1 Speicherverwaltung

Historisch gesehen benutzten Programmierer zunächst für jedes Programm einen festen Speicherbereich im physikalischen Hauptspeicher. Als es mit wachsender Speichergröße und Prozessorleistung möglich wurde, mehrere Prozesse gleichzeitig ablaufen zu lassen, mußte man jeden Prozeß dynamisch im Speicher an einer freien Stelle plazieren können. Wir unterscheiden dann zwischen dem *virtuellen Speicherbild*, das der Anwendungsprogrammierer vor sich sieht, und dem *realen Bild*, das der realen Anordnung im physikalischen Hauptspeicher entspricht.

Um ein **verschiebliches**(*relocatable*) Speicherbild zu erhalten, muß man zur Laufzeit zu jeder Adresse dynamisch eine Basisadresse addieren können. Dies wird von modernen Mikroprozessoren (z.B. Motorola 68020) in Hardware unterstützt. Um freie Stellen im physikalischen Speicher besser nutzen zu können, wurde es zunächst nötig, den Gesamtspeicher des Prozesses in kleinere **Segmente**(*segments*) aufzuteilen, die separat verschoben werden können.

Mit der Entwicklung von komplexerer Hardware zur **Adreßumsetzung** (*address translation*), sog. **Speicherverwaltungseinheiten**(*MMU—memory management unit*) konnte man schließlich die Segmente in noch kleinere **Seiten**(*pages*) fester Größe zerlegen. Der physikalische Hauptspeicher wurde in feste **Rahmen** oder **Kacheln** (*page frames, tiles*) (z.B. einer Größe zwischen 0.5 und 4 KB) aufgeteilt, und jede Seite kann mit Hilfe der MMU auf jede beliebige Kachel gelegt werden. Zur Adreßumsetzung verwaltet die MMU sog. **Seitentabellen** (*page tables*).

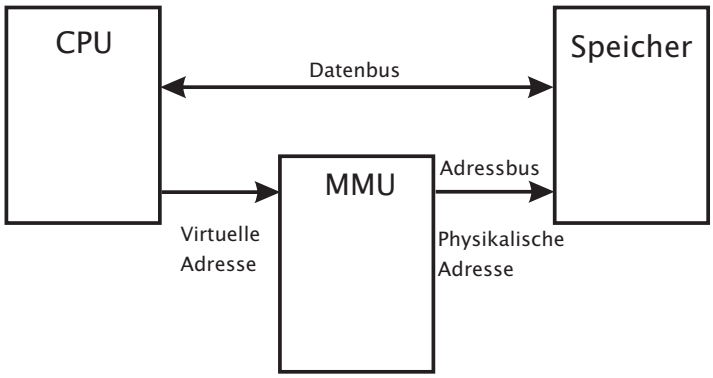


Abbildung 3.8: Speicherzugriff mit MMU

Wird in eine neue Seite adressiert, die noch nicht auf eine Kachel abgebildet wurde, so generiert die MMU bei der Adreßumsetzung einen **Seitenfehler** (*page fault*). Dies ist ein Interrupt, den der (*page fault handler*) im Betriebssystem bedient. Er sucht eine freie Kachel, bringt die Seite per DMA in den Hauptspeicher und korrigiert die Seitentabelle entsprechend. Als Nebeneffekt der Seitenfehlerbehandlung überwacht UNIX die Speichergrenzen des Prozesses, indem es vor der Einlagerung zunächst überprüft, ob der Prozeß auf diese Seite überhaupt zugreifen darf. Falls nicht, liefert UNIX das Signal SIGSEGV (segmentation violation) an den Prozeß aus.

Durch die automatische Einlagerung von Seiten kann ein Prozeß ablaufen, ohne daß sein gesamter eigentlich nötiger Speicher je zu einer Zeit im physikalischen Hauptspeicher vor-

handen ist; der eigentliche Speicher kann also sogar größer sein als der reale Speicher. Dieses Konzept, inklusive der zugehörigen Hardware und Software, die für eine effiziente Umsetzung des Konzepts nötig ist, bezeichnet man mit dem Fachbegriff **virtueller Speicher** (*virtual memory*).

In der Praxis kann virtueller Speicher nur dann effizient funktionieren, wenn das Programm eine gewisse **Lokalität der Speicherzugriffe** (*locality of memory references*) aufweist. Das Programm darf zu jedem Zeitpunkt im wesentlichen nur auf eine begrenzte Menge von Seiten zugreifen, die gesamthaft in den Hauptspeicher passen; Peter Denning hat hierfür den Namen (*working set*) geprägt. Der **Seitentauschalgorithmus** (*pager*) des Betriebssystems muß häufig genutzte von wenig genutzten Seiten unterscheiden können und die häufig genutzten Seiten im Hauptspeicher halten. Trifft dagegen die Mehrzahl der Speicherzugriffe auf fehlende Seiten (z.B. weil das Programm in unvorhersehbarer Weise auf den Speicher zugreift) dann läuft jeder Speicherzugriff effektiv nur noch mir der Geschwindigkeit des Plattenspeichers ab, also ca. 100.000 mal langsamer.

3.6.2 Das Speicherbild eines Prozesses

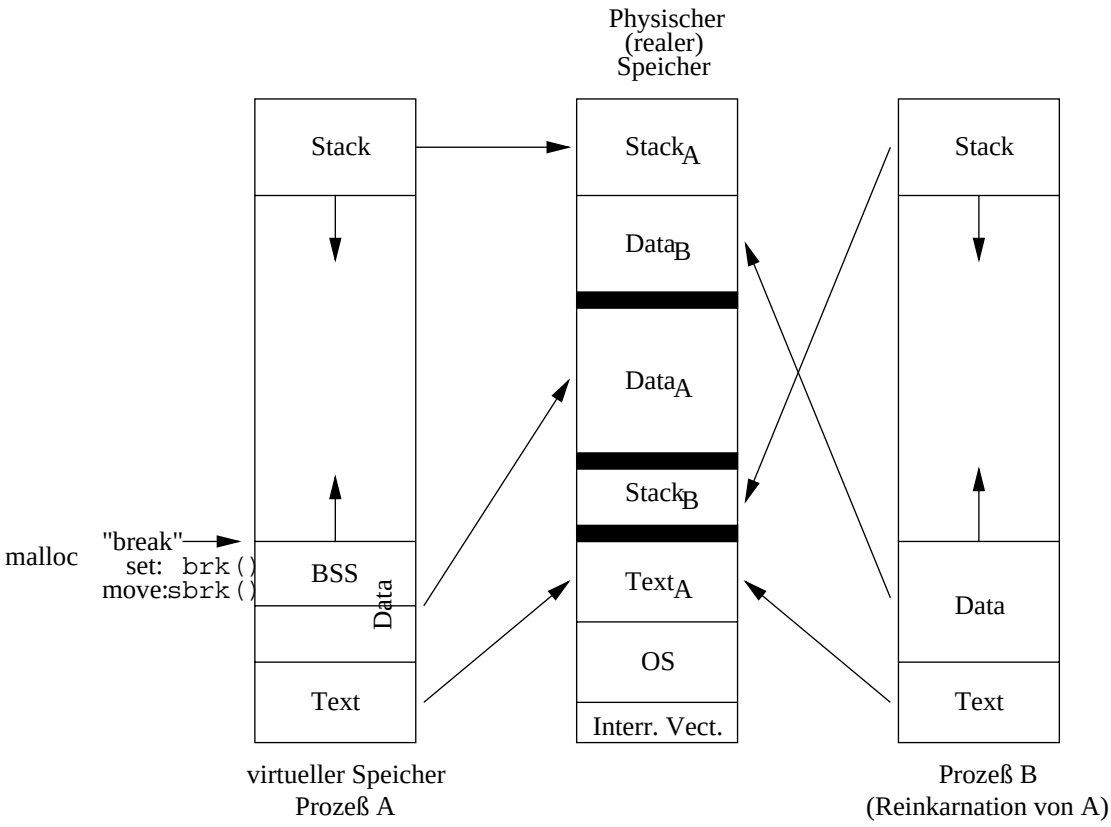


Abbildung 3.9: Speichermodell von UNIX Prozessen

Der Speicher eines UNIX Prozesses besteht im klassischen Sinn aus mehreren **Segmenten**(*segments*). Dies sind logisch zusammengehörige und auch logisch zusammenhängende

(*contiguous*) Speicherbereiche, die der Prozeß tatsächlich mit Daten bevölkert (*populate*), d.h. benutzt. Dazwischen können grosse Speicherbereiche liegen, die der Prozeß (noch) nicht benutzt.
Jeder Prozeß hat ein

- text segment (mit ausführbarem Code)
- data segment (Datenspeicher, static memory) mit 2 Teilen
  - BSS uninitialisierte Daten (z.B. globale uninitialisierte C-Variablen, die alle implizit mit 0 vorbesetzt sind)
  - initialisierte Daten (z.B. `const a = 5; static int a = 5;`)
- Stack (Stapelspeicher, dynamic memory)

Auf dem Stack-Segment wird der Laufzeitstapel des Programms angelegt; dort befinden sich z.B. die lokalen Variablen einer Prozedur. Auf dem Daten-Segment wird u.a. der Heap des Programms angelegt; dort befinden sich z.B. alle mit `new` erzeugten neuen Objekte oder Verbunde des Programms.

Sowohl Stack als auch Datensegment können wachsen. Der Stack wird implizit vergrößert, falls die Hardware eine Überschreibung der Stackgrenze signalisiert (spezielle HW Unterstützung für Stacks). Das data segment kann vom Anwenderprogramm mittels Systemaufruf vergrößert werden (`brk, sbrk`). Dieser wird von der C-Bibliotheksfunktion `malloc` benutzt.

Das text segment ist statisch fest, da der Programmcode nicht dynamisch verändert werden kann. Es kann aber von verschiedenen Inkarnationen desselben Prozesses geteilt werden (z.B. falls mehrere Benutzer gleichzeitig denselben Editor benutzen).

Moderne Organisationsformen des Adressraums benutzen zusätzlich eine Seitenaufteilung des Hauptspeichers und bieten die Möglichkeit, weitere Speicherobjekte (in der Regel Dateien) in dem (heutzutage großen) freien Raum zwischen Stack- und Data-Segment anzulegen (vgl. Abb. 3.14).

3.6.2.1 Segmente, Seiten und die MMU

Segmentierung allein hat den Vorteil, daß logisch Zusammengehöriges auch so gesehen wird und daß Segmente dynamisch wachsen können. Alle Segmente sind im Speicher verschieblich und die Segmentgrenzen werden vom BS verwaltet. Soll das Segment wachsen, so wird entweder die Grenze in den angrenzenden ungenutzten Speicher hinausgeschoben oder für das vergrößerte Segment wird anderswo im HSP ein neuer Platz gesucht; die virtuellen Adressen ändern sich nicht, wohl aber die Umsetzung.

Ein Problem der Segmentierung ist, daß kleine Lücken zwischen den Segmenten übrigbleiben, die nicht genutzt werden können. Dies ist die sog. **externe Fragmentierung** (*external fragmentation*) (Zerfaserung) des HSP. Dem tritt man entgegen, indem man den HSP aufteilt in *Seiten* (pages), d.h. in fixe Allokationsblöcke, aus denen die Segmente zusammengesetzt werden. Jetzt kann jede freie Seite genutzt werden, sofern man Segmente aus

nicht zusammenhängenden (non-contiguous) Seiten aufbauen kann. Wieder wird ein virtuell zusammenhängender Speicherbereich in eine physisch verstreute Realisierung abgebildet mittels einer Umsetzfunktion $r = m(v)$, die virtuelle Adressen zu physischen Adressen umsetzt. Aus Effizienzgründen geschieht dies in Hardware mit einer **Speicherverwaltungseinheit** (*memory management unit—MMU*).

Das Grundprinzip ist folgendes. Seiten haben die Größe 2^n und es gibt 2^K Seiten, so daß $n + K = a$ die Adresslänge. Die niedrigen n Bits der virtuellen Adresse v sind gleich den niedrigen n Bits der realen Adresse r. Die hohen K Bit adressieren die Seite $s = p(K)$. p ist über eine **Seitentabelle** (*page table*) implementiert. Im Beispiel wurden die virtuellen Seiten 2 und 3 an den Anfang des physischen Speichers gelegt.

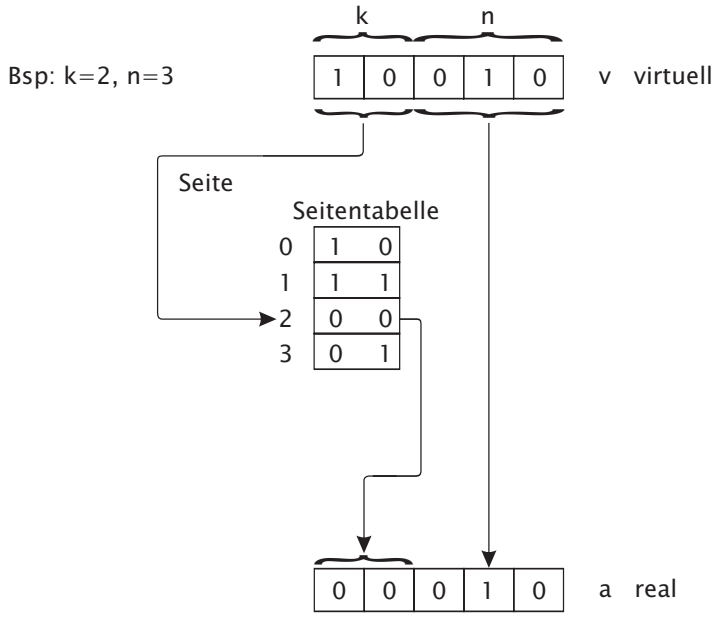


Abbildung 3.10: Prinzip der Adreßumsetzung virtuell → real

Reale MMU's sind noch um einiges komplexer, da sie die Verwendung sehr großer Speicherbereiche unterstützen, die dünn besetzt sind. Z.B. kann es vorteilhaft sein, innerhalb des mit 32 Bit adressierbaren Speichers von 4 GB an mehreren weit auseinanderliegenden Stellen jeweils dynamisch vergrößerbare **Speicherregionen** (*regions*) anzulegen, die jeweils wieder aus mehreren Segmenten bestehen können. Klassische Regionen sind zunächst wieder Text, Stack und Daten. Weitere Regionen können z.B. dazu dienen, verschiedene Dateien in den Hauptspeicher abzubilden um sie dort bequemer zu verarbeiten. (Vgl. dazu die Organisation des Adressraums von Prozessen in 4.4 BSD UNIX in Abbildung 3.14.) Nun würde dies zunächst bedeuten, daß für den gesamten Speicher inklusive der leeren Zwischenräume eine Seitentabelle nötig wäre. Bei einer Seitengröße von 4KB hätte die Tabelle 1 Million Einträge (z.B. von 24 Bit) und dies für jeden Prozeß im System; bei Verwendung von 64 Bit Adressen verschärft sich das Problem noch dramatisch.

Heutige MMU's definieren neben Tabellen von Seiten daher auch Tabellen von Segmenten und Regionen. Außerdem verwaltet die MMU die **Kontext-Tabelle** (*Context Table*), die einen Eintrag für den gesamten Speicherbereich eines jeden Prozesses hält. Wir spielen dies

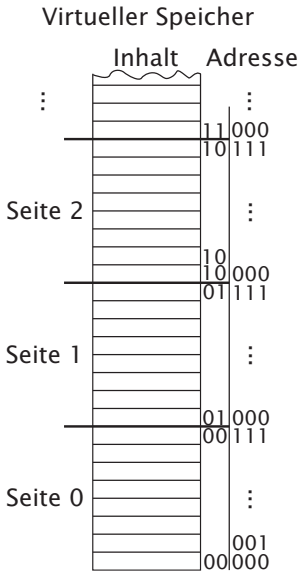


Abbildung 3.11: virtueller Speicher

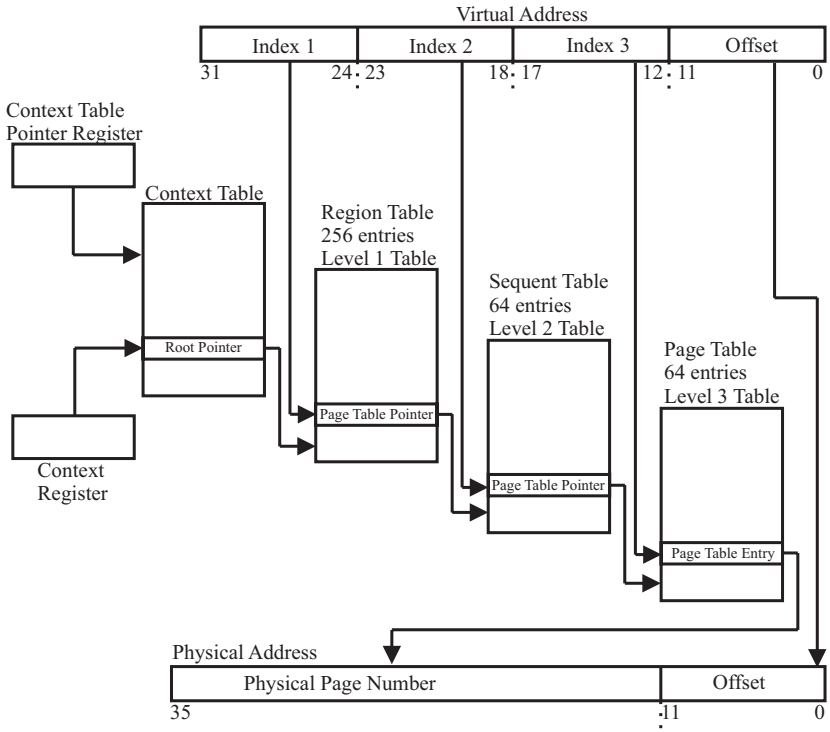
am Beispiel der SuperSPARC Referenz-MMU durch .

Aus einer 32 Bit virtuellen Adresse wird eine 36 Bit physikalische Adresse gemacht. Die virtuelle Adresse adressiert ein Byte innerhalb eines 4 GB Prozeß-Kontexts. Ihre Bestandteile sind (vom signifikantesten Bit an)

1. ein 8 Bit Index (Bits 24–31) in eine Regionen-Tabelle (256 Einträge)
2. ein 6 Bit Index (Bits 18–23) in eine Segment-Tabelle (64 Einträge)
3. ein 6 Bit Index (Bits 12–22) in eine Seiten-Tabelle (64 Einträge)
4. ein 12 Bit Versatz (Bits 0–11) innerhalb der Seiten-Tabelle

Damit betragen die Speichergrössen 4 GB für einen Kontext, 16 MB für eine Region, 256 KB für ein Segment und 4 KB für eine Seite. Falls ein Prozeß also den ganzen 4GB Adreßraum (=1M Seiten) benutzt, so braucht man neben den 16K Seitentabellen auch noch 256 Segment-Tabellen und 1 Regionen-Tabelle. Falls der Prozeß aber ein ganzes Segment oder eine ganze Region gar nicht benutzt, dann vermerkt man dies schon im Eintrag der Segment- oder Regionentabelle und legt die darunterliegenden Tabellen gar nicht erst an. Dies ist der häufigste Fall, in dem man also sehr viel Platz spart; dies wird beim Übergang zu 64 Bit Adreßräumen noch sehr viel wichtiger.

Alle Tabellen liegen im Hauptspeicher. Die MMU interpretiert die virtuelle Adresse und benutzt die Indices und die Tabellen um die zur virtuellen Seite zugehörige physikalische Seite zu finden. Deren Adresse, die physikalische Seitenadresse, ist in dem Eintrag der Seitentabelle gespeichert, zu dem sie durch die Indices in Kontext-, Regionen- und Segment-Tabelle geleitet wird. Dazu implementiert sie den entsprechenden **Tabellenwanderungsalgorithmus** (*table walk algorithm*) in Hardware. Die 24 Bit Adresse der physikalischen Seite wird mit dem Versatz zu einer 36 Bit grossen physikalischen Adresse verklebt.



4 KB	page size
256 KB	segment size
16 MB	region size
4 GB	context size
64	TLB-entries

Abbildung 3.12: Die Architektur der SuperSPARC on-chip MMU

Die Tabelleneinträge sind 32 Bit lang, wobei Bits 0–1 den Eintragungstyp *(Entry Type)* darstellen. Die ET-Bits entscheiden, ob es sich um einen **Seitentabellen-Eintrag** (*page table entry*) handelt ($ET = 2$) oder um einen **Tabellendeskriptor** (*page table descriptor*) mit einem Tabellenzeiger in Bits 2–31 ($ET = 1$). Außerdem enthalten die ET-Bits die Gültigkeits-Information *(validity)*. Falls $ET = 0$, so befindet sich die entsprechende Seite nicht im Hauptspeicher, bzw. bei einem Tabellenzeiger befindet sich keine Seite des Segments oder der Region im Hauptspeicher.

Tabellenzeiger sind 30 Bit Zeiger auf weitere Tabelleneinträge (jeweils ganze Wörter). Ein Eintrag in einer Seitentabelle besteht aus der 24 Bit langen physikalischen Seitenadresse sowie aus Verwaltungsinformation von 8 Bit nach folgendem Muster.

Cacheable (Bit 7). Falls $C = 1$ kann die Seite in einen Cache aufgenommen werden. Falls in der Seite I/O Geräte liegen, die in den Speicher abgebildet wurden, so muß $C = 0$ sein.

Modified (Bit 6). Wird durch die MMU auf 1 gesetzt, wenn auf die Seite ein Schreibzugriff stattgefunden hat.

Referenced (Bit 5). Wird durch die MMU auf 1 gesetzt, wenn auf die Seite ein Zugriff

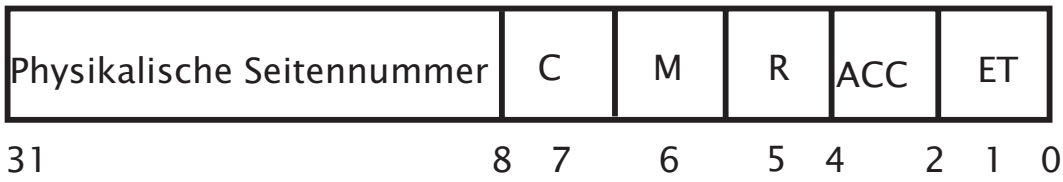


Abbildung 3.13: Seitentabellen-Eintrag der SuperSPARC MMU

stattgefunden hat.

Access Permissions (Bits 2–4). Zugriffsberechtigung auf die Seite, wird nach der folgenden Tabelle in Abhängigkeit davon interpretiert, ob der Zugriff in User Mode oder Supervisor Mode erfolgt.

ACC	User Access	Supervisor Access
0	Read Only	Read Only
1	Read / Write	Read / Write
2	Read / Execute	Read / Execute
3	Read / Write / Execute	Read / Write / Execute
4	Execute Only	Execute Only
5	Read Only	Read / Write
6	No Access	Read / Execute
7	No Access	Read / Write / Execute

Entry Type and Validity (Bits 0–1). Falls $ET = 0$ so ist der Eintrag ungültig, d.h. die Seite ist nicht im Hauptspeicher. Ansonsten stellen bei $ET = 2$ die Bits 8–31 die entsprechende physikalische Seitennummer im Hauptspeicher dar.

Da die Tabellenwanderung aufwendig ist, setzt man zur Effizienzsteigerung typischerweise ein Stück Assoziativspeicher als **TLB** (*translation look-aside buffer*) ein, der die letzten m Umsetzungen von virtuellen Seitenadressen in physische Seitenadressen speichert (z.B. m=64). (Mit jedem TLB Eintrag unterstützt man also die Umsetzung von insgesamt 4 K Adressen.)
Zusammenfassend ist die MMU also für folgendes verantwortlich:

- Adreßumsetzung: virtuelle Seitenadresse → physikalische Seitenadresse.
- Unterstützung des BS bei der Überwachung von Zugriffsrechten.
- Erkennung von Seitenfehlern.
- Unterstützung der Seitentauschverfahren (z.B. durch *referenced* und *modified* Bits).

3.6.3 Behandlung von Seitenfehlern

Wenn ein Prozeß versucht, auf ein Stück seines virtuellen Adressraums zuzugreifen, das im Moment nicht im Hauptspeicher präsent (resident) ist, wird ein Seitenfehler ausgelöst. Der Bearbeitungsroutine für Seitenfehler (page fault handler) wird die virtuelle Adresse übergeben,

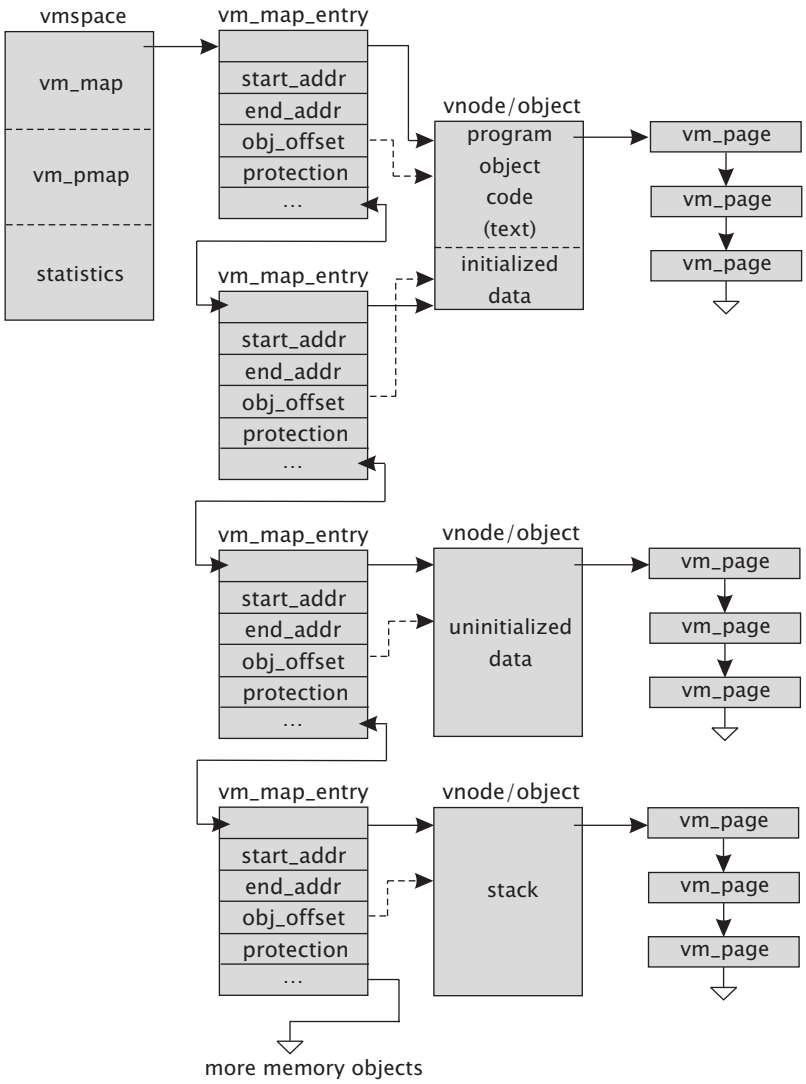


Abbildung 3.14: Layout des Adressraums eines Prozesses in 4.4 BSD UNIX

die den Fehler ausgelöst hat. Diese muss feststellen, ob sich überhaupt an der betreffenden Stelle des virtuellen Adressraums ein Speicherobjekt befindet (andernfalls *segmentation violation*) und falls ja, zu welchem Speicherobjekt die betreffende Adresse gehört und von wo man die betreffende Seite nachladen kann.

In der Ursprungsform des UNIX Adressraums kann man dies durch Vergleich mit den Grenzen von Text, Stack- und Datensegment relativ leicht feststellen. Mit der allgemeineren Organisation des Adressraums von 4.4 BSD wird der Seitenfehler gemäß den folgenden vier Schritten behandelt:

1. Aus der `vmspace` Datenstruktur des Prozesses, der den Fehler ausgelöst hat, wird der Kopf der `vm_map_entry` Liste ausgelesen.

2. Die `vm_entry_list` wird traversiert. Für jeden Eintrag (= Speicherobjekt) wird überprüft, ob sich die Adresse, die den Seitenfehler produziert hat, zwischen Start- und

Endadresse des Eintrags befindet. Wenn der Kernel erfolglos das Ende der Liste erreicht, dann befindet sich die fehlerhafte Adresse nicht in einem gültigen Bereich des Adressraumes und daher sendet der Prozeß ein Segmentfehler-Signal (SIGSEGV).

3. Wurde hingegen ein gültiger `vm_map_entry` gefunden, dann kann die Adresse wie folgt in einen Offset für das darunterliegende Objekt umgewandelt werden:

```
object_offset = fault_address
   - vm_map_entry->start_adress
   + vm_map_entry->object_offset
```

   Durch die Subtraktion der Startadresse erhält man den Offset für das Objekt, das der `vm_map_entry` abbildet. Durch die anschließende Addition des Objekt Offsets, an dem der `vm_map_entry` innerhalb des zugrunde liegenden Objekts (Files) beginnt, ergibt sich der absoluten Offset der Seite innerhalb des zugrunde liegenden Objektes (Files).

4. Der berechnete absolute Offset wird an das darunterliegende Objekt übergeben, das eine `vm_page` alloziert und diese (mit Hilfe seines pagers) füllt. Das Objekt gibt dann einen Zeiger auf die `vm_page` zurück, die an die Stelle im Adressraum abgebildet wird, an der der Fehler erzeugt wurde.

Wenn die fehlende Seite an die passende Stelle im Adressraum abgebildet worden ist, dann kehrt die Fehlerroutine zurück und führt die Instruktion, die den Seitenfehler ausgelöst hat, nochmals aus.

3.6.4 Virtueller Speicher

Mittels der Speicherabbildungen ergibt sich nun die Möglichkeit, virtuell (*virtually* = eigentlich, im Prinzip) mehr HSP zu benutzen als physisch vorhanden ist. Das funktioniert falls zu jedem gegebenen Zeitpunkt nur höchstens soviel Hauptspeicher tatsächlich aktiv benötigt wird, wie real vorhanden ist. Virtueller Speicher wird benutzertransparent implementiert und stellt somit eine neue Abstraktionsebene dar (großer zusammenhängender Adressraum).

3.6.5 Swapping (Prozeßtausch-Verfahren)

Zunächst kann man auf Prozeß-Ebene arbeiten und virtuell mehr Prozesse gleichzeitig ablaufen lassen, als zusammen in den HSP passen. Dies geschieht durch ein **Prozeßtausch-Verfahren** (swapping), d.h. das BS lagert Prozesse niedriger Priorität temporär gesamthaft auf Platte aus, um Platz zu schaffen (Voraussetzung hierfür ist lediglich verschieblicher Code.)

Ein Prozeß muß ausgelagert werden, falls kein Platz vorhanden ist für

- einen neuen durch `fork()` erzeugten Prozeß
- ein größeres Datensegment (`brk(), sbrk()`)
- einen größeren Stack

- die Einlagerung eines Prozesses, der höhere Priorität hat.

Außerdem werden ganze Prozesse ausgelagert, falls der *pager* unter den laufenden Prozessen nicht genügend unbenutzte Seiten finden kann. Dadurch soll das sog. (*thrashing*) (wildes Seitentauschen) verhindert werden. Erst werden untätige Prozesse ausgelagert (≥ 20 sec untätig), dann derjenige der 4 größten, der am längsten gerechnet hat.

3.6.6 Paging (Seitentausch-Verfahren)

UNIX hatte zunächst nur swapping; 3BSD hatte auch paging, da immer größere Programme geschrieben wurden. Paging ist allgemeiner als swapping, da sowohl alle Seiten eines Prozesses ausgelagert werden können als auch nur einige seiner Seiten, die gerade nicht (schon lange nicht mehr) benötigt werden. In UNIX existiert aber beides nebeneinander:
Process 0 ist der swapper, (Proc 1 = init)
und Process 2 ist der *page daemon*.

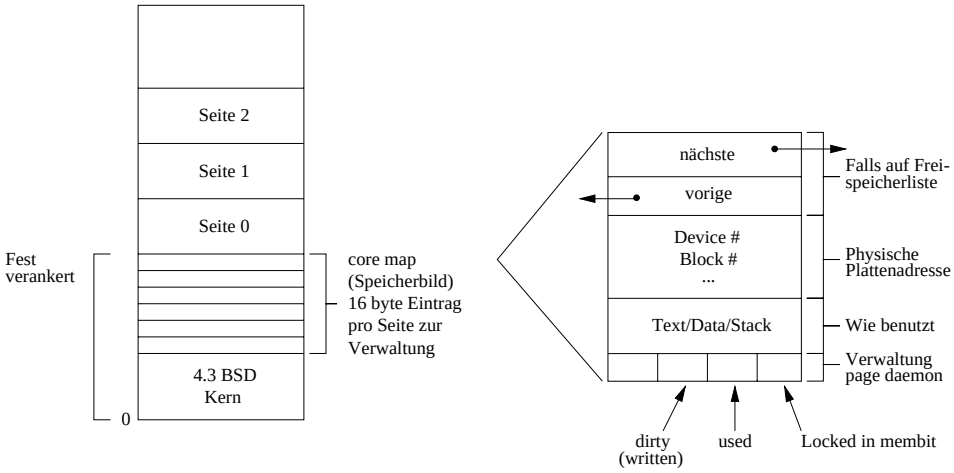


Abbildung 3.15: Seiteneinteilung im HSP

3.6.7 Der UNIX Seitentausch-Algorithmus

Alle 250ms wird der page-daemon geweckt. Der sieht nach, ob $\geq$ **lotsfree** Seiten frei sind (typisch: 25 % HSP). Falls nicht, werden entsprechend Seiten ausgelagert (**paged out**); falls ja, schläft der Dämon weiter.

Der Auslagerungsalgorithmus folgt dem Einzeiger-Uhr-Prinzip. Die Speichereinträge werden reihum durchgegangen und das „benutzt“ -Bit wird zurückgesetzt. Beim 2. Durchgang werden alle die Seiten freigesetzt (in die Freispeicherliste eingekettet), die seit dem 1. Durchgang nicht benutzt wurden. „Schmutzige“ Seiten (auf die geschrieben wurde) müssen auf Platte zurückgeschrieben werden.

Moderne MMU's unterstützen „benutzt" -Bits (usage-bits) in Hardware (vgl. Abschnitt 3.6.2.1). Auf der DEC VAX ohne benutzt-Bit wird das wie folgt simuliert. Die Seite wird in der Seitentabelle als ungültig markiert und ein trap folgt. Darauf werden in Software die Adreßraumgrenzen geprüft und ggf. das „benutzt" -Bit gesetzt und der Eintrag in Ordnung gebracht. (Dieser Trick, den Seitenfehlertrap zu mißbrauchen, wird auch in einigen Speicherbereinigungs-Algorithmen (garbage collection) angewendet).

Variationen des Uhr-Algorithmus:

- Es werden 2 Zeiger benutzt, die einander in dichterem Abstand folgen, damit mehr Seiten ausgelagert werden können.
- Es werden 2 Markierungen, *hoch* und *tief* benutzt und ausgelagert, bis die Zahl freier Seiten *hoch* ist, aber erst bei *tief* wird erneut ausgelagert.

3.7 Das UNIX Dateisystem

Die Dateien sind im Dateibaum organisiert. Zugriff über *absoluten* Pfadnamen (/usr/ast/book) oder *relativ* zur augenblicklichen Position (z.B. book bei Position im Katalog /usr/ast). Es können Dateien eingerichtet werden, die auf andere verweisen (*links*). Dadurch kann auf einer bestehenden Organisation eine andere logische Organisation überlagert werden.
hard links: innerhalb eines Geräts
soft links: über Gerätegrenzen hinweg
(z.B. ln -s /usr/ast/book OS_book).

Es ist möglich, einen Dateibaum von einem Gerät in den Dateibaum eines anderen Geräts einzutragen (*mounting*) und somit logisch einzugliedern. Der Benutzer merkt das Überschreiten der Gerätegrenzen dann nicht und arbeitet logisch mit einem einzigen Dateibaum.

Nach POSIX können Dateien für Exklusivzugriffe reserviert werden (locking). Dies ist zur Synchronisation paralleler Dateizugriffe nötig.
2 Arten von *locks*:

- shared - Reservierung für Lesezugriff (vgl. readers/writers locks)
- exclusive - Reservierung für Schreibzugriff

Es kann jede zusammenhängende Bytesequenz reserviert werden, und shared locks können sich überlappen. Ein Exklusivzugriff ist nur möglich, nachdem alle anderen Reservierungen aufgehoben wurden.

Reservierungen sind nötig, um bei Parallelzugriff Inkonsistenzen und andere semantische Fehler zu vermeiden:
Gegeben Datei D und zwei Prozesse P_1 und P_2:

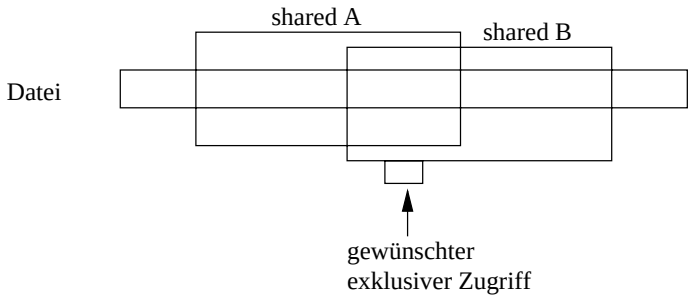


Abbildung 3.16: Datei locking nach POSIX

- Fehlertyp: **Ausfall**
  (1) P_1 liest Byte B = C.
  (2) P_2 liest Byte B = C.
  (3) P_1 schreibt B = C_1.
  (4) P_2 schreibt B = C_2.
  Die Aktion von Prozeß P_1 „fällt unter den Tisch“. Falls zufällig (3) und (4) vertauscht ausgeführt werden, fällt Aktion von P_2 weg. Weder P_1 noch P_2 bemerken dies.

- Fehlertyp: **Inkonsistenz**. Gegeben Wort W = B_1B_0 = $C_{1_1}C_{1_0}$
  (1) P_1 liest Byte $B_0 = C_{1_0}$
  (2) P_2 schreibt $B_0 = C_{2_0}$
  (3) P_2 schreibt $B_1 = C_{2_1}$
  (4) P_1 liest $B_1 = C_{2_1}$
  P_1 liest inkonsistentes Datum $C_{2_1}C_{1_0}$ statt entweder $C_{1_1}C_{1_0}$ *oder* $C_{2_1}C_{2_0}$.

Grundregel: Bei Parallelverarbeitung müssen für Schreibzugriff gemeinsame Daten *immer* reserviert werden.

3.7.1 Implementierung des Dateisystems

3.7.1.1 Organisation der Platte

3.7.1.2 Organisation von I-node und Datei

I-nodes existieren sowohl auf Disk als auch im HSP, wenn die Datei offen ist. Oben gezeigt sind die disk I-nodes. HSP I-nodes haben zusätzliche Felder, u.a. die Position des zugehörigen disk I-node Platzes und ein „dirty bit“, sowie Zeiger für die Liste freier I-nodes und für Verkettung in der Hash-Tabelle der HSP-Inodes. Eine Schloß-Variable schützt den I-node während eines Systemaufrufs vor dem Zugriff durch andere Prozesse.

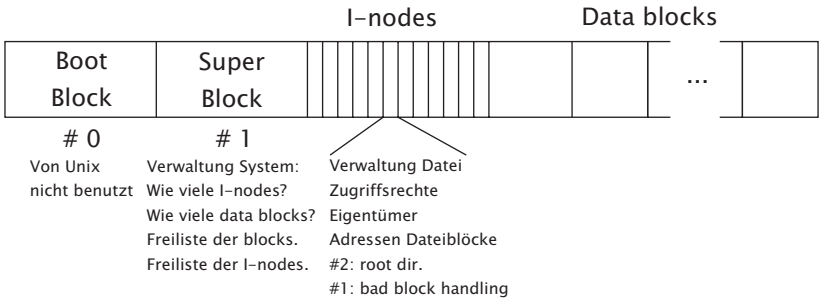


Abbildung 3.17: Plattenorganisation

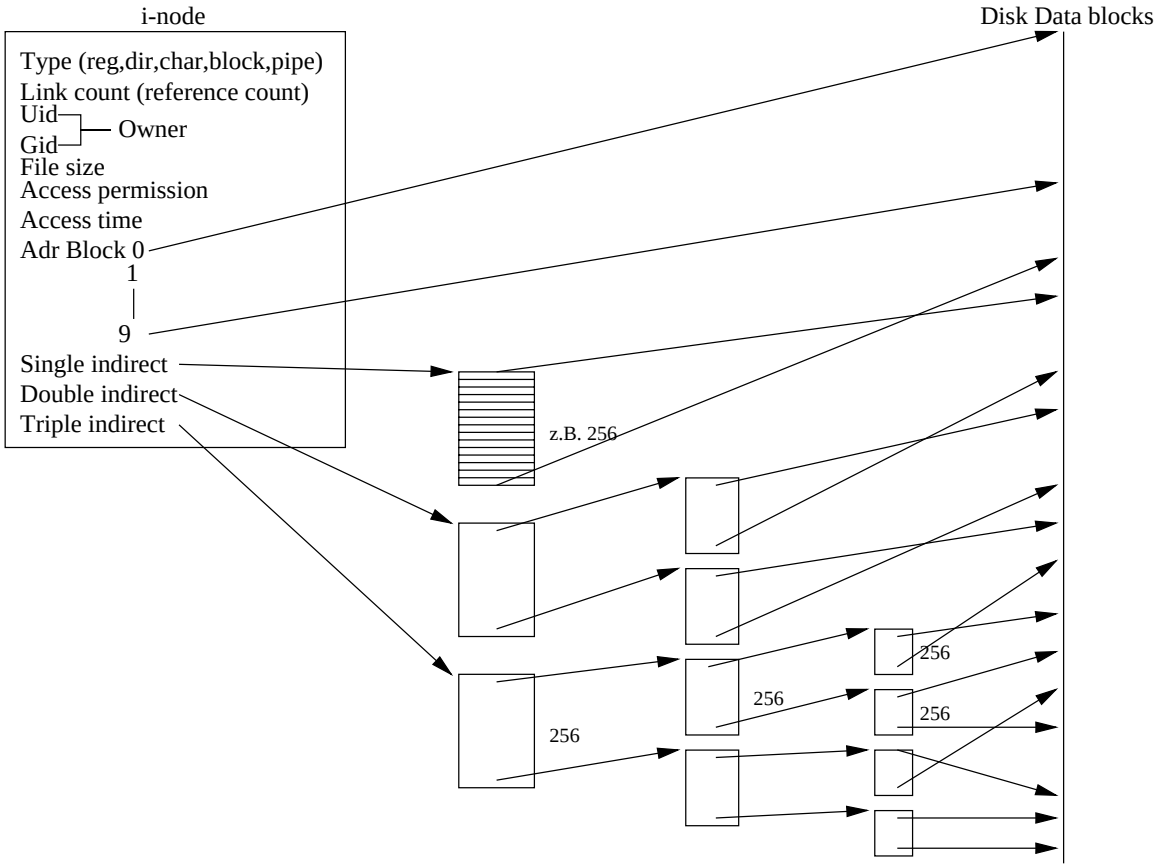


Abbildung 3.18: Zugriffsstruktur für UNIX Dateien

Das Dateisystem errechnet den gefragten Block aus der Dateiposition (in Byte), dem Dateianfang und der Blockgröße. Mit 32 bit Wort können 4 GB adressiert werden. Mit 1 KB Blöcken und 32 bit Blockadressen können im obigen Schema über 16 GB gespeichert werden.

3.7.1.3 File Datenstrukturen im Kern

Zusätzlich zur Repräsentation auf der Platte existieren zu jeder benutzten Datei noch weitere Datenstrukturen im Kern, die den Zugriff regeln und erleichtern.
Der inode auf der Platte wird in einen Eintrag der Inode-Tabelle im Kern (**in-core inode table**) geladen (**caching**).

Jeder Zugang zur Datei (vgl. `open` s.u.) wird durch einen Eintrag in der File-Tabelle des Kerns vermerkt mit Zugangsart (read/write/read-write) und momentaner Lese/Schreibposition (**position**) (vgl. `lseek` s.u.).

Die Prozesse selbst greifen auf Dateien über Deskriptoren (**file descriptor**) zu. Ein Deskriptor ist ein Index in die private File-Deskriptor-Tabelle des Prozesses, die in der u-structure angelegt ist. Dort stehen Verweise auf die Einträge in der File-Tabelle. Die ersten drei Einträge sind per default `stdin (==0), stdout (==1), stderr (==2)`. Die Zwei-Schichten-Struktur aus fdt und ft erlaubt es verschiedenen Prozessen, sich einen Zugang zu einer Datei mit einer gemeinsamen Lese/Schreibposition zu teilen.

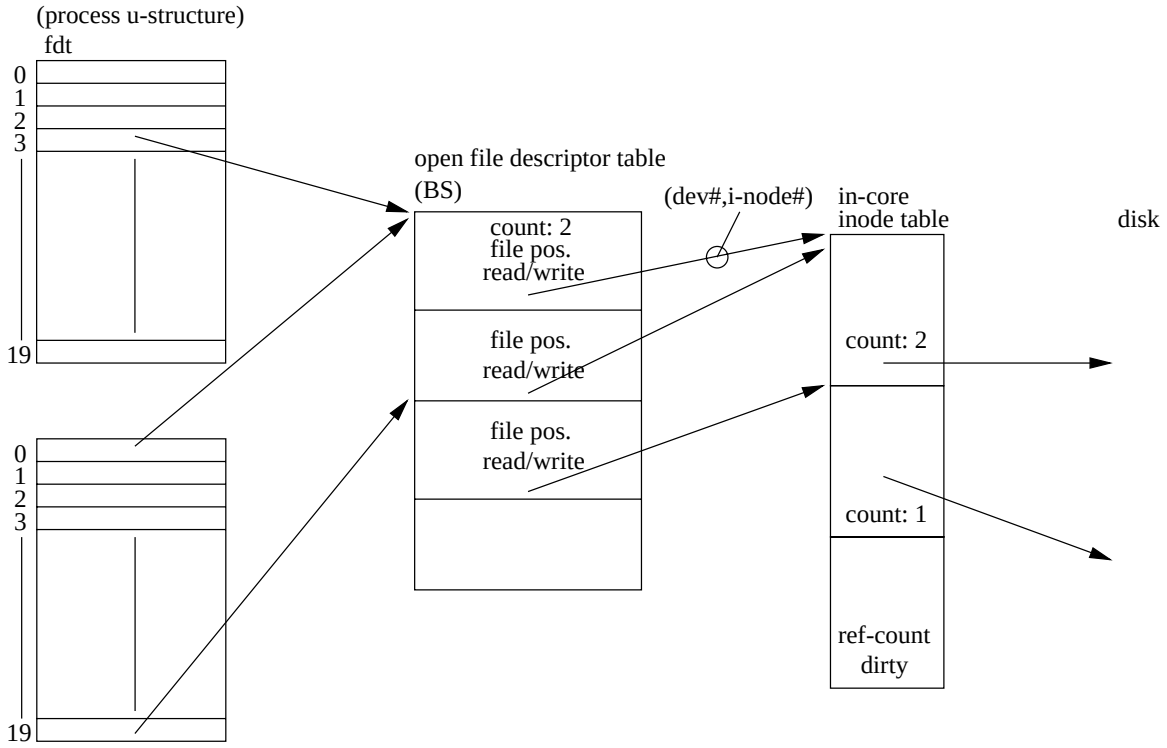


Abbildung 3.19: Datenstrukturen des Filesystems im UNIX-Kern

3.7.2 Öffnen und Schließen einer Datei

Vor der ersten Benutzung muß eine Datei **geöffnet** werden. Aus objekt-orientierter Sicht: das neue Objekt der Klasse Datei muß initialisiert werden. System-Aufruf:

```
fd = open(pathname,flags,modes)
```

`pathname`	ist der Dateiname als String.
`flags`	geben die Art des Öffnens an, z.B. zum Lesen oder Schreiben.
`modes`	gegen die Zugriffsrechte an, falls die Datei neu geschaffen (kreiert) wird.
`fd`	ist der **File-Deskriptor**. Dies ist ein integer Index in die private Deskriptor-Tabelle des Prozesses (**file descriptor table** – fdt). Diese Tabelle ist in der u-structure verankert. `fd == -1` zeigt einen Fehler an.

Entsprechend schließt

```
close(fd)
```

die Datei mit Deskriptor `fd`. Close ist implizit bei Programmende.

Open muß zunächst zum Dateinamen (Pfadnamen) den zugehörigen Inode finden. Hierzu werden Kataloge (**directories**) durchsucht.

Ein Katalog ist eine Datei mit einer Tabelle aus (Name, I-node-index) Einträgen (Abbildungsfunktion). Um Datei zu finden, wird die Tabelle durchsucht und der I-node Index extrahiert, dann können via I-node die Blöcke gefunden werden.

Bei Angabe eines absoluten Pfads /usr/ast/book beginnt die Suche in I-node ♯2, dem root Katalog.

Ist der gefundene Inode noch nicht im Kern, so wird er von der Platte in die Inode-Tabelle des Kerns geladen.

Nach Prüfen der Zugriffsrechte wird ein Eintrag in der File-Tabelle angelegt und die Lese/Schreibposition (**offset**) initialisiert. (Im normalen Lese/Schreibmodus ist dies der Dateianfang 0, im Schreibanhangmodus **write-append** ist dies die Größe der Datei). Danach wird ein Verweis auf diesen File-Table-Eintrag im privaten file-descriptor table des Prozesses eingetragen und zwar an der ersten freien Stelle. Das Ergebnis fd ist dann der Index dieses letzten Eintrags.

Es ist durchaus möglich, eine Datei zweimal zu öffnen, z.B. einmal zum Lesen und einmal zum Schreiben.

Der Aufruf `newfd = dup(fd)` dupliziert einen Deskriptor. Eine Kopie des Eintrags mit Index `fd` wird im ersten freien Platz der Deskriptorstelle untergebracht, sein Index in `newfd` übergeben. Mit `close(stdin); newfd = dup(3);` nimmt z.B. File # 3 den Platz von `stdin` ein.

Dateien werden durch `close()` geschlossen. `close(fd)` löscht den Eintrag mit Index `fd` in der fdt und schafft so einen freien Eintrag. `close` dekrementiert den Referenzzähler **count** im entsprechenden ft-Eintrag und im in-core inode und gibt diese Datenstrukturen frei, wenn **count** = 0 wird.

3.7.3 Lesen einer Datei `read(fd, buffer, nbytes)`

Man muß von fd zum I-node kommen. fd ist ein Index in die Filedescriptor Tabelle (**file descriptor table** – fdt) in der Benutzerstruktur (u structure).

Man könnte in der fdt direkt den I-node Index eintragen, tut dies aber aus folgendem Grund nicht:
Man möchte es ermöglichen, daß grundverschiedene Prozesse verschiedene Dateipositionen halten, daß aber verwandte Prozesse (z.B. shell mit 2 Kindern) sich eine Dateiposition teilen. (shell kann z.B. positionieren und Kind kann dann von dort aus lesen.) Dazu dient die Beschreibungstabelle offener Dateien (**open file descriptor table** – ofdt oder auch **file table** – ft).)

Zwei verschiedene Prozesse können somit dieselbe Datei gemeinsam bearbeiten. Da bei `fork()` mit der u-structure auch der Prozeß fdt kopiert wird, haben Vater und Kind zunächst gleiche fdt-Einträge. Wenn der Vater die Lese/Schreibposition in der Datei verändert, sieht auch das Kind diese Veränderung (und umgekehrt). Dadurch können z.B. 2 Prozesse fortlaufende Einträge in einer Auftragsdatei abwechselnd lesen (und verarbeiten).

3.7.4 Pipes

Eine Pipe (Rohrleitung) ist eine FIFO Kommunikationsstruktur. Pipes wurden traditionell mittels des Dateisystems realisiert. Eine neue Realisierung benutzt sockets (→ BS II). Nach dem Öffnen können sie logisch wie eine Datei beschrieben bzw. gelesen werden, was für Transparenz und Überschaubarkeit sehr wichtig ist. Der Pufferspeicher der Pipe wird durch die 10 direkten Blöcke der Datei realisiert, die logisch als Ringpuffer organisiert werden. Zudem gibt es einen Lesezeiger und einen Schreibzeiger, die auf die nächste zu lesende bzw. zu schreibende Position verweisen. Lese- und Schreibzeiger überholen sich nicht, sondern die entsprechenden Prozesse werden zuvor blockiert und dann wieder aufgeweckt, wenn wieder Platz ist.

Natürlich können mehrere Prozesse in die pipe schreiben oder aus ihr lesen.

Es gibt benannte (**named**) und unbenannte (**unnamed**) pipes zur Kommunikation zwischen verwandten bzw. nicht verwandten Prozessen.

Systemaufrufe:

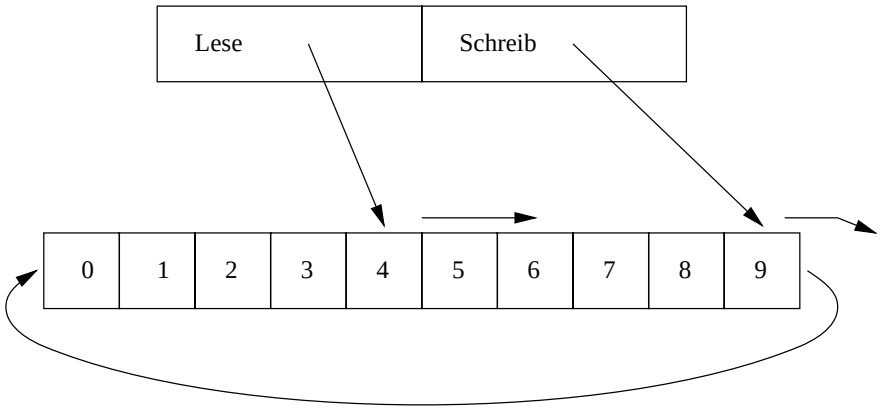


Abbildung 3.20: Implementierung einer pipe über inode

- `int fds[2];`
  `pipe(fds);`
  Liefert eine unbenannte pipe mittels der beiden Filedeskriptoren fds[0] für das Lesen und fds[1] für das Schreiben.

- `mkfifo(path, access-mode)`
  `char* path, mode_t access-mode;`
  Liefert eine pipe mit Namen `path`. Die Lese- und Schreibdeskriptoren werden wie üblich durch nachfolgendes Öffnen der Datei zum Lesen bzw. Schreiben erhalten. Da die pipe einen Namen hat und explizit zum Lesen/Schreiben geöffnet wird, eignet sie sich zur Kommunikation nicht verwandter Prozesse.

Kapitel 4

Interprozeßkommunikation (IPC)

4.1 Zugriffskonflikte (race conditions)

Zugriffskonflikte können immer dann auftreten, wenn mehrere Akteure eine gemeinsame Speicherstelle benutzen, in die von mindestens einem Thread geschrieben wird. Gemeinsames Lesen ist natürlich unkritisch. Solche Konflikte werden durch **Synchronisation** bereinigt. Durch **Kommunikation** wird darüberhinaus erreicht, daß verschiedene Akteure gemeinsam ein Problem lösen können. Beides gemeinsam nennen wir **Interaktion**.

Zur Interaktion zwischen verschiedenen Akteuren wird entweder gemeinsamer Speicher verwendet, oder es werden Nachrichten ausgetauscht. Akteure sind immer Kontrollströme (**Threads of control**). Liegen die Threads alle innerhalb eines Prozesses mit gemeinsamem HSP (SM-Threads), so kann die Interaktion durch Bibliothekscode geschehen, der auf Benutzerebene im gemeinsamen HSP agiert. Liegen die Threads in verschiedenen Prozessen mit verteiltem Speicher (DM-Threads), so stellt das BS die Mittel zur Interaktion bereit.

Viele IPC Probleme lassen sich am einfachsten an SM-Threads studieren. Es gilt grundsätzlich: alle als `global` oder `static` deklarierten Variablen in C haben einen gemeinsamen Wert für alle Threads, alle anderen Variablen sind Thread-lokal, d.h. sie haben in jedem Thread einen lokalen Wert.

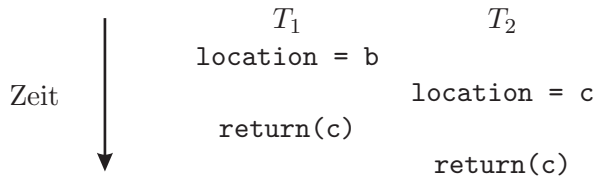
Bei Zugriffskonflikten führt Pseudoparallelität durch verschachteltes Ausführen (**preemptive scheduling**) mehrerer Threads auf einem einzigen Prozessor zu ähnlichen Problemen wie echte Parallelität auf mehreren Prozessoren. Grundsätzlich gibt es zwei Problemtypen, **Ausfall** und **Inkonsistenz**.

- **Ausfall**:

```
    char writer (char arg)
{ static char location;
  location = arg;
```

```
    return(location);
  }
```

Sei `location == a` und seien `x = writer(b)` und `y = writer(c)` auf Threads T_1 und T_2 aktiv. Folgendes kann geschehen:

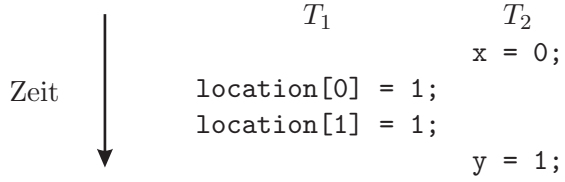


Eine Schreib-Anweisung geht also verloren. Analog wird, falls T_2 zuerst schreibt, das Resultat `"b"` zurückgegeben. Selbst wenn T_1 und T_2 die Schreib-Anweisung exakt gleichzeitig ausführen, so wird doch eine *nach* der anderen im HSP wirksam, da der Bus-Schiedsrichter (**bus arbiter**) den Bus erst dem einen und dann dem anderen Schreibzugriff zuteilt.
Anmerkung: Zukünftig wird es auch Maschinen mit **schwacher Schreibordnung** (**weak store order**) geben. Hier schreibt jeder Prozessor zunächst in einen Puffer (store buffer). Der Puffer wird erst später in den Cache und den HSP übertragen. Da nur die Caches kohärent sind, nicht aber die Schreibpuffer, ist auf solchen Maschinen auch das Resultat `"b"` auf T_1 und `"c"` auf T_2 möglich. Schreibpuffer müssen bei Bedarf aktiv (durch System calls) in den Cache gespült (*flush*) werden. □

- **Inkonsistenz**:

```
  writer (a)                    reader (a)
  int a[2];                     int a[2];
{ a[0] = 1;                   { x = a[0];
  a[1] = 1;                     y = a[1];
}                             }
```

Gilt `int location[] = {0,0};` und führen wir `writer(location);` und `reader(location);` auf T_1 und T_2 gleichzeitig aus, so kann folgendes geschehen:



Der reader auf T_2 liest einen inkonsistenten Zustand der Reihung. Dies kann immer dann geschehen, wenn mindestens einer der Akteure schreibt und die zu schreibenden Daten in mehr als einer elementaren Operation geschrieben werden. (Normalerweise gibt die Breite des Busses an, wieviele Bits elementar übertragen werden. Üblicherweise sind dies 32 oder 64, beginnend von einer Wort-Grenze.)

Es gibt zwei Ansätze zur Bereinigung von Zugriffskonflikten: Das Synchronisieren von Code-Sequenzen und das Schützen von Datenstrukturen. Code-Sequenzen, deren Ausführung zu Zugriffskonflikten führen kann, heißen **kritische Abschnitte** (KA) (**critical section**, **critical region**). Wir werden allgemein von **kritischen Ressourcen** (KR) sprechen, wenn wir kritische Abschnitte oder gemeinsame Datenstrukturen (shared datastructures) meinen. Zugriffe auf KR werden dadurch synchronisiert, daß man die KR für einen einzigen Akteur exklusiv **reserviert**. Der Exklusivzugriff heißt **lock**; demgemäß sprechen wir von **code locking** oder **data locking**.

Beispiel: Gegeben eine Prozedur `access(data)` zum Zugriff auf data. Bei **code locking** sorgt man dafür, daß `access` immer nur von einem einzigen Thread exklusiv ausgeführt werden darf. Bei **data locking** sorgt man dafür, daß die übergebenen aktuellen Parameter individuell geschützt werden. `access` darf von mehreren Threads ausgeführt werden, allerdings nicht gleichzeitig auf derselben Datenstruktur. □

Elementares Hilfsmittel zum locking ist die **Schloßvariable** (**mutual exclusion lock**, **mutex**) mit den Zugriffsfunktionen `lock()` und `unlock()`. Die C Threads Schnittstelle enthält die Funktionen

```
void mutex_lock(m)
     mutex_t m;
```

und

```
void mutex_unlock(m)
     mutex_t m;
```

Ein Aufruf von `mutex_lock` blockiert, bis m geschlossen ist, `mutex_unlock` öffnet m. Von mehreren gleichzeitigen Aufrufen zu `mutex_lock` ist maximal *einer* erfolgreich.

Jeder KR kann nun eine Schloßvariable zugeordnet werden. Durch Ausführen von `mutex_lock()` wird die KR exklusiv reserviert.

Code-locking von KA kann durch die Programmiersprache unterstützt werden, etwa durch die Konstrukte `enter` KA und `leave` KA (vgl. das **Monitor** Konzept bzw. `synchronized` in JAVA).

Ein wichtiger Gesichtspunkt für die Effizienz ist die **Granularität** (**grain size**) der Reservation. Die Reservation (**locking**) ist **grob-granular** (**coarse grained**), falls sich die Akteure relativ lange in den KR aufhalten, andernfalls **mittel-granular** (**medium grained**) oder **feingranular** (**fine grained**). Grobgranular wäre etwa die Ebene einer Datei, mittelgranular eine Gruppe von Records, feingranular ein Record oder einzelne Komponenten eines Record.

Grundsätzlich gilt folgender Antagonismus:
Feingranulare Reservation erhöht den Parallelismus im Programm, erhöht aber auch die Kosten (*overhead*) der Reservationen. Grobgranulare Reservation minimiert die Kosten, aber auch den Parallelitätsgrad.

Code-Synchronisation ist typischerweise grobgranular (wie im obigen Beispiel zu sehen), da es nur wenig Code gibt. Deswegen setzt sich mehr und mehr die Daten-Synchronisation durch, die feiner granular sein kann, weil es mehr Daten gibt. Daten-Synchronisation ist typisch für Threads-Programmierung, wo das *locking* effizient als user-code laufen kann.

Wir betrachten im folgenden Techniken zum Erreichen **wechselseitigen Ausschlusses** (*mutual exclusion*) zweier (oder mehrerer) Prozesse in kritischen Abschnitten. Für eine gute Lösung müssen 4 Bedingungen erfüllt sein:

1. [Korrektheit]
   Nie dürfen 2 Prozesse gleichzeitig in einem gemeinsamen kritischen Abschnitt sein.

2. [Allgemeinheit, Portabilität]
   Es dürfen keine Annahmen über Geschwindigkeit oder Anzahl der CPUs in die Lösung eingehen.

3. [Effizienz]
   Kein Prozeß, der außerhalb eines kritischen Abschnitts läuft, darf einen anderen Prozeß aufhalten (blockieren).

4. [Fairness]
   Kein Prozeß muß je ewig darauf warten, einen kritischen Abschnitt ausführen zu dürfen.

Lösungen zu diesem Problem fallen traditionell in den Bereich der Systemprogrammierung, da nur das Betriebssystem mehrere Prozesse verwaltet und Zugriff auf gemeinsame Betriebsmittel synchronisiert. Mit der Einführung von Parallelrechnern, auch im Workstation-Bereich, muß sich nun auch der Ersteller paralleler Anwendungsprogramme mit Synchronisation beschäftigen. Typischerweise werden hierzu Threads Systeme verwendet.

4.2 Wechselseitiger Ausschluß

4.2.1 Deaktivieren von Unterbrechungen

Funktioniert theoretisch bei einer einzigen CPU. Der Prozeß kann bei deaktivierten Interrupts nicht mehr unterbrochen und somit auch nicht von der CPU genommen werden. Er kann damit einen kritischen Abschnitt exklusiv bearbeiten.

Nachteil: Interrupts sollten nur vom BS deaktiviert werden können (System-Aufgabe). Funktioniert nicht bei Mehrprozessorsystemen (verletzt 2, anfällig für 3+4 bei Programmfehlern).

4.2.2 Strikte Abwechslung (strict alternation)

Aus historischen Gründen und zum Vergleich betrachten wir eine Lösung rein in Software. Auf Einzel- und eng gekoppelten Mehrprozessorsystemen mit gemeinsamem HSP sind diese Lösungen zu kompliziert und ineffizient.

Gegeben 2 Prozesse P_0 und P_1. Eine gemeinsame Variable `turn` wird auf 0 oder 1 gesetzt und zeigt an welcher Prozeß „dran“ ist in den KA einzutreten.

```
    P0                                    P1
  while (1)                             while (1)
{ while (turn != 0)   /* wait */;   { while (turn != 1);
  critical_section();                   critical_section();
  turn = 1;                             turn = 0;
  rest0();                               rest1();
}                                     }
```

Problem: Nachfolgeprozeß von $P_0(P_1)$ ist in $P_0(P_1)$ hart codiert durch Setzen von turn $\longrightarrow$ nicht modulares, fehleranfälliges Programmieren; schwer durchblickbare Abhängigkeiten. Verletzt Effizienzbedingung: Falls P_0 schnell ist, muß P_0 u.U. warten bis der langsame P_1 einen nicht kritischen Abschnitt beendet hat, um dann den KA bearbeiten zu können. $\longrightarrow$ Ganz schlechte Lösung.

4.2.3 Peterson's Lösung (in Software)

Gegeben P_0 und P_1. Das array `interested[2]` zeigt an, welcher Prozeß Interesse hat, in den KA einzutreten. Bei Interessenkonflikt entscheidet die gemeinsame Variable `loser`, wer verliert. Es wartet, wer `loser` zuletzt mit seiner Prozeßnummer gesetzt hat. Die Prozesse führen folgende Prozeduren zu Beginn und Ende des KA aus.

```
void  enter_region (process)
int  process;
{  int other = 1 - process;       /* other process */
   interested [process] = 1;
   loser = process;
   while (loser == process && interested [other])  /* wait */;
}
void  leave_region (process)
int  process;
{ interested [process] = 0; }
```

Falls P_0 und P_1 enter_region() gleichzeitig ausführen, so wird trotzdem `loser` nacheinander mit 2 verschiedenen Werten beschrieben und der letzte Wert bleibt. Der letzte Prozessor

wartet somit. Korrektheit des Protokolls wird bewiesen, indem man alle möglichen Verschachtelungen der Befehlssequenzen in 2 Aufrufen von enter_region durchspielt.

Die Lösung ist nicht allgemein, da sie nur für 2 Prozesse funktioniert. Die Lösung ist nur für Prozeßnummern aus 1 Byte portabel, da mehrere Bytes nicht unbedingt atomar geschrieben werden.

Bemerkung: Bei Maschinen mit weak store order ist diese Lösung nicht korrekt: P_0 setzt `interested[0] = 1`; P_1 sieht dies aber nicht, wenn der store buffer nicht in den HSP übertragen wurde, und tritt auch nach P_0 in die KR ein.

4.3 Mutual Exclusion Lock (Mutex)

4.3.1 Lock Variables (Schloßvariablen)

Schloßvariablen: 1 Bit, entweder s==0 oder s==1. Vor Eintritt in den KA testet der Prozeß s. Falls s==0, wird s=1 gesetzt und fortgefahren. Falls s==1, wird aktiv gewartet `while (s) /* wait */;`.

Lösung ist in dieser Form falsch, Problem ist nur verschoben: Wenn Prozeß nach Test s==0 unterbrochen wird, kann ein anderer Prozeß s==0 lesen und s=1 setzen, danach setzt erster Prozeß s=1 und beide Prozesse sind im KA. Eine wirkliche Lösung erhalten wir durch einen nicht-unterbrechbaren Befehl: „Teste und Setze“ s.u.

4.3.2 Die TSL Instruktion

Praktisch alle modernen Mikroprozessoren sind durch eine TSL Instruktion (test and set lock, Teste & Setze) auf Verwendung in Multiprozessorsystemen vorbereitet.
`tsl register, lock`
Liest das Speicherwort `lock` in `register` und speichert anschließend 1 (bzw. `0xF`) nach `lock`. Der Speicherbus wird während der Ausführung gesperrt, sodaß Lesen und Schreiben zusammen *atomar* ablaufen, also nicht nach dem Lesen ein anderer gleich schreiben kann.

Beispiel: Instruktion LDSTUB (Atomic Load-Store Unsigned Byte) in der SPARC Architektur.

4.3.3 Mutex als Spin-Lock

Wir nehmen an, der TSL Befehl sei in C als Prozedur `int TSL(& lock)` aufrufbar, wobei als Ergebnis der alte Wert der `lock`-Variablen zurückgegeben wird.

Mit dieser Variablen kann wie folgt ein wechselseitiger Ausschluß mit aktivem Warten (ein *spin lock*) realisiert werden:

```
            void mutex_lock (lock);
            int  * lock;
  {  while  (TSL (lock))    /* wait */;  }

            void mutex_unlock (lock);
            int  * lock;
  {  * lock  = 0; }
```

Enter_region und leave_region ergeben sich nun aus mutex_lock und mutex_unlock mit einer lock-Variable *mutex* pro Region, deren Adresse übergeben wird.

Hiermit lassen sich nun Zugriffskonflikte bereinigen. Aktives Warten kostet aber Prozessorzeit, so daß es zur Realisierung langer Wartezeiten in kooperierenden Threads nicht geeignet ist.

4.3.3.1 Mutex für Multiprozessoren

Die obige Implementierung kann auf bestehenden Mehrprozessorsystemen sehr ungünstig sein. Nehmen wir an, Prozessor P_0 hält ein mutex m, und Prozessoren $P_1, \ldots, P_n$ führen mutex_lock (m) aus.

Üblicherweise wird m (bzw. die Schloßvariable selbst) in jeden Prozessor-Cache geladen. Da jede Ausführung von TSL aber den Wert 1 in die Schloßvariable schreibt, werden alle Cache-Kopien fortlaufend invalidiert und vom Hauptspeicher nachgeladen. Dies führt zu erheblichem Betrieb auf dem Bus, obwohl sich am Inhalt der Schloßvariablen nichts ändert. Hierdurch wird P_0 erheblich an HSP-Zugriffen gehindert, so daß ein unnötig langer sequentieller Flaschenhals entsteht.

Man kann diese Situation zunächst verbessern, indem $P_1, \ldots, P_n$ an einer cache-lokalen Kopie der Schloßvariablen warten:

```
            void mutex_lock (lock_p)
            char* lock_p;
{
   while ( TSL(lock_p)) {
         while (* lock_p)          /* wait on local cache copy */ ;
         }
}
```

Wir warten nun aktiv auf eine Änderung in `* lock_p`. Dies geschieht durch fortlaufendes reines Lesen in der lokalen Kopie von `* lock_p` in jedem Cache. Bus und Hauptspeicher bleiben unbelastet.

Irgendwann wird P_0 ein `mutex_unlock(lock_p)` ausführen. Dadurch wird `* lock_p = 0` gesetzt und die Kopien in allen Caches invalidiert und mit 0 nachgeladen. Daraufhin brechen die Warte-Schleifen und jeder Prozessor versucht erneut ein `TSL(lock_p)`.

Hierdurch gibt es erneut einen Aktivitätsschub (*burst of activity*) auf dem Bus. Erst werden alle Kopien von `* lock_p` zu 0 zurückgesetzt; danach führt jeder Prozessor ein TSL aus und einer gewinnt; dadurch werden alle Kopien von `* lock_p` invalidiert und auf 1 gesetzt. Eine Lösung ist mit zusätzlichem Aufwand möglich, indem die Änderung `mutex_unlock()` gezielt an einen der Prozessoren $P_1, \ldots, P_n$ signalisiert wird.

4.3.3.2 Prioritäten-Inversion

Aktives Warten kann aber den Effekt der *Prioritäten-Inversion* haben. Seien zwei Prozesse P_1 und P_2 gegeben, wobei P_2 höhere Priorität als P_1 habe. P_1 hält ein lock m, P_2 ist blockiert. Wird P_2 nun deblockiert, so verdrängt er P_1; führt P_2 nun `while (TSL (m))`; aus, dann wartet er auf den P_1 niederer Priorität, der m aber nicht hergeben kann, da er verdrängt ist.

4.3.3.3 Mutex mit Warten

In einer Implementation von `mutex_lock` in C Threads wird daher üblicherweise statt `while (TSL(m));` nur

```
for (i=0; i < mutex_spin_limit; i++)
    if (! TSL(m)) return;
```

ausgeführt und, falls das `spin_limit` überschritten wurde, der Prozeß in der Warteschlange vor m blockiert (d.h. zu passivem Warten umgeschaltet). `Mutex_unlock` weckt dann alle Prozesse in der WS wieder auf.

Bemerkung: Die WS muß nun durch ein separates reines Spin lock geschützt werden. Dies ist kein Problem, da das Ein- und Ausketten garantiert schnell geht und niemand lang warten muß.

4.4 Inaktives Warten (Sleep, Wakeup)

Neben der Synchronisation ist von ganz besonderer Bedeutung, daß verschiedene Prozesse **kooperieren** können. Typisch ist der Produzent/Konsument-Fall, wo der Produzent Daten in einem Puffer ablegt, die der Konsument entfernt. Bei leerem Puffer *wartet* der Konsument, bis der Puffer gefüllt ist, bei vollem Puffer wartet der Produzent, bis er geleert ist.

Dieses Warten dauert i.a. zu lange, als daß es effizient als busy waiting realisiert werden könnte, z.B. durch Blockieren in mutex_lock().

Hierzu werden zwei neue Systemaufrufe eingeführt.

`SLEEP()` blockiert den Aufrufer, so daß er von Arbeit suspendiert wird, ohne einen Prozessor zu belasten.

`WAKEUP(pid)` weckt den Prozeß mit der ID pid auf.

Alternativ wird eine *Ereignisvariable* event eingeführt und die Aufrufe
`WAIT(event)` bzw. `SIGNAL(event)`
realisieren das Warten auf das Ereignis, das logisch zu event dazugehört. (In der Implementierung werden die Leitblöcke wartender Prozesse in einer in `event` verankerten Warteschlange gehalten. `SIGNAL(event)` löst einen (oder alle) Prozesse aus der Schlange und macht sie lauffähig durch Einketten in die run_queue.)

Betrachten wir das Produzent/Konsument-Problem. Wir nehmen an, enter(item) und remove(& item) regeln den Zugriff auf den Puffer der Größe N.

Mit einer gemeinsamen Variablen `count` haben wir dann:

Producer/Consumer (1. Näherung, inkorrekt)

```
void producer()
{ int item;
  while (1)
  { produce_item (& item);
    if (count == N)  sleep();      /* if buffer full */
    enter(item);
    count = count + 1;
    if (count == 1)  wakeup(consumer);
  }                                /* if buffer was empty,
}                                     hence consumer slept */

void consumer()
{ int item;
  while (1)
  { if (count == 0)  sleep();      /* if buffer empty */
    remove (& item);
    count = count - 1;
    if (count == N-1)  wakeup(producer);
    consume (item);                /* if buffer was full,
  }                                   hence producer slept */
}
```

Diese Version ist offensichtlich falsch, da der Zugriff auf die gemeinsame Variable `count` nicht synchronisiert wird. Für `count` besteht das Problem „Ausfall“, nicht aber das Problem „Konsistenz“. Allerdings besteht das Problem „Konsistenz“ zwischen dem tatsächlichen Zustand des Puffers und der Variable `count`. Es könnte nämlich sein, daß ein Producer den Puffer verändert, während der Consumer `count` verändert. Deshalb muss man die Bedingung `count=0` erneut testen, also `if` durch `while` ersetzen (bzw. `count==N` bei mehreren Producern/Consumern).

Producer/Consumer (2. Näherung):

```
mutex_t lock;
void producer()
{int item;
 while (1)
 { produce(& item);
   mutex_lock(lock);
   if (count == N)  {mutex_unlock(lock);  sleep();  mutex_lock(lock);}
   enter(item);
   count = count + 1;
   if (count == 1)  wakeup(consumer);
   mutex_unlock(lock);
 }
}


void consumer()
int item;
{while (1)
 { mutex_lock(lock);
   if (count == 0)  {mutex_unlock(lock);  sleep();  mutex_lock(lock);}
   remove(& item);
   count = count - 1;
   if (count == N-1)  wakeup(producer);
   mutex_unlock(lock);
   consume(item);
 }
}
```

Diese Lösung ist immer noch falsch, da z.B. der Produzent zwischen mutex_unlock() und sleep() unterbrochen werden kann. Ein dann folgender wakeup-Aufruf des Konsumenten verpufft wirkungslos, da der Produzent noch nicht schläft. In der Folge schlafen bald beide Prozesse.

Falls die statements `if (count > 0) wakeup(consumer);` bzw.
`if (count < N) wakeup(producer)` benutzt werden, verringert sich nur die Wahrscheinlichkeit, daß der Fehler auftritt; die Lösung ist immer noch fehlerhaft bzw. verletzt Bedingung 2.

Es gibt hierzu 2 weitere Lösungen.

- Man kann das Aufschließen und Einschlafen in einem Befehl kombinieren, wogegen
- Dijkstras *Semaphore* die gesendeten Signale zählen, damit keines verloren geht.

4.4.1 C Threads Conditions

Conditions sind die Realisierung der Ereignisvariable in C Threads. Sie erlauben es, Zustandsänderungen von gemeinsamen Speicherobjekten zu signalisieren bzw. auf sie zu warten. Jedes Speicherobjekt ist typischerweise durch ein mutex geschützt. Ein thread erwirbt das Zugriffsrecht, untersucht das Objekt und findet, daß er auf eine Bedingung warten muß. Das Zugriffsrecht muß abgegeben werden, damit ein anderer thread die Daten verändern kann, bis die Bedingung erfüllt ist. Dies geschieht atomar, d.h. `mutex` wird erst (vom System) freigegeben, wenn der Thread schläft und das Aufwach-Signal empfangen kann.

Nach Empfangen des Signals, die Bedingung sei nun erfüllt, wacht der blockierte thread auf mit dem Zugriffsrecht in der Hand, d.h. im Zustand, in dem er zuletzt war. Er muß jedoch die Bedingung erneut prüfen, denn ein Dritter, der zusammen mit ihm auf die Erfüllung der Bedingung gewartet hat, könnte ihm zuvorgekommen sein und die Daten schon wieder verändert haben (wird die Bedingung nicht geprüft, ist Anforderung 2 verletzt).

```
void  condition_signal(c)
      condition_t c;

void  condition_wait(c,m)
      condition_t c;
      mutex_t m;
```

Producer/Consumer: Lösung mit C Threads.

Mittels C Threads conditions: Hausaufgabe. (C Threads paper p. 15).

4.5 Semaphore

Das Semaphor-Konzept wurde von E. W. Dijkstra erfunden. Im ursprünglichen Sinn reguliert ein Semaphor die Anzahl der Prozesse, die sich in einer KR befinden können bzw. die Anzahl der Exemplare einer KR, die verfügbar sind. Das Semaphor wird mit dieser Zahl initialisiert. Es gibt 2 Operationen

P(s) „Passiere“ den Eingang zur KR (pass).

V(s) „Verlasse“ die KR (leave).

P(s) blockiert (schläft) nötigenfalls bis s>0 und führt dann s=s-1 aus. Testen, Setzen und Schlafengehen ist eine atomare Aktion.

V(s) führt s=s+1 aus und weckt *einen* auf s wartenden Prozeß auf, falls es einen solchen gibt.

Implementierung:

Ein Semaphor kann als Kombination von Zähler und mutex betrachtet werden. Ein Spezialfall sind *binäre* Semaphore, initialisiert zu 1. Auf diesen sind P(s) und V(s) äquivalent zu mutex_lock und mutex_unlock.

Üblicherweise assoziiert man mit Semaphoren aber inaktives Warten. In diesem Fall benötigt man aus Sicht von C Threads zusätzlich eine Bedingungsvariable, etwa `non_empty`.

Allgemein regelt ein Semaphor die Verteilung von k Exemplaren eines beschränkten Betriebsmittels KR. `P(s)=Down(s)` vergibt ein Exemplar, `V(s)=Up(s)` gibt ein Exemplar zurück.

Ein Semaphor mit inaktivem Warten kann aber auch zur Realisierung von *Sleep, Wakeup* benutzt werden. `Down(e)` wartet auf das Ereignis e, `Up(e)` signalisiert e. Es gehen keine Weckrufe verloren, da sie von der Semaphor-Variable gezählt werden.

Producer/Consumer: Lösung mit Semaphoren.

```
            typedef int * semaphore; semaphore mutex, empty, full;
            producer
            {  produce(& item);     /* Außerhalb des KR nebenläufig */
               down(empty);         /* Warte auf ein leeres Fach */

synchron.      down(mutex);         /* Zugangsrecht vergeben */
Eintragen      enter(item);
von item       up(mutex);           /* Zugangsrecht zurückgegeben */

               up(full);            /* Signalisiere Existenz eines vollen Fachs */
            }


            consumer
                :
            {  down(full);          /* Warte auf ein volles Fach */

synchron.      down(mutex);         /* Zugangsrecht */
Austragen      remove(& item);
von item       up(mutex);           /* Zugangsrecht zurück */

               up(empty);           /* Signalisiere Existenz eines leeren Fachs */
               consume(item);       /* außerhalb KR nebenläufig */
            }
```

Die Semaphore `empty` und `full` werden als Bedingungsvariable benutzt, das Semaphor `mutex` wird als Schloßvariable benutzt. Das Semaphor `empty` wird mit der Anzahl der leeren Fächer initialisiert, das Semaphor `full` mit 0.

4.6 Nachrichtenaustausch (Message Passing)

Die bisher betrachteten Konzepte verlangen alle einen gemeinsamen Speicher, der für verteilte Systeme nicht unbedingt gegeben ist. Selbst bei gemeinsamem Speicher ist es nützlich, wenn logisch getrennte Prozesse sich Nachrichten zuschicken, statt explizit gemeinsame Variablen zu benutzen. Das BS kann die Nachrichten dann immer noch via gemeinsame Speicherbereiche zustellen, aber eben auch über das Netz, falls die Prozesse verteilt laufen. Dadurch ist auch besserer Speicherschutz gewährleistet. Dies ist die Philosophie von MACH mit *message ports* (Nachrichtenpforten) in jedem Prozeß.

Das Producer/consumer-Problem kann nun gelöst werden, indem jedes „item“ in einer Nachricht vom Produzenten zum Konsumenten verschickt wird. Eine feste Anzahl von Nachrichtenumschlägen entspricht einem Puffer fester Größe.

Nachrichten können im *Rendezvous* synchron übermittelt werden oder asynchron, wobei sie in einem *Briefkasten* (mailbox) zwischengespeichert werden. (MACH *ports* sind vom BS verwaltete und geschützte Briefkästen.)

Rendezvous (siehe Ada)

Ein send(Empfänger) des Senders muß mit einem receive(Sender) gleichzeitig aktiv sein. Einzelne send- bzw. receive-Aufrufe blockieren, bis sich ein Partner einstellt. Vorteil: Keine Zwischenpufferung nötig. Nachteil: Prozesse sind in Realzeit gekoppelt.

Mailbox

Ein Briefkasten ist ein Speicherplatz für Meldungen. Ein send(mailbox) blockiert nur dann, wenn mailbox bereits voll ist, ein receive(mailbox) nur dann, wenn mailbox leer ist.

Producer/Consumer: Lösung mit Messages

```
mailbox_t fullbox, emptybox;
      producer()
{ int item; message_t m;
  while (1)
  { produce(&item);
    receive(emptybox, &m);     /* Receive empty envelope message */
    build(&m, item);           /* Put item in a message envelope */
    send(fullbox, &m);
  }
}
      consumer()
{ int item; message_t m;       /* Initialize: Create N empty envelopes */
  for (i = 0; i < N, i ++) send (emptybox, &m);
  while (1)
  { receive(fullbox, &m);
    extract(&m, &item);        /* Get item from message */
```

```
    send(emptybox, &m);        /* Send envelope back */
    consume(item);
  }
}
```

4.7 Monitors (Ausführungskontrolleure)

Monitors (erfunden von Per Brinch Hansen und verbessert von Hoare) sind ein höheres Programmiersprachenkonzept, das das Schreiben nebenläufiger Programme unterstützt.

Dem objektorientierten Programmieren verwandt, werden Datenobjekte und darauf agierende Prozeduren (z.B. Puffer, Produzent, Konsument) zusammengefaßt. Der Compiler übersetzt solch einen Monitor derart, daß sich jeweils nur ein einziger Prozeß im Monitor aufhalten kann, d.h. code aus dem Monitor ausführen darf. Damit entfällt für den Programmierer die Notwendigkeit, einzelne Datenobjekte separat durch locks zu schützen. Es entsteht aber die Gefahr der Ineffizienz durch zu grobe Granularität, da eventuell nur kleine Teile der Monitorprozeduren wirklich exklusiv zu sein brauchen.

Nehmen wir

```
 Monitor producer_consumer
{buffer b ...
 producer
  { ...
  }
 consumer
  { ...
  }
}
```

So laufen nun auch `produce(&item)` und `consume(item)` als KR's. Besser ist es, nur `enter` und `remove` zu schützen.

Wiederum benötigen wir die Möglichkeit der Kooperation, d.h. daß ein Prozeß den Monitor aufgibt, um auf eine Bedingung zu warten und dann im Monitor wieder aufwacht.

Dies geschieht mit Bedingungsvariablen (condition variables) und den Aufrufen

```
condition c;
wait(c);
signal(c);
```

Signal beinhaltet dabei ein Verlassen des Monitors, da sonst der aufgeweckte Prozeß als zweiter im Monitor aktiv wäre.

Bemerkung: Vergleiche, wie in C Threads wait(c,m) durch das mutex m des Monitor-Begriffs unterstützt wird. m entspricht dem impliziten mutex eines Monitors. Der aufgeweckte Prozeß

blockiert (aktiv) außerhalb des Monitors, bis er durch Erwerb des mutex wieder eintreten darf.

Producer/Consumer: Lösung mit Monitor

```
Monitor ProCon
{ condition nonfull, nonempty;
  buffer B[N];
  int count = 0;

  void enter(x)
  int x;
  { if (count == N) wait(nonfull);
    B[count] = x;
    count = count + 1;
    if (count == 1) signal(nonempty);
  }
  void remove(y)
  int * y;
  { if (count ==0) wait(nonempty);
    *y = B[count];
    count = count - 1;
    if (count == N - 1) signal(nonfull);
  };
}
    void producer()
  { item_t item;
    while (1) { produce(& item);
                ProCon.enter(item);
              }
  }
    void consumer()
  { item_t item;
    while (1) { ProCon.remove(& item);
                consume(item);
              }
  }
```

4.8 Äquivalenz der Konzepte

Alle betrachteten Konzepte (Messages, Monitors, Semaphores, mutex/conditions) sind äquivalent, d.h. sie können wechselseitig implementiert werden. Die Programmiersprachen concurrent Euklid und -Pascal bieten Monitors an, `synchronized` in JAVA ist äquivalent zu Monitors, UNIX System V bietet Semaphore und Nachrichten, C Threads bieten mutex/conditions.

4.9 Threads und Synchronisation in Java

Die Programmiersprache Java stellt eigene Threads zur Verfügung. Das Synchronisationskonzept beruht auf Monitoren, die implizit Locks auf Klassen bzw. auf jedem Objekt nutzen. Monitore sind einzelne Methoden oder Anweisungs-Sequenzen. Aus Datenstruktur-Sicht ist die Locking-Granularität die einzelner Objekte. Das Kooperationskonzept beruht auf Signal (in Java: Notify) und Wait, wobei pro Klasse nur eine implizite Bedingung (condition) vorgesehen ist.

4.9.1 Java Threads

Ein Java Thread ist durch ein `Thread` Objekt gegeben. `Thread` Objekte haben die Methoden `public void run()` und `void start()`. Sei `Thread th;` gegeben. Ein Aufruf `th.start();` startet die Methode `run();` von `th` als separaten Thread of Control.

Es gibt zwei Methoden, eine gegebene Funktion `f()` als Thread zu starten. Entweder man leitet von `Thread` eine Klasse `ThreadF` ab, in der man die `run()` Methode geeignet überschreibt. Oder aber man definiert eine neue Klasse `RunF`, die das Interface `Runnable` durch die Definition einer geeigneten `run()` Methode implementiert, und man konfiguriert das passende Thread-Objekt über den Konstruktor-Aufruf `Thread(runF)`, wobei nun `runF` ein Objekt vom Typ `RunF` ist.

Im Beispiel des Produzenten und Konsumenten haben wir die erste, etwas einfachere Methode gewählt (siehe Abbildung 4.2 und 4.3. Falls allerdings `f()` bereits in einer abgeleiteten Klasse definiert war, muß die zweite Methode gewählt werden, da man in Java nur von einer einzigen Klasse erben kann.

4.9.2 Synchronisations- und Kooperations-Konstrukte in Java

4.9.2.1 Producer / Consumer in Java

```
package ProducerConsumer;

class Buffer {

    private Object[] buffer;
    private int count;         // count many objects in buffer

    public Buffer(int size) {
        buffer = new Object[size];
        count=0;
    }

    synchronized void display() {
        for (int i = 0; i < buffer.length; i++) {
            System.out.print(buffer[i] + " ");
        }
        System.out.println();
    }

    synchronized void enter(Object x)
    throws InterruptedException {
        while (count == buffer.length) wait();
        buffer[count++] = x;  // insert position is count because buffer starts at 0
        if (count == 1) {
            display();
            notify();
        }
    }

    synchronized Object remove()
    throws InterruptedException {
        while (count == 0) wait();

        if (count == buffer.length) {
            display();
            notify();
        }
        return(buffer[--count]); //remove position is count-1 because buffer starts at 0
    }
}
```

Abbildung 4.1: Endlicher Puffer in Java

```
package ProducerConsumer;

class Producer extends Thread {

    static int seed = 0;

    Buffer buf;

    public Producer(Buffer buffer) {
        buf = buffer;
    }

    public void run() {
        Object item;
        while (true) {
            item = produce();
            try {
                buf.enter(item);
            } catch (InterruptedException e) {
                return; // end this thread
            }
        }
    }

    Object produce() {
        //Object res = new Integer((int)Math.random());
        Object res = new Integer(seed++);
        System.out.println(res + " produced.");
        return(res);
    }

    public static void main(String[] args) {
        Buffer buf = new Buffer(10);
        new Producer(buf).start();
        new Consumer(buf).start();
    }
}
```

Abbildung 4.2: Produzent in Java

```
package ProducerConsumer;

class Consumer extends Thread {
    Buffer buf;

    public Consumer(Buffer buffer) {
        buf = buffer;
    }

    public void run() {
        Object item;
        while (true) {
            try {
                item=buf.remove();
            } catch (InterruptedException e) {
                return; // end this thread
            }
            consume(item);
        }
    }

    void consume(Object item) {
        System.out.println(item + " consumed.");
    }

    public static void main(String[] args) {
        Buffer buf = new Buffer(10);
        new Consumer(buf).start();
        new Producer(buf).start();
    }
}
```

Abbildung 4.3: Konsument in Java

4.10 Synchronisation in UNIX

4.10.1 Mutex

Die einzige voll portable Synchronisationsoperation für Anwender ist die Realisierung von mutex_lock durch das Erzeugen eines Lock-Files. `Creat` läuft atomar ab und liefert einen Fehler, falls das File schon existiert, d.h. ein anderer Prozeß das lock hat. Es ist problematisch, das Filesystem so zu mißbrauchen; z.B. bleiben lock-files liegen, wenn ein Prozeß abstürzt oder ein Prozeß hängt, wenn zufällig ein anderes File mit dem gleichen Namen wie das lock-file existiert.

4.10.2 IPC in System V

UNIX System V enthält drei Mechanismen für IPC:

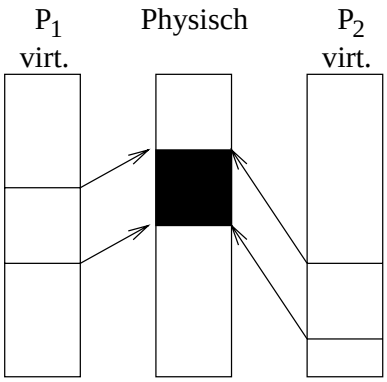
Shared Memory	erlaubt es Prozessen, einen Teil ihres HSP gemeinsam zu nutzen und so zu kommunizieren.
Messages	erlauben es Prozessen, sich gegenseitig formatierte Datenströme zuzusenden.
Semaphores	erlauben es Prozessen, ihre Ausführung zu synchronisieren.

Diesen Mechanismen ist folgendes gemeinsam:

- Eine Systemtabelle enthält alle Instanzen der Mechanismen.
- Jeder Eintrag enthält einen Benutzer definierten numerischen Schlüssel `key`.
- Ein `get` Systemaufruf durchsucht die Tabelle nach einem Eintrag mit Schlüssel `key`. Modifiziert durch `flags` können auch neue Einträge erzeugt oder Fehler geliefert werden, falls `key` schon existiert. `get` liefert einen Deskriptor, den die anderen Aufrufe benutzen.
- Jeder Eintrag enthält Zugriffsrechte und andere Statusinformation (z.B. letzter zugreifender Prozeß und Zeit des Zugriffs).
- Jeder Mechanismus enthält einen `control (ctl)` Aufruf, um den Zustand eines Eintrags abzufragen oder zu verändern oder den Eintrag zu löschen.

4.10.2.1 Shared Memory

Prozesse können direkt kommunizieren, indem sie sich einen Teil ihres virtuellen Adreßraums teilen und dort Daten schreiben und lesen.



`shmget`	gibt Zugriff auf oder erzeugt eine neue gemeinsame Speicherregion
`shmat`	bildet eine Speicherregion in den virtuellen Adreßraum eines Prozesses ab.
`shmdt`	löscht eine Speicherregion aus dem virtuellen Adreßraum eines Prozesses.
`shmctl`	verändert gewisse Parameter einer gemeinsamen Speicherregion.

Die gemeinsamen Speicherregionen werden in einer Tabelle gehalten.

```
shmid = shmget (key,size,flag)
```

sucht zuerst nach einer Region `key` in der Tabelle. Falls keine da ist (und flag = IPC_CREAT), wird ein neuer Eintrag erzeugt.

```
virtaddr = shmat (id,addr,flags)
```

bildet eine Speicherregion in den virtuellen Adreßraum des Aufrufenden ab (möglichst an der Stelle `addr`). Der tatsächliche Ankerpunkt ist virtaddr. Ist addr = 0, ist die Wahl dem System überlassen; sonst muß der Aufrufer selbst Konflikte (z.B. mit dem Stack) vermeiden. Seitentabellen für die Region werden erst beim ersten `shmat` erzeugt.

`shmdt(addr)` löst die Bindung einer Speicherregion an die virtuelle Adresse addr.

`shmctl(id,cmd,shmstatbuf)` kann den Status (z.B. Zugriffsrechte) einer Region ändern. `shmstatbuf` enthält den neuen Status. `shmctl(id,IPC_RMID,0)` gibt die Region zum Auflösen frei (dies geschieht erst beim letzten `shmdt`).

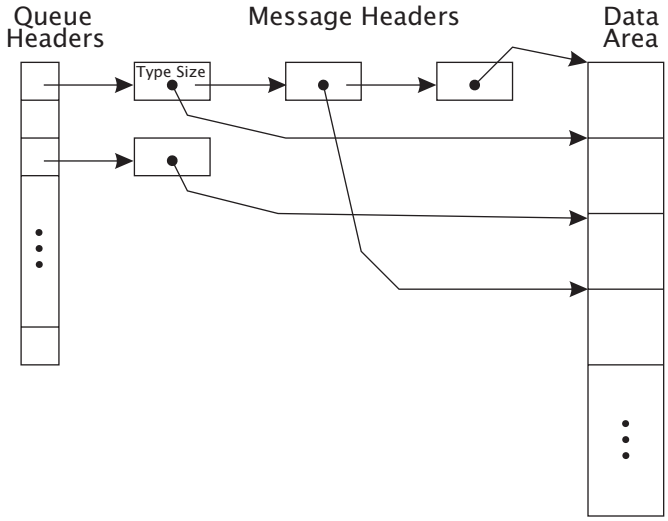
4.10.2.2 Messages

Messages dienen zum Austausch formatierter Daten.

`msgget`	liefert (und falls nötig erzeugt) einen Deskriptor für eine Nachrichtenwarteschlange (Briefkasten).
`msgsend`	dient zum Senden,
`msgrcv`	zum Empfangen einer Nachricht.
`msgctl`	setzt und liest Parameter (z.B. Zugriffsrechte) eines Deskriptors. Eine Option dient zum Löschen der Queue.

```
msgqid = msgget (key,flag)
```

liefert einen Deskriptor auf eine Liste von Nachrichten (Briefkasten).




```
msgsnd (msgqid,msg,count,flag)
```

sendet Nachricht `msg` der Länge `count` in die WS `msgqid`. `msg` selbst ist eine Struktur aus einem Integer Typ und der Nachricht als char array.

Es werden Prozesse geweckt, die auf eine Nachricht von diesem Typ warten.

```
count = msgrcv (id,msg,maxcount,type,flag)
```

empfängt eine Nachricht des Typs `type` und der max. Länge `maxcount` von Schlange `id`, die in der Struktur `msg` abgelegt werden soll. `count` ist die aktuelle Anzahl übermittelter Bytes. Es wird die erste Nachricht übermittelt, deren Typ = `type` ist; falls `type` = 0, wird die erste Nachricht übermittelt.

`msgctl(id,cmd,mstatbuf)` liest oder setzt den Status eines Nachrichtendeskriptors.

`msgctl(msgid,IPC_RMID,0)` löscht einen Deskriptor (eine NachrichtenWS).

4.10.2.3 Semaphore

Ein System V Semaphor besteht aus:

- Dem Wert des Semaphors.
- Der Prozeß ID des letzten Prozesses, der das Semaphor manipuliert hat.
- Der WS der Prozesse, die darauf warten, daß sich der Wert des Semaphors erhöht.
- Der WS der Prozesse, die darauf warten, daß der Wert des Semaphors Null wird.

Die Systemaufrufe sind

`semget`	Erzeugen und Erhalten von Zugriffsberechtigung auf eine Semaphor-Menge.
`semctl`	Verschiedene Kontrolloperationen auf der Menge.
`semop`	Werte der Semaphore verändern (entsprechend P und V).

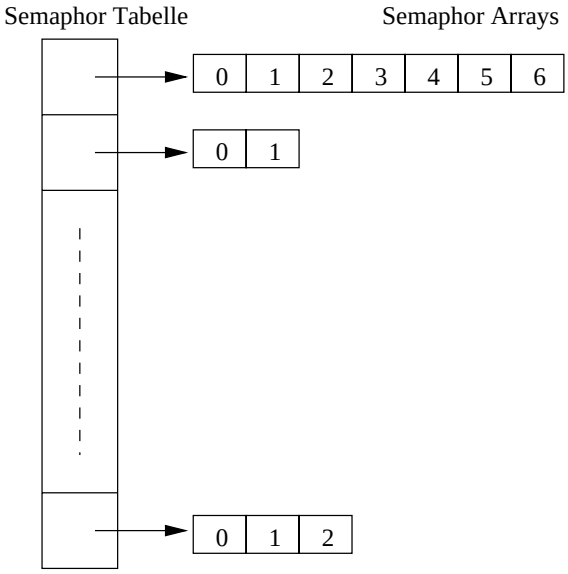


Abbildung 4.4: Semaphor Datenstruktur

```
Semget  id = semget (key,count,flag);
```

gibt Zugriff auf einen Eintrag in der Semaphor-Tabelle mit einem Array aus Semaphoren mit `count` Elementen. `key` ist ein Benutzer definierter Name für den Eintrag und `flag` spezifiziert Zugriffsarten (z.B. ob Eintrag erzeugt werden soll, falls er noch nicht existiert). Der Tabelleneintrag enthält auch die Zeiten der letzten semop und semctl Aufrufe sowie Zugriffsrechte für semop Aufrufe.

```
Semop  oldval = semop (id,oplist,count)
```

`id` ist der Deskriptor aus `semget`, `oplist` ist ein Zeiger auf ein Array der Länge `count` aus Semaphor-Operationen. `oldval` ist der Wert des letzten Semaphors in der Liste vor der Operation. Jedes Element von `oplist` enthält:

- den Index des Semaphors im Array
- die Semaphor-Operation in Form einer Konstanten Op die zum Sem-Wert addiert wird
- Flags

Als Resultat einer `semop` ändert der Kern die Werte der beteiligten Semaphore gemäß den spezifizierten Operationen. Falls der Wert erhöht wird (Op > 0), werden alle Prozesse geweckt, die darauf warten.

Falls Op $= 0$, wartet der Prozeß so lange, bis der Semaphor-Wert $= 0$ wird.

Falls Op < 0, so wird Wert $=$ Wert $+$ Op ausgeführt, falls der neue Wert dadurch nicht negativ wird (sonst wird auf die Erhöhung des Semaphors gewartet). Ist der neue Wert 0, so werden die auf 0 wartenden Prozesse geweckt.

In jedem Fall wird eine `semop` atomar durchgeführt. Blockiert der Prozeß bei einer Teiloperation, so werden auf dem Semaphor-Array die Originalwerte temporär wieder hergestellt. Dies macht System V Semaphore sehr kompliziert und aufwendig.

Semaphore werden durch den `semctl`-Aufruf gelöscht, der auch noch viele andere Funktionen hat.

4.11 Deadlocks (Verklemmungen)

Verklemmungen entstehen, wenn mehrere Prozesse sich gegenseitig beim Zugriff auf Betriebsmittel (BM) blockieren.

Es gibt wegnehmbare (*preemptable*) und nicht wegnehmbare (*non preemptable*) Ressourcen. Z.B. virtueller, in Seiten aufgeteilter Speicher ist preemptable, Printer non-preemptable. Deadlocks geschehen mit non-preemptable Betriebsmitteln.

Einfachster Deadlock:
P_1 hat BM_1, will BM_2.
P_2 hat BM_2, will BM_1.

Eine Menge von Prozessen ist verklemmt, wenn jeder auf ein Ereignis wartet (z.B. Freigabe eines BM), das nur ein anderer Prozeß in der Menge herbeiführen kann.

4.11.1 Beispiel: Essende Philosophen (Dining Philosophers)

Dieses Problem wurde von Dijkstra gestellt. 5 Philosophen essen, denken oder sind hungrig. Kritische Betriebsmittel sind Eßutensilien: Es gibt 5 Teller mit Reis und zu beiden Seiten jeden Tellers je ein Eßstäbchen (*chopstick*). Man benötigt 2 Stäbchen, um essen zu können.

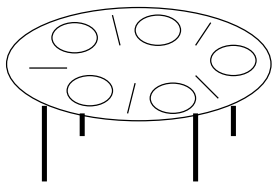


Abbildung 4.5: Der Tisch der Philosophen

Probleme:

1. Synchronisation: Die Betriebsmittel Teller und Stäbchen müssen zugeteilt werden.
2. Effizienz: Es soll kein Philosoph unnötig warten müssen (es soll nicht immer nur einer am Tisch sitzen).
3. Kein Verhungern: Es soll nicht vorkommen können, daß ein Philosoph verhungert (starvation), weil sich die anderen gegen ihn verbünden und ihm immer die Stäbchen blockieren.
4. Keine Verklemmung: Die Lösung soll nicht zu Verklemmungen führen können.

Die folgende Lösung mit Semaphoren erfüllt 1, 2 und 4. Problem 3 kann dagegen zufällig auftreten. Wir gehen von genau fünf Philosophen aus. Mehr Philosophen können durch ein Semaphor kanalisiert werden.

```
/* Dining Philosophers with Semaphores */

int state[5];
semaphore s[5];
semaphore mutex = 1;
#define left(i) ((i-1)%5)
#define right(i) ((i+1)%5)

void philosopher(i)
int i;
{ while(1)
  { think();
    take_sticks(i);
    eat();
```

```
void test(i)
int i;
{ if (state[i] == HUNGRY
  && state[left(i)]!= EATING
  && state[right(i)]!= EATING
  { state[i] = EATING;
```

```
    put_sticks(i);
  }
}

void take_sticks(i)
int i;
{ down(&mutex);
  state[i] = HUNGRY;
  test(i); up(&mutex);
  down(& s[i]); /* wait for test */
                /* to succeed */
}
```

```
    up(&s[i]);  /* signal successful test */
  }
}

void put_sticks(i)
int i;
{ down(&mutex);
  state[i] = THINKING;
  test(left(i));          /* left */
  test(right(i));         /* right */
  up(&mutex);
}
```

Problem 4 kann insbesondere auftreten, wenn zuerst das eine, dann das andere Stäbchen zugeteilt wird. Nehmen alle erst das linke Stäbchen, haben wir eine Verklemmung. Deswegen wird kein weiterer Zustandsübergang erlaubt, während Philosoph i testet, ob links und rechts nicht gegessen wird.

```
/* Dining Philosophers in C Threads */

int state[5];
condition_t sticks_available[5];
mutex_t table;

void philosopher(i)
int i;
{ while(1)

  { think();

    take_sticks(i);

    eat();

    put_sticks(i);

  }
}

void take_sticks(i)
int i;
{ mutex_lock(table);
  state[i] = HUNGRY;
  while(! ready(i)) condition_wait(sticks_available[i],table);
  state[i] =  EATING;
  mutex_unlock(table);
```

```
int ready(i)
int i;

{ if (state[(i-1)%5] !=  EATING

      && state[(i+1)%5] != EATING)

  return(1);

  else return(0);

}
```

```
}

void put_sticks(i)
int i;
{ mutex_lock(table);
  state[i] = THINKING;
  condition_signal(sticks_available[(i-1)%5]);
  condition_signal(sticks_available[(i+1)%5]);
  mutex_unlock(table);
}
```

4.11.2 Bedingungen für Verklemmungen

Verklemmungen können nur entstehen, wenn *alle* der folgenden Bedingungen erfüllt sind.

1. **Wechselseitiger Ausschluß:** Jedes BM ist entweder von genau einem Prozeß in Gebrauch oder es ist frei.
2. **Halte und Warte:** Prozesse, die schon BM haben, können später neue anfordern.
3. **Keine Wegnahme:** BM können nicht weggenommen, nur freigegeben werden.
4. **Zirkuläres Warten:** Zwei oder mehr Prozesse warten zirkulär auf Freigabe von BM.

Zur Vermeidung von Verklemmungen reicht es, jeweils eine der Bedingungen zu vermeiden. Es gibt vier Strategien:

1. **Ignorieren des Problems**
   Man sorgt für so viele BM, daß das Problem höchst selten auftritt; behandelt es aber nicht automatisch.
2. **Erkennung und Wiederaufsetzen**
   Man erkennt das Problem, wenn es aufgetreten ist und beseitigt es danach automatisch (z.B. durch reboot).
3. **Dynamische Vermeidung**
   Man kontrolliert die BM Zuteilung dynamisch, um Verklemmungen zu vermeiden (z.B. via Bed. 3 und 4).
4. **Statische Vermeidung**
   Man baut das System von vornherein so, daß eine der Bed. 1-4 gar nie eintreten kann (z.B. via 2 und 4).

4.11.3 Modellieren von Verklemmungen

Verklemmungssituationen können durch BM-Graphen modelliert werden (*resource graphs*).

Gibt es jedes BM nur einmal, so kann man den BM-Graphen bilden und algorithmisch auf Zyklen untersuchen (Bed. 4).

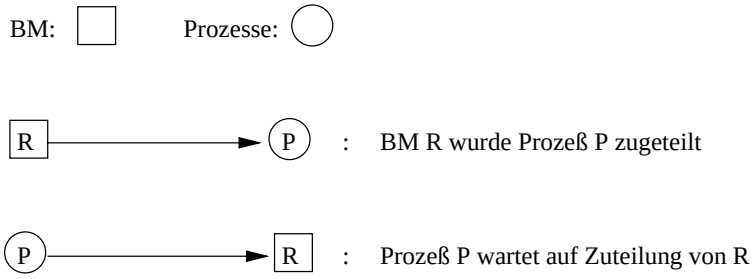


Abbildung 4.6: Modellierung von Verklemmungssituationen

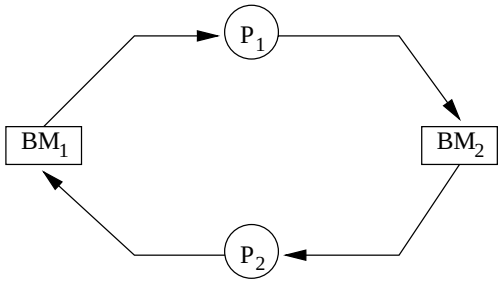


Abbildung 4.7: Einfachster Deadlock

4.11.4 Wiederaufsetzen nach Verklemmungen

Hierzu nutzt man Bed. 3 aus (no preemption). Man nimmt einem Prozeß einige seiner BM weg, falls möglich, oder man terminiert den ganzen Prozeß. Systeme mit Seiteneffekten (Datenbanksysteme) müssen von vorneherein so entworfen werden, daß *Transaktionen* zurückgesetzt werden können (mit Freigabe der BM), falls eine Verklemmung eintritt.

4.11.5 Vermeiden von Verklemmungen / Banker's Algorithm

Der folgende Algorithmus entdeckt Situationen, die zu Verklemmungen führen werden in Systemen, wo Betriebsmittel mehrfach vorhanden sind (z.B. 2 CDROM). Voraussetzung ist, daß man in jedem Systemzustand weiß, wieviele BM zur Beendigung jedes Prozesses maximal benötigt werden.

Ein Systemzustand besteht aus

E	$=$	$(E_1, \ldots, E_n)$	Vektor der existierenden BM
A	$=$	$(A_1, \ldots, A_n)$	Vektor der vorhandenen (noch freien) BM
Für jeden Prozeß i:			
C_i	$=$	$(C_{i_1}, \ldots, C_{i_n})$	Vektor der bereits zugeteilten BM
R_i	$=$	$(R_{i_1}, \ldots, R_{i_n})$	Vektor der noch benötigten BM

Ein Zustand, der (evtl. zukünftig) zu einer Verklemmung führen wird, heißt *unsicher*; ein Zustand, der bei geeignetem scheduling nicht zu Verklemmungen führen wird, heißt *sicher*.

Der Bankiers-Algorithmus von Dijkstra gewährt BM Anforderungen dann, wenn sie in einen sicheren Zustand münden. D.h. man nimmt an, die Anforderung werde gewährt und setzt die zugehörigen Vektoren C_i, R_i auf. Dann prüft man gemäß dem folgenden Verfahren, ob der Zustand sicher ist.

Bestimmung von sicheren Zuständen

Für alle lauffähigen Prozesse simuliert man die Auswirkungen ihres Ablaufs auf den BM Vektor A. Bleiben zum Schluß nicht lauffähige Prozesse übrig, so sind diese verklemmt.

1. Suche einen Prozeß P_i mit $R_i \leq A$ (komponentenweise $\leq$).
2. Falls es keinen solchen Prozeß gibt, terminiere. Die übrigbleibenden Prozesse sind verklemmt; sind keine übrig, ist der Zustand sicher.
3. Lasse den gefundenen Prozeß logisch ablaufen: $A := A + C_i$; entferne C_i und R_i, entferne den Prozeß.
4. Go to 1 $\Box$.

Bem.: Die Reihenfolge in 1. ist egal, da nach jedem Schritt A anwächst.

Der Bankiers-Algorithmus berücksichtigt nicht, daß

- Ressourcen auch zurückgegeben werden können, bevor ein Prozeß terminiert. (Nicht jeder unsichere Zustand muß daher zu einer Verklemmung führen.)
- Die maximal gebrauchten BM nicht immer von vornherein bekannt sind.
- Neue Prozesse entstehen können durch `fork` oder durch Benutzer.

Kapitel 5

Prozeßablaufplanung (process scheduling)

Dies ist die Frage, welcher Prozeß wann wie lange läuft. Grundprinzip ist die Existenz einer Warteschlange (*run-queue*) von Prozeßleitblöcken lauffähiger Prozesse. Der Ablaufplaner (*scheduler*) kettet lauffähige Prozesse dort ein und wählt bei Freiwerden eines Prozessors nach einer gewissen Strategie den nächsten zu bearbeitenden Prozeß daraus aus. Je nach Strategie und Komplexität der Warteschlange werden verschiedene Scheduler unterschieden.

Ein Prozeß befindet sich immer in einem von drei Grundzuständen: *running* (auf einer CPU), *runnable* (ready; kann bei Freiwerden einer CPU sofort laufen. Alle Betriebsmittel bereit) und *blockiert* (wartet auf ein Betriebsmittel).

Diese Zustände ändern sich gemäß dem *Prozeßzustandsdiagramm*.

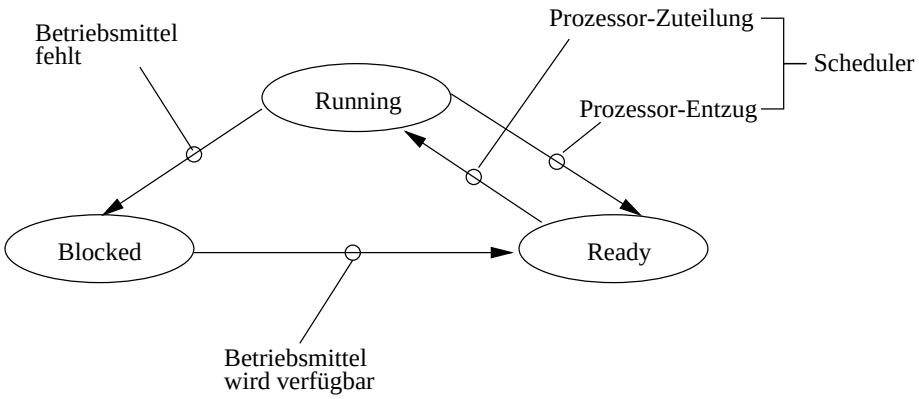


Abbildung 5.1: Kern des Prozeßzustandsdiagramms

Verfeinerungen sind möglich, z.B. Unterscheidung zwischen blocked on disk i/o, blocked on tape i/o, blocked on terminal i/o, blocked waiting on signal, ... und je nachdem Auslagerung auf disk oder nicht.

Ein *laufender* Prozeß belegt eine CPU, bis er entweder sein Zeitquantum verbraucht hat oder er einen Systemaufruf macht, der dazu führt, daß er blockiert. Das Zeitquantum wird vom BS (in einem HW Register) gesetzt, bevor der Prozeß gestartet wurde. Die HW zählt das Register herunter parallel zur Verarbeitung des Prozesses; wenn das Register = 0 ist, findet ein HW interrupt statt, und es wird (in HW) zu dessen Service-Routine umgeschaltet, in diesem Fall dem scheduler, der die Prozeßumschaltung vornimmt. Im Falle eines Systemaufrufs kommt es zur Blockade und damit zur Prozeßumschaltung, wenn ein Betriebsmittel angefordert wurde, das nicht sofort vorhanden ist (Speicherzeile, file, ...), etwa auch durch down(semaphore) als Systemaufruf.

Bei der Prozeßumschaltung werden die Register (auch Coprozessorregister!) in den Leitblock oder in den Stack gesichert und dann vom neuen Prozeßleitblock ein neuer Registersatz geladen. Ist zuletzt der PC geladen, ist die Umschaltung vollzogen.

Beispiel: `ctreads_reschedule`!

Beispiel: `_cproc_switch` aus C Threads

5.1 Ziel einer Prozeßlaufplanung

1. Fairneß: Jeder Prozeß bekommt auf faire Weise Rechenzeit zugeteilt.
2. Effizienz: Die CPU soll nie leer laufen.
3. Antwortzeiten: sollen für interaktive Benutzer minimal sein.
4. Prozeßabwicklungszeit (*turnaround time*): soll für Benutzer des Stapelbetriebs minimal sein.
5. Durchsatz: Der Gesamtdurchsatz von Prozessen soll maximal sein.
6. Der Planungsaufwand selbst soll minimal sein.

Einige dieser Ziele widersprechen sich; außerdem ist die Planung selbst zeitaufwendig, besonders wenn sie optimal sein soll. So sind vor allem gute Kompromisse gefragt. Theoretische Methoden vernachlässigen oft (6.) bis zur Impraktikabilität.

Grundsätzliche Unterscheidung:

- *Non-preemptive scheduling*: Es werden keine künstlichen Zeitquanten gesetzt.
- *Preemptive scheduling*: Laufende (und weiter lauffähige) Prozesse können unterbrochen und durch einen anderen ersetzt werden (*preemption*).

Im Normalfall nicht-kooperierender Benutzer muß preemptive scheduling eingesetzt werden.

5.2 Karussell Laufplanung (round robin scheduling)

Es wird stets der 1. Prozeß der WS ausgeführt. Bei Umschaltung kommt der Prozeß als letzter in die Warteschlange. Diese Methode wird oft angewendet. Problem ist die Länge des Zeitquantums Q. UNIX Prozeßwechsel dauert ca. 5 ms@1MIPS. Falls Q = 5 ms, haben wir 100% overhead. Falls Q = 500 ms haben wir 1 % overhead, aber der 11. Prozeß in der WS muß 5,05 s warten, bis er wieder die CPU bekommt. Ein gebräuchlicher Wert ist 100 ms.

5.3 Laufplanung mit Prioritäten (priority scheduling)

Prioritäten können statisch zugeteilt werden (Prof., Student, ...), benutzergesteuert (`nice`) oder dynamisch aus dem Laufzeitverhalten errechnet werden. (Vergleiche die Lösung bei UNIX.)

Interaktive Prozesse können bevorzugt werden, um kurze Antwortzeiten zu gewährleisten und weil sie sowieso den Prozessor schnell wieder frei machen. Z.B. kann die Priorität zu 1/f gesetzt werden, wobei $f = \frac{r}{Q}$, Laufzeit/Zeit-Quantum.

5.4 Mehrfach-Warteschlangen

Oft gibt es für jede Priorität eine eigene run-queue, die selbst in round robin Weise verwaltet wird.

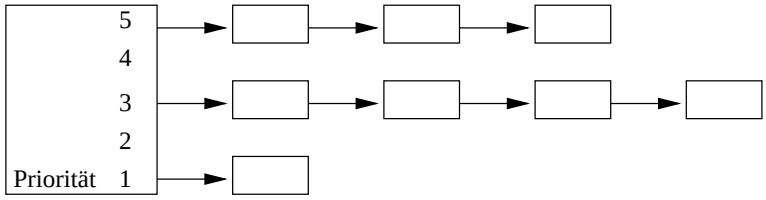


Abbildung 5.2: Mehrfachwarteschlangen

5.5 Kürzester Auftrag zuerst

Falls gleichzeitig k Aufträge eintreffen, deren Bearbeitungszeit im voraus bekannt ist, kann die mittlere Wartezeit auf die einzelnen Resultate minimiert werden, indem man immer den kürzesten Auftrag zuerst bearbeitet. Denn die mittlere Wartezeit ist

$$\begin{aligned} & [a_1 + (a_1 + a_2) + (a_1 + a_2 + a_3) + \ldots] / k \\ = \quad & [k \cdot a_1 + (k-1)a_2 + \ldots + a_k] / k =: m_k \end{aligned}$$

Gilt nun $a_i > a_{i+1}$ bei einem m_k, so erhalten wir $(j = i + 1)$

$$\begin{aligned}(k-(i-1))a_i + (k+1-(i+1))\,a_j = \\ (k+1-i)(a_i+a_j) - a_j > (k+1-i)(a_i+a_j) - a_i\end{aligned}$$

Gilt nun $a_i > a_j$ bei $j = i + d$, so erhalten wir

$$\begin{aligned}&(k-(i-1))a_i + (k-(j-1))a_j \\ &= (k+1-i)a_i + (k+1-j)a_j \\ &= (k+1)(a_i+a_j) - i \cdot a_i - (i+d)a_j \\ &= (k+1)(a_i+a_j) - i(a_i+a_j) - da_j\end{aligned}$$

d.h. durch Vertauschen von a_i und a_j erhält man $m'_k < m_k$.

Die Auftragslänge kann z.B. in monolithischen Transaktionsverarbeitungssystemen gut geschätzt werden.

Oft kann aber nur geschätzt werden, z.B. aus dem Verhalten der Vergangenheit. Der Schätzwert S muß dann immer wieder neu errechnet werden. Ein guter Kompromiß aus Stabilität und Flexibilität ist es, den neuen Wert S' zu gleichen Teilen aus dem alten und dem jüngsten Wert zu bestimmen $S' = \frac{S_{\text{alt}}+S_{\text{neu}}}{2}$. Damit *altert* der Einfluß früheren Verhaltens

$$\frac{\frac{\frac{S_0+S_1}{2}+S_2}{2} + S_3}{2} \ldots$$

Kapitel 6

Speicherverwaltung

6.1 Seitentausch-Algorithmen

Wenn die MMU einen Seitenfehler entdeckt (d.h. die angebotene virtuelle Adresse nicht umsetzen kann), erzeugt sie einen Interrupt, der den Prozeß unterbricht und in das BS verzweigt. Das BS muß den Grund für den Interrupt feststellen: Existiert die Speicherstelle gar nicht oder liegt sie auf einer Seite, die nicht im HSP eingelagert ist. Im ersten Fall bekommt der Prozess ein `SIGSEGV` Signal, im zweiten Fall muß das BS die Seite einlagern (lassen) und die Seitentabellen nachtragen. U.U. muß zuvor eine Seite ausgelagert werden, um Platz zu schaffen. Hierzu gibt es verschiedene Algorithmen.

6.2 Optimaler Seitentausch

Der optimale Algorithmus ersetzt diejenige Seite, die am spätesten referenziert werden wird. Dadurch wird das Intervall maximiert, in dem kein Seitentausch nötig ist. Diese Auswahl ist i.a. nicht konstruktiv möglich. Innerhalb eines einzelnen Programmes, dessen Laufzeitverhalten genau bekannt ist, geht dies aber (z.B. durch einen externen *Pager*). Alle Verfahren versuchen, dieses Ideal zu approximieren.

Da man nicht in die Zukunft blicken kann, schließt man aus der Vergangenheit auf die Zukunft. Man merkt sich also, ob eine Seite kürzlich benutzt wurde oder nicht.

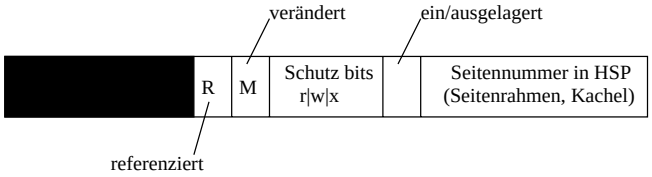


Abbildung 6.1: Eintrag in Seitentabelle

6.3 NRU - not recently used

Die R und M bits werden gesetzt, wenn die Seite referenziert bzw. verändert wurde. Periodisch wird das R Bit zurückgesetzt, um Seiten finden zu können, die „kürzlich“ verändert wurden. Die Bits RM geben den Seiten eine Priorität. NRU tauscht eine Seite niedrigster Wichtigkeit gegen die benötigte. Der Algorithmus hat wenig Aufwand, falls die Status bits von der HW gesetzt werden, und in der Praxis gutes Verhalten.

6.4 FIFO

Die am längsten im System befindliche Seite wird ersetzt. Geringer Aufwand, aber auch alte Seiten können wichtig sein. (Langlaufende Programme werden torpediert.)

6.5 2^{nd} chance

Modifikation von FIFO. Man sortiert die Seiten nach der Zeit als sie geladen wurden. Man wählt die älteste gemäß FIFO zum Ersetzen aus; dies tut man tatsächlich, falls R = 0. Falls R = 1, erhält die Seite eine zweite Chance, drin zu bleiben. Man setzt R = 0, ändert die Ladezeit zur momentanen Zeit und kettet die Seite ans Ende der Liste.

So wird die älteste Seite ersetzt, die R = 0 hat (nicht kürzlich benutzt wurde). Falls alle R = 1 haben, wird die älteste Seite ersetzt.

6.6 Clock

Um die Listenverwaltung in 2^{nd} chance zu optimieren, geht man mit einem Zeiger durch die (logisch zirkuläre) Liste aller Seiten.

Man ersetzt die Seite unter dem Zeiger, falls R = 0, sonst setzt man R = 0 und probiert die nächste Seite, bis eine mit R = 0 gefunden wurde.

6.7 LRU - least recently used

LRU tauscht diejenige Seite, die am längsten nicht benutzt wurde. Eine volle Implementierung ist sehr teuer, aber Approximationen sind möglich.

6.7.1 HW Implementierungen

1. Statt des R bits wird ein Zeitstempel im Seitenrahmen gespeichert, z.B. ein 64 Bit Zyklenzähler. Der Rahmen mit dem kleinsten Zähler wird getauscht. Das Durchsuchen *aller* Rahmen ist immer noch teuer (z.B. 128 MB HSP $\approx$ 128 K Rahmen = 128 K $*$ 64 bit = 2^{13} Kbits = 2^3 Mbit = 1 MB).

2. Bei n Rahmen sei eine Matrix aus $n \times n$ Bits gegeben. Wenn Rahmen k referenziert wird, setzt die HW Zeile k auf $11\ldots1$ und Spalte k auf $00\ldots0$. Die kleinste Zeile entspricht somit dem ältesten Rahmen. Probleme sind die Größe des Arrays und Kosten der HW. 128 MB HSP = 128 K Rahmen, benötigt 128 K $*$ 128 Kbit Matrix = $16 * 128$ MB = 2 GB.

Generelles Problem ist, daß SpezialHW die Portabilität negativ beeinflußt.

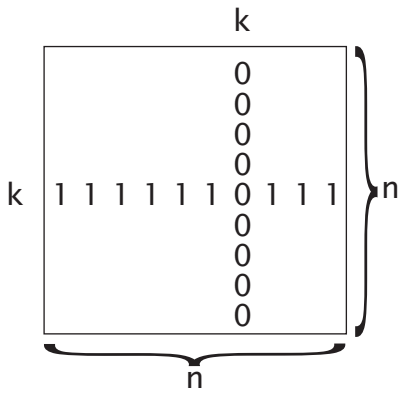


Abbildung 6.2: Referenzmatrix

6.7.2 SW Implementierung

1. Rahmen werden zu einer Liste verkettet. Referenzierte Rahmen kommen nach vorn. Der älteste Rahmen steht hinten.
   Problem: zu teuer (Umketten bei fast jeder Referenz).

2. Man wird nur noch nach jeder Zeitscheibe in SW tätig; die HW setzt das R-bit. *aging*: In jedem Rahmen speichert man einen Geschichtsvektor $\vec{v}$, der die R-bits der letzten k Zeitscheiben speichert. Das jüngste R-bit steht links. Es wird also $\vec{v_i} = (R_i << (k-1)) \mid (\vec{v_i} >> 1)$ in jeder Zeitscheibe ausgeführt.
   Die ältesten Rahmen haben das kleinste $\vec{v}$, aber alles älter als k Zeitscheiben wird gleich behandelt.
   In der Praxis ist $k = 8$ gut genug.
   Problem: Aufwand ist immer noch hoch bei großem Speicher.

6.8 Modellierung von Seitentauschverfahren

Das Verhalten eines Seitentauschsystems wird charakterisiert durch:

1. Die Referenzsequenz des Prozesses

2. Das Seitentauschverfahren

3. Die Anzahl m der Seitenrahmen

Die Referenzsequenz besteht aus der Sequenz der (virtuellen) Seitennummern, die von den Speicherzugriffen berührt werden.

LRU kann dann wie folgt modelliert werden.

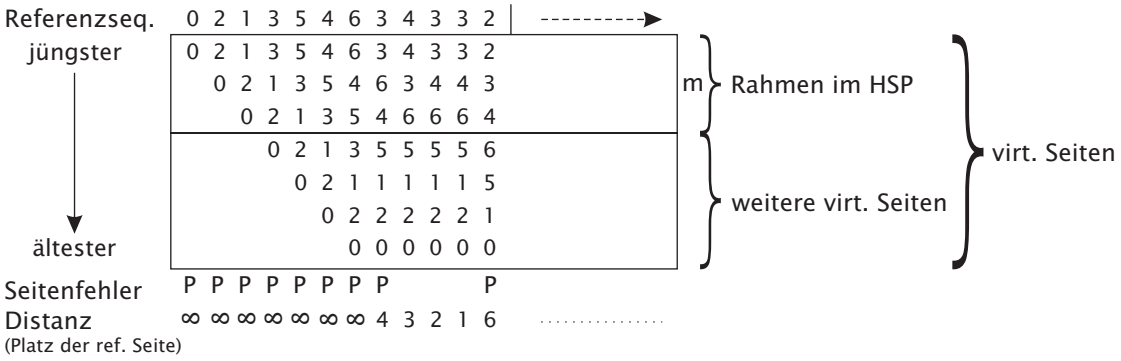


Abbildung 6.3: Modellierung von LRU

Die Tabelle listet die Seitenrahmen mit der jeweils zugeordneten Seitennummer auf, sortiert vom jüngsten zum ältesten Rahmen (nach dem Zeitpunkt der letzten Referenzierung). Zu jedem Zeitpunkt definiert eine Spalte der Tabelle also einen Stapel aus Seiten. Die Zeile „Distanz“ gibt an, welche Position die referenzierte Seite auf dem Stapel einnimmt.

6.9 Das Lokalitätsprinzip

Die Häufigkeit (Dichte) der Distanzen ist oft wie in Abbildung 6.4.

D.h. die Referenzen haben eine gewisse *Lokalität*. Aus der Distanzsequenz kann man nun für verschiedene m die Anzahl der Seitenfehler berechnen. Diese Funktion sieht oft wie Abbildung 6.5 aus.

Es gilt also, das Optimum zu finden, wo man den größten Teil der Seitenfehler vermeidet, ohne allzu viele Seiten zu verbrauchen.

Peter Denning nennt die momentan (im letzten Intervall) gebrauchten Seiten *working set* (Arbeitsmenge). Er stellt fest, daß viele Programme *Lokalität* der Referenz aufweisen, d.h. lange bei einer Arbeitsmenge bleiben, bevor sie zur nächsten übergehen. Diese Prozesse laufen schon gut, wenn sie genügend Seiten für das working set haben. Seitenfehler treten beim Wechsel der Arbeitsmenge auf. Ist die Seitenzahl im HSP zu klein, so kommt es zu exzessivem Seitentauschen (thrashing, „dreschen“). Dann ist es besser, den ganzen Prozeß auszulagern (swappen).

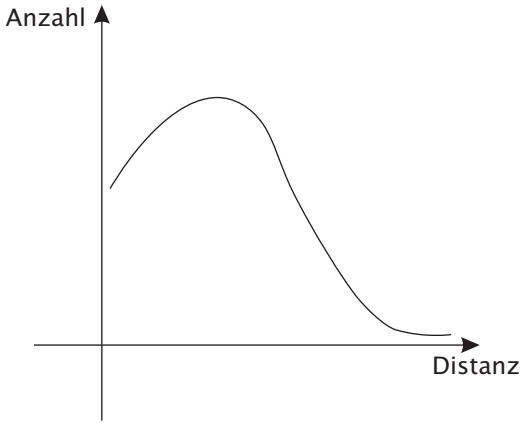


Abbildung 6.4: Häufigkeit der Distanzen

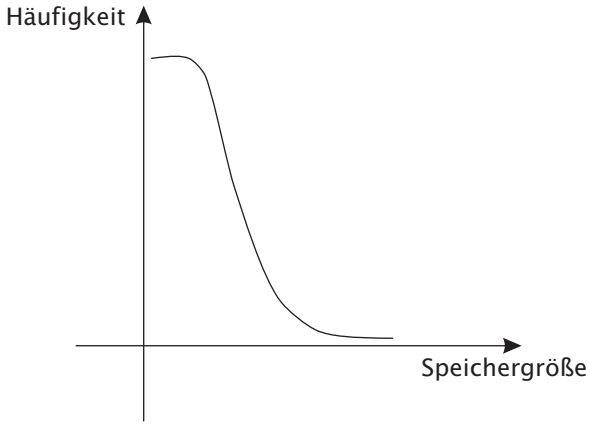


Abbildung 6.5: Seitenfehlerhäufigkeit in Abhängigkeit von Speichergröße

Am Anfang muß sich ein Prozeß die Arbeitsmenge aufbauen. Bei *demand paging* geschieht dies über Seitenfehler. *Prepaging* versucht dafür zu sorgen, daß ein Prozeß sofort eine Arbeitsmenge bekommt (z.B. nach Wiedereinlagern). Der Blocktransfer der Arbeitsmenge ist effizienter als demand paging.

Übrigens ist auch der Block-Transfer HSP → Platte des page-daemons effizienter als einzelnes Auslagern.

6.10 Implementierung des Seitentausches

Anforderung an die HW:

1. MMU aus Effizienzgründen.
2. Unterbrechbare und wiederaufsetzbare Instruktionen.

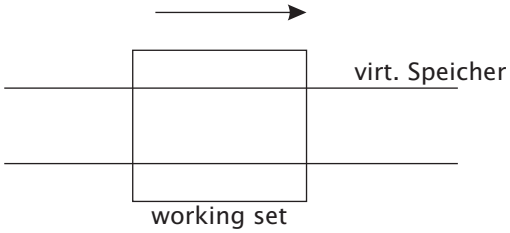


Abbildung 6.6: Working Set und virt. Speicher

Zu 1. Adreßübersetzung und validity bit muß von der HW kommen. M und R bits können in SW gesetzt werden, aber besser in HW.
Um das R bit in SW zu setzen, wird die Seite als nicht vorhanden ($V = 0$) gesetzt. Man merkt sich anderswo, daß die Seite in Wahrheit doch im HSP ist. Meldet die MMU die ungültige Adresse, stellt man in SW fest, daß sie doch gültig ist. Man setzt R = 1, V = 1 und wiederholt die Instruktion.

Zu 2. Bei $a = b * c$ können drei Seitenfehler auftreten. Nach der Fehlerbehandlung muß die Instruktion neu gestartet werden und alle Seiteneffekte müssen zurückgesetzt werden (vgl. $*p++$).

6.11 Schritte des Seitentausches

0. MMU bekommt virtuelle Adresse, die auf ungültiger Seite liegt ($V = 0$).
1. HW sichert Info zum Wiederaufsetzen der Instruktion. Kernel Trap.
2. Weiterer Kontextwechsel in SW (Sichern von Registern etc.); schließlich Aufruf der Seitenfehlerbehandlungsroutine PFH im BS.
3. PFH stellt die verursachende virtuelle Adresse und Seite fest (aus MMU oder indirekt aus Instruktion).
4. PFH stellt fest, ob Pseudofehler oder echter Fehler. Bei echtem Fehler `SIGSEGV` an Prozeß und neues Scheduling; exit.
Pseudofehler:

   a Seite nicht eingelagert.

   b Seite schreibgeschützt wegen copy-on-write.

   c Seite temporär schreibgeschützt durch pager.

   d Seite pseudo ungültig markiert, um $R = 1$ in SW zu setzen.

   c und d benötigen keinen neuen Rahmen und werden hier nicht betrachtet. Bei a und b wird ein leerer Rahmen auf der Freiliste gesucht. Ist keiner da, wird der pager gerufen, um Rahmen freizumachen.

5. Ist der ausgewählte Rahmen schmutzig, oder soll der pager viele Rahmen einsammeln, so wird der verursachende Prozeß suspendiert und ein anderer lauffähig gemacht. Der schmutzige Rahmen wird vom pager auf Platte zurückgeschrieben, möglichst zusammen mit anderen als Block Transfer. (Diese Rahmen werden geschützt, bis sie draußen sind.)

6. Ist ein freier Rahmen vorhanden, so findet das BS in der Tabelle virtueller Seiten des verursachenden Prozesses die Disk-Adresse der benötigten Seite und liest sie (per DMA) ein. Während der DMA kann ein anderer Prozeß laufen; der Verursacher bleibt suspendiert.

7. Ein HW interrupt signalisiert dem BS, daß die Seite da ist. Die Seitentabelle des Verursachers wird nachgeführt.

8. Der Verursacher wird in den Stand vor dem Fehler gebracht (Rücksetzen PC, Laden seiner Register).

9. Der Verursacher wird lauffähig gemacht und fährt mit der Ausführung fort.

Kapitel 7

I/O

7.1 Allgemeine Struktur I/O

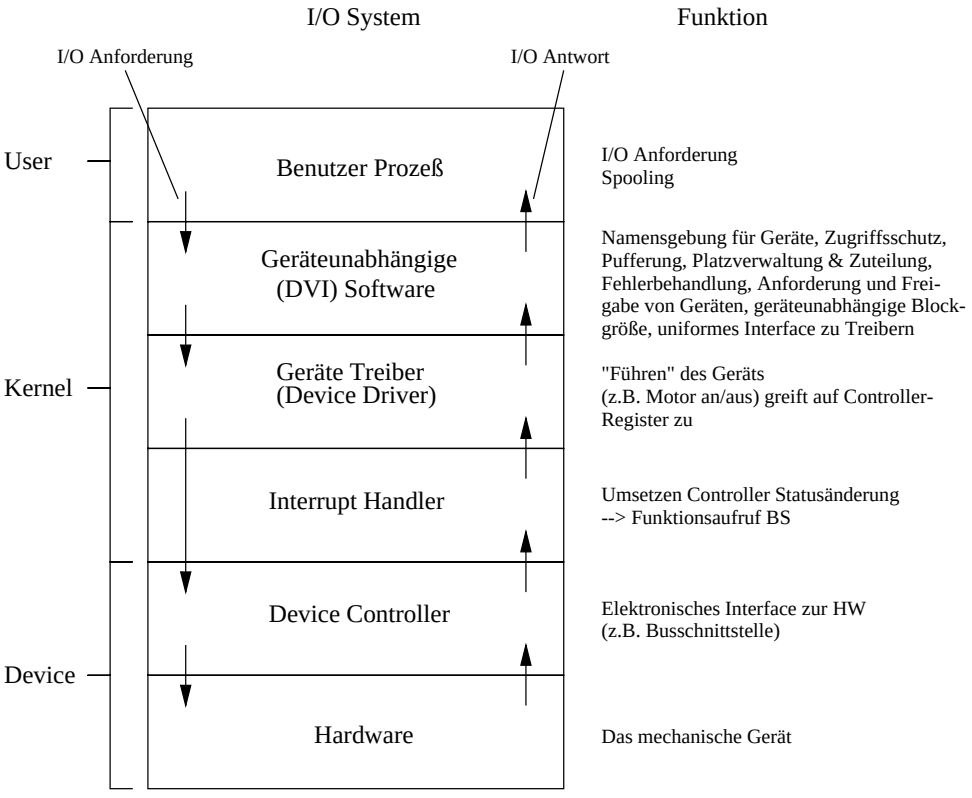


Abbildung 7.1: Grundstruktur der I/O

7.1.1 Geräte

Wichtige Unterscheidung: *Block Devices* erlauben wahlweisen Zugriff auf einen beliebigen Datenblock (`seek`). Typisch: Platten.

Character Devices generieren oder absorbieren einen Zeichenstrom.

7.1.2 Controller

Das elektronische Interface zum mechanischen Gerät. Z.B. Disk: Positionierbefehle werden als Bitmuster in die Register des Controllers geschrieben; der Controller generiert die elektronischen Ströme, die die Positionierung zur Folge haben. Zusatzinformationen werden aus Bitstrom herausgefiltert und die reinen Daten werden an den Speicher übertragen (DMA). Zwischenpufferung auch vom Controller.

7.1.3 DMA (Direct Memory Access)

Controller erhält Speicheradresse und Auftrag, Daten dort abzulegen. Controller generiert Schreibbefehle mit Adressen und Daten auf dem Bus, unabhängig von CPU. Speicher gehorcht, als ob die CPU die Befehle generiert hätte.

7.1.4 Interrupt Handlers

Die Routine, die gestartet wird, wenn ein HW interrupt erfolgt. Hier zum Deblockieren auf I/O wartender Prozesse nach dem Interrupt, den der Controller bei I/O Ende erzeugt.

7.1.5 Device Drivers (Gerätetreiber)

Geräteabhängige Software, die abstrakte, geräteunabhängige Befehle umsetzt in konkrete, vom Controller ausführbare Aktionen. Driver kennt die Adressen der Controller-Register und den Befehlsvorrat sowie die Geräteeigenheiten. (z.B. Zeilenlänge, Anzahl Zylinder etc.) Außerdem werden verschiedene Aufträge koordiniert. Disk Controller setzen logische Blocknummern in physikalische Adressen um mit Zylinder, Spur und Sektor. Fehlerbehandlung, z.B. Wiederholung des Lesens.

7.1.6 Geräteunabhängige Software

Stellt möglichst geräteunabhängiges Interface dar zwischen Anwendersoftware und Gerätetreiber. Z.B.

- Umsetzung Dateinamen auf Geräte(treiber).
  `/dev/tty0` → (major device ♯, minor device ♯)
  = (Gerät, Ort im Gerät)
  gespeichert im i-node eines Spezialfiles.

- Puffern von Daten, wenn `char` gebraucht, aber Block gelesen werden muß.
- Verbergen der physikalischen Blockgrößen.
- Zugriffsschutz, z.B. via mode bits der Gerätedateien.

7.1.7 Platte

Platte = Zylinder, Spuren (Tracks), Sektoren (Sectors) (8-32)

Heute z.B. SCSI oder IDE, $<$ 10ms, 100 GB

Zugriffszeit = Positionierzeit des Arms (*seek time*), Rotationswartezeit (*latency*), Transferzeit.

Zugriffszeit kann im Mittel verkürzt werden durch geschicktes Einfädeln der Aufträge, sowie durch parallele Datenübertragung in RAID Systemen.

7.1.8 Arm Scheduling

7.1.8.1 FCFS - First Come First Served

Nicht optimal, da durch Bündelung nahegelegener Aufträge die Suchzeiten minimiert werden können (z.B. Lese Zylinder 1, 500, 2, 400, 3, ...).

7.1.8.2 SSF - shortest seek first

Nimmt den Auftrag als nächstes, der zur kürzesten Positionierzeit führt. Unfair, da Ausreißerauftrag u.U. nie behandelt wird.

7.1.8.3 Elevator (Fahrstuhl)

Kopf wird in der Liste der Zylinder Auf und Ab bewegt, jeweils max. lange in einer Richtung wie ein Fahrstuhl.
Variante: Liste der Zylinder wird logisch zum Ring geschlossen.

Ähnliches ist für Sektoren möglich, wenn Sektoradressen on-the-fly vom Driver gelesen werden können.

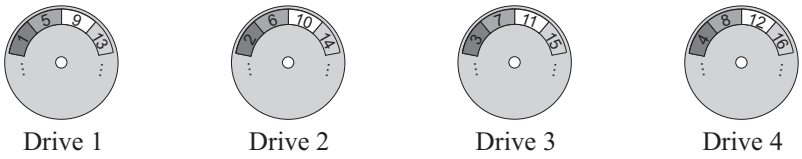
7.1.9 RAID

Die Datentransferrate ist durch die Rotationsgeschwindigkeit begrenzt. Ausweg ist Parallelismus.

RAID = Redundant Array of Inexpensive Disks.

Raid gibt es in mehreren Ausbaustufen. Z.B. 38 Disks parallel. Ein 32 bit Wort wird mit 6 bits zur Fehlerkorrektur zu 38 bit aufgeweitet. Jedes der bits wird auf einer Disk gespeichert. Fällt eine Disk aus, kann das 32 bit Wort aus 37 bits rekonstruiert werden (Fehlerkorrigierender Code). Dies ermöglicht auch den Austausch einer fehlerhaften Platte im laufenden Betrieb (hot swapping).
Es gibt verschiedene Typen (genannt „Level“).

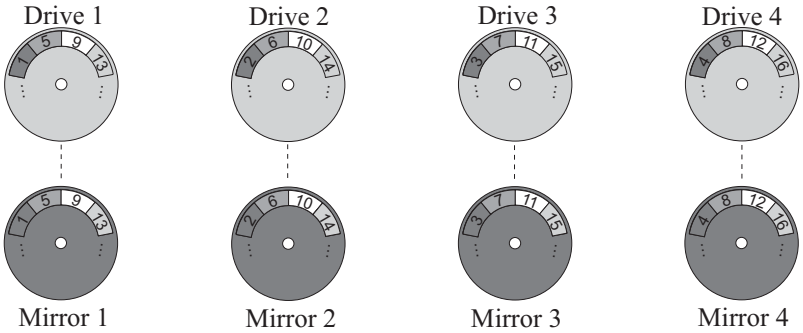
- Level 0 (Striping): Die Bytes eines Datenwortes werden auf 4 Disks verteilt.



  Vorteil: 4-fache Schreib-/Lesegeschwindigkeit bei langen Zugriffen (große Datenblöcke)

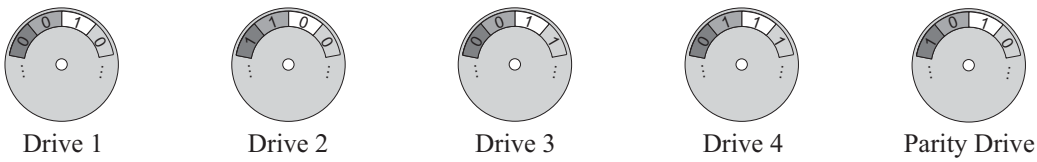
- Level 1 (Mirroring): Spiegeln der Platten zur Ausfallsicherheit. Lesen: 2-fache Geschwindigkeit, Schreiben: 1-fach.

- Level 0+1 (auch 10 genannt):



  Nachteil: Kosten

- Level 4: Striping mit Parity Bit auf extra Platte



Parity Byte auf der Parity Disk besteht aus den Parity Bits zu den Datenbits in Bytes 1-4. Also: Bit 1 des Partity Bytes ist das Parity Bit zu Bit 1 in Bytes 1-4.

Problem: Parity-Platte Flaschenhals beim Schreiben.

- Level 5: Wie Level 4, aber je nach Platten-Region Parity Bits auf anderen Platten.

Bemerkung: Konstruktion verteilter Systeme

1. Total verteilte Architektur:
   Auf jedem Rechner ein gleichberechtigter Teil des Systems, die miteinander kommunizieren. Erstrebenswert, aber kompliziert.
2. Eine zentrale Instanz mit Stellvertretern auf jedem Rechner. Einfacher, aber Gefahr von Performance-Bottleneck.

7.2 I/O in UNIX

E/A-Geräte werden logisch als spezielle Dateien behandelt, da sie geschrieben/gelesen werden können. (Üblicherweise in /dev). Sie können wie Dateien behandelt werden, z.B. cat file /dev/lp.
(Üblicherweise wird aber das Schreiben auf /dev/lp nur dem printer daemon gestattet).

Es werden 2 Dateitypen unterschieden

1. Block special files
   Datei ist Folge numerierter Blöcke.
   Jeder Block kann individuell adressiert und gelesen/geschrieben werden. (Bsp. Platte)
2. Character special files
   Datei ist Strom von Zeichen mit sequentiellem Zugriff. (Bsp. printer, Terminal, ...)

Die Gerätesteuerung selbst wird von Gerätetreibern erledigt (**device drivers**). Geräteeigenheiten werden unter Standard-Schnittstellen verborgen. Dies kann auch über UNIX hinausgehen z.B. SCSI.

Üblicherweise werden die Blöcke von Blockdateien in einem „Puffer-Hort“ (**buffer cache**) zwischengespeichert, so daß Lese/Schreibzugriffe zunächst in den cache gehen und nur periodisch der Block zurückgeschrieben wird.

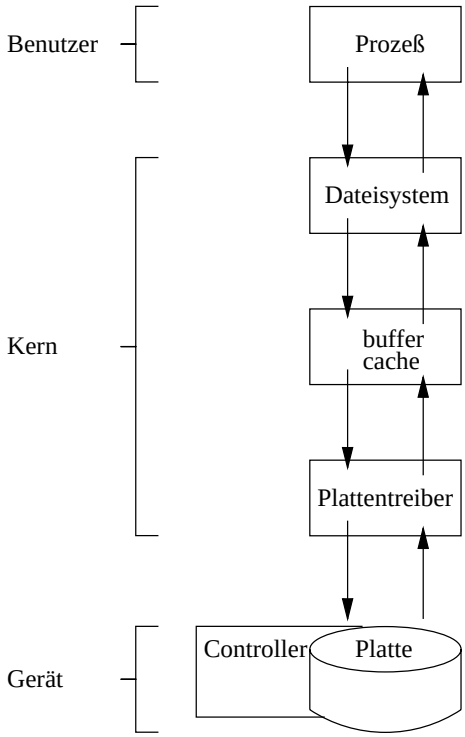


Abbildung 7.2: Platten-I/O in UNIX

Rohe Zeichenströme von zeichensequentiellen Dateien werden zunächst in „C-Listen“ zusammengefaßt (Liste aus 64 char Blöcken) und dann von einem Filter namens „line discipline“ *zubereitet*: carriage return wird durch line feed ersetzt, gelöschte Zeichen und Zeilen werden entfernt (d.h. B BKSP C wird durch C ersetzt) etc. Prozesse arbeiten üblicherweise mit gekochten Zeichenströmen, können wahlweise aber auch die Verbindung auf „*roh*“ setzen.

Für die Ausgabe gilt entsprechendes. Gekochte Ströme enthalten z.B. zusätzliche Füllzeichen, um die Wagenrücklaufzeit eines Typenraddruckers zu überbrücken.

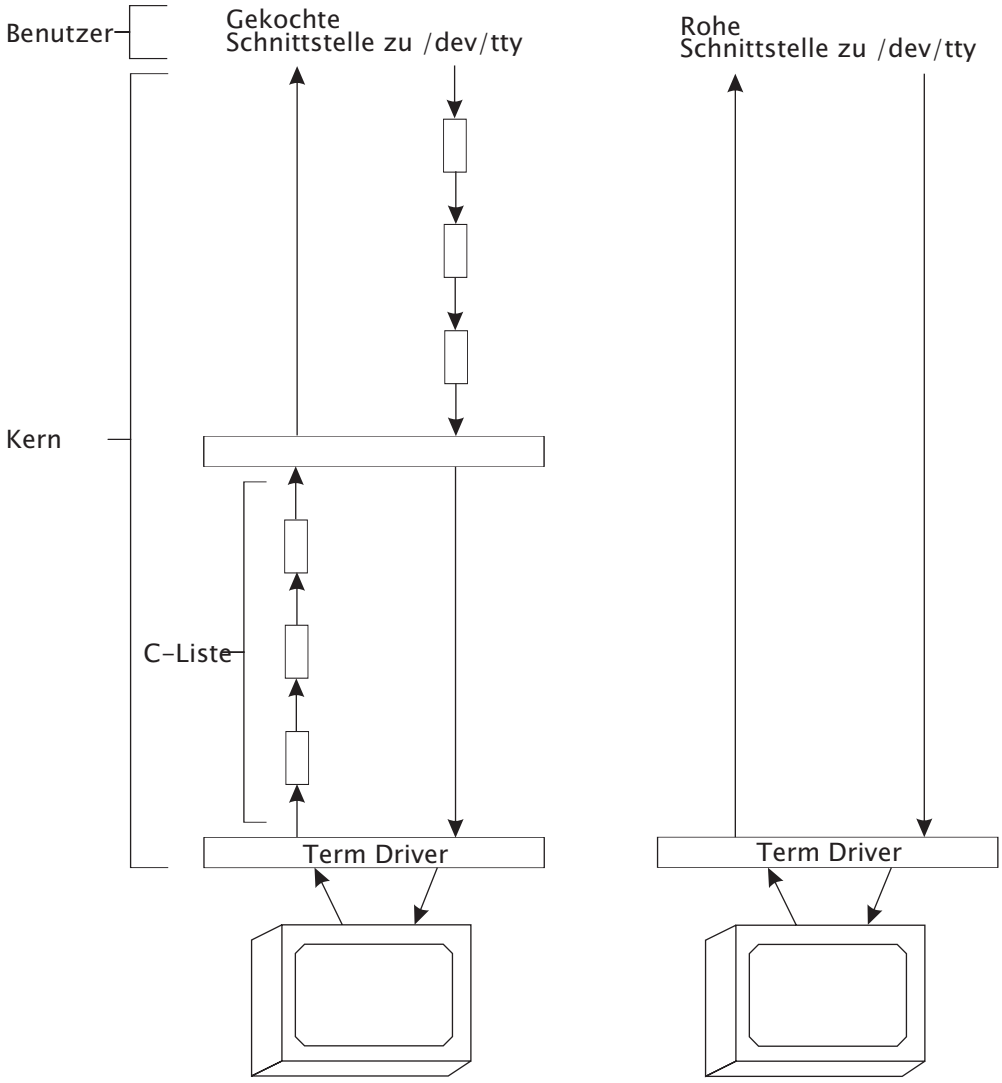


Abbildung 7.3: Terminal I/O in UNIX

7.2.1 E/A Ströme (System V Streams)

E/A Ströme (**streams**) wurden von Ritchie entworfen und in System V eingeführt, um mehr Modularität in der UNIX E/A zu erhalten. (Auf Programmiersprachen-Ebene erzielt die C++ -

streams-Bibliothek etwas ähnliches.) Z.B. bestehen Netzwerk-Treiber (**drivers**) aus einem Protokollteil und einem Netzwerk-spezifischen Teil. Der Protokollteil kann für verschiedene Netzwerke gleich sein, wurde aber bei konventionellen Treibern jeweils dupliziert.

Ein (System V) Strom ist eine Voll-Duplex-Verbindung zwischen einem Prozeß und einem Gerätetreiber. Er besteht aus einer Kette von Strom-Gliedern. Ein Strom-Glied besteht aus einem Paar von Manipulator-Einheiten. Eine Manipulator-Einheit besteht aus einer Service-Routine (Filter, Manipulator) wie z.B. eine line-discipline, ein Protokoll oder ein Treiber, sowie einer Warteschlange. Die Manipulator-Einheit bearbeitet die Nachrichten in der Warteschlange mit der Service-Routine und reicht die Nachricht in der Kette weiter.

Jede Manipulator-Einheit ist also ein Objekt einer Klasse mit folgenden Elementen (**members**):

- Initialisator, `open`-Prozedur vom Systemaufruf `open` angesprochen.
- Aufräumer, `close`-Prozedur, vom Systemaufruf `close` angesprochen.
- `put`-Prozedur, fügt eine Nachricht in die Warteschlange ein.
- `service`-Prozedur, bearbeitet die Nachrichten in dieser Einheit.
- Ein Zeiger auf die nächste Manipulator-Einheit.
- Die Warteschlange der zu bearbeitenden Nachrichten.
- Ein Zeiger auf eine private Datenstruktur mit dem Status der Einheit, z.B. `flags` und Hoch- und Tiefpunktmarken für Flußkontrolle etc.

Jede Nachricht kann aus mehreren Blöcken bestehen. Eine Warteschlange aus Nachrichten ist also eine zweidimensionale Liste.

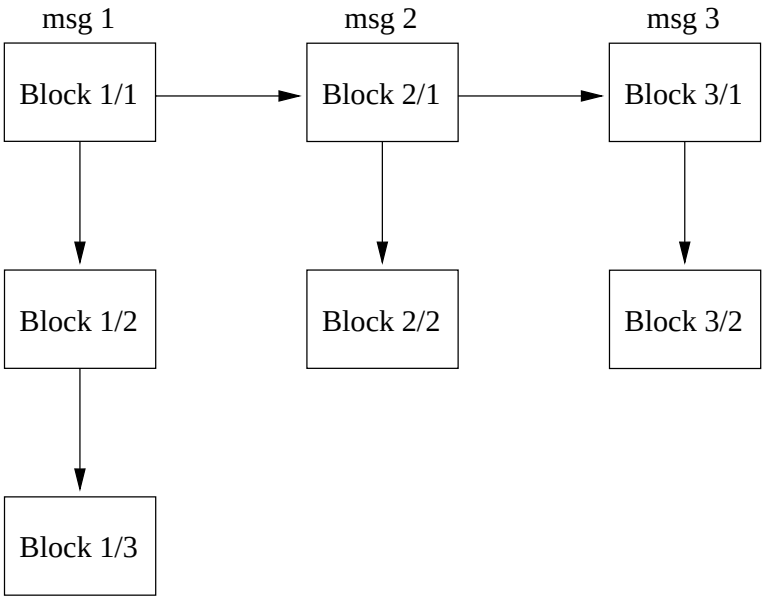
Jedes Gerät mit einem Streams-Treiber ist ein **character device**. Wird eine Geräte-Datei eines solchen Gerätes geöffnet, so wird im I-node nachgesehen, ob der zugehörige Treiber einen Strom benutzt. Falls ja, so wird ein Strom zwischen I-node und Treiber initialisiert. Alle Ströme beginnen mit einem uniformen Kopf mit generischen `put` und `service` Routinen. Der Kopf stellt z.B. Nachrichten-Blöcke zur Verfügung.

Ein Strom kann dadurch erweitert werden, daß ein Prozeß eine Manipulator-Einheit als neuen Modul direkt hinter dem Kopf des Stroms einsetzt – er „`push`t den Modul auf den Strom“. Dies geschieht mit dem Systemaufruf **ioctl**. Man könnte etwa wie folgt ein Terminal öffnen und eine **line-discipline** TTYLD in den Strom einfügen:

```
fd = open("/dev/ttznn",O_RDWR);
ioctl(fd,PUSH,TTYLD);
```

Durch `ioctl(fd,POP,0)` können die Module in LIFO-Manier wieder entfernt werden.

Jeder Modul benutzt die put-Prozedur des nächsten Moduls, um dort Nachrichten einzufügen. Die zugehörige service-Routine wird dann von einem Strom-Scheduler aufgerufen, falls die Nachrichtenschlange nicht leer ist.



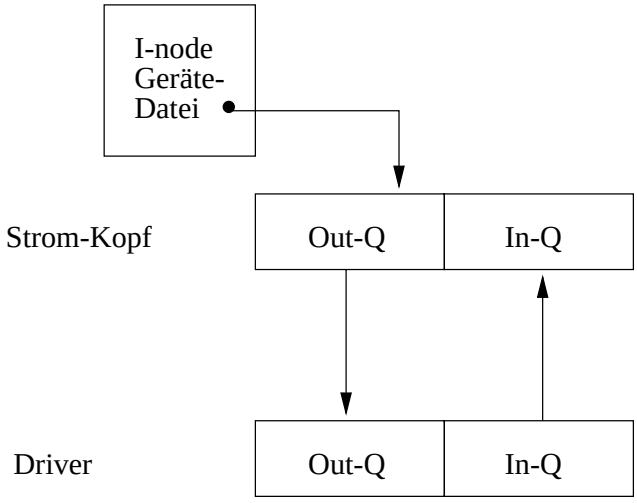


Abbildung 7.4: Strom nach Öffnen der Geräte-Datei

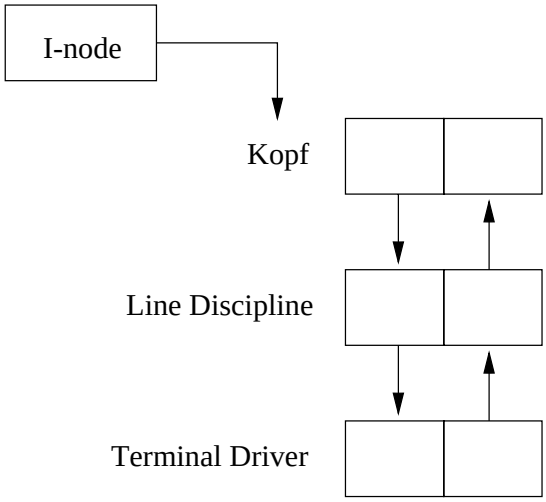


Abbildung 7.5: Strom nach Einfügen einer line discipline

Kapitel 8

MACH

Das Mach Betriebssystem wird an der Carnegie Mellon University entwickelt. Eine kommerzielle Version ist OSF/1 der Open Software Foundation (IBM, DEC, HP ...).

Es ist ein Mikrokernsystem. Ein kleiner Mikrokern bildet die Basis, auf der viele verschiedene Betriebssystemschnittstellen (z.B. UNIX) implementiert werden können. Der Kern ist strukturiert nach objekt-orientierten Prinzipien, d.h. in Gruppen von Datenstrukturen und Prozessen, die über Nachrichtenaustausch kommunizieren, u.U. auch über ein Netz.

Mach 2.5 hatte noch UNIX code im Kern, in Mach 3.0 (OSF/1) ist der UNIX code im Benutzerraum.

Gegenwärtige Ziele sind:

- Implementierung weiterer Betriebssysteme auf der Basis des Mikrokerns (z.B. OS/2).
- Unterstützung großer dünn belegter Adreßräume.
- Unterstützung transparenter Zugriffe auf Netzwerkressourcen.
- Ausnutzung von Parallelität im System und in den Anwendungen (Threads, Netzwerk).
- Weiterentwickeln der Portabilität.

8.1 Der Mach Mikrokern

Ein Mikrokern leistet Prozeßverwaltung, Speicherverwaltung, Kommunikation und I/0 Dienste. Dateisysteme werden im Benutzerraum verwaltet.

Der Kern kennt fünf Hauptobjekte:

1. Prozesse
2. Threads

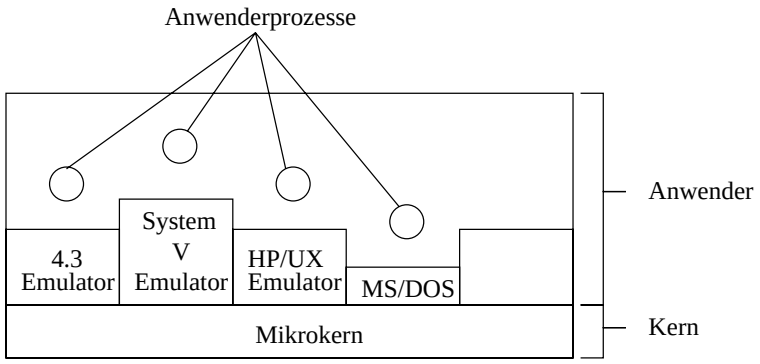


Abbildung 8.1: Struktur von MACH

3. Speicherobjekte
4. Ports
5. Nachrichten

Ein Prozeß (*task*) ist die übliche Verwaltungseinheit. In ihm können mehrere Threads ablaufen, die sich die task-Umgebung teilen. Ein üblicher UNIX Prozeß entspricht einer MACH task mit nur *einem* thread.

Ein Speicherobjekt besteht aus einer Menge von virtuellen Seiten, die in den Adreßraum eines Prozesses abgebildet werden können. Bei einem Seitenfehler kann ein Memorymanager des *Anwenders* durch Nachricht von Mach beauftragt werden, die Seiten einzulagern.

Ein *Port* ist ein von Mach verwalteter und geschützter Briefkasten.

8.1.1 Ports

Ein Port unterstützt einen ausfallsicheren, sequentiellen, unidirektionalen Nachrichtenstrom. Es hat immer nur ein thread pro port das Empfangsrecht, aber es können mehrere das Senderecht haben.

Es werden pro port max n Nachrichten gespeichert, danach blockiert jedes weitere `send`.

Ein Prozeß kann ports anlegen und Sende- und Empfangsrechte an andere Prozesse weitergeben (*capabilities*).

Nachrichten werden vom Kern weitergeleitet, wenn möglich durch Kopieren von Zeigern, in der Regel durch Kopieren der Nachricht. Der MACH interface generator (MIG) unterstützt das Vereinbaren von Nachrichtenformaten (und das Generieren des entsprechenden C Codes).

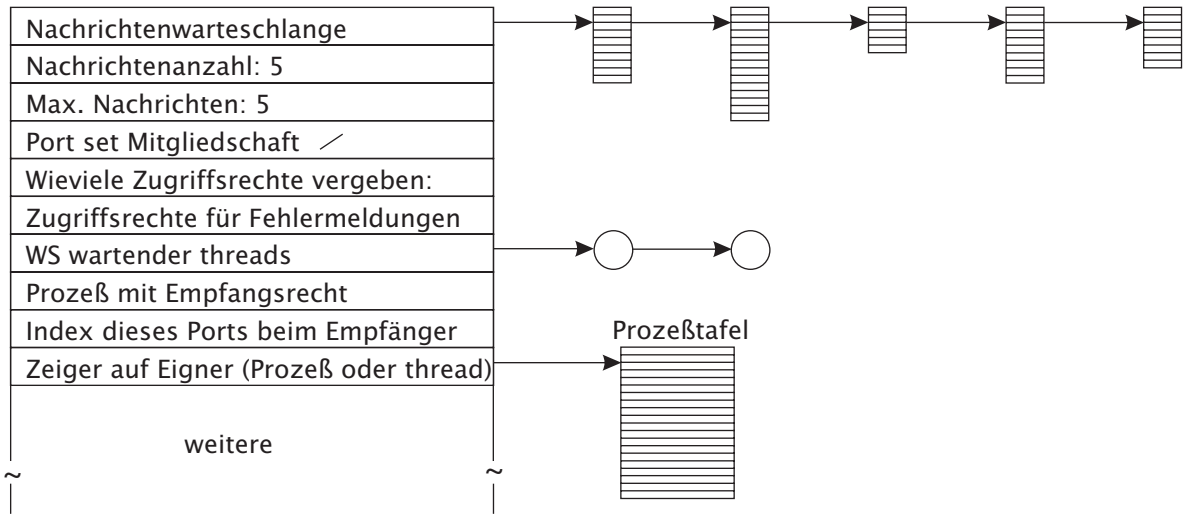


Abbildung 8.2: Port Struktur

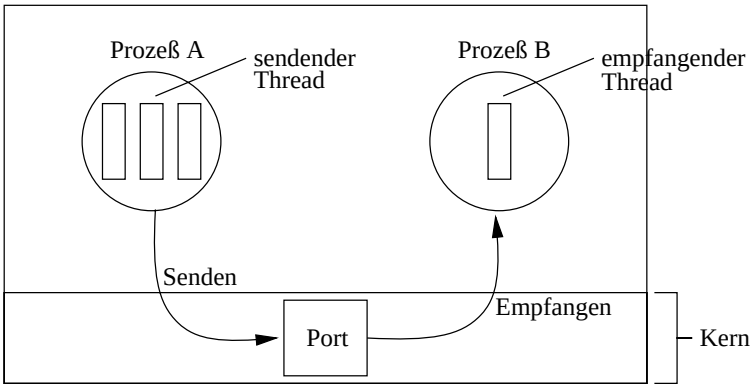


Abbildung 8.3: Nachrichtenübermittlung via port

Ein port hat u.a. folgende Felder (siehe Abbildung 8.2)

Ports können in Gruppen zusammengefaßt werden. Der Empfänger empfängt dann eine Nachricht von irgendeinem der Ports und blockiert nur, wenn die Gruppe leer ist.

Selbst die Kommunikation mit dem Kern und den Standard Servern läuft über (spezielle) ports. Das bedeutet insbesondere, daß server Dienste auch von anderen Rechnern erbracht werden können. Die Fähigkeit (*capability*) eines Prozesses, auf ports zuzugreifen, wird als Liste der ports mit jeweiligem Zugriffsrecht realisiert. Der Index in die Liste identifiziert einen port mit Zugriffsrecht analog einem filedescriptor. Rechte sind RECEIVE, SEND oder SEND_ONCE. SEND_ONCE erlaubt eine Einmal-Antwort z.B. von einem Server mit anschließender autom. Auflösung des Ports. Ein Prozeß mit RECEIVE-Recht auf P kann dieses Recht transferieren oder auch das SEND-Recht auf P erteilen.

Ein Eintrag in einer capability Liste kann sein

1. ein Zugriffsrecht auf einen Port.
2. Ein Zugriffsrecht auf eine Port-Menge.
3. Der Null-Eintrag. (Nach Deallokation der capability).
4. Ein Hinweis, daß der entsprechende Port dealloziert ist (weil der Empfänger gestorben ist), obwohl hier noch eine `send` capability steht.

Auf Benutzerebene ist eine capability der Index in die Liste.

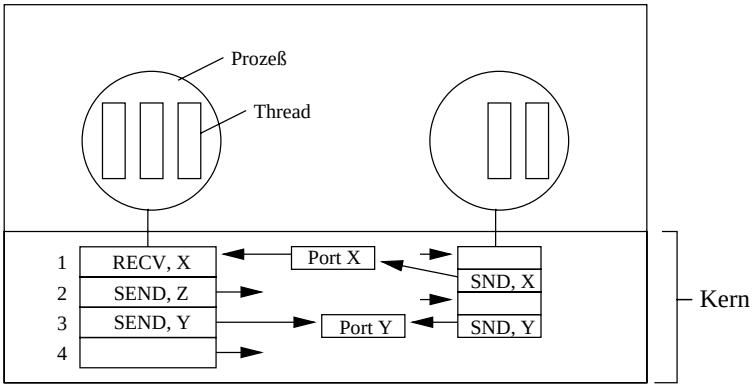


Abbildung 8.4: Ports und Capabilities

8.2 Nachrichten in MACH

Nachrichten werden mit dem `mach_msg()` system call gesendet oder empfangen.

```
mach_msg ( & hdr,                /* Pointer to msg buffer */
           options,              /* Send and/or receive */
           send_size,            /* How long is msg */
           rcv_size,             /* How big is buffer */
           rcv_port,             /* Capability identifying receive port */
           timeout,              /* ms to wait on call completion */
           notify_port )         /* Where to send status report */
```

Das Nachrichtenformat geht aus Abbildung 8.5 hervor.

simple: Message ohne capabilities oder out-of-line Daten.

local port: receive port (bei Empfang), reply port (bei Senden).

remote port: send port (bei Senden), - (bei Empfang).

complex: Nachricht mit capabilities (müssen geschützt werden) oder out-of-line Daten.

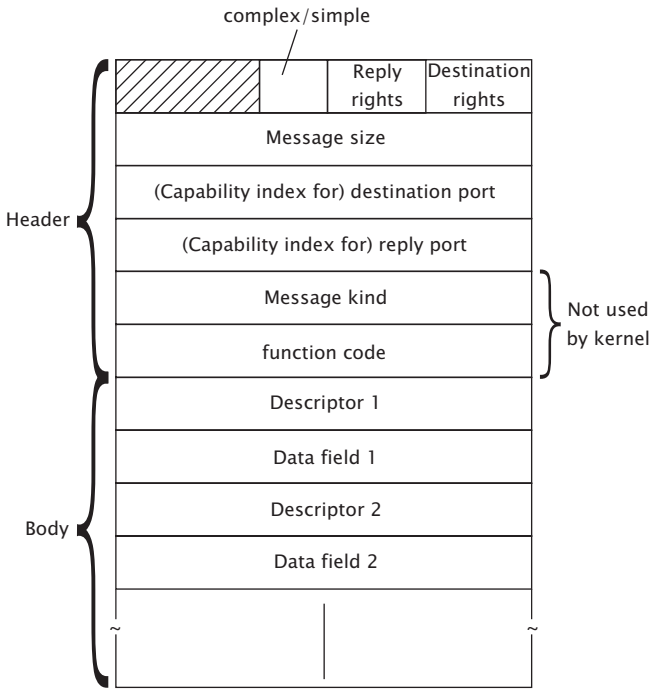


Abbildung 8.5: Nachrichtenformat (für komplexe Nachrichten)

message size: Größe von Kopf und Nachricht.

reply port: Wird z.B. bei RPC gebraucht.

Beim Senden muß die Nachricht in einem Puffer P zusammengestellt werden; sie wird dann mit mach_msg(&P, ...) abgeschickt. Beim Empfang wird der Puffer P angegeben und die Nachricht wird mit Header dort hineinkopiert. MIG kompiliert die entsprechenden Prozeduren für das Einpacken und Auspacken der Daten in eine Nachricht.

8.2.1 Das Nachrichtenformat

Mit der Deskriptor-Information (vgl. Abbildung 8.6) können Daten items zwischen Formaten umgesetzt werden (ASCII-EBCDIC, big endian - little endian).

Capabilities können übertragen werden mit oder ohne Löschung beim Sender (siehe Abbildung 8.7). RCV beim Sender kann zu SND beim Receiver umgesetzt werden. D.h. wer ein Receive Right hat, kann zu diesem Port ein send right vergeben.

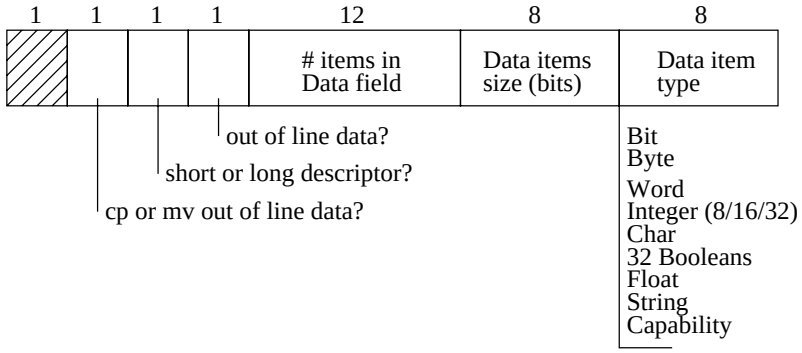


Abbildung 8.6: Deskriptor in einer komplexen Nachricht

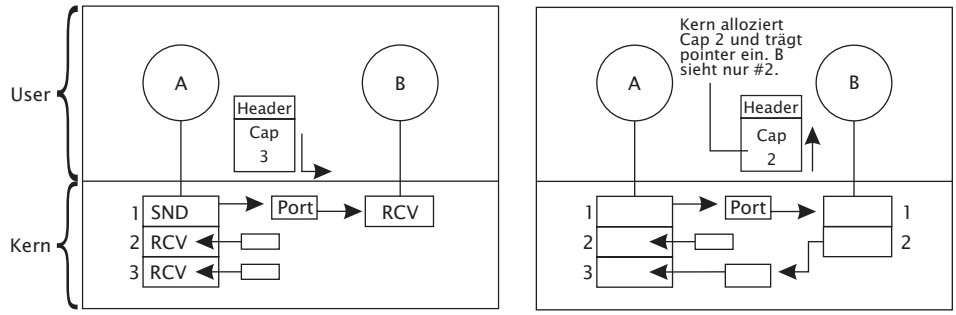


Abbildung 8.7: Senden einer Capability

8.2.2 out-of-line data

Zwischen Prozessen mit gemeinsamem HSP können Daten schneller übertragen werden, indem das zu sendende Objekt in den virtuellen Adreßraum des Empfängers abgebildet wird (copy-on-write). Dazu wird im Datenteil ein Zeiger auf den Beginn des Datenobjekts übertragen und in den *Size* und *Number* Feldern des Descriptors eine 20 bit breite Längenangabe. Im virtuellen Adreßraum des Empfängers wird dann ein freier Platz dieser Größe gesucht und die entsprechenden Seiten des Senders da hinein copy on write abgebildet. Der Kernel kann dies tun, da er in der Memory Map des Senders die betreffenden Seiten nachsetzen und eine entsprechende Map beim Receiver anlegen kann. Out-of-line data zwischen verschiedenen Kernen ist sowieso nicht möglich.

8.2.3 Der Netzwerk Nachrichten Dienst (NND)

Der NND leitet Nachrichten über Maschinengrenzen weiter. Nachrichten werden z.B. verschlüsselt, oder es finden Formatkonversionen statt.

Ein NND thread horcht die Eingangsportmenge ab und leitet Nachrichten an die entsprechenden Netzwerkports weiter.

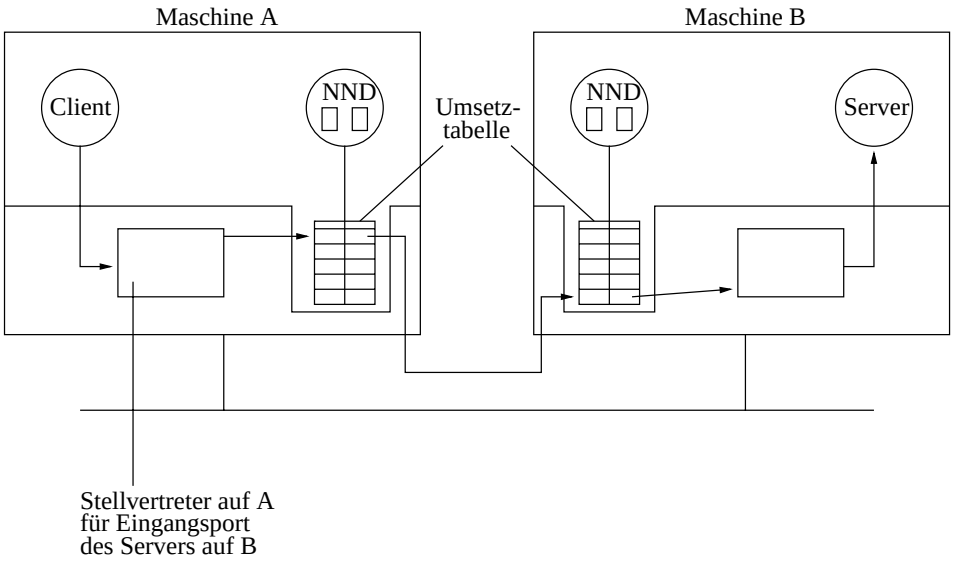


Abbildung 8.8: Nachrichtentransport über den Netzwerk-Nachrichtendienst

Der NND bemüht sich, die Methode lokaler Kommunikation in MACH möglichst homogen auf das Netz auszudehnen. Er benutzt spezielle *network ports*, um Nachrichten über das Netz zu senden. Diese macht er für die Anwenderprozesse transparent, indem er den Anwenderprozessen lokale MACH ports zur Kommunikation bereitstellt und diese auf Netzwerk-ports abbildet. Z.B. kann ein external pager auf einem file server abgesetzt laufen, wenn er einen lokalen Stellvertreter hat, der für ihn die Nachrichten entgegennimmt und weiterleitet.

Der NND läuft verteilt auf jeder Maschine. Ein Server kann sich beim NND anmelden mit einem lokalen RECEIVE port. Der NND koppelt daran einen Netzwerk port für Kommunikation von außen. Der NND erzeugt auf allen Maschinen je einen Stellvertreter (proxy) port und bildet diese auf den Netzwerk port ab. Der Netzwerk port hat eine Netz-Adresse, so daß das Netz Daten an ihn übertragen kann.

Die Zuordnung Name des Dienstes (server Prozeß) und Proxy Port kann vom name server verwaltet werden; ein Klient erhält so den proxy port, wenn er den Server Namen kennt. (Der Port des name servers ist ein registered port, der dem Klient vom Kern in die Wiege gelegt wird.)

Nachrichten von Klient zu Server werden in fünf Schritten übertragen (vgl. Abb. 2.24).

1. Lokale Nachricht an den Proxy Port.
2. Lokaler Empfang des lokalen NND am Proxy Port.
3. Abbildung auf das Netz (via Tabelle Proxy → Netzport) und Verschicken durch das Netz. Evtl. verschlüsseln, einkopieren von out-of-line data.
4. Empfang am Netzport durch remote NND. Abbildung auf lokalen port des Servers (Tabelle) und Abschicken.

5. Lokaler Empfang am Receive Port des Servers.

Falls Capabilities verschickt werden, muß dem verschickten lokalen Port ein Netzport zugeordnet werden und an der Zielmaschine ein Proxy Port.

8.3 Mach Speicherverwaltung

Mach unterstützt sehr großen, dünn besetzten virtuellen Speicher. Theoretisch tut das jedes VM System, das Problem ist aber, daß 32 bit Adressen 4 GB adressieren können auf z.B. 4.000.000 1K Seiten. Mach stellt *Regionen* zur Verfügung, bestimmt durch Basisadresse und Größe, in die Speicherobjekte abgebildet werden können (etwa ein File oder ein Satz Seiten oder ein Array). Adressen, die zwischen Regionen liegen, werden abgefangen und erzeugen einen trap, ohne daß Einträge in der Seitentabelle nötig sind.

Eine in den HSP projezierte Datei kann wie ein *char array* gelesen und geschrieben werden. Es herrscht die übliche Seiteneinteilung, die Daten können als Nachricht verschickt werden (etwa out-of-line) und nach Programmende wird alles zurückgeschrieben.

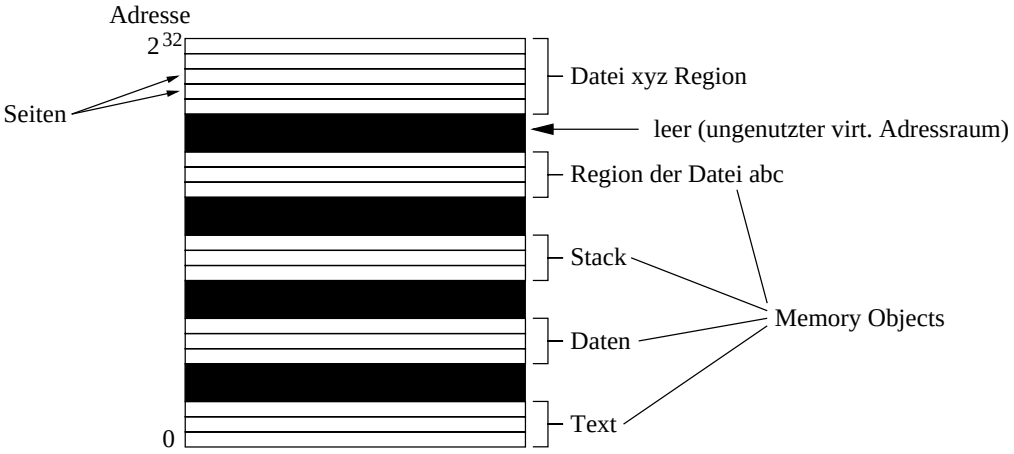


Abbildung 8.9: MACH Adreßraum (Beispiel)

8.3.1 Speichermehrfachbenutzung

Besonders in Multiprozessoren kann es vorkommen, daß mehrere Prozesse (gleichzeitig) dasselbe Speicherobjekt bearbeiten (etwa ein Bildraster).

Ein und dasselbe Speicherobjekt kann in verschiedene Adreßräume projiziert werden - man baut einfach die verschiedenen Seitentabellen so auf, daß verschiedene virtuelle Adressen immer zu denselben Seiten umgesetzt werden.

Dieses Verfahren kann auch bei `fork()` angewendet werden. Jede Region hat daher ein Vererbungsattribut mit einem von drei Werten

1 Nicht vererbt (im Kind ungenutzt)

2 Vererbt (von Vater und Kind gemeinsam genutzt)

3 Als Kopie vererbt.

3 wird als „copy on write “implementiert. Beim ersten Schreibzugriff des Vaters oder des Kindes auf eine als Kopie geerbte Seite wird ein Trap erzeugt und die Seite kopiert, die Seitentabelle repariert und erst dann auf die echte Kopie geschrieben. Dies könnte man auch mit „lazy copying “bezeichnen.

8.3.2 Verteilter mehrfachgenutzter Speicher

Ein zentraler Seitendienst kann z.B. die Seiten verteilen. Wird eine Seite zum Lesen angefordert, wird sie einfach ausgeteilt. Wird eine Seite zum Schreiben angefordert, werden zunächst Meldungen an alle anderen Benutzer versandt, die Benutzung einzustellen. Wenn dies zugesichert ist, wird die Seite zum Schreiben zugeteilt und am Ende wieder im Dienst zurückgeschrieben.

Dies ist nur *ein* Implementierungsbeispiel. Verschiedene Konzepte können leicht realisiert werden, da Mach das Schreiben externer Seitenbehandlungsdienste (*pager*) unterstützt. Der Kern fordert über einen bestimmten port eine Seite an, und der externe Dienst liefert sie über einen anderen port.

8.4 Prozesse in MACH

Primär besteht ein Prozeß aus einer Sammlung von Threads und den allen Threads gemeinsamen Betriebsmitteln. Das wichtigste ist der gemeinsame Adreßraum; außerdem gibt es `ports` zur Kommunikation mit dem Kern, Statistiken, die Adresse der Emulationsbibliothek, falls ein fremdes BS emuliert wird, und scheduling Informationen.

Fast alle Kommunikation mit dem Kern findet via Nachrichten und ports statt; es gibt nur wenige klassische Systemaufrufe (z.B. `mach_msg`).

Process port:	Hier liest der Kern Nachrichten, die Systemaufrufe ersetzen.
Bootstrap port:	Der erste Prozeß findet hier die Namen der essentiellen Kern-Ports.
Exception port:	Hier liest der Prozeß Fehlermeldungen wie Division durch Null.
Registered ports:	Standard ports, die die Standard Server beim Kern angemeldet haben (z.B. name server).
Thread ports:	Threads-spezifische Anweisungen an den Kern, z.B. exit, yield, ...

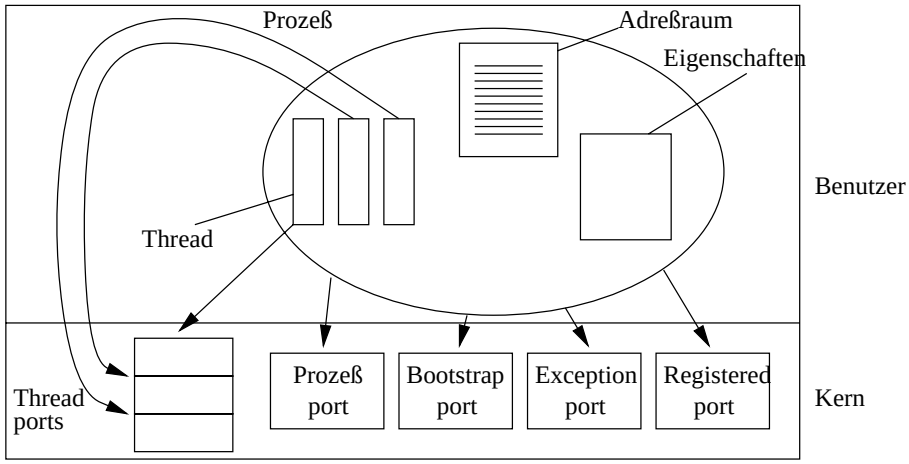


Abbildung 8.10: Prozesse in MACH

8.5 Systemaufrufe zur Prozeßverwaltung

Aufruf	Beschreibung
`Create`	Neuer Prozeß. Kann mit oder ohne Kopie des Adreßraums des Vaters erzeugt werden (für nachfolgendes `exec`). Kind bekommt process, bootstrap und exception port.
`Terminate`	
`Suspend`	Suspendiere Prozeß mittels P(suspend_counter).
`Resume`	Reaktiviere mittels V(suspend_counter). Prozeß lauffähig, falls suspend_counter = 0.
`Priority`	Setze Rahmenpriorität für alle Threads.
`Assign`	Teile den nachfolgend erzeugten Threads ein ausführendes Processor-set zu.
`Info`	Gib Laufzeitstatistik aus.
`Threads`	Liste aller Threads im Prozeß.

8.6 MACH Threads

Jeder Kernel thread hat einen Thread Port, um Thread spezifische BS Direktiven absetzen zu können.

Die Benutzerschnittstelle für Mach Threads ist C Threads. In Mach 2.5 (OSF/1) gibt es zu jedem C Thread einen Kernel thread. Daneben existiert eine Emulation von C Threads als Coroutinen (nicht- preemptive) auf einem einzigen Kernel thread. Dies ist für Testen/Debuggen geeignet, da die Ausführung deterministisch ist. Jeder Systemaufruf blockiert aber alle Threads. Mach 3.0 hat ein user-level threads Paket, in dem die User Threads auf einen Pool von Kernel threads abgebildet werden. Dies reduziert Overhead, da Threads Kontext-Wechsel nicht mehr über den Kern abgewickelt werden, wenn der Kernel thread gleich bleibt. Außerdem teilen sich viele Threads einen Kernel-Stack, den thread-port etc.

8.7 Scheduling

Objekte des Scheduling sind die (Kernel) Threads. Prozessoren werden zu `processor-sets` zusammengefaßt, und Threads können einem processor-set zugeordnet werden. Jedes processor-set hat eine globale 32-Schichten run-queue. Jeder Prozessor hat zusätzlich eine lokale run-queue, auf die er ohne Synchronisation zugreift.

8.8 UNIX Emulation

Für die Emulation existiert ein (systemweiter) UNIX server, der selbst multi-threaded ist.

Mit jeder UNIX Binärdatei wird eine Emulationsbibliothek gebunden. Der Mach Kern fängt UNIX system calls ab (1) und ruft (up-call) die entsprechende Funktion in der Emulationsbibliothek auf (up-call) (2). Diese schickt eine Nachricht an den UNIX server (3). Ein Thread im Server liest die Nachricht, führt den entsprechenden Aufruf aus und schickt das Ergebnis über den bootstrap port des Prozesses zurück (4).

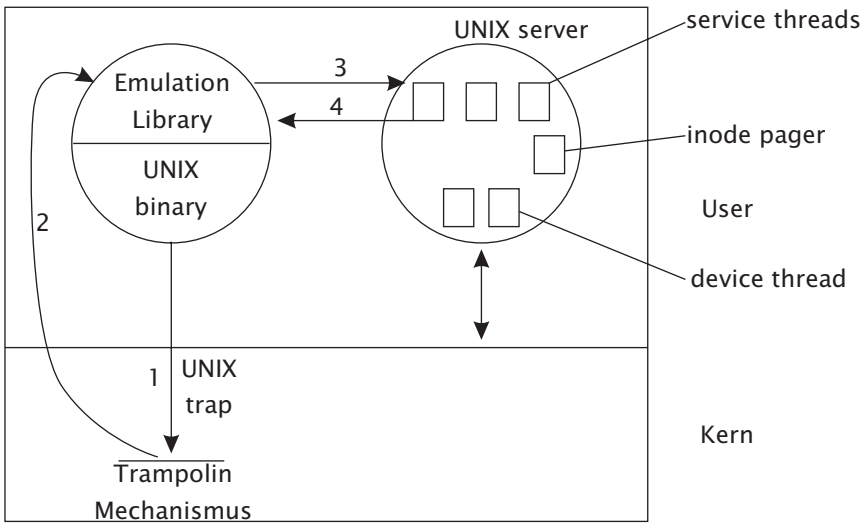


Abbildung 8.11: UNIX-Emulation in Mach

Die Emulationsbibliothek wird bei `fork` vererbt und bei `exec` nicht ersetzt.

File I/O vermeidet aus Effizienzgründen diesen Weg. Jedes UNIX file wird in den Adreßraum der Task abgebildet, mit dem `inode-pager` des UNIX servers als externem pager. Seitenfehler beim Dateizugriff werden also vom inode-pager behandelt, ebenso Zurückschreiben der Datei auf Platte.

8.9 Externe Speicherverwaltung

In Mach können verschiedene Speicherobjekte von verschiedenen Managern verwaltet werden. Diese können verschiedene Paging Strategien implementieren und Objekte maßgeschneidert auf den Platten ablegen.

Kernel und Manager kommunizieren über drei Ports.

`object port`	Kernel schickt Anforderungen an Manager zum object port.
`control port`	Manager schickt Antwort an Kernel zum control port.
`name port`	Identifiziert das Speicherobjekt eindeutig. (Der Kernel antwortet mit dem name port des zugehörigen Objekts, wenn er gefragt wird, wohin eine Speicheradresse gehört).

Am Anfang stellt der Manager den object port zur Verfügung und der Kern den control und name port. Der Manager wartet am object port auf eine Anforderung des Kerns.

Bei einem Seitenfehler sendet der Kern dem entsprechenden Manager die Aufforderung, die Seite einzulagern. Der Kern suspendiert den laufenden Thread vorsorglich, da ja nun der Manager die CPU bekommen muß und zudem keine Antwortzeiten vom Manager als user Prozeß garantiert werden können.

Der Manager holt darauf die Seite in seinen Adreßraum im HSP und schickt dem Kern eine Nachricht mit der Adresse. Hier kann der Manager auf Vorrat auch andere Seiten holen gemäß einer Strategie, damit die nächste Anforderung schneller bedient werden kann. Der Kern bildet die Seite in den Adreßraum des anfordernden Threads ab und macht ihn wieder lauffähig (er läuft nicht unbedingt sofort weiter, da er vielleicht nicht mehr die höchste Priorität hat).

Um im HSP Platz zu bekommen, sucht der `paging daemon` Thread des Kerns von Zeit zu Zeit nach nicht mehr benötigten Seiten. Saubere Seiten sind problemlos. Bei schmutzigen Seiten muß der entsprechende Manager beauftragt werden, sie auf Platte zurückzuschreiben. Der paging daemon operiert global in allen Adreßräumen, da dies am effektivsten ist.

Natürlich gibt es Standard-Managers, z.B. den inode manager für UNIX files und einen, der die malloc Aufrufe mit leeren Seiten bedient.

Das Kommunikationsprotokoll zwischen Kern und Manager benutzt folgende Nachrichten:

`init` (Kern → Manager)	Der Manager soll sich initialisieren, um ein Objekt zu bedienen, was ein Prozeß durch `map` in seinen virtuellen Adreßraum abgebildet hat. Ports werden festgelegt.
`Set_attributes` (M → K)	Antwort auf init. Teilt dem Kern Zugriffsrechte (mode-bits) mit.
`Data_request` (K → M)	Liefere dem Kern eine Seite, um einen Seitenfehler zu beheben.
`Data_provided` (M → K)	Adresse der angeforderten Seite.
`Data_unavailable` (M → K)	Angeforderte Seite ist nicht vorhanden (z.B. Attempt to read through end of mapped file).
`Data_write` (K → M)	Schreibe Seite auf Platte zurück (z.B. Aufforderung des page daemons).
`Lock_request` (M → K)	Schütze Seite gegen Schreiben, damit sie der Manager auf Platte zurückschreiben kann (unterstützt paging Strategie des Managers). Alternativ: ändere mode-bits.
`Lock_completed` (K → M)	Antwort auf lock_request (nötig, da asynchrone Nachrichten, keine system calls benutzt werden).
`Terminate` (K → M)	Objekt wird nicht mehr benutzt (Prozeß terminierte oder hat Objekt dealloziert, z.B. file close).
`Destroy` (M → K)	Zerstöre Objekt (Manager hat geschlossene Datei vollständig zurückgeschrieben; HSP-Region kann freigegeben werden).

Kapitel 9

Realzeitsysteme

Wir geben einen Einblick in den Bereich der Betriebssysteme für Anwendungen, die in Echtzeit reagieren und kommunizieren müssen. Wir beschränken uns hierbei auf Funktionalität, die in dem POSIX.4 Standard spezifiziert ist und eine UNIX Erweiterung darstellt. POSIX.4 deckt im weiteren Sinne den „weichen“ Realzeitbereich (*soft realtime*) ab; der „harte“ Realzeitbereich ist Gegenstand spezieller Vorlesungen zur Prozeßrechner- und Realzeitprogrammierung in der Automatisierungstechnik.

Grob gesagt ist ein Prozeß ein **Realzeitprozeß**, falls sein Ergebnis zu einer bestimmten, realen (system-unabhängigen) Zeit t vorliegen muß, um nützlich zu sein. Ist eine Abweichung δ erlaubt, so spricht man von einer weichen Realzeitforderung, wobei die Skala zwischen weich und hart gleitend ist. Üblicherweise spricht man nur dann von einem Realzeitprozeß, wenn er direkt ein Gerät steuert bzw. über Eingaben und Ausgaben direkt mit der Umwelt kommuniziert.

Ein Wetterprogramm ist kein RT-Prozeß, wenn es Ausgaben auf Papier druckt, auch wenn diese früh morgens fertig sein müssen. Ein Bankautomat wird von einem weichen RT Prozeß gesteuert – die Reaktion darf mit Schwankung δ erfolgen, falls δ akzeptabel klein ist. Eine integrierte Motorsteuerung ist einer harter RT-Prozeß – die Steuersignale müssen auf die Zylinderbewegung genau abgestimmt sein oder der Motor wird zerstört.

Der RT-Bereich umfaßt also alle Gerätesteuerungen vom Flugzeug (*fly-by-wire*) über Bremssysteme (ABS), Industrieroboter, bis hin zu Bankautomaten und Multi-Media Spielgeräten. Klassischerweise war der nicht RT-Bereich die Domäne von Großrechnern und später UNIX-Servern. Im RT-Bereich dominierten als Hardware Microcontroller oder Spezialhardware mit speziellen, proprietären, maßgeschneiderten Betriebssystemen. Diese mußten möglichst klein sein, um in den beschränkten Speicher zu passen und schnelle Prozeßwechsel etc. zu gewährleisten.

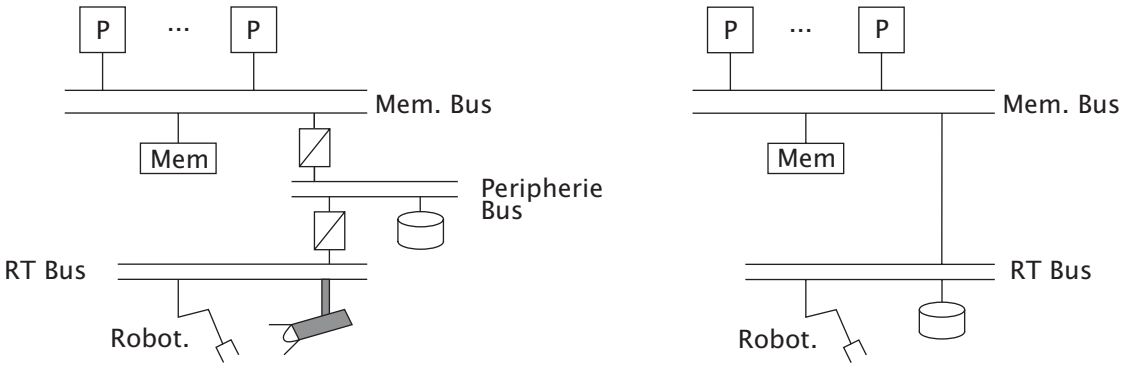
Heute wachsen beide Bereiche auf Systemen der Workstation Klasse zusammen. Da Prozessoren schneller und Speicher größer werden, kann man Standardsysteme einsetzen, wo man Spezialsysteme hatte. Durch den Trend zu Multimedia müssen Arbeitsplatzrechner für den RT-Bereich gerüstet sein.

Typisch für den RT-Bereich sind

- Parallelität, da oft mehrere Geräte gleichzeitig zu steuern sind (z.B. Ein- und Ausgaben, mehrere Motoren in einem Roboter etc.).
- Prioritäten, die vom Benutzer in einer Hierarchie selbst definiert werden, bis hin zur höchsten Systempriorität. Prioritäten in *allen* Warteschlangen im System.
- Paralleles Betriebssystem (*reentrant code*) mit möglichst kurzen kritischen Abschnitten.
- Feingranulare Interprozeßkommunikation auf Benutzerebene.
- Leichte Prozesse, um Prozeßumschaltzeiten möglichst kurz zu halten.
- Asynchrone I/O, damit ein Prozeß nicht unvorhersehbar lange in I/O aufgehalten werden kann.

Damit stellt sich heraus, daß ein für Multiprozessoren geeignetes BS (mit leichten Prozessen und evtl. Mikrokern) bereits die meisten Anforderungen an ein RT-System erfüllt. Es ist daher ein vergleichsweise kleiner Schritt, ein System wie Solaris 2.x auch (weich) RT-fähig zu machen.

Wir gehen im folgenden von einem Standardsystem der Workstation-Klasse aus, bei dem die RT-Geräte durch einen zweiten Bus angesteuert werden. Es gibt zwei Varianten.



9.1 Realzeit IPC

In einem Realzeitsystem muß Kommunikation sicher und schnell sein. Ein portabler Standard muß auch einfach sein. POSIX.4 erweitert die Signale aus UNIX und definiert Messages, Shared Memory und Semaphore neu, die in System V zu kompliziert und langsam geraten sind.

9.1.1 POSIX

Portable OS Interface for UNIX based systems ist ein Interface Standard für UNIX-ähnliche Betriebssysteme. Es werden verschiedene Funktionalitäten spezifiziert. Es bleibt dem BS überlassen, wie es diese implementiert.

Überblick:

POSIX.1 (neu 1003.1 – 1990)	Grundlegende BS-Schnittstellen
POSIX.2	Kommando-Sprache (sh etc.)
POSIX.4 (1003.1b – 1993)	Realzeit-Erweiterungen
POSIX.4a (1003.1c – 1994)	Threads-Erweiterungen
POSIX.4b (1003.1d)	Weitere Realzeit-Erweiterungen

9.1.2 Signale

Es gibt verschiedene Probleme mit den bekannten POSIX.1 Signalen, die eine Erweiterung für den Realzeitgebrauch nötig machen.

Es gibt zu wenig Signale in POSIX.1 (nur SIGUSR1 und SIGUSR2). In POSIX.4 gibt es RTSIG_MAX (≥ 8) Realzeitsignale mit Werten zwischen SIGRTMIN und SIGRTMAX.

POSIX.1 Signale können verlorengehen, falls ein zweites Signal eintrifft, bevor das erste Signal bearbeitet werden konnte (z.B. weil es blockiert war). Das Eintreffen eines Signals wird nur durch ein einziges Bit gespeichert. UNIX versucht deshalb, ein Signal sofort auszuliefern. Es wird zum Empfängerprozeß verzweigt und dessen Signalbearbeiter gerufen. Dieser maskiert üblicherweise das Signal, damit er nicht ein zweites Mal gerufen wird. Ein Realzeitprozeß hoher Priorität könnte jetzt den Handler verdrängen und weitere Signale senden. Dadurch, daß lediglich ein einziges Bit pro Signal gesetzt wird, kann auch nur ein Informationsgehalt von einem Bit direkt übertragen werden (z.B. Temperatur steigt – aber nicht um wieviel).

POSIX.4 RT-Signale werden aufgereiht (*queued*) und können zusätzlich einen Wert in Wortgröße übertragen. Eine Handler-Funktion hat dann den Prototyp

```
void handler(int signum, siginfo_t* data)
```

Entsprechend nimmt die Funktion `sigqueue`, die `kill` ersetzt, einen `siginfo_t*` als entsprechendes Argument.

Unter den POSIX.1 Signalen gibt es keine Priorität. Werden mehrere Signale deblockiert, ist undefiniert, welches zuerst ausgeliefert wird. POSIX.4 Signale haben eine Priorität von

SIGRTMAX (niederste) bis SIGRTMIN (höchste). Es wird stets das aufgereihte Signal der höchsten Priorität als nächstes ausgeliefert.

UNIX Signale sind grundsätzlich ein **asynchrones** Kommunikationsmittel. Ein Prozeß kann ein Signal empfangen, ohne daß er explizit darauf gewartet hat. Dies ist manchmal genau so gewünscht, insbesondere bei Ausnahmen (z.B. SIGSEGV) oder wenn es keine Threads gibt. Die Asynchronität kann Probleme bereiten, z.B. wenn man zwischen Signal Handler und Haupt-Thread synchronisieren muß, um race-conditions zu vermeiden. (Das Signal kann wirklich **jederzeit** eintreffen). Sie verursacht auch Kosten, denn die Umschaltung zum Handler durch das BS ist teurer als ein normaler Funktionsaufruf.

Falls der asynchrone Signalmechanismus aber zur Synchronisation zwischen Prozessen benutzt wird (einer wartet auf das Signal des anderen), so läßt sich der Aufwand verringern. `sigwaitinfo()` blockiert, bis eines der Signale aus einer gegebenen Menge ansteht und liefert seine Nummer ab. Es wird kein Signalhandler gerufen, sondern dies bleibt dem rufenden Prozeß überlassen, der ja ohnehin gewartet hatte. Man vergleiche dazu `sigsuspend()`, das blockiert, bis ein unmaskiertes Signal eintrifft. Erst ruft das BS den Signalhandler, und nachdem dieser fertig ist, wird der Prozeß deblockiert.

Zu `sigwaitinfo()` gibt es auch die Variante `sigtimedwait()` mit timeout. Da POSIX.4-Signale gespeichert werden, kann man race-conditions mit Signalhandlern nun vermeiden, indem man das Signal während eines KA maskiert und danach mit `sigtimedwait()` nachsieht, ob es in der Zwischenzeit eingetroffen war (falls nein, vermeidet man Warten durch sofortigen timeout).

9.1.3 Messages

POSIX.4 definiert Nachrichtenwarteschlangen (*message queues*) mit Prioritäten der Nachrichten. Die Grundfunktionalität ist ähnlich System V, aber die Schnittstelle ist einfacher zu benutzen, da sie analog zu file-handling aufgebaut ist. Die Warteschlangen sind auch einfacher zu implementieren, da z.B. pro WS max. Anzahl und max. Größe der Nachrichten spezifiziert wird.

Überblick über die Funktionalität:

```
/* Öffnen; die letzten zwei Argumente nur für das Kreieren */
mqd_t mq_open(const char *mq_name, int oflag,
    mode_t create_mode, struct mq_attr *create_attrs);

/* Schließen */
int mq_close(mqd_t mqueue);

/* Operation */
int mq_send(mqd_t mymq, const char* msqbuf,
            size_t msgsize, unsigned int msgprio);
i=mq_receive(mqd_t mymq, const char* msgbuf,
            size_t * msgsize, unsigned int* msgprio);
```

/∗ receive oldest msg of highest priority.
Blocks unless queue status was nonblocking.
set_attr, get_attr manipulieren Status. ∗/

Mit `mq_notify()` kann eine Queue beauftragt werden, einem Prozeß ein Signal zu schicken, wenn eine Nachricht eintrifft.

9.1.4 POSIX.4 Semaphore

Es gibt zwei Arten von Semaphoren, benannte und namenlose. Es folgt ein Überblick über die Schnittstelle:

```
/* Memory-based (unnamed) semaphores */
int sem_init (sem_t* semaphore_location.
              int pshared,                  // 1 for inter-process, else 0.
              unsigned int initial_value);

int sem_destroy (sem_t* semaphore_location);

/* Named semaphores */
sem_t* sem_open(const char* semaphore_name,
       int oflags,                          // O_CREAT or O_EXCL
       mode_t creation_mode,                // like file permission
       unsigned int initial_value);
int sem_close(sem_t* semaphore);
int sem_unlink(const char* semaphore_name);

/* Semaphore operations (for both kinds) */

int sem_wait(sem_t* semaphore);
int sem_trywait(sem_t* semaphore);
int sem_post(sem_t* semaphore);
int sem_getvalue(sem_t* semaphore,
                 int* value);
```

9.1.4.1 Semaphore mit Namen

Diese sind eine systemweite Ressource des BS, ähnlich wie in System V. Ähnlich wie z.B. message queues werden existierende Semaphore mit Namen vor dem ersten Gebrauch geöffnet. Ein oflag O_CREAT zeigt an, daß das Semaphor neu geschaffen werden soll, falls es noch nicht existiert; O_EXCL zeigt an, daß der Aufruf mit Fehler (-1) abbrechen soll, falls das Semaphor schon existiert. Weitere flags (O_RDWR) sind implizit und können nicht explizit angegeben werden. `creation_mode` spezifiziert wie bei einem File grundsätzliche Rechte am Semaphor.

Die Makros S_IRWXU, S_IRWXG und S_IRWXO machen das Semaphor für user, group und others benutzbar.

`Close` und `unlink` funktionieren ähnlich wie bei Nachrichtenwarteschlangen.

Wait, Post und Try_wait funktionieren wie erwartet (down, up, down mit timeout). Unter einem RT-Scheduler ist das Warten priorisiert, d.h. der wichtigste Prozeß bekommt das Semaphor als nächstes.

Getvalue kann (u.a. bei der Fehlersuche) eingesetzt werden, um den Zustand des Semaphors auszugeben.

9.1.4.2 Speicherbasierte Semaphore

Diese sind eine Ressource des Prozesses und werden im Speicher angelegt durch `sem_init`. `pshared == 1` ist für den Gebrauch zwischen Prozessen, `pshared == 0` für den Gebrauch zwischen Threads innerhalb eines Prozesses. `sem_init` nimmt als Argument einen Zeiger auf die Speicherstelle, an der das Semaphor initialisiert werden soll. Das Semaphor kann also in Datenstrukturen eingegliedert werden.

9.2 Shared Memory

Gemeinsamer Speicher wird in POSIX.4 logisch als File angesehen, das mittels *mmap* in den HSP abgebildet wird. Ein SM-Objekt wird deshalb geöffnet, abgebildet, geschlossen und evtl. zerstört. Offene und abgebildete Speicherobjekte werden bei `fork()` vererbt und sind dann wie Files beiden Prozessen zugänglich.

9.2.1 Öffnen

```
int shm_open(const char* name, int oflag, mode_t mode);
```

gibt einen File-Deskriptor zurück, der das Shmem-Objekt bezeichnet. `shm.open` funktioniert wie `open`: das oflag O_CREAT sorgt dafür, daß das Objekt wenn nötig erzeugt wird und `mode` spezifiziert Zugriffsrechte wie O_RDWR oder O_RDONLY.

9.2.2 Größe bestimmen

```
int ftruncate(int fd, off_t total_size);
```

setzt die Größe des Objektes fd fest (für alle Benutzer des Objekts). Ein Ergebnis < 0 zeigt einen Fehler an.

9.2.3 Aufräumen

```
int close(int fd)
```

schließt den Deskriptor eines SM-Objektes. Das SM-Objekt selbst wird aber nicht entfernt.

```
int shm_unlink(const char* name);
```

markiert das SM-Objekt zur Zerstörung vor. Die Zerstörung findet statt, wenn jeder beteiligte Prozeß die Objekt-Abbildung aufgehoben hat.

```
int munmap(void* begin, size_t length);
```

hebt die Abbildung der Seiten auf, die die Adressen von `begin` bis `begin` + `length` enthalten. Es ist deshalb sinnvoll, den Speicherbereich auf Seitengrenzen beginnen und enden zu lassen.

9.2.4 Speicherabbildung

Ein durch einen File-Deskriptor bezeichnetes Objekt (File oder SM-Objekt) kann durch `mmap` in den HSP abgebildet werden.

```
void* mmap(void* where,              // Wunsch 0=anywhere
           size_t length,
           int memory_protections,   // Zugriffsrechte
           int mapping_flags,        // Shared, private, fixed
           int fd,                   // Objekt
           off_t offset              // im Objekt
          );
```

`length` Bytes, beginnend bei `offset` im Objekt `fd`, werden in den HSP abgebildet, wenn möglich beginnend mit Adresse `where`. (0 steht für egal wo). Die tatsächlich gewählte Adresse wird als Ergebnis zurückgegeben. Die Seiten des HSP Bilds bekommen die Zugriffsrechte `memory_protections` in der MMU:
PROT_READ, PROT_WRITE, PROT_EXEC oder PROT_NONE
EXEC erlaubt lediglich das Ausführen von Code, NONE erlaubt nichts und eignet sich zum Anlegen von Seiten, die als Brandmauern Speicherbereiche abgrenzen.

Das Flag `shared` bestimmt den Gebrauch zu shared memory, `fixed` spezifiziert, daß `where` nicht Wunsch, sondern Befehl ist, und `private` verhindert, daß Änderungen in das SM-Objekt zurückgeschrieben werden.

9.2.5 Seitenschutz

Es ist möglich, die MMU Schutzbits einzelner Seiten individuell zu bestimmen. Z.B. kann ein Stacksegment abgeschlossen werden durch eine Seite, auf die nicht zugegriffen werden darf. Dies realisiert eine Art Brandmauer gegen das nächste Stacksegment – läuft ein Pointer über und in die geschützte Seite hinein, wird das Signal SIGSEGV erzeugt. Der Aufruf ist

```
int mprotect(void* begin, size_t length, int memory_protections);
```

9.2.6 Synchronisation

Falls das abgebildete Objekt tatsächlich eine Datei ist, kann es nützlich sein, Datei und HSP Bild zu synchronisieren, so daß beide garantiert übereinstimmen. Der Aufruf ist

```
int msync(void* begin, size_t length, int flags);
```

Ist das flag MS_SYNC gesetzt, blockiert `msync`, bis Datei und Bild synchron sind. Ist MS_ASYNC gesetzt, wird eine Synchronisation eingeleitet, aber deren Ende nicht abgewartet. Ist MS_INVALIDATE gesetzt, werden die entsprechenden Seiten in jedem anderen beteiligten Prozeß invalidiert und mit dem synchronisierten Bild nachgeladen.

9.3 Realer Hauptspeicher

Seitentauschaktivitäten in virtuellem Speicher können ein Realzeitsystem empfindlich stören. Falls ein Speicherzugriff zu einem Seitenfehler führt, verlangsamt er sich um einen Faktor 10^5 bis 10^6 (Millisekunden statt Nanosekunden). Besonders schlimm ist, daß das Seitentauschverhalten des BS nicht vorhersehbar ist und ein RT-Prozeß deshalb von anderen Prozessen in undurchschaubarer Weise abhängig wird.

In POSIX.4 kann man einen Prozeß ganz oder in Teilen im HSP festnageln. Man konvertiert somit virtuellen HSP zu realem HSP.

```
int mlockall(int flags)
```

nagelt den ganzen Prozeß fest – Text, Daten, Heap, Stack, shared libs, shared memory. flags sind MCL_CURRENT und MCL_FUTURE und sagen aus, welche mappings mit einbezogen werden sollen – die jetzt vorhanden oder die zukünftigen.

```
int munlockall(void)
```

hebt die Festlegung des Prozesses wieder auf. Eine Festlegung von Teilen des Speichers geschieht mit

```
int mlock(void* address, size_t length)
```

im Bereich von address bis address + length. Natürlich werden immer ganze Seiten beeinflußt, so daß u.U. mehr Speicher festgelegt, aber auch befreit wird. (PAGESIZE wird in `<limits.h>` festgelegt).

```
int munlock(void* address, size_t length)
```

befreit den Bereich von address bis address + length. Die Festlegung einer Seite wird nur in einem Bit gespeichert, so daß ein einziges `unlock` beliebig viele Festlegungen der Seite aufhebt.

9.4 Real-Time Scheduling

9.4.1 Lösungen in Standard UNIX

Als Super-User kan man *nice* dazu benutzen, die Priorität eines Prozesses zu erhöhen. Dadurch wird der Prozeß statistisch gesehen besser/schneller laufen. Da der UNIX Scheduler aber nicht im einzelnen spezifiziert ist, gibt es keine Garantie, daß der Prozeß auch immer zuverlässig läuft, wenn er lauffähig ist. (*nice* ist nur einer von mehreren Parametern, die in die effektive Prozeßpriorität eingehen).

Eine bessere, aber auch umständliche Lösung ergibt sich in System V mit *priocntl*:

```
long priocntl(idtype_t idtype,
              id_t id,
              ind cmd,
              ... )
```

Ist idtype gleich P_PPID, so wird die Priorität aller Kinder des Prozesses *id* gesetzt, ansonsten die von Prozeß *id* selbst. Weitere Optionen beziehen sich auf Gruppen-, Sitzungs- oder User-Id oder auf jeden Prozeß im System.

cmd kann benutzt werden um Informationen über die Prozeßpriorität zu bekommen oder um die Priorität zu setzen. Es gibt drei Schedulingklassen: Real-Time, System, Time-Sharing. Man kann einen Prozeß in die RT-Klasse heben, ihm dann eine RT-Priorität geben und die maximale Zeitscheibe definieren, die er läuft, bevor er vom nächsten Prozeß gleicher Priorität abgelöst wird.

Wichtig ist, daß RT über System liegt, d.h. ein RT-Prozeß wird nicht durch System-Aktivitäten wie Dämonen etc. unterbrochen, sondern lediglich durch andere RT-Prozesse höherer Priorität.

9.4.2 POSIX.4 RT Scheduling

Wir geben zunächst einen Überblick über die Schnittstelle:

```
#include <unistd.h>
#include <sched.h>
int i, policy;
struct sched_param scheduling_parameters;
pid_t pid;

int sched_scheduler(pid_t pid,               //process id
                    int policy,              //Algorithmus bzw. Verfahren
                    struct sched_param* scheduling_parameters);
```

```
int sched_getscheduler(pid_t pid);

int sched_getparam(pid_t pid,
                   struct sched_param* scheduling_parameters);

int sched_setparam(pid_t pid,
                   struct sched_param* scheduling_parameters);

int sched_yield(void);

int sched_get_priority_min(int policy);

int sched_get_priority_max(int policy);
```

Scheduling Parameter werden in einer Struktur abgelegt; momentan ist in `sched_param` als einziges Feld `int sched_pirority;` definiert. Später können weitere hinzu kommen.

9.4.3 POSIX.4 Scheduling Verfahren

POSIX.4 kennt bisher drei Verfahren *policy* für die Ablaufplanung:

1. SCHED_FIFO: (preemptive, priority-based scheduling). Prioritätsbasierte Ablaufplanung mit Prozessorentzug nur durch Prozesse höherer Priorität.
2. SCHED_RR: (round-robin, preemptive priority-based scheduling with quanta). Prioritätsbasierte Ablaufplanung mit Prozessorentzug sowohl durch Prozesse höherer Priorität als auch durch Prozesse gleicher Priorität nach Ablauf einer Zeitscheibe.
3. SCHED_OTHER: (implementation defined scheduler). In der Praxis z.B. der übliche UNIX timesharing Scheduler.

Das wichtigste Verfahren für RT-Prozesse ist SCHED_FIFO. Ein solcher Prozeß hat die CPU so lange, bis er sie selbst aufgibt oder bis ein anderer mit höherer Priorität lauffähig wird. In der Praxis weiß man dann (auf einem Uniprozessor), daß der Prozeß seine Arbeit zu Ende bringen kann, wenn er nicht selbst blockiert, einem wichtigeren ein Signal schickt oder ein anderer durch externe HW-Interrupts lauffähig wird. Diese Eigenschaft ist besonders wichtig, denn ein RT-Prozeß sollte nicht unterbrochen werden, so lange er Geräte, E/A-Kanäle oder andere Ressourcen exklusiv reserviert hat.

SCHED_RR ist ähnlich zu SCHED_FIFO, die Prozesse laufen aber höchstens eine Zeitscheibe lang, die durch das System bestimmt wird. (In System V kann die Zeitscheibe durch den Benutzer gesetzt werden. Mit dem Quantum TQ_INF erhält man das Äquivalent zu SCHED_FIFO). Mit

```
sched_rr_get_interval(pid_t pid)
```

kann man das momentane Quantum erfahren.

SCHED_RR eignet sich gut für Hintergrundprozesse (z.B. logging), die während der RT-Verarbeitung tätig sind, die Maschine aber nicht blockieren dürfen. Solche reinen Rechenprozesse, die keine Geräte exklusiv reservieren, können jederzeit unterbrochen werden.

9.4.4 RT Scheduling Theorie

Zur Bestimmung der Prioritäten in einem RT System gibt es zwei wichtige Verfahren: rate-monotonic scheduling und earliest deadline first scheduling.

earliest deadline first (EDF):
Es wird immer derjenige Prozeß als nächstes auf den Prozessor gebracht, der am ehesten fertig sein muß. Dieses Verfahren ist *feasible* und optimal. (Ein Verfahren ist *feasible*, wenn es immer dann einen Laufplan unter Einhaltung der Beschränkungen generiert, falls dies überhaupt möglich ist). EDF wird von POSIX.4 nicht direkt unterstützt.

rate-monotonic scheduling (RMS):
Dies ist das in der Praxis am häufigsten angewendete Verfahren. Man kann es als Näherungslösung zu EDF verstehen. RMS gibt dem Prozeß mit der höchsten Frequenz an Rechenperioden die höchste Priorität. Diese Priorität kann mit POSIX.4 `sched_setparam` gesetzt werden. Der FIFO Scheduler führt dann den durch RMS gefundenen Plan aus.

Um eine Prozeßpriorität zu bestimmen, bestimmt man also zuerst, wie häufig der Prozeß ablaufen muß (z.B. muß er 50mal/sec einen Meßwert übernehmen). Man sortiert diese Frequenzen absteigend und vergibt die Prioritäten in dieser Reihenfolge (Gleichstände werden priorisiert um Zufälligkeiten auszuschließen).

Die Idee des Verfahrens ist, die Prozesse so einzuplanen, daß möglichst wenige Termine verpaßt werden. Hochfrequente Prozesse haben die häufigsten Termine, bekommen also die höchste Priorität. Außerdem ist bei einem hochfrequenten Prozeß die Wahrscheinlichkeit am größten, daß er bald einen Termin hat, also sowieso nach EDF eingeplant werden müßte.

9.5 Zeit-Dienste

Zeit-Dienste sind eine essentielle Aufgabe des BS Kerns. Wir unterscheiden zwischen (log.) Uhren (*clocks*), die die Zeit enthalten, und (Kurzzeit-)Weckern (*timer*), die losgehen und den Prozeß wecken. Es gibt einmalige und periodische Wecker.

UNIX (d.h. POSIX.3) enthält eine Uhr, die über einen Zähler fortlaufend die Zeit angibt, die seit dem 01.01.1970, 0 Uhr morgens, verstrichen ist.

In System V und POSIX.1 wird diese Zeit in Sekunden ausgelesen durch

```
time_t time(time_t* time_p)
```

Falls `time_p != 0`, wird die Zeit als Seiteneffekt dort auch abgelegt.

In BSD und S V R4 gibt es

```
int gettimeofday(struct timeval* time_p)
```

Ausgabe ist 0 oder -1 für Erfolg und Mißerfolg; die Zeit wird abgelegt in einer Struktur

```
struct timeval{
               time_t tv_sec;        //seconds
               time_t tv_usec;       //microseconds
              };
```

Üblicherweise ist die Auflösung nur *clock ticks* und die Uhr tickt mit 100 Hz; d.h. jeder tick schaltet 10.000 μsec weiter.

Bemerkung: Auf den meisten UNIX Maschinen wird Zeit in 32 bit long int gezählt. Diese Zähler werden im Januar 2038 überlaufen und vermutlich viele Programme zum Absturz bringen (falls es 32 bit UNIX Software dann noch gibt). □

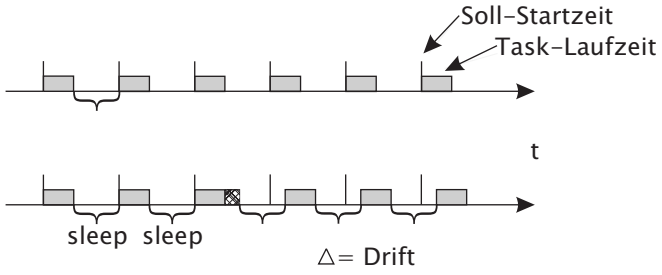
9.5.1 Zeitintervalle

RT-Anwendungen brauchen typischerweise periodische Tasks, die regelmäßig wiederkehrende Arbeit verrichten und dann bis zum nächsten Mal schlafen.

In UNIX schläft man mit

```
unsigned int sleep(unsigned int seconds);
```

Dies ist für RT-Anwendungen nicht geeignet. Die Sekunden-Auflösung ist zu ungenau, und `sleep()` kann durch Signale unterbrochen werden (in System V wird dann die noch zu schlafende Zeit zurückgegeben). Außerdem schläft man relativ zur Einschlafzeit, d.h. bei Varianzen in der Einschlafzeit driftet die Periode der Aufwachzeit.



9.5.2 UNIX Intervall Wecker

BSD und SVR4 haben folgende Schnittstelle:

```
struct itimerval{
       struct timeval it_value;        //initial value, 0 = off
       struct timeval it_interval;     //period value, 0 = once
 }

int setitimer(int which_timer,         //REAL, VIRTUAL, PROF
              const struct itimerval* new_itimer_value,
              struct itimerval* old_itimer_value)

int getitimer(int which_timer,
              struct itimerval* itimer_value);
```

Es gibt drei verschiedene Timer:
ITIMER_REAL, der in Realzeit operiert,
ITIMER_VIRTUAL, der in Systemzeit operiert, und
ITIMER_PROF für Unterbrechungen zum Profiling in Systemzeit.

Alle Timer schicken dem Prozeß ein Signal, nämlich SIGALRM, SIGVTARLM und SIGPROF.

Für RT-Applikationen ist nur ITIMER_REAL brauchbar. Mit den anderen Timern kann man die Zeit messen, die ein Prozeß in einem bestimmten Codesegment braucht. Man beachte wieder, daß die Mikrosekundengenauigkeit der Itimer-Struktur i.a. nicht wirklich genutzt wird. Zeit wird nur HZ mal pro Sekunde inkrementiert. Es kommen also jedesmal gleich 1.000.000/HZ μsec dazu. Setzt man den Itimer z.B. auf ein 9000 μsec Intervall, so ist es wahrscheinlich, daß er erst nach 1.000.000/100 = 10.000 μsec losgeht. In Realzeit driftet der Prozeß also wieder.

Schwachstellen der UNIX Uhren und Timer sind

- nur 1 RT-Uhr
- Zu geringe Auflösung, sogar in der Time-Struktur; moderne Hardware-Uhren können Nanosekunden zählen (100 MHz $\hat{=}$ 10 ns Intervall).
- Keine Möglichkeit, Timer-Überläufe festzustellen (der Wecker klingelt mehrfach, bevor man reagiert hat).
- Beschränkung auf SIGALARM als Signal.

9.5.3 POSIX.4 Uhren und Timer

Wir geben zunächst einen Überblick über die Funktionalität:

```
#include <unistd.h>
#include <signal.h>
#include <time.h>

struct timespec{
                time_t tv_sec;           //seconds
                long tv_nsec;            //nanoseconds
               };

// Clocks
int clock_settime(clockid_t clock_id,    //clock ID
                  const struct timespec* current_time);
int clock_gettime(clockid_t clock_id,
                  struct timespec* current_time);
int clock_getres(clockid_t clock_id,
                 struct timespec* resolution);

// Timers
int timer_create(clockid_t clock_id,     //Basis-Uhr
    const struct sigevent* signal_spec,  //Assoz. Signal
    timer_t* timer_id);                  //ID des neuen Timers

int timer_settime(timer_t timer_id,
          int flags,                         //Rel. Zeit, Abs. Zeit
    const struct itimerspec* new_interval,   // Neu
          struct itimerspec* old_interval);  //Bisher

int timer_gettime(timer_t timer_id,
          struct itimerspec* cur_interval);

int timer_getoverrun(timer_t timer_id);
int timer_delete(timer_t timer_id);

// High-Resolution Sleep

int nanosleep(
              const struct timespec* requested_time_interval,
              struct timespec* time_remaining);
```

9.5.4 Uhren

POSIX.4 verlangt lediglich eine einzige Uhr, CLOCK_REALTIME, ermöglicht aber das Bereitstellen weiterer Uhren, z.B. für virtuelle Systemzeit, externe Zeit etc. Man bekommt die Zeit z.B. durch

```
struct timespec current_time;
clock_gettime(CLOCK_REALTIME,
              &current_time);
```

Durch die ANSI-C Funktionen *ctime*, *localtime* und *qmtime* kann der Sekunden-anteil der Zeit in lesbare Strings umgewandelt werden. Hardware-Uhren haben heute eine Auflösung bis 500 ns (SPARC) und 21 ns (SGI).

Mit `clock_getres()` erhält man die Auflösung einer bestimmten POSIX.4 Uhr. (Man beachte, daß die POSIX-Uhren **virtuelle** Uhren sind, die von der Hardware periodisch hochgezählt werden. 50 MHz SPARC z.B. hat offensichtlich eine 20ns HW-Uhr, aber nur eine 500ns POSIX-Uhr. Die Mindestauflösung von CLOCK_REALTIME beträgt 20ms ($\hat{=}$ 50 Hz).

Falls `clock_getres()` einen Fehlerwet (< 0) liefert, existiert die gewünschte Uhr auf dem System nicht.

9.5.5 Sleep

`nanosleep()` funktioniert ähnlich wie `sleep()` nur mit höherer Auflösung. `*time_remaining` liefert die Restzeit zurück, falls der Schlaf durch ein Signal vorzeitig unterbrochen wurde (errno = EINTR).

9.5.6 Intervall Timer

Intervalle werden gemessen mit

```
struct itimerspec{
       struct timespec it_value;        //erster Weckruf
       struct timespec it_interval;     //Weck-Intervall
       };
```

`it_value` gibt die Zeit des ersten Weckrufs an, `it_interval` die Periode, nach der jeweils weitere Weckrufe erfolgen.

POSIX.4 timer werden für jeden Prozeß dynamisch erzeugt und werden von einer der POSIX.4 Uhren abgeleitet. Außerdem läßt sich für jeden Timer spezifizieren, welches POSIX Signal er ausliefern soll; typischerweise benutzt man die POSIX.4 Realzeitsignale. `timer_id` ist dann die ID des erzeugten Timers.

Mit `timer_delete` räumt man nicht mehr benötigte Timer auf, um Verwaltungsaufwand im BS zu sparen.

Mit `timer_settime()` stellt man den Timer ein. Mit `new_interval` gibt man die erste Weckzeit an und das Intervall, nach dem immer weitere Weckrufe erfolgen sollen. In `old_interval()` erhält man den alten Wert zur Kontrolle zurück. Mit `flags` spezifiziert man, ob die erste Weckzeit relativ zur momentanen Zeit angegeben wurde oder ob dies eine absolute Zeit ist. Die Angabe einer relativen Startzeit ist im RT-Bereich gefährlich, da man kurz vor dem Setzen der Zeit schlafen gelegt werden kann. Diese Schlafdauer wird dann implizit zur relativen Startzeit addiert, da man erst nach der Schlafdauer den Timer wirklich setzt.

Literaturverzeichnis

[Literatur für das Kapitel: Aufbau und Funktionsweise eines Computers]

[Bra98] R. Brause. *Betriebssysteme – Grundlagen und Konzepte.* Springer-Verlag, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988. Zweite Auflage.

[KW05] Wolfgang Küchlin and Andreas Weber. *Einführung in die Informatik.* Springer-Verlag, 2005. Dritte Auflage.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts.* Addison-Wesley, 1998. Fünfte Auflage.

[Str93] B. Stroustrup. *The C++ Programming Language, Zweite Auflage.* Addison-Wesley, 1993.

[Str97] B. Stroustrup. *The C++ Programming Language, Dritte Auflage.* Addison-Wesley, 1997.

[Tan76] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, 1976.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 2001.

[TG01] Andrew S. Tanenbaum and James Goodman. *Computerarchitektur.* Pearson Studium, 2001. Original: Structured Computer Organization, Prentice Hall, 1999.

[Wir95] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1995.