Betriebssysteme I

Wintersemester 1993/94

Inhaltsverzeichnis

1 Einführung und Überblick **1**

2 *UNIX* Einführung **3**
2.1 Shell . . . 3
2.2 *UNIX* Dienstleistungsprogramme (Utilities) . . . 4

3 Grundlegende Konzepte von *UNIX* **5**
3.1 Prozesse . . . 5
3.2 *POSIX*-Standard . . . 6
3.3 Implementierung von Prozessen in *UNIX* . . . 6
3.4 Information in der Prozeßtabelle . . . 6
3.5 User structure . . . 7
3.6 Realisierung von *fork*() . . . 7
3.7 Leichte Prozesse . . . 7
3.8 C-Thread „System Aufrufe“ . . . 8
3.9 Scheduling (Prozeßlaufplanung) . . . 9
3.10 Berechnung der Priorität eines Prozesses . . . 10

4 Das *UNIX* Speichermodell **11**
4.1 Virtueller Speicher . . . 12
4.2 Swapping (Prozeßtauschverfahren) . . . 12
4.3 Paging (Seitentauschverfahren) . . . 13
4.4 Der *UNIX* Seitentauschalgorithmus . . . 13

5 Das *UNIX* Dateisystem **15**
5.1 Implementierung des Dateisystems . . . 16
5.2 Lesen einer Datei . . . 17
5.3 Pipes . . . 18
5.4 Input/Output . . . 18

6 Interprozeßkommunikation (*IPC*) **20**
6.1 Zugriffskonflikte (race conditions) . . . 20
6.2 Wechselseitiger Ausschluß mit aktivem Warten . . . 20
6.2.1 Deaktivieren von Unterbrechungen . . . 20
6.2.2 Lock Variables (Schloßvariablen) . . . 20
6.2.3 Strikte Abwechslung (strict alternation) . . . 20
6.2.4 Peterson's Lösung . . . 21
6.3 Die TSL Instruktion (test and set lock) . . . 21
6.4 Sleep, Wakeup . . . 22
6.5 C Thread Conditions . . . 24
6.6 Semaphore . . . 24
6.7 Monitore (Ausführungskontrolleure) . . . 25
6.8 Nachrichtenaustausch (Message Passing) . . . 26
6.9 Rendez-vous (siehe ADA) . . . 27
6.10 Äquivalenz der Konzepte . . . 27
6.11 IPC in *UNIX* . . . 27
6.12 IPC in System V . . . 28
6.12.1 Semaphore in System V . . . 28
6.12.2 Shared Memory . . . 29
6.12.3 Messages . . . 30
6.13 Bekannte IPC Probleme . . . 30
6.13.1 Essende Philosophen (Dijkstra) . . . 30

6.14 Prozeßablaufplan (process scheduling) . . . 33
6.15 Karussell Laufplanung (round robin scheduling) . . . 35
6.16 Laufplanung mit Prioritäten . . . 35
6.17 Mehrfachwarteschlangen . . . 35
6.18 Zwei Ebenen Ablaufplanung . . . 36

7 Das Betriebssystem MACH 37
7.1 Der MACH Mikrokern . . . 37
7.2 Nachrichten in MACH . . . 39
7.3 Speicherverwaltung . . . 41
7.4 MACH Prozesse . . . 42
7.5 Systemaufrufe zur Prozeßverwaltung . . . 43
7.6 MACH Threads . . . 43
7.7 Scheduling . . . 43
7.8 *UNIX* Emulation . . . 43
7.9 Externe Speicherverwaltung . . . 44
7.10 Der Netzwerk Nachrichtendienst (NND) . . . 46

8 Speicherverwaltung **48**
8.1 Seitentauschalgorithmen . . . 48
8.1.1 Optimaler Seitentausch . . . 48
8.1.2 FIFO . . . 48
8.1.3 NRU (not recently used) . . . 48
8.1.4 2^{nd}Chance . . . 48
8.1.5 Clock . . . 49
8.1.6 LRU (least recently used) . . . 49
8.2 HW Implementierung . . . 49
8.3 SW Implementierung . . . 49
8.4 Modellierung des Seitentauschs . . . 49
8.5 Das Lokalitätsprinzip . . . 50
8.6 Implementierung des Seitentauschs . . . 51
8.7 Schritte des Seitentauschs . . . 51

9 Input / Output **53**
9.1 Geräte . . . 53
9.2 Controller . . . 53
9.3 DMA (direct memory access) . . . 53
9.4 Interrupt Handlers . . . 53
9.5 Device Drivers (Gerätetreiber) . . . 53
9.6 Geräteunabhängige Software . . . 54
9.7 Festplatten . . . 54
9.8 Arm Scheduling . . . 54
9.9 Elevator (Fahrstuhl) . . . 54
9.10 RAID . . . 54
9.11 Uhren (Timer) . . . 55

10 Deadlocks (Verklemmungen) **56**
10.1 Bedingungen für Verklemmungen . . . 56
10.2 Modellierung von Verklemmungen . . . 57
10.3 Wiederaufsetzen nach Verklemmung . . . 57
10.4 Mehrfachbetriebsmittel . . . 57

Abbildungsverzeichnis

1 Multiprozessorsystem mit gemeinsamen Hauptspeicher (z.B. Solbourne 704) 7
2 Seiteneinteilung im HSP 14
3 Semaphordatenstruktur 28
4 Prozeßzustandsdiagramm 33
5 Deskriptor in einer komplexen Nachricht 40
6 Beispiel für einen MACH Adreßraum 42
7 Eintrag in eine Seitentabelle 48
8 Einfachster Deadlock 57

1 Einführung und Überblick

Das Betriebssystem (BS) ist die Vergabe- und Verwaltungsstelle für alle Betriebsmittel. Zentrale Aufgaben des BS 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 geräteunabhängige Schnittstelle zur darunterliegenden Hardware.
Es liefert abstrakte, höhere Zugriffsroutinen als es die einzelnen Gerätefunktionen sein können. Es realisiert eine (möglichst portable) abstrakte Maschine auf dem jeweiligen physischen Gerät, die einfacher und geräteunabhängig handhabbar ist.

Einheiten der Vorlesung:

1. Einführung und Überblick
2. Prozesse
3. Speicherverwaltung
4. Dateisysteme
5. Ein-/Ausgabe
6. Synchronisation und Verklemmungen
7. Eng gekoppelte parallele Systeme und MACH

Einführung:

5	Lohn/Gehalt, ..., Spiele
4	Compiler, Editoren, ...
3	Betriebssystem
2	Maschinensprache
1	Mikroprogramm
0	Physikalische Geräte

0: Hardware, wie sie physikalisch beschaffen ist

1: μProgramme setzen Befehle der Maschinensprache um in die Steuerung einzelner Gatter (gates)

2: Softwareschnittstelle, die von Ebene 1 realisiert wird. Definiert Architektur unabhängig vom Gerät

3: Das BS stellt SW-Architektur unabhängig von der HW-Architektur dar (Filesystem, Prozesse, virtueller Speicher, ...)

4: Utilities (Dienstleistungsprogramme) stellen SW-Entwicklungsarchitektur (SW-Systemumgebung) unabhängig von der Systemarchitektur dar.

5: Applikationen-Architektur unabhängig von Entwicklungsarchitektur.

Beispiel zur Abstraktion:

Plattenzugriff auf Ebene 2:

Lade Geräteregister (spezielle Adresse)
mit: Plattenadresse, Zylinder, Sektor, Offset
Hauptspeicheradresse
Anzahl der zu übertragenen Bytes
Lese / Schreibe

Betriebssystemaufruf auf Ebene 3:

read(file, buffer)
/* liest 1 Block aus Datei „file“ in „buffer“
siehe „man read“ */
int read(fd, buf, nbyte)
int fd;
char *buf;
int nbyte;
/* Read() attempts to read bytes of data from object referenced by the descriptor *fd* into the buffer pointed to by *buf*. The result is the number of bytes actually read */

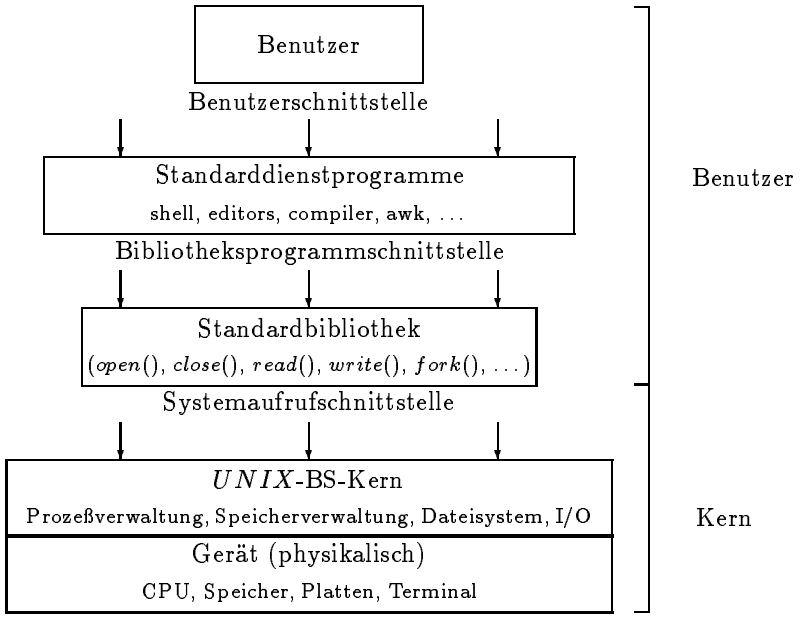
Bibliotheksaufruf auf Ebene 4:

$fscanf(file, format)$ liest Zeichenreihe gemäß Format „format“ (z.B. 1 Integer und 1 Real) von Datei „file“ ein, egal ob kürzer oder länger als ein Block.

Beispiel zur Verwendung:

```
#include <stdio.h>
main()
{
  char *buf = "Test.\n";
  read(0,buf,4); /* lese 4 Bytes von 0=stdin */
  printf(buf);
}
```

2 *UNIX* Einführung



- Systemaufrufe (system calls) erfolgen über die Bibliotheksprogrammschnittstelle per C-Prozeduraufruf (s. Bsp. *read*()).
- Aufrufe über die Systemaufrufschnittstelle hinweg erfolgen durch Aufruf von Assemblerroutinen (je eine pro Aufruf). Jede legt Parameter im Register oder auf den Stack ab und führt eine **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.

2.1 Shell

UNIX enthält ein „Schalenprogramm“, das mit dem Benutzer interagiert. Es akzeptiert seine Kommandos, führt die entsprechenden Programme (gleichen Namens) aus und leitet deren Ausgaben weiter. Die *Shell* ist ein Benutzerprogramm, das vom Terminal lesen und darauf schreiben, sowie Programme ausführen kann (Flavors: sh, csh, tcsh, ksh). Eine Shell wird vom **login** Programm automatisch gestartet, weitere können jederzeit später gestartet werden.
Dienstleistungen der Shell u.a.:

- I/O-Umleitung (sort < in > out)
- Pipes: sort < in | more
- Kommandoergänzungen: ls *.c
- Hintergrundprozesse: a.out &

Shelleingabe ist selbst wieder eine Sprache. Shellprogramme (scripts) sind Sequenzen in Shellsprache.

Beispiel:

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

Beliebte Dateiverzeichnisse:

/bin	ausführbare Systembinärdateien
/dev	Dateien, die für Geräte stehen (/dev/null: leeres Gerät)
/etc	Systemverwaltung
/bib	Bibliotheksfunktionen
/tmp	Temporärdateien
/usr	Benutzerdateien
/usr/adm	Systemverwaltung
/usr/*name*	der Katalog für den User mit login *name*
/usr/bin	weitere bin's
/usr/include	Systemvorspanndateien
/usr/lib	Compiler, ...
/usr/spool	Zwischendateien für Drucker
/usr/scr	System Quellcode
/usr/tmp	Temporärdateien

2.2 *UNIX* Dienstleistungsprogramme (Utilities)

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

(*POSIX* 1003.2)

3 Grundlegende Konzepte von *UNIX*

3.1 Prozesse

Ein Prozeß ist die elementare Ablauf- und Verwaltungseinheit im Rechner. Er besteht aus Programmcode und der gesamten, für dessen Ablauf nötigen Umgebung (**context**) (z.B. Hauptspeicher, Dateien, Programmzähler, Abrechnungsinformationen, Signalbehandlung, ...).
UNIX ist ein Mehrprozeßbetriebssystem. Es kann mehrere Prozesse gleichzeitig verwalten. Es kann zwischen lauffähigen Prozessen hin- und herschalten, so daß sie scheinbar gleichzeitig ablaufen (in Multiprozessormaschinen kann dies tatsächlich gleichzeitig geschehen).
UNIX macht es für den Benutzer einfach, neue Prozesse zu erzeugen (via **fork()**). Systemprozesse im Hintergrund heißen „Dämonen“ (demons).

Beispiel:
cron wacht einmal pro Minute auf, sieht sich in der Tabelle „crontab“ nach Arbeit um und führt diese dann aus. Danach schläft er wieder.

Kind-Prozesse werden durch $fork()$ vom Vater erzeugt ($pid = fork()$). Das Kind beginnt als exakte Kopie des Vaters (Programm+Daten). Danach kann der Kontrollfluß zwischen Vater und Kind verschiedene Wege gehen. Das Resultat *pid* von $fork()$ ist $= 0$ im Kindprozeß, im Vater jedoch > 0 (die Speicherbereiche sind getrennt).

Beispiel:

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

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

Prozeßsynchronisation

1. Prozesse können zu „Leitungen“ (*pipelines*) verbunden werden
   (Baukastenprinzip). Bsp.: ls | more
   Generell: 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 oder gefüllt werden kann. Der Systemaufruf heißt **pipe** (s. File-System).

2. Signale (Software-Unterbrechung)
   Ein Prozeß kann einem anderen seiner Prozeßgruppe ein **Signal** senden (via BS). Der Prozeßgruppe gehören Ahnen, Geschwister und Nachfahren an.
   Der BS-Aufruf zum Senden eines Signals ist:
   $kill(pid, signum)$, wobei pid=Prozeß-ID und signum=signal

3.2 *POSIX*-Standard

SIGABRT	abort & core dump
SIGALRM	alarm rings
SIGFPE	floating point error
SIGHUP	hang up (phone line)
SIGILL	illegal instruction
SIGINT	interrupt (by user)
SIGKILL	kill process unconditionally
SIGPIPE	wrote to pipe with no reader
SIGSEGV	segmentation violation
SIGTERM	please terminate gracefully
*SIGUSR*1	userdefined #1
*SIGUSR*2	userdefined #2

Ein Prozeß bereitet sich darauf vor, ein Signal zu verarbeiten, indem er dem BS mitteilt, welche Routine (signal handler) beim Eintreffen eines Signals ausgeführt werden soll.
signal(SIGNAL, handler)
Bei Eintreffen eines Signals (BS führt *kill* eines anderen Prozesses aus) unterbricht das BS den signalisierten Prozeß und führt **handler** aus. (Falls nichts vereinbart wurde, wird der Prozeß abgebrochen.) Danach fährt der Prozeß an der alten Stelle fort. Signale können auch an die gesamte Prozeßgruppe gleichzeitig gesendet werden.

3.3 Implementierung von Prozessen in *UNIX*

Abstrakt:
Alle Verwaltungsinformationen über einen Prozeß sind im *Prozeßleitblock* (process control block **pcb**) zusammengefaßt (z.B. Segmentgrenzen, Programmtext, Daten, Stack, ...). In *UNIX* liegt die Verwaltungsinformation (Prozeßumgebung, process context) in zwei Strukturen vor.

1. Prozeßtabelle (process table)
   (Eintrag in) HSP-residente Tabelle aller Prozesse

2. Benutzerstruktur (user structure, u-dot structure)
   wird mit dem Prozeß ausgelagert.

3.4 Information in der Prozeßtabelle

(was man braucht, auch wenn der Prozeß ausgelagert ist)

1. Parameter der Prozeßbehandlung (scheduling parameters)
   Priorität, CPU-Zeit und Blockzeit im letzten Zeitquantum.

2. Speicherabbild
   Zeiger auf Programmtext, Daten, Stacksegmente oder deren Seitentabellen oder die externen Adressen, falls Prozeß ausgelagert.

3. Signale
   Welche Signale werden ignoriert, temporär ausgesperrt, von Verarbeitern behandelt oder gerade gesendet ?

4. Verschiedenes
   Prozeßstatus (blocked, running, ready, ...)
   PID, PID des Vaters, UserID, GroupID, Zeit bis zum nächsten Alarm, Ereignis (event), auf das gewartet wird.

3.5 User structure

(Informationen, die nur benötigt werden, wenn der Prozeß im Hauptspeicher und lauffähig ist)

1. Registerinhalte (Prozessor und Coprozessor)
   gehören zum Prozeßstatus, also zum Prozeßkontext. Sie werden gesichert, wenn der Prozeß durch *kerneltrap* unterbrochen wird, z.B. zur Auslagerung.

2. Systemaufrufstatus
   die Parameter eines laufenden Systemaufrufs werden damit gespeichert.

3. Dateibeschreibungstabelle (descriptor table)
   wird hierdurch für Systemaufrufe zugänglich, d.h. $read(d, buf, nbytes)$; benutzt d als Index in diese Tabelle, um z.B. die Geräteadresse zu finden, von der zu lesen ist.

4. Abrechnung (accounting)
   Zeiger auf Tabelle mit Abrechnungsinformation ($\rightarrow$ getrusage), Zeit, Speicherplatz.
   Auch Begrenzungen, die eingehalten werden müssen.

5. Kernstapel (kernel stack)
   Der BS-Kern benutzt seinen eigenen Stack, um Systemaufrufe dieses Prozesses zu bearbeiten.

3.6 Realisierung von $fork()$

Nach dem *kernel trap* sucht das BS einen freien Eintrag in der Prozeßtabelle und kopiert (fast) den ganzen Eintrag des Vaterprozesses. Wichtigste Ausnahme:
Es werden neue, gleichgroße Segmente für Text, Daten, Stack und u-structure angelegt und deren Grenzen gespeichert. Dann wird der Inhalt dieser Segmente vom Vater zum Kind kopiert. Mögliche Optimierung ($\rightarrow$MACH):
copy_on_write, d.h. zunächst wird auf das Segment des Vaters verwiesen und kopiert wird erst, falls das Segment (durch Schreiben) verändert wird.

3.7 Leichte Prozesse

(threads of control) $\rightarrow$MACH, Solaris 2.2

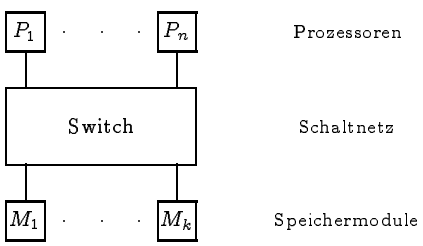


Abbildung 1: Multiprozessorsystem mit gemeinsamen Hauptspeicher (z.B. Solbourne 704)

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

Beispiel *sortiere*(divide & conquer):

```
Sortiere(A[1..n],n)
[ sortiere Array A der Länge n ]
{
```

1. Trivialfall. if (n $\leq$ 1) return;

2. Teile. Errechne k und verteile A intern um, so daß A[i] $\leq$ A[j], $\forall\, i \leq k < j$

3. Herrsche. Sortiere(A[1..k],k);
Sortiere(A[k+1..n],n-k);

4. Ende. return;

}

Die beiden rekursiven Aufrufe im Schritt 3 können parallel ausgeführt werden. Würde für den zweiten Aufruf ein neuer $UNIX$-Prozeß erzeugt werden, so hätten wir unnötigen Aufwand (overhead). Kopie des PCB, neue Segmente (text, data, stack) samt Seitentabellen. Text, Daten und PCB-Einträge bleiben aber gleich. Der zweite Sortierprozeß braucht lediglich einen eigenen Stack und einen sehr reduzierten PCB, der zur Identifikation des Prozesses dient und in dem bei Unterbrechung der Registersatz gespeichert werden kann. Wir nennen dies einen **leichten Prozeß**, da er wegen des kleineren Kontexts viel schneller gestartet werden kann (ca. 10x schneller) als ein $UNIX$-Prozeß. Threads haben keinen eigenen geschützten Datenbereich, nicht einmal ihr Stacksegment ist geschützt. Threads wurden in *POSIX* (4.3) standardisiert (*pthreads* $\rightarrow$ 12.1.6). Ein defacto Standard sind ferner $C-Threads$, wie in MACH (bzw. OSF/1). C-Threads sind weniger komfortabel, aber sehr portabel und effizient.

3.8 C-Thread „System Aufrufe“

1. *cthread_fork*($func, arg$)
   /* Erzeugt neuen *thread*, der die (Haupt)Funktion $func(arg)$ ausführt. Das Resultat ist die ID des neuen *threads*. Nur ein Argument arg ist möglich; mehrere müssen gepackt werden (und $func$ muß das Auspacken besorgen). */

2. *any_t cthread_join*(ID)
   cthread_t ID;
   /* Wartet auf das Ende des *threads* mit ID. Das Resultat des *join* ist das Resultat der Hauptfunktion von *thread* ID. Join (oder *detach* s.u.) auf jedes Kind muß sein. **Warnung:** Resultat darf nicht auf Stack des Kindes liegen. */

3. *cthread_detach*()
   /* Deklariert, daß niemand auf das Ergebnis des laufenden *threads* warten wird. */

4. *cthread_exit*(res)
   any_t res;
   /* Terminiert den laufenden *thread*; *res* ist dann das Resultat des *thread*. Ein *cthread_exit* wird implizit am Ende der Hauptfunktion des *threads* ausgeführt. */

5. *cthread_t cthread_self*()
   /* Liefert die ID des laufenden *threads*. */

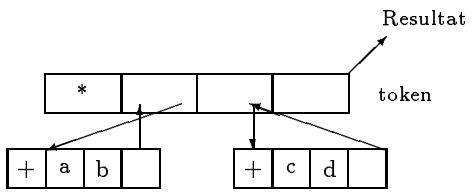
6. *cthread_yield*()
   /* Nimmt freiwillig den laufenden *thread* von der CPU herunter (Benutzerscheduling). Nötig, falls C-Threads ohne *preemptive scheduling* implementiert. */

7. *void cthread_set_data*(t, *data*)
   c_thread t;
   any_t data;
   /* Assoziiert eine Größe *data* (von der Länge eines Wortes) mit einem thread *t* */

8. *any_t cthread_data*(t)
   cthread_t t;
   /* Liest (returniert) die mit t assoziierte Größe */

Anmerkung:
Die leichtesten Prozesse sind sogenannte Datenflußmarken (dataflow tokens). Sie bestehen nur aus Funktions- und Argumentzeiger, sowie aus einem Zeiger auf den Platz, an dem das Resultat abzulegen ist. Sie können nur von speziellen Datenflußmaschinen [1] verarbeitet werden (außer durch Simulation), ermöglichen aber extrem feinkörnigen Parallelismus.

Beispiel: $(a + b) * (c + d)$



3.9 Scheduling (Prozeßlaufplanung)

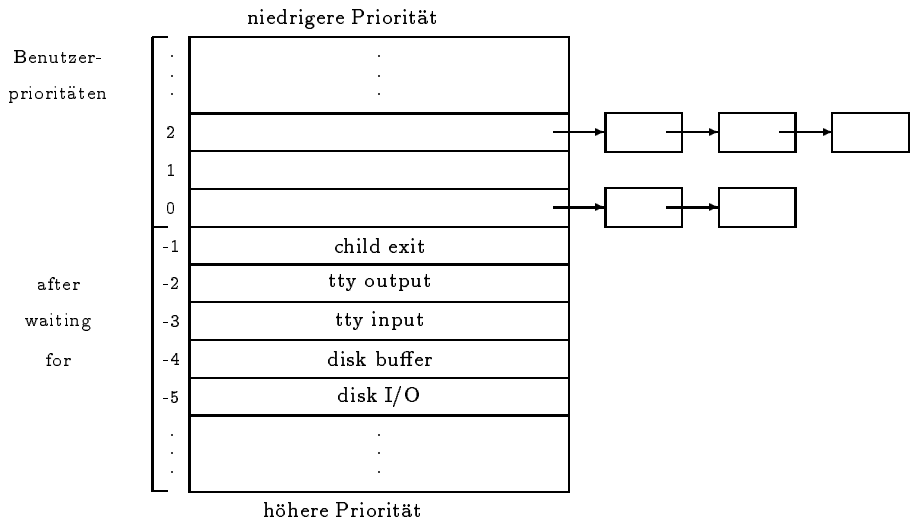
In *UNIX* optimiert auf gute Leistung für interaktive Prozesse.

- obere Ebene (high level): swapper
  lagert ganze Prozesse im HSP ein und aus, unter Benutzung von Sekundärspeicher (Platten).

- untere Ebene (low level):
  Auswahl des nächsten vom Prozessor zu bearbeitenden Prozesses aus der Warteschlange lauffähiger Prozesse (run queue).

UNIX hat eine Mehrschichtenwarteschlange (multiple run queues).

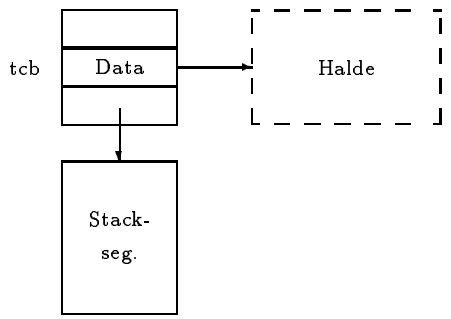
[1] z.B. MIT Monsoon



Scheduler wählt ersten Prozeß höchster Stufe. Der läuft ein Quantum (z.B. 5 $\cdot \frac{1}{60}$ sec), dann wird er wieder hinten in die Queue eingehängt. Einmal pro Sekunde werden die Prioritäten neu berechnet aus:

$$New = base + \frac{CPUusage}{2}$$

Base = 0, außer falls durch $nice()$ heraufgesetzt.



3.10 Berechnung der Priorität eines Prozesses

Die Priorität (im user mode) wird alle 4 Uhrtakte nach folgender Formel berechnet:

$$p_usrpri = PUSER + \frac{p_cpu}{4} + 2 \cdot p_nice$$

$PUSER$ ist die Basispriorität für den Usermode. 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.

$$p_cpu = \frac{2 \cdot load}{2 \cdot load + 1} \cdot p_cpu + p_nice$$

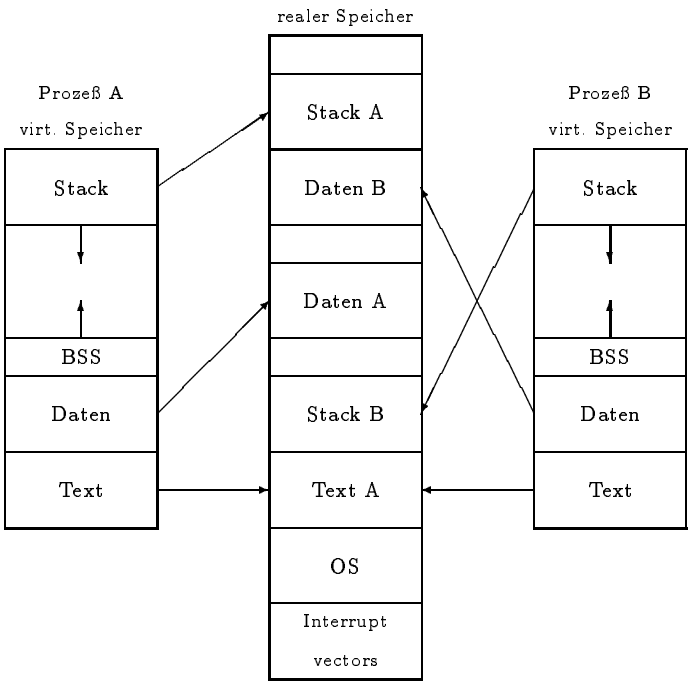
$load$ ist die durchschnittliche Länge der run queue während der letzten Minute. Bei zwei Prozessen im System wird dadurch nach fünf Sekunden etwa 90% des vergangenen CPU-Verbrauchs ausgefiltert.

4 Das *UNIX* Speichermodell

Jeder Prozeß hat ein(en)

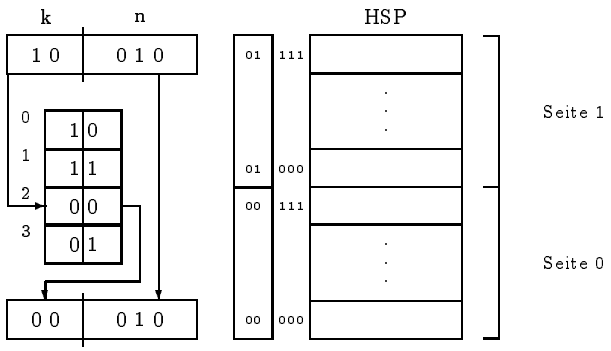
- text segment (mit ausführbarem Code)
- data segment (Datenspeicher) mit zwei Teilen
  - BSS uninitialisierte Daten (globale uninitialisierte *C* Variablen, die alle implizit mit 0 vorbesetzt sind.)
  - initialisierte Daten
- Stack (Stapelspeicher, dynamic memory)

Sowohl *Stack*, als auch *Datensegment* können wachsen. Der Stack wird implizit vergrößert, falls die Hardware eine Überschreitung der Stackgrenze erkennt.
Das *Datensegment* kann mittels system call vergrößert werden (brk, sbrk). Diese werden von *malloc* (C Bibliothek) benutzt.
Das *Textsegment* ist statisch fest, da der Programmcode nicht dynamisch verändert werden kann. Es kann von verschiedenen Inkarnationen des Programms gemeinsam benutzt werden (z.B. Editor). Alle Segmente sind im Speicher verschiebbar. Wir unterscheiden zwischen dem **virtuellen Speicherbild** des Programmierers und dem **realen Speicherbild** zur Laufzeit im Hauptspeicher. Die Umsetzung virtueller auf reale Adressen wird meist von der Hardware unterstützt. Die Umsetzung kann mehrstufig sein, z.B. virtuelle Segmente → reale Segmente mit virtuellen Seiten → realen Seiten.



Segmentverschiebung durch Addition der Segmentbasisadresse zur logischen Speicheradresse.
Vorteile der Segmentierung : logisch Zusammengehöriges wird auch so gesehen; Segmente können dynamisch wachsen. Wachsen geschieht, wenn nötig, durch Verschieben des Segments in eine größere Lücke im Hauptspeicher. Nur die Basisadresse ändert sich, nicht aber die virtuelle Adresse.

Problem:
Zerfaserung (externe Fragmentierung) des Hauptspeichers, da kleine Lücken ungenutzt bleiben. Deshalb werden Segmente in kleinere Einheiten aufgebrochen, die sog. Seiten (**pages**). Seiten sind fixe Allokationsblöcke. Jetzt kann jede freie Hauptspeicherseite benutzt werden, falls man die Adreßabbildung seitenweise individuell machen kann. Segmente werden dann nicht zusammenhängend realisiert (non contiguous). Der virtuell zusammenhängende Speicherbereich wird mittels einer Umsetzfunktion $r = m(v)$ in eine verstreute Realisierung abgebildet.
Aus Effizienzgründen geschieht dies nach folgendem Prinzip:
(wenn möglich in der Hardware) MMU (memory management unit). Seiten haben die Größe 2^n und es gibt 2^k Seiten, so daß $n+k = a$ die Adreßlänge ist. Die niedrigsten Bits der virtuellen Adresse v sind gleich den niedrigsten Bits der realen Adresse r. Die hohen k Bit adressieren die Seite $s = p(k)$. p ist über eine **Seitentabelle** implementiert.



4.1 Virtueller Speicher

Mittels der Speicherabbildungen ergibt sich die Möglichkeit, virtuell (virtually = eigentlich, im Prinzip) mehr HSP zu benutzen, als real vorhanden ist. Das funktioniert, falls zu jedem Zeitpunkt nur höchstens soviel HSP tatsächlich aktiv benötigt wird, wie real vorhanden ist. Nicht benötigte Seiten werden auf Platte ausgelagert und bei Bedarf eingelagert. Virtueller Speicher wird benutzertransparent implementiert und stellt somit eine neue Abstraktionsebene dar (großer Adreßraum).
Virtueller Speicher ist effizient, falls sich ein „Fenster der Aktivität " langsam durch den Gesamttext und die Gesamtdaten durchschiebt. Das Programm muß eine gewisse Lokalität aufweisen.

4.2 Swapping (Prozeßtauschverfahren)

Man kann auf Prozeßebene arbeiten und virtuell mehr Prozesse gleichzeitig ablaufen lassen, als zusammen in den HSP passen. Dies geschieht durch *swapping*, d.h. das BS lagert Prozesse mit niedrigerer Priorität temporär auf Platte aus, um Platz zu schaffen (Voraussetzung: relocatable code; Adressen relativ zu verschiebbarer Basis).
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 eine höhere Priorität hat.

Mögliche Prioritätskriterien:

- Hohe Priorität
  - schon lange nicht mehr gerechnet
  - interaktiver Prozeß
  - erst ganz kurz im HSP
- Niedrige Priorität
  - lange gerechnet
  - blockiert (wartet auf Ereignis)
  - Hintergrundprozeß
  - benutzt viel Speicher

4.3 Paging (Seitentauschverfahren)

UNIX hatte zunächst nur *swapping*; ab 3BSD dann auch *paging*, wegen der immer größer werdenden Programme. *Paging* ist allgemeiner, da im Prinzip auch alle Seiten eines Prozesses ausgelagert werden können; normalerweise werden nur die Seiten (außerhalb des Aktivitätsfensters) ausgelagert, die schon lange nicht mehr benötigt wurden.
In *UNIX* existieren beide Verfahren nebeneinander:

- Prozeß 0 ist der **swapper**
- Prozeß 1 ist der **init**
- Prozeß 2 ist der **page daemon**

4.4 Der *UNIX* Seitentauschalgorithmus

Alle 250ms wird der *page daemon* geweckt. Der sieht nach, ob mehr als **lotsfree** Seiten frei sind (typisch 25% HSP). Falls nicht, werden entsprechend viele Seiten ausgelagert (paged out); falls ja, schläft der Dämon wieder ein.
Der Auslagerungsalgorithmus folgt dem *Einzeigeruhr*prinzip . Die Seiten werden reihum durchgegangen und das „Benutztbit“ wird zurückgesetzt.
Beim zweiten Durchgang werden alle Seiten freigesetzt (in die Freispeicherliste eingekettet), die seit dem ersten Durchgang nicht benutzt wurden. „Schmutzige“ Seiten (auf die geschrieben wurden) müssen auf Platte zurückgeschrieben werden. Einige Maschinen unterstützen „Benutztbits“ in der Hardware. Auf der **VAX** wird das wie folgt simuliert:
Die Seite wird in der Seitentabelle als ungültig markiert und ein Softwaretrap folgt, der dann das „Benutztbit“ setzt und den Eintrag in Ordnung bringt.
Variationen des Uhralgorithmus:

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

Falls nicht genug HSP freigemacht werden kann, werden ganze Prozesse ausgelagert, um wildes Seitentauschen zu verhindern (**thrashing**).
Erst werden untätige Prozesse ausgelagert ($\geq$ 20sec untätig), dann derjenige der vier größten, der am längsten gerechnet hat.

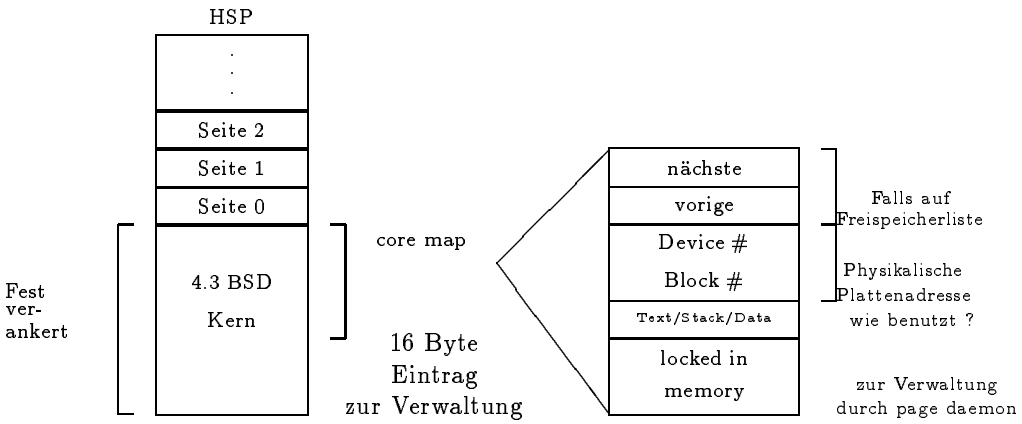


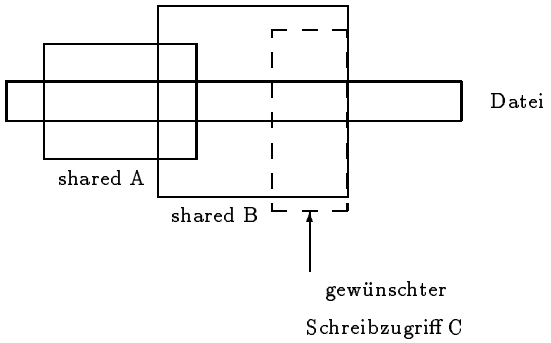
Abbildung 2: Seiteneinteilung im HSP

5 Das *UNIX* Dateisystem

Die Dateien sind in einem Datenbaum organisiert. Der Zugriff erfolgt über **absolute** Pfadnamen (z.B. /home/amrhein/Uebungen/Ueb4) oder **relativ** zur augenblicklichen Position (z.B. Uebungen/Ueb4 für Benutzer amrhein). Es können Dateien eingerichtet werden, die auf andere verweisen (**links**). **Soft links** über Gerätegrenzen hinweg, **hard links** innerhalb eines Geräts.
Ein Dateibaum von einem Gerät kann in den Dateibaum eines anderen Geräts eingetragen und damit logisch eingegliedert werden (**mounting**). Der Benutzer merkt das Überschreiten der Gerätegrenzen nicht und arbeitet logisch mit dem Dateibaum.
Nach *POSIX* können Dateien für Exklusivzugriffe reserviert werden (**locking**). Dies ist zur Synchronisation paralleler Dateizugriffe nötig. Es gibt zwei Arten von *locks*:

- *shared*: Reservierung für Lesezugriff
- *exclusive*: Reservierung für Schreibzugriff

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



Reservierungen (locks) sind nötig, um bei Parallelzugriff Inkonsistenzen und andere semantische Fehler zu vermeiden.

Gegeben sei eine Datei D mit Datenbereich C und zwei Prozessen P_1 und P_2.

- Fehlertyp *Ausfall*

  1. P_1 liest Datum $B_1 \leftarrow C$
  2. P_2 liest Datum $B_2 \leftarrow C$
  3. P_1 modifiziert B_1 und schreibt es zurück $B_1^{'} \rightarrow C$
  4. P_2 modifiziert B_2 auf andere Art und schreibt es zurück $B_2^{'} \rightarrow C$

  Resultat: Die Aktion von P_1 fällt unter den Tisch. Falls zufällig (3) und (4) vertauscht ausgeführt werden, fällt die Aktion P_2 weg. Weder P_1 noch P_2 bemerken dies.
  Spezialfälle: Selbst wenn P_1 und P_2 die gleiche Modifikation ausführen, kann es Probleme geben: z.B. $B = B + 1$; C wird nur **einmal** inkrementiert. Exklusivzugriff ist nur dann nicht nötig, wenn P_1 und P_2 das Datum mit derselben neuen Konstanten überschreiben, z.B. $B = 0$;.

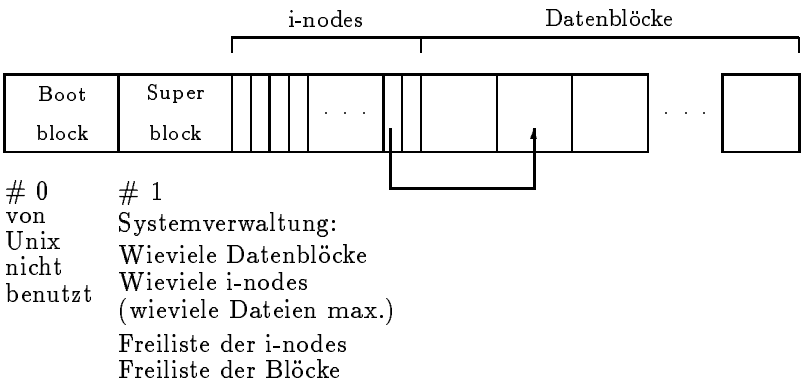
- Fehlertyp *Inkonsistenz*
  Gegeben sei ein Wort $W = B_1 B_0$

1. P_1 liest Byte B_0
2. P_2 überschreibt Byte B_0
3. P_2 überschreibt Byte B_1
4. P_1 liest Byte B_1

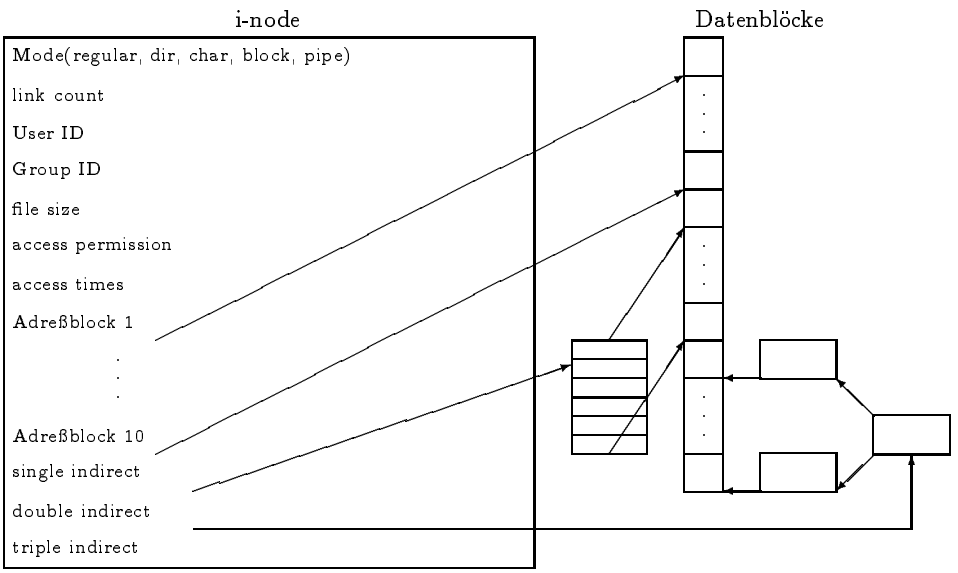
P_1 liest ein inkonsistentes Datum. Deshalb muß schon bei einem einzigen Schreiber in der Gegenwart von Lesern gesperrt werden.
Ausnahme: Falls nur kleine Einheiten **atomar** geschrieben werden (Synchronisation durch Hardware). Z.B. Hardware garantiert atomares Schreiben eines 32bit Wortes.

5.1 Implementierung des Dateisystems

Plattenorganisation:



„Ein $I-node$ ist für eine Datei, was der PCB für einen Prozeß ist."



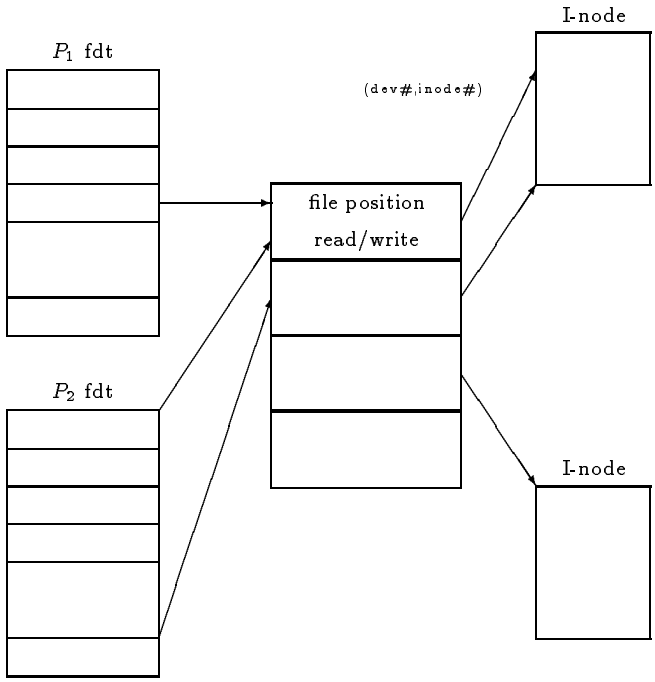
I-nodes existieren sowohl auf Platte, als auch im HSP, wenn die Datei offen ist. Die Abb. zeigt die Platten I-nodes. HSP I-nodes haben zusätzliche Felder, z.B. die Position des zugehörigen Platten I-nodes und ein „dirty bit".

Beim Dateizugriff errechnet das Dateisystem den gefragten Block aus der Dateiposition und der Blockgröße. Mit 32bit Worten können 4 GB adressiert werden. Mit 1 kB Blöcken und 32bit Blockadressen können über 16 GB gespeichert werden. Ein Katalog (*directory*) ist eine Datei mit einer Tabelle aus Einträgen (Name, I-node Index). Um die 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 (z.B. /home/amrhein/Uebungen) beginnt die Suche in I-node #2, dem $root - Katalog$.

5.2 Lesen einer Datei

mit $read(fd, buffer, nbytes)$

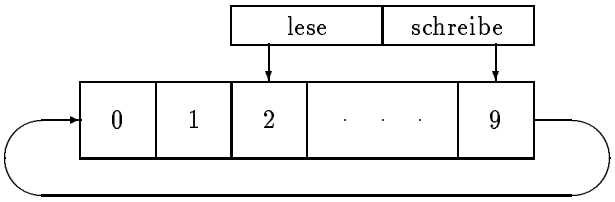
Man muß vom Filedescriptor fd zum I-node kommen. fd ist ein Index in die Tabelle der Descriptoren (fdt) in der Benutzerstruktur (u-structure). Man könnte in der fdt direkt den I-node Index (Adresse) eintragen, tut dies aber aus folgendem Grund nicht:
Man könnte es ermöglichen, daß verschiedene Prozesse verschiedene Dateipositionen halten, daß aber verwandte Prozesse (z.B. shell mit zwei Kindern) sich eine Dateiposition teilen (shell legt Datei an, Kind 1 schreibt zuerst, Kind 2 schreibt danach). Dazu wird die Beschreibungstabelle offener Dateien eingeführt (**o**pen **f**ile **d**escriptor **t**able $ofdt$).



Beim Aufruf von $open()$ wird ein neuer Eintrag in der $ofdt$ generiert und vom ersten freien Platz im $process\ fdt$ darauf verwiesen.
Zwei verschiedene Prozesse können dieselbe Datei an verschiedenen Stellen 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 der Datei verändert, sieht auch das Kind diese Veränderung (und umgekehrt). Dadurch können z.B. zwei Prozesse fortlaufende Einträge in einer Auftragsdatei abwechselnd lesen (und verarbeiten).

5.3 Pipes

Pipes werden mittels des Dateisystems realisiert. Nach dem Öffnen können sie logisch wie eine Datei geschrieben bzw. gelesen werden. Es existiert ein Pufferspeicher, der durch die zehn direkten Blöcke der Datei realisiert wird, die logisch als Ringpuffer organisiert werden. Zudem gibt es einen Lese- und einen Schreibzeiger. Lese- und Schreibzeiger überholen sich nicht, sondern Produzent und Konsument werden zuvor blockiert (und wieder aufgeweckt, wenn wieder Platz ist).



Es können mehrere Prozesse in eine *pipe* schreiben oder aus ihr lesen. Es gibt benannte (**named**) pipes und unbenannte (**unnamed**) pipes zur Kommunikation zwischen verwandten bzw. nicht verwandten Prozessen.

Systemaufrufe

- int fds[2];
  pipe(fds);
  /* liefert eine unbenannte *pipe* mittels der beiden Deskriptoren $fds[0]$ fürs Lesen und $fds[1]$ fürs Schreiben */

- mkfifo(path,access_mode)
  char *path; mode_t access_mode;
  /* liefert eine *pipe* mit dem Namen *path* */

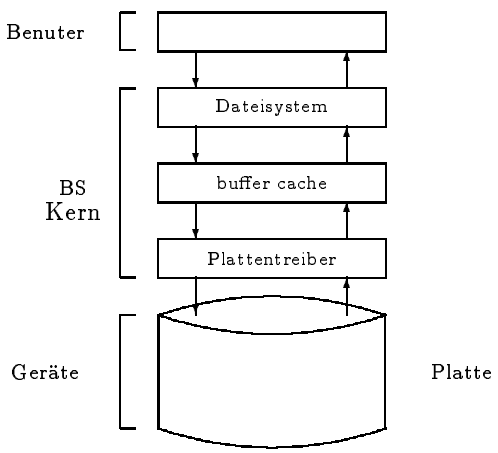
5.4 Input/Output

Ein-/Ausgabegeräte werden logisch als spezielle Datei behandelt, da sie beschrieben/gelesen werden können (üblicherweise in */dev*). Z.B. *cp file /dev/lp*.
Es werden zwei Dateitypen unterschieden:

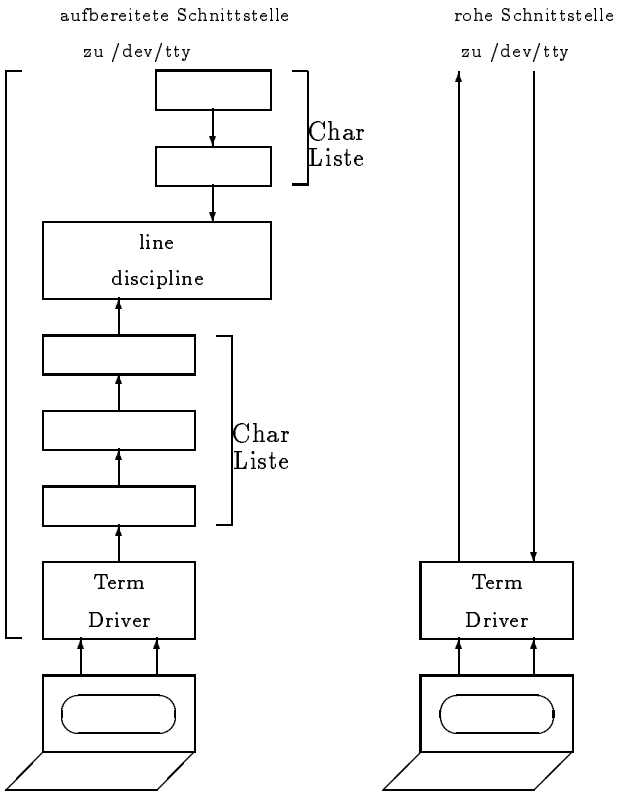
1. Block special file
   Datei ist eine Folge nummerierter Blöcke. Jeder Block kann individuell adressiert und gelesen/beschrieben werden (Bsp. Platte).

2. Character special files
   Datei ist Strom von Zeichen mit sequentiellem Zugriff (Bsp. printer, tty, /dev/null).

Die Gerätesteuerung selbst wird von Gerätetreibern erledigt (*device drivers*). Geräteeigenheiten werden unter Standardschnittstellen verborgen ($\rightarrow SCSI$). 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 (*sync*).



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



6 Interprozeßkommunikation (IPC)

6.1 Zugriffskonflikte (race conditions)

Codesequenzen, während deren Ausführung es zu Zugriffskonflikten kommen kann, heißen **kritische Abschnitte** (kA, critical sections). Die Ausführung der kAe muß **synchronisiert** werden.
Im allgemeinen befindet sich ein Programm nur selten in einem kritischen Abschnitt; der meiste Code ist unkritisch. Wir betrachten nun Techniken zum Erreichen **wechselseitigen Ausschlusses** (**mut**ual **ex**clusion) zweier oder mehrerer Prozesse in kritischen Abschnitten. Für eine gute Lösung müssen vier Bedingungen erfüllt sein:

1. Korrektheit:
   Nie dürfen zwei Prozesse gleichzeitig in einem kA 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 kA läuft, darf einen anderen Prozeß aufhalten (blockieren)

4. Fairneß:
   Kein Prozeß muß je ewig darauf warten, einen kA ausführen zu dürfen.

6.2 Wechselseitiger Ausschluß mit aktivem Warten

(mutual exclusion with busy waiting)

6.2.1 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 kA exklusiv bearbeiten.
Nachteil: Interrupts sollten nur vom BS deaktiviert werden können, z.B. wegen Mißbrauch.
Funktioniert nicht bei Mehrprozessorsystemen (verletzt 2.).

6.2.2 Lock Variables (Schloßvariablen)

Schloßvariable: 1 Bit, entweder $s == 0$ oder $s == 1$;
Vor Eintritt in den kA testet der Prozeß s. Falls $s == 0$, so wird $s = 1$ gesetzt und fortgefahren. Falls $s == 1$ wird aktiv gewartet: *while* s;
Lösung ist so falsch, da das Problem nur verschoben ist:
Mehrere Prozesse schreiben und lesen s gleichzeitig. Eine echte Lösung erhalten wir durch einen nichtunterbrechbaren Befehl „Test und Setze“, der selbst nur exklusiv ausgeführt werden kann.

Aus Vergleichsgründen betrachten wir zunächst zwei Lösungen des $Mutex$ Problems in der Software. Auf Systemen mit gemeinsamen HSP sind diese Lösungen zu kompliziert und ineffizient.

6.2.3 Strikte Abwechslung (strict alternation)

Gegeben seien zwei Prozesse P_0 und P_1. Eine gemeinsame Variable `turn` wird 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) /* wait */;
  critical_section();             critical_section();
  turn=1;                         turn=0;
  rest();                         rest();
}                               }
```

Problem:
Nachfolgeprozeß von P_0 (bzw. P_1) ist hart codiert durch Setzen von `turn`.
Verletzt Effizienzbedingung: Falls P_0 schnell ist, muß P_0 u.U. warten, bis der langsame P_1 seinen unkritischen Rest beendet hat, bevor P_0 wieder in den kA kann.

6.2.4 Peterson's Lösung

(in Software)
Gegeben seien P_0 und P_1. Das array `interested[2]` zeigt an, welcher Prozeß Interesse hat, in den kA einzutreten. Bei einem Konflikt entscheidet die gemeinsame Variable `turn`, wer daran ist zu warten. Es wartet, wer `turn` zuletzt mit seiner Prozeßnummer gesetzt hat. Die Prozesse führen folgende Prozeduren zu Beginn und am Ende des kA aus:

```
void enter_region(process) /* two processes */
int process;
{ int other = 1-process; /* other process */
  interested[process]=1;
  turn=process;
  while(turn==process && interested[other] /* wait */;
}

void leave_region(process)
int process;
{ interested[process]=0; }
```

Falls P_1 und P_0 *enter_region*() gleichzeitig ausführen, so wird `turn` nacheinander mit zwei verschiedenen Werten beschrieben, und der letzte Wert bleibt. (Der **Bus Arbiter** teilt den Speicherbus jeweils nur **einem** Prozessor zu.) Der letzte Prozessor wartet somit.
Korrektheit des Protokolls wird bewiesen, indem man alle möglichen Verschachtelungen von Befehlssequenzen in zwei Aufrufen von *enter_region*() durchspielt.

6.3 Die TSL Instruktion (test and set lock)

(Teste und Setze)
Moderne Prozessoren sind durch eine TSL Instruktion auf die Verwendung in Multiprozessorsystemen vorbereitet. Die Instruktion

```
  tsl register, lock
```

liest das Speicherwort `lock` in das Register `register` und speichert anschließend 1 nach `lock`. Die Instruktion läuft **atomar** ab, d.h. sie kann zwischendurch nicht unterbrochen werden (z.B. wird der Speicherbus während der Ausführung gesperrt, so daß `lock` nicht verändert oder gelesen werden kann, bevor es zurückgeschrieben wurde).

Nehmen wir an, der TSL Befehl sei in *C* als Prozedur

```
int TSL(lock)
```

aufrufbar. Dann erhalten wir:

```
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 aus *mutex_lock*() und *mutex_unlock*() mit einer Lock-Variablen, deren Adresse übergeben wird.
Die **C Thread** Schnittstelle enthält die Funktionen

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

und

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

mutex_lock() blockiert, bis `m` geschlossen werden kann, *mutex_unlock*() öffnet `m`. Von mehreren gleichzeitigen Aufrufen zu *mutex_lock*() ist maximal einer erfolgreich. Hiermit lassen sich nun Zugriffskonflikte bereinigen.
Wichtig ist daneben aber auch, 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*().

6.4 Sleep, Wakeup

Hierzu werden zwei neue Systemaufrufe eingeführt. *sleep*() suspendiert den Aufrufer; er wird blockiert, ohne einen Prozessor zu belasten.
wakeup(*pid*) weckt den Prozeß mit der ID *pid* auf (wenn der Prozeß nicht schläft, passiert idR nichts).
Alternativ wird eine Ereignisvariable **event** und die Aufrufe *wait*(*event*) bzw. *signal*(*event*) eingeführt. Sie realisieren das Warten auf das Ereignis, das logisch zu **event** gehört. In der Implementierung werden die Leitblöcke wartender Prozesse in einer in **event** verankerten Warteschlange gehalten. *signal*(*event*) löst einen oder alle Prozesse aus der Warteschlange 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 in 1. Näherung die Lösung:

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

Diese Version ist fehlerhaft, da der Zugriff auf die gemeinsame Variable **count** nicht synchronisiert wurde.
Zweite Näherung der Lösung:

```
mutex_t lock;
void producer()                         void consumer()
{ int item;                             { int item;
  while(1)                                while(1)
  { produce(&itme);                       { mutex_lock(lock);
    mutex_lock(lock)                        if(count==0)
    if(count==N)                            { mutex_unlock(lock);
    { mutex_unlock(lock);                     sleep();
      sleep();                                mutex_lock(lock);
      mutex_lock(lock);                     }
    }                                       remove(&item);
    enter(item);                            count--;
    count++;                                if(count==N-1)
    if(count==1)                              wakeup(producer);
      wakeup(consumer);                     mutex_unlock(lock);
    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.

Anmerkung:
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 bleibt fehlerhaft, bzw. verletzt Bedingung 2.
Es gibt zwei Lösungen:

1. Man kann das Aufschließen und Einschlafen in einem ununterbrechbaren Befehl kombinieren (oder man kann dafür sorgen, daß erst aufgeschlossen wird, nachdem der Prozeß schläft).

2. Dijkstra's **Semaphore** zählen die gesendeten Signale, damit keines verloren geht.

6.5 C Thread Conditions

Sie erlauben es, Zustandsänderungen von gemeinsamen Speicherobjekten zu signalisieren bzw. auf sie zu warten. Jedes Speicherobjekt ist durch ein *mutex* geschützt. Ein *thread* erwirbt das Zugriffsrecht, untersucht das Objekt und findet, daß er auf eine Bedingung warten muß. Das Zugriffsrecht muß abgetreten werden, damit ein anderer *thread* das Objekt verändern kann, bis die Bedingung erfüllt ist. Nach Empfangen des Signals, die Bedingung sei nun erfüllt, wacht der blockierte *thread* mit dem Zugriffsrecht auf, d.h. im Zustand, in dem er zuletzt war. Er muß jedoch die Bedingung erneut prüfen, denn ein Dritter könnte nach Abtreten des *locks* durch den Sender des Signals inzwischen das Objekt schon wieder verändert haben.

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

6.6 Semaphore

Im ursprünglichem Sinn reguliert ein **Semaphore** die Anzahl der Prozesse, die sich in einem kA befinden. Semaphore wird mit dieser Anzahl initialisiert. Es gibt zwei Operationen:

1. **P(s)** „Passiere“ den Eingang zum kA (=$Down(s)$)
2. **V(s)** „Verlasse“ den kA (=$Up(s)$)

$P(s)$ 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.

Anmerkung:
Ein Spezialfall sind binäre Semaphore, initialisiert zu 1. Auf diesen sind $P(s)$ und $V(s)$ äquivalent zu *mutex_lock*() und *mutex_unlock*().

Auf einem Einzelprozessor kann die Ungeteiltheit der Semaphoreoperationen durch Ausblenden der Unterbrechungen erzielt werden (da Operation kurzzeitig abläuft). Auf Multiprozessoren muß das Semaphore zusätzlich durch ein *mutex* geschützt werden.
Allgemein regelt ein Semaphore die Verteilung von k Exemplaren eines beschränkten Betriebsmittels. $P(s) = Down(s)$ vergibt ein Exemplar, $V(s) = Up(s)$ gibt ein Exemplar zurück. Die Verallgemeinerung auf k Exemplare mit $Down(s, k)$ und $Up(s, k)$ ist offensichtlich.

Vierte Lösung (mit Semaphoren)
(Dritte Lösung = zweite Lösung mit cthread condition_wait())

```
typedef int* semaphore;
semaphore mutex,empty,full;

void producer()
{
  produce(&item);
  /* Ausserhalb des kA */
  down(empty);
  /* Ein leeres Fach zuteilen */
  down(mutex);
  enter(item);
  up(mutex);
  /* Atomares Eintragen
     von item */
  up(full);
  /* Volles Fach zuruekgeben */
}
```

```
void consumer()
{
  down(full);
  /* Volles Fach zuteilen */
  down(mutex);
  remove(&item);
  up(mutex);
  /* Atomares Austragen
     von item */
  up(empty);
  /* Leeres Fach zurueckgeben */
  consume(item);
   /* Ausserhalb des kA */
}
```

6.7 Monitore (Ausführungskontrolleure)

Monitore (Hoare, Binch Hansen) sind ein höheres Programmiersprachenkonzept, das das Schreiben nebenläufiger (paralleler) 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 jeweils nur ein einziger Prozeß Code aus dem Monitor ausführen darf (sich im Monitor aufhalten kann). Damit entfällt für den Programmierer die Notwendigkeit, einzelne Datenobjekte separat zu schützen. Es besteht aber die Gefahr der Ineffizienz, da evtl. 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 kAe.
Besser ist es, nur *enter*() und *remove*() zu schützen. Ferner benötigen wir zusätzlich die Möglichkeit der Kooperation, d.h. daß der Prozeß den Monitor aufgibt, um auf eine Bedingung zu warten und dann im Monitor wieder aufwacht. Das geschieht mit Bedingungsvariablen (conditions variables) und den Aufrufen

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

Bemerkung:
In C Threads wird durch `condition_wait(c,m)` der Monitorbegriff unterstützt; `m` ist das dem Monitor zugeordnete Mutex. Der aufgeweckte Prozeß blockiert aktiv außerhalb des Monitors bis er durch Erwerb des Mutex wieder eintreten darf.

Fünfte 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++;
   if(count==1) signal(nonempty);
  }

  void remove(y)
  int *y;
  {
   if(count==0) wait(nonempty);
   *y=B[count]; count--;
   if(count==N-1) signal(nonfull);
  }
} /* End Monitor */

void producer()
int item;
{
 while(1)
 {
  produce(&item);
  ProCon.enter(item);
 }
}

void consumer()
int item;
{
 while(1)
 {
  ProCon.remove(&item);
  consume(item);
 }
}
```

6.8 Nachrichtenaustausch (Message Passing)

Die bisher betrachteten Konzepte verlangen einen gemeinsamen HSP, der nicht immer gegeben ist. Nachrichtenaustausch funktioniert insbesondere auch über Netze, kann aber bei gemeinsamen HSP effizient realisiert werden. Außerdem ist dies ein nützliches Programmierkonzept. Es ist auch ein besserer Speicherschutz gewährleistet, wenn die kommunizierenden Prozesse in separatem HSP agieren und nur das BS Nachrichten zustellt.
MACH (OSF/1) unterstützt dieses Konzept mit **message ports** (Nachrichtenpforten) in jedem Prozeß.

Das Produzent/Konsument-Problem kann nun gelöst werden, in dem jedes „item“ in einer Nachricht vom Produzenten zum Konsumenten geschickt wird. Eine feste Anzahl von Nachrichtencontainern entspricht einem Puffer fester Größe. Der Konsument schickt dem Produzenten einen leeren Container zum Füllen. Nachrichten können in einem *Rendez−vous* synchron übermittelt werden oder asynchron, wobei sie in einem Briefkasten (mailbox) zwischengespeichert werden. (MACH ports sind vom BS verwaltete und geschützte Briefkästen.)

Sechste Lösung (mit messages)

```
mailbox_t fullbox, emptybox;
producer()                    consumer()
{                             {
 int item;                     int item;
 message_t m;                  message_t m;
 while(1)                      for(i=0;i<N;i++) /* initialize */
 {                              send(emptybox,&m);
  produce(&item);              while(1)
  /* receive empty             {
     message envelope */        receive(fullbox,&m);
  build(&m,item);               extract(&m,&item);
  /* Put item to                consume(&item);
     envelope */                }
  send(fullbox,&m);            }
  /* send to fullbox */
 }
}
```

6.9 Rendez-vous (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 (mailbox) ist ein Speicherplatz für Nachrichten. Ein *send*(*mailbox*) blockiert nur dann, wenn *mailbox* voll ist, ein *receive*() nur dann, wenn *mailbox* ganz leer ist. Nachrichten werden separat gespeichert (Nachrichtengrenzen werden gewährt).

6.10 Äquivalenz der Konzepte

Alle betrachteten Konzepte (Messages, Monitors, Semaphores, Mutex, Conditions) sind äquivalent, d.h. sie können wechselseitig implementiert werden. Die Programmiersprachen *Concurrent Pascal* und *Concurrent Euclid* bieten Monitore an. C Threads bieten *mutex/condition*, *UNIX* System V bietet *Messages* and *Semaphore*.

6.11 IPC in *UNIX*

Die einzige voll portable Synchronisationsoperation für Anwender ist die Realisierung von *mutex_lock*() durch das Erzeugen eines Lockfiles. **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 Lockfiles liegen, wenn ein Prozeß abstürzt oder ein Prozeß blockiert unnötig, wenn zufällig ein anderes File mit dem gleichen Namen wie das Lockfile existiert.

6.12 IPC in System V

Im System V gibt es gemeinsamen HSP zwischen den Prozessen, Nachrichten und Semaphore.

6.12.1 Semaphore in System V

Diese sind sehr kompliziert geraten. Ein Semaphore besteht aus:

- Dem Wert des Semaphores
- Der Prozeß ID des letzten Prozesses, der das Semaphore manipuliert hat
- Den ID's der Prozesse, die darauf warten, daß sich der Wert des Semaphores erhöht
- Den ID's der Prozesse, die darauf warten, daß der Wert des Semaphores NULL wird

Die Systemaufrufe sind:

semget
Erzeugen und Erhalten der Zugriffsberechtigung auf einer Semaphormenge

semctl
Verschiedene Kontrolloperationen (z.B. Schließen) auf der Menge.

semop
Werte der Semaphore verändern (entsprechen $P()$ und $V()$)

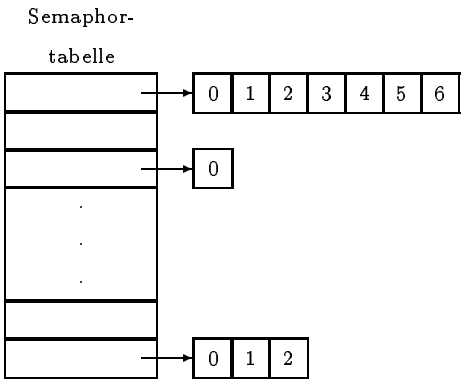


Abbildung 3: Semaphordatenstruktur

semget $id = semget(key, count, flag)$;
Erzeugt einen Eintrag in der Semaphortabelle in einem Array aus Semaphoren mit *count* Elementen. key ist ein benutzerdefinierter Name für das Semaphor und $flag$ spezifiziert die Zugriffsarten (z.B. was passiert, wenn kein Semaphor key existiert). Der Tabelleneintrag enthält auch die Zeiten der letzten *semop* und *semctl* Aufrufe.

semop $id = oldval = semop(id, oplist, count)$;
Verändert die Semaphorwerte. id ist der Deskriptor aus $semget()$, $oplist$ ist ein

Zeiger auf ein array der Länge *count* aus Semaphoroperationen. *oldval* ist der Wert des letzten Semaphore in der Liste vor der Operation. Jedes Element von *oplist* enthält:

- den Index des Semaphors im Array
- die Semaphoroperation als Konstante Op, die zum 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 $Op < 0$ ist, wird $Wert = Wert + Op$ ausgeführt, falls der neue Wert 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. Muß der Prozeß bei einer Teiloperation warten, so werden auf dem Semaphorarray die Orginalwerte wieder hergestellt. (Dies macht System V Semaphore kompliziert und aufwendig.)

Durch den *semctl*() Aufruf kann der Status eines Semaphors gelesen oder geändert werden, oder das Segment kann gelöscht werden.

UNIX System V enthält drei IPC Mechanismen:

1. Shared Memory
2. Messages
3. Semaphore

Den Implementierungen ist folgendes gemeinsam:

- Eine Systemtabelle enthält alle Instanzen der Mechanismen.
- Jeder Eintrag enthält einen benutzerdefinierten numerischen Schlüssel **key**.
- Ein **get** Systemaufruf durchsucht die Tabelle nach einem Eintrag mit Schlüssel **key**. Wenn kein Eintrag existiert kann (je nach gesetzten **flags**) ein neuer Eintrag erzeugt werden. **get** liefert einen Deskriptor, den die anderen Aufrufe benutzen.
- Jeder Eintrag enthält Statusinformationen (z.B. Zugriffsrechte, letzter zugreifender Prozeß, letzte Zugriffszeit).
- Jeder Mechanismus enthält einen **ctl** Aufruf, um den Status eines Eintrags abzufragen oder zu verändern oder den Eintrag zu löschen.

6.12.2 Shared Memory

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

Systemaufrufe:

shmget gibt Zugriff auf, oder erzeugt eine gemeinsame Speicherregion.
`shmid = shmget(key,size,flags);` sucht zuerst nach einer Region `key` in der Tabelle. Falls keine existiert und `flags = IPC_CREAT`, wird ein neuer Eintrag erzeugt.

shmat bildet eine Speicherregion in den virtuellen Adreßraum eines Prozesses ab. `virtaddr = shmat(shmid,addr,flags)`; bindet die Region `shmid` (möglichst) an der Adresse `addr` in den Adreßraum des Aufrufers ein. Der tatsächliche Ankerpunkt ist `virtaddr`. Ist `addr=0`, wählt das System, sonst muß der Aufrufer Konflikte (z.B. mit Stack) selbst vermeiden.

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

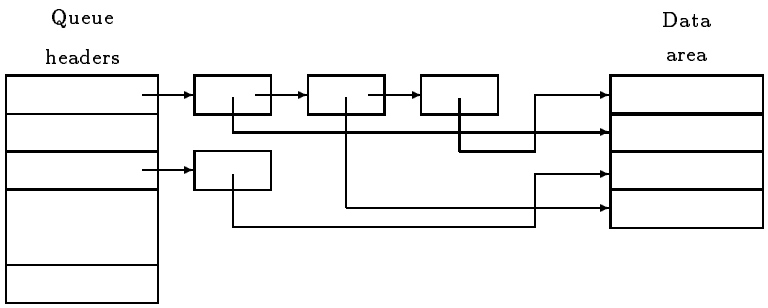
shmctl(id,cmd,shmstatbuf) verändert oder liest den Status einer Region. `shmstatbuf` enthält den neuen Status.

shmctl(id,IPC_RMID,0) gibt die Region zum Auflösen frei (das geschieht erst beim letzten *shmdt*()).

6.12.3 Messages

Nachrichten dienen zum Austausch formatierter Daten.
Systemaufrufe:

msgid = msgget(key,flag) liefert (evtl. erzeugt) einen Deskriptor für eine Nachrichtenwarteschlange (WS).



msgsnd(msgqid,msg,count,flag) sendet eine Nachricht `msg` der Länge `count` in die WS `msgqid`. `msg` selbst ist eine Struktur aus einem Integertyp und der Nachricht als *char array*. Es werden Prozesse geweckt, die auf eine Nachricht von diesem Typ warten.

count = msgrev(id,msg,maxcount,type,flags) empfängt eine Nachricht der maximalen Länge `maxcount` von WS `id`, die in der Struktur `msg` abgelegt werden soll. `count` ist die Zahl der tatsächlich übermittelten Bytes. Es wird die erste Nachricht übermittelt, wenn `type==0` ist, sonst die erste Nachricht vom Typ=`type`.

msgctl(id,cmd,mstatbuf) liest oder setzt den Status der WS `id`. `msgctl(msgid,IPC_RMID,0)` löscht eine WS.

6.13 Bekannte IPC Probleme

6.13.1 Essende Philosophen (Dijkstra)

Fünf Philosophen essen, trinken oder sind hungrig. Kritische Betriebsmittel sind die Eßutensilien. Es gibt fünf Teller mit Reis und zu beiden Seiten eines jeden Tellers je ein Eßstäbchen (chopstick). Man benötigt zwei Stäbchen, um essen zu können.

Problem:

1. Synchronisation: Die Betriebsmittel Teller und Stäbchen müssen zugeteilt werden.

2. Effizienz: Es soll kein Philosoph unnötig warten müssen (d.h. es soll nicht nur einer essen).

3. Kein Verhungern (starvation): Es soll nicht vorkommen können, daß ein Philosoph verhungert, weil die anderen sich gegen ihn verbünden und ihm immer Stäbchen blockieren.

4. Keine Verklemmungen: Die Lösung soll nicht zu Verklemmungen führen können.

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

```
#define LEFT  (i-1)%5
#define RIGHT (i+1)%5

int state[5];
semaphore mutex=1;
void philosopher(i)
int i;
{
 while(1)
 {
  think();
  take_stick(i);
  eat(i);
  put_stick(i);
 }
}

void take_stick(i)
int i;
{
 down(&mutex);
 state[i]=HUNGRY;
 test(i);
 up(&mutex);
 down(&s[i]);
}

void test(i)
int i;
{
 if(state[i]==HUNGRY
    && state[LEFT]!=EATING
    && state[RIGHT]!=EATING)
 {
  state[i]=EATING;
  up(&s[i]);
 }
}

void put_stick(i)
int i;
{
 down(&mutex);
 state[i]=THINKING;
```

```
 test(LEFT);
 test(RIGHT);
 up(&mutex);
}
```

Lösung mit condition/mutex:

```
#define LEFT  (i-1)%5
#define RIGHT (i+1)%5

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

void philosopher(i) /* wie vorher */

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

void put_stick(i)
int i;
{
 mutex_lock(table);
 state[i]=THINKING;
 mutex_unlock(table);
 condition_signal(sticks_available[LEFT]);
 condition_signal(sticks_available[RIGHT]);
}

int ready(i)
int i;
{
 if(state[LEFT]!=EATING && state[RIGHT]!=EATING) return(1);
 else return(0);
}
```

6.14 Prozeßablaufplan (process scheduling)

Das ist die Frage, welcher Prozeß wann wie lange läuft. Grundtechnik 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 Prozeß aus. Ein Prozeß befindet sich immer in einem von drei Grundzuständen:

1. **running** (auf einer CPU momentan laufend)
2. **runnable** (lauffähig, ready; kann bei Freiwerden einer CPU sofort laufen, alle Betriebsmittel bereit)
3. **blocked** (blockiert; wartet auf ein Betriebsmittel)

Diese Zustände ändern sich gemäß dem Prozeßzustandsdiagramm.

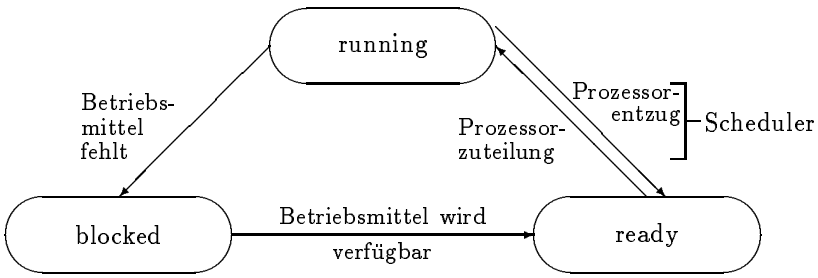
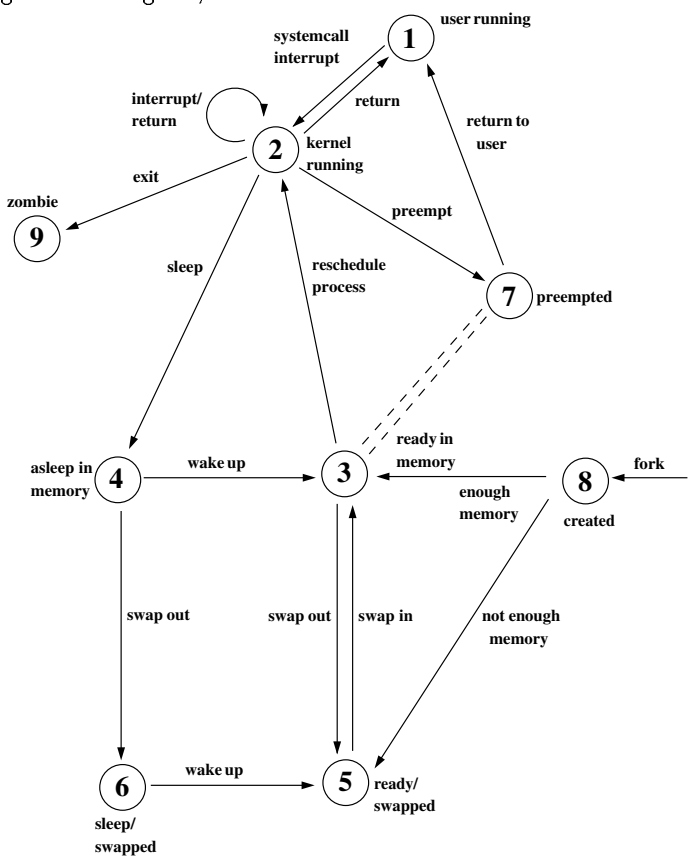


Abbildung 4: Prozeßzustandsdiagramm

Verfeinerungen sind möglich, z.B. in *UNIX*:



1. Prozeß läuft im user mode
2. Prozeß läuft im kernel mode
3. Prozeß kehrt vom kernel mode in den user mode zurück, aber der Kern entzieht ihm den Prozessor
4. Prozeß lauffähig, sobald ihm Prozessor zugeteilt wird
5. Der Prozeß schläft im HSP
6. Der Prozeß ist lauffähig, muß aber vom swapper eingelagert werden
7. Der Prozeß schläft und wurde eingelagert
8. Der Prozeß wurde neu erzeugt und ist in einem Zwischenzustand
9. Der Prozeß hat *exit*() ausgeführt. Der Prozeß existiert nicht mehr, aber er hinterläßt einen exitcode und Abrechnungsinformationen

Ein Prozeß ist *user running*, bis er sein Zeitquantum verbraucht hat oder einen Systemaufruf ausführt (oder von einem HW Interrupt unterbrochen wird). Ein Zeitquantum wird vom BS (in einem HW Register) gesetzt, bevor der Prozeß in den user running mode kommt. Die HW zählt das Register parallel zur Bearbeitung des Prozesses herunter; wenn das Register=0 ist findet ein HW Interrupt statt und es wird (in der HW) zu diesen Serviceroutinen umgeschaltet, in diesem Fall in den scheduler. Im Falle eines Systemaufrufes kommt es zur Blockade, wenn ein Betriebsmittel angefordert wurde, das nicht sofort vorhanden ist. Am Ende eines Systemaufrufs

läuft der scheduler und entscheidet, ob dem Prozeß der Prozessor entzogen wird (preemption) . Bei der Prozeßumschaltung werden alle Register in den Leitblock oder in den Stack gesichert und dann von der entsprechenden Stelle des neuen Prozesses ein neuer Registersatz geladen. Ist zuletzt der PC (program counter) geladen, ist die Umschaltung vollzogen.

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 batch Benutzer minimal sein
5. Durchsatz: Der Gesamtdurchsatz von Prozessen soll maximal sein
6. Der Planungsaufwand selbst soll minimal sein

Einige der Ziele widersprechen sich, gute Kompromisse sind gefragt. 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.

6.15 Karussell Laufplanung (round robin scheduling)

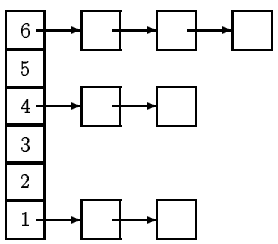
Es wird stets der erste Prozeß der run-queue ausgeführt. Beim Umschalten kommt der Prozeß als letzter in die WS. Dies ist ein oft angewandtes Prinzip, da es einfach und fair ist.
Das Problem ist die Länge des Zeitquantums Q. In $UNIX$ dauert ein Prozeßwechsel ca. 5ms bei 1 MIPS. Falls Q=5ms haben wir 100% overhead. Falls Q=500ms haben wir nur 1% overhead, aber der elfte Prozeß in der WS muß 5sec warten, bis er die CPU bekommt. Ein gebräuchlicher Wert für Q ist 100ms.

6.16 Laufplanung mit Prioritäten

Prioritäten können zugeteilt werden (Prof., Student, Notabschaltung, ...) oder dynamisch aus dem Laufzeitverhalten errechnet werden. Z.B. kann die Priorität zu $\frac{1}{f}$ gesetzt werden, wobei $f = \frac{Laufzeit}{Quantum}$, um kurze interaktive Prozesse zu bevorzugen.

6.17 Mehrfachwarteschlangen

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



Beispiel einer Laufplanung
Kürzester Auftrag zuerst
Falls 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 als nächstes bearbeitet. Die mittlere Wartezeit ist:
$[a_1 + (a_1 + a_2) + (a_1 + a_2 + a_3) \cdots]/k$
$= [k \cdot a_1 + (k-1) \cdot a_2 + \cdots + a_k]/k =: m_k$
Gilt nun $a_i > a_j$ bei $j = i + d$ so erhalten wir
$(k - (i-1)) \cdot a_i + (k - (j-1)) \cdot a_j$
$= (k + 1 - i) \cdot a_i + (k + 1 - j) \cdot a_j$
$= (k+1) \cdot (a_i + a_j) - i \cdot a_i - (i+d) \cdot a_j$
$= (k+1) \cdot (a_i - a_j) - i \cdot (a_i + a_j) - d \cdot a_j,$

d.h. durch Vertauschen von a_i und a_j wird der Betrag von a_i und a_j in m_k kleiner. Die Auftragslänge und der Auftragseingang können z.B. in stapelorientierten Transaktionsverarbeitungssystemen gut geschätzt werden.
Oft kann die Bearbeitungszeit nur geschätzt werden, z.B. aus dem Verhalten in der Vergangenheit. Der Schätzwert S muß dann aber immer wieder neu berechnet werden. Ein guter Kompromiß ist es, den neuen Wert zu gleichen Teilen aus dem alten und dem jüngsten Wert zu bestimmen: $S' = \frac{S_{alt}+S_{neu}}{2}$. Damit **altert** der Einfluß früheren Verhaltens.

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

6.18 Zwei Ebenen Ablaufplanung

Sind sehr viele Prozesse vorhanden, so sollen sehr lang markierte Prozesse auf Disk ausgelagert werden (und sehr lang 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.

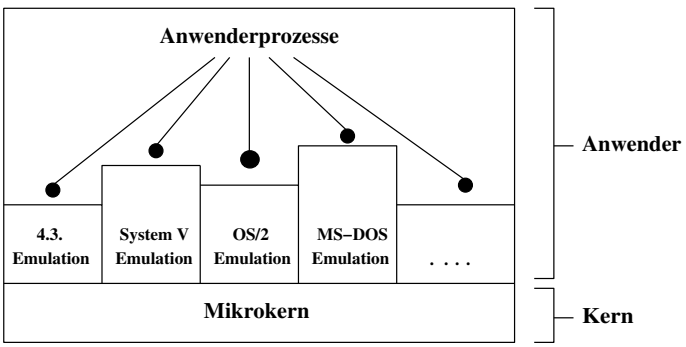
7 Das Betriebssystem MACH

Das MACH Betriebssystem wird an der Carnegie Mellon University (CMU) entwickelt. Eine kommerzielle Version ist OSF/1 der Open Software Foundation (IBM, DEC, HP, ...). Es ist ein Mikrokernelsystem. Ein kleiner Mikrokern bildet die Basis, auf der viele verschiedene Betriebssystemschnittstellen (z.B. *UNIX*, OS/2) implementiert werden können. Der Kern ist strukturiert nach objektorientierten Prinzipien, d.h. in Gruppen von Datenstrukturen und Prozessen, die über Nachrichtenaustausch kommunizieren (und zwar auch über das Netz). MACH 2.5 hatte noch *UNIX* Code im Kern, MACH 3.0 hat einen Mikrokern mit *UNIX* Code im Benutzerraum. Gegenwärtige Ziele sind:

- Implementierung weiterer BS Schnittstellen auf der Basis des Mikrokerns
- Unterstützung großer, dünn belegter Adreßräume
- Unterstützung transparenter Zugriffe auf Netzwerkressourcen
- Ausnutzung von Parallelität im System und in den Anwendungen (Threads, Netzwerk)
- Weiterentwickeln der Portabilität

7.1 Der MACH Mikrokern

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

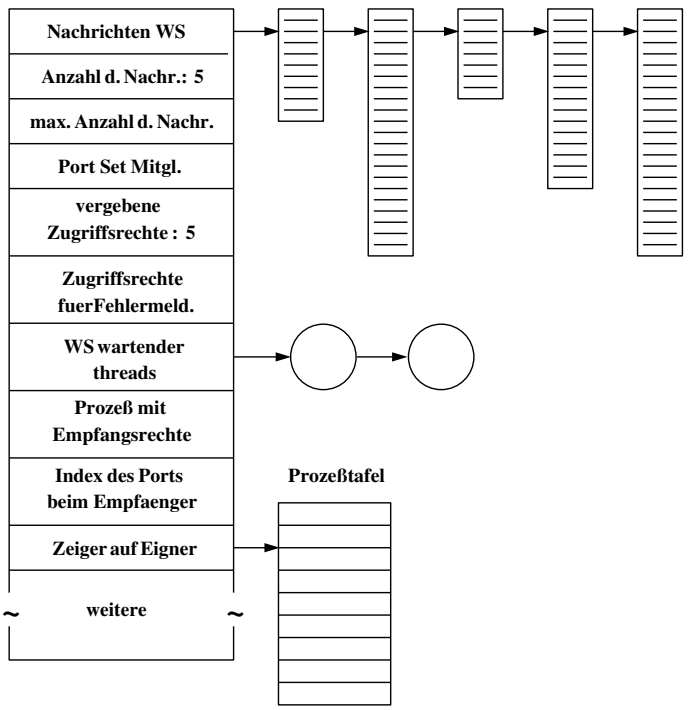


Der Kern kennt fünf Hauptobjekte:

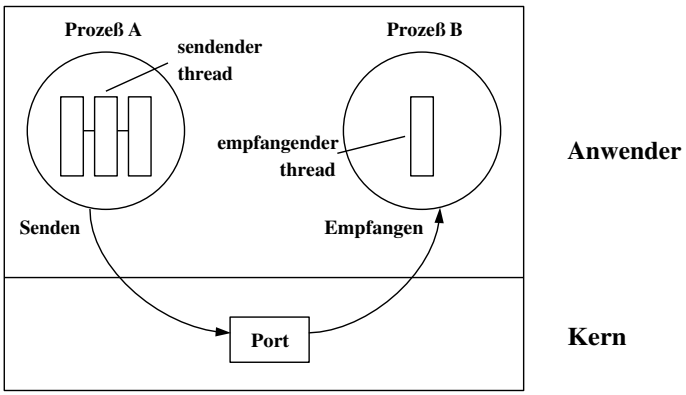
1. Prozesse
2. Threads
3. Speicherobjekte
4. Ports
5. Nachrichten

Ein Prozeß (task) ist die übliche Verwaltungseinheit. In ihm können mehrere threads ablaufen, die sich die task-Umgebung teilen. Ein *UNIX* Prozeß entspricht einer 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 auszulagern.
Ein **Port** ist ein von MACH verwalteter und geschützter Briefkasten. Es werden pro Port maximal 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, in der Regel durch Kopieren der Nachricht, wenn möglich aber auch durch Kopieren von Zeigern. Ein **message interface generator** (*MIG*) unterstützt das Vereinbaren von Nachrichtenformaten (und das Generieren entsprechenden C codes zum Senden und Empfangen).
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.
Ein Port hat u.a. folgende Felder:



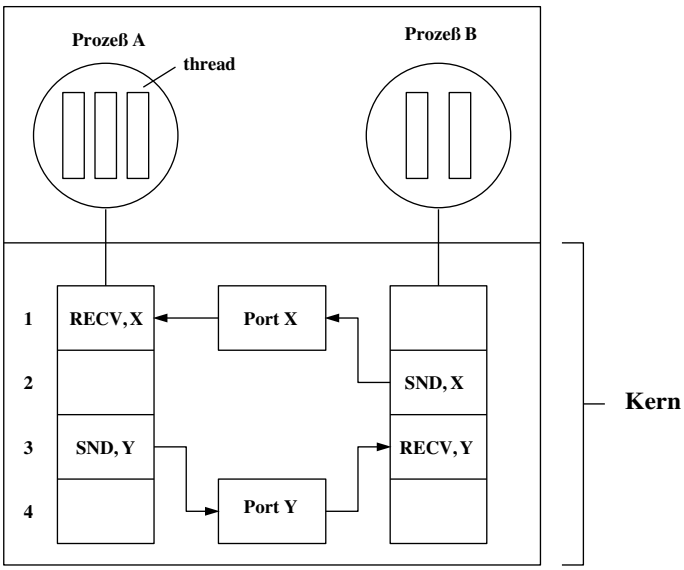
Ports können in Gruppen zusammengefaßt werden. Der Empfänger empfängt dann eine Nachricht von irgendeinem der Ports und blockiert nur, wenn die Gruppe leer ist.
Selbst die Kommunikation mit dem Kern und den Standardservern läuft über (spezielle) Ports. Das bedeutet, daß Serverdienste auch von anderen Rechnern erbracht werden können. Die Fähigkeit (capability) eines Prozesses, auf Ports zuzugreifen, wird als Liste für Ports mit jeweiligem Zugriffsrecht realisiert. Der Index in die Liste identifiziert einen Port mit Zugriffsrecht, analog einem File deskriptor. Rechte sind RECEIVE, SEND oder SEND_ONCE. (Einmal Senden wird von Servern zur Antwort benutzt mit automatischer Auflösung der capability.) Ein Prozeß mit RECEIVE Recht auf einen Port 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 Portmenge
3. Der NULL-Eintrag
4. Der Hinweis, daß der entsprechende Port dealloziiert wurde (weil der Empfänger verstorben ist)

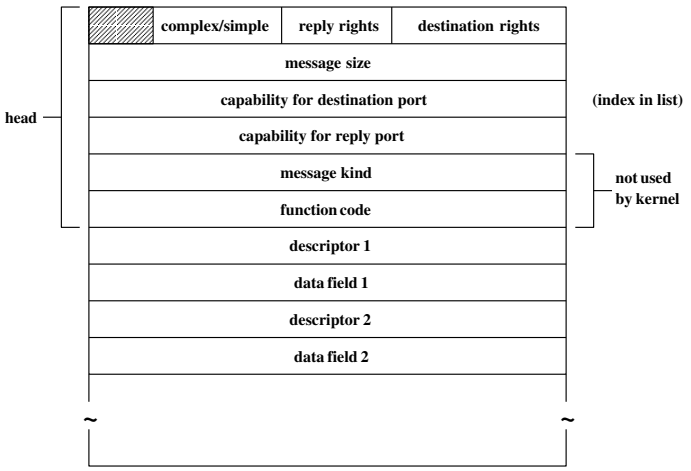
Auf Benutzerebene ist eine capability der Index in diese Liste.



7.2 Nachrichten in MACH

Nachrichten werden mit dem *mach_msg*() Systemaufruf gesendet oder empfangen.

```
mach_msg(&hdr,        /* pointer to message buffer          */
         options,     /* send or receive                    */
         send_size,   /* how long is message ?              */
         rcv_size,    /* how big is buffer ?                */
         rcv_port,    /* capability identifying receive port */
         timeout,     /* ms wait on call completion         */
         notify_port) /* where to send status report        */
```



- simple: Message enthält keine capabilities
- complex: Message mit capabilities (müssen geschützt werden)
- message size: Größe von Kopf und Nachricht
- reply port: wird z.B. bei remote procedure call gebraucht

Beim Senden muß die Nachricht in einem Puffer P zusammengestellt werden. Sie wird dann mit $mach_msg(\&P, \ldots)$ verschickt. Beim Empfang wird ein Puffer P angegeben, und die Nachricht wird mit dem Header dort hineinkopiert. Der MIG kompiliert die entsprechenden Prozeduren für das Ein- und Auspacken der Daten in eine Nachricht.

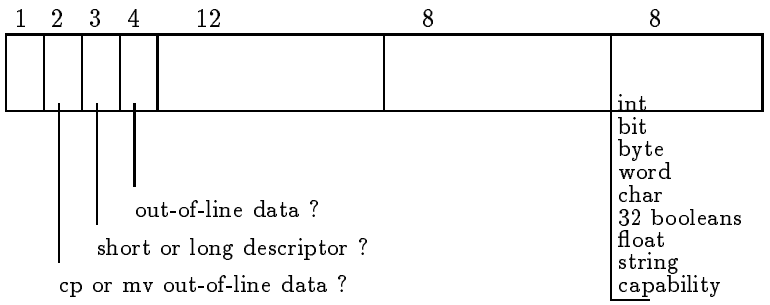
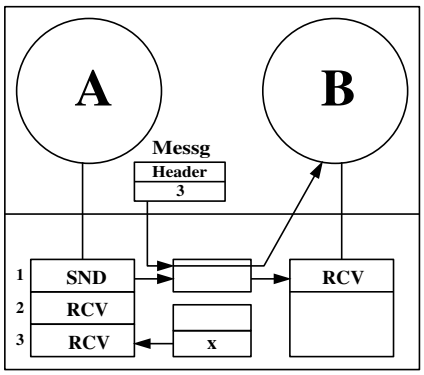


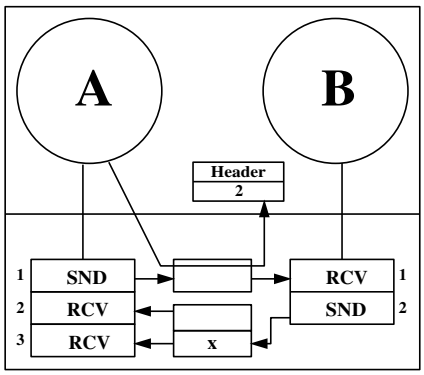
Abbildung 5: Deskriptor in einer komplexen Nachricht

Mit der Deskriptorinformation können Daten zwischen Formaten umgesetzt werden (ASCII - EBCDIC, big endian - little endian).

In *complex messages* können capabilities oder $out-of-line$ Daten übertragen werden. Capabilities können mit oder ohne Löschung beim Sender übertragen werden. (Es gibt aber pro Port nur **ein** RECEIVE Recht.) Wer ein RECEIVE Recht hat, kann zu diesem Port ein SEND Recht vergeben. Dazu wird das RECEIVE Recht, das in der Nachricht eingetragen ist, vom Kern in ein SEND Recht beim Empfänger umgewandelt.



A will B ein SND Recht auf port x uebertragen.



Bei Auslieferung der Nachricht alloziiert der Kern in der Porttabelle von B eine capability (SND, x). Den Index des Eintrags setzt er dann in die Nachricht ein, anstelle des Index, den A dort eingetragen hatte.

Zwischen Prozessen mit gemeinsamen HSP können Daten schneller übertragen werden, indem das zu sendende Objekt in den virtuellen Adreßraum des Empfängers abgebildet wird. Dies geschieht *COW* (copy_on_write). Beim ersten Schreibzugriff des Empfängers wird ein *trap* ausgelöst, und der *trap handler* kopiert die entsprechende Seite.
Zur Übertragung wird im Deskriptor das $out-of-line\ bit$ gesetzt. Im Datenteil wird ein Zeiger auf den Beginn des Datenobjektes übertragen (virtuelle Adresse von Prozeß A) und in den **Size** und **Number** Feldern des Deskriptors eine 20bit breite Längenangabe eingetragen.
Vor Auslieferung der Nachricht sucht der Kern im virtuellen Adreßraum des Empfängers B einen freien Platz dieser Größe. Die entsprechenden Seiten des Datenobjekts werden dorthin abgebildet (durch Aufsetzen der Seitentabellen). Der Kern kann das, da er in der MEMORY MAP des Senders die betreffenden Seiten nachsehen kann und die MEMORY MAP des Empfängers entsprechend verändern kann. $Out-of-line$ Datenübertragung zwischen verschiedenen Kernen ist (sowieso) nicht möglich.
Ausgeliefert wird dann die virtuelle Anfangsadresse des Objekts in Prozeß B (Empfänger).

7.3 Speicherverwaltung

MACH unterstützt sehr großen, dünn besetzten virtuellen Speicher. Theoretisch tut das jedes VM System. Das Problem ist aber, daß 32bit Adressen 4 GB adressieren können, z.B. 4 Mio Seiten à 1K. Das Problem ist die Größe der Seitentabellen. MACH stellt **Regionen** zur Verfügung, bestimmt durch Basisadresse und Größe, in die **Speicherobjekte** abgebildet werden können (etwa in ein File oder in ein großes Array, einen Suchbaum, ...). Adressen, die zwischen Regionen liegen, werden abgefangen und erzeugen einen *trap*, ohne das Einträge in die Seitentabelle nötig sind. Eine in den HSP projizierte Datei kann wie ein char-Array gelesen und beschrieben werden. Es herrscht die übliche Seiteneinteilung. Die Daten können als Nachricht verschickt werden (etwa $out-of-line$) und nach Programmende wird alles Veränderte zurückgeschrieben.

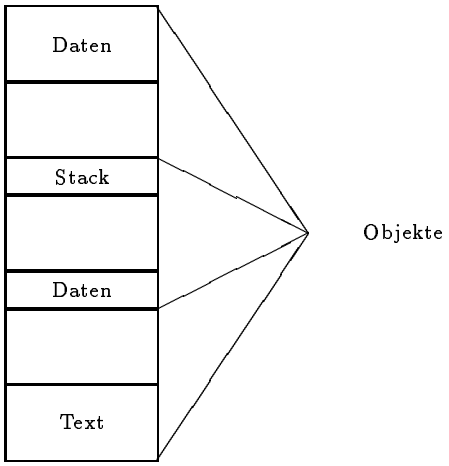
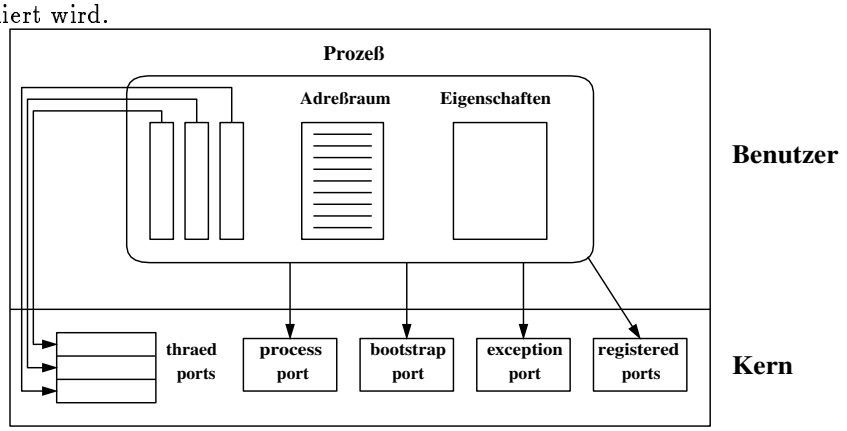


Abbildung 6: Beispiel für einen MACH Adreßraum

7.4 MACH Prozesse

Primär besteht ein MACH 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, scheduling Information und die Adresse der Emulationsbibliothek, falls ein fremdes BS emuliert wird.



Möglichst viel Kommunikation mit dem Kern findet mit Nachrichten via 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 Kernports. Wird hinterher für die Fremdsystememulation benutzt.

exception port: Hier liest der Prozeß Fehlermeldungen wie *Division durch Null*.

registered ports: Standardports, die die Standardserver beim Kern angemeldet haben (z.B. name server).

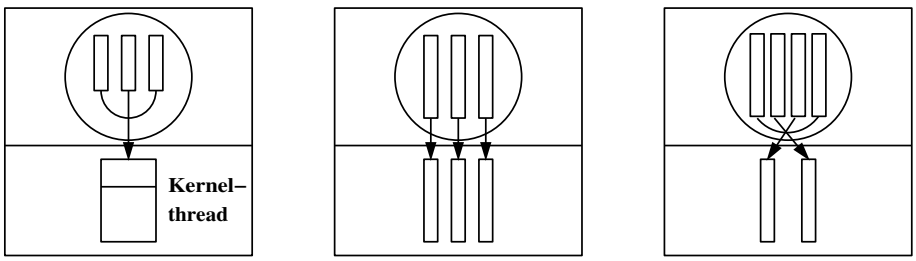
thread ports: Threadspezifische Anweisungen an den Kern, z.B. exit, yield, ...

7.5 Systemaufrufe zur Prozeßverwaltung

Aufruf	Beschreibung
Create	Neuer Prozeß mit oder ohne Kopie des Adreßraums des Vaters. Kind bekommt process, bootstrap und exception port.
Terminate	-
Suspend	Suspendiert Prozeß
Priority	Reaktiviert Prozeß
Assign	Teile den nachfolgend erzeugten threads ein ausführendes processor-set zu
Info	Gibt Laufzeitstatistik aus
Threads	Liste aller threads im Prozeß

7.6 MACH Threads

Jeder Kernelthread hat einen Threadport, um threadspezifische BS Direktiven absetzen zu können.
Die Benutzerschnittstelle (*API*) für MACH Threads ist CThreads. In MACH 2.5 (OSF/1) gibt es zu jedem CThread einen Kernelthread. Daneben existiert eine Emulation von CThreads als Koroutinen (non preemptive) auf einem einzigen Kernelthread. Dies ist fürs Testen/Debuggen geeignet, da die Ausführung deterministisch ist. Jeder blockierende Systemaufruf blockiert aber alle threads.
MACH 3.0 hat (ebenso wie Solaris 2.3) ein user-level thread Paket, in dem die Userthreads auf einem Pool von Kernelthreads abgebildet werden. Dies reduziert den Overhead, da threads Kontextwechsel nicht mehr über den Kern abgewickelt werden, wenn der Kernelthread gleich bleibt. Außerdem teilen sich viele Userthreads einen Stack, den Threadport, etc.

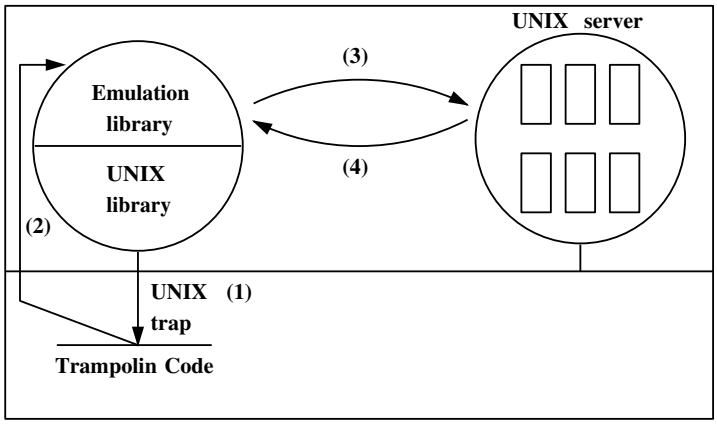


7.7 Scheduling

Objekte des Schedulings sind die Kernelthreads. Prozessoren werden zu Processorsets zusammengefaßt und threads können einem Processorset zugeordnet werden. Jedes Processorset hat eine globale 32Schichten run-queue. Jeder Prozessor hat zusätzlich eine lokale run-queue, auf die er ohne Synchronisation zugreift.

7.8 *UNIX* Emulation

Für die Emulation existiert ein systemweiter *UNIX* server, der selbst multithreaded ist.



Mit jeder *UNIX* Binärdatei wird eine Emulationsbibliothek gebunden. Der MACH Kern fängt *UNIX* systemcalls ab (1) und ruft die entsprechende Funktion in der Emulationsbibliothek auf (up-call) (2). Diese schickt eine Nachricht an den *UNIX* server (3). Ein thread im *UNIX* server liest die Nachricht, führt den entsprechenden Aufruf aus und schickt das Ergebnis über den bootstrap port des Prozesses zurück (4). Die Emulationsbibliothek wird beim $fork()$ vererbt und bei $exec()$ nicht ersetzt. File I/O vermeidet aus Effizienzgründen diesen Weg. Jedes *UNIX* file wird in den Adreßraum der Task abgebildet. Dem zugeordnet ist der **inode-pager** des *UNIX* servers als externer Memory Manager. Seitenfehler beim Dateizugriff werden also vom inode-pager behandelt, ebenso wie Zurückschreiben der Datei auf die Platte.

7.9 Externe Speicherverwaltung

In MACH können verschiedene Speicherobjekte von verschiedenen Managern verwaltet werden. Diese können verschiedene Filesysteme implementieren (z.B. *UNIX* inode-pager) oder maßgeschneiderte paging-Strategien oder Plattenorganisation realisieren.
Kern und Manager kommunizieren über drei Ports:

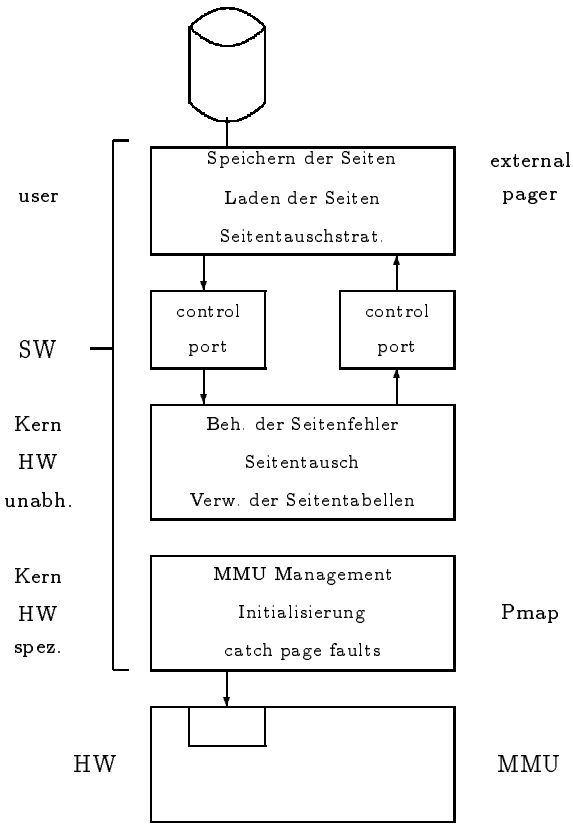
1. **object port** : Der Kern schickt die Anforderungen an den Manager zum object port.

2. **control port** : Der Manager schickt die Antwort an den Kern zum control port.

3. **name port** : Identifiziert das Speicherobjekt eindeutig. (Der Kern antwortet über den name port, wenn er gefragt wird, wohin eine Speicheradresse gehört.)

Am Anfang stellt der Manager den object port zur Verfügung, und der Kern den control port und den name port. Der Manager wartet am object port auf eine Aufforderung 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 keinerlei Antworten 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 noch andere Seiten holen, gemäß irgendeiner Strategie, damit die nächste Antwort schneller gegeben werden kann.
Der Kern bildet die Seite in den Adreßraum des anfordernden threads ab und macht den thread wieder lauffähig. 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 Standardmanager, z.B. den inode-pager für $UNIX$files und einen, der die *malloc*() Aufrufe mit leeren Seiten bedient.



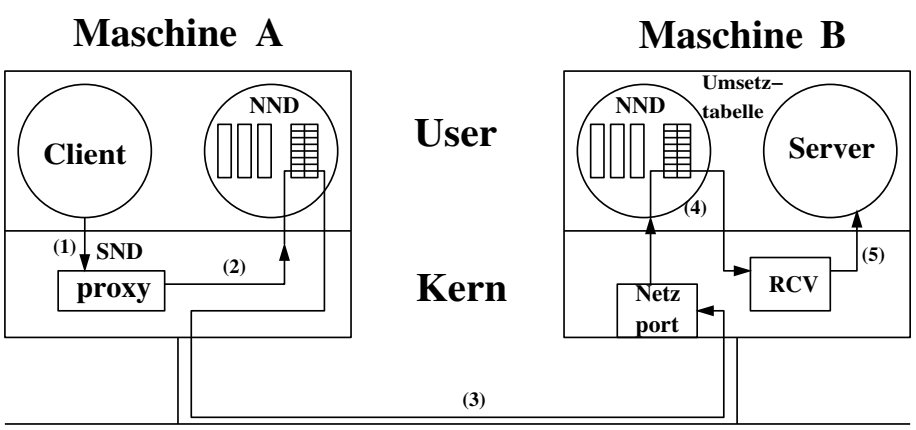
Das Kommunikationsprotokoll zwischen dem Kern und dem Manager benutzt folgende Nachrichten:

init (Kern→Manager)	Der Manager soll sich initialisieren, um ein Objekt zu bedienen, was ein Prozeß durch **map** in seinem virtuellen Adreßraum abgebildet hat. Ports werden festgelegt.
set_attributes (Manager→Kern)	Antwort auf **init**. Teilt dem Kern Zugriffrechte mit (mode bits)
data_request (Kern→Manager)	Liefere dem Kern eine Seite, um einen Seitenfehler zu beheben.
data_provided (Manager→Kern)	Adresse der angeforderten Seite
data_unavailable (Manager→Kern)	Angeforderte Seite ist nicht vorhanden

data_write (Kern→Manager)	Schreibe die Seite auf die Platte zurück (z.B. Aufforderung des paging daemon)
lock_request (Manager→Kern)	Schütze die Seite gegen Schreiben, damit sie der Manager austauschen kann (unterstützt paging Strategie des Managers).
lock_completed (Kern→Manager)	Antwort auf **lock_request** (nötig, da asynchrone Nachrichten statt Systemaufrufe benutzt werden).
terminate (Manager→Kern)	Objekt wird nicht mehr benutzt (Prozeß terminierte oder hat Objekt dealloziiert, z.B. close)
destroy (Manager→Kern)	Zerstöre Objekt (Manager hat Objekt vollständig gesichert. HSP Region kann freigegeben werden).

7.10 Der Netzwerk Nachrichtendienst (NND)

Der NND leitet Nachrichten über Maschinengrenzen weiter. Nachrichten werden z.B. verschlüsselt[2] oder es finden Formatkonversionen statt. 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 network ports macht er für die Anwenderprozesse transparent, indem er lokale Ports zur Kommunikation bereitstellt und diese auf network 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 mit einem lokalen **receive port** anmelden. Der NND koppelt an diesen receive port einen network port für die Kommunikation von außen. Der NND erzeugt auf allen Maschinen je einen Stellvertreter (*proxy*) port und bildet diese auf den network port ab. Der network port hat eine Netzadresse, so daß das Netz Daten an ihn übertragen kann. Die Zuordnung, Name des Dienstes und proxy port, kann von einem name server verwaltet werden; ein Klient erhält so den proxy port, wenn er den Servernamen kennt. (Der Port des name servers ist ein registered port, der dem Klienten vom Kern in die Wiege gelegt wird.)



[2] RSA, DES

Nachrichten werden in fünf Schritten vom Klienten zum Server übertragen:

1. Lokale Nachrichten an den proxy port
2. Lokaler Empfang des lokalen NND am proxy port
3. Abbildung auf das Netz (via Tabelle proxy→network port) und verschicken durch das Netz (evtl. verschlüsseln). Einkopieren von out-of-line Daten.
4. Empfangen am network port durch remote NND. Abbildung auf lokalen Port des Servers (Tabelle) und abschicken (evtl. dechriffieren)
5. Lokaler Empfang am receive port des Servers

Falls capabilities verschickt werden, muß dem verschickten lokalen Port ein network port zugeordnet werden und an der Zielmaschine ein proxy port.

8 Speicherverwaltung

8.1 Seitentauschalgorithmen

Wenn die Memory Management Unit (MMU) einen Seitenfehler entdeckt (d.h. die angebotene virtuelle Adresse nicht umsetzen kann), erzeugt sie einen **trap** ins BS. Das BS muß nun 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.

8.1.1 Optimaler Seitentausch

Der optimale Seitentauschalgorithmus ersetzt diejenige Seite, die am spätesten wieder referenziert werden wird. Dadurch wird das Intervall maximiert, in dem kein Seitentausch nötig ist. Die Auswahl ist i.a. nicht konstruktiv möglich. Innerhalb eines einzelnen Programms geht das zum Teil, dann kann ein spezieller externer pager geschrieben werden. Alle Verfahren versuchen, aus der Vergangenheit auf die Zukunft zu schließen. Man merkt sich also, ob eine Seite kürzlich benutzt wurde oder nicht.

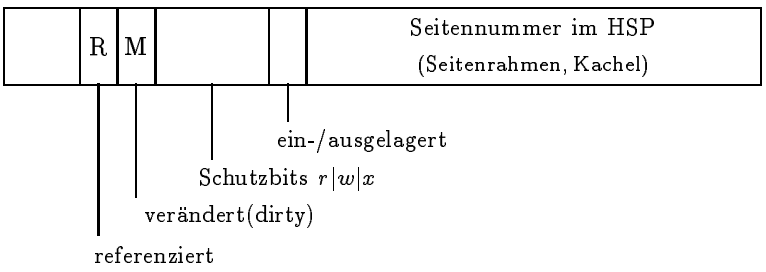


Abbildung 7: Eintrag in eine Seitentabelle

8.1.2 FIFO

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

8.1.3 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“ referenziert wurden. Die Bits RM geben den Seiten eine Priorität.
NRU tauscht eine Seite niedrigster Wichtigkeit gegen die benötigte Seite. Der Algorithmus hat wenig Aufwand, falls die Statusbits von der HW gesetzt werden und in der Praxis ein gutes Verhalten.

8.1.4 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, man ersetzt aber nur, 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. Falls alle R=1 haben, wird die älteste Seite ersetzt.

8.1.5 Clock

Um die Listenverwaltung im 2^{nd}Chance zu optimieren, geht man mit einem Zeiger durch die (logisch zirkuläre) Liste aller Seitenrahmen. Man ersetzt die Seite unter dem Zeiger, falls R=0, sonst setzt man R=0 und probiert die nächste Seite, bis eine Seite mit R=0 gefunden wurde.

8.1.6 LRU (least recently used)

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

8.2 HW Implementierung

1. Statt des R-bits wird ein Zeitstempel im Seitenrahmen gespeichert, z.B. ein 64bit Zyklenzähler. Der Seitenrahmen mit dem kleinsten Zähler wird getauscht. Das Durchsuchen **aller** Seitenrahmen ist immer noch teuer (z.B. 128MB HSP = 128K Seitenrahmen, bei 1K Seitengröße).

2. Bei n Rahmen sei eine Matrix aus $n \times n$ Bits gegeben. Wenn Seitenrahmen k referenziert wird, setzt die HW die Zeile k auf 1 1...1 und die Spalte k auf 0 0...0. Die kleinste Zeile entspricht dann dem ältesten Seitenrahmen. Probleme sind die Größe des Arrays und die Kosten der HW.
   Beispiel: 128MB HSP→256KB Matrix→256 Seitenrahmen.

Ein generelles Problem ist, daß SpezialHW die Portabilität negativ beeinflußt.

8.3 SW Implementierung

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

2. Man wird nur noch nach jeder Zeitscheibe in der SW tätig, die HW setzt das R-bit.
   aging: In jedem Seitenrahmen speichert man in einem Geschichtsvektor $\vec{v}$, der die R-bits der letzten k Zeitscheiben speichert. Das jüngste R-bit steht links.
   $\vec{v_i} = (\vec{v_i} >> 1)|(R_i << (k-1))$ wird in jeder Zeitscheibe ausgeführt. Die ältesten Seitenrahmen haben das kleinste $\vec{v}$ (aber alles älter als k Zeitscheiben wird gleichbehandelt). In der Praxis ist $k = 8$ gut genug.

8.4 Modellierung des Seitentauschs

Das Verhalten eines Seitentauschssystems wird charakterisiert durch:

1. Die Referenzsequenz des Prozesses

2. Das Seitentauschverfahren

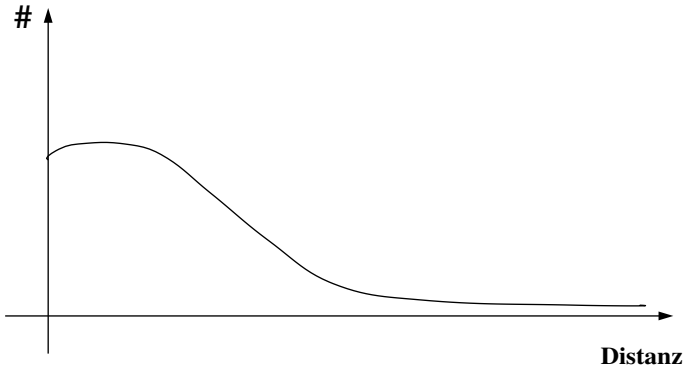
3. Die Anzahl der Seitenrahmen

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

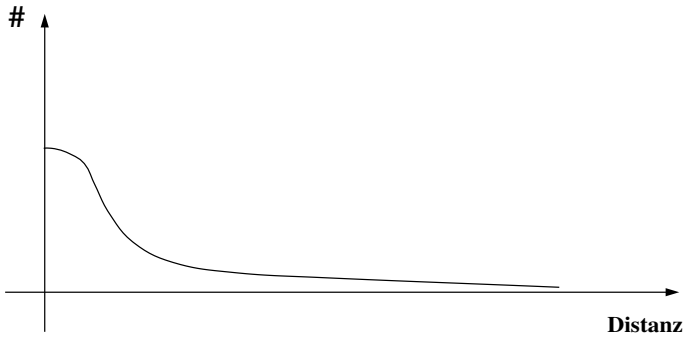
LRU kann wie folgt modelliert werden:

0	2	1	3	5	4	6	3	4	3	3	2	···	Seite
0	2	1	3	5	4	6	3	4	3	3	2		m
	0	2	1	3	5	4	6	3	4	4	3		Rahmen
		0	2	1	3	5	4	6	6	6	4		im HSP
			0	2	1	3	5	5	5	5	6		weitere
				2	1	1	1	1	1	1	6		virtuelle
					0	2	2	2	2	2	1		Seiten
						0	0	0	0	0	0		
∞	∞	∞	∞	∞	∞	∞	4	3	2	1	6	···	Distanz

Die Häufigkeit (Dichte) der Distanzen ist oft



Die Referenzen haben eine gewisse **Lokalität**. Aus der Distanzsequenz kann man für verschiedene m die Anzahl der Seitenfehler berechnen. Dies Fuktion sieht oft so aus:



8.5 Das Lokalitätsprinzip

Es gilt also, das Optimum zu finden, wo man den größten Teil der Seitenfehler vermeidet, ohne allzuviele Seiten zu verbrauchen.
Denning nennt die momentan (im letzten Intervall) gebrauchten Seiten **working set** (Arbeitsmenge). Programme bleiben eine Weile bei einer Arbeitsmenge, bevor sie zur nächsten übergehen. Diese Prozesse laufen gut, wenn sie genügend Seiten für das working set haben. Seitenfehler treten bei einem Wechsel der Arbeitsmenge auf. Ist die Seitenzahl im HSP zu klein, so kommt es zu exzessiven Seitentauschen (**trashing**,„dreschen“). Dann ist es besser, den ganzen Prozeß auszulagern (swappen).
Am Anfang muß ein Prozeß die Arbeitsmenge aufbauen. Bei **demand paging** ge-

schieht dies über Seitenfehler.
Prepaging versucht, dem Prozeß sofort die Arbeitsmenge zu geben (z.B. nach Wiedereinlagerung). Blocktransfer der Arbeitsmenge ist effizienter als einzelne Seitenfehler.

8.6 Implementierung des Seitentauschs

Anforderungen an die HW:

1. MMU aus Effizienzgründen
2. Unterbrechbare und damit wieder aufsetzbare Instruktionen

Zu 1. Adreßübersetzung und **validity bit** müssen gesetzt werden, aber besser in der HW. Um das R-bit in der SW zu setzen, wird die Seite als nicht vorhanden (`v=0`) markiert. Man merkt sich anderswo, daß die Seite doch da ist. Meldet die MMU die ungültige Adresse, stellt man in der SW fest, daß sie doch gültig ist. Man setzt dann `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, nachdem alle Seiteneffekte zurückgesetzt wurden (vgl. `*p++`).

8.7 Schritte des Seitentauschs

1. MMU bekommt virtuelle Adresse, die auf ungültiger Seite liegt (`v=0`)
2. HW speichert Information zum Wiederaufsetzten der Instruktion. Kernel trap.
3. Weitere Kontextwechsel in der SW (sichern der restlichen Register etc.), schließlich Aufruf der Seitenfehlerbehandlung im BS (PFH[3]).
4. PFH stellt die verursachende virtuelle Adresse und die Seite fest (aus MMU oder indirekt aus Instruktion).
5. PFH stellt fest, ob ein Pseudofehler oder ein echter Fehler auftrat. Bei einem echten Fehler wird SIGSEGV an den Prozeß gesendet und ein neues scheduling, exit. Pseudofehler:
   (a) Seite nicht eingelagert
   (b) Seite schreibgeschützt wegen `copy_on_write`
   (c) Seite temporär schreibgeschützt durch pager
   (d) Seite pseudogültig markiert, um `R=1` in SW zu setzen

   (c) und (d) brauchen keinen neuen Seitenrahmen und werden hier nicht weiter betrachtet.
   Bei (a) und (b) wird ein leerer Seitenrahmen in der Freiliste gesucht. Ist keiner da, wird der pager aufgerufen, um Seitenrahmen frei zu machen.
6. Ist der ausgewählte Seitenrahmen schmutzig, so wird der verursachende Prozeß suspendiert und ein anderer lauffähig gemacht. Der schmutzige Seitenrahmen wird vom pager auf die Festplatte zurückgeschrieben (möglichst im block transfer mit anderen).

[3] Page Fault Handler

7. Ist ein freier Seitenrahmen vorhanden, so findet das BS in der Seitentabelle des Prozesses die Diskadresse der benötigten Seite und liest sie (per DMA [4]). Während des DMA kann ein anderer Prozeß laufen.

8. Ein HWinterrupt signalisiert dem BS, daß die Seite da ist. Die Seitentabelle des Verursachers wird nachgeführt.

9. Der Verursacher wird in den status quo ante gebracht (Rücksetzen des PC, Laden der Register).

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

[4] Direct Memory Access

9 Input / Output

Benutzer Prozeß	I/O Anforderung
Geräteunabhängige Software	Namensgebung für Geräte, Anforderung/Freigabe, Zugriffsschutz, Pufferung, Platzverwaltung, Fehlerbehandlung.
Gerätetrieber (device driver)	Führen des Geräts (z.B. Motor ein/aus) Greift auf Controllerregister zu
Interrupt handler	Umsetzung Controller Statusänderung Funktionsaufruf → BS
Device Controller	Elektronische Schnittstelle zur HW
Hardware	Das mechanische Gerät

9.1 Geräte

Es gibt zwei wichtige Unterscheidungen:

- **Block Devices** erlauben den wahlfreien Zugriff auf einen beliebigen Datenblock. Typisches Gerät: Festplatte.
- **Character Devices** generieren oder absorbieren einen Zeichenstrom. Typisches Gerät: Printer, Netz.

9.2 Controller

Der Controller ist das elektronische Interface zum mechanischen Gerät., z.B. Disk. Die Positionierbefehle werden als Bitmuster in die Register des Controllers geschreiben. Der Controller generiert die Ströme, die die Positionierung zur Folge haben. Zusatzinformationen werden aus dem Bitstrom herausgefiltert und die reinen Daten werden an den Speicher übertragen (DMA). Zwischenpufferung (Bit→Wort).

9.3 DMA (direct memory access)

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

9.4 Interrupt Handlers

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

9.5 Device Drivers (Gerätetreiber)

Geräteabhängige SW, die abstrakte, geräteunabhängige Befehle umsetzt in konkrete, vom Controller ausführbare Aktionen. Der Driver kennt die Adressen der Controllerregister und den Befehlsvorrat, sowie die Geräteeigenheiten (Zeilenlänge, Zylinderanzahl etc.). Außerdem werden verschiedene Aufträge koordiniert. Die Diskcontroller setzen logische Blocknummern in physikalische Adressen um, mit Zylinder, Spur und Sektor. Der Driver unterstützt auch die Fehlerbehandlung, z.B. Wiederholen des Lesens.

9.6 Geräteunabhängige Software

Sie stellt ein möglichst geräteunabhängiges Interface dar, zwischen Anwendersoftware und Gerätetreiber, z.B.

- Umsetzung des Dateinamens auf das Gerät

  /dev/tty0 → (major device #, minor device #)
  =(Gerät, Ort im Gerät)
  Gespeichert im i-node eines special file

- Puffern von Daten, wenn ein Character gebraucht wird, aber nur ein Block gelesen werden kann.
- Verlegen von physikalischen Blockgrößen
- Zugriffsschutz, z.B. via mode bits in den Gerätedateien

9.7 Festplatten

Festplatte = Zylinder, Spuren (*tracks*), Sektoren (8-32) (*sectors*)
Heute z.B. SCSI, 10ms, 1GB, DM 1500,- [5]
Zugriffszeit = Positionierzeit des Arms (seek time), Rotationswartezeit, Transferzeit
Zugiffszeit kann im Mittel reduziert werden durch geschicktes Einfädeln der Aufträge.

9.8 Arm Scheduling

- **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, ...).
- **SSF** - shorted seek first
  Nimmt den Auftrag als nächstes, der zur kürzesten Positionierzeit führt. Unfair, da Ausreißerauftrag u.U. nie behandelt wird.

9.9 Elevator (Fahrstuhl)

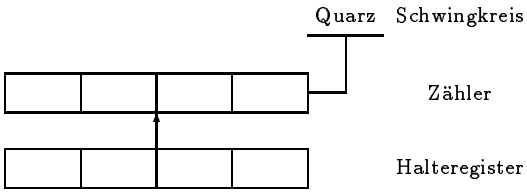
Der Schreib-/Lesekopf wird in der Liste der Zylinder auf und ab bewegt, jeweils maximal lang in einer Richtung, wie ein Fahrstuhl. Alle auf dem Weg liegenden Aufträge werden behandelt. Variante: Die Liste der Zylinder wird logisch zum Ring geschlossen. Ähnliches ist für Sektoren möglich, wenn Sektoradressen on-the-fly vom Driver gelesen werden können.

9.10 RAID

Die Datentransferrate ist durch die Rotationsgeschwindigkeit begrenzt. Der Ausweg ist der Parallelismus. RAID = Redundant Array of Inexpensive Disks. RAID gibt es in mehreren Ausbaustufen, z.B. 38 Disks parallel. Ein 32bit 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 32bit Wort aus 37 Bit rekonstruiert werden (fehlerkorrigierender Code).

[5] Stand 10.2.1994

9.11 Uhren (Timer)



Das Halteregister wird durch die SW mit einem Wert geladen. Der Zähler wird bei jeder Schwingung des Quarzes dekrementiert. Ist der Zähler=0, so wird der Timerinterrupt ausgelöst und das Halteregister in den Zähler kopiert. Die Periode zwischen den Interrupts ist ein **clock tick**.
Clock Software hat folgende Funktionen:

- Uhrzeit nachführen (z.B. in Sekunden)
- Zeitscheibe für Prozesse bereitstellen (z.B. 100ms, 10ms)
- Accounting / Abrechnung
- *ALARM* Systemaufruf verwalten
- Watchdog timer fürs System bereitstellen

Der Timerinterrupt 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, *SIGALRM* gegeben werden muß etc.

10 Deadlocks (Verklemmungen)

Eine Menge von Prozessen ist verklemmt, wenn jeder auf ein Ereignis wartet (z.B. Freigabe eines Betriebsmittels (BM)), das nur ein anderer Prozeß in der Menge herbeiführen kann. Verklemmungen entstehen, wenn mehrere Prozesse sich gegenseitig beim Zugriff auf BM blockieren.
Einfachstes Bsp.:
P_1 hat BM_1, will BM_2.
P_2 hat BM_2, will BM_1.
Es gibt wegnehmbare (preemptable) und nicht wegnehmbare (non preemptable) Ressourcen, z.B. virtueller Speicher ist preemptable, Printer ist non preemptable.

10.1 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 freiwillig freigegeben werden.

4. **Zirkuläres Warten**:
   Zwei oder mehrere Prozesse warten zirkulär auf die 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 Wiederaufsetzung**:
   Man erkennt das Problem, wenn es aufgetreten ist und beseitigt es danach automatisch (z.B. reboot).

3. **Dynamische Vermeidung**:
   Man kontrolliert die BMzuteilung dynamisch, um Verklemmungen zu vermeiden (z.B. via Bed. 3 u. 4).

4. **Statische Vermeidung**:
   Man baut das System von vornherein so, daß eine der Bedingungen 1–4 gar nie auftreten kann (z.B. Bed. 2 u. 4).

10.2 Modellierung von Verklemmungen

Resource graphs (BMgraphen), wait-for-graphs

BM: □ , Prozesse: ○

R → P : BM R wurde Prozeß P zugeteilt

P → R : Prozeß P wartet auf Zuteilung von R

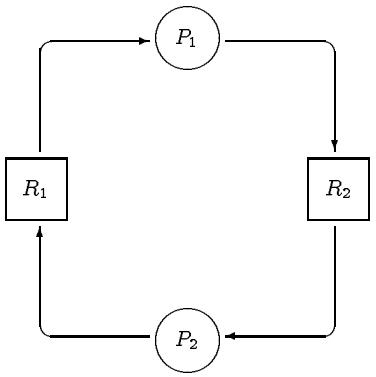


Abbildung 8: Einfachster Deadlock

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

10.3 Wiederaufsetzen nach Verklemmung

Hierzu nutzt man Bed. 3 aus. Man nimmt einem Prozeß einige seiner BM weg oder man terminiert den ganzen Prozeß. Systeme mit Seiteneffekten (Datenbanksysteme) müssen von vornherein so entworfen werden, daß **Transaktionen** rückgesetzt werden können (mit Freigabe der BM), falls eine Verklemmung eintritt.

10.4 Mehrfachbetriebsmittel

Der folgende Algorithmus entdeckt Situationen, die zu Verklemmungen führen, auch in Systemen, wo BM mehrfach vorhanden sind. Voraussetzung: man kennt von vornherein die max. Anzahl benötigter BM.
Wir kennen:

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

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 Banker's Algorithmus von Dijkstra gewährt BM Anforderungen dann, wenn sie in einem 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.
Für alle lauffähigen Prozesse simuliert man die Auswirkungen ihres Ablaufs auf dem BMvektor 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 übriggebliebenen 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. Goto 1.

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

Der Banker's Algorithmus berücksichtigt **nicht**, daß

- Ressourcen auch zurückgegeben werden können, bevor der Prozß terminiert (nicht jeder unsichere Zustand muß daher zu einer Verklemmung führen.),
- die maximal gebrauchten BM nicht immer von vornherein bekannt sind,
- neue Prozesse durch $fork()$ oder durch den Benutzer entstehen können.

Index

UNIX, 3
- Benutzerstruktur, 6
- Block special file, 18
- C-Threads, 8
- Character special file, 18
- data segment, 11
- Dateisystem, 15
- Dateiverzeichnisse, 4
- Datenflusmarken, 9
- Einzeigeruhrprinzip, 13
- fork(), 7
- Fragmentierung, 12
- i-node, 16
- Input/Output, 18
- Kind-Prozesse, 5
- Leichte Prozesse, 7
- Leitungen, 5
- links, 15
  - hard links, 15
  - soft links, 15
- locking, 15
- Mehrprozesbetriebssystem, 5
- mounting, 15
- page daemon, 13
- Paging, 13
- Pipes, 18
- Plattenorganisation, 16
- POSIX-Standard, 6
- Prioritat eines Prozesses, 10
- Prozesleitblock, 6
- Prozesse, 5
- Prozessynchronisation, 5
- Prozestabelle, 6
- reales Speicherbild, 11
- Reservierungen, 15
- Scheduling, 9
- Segmentierung, 11
- Seiten, 12
- Seitentabelle, 12
- Seitentauschalgorithmus, 13
- Shell, 3
- Signale, 5
- Speichermodell, 11
- swapper, 9
- Swapping, 12
- text segment, 11
- Threads, 8
- trap-Instruktion, 3
- User structure, 7
- virtueller Speicher, 12
- virtuelles Speicherbild, 11

MACH
- *UNIX* Emulation, 43
- capabilities, 38
- complex messages, 40
- externe Speicherverwaltung, 44
- Kommunikationsprotokoll, 45
- message interface generator, 38
- Nachrichten, 39
- name port, 44
- Netzwerk Nachrichtendienst, 46
- object port, 44
- Port, 38
- Prozesse, 42
- Scheduling, 43
- Speicherverwaltung, 41
- Threads, 43

Deadlocks, 56
- non-preemtable Ressourcen, 56
- preemptable Ressourcen, 56

Input / Output, 53
- RAID, 54
- arm scheduling, 54
- Block Devices, 53
- Character Devices, 53
- Controller, 53
- direct memory access, 53
- elevator, 54
- first come first served, 54
- Geratetreiber, 53
- interrupt handlers, 53
- shorted seek first, 54
- Uhren, 55

Interprozeskommunikation, 20
- Ablaufplaner, 33
- atomar, 21
- C Thread Conditions, 24
- dispatcher, 36
- essende Philosophen, 30
- high level scheduler, 36
- Karussell Laufplanung, 35
- Laufplanung mit Prioritaten, 35
- Lock Variables, 20
- Mehrfachwarteschlangen, 35
- message passing, 26
- messages, 30
- Monitore, 25
- non-preemptive scheduling, 35
- preemption, 35
- preemptive scheduling, 35

Prozesablaufplan, 33
Prozeszustandsdiagramm, 33
Rendez-vous, 27
Semaphore, 24
shared memory, 29
Sleep, 22
Strikte Abwechslung, 20
Wakeup, 22
wechselseitiger Ausschlus, 20
Zugriffskonflikte, 20
Zwei Ebenen Ablaufplanung, 36

Speicherverwaltung
2^{nd}Chance, 48
Clock, 49
demand paging, 50
FIFO, 48
Implementierung des Seitentauschs, 51
least recently used, 49
Lokalitatsprinzip, 50
Modellierung des Seitentauschs, 49
optimaler Seitentausch, 48
prepaging, 51
Seitentauschalgorithmen, 48
System V, 28
Semaphore in System V, 28