Gunther Heidemann
Technische Informatik 1 (Betriebssysteme)

Markus Gärtner

1. Februar 2004

Inhaltsverzeichnis

Vorwort

Dies ist eine Vorlesungsmitschrift der Vorlesung Technische Informatik I im Sommersemester 2000 an der Universität Bielefeld bei Gunther Heidemann. Es handelt sich hierbei nicht um ein Skript, da es nicht auf inhaltliche und grammatikalische Fehler überprüft und überarbeitet wurde. Für Fehler jeglicher Art wird keine Haftung übernommen. Wer einen Fehler findet, kann diesen gerne an mgaertne@techfak.uni-bielefeld.de schicken, und ich werde mich so schnell wie möglich um dessen Beseitigung kümmern.

Sämtliche Grafiken basieren auf den Folien zur Vorlesung, die so gut wie möglich nachgebildet wurden.

Sollte sich jemand finden, der Lust daran hat, an diesem Skript aktiv weiterzuarbeiten - z.B. Erstellung eines Indexes, Überarbeitung, etc. - kann er sich gerne bei mir per Mail an obiger Adresse melden.

1 Einleitung

1.1 Hardware

1.1.1 Von Neumann-Prinzipien (1946)

1. Programmsteuerung: Struktur des Rechners ist unabhängig vom behandelten Problem
2. Rechner:
   - Hauptspeicher für Programme und Daten
   - Steuerwerk interpretiert Programme
   - Rechenwerk verarbeitet Daten nach „Anweisungen“ des Steuerwerks
   - Input-/Output-Geräte zur Kommunikation mit der „Umwelt“
   - Sekundär- oder Langzeit-Speicher
3. Binäre Darstellung von Daten und Programmen
4. Daten und Programme befinden sich im selben Hauptspeicher
   Maschine kann Daten und Programme verändern
5. Hauptspeicher besteht aus Zellen gleicher Größe, die mit „Adressen“ durchnummeriert werden
6. Programm ist eine Folge von Befehlen, die sequentiell abgearbeitet werden
7. Abweichung: bedingte oder unbedingte Sprungbefehle

Beispiele für keine Von Neumann-Rechner:

- Analogrechner: z.B. neuronale Rechner
- digitale Special Purpose Rechner: z.B. für Bildverarbeitung

1.1.2 Reale Hardware

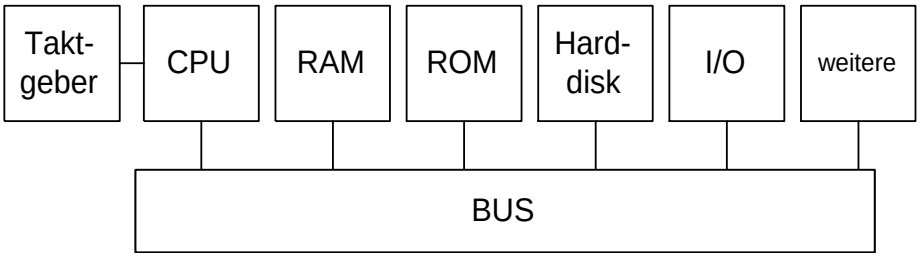


Abbildung 1: Hardwareschema eines Rechners

Bus Meist gemeinsames (d.h. nicht spezialisiertes) Bündel von Signalleitungen zwischen CPU, RAM, usw. bestehend aus:

Datenbus Transportiert Daten zum oder vom Hauptspeicher; besteht typischerweise aus 16, 32 oder 64 Leitungen, die Signale binär als Spannungen (0 V / 3–5 V) tragen; z.B. 32 Leitungen („32 Bit Breite“) ⇒ 4 Byte parallel übertragbar

Adressbus Transportiert Ziel-/Quell-Adresse typische Breiten 8-64 Bit; z.B. „32 Bit Breite“ ⇒ 4 GByte adressierbar

Steuerbus steuert z.B. Lesen/Schreiben

Versorgungsbus Strom, Takt, usw.

RAM Random Access Memory

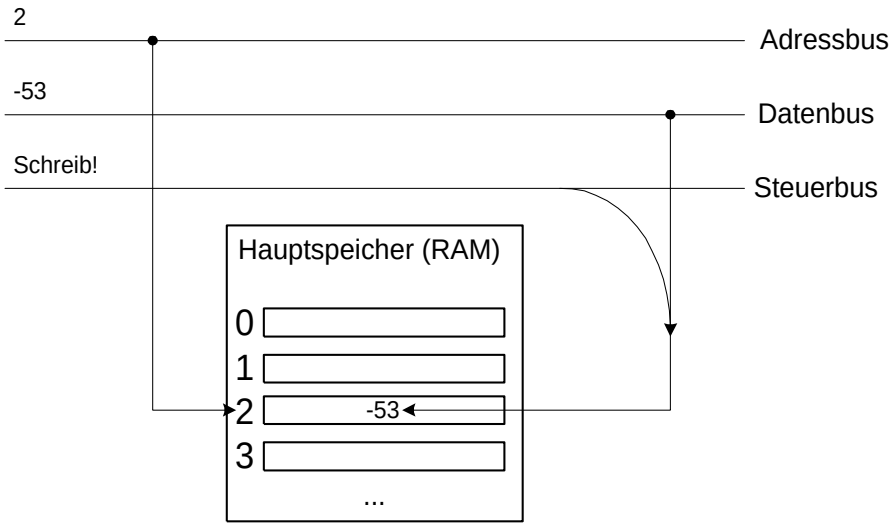


Abbildung 2: Funktionsweise des RAMs

CPU Central Processing Unit

Steuerwerk Lädt Befehle gemäß dem Befehlszähler in das Befehls*register*; generiert Steuersignale

Rechenwerk Lädt Daten (Operanden) gemäß den Signalen vom Steuerwerk in Register und verarbeitet sie in ALU(s) (Arithmetic Logic Unit)

1.1.3 Abweichungen von den von Neumann-Prinzipien

- teilweise Parallelisierung
- mehrstufiger Speicher
- *keine* Selbstmodifikation von Programmen

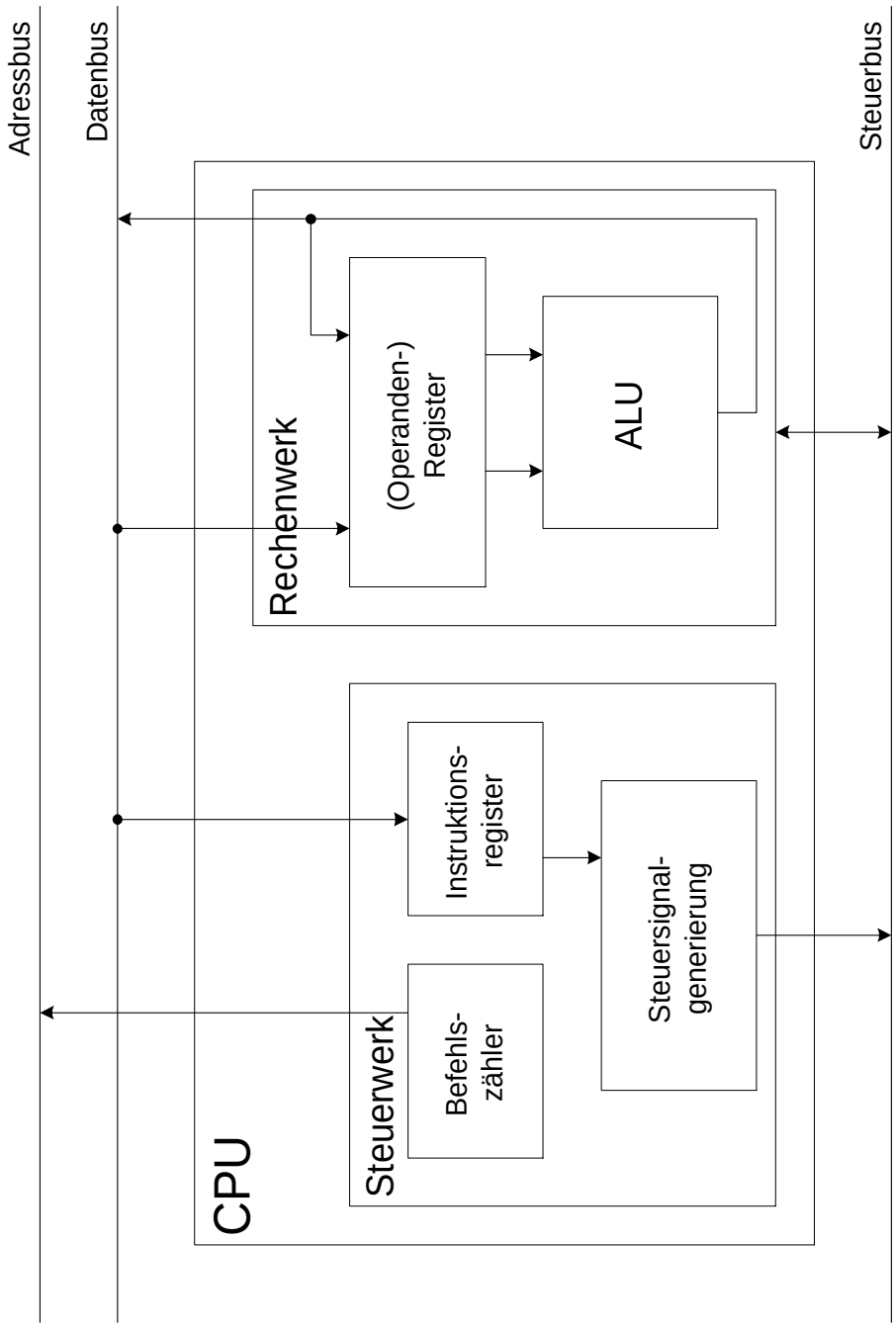


Abbildung 3: Aufbau einer CPU

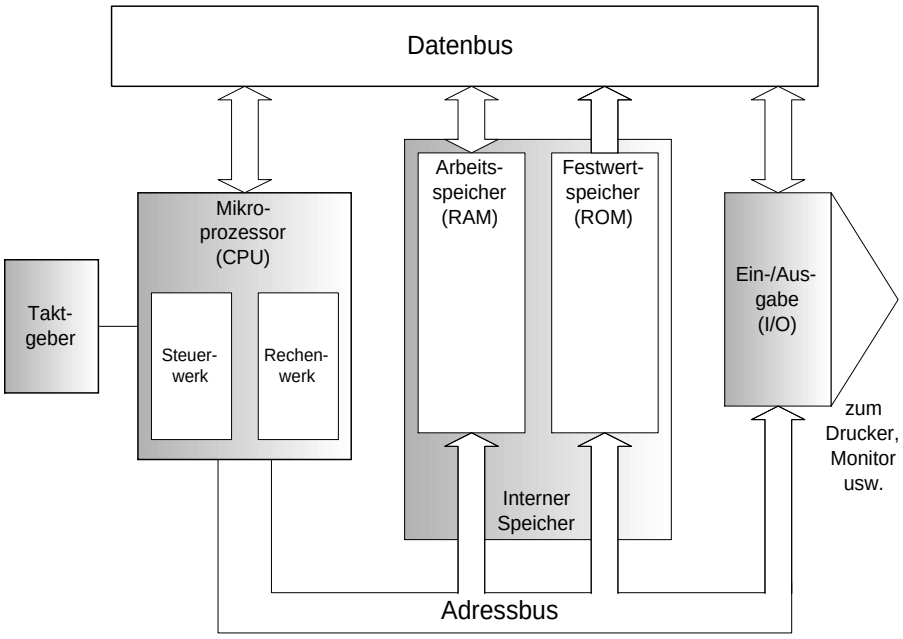


Abbildung 4: Allgemeiner Aufbau eines Rechners

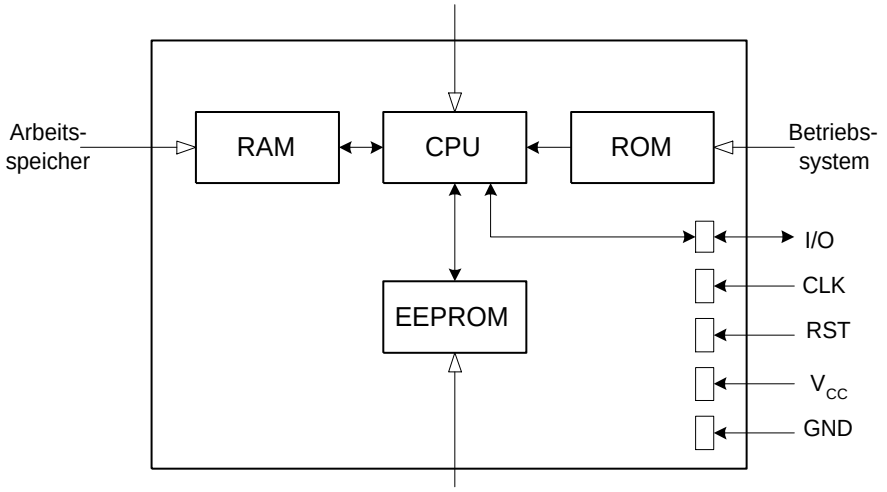


Abbildung 5: Grundlegende Architektur von Chipkarten

EEPROM = Electrically Erasable Programmable Read-Only Memory

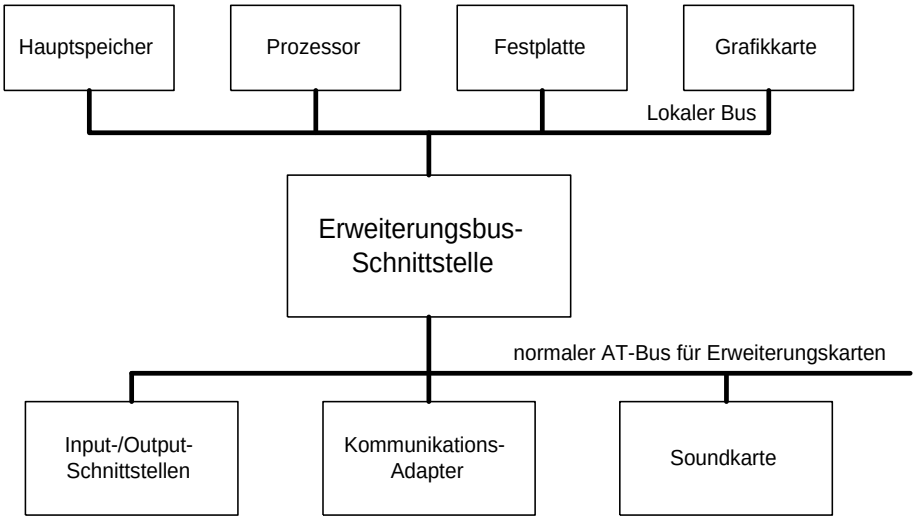


Abbildung 6: Die Local-Bus-Architektur

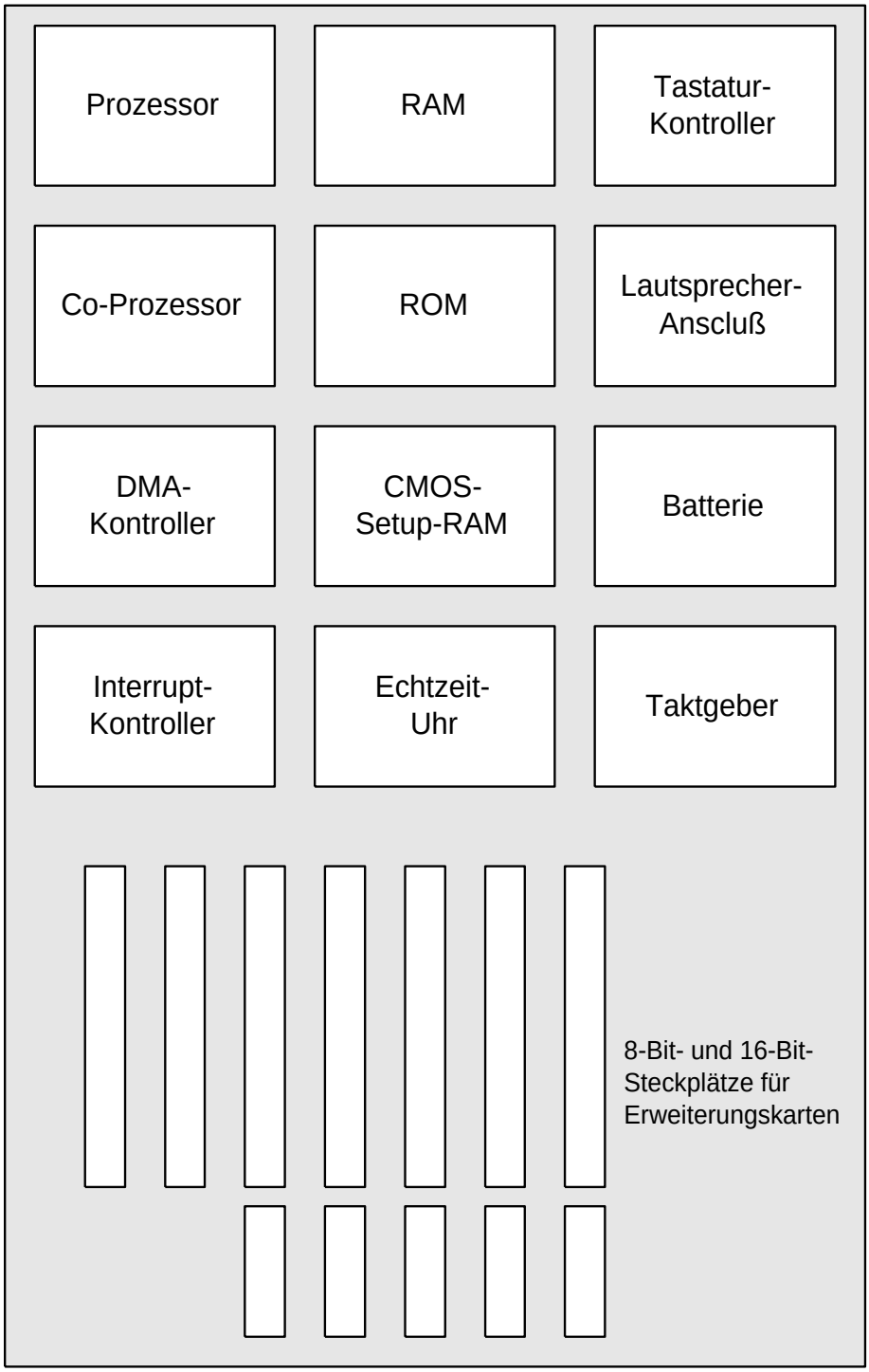


Abbildung 7: Schematische Darstellung einer Hauptplatine

1.2 Betriebssysteme

> Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.
>
> DIN 44300

> A set of programs and routines which guide a computer in the performance of its tasks and assist the programs (and programmers) with certain supporting functions.
>
> *Operating System Survey*
> Sayers

> Ein Betriebssystem [*ist*] eine Menge von Programmen, die die Ausführung von Benutzeroprogrammen und die Benutzung von Betriebsmitteln steuern.
>
> *Introduction to Operating System Design*
> Haberman

> Der Zweck eines Betriebssstems [*besteht*] in der Verteilung von Betriebsmitteln auf sich bewerbende Benutzer.
>
> *Betriebsysteme*
> Brinch Hansen

1.2.1 Hauptaufgaben

Betriebsmittel so verwalten, dass eine „virtuelle Maschine“ entsteht, die eine direkte Ansteuerung der Hardware (interaktiv oder durch Programmierung) überflüssig macht.

Beispiele:

- logische I/O-Geräte statt physikalische
- Abstraktion:
  laufendes Programm → Prozess (CPU-Vergabe an mehrere Prozesse erzeugt scheinbare Parallelität)
- Speicher:
  explizite Adressierung vermeiden
  verschiedenartige Medien (z.B. RAM und Festplatte) zu logischem Speicher zusammenfassen
- Bereitstellung von Schnittstellen
  1. für Benutzer
  2. für Programmentwicklung

1.2.2 Stellung des Betriebssystems

2 Prozesse, Scheduling & Prozesskommunikation

2.1 Prozesse

2.1.1 Begriffsbestimmung

Ein Prozess ist ein Programm in Ausführung mit einer Prozess-„Umgebung“

Vergleich:

Programm	$\leftrightarrow$	Rezept
Prozess	$\leftrightarrow$	Kochen nach Rezept
Prozessor	$\leftrightarrow$	Koch, der das Rezept abarbeitet unter Umständen auch mehrere gleichzeitig

2.1.2 Mehrprozessbetrieb (Multitasking)

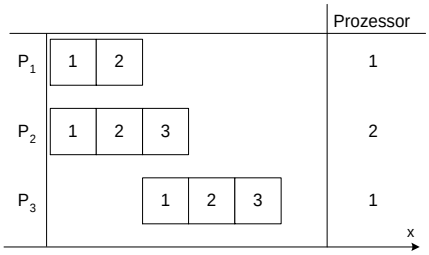
Motivtaion für Multitasking:

- mehrere User gleichzeitig
- I/O (Tastatur, Maus, Plattenzugriff)
- Interaktion mit Anwendungen
- Hintergrundjobs zur Ausnutzung der Ruhezeiten
- Netzwerkprozesse (Automounter, Mail)
- Betriebssystem selbst

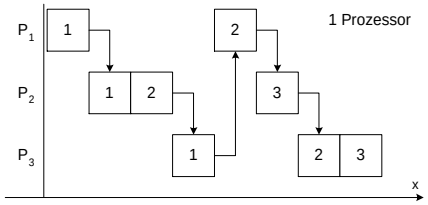
Physikalischer Ablauf

- Echtparallel Anzahl Prozessoren $\geq$ Anzahl (gleichzeitiger) Prozesse
- in der Praxis: Anzahl Prozessoren $<$ Anzahl Prozesse

  $\Rightarrow$ Pseudo–Parallelität:
  Durch sequentielles Bearbeiten mehrerer Prozesse wird Parallelität vorgetäuscht. („Nebenläufigkeit“, „concurrency“)



(a) Echtparalleler Ablauf von Prozessen



(b) Pseudo-Paralleler Ablauf von Prozessen

Gleichzeitigkeit ist nur eine Abstraktion

Die Ablaufkontrolle muss in diesem Fall ein sogenannter Scheduler übernehmen. Dabei ist der Scheduler selbst Prozess des Betriebssystems.

2.1.3 Prozesszustände

Problem: Prozess muß z.B. ein Ereignis in einem anderen Prozess oder I/O abwarten.
Beispiel: (Unix)

cat file.c | grep blabla

Während *cat* eine Datei liest, sucht der Befehl *grep* eine Zeichenkette.

Die Lösung ist ineffektiv, wenn *grep* eine Warteschleife macht, da der Input von *cat* fehlt.
⇒ *3 Prozesszustände*

1. bereit: der Prozess wartet auf Zuteilung der CPU
2. rechnend: dem Prozess ist Zeit von der CPU zugeteilt
3. blockiert: der Prozess wartet auf ein Ereignis, Rechenzeit könnte nicht verwertet werden

2.1.4 Repräsentation von Prozessen im Betriebssystem

Was muß repräsentiert werden?

Ein vom Scheduler unterbrochener Prozess muß später in exakt demselben Zustand wieder aufgenommen werden

⇒ Speichern in „Process Control Block“ (PCB)
= Datenstruktur der Zustandsinformation
= „Kontext“ des Prozesses

Der PCB enthält u.a.:

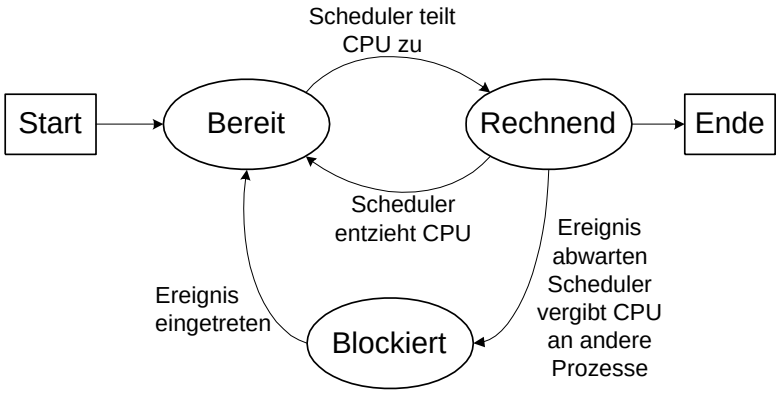


Abbildung 8: Zustandsübergänge

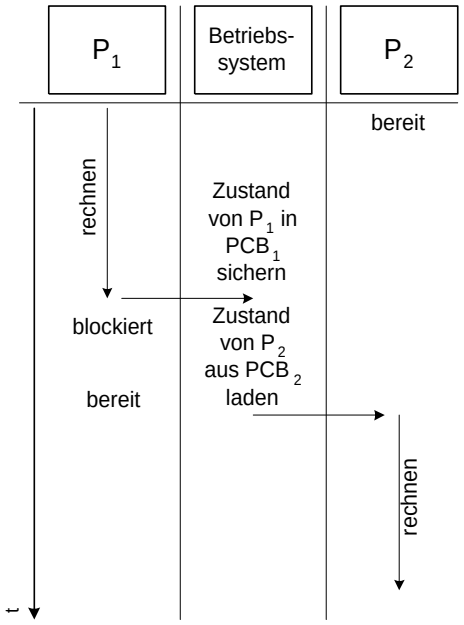


Abbildung 9: Prozesswechsel mit PCB

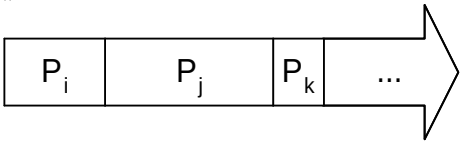
- CPU
  - Befehlszähler
  - Daten- und Adressregister
- Speicher
  - Zeiger auf ein Textsegment
  - Zeiger auf ein Datensegment

- Dateiverwaltung
  - Dateideskriptoren und Puffer
  - aktuelles Verzeichnis
- Kommunikation
  - unbearbeitete Signale
  - Zeiger auf Nachrichten
- Prozessverwaltung
  - Prozesszustand
  - Prozessnummer
  - Erzeugungspunkt
  - verbrauchte CPU-Zeit
  - Benutzer
  - Priorität
- ...

2.1.5 Vorgriff: Segmentierung

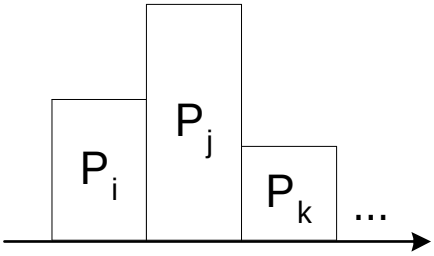
vgl. Abschnitt 3.4, S. 55

„1-dimensionaler“ Adressraum:



Problem: Speicherbereiche der Prozesse variieren in der Größe

Lösung: Segmentierter „2-dimensionaler“ Adressraum:



- Betriebssystem nimmt dem Prozess die physikalische Speicherverwaltung ab
- Jeder Prozess hat eigene Segmente für Text und Daten

- Segmente haben Zugriffsrechte
  - Erleichtert Debugging
  - Schutz vor Zugriff anderer Prozesse

2.2 Scheduler

2.2.1 Ablauf der CPU-Zuteilung

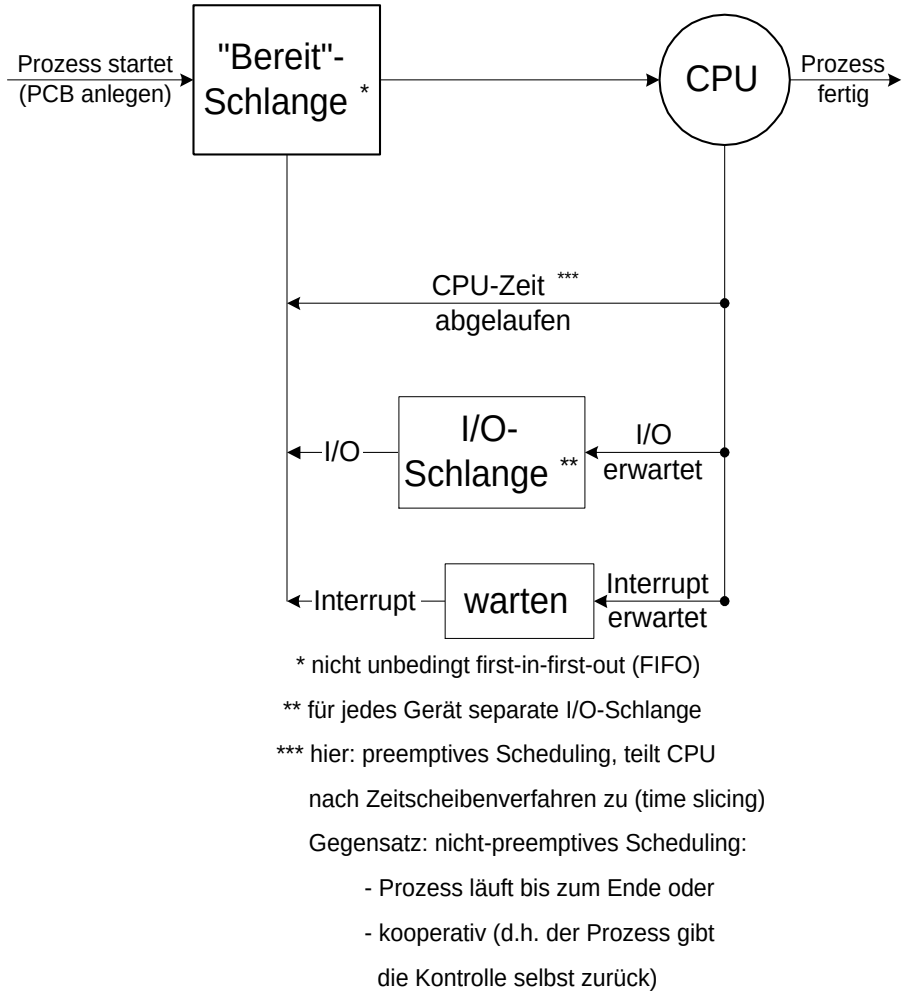


Abbildung 10: Ablauf der CPU-Zuteilung

2.2.2 Scheduling-Algorithmen

Aufgabe:

- aus „Bereit“-Schlange Prozess auswählen
- CPU-Zeit zuteilen

CPU-Vergabe-Kriterien

1. Effizienz: max. CPU-Auslastung
2. Durchsatz: maximiere Anzahl Aufträge pro Zeit
3. Verweilzeit: minimiere Verweilzeit der Aufträge
4. Antwortzeit: minimiere Antwortzeit für interaktive Arbeit
5. Fairneß: Gleichverteilung der CPU-Zeit
6. Termine: termingerechte Vollendung einzelner Prozesse

Wiedersprüchliche Ziele!
z.B.:
1. möglichst wenig Prozesswechsel $\Leftrightarrow$ 4. Prozesswechsel bei jeder Eingabe

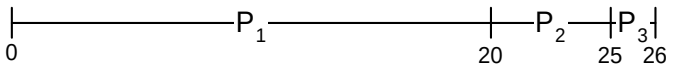
Algorithmen

First-come, first-served (FCFS)

- FIFO, wurde früher im reinen Stapelbetrieb (batch operation) eingesetzt
- „Run to completion“
- nicht-preemptiv, da rechnende Prozesse nicht abgebrochen werden
- Optimiert Punkt 1
- kein Multitasking

Die mittlere Verweilzeit hängt von der Reihenfolge ab!
Beispiel:

Prozess	Zeit
P_1	20
P_2	5
P_3	1



Mittlere Verweilzeiten:

$$P_1, P_2, P_3 \quad : \quad \frac{20 + 25 + 26}{3} = 23,6$$
$$P_3, P_2, P_1 \quad : \quad \frac{1 + 6 + 26}{3} = 11$$

Shortest Job First (SJF)

- wie FCFS mit Ordnung nach Länge
- die Länge ist aber nur schwer voraussagbar
- preemptive Version: Prozess wird suspendiert, wenn ein neuer Prozess kürzer ist als die Restlaufzeit

Round-Robin-Scheduling (RR)

- preemptives Verfahren
- bearbeitet alle Prozesse reihum für ein festes Zeitquant Q

Q	Antwortzeit	Overhead
klein	klein	groß
groß	groß	klein

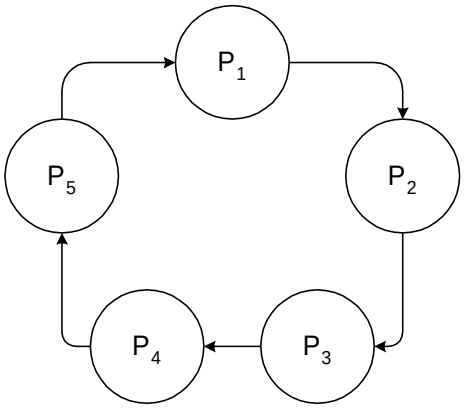


Abbildung 11: Round-Robin-Scheduling für 5 Prozesse

Prioritäts-Scheduling (PS)

- jeder Prozess hat eine eigene Priorität
- ein Prozess mit höchster Priorität erhält als erstes CPU-Zuteilung
- Prioritäten ändern sich zur Laufzeit
- Prozesswechsel bei Unterschreiten der Priorität eines anderen Prozesses

Strategien zur Prioritäten-Festlegung:

- bei Start (statisch)
  - nach Benutzer
  - individuelle Einstufung (Unix: nice)
  - vom Betriebssystem: (absteigend nach Prioritäten geordnet)
    1. zeitkritische Betriebssystemprozesse, z.B. Gerätetreiber
    2. andere Betriebssystemprozesse
    3. Interaktive Prozesse
    4. Stapelaufträge
  - nach vermutetem Prozessverhalten, z.B. hohe Priorität bei viel I/O

- zur Laufzeit durch Betriebssystem (dynamisch)
  - erniedrige Priorität mit zunehmender CPU-Zeit
  - erhöhe Priorität mit zunehmender Warte-Zeit
  - Priorität(P) $\sim \frac{1}{f(P)}$, $f(P)$ = Anteil CPU-Zeit, den der Prozess bis zum Blockieren verbraucht hat

Meist findet sich eine Kopplung von Prioritäts- und Round-Robin-Scheduling, z.B. durch Einteilung in

Prioritätsklassen

4 Klassen

- für jede Prioritätsklasse eine Round-Robin-Schlange
- CPU-Zeit steigt mit fallender Priorität
- Prozess, der alle Quanten aufbraucht, wird abgestuft
- Aufstieg z.B. bei I/O

Priorität	Zeit
4	1 Q
3	2 Q
2	4 Q
1	8 Q

Vorteile:

- interaktive Prozesse sind meist nur kurz und haben damit automatisch eine hohe Priorität
- Prozesse mit viel I/O haben ebenfalls automatisch eine hohe Priorität
- rechenintensive Prozesse rechnen automatisch selten, aber dafür lang
- keine absolute Priorität

Nachteil:

- Prozesse in unterster Klasse müssen bei I/O erst aufgestuft werden

Garantierendes Scheduling

garantiere bei n Prozessen jedem $\frac{1}{n}$ CPU-Zeit:
führe Prozess mit kleinstem $v = \frac{\text{verbrauchte Zeit}}{\text{zustehende Zeit}}$ so lange aus, bis ein anderer Prozess ein kleineres v hat

Deadline Scheduling

Garantiere Fertigstellungstermin in Echtzeit-Systemen:

Earliest Deadline First höchste dynamische Priorität für den Prozess, dessen Termin am nächsten liegt

Least Laxity Bearbeite Prozess mit kleinster Differenz

Zeit bis Termin(Prozess) - Abarbeitungszeit(Prozess)

2.2.3 Interrupts

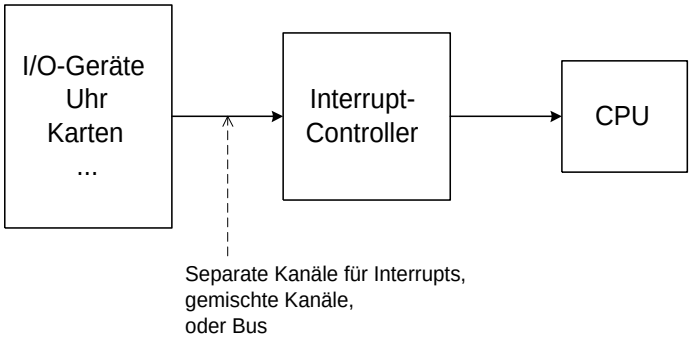
Beim Scheduling gibt es zwei Probleme:

1. Scheduler selbst muss aufgerufen werden
   Schlechte Lösung: Die Prozesse geben selbst die Kontrolle an den Scheduler zurück
2. I/O-Geräte in das Scheduling einbeziehen
   Schlechte Lösung: Scheduler fragt die Geräte ab

Lösung: CPU unterbricht laufenden Prozess bei Interrupt

Zunächst soll es sich hierbei um Hardware-Interrupts handeln, später werden wir noch Software-Interrupts kennenlernen.

Hardware:



Controller Identifiziert die Interrupts und ordnet sie nach Prioritäten

CPU unterbricht laufenden Prozess und sichert ihn, lädt den Interruptvektor (Routine)

Die Interruptroutine des Timerinterrupts stößt den Scheduler an.

2.3 Threads

Threads sind Leichtgewichtprozesse (engl.: leight weight process, LWP) innerhalb normaler Prozesse. Für bestimmte Aufgaben ist eine einfachere Interprozesskommunikation (IPC) möglich.

2.3.1 Motivation

Warum der Prozess im Prozess?

Naive Annahme: Wenn man sich innerhalb eines Prozesses befindet, braucht man auch keine *Inter-Prozess*-Kommunikation. Die Handhabung wird dadurch einfacher gestaltet.

In der Realität ist es allerdings unmöglich auf Inter-Prozess-Kommunikation zu verzichten.

IRQ	belegt durch	Priorität
1	Tastatur	14
2	EGA-/VGA-Grafikkarte	13
3	2. serielle Schnittstelle (COM2)	3
4	1. serielle Schnittstelle (COM1)	4
5	2. parallele Schnittstelle (LPT2)	2
6	Diskettenlaufwerk	1
7	1. parallele Schnittstelle (LPT1)	0
8	Echtzeituhr	12
9	Frei, wenn keine andere Karte	11
10	Frei, wenn keine andere Karte	10
11	Frei, wenn keine andere Karte	9
12	Frei, wenn keine andere Karte	8
13	Coprozessor	7
14	Festplatte	6
15	Frei, wenn keine andere Karte	5

Tabelle 1: Interruptverteilung bei einem Standard-PC

Gleichzeitigkeit

Szenario: handelnde Akteure (kommunizierende Agenten, Doom)

Naiv: Das Hauptprogramm gibt jedem Objekt die Gelegenheit zu agieren

Problem: Wieviel und Wann soll es agieren?

Eleganter: Programmiere Objekte mit Verhalten, die im eigenen Thread laufen, und mit IPC kommunizieren

Parallelität Unabhängige Berechnungen parallelisieren z.B. in der Bildverarbeitung auf Bildausschnitten (s. Abbildung 12).

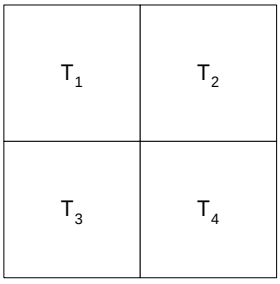


Abbildung 12: Beispiel für die Aufteilung in Threads in der Bildverarbeitung

Pipelining Eine Aufgabe wird von Thread zu Thread durchgereicht. (Parallelität ist hier nicht zwingend nötig, aber einfacher zu programmieren) z.B. Erzeuger-Verbraucher-Problem (s. 2.4.4, S. 30)

Blockierende Systemaufrufe Systemaufrufe wie z.B. `read();` blockieren den Thread, bis die Festplatte (oder Netzwerk, Tastatur) die benötigten Daten verfügbar gemacht haben.

Szenario: Web-Server

Schlechte Lösung:

```
while (1) {
    netzverbindung = warte_auf_client();
    anfrage = empfange_anfrage(netzverbindung);
    ergebnis = lies_datei(anfrage);
    schicke_datei_an_client(netzverbindung, ergebnis);
}
```

Bessere Lösung:

```
while (1) {
    netzverbindung = warte_auf_client();
    bediene_client_mit_thread(netzverbindung);
}
```

Eigener Thread:

```
bediene_client_mit_thread(netzverbindung)
{
    anfrage = empfange_anfrage(netzverbindung);
    ergebnis = lies_datei(anfrage);
    schicke_datei_an_client(netzverbindung, ergebnis);
} /* bediene_client_mit_thread */
```

An diesem Szenario kann man erkennen, dass man möglichst früh blockierende Systemaufrufe in Threads auslagern sollte.

2.3.2 Implementation

Gängige Betriebssysteme haben Threads (Java™ ist hier ähnlich einem Betriebssystem)

Threads sind wie Prozesse:

- Programmzähler
- Stack
- Zustand (bereit, warten, läuft)

Aber: Threads teilen

- Adressraum
- globale Variablen
- Dateideskriptoren

Eigenschaften pro Thread
Programmzähler
Stack
Registersatz
Unterthreads
Zustand: bereit, warten, läuft

Eigenschaften pro Prozess
Adressraum
globale Variablen
Dateideskriptoren
Unterprozesse
Timer
Signale
Semaphoren
Benutzerzugehörigkeiten

Threads sind nicht voreinander geschützt, weil:

- das per Design nicht geht
- das auch nicht sein muß, da Threads immer vom gleichen Benutzer stammen

Kernel Space Threads sind als Prozesse implementiert und werden auch wie diese gescheduled. (Prozesse ohne Threads haben genau einen Thread)

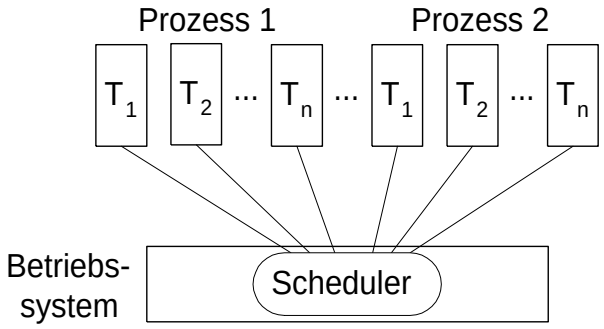


Abbildung 13: Kernel Space Threads

Vorteile:

- „einfach“ zu implementieren
- nutzt Multiprozessorsystem (verteilt die Threads auf die CPUs)

Nachteile:

- der Threadwechsel ist so „teuer“ wie ein Prozesswechsel

User Space Eigener Scheduler (als Teil des Laufzeitsystems) im Prozess. Blockierende Systemaufrufe (z.B. ready) müssen abgefangen werden, damit andere Threads rechnen können.

Vorteile:

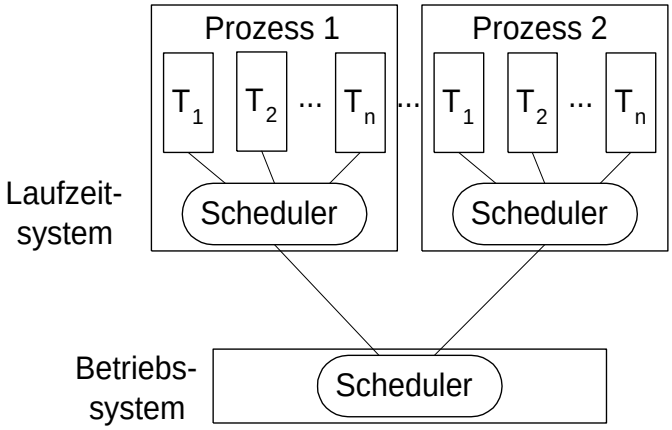


Abbildung 14: User Space Threads

- statt Prozesswechsel nur Austausch der CPU-Register (process counter, Register)
- skaliert sehr gut
- frei wählbare Scheduling Strategie

Nachteile:

- keine Beschleunigung bei Multiprozessor Rechnern
- nur kooperatives Multiprocessing im Thread-Scheduler
- benötigt Reimplementation der Laufzeitumgebung

Hybride Lösung Jeweils N Threads im User-Mode im eigenen Prozess laufen lassen.

Vorteile:

- skaliert gut
- Multiprozessor-fähig

Variabel:

- Anteil „teurer“ Threadwechsel durch Pool-Größe bestimmt

Nachteile:

- Laufzeitumgebung (s.o.)

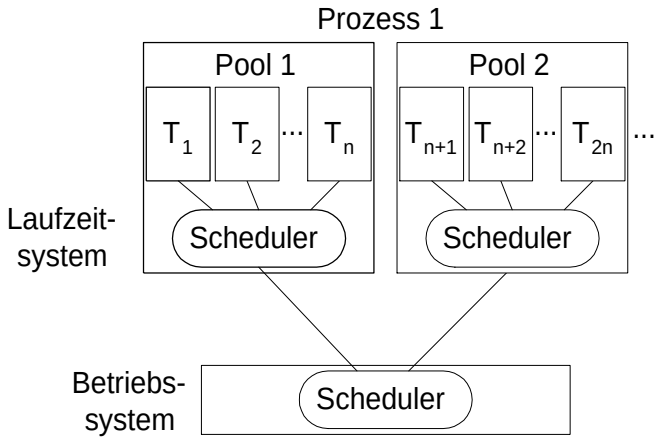


Abbildung 15: Hybrider Ansatz

2.3.3 Nachteile von Threads

Das gesamte Programm inklusive *aller* benutzter Bibliotheken muß Multithread (MT) fähig sein. Das Signal Handling ist zudem kompliziert.

2.4 Prozesskommunikation (IPC)

IPC = interprocess communication

Zweck:

- Prozess-Synchronisation
- Datenaustausch

Synchronisation ist nötig, wenn die Prozesse von einander abhängen, z.B.

- Datenaustausch durch gemeinsamen Speicher (Datei oder „shared memory“)
- Erzeuger-Verbraucher-Problem

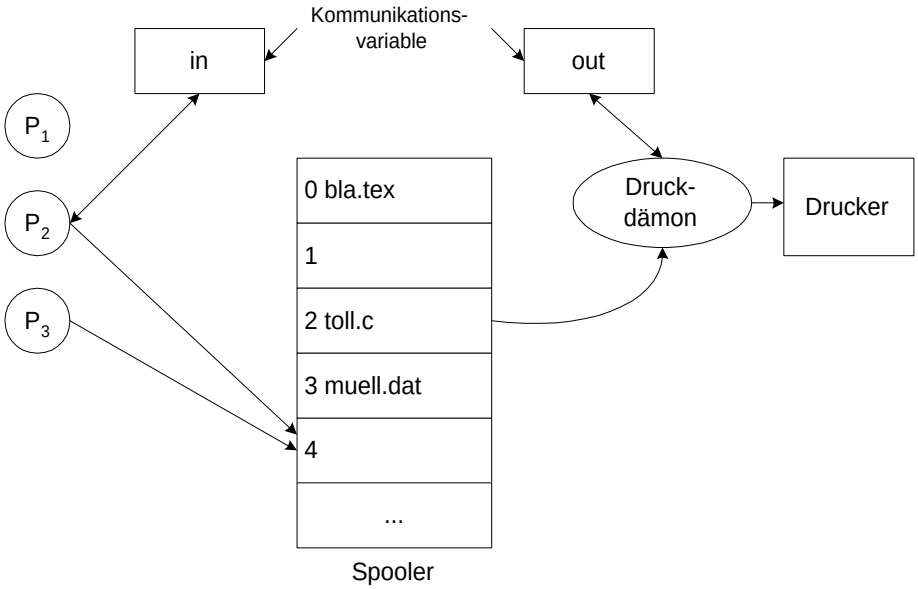
Dabei gibt es:

2.4.1 Zeitkritische Abläufe (race conditions)

„Race conditions“ sind Interaktionen zwischen Prozessen, deren Ergebnis von einer (zufälligen) Bearbeitungsreihenfolge durch das Scheduling abhängt.

Beispiel: shared memory für einen Druckerspooler

Der Ausgang ist nicht-deterministisch und hängt von der Reihenfolge der Prozesse ab!



in	P_1	P_2	P_3	Spooler	out	P_{Druck}
3	liest 3 von in				3	
3	schreibt muell.dat in den Spooler			3: muell.dat	3	
4	schreibt 3+1 nach in			3: muell.dat	3	
4				3: muell.dat	3	liest von out
4				3: muell.dat	3	liest 3 vom Spooler und druckt
4					4	schreibt 3+1 nach out

Ablauf eines unkritischer Spoolauftrages

in	P_1	P_2	P_3	Spooler	out	P_{Druck}
4		liest 4 von in				
4			liest 4 von in			
4			schreibt aha.txt in den Spooler	4: aha.txt		
5			schreibt 4+1 nach in			
5		schreibt grauen.f in den Spooler		4: grauen.f		
5		schreibt 4+1 nach in		aha.txt ist verloren		

Durch den Scheduler wird ein Konflikt beim Drucken verursacht

2.4.2 Wechselseitiger Ausschluß (mutual exclusion)

Race conditions treten nur in „kritischen Bereichen“ eines Prozesses auf

Lösung: Nur jeweils ein Prozess darf in einem kritischen Bereich sein, d.h. Prozesse müssen sich wechselseitig ausschließen.

Methoden des Betriebssystems zum wechselseitigen Ausschließen

Interrupt-Sperre (interrupt disable) Bei Eintritt in einen kritischen Bereich sperrt der Prozess alle (auch Timer-) Interrupts.

Nachteile:

- User Prozess kann CPU beliebig blockieren
- wechselseitiges Ausschließen *aller* kritischen Bereiche nicht nötig

  ⇒ Interupt-Sperre nur für Betriebssystemprozesse

Verdrängungssperre (preemption lock) Sperrt nur andere Prozesse, nicht Interrupts ⇒ kein Prozesswechsel

Vor- und Nachteil:

Interruptbehandlung weiter möglich (auch Interruptbehandlung kann kritischen Bereich enthalten!)

Sperrvariable (spin lock) Idee: Initialisiere eine gemeinsame Sperrvariable „lock“ mit FALSE und setze diese auf TRUE, wenn der Prozess in einen kritischen Bereich gelangt.

Naiver Ansatz:

```
1 noncritical_section();
2 while(lock)
3     ;
4 lock = TRUE;
5 critical_section();
6 lock = FALSE;
7 noncritical_section();
```

Dieses funktioniert nicht, weil die Zeilen 2 bis 4 selbst einen kritischen Bereich bilden. (s. Tabelle 2, S. 28 und Tabelle 3, S. 28)

Da die Kollision allerdings unwahrscheinlich ist, ist es auch schwer, das Problem zu debuggen!

Zeile	P_1	lock	P_2	Zeile
3	weiter	F		
4	setze lock	T		
5	criticalsection	T		
		T	warte	2
		T	warte	2
5	criticalsection	T		
6	setze lock	F		
		F	weiter	3
		T	setze lock	4
		T	criticalsection	5

Tabelle 2: Glücklicher Prozesswechsel

Zeile	P_1	lock	P_2	Zeile
3	weiter	F		
		F	weiter	3
		T	setze lock	4
		T	criticalsection	5
4	setze lock	T		
5	criticalsection	T		

Tabelle 3: Unglücklicher Prozesswechsel

2 Lösungen

1. Hardware-basiert: *(TSL-Instruktion)*

   Die Hardware stellt einen unteilbaren („atomaren“) Maschinenbefehl „Test and Set Lock“ (TSL) bereit, der Lesen und Setzen einer Variablen zusammenfaßt. TSL wird für User als Betriebssystemaufruf gekapselt.

   Anwendung:

   Ersetze Zeile 2 bis 4 durch `while(TSL(lock)); /* (Pseudo-C!) */`

   TSL setzt lock=TRUE und gibt *alten* Wert zurück.

2. Software-basiert: *(Striktes Alternieren)*

   „turn“: gemeinsame Variable zweier Prozesse

   Prinzip: Das Sperren und Entsperren erfolgt jeweils mit *einer* Operation.

```
while(TRUE)                    }
{                              while(TRUE)
    while (turn != 0)          {
        ;                          while (turn != 1)
    /* warte */                        ;
    critical_section();        /* warte */
    turn = 1;                  critical_section();
    noncritical_section();     turn = 0;
```

```
    }                                   noncritical_section();
```

Nachteile:

- schnellerer Prozess muß auf langsameren warten
- „aktives Warten“ verschwendet CPU-Zeit
- Prozesse *müssen* alternieren, d.h. kein Prozess kann zweimal in Folge in einen kritischen Bereich

Petersons Verfahren vermeidet Alternieren, aber nicht aktives Warten:

```
int turn;               /* Gemeinsame Variablen */
int interested[2];

void enter_critical_region(int process_nb)
{
    int other_process_nb;

    other_process_nb = 1 - process_nb;
    interested[process_nb] = TRUE;
    turn = process_nb;
    while (turn == process_nb &&
           interested[other_process_nb] == TRUE)
        ;
}

void leave_critical_region (int process_nb)
{
    interested[process_nb] = FALSE;
}
```

Das schlimmste Problem beim aktiven Warten ist die

Prioritäteninversion zwei Prozesse:

P_H mit hoher Priorität P_N mit niedriger Priorität

Wenn P_N in einem kritischen Bereich ist, und P_H aktiv wartet, teilt der Scheduler P_H alle oder die meiste CPU-Zeit zu. Dann kann P_N gar nicht oder nur sehr langsam weiterkommen. Dadurch wartet P_H immer oder sehr lange.

Lösung: z.B. `while (lock) do { usleep(10000); }`

Allerdings ist dazu eine Annahme über die Geschwindigkeit, bzw. die Zeit, die vom anderen Prozess zum Abarbeiten des kritischen Bereichs benötigt wird, nötig.

2.4.3 Semaphore

(Dijkstra 1965)

Idee: Statt aktiv zu warten legt P_1 sich „schlafen“ (per Systemaufruf) und wird von P_2 durch ein Signal (Systemaufruf) geweckt, sobald P_2 den kritischen Bereich verläßt.

Vorgehen: Integer-Variable i mit zwei unteilbaren Operationen ($i \geq 0$)

down(i) blockiert:

- Wenn $i > 0$ dann setze $i = i - 1$
- Wenn $i = 0$ wird der aufrufende Prozess „schlafen“ gelegt, d.h. aus der Prozesstabelle entfernt und in eine Warteschlange für schlafende Prozesse eingereiht.

up(i) entsperrt:

- Falls $i = 0$ dann schläft mindestens ein Prozess bezüglich i. Also muss aus der Warteschlange für schlafende Prozesse ein Prozess ausgewählt und wieder in die Prozesstabelle eingehangen werden.
- Falls $i > 0$, dann speichere das Wecksignal durch Setzen von $i = i+1$

Beispiel:

```
int mutex = 1; /* gemeinsame Variable */

noncritical_section();
down(&mutex);
critical_section();
up(&mutex);
```

2.4.4 Erzeuger-Verbraucher-Problem

engl.: Producer-Consumer-Problem

Bisher haben wir „mutex“ nur als einen binären Semaphor behandelt. Wozu wären höhere Werte sinnvoll?

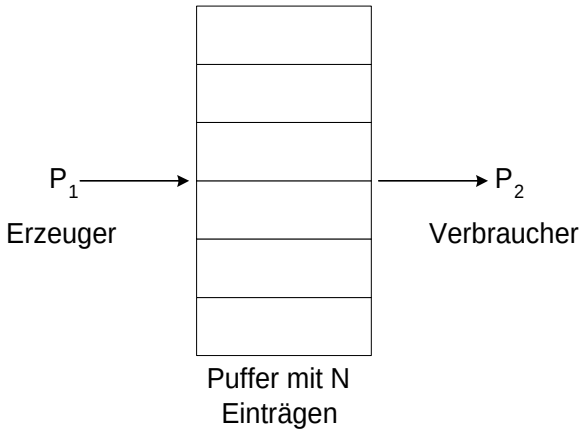
Idee:

- P_1 geht schlafen, wenn der Puffer voll ist
- P_2 geht schlafen, wenn der Puffer leer ist

Producer-Consumer-Problem mit drei Semaphoren:

```
#define N 100      /* Puffergroesse */

int mutex   = 1; /* mutual exclusion */
int n_empty = N; /* Anzahl leerer Pufferplaetze */
int n_full  = 0; /* Anzahl belegter Pufferplaetze */
```




```
void producer()
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&n_empty); /* Warte, wenn kein Platz frei ist */
        down(&mutex);   /* Anfang kritischer Bereich */
        put_item_to_buffer(item);
        up(&mutex);     /* Ende kritischer Bereich */
        up(&n_full);
    }
} /* producer */

void consumer()
{
    int item;

    while (TRUE) {
        down(&n_full);  /* Warte, wenn kein
                           Eintrag existiert */
        down(&mutex);   /* Anfang kritischer Bereich */
        item = get_item_from_buffer();
        up(&mutex);     /* Ende kritischer Bereich */
        up(&n_empty);
        consume_item(item);
    }
} /* consumer */
```

2.4.5 Weitere Kommunikationsprimitiva

Ereigniszähler Integervariable i mit drei Operationen:

- `read(i)` : liefert den Wert i
- `advance(i)` : $i = i + 1$ atomar
- `await(i,j)` : wartet bis $i \geq j$

Producer-Consumer-Problem mit zwei Ereigniszählern:

```
#define N 100; /* Puffergroesse */

int n_in  = 0;  /* Anzahl insgesamt
                   eingefuegter Eintraege */
int n_out = 0;  /* Anzahl insgesamt
                   entnommener Eintraege */

void producer()
{
    int item;
    int n_items = 0;

    while (TRUE) {
        item = produce_item();
        n_items = n_items + 1;
        await(n_out, n_items - N); /* Warte, falls kein
                                      Platz im Puffer */
        put_item_to_buffer(item);
        advance(&n_in); /* Zaehle eingefuegten Eintrag */
    }
} /* producer */

void consumer()
{
    int item;
    int n_items = 0;

    while (TRUE) {
        n_items = n_items + 1;
        await(n_in, n_items); /* Warte, falls kein
                                 Eintrag im Puffer */
        item = get_item_from_buffer();
        advance(&n_out); /* Zaehle entnommenen Eintrag */
        consume_item(item);
    }
} /* consumer */
```

Pipes „Kanal“ zwischen zwei Prozessen

Vorteil: wie Datei-Kommunikation, aber schneller

Nachrichtenaustausch (message passing)

- ist erforderlich, wenn kein gemeinsamer Speicher zwischen den Prozessen existiert, zum Beispiel in einem Rechnernetz.
- dient auch zur Synchronisation:

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

  Addressierung erfolgt zum Beispiel über process@machine.domain

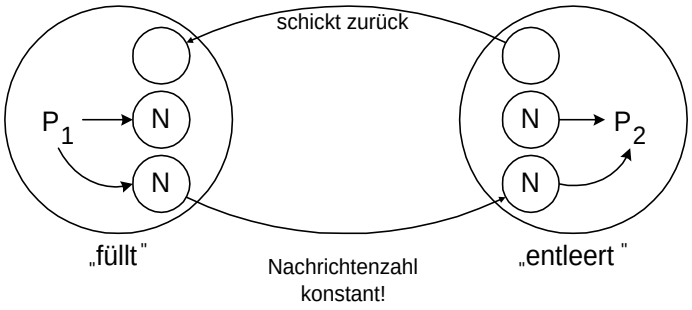


Abbildung 16: Producer-Consumer-Problem mit Nachrichtenaustausch

Producer Consumer Problem mit N Nachrichten:

```
#define N 100

typedef data ...;
typedef message ...;

void producer()
{
    data dat;
    message m;

    while (TRUE) {
        produce_data(&dat);
        receive(consumer_address, &m); /* Warte auf leere
                                          Nachricht */
        put_data_to_message(dat, &m);  /* Fuelle leere
                                          Nachricht */
        send(consumer_address, &m);
    }
} /* producer */

void consumer()
{
    data dat;
```

```
    message m;
    int i;

    /* sende N leere Nachrichten */
    for (i = 0; i < N; i++) {
        send(producer_address, &m);
    }
    while (TRUE) {
        receive(producer_address, &m);
        extract_data_from_message(&m, &dat); /* "entleere"
                                                Nachricht */
        send(producer_address, &m);
        consume_data(&dat);
    }
} /* consumer */
```

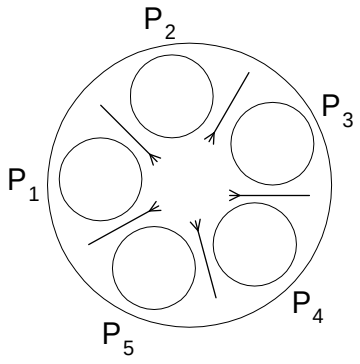
2.5 Klassische Kommunikationsprobleme

2.5.1 Philosophenproblem

(Dijkstra 1965)

Beim Zugriff auf gemeinsame Betriebsmittel kann es zu Synchronisationsproblemen kommen:

Fünf Philosophen sitzen um einen Tisch herum. Jeder Philosoph hat einen Teller mit Spaghetti vor sich. Damit ein Philosoph die Spaghetti essen kann, braucht er zwei Gabeln. Zwischen je zwei Tellern liegt eine Gabel. Das Leben eines Philosophen besteht aus sich abwechselnden Phasen des Essens und des Denkens. Wenn ein Philosoph hungrig wird, versucht er, in einer beliebigen Reihenfolge nacheinander seine linke und seine rechte Gabel aufzunehmen. Hat er erfolgreich beide Gabeln aufgenommen, ißt er eine Weile, legt dann die Gabeln wieder ab und setzt das Denken fort.



Ein entsprechendes Beispiel aus der Rechnerwelt wäre, wenn mehrere Prozesse versuchen auf Floppy und/oder Festplatte zuzugreifen.

Funktion des einzelnen Philosophen:

```
#define N 5 /* philosophers */

void philosopher(int phil_nb)
{
    while (TRUE) {
        think();
        take_both_forks(phil_nb);
        eat();
        put_both_forks(phil_nb);
    }
} /* philosopher */
```

Das Problem liegt jetzt in der Realisierung der Funktion `take_both_forks` und `put_both_forks`. Eine naive Lösung wäre die folgende:

```
void take_both_forks(int phil_nb)
{
    take_fork(phil_nb);   /* left fork */
    take_fork(phil_nb+1); /* right fork */
} /* take_both_forks */

void put_both_forks(int phil_nb)
{
    put_fork(phil_nb);    /* left fork */
    put_fork(phil_nb+1);  /* right fork */
} /* put_both_forks */
```

Die Prozedur `take_fork` wartet, bis die angegebene Gabel frei ist und ergreift sie. Leider hat die offensichtliche Lösung einen Fehler. Falls alle fünf Philosophen gleichzeitig ihre linke Gabel aufnehmen, ist es keinem mehr möglich, auch die rechte aufzunehmen, und es liegt ein Deadlock vor.

Eine kleine Änderung der Prozedur `take_both_forks` kann dieses Problem beheben:

```
void take_both_forks(int phil_nb)
{
    take_fork(phil_nb);
    /* take_fork liefert FALSE, wenn
       die Gabel nicht frei ist */
    while (!take_fork(phil_nb+1)) {
        put_fork(phil_nb);  /* zuruecklegen */
        sleep(100);         /* warten */
        take_fork(phil_nb); /* wieder aufnehmen */
    }
} /* take_both_forks */
```

Bei diesem Versuch führt die Vorgehensweise allerdings zu einer race condition. Aus diesem Grund müssen wir zu einer anderen Methode greifen, um das Problem zu beseitigen:

```
#define THINKING  0
#define HUNGRY    1
#define EATING    2

int state[N]; /* THINKING, HUNGRY, EATING */
int mutex = 1; /* Semaphore, protects state */
int have_forks_sem[N]; /* Semaphores, initialized
                          with 0 */

void take_both_forks(int phil_nb)
{
    down(&mutex);
    state[phil_nb] = HUNGRY;
    take_forks_if_possible(phil_nb);
    up(&mutex);
    down(have_forks_sem + phil_nb);
} /* take_both_forks */

void put_both_forks(int phil_nb)
{
    down(&mutex);
    state[phil_nb] = THINKING;
    put_fork(phil_nb);
    put_fork(phil_nb + 1);
    take_forks_if_possible((phil_nb + N - 1) % N); /* links */
    take_forks_if_possible((phil_nb     + 1) % N); /* rechts */
    up(&mutex);
} /* put_both_forks */

void take_forks_if_possible(int phil_nb)
{
    if (state[phil_nb]               == HUNGRY &&
        state[(phil_nb + N - 1) % N] != EATING &&
        state[(phil_nb     + 1) % N] != EATING)
    {
        state[phil_nb] = EATING;
        take_fork(phil_nb);
        take_fork(phil_nb + 1);
        up(have_forks_sem + phil_nb);
    }
} /* take_forks_if_possible */
```

2.5.2 Leser-Schreiber-Problem

Beispiel: Datenbank

- viele Prozesse dürfen gleichzeitig lesen
- nur ein Prozess darf schreiben und kein anderer Prozess darf zugleich lesen

```
int mutex          = 1; /* Semaphore, protects db */
int db             = 1; /* Semaphore */
int reader_counter = 0;

void reader()
{
    while (TRUE) {
        down(&mutex);
        reader_counter = reader_counter + 1;
        /* Erster Leser schliesst die
           Datenbank bei Eintritt */
        if(reader_counter == 1) { down(&db); }
        up(&mutex);
        read_data_base();
        down(&mutex);
        reader_counter = reader_counter - 1;
        /* Letzter Leser oeffnet die Datenbank */
        if(reader_counter == 0) { up(&db); }
        up(&mutex);
        use_data();
    }
} /* reader */

void writer()
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
} /* writer */
```

2.5.3 Problem des schlafenden Friseurs

In einem Friseurladen gibt es 5 Warteplätze. Wenn der Friseur Kunden im Laden hat, schneidet er ihnen die Haare, hat er keine Kunden im Laden, schläft er. Wenn alle 5 Warteplätze besetzt sind, verlassen neu hinzukommende Kunden wieder den Laden, ohne zu warten.

```
#define N_CHAIRS 5

int n_barbers    = 0; /* Semaphore */
int mutex        = 1; /* Semaphore */
int n_customers  = 0; /* Semaphore:
                         number of waiting customers */
int n_waiting    = 1; /* number of waiting customers
                         (no semaphore!) */
```

```
void barber()
{
    while (TRUE) {
        down(&n_customers);
        down(&mutex);
        n_waiting = n_waiting - 1;
        up(&n_barbers);
        up(&mutex);
        cut_hair();
    }
} /* barber */

void customer()
{
    down(&mutex);
    if (n_waiting < N_CHAIRS) {
        n_waiting = n_waiting + 1;
        up(&n_customers);
        up(&mutex);
        down(&n_barbers);
        get_hair_cut();
    } else {
        up(&mutex);
    }
} /* customer */
```

3 Speicherverwaltung

Teil des Betriebssystems, das den physischen Speicher verwaltet.

Aufgabe:

- Verwaltung freier und belegter Speicherbereiche, Zuweisung an Prozesse, Freigabe, gegebenenfalls Auslagerung auf Sekundärspeicher
- Speicherverwalter sorgt für Systemstabilität, da die Prozesse getrennten Speicher zugewiesen bekommen
- Speicherverwaltung ist aus Geschwindigkeitsgründen eng an die Hardware gekoppelt

3.1 Speicherverwaltung ohne Swapping und Paging

3.1.1 Einprogrammbetrieb ohne Swapping und Paging

Frühe Rechner liefen komplett unter Kontrolle *eines* Prozesses, der die gesamte Kontrolle über Rechner und Speicher hatte.

besser: Teilung von Betriebssystem und einem Prozess, etwa ein IBM-PC unter DOS

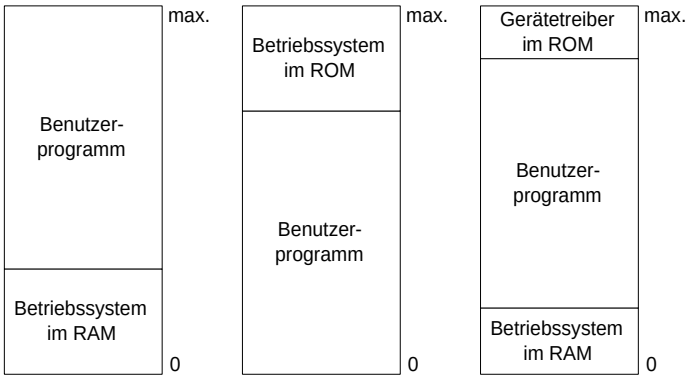
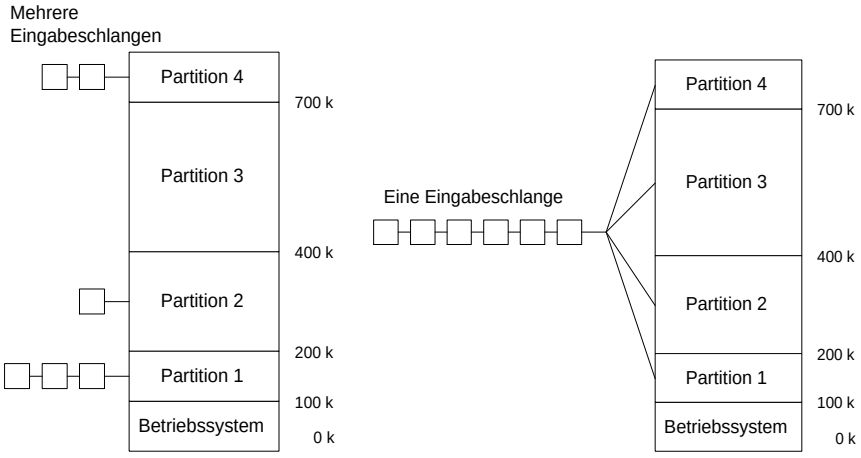


Abbildung 17: Drei Arten der Speicherorganisation mit dem Betriebssystem und einem Benutzerprozess

3.1.2 Mehrprogrammbetrieb mit fixen Partitionen

Beim Start des Rechners werden mehrere feste Partitionen eingerichtet, Prozesse werden in einer getrennten (Abb. 18(a)) oder einer gemeinsamen (Abb. 18(b)) Warteschlange gehalten.



(a) Feste Speicherpartitionen mit getrennten Eingabeschlangen für jede Partition

(b) Feste Speicherpartitionen mit einer einzigen Eingabeschlange

Abbildung 18: Mehrprogrammbetrieb mit fixen Partitionen

Nachteil: mögliche Speicherverschwendung

Implementiert wurde diese Methode etwa unter IBM Mainframe (OS/360)

Implikation der verschiedenen Basisadressen führt zu einer Relokation (Sprungadressen müssen angepaßt werden). Diese ist möglich durch

(a) Modifikation des Programmcodes beim Laden

(b) Basisadressen im Registersatz (Hardwarelösung)

Speicherschutz: Verhindern von Speicherübergriffen der verschiedenen Prozesse, zum Beispiel mit Teilung des Speichers in Blöcke (z.B. 2kB) und Zuweisung eines prozesseigenen Schutzcodes zu jedem Block.

alternativ: Hardwarelösung mit Grenzregistern

3.2 Swapping

Auf Multiusersystemen existieren oft viele Prozesse (bei interaktiver Bedienung, Terminals). Dadurch genügt der Speicher nicht, um alle Prozesse darin aufzunehmen.

Lösung: Prozesse auslagern auf einen Sekundärspeicher

3.2.1 Mehrprogrammbetrieb mit variablen Partitionen

Flexible Größen der Partition verbessern die Speicherausnutzung, *aber* dadurch entsteht auch eine komplizierte Allokation und Deallokation des Speichers. Über längere Zeit entstehen Speicherlücken, die durch Verschiebung von Prozessen in *ein* Loch entfernt werden (Speicherverdichtung). Diese Verschiebung ist allerdings aufwendig.

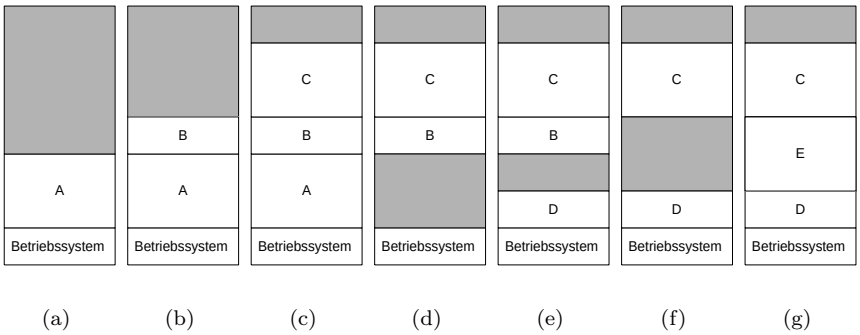
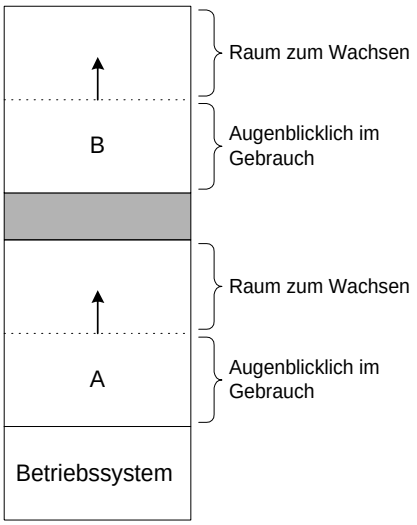


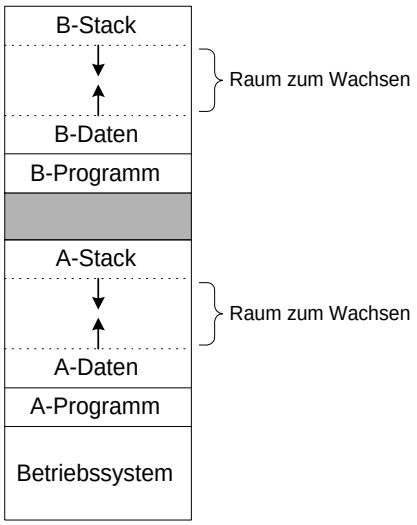
Abbildung 19: Die Speicherallokation ändert sich, wenn Prozesse in den Speicher kommen oder ihn verlassen. Die schattierten Bereiche sind unbenutzter Speicher

Probleme gibt es auch bei dynamischen Speicherallokationen (Heap/Stack).

Speicherverwaltung geschieht in diesem Zusammenhang oftmals mit Hilfe von



(a) Allokation von Speicher für wachsende Datensegmente



(b) Allokation von Speicher für einen wachsenden Stack und wachsende Datensegmente

- Listen
- Bitmaps
- Buddy-System (Abb. 20, S. 41)

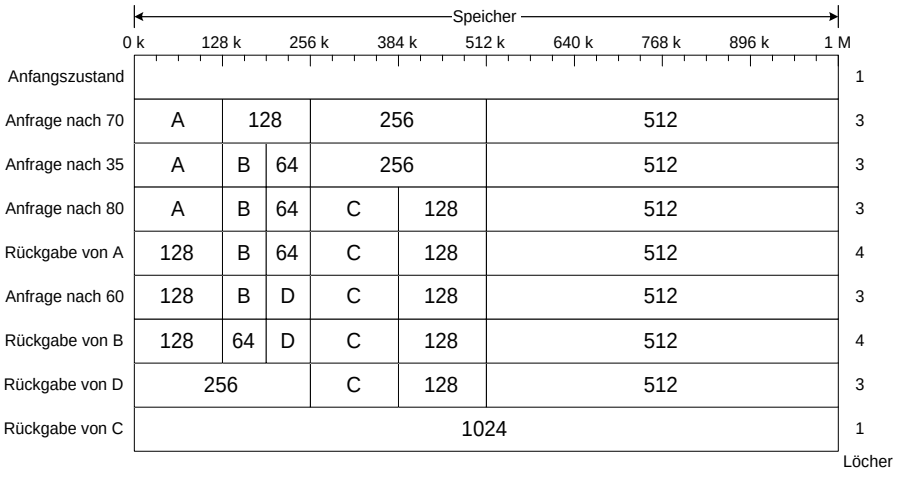


Abbildung 20: Das Buddy-System. Die horizontalen Achsen repräsentieren die Speicheradressen. Die Nummern sind die Größe der nichtallokierten Speicherblöcke in kB. Die Buchstaben stellen allokierte Speicherblöcke dar.

3.3 Virtueller Speicher

Problem: Prozesse sind einzeln zu groß für den Hauptspeicher

Frühere Lösung: Überlagerungen (aufwendig für Programmierer)

Heutige Lösung: Virtueller Speicher. Das Betriebssystem sorgt dafür, dass die nötigen Programmteile im Hauptspeicher sind, der Rest befindet sich auf der Festplatte.

3.3.1 Paging

Das Programm verwendet „virtuelle Adressen“ in einem virtuellen Adressraum. Eine Speicherverwaltungseinheit (MMU) im Rechner sorgt dafür, dass die Adressen auf den physischen Speicher abgebildet werden. Die virtuellen Adressen gelangen nicht auf den Adressbus!

Der virtuelle Speicher ist in Seiten eingeteilt, im physischen Speicher heißt die korrespondierende Speichermenge „Seitenrahmen“. Beide haben eine konstante Größe (4kB bei i386 Linux). Die Auslagerung geschieht dann immer seitenweise, gekennzeichnet werden die Auslagerungen durch eine Present/Absent Flag in der Seitentabelle. (siehe Abbildung 21, S. 43)

Das Ansprechen ausgelagerter Seiten verursacht per Hardware (MMU) einen Seitenfehler (page fault). Dann wählt das Betriebssystem eine wenig benutzte Seite aus, und ersetzt sie durch die angeforderte. Dabei kommt es zu:

- Änderung der Seitenliste
- Wiederaufnahme des unterbrochenen Programms

Zusammenfassung: Probleme der Speicherverwaltung

1. Architekturbedingt:
   - adressierbarer Bereich ist kleiner als das Programm, das darin laufen soll
   - vorgegebene Aufteilung des Adressraums (z.B. bei MS-Dos)
2. durch Multiprozessbetrieb bedingt
   (a) dynamische Verwaltung belegter und freier Bereiche
   (b) Swapping:
   Auslagerung von Prozessen oder Prozessteilen auf Platte, falls der Speicher zu knapp wird
   (c) Adressierung
   Anpassung der Adressreferenzen eines Programms an physische Adresse durch:
   - Übersetzung virtueller in physische Adressen mit Übersetzungshardware (memory managment unit - MMU) oder

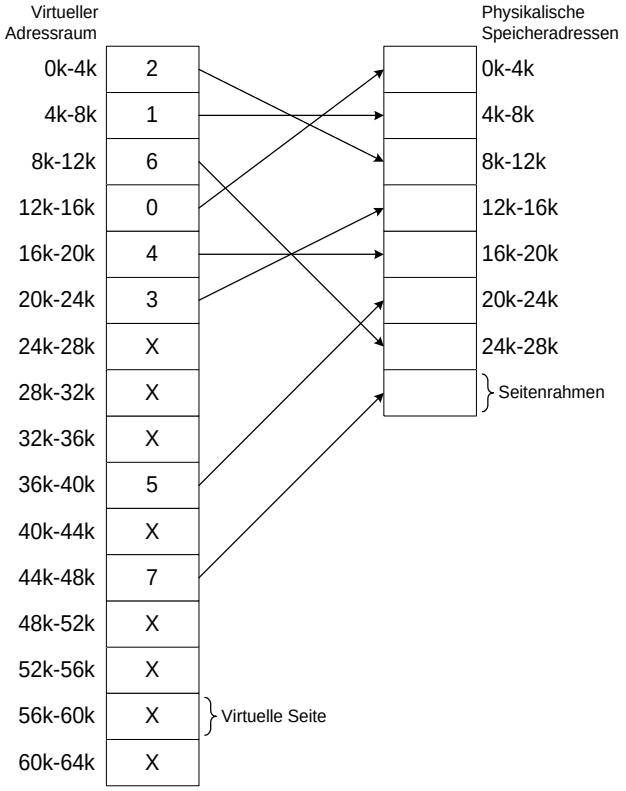


Abbildung 21: Die Beziehung zwischen virtuellen Adressen und physischen Speicheradressen wird durch eine Seitentabelle gegeben

- Relokation:
  modifiziere absolute Adressreferenzen eines Programms beim Laden
- Alternative: *position independent code*
  Compiler erzeugt alle Adressreferenzen relativ zum Stand des Befehlszählers

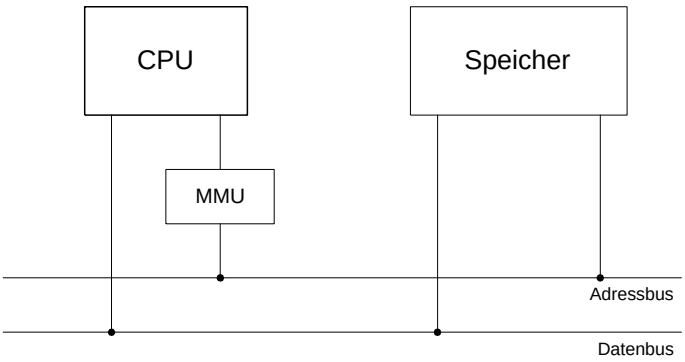
(d) Schutz:
Prozess darf nur auf eigenen Speicherbereich zugreifen

Lösung: Virtueller Speicher

Paging

1. Unterteile den physischen Speicher in einen „Seitenrahmen“ gleicher Größe (ca. 0,5–32 kB)
2. Unterteile den virtuellen Adressraum in „Seite“ entsprechender Größe
3. Ordne Text- bzw. Datenseiten der Prozesse dynamisch den Seitenrahmen zu

Adressierung Der Prozess arbeitet mit virtuellen Adressen, die von der MMU in physische transformiert werden.



Vorteile:

- es ist möglich, den Prozessen mehr Speicher zur Verfügung zu stellen, als in der Realität physisch vorhanden ist (sowohl allen Prozessen als auch einem einzigen)
- nur die benötigten Seiten eines Prozesses müssen von der Platte geladen werden
- der virtuelle Adressraum eines Prozesses ist unabhängig von der realen Speichergröße/-aufteilung
- Schutz der virtuellen Adressbereiche untereinander
- Swap-Vorgänge feingranular und bedarfsgesteuert

3.3.2 Adressübersetzung

Funktion der MMU:

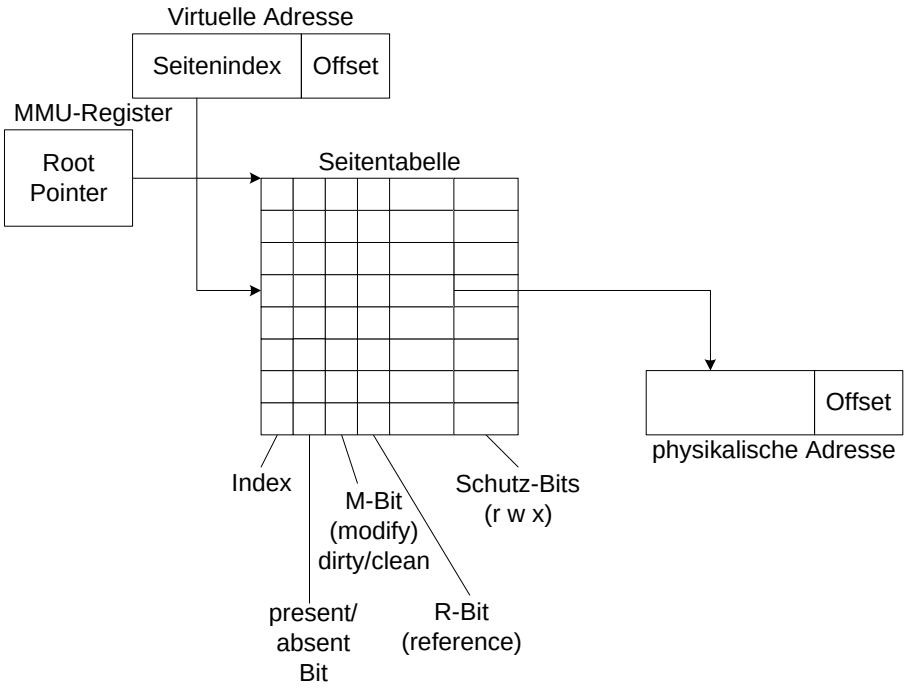
Falls die benötigte Seite nicht im Speicher ist („absent flag“), kommt es zu einem „page fault“, dadurch wird in das Betriebssystem gesprungen, das einen Seitenrahmen freimacht (falls Seite „direkt“ auslagern). Danach wird die benötigte Seite geladen und die Seitentabelle aktualisiert.

Probleme:

- Größe der Seitentabelle

  z.B. 32-Bit Adressraum (ca. 4 GB)

  Seitengröße 4 kB: 20 Bit Seitenindex (1M Seiten) + 12 Bit Offset
- Jeder Prozess hat eine eigene Seitentabelle



- Konflikt:

Anteil der Seitentabelle in MMU-Registern	hoch	niedrig
Zugriffsgeschwindigkeit	hoch	niedrig
Ladeaufwand bei Prozesswechsel	hoch	niedrig

Lösungen

A Mehrstufige Seitentabellen

Beispiel: 2-stufig im 32-Bit Adressraum mit 4kB Seiten.

Für jeden Prozess ist nur die Basistabelle vorhanden, die Seitentabellen werden nach Bedarf benutzt.

2–4 Stufen, Übersetzungsvorgang („table walk“) ist entweder in der MMU mikrokodiert (z.B. Motorola 680x0, Intel x86, SPARC) oder wird von der CPU ausgeführt (z.B. RISC)

B Translation Lookaside Buffer (TLB) („Assoziativspeicher“)
Prozesse brauchen nur weinge Seiten, diese aber sehr oft. Daraus ergibt sich eine Beschleunigung mit Hilfe des TLB in der MMU. Der TLB enthält Kopien der zuletzt benutzten 8–32 Einträge der Seitentabelle. Bei einem Zugriff wird zuerst der TLB und erst dann die Seitentabelle durchsucht.

Die Performance hängt bei diesem Verfahren ab von:

- der Trefferrate

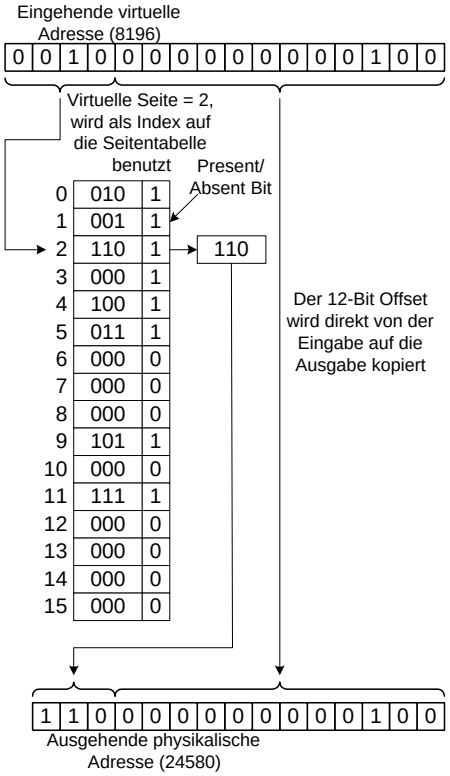
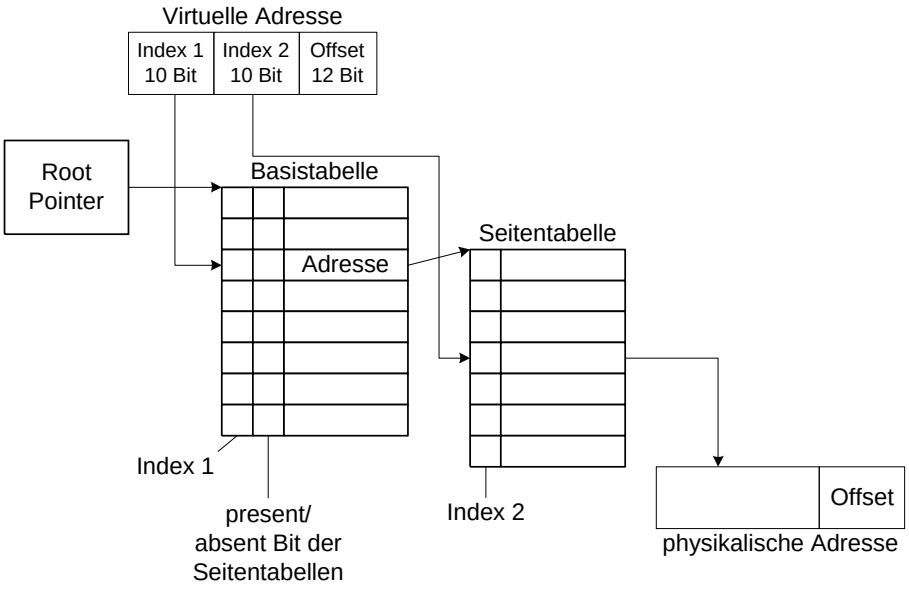
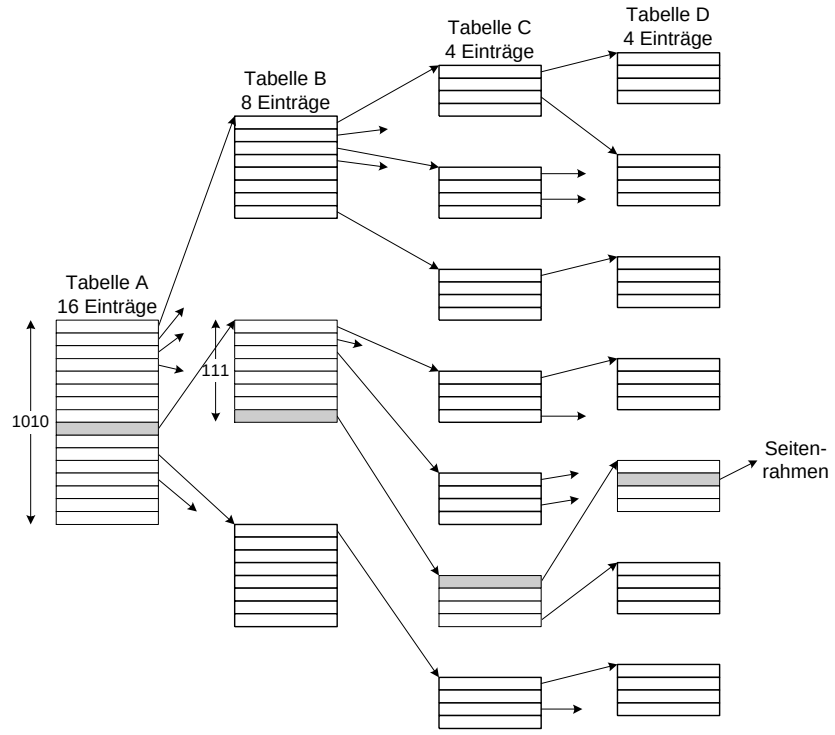


Abbildung 22: Übersetzung einer Adresse durch die Seitentabelle





- den Zugriffszeiten auf den TLB bzw. die Seitentabelle

Bei einem Prozesswechsel muss entweder der TLB gelöscht werden oder der TLB für den Prozess geladen werden.

C Invertierte Übersetzungstabelle

Diese ist notwendig, wenn der virtuelle Adressraum zu groß ist (z.B. 64 Bit).

D Virtuelle Seitentabelle

Die Seitentabelle selbst ist im virtuellen Speicher enthalten. Damit werden Mechanismen des virtuellen Speichers selbst benutzt anstatt einer expliziten Strukturierung mehrstufiger Seitentabellen.

Aber:

- rekursives Problem:
  muss vom Betriebssystem auf unterster Stufe der Rekursion gelöst werden
- table walk per Hardware ist schwierig
  ⇒ Software table walk z.B. MIPS-Prozessoren

3.3.3 Seitenverwaltung

Wenn der Hauptspeicher knapp wird, werden Strategien benötigt für:

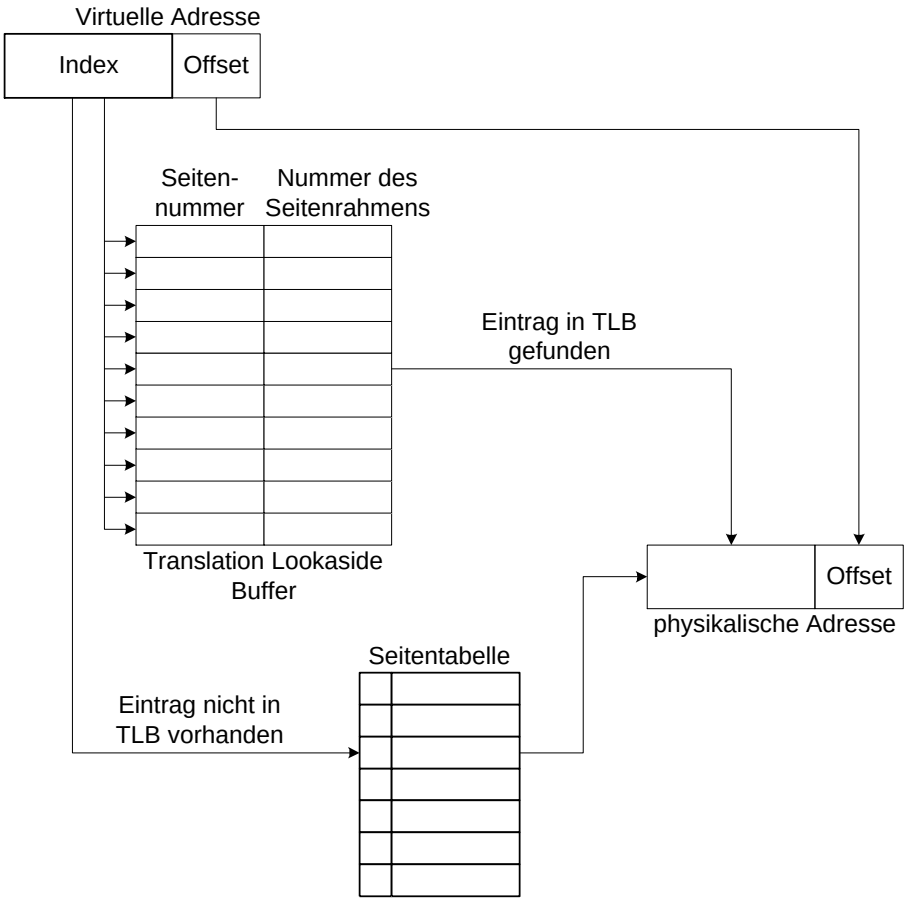


Abbildung 23: Translation Lookaside Buffer

1. Einlagerungszeitpunkt

   - rein bedarfsgesteuert (demand paging)
   - spekulativ (prepaging, prefetching)

2. Auslagerungszeitpunkt

   - nur auslagern, wenn Auslagerung unabwendbar ist, d.h. wenn kein Rahmen mehr frei ist

     Achtung: für den Auslagerungsvorgang selbst wird eine Speicherreserve benötigt

   - spekulativ

3. Seitenersetzungsstrategie

   wählt bei einem page fault die zu ersetzende Seite aus.

   Ziel: Wähle die Seite, die in naher Zukunft nicht gebraucht wird.

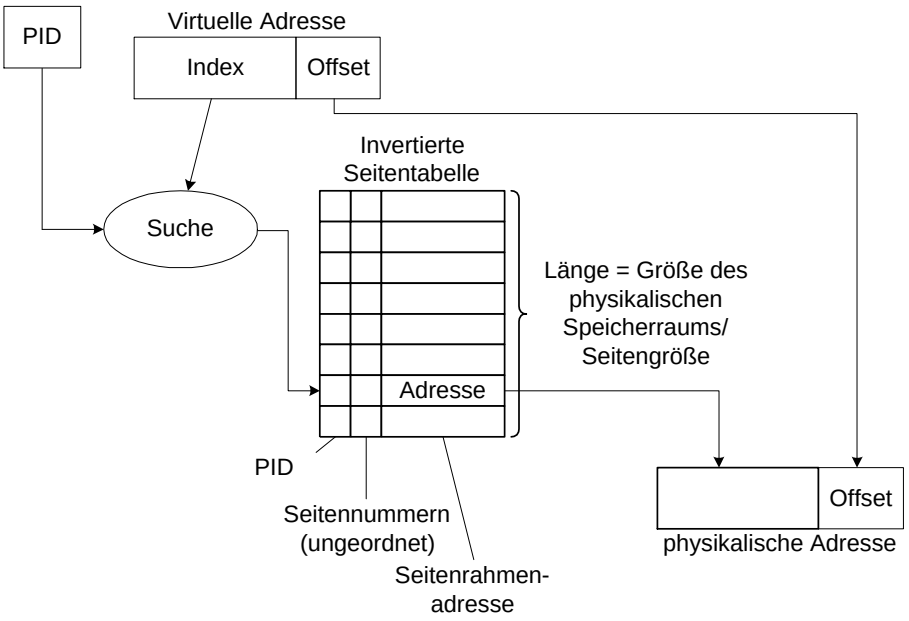


Abbildung 24: Invertierte Seitentabelle

3.3.4 Seitenersetzungsalgorithmen

Problem: Sage aus vergangenem Gebrauch die Seite voraus, deren Wiedergebrauch am weitesten in der Zukunft liegt.

Informationen dafür:

- Zugriffsinformation:

  Referenz-Bit (R-Bit): 1, falls die Seite referenziert wurde (lesend oder schreibend)

- Zustand:

  Modifikations-Bit (M-Bit): 1, falls die Seite modifiziert wurde

- Einlagerungszeitpunkt
- Prozesszugehörigkeit

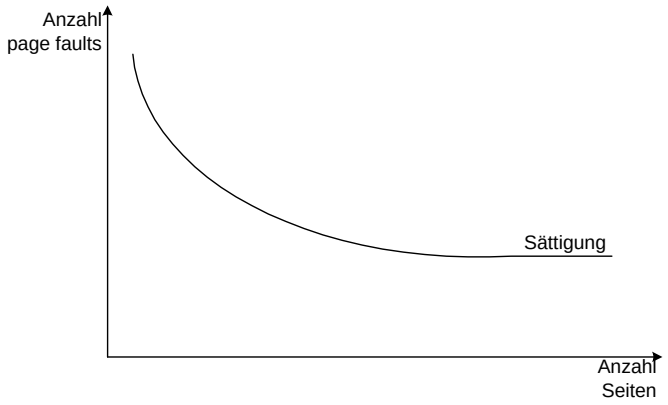
Allgemein gibt es zwei Strategie-Typen:

lokal berücksichtigt die Prozesszugehörigkeit

global wählt aus allen Seiten aus z.B. Page Fault Frequency (PFF) Algorithmus

Page Fault Frequency weist jedem Prozess individuelle optimale (im Sinne des Gesamtsystems) Seitenzahl zu.

Ersetzungsalgorithmen



Not-Recently-Used (NRU) Periodischer Reset des R-Bit, z.B. bei einem timer interrupt.

Entferne Seite mit niedrigster Priorität:

Priorität	R-Bit	M-Bit
0	0	0
1	0	1
2	1	0
3	1	1

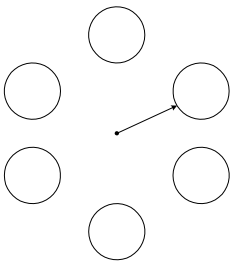
FIFO: Führe eine Seitenliste, die nach Alter der Seiten sortiert ist. Die „Älteste“ Seite wird dann bei einem page fault ersetzt.

Fehler: auch ständig benutzte Seiten werden entfernt

Second-Chance Funktionsweise wie FIFO, aber, falls das R-Bit gesetzt ist für die älteste Seite, wird diese Seite ans Ende der Liste gesetzt (“Verjüngung“) und das R-Bit auf 0 gesetzt.

bessere Implementierung statt linearer Liste:

Clock Zyklische Liste



Es gibt einen Zeiger auf die älteste Seite. Ist bei dieser das R-Bit ungesetzt wird diese gelöscht. Ist es gesetzt, wird der Zeiger um einen Eintrag weiterversetzt.

Least-Recently-Used (LRU) Entferne die Seite, die am längsten unbenutzt ist.

Implementierung: verkettete Liste aller Seiten, die jüngst referenzierte steht am Anfang

Problem: Aktualisierung bei *jeder* Referenz (Suchen und Ersetzen)

Lösungen:

1. Spezial-Hardware

   - Ein 64-Bit-Zähler sowie ein 64-Bit-Feld an jedem Seiteneintrag. Nach einer Referenz wird der Zähler inkrementiert und in das Feld der Seite kopiert. Bei einem page fault wird nach dem kleinsten Wert im Feld gesucht.
   - Matrix

     Wir machen die Funktionsweise an einem Beispiel mit 3 Seiten deutlich:

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0

     Die Ausgangsmatrix. Zu jeder Seite gibt es eine Zeile und eine Spalte. Jetzt wird auf Seite 0 referenziert:

	0	1	2
0	0	1	1
1	0	0	0
2	0	0	0

     Die Bits in der Zeile der Seite, auf die referenziert wird, werden in den Spalten mit den beiden nicht referenzierten Seiten gesetzt. Nun wird auf Seite 1 referenziert:

	0	1	2
0	0	0	1
1	1	0	1
2	0	0	0

     Nach einer Referenzierung auf Seite 2 entsteht folgende Matrix:

	0	1	2
0	0	0	0
1	1	0	0
2	1	1	0

     Ein weiterer Zugriff auf Seite 1 führt zu dieser Matrix:

	0	1	2
0	0	0	0
1	1	0	1
2	1	0	0

Bei einem page fault wird die Seite mit den meisten ungesetzten Bits in der entsprechenden Zeile ersetzt. Für die letzte Matrix wäre das offensichtlich die Seite 0.

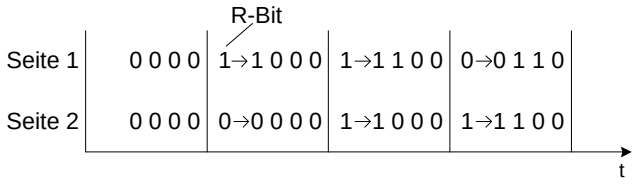
2. Approximation durch Software
   - Not-Recently-Used (grob)
   - Not-Frequently-Used (feiner)

Not-Frequently-Used (NFU) Für jede Seite gibt es einen individuellen Zähler. Das R-Bit wird bei jedem timer interrupt aufaddiert und zurückgesetzt. Bei einem page fault wird die Seite mit dem kleinsten Zähler ersetzt.

Problem: Gleichgewichtung alter und neuer Referenzen

Lösung: Gewichtung durch *Aging*

- bei jedem timer interrupt werden alle Bits aller Zähler um eins nach rechts verschoben
- am höchstwertigen Bit (links) wird das R-Bit aufaddiert
- bei einem page fault wird die Seite mit dem kleinsten Wert ersetzt



Aber:

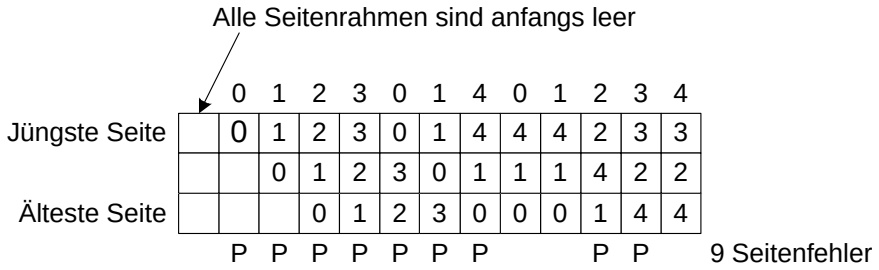
- totales Vergessen nach durch Anzahl der Bits festgelegter Zeit
- 100000 wird erst nach 011111 ersetzt

3.3.5 Theoretische Überlegungen zu Seitenersetzungsalgorithmen

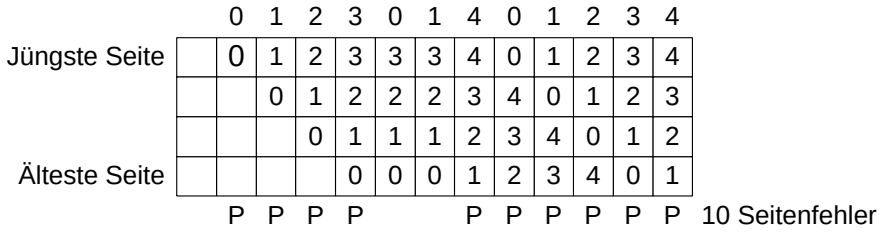
Belady's Anomalie Für ein System mit fünf Seiten existiert eine Referenzierungsreihenfolge, für die 4 Seitenrahmen *mehr* page faults erzeugen als 3 Seitenrahmen.

Stack-Algorithmen Zur Vereinfachung nehmen wir eine Maschine mit nur einem Prozess.

1. kein nicht-deterministischer Einfluss des Scheduling
2. kein „Strukturwechsel“ in den Referenzierungen bei einem Prozesswechsel



(a)



(b)

Abbildung 25: Belady's Anomalie

Dann hängt das Paging nur ab von:

- Referenzzeichenkette (= Folge referenzierter Seitennummern) eines Prozesses
- Algorithmus
- Zahl m verfügbarer Seitenrahmen

Modell der Ersetzungsverwaltung beispielsweise für den Least-Recently-Used Algorithmus:

Setze eine referenzierte Seite S in den ersten Eintrag von M. Falls S noch nicht in M vorhanden ist, setze alle anderen um eins nach unten. Sonst setze alle Seiten oberhalb von S um eins nach unten. (vgl. Abb. 26, S. 54)

LRU gehört zu den theoretisch bevorzugten, sog. Stack-Algorithmen, da alle für m Rahmen nach r Referenzen präsenten Seiten auch bei m+1 Rahmen nach r Referenzen präsent sind. Es liegt also keine Belady-Anomalie vor.

Distanzzeichenkette Kodiere die Referenzzeichenkette um die Distanz d der referenzierten Seite im Feld M zum Beginn von M. d hängt vom Algorithmus ab.

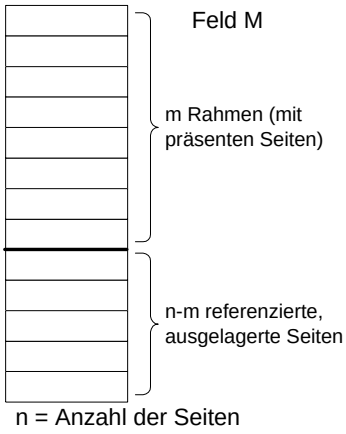


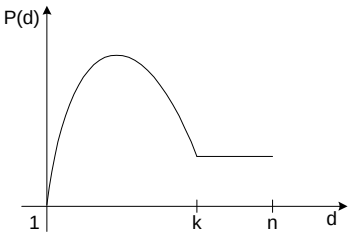
Abbildung 26: Aufbau der Ersetzungsverwaltung

Referenzzeichenkette	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Seitenfehler	P	P	P	P	P	P	P		P					P		P							P	
Distanzzeichenkette	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

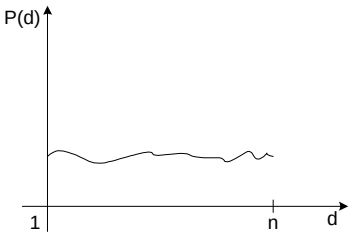
Abbildung 27: Ersetzungsverwaltung mit einer Distanzzeichenkette

Nutzen der Distanzzeichenkette:

1. Wahrscheinlichkeitsdichte P(d) verschiedener Distanzen geben Hinweis auf eine sinnvolle Rahmenzahl



(a) k Seitenrahmen genügen



(b) möglichst viele Seitenrahmen

2. Abschätzung Nummer von page faults in Abhängigkeit von der Speichergröße

   Berechne Histogramm H der Distanzzeichenkette

   $$H(i) = \sum_{j=1}^{N} \delta_{i,d(j)}$$

   Wobei N die Anzahl der Referenzen ist.

   Das heißt, dass $H(i)$ die Anzahl der Elemente von d mit Wert ist. Berechne die Anzahl der page faults $F(m)$ für m Rahmen:

   $$F(m) = \left(\sum_{j=m+1}^{n} H(j) \right) + H(\infty)$$

   n ist die Anzahl der Seiten

Aber:

- gilt nur für eine bereits bekannte Distanzzeichenkette d
- statistische Modellierung von d schwierig, weil d nicht zufällig ist
- Bewertung eines Algorithmus muss neben „Struktur“ von d auch m und n berücksichtigen
- Alternative: Individuelle Algorithmenwahl für einzelne Probleme

3.4 Segmentierung

vgl. Einschub 2.1.5

Bisher haben wir nur 1-dimensionalen Speicher behandelt.

Virtueller Speicher ermöglicht zwar die Nutzung des gesamten adressierbaren Bereichs, allerdings war dieser bisher ohne jede Strukturierung.

Daher *Segmentierung*:

- Technik zur 2 dimensionalen Strukturierung virtuellen oder physischen Speichers mit Segmenten variabler Länge
- Segmente dienen als von einander unabhängige Adressräume, die jeweils bei „0“ beginnen
- jedes Segment beinhaltet nur einen „Objekttyp“
- jedes Segment hat individuelle Zugriffsrechte (r w x)

Vorteile:

- Trennung logisch voneinander verschiedener Programmteile in verschiedenen Adressräumen z.B. Code, Daten, Konstanten, Stack

- dadurch Schutz, z.B. können Daten nicht in Codesegment geschrieben werden
- das Betriebssystem nimmt Programmierer Verwaltungsaufwand für Datenfelder variabler Größe ab
- Effiziente Speicherverwaltung, das Betriebssystem kann unsichtbar für Prozesse:
  1. wachsenden Segmenten den Platz schrumpfender zuweisen
  2. Segmente umsortieren
  3. Segmente verdichten (s. Abb. 28, S. 56)

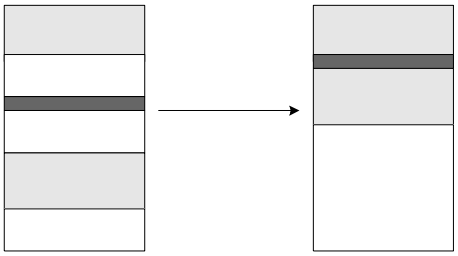


Abbildung 28: Segmentverdichtung

- verschiedene Prozesse können Daten oder Prozeduren[1] gemeinsam nutzen, z.B. in „shared libraries“ verpackte Grafik-Prozeduren

Kombination Seitenersetzung und Segmentierung *Unterschiede*

- **Seitenrahmen** sind die fixe Unterteilung physischen Speichers, um **Seiten** fester Größe zeitweise auf verschiedenen physischen Medien abzulegen
- **Segmente** sind die variable Aufteilung des Adressraumes. Ein oder mehrere Segmente bilden den logischen Adressraum eines Prozesses

Die 4 entsprechenden Speicherverwaltungsmodelle benutzen diese Strategien wie folgt:

	Segmentierung	Paging	
FLAT	-	-	logische = lineare = physische Adresse
Segmented	+	-	logische → lineare = physische Adresse
Paged	-	+	logische = lineare → physische Adresse
Segmented & Paged	+	+	logische → lineare → physische Adresse

[1] die in eigenen Segmenten sind

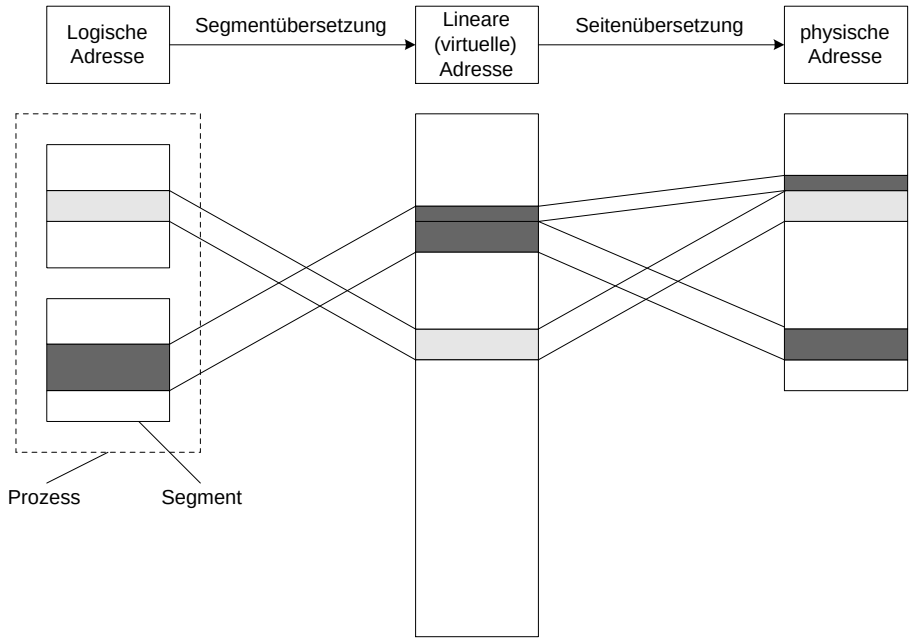


Abbildung 29: Adressübersetzung

4 Dateisystem

Datei (file):

- Sammlung von Daten auf Permanentspeicher
- Abstraktionsmechanismus, um Daten für Benutzer unabhängig von Details der Speicherung zugänglich zu machen

Vorteile des Permanentspeichers:

- groß und billig
- Datenerhaltung bei Prozessende oder Rechnerausfall
- mehrere Prozesse können gleichzeitig auf dieselben Daten zugreifen

Das Betriebssystem stellt dem Benutzer ein logisches Dateisystem zur Verfügung und verwaltet das physische Dateisystem.

4.1 Logisches Dateisystem

Dateien werden in Verzeichnissen (directories) eingetragen. Da Verzeichnisse i.A. selbst Dateien mit vom Betriebssystem definierter Semantik sind, ist eine Kaskadierung der Verzeichnisse möglich.

4.1.1 Dateien

Bennenung:

- durch Zeichenketten (mindestens 8 Zeichen werden als maximale Länge von allen Betriebssystemen unterstützt)
- UNIX unterscheidet Groß-/Kleinschreibung, MS-Dos nicht
- üblich: *filename.extension*

  Extension kennzeichnet den Typ der Datei, z.B.

.c	C-Quellcode
.gif	Bild im GIF-Format
.jpg	Bild im JPEG-Format
.tex	LaTeX/TeX
.txt	Text
.gz	mit gnu-Zip (gzip) komprimiertes file

  Manche Programme verlangen eine passende Extension, andere nicht

Lineare Dateistrukturierung ist eine Frage der Interpretation durch den Prozess

1. Strukturlos, d.h. die Interpretation findet nur durch das Anwendungsprogramm statt
2. Strukturierung in „records“ konstanter Länge, Daten I/O immer in Einheiten von ganzen records

   Dieses System hat seine Grundlagen im Lochkartenzeitalter
3. records variabler Länge werden vom Betriebssystem (nicht vom User) angeordnet. Jeder record besitzt ein „Schlüsselfeld“, das den direkten Zugriff ohne Kenntnis der Position ermöglicht. Die Verwaltung durch das Betriebssystem geschieht mittels einer Baumstruktur.

Aufbau einer Datei *Problem:* Eine Datei ist nur eine Folge von Bytes. Die Semantik wird durch das erzeugende oder benutzende Programm definiert. Woran erkennt das Programm, ob der Typ auch paßt?

Unterscheide z.B. gewöhnliche Dateien von Verzeichnissen und speziellen Dateien (etwa Gerätedateien)[2]

Lösungen:

1. Kodierung in der Extension. Damit ist die Extension auch verpflichtend.

   Nachteil dieser Lösung: Wenn z.B.

   (a) Programmdateien (nur kompilierbar) und

   (b) ASCII-Dateien

[2]Eine minimale Anforderung, die wir stellen müssen, ist die Unterscheidung von nicht-ausführbaren Dateien von ausführbaren!

unterschieden werden sollen, ergibt sich das Problem, dass die Programmdateien eine Untermenge der ASCII-Dateien sind.

2. Dateiattribut ausführbar/nicht ausführbar

   Attribute sind Infos, die das Betriebssystem (versteckt) zu jeder Datei speichert.

   Weitere Beispiele für Attribute (nicht inhaltsbezogen) sind:

   - erzeugender User
   - besitzender User
   - Zugriffsrechte:

     z.B. unter UNIX:

r w x	r w x	r w x
Besitzer	Gruppe	Welt

   - Passwort
   - Erzeugungszeit
   - Zeit der letzten Änderung
   - Größe
   - System-Datei
   - temporär (d.h. bei Prozessende wird die Datei gelöscht)
   - versteckt (Auftreten in Listing)

3. Typ-Kodierung in Kopf (Header)

   Header steht *in* der Datei vor dem eigentlichen Inhalt (s. Abb. 30, S. 60)

Unter UNIX findet sich Typ 3 (ziemlich sicher) und Typ 1 und 2 (veränderbar durch den User). Bei Apple findet sich Typ 2 (sicher).

Zugriff auf Dateien

sequentiell stammt aus der Zeit der Magnetbänder. Die Daten können nur vom Anfang an nacheinander gelesen werden. Die einzige Positionierungsoperation ist Rewind (zurückspulen).

direkt Zugriff kann in beliebiger Reihenfolge und an jeder beliebigen Stelle geschehen. Nötig ist dieses z.B. in Datenbanken

Operationen

create Erzeugung einer Datei, Festlegung von Namen und Attributen

delete Platz freigeben

rename Dateinamen ändern

open Einlesen der Attribute und Plattenadressen vor dem ersten Zugriff

close Freigabe dieses Speichers

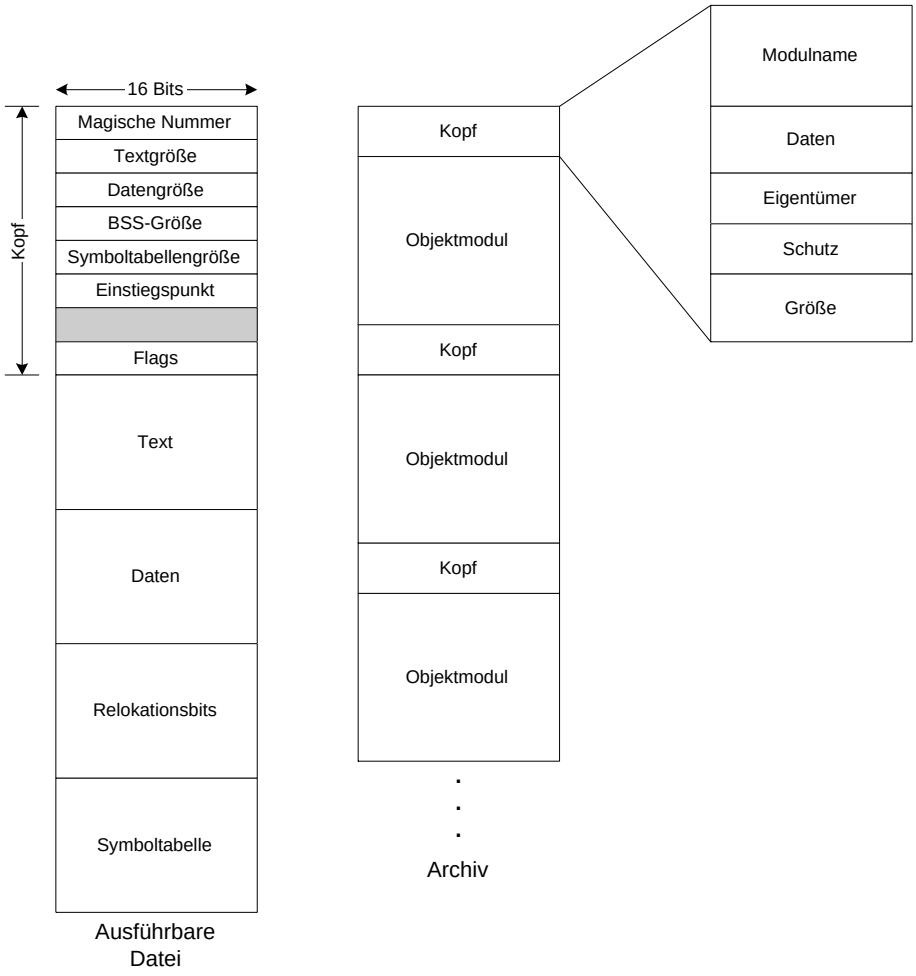


Abbildung 30: Typ-Kodierung im Datei-Header

read Entweder an einer beliebigen Position oder an der aktuellen Position. Die Datenlänge ist das Argument dieser Operation.

write (Über-)Schreiben

append Funktionsweise wie write, es ist aber nur möglich, Daten am Ende anzufügen

seek Setzen der aktuellen Position bei einem direktem Zugriff falls Read sich auf diese bezieht

Get/Set Attributes Lesen/Schreiben von Dateiattributen

Map/Unmap Speicherabbildung der Datei

Lade die Datei in den Hauptspeicher für einen schnelleren Zugriff

Intern: Datei einfach als Hintergrundspeicher umdeklarieren. Der erste Zugriff verursacht dann einen page fault, so dass ein Teil der Datei in

Form einer Seite geladen wird. Bei Segmentierung erhält die Datei ein eigenes Segment.

Problem: Verändert ein Prozess P_1 eine Datei, so wird dies erst bei Auslagerung der Seite(n) für einen weiteren Prozess P_2 sichtbar (das führt zu einem Problem z.B. bei Dateikommunikation)

4.1.2 Verzeichnisse

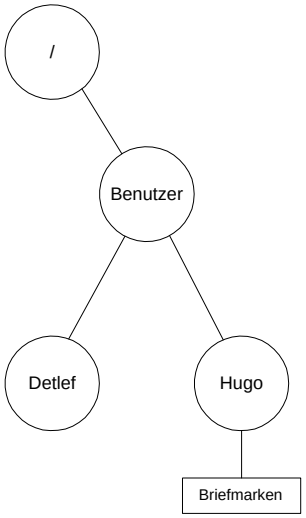


Abbildung 31: Der Pfad der Datei /Benutzer/Hugo/Briefmarken

Ein Verzeichnis ist meistens eine spezielle Datei, die für jede Datei einen Eintrag enthält (Name, Attribute, Plattenadresse). Da Verzeichnisse selbst Dateien sind, ist auch ein Verzeichnisbaum möglich.

Eine Dateireferenz erfolgt entweder

- durch absoluten Pfad (relativ zum Wurzelverzeichnis)
  z.B. unter UNIX /usr/bin/lpr
- durch Pfad relativ zum Bezugsverzeichnis (current directory)
  z.B. unter UNIX ../../pub/aufgabe1.txt

Operationen:

create leeres Verzeichnis erstellen

delete leeres Verzeichnis löschen

open dir vor dem Lesen/Schreiben z.B. bei der Suche nach einer Datei. Meist wird ein Puffer angelegt.

close dir nach dem Lesen/Schreiben

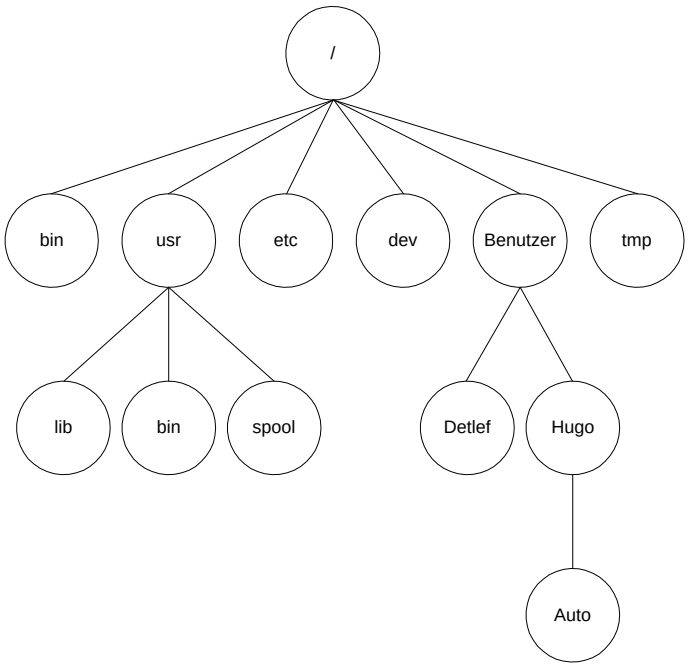


Abbildung 32: Der Dateibaum von UNIX

read dir Lies/Schreib nächsten Eintrag

link Erzeuge im Verzeichnis einen Verweis auf eine Datei in diesem oder einem anderen Verzeichnis

z.B. unter UNIX

```
> cd /homes/schulze
> ln -s /homes/schulze/tmp/bla.tex toll.tex
> ls -l
(...)
lrwxrwxrwx 1 (...) toll.tex -> /homes/schulze/tmp/bla.tex
(...)
```

unlink wenn auf eine Referenz angewandt, wird nur diese entfernt

wenn auf eine Datei selbst angewandt, wird die Datei gelöscht

4.2 Dateisystemimplementierung

Aufgabe: Abbildung des logischen Dateisystems auf z.B. Plattenoberflächen, Plattenzylinder, Plattensektoren

Die Abbildung erfolgt in zwei Schritten:

logische Datei	→ (a)	fortlaufend nummerierte Datenblöcke fester Länge	→ (b)	Geometrie des Speichermediums

(b) wird vom Controller (z.B. AT-Bus- oder SCSI-Controller) ausgeführt
Wir wollen den ersten Teil der Abbildung betrachten (a):
Aufgaben des Betriebssystems:

- Zuordnung Datenblöcke ↔ Dateien
- Verwaltung freier Datenblöcke
- Verwaltung defekter Datenblöcke
- Verwaltung von Verzeichnissen
- schnellen Zugriff ermöglichen
- geringer Datenverlust bei Fehler

4.2.1 Allokation und Verwaltung von Datenblöcken

Kontinuierlich:
Blöcke jeder Datei liegen nebeneinander

- einfache Verwaltung
- schnell

Aber:

- die Dateilänge ist beim Anlegen einer Datei meist unbekannt
- es kommt zu einer Fragmentierung

Besser: Verteilen der Datei auf beliebig positionierte Blöcke
Aber: Das erfordert

Verwaltung

interne Verkettung *Nachteil:* eine Datei ist nur sequentiell lesbar

externe Verkettung Die externe File Allocation Table (FAT) hat eine Zelle für jeden Block, die einen Verweis auf den nächsten Block enthält. Die FAT hat einen reservierten Platz.

Bei einer Suche muss nur die FAT sequentiell gelesen werden.
Beispiel: MS-Dos

- Verlust von Teilen der FAT führt zu dem Verlust referenzierter Daten
  ⇒ 2 redundante FAT's
- Blockgröße ist proportional zur Datenträgergröße um die FAT-Größe zu begrenzen

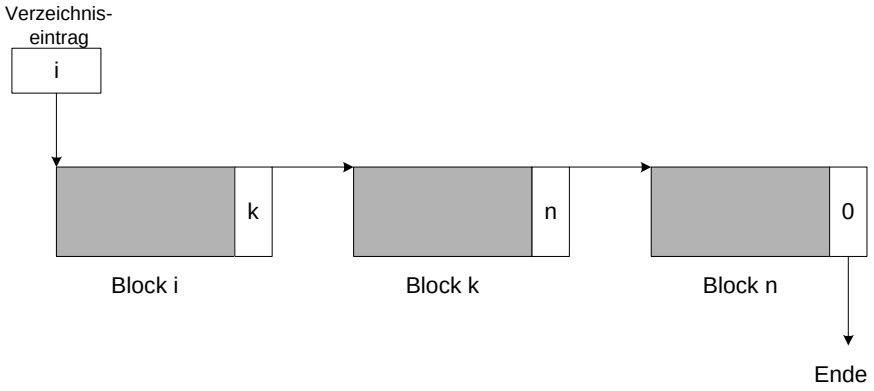


Abbildung 33: Interne Verkettung von Dateiblöcken

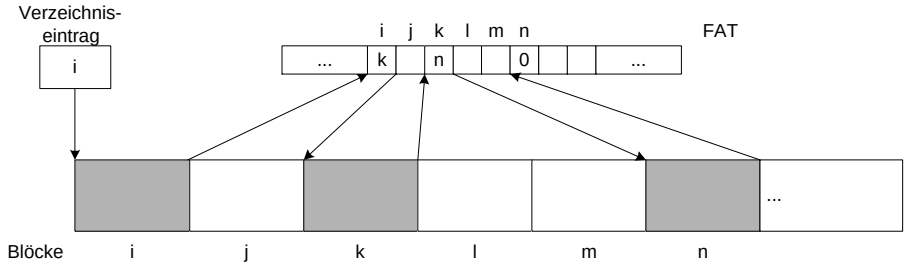


Abbildung 34: Externe Verkettung durch die FAT

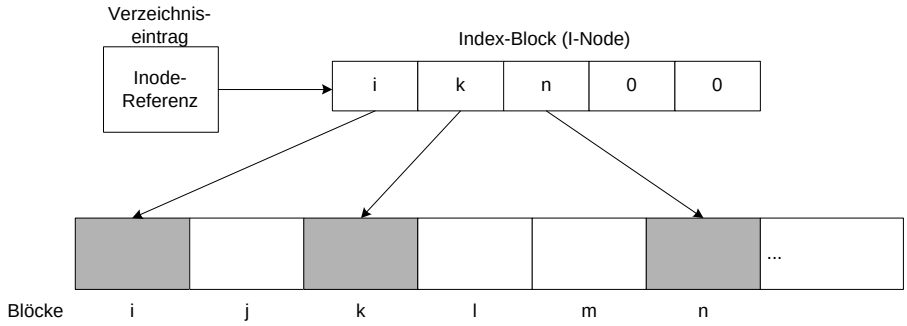


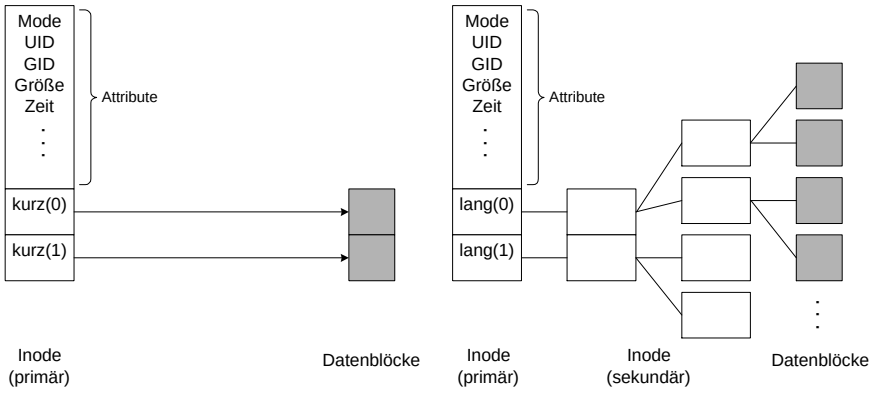
Abbildung 35: Indexblockverkettung (Inode)

Indexblock (I-Node)

- keine Verkettung
- Verzeichnis verweist nicht auf den ersten Datenblock, sondern auf den Inode (der selbst wieder ein Block ist), der die Verweise auf weitere Blöcke enthält
- falls ein Inode zu kurz für eine Datei ist, kommt es zu einer Hierarchisie-

rung, d.h. der Inode-Eintrag verweist auf einen anderen Inode analog zu einer mehrstufigen Seitentabelle

- *Beispiel* UNIX



(a) Inodes einer kurzen Datei (b) Inodes einer langen Datei

Abbildung 36: Inodes verschieden langer Dateien sind unterschiedlich aufgebaut

Master File Table (MFT) *Beispiel:* NTFS (Dateisystem von Windows NT)

Es gibt eine zentrale Metadatei für *alle* Dateien, die folgendes enthält:

- Attribute
- Inhalt:
  1. kurze Dateien: direkt in der MFT (resident)
  2. längere Dateien:
     In Form mehrerer „runs“. runs sind Cluster von Blöcken auf dem Datenträger. Spart Indexinfo im Vergleich zu UNIX.

4.2.2 Verwaltung freier und fehlerhafter Blöcke

Dateisystem mit FAT Markierung frei oder defekt geschieht direkt in der FAT

Dateisystem ohne FAT Hier ist eine zusätzliche Datenstruktur nötig:

(a) Vektor mit einem Bit pro Block. In diesen Bits steht jeweils, ob der Block frei/belegt oder defekt ist. Die Größe ist somit die Anzahl der Blocks.

(b) freie Blöcke werden in einer verketteten Liste und defekte Blöcke in einer Pseudo-Datei geführt. Die Größe ist bestimmt durch die Anzahl der freien Blöcke mal die Anzahl der Bits für die Blocknummer. Im Gegensatz zu der Größe bei (a) ist die Größe der Datenstruktur geringer, wenn die Platte sehr voll ist.

4.2.3 Zentrale Dateisystemdaten

- globale Strukturen (z.B. FAT, Freiliste oder Bitmap) liegen in einem ausgezeichneten Block („superblock“, „home block“, „bootsector“)
- dort liegt auch der Verweis auf die Wurzel des Verzeichnisbaumes

  UNIX Inode von `/`

  NTFS MFT

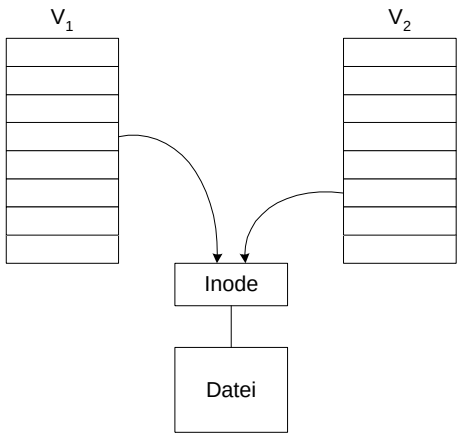
  MS-Dos physischer Bereich

  Bei UNIX und NTFS ist der Verweis nur ein Verweis auf eine gwöhnliche Datei, während bei MS-Dos auf den physischen Bereich verwiesen wird.

4.2.4 Links

Es gibt zwei Realisierungen der Verzeichnisoperation `LINK`:

(a) Ein Link ist ein Verweis im Verzeichnis auf den Inode einer Datei



Problem: In V_1 wird die Datei gelöscht. Inode enthält keinen „Rückwärtszeiger“ auf V_2, um dort ebenfalls den Eintrag zu löschen.

⇒ Der Inode kann nicht gelöscht werden, aber der Besitzer kann ggf. nicht auf den Besitzer von V_2 geändert werden.

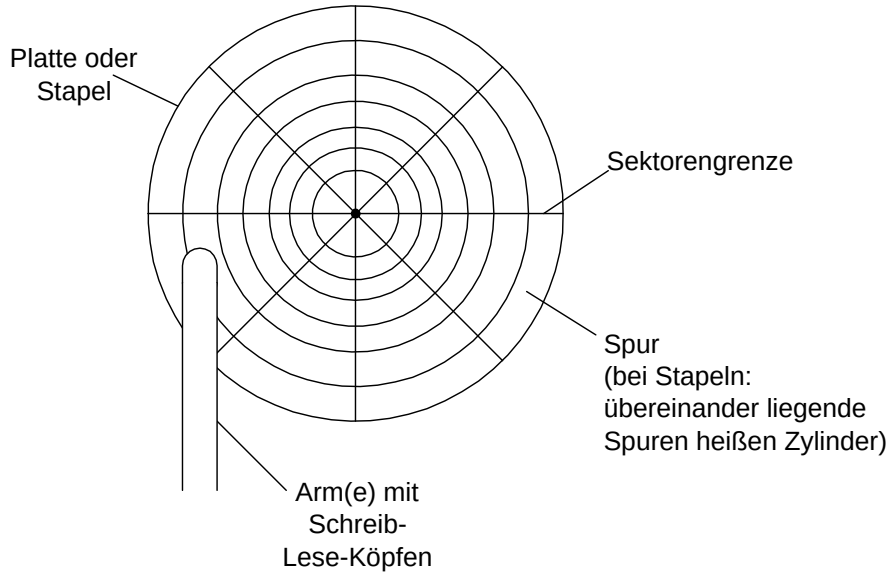
(b) Ein Link ist eine Datei mit Pfadnamen (symbolic link)

Der Problem wie bei (a) besteht nicht.

Aber: Overhead (zusätzlicher Inode und Block, Pfadverfolgung)

4.3 Festplattenmanagment

4.3.1 Hardware



Typische Daten einer Festplatte sind:

Umdrehungen pro Minute 10.000 Upm

Datendurchsatz 30 MB/sec.

Abstand Kopf-Platte < 500 nm

Spurbreite $< 1\mu$m

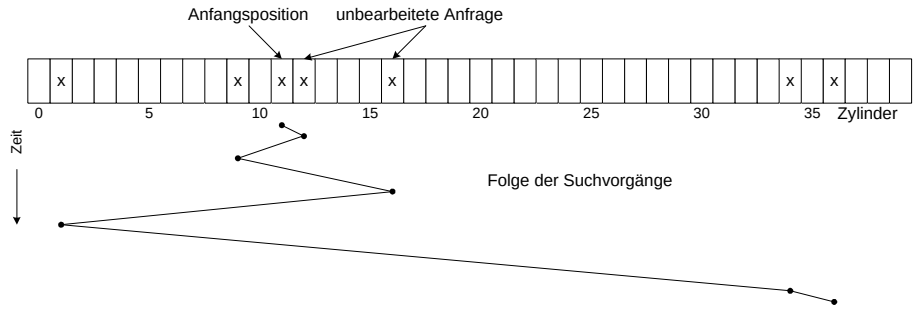
Die Performance hängt ab von:

1. Zylinder anfahren
2. Lesen/Schreiben
3. Übertragung

Der erste Punkt dominiert somit die Performance. Somit werden Strategien zur Bearbeitung benötigt. Hierzu gibt es mehrere Alternativen:

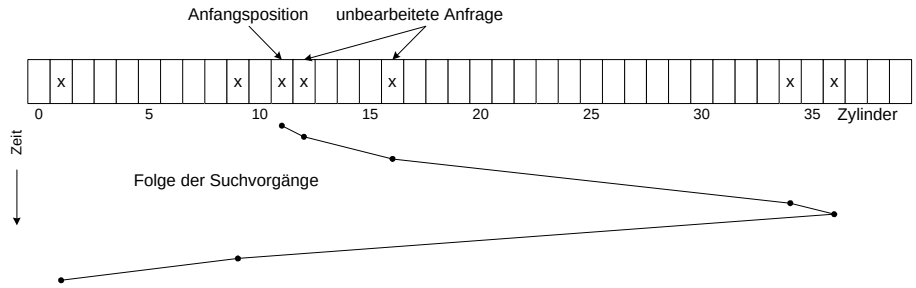
FCFS ungünstig

Shortest-Seek-First Bearbeite die Anfrage, für die die Distanz Arm–Zylinder minimal ist, zuerst.



Problem: Der Arm bevorzugt die Mitte der Platte. Unter Umständen wird auf die äußeren Zylinder selten oder nie zugegriffen!

Elevator Fahre den Arm solange in eine Richtung, bis keine Anfrage mehr in dieser Richtung erreichbar ist.



4.3.2 Block-Managment

Warum Daten nur als Blöcke *fester* Größe auf der Platte speichern?

Alternative: variable Segmente (wie im Hauptspeicher)

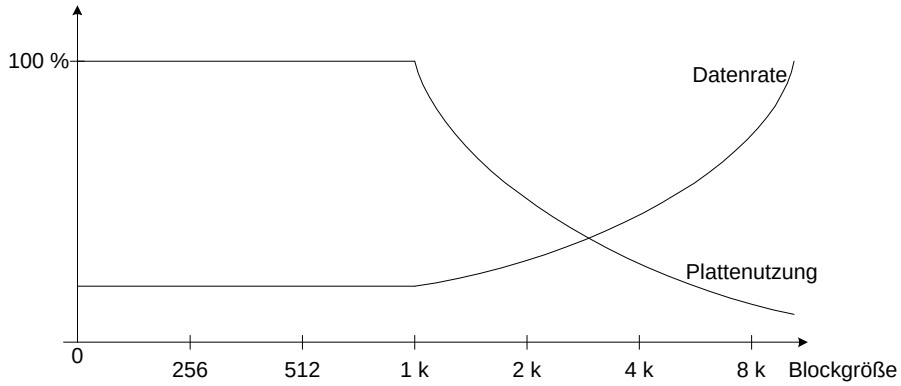
Geht nicht, da das Verschieben auf der Platte zu langsam ist.

Block-Größe

groß Verschwendung bei kleinen Dateien

klein viele Blöcke für große Dateien führt zu einem langsamen Zugriff

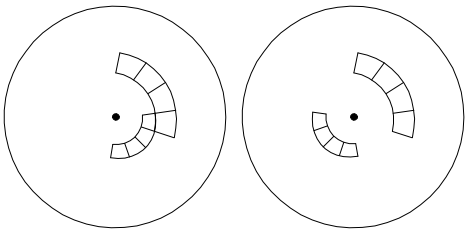
Beispiel: 1k große Dateien



Gebräuchlich sind 512 – 2k

Block-Positionierung

1. Blöcke, auf die nacheinander zugegriffen wird, in der selben Spur (oder in benachbarten Spuren) speichern
2. Vor jedem Dateizugriff muß mindestens ein Inode gelesen werden. Aber wenn alle Inodes an einer zentralen Position gespeichert werden, müssen auch viele Armbewegungen gemacht werden. Deshalb sollten Zylindergruppen mit lokalen Inodes gebildet werden.
3. Berücksichtigung der Rotationszeit



(a) schlecht (b) besser

Abbildung 37: Blockpositionierung ohne und mit Berücksichtigung der Rotationszeit

4.3.3 Plattencache

Häufig angefragte Blöcke in Puffer („cache“) speichern, um die Anzahl der Zugriffe zu reduzieren

Bei einer Anfrage wird überprüft, ob der Block bereits im Cache vorhanden ist. Wenn er es nicht ist, muss er geladen werden. Falls der Cache voll ist, wird ein Block zurückgeschrieben und anschließend überschrieben

Gleiche Vorgehensweise wie bei der Seitenersetzung

Aber: Der beste Algorithmus LRU läßt die Blöcke auch am längsten im Cache. Bei einem Absturz ist die Gefahr der Inkonsistenz dann am größten.

Daher werden Blöcke eingeteilt nach folgenden Kriterien:

1. Für Konsistenz wichtige Blöcke, z.B. Inodes, sofort oder automatisch in kurzen Abständen zurückschreiben

   UNIX: sync alle 30 sec.

2. Blöcke, die *wahrscheinlich* nicht zweimal hintereinander benutzt werden, kommen an eine schlechte Position in der LRU-Liste

4.3.4 Dateisystem-Konsistenz

Problem: Systemabsturz, ehe modifizierte Blöcke zurückgeschrieben wurden, die das Dateisystem verwalten (Inodes, Verzeichnisse, Bitmap freier Blöcke).

UNIX: Check beim Systemstart

Block-Konsistenz Jeder Block muß genau einmal in einer Datei oder alternativ in der Freiliste auftreten.

Fehler:

1. „verlorener“ Block (d.h. der Block ist nicht verzeichnet) → Eintrag in die Freiliste
2. mehrfach in der Freiliste verzeichnet → entfernen der mehrfachen Einträge
3. in Datei und in der Freiliste eingetragen → entfernen des Blocks aus der Freiliste
4. mehrfach in Datei(en) allokiert → entsprechenden Block in einen freien Block kopieren und diesen einer anderen Datei zuordnen

Verzeichnis-Konsistenz Bei **/** beginnend wird der Verzeichnisbaum rekursiv untersucht. Für jeden Inode wird die Zahl N der Verzeichnis-Referenzen gezählt und mit dem Linkzähler L der Inodes verglichen:

Fehler:

1. $N < L$: Auch beim Löschen aller referenzierten Dateien würde der Inode nicht gelöscht → setze $L := N$
2. $N > L$: Beim Löschen referenzierender Dateien fällt L auf 0, ehe alle Dateien gelöscht sind, so dass ein Inode fälscherlicherweise „frei“ wird → setze $L := N$

Merkwürdigkeiten:

Warning bei z.B. Zugriffsrechten

- - -	- - x	r w x
Besitzer	Gruppe	Welt

4.4 Sicherheit

Entwurfsprinzipien für Schutz vor unbefugtem Zugriff:

- Einfachheit und Übersichtlichkeit (optimal: beweisbarer Schutz)
- Geschlossenheit: keine Umgehungsmöglichkeiten außerhalb des Betriebssystemkonzeptes
- keine Abhängigkeit von der Geheimhaltung der Funktionsweise des Konzeptes
- nur minimal erforderliche Rechte beim Zugriff auf Betriebsmittel für jeden User bzw. jedes Programm (need-to-know-Prinzip)
- Benutzbarkeit nicht unzumutbar einschränken

Begriffe in Sicherheitskonzepten:

Subjekte Benutzer, Prozesse, ...

Objekte Betriebsmittel

Operationen Lesen, Schreiben, ...

Idee: Subjekte dürfen nur bestimmte Operationen auf Objekten ausführen

Menge der auf bestimmten Objekten erlaubten Operationen heißt „Schutzbereich“ („protection domain“)

Voraussetzungen:

- Identifizierung der Subjekte (z.B. Passwort)
- Operationen nur unter Kontrolle des Betriebssystems, nicht direkt

  Gegenbeispiel: MS-Dos erlaubt den Zugriff auf die I/O-Geräte
- keine „verdeckten Kanäle“

Verwaltung ist möglich auf Basis der Subjekte oder Objekte:

- Access Control Lists (ACLs)

  spezifizieren für ein *Objekt*, welches Subjekt welche Operationen ausführen darf ⇒ feingranulare Kontrolle

  Beispiel Windows NT
- Capabilities

  spezifizieren Berechtigung eines *Subjektes*, auf Objekte zuzugreifen (Liste)

Was ist mit Sicherheitskonzepten unter UNIX?

- Prozess ist user- oder system-process → Verstoß gegen minimale Berechtigungen

- grobe ACL-Realisierung:
  nur für Besitzer, Gruppen und Rest können Operationen auf Dateien spezifiziert werden

Bekannte Angriffsmöglichkeiten:

- Lesen freigegebener, nicht gelöschter Speicherbereiche
- Unterbrechung der Identifizierungsprozedur
- Nutzen (absichtlich offener) Schlupflöcher

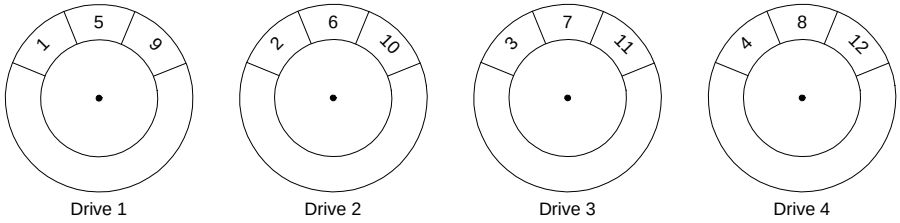
  ⋮

A RAID

(*R*edundant *A*rray of *I*ndependent *D*isks)
RAID verfolgt zwei Ziele:

- höhere Transferrate
- höhere Ausfallsicherheit

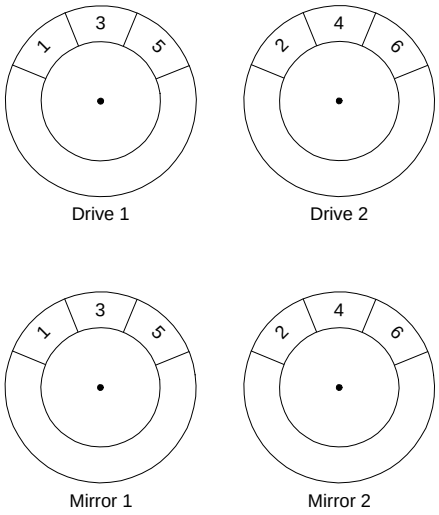
A.1 Striping (RAID 0)



RAID 0: Die Daten werden auf mehrere Platten verteilt geschrieben

- erhöht den Transfer
- aber erhöht auch die Ausfallwahrscheinlichkeit

A.2 Striping + Mirroring (RAID 0+1)

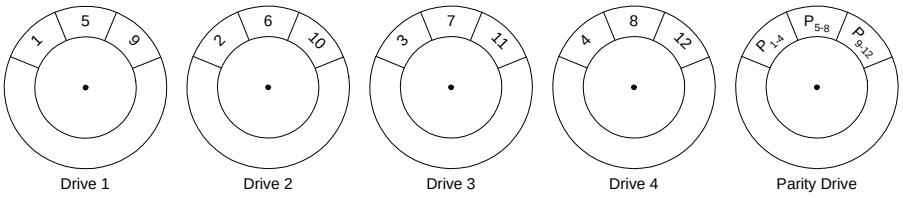


RAID 0+1: Zu jeder Platte existiert ein Mirror

- hoher Transfer

- hohe Sicherheit
- aber doppelte Plattenzahl benötigt

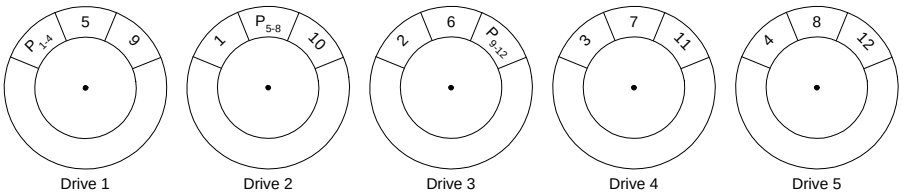
A.3 RAID 4 (Parity-Platte)



RAID 4: Die Parität aller 4 Platten wird auf einer seperaten Platte gespeichert

- hoher Transfer
- hohe Ausfallsicherheit
- nur ein zusätzliches Laufwerk
- aber die Parität muß bei jedem Schreiben berechnet und geschrieben werden

A.4 RAID 5 (verteilte Parität)



RAID 5: Die Parität wird auf alle Platten verteilt

- wie RAID 4, aber Lesezugriffe sind etwas schneller

Literatur

[1] Tanenbaum, Andrew S. (1995): *Moderne Betriebssysteme* (engl. *Modern operating Systems*), Hanser

[2] Tanenbaum, Andrew S.: *Betriebssysteme: Entwurf und Realisierung* (engl. *Operating Systems*), Hanser

[3] Silberschatz, Abraham (1998): *Operating system concepts*, Addison-Wesley

[4] Pomberger, Gustav und Rechenberg, Peter (1999): *Informatik Handbuch*, Hanser

[5] Kernighan, Brian W. und Ritchie, Dennis M. (1978): *The C programming language*, Prentice-Hall