Betriebssysteme

Skript zur Vorlesung
von Prof. Dr.-Ing. J. Brauer

erstellt unter Mitarbeit von Raymond Fürst

Vorwort

Dieses Skript stellt das Begleitmaterial für die Vorlesung *Systemarchitekturen* dar, die im Rahmen der des Vorlesungszyklus *Systemtechnik* stattfindet. Es wird im wesentlichen das Thema *Betriebssysteme* behandelt.

Danksagung: Großen Anteil am Zustandekommen dieses Skripts hat Herr Raymond Fürst, der zu den ersten Diplom-Wirtschaftsinformatikern der NORD-AKADEMIE gehört.

Inhaltsverzeichnis

Kapitel 1

Einführung

In LV-Zyklus Systemtechnik werden verschiedene Arten von Systemen behandelt. Deshalb soll zunächst eine Klärung des Begriffs *System* versucht werden.

1.1 Systembegriff

Problemlösung mit Hilfe von Rechnern:

- einfache Probleme → einfache, überschaubare Programme
- komplexe Probleme
  → Erarbeitung von Teillösungen, es entstehen Komponenten einer Gesamtlösung, die zusammenarbeiten müssen.
  ⇒ System
- in DV-Systemen typisch:
  parallel ablaufende Aktivitäten auf verschiedenen technischen Komponenten (Betriebsmittel)

1.2 Informationsverarbeitende Systeme

Informationsverarbeitende Systeme bestehen - hardware-technisch - gesehen aus einer Vielzahl von aktiven und passiven Komponenten:

- *aktiv:* Prozessoren, Peripheriegeräte

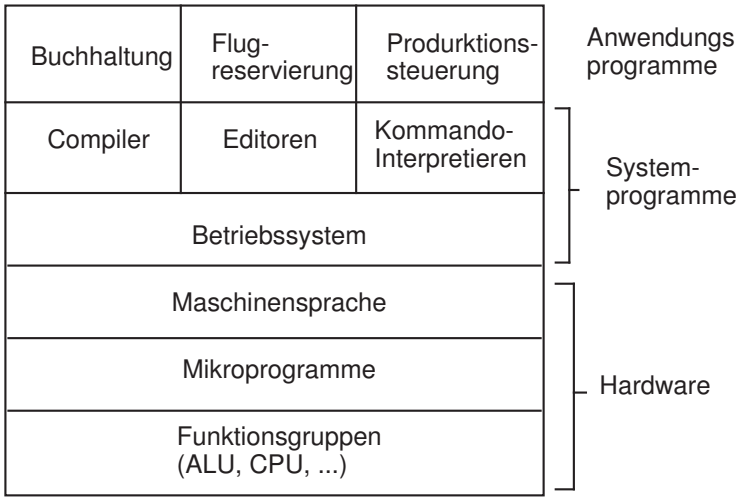


Abbildung 1.1: Beschreibungsebenen eines informationsverarbeitenden Systems

- *passiv:* Speicher

Abbildung 1.1 zeigt die Beschreibungsebenen eines informationsverarbeitenden Systems.

Das Betriebssystem stellt die fundamentalste Komponente der Systemsoftware dar. Aufgaben, Strukturen und Prinzipien von Betriebssystemen finden sich in ähnlicher Form auch in anderen informationsverarbeitenden Systemen wieder (z.B. Datenbanksysteme, Flugbuchung, Produktionsüberwachung und -steuerung).

Architekturen von Betriebssystemen werden im folgenden stellvertretend für andere Systemarchitekturen betrachtet.

Erste grundlegende Aufgabe eines Betriebssystems: Bereitstellen einer virtuellen (abstrakten) Maschine, d.h. Befreiung des Programmierers von Hardwaredetails.

Beispiel: Lesen eines Datenblocks von einer Festplatte
Vorteile:

- Drucken mit endlosem Papiervorrat

- fehlerfreie Datenübertragung

Nachteile:

- u.U. nur unzureichende Unterstützung der Leistungsfähigkeit spezieller Geräte (Beispiel: hochauflösender Grafikbildschirm wird zum Fernschreiber „degradiert“)
- „primitives“ Dateikonzept (vgl. z.B. PASCAL)

Definition nach DIN 44300 „Betriebssystem (*operating system*): Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des ditigalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.
Eine Sprache, der ein Betriebssystem gehorcht, heißt Betriebssprache (*operating language*).“

Zweite grundlegende Aufgabe eines Betriebssystems: Verwaltung der Betriebsmittel einer Rechenanlage
Beispiele: Druckerverwaltung, Speicherverwaltung

1.3 Betriebssystemkategorien und Historie

Rechner der ersten Generation (Röhren, 1945-1955)
Rechner der ersten Generation kannten keine Betriebssysteme. Die Programmierung erfolgte durch Stecktafeln, später mit Lochkarten.

Rechner der zweiten Generation (Transistoren)
Programm- und Dateneingabe erfolgte mit Lochkarten, Ausgabe auf Drucker. Es gab zwei Betriebsarten:

- *online*-Betrieb
- *offline*-Betrieb (Puffern von Ein- und Ausgabe über Magnetbänder) sog. *Batch*-Systeme

Abbildung 1.2 zeigt das Beispiel eines „frühen“ Batch-Systems.

Die Ausführung eines sogenannten *Batch-Jobs* umfaßte dabei die folgenden Schritte:

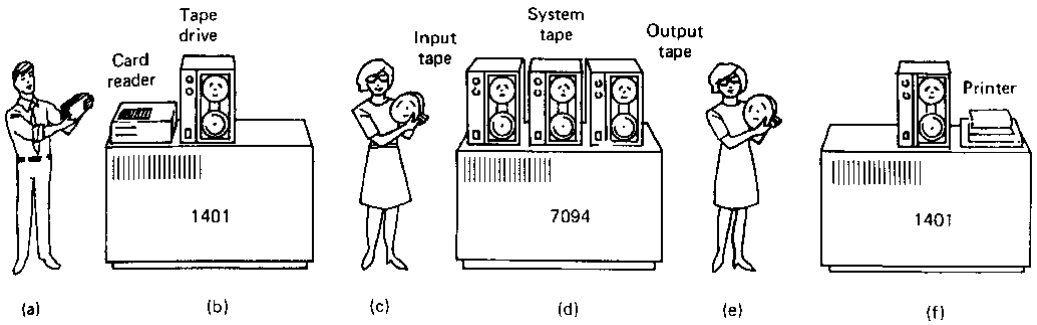


Abbildung 1.2: Batch-System, Bild entnommen aus [4]

(a) Programmierer liefert die Karten ab

(b) Vorrechner (IBM 1401) liest mehrere Benutzeraufträge (Jobs) ein und schreibt sie auf ein Magnetband

(c) Operator legt Band in Station des Hauptrechners (IBM 7094) ein.

(d) Hauptrechner führt Benutzeraufträge aus und schreibt deren Ausgabe auf ein weiteres Magnetband

(e, f) Ausgabeband wird zum Vorrechner transportiert und Ergebnisse werden dort ausgegeben.

Abbildung 1.3 zeigt ein Beispiel eines typischen Batch-Jobs.

Einbenutzersystem im Einprogrammbetrieb
Es gab drei Phasen der Job-Ausführung:

- Programmeingabe
- Programmausführung
- Ergebnisausgabe

Für die folgenden Betrachtungen sollen für die Berechnung von *Prozessorauslastung* und *Durchsatz* diese Formeln benutzt werden:

$$\text{Prozessorauslastung} = \frac{\text{Ausführungszeit}}{\text{Verweildauer des Jobs in der Anlage}}$$

$$\text{Durchsatz} = \frac{\text{Anzahl Jobs}}{\text{Zeiteinheit}}$$

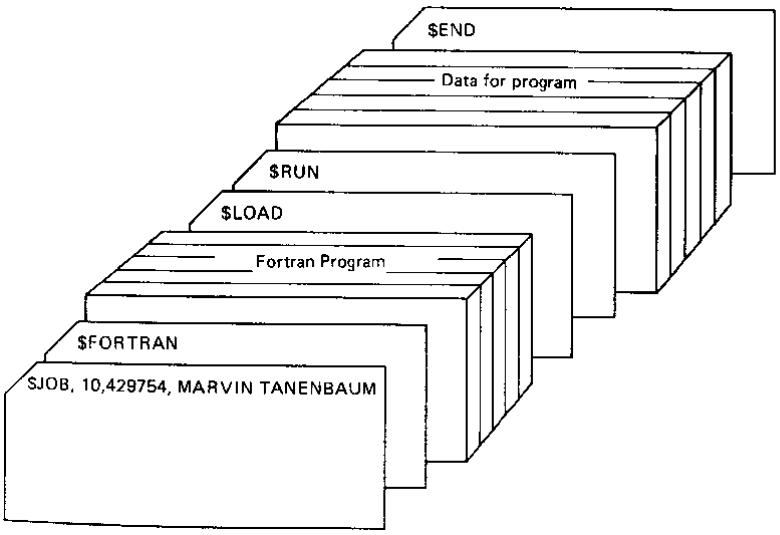


Abbildung 1.3: Batch-Job als Kartenstapel, Bild entnommen aus [4]

Beide Kenngrößen sind bei Terminal-Betrieb (wie z.B. durch einen Benutzer vor einer Workstation) beliebig schlecht, u.U. aber auch nicht von großer Bedeutung. Bei einem Betrieb mit Kartenleser und Drucker kann insbesondere die Eingabezeit drastisch vermindert werden.

	Eingabezeit (300 Karten):	0,3 min.
Zahlenbeispiel:	Ausgabezeit (500 Zeilen):	0,5 min.
	Berechnungszeit:	1 min.

Ausgehend von diesem Zahlenbeispiel zeigt die Tabelle in Abbildung 1.4 in der linken Spalte mögliche Zeitangaben für die Durchführung des Benutzerauftrages im Online-Betrieb, d.h. die Rechenanlage steht dem Anwender für die Durchführung seines Jobs exklusiv zur Verfügung.

Betriebssysteme vor 1955 bestanden nur aus einen Einlese- und einem Druckprogramm. Eine weitergehende Bedienerunterstützung gab es nicht. Heute kann man sich solche Betreibssysteme nur leisten, wenn Auslastung und Durchsatz keine Rolle spielen (Personal Computer). Diesen Fall wollen wir vorerst nicht weiter betrachten.

Um eine Verbesserung von Durchsatz und Auslastung der damals teuren Hardware zu erzielen, entzog man den Rechner dem direkten Zugriff durch den Benutzer (closed-shop-Betrieb, offline). Dadurch konnte die Verweildauer

online-Betrieb			*offline*-Betrieb		
Gesamtzeit	=	15 min.	Gesamtzeit	=	1.8 min.
Prozessorauslastung	=	$\frac{1}{15} \approx 7\%$	Prozessorauslastung	=	$\frac{1}{1.8} \approx 55\%$
Durchsatz	$\approx$	$4\frac{\text{Jobs}}{h}$	Durchsatz	$\approx$	$33\frac{\text{Jobs}}{h}$

Abbildung 1.4: Prozessorauslastung und Durchsatz bei Online- und Offline-Betrieb

eines Auftrags in der Anlage nahezu auf die Summe von Eingabezeit, Ausgabezeit und Berechnungszeit reduziert werden. Damit ergeben sich die in der rechten Spalte in Abbildung 1.4 angegebenen Werte.

Voraussetzung für diesen Betrieb war, das die Ablaufsteuerung durch ein Betriebssystem vorgenommen wurde, daß sich ständig im Hauptspeicher befinden mußte. Damit entstanden gleichzeitig neue Probleme:

- Das Betriebssystem mußte gegen „Angriffe“ durch die Benutzerprogramme geschützt werden.
- Die Benutzerprogramme müssen nach Ablauf die Kontrolle an das Betriebssystem zurückgeben.

Rechner der dritten Generation (1965-1980)
waren u.a. gekennzeichnet durch:

- die Verwendung von Integrierte Schaltkreise (ICs, Chips)
- die Benutzung sowohl für technisch-wissenschaftliche (rechenintensive) und kommerzielle (E/A-intensive) Aufgaben

Typische Verteterin war die IBM 360 und ihre Nachfolger (370, 4300, 3080, 3090). Das wichtigste zu lösende Problem war die Verbesserung der Prozessorauslastung. Abbildung 1.5 zeigt exemplarisch die Zeitaufteilung zwischen dem Zentralprozessor und angeschlossenen Peripheriegeräten, die für die Abwicklung eines Programms benötigt werden. Die deutlich erkennbaren „Beschäftigunglücken“ der CPU gaben Anlaß zu der grundlegenden Idee, diese Beschäftigunglücken für die ineinander verzahnte Bearbeitung mehrerer Aufträge zu nutzen, wie es in Abbilung 1.6 dargestellt ist.

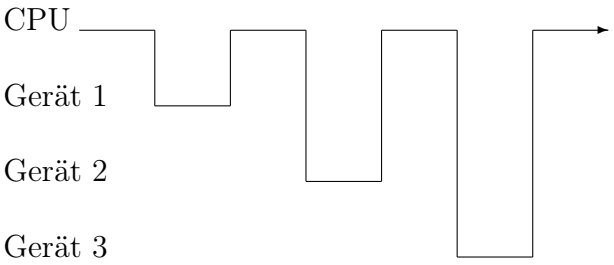


Abbildung 1.5: Aufteilung der Bearbeitungszeit zwischen Prozessor und Peripheriegeräten

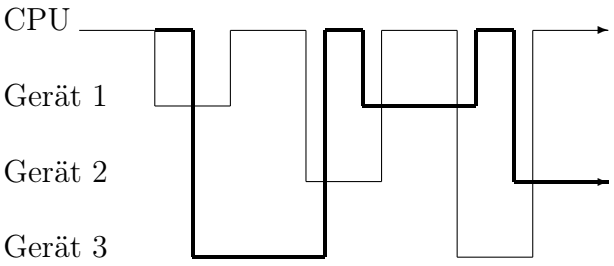


Abbildung 1.6: Mehrfachnutzung der CPU durch zwei Programme

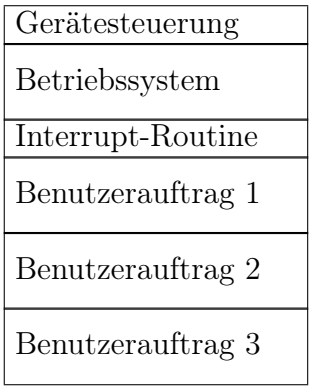


Abbildung 1.7: Aufteilung des Hauptspeichers bei Mehrprogrammbetrieb

Mehrprogrammbetrieb
Beim *Mehrprogrammbetrieb* (*Multiprogramming*) werden die Zeiten, während der die CPU auf die Fertigstellung einer E/A-Anforderung warten müßte, für die Bearbeitung eines anderen Benutzerauftrages genutzt. Dies ist nur unter der Vorraussetzung möglich, daß sich mehrere Programme (Benutzeraufträge) gleichzeitig im Hauptspeicher befinden. Abbildung 1.7 zeigt eine mögliche Aufteilung des Hauptspeichers.

Beschränkung der Parallelarbeit ist im wesentlichen gegeben durch:

- verfügbare Rechenzeit
- Platz im Hauptspeicher

Das Betriebssystem muß entscheiden können, zu welchen Zeitpunkten welcher Benutzerauftrag bearbeitet wird. Unterbrechungen (*Interrupt*) des Zentralprozessors müssen ermöglicht werden. Bei jeder Unterbrechung muß ein Prozessorverwaltungsprogramm aufgerufen werden. „Langläufer“ müssen unterbrochen werden können (Zeitgeber).

Ziel: gerechte Verteilung der verfügbaren Prozessorzeit auf alle Benutzeraufträge
Unter *Prozessen* sollen vorerst „Träger parallel ablaufender Aktivitäten“ verstanden werden, die die CPU benötigen und selbst jeweils sequentiell ablaufen.

Aufgaben (und Probleme) des Betriebssystems:

1. *Scheduling* (Prozessorverwaltung)
2. gerechte Bedienung von Geräteanforderungen
3. *Overhead*: Betriebssystem nimmt den Benutzeraufträgen Rechenzeit weg, diese benötigen daher mehr Zeit (Verweildauer im System)
4. Schutz der Prozesse vor gegenseitigen Fehlfunktionen (seperater Adreßraum)
5. Zugriffsrechte auf gemeinsam benutzte Daten verwalten
6. Buchführung des Ressourcenverbrauchs
7. „*Spooling*“ (über Platten) der Job-Ein- und -Ausgabe

Mehrprogrammbetriebssysteme mit Verdrängung beziehen auch Benutzeraufträge, die sich z. Zt. nicht im Hauptspeicher befinden, in die Vergabeplanung von Betriebsmitteln ein. Dies führt zur Ein- und Auslagerung von Prozessen.

Interaktive Mehrbenutzersysteme im Mehrprogrammbetrieb: *Time-Sharing*-Betriebssysteme sind eine Weiterentwicklung des Mehrprogrammbetiebs, bei dem viele Benutzer über Terminals online mit der Rechenanlage verbunden sind. Als zusätzliche Aufgabe für das Betriebssystem entsteht die Forderung, jedem Benutzer durch geeignete Prozessorzuteilung den Eindruck zu vermitteln, die Rechenanlage würde nur für ihn arbeiten.

Rechner der vierten Generation (1980 - heute)

Schlagworte:

- Hardware
  - Höchstintegration (VLSI)
  - Personal Computer
  - Workstations

- Typische Betriebssysteme
  - MS-DOS (PC)
  - UNIX
  - MVS (IBM), VMS (digital)
- Neuere Entwicklungen
  - Netzwerk-Betriebssysteme
  - verteilte Betriebssysteme

1.4 Zusammenfassung

- Betriebssysteme stellen virtuelle Maschinen mit virtuellen Befehlen (genannt Systemaufrufe, *system calls*) bereit. Sie erlauben Benutzerprogrammen, Dienstleistungen des Betriebssystems in Anspruch zu nehmen.
- Prozesse (Träger parallel ablaufender Aktivitäten) enthalten ein Benutzer- oder Systemprogramm sowie alle für die Ausführung notwendigen Informationen (u.a. Befehlszählerstand, Registerinhalte, Adreßraum (Speicherabbild, *core-image*)).
  - Prozeßverwaltung: Erzeugen, Beenden, Suspendieren von Prozessen
  - Betriebsmittelverwaltung: Hauptspeicherverwaltung, Prozessorverwaltung
- Dateisystem: Verwaltung von Daten mit peripheren Speichermedien
- Benutzerverwaltung

Kapitel 2

Prozesse

Für die Behandlung der Anforderungen in Mehrprogrammbetriebssystemen sind primitive Ad-hoc-Lösungen nicht mehr möglich. Das Verständnis des Gesamtsystems ist nicht mehr durch die Beschreibung des Verhaltens der CPU zu jedem Zeitpunkt möglich, da das Verhalten der CPU in Mehrbenutzersystemen im Mehrprogrammbetrieb stark von nicht vorhersagbaren externen Ereignissen (Unterbrechungen) abhängig ist. Das Betriebssystem wird als Ansammlung von funktionellen Einheiten betrachtet, die zunächst unabhängig voneinander arbeiten aber über wohldefinierte Schnittstellen miteinander kommunizieren müssen. Diese funktionellen Einheiten bezeichnet man als Prozesse.

2.1 Prozeßbegriff

Typische Merkmale von Prozessen:

- brauchen Prozessor
- enthalten jeweils ein sequentielles Programm
- können grundsätzlich parallel ablaufen

Zur Abgrenzung zum Begriff *Benutzerauftrag* (*Job*): Zur Abarbeitung eines Benutzerauftrags sind in der Regel mehrere Prozesse notwendig.

Formen der Parallelität

- mehrere Prozesse laufen auf unterschiedlichen Prozessoren ab (tatsächlich parallel)
- ein Prozessor wird „scheibchenweise“ den Prozessen zugeordnet, so daß diese „überlappt“ ablaufen (quasi-parallel)

Die im Zusammenhang mit der Parallelität von Prozessen auftretenden Probleme sind davon aber unabhängig.

2.2 Synchronisation konkurrierender und kooperierender Prozesse

2.2.1 Problem des wechselseitigen Ausschlusses

Das Problem des wechselseitigen Ausschlusses (*mutual exclusion*) wurde erstmals 1965 von Edsger W. Dijkstra formuliert.

Beispiel 1: Zwei zyklische Prozesse p_1 und p_2 benutzen von Zeit zu Zeit ein Magnetband. Es steht nur ein Gerät zur Verfügung, das nicht von mehr als einem Prozeß gleichzeitig benutzt werden kann.

1. Lösungsversuch: man definiert eine boolesche Variable `frei`

p_1:

```
001  wiederhole
002    wiederhole bis frei;
003    frei := false;
004    benutze(magnetband)
005    frei := true;
⋮      ⋮
FFF  ständig
```

p_2:

```
001  wiederhole
002    wiederhole bis frei;
003    frei := false;
004    benutze(magnetband)
005    frei := true;
⋮      ⋮
FFF  ständig
```

Problem: Wenn p_1 und p_2 parallel ablaufen, können sie auch gleichzeitig das Magnetband als frei erkennen. Das gleiche Problem kann auch bei quasi-parallel ablaufenden Prozessen auftreten, da jeder Prozeß zwischen `002` und `003` unterbrochen werden kann.

Weiterer Nachteil dieser „Lösung“: Durch die Warteschleife wird Prozessorzeit beansprucht (*busy waiting*).

2. Lösungsversuch man definiert eine boolesche Variable `p1anderReihe`

p_1:

```
001  wiederhole
002    wiederhole bis p1anderReihe;
003    benutze(magnetband)
004    p1anderReihe := false;
⋮      ⋮
FFF  ständig
```

p_2:

```
001  wiederhole
002    wiederhole bis nicht p1anderReihe;
003    benutze(magnetband)
004    p1anderReihe := true;
⋮      ⋮
FFF  ständig
```

Ein wechselseitiger Ausschluß ist zwar gewährleistet, allerdings müssen die Prozesse das Magnetband abwechselnd benutzen. Beide Prozesse müssen außerdem „am Leben“ bleiben. *Busy waiting* tritt auch hier auf.

3. Lösungsversuch Definition zweier boolesche Variablen `p1istdran` und `p2istdran`

Initialisierung:

```
p1istdran := false
p2istdran := false
```

p1:

```
wiederhole
  p1istdran := true
  wiederhole bis nicht p2istdran
  benutze(magnetband)
  p1istdran := false
  ⋮
ständig
```

p2:

```
wiederhole
  p2istdran := true
  wiederhole bis nicht p1istdran
  benutze(magnetband)
  p2istdran := false
  ⋮
ständig
```

Wechselseitiger Ausschluß ist zwar garantiert, es besteht aber die Gefahr der Verklemmung (*deadlock*).

Anforderungen an eine Lösung für das Problem des wechselseitigen Ausschlusses:

1. Das Betriebsmittel wird nach endlicher Zeit zugewiesen.
2. Ein Prozeß gibt das Betriebsmittel nach endlicher Zeit wieder frei.
3. Ein Prozeß, der wartet, soll keine Rechenzeit verbrauchen.
4. Eine Problemlösung soll von den Prozessen in eine gemeinsame Umgebung verlagert werden.

Das grundsätzliche Problem resultiert aus der „unkontrollierten“ Benutzung gemeinsamer Daten.

Weitere Beispiele für das Auftreten des Problems des wechselseitigen Ausschlusses:

1. Veränderung von Datensätzen in einer von mehreren Prozessen gemeinsam benutzten Datei
2. gemeinsame Benutzung von Unterprogrammen mit lokalen Variablen für Zwischenergebnisse

Definition: Programmabschnitte, in denen sich zu einem Zeitpunkt nur jeweils ein Prozeß befinden darf, heißen *kritische Abschnitte* (*critical sections*).

Lösung: `P`- und `V`-Operationen nach Edsger W. Dijkstra

`P` und `V` sind zwei Operationen auf einer gemeinsamen Variablen, genannt *Semaphorvariable*. Jedem kritischen Abschnitt wird eine Semaphore zugeordnet.

Definition von `P` und `V`:

```
P(s):
   wenn s=1
   dann s:=0
   sonst blockiere den aufrufenden Prozess
         und schalte auf anderen Prozess um

V(s):
   wenn ein Prozess auf s wartet
   dann loese den Prozeß aus Wartezustand
   sonst s:=1
```

Beispiel für die Sicherung eines kritischen Abschnitts (Benutzung eines Magnetbandgeräts) durch eine Semaphore `s`:

```
P(s)
benutze(magnetband)
V(s)
```

`P`- und `V`-Operationen sind selbst kritische Abschnitte und müssen atomar sein (dürfen nicht selbst unterbrochen werden). Es handelt sich aber um kurze kritische Abschnitte, die im Systemkern realisiert werden, wo wechselseitiger Ausschluß einfach zu implementieren ist. Sie werden häufig mithilfe eines Spezialbefehls des Prozessors realisiert, wobei ein aktives Warten in Kauf genommen wird. Dazu wird eine Sperrvariable `pv` mit folgender Bedeutung eingeführt:

`pv=1`	:	`P`- und `V`-Operationen können ausgeführt werden
`pv=0`	:	`P`- und `V`-Operationen können nicht ausgeführt werden

Der Spezialbefehl `teste_und_setze(pv)` ist eine unteilbare Operation, die folgendermaßen arbeitet:

```
wiederhole solange pv = 0; (* tue nichts, busy waiting *)\\
pv := 0
```

Mithilfe dieses Befehls werden nun zwei modifizierte Operationen `P'` und `V'` eingeführt, die dann zur Sicherung eines kritischen Abschnitts eingesetzt werden können.

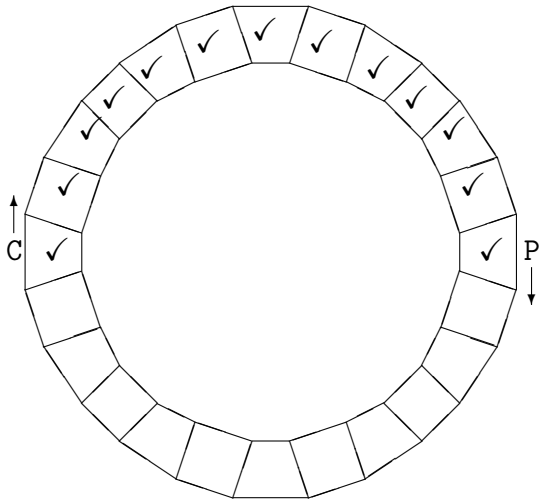


Abbildung 2.1: Ringpuffer für den Nachrichtenaustausch

```
P'(s)                          V'(s)
  teste_und_setze(pv)            teste_und_setze(pv)
  P(s)                           V(s)
  pv:=1                          pv:=1
```

Bisher wurden nur um gemeinsame Betriebsmittel konkurrierende Prozesse betrachtet, die sonst nichts miteinander zu tun hatten.

2.2.2 Synchronisation kooperierender Prozesse

Kooperation zwischen Prozessen kann z.B. heißen, daß Nachrichten zwischen einem Erzeuger und einem Verbraucher ausgetauscht werden (*producer-consumer-problem*).

Nachrichtenaustausch soll gepuffert erfolgen, um Erzeuger und Verbraucher bezüglich ihrer Arbeitsgeschwindigkeit zu entkoppeln. Ringpuffer fester Größe kann nur eine feste Anzahl von Nachrichten speichern. Abbildung 2.1 zeigt einen teilweise gefüllten Ringpuffer mit zwei Zeigern, `c` für den Verbraucher und `p` für den Erzeuger. Beide Prozesse bearbeiten den Puffer im Uhrzeigersinn. Durch die Prozeßsynchonisation muß verhindert werden, daß sie sich gegenseitig „überholen“.

Die Synchronisation von *producer* und *consumer* erfolgt über verallgemeinerte (nicht binäre) Semaphoroperationen.

Semaphoroperationen: `down`, `up`

```
down(s)                                  up(s)
  wenn s>0                                 s:=s+1
  dann s:=s-1                              löse auf s wartenden Prozeß
  sonst blockiere laufenden Prozeß         aus Wartezustand
```

Die Prozesse benutzen jeweils eine Kommunikationsprozedur, `SendeNachricht` und `EmpfangeNachricht`, die dafür sorgen, daß der Erzeuger wartet, wenn der Puffer voll ist, und der Verbraucher, wenn der Puffer leer ist.

```
EmpfangeNachricht(puffer,nachricht)
   down(voll)
   P(sp)
   nachricht:=puffer.nachricht[puffer.c]
   puffer.c:=puffer.c+1 mod max
   V(sp)
   up(leer)

SendeNachricht(puffer,nachricht)
   down(leer)
   P(sp)
   puffer.nachricht[puffer.p]:=nachricht
   puffer.p:=puffer.p+1 mod max
   V(sp)
   up(voll)

Initialisierung:
   leer:=max
   voll:=0
   sp:=1
   p:=0
   c:=0
```

Der Nachteil dieser Lösung des Synchronisationsproblems liegt darin, daß die Verantwortung für die korrekte Synchronisation bei den Prozessen bzw. deren korrekte Programmierung liegt. Programmierfehler können dabei zu schwer reproduzierbarem Fehlverhalten (z.B. Verklemmungen) führen.

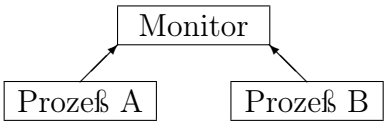


Abbildung 2.2: Hierarchische Prozeßstruktur mit Monitor

Abhilfe schafft z.B. daß von Hoare bzw. Brinch Hansen ([1]) beschriebene

Monitor-Konzept

das eine, wie in Abbildung 2.2 gezeigt, hierarchische Struktur vorsieht. Dabei sind alle kritischen Abschnitte als Routinen des Monitors implementiert.

2.2.3 Synchronisation durch Nachrichtenaustausch

Die bisher betrachteten Synchronisationsprimitive sind nur einsetzbar, wenn die beteiligten Prozesse Zugriff auf einen gemeinsamen Speicherbereich (shared memory) haben, in dem sich z.B. die Semaphorvariablen befinden. Auf diese Art ist daher die Synchronisation in Verteilten Systemen, wo Prozesse auf unterschiedlichen Maschinen ablaufen können, nicht möglich.

Hierfür werden neue Synchronisationsprimitive (Aufrufe des Systemkerns), die auf dem Austausch von Nachrichten (*message passing*) basieren, eingeführt:

- `send(destination,message)`
- `receive(source,message)`

Mit `send` und `receive` können Prozesse synchronisiert werden, die auf Prozessoren ohne gemeinsamen Speicher ablaufen. Bei einem Aufruf von `send` wird der Prozeß blockiert, wenn keine Nachricht übermittelt werden kann. Bei einem Aufruf von `receive` wird der Prozeß blockiert, wenn keine Nachricht verfügbar ist.

Mögliche Schwierigkeiten bei der Nachrichtenübermittlung:

- Verlust einer Nachricht
  Abhilfe: jede gesendete Nachricht muß quittiert werden (*acknowledgement*), wiederholen der Nachricht beim Ausbleiben der Quittung

- Verlust der Quittung
- doppeltes Eintreffen einer Nachricht beim Empfänger
  Abhilfe: Numerieren der Nachrichten
- Eindeutige Benennung (Adressierung) von:
  - Prozessoren
  - Maschinen
  - *domains*
- Sicherheitsprobleme
- Effizienz, wenn Sender und Empfänger auf der gleichen Maschine laufen

Behandlung des *Producer-Consumer-Problems* mit *message passing*:
Annahmen:

- Nachrichten haben feste Länge.
- gesendete, aber noch nicht empfangene Nachrichten werden vom Betriebssystem automatisch gepuffert.
- maximal `max` Nachrichten können gepuffert werden.

```
Consumer:
for i := 1 to max do send(producer,emptymessage);
while true do begin
   receive(producer,message);
   extract_data(message);
   send(producer,emptymessage);
   process(data);
end

Producer:
while true do begin
   produce_data(data);
   receive(consumer,emptymessage);
   build_message(message,data);
   send(consumer,message);
end
```

Anmerkungen:

- die Zahl der Nachrichten bleibt konstant
- für Pufferung ist ein fester Speicherbereich vorgesehen
- Pufferung und Adressierung erfolgt durch sog. *mailboxes* bei Sender und Empfänger
- *Rendez-vous-Technik*: Kommunikation ohne Pufferung
- in UNIX entsprechen sogenannte *pipes* den *mailboxes*

2.3 Verklemmungen

Definition: Eine Verklemmung (*deadlock*) bedeutet, daß zwei oder mehr Prozesse auf Ereignisse warten, die niemals eintreten werden („Nach-Ihnen-Nach-Ihnen"-Schleifen, Warten im „Kreis").

Beispiel: Verschachtelung von kritischen Abschnitten

```
P1 : P(a) ... P(b) ... V(b) ... V(a)
P2 : P(b) ... P(a) ... V(a) ... V(b)
```

Das Auftreten von Verklemmungen ist zeitabhängig. Ursachen sind im laufenden System schwer feststellbar und nicht ohne weiteres reproduzierbar. Das Problem wurde zuerst von Edsger W. Dijkstra1965 erkannt und analysiert.

Vier notwendige Bedingungen für das Auftreten von Verklemmungen:

1. „Wechselseitiger Ausschluß"-Bedingung:
   Ein Betriebsmittel, um das Prozesse konkurrieren, ist entweder frei oder genau einem Prozeß zugewiesen.
2. „Halte-und-warte"-Bedingung:
   Prozesse mit bereits zugewiesenen Betriebsmitteln dürfen weitere Betriebsmittel anfordern.

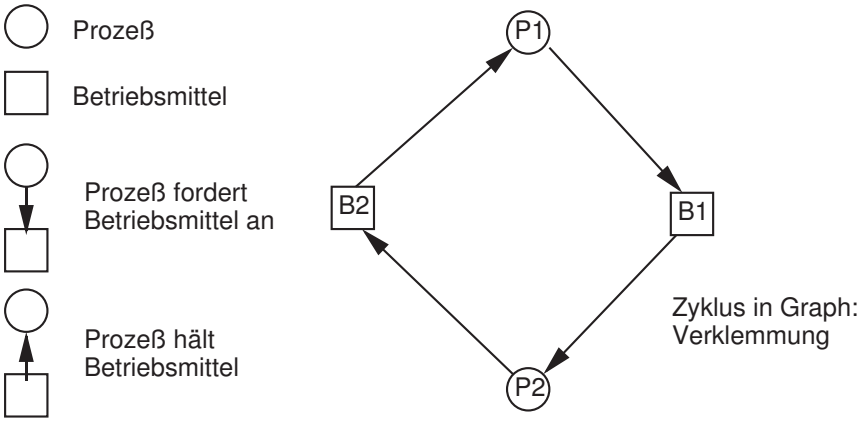


Abbildung 2.3: Modellierung einer Verklemmungssituation durch einen Graphen

3. „*No-preemption*"-Bedingung:
   Prozesse geben Betriebsmittel nur von sich aus frei. Betriebsmittel können ihnen nicht zwangsweise entzogen werden.

4. „*Circular-wait*"-Bedingung:
   Zwei oder mehr Prozessse warten wechselseitig auf Betriebsmittel, die von dem/den jeweils anderen gehalten werden.

Die vierte Bedingung wird, wie in Abbildung 2.3 gezeigt, zur Modellierung von Verklemmungssituationen durch Graphen zum Zwecke der Verklemmungserkennung benutzt.

Vier Strategien mit dem Verklemmungsproblem umzugehen:

1. „Vogel-Strauß"-Algorithmus (Beispiel: UNIX)

2. Erkennung und Auflösung (Beispiel: Datenbanktransaktionen)

3. Verklemmungsfreiheit durch besondere Zuteilungsalgorithmen (Verklemmungsvermeidung, *deadlock avoidance*)

4. Verhinderung von Verklemmungen durch Aufheben einer der vier notwendigen Bedingungen

Beispiele für Verhinderungsmaßnahmen:

1. mehrfachen Zugriff erlauben (Bedingung 1. ist aufgehoben)
   Im allgemeinen nicht sinnvoll.

2. *preemptive scheduling* (Bedingung 3. ist aufgehoben)
   Bei CPU selbstverständlich, bei E/A-Geräten meist nicht sinnvoll.

3. Vorwegzuteilung aller von einem Prozeß benötigten Betriebsmittel

   - im Prinzip möglich
   - Nachteil: Parallelität stark eingeschränkt
   - Betriebsmittelbedarf muß im Vorhinein bekannt sein.

Verklemmungsvermeidung beruht auf der Grundidee, die Betriebsmittelanforderungen der Prozesse in eine „verklemmungsfreie“ Reihenfolge zu bringen. Die Algorithmen (Vgl. Bankiersalgorithmus) hierfür sind teiweise sehr komplex und auch nur anwendbar, wenn der gesamte Betriebsmittelbedarf der Prozesse im vorhinein bekannt ist, was in der Praxis häufig nicht der Fall ist. Nichtsdestotrotz hat sich um dieses Thema herum eine eigene mathematische Theorie entwickelt, auf die hier aber nicht eingegangen wird.

2.4 Prozessorverwaltung

2.4.1 Funktion des Schedulers

Kurzfristige Steuerung (*short term scheduling*): Zuteilung von Betriebsmitteln (hier CPU) zu Prozessen, sobald verfügbar, bei möglichst guter Auslastung der Betriebsmittel und fairer Behandlung aller Prozesse. Sie stellt die Synchronisationsprimitive zur Verfügung.

(Daneben gibt es die - hier nicht behandelte - mittelfristige Steuerung, die auf der Job-Ebene angesiedelt ist und einzelnen Prozessen bzw. Jobs z.B. auf der Basis von Benutzerklassen oder des zu erwartenden Speicher- und Rechenzeitbedarfs Prioritäten für die Betriebsmittelvergabe zuweist.)

Für die folgenden Betrachtungen gehen wir von einer Systemkonfiguration aus, bei der die Zahl der Prozesse im Hauptspeicher größer ist als die Zahl der Prozessoren.

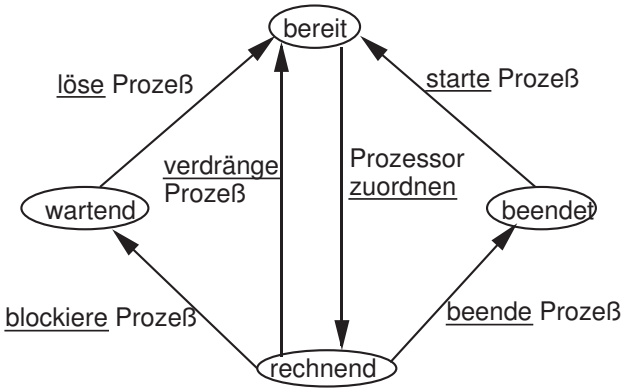


Abbildung 2.4: Mögliche Zustandsübergänge für einen Prozeß

Prozeßbeschreibung

Prozeßzustände:

- **beendet**
  Prozeß, der beendet bzw. nicht vorhanden ist.
- **rechnend**
  Prozeß, dem ein Prozessor zugeordnet ist.
- **wartend**
  Prozeß, der auf den Eintritt eines Ereignisses wartet.
- **bereit**
  Prozeß, der gestartet oder fortgesetzt werden kann, dem aber noch kein Prozessor zugeteilt wurde.

Abbildung 2.4 zeigt die möglichen Zustandsübergänge. Im folgenden werden die diese Übergänge auslösenden Operationen bzw. Situtationen beschrieben. Der Übergang vom Zustand `rechnend` in den Zustand `bereit` erfolgt hier durch die sogenannte *Verdrängung* (*preemptive scheduling*) - auch *unterbrechende Steuerung* genannt. Der einfachere Fall, daß ein Prozeß den Prozessor von sich aus abgibt (Prozeß rechnet bis er fertig ist), wird hier nicht weiter betrachtet.

Prozeßkontrollblöcke:
Das Betriebssystem (genauer: der Scheduler) verwaltet die Prozesse in einer aus *Prozeßkontrollblöcken* (PKB) bestehenden Prozeßtabelle:.

```
PKB=registerfeld[1..n]
   {für die Aufnahme der Registerinhalte incl PC}
prozeßtabelle[1..p] von PKB
   {Index identifiziert Prozeß}
```

Ein Prozeß ist durch seinen PKB und seinen Zustand beschrieben. Für die unterbrechende Steuerung ist ein Zeitgeber erforderlich, der den Prozessor periodisch unterbricht und den Scheduler aufruft. Die Intervallänge bestimmt den Rhythmus der Prozessorzuteilung. Sehr kurze Intervalle bringen viel „Overhead“, sehr lange Intervalle behindern eventuell Prozesse höherer Priorität.

Zeitgeber:

```
wiederhole
  Uhr:=intervalllänge
  wiederhole
    uhr:=uhr-1
  bis uhr=0
  setze interrupt
ständig
```

Befehlszyklus:

```
wiederhole
  wenn interrupt und intErlaubt
  dann rücksetze interrupt
       interruptbehandlung
       monitor(verdrängeProzeß)
  führe aktuellen Befehl aus
ständig
```

Die `interruptbehandlung` prüft die Ursache der Unterbrechung und führt die Routine `monitor(verdrängeProzeß)` nur dann aus, wenn die Unterbrechung durch den Zeitgeber ausgelöst wurde. Da alle Prozessoren denselben `monitor` benutzen, stellt dieser einen kritischen Abschnitt dar. Der **Monitor** hat folgenden Aufbau:

```
monitor(funktion)
  teste und setze(sm)
  prozeßtabelle[aktiverProzeß].registerfeld:=Prozessorregister
  CASE funktion
     ⋮                    ⎫
     verdrängeProzeß      ⎬ scheduling
     ⋮                    ⎭
  prozeß:=aktiverProzeß
  Prozessorregister:=prozeßtabelle[prozeß].registerfeld
  sm:=1
end-monitor
```

Das Retten und Zurückholen der Prozessorregister bezeichnet man auch als *context switching*. `aktiverProzeß` ist aus der Menge der rechenwilligen Prozesse zu ermitteln (verwaltet in einer Liste, der `bereitliste`). Die `bereitliste` kennt zwei Funktionen:

- `einfüge(prozeß,bereitliste)`, um einen Prozeß der Liste hinzuzufügen
- `entferne(prozeß,bereitliste)`, um einen Prozeß aus der Liste zu entfernen

Mithilfe dieser Funktionen kann die Monitorroutine `verdrängeProzeß` implementiert werden.

```
verdrängeProzeß
  einfüge(prozeß,bereitliste)
  entferne(prozeß,bereitliste)
```

Die Bereitsliste enthält immer mindestens einen rechenwilligen Prozeß, den sogenannten Nullprozeß, der bei der Systeminitialisierung zum aktiven Prozeß gemacht wird.

Weitere scheduler-Routinen :

```
starteProzeß(anfangszustand)
  hole freien PKB(neuerProzeß)
  prozeßtabelle[neuerProzeß].registerfeld:=anfangszustand
  einfüge(neuerProzeß,bereitliste)
```

```
beende(prozeß) {Prozeß beendet sich selbst}
  PKB freigeben
  entferne(aktiverProzeß,bereitliste)
```

Semaphoroperationen Die im Abschnitt 2.2 beschriebenen Semaphoroperationen `up` und `down` können auch als Monitorroutinen implmentiert werden. Dabei ist jeder Semaphore folgende Struktur zugeordnet:

```
semaphore[sem] = record
                   zaehler
                   warteschlange
                 end
```

Damit ergibt sich für `up` und `down`:

```
down(sem):
   wenn semaphore[sem].zaehler > 0
   dann semaphore[sem].zaehler := semaphore[sem].zaehler - 1
        aktiverProzess := prozess {oder verdraenge(prozess)}
   sonst einfuege(semaphore[sem].warteschlane, prozess)
         entferne(bereitliste, aktiverProzess)

up(sem):
   wenn nicht leer(semaphore[sem].warteschlange)
   dann entferne(semaphore[sem].warteschlange, aktiverProzess)
        eifuege(bereitliste, aktiverProzess)
   sonst semaphore[sem].zaehler := semaphore[sem].zaehler + 1
   aktiverProzess := prozess {oder verdraenge(prozess)}
```

Diese Implementierungen benutzen nicht mehr die „beschäftigte Form des Wartens“.

2.4.2 Steuerungsstrategien (*scheduling strategies*)

a) ***round robin* scheduling:**
Jeder Prozeß erhält ein festes Zeitquantum. Wenn dieses abgelaufen ist, wird auf den nächsten Prozeß umgeschaltet. Der unterbrochene Prozeß kommt ans Ende der `bereitliste`. Es handlet sich um eine einfache, faire und häufig benutzte Strategie.
Parameter: Länge des Zeitquantums

- zu kurz: Verwaltungsaufwand (*context switching*) wird sehr hoch
- zu lang: Antwortzeiten werden lang

Erfahrungswert: $100ms$

b) **Prioritätssteuerung:**
Prozesse haben unterschiedliche Prioritäten und werden danach eingeplant. Nur der Prozeß mit der höchsten Priorität darf rechnen. Um „Langläufer“ zu „bremsen“ könnte der Scheduler die Priorität schrittweise vermindern. Die Priorität kann *statisch* oder *dynamisch* zugeordnet werden. Bei dynamischer Zuordnung ändert sich die Priorität nach bestimmten Regeln:

- I/O-bound $\rightarrow$ hohe Priorität
- CPU-bound $\rightarrow$ niedrige Priorität

Einfache Zuordnungsformel: $p = \frac{1}{f}$, mit f = benutzter Anteil des Prozesses an der Rechenzeit während des letzten Zeitquantums.

Erweiterung durch Prioritätsklassen:

- Jeder Prozeß wird einer Klasse zugeordnet.
- Innerhalb einer Klasse wird nach *round robin* scheduling verfahren

Prozesse der zweiten Klasse werden erst bedient, wenn die höchste Klasse leer ist.

c) ***shortest job first*:**
Diese Methode ist anwendbar für Prozesse, deren Gesamtrechenzeit bekannt ist (z.B. *batch-jobs*). **Beispiel:** angenommene Jobfolge:

Job-Nr.:	1	2	3	4	5
CPU-Zeit:	20	10	50	15	5

Die durchschnittliche Rechenzeit (Verweildauer) ergibt:

in der Reihenfolge:

1 :	20	
2 :	30	
3 :	80	
4 :	95	
5 :	100	
$\sum$	325	$\div 5 = 65$

shortest job first:

5 :	5	
2 :	15	
4 :	30	
1 :	50	
3 :	100	
$\sum$	200	$\div 5 = 40$

⇒ Die durchschnittliche Rechenzeit wird minimiert.

d) **guaranteed scheduling:**
z.B. Realzeitsysteme

e) **Scheduling mit Verdrängung von Prozessen**

- Auslagern eines Prozesses (*swapping*)
- zwei-Ebenen-Scheduling
  1.Ebene: Scheduling der Prozesse im Hauptspeicher
  2.Ebene: Ein-/Auslagern von Prozessen

2.5 Zusammenfassung

Sequentielle Prozesse sind Abstraktionsmittel, um von Details auf Maschinenebene (z.B. Interrupts) abstrahieren zu können Ein Betriebssystem ist eine Sammlung parallel ablaufender sequentieller Prozesse. Jeder Prozeß hat eigene virtuelle Betriebsmittel.

Weitere wichtige Konzepte:

- Konzept des kritischen Abschnittes
- Prozeß-Synchronisations-Primitive
- Prozeßzustände
- Scheduling-Strategien

Kapitel 3

Speicherverwaltung

Trotz der rasanten Entwicklung der Speichertechnologie gilt nach wie vor, daß

- schnelle Speicher eine geringe Kapazität und hohe Kosten pro gespeichertem Bit aufweisen,
- große Speicher langsam sind aber geringe Kosten pro gespeichertem Bit aufweisen.

Das Ziel jeder Speicherverwaltung ist es, den Prozessen einen schnellen Speicher scheinbar unbegrenzter Kapazität zur Verfügung zu stellen. Gegenstand der Speicherverwaltung des Betriebssystems sind die Speichertypen:

- Massenspeicher (Plattenspeicher)
- Hauptspeicher (Halbleiter-RAMs)

Abbildung 3.1 zeigt die Aufteilung des Hauptspeichers.

Um einen idealen Speicher, d.h. einen beliebig großen, schnellen und obendrein noch billigen Speicher, mit organisatorischen Mittell nachzubilden, baut man aus verschiedenen Speicherarten eine Hierarchie auf (s. Abbildung 3.2). Die unterschiedlichen Zugriffszeiten resultieren u.a. auch aus unterschiedlichen Zugriffsverfahren. Während Registerspeicher, Cache und Hauptspeicher wahlfreien Zugriff erlauben, ist bei Massenspeichern sequentieller Zugriff erforderlich.

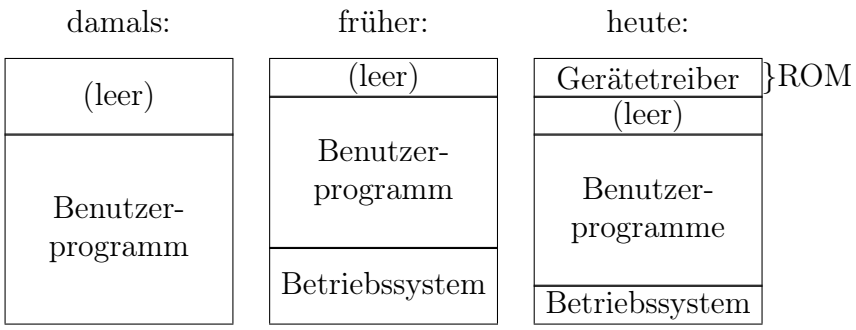


Abbildung 3.1: Aufteilung des Hauptspeichers

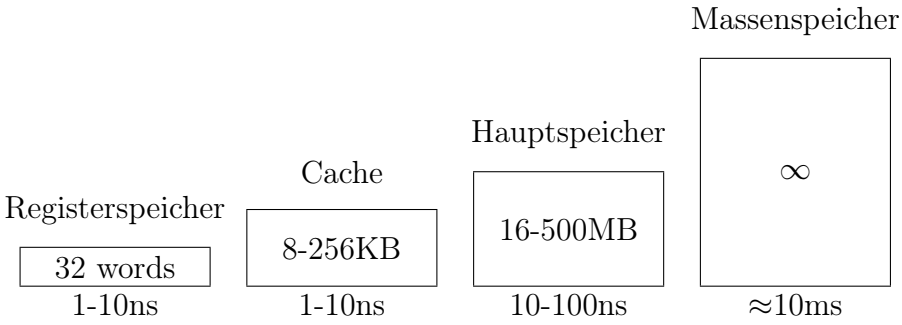
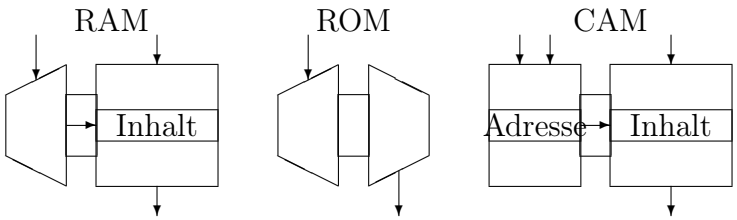


Abbildung 3.2: Speicherhierarchie



CAM = Contents Addressable Memory (*Assoziativ-Speicher*)

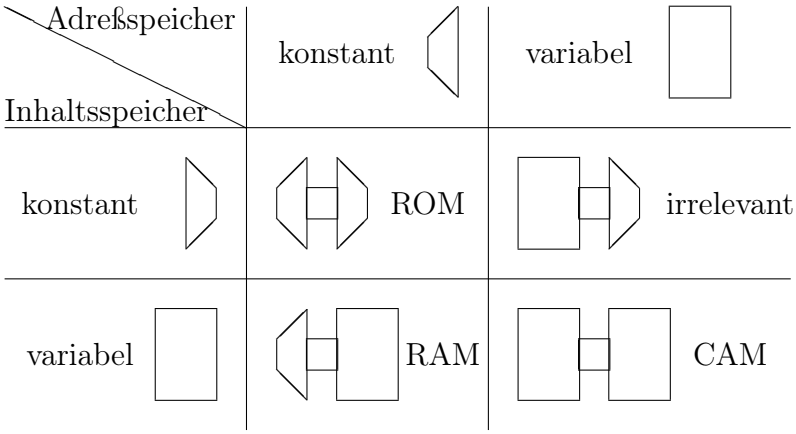


Abbildung 3.3: Varianten von Speichern mit wahlfreiem Zugriff

3.1 Pufferspeicher (Cache-Memory)

Obwohl das Zusammenspiel zwischen Haupt- und Pufferspeicher vollständig durch die Hardware gesteuert wird, soll in diesem Abschnitt kurz die prinzipielle Pufferspeicher-Organisation mit Hilfe eines Assoziativ-Speichers gezeigt werden, da die Prinzipien des Zusammenwirkens zwischen Hauptspeicher und virtuellem Speicher ähnlich sind. Abbildung 3.3 zeigt die heute in Rechnern vorkommenden Speichertypen mit wahlfreiem Zugriff. Ein Assoziativspeicher zeichnet sich dadurch aus, daß nicht nur der Inhaltsspeicher variabel ist, sondern auch der Adreßspeicher.

Der Hauptspeicher wird durch den Pufferspeicher gepuffert. Das bedeutet, daß die aktuell benötigten Daten sich im Pufferspeicher befinden. Wenn sichergestellt werden kann, daß die jeweils vom Prozessor benötigten Daten rechtzeitig vom Hauptspeicher in den Pufferspeicher gebracht werden, wirkt

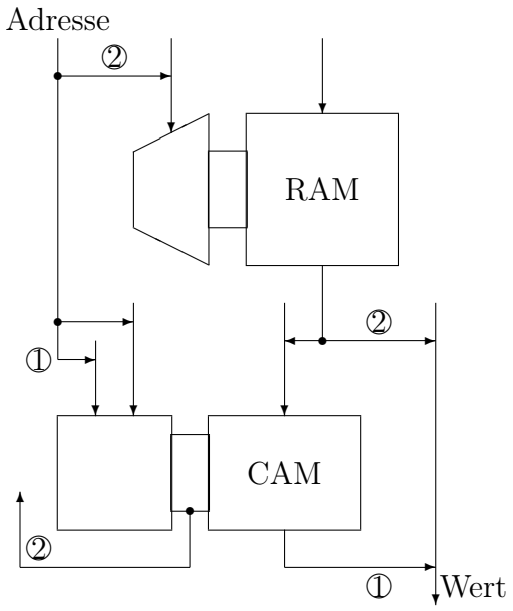


Abbildung 3.4: Zusammenwirken zwischen Haupt- und Pufferspeicher

das System Hauptspeicher/Pufferspeicher wie ein Speicher mit der Geschwindigkeit des Pufferspeichers und der Kapazität des Hauptspeichers. Anders ausgedrückt: Der Pufferspeicher besitzt scheinbar (virtuell) die Kapazität des Hauptspeichers. Die Bezeichnung *virtueller Speicher* ist hier aber nicht gebräuchlich.

Abbildung 3.4 zeigt das Zusammenwirken von Haupt- (RAM) und Pufferspeicher (CAM) bei einem Lesezugriff. Zunächst wird versucht, die Adresse im Cache zu finden. Bei einem Treffer wird der Inhalt ausgegeben (mit 1 bezeichnete Pfeile). Befindet sich die Adresse nicht im Cache, wird auf den Hauptspeicher zugegriffen und der Inhalt ausgegeben. Außerdem werden Adresse und Inhalt in den Pufferspeicher kopiert (mit 2 bezeichnete Pfeile).

Bei Kopieren einer Speicherzelle vom Haupt- in den Pufferspeicher muß dort eine Zelle üeberschrieben werden. Hierfür sind verschiedene Ersetzungsstrategien gebräuchlich:

- FIFO (First In First Out)
- LRU (Least Recently Used)

Bei Schreibzugriffen sind zwei Strategien gebräuchlich um die Inhalte von Haupt- und Pufferspeicher konsistent zu halten:

- *copy back*: Hauptspeicher wird erst beschrieben, wenn die Stelle im Pufferspeicher ersetzt werden soll
- *write through*: Aktualisierung bei jedem Schreibzugriff

Die Effizienz der Pufferspeicher-Organisation hängt von der Trefferquote ab, diese wiederum von dem Verhältnis der Kapazitäten von Haupt- und Pufferspeicher sowie den Ersetzungsalgorithmen.

3.2 Virtueller Speicher

Da schon immer die zur Verfügung stehende Hauptspeicherkapazität für größere Anwendungen nicht ausreichte, gab es auch schon früher Techniken, die Problem zu umgehen. Früher wurde dafür die sogenannte *Overlay-Technik* verwendet, bei der der Programmierer selbst sein Programm in in den Speicher passende Stücke (Overlays) zerlegen mußte und für das Ein- und Auslagern der Teile selbst sorgen mußte. Ausgelagerte Programmteile befinden sich dabei auf einem Hintergrundspeicher (i.d.R. Plattenspeicher).

In modernen Rechenanlagen obliegen diese Aufgaben dem Betriebssystem, das hierzu die *virtueller Speichertechnik* benutzt. Die virtuelle Speichertechnik ist erst mit Mehrprogrammbetrieb effizient nutzbar. Erfordert nämlich die Ausführung eines Prozesses das Nachladen von Speicherbereichen vom Hintergrundspeicher, was, verglichen mit einem Hauptspeicherzugriff, sehr viel Zeit erfordert, so wird die Ausführung des Prozesses unterbrochen und die CPU einem anderen rechenwilligen Prozeß zugeordnet.

Um die Organisation eines virtuellen Speichers zu verstehen, ist zunächst der Begriff des *Adreßraums* zu klären. Im einfachsten Fall sind die im Programm verwendeten Adressen genau die, mit denen auf den Speicher zugegriffen wird: Programm-Adreßraum = Speicher-Adreßraum

Bei virtuellen Speichern werden diese Adreßräume voneinander entkoppelt, d.h. der Programm-Adreßraum wird in den Speicher-Adreßraum umgesetzt. Abbildung 3.5 zeigt schematisch, wie eine Programmadresse (virtuelle Adresse) in eine Speicheradresse mithilfe einer Adreßumsetzungstabelle umgesetzt wird.

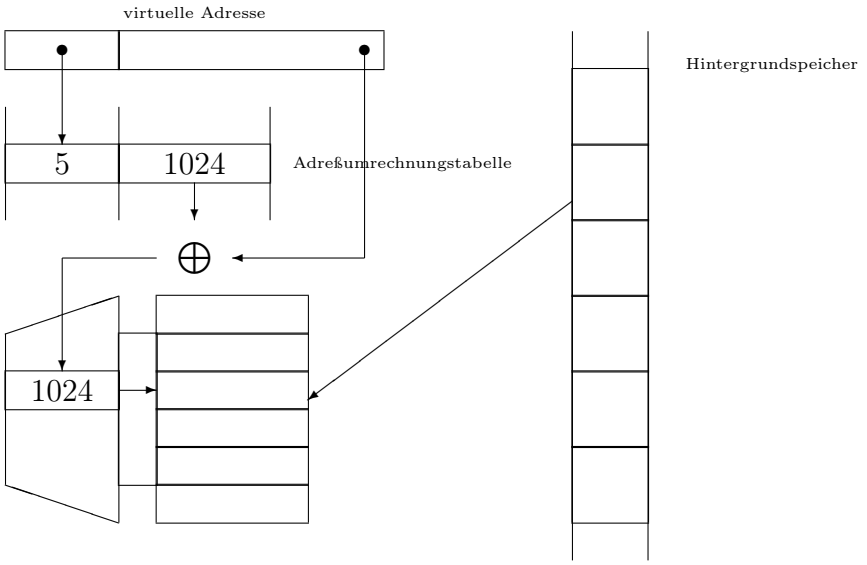


Abbildung 3.5: Umsetzung einer virtuellen in eine reale Adresse

Zu den Aufgaben des Betriebssystems zählt damit auch die Verwaltung des Hauptspeichers hinsichtlich belegter und freier Speicherbereiche. Da Programme vom Betriebssystem an beliebigen realen Adressen plaziert werden können, müssen Programme mit ihren Daten verschiebbar (*relocatable*) sein. Das Betriebssystem muß Sicherheit vor Adreßraumverletzung durch Zugriffsschutz gewährleisten.

Sicherheit bedeutet hierbei:

Bereichsschutz Programm- und Datenteile dürfen bei der Adressierung nicht überschritten werden

Zugriffsschutz Schutz vor unzulässigen Zugriffen.

Aufgaben wie Adreßumsetzung, Bereichsschutz und Zugriffsschutz werden vom Betriebssystem in Zusammenarbeit mit einer Speicherverwaltungseinheit (*Memory Management Unit*, MMU, Spezialhardware) erledigt.

Bezüglich der Zerlegung von Programmen und Daten in Teile, die dann ein- bzw. ausgelagert werden können unterscheidet man grundsätzlich zwei Möglichkeiten: Zerlegung in *Segmente* oder *Seiten*.

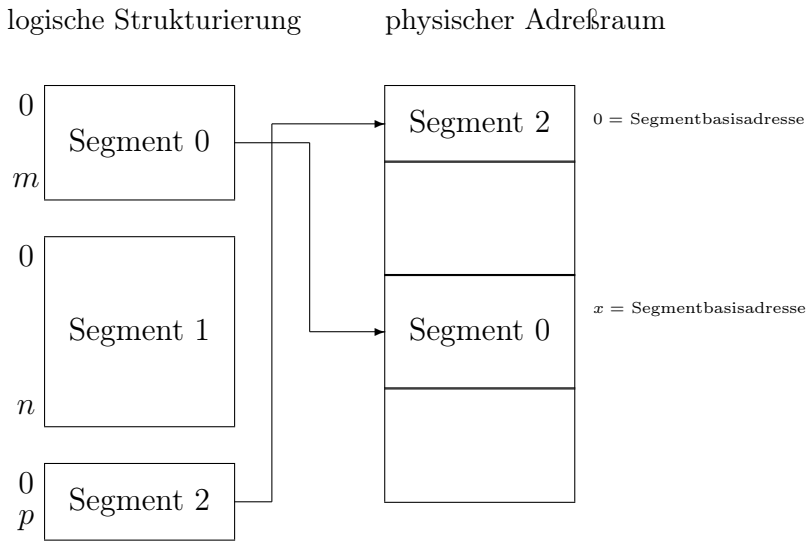


Abbildung 3.6: Abbildung logischer Segmente des virtuellen Adreßraum auf physische Segmente im Hauptspeicher

3.2.1 Logische Zerlegung in Segmente

Die Zerlegung in *Segmente* ist an der Programmstruktur orientiert, z.B. in Prozedursegmente und Datensegmente. Segmente haben variable Längen.

virtuelle Adressierung: (eines Bytes)

- Angabe einer *Segmentnummer*
- Angabe einer *Bytenummer* innerhalb des Segments

reale Adressierung:

- Angabe der *Segment-Basisadresse*
- hinzuaddieren der Bytenummer

Abbildung 3.6 zeigt den Zusammenhang zwischen logischen und physischen Segmenten.

Segmentverwaltung: Jedem Segment ist sind die folgenden Informationen zugeordnet:

- Segmentbasisadresse
- Längenangabe für Bereichsschutz
- Zugriffsattribut für Zugriffsschutz
- Hinweis, ob das Segment im Hauptspeicher geladen ist oder nicht
- Hinweis, ob das Segment im Hauptspeicher verändert wurde (*dirty tag*)

Diese Angaben werden in einem *Segmentdeskriptor* zusammengefaßt. Die Segmentdeskriptoren werden in einer *Segmenttabelle* verwaltet.

Vorteile der Segmentierung:

- Segmente sind *logische* Einheiten, denen spezifische Merkmale zugeordnet werden können.
- Überlappende Segmente sind durch geeignete Wahl von Basisadressen und Längen möglich (z.B. *shared code*, d.h. gemeinsame Nutzung von Programm(teil)en durch verschiedene Prozesse).
- Die Segmentlänge kann dynamisch verändert werden (z.B. *Stack*-Segmente).

Nachteile:

- Die variable Länge der Segmente führt zu aufwendiger Verwaltung der freien und belegten Bereiche im Hauptspeicher (*Freispeicherverwaltung*).
- Segmente müssen komplett geladen bzw. ausgelagert werden.
- Das Ein- und Auslagern variabel langer Segmente kann zu einer ungünstigen Stückelung des freien Speichers im Hauptspeicher führen. Das kann zur Folge haben, daß ein großes Segment nicht geladen werden kann, weil kein ausreichend großes zusammenhängendes Stück Speicher mehr frei ist, obwohl insgesamt noch genügend Speicher vorhanden wäre.

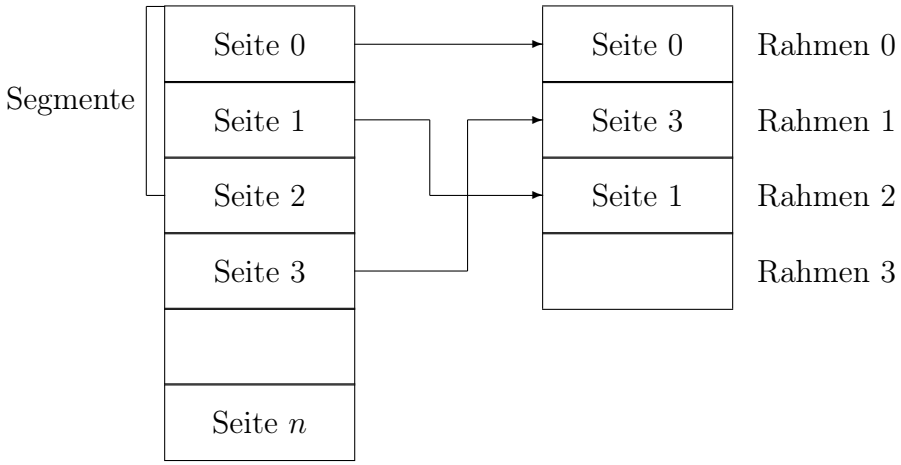


Abbildung 3.7: Abbildung von virtuellen Seiten auf Rahmen im Hauptspeicher

3.2.2 Physische Zerlegung in Seiten

Die Verwaltung des Adreßraumes erfolgt hier nach physischen Gesichtspunkten. Virtueller und realer Adreßraum werden in Bereiche konstanter Größe (Seiten (*pages*) oder Rahmen (*frames*)) zerlegt. Die Größe einer Seite ist immer eine Zweier-Potenz (zwischen 512 Bytes und 8KB). Abbildung 3.7 zeigt die Zuordnung von Seiten des virtuellen Adreßraums zu Rahmen des Hauptspeichers.

Jede Seite wird durch einen Deskriptor beschrieben:

- Seitennummer (Rahmennummer)
- Zugriffsattribut
- Hinweis, ob die Seite geladen ist
- Hinweis, ob die Seite verändert wurde

Abbildung 3.8 zeigt, wie eine realen Hauptspeicheradresse aus Rahmennummer und Bytenummer zusammensetzt wird.

Vorteile der Aufteilung des Speichers in Seiten:

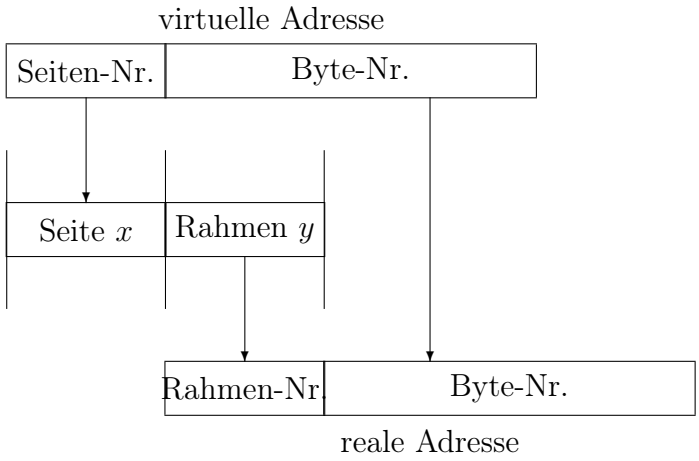


Abbildung 3.8: Zusammensetzung einer realen Hauptspeicheradresse

- Nur die aktuell benötigten Teile des Programms und seiner Daten werden im Hauptspeicher gehalten.
- Das Problem der Freispeicherverwaltung entfällt.

Nachteile:

- Die Zuordnung logischer Merkmale (von Segmenten) muß zu allen betroffenen Seiten erfolgen
- Eine Seitentabelle ist viel größer als eine Segmenttabelle.

Im Zusammenhang mit der virtuellen Speicherverwaltung sind noch folgende Begriffe von Bedeutung:

***Working set*:** Die Menge der einem Prozeß aktuell zugehörigen Seiten (Rahmen) im Hauptspeicher

***Page fault*:** Zugriff auf eine Seite, die sich nicht im Hauptspeicher befindet. Ein Page fault führt zur Unterbechung des Prozessors und Aktivierung des Betriebssystems (Scheduler).

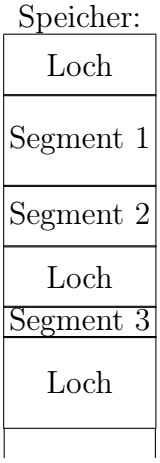


Abbildung 3.9: „Momentaufnahme“ einer Speicheraufteilung

Seitenflattern (*thrashing*): Das Betriebssystem lagert aufgrund einer Überlast rechenwilliger Prozesse ständig Prozesse aus und wieder ein.

Um die Nachteile zu vermeiden bzw. die Vorteile der Segment- und der Seitenaufteilung zu kombinieren, wird häufig eine kombinierte Segment-/Seitenverwaltung benutzt, bei der der Speicher zunächst in logische, variabel lange Segmente und diese wiederum in Seiten fester Länge unterteilt (vgl. z.B. [2]) werden.

3.2.3 Algorithmen für die Plazierung variabel langer Segmente im Hauptspeicher

In diesem Abschnitt soll kurz auf die Probleme einer Freispeicherverwaltung eingegangen werden, wie sie z.B. bei der Verwaltung variabel langer Segmente eines virtuellen Speichers oder auf dem Heap im Laufzeitsystem einer höheren Programmiersprache auftreten. Grundsätzlich kann hierbei die Speicheraufteilung in freie und belegte Bereiche nicht vorhergesagt werden. Eine mögliche Situation zeigt Abbildung 3.9.

Beim Versuch, ein neues Segment im Hauptspeicher zu plazieren, könnten die folgenden Situationen auftreten:

1. Ein Segment soll plaziert werden, mindestens ein Loch ausreichender Größe ist vorhanden.

2. Kein Loch ist groß genug für das zu plazierende Segment, aber die Summe mehrerer Löcher würde ausreichen.

3. Der freie Platz insgesamt reicht nicht aus, um einen anstehendes Segment aufzunehmen.

Fall 1.: Es gibt folgende Strategien:

1. Man wähle das kleinste Loch (*best fit*)
   Vorteil: Große Löcher werden nicht zerlegt.
   Nachteil: Fragmentierung in viele kleine Löcher

2. Man wähle das erste Loch (*first fit*)
   Vorteil: Diese Methode ist am schnellsten

3. Man wähle das größte Loch (*worst fit*)
   Ziel: Löcher sollen möglichst groß sein

Fall 2.: Es wird Platz geschaffen durch Verschieben der Segmente, dabei brauchen nur die Basisadressen der Segmente verändert werden.

Fall 3.: In diesem Fall kann Platz nur durch Auslagern von Segmenten zur Verfügung gestellt werden.

Kapitel 4

Dateisysteme

Aufgabe: persistente (dauerhafte) Speicherung von Daten:

- Programme
- Dokumente

Das Dateisystem bestimmt, wie Dateien

- benannt werden
- strukturiert sind
- zugegriffen werden
- beschützt werden
- implementiert werden
- auf dem Datenträger verwaltet werden

4.1 Dateibenennung

Eine Datei ist ein Abstraktionsmechanismus für auf Datenträgern gespeicherte Daten. Wichtigstes Konzept ist die Namensvergabe.

Es gibt spezifische Konventionen hinsichtlich:

- Groß/Kleinschreibung

- Zeichenvorrat
- Länge (8.3)
- Struktur (datei.exe)

4.2 Dateistruktur

Unterscheidung zwischen logischer und physischer Struktur

logisch: z.B. Pascal: `var f: FILE OF Record`

physisch:

- einfacher Bytestrom. Für das Betriebssystem ist der Bytestrom ohne Struktur (UNIX, MS-DOS)
  → Maximale Flexibilität
  → Maximaler Aufwand für Anwendungsprogramme
- Satzstruktur
  - Folge von Sätzen (*records*) fester Länge (z.B. 80 Zeichen, 132 Zeichen)
    Lese und Schreiboperationen sind satzbezogen
  - Folge von Sätzen variabler Länge
- Baumstruktur
  Datensätze variabler Länge mit Schlüsselfeld als Sortier- und Suchkriterium (Großrechner, kommerzielle DV, ISAM-Dateien)

4.3 Dateitypen

- reguläre Dateien für Benutzerdaten
- Systemdateien (z.B. Dateiverzeichnisse, *directories*)
- Spezialdateien (E/A-Geräte als *character special files*)
  z.B.: con, prn, com$x$ etc.

Reguläre Dateien

- Textdateien (i.d.R. mit Zeilenstruktur) ASCII
- Binäre Dateien

4.4 Zugriffsarten

- sequentieller Zugriff
- wahlfreier Zugriff (*random access*)
  - durch Angabe der Satzposition innerhalb der Datei
  - durch Angabe eines Schlüssels

4.5 Dateiattribute

Schutzattribute
Flags (*read only, hidden*)
Satzattribute (Position des Schlüsselfeldes)
Erzeugungsdatum, Datum der letzten Änderung, Größe

4.6 Dateiverzeichnisse

Hierarchische Dateisysteme, Verwaltung in Verzeichnissen. Verzeichnisse können selbst Verzeichnisse enthalten Arbeisverzeichnis (*working directory*)

Literaturverzeichnis

[1] Brinch Hansen: *Betriebssysteme*

[2] Liebig, Flik: *Rechnerorganisation*

[3] Schnupp: *Standard Betriebssysteme*

[4] Tanenbaum: *Modern Operating Systems*

Abbildungsverzeichnis