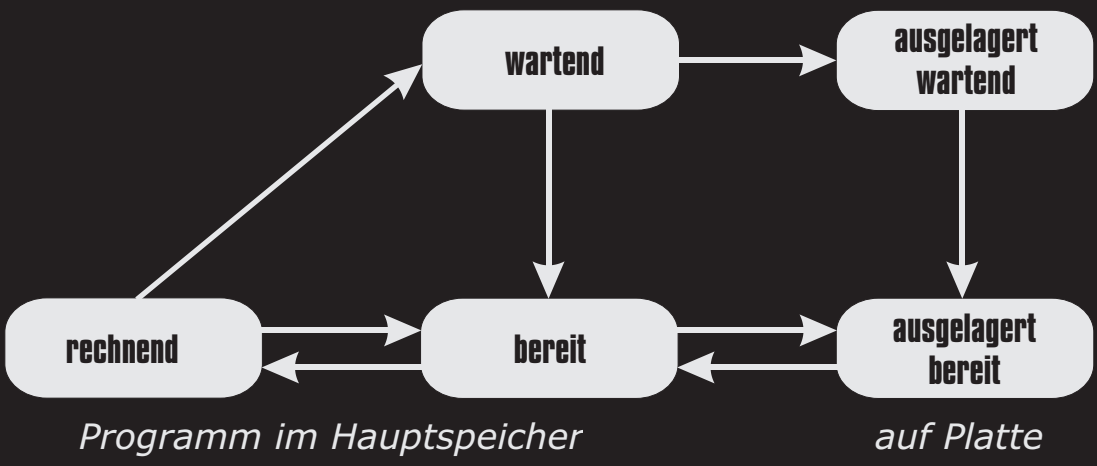
Lutz Wegner

Grundlagen der Betriebssysteme

Universität
Kassel



SKRIPTEN DER
PRAKTISCHEN INFORMATIK

Grundlagen der Betriebssysteme

Lutz Wegner

Grundlagen der Betriebssysteme

3. Auflage Oktober 2003

Skriptum zur Vorlesung „Betriebssysteme“ an der Universität Kassel im Wintersemester 2003/04.

Anschrift:
Prof. Dr. Lutz Wegner
Universität Kassel
FB 17 Mathematik/Informatik
Heinrich-Plett-Straße 40
D-34109 Kassel

`www.db.informatik.uni-kassel.de`

Vorwort

Auf dem Gebiet der Betriebssysteme haben sich über die Jahrzehnte einige Grundlagen, schöne Algorithmen, Klassifikationen, fundamentale Beweise und Einsichten herausgebildet. Genannt seien die Prozeßsynchronisationsverfahren, etwa so klassische Ansätze wie *Dekker's Algorithmus* mit globalen Variablen, daneben *Dijkstra's Semaphore*, *Hoare's Monitore* und die aus UNIX bekannten praktischen Verfahren wie *Streams*, *Pipes*, *Sockets*, *Threads*.

Genauso gehören zum „ehernen Bestand an Informatikwissen" Speicherverwaltungsstrategien wie *First-Fit*, *Best-Fit*, *Worst-Fit*, Seitenverdrängungsmethoden wie *LRU*, *FIFO*, *Second-Chance* und *Belady's Anomalie*. Verklemmungsentdeckung und ~vermeidung, heute mit Betonung auf verteilten Systemen, wären zu nennen und die auf der Warteschlangentheorie aufbauenden Beobachtungen zur Systemauslastung bei *Round-Robin* Auftragsabarbeitung.

Die kurze Liste zeigt bereits, daß es keinen Mangel an Grundlagenwissen gibt und daß die Kunst vermutlich darin besteht, diese fundamentaleren Einsichten mit Beispielen aus Implementierungen von Betriebssystemen anschaulich zu kombinieren. Letzteres ist allerdings aufwendig, zumal die Dokumentation eines Betriebssystems dazu neigt, grundsätzliche Eigenschaften unter einem Berg von technischem Detail und Features zu begraben. Als positive Ausnahmen sei aber auf die Werke von *Maurice Bach* und *Andrew Tanenbaum* hingewiesen, die auf der Grundlage von UNIX eine geglückte Symbiose von Grundlagen und Anwendungen bieten.

Dieses Skript hier baut nicht auf einem speziellen Betriebssystem auf und ist aus einer Sammlung von Folien hervorgegangen, deren Ursprünge bis auf ein Skript des Kollegen *Nehmer* (Kaiserslautern) aus den Siebziger Jahren zurückgehen. Der zweite wesentliche Einfluß stammt aus dem didaktisch hervorragenden Buch von Ben-Ari *Grundlagen der Parallel Programmierung* (1984).

Nicht behandelt werden die anspruchsvolleren Themen zur formalen Spezifikation und Verifikation nebenläufiger Prozesse, z.B. mittels der *Z-Notation* oder der sog. *temporalen Logik*. Genausowenig wird die Verklemmungsentdeckung in verteilten Systemen, zu der es heute eine irritierende Anzahl an Publikationen gibt, weiter vertieft. Diese Themen wären für sich anschließende Spezialvorlesungen geeignet, die auf die im wesentlichen intuitiv begründeten Argumente und Plausibilitätsüberlegungen dieser Vorlesung aufbauen könnten.

Kassel, im Oktober 1997

Lutz Wegner

Inhaltsverzeichnis

1 Einführung

1.1 Aufgaben

Aufgabe eines Betriebssystems (kurz BS) ist es, dem Benutzer die Dienste eines Rechensystems nutzbar zu machen, speziell auch die *effiziente*, *gemeinsame* Nutzung zu ermöglichen. Dahinter verbergen sich verschiedene Anforderungen, die auch als Gütekriterien verwendet werden können.

- Bereitstellung einer ergonomischen Benutzerschnittstelle
- Abstraktion von technischen Einzelheiten des Rechensystems, Konzept der *virtuellen Maschine*
- Einsetzbarkeit für unterschiedliche und wechselnde Einsatzfelder, ggf. Koexistenz mehrerer Betriebssysteme
- robuster Schutz der Benutzer und des Rechensystems vor unbeabsichtigt oder mutwillig herbeigeführten Schäden
- langfristige Haltung von Daten und Programmen und Schutz vor Ausfällen
- effiziente Nutzung des Systems, geringer „Overhead“, ggf. Abrechnung der Leistungsnutzung.

Aus programmtechnischer Sicht ist ein BS eine Ansammlung kooperierender Programmkomponenten, die in der Regel als nebenläufige (quasiparallele) Prozesse ablaufen. Ein *Prozeß* ist dabei einfach *ein Programm in Abarbeitung* (näheres später). Die DIN Norm 44300 definiert ein Betriebssystem wie folgt.

Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlagen die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesonderer die Abwicklung von Programmen steuern und überwachen.

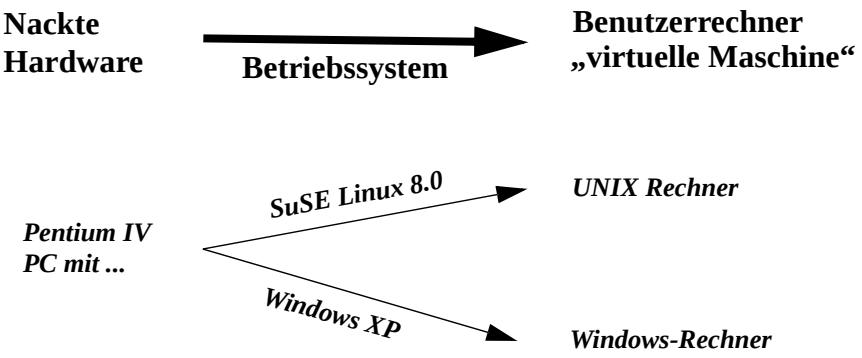


Abb. 1–1 *Rechner als virtuelle Maschine*

1.2 Betriebsarten

Man unterscheidet verschiedene Arten nach ihrer Nutzung.

1.2.1 Stapelbetrieb (batch processing)

Abarbeitung von Aufträgen (Rechenjobs) *ohne Interaktion*, z.B. Abwicklung von Überweisungsverkehr zwischen Großbanken nachts, Rechnungserstellung Stadtwerke, Rentenbescheide. Ein- und Ausgabe werden häufig getrennt zuerst auf Platte zwischengespeichert (sog. *Spooling*[1]), damit die langsamen Belegleser und Drucker die Verarbeitung nicht bremsen.

1.2.2 Dialogbetrieb

Interaktive Form der Nutzung, z.B. für Online-Eingaben, Textverarbeitung, Web-Nutzung. Kann aufgrund der langsamen Reaktionszeiten der

1. Simultaneous Peripheral Operations On Line

Teilnehmer wird der Prozessor der Anlage nur schwach genutzt. Deshalb bietet es sich an, **Dialogbetrieb** mit **Mehrprogrammbetrieb** (multi-programming, multi-processing), ggf. **Mehrbenutzerbetrieb** (multi-user) zu koppeln. Die Vergabe von Rechenzeit an einzelne Teilnehmer kann z.B. in einem **Zeitscheibenverfahren** reihrum geregelt werden. Man spricht dann von **Time-Sharing**.

Mehrbenutzerbetrieb setzt Mehrprogrammbetrieb voraus, aber nicht umgekehrt. Mehrprogrammbetrieb kann auf einer *Einprozessormaschine* nicht echt parallel erfolgen, vielmehr wird der Prozessor *gemultiplexed* (hin- und hergeschaltet), etwa über das Zeitscheibenverfahren. Aufgrund der hohen Rechengeschwindigkeiten entsteht aber der Eindruck der simultanen Nutzung der Anlage. Auf *Mehrprozessormaschinen* ist echte Parallelverarbeitung mit mehreren Arbeitsformen möglich. Eine recht alte Klassifikation geht auf Michael Flynn 1972 [17] zurück und unterscheidet (SISD, SIMD, MISD, MIMD, also single data single instruction, single instruction multiple data, ...). Physisch paralleles Arbeiten kann zusätzlich räumlich verteilt sein, ggf. mit einem Betriebssystem für verteiltes Rechnen. Eine breite Übersicht über *Parallel and Distributed Computing* gibt Leopold [15], wir gehen darauf nicht ein.

Aufgaben des BS im Dialogbetrieb sind u.a.

- „Aufsammeln“ der Eingaben an den Terminals und Inter-pretation als Kommandos; Techniken: polling und interrupt
- Laden und Entladen der Benutzerprozesse von/auf Hintergrundspeicher
- Ausgabe von Information an den Terminals und/oder in einer Mehrfensterumgebung; bei graphischen Benutzeroberflächen z.B. Einsatz eines *X-Servers* (auf dem Anwender-PC!), der einer Anwendung auf einem Anwendungsserver, z. B. einem Datenbankmanagementsystem (DBMS), die graphische Aufbereitung und Interaktion abnimmt.
  Hinweis: Man beachte die wechselseitig verdrehte Form von Client und Server.
- Verwaltung der Benutzerdaten

1.2.3 Transaktionssystem

Spezialfall eines Dialogsystems für hohen Durchsatz, z. B. Flugreservierungen, Fahrkartenterminals, Server für Bankautomaten. Kennzeichnend sind

- sehr viele Terminals
- formatierte Datenübertragung im Blockmodus
- oft nur ein Anwendungsprogramm
- Operationen starr und oft einfach

Ähnlich auch **Serverbetrieb**, etwa für eine Web-Anwendung mit hoher Internet-Seitenzugriffsrate. Der Serverbetriebsart liegt häufig ein Auftragsmodell (Client-Server-Beziehungen) zugrunde. Dies setzt eine geeignete Interprozeßkommunikation (IPC) voraus. Hierauf gehen wir in einer eigenen Vorlesung ein, Teile werden hier in einfacher Form behandelt.

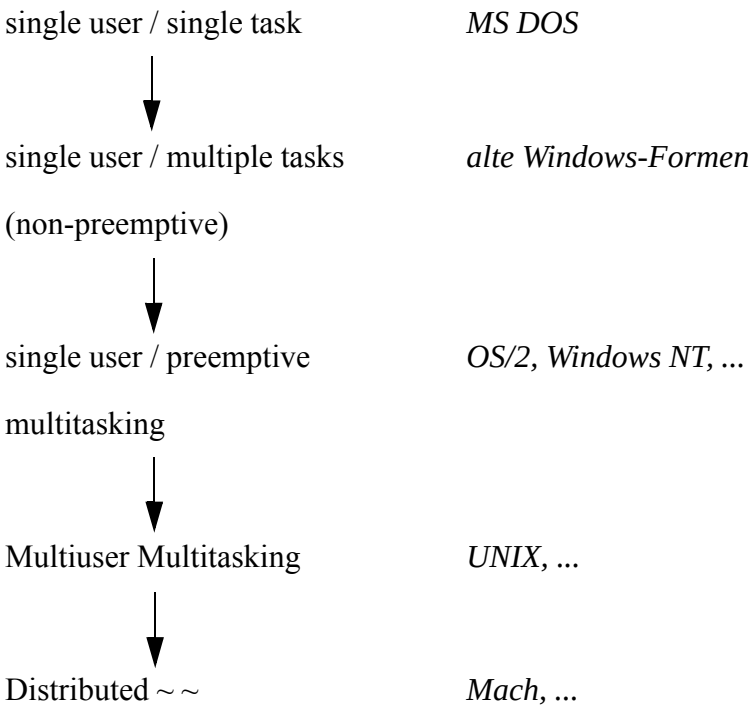


Abb. 1–2 *Weitere Begriffe und Beispiele zu Betriebsarten*

Der in Abb. 1–2 verwendete Begriff *preemptive* bezieht sich auf die Fähigkeit des Betriebssystems, den Prozessor einem arbeitenden Programm jederzeit zu entziehen, etwa aufgrund einer Zeitscheibenregelung. Den Vorgang der Prozessorzuteilung nennt man *Scheduling*. Bei Systemen, die kein preemptive scheduling können, setzt der Prozessorwechsel zu einem anderen Prozeß den Eintritt eines Warteereignisses für den laufenden Prozeß voraus. Ein solches externes Ereignis, z.B. das Erreichen eines Lesebefehls zur Eingabe von Tastatur oder Platte, hat immer eine Abgabe des Prozessors zur Folge. Dazu mehr später.

Die letzte wichtige Betriebsart ist

1.2.4 Echtzeitbetrieb (real time processing)

Hier ist die Rechenanlage meist in einen *technischen Prozeß* eingebettet, etwa zur Prozeßsteuerung in Werkzeugmaschinen, Transport- und Verkehrssystemen, Walzstraßen, usw. Die Peripherie enthält Stellglieder, Taktgeber, Grenzwertmelder, Sensoren aller Art. Benutzerprogramme können in der Regel als Teil des Betriebssystems gesehen werden. An die *Antwortzeiten* (*Reaktionszeiten*) werden „harte“ Anforderungen gestellt, die über Prioritäten gesteuert werden, und für die Zusicherungen gemacht werden müssen.

Mischformen der Betriebsarten sind üblich. So muß ein Server für *Video-on-demand* in einer kommerziellen Anwendung praktisch im Echtzeitbetrieb laufen. Fehler im Plattenzugriff (etwa durch ein periodisches Selbstkalibrieren der Platten) führen zu Bildaussetzern (Ruckeln), die Kunden nicht tolerieren.

1.3 Schichtenarchitektur

Moderne Betriebssysteme sind komplexe Softwaresysteme.

BS	Lines-of-Code (LOC)	davon Treiber
Windows NT 4.0 (96/97)	18,9 Mio	
Windows 2000	29 Mio (35 Mio?) C++	8 Mio
LINUX (2002)	30 Mio (?)	
LINUX 2.1 (1998)	1,5 Mio	
SunOS 4.03 (Sun 3, Sun 4)	2, 4 Mio	
Windows NT 3.1 (1993)	6 Mio	
QDOS (Vorläufer von MS-DOS) von Gates für 50.000$ von Tim Paterson gekauft	ca. 8.000 Zeilen Assembler	davon 300 mit Bugs

Tab. 1–1 *Größe von Betriebssystemen*

Hinweis: Die historische Entwicklung von UNIX kann man z. B. in einem wunderbaren Artikel *The Evolution of the Unix Time-sharing System* von Dennis Ritchie aus dem Jahr 1979 nachlesen.

```
http://cm.bell-labs.com/cm/cs/who/dmr/hist.html
```

Folglich kann man ein solches System nur in einer *Schichtenarchitektur* aufbauen. Allerdings unterscheiden sich die Betriebssysteme stark darin, ob sie *monolithisch* (spez. Windows, BS2000, IBM/VMS) oder *modular* (alle UNIX Varianten) auftreten. Mit der Frage „Ist der Kommandointerpreter austauschbar“ läßt sich dies leicht testen.

Im Fall von UNIX kommt zusätzlich dazu, daß mit dem POSIX-Standard eine genormte Bibliothek für Systemaufrufe zur Verfügung steht. Diese wird neuerdings auch von Windows 2000 in Form eines API (Applications Programmers Interface) unterstützt.

Als Beispiele für Systemarchitekturen seien aus Silberschatz/Peterson/Galvin [6] die OS/2 Architektur und die UNIX-Architektur gezeigt.

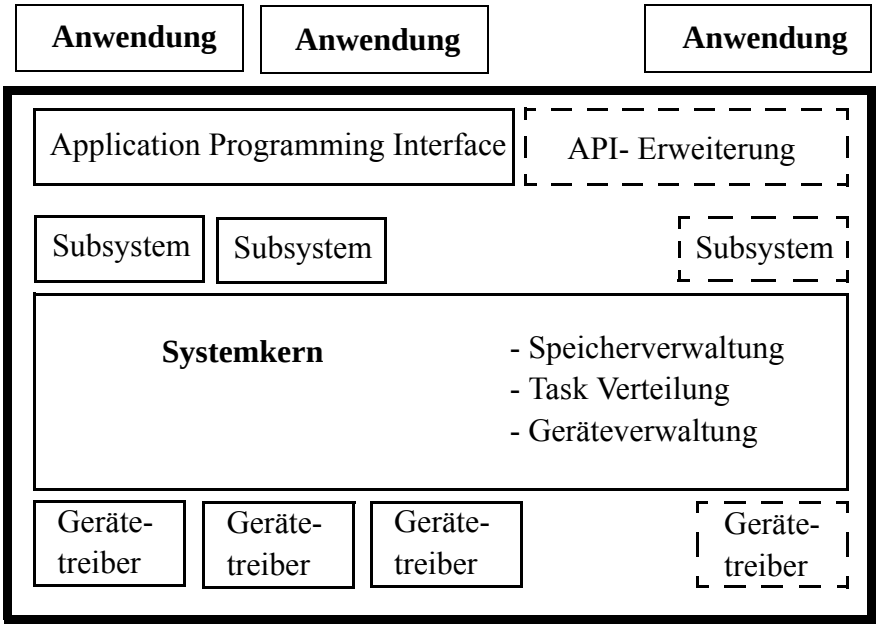


Abb. 1–3 *OS/2 Schichtenarchitektur*

Ein weiterer Architekturansatz ist das Konzept des „micro kernel“ (Mikrokern), der minimale Funktionalität für Prozessorumschaltung, Hauptspeicherverwaltung und Unterbrechungsbehandlung übernimmt.

Alle anderen Aufgaben werden an Dienstprozesse delegiert, die untereinander (über die *Systems Programmers Interface*, SPI, des Mikrokerns) per Botschaftenkonzept miteinander kommunizieren. Die Abb. unten zeigt zusätzlich die Verwendung eines Mikrokerns zur Realisierung virtueller Maschinen.

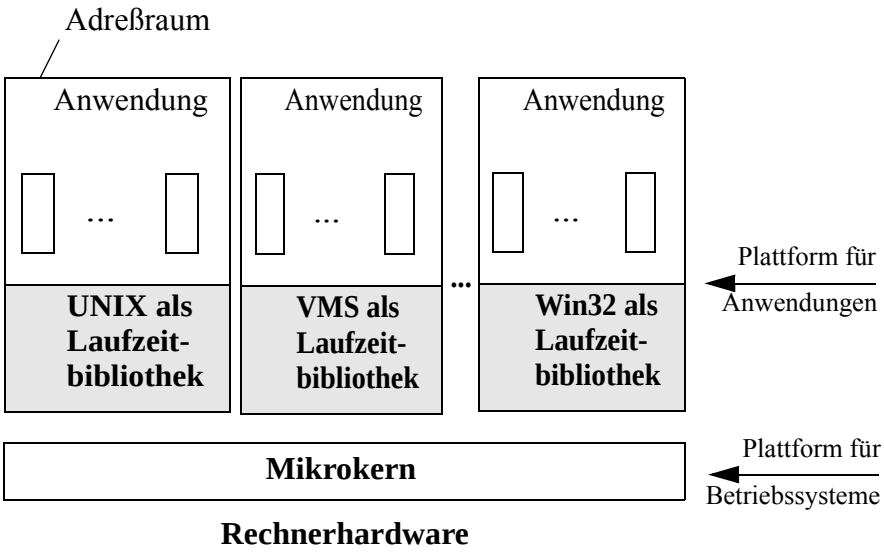


Abb. 1–4 *UNIX Systemstruktur*

Anwender		
Shell und Kommandos / Compiler und Interpreter / Systembibliothek		
Systemaufrufe als Schnittstelle zum BS-Kern		
signals	file system	CPU scheduling
terminal handling	swapping	page replacement
character I/O	block I/O	demand papging
terminal drivers	disk and tape drivers	virtual memory
BS-Kern Schnittstelle zur Hardware		
terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

Abb. 1–5 *Virtuelle Maschinen auf Mikrokernel aufgesetzt*

Anders als ein normaler Prozeß wird der Microkernel „nichtblockierend“ ausgelegt, d.h. seine Betriebszustände sind nur *rechnend* und *bereit*, nicht *wartend* (siehe weiter unten unter „magisches Dreieck“ der Prozeßzustände)

1.4 Programm, Prozessor, Prozeß

Ein zur Ausführung bereites *sequentielles Programm* ist eine Folge von Befehlen (Instruktionen) und Datenfeldern, die u.U. durch Übersetzung aus den Anweisungen und Vereinbarungen eines Quellprogramms in einer höheren Programmiersprache entstanden ist. Seine Ausführung heißt *Prozeß*. Dazu wird ihm ein *Prozessor* als ausführendes Organ zugeordnet, Befehle und Daten stehen (zumindestens die gerade auszuführenden Teile) im Hauptspeicher.

Befehlszyklus (fetch/execute cycle)

Sei `M[0..`*n*`]` der Hauptspeicher (memory), BR das Befehlsregister, BZ der Befehlszähler

```
repeat
  BR := M[BZ];
  BZ := BZ + 1;
  ausführen(BR)
until Prozessor wird abgeschaltet;
```

Dabei sind Befehlszyklen heute so ausgelegt, daß sie „*wiederstartbar*“ (re-startable) sind. So wird im Fall eines Fehlers, etwa wenn ein Wert aus dem virtuellen Speicheradressraum des Prozesses nicht verfügbar ist (z.B. Seite nicht eingelagert, sog. Seitenfehler), zunächst in eine Unterbrechungsroutine verzweigt und danach wird der fehlende Teil wiederholt. Nebeneffekte der Instruktion, z.B. das Hochzählen eines Zählregisters, usw. dürfen natürlich nicht doppelt ausgeführt werden.

Ein Prozessorwechsel von einem Prozeß zum anderen findet in der Regel aber nur nach einem vollständigen fetch/execute-Zyklus statt.

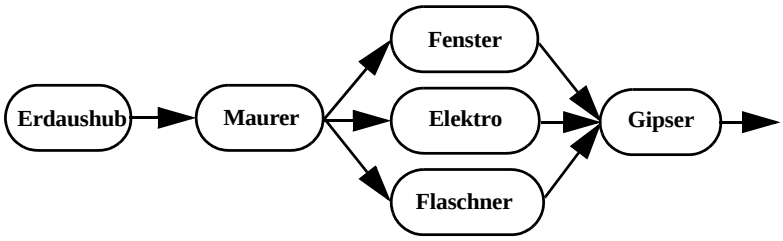
Ein *nebenläufiges* (nicht sequentielles, *concurrent*) Programm besteht aus zwei oder mehr sequentiellen Programmen, deren Ausführung gleichzeitig als *parallele Prozesse* erfolgt.

Dazu kann ein Prozessor zwischen den Prozessen hin- und hergeschaltet (gemultiplexed) werden, oder jedem Prozeß wird ein Prozessor zugeordnet. Entsprechend unterscheidet man *Multiprogramming* und *Multiprocessing*.

Multiprogramming bietet sich besonders dadurch an, daß die langsamen Ein-/Ausgabeoperationen an unabhängige Kanalprozessoren (heute DMA-Bausteine[1]) delegiert werden, der wartende Prozeß in dieser Zeit den Prozessor abgibt, und dadurch andere bereite Prozesse rechnen können.

Analogien für parallele Aktivitäten:

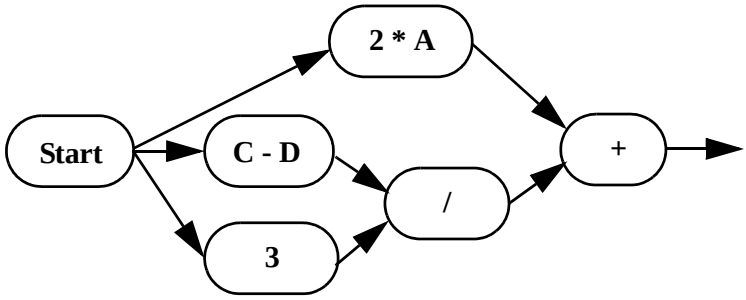
a) großes Projekt, z. B. Hausbau



Darstellung als *Präzedenzgraph*, GANTT- und PERT-Diagramme; geben an, „was kommt vor was“ („what precedes what“) kommen muß. Daraus erkennbar Wartezwänge, kritische Pfade (Aktivitäten ohne Schlupf). Abbildbar mittels ein Prozeß je Aktivität.

1. DMA - *Direct Memory Access*; direkter Datentransfer (ohne Einschaltung des Microprozessors) zwischen einem Peripheriegerät und dem Hauptspeicher.

b) unabhängige Programmteile, z. B. arithmetischer Ausdruck (2*A) + ((C - D) / 3)



c) parallele Suche

```
for i := 1 to Tabellengröße do
if Eintrag[i] = Suchwert then found := true
```

d) Simulation, z.B. Betrieb eines Hafens (Belegung, Verkehr, Auslastung, Engpässe). Jedes Schiff ein Prozeß, ggf. mit Zufallseffekten.

Notation

Einzelne sequentielle Programme als Prozeduren in Pseudo-Pascal mit Endlosschleife. Paralleler Start der Prozesse (Prozeduren) P_1, P_2, ..., P_n im Hauptprogramm mit

cobegin P_1; P_2; ...; P_n **coend**

2 Prozesse

2.1 Prozeßzustände

Ein *Prozeß* war definiert als ein Programm, dessen erster Befehl bereits ausgeführt und dessen letzter noch nicht erreicht wurde, d.h. ein Prozeß ist ein Programm in Abarbeitung.

Die Programmausführung (vgl. fetch/execute-Zyklus) kann als *Folge von Zuständen* betrachtet werden. Die Zustände sind gekennzeichnet durch

- nächste auszuführende Instruktion
- Inhalt aller Register und Programmstatusanzeigen
- Inhalt des zugehörigen Datenteils

Programmunterbrechungen ereignen sich durch Prozessorentzug (z.B. im Time-sharing) oder Prozessorabgabe wegen Wartezwang oder externer Unterbrechung. In diesen Fällen des *Prozeßwechsels* (context switch, task switch) muß der *Programmzustand* (eigentlich genauer Prozeßzustand) gerettet werden.

Beispiel 2–1

Hypothetischer Stack-Computer mit symbolischer Maschinensprache HYML

Notation

[*x*]	Wert (Inhalt) der mit *x* bezeichneten Variablen, bzw. des mit *x* bezeichneten Stapelelements
top	Adresse des oberstes Stapelelements
top-1	Adr. des zweitobersten Stapelelements
pop	Funktion, die oberstes Stapelelement entfernt (top := top - 1)
push(*w*)	Funktion, die *w* als neues oberstes Element auf dem Stapel ablegt, d.h. galt vorher [top] = *v*, dann gilt nachher [top-1] = *v* & [top] = *w*.

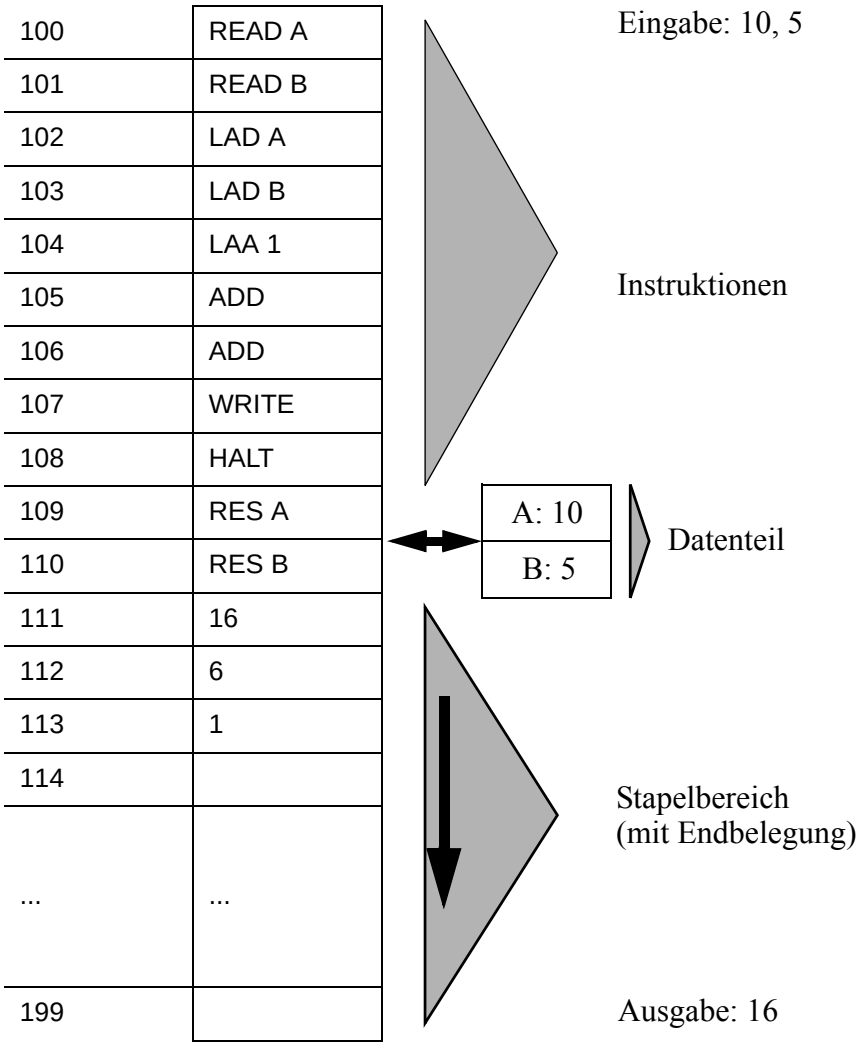
Befehlsvorrat

Befehl	Operand	Semantik
RES	Name *n*	reservieren für Variable *n* [*n*] unbestimmt
READ	Name *n*	[*n*] := Eingabewert
WRITE	-	Ausgabewert := [top]; pop
LAD	Name *n*	push([*n*])
LAA	Zahl *i*	push(*i*)
STO	Name *n*	[*n*] := [top]; pop
ADD	-	[top-1] := [top-1]+[top]; pop
HALT	-	Ende der Programmausführung

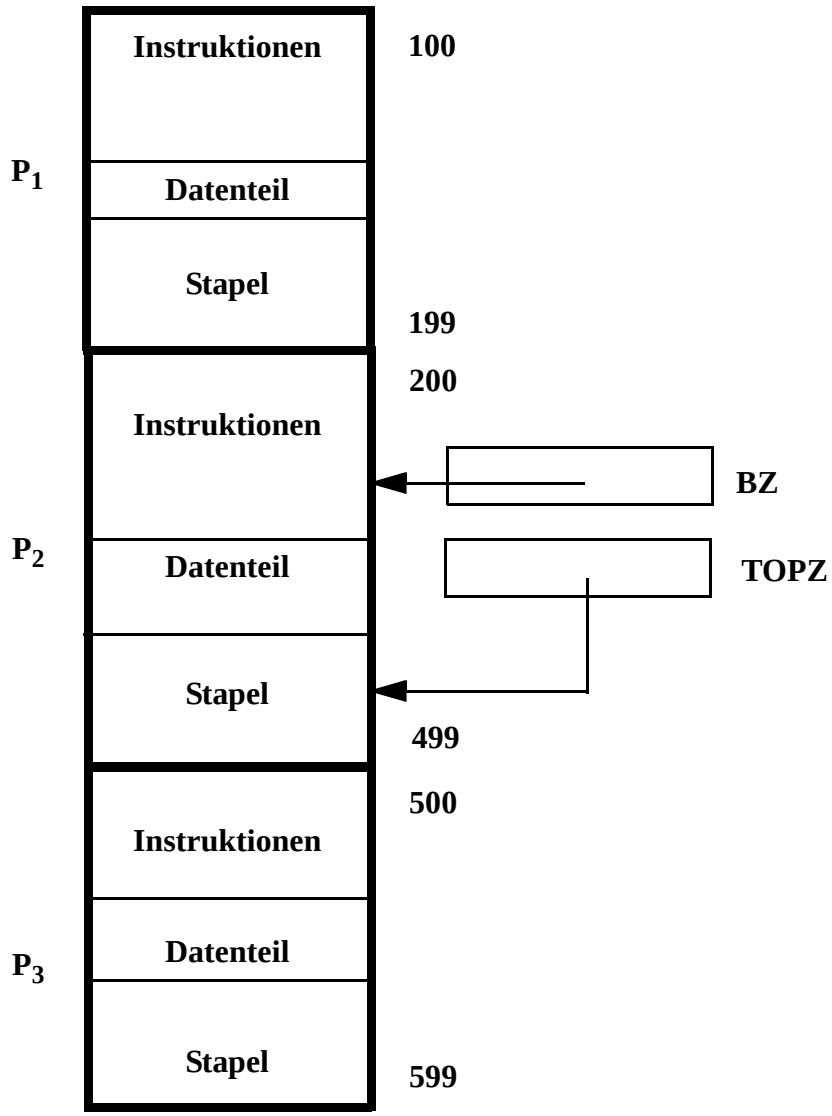
Beispielprogramm

```
lies A, B; drucke A+B+1
```

Speicherbild



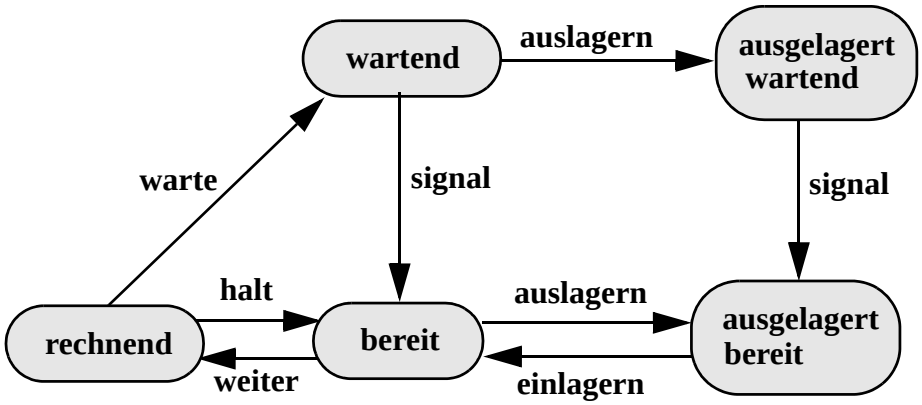
Speicherbild für mehrere HYML-Programme im Hauptspeicher



Bei einer Programmunterbrechung sind demnach zu retten:

> BZ, TOPZ (Stapelzeiger), Stapeluntergrenze, Stapelobergrenze (hier gleich Ende des Adreßraums), event. Adresse der 1. Instruktion, Status
> ferner: Prozeßidentifikator („Name“, ProzeßID), VaterprozeßID, Prozeßbesitzer, ...

Prozeßstatus (Magisches Dreieck der Prozeßzustände)



Programm im Hauptspeicher *auf Platte*

Andere synonyme Begriffe:

- rechnend = aktiv, running
- bereit = ready, runnable, sleeping (UNIX)
- wartend = waiting, unrunnable, blocked, blockiert
- auslagern = swap out, einlagern = swap in

Wichtig: Der Status *bereit* heißt, der Prozeß wartet auf Zuteilung des Prozessors, etwa im time sharing Betrieb.

Der Zustand *wartend* heißt, der Prozeß wartet auf den Eintritt eines externen Ereignisses; er könnte nicht laufen bei Zuteilung des Prozessors.

Richtig oder falsch? (aus: mc 2/1985, S. 50)

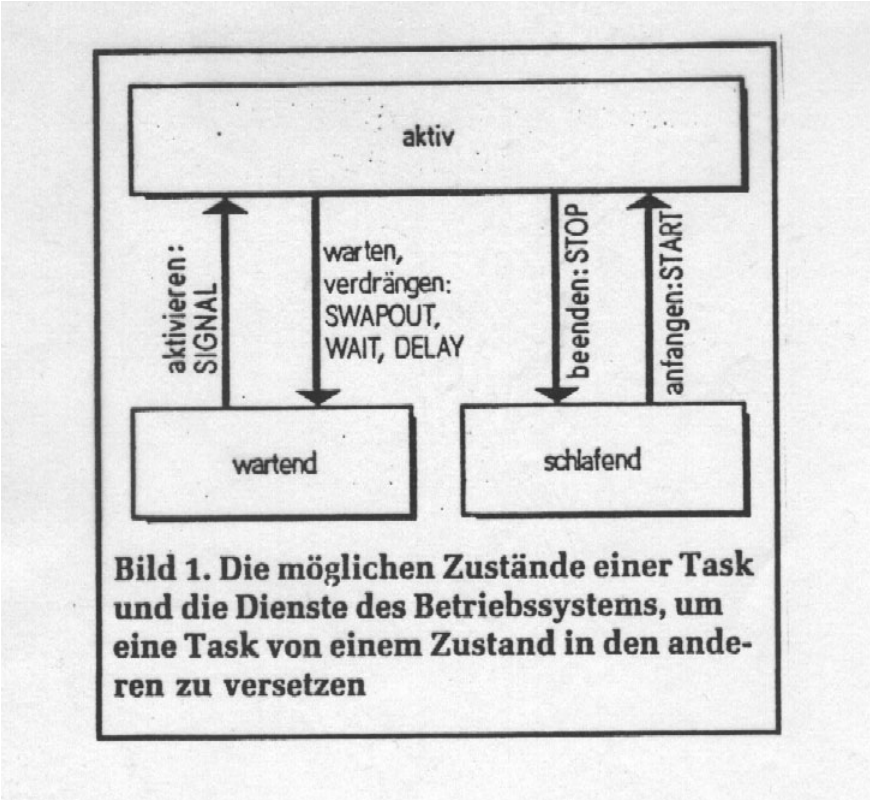


Bild 1. Die möglichen Zustände einer Task und die Dienste des Betriebssystems, um eine Task von einem Zustand in den anderen zu versetzen

2.2 Prozeßverwaltungsinformation (PCB)

Zur Verwaltung der Prozesse existiert im Adreßraum des BS-Kerns eine *Tabelle* und/oder eine *Warteschlange von Prozeßkontrollblöcken*, die als lineare Liste realisiert ist. Die Prozeßkontrollblöcke (PCBs) sind oft unterteilt in

- *Hardware PCB*, der in den Prozessor beim Übergang zu rechnend geladen wird (daher möglichst klein), Registerinhalte und Speichergrenzen
- *Software PCB*, HS-residend, relativ groß, zusätzliche Verwaltungsinformation, z.B. zugeteilte Peripherie, offene Dateien, zugeteilte

Speichergrößen und Rechte, Prioritäten, anhängige Ereignisse, Statusinfo, Besitzer, verbrauchte Zeit, Adresse des HW-PCB, ...

Vgl. UNIX PCB in Dateien `/usr/include/sys/{proc.h, user.h}`

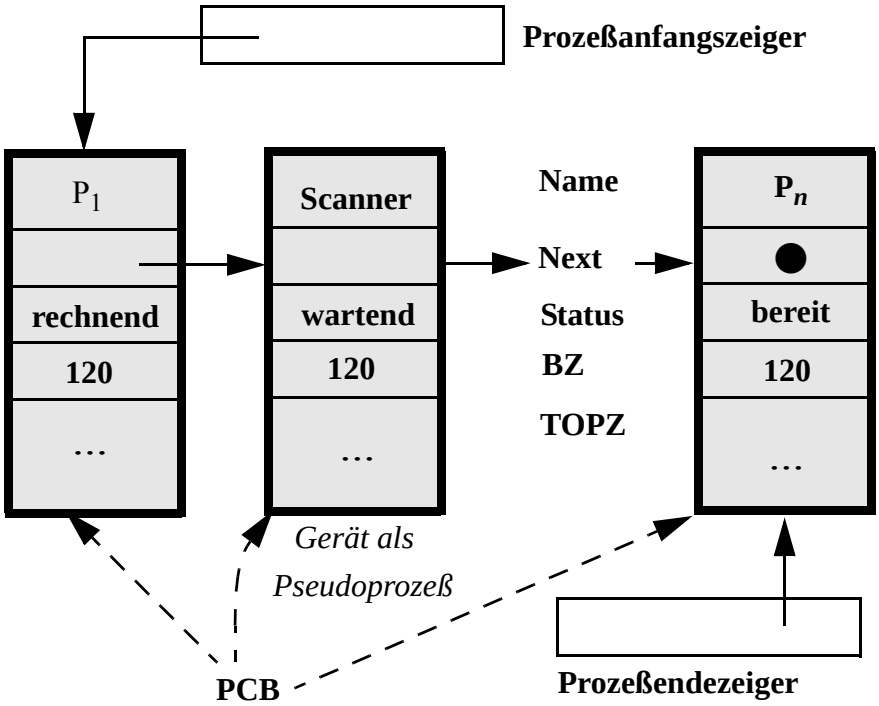


Abb. 2–1 *Verwaltung der Prozeßkontrollblöcke*

2.3 Unterbrechungen

Der Wechsel von einem Prozeß zum anderen wird über Unterbrechungen gesteuert. Dabei sind für das Erkennen eines Unterbrechungsereignisses zwei grundsätzliche Methoden möglich.

1. Regelmäßige Abfragen - Polling
   Unterbrechung nach *n* Takten und Abfrage reihum.

Vorteil: einfach zu realisieren
Nachteil: hoher Zeitaufwand im Ablauf

- runterzählen Taktzähler
- prüfen auf Null
- wenn = 0, prüfen ob Unterbrechungsursache vorliegt
- Gerät muß ggf. bis zu n Takte warten, bis es gestartet wird

2. Mittels Unterbrechungsvektor (interrupt vector)
   Zuordnung *m*-te Unterbrechungsursache (Signal von Gerät *m*) zu *m*-tem Bit im Vektor (ggf. mehrere Ursachen abgebildet auf ein Bit)
   Vorteil: geringer Zeitaufwand, schnelles Reagieren
   Nachteil: Prozessor muß auf Unterbrechungsvektor ausgelegt sein

Neuer Fetch/execute-Zyklus

```
repeat
   if InterruptVektor = 0
   then begin
      BR := M[BZ];
      BZ := BZ + 1;
   end
   else BR := M[Unterbrechungsadresse];
   ausführen(BR)
until Prozessor wird abgeschaltet;
```

Grundstruktur der Unterbrechungsroutine

```
begin
   retten Zustand des unterbrochenen Programms;
   i:= Index des Bits, das Unterbrechung ausgelöst
       hat;
   InterruptVektor[i] := 0 {ausschalten};
   Aufruf Kontrollprogramm mit Startadresse A[i]
   wiederherstellen alten Programmzustand und weiter
end;
```

2.4 Forderungen an einen BS-kern (OS Nucleus)

- Realisierung von Speicherschutz
- Echtzeituhr (für zeitabhängige Wartezustände, vgl. at-Kommando in UNIX)
- Unterbrechungsbehandler (interrupt handler)
- Prozeßumschalter (dispatcher)
- Unterscheidung User-Modus/Supervisor-Modus (letzterer auch System Modus genannt, immer mit priviligierten Instruktionen und im Systemadressraum ablaufend)

2.4.1 Interrupt handler (Exception handler)

Muß Unterbrechungsgrund feststellen und Unterbrechung bedienen (behandeln). Dazu

- retten Hardware-PCB, oft über eigenen Maschinenbefehl (SAVE, RESTORE) oder per Satz eigener Register (dann aber Vorsicht bei kaskadierenden Unterbrechungen)
- Sprung zur Unterbrechungsroutine (1. Instruktion der Routine) in Abhängigkeit vom Index des Bits im Unterbrechungsvektor,
  - ggf. proritätsgesteuert (Maskierung von Unterbrechungen niedrigerer Priorität), z.B. E/A-Unterbrechung unterbrechbar durch HW-Fehler-Unterbrechung, aber nicht umgekehrt;
  - Ausbildung von *Unterbrechungskaskaden*
  - Schutz der Unterbrechungsbehandlung an kritischen Stellen vor erneuter Unterbrechung, z.B. Übertragung des PCB mit mehreren Befehlen nicht unterbrechbar.
- Statusänderung bei unterbrochenem Programm veranlassen; nach Unterbrechungsbehandlung Statusänderung des unterbrochenen Programms
- Rückkehr von Unterbrechungsroutine, z.B. über spezielle Befehle (RTI - return from interrupt, Intel 80x86 IRET) und RESTORE-Befehl (Laden des geretteten HW-PCB)

- Ein-/Ausschalten von Unterbrechungsroutinen durch Löschen von Unterbrechungsadressen mittels ARM/DISARM („Bewaffnen/Entwaffnen“).

2.4.2 Dispatcher (low-level scheduler)

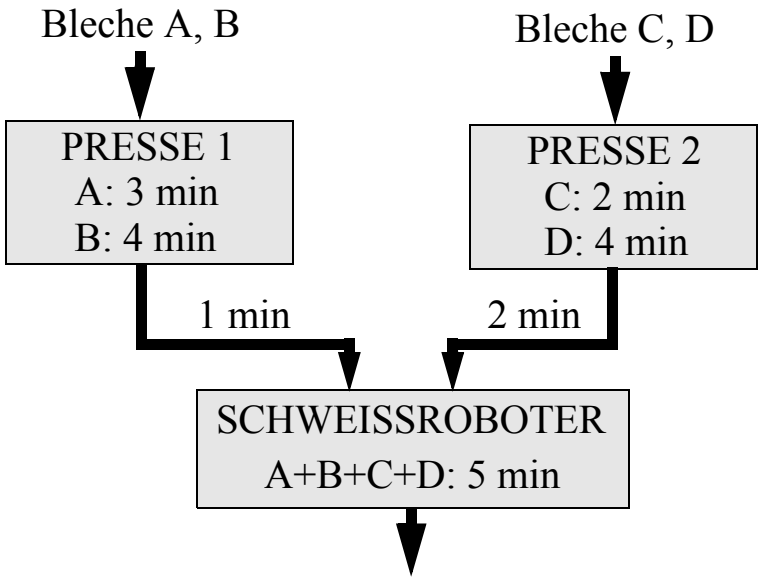
Bestimmt den geeigneten Prozeß, an den der Prozessor abgegeben werden soll und nimmt die Übergabe vor.

- wird angestoßen vom Interrupt-Handler
- reagiert auf extern vorgegebene Prozeßprioritäten (ggf. durch mehrfache Warteschlangen und Alterungsvorgang für Prozesse)
- kann auch direkt wieder zum angehaltenen Prozeß zurückkehren, z.B. bei polling über Uhr
- verwaltet die PCB-Liste
  - trägt Statusänderungen ein
  - hängt PCBs um in Liste(n) – gefährlich, wenn unterbrechbar!
  - hat dazu oft eigene Maschinenbefehle (QUEUE, DEQUEUE)
- Ablauf eines Null-Prozesses (dummy process) falls Warteschlange leer.

3 Parallele Prozesse

3.1 Asynchrone Abläufe

Betrachte man die untenstehende Fertigungsstraße für 4 Arten von Blechen mit unterschiedlichen Bearbeitungszeiten.



Wir vergleichen jetzt diese einfache „Fabrik“, die im Minutentakt gesteuert wird, mit einem Rechner, der mit Takten von 1 ns (10^{-9} s) bis zu - sagen wir - 1 µs (10^{-6} s) arbeitet. Schon bei der moderaten Taktrate von 1 MHz entspricht eine Sekunde Rechnerbetrieb der Ablaufsteuerung von 2 Jahren in der Fabrik (2 Jahre * 365 Tage * 24 h * 60 min = 1.051.200 Minuten).

Genausowenig wie man eine Fabrik „ohne Zwischenfälle“ und Abweichungen vom Takt für zwei Jahre synchron betreiben kann, so wenig wird man ein größeres Rechnersystem vollständig synchron (getaktet) arbeiten lassen können. Also wird man auch das BS auf ereignisgesteuerten, asynchronen Betrieb mit Wartezwängen und Synchronisierung auslegen.

Dies bedeutet für die Programmierung

- keine Annahmen über relative oder absolute Abarbeitungszeiten der Programmschritte, bzw. Gerätegeschwindigkeiten;
- keine Annahmen über den Zeitpunkt des Prozessorwechsels, d. h. beliebige (zufällige) Verschränkungen des Prozeßablaufs sind möglich.

Aber wir setzen voraus, daß

- ein Programmschritt (eine Maschineninstruktion) unteilbar ausgeführt wird.
  Vorsicht: gilt auf Maschinenebene, nicht für Konstrukte höherer Programmiersprachen, wie z. B. `i := i + 1;` oder für `if x = 0 then y := -1`

3.2 Wechselseitiger Ausschluß (mutual exclusion)

Die klassische Prozeßsynchronisierungsaufgabe ist der wechselseitige Ausschluß, etwa in der Benutzung einer Platte oder eines Übertragungskanals, der zu jedem Zeitpunkt von genau einem (oder manchmal von genau *k*) Teilnehmer(n) benutzt werden kann. Analogien aus dem Alltagsleben sind einspurige Brücken oder eingleisige Schienenabschnitte.

Programm 1

```
procedure P1;
begin
  repeat
    rechnen;
    wenn Platte frei
    dann belegen
    sonst warten;
    Platte benutzen;
    Platte freigeben;
  forever
end;
```

Programm 2

```
procedure P2;
begin
  repeat
    rechnen;
    wenn Platte frei
    dann belegen
    sonst warten;
    Platte benutzen;
    Platte freigeben;
  forever
end;
```

Rahmenprogramm

```
program RP;

  procedure P1;
  procedure P2;
begin
  Initialisierung;
  cobegin P1; P2 coend
end.
```

Abb. 3–1 *Pseudocode für wechselseitigen Ausschluß*

Daraus läßt sich die folgende Grundstruktur ableiten. Die nebenläufigen Prozesse sind Endlosschleifen mit

- einem unkritischen Rest (hier: rechnen)
- einem Vorprotokoll (Nutzungsberechtigung klären/prüfen)
- einem kritischen Abschnitt (critical section, hier: Platte benutzen)
- einem Nachprotokoll (Nutzungsrecht abgeben)

Im folgenden untersuchen wir die Konzepte *globale Variable* (*shared memory*), *Semaphore*, *Monitore*, *Botschaftenaustausch*.

3.3 Globale Variable

An eine Lösung der obigen Aufgabe stellen wir die folgenden Anforderungen (nach Dijkstra 1968 [16], S. 61):

- höchstens ein Prozeß darf sich im kritischen Abschnitt befinden
- die Entscheidung, welcher Prozeß zuerst seinen kritischen Abschnitt betritt, darf nicht auf ewig hinausgezögert werden;
- das Anhalten eines Prozesses außerhalb seines kritischen Abschnitts darf die anderen nicht beeinflussen.

Beurteilungskriterien

- **Safety**
  auch *partielle Korrektheit* genannt, z.B. korrekter wechselseitiger Ausschluß im kritischen Abschnitt; statische Eigenschaft
- **Liveliness**
  ergibt mit Safety die *totale Korrektheit*, z.B. daß jeder Prozeß irgendwann (nach endlicher Zeit) den kritischen Abschnitt betreten kann, sofern gewünscht (dynamische Eigenschaft). Zur Frage der Liveliness gehört
  - Deadlock/Livelock
    gegenseitiges Blockieren entweder mit Stillstand der Prozesse oder mit unproduktiven, nichtendenden Prüfschleifen

- Starvation oder Lockout
  einseitiges Blockieren, Aushungern anderer Prozesse
- Fairness

Ansatz 1 mit genau einer globalen Variablen

```
program firstAttempt;
var turn: integer;

procedure P1;
begin
  while (TRUE) do begin
    unkritischer Abschnitt;
    repeat
    until turn = 1;
    kritischer Abschnitt;
    turn := 2;
  end;
end;

procedure P2;
begin
  while (TRUE) do begin
    unkritischer Abschnitt;
    repeat
    until turn = 2;
    kritischer Abschnitt;
    turn := 1;
  end;
end;

begin (*Hauptprogramm*)
  turn := 2;
  cobegin P1; P2 coend
end.
```

Frage: Was passiert wenn ein Prozeß im unkritischen Abschnitt stirbt? Ist Safety gegeben? Wie sieht es mit Deadlock, Starvation, Fairness aus?

Ansatz 2 mit genau zwei globalen Variablen

```
program attempt2;
var c1, c2: integer;
```

```
procedure P1;
var i : integer;
begin
  while (true) do begin
    unkritischer Abschnitt;
    repeat
    until c2 = 1;
    c1 := 0;
    kritischer Abschnitt;
    c1 := 1;
  end;
end;

procedure P2;
var i : integer;
begin
  while (true)do begin
    unkritischer Abschnitt;
    repeat
    until c1 = 1;
    c2 := 0;
    kritischer Abschnitt;
    c2 := 1;
  end;
end;

begin (*Hauptprogramm*)
  c1 := 1; c2 := 1;
  cobegin P1; P2 coend
end.
```

Zum Testen des oberen Programms verwende man die folgende Ablauffolge mit Prozessorwechsel.

Zustand	C1	C2
Initialisierung	1	1
P1 prüft C2		
P2 prüft C1		
P1 setzt C1		
P2 setzt C2		
P1 benützt Platte		

Zustand	C1	C2
P2 benützt Platte		

Ansatz 3 wieder mit zwei Variablen

```
program attempt3;
var c1, c2: integer;

procedure P1;
begin
  while (TRUE)
  do begin
    unkritischer Abschnitt;
    c1 := 0;
    repeat
    until c2 = 1;
    kritischer Abschnitt;
    c1 := 1;
  end;
end;

procedure P2;
var i : integer;
begin
  while (true)
  do begin
    unkritischer Abschnitt;
    c2 := 0;
    repeat
    until c1 = 1;
    kritischer Abschnitt;
    c2 := 1;
  end;
end;

begin (*Hauptprogramm*)
  c1 := 1;
  c2 := 1;
  cobegin P1; P2 coend
end.
```

Testen Sie dieses Programm mit der folgenden Ablauffolge.

Zustand	C1	C2
Initialisierung	1	1
P1 setzt C1		
P2 setzt C2		
P1 prüft C2		
P2 prüft C1		

Ansatz 4 mit zwei Variablen

```
program attempt4;
var c1, c2: integer;

procedure P1;
var i : integer;
begin
  while (TRUE) do begin
    unkritischer Abschnitt;
    c1 := 0;
    repeat
      c1 := 1; (* etwas warten *)
      c1 := 0;
    until c2 = 1;
    kritischer Abschnitt;
    c1 := 1;
  end;
end;

procedure P2;
var i : integer;
begin
  while (TRUE) do begin
    unkritischer Abschnitt;
    c2 := 0;
    repeat
      c2 := 1; (* etwas warten *)
      c2 := 0;
    until c1 = 1;
    kritischer Abschnitt;
    c2 := 1;
```

```
  end;
end;

begin (*Hauptprogramm*)
  c1 := 1;
  c2 := 1;
  cobegin P1; P2 coend
end.
```

Zum Austesten verwende man den folgenden Ablauf.

Zustand	C1	C2
Initialisierung	1	1
P1 setzt C1	0	1
P2 setzt C2	0	0
P1 prüft C2	0	0
P2 prüft C1	0	0
P1 setzt C1	1	0
P2 setzt C2	1	1
P1 setzt C1	0	1
P2 setzt C2	0	0

Lösung nach Dekker (Dekker's Algorithm)

```
program dekker;
var c1, c2, turn: integer;

procedure P1;
var i : integer;
begin
  while (TRUE) do begin
    unkritischer Abschnitt;
    c1 := 0;
    repeat
      if turn = 2 then begin
        c1 := 1;
        repeat until turn = 1;
        c1 := 0;
      end;
```

```
      until c2 = 1;
      kritischer Abschnitt;
      c1 := 1;
      turn := 2;
    end;
end;

procedure P2;
var i : integer;
begin
  while (TRUE)do begin
      unkritischer Abschnitt;
      c2 := 0;
      repeat
        if turn = 1 then begin
          c2 := 1;
          repeat until turn = 2;
          c2 := 0;
        end;
      until c1 = 1;
      kritischer Abschnitt;
      c2 := 1;
      turn := 1;
  end;
end;

begin (*Hauptprogramm*)
  c1 := 1;
  c2 := 1;
  turn := 1;
  cobegin P1; P2 coend
end.
```

Beurteilung des Lösungsansatzes „Globale Variable“

- Verallgemeinerung auf mehr als zwei Prozesse schwierig.
- starke Abhängigkeit der Programme voneinander; getrennter Entwurf praktisch unmöglich
- ineffiziente Warteschleifen (*busy wait*)
- schreiben auf gleiche Speicherstelle (TURN), dadurch ggf. Verletzung von Speicherschutzmechanismen
- nicht anwendbar auf verteilte Speicher (Rechnernetze)

deshalb besser

- Warteschlange für inaktive Prozesse mit *Weckmechanismus*

Implementierungsgesichtspunkte

- *busy wait* z.B. durch Abfrage mit Sprung
  ```
  while TURN = 1 do;
  ```
  realisiert als
  ```
  JE TURN, 1, *
  ```
  mit JE = jump-on-equal, * = gegenwärtige Adresse
  aber Vorsicht, da u. U. nicht unterbrechbar!
- Ausschalten der Unterbrechungsmöglichkeit für Testen und Setzen von Variablen, z.B. Intel EI/DI (enable interrupt/disable ~)
  dann aber Verlust von Unterbrechungsmöglichkeiten und ggf. kein Prozessorwechsel mehr möglich
- unteilbare Operation „*Test-and-Set*“ mit lokaler Variablen *loc* und globaler Variablen *c*:
  TST(*loc*) $\Leftrightarrow$ *loc* := *c*; *c* := 1 (* unteilbar *)
  ggf. realisierbar über Bus-Sperre, d.h. zwischen Lesen und Schreibern eines Speicherworts kann kein anderer den Bus ergreifen.
- Unteilbarer Tausch zweier Registerinhalte:
  `swap r1, r2` oder `EX r1, r2`

Hinweis: (nach Martin C. Rinard, Operating Systems Lecture Notes, Lecture 5, Implementing Synchronization Operations;
`http://www.cag.lcs.mit.edu/~rinard/osnotes/h5.html`)

In einer modernen RISC-Architektur mit nur Load/Store-Instruktionen passen TST und SWAP schlecht. Daher baut man auf eine nichtblockierende Abstraktion: LoadLinked (LL) / StoreConditional (SC).

- *Semantik von LL*: Lade ein Speicherwort in ein Register und markiere es als für diesen Prozessor geladen. Beachte, daß eine Speicherstelle von mehreren Prozessoren als geladen markiert werden kann.

- *Semantik von SC:* Wenn die Hauptspeicherstelle als von diesem Prozessor geladen markiert ist, dann speichere den neuen Wert und entferne alle Markierungen von der Speicherzelle. Sonst führe das Speichern nicht aus. Liefere als Ergebnis (Statusbit) zurück, ob das Speichern erfolgreich war.

So wird LL/SC genutzt um eine Sperroperation zu implementieren.

```
while (1) {
  LL  r1, lock
  if (r1 == 0) {
    LI r2, 1
    if (SC r2, lock) break;
  }
}
```

Lösung des Problems „kritischer Abschnitt“ mittels TST

```
program TestandSet;
var c: integer;

procedure P1
var L: integer;
begin
  repeat
    unkritischer Teil;
    repeat TST(L) until L = 0;
    kritischer Abschnitt;
    c := 0
  forever
end;

procedure P2;
...{genau gleich} ...

begin (* Hauptprogramm *)
  c := 0;
  cobegin P1; P2 coend
end.
```

Verallgemeinerung des Problems „kritischer Abschnitt“ auf *n* Prozesse

Verschiedene Autoren haben Lösungen für dieses Problem gefunden. Wesentliches Qualitätsmerkmal dabei ist die Anzahl der Warteschritte bis zum Erlangen des Abschnitts.

Autor	Anzahl Warteschritte
Dijkstra (1965)	keine obere Schranke
Knuth (1966)	2^n Runden
de Bruijn (1967)	n^2
Eisenberg/McGuire (1972)	n - 1
Lamport (Bakery Algorithm, 1976)	n - 1

Eine Version des Eisenberg/McGuire-Verfahrens in C-Notation findet sich in `http://www.csee.wvu.edu/~jdm/classes/cs356/notes/mutex/Eisenberg.html`

Der Eisenberg and McGuire's Algorithmus in PASCAL

Diese korrekte Version des Problems benötigt die theoretisch minimale Anzahl von *n*-1 Warteschritten, die ein Prozeß zu warten hat, bis er drankommt.

Die Grundidee vereint die Variable `turn` und die Statusflags wie im Algorithmus von Dekker für den 2-Prozeß-Fall. Die Flags haben drei mögliche Werte: idle, want-in, in-cs (Lehrlauf, Anmeldung, Im-kritischen-Abschnitt). In der oben genannten Web-Seite heißen sie IDLE, WAITING, ACTIVE.

Die Prioritätsregelung kann man sich als kreisförmige Warteschlange der Prozesse vorstellen, beginnend mit dem Prozeß, der das „TURN-Token“ hält (dran ist).

Das Eingangsprotokoll eines Prozesses, der in den kritischen Abschnitt (kA) will, prüft die Prozesse von turn bis zu sich selbst (im Alg. durch

Index i bestimmt). Dies sind die Kandidaten, die potentiell vor ihm dran kommen.

Wenn die Prüfung ergibt, daß alle Prozesse davor „idle“ sind, versetzt sich der Prozeß vorläufig in den Zustand „in-cs“. Allerdings kann es passieren, daß ein Prozeß zeitlich nach uns die Prüfung beginnt, der aber gemäß Ordnung vor uns dran käme. Deshalb prüfen wir nochmals, ob es jetzt nicht doch noch einen aktiven („in-cs“) Prozeß gibt.

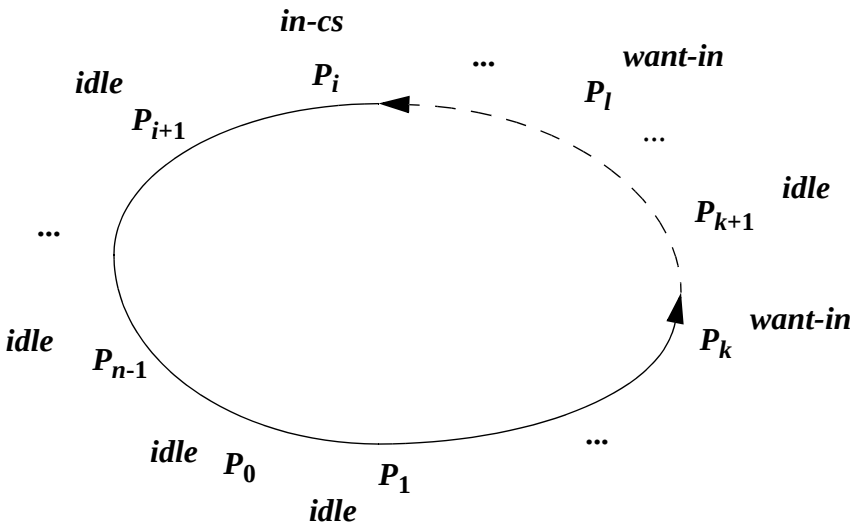


Abb. 3–2 *Vergabe des Eintrittsrechts*

```
program NProzesse;
var flag: array[0..n-1] of (idle, want-in, in-cs);
  turn: 0..n-1; {flag und turn als shared variable}

procedure P(i: integer);
  ...

begin {Hauptprogramm}
  for turn := 0 to n - 1 do flag[turn] := idle;
  turn := 0 {beliebig auf einen Prozess}
  cobegin
    P(1); P(2); ...; P(n-1); P(0)
  coend
end.
```

```
procedure P(i: integer);
var j: integer;
begin
  repeat {forever}
    repeat
      flag[i] := want-in; {Anmeldung ist erfolgt}
      j := turn;
      while j <> i do {prüfen ob alle idle bis zu uns}
        if flag[j] <> idle
        then j := turn {Vorgang wiederholen ab turn}
        else j := j + 1 mod n;
      flag[i] := in-cs; {vorläufig belegen, denn alle}
      j := 0; {vor uns waren idle. Jetzt nochmal prüfen}
      while (j < n) and (j = i or flag[j] <> in-cs) do
        j := j + 1; {hat noch jemand in-cs gesetzt?}
    until (j >= n) and ((turn = i)
      or (flag[turn] = idle));
      {kein anderer aktiv und wir haben turn
      oder turn ist idle, dann Einstieg}
    turn := i; {wir haben turn}

    kritischer Abschnitt

    j := turn + 1 mod n; {turn weitergeben an einen
      Wartenden, wenn keiner da, zurück zu uns selbst}
    while (flag[j] = idle) do j := j + 1 mod n;
    turn := j; {der nächste Prozess, ggf. wir selbst}
    flag[i] := idle; {erst jetzt in-cs zurückgesetzt}

    unkritischer Abschnitt

  until false;
end {P(i)};
```

Eine neue N-Prozeß Lösung: Leslie Lamport's Bakery Algorithmus

Der „Bäckerei-Algorithmus“ ist eine ganz andere Lösung und beruht auf dem Prinzip des „Nummerziehens“. Man beachte, daß man nicht verhindern kann, daß zwei Prozesse dieselbe Nummer ziehen. In diesem Fall entscheidet die Prozeßnummer, hier die Variable i, über den Vortritt.

```
program Bakery;
var choosing: array[0..n-1] of boolean;
    num: array[0..n-1] of integer; {beide shared}
    j: integer;

procedure P(i: integer);
var k : integer;
begin
  repeat {forever}
    choosing[i] := TRUE;  {eine Nummer ziehen}
    num[i] := max(num[0], ..., num[n-1]) + 1;
    choosing[i] := FALSE; {die größer ist als
                           alle anderen}
    for k := 0 to n-1 do begin {für alle Prozesse warte
                                ob dieser wählt}
      while choosing[k] do; {busy wait}
      {Nun prüfen, ob der Prozess Nummer hat
      und vor uns kommt}
      while (num[k] > 0) and ((num[k] < num[i]) or
      ((num[k] = num[i]) and (k < i))) do; {busy wait}
    end;

    kritischer Abschnitt

    num[i] := 0; {unsere Nummer löschen}

    unkritischer Teil

  until false;
end {P(i)};

begin {Hauptprogramm}
  for j := 0 to n-1 do begin
    num[j] := 0;
    choosing[j] := FALSE
  end;
  cobegin
    P(1); P(2); ...; P(n-1); P(0)
  coend
end.
```

Die FETCH-AND-ADD-Instruktion

In der Literatur wird seit Anfang der Achtziger Jahre (vgl. etwa Gottlieb and Kruskal (1981) [19]) diskutiert, wie man hardwaremäßig die Prozeßsynchronisierung analog zu Test-and-Set für $n > 2$ Prozesse unterstützen kann.

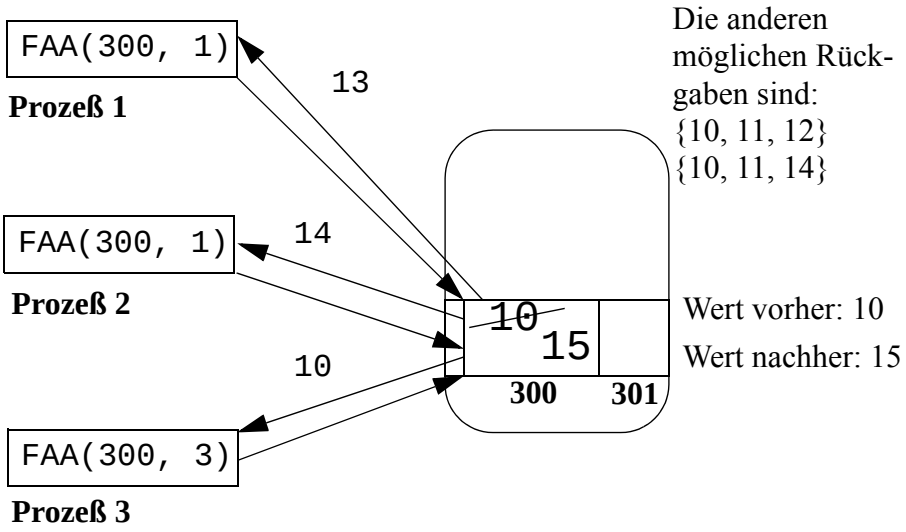
Das Resultat ist die Fetch-and-Add-Instruktion mit der folgenden Syntax und Semantik.

```
FetchAndAdd(Adresse, Inkrement);

begin
  temp := M[Adresse]; {holen Wert aus HS}
  M[Adresse] := temp + Inkrement; {neuer Wert in HS}
  return temp; {alten Wert zurückliefern, z.B. in
                Register}
end;
```

Führen *n* Prozesse gleichzeitig FetchAndAdd auf die gleiche HS-Adresse aus, so steht anschließend dort der ursprüngliche Wert + Summe der Inkremente, jeder Prozeß erhält aber als Resultat einen Wert, der einer beliebigen sequentiellen Abarbeitung entsprochen hätte. Auf den Web-Seiten zu Fetch-and-Add wird die Instruktion so dargestellt, daß auch ein negatives Increment (Decrement) zugelassen ist.

- **Vorteil**:
  Die *n* Prozesse erhalten einen Zeitstempel in praktisch einem Schritt (genauer $\log_2 n$ Schritte).
- **Nachteil**:
  Kaum implementiert, jedoch hat Cray dies für den parallelen Zugriff auf den Hauptspeicher unter Verwendung von sogenannten E-Registern eingesetzt.

Beispiel: Sei M[300] = 10

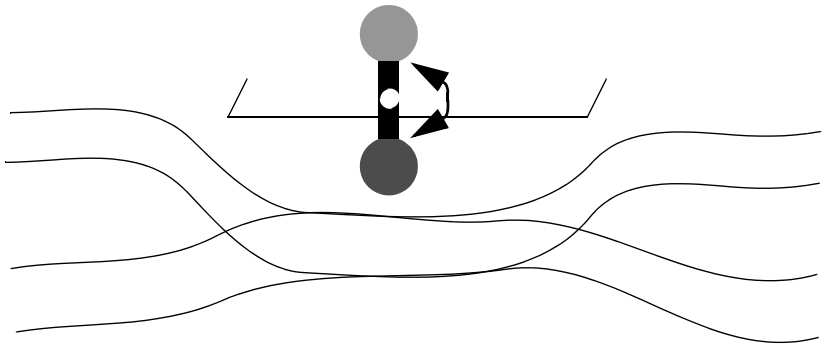


Im nächsten Kapitel wollen wir Semaphore und danach Monitore als höhere Prozeßsynchronisierungskonstrukte besprechen.

4 Semaphore

4.1 Einleitung

Der Begriff *Semaphor* ist der Eisenbahner- und Seefahrtssprache entlehnt und bezeichnet ein Signal, das den Zugriff auf einen kritischen Abschnitt, z.B. einen eingleisigen Abschnitt, oder eine enge Hafenzufahrt, regelt.



E. Dijkstra gilt als Erfinder des Semaphors im Bereich nebenläufige Prozesse [1968].

4.2 Definition und kritischer Abschnitt

Ein Semaphor besteht aus einer „Zähl- oder Bedingungsvariablen“, häufig *COND* genannt, und einer Warteschlange zur Aufnahme blockierter Prozesse. Die beiden elementaren Operationen auf einem Semaphor *s* sind *P*(*s*) und *V*(*s*).

- **P** steht für das holländische „passeren“, also durchgehen, passieren, bzw. „proberen“, probieren, testen, runterzählen.

- **V** steht für „vrygeven“, also freigeben, verlassen, oder „verhogen“, d.h. hochzählen.

Manche Implementierungen verwenden statt *P* die Operation *wait* und statt *V* die Operation *signal*. Wir wollen diese Begriffe hier vermeiden, da wir sie im Zusammenhang mit den Hoareschen Monitoren verwenden.

Grundsätzlich gilt, daß es je kritischen Abschnitt mindestens ein Semaphor (engl. semaphore) gibt.

Definition der P- und V-Operationen

```
procedure P(s: semaphore);
begin
  if s^.cond > 0
  then s^.cond := s^.cond - 1
  else „rufenden Prozeß in Warteschlange von s
       einreihen“
end;

procedure V(s: semaphore);
begin
  if „Warteschlange von s leer“
  then s^.cond := s^.cond + 1
  else „Prozeß in Warteschlange von s wecken“
end;
```

Hinweis: Man beachte, daß bei der *V*-Operation die Zählvariable im Fall einer nichtleeren Warteschlange unverändert bleibt!

Bemerkungen zum Semaphor:

- Wir sprechen von einem *binären Semaphor*, wenn *COND* nur die Werte 0 und 1 (bzw. TRUE und FALSE) annehmen kann.
- Das Semaphor ist ein gekapseltes Objekt, auf das neben einer Initialisierung nur die Operationen (Methoden) *P* und *V* anwendbar sind; insbesondere ist keine direkte Manipulation von *COND* erlaubt.
- *P* und *V* sind unteilbare Operationen
- Die Reihenfolge der Freigabe bei einer *V*-Operation ist nicht festgelegt; FIFO ist aber vernünftig und implementierbar.

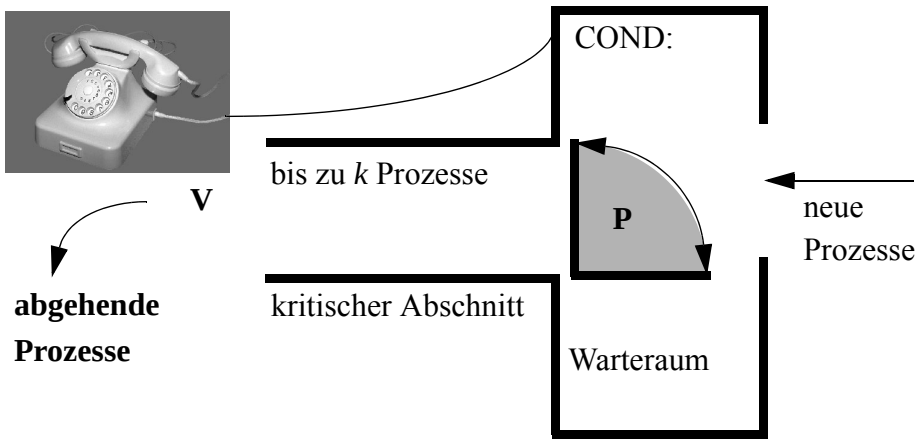
- Warten *k* Prozesse in einem Semaphor *s*, so muß ein *V*(*s*) nicht unbedingt die Freigabe eines dieser Prozesse bedeuten, wenn der Prozeß der *V*(*s*) macht, sofort wieder *P*(*s*) setzt; daher definiert man ein sog. *strongly fair semaphore* mit der Garantie, daß ein wartender Prozeß sicher freikommt, sofern *COND* nur unendlich oft > 0 wird.

Betrachten wir als erstes den einfachen wechselseitigen Ausschluß (engl. mutual exclusion):

```
program Mutex-semaphore;

var s: {binary} semaphore;
procedure P1;
begin
  repeat
    P(S) {wait};
    kritischer Abschnitt
    V(S) {signal};
    unkritischer Abschnitt
  forever
end;

procedure P1;
begin
  repeat
    P(S) {wait};
    kritischer Abschnitt
    V(S) {signal};
    unkritischer Abschnitt
  forever
end;

begin
  semainit(S, 1);
  cobegin P1; P2 coend
end.
```

Man überlege sich, daß in dieser Lösung ein „Prozeßabsturz/~abbruch“ im unkritischen Abschnitt harmlos ist und den Partnerprozeß nicht blokkiert. Genauso kann ein „schneller“ Prozeß mehrere Durchläufe durch den kritischen Abschnitt haben, bevor der langsamere wieder drankommt.

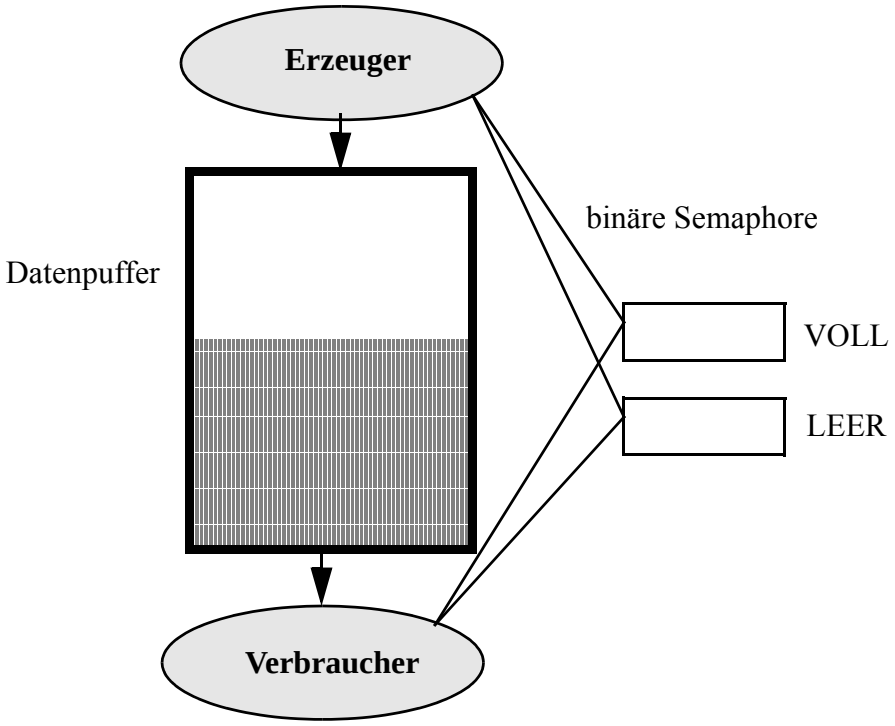
Die Verallgemeinerung auf *n* Prozesse mit genau einem Prozeß im kritischen Abschnitt ist trivial. Genauso kann man leicht eine Lösung programmieren, bei der bis zu *k* Prozesse gleichzeitig in einem kritischen Abschnitt sein dürfen, etwa ein Kanal, der bis zu *k* Teilnehmer multiplexen kann. Hier muß man einfach nur die Zählvariable eines zählenden (nichtbinären) Semaphors mit dem Wert *k* initialisieren.



Wir besprechen jetzt einige wichtige Anwendungsbeispiele.

4.3 Erzeuger-Verbraucher-Systeme

Häufig sind Rechner und Terminal mit einem „type ahead buffer“ verbunden, also einem Puffer mit wechselseitigem Schreiben und Lesen. Ferner muß geregelt werden, daß Prozesse nicht aus einem leeren Puffer lesen und nicht in einen vollen Puffer schreiben.



Das folgende Programmbeispiel setzt voraus, daß der gesamte Dateninhalt der Puffers gelesen oder geschrieben wird.

```
program Producer-Consumer;
var VOLL, LEER: {binäre} Semaphore;

procedure Producer;
begin
  repeat
    Daten erzeugen - unkritisch
    P(LEER); {gehe durch wenn leer}
    füllen Datenpuffer - kritischer Abschnitt
    V(VOLL) {signalisiere, daß Puffer voll}
```

```
  forever
end;

procedure Consumer;
begin
  repeat
    P(VOLL); {gehe durch wenn Puffer voll}
    Entleeren Datenpuffer - kritischer Abschnitt
    V(LEER) {signalisiere, daß Puffer jetzt leer}
    Daten verbrauchen - unkritisch
  forever
end;

begin
  semainit(VOLL, 0); semainit(LEER, 1);
  cobegin Producer; Consumer coend;
end.
```

Anders als bei den Beispielen vorher legen wir Wert darauf, daß die Prozesse nicht beliebig vorauseilen können (siehe aber *unbounded buffer* unten), sondern daß die Prozesse wechselseitig zum Füllen und Leeren aktiviert werden. Die verwendeten Semaphore heißen private Semaphore der beiden Prozesse *Erzeuger* und *Verbraucher*. Charakteristisch ist die Verschränkung der *P*- und *V*-Operationen in den beiden Prozessen.

4.4 Unbegrenzter Puffer

Das folgende Beispiel ist für einen theoretisch unbegrenzten Puffer, also etwa einen Ringpuffer, in dem nur die *n* neuesten Daten relevant sind.

```
program Unbegrenzter-Puffer;
var N {allgemeines}, S {binäres}: Semaphore;

procedure Producer;
begin
  repeat
    Daten erzeugen - unkritisch
    P(S); {gehe durch wenn Abschnitt nicht belegt}
    füllen Datenpuffer - kritischer Abschnitt
    V(S); {kritischen Abschnitt freigeben}
    V(N) {Inhaltszähler hochsetzen}
  forever
end;
```

```
procedure Consumer;
begin
  repeat
    P(N); {gehe durch wenn Inhalt da (Zähler > 0)}
    P(S); {gehe in Puffer wenn frei}
    Entleeren Datenpuffer - kritischer Abschnitt
    V(S); {kritischen Abschnitt freigeben}
    Daten verbrauchen - unkritisch
  forever
end;

begin
  semainit(N, 0); semainit(S, 1);
  cobegin Producer; Consumer coend;
end.
```

Hinweis: Diese Lösung funktioniert auch für mehrere Erzeugen und Verbraucher.

Übung 4–1

a) Man untersuche die Vertauschung von V(S); V(N) im Produzenten.

b) Man untersuche die Vertauschung von P(N); P(S) im Konsumenten.

Ein Nachteil der obigen Lösung ist, daß Prozesse keine „Inspektion“ der Puffer (Variablen N) vornehmen können, ohne bei $N = 0$ gefangen zu werden. Man will ja unter Umständen, daß der Konsument asynchron weiterarbeitet, etwa um eine Eingabeaufforderung abzusetzen.

Ferner wäre die ausschließliche Verwendung binärer Semaphore besser, da diese oft durch die Hardware unterstützt werden, die zwar genau ein Bit atomar testen und setzen kann, nicht aber eine ganzes Wort (Integer).

Zuletzt enthält die Lösung recht viele *P*- und *V*-Operationen, die immer temporär (ca. 10 Maschineninstruktionen je Operation) die Unterbrechungsmöglichkeit ausschalten, was generell gefährlich ist.

Wir betrachten daher zwei weitere Lösungen: ProducerConsumer-A und ~-B. Die Aufgabe wird auch als *sleeping barber-Problem* bezeichnet.

```
program ProducerConsumer-A;
var N: integer;
    S, VERZOEGERUNG {binäre}: Semaphore;

procedure Producer;
begin
  repeat
    Daten erzeugen - unkritisch
    P(S); {gehe durch wenn Abschnitt nicht belegt}
    füllen Datenpuffer - kritischer Abschnitt
    N := N + 1; {ein Kunde mehr}
    if N = 1 {Puffer vorher leer} then V(VERZOEGERUNG);
    V(S) {kritischen Abschnitt freigeben}
  forever
end;

procedure Consumer; {Frisör}
var m: integer; {lokale Hilfsvariable}
begin
  P(VERZOEGERUNG);
  repeat
    P(S); {kritischer Abschnitt (=Wartezimmer) frei}
    Entleeren Datenpuffer - kritischer Abschnitt
    N := N - 1; {ein weiterer Kunde bedient}
    m := N;
    V(S); {kritischen Abschnitt freigeben}
    Daten verbrauchen - unkritisch
    if m= 0 then P(VERZOEGERUNG){wenn vorher letzter
                                 Kunde}
  forever
end;

begin
  N := 0;
  semainit(S, 1); {Zugang kritischer Abschnitt frei}
  semainit(VERZOEGERUNG, 0); {kein Kunde im Wartezimmer}
  cobegin Producer; Consumer coend;
end.
```

Das Problem des schlafenden Frisörs (*Sleeping-Barber-Problem* ist ein Erzeuger-Verbraucher-Problem, wobei der Frisör der Konsument ist und die Frisörkunden von einem fiktiven Produzenten (der Kundschaft) erzeugt werden. Der kritische Abschnitt ist der Blick in den Warteraum (siehe Abb. 4–1).

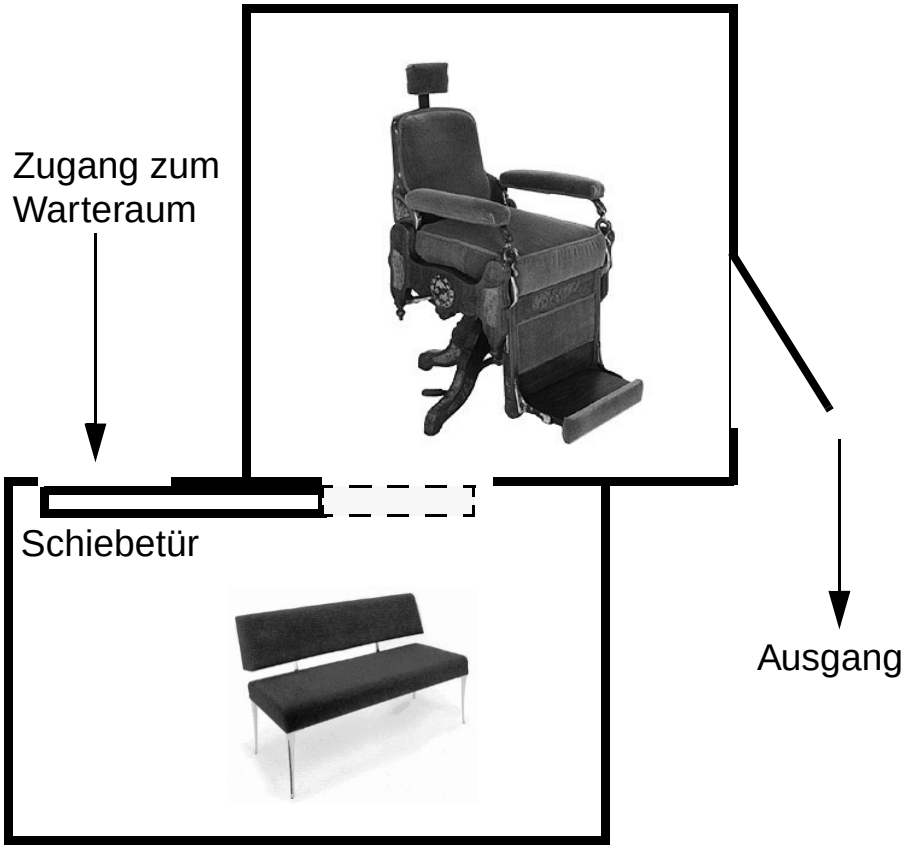


Abb. 4–1 *Sleeping Barber*

In der **Lösung A** prüft der Frisör nach Ende der Bedienung eines Kunden den Warteraum. Findet er einen Kunden, führt er ihn zum Stuhl, sonst schläft er im **Stuhl**.

Kunden betreten den Warteraum und warten dort, wenn noch andere Kunden da sind. Ist das Wartezimmer leer, klopft ein Kunde an und prüft den Behandlungssalon. Ist noch ein Kunde auf dem Stuhl, wartet der neue Kunde im Wartezimmer, sonst weckt er den Frisör.

Diese Lösung enthält einen unnötigen Test und ggf. Kontextwechsel im Fall, daß der Warteraum leer ist. Die folgende Lösung B vermeidet das, indem der Frisör bei leerem Warteraum auf der Bank im Warteraum schläft.

Ein eintretender Kunde weiß dann, daß bei leerer Bank der Frisör noch einen Kunden hat, er kann also warten, bis er geholt wird. Sonst sieht er den Frisör auf der Bank schlafen und weckt ihn.

Der wesentliche Unterschied ist dabei die Variable N, die auch den Wert -1 annehmen kann (= Frisör schläft auf der Bank). Null bedeutet „Bank leer, Frisör bedient gerade“, >0 ist die Anzahl der wartenden Kunden.

```
program ProducerConsumer-B;
var N: integer;
    S, VERZOEGERUNG {binäre}: Semaphore;

procedure Producer;
begin
  repeat
    Daten erzeugen - unkritisch
    P(S); {gehe durch wenn Abschnitt nicht belegt}
    Füllen Datenpuffer - kritischer Abschnitt
    N := N + 1; {ein Kunde mehr}
    if N = 0 {vorher -1, Frisör schlief}
    then V(VERZOEGERUNG);
    V(S) {kritischen Abschnitt freigeben}
  forever
end;

procedure Consumer; {Frisör}
begin
  repeat
    P(S); {kritischer Abschnitt (=Wartezimmer) frei}
    N := N - 1;
    if N = -1 then {war vorher 0, also leeres
                    Wartezimmer}
    begin V(S); {Zimmertür freigeben}
      P(VERZOEGERUNG); {auf Bank schlafen}
      P(S) {Zimmertür zu}
    end;
    Entleeren Datenpuffer - kritischer Abschnitt
    V(S) {kritischen Abschnitt freigeben}
    Daten verbrauchen - unkritisch
```

```
  forever
end;

begin
  N := 0;
  semainit(S, 1); {Zugang kritischer Abschnitt frei}
  semainit(VERZOEGERUNG, 0); {kein Kunde im Wartezimmer}
  cobegin Producer; Consumer coend;
end.
```

Semaphore wurden schon sehr früh in verschiedene Sprachsysteme eingebettet. Eine sehr einfache und übersichtliche gibt es im alten UCSD Pascal. Die Operationen dort sind SEMINIT, WAIT, SIGNAL und ATTACH. Letzte Operation bindet ein Semaphor an einen externen Interrupt. Wird dieser ausgelöst, wird das Semaphore signalisiert (V-Op). Ein Prozeß kann demnach auf einen Interrupt warten und sich damit synchronisieren.

Zur Implementierung unter UNIX siehe Abschnitt 4.7 unten.

4.5 Die Readers-Writers-Probleme

Eine weitere bekannte Klasse von Synchronisierungaufgaben werden als Readers-Writers-Probleme bezeichnet. Dabei werden Datenpuffer geschrieben und gelesen (aber nicht im Sinne von Verbrauch). Schreiben muß immer exklusiv erfolgen, d.h. es sind keine anderen Schreiber oder Leser erlaubt. Lesen kann aber parallel erfolgen, d.h. mehrere Leser stören sich nicht.

Es werden jetzt zwei Lösungen vorgestellt, wobei zuerst die Readers bevorzugt werden. In einer zweiten Lösung dann die schreibenden Prozesse.

Erste Lösung (Readers bevorzugt)

Einem neuen Reader wird sofort der Zugang gewährt, wenn andere Reader bereits lesen, unabhängig von eventuell wartenden Writers. Dabei werden verwendet:

- Variable READCOUNT: zählt die momentanen Leser

- Semaphor MUTEX: synchronisiert die Zugriffe der Leser auf READCOUNT
- Semaphor W: synchronisiert den Zugriff der Leser und Schreiber auf den Datenbereich (Schreiber und Leser warten dort)

```
program ReadersWriters1;
var READCOUNT: integer;
    W, MUTEX{binäre}: Semaphore;

procedure READER(i: integer);
begin
  repeat
    P(MUTEX); {Zugriff auf READCOUNT wenn nicht belegt}
    READCOUNT := READCOUNT + 1; {ein Leser mehr}
    if READCOUNT = 1 then P(W); {erster Leser}
    V(MUTEX);
    Lesen Daten - kritischer Abschnitt
    P(MUTEX);
    READCOUNT := READCOUNT - 1;
    if READCOUNT = 0 {letzter Leser} then V(W);
    V(MUTEX); {READCOUNT freigeben}
    Daten verarbeiten - unkritisch
  forever
end;

procedure Writer(i: integer);
begin
  repeat
    Erzeugen Daten - unkritisch;
    P(W); {kritischer Abschnitt Datenbereich frei?}
    Schreiben in Datenpuffer - kritischer Abschnitt
    V(W) {kritischen Abschnitt freigeben}
  forever
end;

begin
  READCOUNT := 0;
  semainit(W, 1); {Zugang kritischer Abschnitt frei}
  semainit(MUTEX, 1); {Zugriff auf READCOUNT frei}
  cobegin
    READER(1); READER(2); ...; READER(n);
    WRITER(1); WRITER(2); ...; WRITER(m)
  coend;
end.
```

Zweite Lösung (Writers bevorzugt)

Ein Writer, der Zutritt zum kritischen Abschnitt verlangt, wird zum frühestmöglichen Zeitpunkt zugelassen. Ein Reader kommt nur zum Zuge, wenn kein Writer mehr auf Zutritt wartet.

Verwendet werden:

- Zähler READCOUNT, WRITECOUNT
- Semaphore MUTEX1, MUTEX2, MUTEX3, W, R
- READCOUNT und W wie in Lösung 1. MUTEX1 wie MUTEX in Lösung 1.
- MUTEX2 synchronisiert Zugriffe auf WRITECOUNT
- MUTEX3: alle Reader bewerben sich nacheinander um den Eintritt in den kritischen Abschnitt
- R realisiert Priorität der Writer über die Reader indem das Zugriffsrecht hier von Writer zu Writer weitergereicht wird.

Hinweis: Durch MUTEX3 wird gesichert, daß ein Writer höchstens einem Reader den Vortritt lassen muß.

```
program ReadersWriters2;
var READCOUNT, WRITECOUNT: integer;
    W, R, MUTEX1, MUTEX2, MUTEX3 {binäre}: Semaphore;

procedure READER(i: integer);
begin
  repeat
    P(MUTEX3); {Zugriff auf READCOUNT wenn nicht belegt}
      P(R);
        P(MUTEX1);
          READCOUNT := READCOUNT + 1; {ein Leser mehr}
          if READCOUNT = 1 then P(W); {erster Leser}
        V(MUTEX1);
      V(R)
    V(MUTEX3);
    lesen Daten
    P(MUTEX1);
      READCOUNT := READCOUNT - 1;
      if READCOUNT = 0 {letzter Leser} then V(W);
    V(MUTEX1) {READCOUNT freigeben}
    Daten verarbeiten - unkritisch
```

```
  forever
end;

procedure Writer(i: integer);
begin
  repeat
    Erzeugen Daten - unkritisch
    P(MUTEX2) {Schutz von WRITECOUNT}
      WRITECOUNT := WRITECOUNT + 1;
      if WRITECOUNT = 1 then P(R);
    V(MUTEX2);
    P(W); {Datenbereich frei?}
      Schreiben in Datenpuffer
    V(W); {Datenbereich freigeben}
    P(MUTEX2) {Schutz von WRITECOUNT}
      WRITECOUNT := WRITECOUNT - 1;
      if WRITECOUNT = 0 then V(R);
    V(MUTEX2);
  forever
end;

begin
  READCOUNT := 0; WRITECOUNT := 0;
  semainit(W, 1); semainit(R, 1) {alle Zugänge frei}
  semainit(MUTEXi, 1);
  cobegin
    READER(1); READER(2); ...; READER(n);
    WRITER(1); WRITER(2); ...; WRITER(m)
  coend;
end.
```

Den „Instanzenweg“ für Reader und Writer bei Lösung 2 zeigt die folgende Abbildung.

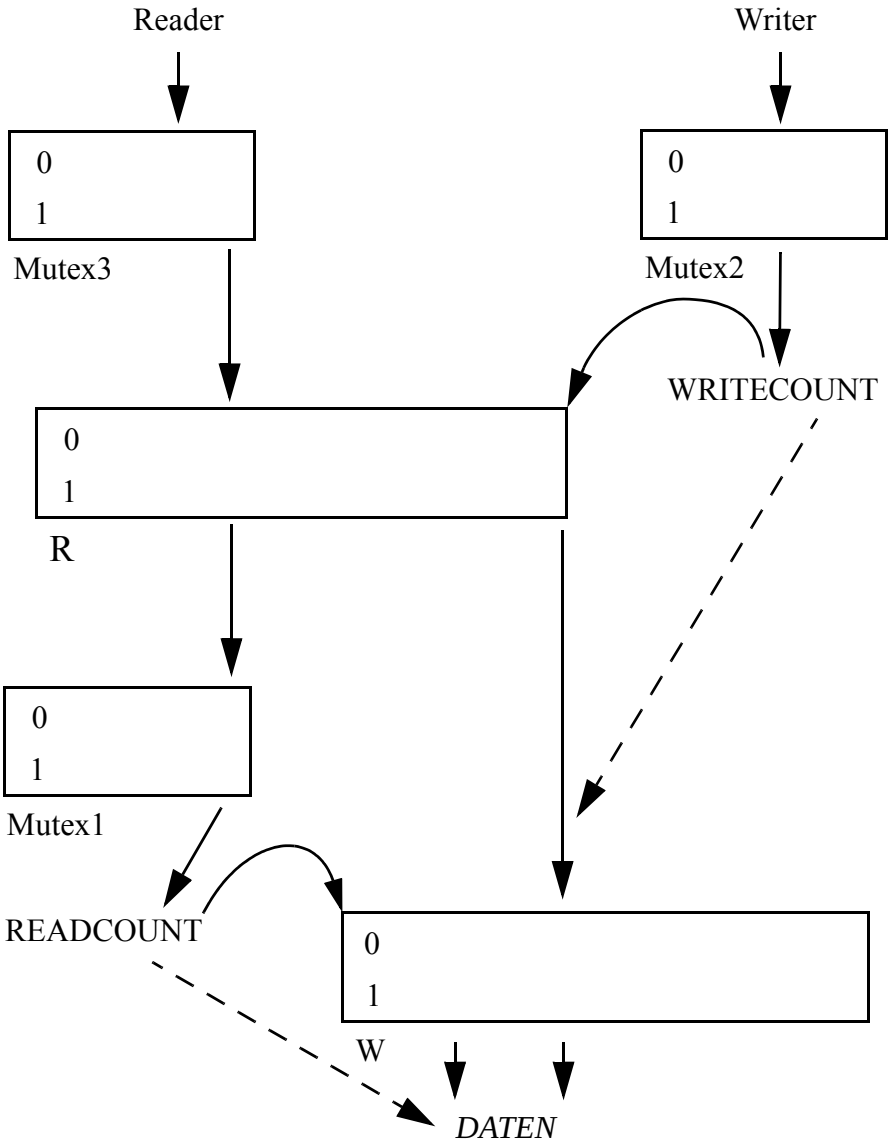


Abb. 4–2 *Zutrittspfad für Readers-Writers Lösung 2*

Zwei mögliche Zutrittshistorien bei gleicher Ankunftsfolge zeigt das BIld unten (gestrichelte Linien bezeichnen Wartezeiten, das graue Kästen ist der Datenbereich).

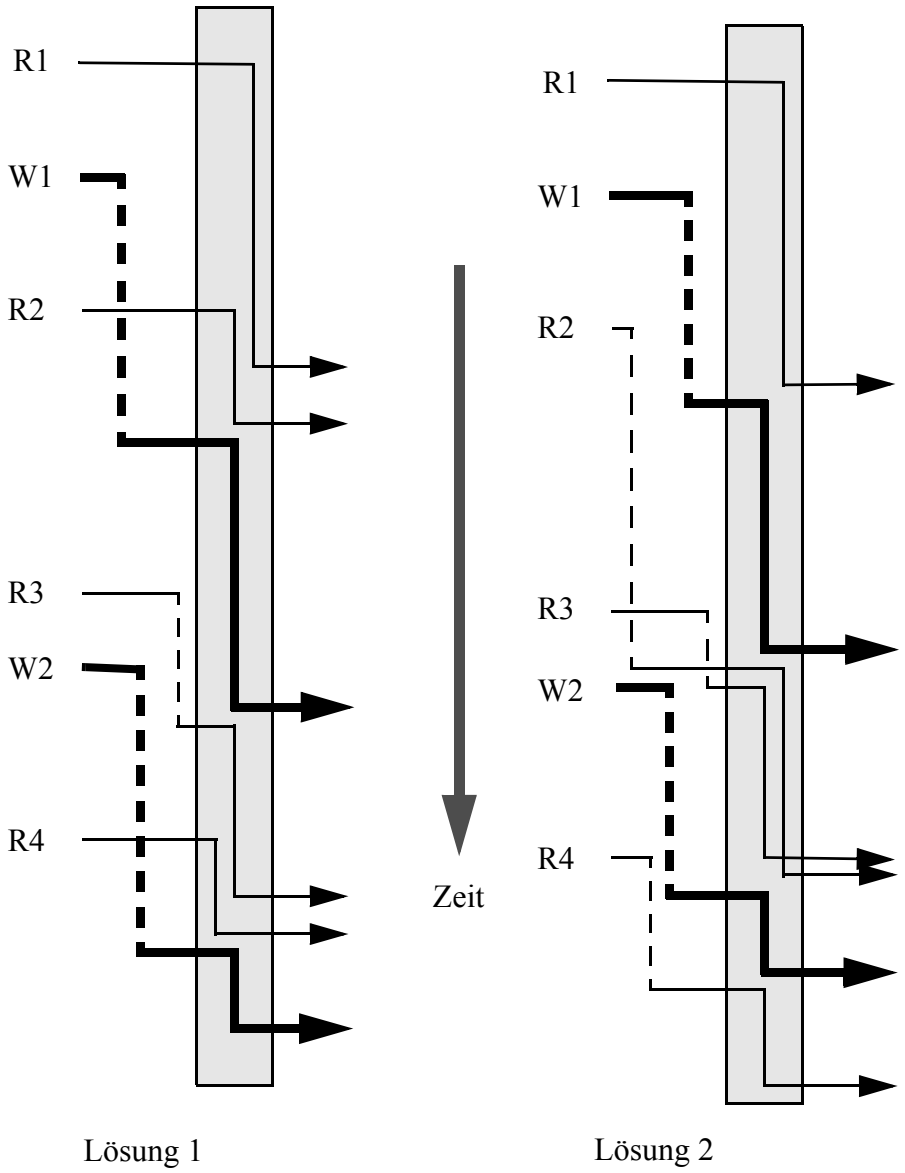


Abb. 4–3 *Zugriffshistorie bei unterschiedlicher Zutrittsbevorzugung*

4.6 Betriebsmittelverwaltung (Vier-Bänder-Problem)

Vier Bandeinheiten werden mit zwei Methoden verwaltet: GETTAPE weist einem Kunden ein Bandgerät zu, so eines frei ist. PUTTAPE gibt das Bandgerät zurück.

```
program VierBänder;
var TAPE: array[1..4] of (frei, belegt); k: integer;
    TAPSEM, MUTEX: semaphore;

procedure GETTAPE(var ACTTAPE: integer); {reentrant}
var i: integer;
begin
  P(TAPSEM);
  P(MUTEX);
  i := 1;
  while TAPE[i] = belegt do i := i + 1;
  TAPE[i] := belegt;
  ACTTAPE := i;
  V(MUTEX)
end;

procedure PUTTAPE(ACTTAPE: integer); {reentrant}
begin
  P(MUTEX);
  TAPE[ACTTAPE] := frei;
  V(MUTEX);
  V(TAPSEM)
end;

procedure TAPEUSER(i: integer); {Anwender}
var ACTTAPE: integer;
begin
  repeat
    ...
    GETTAPE(ACTTAPE);
    benutzen Bandeinheit ACTTAPE
    PUTTAPE(ACTTAPE);
    ...
  forever
end;

begin
  for k := 1 to 4 do TAPE[k] := frei;
  semainit(TAPSEM, 4); semainit(MUTEX, 1);
  cobegin
    TAPEUSER(1); TAPEUSER(2); ...; TAPEUSER(n)
```

```
    coend
end.
```

Wie man sieht regelt TAPSEM die Vergabe und blockiert Anwender, wenn alle 4 Bänder belegt sind. Dazu ist TAPSEM ein zählendes Semaphor. MUTEX schützt die kritische Prüfung des Arrays, weil durch Prozessorwechsel eine inkonsistente Situation beim Test auf „frei/belegt“ entstehen könnte.

Etwas naiv ist die Rückgabe per Nummer ohne weiteren Test. Ein böswilliger Anwender könnte ein Band zurückgeben, das im garnicht gehört. In der Praxis wird man eine Vergabetabelle im Systembereich unterhalten und auch den Fall eines „sterbenden Prozesses“ während der Belegung betrachten müssen.

Aus historischen Gründen steht an den Prozeduren GETTAPE/PUTTAPE noch die Bemerkung „reentrant“. Damit bezeichnet man Programme, deren Textsegment (Programmcode) und Konstanten in mehrfachen Inkarnationen (Instanziierungen, Exemplaren) existieren können. Jeder Prozeß, der mit diesem Programm läuft, benutzt natürlich sein eigenes Datenteil, also Heap und Stapelsegment.

Man sagt anschaulich, Prozesse die auf einem „reentrant-geschriebenen“ Programm beruhen, seien wie Köche, die alle anch dem selben Rezept kochen, aber jeder mit seinem eigenen Kochtopf.

Speziell kann ein Programm, dessen Code reentrant ist, keine selbstmodifizierenden Instruktionen enthalten. Das ist aber ein sehr altmodischer Programmstil, der sich heute im Zeitalter virtueller Adressräume und des Speicherschutzes sowieso verbietet. Bei selbstmodifizierenden Instruktionen wurde z. B. der Rücksprung aus einer Unterroutine ins aufrufende Programm so geregelt, daß in der relativen Adresse 0 des Unterprogramms ein Wort reserviert war zur Aufname eines unbedingten Sprungs (JUMP *Adr*). Als Adresse *Adr* in diesem Befehl wird dann „Absprungadresse + 1“ eingetragen und anschließend in Adresse 1 der Unterroutine verzeigt. Die Unterroutine enthält immer als letzte Instruktion einen Sprung auf die relative Adresse 0 der Unterroutine. Wie man leicht ein-

sieht, versagt diese Methode z. B. beim rekursiven Aufruf der Unterroutine und ist auch sonst Programmierstil aus dem finstersten Mittelalter.

4.7 Semaphore unter UNIX

Die Implementierungssprache von UNIX ist und war in den Sechzigern bekanntlich C. Die Sprache C kennt aber keine Prozeßsynchronisierungsprimitiva! Deshalb wurde UNIX ursprünglich mit „geheimen“ Synchronisierungsmethoden, nämlich unteilbaren Systemaufrufen wie etwa `creat()` zum Anlegen einer Datei, auf korrektes Verhalten bei Nebenläufigkeit programmiert.

Erst sehr viel später wurden Methoden zur Synchronisierung als Teil des *Interprozeßkommunikationspakets* (IPC) hinzugefügt. Dafür dann wieder gleich mehrere Methoden, darunter auch Semaphore, und der Zugriff auf *shared memory.*

Wir besprechen hier nicht die recht undurchsichtige Implementierung. Auch in der Vorlesung Interprozeßkommunikation (Rechnernetze) wird sie nur gestreift. Einen generellen Überblick geben [Rochkind] (S. 185-192) und Maurice Bach [1] (S. 370-383), gründlicher behandeln [Gray] und [Stevens] das Thema. Interessante Details findet man üblicherweise in den Header-Dateien, z. B. in `/usr/include/sys/sem.h`

Generell gibt es drei Systemaufrufe für Semaphore unter UNIX:

`semget`	`anlegen und initialisieren einer` *`Menge`* `von Semaphoren`
`semctl`	`kontrollieren und „befragen“ einer Semaphormenge`
`semop`	`manipuliert Semaphorwerte, also wie P und V`

Die Idee einer Semaphormenge soll die Verwaltung von Betriebsmittelgruppen erleichtern. Dazu werden in den Semaphor-Tabellen Zeiger auf Semaphor-Vektoren angelegt. Die atomaren Operationen arbeiten dann auf *allen* Semaphoren der Menge und sollen damit Verklemmungsvermeidung erleichtern.

Ferner gibt es eine Tabelle mit einem Eintrag (Undo-Header) für jeden Prozeß, der Semaphore verwendet. Stirbt der Prozeß vorzeitig, enthalten die Undo-Header einen Verweis auf eine Kette von Undo-Strukturen, mit denen belegte Ressourcen wieder freigegeben werden können.

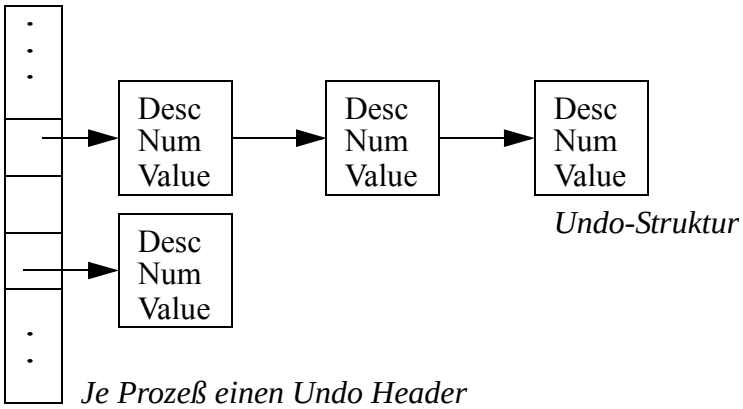


Abb. 4–4 *Undo-Strukturen für Semaphore nach [1]*

Die drei oben genannten UNIX-Funktionen für Semaphore sind wie folgt definiert.

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

/************************************************/
int semget(key, nsems, flags)
    key_t key;      /* semaphore set key */
    int nsems;      /* number of sems */
    int flags;      /* option flags */
    /* semget returns sem_id or -1 on error */

/************************************************/
int semop(sid, ops, nops)
    int sid;                /* semaphore set id */
    struct sembuf (*ops)[]; /* ptr to ops. */
    int nops;               /* number of operations */
    /* semop returns semaphore value prior to last
           operation or -1 on error */
```

```
/************************************************/
int semctl(sid, snum, cmd, arg)
    int sid, snum, cmd;
    char *arg;
    /* semctl returns val or -1 */
```

Die folgende besser lesbare Verwendung von Semaphores stammt ursprünglich aus [Rochkind]. Die Version hier wurde entnommen und übersetzt aus `http://www-cs.canisius.edu/PL_TUTORIALS/C/ADVANCED/ipc`

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

Zunächst besorgt man sich ein Semaphor. Dazu übergibt man dem Aufruf `semtran()` einen eigenen Schlüssel (kleiner Integer) und erhält vom Betriebssystem einen systemweit-eindeutigen Identifier *sid* zurück, mit dem man dann im folgenden arbeitet.

```
int semtran(key) /* translate semaphore key to an ID */
int key;
{
  int sid;

  if ((sid=semget((key_t)key,1,0666|IPC_CREAT)) == -1) {
     printf ("couldn't get semaphore\n");
     exit (-1);
  }
  return (sid);
}
```

So wird im Beispiel das Semaphor benutzt (5 ist unsere Zahl).

```
int sem1 = semtran(5); /* 5 is our number */
```

Jetzt können wir P- und V-Operationen auf das Semaphor loslassen. Zunächst bauen wir uns eine kleine Hilfsfunktion `semcall` als Hülle um `semop` herum.

```
static void semcall (sid, op) /* call semop */
int sid;
int op;
{
  struct sembuf sb;
```

```
  sb.sem_num = 0;
  sb.sem_op = op;
  sb.sem_flg = 0;
  if (semop(sid, &sb, 1) == -1) {
    printf ("Error in semop\n");
    exit (-1);
  }
}

void P(sid) /* acquire semaphore */
int sid;
{
  semcall (sid, -1);
}

void V(sid) /* release semaphore */
int sid;
{
  semcall (sid, 1);
}
```

Der systemweite Semaphore-Identifier wird in den P- und V-Aufrufen verwendet. Das Programm kehrt erst auf dem P-Aufruf zurück, wenn das Semaphor frei wird (P ging durch oder Prozeß wurde geweckt). Unter UNIX kann man das über das ps-Kommando überprüfen (Prozeß wartet). Nach einem V-Aufruf kehrt der Prozeß immer sofort zurück. Ferner muß das Semaphor mit einem V-Aufruf initialisiert werden (Standardinitialisierung scheint 0 zu sein).

Es folgt ein Beispielprogramm zur Programmierung eines kritischen Abschnitts mit den obigen Funktionshüllen.

```
main()
{
  int sid1;

  sid1 = semtran(5);
  V(sid1); /* initialize the semaphore */

  P(sid1);
  printf ("Got semaphore\n");
  printf ("This is the critical section\n");
  V(sid1);
  printf ("Released semaphore\n");
}
```

Stevens bemerkt zu Recht, daß die sog. *Record-Locks* einfacher und fast genauso schnell wie Semaphore unter UNIX sind.

Daher beenden wir die Behandlung hier und gehen zu den Hoareschen Monitoren im nächsten Kapitel über.

5 Monitore

5.1 Semaphor versus Monitor

Die Nachteile des Semaphor-Ansatzes sind

- Kooperation und Gutwilligkeit der beteiligten Prozesse wird vorausgesetzt (in P_i: P(MUTEX), in P_j: V(MUTEX).
- Schachtelung von Semaphoren bedeutet großer Overhead.

Daher gab es schon früh Überlegungen, einen sprachlichen Konstrukt zu finden, der Daten und Operationen als geschützte Einheit mit wechselseitigem, ausschließendem Benutzungsrecht anbietet. Monitore sind ein solcher Ansatz und gehen auf die Arbeiten von Per Brinch Hansen [20] und C. A. R. Hoare [21] aus den Jahren 1973 und 74 zurück.

Der Schutz der Daten und die Regelung des wechselseitigen Ausschlusses werden als Aufgabe des Compilers gesehen. Der Entwerfer des Monitors sichert typgerechte Operationen zu und besorgt eine Warteschlangenverwaltung mit Warte- und Weckanweisungen.

Wie wegweisend diese Arbeiten waren, zeigt ein Zitat aus `http://www.boost.org/libs/thread/doc/bibliography.html` zur Arbeit von Brinch Hansen:

> Also noteworthy because of an introductory quotation from Christopher Alexander; Brinch Hansen was years ahead of others in recognizing pattern concepts applied to software too.

Ein Monitor *M* enthält Operationen (Methoden) in Form von Prozeduren E_i, die durch $M.E_i$ aufgerufen werden. Vereinbarungen im Monitor (die gekapselten Daten) überleben den Aufruf des Monitors.

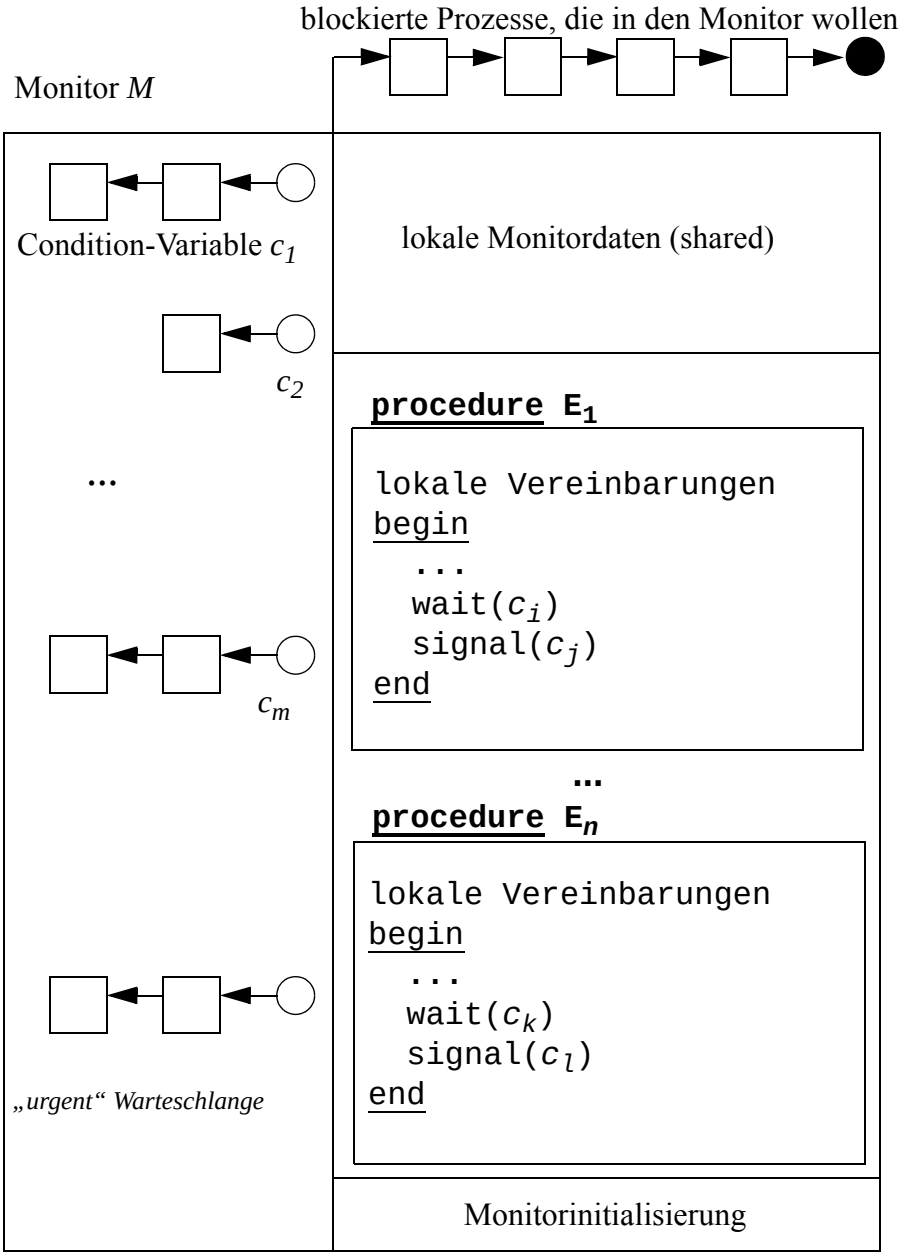


Abb. 5–1 *Aufbau eines Monitors*

Zur Wirkungsweise eines Monitors gilt:

- Der Aufruf einer Monitorprozedur *E* blockiert den Monitor für andere Aufrufe bis der Monitor (von innen her) wieder freigegeben wird.
- Die Condition-Variablen sind FIFO-Warteschlangen und werden manipuliert durch
- **wait(*c*)**
  der rufende Prozeß wird in *c* blockiert, der Monitor wird freigegeben (warum?)
- **signal(*c*)**
  wenn *c* leer, dann nichts, sonst wecke den ersten Prozeß in *c*.

Damit diese Konstruktion sicher funktioniert, nimmt man an, daß *signal*(*c*) nur als letzte Anweisung einer Monitorprozedur auftreten darf. Dann kann der signalisierende Prozeß den Monitor verlassen, bevor der aufgeweckte[1] weitermacht, der die Kontrolle über den Monitor übernimmt.

Beispiele für die Auswirkungen bei Verletzung der Annahmen:

- Sei P_1 in der Warteschlange *c* eines Monitors *M*. P_2 macht **signal**(*c*) und weckt P_1. P_1 macht sofort wieder **wait**(*c*'). Welchen Status hat dann P_2?
- Ein Puffer sei leer ($n = 0$). Ein Verbraucherprozeß P_1 sei in einer Warteschlange *c*. Ein Erzeugerprozeß lädt den Puffer (jetzt $n = 1$), weckt P_1 durch **signal**(*c*) und **gibt den Monitor frei!** Ein zweiter Verbraucherprozeß P_2 betritt den Monitor und leert den Puffer. Jetzt will auch P_1 den Puffer leeren, aber ...?

Hinweis: Der Monitorrumpf wird beim Programmstart durchlaufen und bewirkt die Initialisierung des Monitors.

1. Aufgeweckt im Sinne von „wiederbelebt“; über kluge und dumme Prozesse ist in der Literatur nichts bekannt.

5.2 Einige klassische Beispiele

Wir beginnen mit dem Erzeuger-Verbrauchersystem für einen begrenzten Puffer. Die Daten im Puffer wandern kreisförmig nach rechts durch den Puffer. Es wird an der Stelle `in` geschrieben und an der Stelle `out` gelesen. Eine Variable `n` zeigt den Füllstand.

```
program ProducerConsumer;
const size-of-buffer: ...
monitor BB {bounded buffer};
  var b: array[0..size-of-buffer] of data;
    in, out, n: integer;
    notempty, notfull: condition;

procedure append(v: data); {füllen}
begin
  if n = size-of-buffer + 1 {Puffer voll}
  then wait(notfull);
  b[in] := v; in := in + 1;
  if in = size-of-buffer + 1 then in := 0;
  n := n + 1;
  signal(notempty)
end;

procedure take(var v: data); {leeren}
begin
  if n = 0 {Puffer leer} then wait(notempty);
  v:= b[out]; out := out + 1;
  if out = size-of-buffer + 1 then out := 0;
  n := n - 1;
  signal(notfull)
end;

begin {Monitorrumpf}
  in := 0; out := 0; n := 0
end {Ende Monitor};
```

Verwendung ist wie folgt:

```
procedure Erzeuger;
var v: data;
begin
  repeat
    erzeugen v
    BB.append(v);
```

```
  forever
end;

procedure Verbraucher;
var v: data;
begin
  repeat
    BB.take(v);
    verbrauchen v
  forever
end;

begin {Rahmenprogramm}
  cobegin Erzeuger; Verbraucher coend
end.
```

Das nächste Beispiel greift die Vier-Bänder-Verwaltung auf.

```
monitor TAPMON;
var Tape: array[1..4] of (frei, belegt);
    i,busycount: integer {Anzahl der belegten Einheiten}
    nonbusy: condition;

procedure GetTape(var ActTape: integer);
var j: integer;
begin
  j := 1;
  if busycount = 4 then wait(nonbusy);   {<------------}
  while (Tape[j] = belegt) do j := j + 1;
  Tape[j] := belegt; ActTape := j;
  busycount := busycount + 1
end;

procedure PutTape(ActTape: integer);
begin
  Tape[ActTape] := frei;
  busycount := busycount - 1;
  signal(nonbusy)   {<------------}
end;

begin {Monitorinitialisierung}
  for i := 1 to 4 do Tape[i] := frei;
  busycount := 0
end; {Monitor}
```

Das nächste Beispiel ist die Realisierung des Readers-Writers-Problem durch einen Monitor mit Bevorzugung der Reader.

```
monitor RD;
var readcount: integer;
  writeflag: Boolean {true = schreibend};
  OK_to_read, OK_to_write: condition;

procedure startread;
begin
  if writeflag then wait(OK_to_read);
  readcount := readcount + 1;
  signal(OK_to_read)
end;

procedure endread;
begin
  readcount := readcount - 1;
  if readcount = 0 then signal(OK_to_write);
end;

procedure startwrite;
begin
  if (readcount > 0) or writeflag then
wait(OK_to_write);
  writeflag := true
end;

procedure endwrite;
begin
  writeflag := false;
  if nonempty(OK_to_read) {neue Monitor-Operation}
  then signal(OK_to_read)
  else signal(OK_to_write);
end;

begin {Monitorinitialisierung}
  readcount := 0;
  writeflag := false
end;
```

Man beachte die neue Operation **nonempty**(*c*) die true liefert gdw. die Warteschlange von *c* nicht leer ist.

Die Benutzung des RD-Monitors ist wie folgt.

```
procedure Writer;
begin
  repeat
    RD.startwrite;
    schreiben Daten
    RD.endwrite
  forever
end;
```

Die Leserprozedur ist analog.

5.3 Äquivalenz der Synchronisierungskonstrukte

Damit man zeigen kann, daß Monitor und Semaphor bezüglich der Fähigkeiten gleichwertig sind, gibt man die wechselseitige Simulation an.

Simulation der Semaphore durch einen Monitor

```
program WechselseitigerAusschluss;
monitor semasim;
var busy: Boolean;
    nonbusy: condition;

procedure P;
begin
  if busy then wait(nonbusy);
  busy := true
end;

procedure V;
begin
  busy := false;
  signal(nonbusy);
end;

begin {Monitorinitialisierung}
  busy := false
end {monitor};

procedure P1;
begin
  repeat
    semasim.P;
    kritischer Abschnitt
    semasim.V;
    unkritischer Teil
```

```
  forever
end;

procedure P2;
...

begin {Rahmen}
  cobegin P1; P2; ... coend
end.
```

Simulation eines Monitors

Die Umkehrung ist etwas aufwendiger.

- Der wechselseitige Ausschluß der Monitor-Entryprozeduren E_i muß simuliert werden. Dies erledigt das Semaphor *s* für den Monitor *M*. Jede Monitorprozedur beginnt mit einem *P*(*s*) und endet mit *V*(*s*), außer bei *signal* als letzter Anweisung. Der Anfangswert von *s* ist 1 (*seminit*(*s*, 1)).
- Alle Condition-Variablen werden durch je einen Integerzähler *condcount* (anfangs 0) und ein binäres Semaphor *condsem* simuliert. Das Semaphor *condsem* blockiert wartende Prozesse und ist anfangs auf 0.
- *wait*(*c*) wird zu

  ```
  condcount := condcount + 1;
  V(s);
  P(condsem);
  condcount:= condcount - 1;
  ```

- *signal*(*c*) wird zu

  ```
  if condcount > 0 then V(condsem) else V(s)
  ```

  Dies setzt aber voraus, daß *signal* letzte Operation in der Monitorprozedur ist.

Simulation eines Monitors bei unbeschränkter signal-Operation

Bei allen Prozedureingängen:

```
P(s)
```

Zur Simulation einer Operation *wait*(*c*):

```
condcount := condcount + 1;
if urgent > 0 {falls signalisierende Prozesse}
then V(usem) {warten, laß einen frei}
else V(s); {sonst laß neuen Prozeß rein}
P(condsem);
condcount := condcount - 1;
```

Zur Simulation einer Operation *signal*(*c*):

```
urgent := urgent + 1;
if condcount > 0 {wenn jemand wartet}
then begin {lass ihn rein}
  V(condsem); P(Usem)
end; {und blockiere dich selbst}
urgent := urgent - 1;
```

Beim Prozedurausgang:

```
if urgent > 0 then V(usem) else V(s)
```

Man sieht hier die Verwendung der „*urgent-Warteschlange*“ für signalisierende Prozesse. Die Kontrolle geht gemäß Definition des Monitors an den aufgeweckten Prozeß über. Ist *signal* letzte Operation der Prozedur im signalisierenden Prozeß, ist dies unkritisch, da der Prozeß gleichzeitig den Monitor verläßt. Wenn nicht, hat er höchste Priorität bezüglich des Fertigarbeitens, nachdem der geweckte Prozeß (und alle von ihm geweckten Prozesse!!) ihrerseits fertig wurden.

5.4 Beurteilung

Nachteile des Monitorkonzepts sind z.B. die unklare Situation bei geschachtelten Aufrufen.

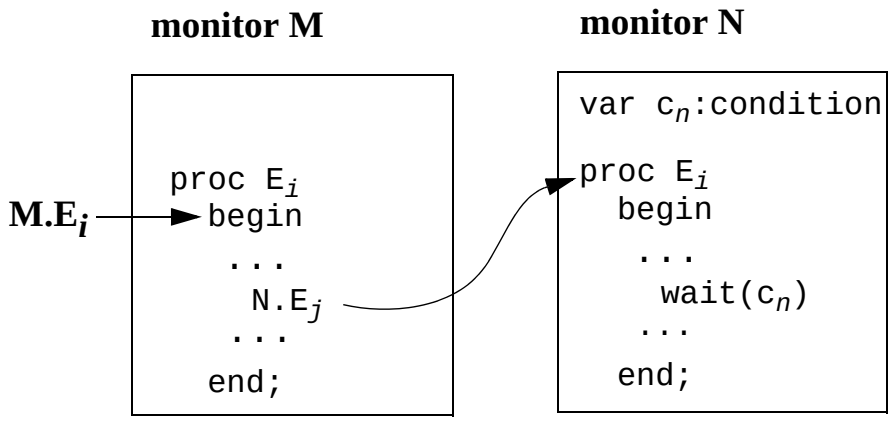


Abb. 5–2 *Geschachtelte Monitoraufrufe – M und N sind blockiert*

Diese Art der Schachtelung ist in MODULA verboten, in Java sind geschachtelte Aufrufe zugelassen. Anders ist die Situation, wenn N in M deklariert ist (geschachtelte Monitore).

Ein zweiter Nachteil ist die Verzögerung des signalisierenden Prozesses, weil ein Prozeß P_1, der aufgrund des Eintritts einer Bedingung B ein *signal*(c) macht, die Kontrolle über den Monitor abgibt. Kehrt die Kontrolle zu P_1 später zurück, kann die Bedingung B von anderen bereits wieder verletzt sein.

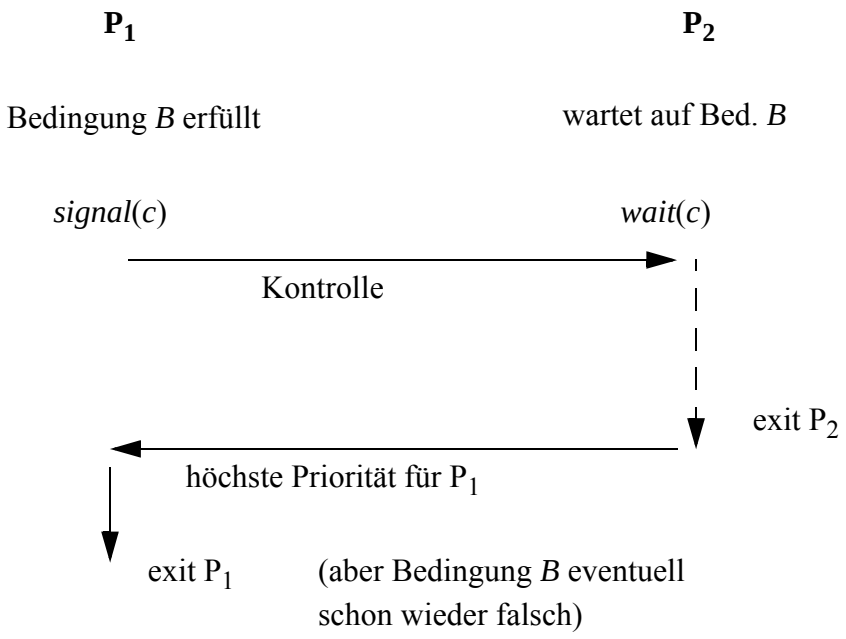


Abb. 5–3 *Verzögerung des Signalisierers*

Beispiel 5–1

Ein Direktor mit Anspruch auf Dienstwagen und Fahrer testet die Bedingung „Dienstwagen verfügbar“ und weckt den Fahrer bei Erfüllung. Wenn der Direktor-Prozeß wieder drankommt, ist aber ggf. der Wagen schon wieder weg, z.B. weil geweckte Fahrer grundsätzlich nach fahrwilligen Direktoren suchen müssen und zwischenzeitlich noch jemand weiter oben in der Liste eingetragen wurde.

Daher gibt es Implementierungen, die zuerst den signalisierenden Prozeß abarbeiten (also P_1), erst dann sofort den signalisierten (also P_2) behandeln. Hier entsteht aber das gleiche, nur spiegelbildlich getauschte Problem, daß jetzt für P_2 die Bedingung *B* bereits wieder falsch sein kann, wenn die Kontrolle an ihn übergeht.

Daher ersetzt man in diesen Monitoren die ursprüngliche Zeile

```
if not B then wait(c);
```

durch

```
while not B do wait(c);
```

Grundsätzlich liegt dies daran, daß – anders als die *V*-Operation bei Semaphoren – *signal* keine Spur hinterläßt.

5.5 Monitor Klassifikation

Die folgenden Einteilungen stammen aus der Arbeit von Buhr, Fortier und Coffin [22] aus dem Jahr 1995.

Danach unterscheidet man nach expliziter Synchronisierung mittels *wait*(*q*) und *signal*(*q*) für Condition-Variablen (Ereignis-Warteschlangen) *q* wie gehabt einerseits und *impliziter Synchronisierung* (Hoare 1974 [21]) ohne *condition variables*, ohne *signal*, stattdessen mit

```
wait Bedingter Ausdruck
```

z. B. wie mit

```
uWaitUntil count != 0
```

wie in der Sprache *Edison* von Per Brinch Hansen 1981 [24] realisiert. Bei letzterer Lösung entsteht aber ein hoher Aufwand durch Kontextumschaltungen, da wartende Prozesse reihum geweckt werden und die Wartebedingung prüfen.

Für die Monitorvergabe betrachtet man die folgende generelle Situation.

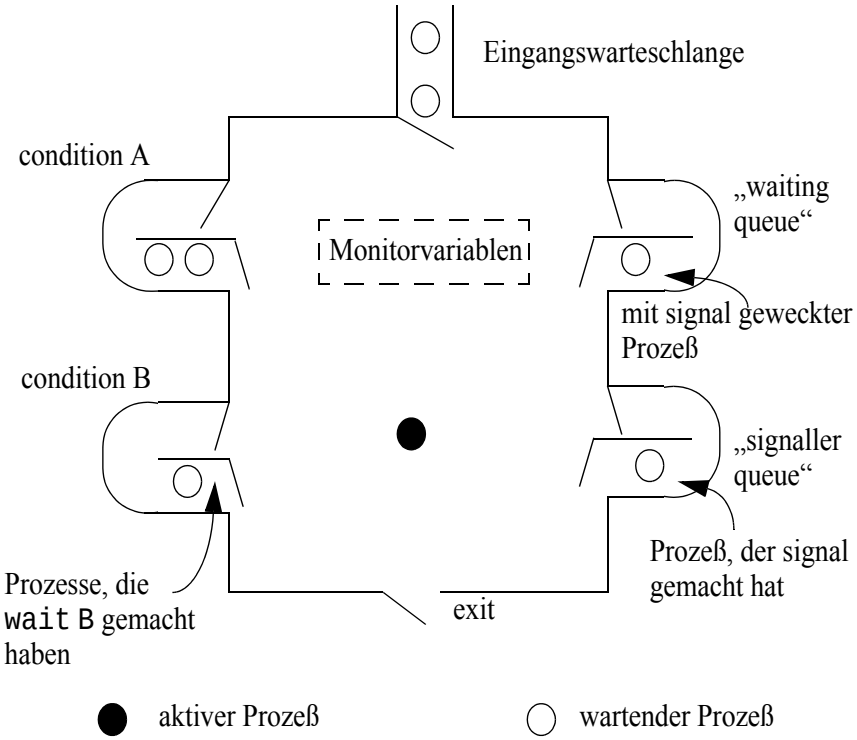


Abb. 5–4 *Prozesse, die auf den Eintritt in den Monitor warten*

Wird der Monitor nach einer signal-Operation oder durch Beendigung einer Entry-Prozedur freigegeben, gibt es drei Warteschlangen, die bereite Prozesse für die Übernahme enthalten:

- entry queue - Priorität E_p
- waiting queue - Priorität W_p
- signaller queue - Priorität S_p

Die Monitorvergabe in dieser Situation ist nicht benutzergesteuert und ist wesentliches Klassifizierungsmerkmal.

Daraus entsteht nach [22] die folgende Tabelle:

	relative Priorität	traditioneller Monitorname
1	$E_p = W_p = S_p$	
2	$E_p = W_p < S_p$	Wait and Notify [Lamson and Redell 1980] Andrew: signal and continue
3	$E_p = S_p < W_p$	Signal and Wait [Howard 1976a]
4	$E_p < W_p = S_p$	
5	$E_p < W_p < S_p$	Signal and Continue [Howard 1976b]
6	$E_p < S_p < W_p$	Signal and Urgent Wait [Hoare 1974]
7	$E_p > W_p = S_p$	wenig sinnvoll (1)
8	$E_p = S_p > W_p$	wenig sinnvoll (2)
9	$S_p > E_p > W_p$	wenig sinnvoll (2)
10	$E_p = W_p > S_p$	wenig sinnvoll (2)
11	$W_p > E_p > S_p$	wenig sinnvoll (2)
12	$E_p > S_p > W_p$	wenig sinnvoll (1)
13	$E_p > W_p > S_p$	wenig sinnvoll (1)

- Wenn E_p höchste Priorität hat, kann es zu einem Aushungern der anderen Prozesse kommen. (1)
- Gilt $E_p > W_p$ oder $E_p > S_p$, gilt gleiches, wenn es einen kontinuierlichen Strom ankommender Prozesse in der *entry queue* gibt. (2)

Bei gleichen Prioritäten wählt der Monitor-Scheduler beliebig. Dies führt zu einem Programmierstil, bei dem *signal* als Hinweis (hint) betrachtet wird, die Freigabebedingung könnte erfüllt sein. Der geweckte Prozeß muß dies prüfen!

Monitor mit sofortiger Beendigung

Immediate-Return Monitor: *signal* ist letzte Anweisung in Entry-Prozedur, aber [Howard76] und [Andrews91] zeigen, daß diese Nebenbedingung die Ausdrucksmächtigkeit verringert.

Extended Immediate-Return Monitor: signal ist ... oder *signal* erfolgt unmittelbar vor *wait* [Howard76]. Dies bedeutet keine Einschränkung der Mächtigkeit.

In beiden Fällen gibt es keine signaller-queue, genausowenig für *automatic signal monitors* (implizite Synchronisierung, vgl. oben). Daher gilt Tabelle 2 auch dafür und Eintrag 2 ($E_p < W_p$) heißt *Automatic Signal* [Hoare74].

	relative Priorität	**traditioneller Monitorname**
1	$E_p = W_p$	
2	$E_p < W_p$	Signal and Return [Brinch Hansen 1975]
3	$E_p > W_p$	wenig sinnvoll

Tab. 5–1 *Relative Prioritäten für Extendet Imm.-Return Monitore*

Zusammenfassung der neuen Klassifikation

1. Unterscheidung:
   Haben Prozesse, die schon im Monitor sind Priorität über außen wartende Prozesse? $W_p > E_p$ und/oder $S_P > E_p$
2. Unterscheidung: Wird signalisierender Prozeß blockiert?
   ja 1. Zeile (Blocking)
   nein 2. Zeile (Non-Blocking)
   vielleicht 3. Zeile (Quasi-Blocking)
   nicht zutreffend 4. und 5. Zeile

Signal Charakteristik	Priorität	Keine Priorität
Blocking	Signal and Urgent Wait $E_p < S_p < W_p$ *Priority Blocking (PB)*	Signal and Wait $E_p = S_p < W_p$ *No Priority Blocking (NPB)*
Non-Blocking	Signal and Continue $E_p < W_p < S_p$ *Priority Non-Blocking (PNB)*	Wait and Notify $E_p = W_p < S_p$ *No Priority Non-Blocking (NPNB)*
Quasi-Blocking	Signal and Wait $E_p < W_p = S_p$ *Priority Quasi-Blocking (PQB)*	$E_p = W_p = S_p$ *No Priority Quasi-Blocking (NPQB)*
Extended Immediate Return	Signal and Return $E_p < W_p$ *Priority Imm. Return (PRET)*	Signal and Wait $E_p = W_p$ *No Priority Imm. Return (NPRET)*
Automatic Signal	Automatic Signal $E_p < W_p$ *Priority Autom. Signal (PAS)*	$E_p = W_p$ *No Priority Autom. Signal (NPAS)*

Tab. 5–2 *Nützliche Monitortypen*

Daraus leiten sich Folgerungen für (a) Zusicherungen und (b) Leistungsvergleiche ab:

a) Monitore mit explizitem Signal sind effizient aber schwierig zu programmieren und zu verifizieren. *Eingeschränkte Automatic Signal Monitore* (kein Test lokaler Variablen und Parameter von Entry-Prozeduren in Wait-Ausdrücken) sind ein guter Kompromiß.

b) Blockierende Monitore machen 30 % früher „dicht“ (*saturate*) als nichtblockierende; nichtblockierende Monitore verbessern die Antwortzeiten von Anwendungen (Readers/Writers) um 5-10 % gegenüber blockierend, sind aber nicht „schneller“ als *immediate-return Monitore*.

Fazit

Der Monitortyp *Priority Non-Blocking* (PNB), Signal and Continue, $E_p < W_p < S_p$) ist eine gute Wahl mit mittlerem Semantikaufwand (Beweisregeln), guter Laufzeit und hoher Grenze für Überlast. Ferner ist er leicht zu programmieren.

6 Das Botschaftenkonzept

Bei der Synchronisierung mittels Semaphor und Monitor können Prozesse globale (Monitor-)Variablen lesen und schreiben. Die eigentlichen Operationen, also *P*, *V*, *wait* und *signal* enthalten aber keine Nachrichten.

Ein neuer Ansatz ist die Synchronisierung über Botschaftenaustausch (messages), da diese eine klare zeitliche Komponente enthalten: eine Botschaft kann nicht früher empfangen werden, als sie abgeschickt wurde!

6.1 Die Grundoperationen *send* und *receive*

Hierzu legen wir die folgenden Grundoperationen für ein Botschaftenkonzept fest.

```
send Ausdrucksliste to Zielbezeichner
receive Variablenliste from Quellbezeichner
```

Die Semantik der *send*-Operation legt fest, daß die Ausdrücke der Ausdrucksliste zum Zeitpunkt der Ausführung der *send*-Operation ausgewertet werden.

Die Semantik der *receive*-Operation sieht die Zuweisung der Botschaftswerte an die Variablen der Liste vor. Anschließend wird die Botschaft zerstört (verbrauchendes Empfangen).

Ansonsten können sinngemäß die Regeln der Schreib- und Leseanweisungen, etwa in der Bourne-Shell von UNIX, genommen werden.

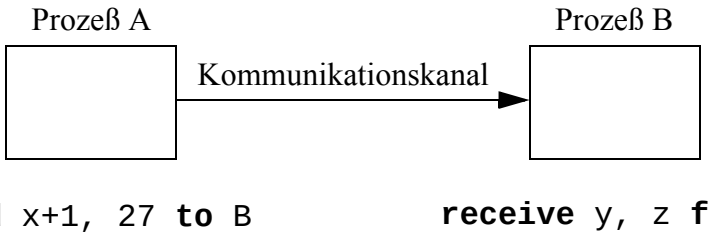


Abb. 6–1 *Botschaftenkonzept*

Damit taucht die Frage auf, wie man Kommunikationskanäle spezifiziert.

Direct Naming (Direkte Namensangabe)

Die Form kann der Abb. 6–1 entnommen werden. Der Vorteil liegt in der einfachen Implementierung, oft als Pipeline realisiert. Viele Client-/Serversysteme arbeiten mit dieser direkten Benennung der Auftragnehmer (Klienten, clients) und Auftraggeber (Server), wobei einer Auftragserteilung mindestens eine Erledigungsbestätigung in die Gegenrichtung folgt.

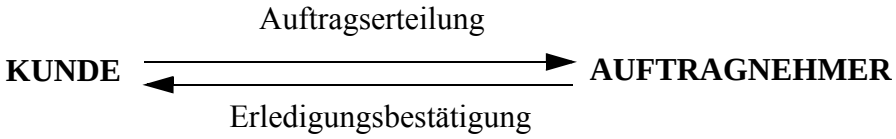


Abb. 6–2 *Botschaftenkonzept in Client-/Serveranwendung*

Das Grundgerüst eines Eingabe-Verarbeitung-Ausgabe Betriebssystems sieht demnach wie folgt aus.

```
program OPSYS;

procedure Reader;
var eingabe: Eingabeliste;
begin
  repeat
    lesen eingabe von Leser;
    send eingabe to Executer;
  forever
end {Reader};

procedure Executer;
var eingabe: Eingabeliste;
    ausgabe: Ausgabezeile;
begin
  repeat
    receive eingabe von Reader;
    verarbeiten Eingabeliste und erzeugen Ausgabezeile;
    send ausgabe to Printer;
  forever
end {Executer};

procedure Printer;
var ausgabe: Ausgabezeile;
begin
  repeat
    receive ausgabe von Executer;
    drucken Ausgabezeile auf Drucker;
  forever
end {Printer};

begin
  cobegin Reader; Executer; Printer coend
end.
```

Die „Direct-Naming-Lösung“ hat offensichtliche Nachteile. Ein Auftragnehmer sollte von jedem beliebigen Kunden Aufträge annehmen, d.h. man möchte (zusätzlich?) ein *receive* ohne *from*.

Analog könnte ein Auftraggeber einen Auftrag pauschal an eine Menge (gleichwertiger) Auftragnehmer vergeben, d.h. man möchte ein *send* ohne *to*.

Briefkastensystem (global names, mailboxes)

Abhilfe schafft ein Briefkastensystem in das Aufträge abgelegt werden und aus dem sich Auftragnehmer Aufträge holen.

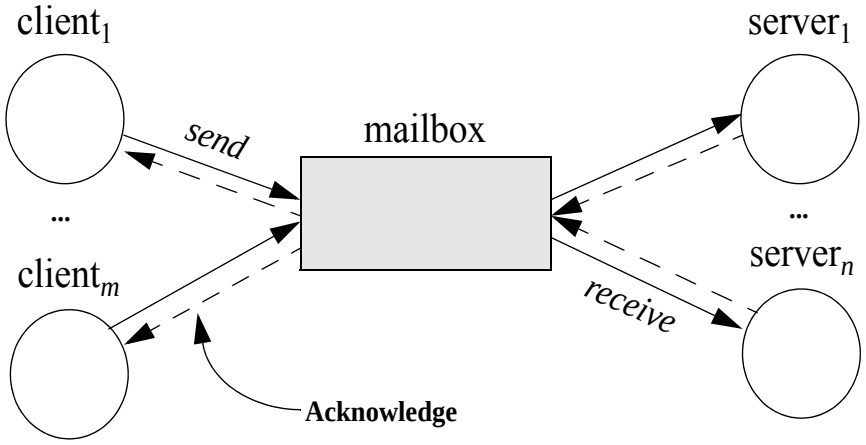


Abb. 6–3 *Client-Server mit indirect naming über Mailbox*

Allerdings setzt dies ein spezialisiertes Kommunikationsnetz voraus. Die Tatsache, daß etwas im Briefkasten eingetroffen ist, muß den Auftragnehmern bekanntgemacht werden. Bei Entleerung muß mitgeteilt werden, daß die Botschaft (der Auftrag) nicht mehr verfügbar ist.

Als Spezialfall eines Briefkastensystems kann man *Ports* bezeichnen, bei denen höchstens ein Server ein *receive* auf den Port machen darf.

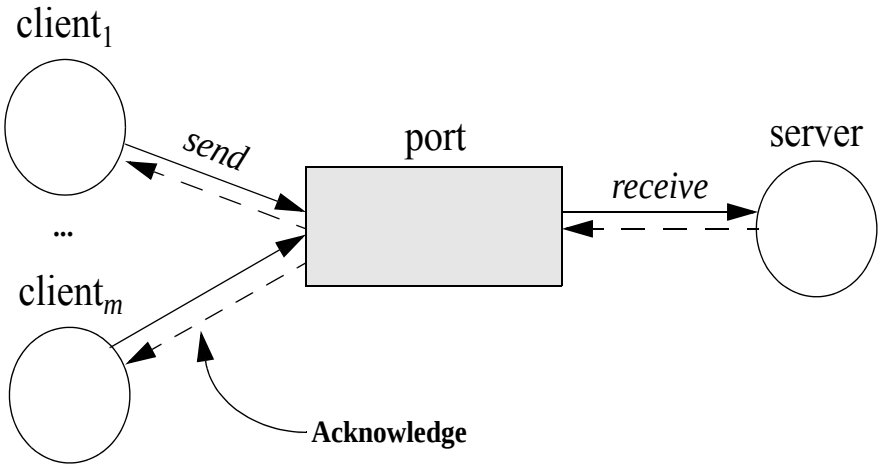


Abb. 6–4 *Client-Server mit indirect naming über Port*

Für die Bindung der Namen (direct naming, ports, mailboxes), die auch als Channel Naming bezeichnet wird, unterscheidet man *statische Bindung* (zur Übersetzungszeit) und *dynamische Bindung* (zur Laufzeit, analog zur Bindung Programm - Dateideskriptor - Datei in UNIX, wobei am Dateideskriptor auch mittels Pipemechanismus ein anderer Prozeß hängen kann).

6.2 Der Synchronisierungseffekt von Botschaften

Es gibt drei offensichtliche Möglichkeiten:

a) kein Warten nach *send*
b) bedingtes Warten nach *send*
c) unbedingtes Warten nach *send*

Den Fall (a) nennt man ein *asynchrones Botschaftensystem*, auch „send-no-wait“ oder *non-blocking send* genannt. Es arbeitet im Prinzip mit einem unbegrenzten Puffer und sendende Prozesse können beliebig vorauseilen, weshalb empfangene Botschaften ggf. stark veraltet sind. Empfänger warten bei leerem Puffer oder geben eine Fehlermeldung aus (analog zu einem *read* in einem „normalen“ Programm).

Der Fall (b) ist ähnlich, arbeitet aber mit einem begrenzten Puffer. Der sendende Prozeß kann damit nicht beliebig vorauseilen.

Der Fall (c) ist ein *synchrones Botschaftensystem*, das ohne Puffer arbeiten kann. Der sendende Prozeß wartet am *Synchronisationspunkt* auf den empfangenden Prozeß.

Ein *receive* wird meistens blockierend ausgelegt sein (wenn nichts da ist, warten). Im Fall unten wartet der Prozeß dann auf die Botschaft von *A*, obwohl vielleicht schon die Nachricht von *B* eingetroffen ist.

```
...
receive X from A;
receive Y from B;
...
verarbeiten X und Y in beliebiger Reihenfolge
...
```

Abhilfe brächte eine Testmöglichkeit zum Feststellen, ob Nachrichten vorliegen, ohne blockiert zu werden.

Betrachtungen zur Korrektheit

Als Grundregel stellt eine Botschaft immer eine Zusicherung des Absenders über den Zustand des Systems dar. Damit ist

- ein synchrones Botschaftensystem immer sehr sicher
- ein asynchrones ~ nur sicher, wenn der Sender die Zusicherung beim Weitermachen nicht verletzt.

Programmiersprachenkonzepte mit Botschaften

Von Hoare [26] stammt aus dem Jahr 1978 die Sprache CSP (Communicating Sequential Processes) für ein synchrones Botschaftensystem, in dem globale Variable nur gelesen werden dürfen und das *direct naming* sowie *statische Bindung* für Kanalnamen verwendet.

Die Operationen lauten

```
Zielangabe ! Ausdruck
```

für die Ausgabe und

```
Quellangabe ? Variable
```

für die Eingabe, also z. B. im Prozeß *Executer* den Aufruf

```
Printer ! 'Dies ist ein Text'
```

und im *Printer*-Prozeß den Aufruf

```
Executer ? MyLine.
```

Eine Sprache für asynchronen Botschaftenaustausch ist PLITS (Programming Language in the Sky) der Univ. of Rochester (Feldman 1979 [25]). Microsoft plant angeblich „next generation asynchronous programming languages (eg. XLANG, xSPresso)“[1]. Java hat Packages für beide Arten von Botschaftenkonzept.

In C (und anderen Sprachen) kann man unter UNIX (und inzwischen auch unter Windows) RPC (remote procedure call) und Socket Systembibliotheken für die Interprozeßkommunikation verwenden. Wir gehen darauf in einer gesonderten Vorlesung ein.

Wir beenden das Kapitel über Synchronisierung mittels Botschaften mit einem Ausflug in die Programmiersprache Ada.

6.3 Das Ada-Rendezvous

Die Programmiersprache Ada[2] wurde in den Jahren 1974 - 1981 für das US-Verteidigungsministerium (DoD) entwickelt. Die Sprache selbst stammt von einem französischen Team unter Leitung von *Jean Ichbiah* nach einer Spezifikation des internationalen Definitionskommittees, das seine Anforderungen in fünf Dokumenten (Strawman, Woodenman, Tinman, Ironman, Steelman) festlegte. Seit Februar 1983 ist Ada ein ANSI Standard, seit 1987 ein ISO Standard. Im Juli 1988 startete das Ada 9X Projekt zur Revision der Ada Programmiersprache. Die Firma Interme-

1. http://research.microsoft.com/behave/TypesAsModels.ppt.
2. nach Augusta Ada Byron, Countess of Lovelace, geboren 10. Dez. 1815, Tochter des Dichters Lord Byron, Kollegin von Charles Babbage, erste Programmiererin und Autorin des ersten Programmierbuchs

trics wurde Hauptauftragnehmer für ein „mapping/revision team“ für den neuen Ada 9X Standard, Technischer Direktor war S. Tucker Taft. Ada95 wurde als gemeinsamer ISO und ANSI Standard im February 1995 angenommen. Vgl. `http://www.cs.fit.edu/~ryan/ada/ada-hist.html`

Interessant für uns ist das Synchronisierungskonzept in Ada, genannt *Rendezvous*. Es ist grundsätzlich ein synchrones Botschaftenkonzept und lehnt sich an die Grundidee der *remote procedure call* an, d.h. ein Auftragnehmer wird durch den Aufruf einer Serviceprozedur aktiviert.

Prozesse heißen Tasks und werden aktiviert, wenn ein Block betreten wird, in dem die Task vereinbart ist. Tasks (Blöcke) dürfen geschachtelt werden und können unbeschränkt globale Variable benützen (vgl. Monitor).

Die Prozeßinteraktion geschieht durch

Aufrufe (calls), **accept**, **select**, **delay**

Die Aufrufe der Prozeduren (ein eigentliches *call*-Statement gibt es nicht) entsprechen einem *send*, **accept** wirkt wie ein *receive*, beide sind blockierend, ihr Synchronisationspunkt ist das *Rendezvous*, das auch mit einem Namen versehen werden kann.

Was bei einem Rendezvous passiert (z. B. die Übergabe einer Botschaft), bestimmt der accept-Rumpf einer Entry-Prozedur, die durch den Aufruf einer anderen Task aktiviert wird. Wie heute üblich werden Tasks und Entries getrennt definiert und deklariert (bekannt gemacht).

Am leichtesten versteht man das Konzept an einem Beispiel, hier dem klassischen Erzeuger-Verbraucher-System (in Pseudo-Ada).

```
procedure ConsumerProducer is

task Buffer is
  entry append(Daten: in Datentyp);
  entry fetch(Daten: out Datentyp);
end Buffer;

task Producer;
task body Producer is
erzeugteDaten: Datentyp; -- kein var
begin
  loop
    produzieren erzeugteDaten;
    Buffer.append(erzeugteDaten);
  end loop; -- forever
end Producer;

task Consumer;
task body Consumer is separate;

task body Buffer is
full: Boolean; Bufferspace: Datentyp;
begin
  full := false;
  loop
    if full then
      accept fetch(Daten: out Datentyp) do
        Daten := Bufferspace;
        full := false;
      end fetch
    else -- Puffer leer
      accept append(Daten: in Datentyp) do
        Bufferspace := Daten;
        full := true;
      end append;
    end if;
  end loop;
end Buffer;

begin
  null;
end ConsumerProducer;
```

Hinweise: Wie man sieht, gibt es kein *cobegin*/*coend*. Die Taskabarbeitung beginnt mit Eintritt in die Prozedur *ConsumerProducer*.

Die gezeigte Lösung selbst ist umständlich und nicht sehr flexibel, da nur eine Folge von *append*, *fetch*, *append*, *fetch*, ... möglich ist. Auch wäre ein direktes Rendez-vous zwischen Consumer und Producer möglich.

Die Entry-Warteschlangen sind FIFO, was hier aber keine Rolle spielt. Auch weiß der Prozeß *Buffer* nicht, von wem er aufgerufen wird.

Die folgenden Abbildungen zeigen den Synchronisationsablauf in CSP und in Ada.

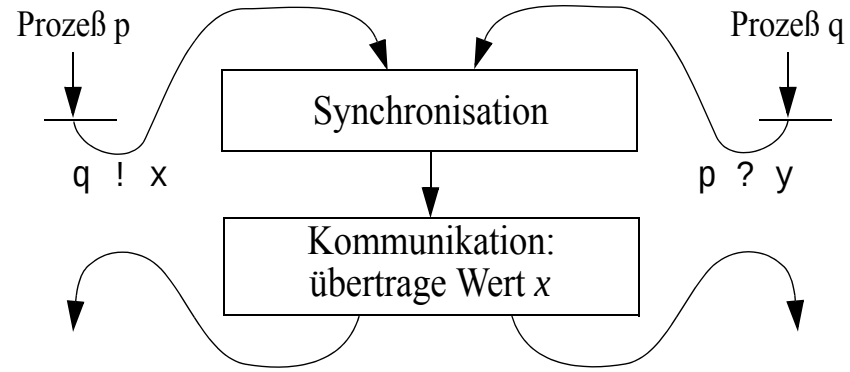


Abb. 6–5 *CSP*

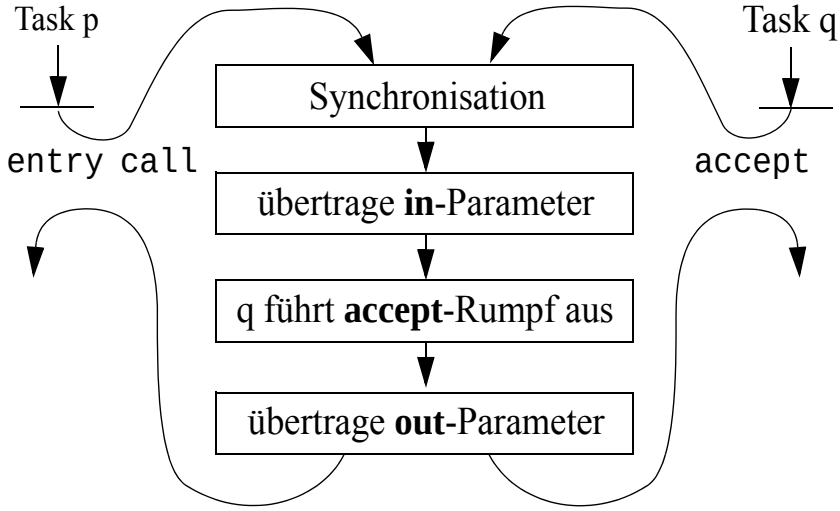


Abb. 6–6 *Ada*

Die einzelnen Schritte sind:

- P wünscht Rendezvous mit Q, d. h. P hat den Aufruf der Entry-Prozedur gemacht (die Anweisung erreicht).
- P signalisiert den Wunsch und blockiert.
- Q registriert den Wunsch von P beim Erreichen der *accept*-Anweisung durch Unterbrechungsvektor oder Polling.
- Q teilt P mit, daß Rendezvous stattfinden kann und blockiert bis P Antwort bestätigt und Parameter (P's out = Q's in) verschickt sind.
- Q führt Rumpf des *accept*-Befehls aus und schickt danach out-Parameter weg.
- P macht weiter nach Erhalt seiner in-Parameter.

Für die Ein-/Ausgabe können Entries an Unterbrechungsvektoren gebunden werden. Damit ist ein Interrupt ein entry call, die Unterbrechungsbedienung erfolgt durch die Task mit dem *accept* auf dem Unterbrechungsvektor.

Wir simulieren nun den wechselseitigen Ausschluß über Semaphore in Ada.

```
procedure Mutex is
task semaphore is
  entry P;
  entry V;
end semaphore;

task body semaphore is
begin
  loop
    accept P; -- kein accept Rumpf nötig
    accept V; -- kein accept Rumpf
  end loop;
end semaphore;

task P1;
task body P1 is
begin
  loop
    unkritischer Teil;
    semaphore.P;
    kritischer Abschnitt;
```

```
      semaphore.V;
    end loop;
end P1;

task P2
...

begin
  null;
end Mutex;
```

Man sieht, daß das Betreten des kritischen Abschnitts über ein Rendezvous mit dem Semaphore über die Entry-Prozedur *P* erfolgt. Ein zweites Rendezvous mittels *V* ist beim Verlassen nötig.

Zuletzt betrachten wir die **select**-Anweisung in Ada. Man benötigt sie z. B. beim Bounded-Buffer Problem. Für einen begrenzten Puffer gilt:

- ist der Puffer leer, dann nur Rendezvous mit Erzeuger
- ist der Puffer voll, dann nur Rendezvous mit Verbraucher
- ist der Puffer weder voll noch leer, dann Rendezvous beliebig, aber keine Blockade wegen „falschem Partner“

Die *select*-Anweisung ähnelt dem *guarded command* von Dijkstra, geschrieben als $b_1 \rightarrow c_1 \mid b_2 \rightarrow c_2 \mid ... \mid$ **`else skip`**, d.h. Anweisungen c_i werden durch Bedingungen (guards) b_i geschützt, die man sich wie Türen vorstellen kann. Wesentlich ist das nicht-deterministische Verhalten, da mehrere Türen offen sein können und der Kontrollfluß dann beliebig durch eine offene Tür geht.

In Ada sieht das dann syntaktisch so aus:

```
select
    when Bedingung_1 =>accept Entry_1 do
                           Entry-Anweisungen
                       end;
                       andere Anweisungen
    or
    when Bedingung_2 =>accept Entry_2 do
                           Entry-Anweisungen
                       end;
                       andere Anweisungen
    or ...
```

```
    else Anweisungen
end select;
```

Für die Semantik gilt:

- Es werden zuerst alle Bedingungen (guards) ausgewertet, um festzustellen, welche Alternativen offen sind.
- Dann wird geprüft, an welchen offenen Alternativen Tasks warten.
- Eine solche Alternative wird ausgewählt (bei mehreren möglichen beliebig).
- Ist keine Alternative offen oder wartet keine Task, dann else-Teil ausführen.
- Fehlt der else-Teil, dann warten. Jetzt ist Rendez-vous möglich mit jeder ankommenden Task, die auf eine offene Alternative stößt.
- Fehlender else-Teil und keine offene Alternative ist ein Fehler.

Wir geben jetzt eine Lösung für den Bounded Buffer an.

```
task BB is
  entry append(v: in integer);
  entry take(v: out integer);
end BB;
task body BB is
  size: constant := ...;
  b: array(0 .. size) of integer;
  inptr, outptr: integer;
  n: integer;

begin
  n := 0; inptr := 0; outptr := 0;
  loop
  select
    when n <= size => -- Spitzennotation :-(
      accept append(v: in integer) do
        b(inptr) := v;
      end append;
      n := n + 1; inptr := (inptr + 1) mod (size + 1);
    or
    when n > 0 =>
      accept take(v: out integer) do
        v := b(outptr);
    end take;
    n := n - 1; outptr := (outptr + 1) mod (size + 1);
```

```
      -- Pointerweitersetzen kann aussen sein
    end select;
    end loop;
  end BB;
```

Als letztes Beispiel simulieren wir ein monitorartiges wait/signal.

```
task Monitorartig is
  entry wait;
  entry signal;
end Monitorartig;

task body Monitorartig is
begin
  loop
    accept signal;
    select
      -- eine immer offene Alternative
        accept wait;
      else null;
      -- weiter wenn kein wartender Prozess
    end select;
  end loop;
end Monitorartig;
```

Die Idee ist, daß `Monitorartig` in der Anweisung `accept signal` auf den *signal-Aufruf* wartet. Beim Aufruf mit *signal* geht `Monitorartig` weiter und wenn eine Task an der offenen *wait-Alternative* wartet, wird diese Task befreit, sonst kehrt Monitorartig zum `accept signal` zurück und das *signal* verpuffte. Man beachte aber: eine Folge `Monitorartig.signal`, `Monitorartig.wait` kann dazu führen, daß nach dem *signal* noch das *wait* bedient wird, d.h. das *signal* hat dann eine Spur hinterlassen. Das widerspricht eigentlich der Semantik der *signal*-Operation im Monitor.

7 Threads

In fast allen Betriebssystemen ist die Prozeßgenerierung (in UNIX z. B. durch die fork/exec-Systemaufrufe), die Prozeßverwaltung und das Kontextumschalten teuer. Ferner ist das Prozeßkonzept häufig schlecht abgestimmt auf Mehrprozessoranlagen und auf klassische Client-Server-Anwendungen (vgl. Abbildung 7–1)

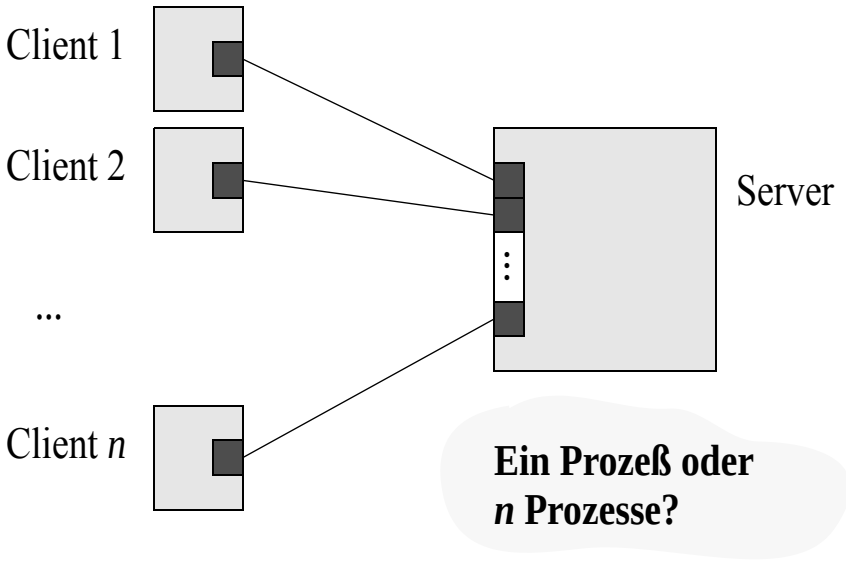


Abb. 7–1 *Client-Server Realisierung*

Ist der Server genau ein Prozeß, dann steht der Server, wenn einer der Clientwünsche einen Seitenfehler produziert („steht“ in dem Sinn, daß das

BS den Prozeß blockiert, bis die Seite von der Platte hereingebracht wurde).

Wird der Server durch $n > 1$ Prozesse realisiert, kann man nicht effizient einen Pool von Puffern verwalten, da der Zugriff auf gemeinsame Adressräume in der Regel verhindert wird.

Abhilfe bieten die sog. *Threads* („Fäden der Ausführung“). Threads sind im POSIX Standard vereinbart und werden u. a. als Bibliothek angeboten. Etwas allgemeiner spricht man auch von Pseudoprozessen oder leichtgewichtigen Prozessen (LWP, light weight processes).

Threads sind implementierbar als

- Programmiersprachenerweiterung
- BS-Kernerweiterung
- Benutzerbibliothek

Im folgenden stellen wir kurz die Distributed Computing Environment (DCE) Threads-Bribliothek vor.

Bei Threads steuert (synchronisiert) der Anwender seine eigenen Pseudoprozesse und regelt die Prozessorvergabe innerhalb eines eigenen Prozeßraums, d. h. er spielt selbst den Scheduler.

Threads kommunizieren miteinander über *shared variables* (innerhalb des eigenen Prozeßraums), deren Nutzung synchronisiert werden muß. Hierzu bieten die DCE Threads drei Möglichkeiten:

- Mutex Objekte
- Condition Variables
- die join Routine

Mutex Objekte

schützen die gemeinsamen Daten durch atomare Aufrufe.

```
pthread_mutex_lock(), pthread_mutex_trylock(),
pthread_mutex_unlock(), pthread_mutex_init(),
pthread_mutex_destroy()
```

Ein Mutex (ähnlich einem Semaphor) nimmt zwei Zustände an: frei und gesperrt. Ein Thread, der ~`lock` für ein bereits gesperrtes Mutex aufruft, wird blockiert. Er läuft weiter, wenn das Mutex mit ~`trylock` getestet wird.

Jedes Mutex muß initialisiert werden; dabei wird der Typ angegeben:

- fast (default, effizient)
- recursive
- non-recursive

Im Falle von **fast** erfolgt keine Kontrolle des Aufrufes/Besitzers der Sperre, daher kommt es zu einer Verklemmung bei erneutem Aufruf des selben Mutex!

Im Falle von **recursive** zählt ein Zähler in der Sperre hoch, wenn der Besitzer es erneut aufruft, der ursprüngliche Besitzerthread (der durchging und erstmals die Sperre einrichtete) wird aber nicht blockiert trotz rekursivem Aufruf. Die Freigabe erfordert die gleiche Anzahl von `unlock()` Aufrufen. Es wird ein Fehler signalisiert, wenn ein anderer Thread `unlock()` aufruft.

Im Fall von **non-recursive** erzeugt ein Thread einen Fehler, wenn er versucht, ein von ihm gesperrtes Mutex erneut zu sperren.

Mutex-Objekte stellen keinen Weckmechanismus bereit. Dazu benötigt man zusätzlich

Condition Variables

für eine anonyme Synchronisation mit

```
pthread_cond_init(), ~_wait(), ~_signal(),
~_broadcast(), ~_timedwait(), ~_destroy().
```

Der Anwender muß mit einem zur Condition Variablen gehörenden Mutex die Variable zunächst selbst sperren.

Dann wird er eine Bedingung (ein Prädikat) prüfen

```
while (! Prädikat) {...}
```

und wenn nicht erfüllt, mittels

```
pthread_cond_wait(&cond_var, &cond_mutex)
```

in `cond_var` auf ein `pthread_cond_signal()` oder `pthread_cond_broadcast()` warten. Als Nebeneffekt wird `cond_mutex` entperrt!

Analog sollte `~_signal(&cv)` von einem `~_lock(&m)` und `~_unlock(&m)` umgeben sein. `Signal` weckt nur *einen* Thread auf, mit `~_broadcast()` werden *alle* wartenden Threads geweckt.

Bei `signal` und bei `broadcast` entscheidet die Priorität des geweckten Threads, wann er die Kontrolle übernimmt. Wegen der fehlenden direkten Übergabe der Kontrolle nach `signal` sollte `wait` in eine Schleife für den Test des Prädikats eingebaut werden.

Fazit: DCE Threads mit Condition Variables sind wie Monitore mit viel Anwenderverantwortung.

Join

`pthread_join()` läßt den Aufrufer auf die Beendigung des benannten Threads warten. Der Aufruf eignet sich gut für Abräumroutinen.

Ferner gibt es in UNIX sog. *Jacket-Routinen* um die UNIX-Systemaufrufe herum, die einen Prozeß (und damit alle seine Threads) blockieren würden: *read, write, open, socket, send, recv, ...*

Ein etwas kitzliger Punkt ist, daß die Thread-Synchronisierung nur mit *reentrant* Routinen funktioniert. Man prüfe daher immer beim Einsatz einer Bibliothek oder einer speziellen Routine in einer Anwendung, die mittels Threads arbeitet, ob die verwendete Bibliothek „*thread-safe*“ ist.

Auch relativ kompliziert ist die Frage, wie mit externen Signalen, Ausnahmen, Prioritäten, Verklemmungen, Time-Outs zu verfahren ist. Beziehen sich die Ereignisse auf den umgebenden Prozeß, einzelne Threads, ...?

Insgesamt gelten Threads als beste Möglichkeit, unter UNIX selbst anspruchsvolle, nebenläufige Anwendungen zu schreiben.

8 Betriebsmittelverwaltung

8.1 Verklemmungen und zeitabhängige Fehler

Die Behandlung von Inkonsistenzen und potentiellen Verklemmungen, deren Auftreten von zufälligen zeitlichen Konstellationen abhängt, schließt sich an die Synchronisationsverfahren von oben an.

Eine Inkonsistenz von Daten ergibt sich, wenn nebenläufige Prozesse Werte produzieren, die so nicht als Ergebnisse einer sequentiellen Ablauffolge entstanden wären (vgl. auch Serialisierung für parallele Transaktionen in Datenbanken).

Beispiel 8–1

Prozeß A	**Prozeß B**
lade Wert aus *i* nach Register *j*	
	lade Wert aus *i* nach Register *k*
add R_j, '1'	
	add R_k, '1'
speichere R_j nach *i*	
	speichere R_k nach *i*

Bei einer Verklemmung wartet eine Gruppe von Prozessen auf den Eintritt von Bedingungen, die nur durch Prozesse dieser Gruppe hergestellt werden können.

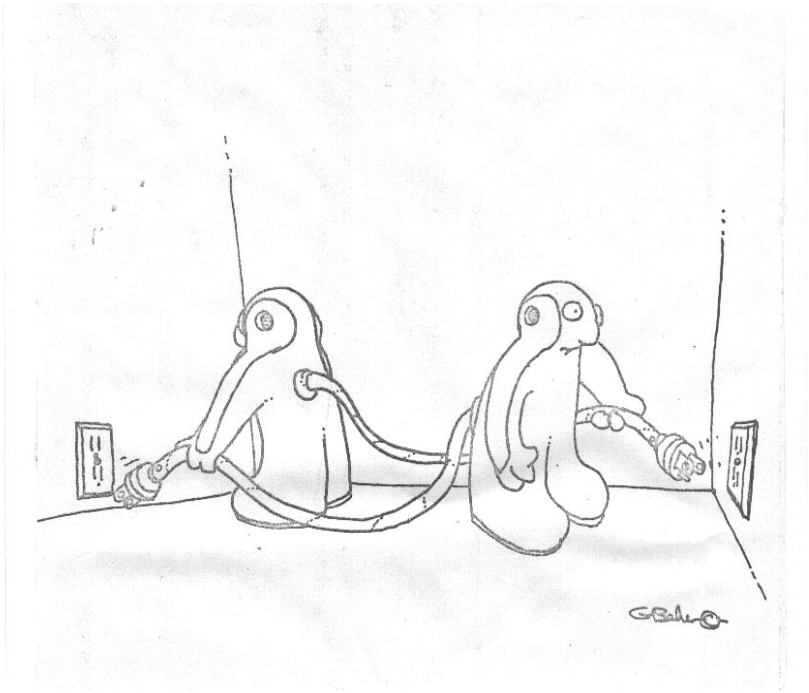


Abb. 8–1 *Verklemmt?*

Prozeß A	Prozeß B
gettape, gettape	
	gettape, gettape
gettape	
	gettape
...	...
puttape, puttape, puttape	puttape, puttape, puttape

Tab. 8–1 *Verschränkte Betriebsmittelvergabe*

Das folgende Beispiel beschreibt eine 4-Bänderverwaltung, bei der jeder Prozeß drei Bänder möchte.

Beispiel 8–2

Das Vier-Bänderbeispiel zeigt sehr deutlich, daß die inkrementelle (stückweise) Vergabe der Betriebsmittel der wesentliche Verklemmungsgrund ist. In einer graphischen Darstellung (vgl. Abb. 8–2) sieht man auch, wie sich der Systemfortschritt (eine monoton wachsende Funktion!) in ein konkaves Gebiet bewegt, das von der verbotenen Zone umgeben ist. Aus dieser Höhle kann er sich nicht mehr selbst befreien.

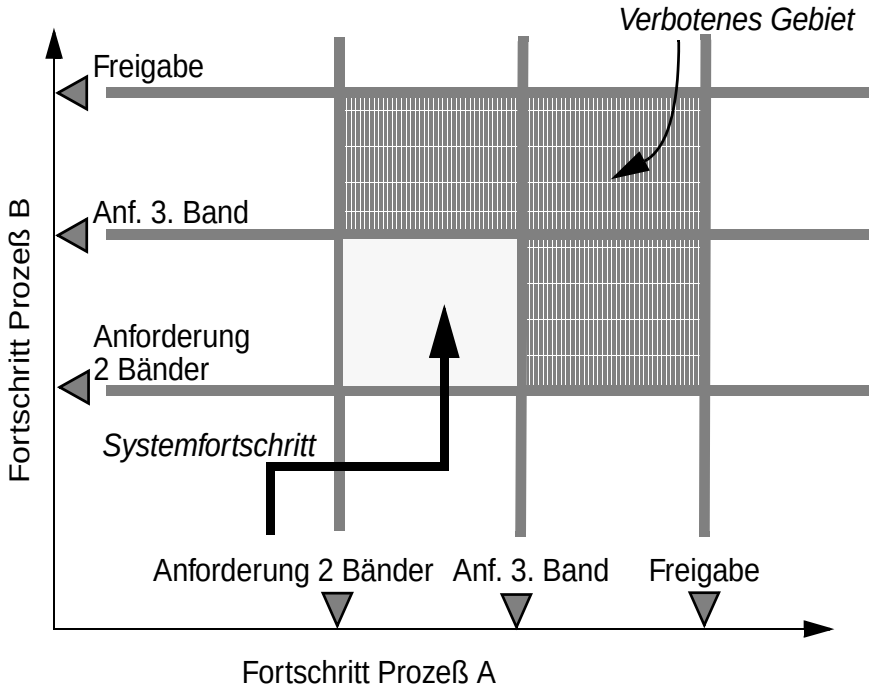


Abb. 8–2 *Inkrementelle Betriebsmittelzuteilung*

Das Problem verschärft sich durch den Zeitfaktor, d.h. der Fehler tritt abhängig von der Reihenfolge der Bearbeitung auf und ist i. a. zufällig und nicht beliebig reproduzierbar. Dadurch ist kein systematisches Austesten möglich.

Verklemmungen können leicht durch geschachtelte kritische Abschnitte entstehen, wie in Abbildung 8–3 gezeigt. Dies ließe sich auch durch Verwendung eines Monitors nicht vermeiden.

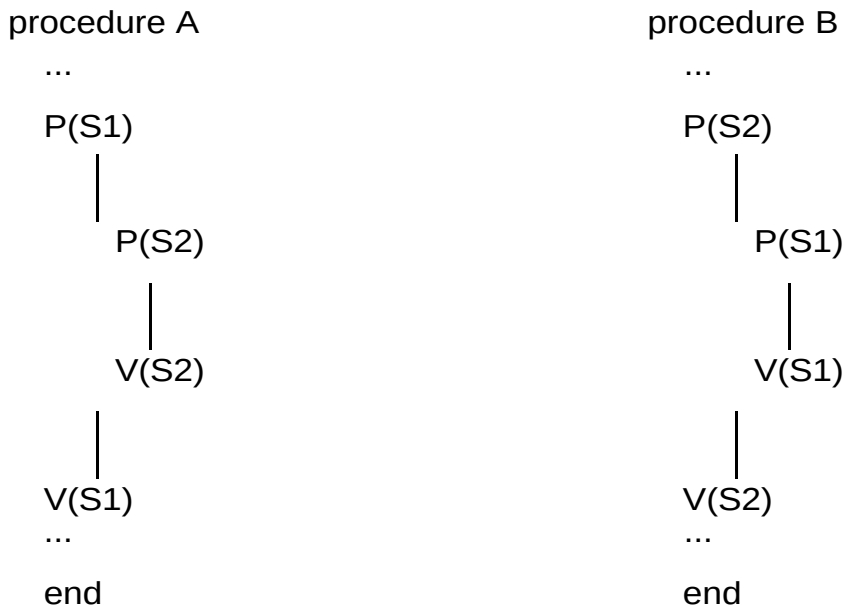


Abb. 8–3 *Geschachtelter kritischer Abschnitt*

Genauso kann dies bei einer zyklischen Auftragsrelation passieren, wobei in Abbildung 8–4 eine gerichtete Kante einem erteilten Auftrag entspricht.

Abb. 8–4 *Zyklische Auftragsrelation*

Die programmtechnische Ausgestaltung in einem synchronen Botschaftenkonzept sähe wie folgt aus.

```
procedure A;
begin
  repeat
    send Auftrag, 'A' to 'C';
    receive Bestätigung from 'C';
    ...
  forever
end;

procedure C;
var Sender: ('A', 'D');
begin
  repeat
    receive Auftrag1, Sender from Sender2;
    case Sender do
    'A': begin
         send Auftrag2, 'C' to 'D';
         receive Bestätigung2 from 'D';
         ...
         send Bestätigung1 to 'A'
       end;
    'D': begin
         ...
         send Bestätigung2 to 'D'
       end
    end {case}
  forever
end {C}; {Prozedur D analog}
```

Als praktisches Beispiel sei ein einfaches Spooling-System genannt (vgl. Abbildung 8–5). Es kann (so unter OS/360 passiert) zu einer Verklemmung kommen, wenn die Platte voll ist mit eingelesenen Programmen, Eingabedaten und unvollständigen Ausgabedaten von noch nicht abgeschlossenen Programmen.

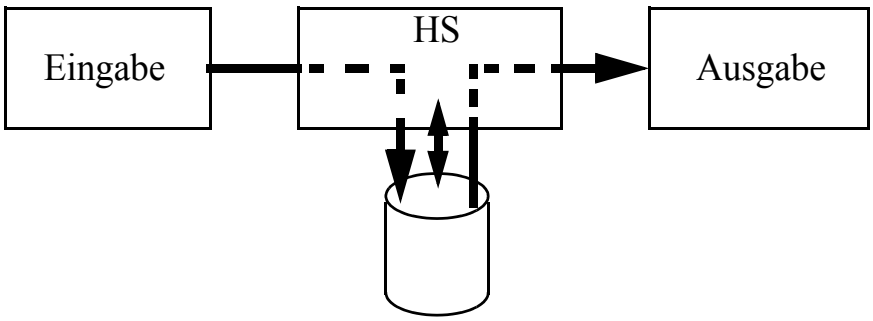


Abb. 8–5 *Verklemmung beim Spooling bei voller Platte*

Genauso kann ein fehlerhaftes Benutzerprogramm, z. B. wenn ein V(S) vergessen wurde, die ihm zugeteilten Betriebsmittel nicht zurückgeben. Nach Holt (1972) sieht das kürzeste PL/I Killerprogramm wie folgt aus:

```
REVENGE: PROCEDURE OPTIONS(MAIN, TASK);
  WAIT(EVENT);
END REVENGE;
```

Wenden wir uns jetzt einer etwas systematischeren Betrachtung von Verklemmungen zu. Die Hypothese lautet, daß die folgenden vier Bedingungen für die Entstehung einer Verklemmung hinreichend und notwendig sind.

1. exklusiver Zugriff (mutual exclusion)
2. inkrementelle Zuteilung (wait for)
3. kein Entzug (no preemption)
4. zyklisches Warten (circular wait)

Am Bild der folgenden Verklemmung (Manhatten Grid) ist dies besonders gut zu veranschaulichen.

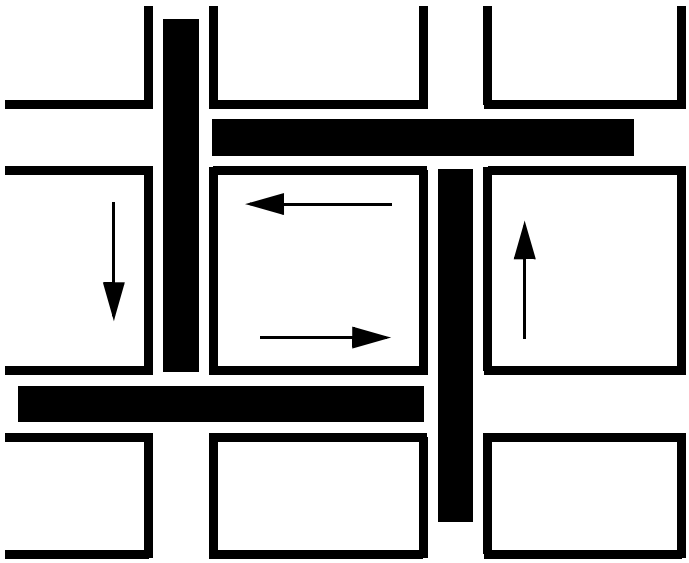


Abb. 8–6 *Verklemmung in Manhatten*

Die dicken schwarzen Balken repräsentieren die einzelnen Fahrzeugkolonnen. Könnten die Fahrzeuge übereinander fahren (kein exklusiver Zugriff) ließe sich die Blockade der Kreuzungen lösen. Genauso, wenn man kurzfristig ein Auto beiseitestellen könnte. Das zyklische Warten ist offensichtlich und die Blockade kam durch die inkrementelle Zuteilung zustande, d.h. könnte man erst in die Kreuzung einfahren, wenn man sicher ganz durchkäme, ließen sich Verklemmungen (engl. *Deadlocks*) vermeiden.

Generell gilt: Wenn die Bedingungen 1 - 3 erfüllt sind, ist Deadlock möglich.

Wird die Bedingung 4 erkannt und vermieden, kann kein Deadlock entstehen, wir sprechen dann von *Verklemmungsvermeidung (deadlock avoidance)* .

Sorgt man dafür, daß Bedingung 1 - 3 nicht erfüllt sind, dann läßt sich *Verklemmungsverhinderung* (*deadlock prevention*) erreichen.

Kann man mit Entzug arbeiten (Bed. 3 nicht erfüllt), dann gibt es die Aufgabe, Verklemmungen zu entdecken und zu beheben (*deadlock detection and recovery*).

In der Praxis muß man abwägen, was realistisch machbar ist. Häufig ist ein exklusiver Zugriff (etwa auf einen Plattensektor) zwingend notwendig.

Die Bedingung 2 (nur vollständige Zuteilung) ist machbar, führt aber zu einer sehr schlechten Auslastung. Die Bedingung 3 führt zur Rückgabe belegter Betriebsmittel bei nichterfüllbaren Wünschen. Das geht nicht bei allen Betriebsmitteln (HS ja, Drucker nein).

Die Bedingung 4 läßt sich umgehen, wenn man die Betriebsmittel in eine (willkürliche) lineare Reihenfolge bringt. Prozesse, die schon Betriebsmittel haben, dürfen nur solche anfordern, die in dieser Reihenfolge den schon belegten folgen. Damit sind keine Zyklen möglich, ggf. muß aber lange gewartet werden, bis der Prozeß mit den höchsten Betriebsmittelnummern fertig geworden ist und seine Mittel freigibt.

Beispiel 8–3

Die Ordnung laute:

HS1, HS2, Disk1, Disk2, Tape, Printer

Dann ist die folgende Anforderungsfolge unzulässig:

A: Tape, HS1, Disk1
B: Disk2, HS2, Printer, Tape

dagegen wären die beiden Folgen

A: HS1, Disk2, Tape
B: HS2, Disk2, Tape, Printer

legal. Häufig ist die Implementierung (wie soll man Plattenbelegungen ordnen, wie HS-Zuteilungen?) aber unklar.

Zuletzt sei darauf hingewiesen, daß im Falle der Aufhebung der Bed. 3 meist der Prozeß mit der niedrigsten Priorität seine Mittel zurückgeben

muß. Das kann zum Aushungern dieses Prozesses führen, ggf. zum Zurücksetzen mehrerer Prozesse. Hier hilft das Bild vom Fluß mit einer Kette von Steinen, auf denen immer nur einer stehen kann (exklusiver Zugriff).

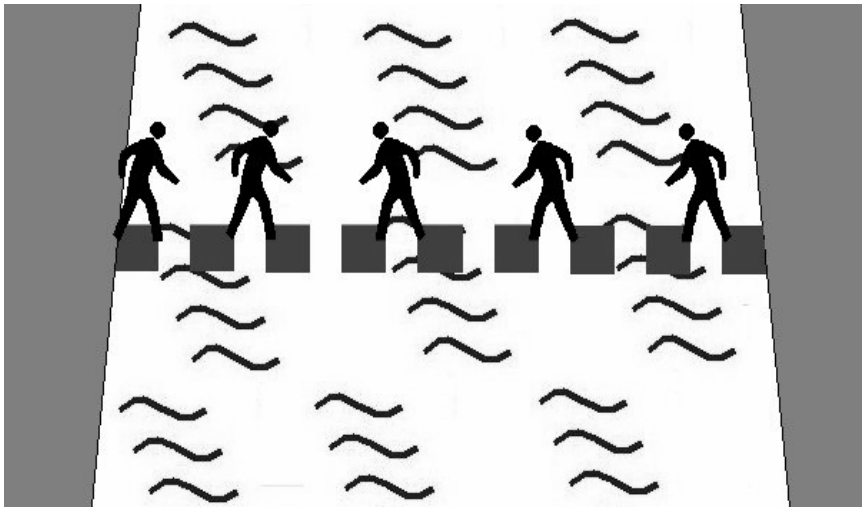


Abb. 8–7 *Verklemmung mit Aushungern*

Es folgt nun ein graphentheoretisches Modell nach Holt (1972) zur Beschreibung von Verklemmungen.

8.2 Verklemmungsentdeckung

Ein Prozeßsystem ist ein Paar (Σ, Π) mit

Σ: Menge {*S, T, U, V, W,* ...} von Systemzuständen
Π: Menge $\{P_1, P_2, ...\}$ von betrachteten Prozessen.

Die graphische Darstellung dieses Prozeßsystems ist ein Graph, bei dem Σ die Knoten und die gerichteten Kanten Übergänge sind, wobei die Kantenbeschriftung mit $p \in \Pi$ den verursachenden Prozeß benennt.

Als Schreibweise führt man

$$\boldsymbol{S} \xrightarrow{\boldsymbol{i}} \boldsymbol{T}$$

ein mit der Sprechweise „P_i führt Zustand S in den Folgezustand T über“. Wie üblich schreibt man für

$$S \xrightarrow{i} T, \quad T \xrightarrow{j} U, \quad \cdots \quad V, \; V \xrightarrow{k} W,$$

kurz

$$S \xrightarrow{*} W$$

und spricht davon, daß S **in null oder mehr Schritten** in W übergeht (also immer $S \xrightarrow{*} S$). Ferner ist nichtdeterministisches Verhalten zugelassen, etwa im Beispielprozeßsystem unten mit $S \xrightarrow{1} T$ und $S \xrightarrow{1} U$. Speziell gilt dort auch $S \xrightarrow{*} U$, weil sowohl $S \xrightarrow{1} T$, $T \xrightarrow{2} V$, $V \xrightarrow{1} U$, also auch direkt $S \xrightarrow{1} U$.

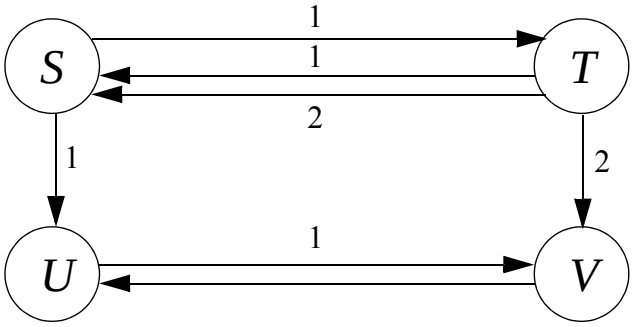


Abb. 8–8 *Zustandsübergangsgraph nach Holt*

Die folgenden Definitionen erlauben es nun, präziser über Verklemmungen zu sprechen.

Definitionen

a) P_i ist in S **blockiert**, wenn es kein T gibt, mit $S \xrightarrow{i} T$.

b) P_i ist in S **verklemmt**, wenn P_i für alle Übergänge $S \xrightarrow{*} T$ in T blockiert ist.

c) ein Zustand S ist **sicher**, wenn für T mit $S \xrightarrow{*} T$, T für keinen Prozeß ein Verklemmungszustand ist.

d) S ist **totaler Verklemmungszustand**, wenn alle P_i in S verklemmt sind.

e) ein **System ist sicher**, wenn es einen sicheren Zustand gibt.

Verbal ausgedrückt gilt also im Fall (a), daß ein blockierter Prozeß nicht selbstständig seinen Zustand ändern kann. Das entspricht z.B. einem Prozeß, der mit $P(S)$ in einem Semaphor S gefangen wurde.

Im Fall (b) ist die Situation ernster, denn egal wohin ihn andere Prozesse „mitnehmen“, kann er nie wieder selbstständig agieren. Trotzdem kann das System insgesamt lebendig sein.

Ein Zustand ist sicher, wenn sich aus ihm keine Verklemmungszustände entwickeln lassen. In einem zusammenhängenden System (dem Normalfall) genügt es dann zu prüfen, ob der Startzustand sicher ist. Totale Verklemmungszustände erkennt man leicht im Graphen. Sie sind Knoten, aus denen keine Kanten herausgehen (man spricht von Senken).

Im Beispiel aus Abbildung 8–8 gilt also: P_2 ist in S blockiert, aber nicht verklemmt; P_2 ist in U und in V verklemmt (also auch blockiert). Das System hat keinen totalen Verklemmungszustand.

Übung 8–1

Wenn in einem Zustand x kein Prozeß verklemmt ist, ist der Zustand x auch sicher (vgl. Punkt (c) oben). Ja oder nein?

Die Antwort lautet nein und läßt sich durch ein Gegenbeispiel beweisen.

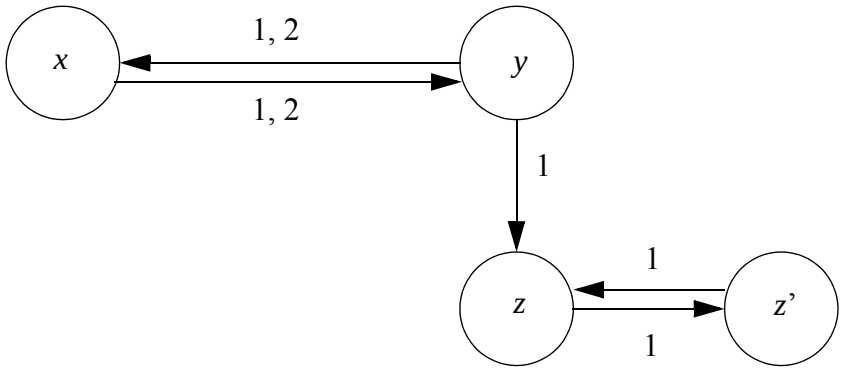


Abb. 8–9 *Nicht verklemmt und doch unsicher*

P_1 und P_2 sind weder in x noch in y verklemmt. Aber x ist nicht sicher, weil $x \xrightarrow{*} z$ und z ein Verklemmungszustand für P_2 ist.

Wir geben nun einen Graphen zur Darstellung der gegenläufigen Schachtelung kritischer Abschnitte durch zwei Prozesse *A* und *B* an (nach Holt 1972).

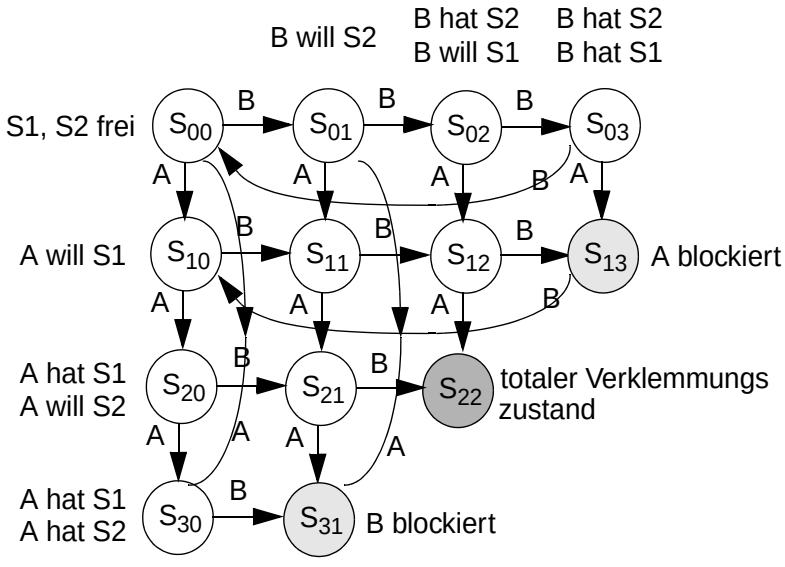


Abb. 8–10 *Schachtelung kritischer Abschnitte nach Holt*

Am Beispiel des nachfolgenden Graphen, der die Betriebsmittelvergabe zweier Exemplare eines Betriebsmittels durch zwei Prozesse darstellt, sieht man bereits, wie unhandlich diese Methode wird.

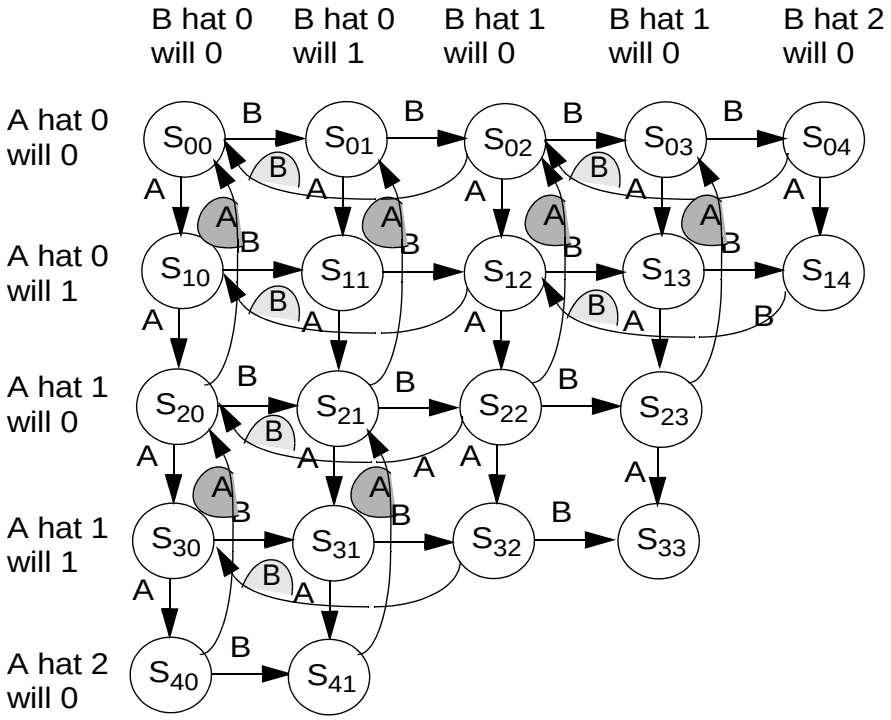


Abb. 8–11 *Akkumulierende Belegung nach Holt*

Der Zustandsgraph nach Holt erlaubt demnach, festzustellen ob

- ein Prozeß P_i blockiert (trivial) oder verklemmt ist
- das System, gegeben durch n Prozesse und deren mögliche Belegungsabsichten, sicher ist.

Damit lassen sich momentane und potentielle Verklemmungen entdecken. Es sind aber keine direkt ableitbaren allgemeinen und effizienten Verfahren hierfür bekannt.

Man erweitert daher die Darstellung zum Betriebsmittelzuteilungsgraphen (resource allocation graph, Holt 1972).

8.2.1 Betriebsmittelzuteilungsgraph nach Holt

Die Grundidee besteht darin, einen bipartiten Graphen $G = (V, E)$ zu bilden, wobei $V = R \cup P$, $R = \{r_1, r_2, ..., r_m\}$ Betriebsmittel (resources), $P = \{p_1, p_2, ..., p_n\}$ Prozesse als Knoten.

Ein Graph $G = (V, E)$ ist *bipartite* gdw. $V = X \cup Y, X \cap Y = \varnothing$ und $\forall (u, v) \in E$: $(u \in X \wedge v \in Y) \vee (u \in Y \wedge v \in X)$.

Demnach besteht die Knotenmenge aus zwei disjunkten Teilmengen und alle Kanten gehen von einer Menge in die andere.

Die Kanten hier werden mit der folgenden Bedeutung belegt.

- $(p_i, r_j) \in E \Leftrightarrow$ Prozeß p_i beansprucht Betriebsmittel r_j und wartet darauf
- $(r_j, p_i) \in E \Leftrightarrow$ Betriebsmittel r_j ist Prozeß p_i zugeteilt.

Bildlich stellen wir Prozesse als runde Knoten dar. Betriebsmittel bilden Rechtecke, Punkte im Inneren sind die Exemplare.

a) Beantragt jetzt ein Prozeß p_i die Zuteilung einer Resource von r_j, dann tragen wir eine Kante von p_i **an** das Kästchen von r_j an.
b) Ist die Zuteilung erfolgt, löschen wir die Kante aus (a) und tragen eine Kante (r_j, p_i) aus dem Kästchen heraus ein.

Beispiel 8–4

Wir betrachten drei Prozesse $P = \{p_1, p_2, p_3\}$ und vier Betriebsmittel $R = \{r_1, r_2, r_3, r_4\}$. Von r_1 und r_3 existieren je eine Einheit, r_2 hat zwei und r_4 drei Einheiten.

Die Anmeldungs- und Belegungshistorie sei wie folgt:

$$E = \{(p_1, r_1), (p_2, r_3), (r_1, p_2), (r_2, p_2), (r_2, p_1), (r_3, p_3)\}$$

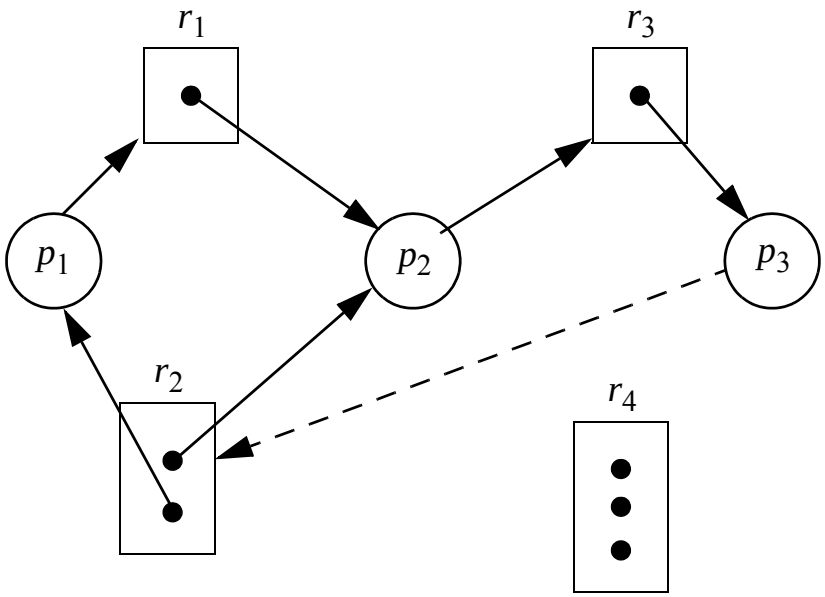


Abb. 8–12 *Betriebsmittelzuteilungsgraph nach Holt*

Ohne die gestrichelte Kante ist dies wie folgt zu interpretieren.

- p_1 hat eine Einheit von r_2 und will r_1.
- p_2 hat eine Einheit von r_2 und will sowohl r_1 als auch r_3
- p_3 hat eine Einheit von r_3.

Momentan ist keine Verklemmung eingetreten, obwohl p_1 und p_2 blokkiert sind. Gibt p_3 sein Betriebsmittel frei, kann p_2 fertigarbeiten und danach seine Resource r_1 freigeben, womit auch p_1 fertigarbeiten kann.

Die Situation ändert sich, wenn auch die Forderung aus der gestrichelten Kante bestünde. Jetzt ist eine Verklemmung eingetreten.

Die Behauptungen lauten nun:

a) Graph hat keine Zyklen $\Rightarrow$ kein Prozeß verklemmt
b) Graph hat Zyklus $\Rightarrow$ Prozeß kann u. U. verklemmt sein, muß es aber nicht notwendigerweise sein.

c) Graph hat Zyklus und jedes Betriebsmittel im Zyklus hat genau eine Einheit ⇒ jeder Prozeß im Zyklus ist verklemmt.

Im folgenden Beispiel (Abb. 8–13) ist eine solche Verklemmung

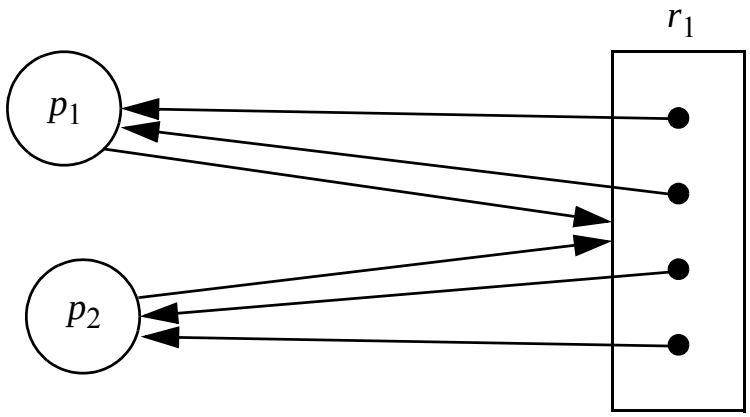


Abb. 8–13 *Beispiel 4 Bänder und zwei Prozesse*

eingetreten. Im folgenden Beispiel (Abb. 8–14) jedoch nicht, obwohl auch ein Zyklus existiert. Die Verklemmung läßt sich hier auflösen, weil p_4 seine Einheit von r_2 an p_3 abgeben kann.

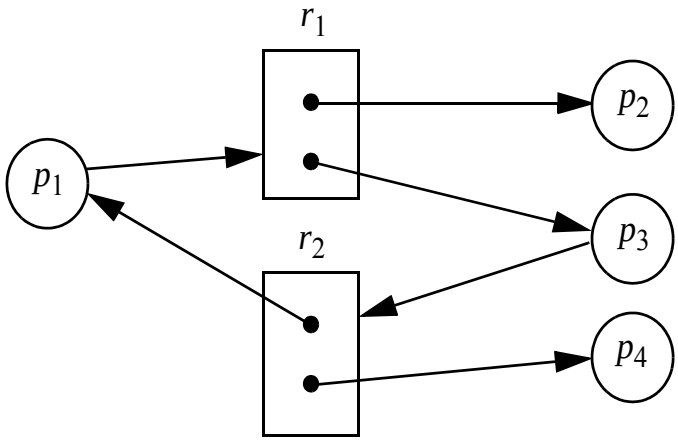


Abb. 8–14 *Zyklus ohne Verklemmung*

Der Zusammenhang mit dem Zustandsgraphen ist nun wie folgt: ein Prozeß p_i blockiert, wenn es eine Kante (p_i, r_j) gibt und r_j keine freie(n) Einheit(en) mehr hat.

Daraus läßt sich jetzt ein effizienter Algorithmus (auf der Basis der Zyklenanalyse) zur *Deadlockerkennung* ableiten.

Wir definieren die *Reduzierung eines Betriebsmittelzuteilungsgraphen* durch einen Prozeß p_i, der nicht isoliert und nicht blockiert ist, durch das Entfernen aller Kanten (p_i, r_j) und (r_j, p_i).

Ein Graph ist *vollständig reduzierbar*, wenn alle Kanten so entfernt werden können.

Satz 8–1

> Ein Prozeß p_i ist nicht im Graphen S verklemmt, gdw. es eine Reduzierungsfolge für S gibt, so daß p_i nicht blockiert.

Korollar

1. S ist in keinem Verklemmungszustand $\Leftrightarrow$ S ist vollständig reduzierbar.
2. alle Reduzierungsfolgen in nicht vollständig reduzierbaren Graphen führen zum selben Zustand, d.h. es ist kein Backtrack nötig.

Hinter dem Satz und seiner Schlußfolgerung steckt die Strategie, daß eine Betriebsmittelzuteilung (bei bekannten Wünschen) dadurch zu einem „guten Ende“ gebracht werden kann, daß man Zug um Zug Prozesse, die alle ihre Wünsche befriedigt bekommen können, zu Ende arbeitet und ihre Betriebsmittel zurückgibt. In welcher Reihenfolge dies geschieht, ist gleichgültig. Auch im Fall des Mißerfolgs kann man sicher sein, daß die Blockade nicht durch eine andere Reihenfolge der Abarbeitung zu vermeiden gewesen wäre.

Allerdings gilt dieser Satz nur für „rückgebbare“ Betriebsmittel. Er läßt sich nicht anwenden auf Fälle, bei denen Betriebsmittel „verbraucht“ werden, z. B. Signale die eintreffen oder einmalbeschreibbare Datenträger, etc.

8.2.2 Deadlock-Erkennungsalgorithmus nach Holt

Im wesentlichen geht es um eine Art Buchhaltung über Belegungen und Wünsche. Dazu führt man eine sog. *Hat-Matrix* und eine *Will-Zusätzlich-Matrix*. Beide sind $n \times m$ Matrizen, wenn n die Anzahl der Prozesse und m die Anzahl der Betriebsmitteltypen ist.

Um eine schnellere Laufzeit zu ermöglichen, führt man zusätzlich zwei Datenstrukturen ein.

Zum einen definiert man einen *Wartezähler* als Vektor mit n Elementen, wobei **Wartezähler**[i] angibt, auf wieviele Betriebsmitteltypen Prozeß p_i wartet, d.h. wieviele er verlangt, aber nicht bekommen kann. Offensichtlich sind Prozesse, deren Zähler auf 0 gesunken ist, Kandidaten, die fertigarbeiten und in der Folge Betriebsmittel freigeben können.

Zum zweiten definiert man je Betriebsmittel eine Liste der wartenden Prozesse, geordnet nach Anzahl der verlangten Einheiten. Dies könnte durch einen Vektor der Dimension m geschehen, der Startzeiger auf lineare verkettete Listen enthält, deren Knoten jeweils die Prozeßnummer und die Anzahl der verlangten Einheiten enthalten.

```
while unbehandelter und reduzierbarer Prozeß P übrig
      (Wartezähler = 0) do {1}
    while Betriebsmittel R im Besitz von P übrig do {2}
    begin
      erhöhe verfügbare Anzahl der Einheiten
      von R um die von P besessene Anzahl; {3}
      while Prozeß Q übrig, dessen Anspruch auf R
      vollständig befriedigt werden kann do {4}
      begin
        Wartezähler[Q] := Wartezähler[Q] - 1; {5}
        if Wartezähler[Q] = 0 {6}
        then nimm Q in Liste als unbehandelten
        und reduzierbaren Prozeß auf
      end {while Prozeß Q übrig}
    end
```

Ist am Ende die Prozeßliste leer, dann ist der Graph vollständig reduziert und das System nicht verklemmt.

Zur **Laufzeit** stellen wir fest, daß *{1}* eine Iteration je Prozeß bedeutet, d.h. der Aufwand ist proportional zu n.

Der Aufwand aus der Schleife *{2}* ist proportional zur Anzahl der maximal belegten Betriebsmittel, also m.

Der Schritt *{4}* benötigt kein Suchen, weil die Liste geordnet ist.

Insgesamt macht man sich leicht klar, daß das Verfahren nach Holt in $O(m \cdot n)$ Schritten läuft, was dem einmaligen Durchlaufen der *Hat-Matrix* entspricht.

Beispiel 8–5

Betriebsmittel		r_1	r_2	r_3	r_4
insgesamt		**2**	**5**	**3**	**2**
Hat	p_1		2	1	1
	p_2		1	1	1
	p_3	1			
	p_4	1	1	1	
Will-Zusätzlich	p_1				
	p_2		1	1	
	p_3		2	3	1
	p_4		1	1	1
		p_1	p_2	p_3	p_4
Wartezähler		**0**	**2**	**3**	**3**

Tab. 8–2 *Verklemmungsentdeckung nach Holt*

Die zugehörige Betriebsmittelliste sieht wie folgt aus.

Betriebs-mitteltyp	Listen der wartenden Prozesse (sortiert nach benötigten Einheiten)	momentan verfügbar
r_1		0
r_2	(p_2, 1); (p_4, 1); (p_3, 2)	1
r_3	(p_2, 1); (p_4, 1); (p_3, 3)	0
r_4	(p_3, 1); (p_4, 1)	0

Tab. 8–3 *Betriebsmittelliste*

Durch Nachverfolgen der Abarbeitung gemäß obigem Verfahren wird man feststellen, daß das System nicht verklemmt ist. Der folgende Betriebsmittelzuteilungsgraph in Abbildung 8–15 zeigt die Situation nach Abarbeitung von Prozeß p_1.

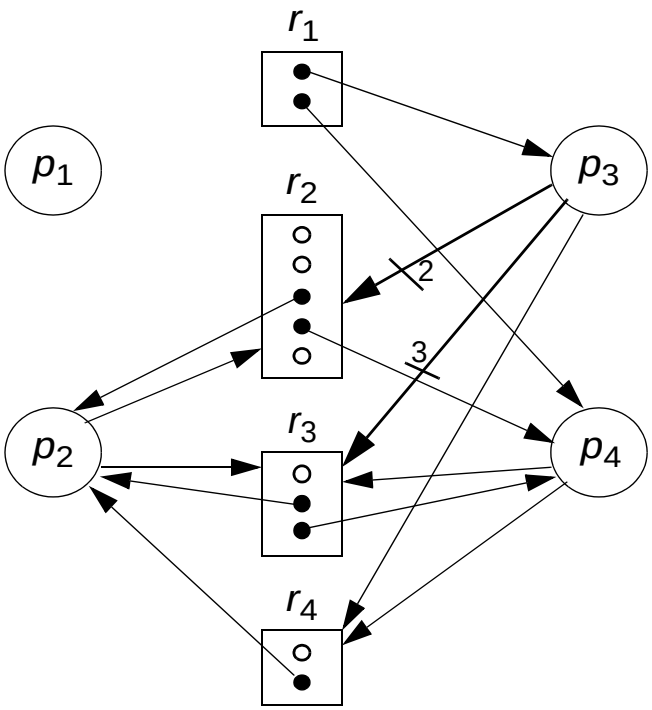


Abb. 8–15 *Betriebsmittelzuteilung und Anforderungen*

Ganz analog funktioniert das *Verfahren von Shoshani und Coffman* aus dem Jahr 1970. Es benötigt $O(m \cdot n^2)$ Schritte.

Eingabe

Einen Integervektor `Verfügbar[1..m]` zur Angabe der momentan verfügbaren Betriebsmittel

Eine Integermatrix `Hat[1..n, 1..m]` für die einem Prozeß i ($1 \le i \le n$) zugeteilten Einheiten des Betriebsmittels j ($1 \le j \le m$).

Eine Integermatrix `Will[1..n, 1..m]` für die von einem Prozeß i ($1 \le i \le n$) zusätzlich benötigten Einheiten des Betriebsmittels j ($1 \le j \le m$).

Methode

1. Markiere alle Zeilen i in `Hat` für die gilt: `Hat[i, j] = 0` $\forall\ 1 \le j \le m$;
2. Finde unmarkierte Zeile i so daß gilt:
   `Will[i, j] <= Verfügbar[j]` $\forall\ 1 \le j \le m$;
   Für gefundene Zeile i setze
   `Verfügbar[j] := Verfügbar[j] + Hat[i, j]`
   $\forall\ 1 \le j \le m$; markiere Zeile i in `Hat`; finde nächste Zeile;
3. Bleibt eine unmarkierte Zeile übrig, ist das System verklemmt, sonst verklemmungsfrei. Eine fehlende Markierung zeigt den Verursacherprozeß an.

Wie im Fall des Verfahrens von Holt prüft der Algorithmus, ob es eine Abarbeitungsreihenfolge gibt, so daß alle Prozesse unter Rückgabe ihrer Betriebsmittel zu Ende gebracht werden können. Beim Durchrechnen von Beispielen achte man darauf, daß ein Prozeß, dessen Anspruch momentan befriedigt werden kann (Will-Zeile kleiner als Verfügbar-Vektor) und damit als abgearbeitet ausscheidet, nur die Hat-Zeile zurückgibt. Man tut also so, als habe der Prozeß momentan die Will-Ansprüche gehabt und auch schon zurückgegeben und spart sich so das Addieren und Subtrahieren im Verfügbar-Vektor.

Spezialfall einer Einheit je Betriebsmitteltyp

Verklemmungsfreiheit prüft man einfach dadurch, daß man aus dem Betriebsmittelzuteilungsgraphen von Holt die Betriebsmittelknoten und die Belegungspfeile entfernt. Dadurch entsteht ein Prozeßwartegraph, den man auf Zyklenfreiheit prüft (geht in $O(n^2)$ Schritten).

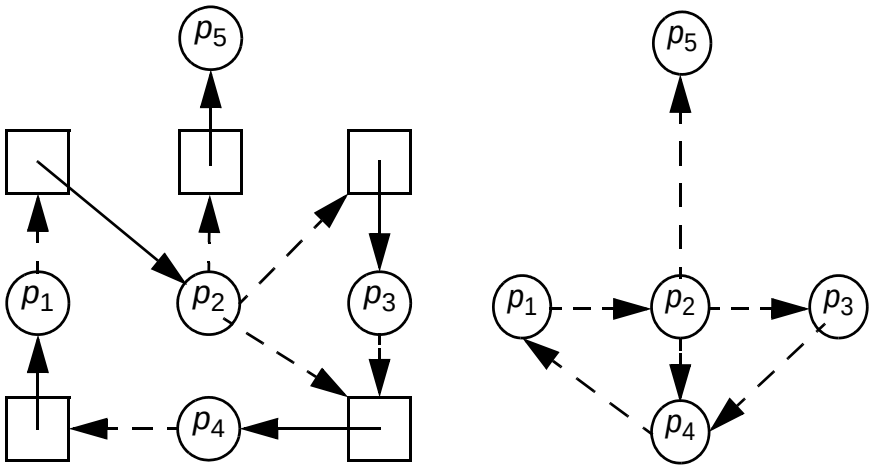


Abb. 8–16 *Vom Betriebsmittelzuteilungsgraph zum Wartegraph*

8.2.3 Maßnahmen bei Entdeckung einer Verklemmung

Man überlegt sich am Beispiel der Flußüberquerung auf der Kette von Steinen leicht die Konsequenzen einer Verklemmungsbeseitigung.

- gleichzeitige Benutzung gestatten (Bed. 1 ausschalten)
- Abbruch eines Prozesses (Bed. 4, zykl. Warten, ausschalt.)
- Entzug (teilweise oder ganz) von Betriebsmitteln (Bed. 3)

Beim Entzug entstehen folgende Fragen:

1. Wer ist das Opfer?
2. Wie weit zurücksetzen?
3. Kann es zum Aushungern kommen?

Zu (1): Prioritäten und der Fortschritt zählen, z. B. 998 von 1000 Steinen überquert; die Anzahl der Opfer zählt, z.B. p_1 und p_2 treffen sich in der Mitte, aber p_1 hat zehn Leute hinter sich.

Zu (2): Alternativen sind *totales Zurücksetzen* (beim Überqueren des Dönchebachs entsteht ein Deadlock, dann Zurücksetzen zum Reiseanfang in München?), *partielles Rücksetzen* (bis zum Ufer, bis zu Ausweichsteinen?).

Zu (3): Hier muß man die Anzahl der Rücksetzungen mitzählen.

Generell kann Rücksetzen auch scheitern, wenn es gerade das Betriebsmittel benötigt, das die Verklemmung hervorrief, z. B. ein Mangel an freiem Plattenplatz.

8.3 Verklemmungsvermeidung

8.3.1 Der Banker's Algorithmus

Die Verfahren gehen auf Dijkstra (1965) und Habermann (1969) zurück und setzen voraus, daß die maximalen Anforderungen aller Prozesse bekannt sind.

Die Idee ist, nur solche Zuteilungen zu genehmigen, die das System in einem sicheren Zustand belassen, wobei der Sicherheitsbegriff anders ist als bei Holt[1]. Hier bedeutet Sicherheit, daß ein System nur dann sicher ist, wenn es eine Reihenfolge gibt, mit der alle Prozesse zu Ende gebracht werden können unter Freigabe ihrer bisher belegten Betriebsmittel. Eine Zuteilung erfolgt also nur, wenn das System sicher bleibt, sonst nicht. Speziell bedeutet dies, daß eine Zuteilung u. U. nicht erfolgt, obwohl noch Betriebsmittel dafür frei sind.

Das unten vorgestellte Verfahren heißt auch Banker's Algorithmus, weil er das Verhalten einer Bank simuliert, die Geld an Kreditnehmer ausgibt, deren Kreditwünsche bekannt sind. Eine Zuteilung an einen Kunden erfolgt nur, wenn die Bank unter Berücksichtigung der Rückzahlung voll-

1. vergleiche dazu die Diskussion der Unterschiede auf Seite Seite 128

ständig erfüllter Darlehen, alle Kreditwünsche ihrer Kunden (ggf. nach längerer Zuteilungswartezeit!) befriedigen kann.

Das Verfahren hat eine Laufzeit $O(m \cdot n^2)$.

Eingabe

Zusätzlich zu dem schon aus den Verfahren von Holt und Coffman/Shoshani bekannten Vektor `Verfügbar` und den Matrizen `Hat` und `Will` gibt es jetzt eine `Max-Matrix[1..n, 1..m]` zur Angabe der maximalen Anforderungen mit
`Will[i, j] := Max[i, j] - Hat[i, j]`.

Ferner liegt ein konkreter Anforderungswunsch $\texttt{Zuteilung}_i\texttt{[1..m]}$ des Prozesses *i* vor, über den entschieden werden muß.

Methode

1. Wenn $\texttt{Zuteilung}_i\texttt{[j]} > \texttt{Will[i, j]}$ für ein *j* ($1 \le j \le m$), dann Fehler, da max. Anforderung überschritten.
2. Wenn $\texttt{Zuteilung}_i\texttt{[j]} > \texttt{Verfügbar[j]}$ für ein *j* ($1 \le j \le m$), dann warten Prozeß *i*.
3. Per forma Eintragung der Zuteilung.
   $\texttt{Verfügbar[j]} := \texttt{Verfügbar[j]} - \texttt{Zuteilung}_i\texttt{[j]} \quad \forall\ 1 \le j \le m$
   $\texttt{Hat[i, j]} := \texttt{Hat[i, j]} + \texttt{Zuteilung}_i\texttt{[j]}\ \forall\ j$
   $\texttt{Will[i, j]} := \texttt{Will[i, j]} - \texttt{Zuteilung}_i\texttt{[j]}\ \forall\ j$
   {*jetzt Sicherheitsüberprüfung mit neuem Hat*}
4. Kopieren `Verfügbar` nach `Arbeitsvektor[1..m]`
5. Markiere alle Zeilen *i* in `Hat` mit $\texttt{Hat[i, j]} = 0\ \forall\ j$
6. Finde unmarkierte Zeile *i* so daß
   $\texttt{Will[i, j]} \le \texttt{Arbeitsvektor[j]}\ \forall\ j$
   Für eine so gefundene Zeile *i* setze $\forall\ j$
   ```
   Arbeitsvektor[j] := Arbeitsvektor[j] +
                      Hat[i,j];
   ```
   markiere Zeile *i*; finde nächste Zeile.

7. Bleibt eine unmarkierte Zeile übrig, ist der Zustand unsicher und die Zuteilung kann nicht erfolgen. Die Zuweisungen aus (3) werden dann rückgängig gemacht. Sonst ist die Zuteilung sicher.

Beispiel 8–6

Betriebsmittel		r_1	r_2	r_3
insgesamt		**2**	**3**	**3**
Hat	p_1	1	1	0
	p_2	0	1	0
	p_3	1	0	1
Max (Will)	p_1	1 (0)	2 (1)	1 (0)
	p_2	1 (1)	2 (1)	3 (3)
	p_3	2 (1)	2 (2)	2 (1)
Verfügbar		0	1	2
Zuteilung$_1$		0	1	1
Hat' (Schattenmatrix)	p_1	1	2	1
	p_2	0	1	0
	p_3	1	0	1
Arbeitsvektor		**0**	**0**	**1**

Tab. 8–4 *Überprüfung auf sichere Zuteilung*

Ein Durchrechnen des Beispiels ergibt, daß die Forderung (0, 1, 1) durch Prozeß 1 erfüllt werden kann. Dagegen würde die gleiche Forderung (0, 1, 1) durch Prozeß 3 die Sicherheit verletzen und würde deshalb zurückgewiesen.

8.3.2 Unterschiede Holt/Coffman/Habermann

Zustandsgraph nach Holt:

Der Graph beschreibt ein Gesamtübergangsverhalten, das bekannt sein muß. Ist ein momentaner Zustand sicher, so ist das System sicher, d. h. unabhängig von allen möglichen Übergängen kann kein Prozeß verklemmt werden. Damit ist keine weitere Überprüfung nötig, wenn das System statisch ist. Allerdings gibt es hierfür kein praktikables Verfahren, da der Zustandsraum „explodiert".

Algorithmen von Holt bzw. Coffman/Shoshani:

Die momentanen Anforderungen aller Prozesse sind bekannt. Gibt es eine Anforderung, die momentan nicht erfüllt werden kann, stellt der Alg. fest, ob eine Verklemmung eingetreten ist. Ist die Antwort nein, gibt es (mindestens) eine Abarbeitungsreihenfolge, so daß alle Prozesse beendet werden können unter den momentanen Anforderungen bei sukzessiver Freigabe belegter Betriebsmittel.

Algorithmen von Dijkstra und Habermann (Banker's Alg.):

Die maximalen Anforderungen aller Prozesse sind bekannt. Der Algorithmus prüft, ob eine gestellte Anforderung bei Erfüllung keine Abarbeitungsreihenfolge mehr zuläßt, so daß alle Prozesse zu Ende kommen. Wenn ja, dann führt die Anforderung in einen unsicheren Systemzustand.

9 Programmallokation und Speicherverwaltung

Wichtigstes Betriebsmittel bei der Abarbeitung eines Jobs ist neben dem Prozessor (vgl. Abschnitt über CPU-Zuteilungsstrategien) der *Arbeitsspeicher.*

9.1 Speicherhierarchie

Üblicherweise sind Programm- und Datenspeicher in Hierarchien unterteilt je nach Kapazität, Kosten/Bit, flüchtig/persistent, Zugriffszeit, usw.

Speicher	**Ebene**	
Cache	primär	von oben nach unten wachsende Kapazitäten und abnehmende Geschwindigkeiten
Hauptspeicher		
Flashkarten usw.	sekundär	
Platten		
MO-Datenträger mit direktem Zugriff	tertiär	
Bänder, Kassetten		

Tab. 9–1 *Speicherhierarchie*

Das Problem der optimalen Verteilung auf die Ebenen regelt

- bei Cache/HS die Hardware
- bei HS/Sekundärspeicher das Betriebssystem
- Sekundär-/Tertiärspeicher der Systemadministrator

Dem liegt seit etwa 50 Jahren unverändert das Modell des „von-Neumann-Rechners“ zugrunde mit dem Maschinenzyklus (vgl. fetch/execute cycle auf Seite 9in Kapitel 1):

a) laden Instruktionen, decodieren
b) laden Operanden, verarbeiten
c) speichern Resultate

Dies führt zur Ausbildung des „von-Neumann Flaschenhalses“ zwischen CPU und HS, den man weitgehend durch große und schnelle Caches zu mildern sucht.

9.2 Adreßbildung

Betrachten wir die Adreßbildung für den Zugriff auf Daten und Instruktionen etwas genauer. Zunächt definieren wir einen *physischen Adreßraum* (den Adreßraum des HS) der Speicheradressen im Bereich 0 bis 2^b - 1 anbietet, wobei *b* die realisierbare Anzahl von Bits in der effektiven Adresse ist, bestimmt z.B. von der Busbreite der Rechnerarchitektur.

Jetzt gilt es eine Zuordnung der Adressen zu den Werten von Variablen, Konstanten, Funktionen, etc. herzustellen. Dabei spielt der Zeitpunkt der Festlegung und die spätere Änderbarkeit eine Rolle.

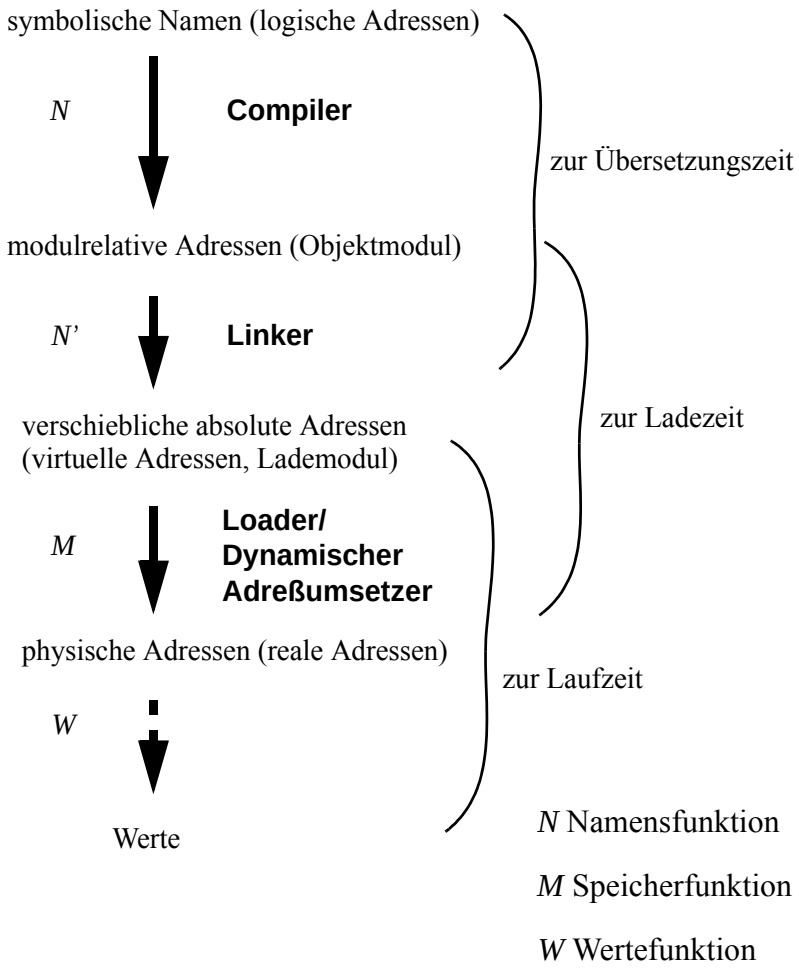


Abb. 9–1 *Adreßfunktionen*

Aus dem Lademodul wird je nach Adressierungsart des Rechners ein ausführbares Modul erzeugt und zwar

- bei *direkter Adressierung* durch Erhöhen aller relativen Adressen um die Startadresse (keine Verschiebung zur Laufzeit möglich)
- bei *relativer Adressierung* durch Errechnung der echten Adresse zur Laufzeit aus relativer Adresse + Wert eines sog. *Basisregisters*. Solche Programme sind *verschieblich* (engl. *relocatable*).

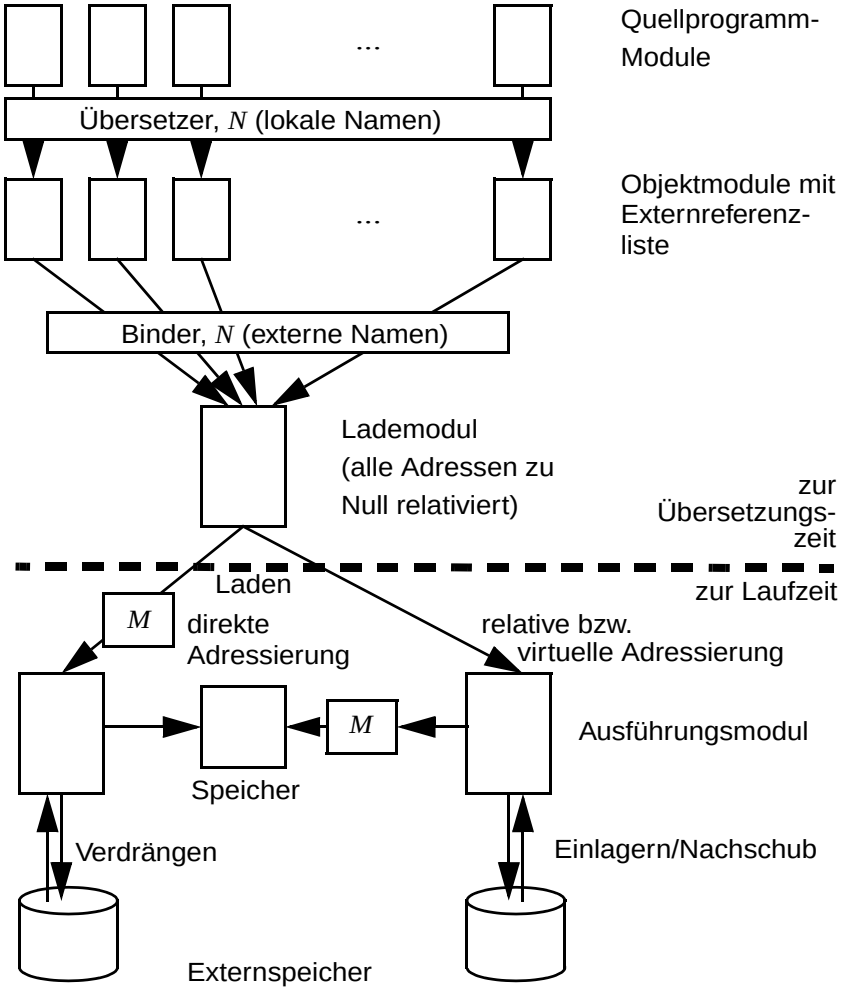


Abb. 9–2 *Bindephasen, Laden und Verdrängen (nach Nehmer)*

Wir beschreiben das Zusammenspiel an einem Anwendungsprogramm, daß zweimal die Quadratwurzelfunktion (SQRT) einer Systembibliothek aufruft. Der im Beispiel in Abbildung 9–3 verwendete Befehl `BAL` (Branch-And-Link) rettet die Rücksprungsadresse (also den Programmzähler + 1) in einem Register und verzweigt zur angegebenen Adresse. Zur Vereinfachung wird angenommen, daß Unterroutinen mittels `BR 14` zurückspringen (besser mittels Stapel zu realisieren).

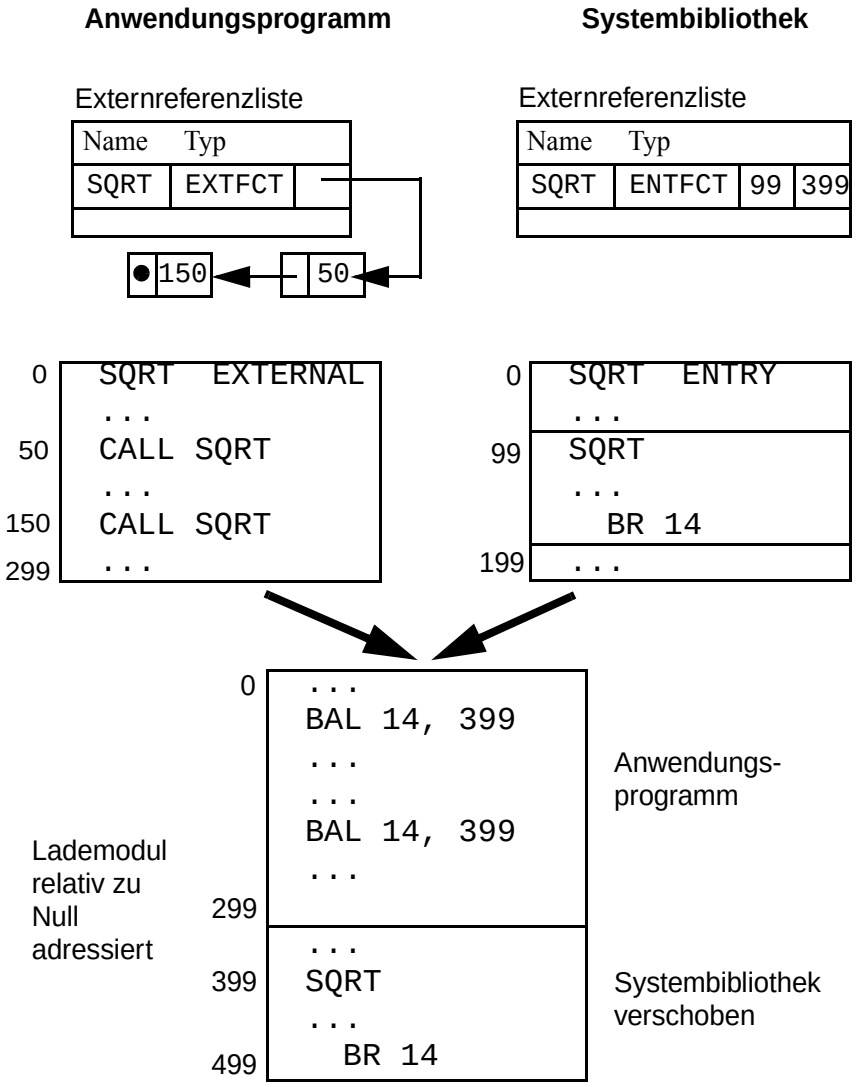


Abb. 9–3 *Beispielprogramm für Binden mit Externreferenzlisten*

9.3 Reelle Adressierung

9.3.1 Ein fester Laufbereich, keine Verdrängung

Bei diesen Formen für sehr kleine Rechner teilt man den HS in zwei Teile:

- residenter Teil des Betriebssystems (sog. Monitor)
- Benutzerprogramm (Code + Datenteil + Laufzeitstack)

Ein Beispiel für eine solche frühe Form der Speicherverwaltung auf Mikrorechnern ist ein 48 KB CP/M[1](Abbildung 9–4).

Man beachte auch das Konzept des Urladers, der zunächst den Lader lädt, der wiederum das Betriebssystem lädt.[2]

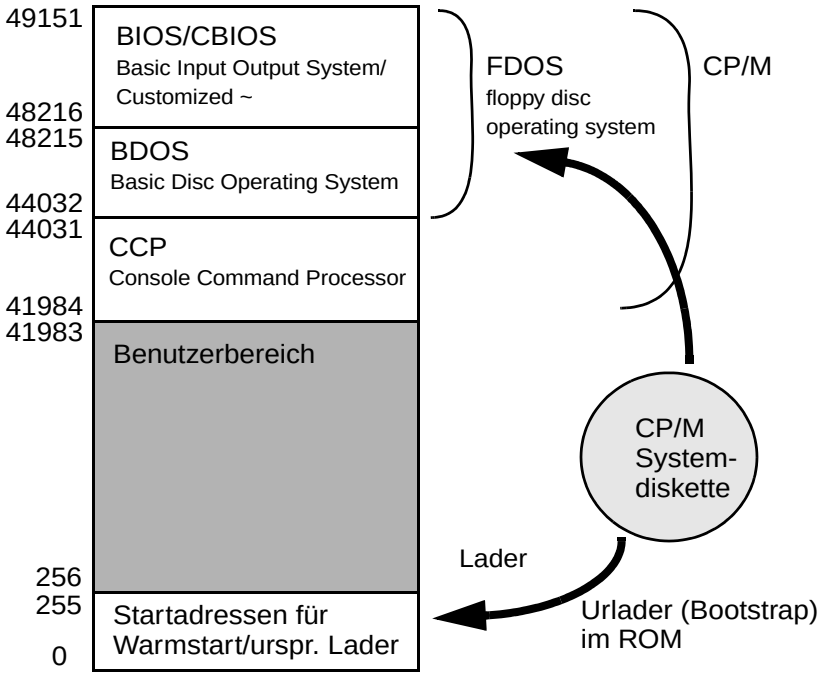


Abb. 9–4 *CP/M (48 K) nach Kaltstart*

1. CP/M ist Warenzeichen der Digital Research Co.
2. Einer alten Informatikerweisheit zufolge hat ein Mensch nicht wirklich gelebt, solange er nicht einmal einen Bootstrap-Lader geschrieben hat.

9.3.2 Fester Laufbereich mit Speicherschutz

In einfachen Systemen wie CP/M war es möglich, den Monitor zu überschreiben (außer dem BIOS), was natürlich extrem gefährlich ist. In „erwachsenen“ Betriebssystemen wird man den residenten Teil des Systems durch einen „Zaun“ schützen. Generell wird man Adreßverletzungen eines Anwendungsprogramms immer entdecken wollen, im Mehrprogrammbetrieb auch gegenüber anderen Anwenderprogrammen.

Früher mußte der Schutz zum Teil selbst einprogrammiert werden (bei jedem potentiell gefährlichen HS-Zugriff, etwa auf Stapeladressen). Das war sehr aufwendig. War zudem ein fester Zaunwert zu verwenden, war kein Wachsen des Monitors möglich.

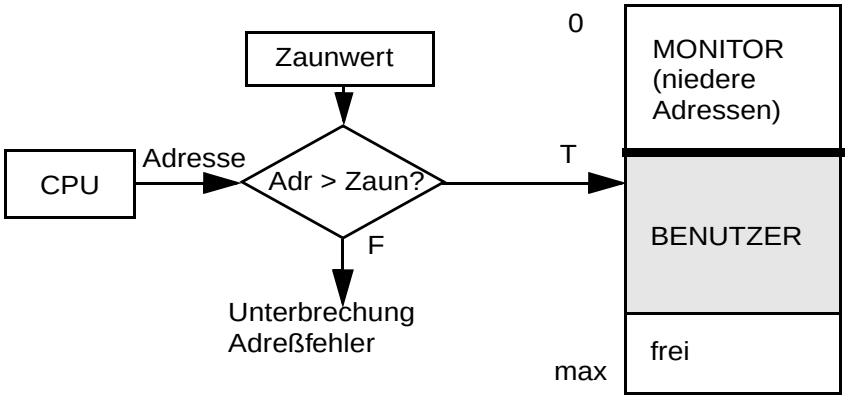


Abb. 9–5 *Schutz des Monitors durch Zaunwert*

Heute unterstützen alle CPUs ladbare Zaunwerte über sog. *fence registers*.

Speicherschutz läßt sich mit Verschieblichkeit kombinieren. Im einfachen Beispiel eines Monitors an den niederen Adressen (hier 0 - 1399) kann durch Verwenden eines Basisregisters mit Wert 1400 sowohl das relativ zu Null adressierte Lademodul zur Laufzeit an jede beliebige Adresse verschoben, als auch der Monitor geschützt werden. Soll allerdings **nach** dem Ladevorgang verschoben werden, muß in einen neuen Bereich umkopiert werden.

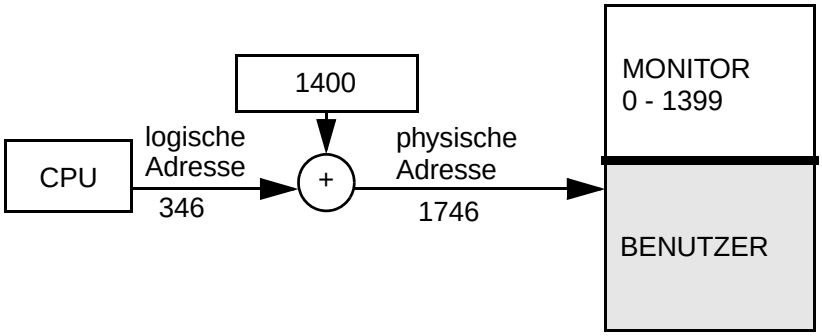


Abb. 9–6 *Verschieblichkeit kombiniert mit Speicherschutz*

9.3.3 Ein fester Laufbereich mit Verdrängung

Diese Vorläufertechnik zu Paging (seitenweise Verdrängung) ist als *Swapping* oder *roll in*/*roll out* bekannt. Ein Prozeß wird komplett auf Sekundärspeicher ausgelagert, z.B. bei einer E/A-Unterbrechung. Die Technik kann mit und ohne Basisregister realisiert werden. Die frühen UNIX-Versionen bis System V arbeiteten häufig mit Swapping, allerdings mit mehreren Laufbereichen, daher hat häufig der Dämon-Prozeß, der die Hauptspeicherverwaltung übernimmt, noch den traditionellen Namen *swapper* (häufiger aber schon *pager*).

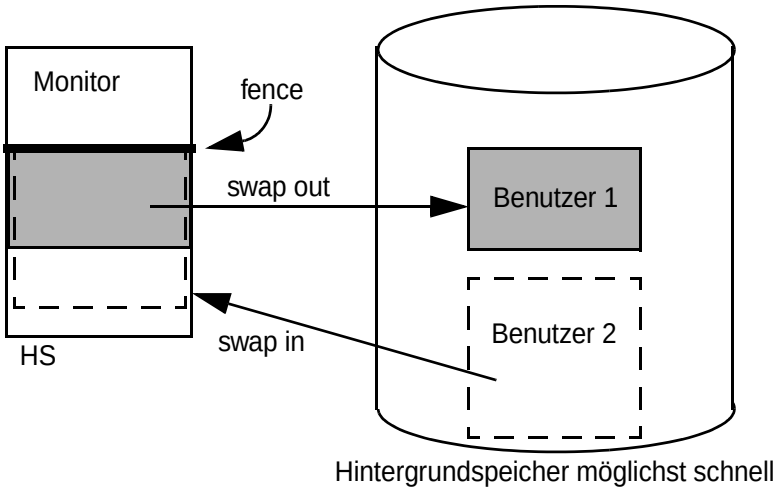


Abb. 9–7 *Swapping mit einem Anwenderprogramm*

Generell sollte für eine sinnvolle Nutzung die Verweilzeit eines Programms sehr viel größer sein als die Auslagerungszeit. Klar ist auch, daß ein wegen einer E/A-Operation ausgelagerter Prozeß seine E/A-Puffer im HS belassen muß!

Vorteile des Swappings sind die generelle Unsichtbarkeit (Transparenz) und der große Speicherplatz für jeden Benutzer.

9.3.4 Mehrere feste Laufbereiche ohne Verdrängung

Diese Technik wurde auf IBM /360 Anlagen als OS/MFT betrieben (Multiprogramming with a Fixed Number of Tasks). Üblich waren drei Bereiche (Foreground1, Foreground2 und Background) mit zahlreichen Einschränkungen.

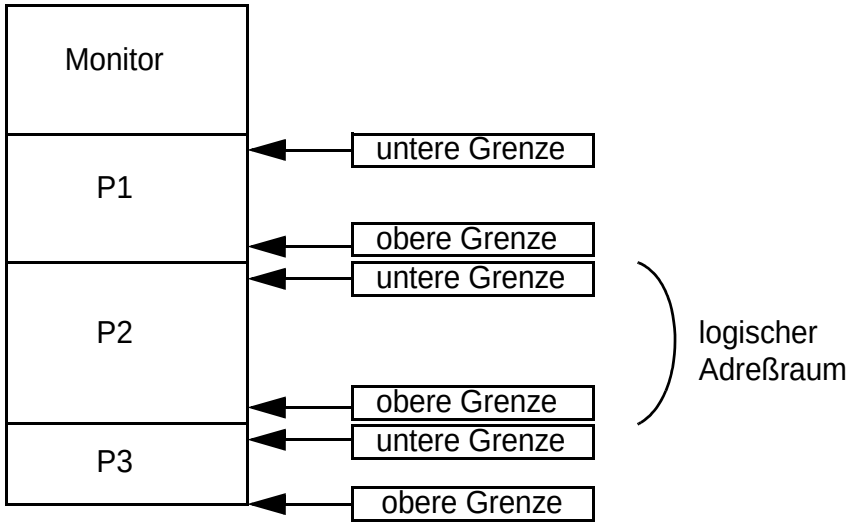


Abb. 9–8 *Beispiel mit drei festen Laufbereichen*

Die Größe der Bereiche wurde vom Operator ein- bis zweimal am Tag eingestellt und war während der Laufzeit eines der Programme nicht änderbar.

Der Speicherschutz der Programme war über *zwei Grenzregister* oder über *ein Basis- und ein Grenzregister* realisierbar.

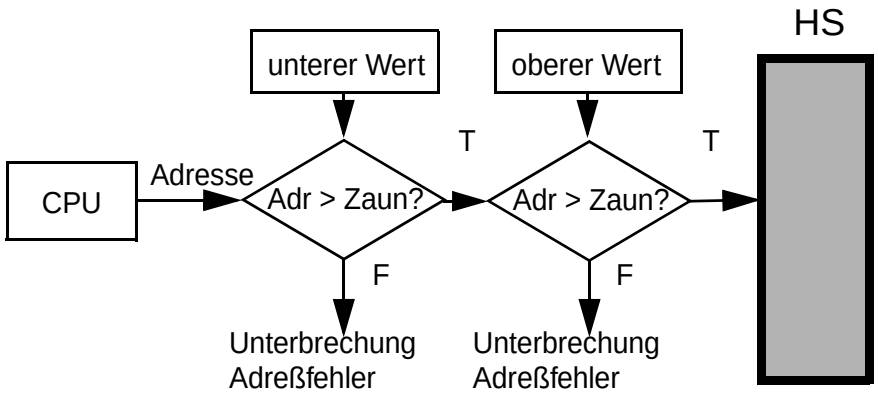


Abb. 9–9 *Speicherschutz mit 2 Grenzregistern*

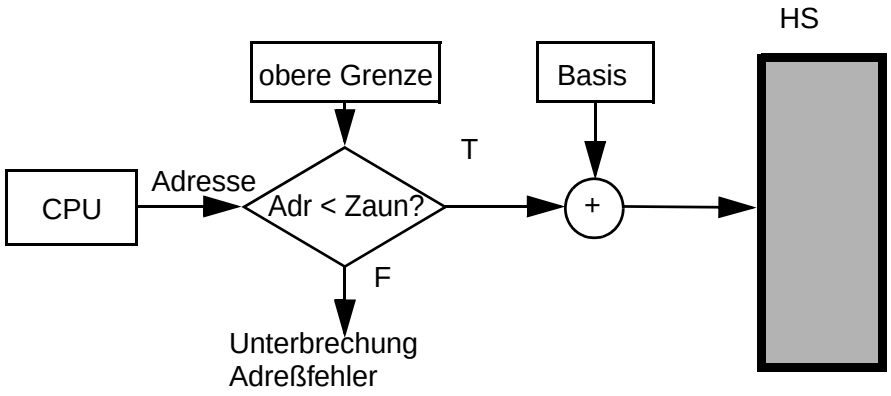


Abb. 9–10 *Speicherschutz mit Basis- und 1Grenzregister*

Probleme bei MFT

- Platzbedarf der Programme sollte bekannt sein.
- Welche Zuteilungsstrategie ist günstig?
  - best fit only?
  - best available fit?

Beispiel 9–1

Der HS habe 320 KB, das BS belegt 100 KB. Drei Anwenderbereiche sind vorgesehen: kleine Jobs mit 20 KB, mittlere mit 60 KB und große mit 140 KB.

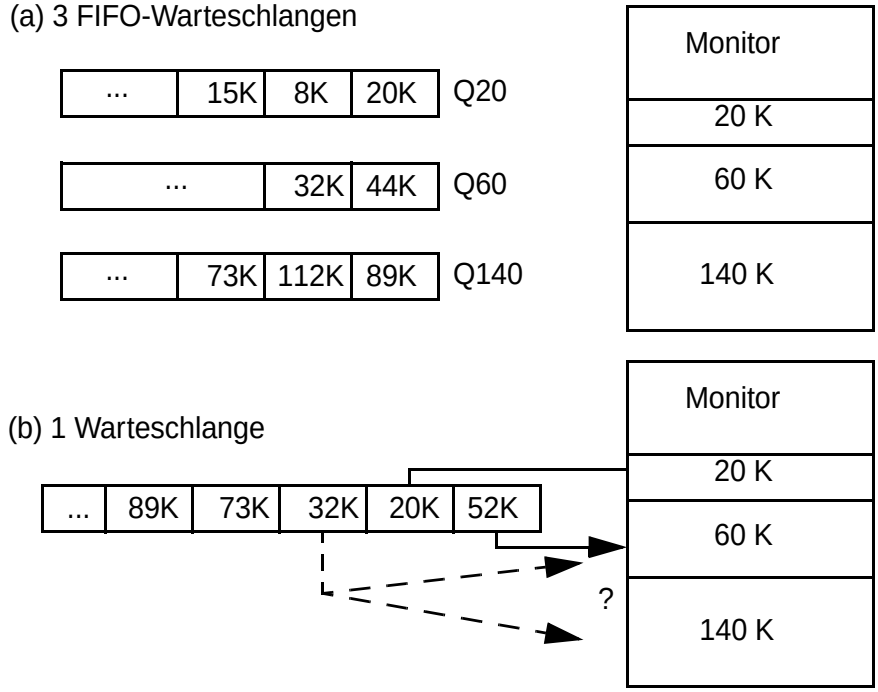


Abb. 9–11 *Realisierungen für best-fit und best-available-fit*

MFT kann mit Swapping kombiniert werden. Dies löst allerdings nicht das Problem der Hauptspeicherzerstückelung (Fragmentierung). Dabei unterscheidet man *interne* und *externe Zerstückelung*.

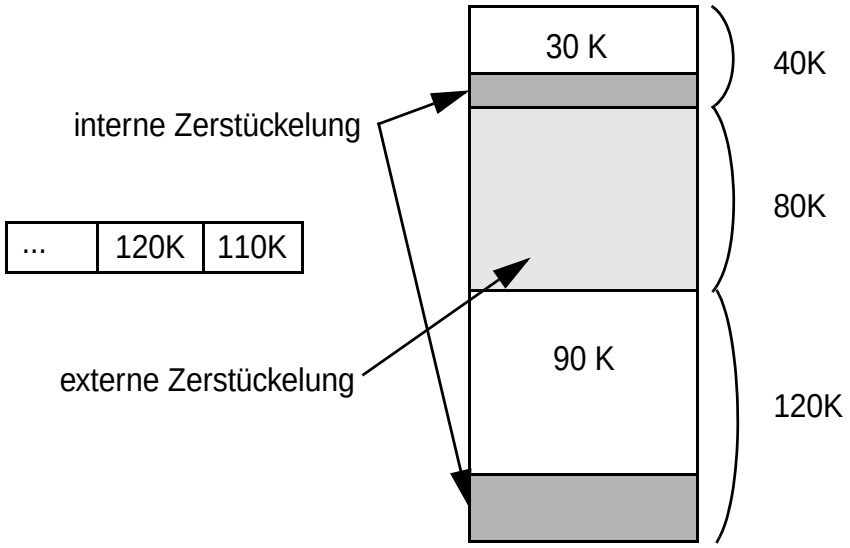


Abb. 9–12 *Fragmentierung bei MFT*

9.3.5 Mehrere variable Laufbereiche

Diese Technik wurde z.B. bei IBM OS/360 MVT (Multitasking with a Variable Number of Tasks) eingesetzt. Sie zeichnet sich aus durch

- dynamische Festlegung der Bereichsgrößen
- Verwaltung und Vereinigung freier Bereiche

Ein Job kann in jedem Bereich laufen, der groß genug ist. Zu große Bereiche werden aufgespalten. Freie Bereiche lassen sich z. B. in einer linearen, verketteten Liste verwalten, wobei man eine Schranke für den kleinsten vergebenen Bereich ansetzt (im Beispiel unten z. B. 2 K). Insgesamt ist diese Technik aus der dynamischen Speicherverwaltung der Programme (sog. *Heap-Segment*) mit `alloc` und `free` bekannt.

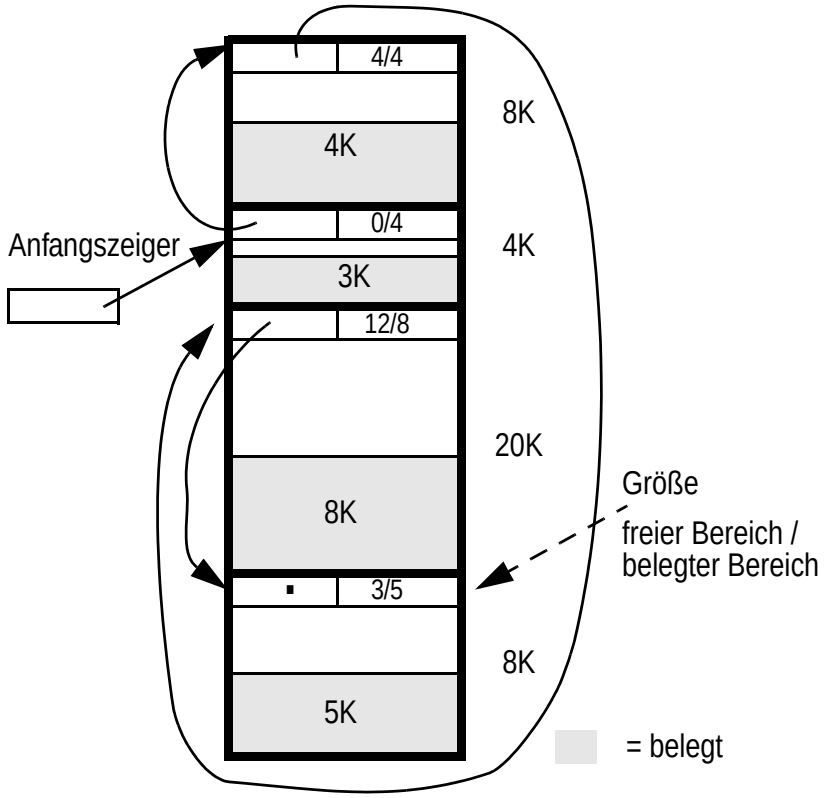


Abb. 9–13 *Speicherverwaltung mit MVT*

Auch bei MVT kommt es zur Speicherzerstückelung. Ein Zusammenschieben (*compaction*, analog zur Defragmentierung der Festplatte) lohnt sich aber nur, wenn es längerlaufende, statische Programme gibt.

Interessant ist ein Vergleich der

9.4 Zuteilungsstrategien

a) **first-fit**
wähle den ersten Block, der groß genug ist (geht schnell)

b) **best-fit**
wähle kleinsten Block, der groß genug ist (gibt kleine Reste, verlangt Durchsuchen der Liste verfügbarer Blöcke)

c) **worst-fit**
wähle größten Block, der verfügbar ist (gibt große Reste, alles durchsuchen).

Bewertung

Simulationen [z. B. Knuth 73] zeigen, daß *first-fit* kaum schlechter ist als *best-fit*, beide jedoch besser als *worst-fit*.

Das folgende Beispiel demonstriert, daß es unter identischen Anfangsbelegungen und identischer Anforderungsfolge bei jeder der drei Strategien zur Blockade kommt, obwohl insgesamt noch ausreichend freier Speicherplatz (aber nicht kontinuierlich) vorhanden ist für die nächste Anfrage.

	first-fit	**best-fit**	**worst-fit**
Anfangsbelegung	**1500, 600, 900**	**1500, 600, 900**	**1500, 600, 900**
500	1000, 600, 900	1500, 100, 900	1000, 600, 900
800	200, 600, 900	1500, 100, 100	200, 600, 900
200	600, 900	1300, 100, 100	200, 600, 700
700	600, 200	600, 100, 100	200, 600
700	blockiert	blockiert	blockiert

Tab. 9–2 *Zuteilungen bei unterschiedlichen Strategien*

Eine schöne Beobachtung mit einem leichten Beweis ist unter dem Begriff „fünfzig-Prozent-Regel“ bekannt (Knuth Vol. I von 1973, S. 445f).

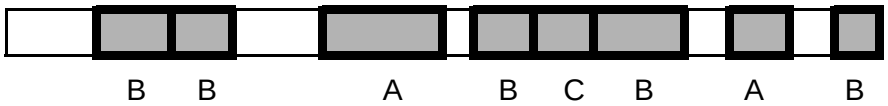
Die Aussage lautet: Bei der Speichervergabe nach first-fit enthält ein System im Gleichgewicht halb so viele freie Blöcke wie belegte. Sei N die

Anzahl der belegten Blöcke, M die Anzahl der freien, dann gilt für große N

$$M \approx ½\, N$$

Der Begriff „Gleichgewicht“ ist so definiert, daß die Anzahl der unbelegten Blöcke im Durchschnitt bei aufeinanderfolgender Freigabe und Belegung eines Blockes gleich bleibt („eingeschwungener Zustand“).

Für den Beweis der Regel betrachtet man eine beliebige Speicherbelegung.



Wie man sieht, kann man drei Arten von Blöcken unterscheiden.

- A: bei Freigabe verringert sich M um eins.
- B: bei Freigabe bleibt M gleich.
- C: bei Freigabe erhöht sich M um eins.

Sei nun a, b, c die Anzahl der Blöcke vom Typ A, B, C. Dann gilt

$$N = a + b + c$$
$$M = ½\,(2a + b)$$

wobei der Rand vernachläßigt wird. Die zweite Gleichung folgt aus der Tatsache, daß jeder A-Block zwei freie Blöcke produziert, jeder B-Block einen. Der Faktor ½ verhindert die Doppelzählung.

Wegen des Gleichgewichtszustands gilt $a = c$. Damit aber

$$N = 2a + b,\ 2M = 2a + b \ \rightarrow\ N = 2M$$

q.e.d.

Eine Speicherverdichtung, so man sie überhaupt ins Auge faßt, läßt sich gut als Nebeneffekt von Swapping machen. Verschiedene Strategien sind möglich, wie man am folgenden Beispiel erkennen kann.

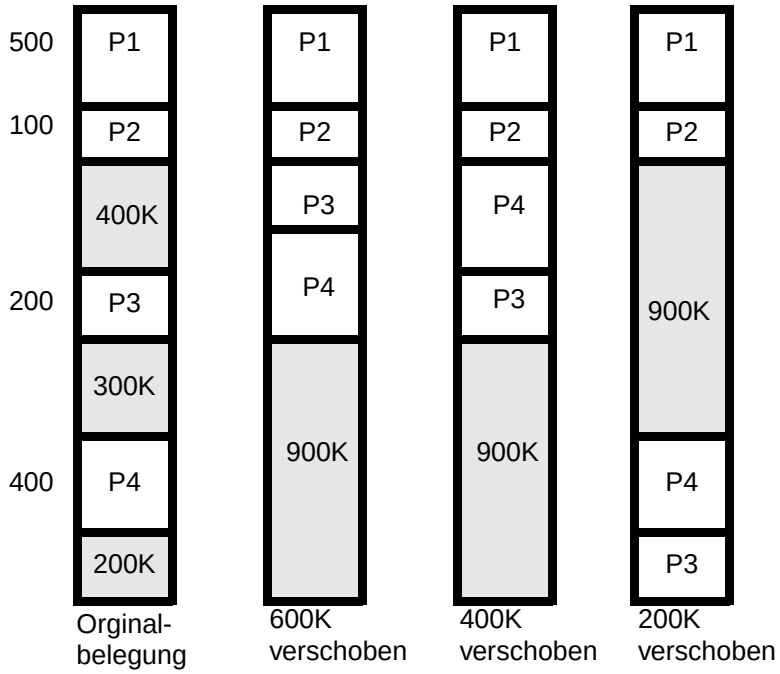


Abb. 9–14 *Auswirkungen verschiedener Strategien bei der Speicherverdichtung*

9.5 Zusammenhänge zwischen der Speicherzuteilungsstrategie und dem „Bin Packing Problem“

Das „Bin Packing Problem“ (Packungsproblem des OR) ist ein bekanntes, schweres Problem des Operations Research. Gegeben eine Liste $L = (a_1, a_2, ..., a_n)$ von n Gütern mit Gewicht (Größe, Wert) $a_i \in (0, 1]$ für $1 \leq i \leq n$, finde die minimale Anzahl von Behältern mit Einheitsgröße um alle n Güter zu verpacken.

Beispiel 9–2

Eine Lagerhalle enthalte $n = 6$ Kisten mit unterschiedlichen Gewichten.

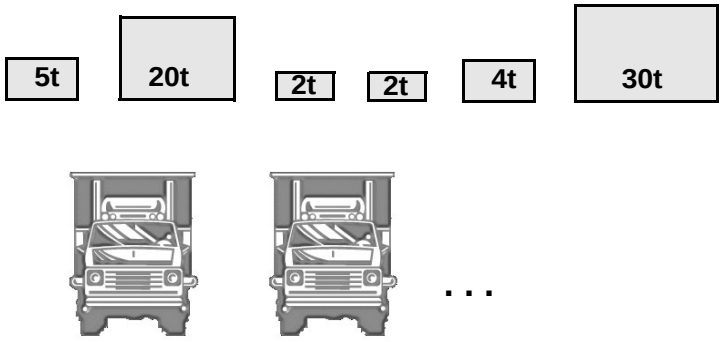


Abb. 9–15 *Güter (Kisten) und Behälter (Lkw)*

Diese Kisten sollen mit Lkw in eine andere Halle gebracht werden. Jeder Lkw kann max. 32t laden, das Volumen spiele keine Rolle. Wieviele Lkw benötigt man, wenn man geschickt packt?

Diese Aufgabe gehört zur Problemklasse der *NP-vollständigen Probleme*, d.h. es ist kein polynomieller Algorithmus (Laufzeit n^k für k konstant) bekannt, aber es ist nicht bewiesen, daß es keinen solchen geben kann. Man behilft sich mit Heuristiken, z. B. First-Fit, Best-Fit, Next-Fit, ...

Die Speicherverwaltungsprobleme ähneln allerdings eher dem sog. *On-line Bin Packing Problem*, bei dem die Entscheidung „wo geht Gut i hin?“ (für $i = 1, 2, ...$) ohne Kenntnis der folgenden Güter getroffen werden muß.

Um die Qualität einer Heuristik A beurteilen zu können, die $A(L)$ Behälter für eine Liste L benötigt, definiert man sie in Relation zur Anzahl L^* der Behälter bei Verwendung eines optimalen Algorithmus:

$$r(A) = \limsup_{L^* \to \infty} \frac{A(L)}{L^*}$$

Nach einer Arbeit von C. C. Lee und D. T. Lee: *A Simple On-Line Bin-Packing Algorithm*, J. ACM, Vol. 32, No. 3, July 1985, pp. 562-572 betrachtet man z. B. das *Next-Fit-Verfahren*.

```
for i := 1 to n do
  if Gut_i paßt in gegenwärtigen Behälter
  then packe Gut_i in diesen Behälter
  else begin packe Gut_i in nächsten leeren Behälter,
    alter Behälter raus;
  end;
```

Die Laufzeit ist $O(n)$, der Platzbedarf $O(1)$. Für die Qualität gilt r(Next-Fit) = 2.

Beim folgenden First-Fit-Verfahren entläßt man die Behälter nicht, es sei denn, einer sei voll.

```
k := 1;
for i := 1 to n do
begin
  j := 1; packed := false;
  while not packed do
    if Gut_i paßt in Behälter j
    then (Gut_i in Behälter j; packed := true)
    else j := j + 1;
  k := max(k, j)
end;
writeln(k, 'Behälter gebraucht)');
```

Die Laufzeit hier ist $O(n \log n)$ und der Platzbedarf ist $O(n)$, wobei die Laufzeit nur durch geschickte Organisation der Angaben zu den freien Kapazitäten erreichbar ist, etwa mit einem balancierten Suchbaum. Für die Qualität gilt nun

$$r(\text{First-Fit}) = 1{,}7.$$

Untere Schranken für On-line Bin-Packing Heuristiken liegen beweisbar bei

- 1,536... wenn Platz $O(n)$ erlaubt ist
- 1,6910... wenn Platz $O(1)$ gefordert wird.

Die z. Zt. besten bekannten Verfahren sind

- HARMONIC: $O(1)$ Platz, $O(n)$ Zeit, $r(\,) = 1{,}692...$
- REFINED-HARMONIC: $O(n)$ Platz, $O(n)$ Zeit, $r(\,) \leq 1.636$

9.6 Das Buddy-Verfahren

Das Buddy-Verfahren ist eine Mischform aus starrer und dynamischer Segmentierung. Die effiziente Implementierung ist trickreich und ist in [Knuth 73] im Detail angegeben.

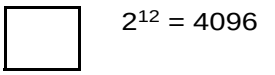
Die Grundidee ist einfach: Speicherblöcke werden nur in Größen vergeben, die Zweierpotenzen sind. Bei Anforderung von l Worten wird also ein Segment 2^k vergeben mit $2^{k-1} < l \leq 2^k$.

Dazu kann ein freies Segment der Länge 2^i, $i > k$, sukzessive in zwei „Buddies“ der Länge 2^{i-1} aufgespalten werden. Beim Freigeben werden benachbarte Buddies verschmolzen, um wieder zu großen Segmenten zu kommen.

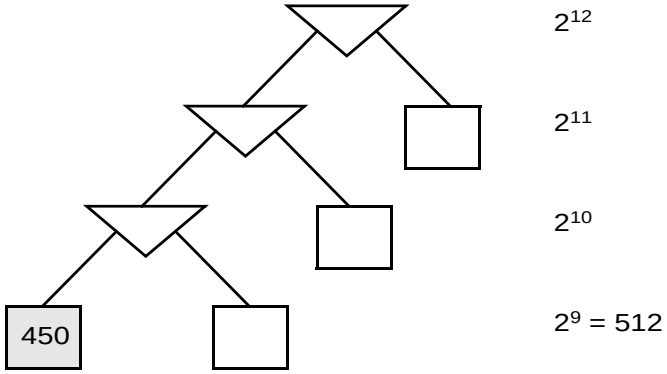
Beispiel 9–3

Es sollen nacheinander Speichersegmente der Größen 450, 1500, 100, 500 belegt werden. Danach werden die Segmente 450 und 100 freigegeben. Darauf folgt eine Belegung 50, 1000. Die Speichergröße sei 4096 (Kilo-, Mega-, Gigabyte).

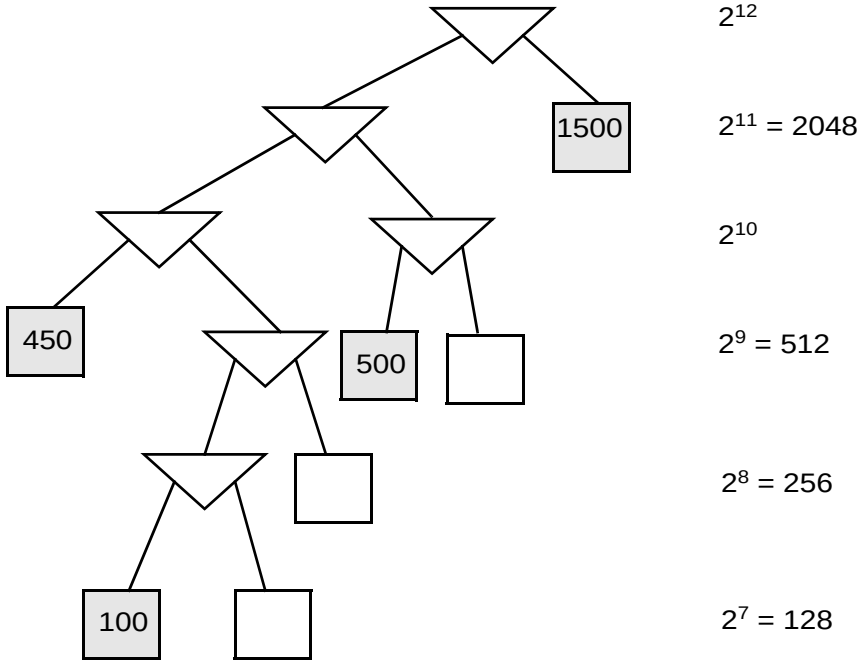
(a) Ausgangssituation



(b) Situation nach Anforderung 450



(c) nach Belegen 1500, 100, 500



(d) nach Freigeben 450, 100

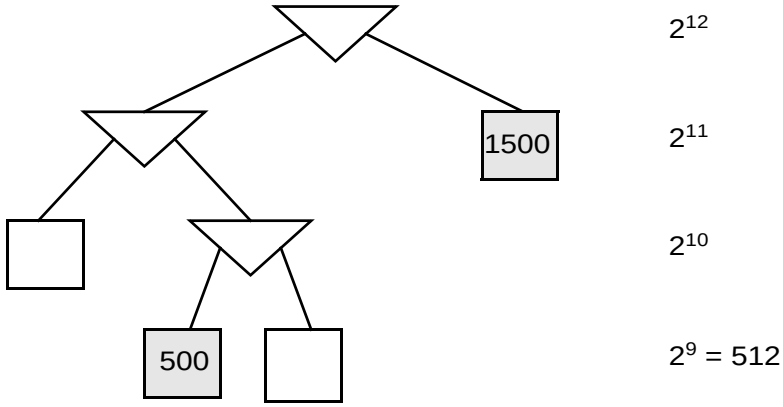


Abb. 9–16 *Beispiel zum Buddy-Verfahren*

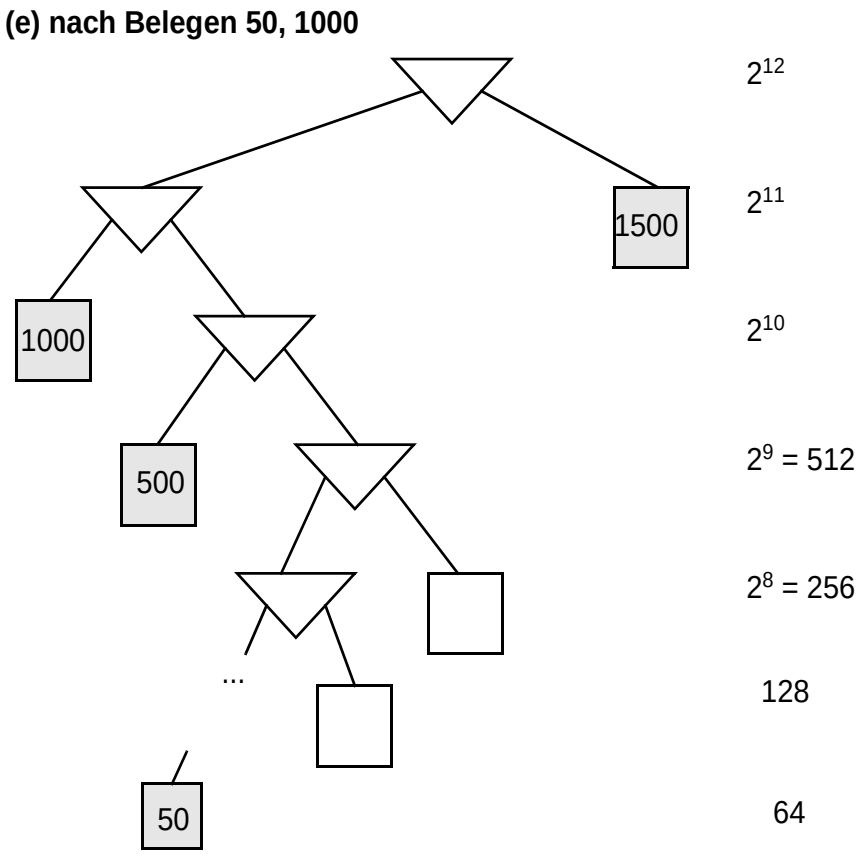


Abb. 9–17 *Endsituation der Segmentvergabe*

Bei Verwendung von Zweierpotenzen gilt: Der Buddy eines Segments der Größe 2^k, das auf Adresse *x*000...0 liegt (*k*+1 Nullen rechts), hat Adresse *x*100...0 (*k* Nullen rechts, nächsthöheres Bit ist Eins), bzw. umgekehrt.

Für die Verwaltung eines Speichers der Größe 2^m (Adressen 0 bis 2^m-1) im Buddy System benötigt man die folgende Verwaltungsinformation:

- 1 bit je freiem oder belegtem Block zur Anzeige ob frei oder belegt.
- 3 Worte je freiem Block für die Angabe der Größe des Blocks und 2 Verweise (vorwärts und rückwärts) auf Blöcke gleicher Größe

- $2m+2$ Anfangszeiger auf $m+1$ Listen zur Verwaltung freier Blöcke der Größen 2^0, 2^1, ..., 2^m.

Das folgende Verfahren belegt einen Block der Größe 2^k.

a) Finde den kleinsten Block der Größe 2^j ($k \leq j \leq m$) der frei ist (Anfangszeiger nicht **nil**).
b) Entferne den Block aus der Liste und markiere ihn als belegt.
c) wenn $j = k$ dann fertig;
sonst splitte den Block; $j := j - 1$; trage unbenutzte Hälfte in Liste ein, markiere diese als unbelegt, gehe nach (c).

Nun brauchen wir noch einen Algorithmus zur Freigabe eines Blocks der Größe 2^k.

a) Gehe zur Buddy-Adresse.
b) Wenn $k = m$ oder Block belegt oder Block frei und
Größe $\neq k$ (es werden nur die Exponenten gespeichert)
dann trage Block in Liste für 2^k als frei ein, markiere als unbelegt, fertig
sonst {*Vereinige*} entferne Buddy-Block aus Freiliste, setze $k := k + 1$, gehe nach (a).

9.7 Reelle Adressierung mit Overlay

Diese ältere Technik benutzt eine vom Anwender zu steuernde Verdrängung und Überlagerung von Programmteilen. Im folgenden Beispiel kommt in einem Speicher der Größe 70 K ein Programm (die Prozedur *C*, 40 K) zu laufen, die Prozeduren *A* und *B* (je 30 K)aufruft, jedoch nicht gleichzeitig.

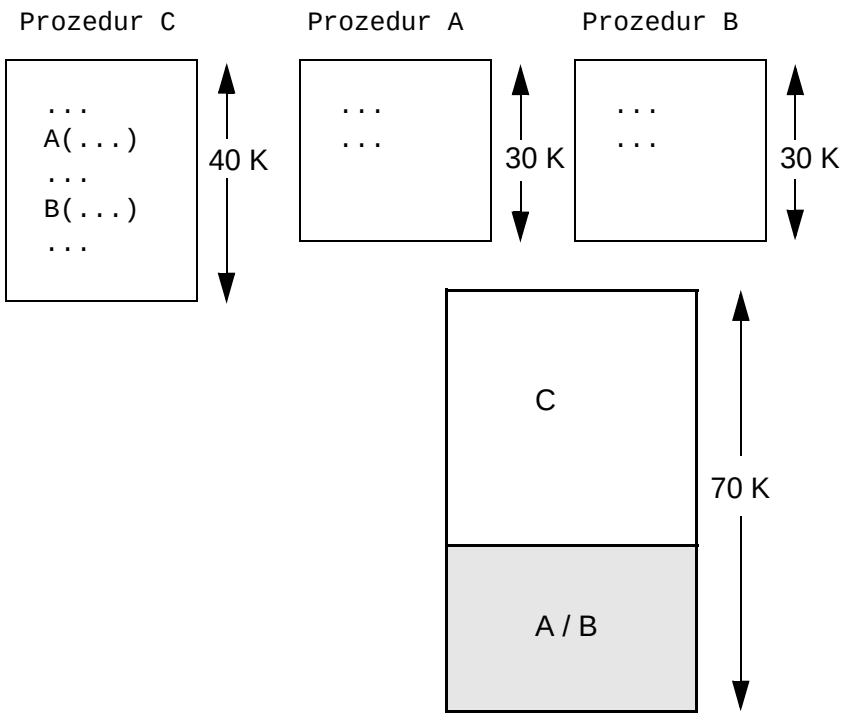


Abb. 9–18 *Overlay Technik*

Der Einsatz erfordert Kenntnis zur Bindezeit, daß z. B. in *A* nicht *B* aufgerufen wird. Problematisch ist auch die Behandlung von Laufzeitstacks, speziell die Rekursion.

10 Virtuelle Adressierung

10.1 Adreßumsetzung über Tabellen

Generell kann es zwei Formen der Abweichung des physischen vom logischen Adreßraum geben (vgl. Abbildung 10–1).

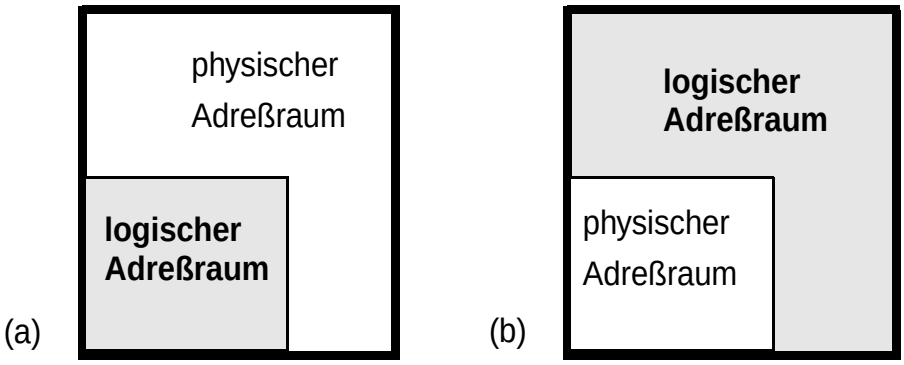


Abb. 10–1 *Logischer und physischer Adreßraum*

Die Situation (a) wird man heute nur noch selten antreffen, etwa wenn ein Rechner mit einem „alten“ Instruktionssatz, z. B. mit 16 bit Adressen, auf einer modernen Hardware mit 1 MB Speicher läuft. In MS-DOS war dies aber lange Zeit gängige Praxis mit allerlei „Klimmzügen“ (extended und expanded memory oberhalb von 640 KB[1]).

Andererseits kommt man mit 32 bit Adressen (2^{32} = 4 Giga) in eine ähnliche Situation, da HS immer billiger und größer wird. Beim Plattenplatz ist ebenfalls evident, daß eine Adreß-Umsetzung erfolgen muß.

---

1. „Nobody will ever need more than 640k RAM!“ – *Bill Gates, 1981*

Die gängige Situation ist aber (b), speziell auch zukünftig beim Übergang auf 64 bit Adressen. Man spricht von virtueller Adressierung, wobei Hauptspeicher und schneller Sekundärspeicher zusammengehen zu einem einstufigen (virtuellen) Speicher.

10.2 Paging

Alle modernen Mikroprozessoren unterstützen heute Paging (virtuelle Speicherverwaltung auf Seitenbasis) „on chip“, früher waren auch separate MMUs (*memory management units*) üblich.

Das Grundprinzip ist, den logischen Adreßraum in Seiten (pages) und den physischen Adreßraum in Kacheln (frames, tiles) zu unterteilen, wobei Seitengröße = Kachelgröße. Eine auch heute noch gängige Seitengröße aus IBM /370 Zeiten ist 4096 bytes (4 KB). Sehr frühe UNIX-Systeme auf der DEC 10 liefen mit 512 Worten.

Programme laufen in einem großen virtuellen Adreßraum. Teile dieses Programms sind physisch im HS vorhanden, aber nicht notwendigerweise kontinuierlich, sondern auf beliebigen Kacheln. Die Umsetzung zwischen logischer und physischer Adresse macht eine Seitentabelle, die jeder log. Seitennummer eine Kachelnummer zuteilt, so die Seite im HS eingelagert ist.

Wird auf eine Adresse im log. Adreßraum zugegriffen, für die es momentan keine Entsprechung im HS gibt, wird ein Seitenfehler erzeugt (Unterbrechung) und die Seite wird nachgeladen, ggf. unter Verdrängung einer anderen Seite. Entsprechend sind die Einträge in der Seitentabelle zu ändern.

Man nennt dies „paging-on-demand“. Es funktioniert gut, weil Programme in der Regel ein lokales und zeitliches Lokalitätsverhalten aufweisen (über längere Zeitabschnitte hinweg werden immer wieder lokal eng begrenzte Speicherbereiche angesprochen, etwa die Instruktionen in einer Schleife oder die Daten auf dem Stack).

Grundlage des Verfahrens ist die Seitentabelle, die für jede Seite des logischen Adreßraums entweder die Kachelnummer enthält, wenn die Seite eingelagert ist, oder den Wert 0 sonst.

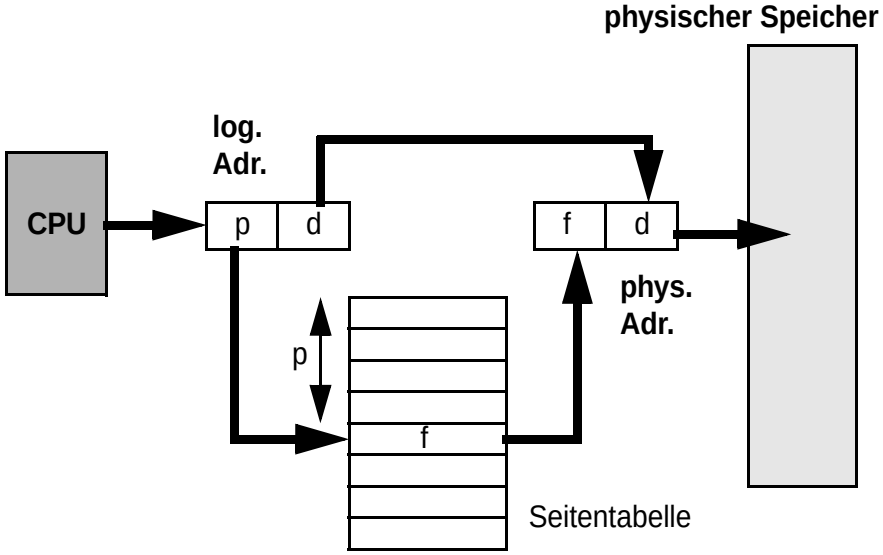


Abb. 10–2 *Adreßumsetzung beim Paging*

Wie in Abbildung 10–2 gezeigt, wird also eine „große“ logische Adresse in eine „kleine“ physische umgesetzt. Dabei bleiben die niedrigwertigen bits (bei 4 KB Seiten die unteren 12 bit) erhalten und bilden die Versetzadresse (displacement *d*) in die Seite, bzw. Kachel hinein.

Würde man jetzt einen 2^{32} log. Adreßraum (4 GB) in einen 2^{28} physischen Adreßraum (256 MB) abbilden, dann wäre eine 20 bit Seitennummer ($2^{20} \approx 1$ Mio Seiten) auf eine 16 bit Kachelnummer (max. 2^{16} Kacheln) abzubilden. Dies würde bedeuten, daß die Seitentabelle nur 2-Byte Einträge[1] aufnehmen muß. Im Falle eines maximal großen Programmadreßraums von 4 GB erfordert diese Seitentabelle trotzdem 2 MB (1 Mio Seiten × 2 Bytes) und das in einem HS von nur max. 256 MB! Gleichzeitig ist die Mehrheit der Einträge Null!

1. Tatsächlich sind weitere Bits je Eintrag üblich, z.B. Schreibschutz, ein Schreibmodifikationsbit und Hilfen für den Verdrängungsalgorithmus

Es drängt sich die Idee auf, die Seitentabelle als Hash-Tabelle anzulegen. Dies ist aber zu langsam, da die Umrechnung bei jedem HS-Zugriff erfolgen muß. Indirekt arbeitet man allerdings mit einer Art Hashing, und zwar in einem speziellen Adreßcache, dem sog. *Translation Look-aside Buffer* (*TLB*). Dort werden mehrere Seitennummern per XOR-Faltung auf eine Cache-Zeile abgebildet. Näheres erfährt man in einer geeigneten Rechnerarchitekturvorlesung.

Ein genereller Nachteil eines Rechners mit Paging ist, daß die Programmgeschwindigkeit geringer ist als bei reeller Adressierung. Das genaue Maß hängt vom Instruktionsmix ab, da nur HS-Operanden die Umsetzung (und damit einen zweiten HS-Zugriff, sofern nicht Adresse im TLB gefunden) erfordern. Register-Operanden benötigen die Umsetzung natürlich nicht, das Laden von Instruktionen dagegen schon, sofern diese nicht bereits im Instruktionscache vorhanden sind, der mit virtuellen Adressen angesprochen wird.

Der große Vorteil des Paging ist, daß *keine externe Fragmentierung* (jede Kachel kann jede beliebige Seite aufnehmen) und nur eine *geringe interne Fragmentierung* (im Mittel eine halbe Seite je Programmsegment) entsteht.

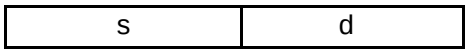
Der Zugriff auf die Seitentabellen (eine je Programm) erfolgt über ein Seitentabellenbasisregister, das die Startadresse der Tabelle enthält. Es wird bei einem Kontextswitch (Prozeßumschaltung) mit dem neuen Wert geladen.

Seitentabellen können auch für die simultane Nutzung von Textsegmenten genutzt werden, wobei natürlich der Instruktionsteil reentrant geschrieben sein muß. In diesem Fall wird von mehreren Seitentabelleneinträgen auf gleiche Kachelnummern verwiesen. In UNIX wird dies aber über Segmenttabellen geregelt, zumal man einen Zähler je Segment braucht, der feststellt wieviele Nutzer noch das Segment in Bearbeitung haben (Segment kann überschrieben werden, wenn Zähler = 0).

Auf die interessanten Fragen der geeigneten Seitenersetzungsstrategien gehen wir unten ein.

10.3 Segmentierung

Hier wird der logische Adreßraum in mehrere Segmente (variabler Länge) aufgeteilt, z. B. nach funktionalen Gesichtspunkten (Textsegment, Stapelsegment, Heap, Datensegment). Diese Aufteilung kann der Compiler vornehmen. Segmenttabellen (für den gesamten logischen Adreßraum) nehmen dann die Abbildung zur physischen Adresse vor, indem je Segment eine Zeile in der Segmenttabelle eingetragen wird mit Basisadresse im HS und Längenangabe des Segments. Die Adressierung im Segment erfolgt wieder mit einem Displacement.



virtuelle Adresse
(s = Segmentadresse, d = Relativadresse im Segment)

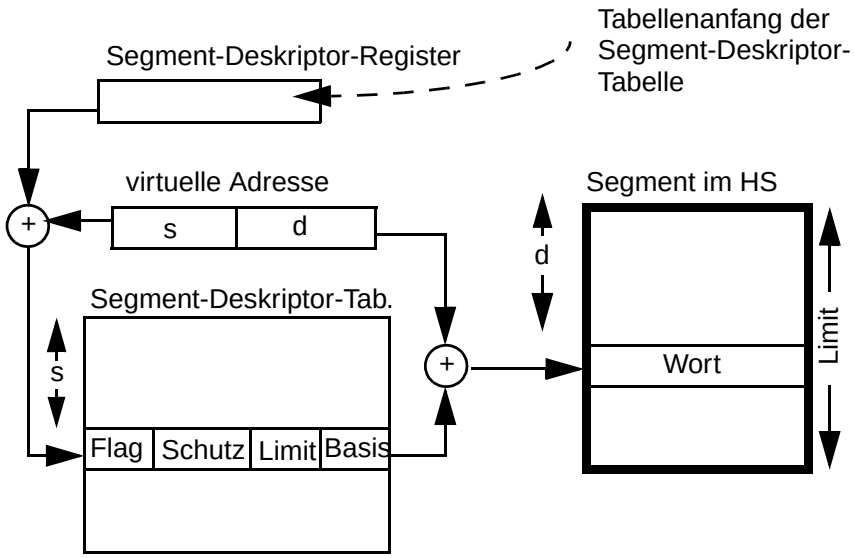


Abb. 10–3 *Prinzip der Adreßtranslation bei Segmentierung*

Eine der Rechner mit Segmentierung war die PDP 11/45 mit 8 Segmentregistern und einer 16 bit logischen Adresse. Davon entfielen demnach 3 bit auf die Segmentnummer, der Rest bildete ein 13 bit Offset, was wiederum 8 KB Segmente erzeugte.

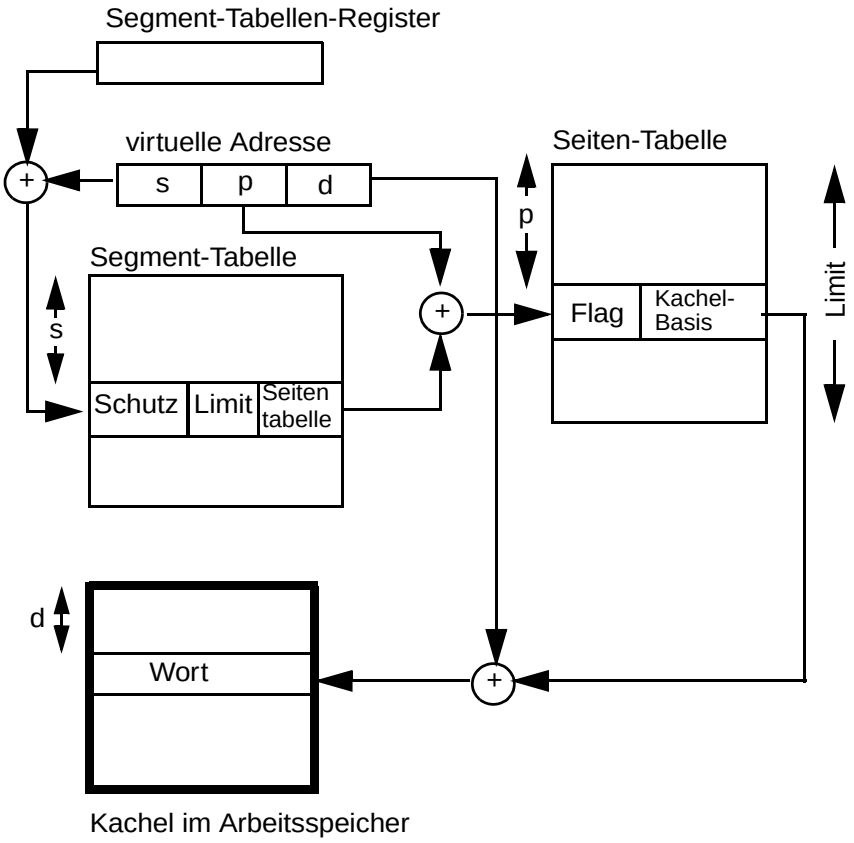


Abb. 10–4 *Prinzip der Adreßtranslation bei Segmentierung mit Seitenadressierung*

Eine sehr fortschrittliche Architektur zeigte das Betriebssystem Multics[1] für die GE 645: 256K Segmente [sic] konnten je bis zu 64K Worte enthalten.

Entsprechend war bei der GE 645 wie auch bei der IBM 360/370 Segmentierung kombiniert mit Paging. Der Adressraum ist also unterteilt in Seiten fester Länge, mehrere (Hundert, Tausend) dieser Seiten bilden Segmente variabler Länge. Das oben beschriebene MVT ist davon ein

1. Der Name UNIX ist gerüchteweise eine Anspielung auf Multics, das in den Bell Labs Ende des Sechziger im Einsatz war.

Spezialfall, indem ein Job ein Segment bildet. Abbildung 10–4 zeigt die Adreßbildung.

10.4 Seitenverdrängungsalgorithmen

Wie oben angedeutet, ist die Normalform des Pagings das *demand-paging*, d.h. eine Seite wird nur geladen, wenn eine nichtvorhandene Adresse referenziert wird. Dazu erzeugt das System einen Seitenfehler (interrupt), der Status des programms wird gerettet, es erfolgt die I/O-Operation zum Laden des Seite von Platte, danach muß die Instruktion neugestartet werden. Der Neustart von Instruktionen in den heutigen Pipelinearchitekturen ist dabei nicht unproblematisch. Sinnigerweise wird man bei einer HS-HS-Operation und Seitenfehler für Operand A nicht gerade die Seite verdrängen, von der Operand B geladen oder auf die er geschrieben werden soll.

Neben *paging-on-demand* gibt es noch *pre-paging*, d.h. vorausschauendes Laden. Dies setzt genaue Kenntnis des Lokalitätsverhaltens voraus. Üblicherweise wird man es nur beim Programmstart verwenden können.

Somit müssen Seitenverdrängungsalgorithmen das zukünftige, unbekannte Seitenzugriffsverhalten aus der Vergangenheit approximieren. Um ein Vergleichsmaß zu haben, beginnt man mit der nicht implementierbaren optimalen Verdrängung.

a) **Optimale Verdrängung (Belady's Strategy)**
   Verdränge die Seite, auf die die längste Zeit nicht wieder zugegriffen wird.

Implementierbar mit wechselndem Aufwand sind die folgenden Verfahren:

b) **Zufallsauswahl**
   Verdränge beliebig eine per Zufallsgenerator ausgewählte Seite; einfach implementierbar, schlechter Wirkungsgrad.

c) **First-In-First-Out (FIFO)**
   Verdränge die am längsten im Speicher befiendliche Seite; erfordert einen Zähler je Seite oder läßt sich über eine Schlange von

Seiten (Verkettung) realisieren, die nur zur Einlagerungszeit lokal modifiziert wird; berücksichtigt aber nicht die *Benutzungshäufigkeit*.

Beispiel 10–1

Wir betrachten die Seitenzugriffsfolge 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Im Fall I seien dem Programm drei Kacheln zugeteilt. In den Tabellen sind Kacheln, deren Seiten verdrängt werden, grau unterlegt.

	1	1	1	4	4	4	5	5	5	...
		2	2	2	1	1	1	3	3	...
			3	3	3	2	2	2	4	...

Tab. 10–1 *Fall I - FIFO mit 3 Kacheln*

Wie man sieht, treten 9 Seitenfehler auf, davon 6 „echte“.

Im Fall II teilen wir jetzt 4 Kacheln zu.

	1	1	1	1	5	5	5	5	4	4	...
		2	2	2	2	1	1	1	1	5	...
			3	3	3	3	2	2	2	2	...
				4	4	4	4	3	3	3	...

Tab. 10–2 *Fall II - FIFO mit 4 Kacheln*

Bei gleicher Zugriffsfolge beobachten wir 10 Seitenfehler, davon 6 echte, obwohl das Programm mehr Kacheln hatte als im Fall I. Diese seltsame Verhalten (mehr Seitenfehler trotz größerer HS-Zuteilung) nennt man *Belady's Anomalie*.

d) **Least-recently-used (LRU)**
Verdrängt wird die am längsten nicht referenzierte Seite, d.h. das vergangene Verhalten wird extrapoliert. Benötigt wird ein USED-

Bit in der Seitentabelle und ein Zähler je Seite (ggf. seperat gespeichert). Beim Zugriff auf die Seite wird automatisch das USED-Bit auf 1 gesetzt. In periodischen Abständen (z.B. alle 0,1s) werden alle Zähler deren Seiten ein USED-BIT = 0 aufweisen (auf die also nicht in den letzten 0,1s zugegriffen wurde) um 1 erhöht. War das USED-Bit = 1 wird es auf Null gesetzt wie auch 0 beim Zähler eingetragen. Das „Opfer“ für eine Verdrängung ist die Seite mit dem höchsten Zählerstand.

Generell ist LRU äquivalent mit einer doppelt verketteten Schlange in Kombination mit der *move-to-front Strategie*, d. h. bei einem Zugriff wird das betroffene Element an die Spitze gebracht[1]. Opfer ist das letzte Element der Schlange. Allerdings ist diese Implementierung für Seitenzugriffsverfahren zu langsam, da die Pointerzuweisungen anders als bei FIFO für das Umhängen *bei jedem Speicherzugriff* erfolgen müßten.

LRU zeigt ein sehr gutes Verhalten.

e) **Clock-Algorithmus**

Die Seiten sind kreisförmig verkettet (z.B. auf der Basis FIFO). Wird ein Opfer gesucht, streicht der „Uhrzeiger“ von der letzen Stelle weiter über die Seiten, bis eine gefunden wird, deren USED-Bit Null ist. Dies ist das Opfer. Seiten mit einem USED-Bit Eins werden auf Null gesetzt und übergangen. Opfer sind also Seiten, die während einer ganzen „Uhrumdrehung“ nicht referenziert wurden. Das Verfahren ist eine Approximation von LRU.

f) **Second-Chance-Replacement**

Verdrängt wird wie beim Clock-Algorithmus, allerdings erhält die Seite eine 2. Chance, wenn ihr DIRTY-Bit gesetzt ist, sie also im letzten Zyklus zwar nicht im Gebrauch war, davor aber beschrieben wurde. Solche Seiten müssen zurückgeschrieben werden, brauchen also einen 2. Plattenzugriff, ihr Austausch ist damit besonders teuer. Das Setzen des DIRTY-Bits erfolgt wie das Setzen des USED-Bits automatisch bei jedem Seitenzugriff.

---

1. Alternativ gibt es für häufigkeitssensitive Speicherverfahren auch das langsamer reagierende Swap-Verfahren, bei dem ein Element mit einem Zugriff den Platz tauscht mit dem Nachbar davor. Dies läßt sich auch mit einer sequentiellen Speicherung für Elemente gleicher Größe realisieren.

vor Uhrzeigerdurchgang		nach Uhrzeigerdurchgang	
USED-Bit	**DIRTY-Bit**	**USED-Bit**	**DIRTY-Bit**
0	0	Seite ersetzen	
0	1	0	0
1	0	0	0
1	1	0	1

Tab. 10–3 *Second-Chance-Replacement*

Generell gelten die oben vorgestellten Verfahren für Verdrängungen innerhalb eines Prozeßadressraums oder für den Adreßraum insgesamt. Im letzteren Fall stiehlt ein Prozeß einem anderen eine Seite. Hierzu mehr unten.

10.5 Leistungsbeurteilung von Paging-Verfahren

Die folgenden Ausführungen beruhen auf der Arbeit von Sleator/Tarjan: Armotized Efficiency of List Updates and Paging Rules, Comm. ACM, Vol. 28:2, Feb. 1985, pp. 202-208.

Als Verfahren betrachten wir

- Least recently used (LRU)
  ersetze die Seite, auf die die längste Zeit nicht zugegriffen wurde
- First-in, First-out (FIFO)
  ersetze die Seite, die am längsten im HS ist.
- Last-in-First-out (LIFO)
  ersetze die als letzte in den HS gebrachte Seite.
- Least frequently used (LFU)
  ersetze die am wenigsten benutze Seite.
- Längste Distanz in die Zukunft (MIN, Belady)
  ersetze die für die längste Zeit nicht gebrauchte Seite.

Alle diese Verfahren beruhen auf Seitenverdrängung nach Bedarf.

Als erste *Beobachtung* stellen wir fest, daß durch *vorausschauendes Paging* die Seitenfehlerrate nicht verringert werden kann.

Die ersten fünf Algorithmen heißen on-line Verfahren; Belady's Strategie ist off-line, damit nicht praktikabel und dient nur als Vergleichsmaßstab zum Optimum.

Wie gut kann nun ein on-line Algorithmus im Vergleich zu *MIN* sein, bzw. wie schlecht?

Sei *A* irgend ein Algorithmus, sei *s* eine Folge von *m* Seitenzugriffen, dann bezeichnen wir mit n_A die Anzahl von Seiten in *A*'s HS und mit $F_A(s)$ die Anzahl der Seitenfehler, die *A* bei der Folge *s* macht. Beim Vergleich von A mit *MIN* gelte ferner $n_A \leq n_{MIN}$.

Der erste Satz zeigt, wie schlecht die Algorithmen gegenüber *MIN* abschneiden.

Satz 10–1

Sei *A* irgend ein on-line Algorithmus. Dann gibt es beliebig lange Folgen *s*, so daß

$$F_A(s) \geq \frac{n_A}{n_A - n_{MIN} + 1} \cdot F_{MIN}(s)$$

Beispiel 10–2

Sei die Anzahl der Seiten n_A im HS für *A* sogar doppelt so groß wie für *MIN*, dann macht *A* unter Umständen trotzdem doppelt so viele Seitenfehler wie *MIN*:

$$F_A(s) \geq \frac{2x}{2x - x + 1} \cdot F_{MIN}(s)$$

Übung 10–1

Wie sieht es aus für $n_A = n_{MIN} = x$?

Beispiel 10–3

Wir konstruieren eine beliebig lange Folge für den Fall $n_A = 2 \cdot n_{MIN} = 6$. Dabei soll A $6/(6 - 3 + 1) = 3/2$-mal mehr Fehler machen als *MIN*.

Es genügt, eine Folge von $n_A = 6$ Zugriffen und eine Seitenmenge S von $n_A + 1$ Seiten (Nr. 1, 2, ..., 7) zu betrachten.

Für die ersten $n_A - n_{MIN} + 1 = 4$ Zugriffe erzeuge sowohl A als auch *MIN* je 4 Seitenfehler. Für die verbleibenden 2 Zugriffe suchen wir uns eine Seite aus S aus, die gerade nicht in A's HS ist (egal wie A's Strategie aussieht, eine solche Seite gibt es immer, weil S eine Seite mehr als n_A hat).

Bei *MIN* achten wir aber darauf, daß für diese $n_{MIN} - 1 = 2$ Zugriffe die Seiten vorher nicht verdrängt wurden. Wenn wir n_{MIN} Seiten für *MIN* im HS haben, können wir die Folge s immer so einrichten, daß die letzten $n_{MIN} - 1$ Zugriffe auf Seiten im HS erfolgen, egal wieviele Zugriffe wir vorher hatten.

Folge s	HS A	HS MIN
Anfangsbelegung	{1, 2, 3, 4, 5, 6}	{1, 4, 5}
7	verdränge z. B. 1 {2, 3, 4, 5, 6, 7}	{7, 4, 5}
1	verdränge z. B. 2 {1, 3, 4, 5, 6, 7}	{1, 4, 5}
2	...	{1, 4, 5}
3	...	{1, 4, 5}
4	...	{1, 4, 5}
5	verdränge z. B. 6 {1, 2, 3, 4, 5, 7}	{1, 4, 5}
6	verdränge z.B. ...	

Tab. 10–4 *Worst-case Folgen*

Bemerkung:

Gilt $n_A < n_{MIN}$, dann gibt es beliebig lange Folgen, so daß *A* ständig Seitenfehler erzeugt und *MIN* nie.

Übung 10–2

Für $n_A = 4$ gebe man Folgen für LRU, FIFO, LIFO, LFU an, möglichst mit 5 Seiten.

Andererseits können wir aber auch zeigen, daß sich *LRU* und *FIF*O, nicht schlechter als oben gezeigt verhalten (gilt nicht für *LFU* und *LIFO*), d. h. die Grenze oben ist bis auf eine additive Konstante „scharf“ wie man sagt.

Satz 10–2

Für beliebige Folgen *s* gilt

$$F_{LRU}(s) \leq \frac{n_{LRU}}{n_{LRU} - n_{MIN} + 1} \cdot F_{MIN}(s) + n_{MIN}$$

Für die Praxis heißt dies (n_{LRU} fest), daß *LRU* auch im schlechtesten Fall nie um mehr als einen multiplikativen Faktor schlechter ist als die optimale Strategie *MIN*, wobei der Faktor abhängt von der Seitenzahl im HS, den man bei *MIN* benützen würde, und höchstens gleich n_{LRU} sein kann.

10.6 Die „Longest Next Reference“ Strategie

Bei Wechselspeichern ist eine falsche Ersetzungsstrategie besonders teuer. Daher wird hier eine Strategie entwickelt, die gegenüber *LRU* mehr Historie, spez. die Gleichmäßigkeit der Zugriffe, berücksichtigt. Die Darstellung hier geht zurück auf Daniel A. Ford: Dismountable Media Management in Tertiary Storage Systems, IEEE TKDE, Vol. 9, No. 2, March-April 1997, pp. 350-1.

Die Idee ist, jeder Wechseleinheit (Platte, Band, Seite im HS) eine *Leerlaufspanne* (engl. *idle horizon*) zuzuordnen, die angibt, wie lange eine Einheit unreferenziert bleiben kann, bevor sie als „idle“ (nicht mehr

gebraucht) zu betrachten ist und Opfer einer Ersetzung werden sollte. Diese Spannen werden auf der Basis längerlaufender Beobachtungen dynamisch angepaßt, indem man Vergleiche mit der optimalen Strategie macht:

Wird eine Einheit angefordert, die gerade erst Opfer war, dann war die damalige Entscheidung ...

a) **richtig**, wenn seit der Verdrängung alle anderen Seiten (Einheiten) referenziert wurden
b) **falsch**, wenn es seit der Verdrängung Seiten (Einheiten) gibt, die *nicht* referenziert wurden.

Aus (a) folgert man, daß der Horizont des Opfers zu halbieren ist, damit es häufiger ersetzt wird. Aus (b) schließt man, daß der Horizont des Opfers zu verlängern ist, damit es vor Ersetzung besser geschützt wird.

Die Opferauswahl erfolgt grundsätzlich unter den Einheiten, deren letzte Referenz so weit zurückliegt, daß der Abstand zu „jetzt“ den Horizont übersteigt. Die Referenzzeit wird bei jedem Zugriff gesetzt (z. B. aus Zählregister).

Unter den möglichen Opfern wähle man das mit dem kleinsten Referenzzähler (der ältesten Zeit). Der Referenzzähler wird für alle Einheiten auf Null gesetzt, wenn eine Seite ersetzt wird (diese Seite bekommt dann den Wert 1).

10.7 Leistungsverhalten und Systemparameter

Haben ein oder mehrere Prozesse weniger Seiten zugeteilt, als sie ständig benötigen, dann steigt die Seitenfehlerrate steil an. Man spricht von *Seitenflattern* (*thrashing*), die Prozesse kommen insgesamt nur schlecht voran, da der Rechner mit unproduktivem Ein-/Auslagern von Seiten beschäftigt ist, was wiederum die Belastung noch mehr steigert, wenn die Ankunftsrate neuer Prozesse konstant ist. Abhilfe kann hier die Beobachtung der Seitenfehlerrate bringen, ggf. müssen Programme zwangsweise deaktiviert werden.

Die Seitenzuteilung kann (wie oben angedeutet)

- global erfolgen, d.h. Prozeß *i* nimmt Prozeß *j* eine Seite weg
- lokal erfolgen, wobei jeder Prozeß einen festen Anteil am HS hat, z. B. #Kacheln insgesamt / #Prozesse, oder
  #Kacheln inges. * (virtuelle Programmgröße / Summe der virtuellen Größen aller Programme).

Ein wirksames Mittel zum Beobachten des Laufzeitverhaltens ist der *working set*, d. h. die Menge der Seiten, die in einer bestimmten Zeitspanne, z. B. den Δ letzten Zugriffen (Δ = 10.000) aktiv waren.

Beispiel 10–4

...5 1 5 3 7 7 3 7 1 5 2 7 2 2 1 5 1 1 5 2 7 7 2 2 7 2 2 5

$WS(t_1) = \{1, 3, 5, 7\}$ $WS(t_2) = \{2, 5, 7\}$

Abb. 10–5 *Working Set (Δ = 10)*

Den schon oben erwähnten Zusammenhang zwischen Seitenfehlerrate, Größe des Workingsets und Lokalitätsverhalten des Programms zeigt Abbildung 10–6.

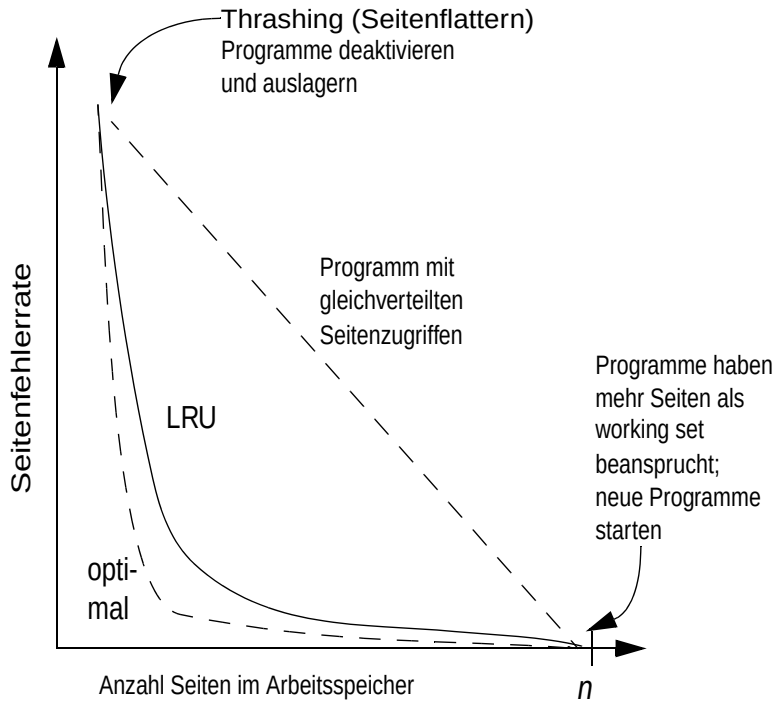


Abb. 10–6 *Seitenfehlerrate in Abhängigkeit von der Größe der Working Sets*

Ein anderer Einflußfaktor ist die Wahl der Seitenlänge.

- Kleine Seiten bieten eine gute Speicherausnutzung wegen geringer *interner Fragmentierung*, aber hohe Seitenfehlerraten (viele Unterbrechungen, die reinen Datentransfervolumina bleiben in der Summe ggf. gleich) und große Seitentabellen
- große Seiten haben eine schlechte Speicherausnutzung, geringe Seitenfehlerraten und kleine Seitentabellen.

Generell wird der Anwender keinen Einfluß auf Seitenlängen nehmen können. Vielmehr sind diese auch auf den Externspeicher abgestimmt (1 Seite = 1 Segment einer Plattenspur, usw.).

Besser als Platten wären Externspeicher, die keine Kopfpositionierzeit (früher sog. Trommeln) haben. Bei kleinen Seiten kommt auch immer ein hoher Anteil an *Drehwartezeit* (*latency time*) dazu.

Ferner begünstigen manche Programmstrukturen Paging, z. B. Stapel, andere nicht, z. B. große Streuspeichertabellen (Hashspeichertabelle).

Zu erwähnen ist noch, daß die Auslagerung gewisser Seiten verboten werden kann. Dazu gehören E/A-Puffer von Prozessen, die wegen einer E/A-Unterbrechung nicht aktiv sind. Ferner wird man BS-Monitor-Seiten vor Auslagerung schützen mit besonderem Augenmerk auf dem Paging Algorithmus!

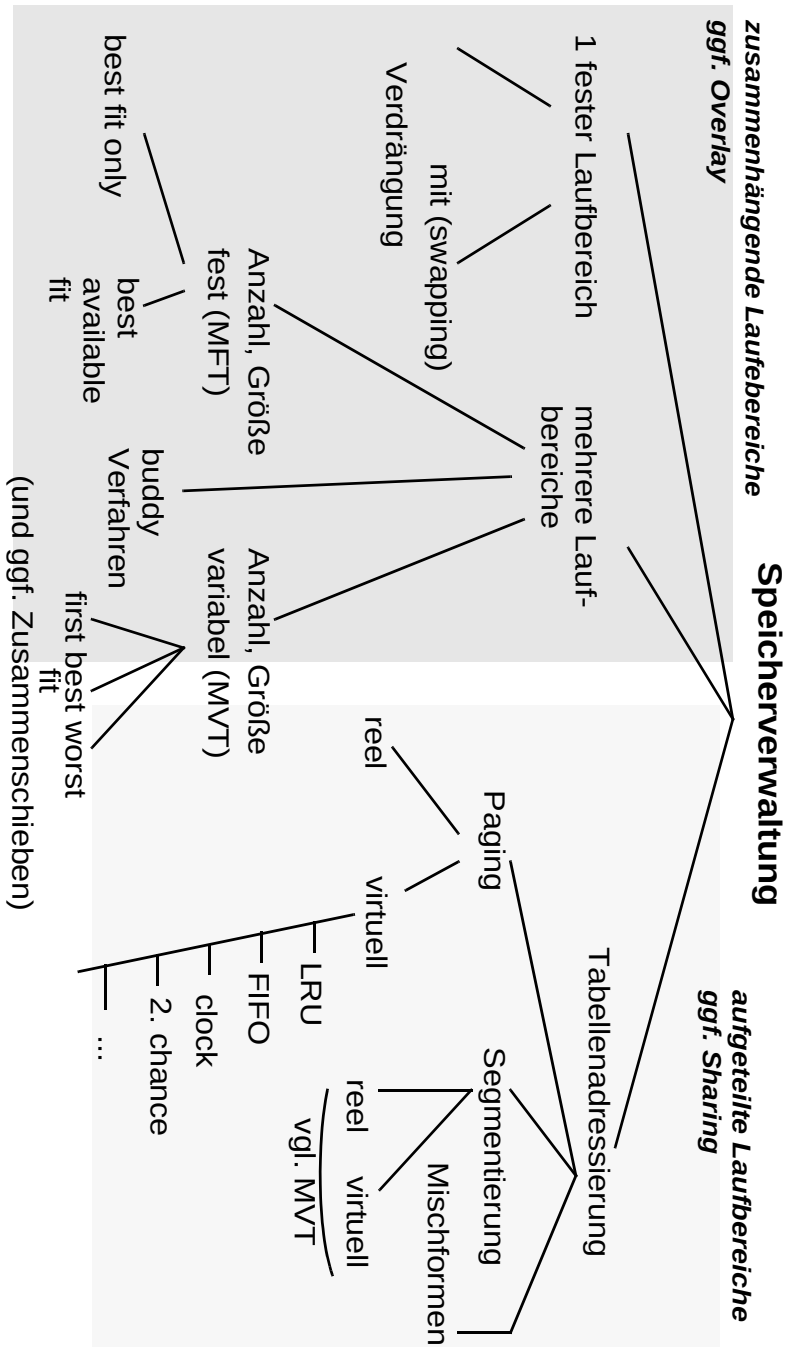


Abb. 10–7 *Übersicht Speicherverwaltung*

11 Auftragssteuerung

11.1 Aufgaben und Begriffe

Die Aufgabe der Auftragssteuerung ist die Festlegung einer Reihenfolge, in der die Auftragswarteschlange abgearbeitet werden soll. Diese Festlegung heißt engl. *scheduling*, die befolgte Strategie *scheduling discipline*.

Ferner kann die Abwicklung der Teilaufträge (Jobschritte) Aufgabe der Steuerung sein. Der konkrete Ablauf wird natürlich beeinflußt durch

- Job-Control-Anweisungen im Stapelbetrieb (z. B. Prioritäten, vgl. `nice`-Kommando in UNIX)
- Kommandoanweisungen (interaktiv) im Dialogbetrieb (z. B. suspend, resume)

Die Realisierung des Scheduling erfolgt durch einen Prozeß, den *scheduler* (nicht zu verwechseln mit dem *dispatcher*, der für die CPU-Umschaltung zuständig ist). Ferner kann es ggf. mehrere Auftragsabwicklerprozesse geben, im Dialogbetrieb ferner Terminalprogramme für die Ausführung der Steuerkommandos.

Die wichtigste Teilaufgabe der Auftragssteuerung ist die Festlegung einer Prozessor-Vergabestrategie, das CPU-Scheduling. Dabei gehen wir voraus, daß ein beliebiger *Entzug des Prozessors* möglich ist (*Job Preemption*).

11.2 Scheduling-Strategien

Optimierungsgesichtspunkte sind

- kleine Bedienzeiten für die Benutzer
- gute Auslastung aller (teuren) Betriebsmittel
- hoher Durchsatz (throughput) gemessen in Jobs/Zeiteinheit
- „faire“ Wartezeiten für alle Benutzer
- garantierte Abwicklung wichtiger Jobs
- ...

Dabei wird man

- Stapelbetrieb
- Mehrprogrammbetrieb
- Echtzeitbetrieb

unterschiedlich gewichten.

Alle Scheduling-Algorithmen beruhen auf expliziten oder impliziten Prioritätsentscheidungen, gegeben durch die Jobumgebung (geschätzte Laufzeit, Betriebsmittelanforderungen, Benutzerpriorität[1], ...) und Rechnerumgebung (Prozessorauslastung, Seitenfehlerrate, Warteschlangenlänge, ...). Wir untersuchen einige einfache Effekte an Beispielen.

Beispiel 11–1

Es liege einfacher Stapelbetrieb vor. Der Begriff *Turnaround* (Verweildauer) sei definiert als Zeitpunkt_Jobende - Zeitpunkt_Jobabgabe. Sei n die Anzahl der Jobs, T_i der Turnaround von Job i, dann ist die durchschnittliche Verweildauer (average turnaround)

$$\frac{1}{n} \cdot \sum_{i=1}^{n} T_i$$

1. Einem Gerücht zufolge fand man 1973 am MIT beim Abschalten der IBM 7094 einen Job mit niedriger Priorität, der 1967 abgegeben worden und noch nie gelaufen war.

Für die folgenden Beispieldaten seien die Laufzeitschätzungen (in Dezimalstunden) der Benutzer richtig.

Job No.	Arrival Time	Run Time
1	10.00	2.00 hrs
2	10.10	1.00 hrs
3	10.25	0.25 hrs

Tab. 11–1 *Beispieldaten*

Fall 1.1 First-come, first-served (FCFS)

Abarbeitung in der natürlichen Reihenfolge der Ankunft.

Job. No.	Arrival Time	Start Time	Finish Time	Turnaround
1	10.00	10.00	12.00	2.00 hrs
2	10.10	12.00	13.00	2.90 hrs
3	10.25	13.00	13.25	3.00 hrs
Summe Verweilzeit				**7.90 hrs**
durchschnittliche Verweilzeit = 2.63 hrs				

Tab. 11–2 *First-come, first served*

Man beobachtet eine hohe Varianz der Durchschnittszeiten als Folge des *Konvoi-Effekts* (lange Jobs behindern kurze Nachfolger).

Fall 1.2 Shortest job first (SJF)

Soweit möglich, werden kürzere Jobs vorgezogen (eine an Kopierern häufig angewandte Strategie, an Supermarktkassen gelegentlich anzutreffen).

Job. No.	Arrival Time	Start Time	Finish Time	Turnaround
1	10.00	10.00	12.00	2.00 hrs
2	10.10	12.25	13.25	3.15 hrs
3	10.25	12.00	12.25	2.00 hrs
Summe Verweilzeit				**7.15 hrs**
durchschnittliche Verweilzeit = 2.38 hrs				

Tab. 11–3 *Shortest job first*

Fall 1.3 Zuteilung mit „Hellseherfähigkeit“

Um 10.00 Uhr sei bekannt, daß in Kürze zwei kleine Jobs eintreffen.

Job. No.	Arrival Time	Start Time	Finish Time	Turnaround
1	10.00	11.50	13.50	3.50 hrs
2	10.10	10.50	11.50	1.40 hrs
3	10.25	10.25	10.50	0.25 hrs
Summe Verweilzeit				**5.15 hrs**
durchschnittliche Verweilzeit = 1.72 hrs (dabei CPU leer 0.25 hrs)				

Tab. 11–4 *With hindsight*

Beispiel 11–2

Wir vergleichen jetzt Mono- mit Multiprogramming für eine gegebene Jobfolge. Zusätzlich führen wir ein weiteres Gütekriterium ein, die gewichtete Verweilzeit (weighted turnaround) = Verweilzeit / Laufzeit.

Offensichtlich ist es besonders ärgerlich, wenn ein kurzlaufenden Job besonders lange im System warten muß.

Job No.	Arrival TIme	Run Time
1	10.0	0.3 hrs
2	10.2	0.5 hrs
3	10.4	0.1 hrs
4	10.5	0.4 hrs
5	10.8	0.1 hrs

Tab. 11–5 *Jobfolge*

Fall 2.1 Einfacher Stapelbetrieb

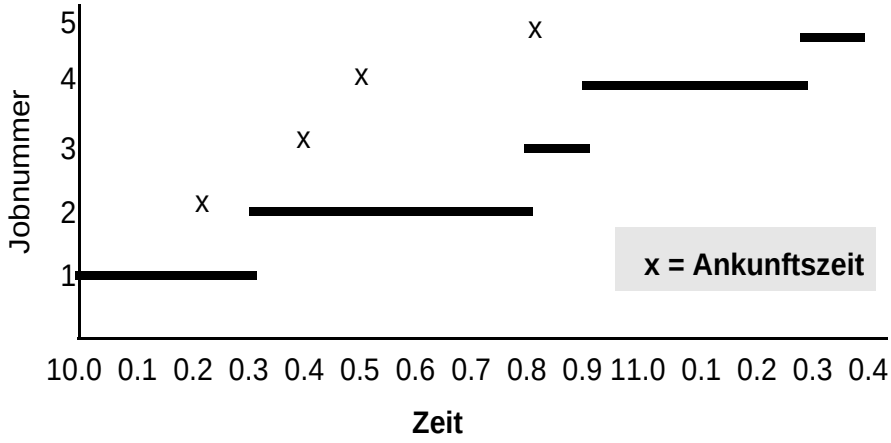


Abb. 11–1 *Jobabfolge im Stapelbetrieb*

Den Unterschied zum Multiprogramming sieht man sofort im Fall 2.2, wobei jeder Job sofort zugelassen wird und gleichgroße Zeitscheiben zugeteilt werden. E/A-Operationen werden nicht betrachtet. Insgesamt bleibt die Gesamtdauer natürlich gleich, in der Realität würde sie sogar geringfügig durch den Prozessorwechsel steigen.

Job No.	Ankunfts-zeit	Startzeit	Endzeit	Verweil-dauer	gewichtete Verweil-dauer
1	10.0	10.0	10.3	0.3	1.00
2	10.2	10.3	10.8	0.6	1.2
3	10.4	10.8	10.9	0.5	5.00
4	10.5	10.9	11.3	0.8	2.00
5	10.8	11.3	11.4	0.6	6.00
Summe				**2.8 hrs**	**15.20**
durchschnittliche Verweildauer T = 0.56 gew. durchschn. Verweildauer W = 3.04					

Tab. 11–6 *Beurteilung Stapelbetrieb*

Fall 2.2 Multiprogramming

Job No.	Joblänge	Startzeit	Endzeit	Verweil-dauer	gewichtete Verweil-dauer
1	0.3	10.0	10.4	0.4	1.33
2	0.5	10.2	11.35	1.15	2.3
3	0.1	10.4	10.65	0.25	2.5
4	0.4	10.5	11.4	0.9	2.25
5	0.1	10.8	11.1	0.3	3.00
Summe				**3.00 hrs**	**11.38**
durchschnittliche Verweildauer T = 0.6 gew. durchschn. Verweildauer W = 2.276					

Tab. 11–7 *Beurteilung Multiuserbetrieb*

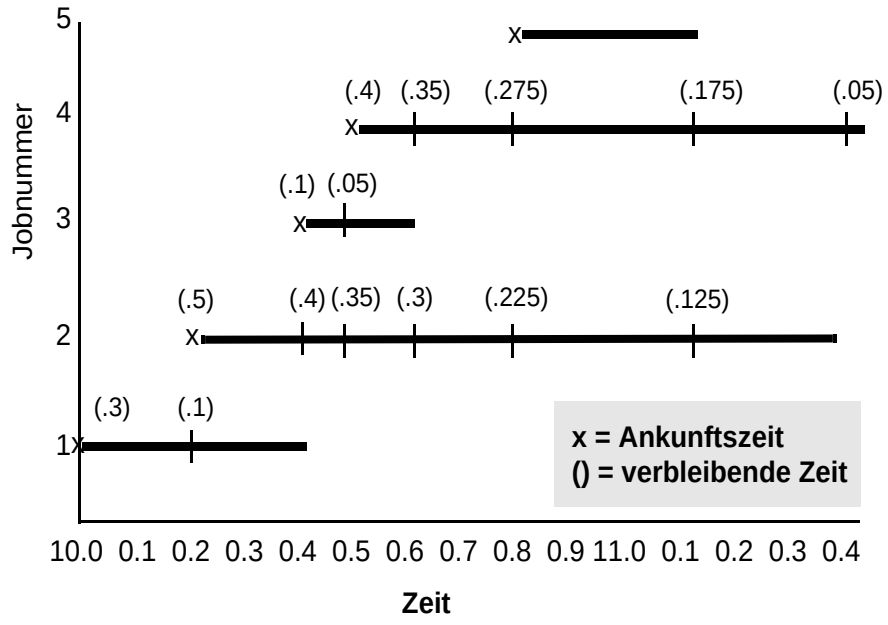
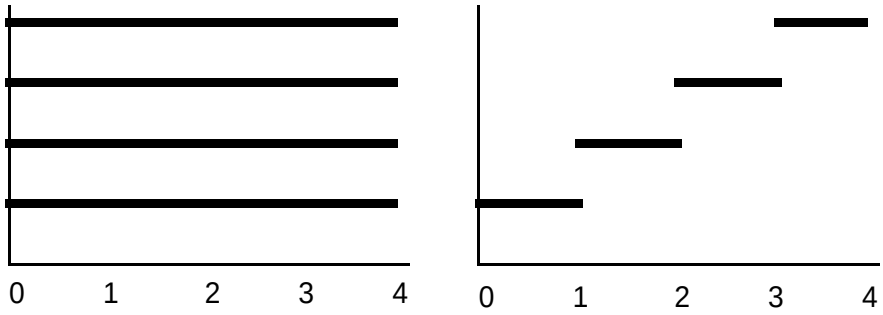


Abb. 11–2 *Jobabfolge im Multiuserbetrieb*

Wie man sieht, ist die durchschnittliche Verweildauer geringfügig von 0.56 auf 0.6 Stunden angestiegen. Dagegen ist die gewichtete Verweildauer um ca. 25% von 3.04 auf 2.276 gesunken. Subjektiv ist wichtig, daß man den Eindruck hat, es gehe sofort los (auch wenn man dann feststellt, daß es langsam voran geht).

Beispiel 11–3

In diesem Extrembeispiel betrachten wir 4 Jobs zu je 1 Stunde Laufzeit, die gleichzeitig ankommen.



(a) Jobs im Multiprogrammbetrieb (b) Jobs nacheinander

Abb. 11–3 *Multiprogrammbetrieb versus serielle Abfolge*

Abbildung 11–3 zeigt in diesem Fall, daß Multiprogramming auch Nachteile für alle Beteiligten bringen kann, weshalb Multiprogramming vorzugsweise mit den Wartezeiten einzelner Programme kombiniert werden sollte und meist im Jobmix besondere Vorteile ausspielen kann.

Das folgende Beispiel mit einem langlaufenden Programm und drei Kurzläufern zeigt eine solche Situation, wobei im Monoprogrammbetrieb gemeinerweise das langlaufende Programm zuerst gestartet wird.

Beispiel 11–4

Job No.	Joblänge	Startzeit	Endzeit	Verweil-dauer	gewichtete Verweil-dauer
1	3.0	10.0	14.0	4.0	1.3
2	0.5	10.0	11.5	1.5	3.0
3	0.25	10.0	11.0	1.0	4.0
4	0.25	10.0	11.0	1.0	4.0
Summe				**7.5 hrs**	**12.3**
durchschnittliche Verweildauer T = 1.88 gew. durchschn. Verweildauer W = 3.1					

Tab. 11–8 *Beispiel Mehrprogrammbetrieb*

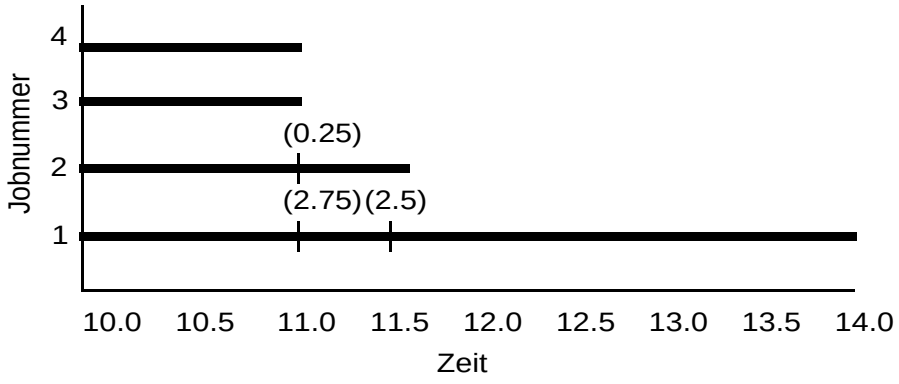


Abb. 11–4 *Multiprogramming*

Job No.	Joblänge	Startzeit	Endzeit	Verweil-dauer	gewichtete Verweil-dauer
1	3.0	10.0	13.0	3.0	1.0
2	0.5	10.0	13.5	3.5	7.0
3	0.25	10.0	13.75	3.75	15.0
4	0.25	10.0	14.0	4.0	16.0
Summe				**14.25 hrs**	**39.0**
durchschnittliche Verweildauer T = 3.56 gew. durchschn. Verweildauer W = 9.75					

Tab. 11–9 *Beispiel Monoprogrammbetrieb*

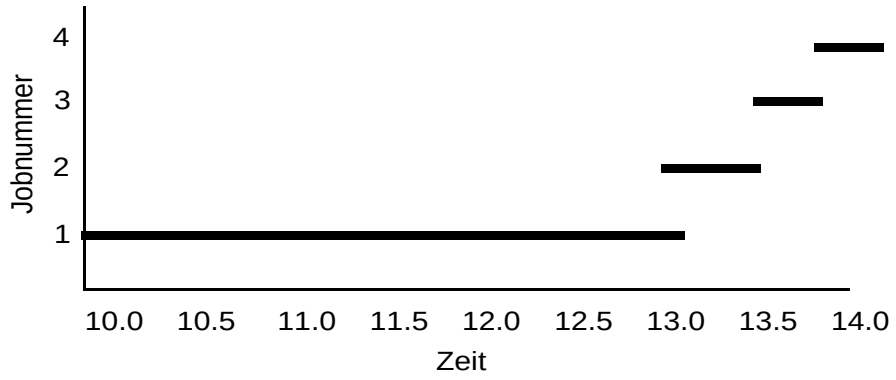


Abb. 11–5 *Monoprogramming*

11.3 Prioritätsbestimmung aus der Laufzeit

11.3.1 Shortest Job First (SJF) - nicht preemptiv

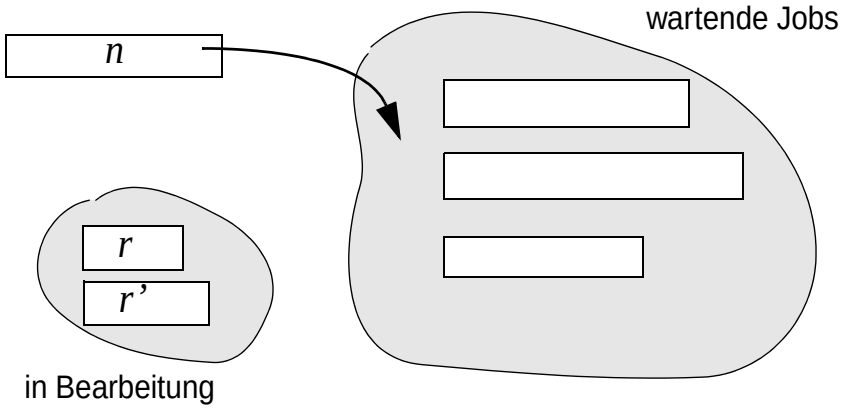
Aus der Auftragswarteschlange wird der Job mit der (vom Benutzer geschätzten) kürzesten Laufzeit gewählt und vollständig bearbeitet.

Jobs, die ihre geschätzte Laufzeit überschreiten, müssen „gestraft“ werden (abbrechen, überzogene Zeit teurer abrechnen, ...).

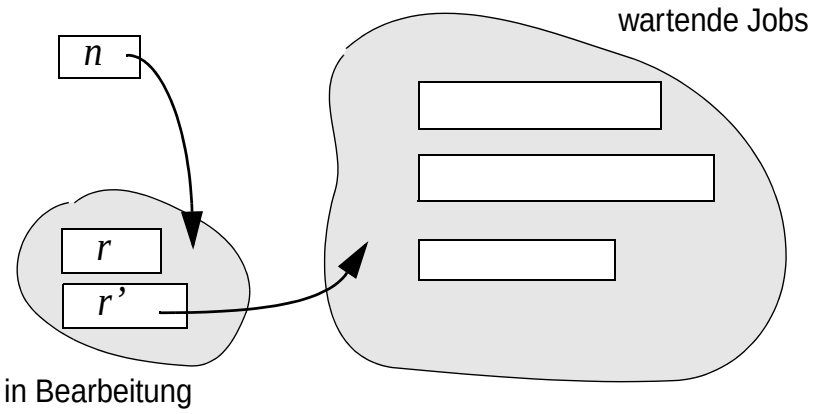
11.3.2 Shortest Job First - preemptiv (PSJF)

Der Entzug des Prozessors (die Preemption) geschieht nur bei Ankunft eines neuen Jobs. Daher heißt die Vergabepolitik auch *shortest remaining time first*. Zwei Fälle sind möglich in Abhängigkeit von der Laufzeitangabe *n* des neuen Jobs.

Fall 1: Laufzeit neuer Job *n* ≥ Restlaufzeiten *r*



Fall 2: Laufzeit neuer Job *n* < Restlaufzeiten *r*



11.3.3 Round-Robin (RR)

Das zugeteilte Zeitquantum q sei fest vorgegeben. Dann schneidet eine Zeitscheibe (ein Umlauf) eine Zeiteinheit q von der Restlaufzeit ab. Für $q \rightarrow 0$ haben wir eine quasi gleichzeitige Bedienung aller Jobs. Für $q \rightarrow \infty$ wird die Strategie zu FCFS.

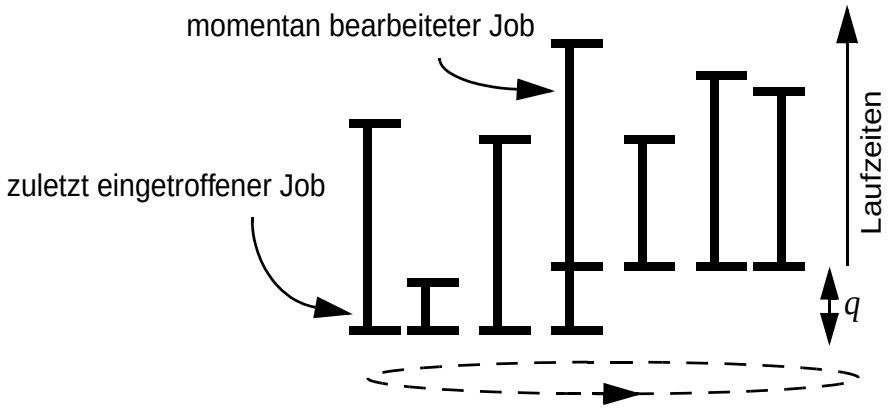


***Abb. 11–6** Round-Robin mit Zeitquantum q*

Die Abb. 11–6 zeigt den Effekt. Die grundsätzliche Organisation zeigt Abb. 11–7.

Neben der Organisationsform mit *einer* Warteschlange gibt es Alternativmöglichkeiten, etwa RR für interaktive (Vordergrund-)Jobs und FIFO für Batch (Hintergrund-)Jobs. Die Aufteilung foreground/background könnte dann z. B. 80/20 oder prioritätsgesteuert geregelt werden.

Weiterhin wären verschiedene Warteschlangen möglich (multi-level feedback), siehe Abb. 11–8.

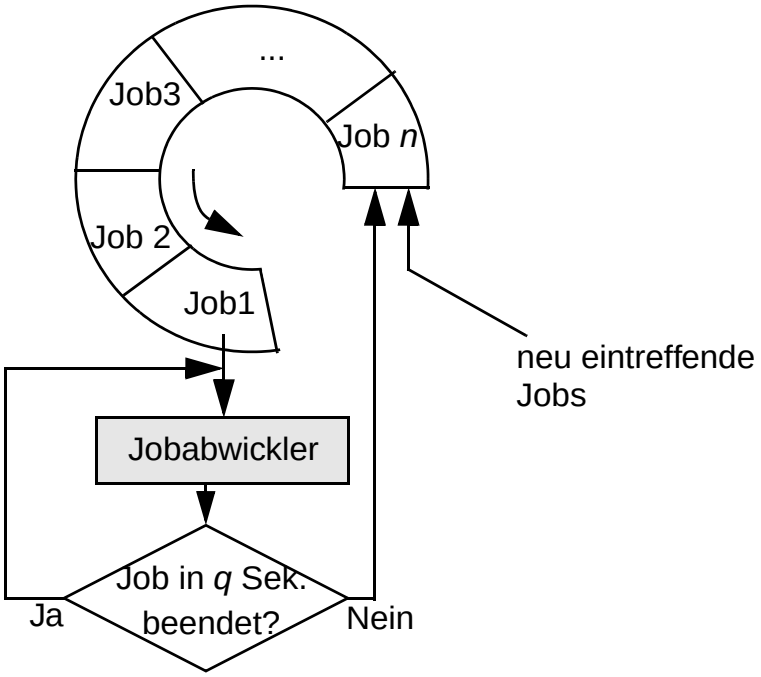


Abb. 11–7 *Grundprinzip einer RR-Verwaltungsstrategie*

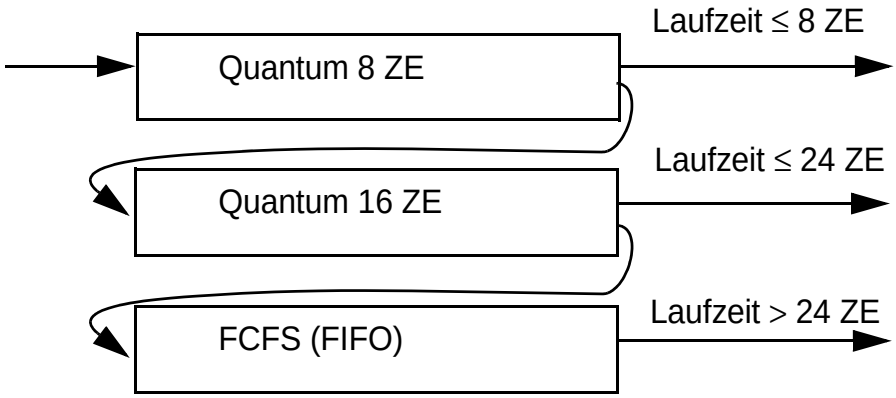


Abb. 11–8 *Auftragssteuerung mit drei Warteschlangen*

Wird für RR eine konstante Bearbeitungszeit/Job vorgesehen, d. h. ist die Zeitspanne zwischen den Bearbeitungsstarts von Job *i* und Job *i*+1 fest, dann wächst der Overhead für die RR-Wechsel und die Verwaltung bei starker Belastung nicht. Allerdings wird das Antwortverhalten träger, da die Zeitspanne bis zur erneuten Bearbeitung eines Jobs in der nächsten Runde wächst. Umgekehrt wächst der Overhead/ZE, wenn der Zeitanteil jedes Nutzers mit Zunahme der Teilnehmeranzahl verkleinert wird, damit Teilnehmer schneller wieder dran sind.

Wir beenden diese eher intuitiven Bemerkungen mit einem etwas gründlicheren Vergleich von FIFO- mit RR-Warteschlangen und machen dazu einen Ausflug in die Warteschlangentheorie.

11.4 FIFO- und RR-Warteschlangen

Die einfachsten und bestuntersuchtesten Warteschlangen sind die sog. M/M/1 Warteschlangen (FIFO), bei denen die Auftragserteilungen voneinander unabhängig sind, d. h. Phänomene, wo Kunden wegbleiben, wenn die Warteschlange zu groß wird, usw. werden nicht betrachtet. Die Anzahl der Auftragserteilungen in *t* hat als Zufallsvariable eine Poissson-Verteilung. Analog sind die Bearbeitungszeiten voneinander unabhängig (also betrachtet man nicht Fälle, bei denen ein System bei hoher Belastung langsamer arbeitet).

Sei α die mittlere *Ankunftsrate* (z. B. Jobs/s), β die mittlere *Abfertigungsrate* (z. B. Jobs/s). Den Wert $T_s = 1/\beta$ bezeichnet man dann als *Servicezeit* [s/Job].

Die Zeitspanne zwischen zwei aufeinanderfolgenden Auftragserteilungen habe die Dichtefunktion

$$\mathbf{f(}s\mathbf{)} = \alpha\, \mathbf{e}^{-\alpha s}$$

Man sagt, die *Zwischenankunftszeiten* sind exponentialverteilt. Der Erwartungswert dieser Verteilung ist $1/\alpha$.

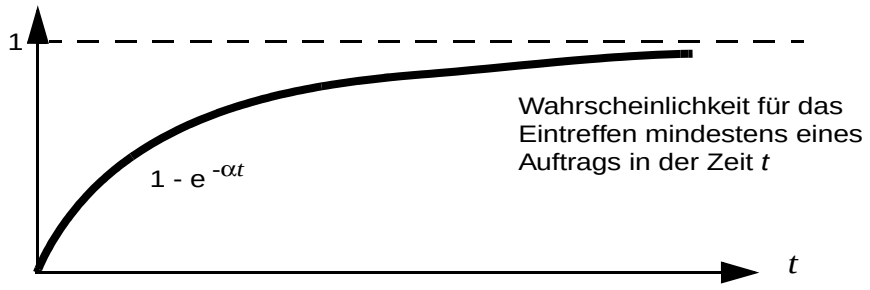


Abb. 11–9 *Kundenankunft als Poisson-Prozeß*

Mit $\rho = \alpha/\beta$ $(0 \leq \rho < 1)$ bezeichnen wir den *Auslastungsfaktor*. Wir werden zeigen, daß Kundensysteme, deren Auslastungsfaktor gegen 1 getrieben wird (z. B. durch eine Auftragsschwemme oder den Anstieg der Bedienzeiten durch weniger Servicestationen) explodierende Warteschlangen zur Folge haben. Mit anderen Worten: Leerzeiten sind ein unvermeidbarer, inhärenter Anteil von M/M/1 Kundensystemen.

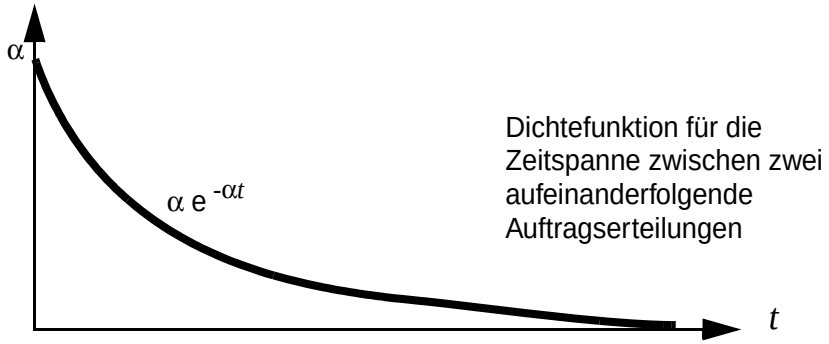


Abb. 11–10 *Zwischenankunftszeiten mit Erwartungswert 1/α*

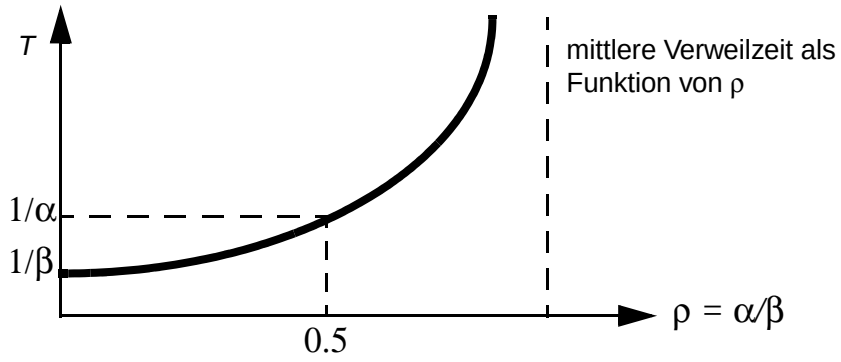


Abb. 11–11 *Explodierende Verweilzeiten bei Auslastungfaktor → 1*

Für diese gängige Form des Warteschlangensystems (Bäckerei, Postschalter, Kfz-Werkstatt) gilt nun *Little's Formel*.

Satz 11–1

> In einem M/M/1-Warteschlangensystem befinden sich im Mittel $n = \alpha T$ Aufträge im System mit T Erwartungswert für die *Verweilzeit*, d. h. *Wartezeit* + *Bedienzeit*.

Hierzu kann man die folgende Plausibilitätsüberlegung anstellen (Abbildung 11–12).

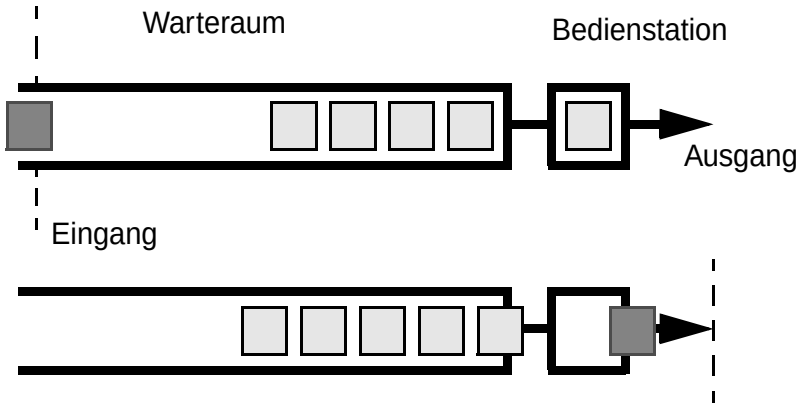


Abb. 11–12 *Situation zwischen Eintreffen und Abgang eines ausgezeichneten Kundens*

Wir wählen beliebig einen eintretenden Kunden und setzen ihm zur besseren Unterscheidung eine rote Mütze auf. Sein Eintrittzeitpunkt ist t_E. Nach einer Verweilzeit von T Zeiteinheiten verläßt dieser Kunde zum Zeitpunkt t_A das System, $T = t_A - t_E$.

Operiert man mit Mittelwerten, dann kommen α Jobs je Zeiteinheit an, in T Zeiteinheiten damit $n = \alpha\, T$ viele. Das sind aber genau die in der Warteschlange wartenden Kunden $(n - 1)$ zuzüglich dem einen in der Bedienstation, die nach dem Satz von Little immer im Mittel im System sind.

11.5 Jobverweilzeiten in einem Mehrprogramm-BS

Grundlage der Beurteilung ist die M/M/1-Warteschlange im Gleichgewicht, die eine Jobverwaltung modelliert, in die über einen längeren Zeitraum gesehen gleichviele Jobs hineingehen wie herauskommen, wobei ein Puffer (die Warteschlange) die Zufälligkeiten der Zwischenankunftszeiten und Bedienzeiten ausgleicht.

Eine Besonderheit ist, daß der Zustand eines M/M/1-Systems durch die Zahl k der sich in Warteschlange und Bedienstation befindenden Kunden beschrieben ist und sich vom Zustand k nur entweder in den Zustand k+1

(Zugang eines Kunden) oder in den Zustand k-1 (Abgang nach Abfertigung) bewegen kann. Ein solches Modell heißt *Geburten-Sterbe-System*.

Sei p_i die Wahrscheinlichkeit, daß das System im Zustand i ist (i Kunden da). Jetzt muß man die Wahrscheinlichkeiten für die Übergänge von i nach i+1, bzw. von i nach i-1 kennen, um das Gesamtsystem messen zu können (Abb. 11–13).

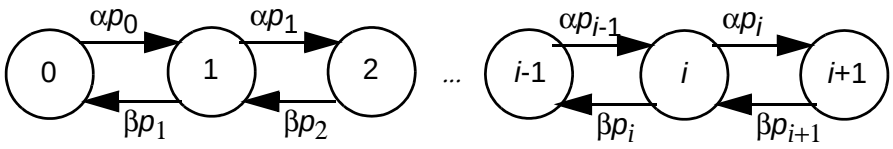


Abb. 11–13 *Zustandsdiagramm für Warteschlange mit einem Bediener (Übergangsraten als Kantenbeschriftung)*

Da die Ankunftsrate α bekannt ist, ist die Übergangsrate von 0 nach 1 (Übergänge je ZE) gerade $\alpha \cdot p_0$. Genauso hängt die Übergangsrate von i+1 nach i von der Bedienrate β und der Wahrscheinlichkeit für den Aufenthalt im Zustand i+1 (nicht i!) ab, d.h. sie ist $\beta \cdot p_{i+1}$.

Beispiel 11–5

Das System sei 20% der Zeit leer, 15% der Zeit wird ein Kunde bedient und 0 warten. Im Mittel kommen 40 Kunden/s an.

Damit ist die Übergangsrate von 0 nach 1 dann $40 \cdot 0{,}2 = 8$ Übergänge/s, von 1 nach 2 genau $40 \cdot 0{,}15 = 6$ Übergänge/s.

Befindet sich das System nun im Gleichgewicht, muß gelten: die Übergangsraten zwischen je zwei benachbarten Zuständen sind gleich. Wäre dies nicht so, wären etwa mehr Übergänge von 4 nach 5 zu beobachten als von 5 nach 4, dann würden sich langfristig immer mehr Kunden in Zuständen >4 ansammeln, d.h. die Gleichgewichtsannahme wäre verletzt.

Aus

$$\alpha p_0 = \beta p_1$$
$$\alpha p_1 = \beta p_2$$
$$\ldots$$
$$\alpha p_k = \beta p_{k+1}$$

folgt (aufgelöst nach p_1)

$$p_1 = \frac{\alpha}{\beta} \cdot p_0$$

$$p_2 = \frac{\alpha}{\beta} \cdot p_1 = \left(\frac{\alpha}{\beta}\right)^2 p_0$$

$$\ldots$$
$$p_k = \left(\frac{\alpha}{\beta}\right)^k p_0$$

oder mit dem Auslastungsfaktor $\rho = \alpha/\beta < 1$

$$p_k = \rho^k p_0$$

Wegen $\sum_{k=0}^{\infty} p_k = 1$ (das System ist immer in einem der Zustände $k = 0, 1, 2, \ldots$) gilt

$$\sum_{k=0}^{\infty} \rho^k p_0 = 1$$

und wegen der Summenformel der geometrischen Reihe

$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ für $x < 1$ folgt

$$p_0 \sum \rho^k = 1 \Leftrightarrow p_0 \frac{1}{1-\rho} = 1 \Leftrightarrow p_0 = 1 - \rho$$

und damit

$$p_k = \rho^k (1 - \rho)$$

Der Erwartungswert für die Anzahl der Kunden N im System ist dann

$$N = \sum_{k=0}^{\infty} k \cdot p_k = (1-\rho)\sum_{k=0}^{\infty} k \cdot \rho^k = \frac{\rho}{1-\rho}$$

Das Resultat von Little aus dem Jahr 1961 lautet $N = T \cdot \alpha$, d.h.die Anzahl der Kunden, die sich im Mittel im System befinden ist gleich der Ankunftsrate a mal der durchschnittlichen Verweilzeit T (Wartezeit + Bedienzeit) im System. J. Bentley drückt es in den Comm. ACM Vol. 29, No. 3 vom März 1986 so aus:

> Die durchschnittliche Anzahl von Dingen in einem System ist das Produkt der durchschnittlichen Rate, mit der die Dinge das System betreten, und der durchschnittlichen Zeit, die sie im System verbringen.

Für T folgt

$$T = \frac{N}{\alpha} = \frac{\rho}{1-\rho} \cdot \frac{1}{\alpha} = \frac{\frac{\alpha}{\beta}}{1-\frac{\alpha}{\beta}} \cdot \frac{1}{\alpha} = \frac{\alpha \cdot \beta}{\beta(\beta-\alpha)\cdot\alpha} = \frac{1}{\beta-\alpha}$$

Wir fassen zusammen:

- Die Wahrscheinlichkeit, daß i Aufträge im System sind, ist
$$p_i = \rho^i(1-\rho)$$
mit $\rho = \alpha/\beta$.
- Der Erwartungswert für die Anzahl der Kunden im System ist dann
$$n = \sum_{i=0}^{\infty} i p_i = \frac{\rho}{1-\rho}$$
wobei n nur definiert ist für $\alpha < \beta$. Für $\rho \to 1$ geht $n \to \infty$.

- Aus Little's Formel folgt die mittlere Verweilzeit

$$T = \frac{n}{\alpha} = \frac{1}{\beta - \alpha}$$

11.6 RR als M/M/1-Warteschlangensystem

Wir betrachten eine Zeitscheibenzuteilung je Job von s [z. B. in Sekunden] und eine Gesamtbearbeitungsdauer $1/\beta$. Die Ankunftsrate (neuer) Jobs sei α.

Dann ist die Anzahl der benötigten Zeitscheiben $k = \frac{1}{\beta \cdot s}$ und die mittlere Verweilzeit (die unabhängig von RR ist) wird zu

$$T(k) \approx k\left(s \cdot \frac{\rho}{1-\rho} + s\right) = \frac{k \cdot s}{1-\rho} = \frac{1}{\beta - \alpha}$$

Diese Näherung stammt von L. Kleinrock aus dem Jahr 1964.

Anwendungen

Beispiel 11–6

Die Ankunftsrate sei $\alpha = 3$ Jobs/s, die Bearbeitungsrate $\beta = 5$ Jobs/s, damit 60% Auslastung. Die Bearbeitungszeit je Job ist $1/\beta = 0{,}2$ s, die mittlere Verweildauer $T = 1/(\beta - \alpha) = 0.5$ s. Die Anzahl der Jobs im System ergibt sich zu $n = T\alpha = 3/2 = 1{,}5$ Jobs.

Erhöht man jetzt die Auslastung um 20% auf 80%, d. h. $\alpha = 4$ Jobs/s und $\beta = 5$ Jobs/s, dann wird $T = 1/(\beta - \alpha) = 1$ s. Die Verweildauer hat sich verdoppelt! Die Anzahl der Jobs im System ist jetzt $n = \alpha T = 4$ Jobs.

Für den Fall $\alpha = 3$ mit RR und einem Zeitscheibenanteil von $s = 0{,}05$ s benötigen wir $k = 1/(\beta \cdot s) = 1/(5 \cdot 0{,}05) = 4$ Zeitscheiben. Die mittlere Verweildauer bleibt bei 0,5 s.

Beispiel 11–7

Es gebe zwei Arten von Jobs (kleine mit $\beta_1 = 10$, große mit $\beta_2 = 5$ Jobs/s). Wir verwenden den Satz, wonach zwei überlagerte Poissonprozesse wieder einen Poissonprozeß ergeben.

Die Ankunftsraten seien $\alpha_1 = 3$, $\alpha_2 = 1$, $\alpha = \alpha_1 + \alpha_2 = 4$.

Die durchschnittliche Bearbeitungszeit je Job ist $1/\beta = \alpha_1/(\alpha_1 + \alpha_2)\cdot 1/\beta_1 + \alpha_2/(\alpha_1 + \alpha_2)\cdot 1/\beta_2 = 3/4 \cdot 1/10 + 1/4 \cdot 1/5 = 5/40 = 1/8$ s.

Die Auslastung errechnet sich zu $\rho = \rho_1 + \rho_2 = \alpha_1/\beta_1 + \alpha_2/\beta_2 = 3/10 + 1/5 = 5/10 = \alpha/\beta$. Die durchschnittliche Verweilzeit ist $T = 1/(\beta - \alpha) = 1(8 - 4) = 1/4$ s. Die Anzahl der Jobs im System ist $n = T\alpha = 1/4 \cdot 4 = 1$ Job.

Bei getrennter Bearbeitung werden die Verweilzeiten zu:

$T_1 = 1/(\beta_1 - \alpha_1) = 1/(10 - 3) = 1/7$ s, $\rho_1 = \alpha_1/\beta_1 = 3/10 = 0{,}3$

$T_2 = 1/(\beta_2 - \alpha_2) = 1/(5 - 1) = 1/4$ s, $\rho_2 = \alpha_2/\beta_2 = 1/5 = 0{,}2$.

Betrachten wir nun eine RR-Bearbeitung mit $s = 0{,}1$ s Zeitscheibenanteil.

Die Anzahl der Zeitscheiben ergibt sich zu $k_1 = 1/(s\cdot\beta_1) = 1$ (kleine Jobs) und $k_2 = 1/s\cdot\beta_2 = 2$ (große Jobs).

Die Verweildauern sind:

$T_1(k_1) = k_1\cdot s/(1 - \rho) = 0{,}1 / 0.5 = 1/5$ s

$T_2(k_2) = k_2\cdot s/(1 - \rho) = 0{,}2 / 0.5 = 2/5$ s

$T(k) = k\cdot s/(1 - \rho) = 1/\beta : (1 - \rho) = 1/8 : 1/2 = 1/4$ s (wie FIFO)

Beurteilung

Bei FIFO zerfällt T für kleine Jobs in $T_s = 1/\beta_1 = 0{,}1$ s und $T_w = T - T_s = 0{,}15$ s. Für die großen Jobs gilt $T_s = 1/\beta_2 = 0{,}2$ s und $T_w = T - T_s = 0{,}05$ s. Damit sind die kleinen Jobs eklatant benachteiligt.

Bei RR ergibt sich für kleine Jobs $T_1(k_1) = 0{,}2$ s ($T_s = 0{,}1$) und $T_2(k_2) = 0{,}4$ s ($T_s = 0{,}2$), d. h. die Verweilzeiten sind proportional zu den Servicezeiten, man kann von einem „fairen“ Verhalten sprechen.

11.7 Abarbeitung von Plattenaufträgen

Fast alle Rechenjobs verlangen Plattenzugriffe für die Ein-/Ausgabe. Die eigentliche Verarbeitungszeit beim Zugriff setzt sich zusammen aus

- Kopfbewegungszeit (*seek time*, inkl. *head settle time*)
- Drehwartezeit (Latenzzeit, *latency time*)
- Übertragungszeit (*transfer time*)

Benötigt ein Prozeß eine E/A von der Platte, wird er einen Systemaufruf an den BS-Kern schicken mit den folgenden Angaben

- wird Ein- oder Ausgabe gewünscht?
- Plattenadresse (log. Blocknummer, Gerätenummer, ggf. Zylinder, Oberfläche, Sektor)
- Hauptspeicheradresse
- Übertragungslänge (Bytes, Wörter, Blöcke)

Bei hoher Auslastung und großer Platte stehen häufig mehrere Plattenaufträge in einer Warteschlange und es ergibt sich die Frage der Abarbeitungsreihenfolge (*disk scheduling discipline*).

FCFS - First-Come First-Serve

Für Plattenzugriffe einfach, im Prinzip auch fair, aber ggf. nicht effizient.

Beispiel: Gegeben sei die Folge von Zugriffswünschen (Spurnummern) und die Ausgangsspur 53. Bei FCFS erhalten wir „heftige“ Kopfbewegungen mit einer „Wegstrecke“ von zusammen *640 Spuren*.

Warteschlange: 98, 183, 37, 122, 14, 124, 65, 67 Start bei 53

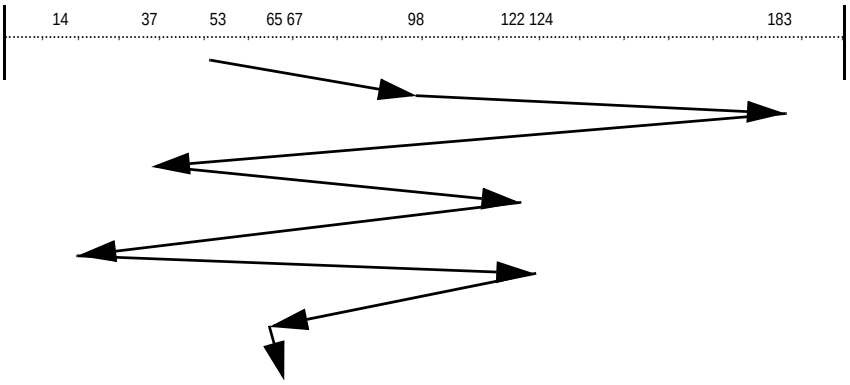


Abb. 11–14 *FCFS disk scheduling*

SSTF - Shortest Seek Time First

Wir wählen jetzt die Anforderung als nächste aus, die den kürzesten Zugriffsweg von der gegenwärtigen Position hat.

Im Beispiel unten beträgt der Gesamtweg jetzt nur noch *236 Spuren*. SSTF entspricht SJF (shortest job first), analog hat man hier das Problem des Aushungerns.

Warteschlange: 98, 183, 37, 122, 14, 124, 65, 67 Start bei 53

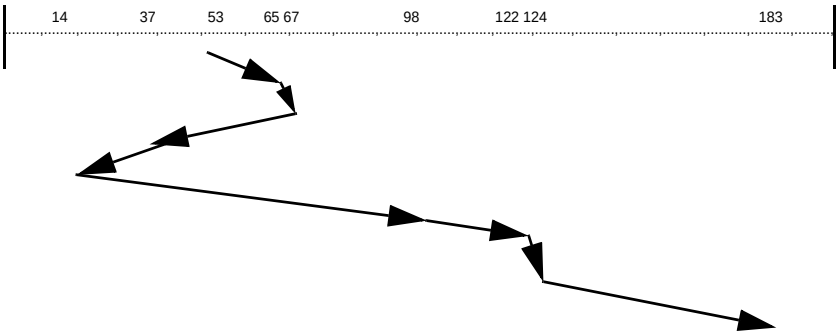


Abb. 11–15 *SSTF disk scheduling*

Scan Strategie

Der Kopf bewegt sich immer in eine Richtung bis ans Ende (Spur 0 bzw. N_{max}) und wechselt dann die Richtung. Dies entspricht den Fahrten eines Aufzugs (*elevator strategy*). Neue Anforderungen werden im gleichen Zug erledigt, wenn sie voraus liegen, sonst erst auf der Rückfahrt.

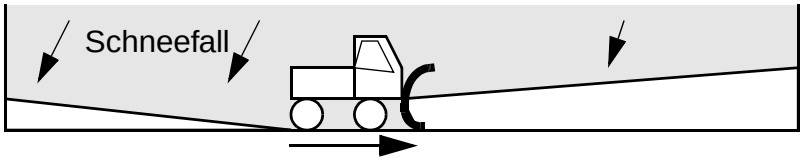


Abb. 11–16 *Schneepflug-Phänomen (hier: oszillierend)*

Der Nachteil ist, daß sich viele Aufträge am entfernten Ende hinter dem Kopf sammeln und diese sehr lange warten müssen (vgl. auch oszillierenden Schneepflug, Abb. 11–16)

Warteschlange: 98, 183, 37, 122, 14, 124, 65, 67 Start bei 53

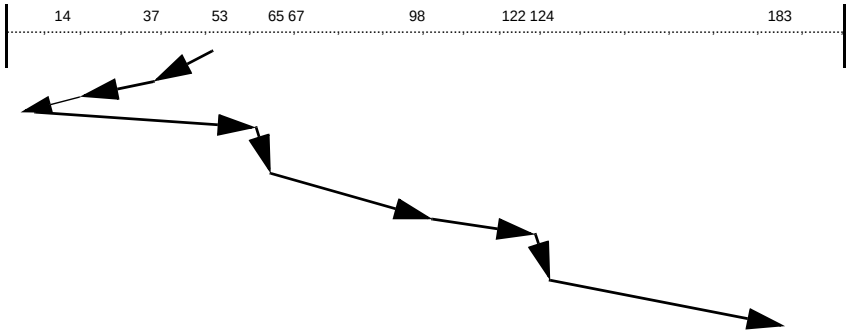


Abb. 11–17 *SCAN disk scheduling*

C-Scan und C-Look

Beim *C-Scan* (*Circular Scan*) kehrt der Kopf ohne Halt zu Spur 0 zurück. Damit wirkt die Strategie wie wenn Spur 0 mit Spur N_{max} kreisförmig verbunden wäre.

Bei *Look* oder *C-Look* hören wir mit der Fahrt auf, wenn keine weiteren Aufträge in die selbe Richtung vorliegen. *C-Look* im Beispiel unten ergibt *Distanz 322*, *Look* mit den Voraussetzungen von Abb. 11–17 ergibt *Distanz 208*.

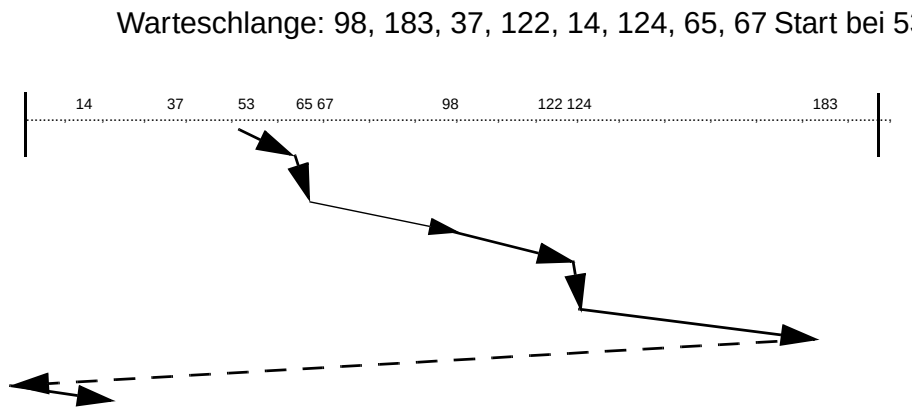


Abb. 11–18 *C-LOOK disk scheduling*

Welche Strategie soll man wählen?

Das hängt von vielen Faktoren ab, z. B. Verteilung der Daten- und Indexblöcke, von der Verzeichnisstruktur. Ebenfalls möglich wäre eine Sektoroptimierung zur Reduzierung der Latenzzeit mittels einer Sektorwarteschlange, d.h. die Anfragen werden nach Sektornummern aufgeteilt.

Innerhalb eines Zylinders werden die Aufträge so abgearbeitet, daß gerade die als nächste drankommen, deren Sektoren in Kürze sich unter dem Kopf durchbewegen.

Arbeitet die Platte mit einem RAM-Cache, in den eine ganze Spur eingelesen wird - beginnend mit dem Sektor, der gerade unter dem Kopf vorbeikommt - bringt *sector queuing* nichts.

Ferner ... verzögertes Schreiben (*deferred writing*), *versetzte Sektoren*, *gespiegelte Platten*, *journaled files*, RAID-Technik, CD-ROM, CD-R, CD-RW, DVD, ...

Literatur

Lernmaterial und Online-Stoffsammlungen der FG Betriebssysteme der Ges. für Informatik (GI) unter

```
http://www.nt.fh-koeln.de/vogt/lehre_bs.html
```

Textbücher

[1] Bach, Maurice J.: *The design of the UNIX operating system*. Prentice Hall US, 1986, 903 pp. Illustrations hardback, £30.50, ISBN 0132017997

[2] Chow, Randy; Johnson, Ted: *Distributed operating systems*. Addison-Wesley Long Hi Ed, 1997. 550 pp. paperback, £28.95, ISBN 0201498383

[3] Tanenbaum, Andrew S.: *Distributed operating systems*. 1994. 496 pp. paperback, £26.95, Prentice Hall IPE, ISBN 0131439340

[4] Tanenbaum, Andrew S.: *Modern operating systems*. 1992. 1055 pp. paperback, £26.95, Prentice Hall US, ISBN 0135957524

[5] Stallings, William: *Operating systems*. Prentice Hall IPE, 2nd ed 1995. 608 pp. paperback, £25.95, ISBN 0131809776

[6] Silberschatz, Abraham; Galvin, Peter: *Operating system concepts*. Addison-Wesley ISE, 4th ed 1993. 704 pp. Bibliography, index paperback, £25.50, ISBN 0201592924

[7] Nutt, Gary J.: *Operating systems: a modern perspective*. Benjamin Cummins, 1997. 750 pp. paperback, £26.95, ISBN 0805312951

[8] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. 2., verb. Aufl. 1995. 877 S., DM 98.00, Hanser, Prentice-Hall, ISBN 3446184023

[9] Zimmermann, Christoph; Kraas, Albrecht W.: *MACH. Konzepte und Programmierung*. 1993. 192 S., DM 78.00, Springer, ISBN 3540558063

[10] Werner, Dieter: *Theorie der Betriebssysteme. Eine Einführung in die Koordinierung paralleler Prozesse*. 1992. 252 S., DM 48.00, Hanser, ISBN 3446165479

[11] Spies, Peter P.; Baumgarten, Bernd: *Betriebssysteme – Konzepte, Methoden und Modelle*. 1997. 600 S., DM 69.90, Oldenbourg, ISBN 3486243853

[12] Brause, Rüdiger: *Betriebssysteme – Grundlagen und Konzepte*. 1997. 350 S., DM 49.90, Springer, ISBN 3540629297

[13] Weinländer, Markus: *Entwicklung paralleler Betriebssysteme*, m. Diskette (3 1/2 Zoll) Design und Implementierung von Multithreading-Konzepten in C Plusplus. 1995. 262 S., DM 78.00, Vieweg, ISBN 3528054972

[14] Tanenbaum, Andrew S.: *Verteilte Betriebssysteme*. 1995. 704 S., DM 98.00, Prentice-Hall, ISBN 393043623X

[15] Leopold, Claudia: *Parallel and Distributed Computing. A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, New York, 2001

[16] Dijkstra, E. W.: Cooperating sequential processes. In *Programming Languages.* F. Genuys (ed.), New York, Academic Press, pp. 43-112, 1968.

Artikel

[17] Flynn, M. J.: *Some computer organizations and their effectiveness*. IEEE Transactions on Computers, 21(9): 948-960, Sept. 1972.

[18] Lamport, L: *The synchronization of independent processes*. Acta Inf. 7 (1976), 15-34.

[19] Gottlieb, Allan; Kruskal, Clyde P.: *Coordinating parallel processors: A partial unification*. Computer Architecture News, 9(6):16-24, October 1981.

[20] Hansen, Per Brinch: *Concurrent Programming Concepts*. ACM Computing Surveys, Vol. 5, No. 4, December, 1973. http://www.acm.org/pubs/articles/journals/surveys/1973-5-4/p223-hansen/

[21] Hoare, C. A. R.: *Monitors: An Operating System Structuring Concept*. Communications of the ACM, Vol. 17, No. 10. October 1974, pp. 549-557 http://www.acm.org/classics/feb96/

[22] Buhr, Peter A.; Fortier, Michael; Coffin, Michael H.: *Monitor Classification*. ACM Comput. Surveys, Vol. 27, No. 1, March 1995

[23] Hansen, Per Brinch: *Java's Insecure Parallelism*. SIGPLAN Notices 34(4): 38-45 (1999)

[24] Hansen, Per Brinch: *Edison-a Multiprocessor Language*. Software – Practice and Experience 11(4): 325-361 (1981)

[25] Feldman, Jerome A.: *High level programming for distributed computing*. Communications of the ACM, 22(6):353-368, June 1979.

[26] Hoare, C. A. R.: *Communicating Sequential Processes*. Communications of the ACM, 21(8) 1978

[27] Sleator/Tarjan: *Armortized Efficiency of List Updates and Paging Rules*. Communications of the ACM, Vol. 28:2, Feb. 1985, pp. 202-208.

[28] Ford, Daniel A.: *Dismountable Media Management in Tertiary Storage Systems*. IEEE TKDE, Vol. 9, No. 2, March-April 1997, pp. 350-1.

Index

P

R

S

T

U

V

W

X

Z