Systemprogrammierung

Skript zur Vorlesung an der RWTH Aachen

Otto Spaniol
Claudia Linnhoff-Popien
Peter Reichl
Marko Schuba

Einführung

1 Betriebssysteme in Geschichte und Gegenwart, Raum und Zeit

1.1 Der Begriff des Betriebssystems

Das **Betriebssystem** eines Rechners ist ganz allgemein ein Programm, das sich zwischen dem Nutzer und der Hardware einordnen läßt.
Ein Betriebssystem soll dem Benutzer eine Umgebung bereitstellen, in der er Programme ausführen kann, und zwar in komfortabler und effizienter Weise. Um diese Aufgabe zu erfüllen, muß ein Betriebssystem die ihm zur Verfügung stehenden Hardware-Ressourcen (wie z. B. CPU-Zeit, Speicher ...) verwalten und dem Anwender(programm) unter Beachtung gewisser Kriterien wie Effizienz oder Fairneß zur Verfügung stellen.

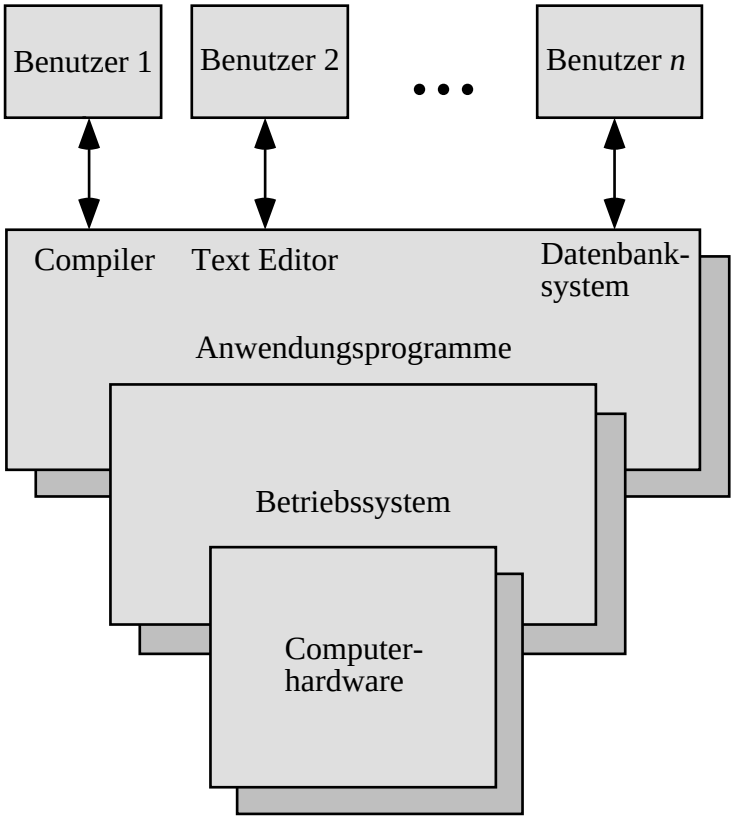
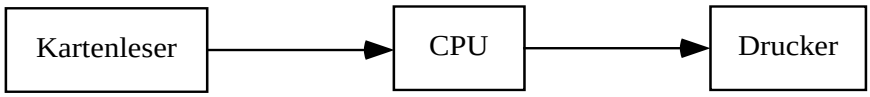


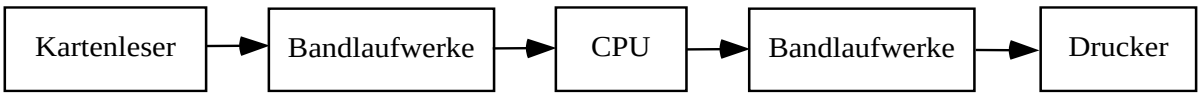
Abb. 1.1: Bestandteile eines Computersystems

1.2 Zur Geschichte der Betriebssysteme

Die ersten Computer kannten noch kein Betriebssystem. Sämtliche Aufgaben, die heutzutage ein Betriebssystem übernimmt, mußten vom Programmierer manuell durchgeführt werden. Beispielsweise erfolgte das Laden eines Programms durch direktes Eingeben binärer Befehle in den Speicher oder (schon komfortabler) über Lochkarten. Später wurden diese Aufgaben durch zusätzliche Hard- und Software erleichtert (Lochkartenleser zum Laden und Speichern, Assembler zur Programmierung etc.). Trotzdem war die Zeitdauer, die allein für die Vorbereitung eines Programmlaufs notwendig war, noch sehr groß. Dies hatte zur Folge, daß die CPU der damaligen Rechner nur ineffizient genutzt werden konnte. Mit dem Aufkommen der ersten höheren Programmiersprachen (wie FORTRAN und COBOL) wurde es möglich, Programme, die z. B. den gleichen Compiler benötigten, direkt hintereinander ablaufen zu lassen. Dadurch konnte ein erneutes Laden des Compilers vermieden werden. Dieser sogenannte **Batch-Betrieb** wurde zuerst manuell und später automatisch über einen residenten Monitor durchgeführt.
Trotz dieser Verbesserungen blieb die CPU-Auslastung immer noch gering. Der Grund lag in der Geschwindigkeitsdiskrepanz zwischen der (elektronischen) CPU und den (oft mechanischen) E/A-Geräten. Dieser Unterschied konnte jedoch durch eine Entkopplung der CPU von den langsamen E/A-Geräten verringert werden (**off-line-Betrieb**). Zu diesem Zweck wurden modernere Magnetbänder benutzt, um z. B. Ausgaben der CPU an den Drucker (schnell) zwischenzuspeichern. Ein späterer Wechsel des Bandes ermöglichte so den (langsamen) Ausdruck der Daten bei gleichzeitiger Weiterarbeit der CPU mit einem neuen Magnetband.



(a) On-line Operationen mit I/O-Geräten



(b) Off-line Operationen mit I/O-Geräten

Abb. 1.2: Operationsformen mit I/O-Geräten

Mit der Entwicklung von Plattenspeichern entstand eine neue Art der Abarbeitung von Jobs: das sogenannte **Spooling** (**S**imultaneous **p**eripheral **o**peration **o**n-**l**ine). Die Platte diente dabei als Puffer für Ein- und Ausgabeoperationen. So war es möglich, daß die CPU weiterarbeitete, während z. B. Daten von der Platte gedruckt oder von Kartenlesern auf Platte gespeichert wurden. Spooling führte schließlich dazu, mehrere Jobs "gleichzeitig" ausführen zu können (**Multiprogramming**-Betrieb), und zwar in dem Sinn, daß der aktuell in der CPU arbeitende Job diese freigab, sobald er eine E/A-Operation machte, mit der eine längere Wartezeit verbunden war. Die CPU konnte dann in der Zwischenzeit von einem anderen Job benutzt werden, wodurch sich Leerlaufzeiten der CPU nahezu vollständig vermeiden ließen.
Mit dem **Time-Sharing** (oder **Multitasking**) wurde eine Variante des Multiprogramming entwickelt, bei dem ein Wechsel der Jobs jeweils nach Ablauf einer gewissen

Zeitspanne (Zeitscheibe) erfolgte. Dadurch ließ sich z. B. die CPU-Zeit gerechter unter den Jobs aufteilen, da eine Monopolisierung des Systems durch rechenintensive Jobs vermieden werden konnte. Time-Sharing-Betriebssysteme eigneten sich insbesondere für den **Multiuser**-Betrieb eines Rechners: Mehrere Benutzer konnten damit "gleichzeitig" eine CPU interaktiv für ihre Arbeit nutzen.
In den letzten Jahren zeichnen sich neben der Verbesserung der Einprozessor-Betriebssysteme zwei weitere Trends ab: Zum einen stellt die Entwicklung von **Multiprozessorsystemen** (Parallelrechner) neue Anforderungen an Betriebssysteme (wie kann z. B. ein einzelner Job automatisch parallelisiert werden?). Zum anderen ermöglicht die zunehmende Vernetzung der Rechner die Nutzung verteilter Ressourcen, für die **verteilte Betriebssysteme** notwendig werden.

Ein Beispiel für die Fortschritte im Bereich der Betriebssysteme ist das Betriebssystem **MULTICS**, welches am Massachusetts Institute of Technology (MIT) für den dortigen Großrechner entwickelt wurde. Später gebaute Mini- und Microcomputer nutzten die Ideen von MULTICS, und so entstand das verbreitete UNIX-Betriebssystem (Bell Labs). Viele der wesentlichen Ideen wurden z. B. auch in Microsoft Windows oder IBM OS/2 verwendet.

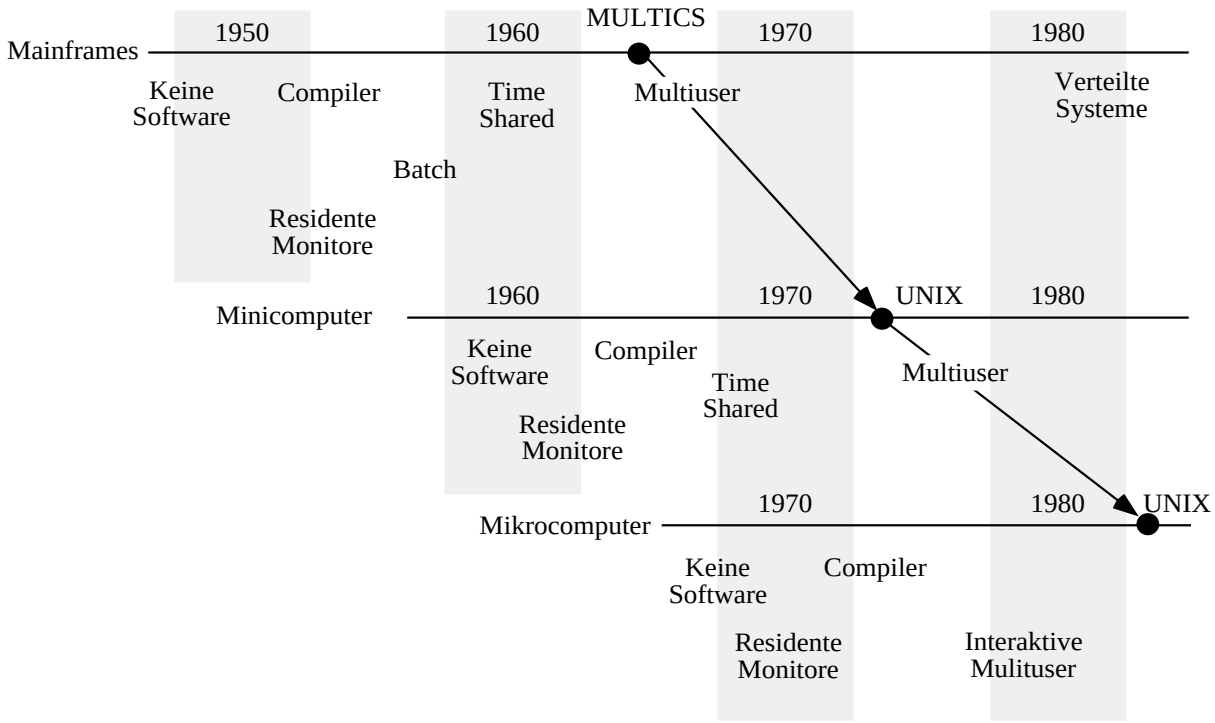


Abb. 1.3: Migration der Betriebssystemkonzepte

1.3 Aufbau eines Rechners

Will man die Struktur heutiger Computersysteme allgemein skizzieren, so erhält man in der Regel etwa folgendes Bild:

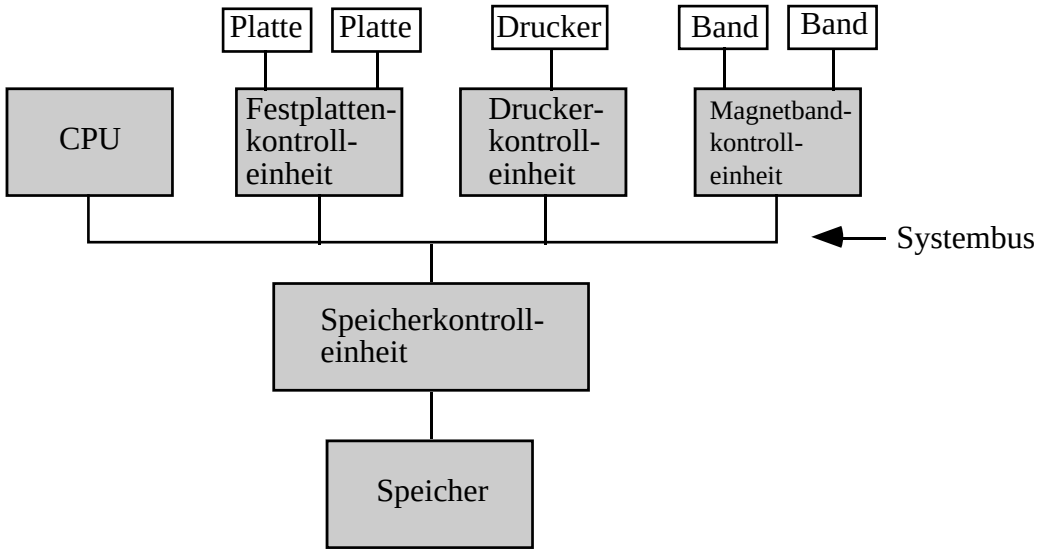


Abb. 1.4: Struktur moderner Computersysteme

Die **CPU** ist über den **Systembus** mit den **Kontrolleinheiten** (device controller) verschiedener Geräte (z. B. Platte, Drucker oder Speicher) verbunden. Jede Kontrolleinheit steuert den Zugriff auf das entsprechende Gerät. Dies ist z. B. dann notwendig, wenn mehrere Stellen gleichzeitig auf ein Gerät zugreifen. Außerdem besitzen die Kontrolleinheiten Puffer, mit denen die Geschwindigkeitsunterschiede zwischen den verschiedenen Teilen des Rechners ausgeglichen werden können.

Nach dem Einschalten eines Computers wird zuerst ein **Boot-Programm** gestartet, um das System (CPU-Register, Controller, Speicher ...) zu initialisieren. Danach wird das Betriebssystem in den Speicher geladen und gestartet. Anschließend wartet das System, bis ein "Ereignis" geschieht. Ereignisse werden der CPU mit Hilfe von **Interrupts** signalisiert. Initiator eines Interrupts kann entweder die Hardware (Drucker: "Ich bin bereit!") oder die Software (Programm: "Schreibe auf Festplatte!") sein. Im Falle eines Interrupts stoppt die CPU ihre aktuelle Arbeit, speichert ihren Zustand und startet eine dem Interrupt entsprechende Routine. Nach der Ausführung dieser Routine wird die vorher unterbrochene Arbeit an der gleichen Stelle wiederaufgenommen.

1.3.1 Grundsätzlicher Ablauf einer E/A-Operation

Das folgende Beispiel soll den Ablauf der Interruptbehandlung am Beispiel einer einfachen E/A-Operation verdeutlichen:

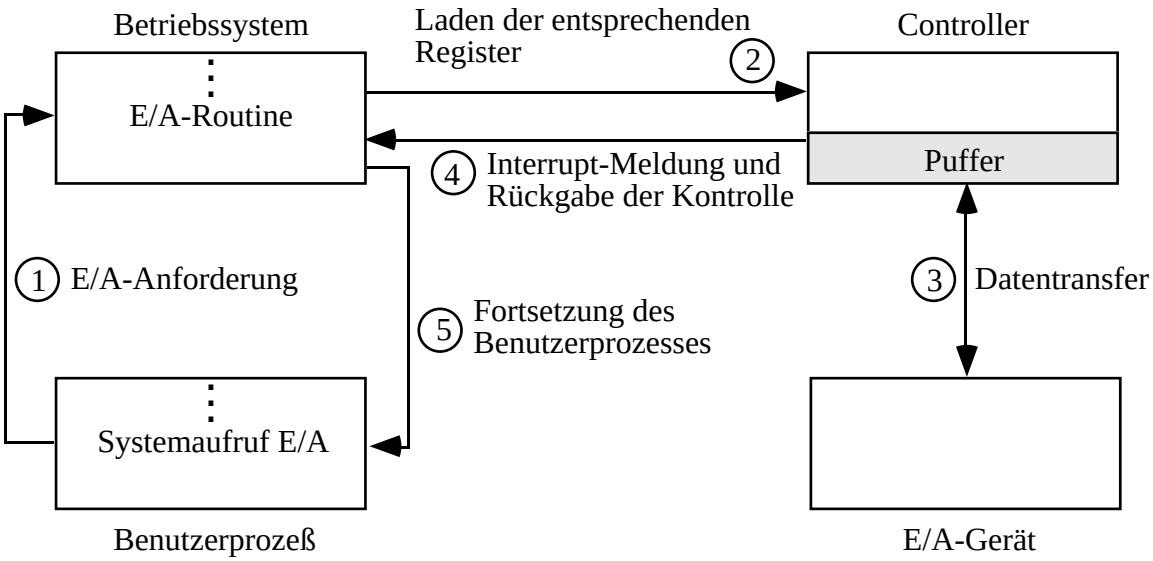


Abb. 1.5: Grundsätzlicher Ablauf einer E/A-Operation

Ein Benutzerprozeß fordert über einen Systemaufruf (Interrupt) eine E/A-Operation an (**1**). Der Systemaufruf unterbricht den Benutzerprozeß und gibt die Kontrolle an das Betriebssystem, welches die entsprechende E/A-Routine aufruft. Diese Routine lädt die notwendigen Werte in die Register und den Puffer des Controllers und stößt die E/A-Operation des Controllers an (**2**). Die CPU kann sich während des Datentransfers (**3**) anderen Aufgaben widmen oder auf dessen Beendigung warten. Durch einen Interrupt (**4**) meldet das E/A-Gerät das Ende der E/A-Operation. Das Betriebssystem ruft eine entsprechende Interrupt-Routine auf und setzt den zuvor unterbrochenen Prozeß fort (**5**).

Synchrone und asynchrone Ein-/Ausgabe
Wird nach der Initiierung des Datentransfers (3) die Kontrolle an das Betriebssystem zurückgegeben, so spricht man von **asynchroner** E/A. Asynchrone E/A führt zu einer Effizienzsteigerung, da die CPU trotz E/A weiterarbeiten kann. Allerdings kommt es auch zu erhöhtem Verwaltungsaufwand, da das Betriebssystem Kenntnis über die Zustände mehrerer E/A-Geräte und Prozesse haben muß.
Synchrone E/A wartet nach dem Anstoßen einer E/A-Operation, bis diese beendet wird. Erst dann wird die Kontrolle an das Betriebssystem zurückgegeben. Der Hauptvorteil sychroner E/A liegt in ihrer Einfachheit.

Direct Memory Access (DMA)
Da sehr oft große Datenmengen z. B. zwischen den Puffern der E/A-Geräte und dem Hauptspeicher bewegt werden müssen, ist die CPU häufig ausschließlich mit Datentransport-Befehlen beschäftigt. Um sie von dieser Arbeit zu entlasten, kann **Direct Memory Access** (DMA) verwendet werden. DMA ermöglicht es einer E/A-Kontrolleinheit, die Übertragung der Daten auf eigene Faust (d. h. nahezu unabhängig von der CPU) durchzuführen. Die CPU initialisiert dazu zunächst die entsprechenden Register des E/A-Controllers. Der eigentliche Datentransport wird vom Controller eigenständig bewerkstelligt. Die so gewonnene Zeit kann von der CPU zur Erledigung anderer Aufgaben verwendet werden.

1.3.2 Speicherstrukturen

Damit ein Computer ein Programm ausführen kann, muß sich das Programm in einem Speicher befinden, auf den die CPU direkt zugreifen kann. Außerdem muß der Speicher so groß sein, daß das entsprechende (Teil-) Programm in ihm Platz hat. Beide Bedingungen werden nur vom **Hauptspeicher** eines Computers erfüllt. Trotz dieser Eigenschaft hat der Hauptspeicher auch Nachteile:

- er ist sehr viel langsamer als die CPU,
- er ist flüchtig (d. h. die Daten sind nur vorhanden, solange der Rechner eingeschaltet ist), und
- er ist nicht groß genug, um zusätzlich Daten zu speichern, die momentan nicht benötigt werden.

Diese Probleme werden durch einen hierarchischen Aufbau des Speichers gelöst. An der Spitze dieser Hierarchie stehen sehr schnelle Speicher, die häufig benötigte Daten speichern (**Register, Cache**). Da diese Art des Speichers pro Speicherwort relativ teuer ist, haben Register und Cache nur eine geringe Kapazität.

Um Daten dauerhaft zu speichern, werden permanente Speichermedien (Festplatte, optische Platte, Magnetbänder) als **Sekundär-** oder **Hintergrundspeicher** benutzt. Die Zugriffszeit auf den Sekundärspeicher ist zwar wesentlich größer als die Zugriffszeit auf den Hauptspeicher, dafür kann dort jedoch ein Vielfaches an Daten gespeichert werden. Bild 1.6 zeigt eine übliche Speicherhierarchie heutiger Rechner.

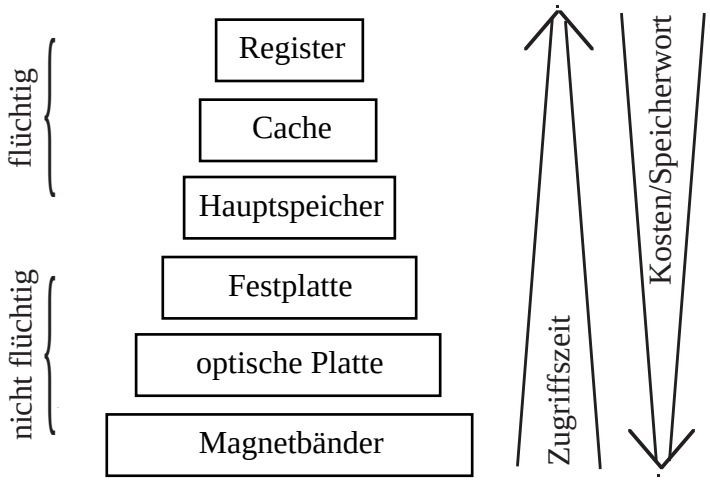


Abb. 1.6: Typische Speicherhierarchie

1.4 Aufbau eines Betriebssystems

1.4.1 Komponenten eines Betriebssystems

Wie uns der kurze Blick in die Geschichte zeigte, sind Betriebssysteme heutzutage hochkomplexe Programme, die eine Unmenge von Aufgaben erfüllen müssen. Um die Realisierung zu vereinfachen, kann ein Betriebssystem in einzelne Komponenten unterteilt werden, die man (relativ) unabhängig voneinander entwickelt.

Prozeßverwaltung

Ein Prozeß ist (vereinfacht gesagt) ein "Programm in Ausführung". Sind auf einem Rechner mehrere Prozesse "gleichzeitig" aktiv, so müssen diese vom Betriebssystem verwaltet werden. Es muß

- neue Prozesse erzeugen und alte vernichten,
- die zur Verfügung stehende CPU-Zeit auf die Prozesse verteilen (Scheduling),
- Prozesse, die z. B. gemeinsame Variablen benutzen, synchronisieren, um Inkonsistenzen zu vermeiden,
- Verklemmungen (Deadlocks, d. h. keiner kann mehr weiterarbeiten) beheben, und
- Mittel zur Interprozeß-Kommunikation zur Verfügung stellen.

Hauptspeicherverwaltung
Ein Prozeß benötigt für seine Arbeit Speicherplatz. Vor seiner Ausführung müssen daher alle relevanten Daten in den Hauptspeicher geladen werden. Aufgabe des Betriebssystems ist es,

- die belegten und freien Speicherbereiche zu verwalten,
- zu entscheiden, welcher neue Prozeß bei freiwerdendem Speicherplatz geladen wird und
- wieviel Speicherplatz einem Prozeß zugewiesen wird (Zuteilung/Entzug).

Sekundärspeicherverwaltung
Eigentlich ist diese der Hauptspeicherwaltung recht ähnlich. Ein Unterschied zwischen Haupt- und Sekundärspeicher liegt jedoch in der Zugriffszeit. Diese dauert beim Sekundärspeicher nicht nur länger, sondern kann insbesondere stark variieren (Spulen an eine geeignete Stelle beim Magnetband, Bewegung des Lese/Schreibkopfes bei einer Platte). Dadurch ergeben sich für die Sekundärspeicherverwaltung zusätzlich Aufgaben wie:

- Disk Scheduling (mehrere Anfragen können beispielsweise gesammelt werden und dann in einer günstigen Reihenfolge abgearbeitet werden)
- Speicherplatzzuteilung, da die Lage der Daten direkten Einfluß auf die Zugriffszeit hat, sowie
- Verwaltung der freien Speicherbereiche.

Dateiverwaltung
Die Dateiverwaltung abstrahiert von der physikalischen Speicherung der Daten. Dazu werden diese logisch als Dateien und Verzeichnisse angesehen, die dem Anwender die Arbeit mit den Daten vereinfachen. Das Betriebssystem ermöglicht dabei z. B.

- das Erstellen, Manipulieren und Löschen von Dateien bzw. Verzeichnissen,
- eine geeignete Abbildung der logischen auf die physikalische Struktur des Speichers, und
- ein automatisches Sichern wichtiger Daten als Backup.

Kommandointerpreter
Ein Kommandointerpreter bildet eine Schnittstelle zwischen Betriebssystem und Benutzer. Sie bietet dem Benutzer eine Möglichkeit, die Funktionalität des Betriebssystems zu nutzen, z. B. Speicherung einer Datei, Start eines Programms usw. Kommandozeilen-Interpreter lesen dazu die Texteingabe eines Benutzers und starten dann eine entsprechende Aktion. Andere Interpreter stellen dafür Icons oder Menüs zur Verfügung. Der Kommandointerpreter ist entweder als Teil des Betriebssystems oder als separates Systemprogramm realisiert.

1.4.2 Systemaufrufe

Über Systemaufrufe können Prozesse direkt mit dem Betriebssystem kommunizieren. Der Aufruf erfolgt dabei über eine Bibliotheksprozedur. Diese hat einen Namen und je nach Typ eine Reihe von Parametern. Realisiert sind Bibliotheksprozeduren meist in Assembler. Es gibt jedoch auch Systeme, die dafür höhere Programmiersprachen (wie z. B. C) benutzen.

Beispiel:
Benötigt ein Prozeß während der Ausführung eines Programms Daten aus einer Datei, so muß er die Datei öffnen und die entsprechenden Daten lesen. Eine solche Leseoperation könnte in einem C-Programm wie folgt aussehen:

```
count = read(file, buffer, n_bytes);
```

Der Systemaufruf hat den Namen `read`, übergibt die drei Parameter `file, buffer` und `n_bytes` und gibt als Resultat des Aufrufs einen Wert zurück, der in einer Variable `count` gespeichert wird.
Bedeutung des Aufrufs: Lies eine Anzahl von Bytes (`n_bytes`) aus der Datei (`file`) in einen Puffer (`buffer`). Nach dem Aufruf gibt der Wert `count` an, wieviele Bytes tatsächlich gelesen wurden. Normalerweise gilt `count = n_bytes`, wegen EndOfFile oder bei fehlerhaftem Aufruf sind jedoch auch andere Werte denkbar.

Systemaufrufe können grob in fünf Kategorien eingeteilt werden: Prozeßkontrolle, Dateimanipulation, Gerätemanipulation, Kommunikation und Abruf von Informationen. Typische Beispiele für Aufrufe der jeweiligen Bereiche sind:

Prozeßkontrolle:
- create/terminate process
- wait/signal event
- allocate/free memory

Dateimanipulation:
- create/delete file
- open/close file
- read/write file

Gerätemanipulation:
- request/release device
- read/write/reposition device

Kommunikation:
- create/delete communication connection
- send/receive message

Abruf von Informationen:
- get process, get file, get data attributes
- get/set time, get/set date

1.4.3 Systemprogramme

Systemprogramme sollen dem Benutzer die Arbeit mit dem Computer erleichtern. Sie sollen ihm helfen, Programme zu erstellen, sie auszuführen usw. Um diese Unterstützung zu erreichen, stellen Rechner eine Reihe von Programmen zur Verfügung, die direkt auf das Betriebssystem aufsetzen und dessen Komplexität vor dem

Benutzer verbergen. Sie bieten eine im Vergleich zu den Systemaufrufen komfortable Schnittstelle.
Zu den Programmen, die häufig als Systemprogramme realisiert sind, gehören Kommandointerpreter (z. B. Shell bei UNIX, command.com bei MS-DOS), Compiler, einfache Texteditoren usw.
Systemprogramme sind nicht Teil des Betriebssystems. Beim Kauf eines Betriebssystems werden jedoch häufig die wichtigsten Systemprogramme mitgeliefert.

1.5 Beispielarchitekturen

Nachdem in einem ersten Anlauf die wesentlichen Konzepte von Betriebssystemen angerissen wurden, wollen wir nun beispielhaft einige real existierende Betriebssysteme vorstellen.

1.5.1 MS-DOS

MS-DOS ist das zur Zeit am weitesten verbreitete Betriebssystem (Intel 8088, 286, 386, 486, Pentium). Bei seiner Entwicklung war sich niemand darüber im klaren, wie populär dieses System einmal werden würde. Deswegen weist die Struktur von MS-DOS auch einige Mängel auf.

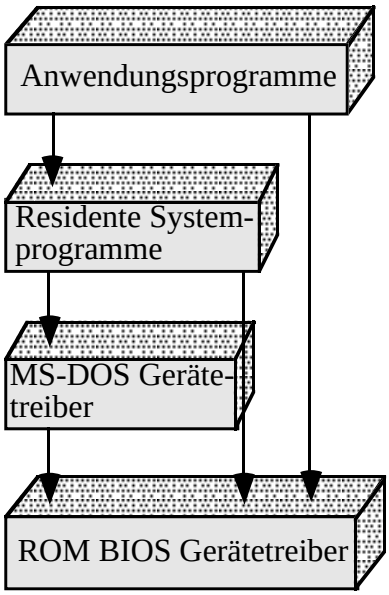


Abb. 1.7: Prinzipielle Struktur von MS-DOS

So wird bei MS-DOS z. B. nicht zwischen einem **User-** und einem **Supervisor-Mode** unterschieden. Dies bedeutet, daß ein Benutzerprozeß gleiche Rechte und Zugriffsmöglichkeiten hat wie beispielsweise das Betriebssystem selbst. Dadurch kann es zu Fehlern kommen, wenn ein Programm mit seinen Daten Teile des Betriebssystems überschreibt. Ein weiterer "Nachteil" von MS-DOS besteht darin, daß es Multitasking nicht unterstützt. Startet ein Elternprozeß einen neuen Kindprozeß, so wird der Elternprozeß erst dann fortgesetzt, wenn der Kindprozeß endet.

1.5.2 UNIX

UNIX ist das derzeit dominierende Betriebssystem im Bereich der Workstations. Im Gegensatz zu MS-DOS gibt es UNIX-Versionen für die unterschiedlichsten Rechner-

typen. Die Schnittstellen zwischen den einzelnen Komponenten von UNIX sind besser definiert als bei MS-DOS.

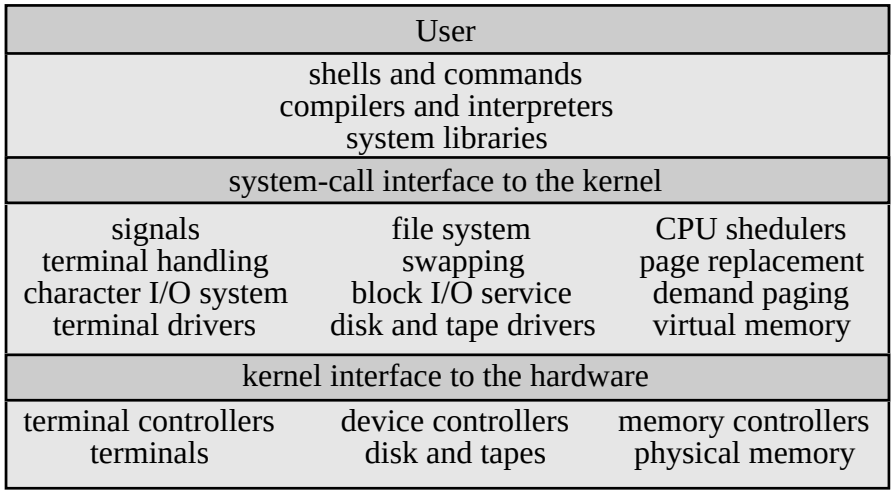


Abb. 1.8: UNIX-Systemstruktur

Der eigentliche Betriebssystemkern kann nur im Supervisor-Mode verändert werden (Modebit in der Hardware). Dadurch wird das System vor allzu eifrigen Benutzern oder Prozessen geschützt. Im Gegensatz zu MS-DOS können bei UNIX außerdem mehrere Prozesse "quasi gleichzeitig" ablaufen (Multitasking).

1.5.3 OS/2

OS/2 ist eine Weiterentwicklung von MS-DOS und soll insbesondere die Schwächen von MS/DOS beheben. Z. B. wird das seit dem Intel 80486 vorhandene Hardware-Bit für den User-/Supervisor-Mode benutzt. Außerdem erweitert OS/2 das MS-DOS-Konzept um Multitasking. Die folgende Abbildung 1.9 spiegelt die Struktur von OS/2 wider:

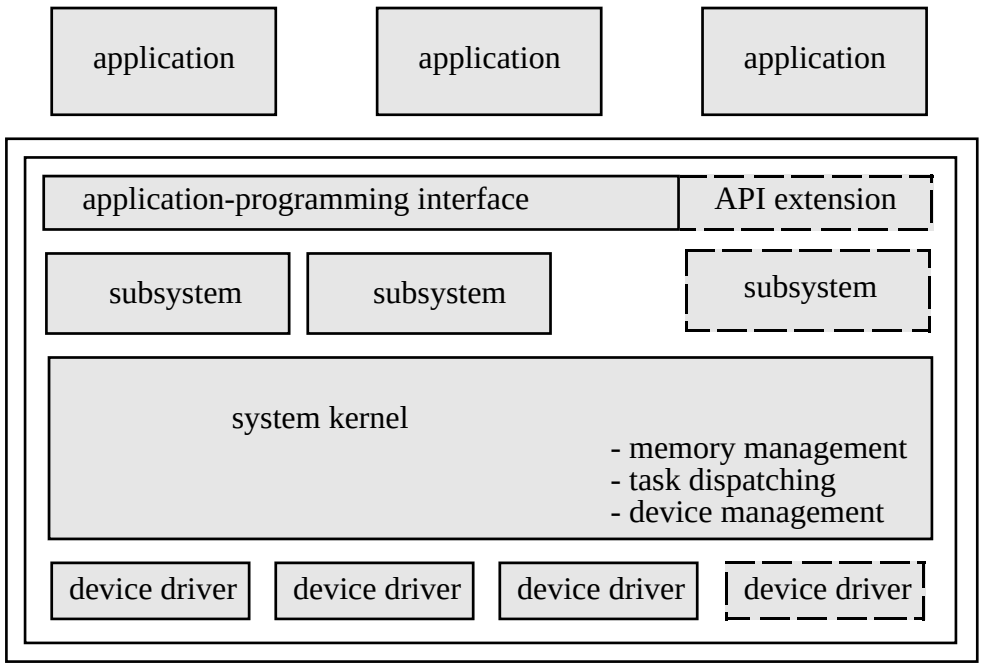


Abb. 1.9: Die OS/2-Ebenenstruktur

1.5.4 Schichtenkonzepte

Neben den kommerziell verfügbaren Produkten gibt es einige Entwicklungen, die vorwiegend eine geeignete **Strukturierung** des Betriebssystems zum Ziel haben.
Ein häufig verwendeter Ansatz ist die Unterteilung eines komplexen Systems in **Schichten**. Die unteren Schichten sind dabei eher "hardwarenah", die oberen "anwendungsnah".
Eine Schicht nutzt jeweils die Funktionalität, die ihr von einer darunterliegenden Schicht geboten wird, und bietet ihrerseits der darüberliegenden Schicht Dienste an.

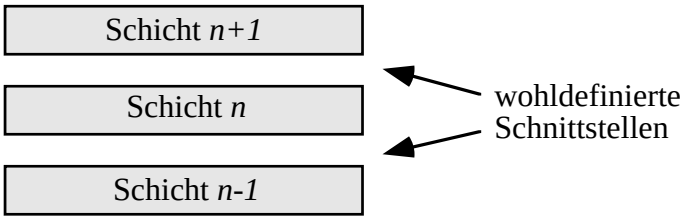


Abb. 1.10: Prinzipielle Struktur einer Schichtenarchitektur

Man erkennt, daß bei einer festen Schnittstellendefinition die alte Realisierung einer Schicht problemlos gegen eine neue Realisierung ausgetauscht werden kann. Der Nachteil des Schichtenprinzips liegt in einer insgesamt aufwendigen Realisierung.

Zur Veranschaulichung zeigen wir in Abb. 1.11 und Abb. 1.12 die Schichtenstrukturen zweier Betriebssysteme, die nicht allzu weit verbreitet sind: THE (Technische Hogeschool Eindhoven) und Venus.

Schicht 5: Benutzerprogramme
Schicht 4: Ein-/Ausgabepufferung
Schicht 3: Gerätetreiber
Schicht 2: Speicherverwaltung
Schicht 1: CPU Scheduling
Schicht 0: Hardware

Abb. 1.11: Die THE Schichtenstruktur

Schicht 6: Benutzerprogramme
Schicht 5: Gerätetreiber
Schicht 4: Virtueller Speicher
Schicht 3: Ein-/Ausgabekanäle
Schicht 2: CPU-Scheduling
Schicht 1: Befehlsinterpreter
Schicht 0: Hardware

Abb. 1.12: Die Venus Schichtenstruktur

Schichtenkonzepte in der Datenkommunikation
Die Kommunikation zwischen Rechnern erfolgt über sogenannte Protokolle. Diese legen fest, wie Daten, die über Netz von einem anderen Rechner ankommen, zu interpretieren sind. Da die hierfür benötigte Software sehr komplex ist, wird auch hier ein Schichtenkonzept verwendet:

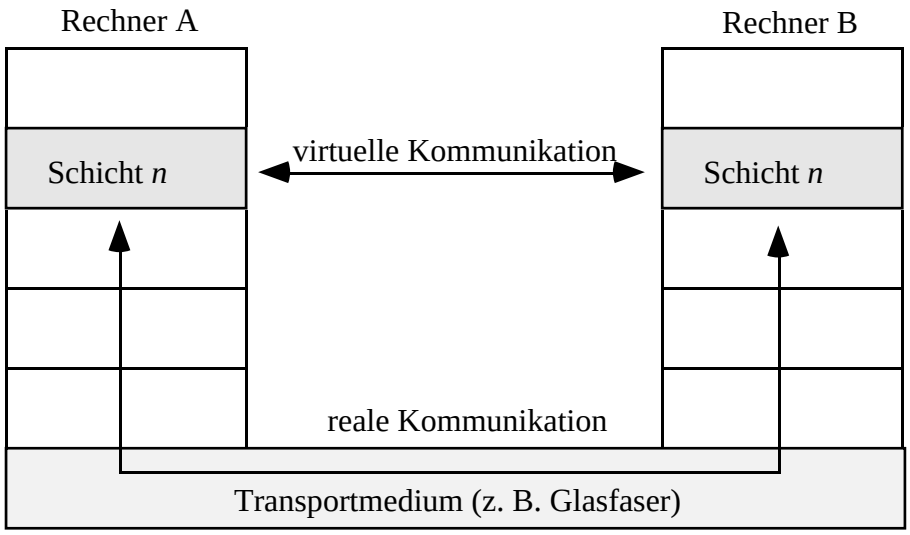


Abb. 1.13: Nutzung einer Schichtenarchitektur bei der Rechnerkommunikation

Eine Anwendung eines Rechners A, die mit einer Anwendung des Rechners B kommunizieren möchte, gibt die Daten an ihre oberste Kommunikationsschicht weiter. Diese Schicht (die durch ein Protokoll mit der entsprechenden Schicht des Rechners B kommuniziert) leitet die Daten an die darunterliegenden Schichten weiter. Die Daten werden schließlich auf das Netz gegeben und durchlaufen beim Rechner B die Schichten von unten.
Es erfolgt also eine virtuelle Kommunikation zwischen gleichen Schichten, die reale Kommunikation findet jedoch mit der darüber- bzw. darunterliegenden Schicht statt.

Das **ISO/OSI-** (**I**nternational **S**tandard **O**rganization / **O**pen **S**ystems **I**nterconnection) **Referenzmodell**

Das Modell basiert auf einem Vorschlag, der von der ISO für die Standardisierung von Kommunikationsprotokollen entwickelt wurde. Es besteht aus 7 Schichten und dem Kommunikationsmedium. Die Bezeichnung der einzelnen Schichten sowie ein Versuch, die verschiedenen Funktionalitäten anhand einer "Eselsbrücke" zu veranschaulichen, findet sich in Abb. 1.14.

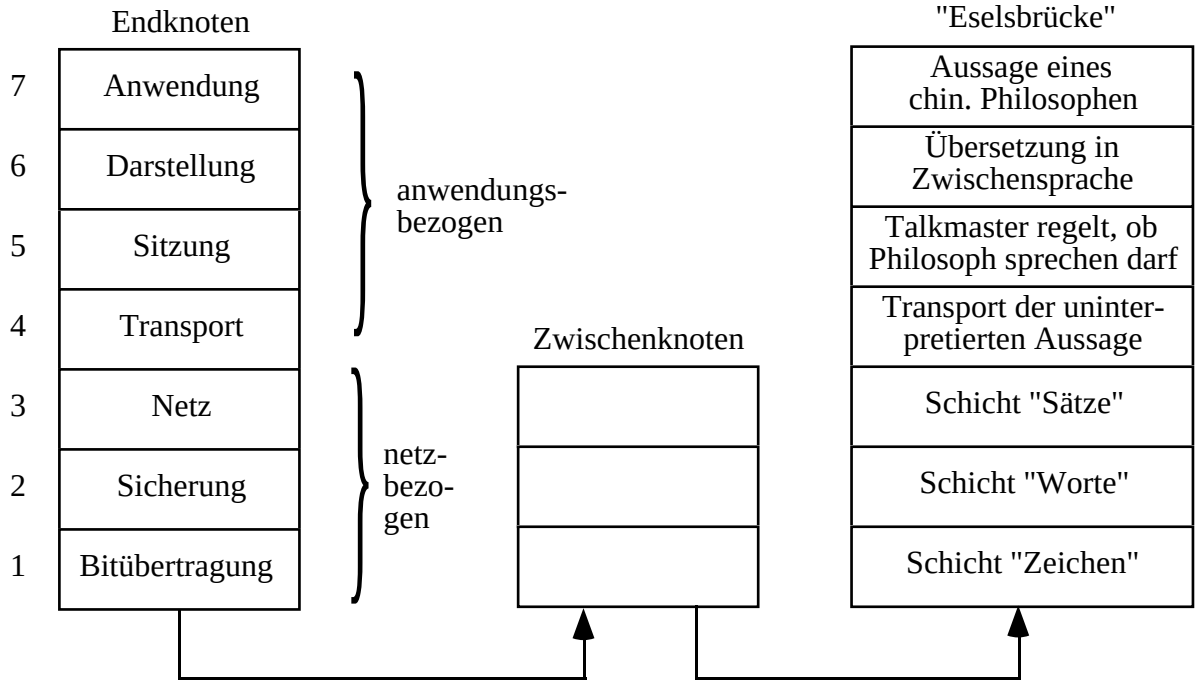


Abb. 1.14: Kommunikationsstandardisierung gemäß ISO/OSI

Prozeßverwaltung

2 Prozesse

2.1 Zum Prozeßbegriff

Der Begriff des Prozesses stellt eines der grundlegendsten Konzepte im Bereich der Systemprogrammierung dar. In der Einleitung wurde ein **Prozeß** kurz als "**Programm im Stadium der Ausführung**" definiert, um dadurch den "dynamischen" Charakter eines Prozesses im Gegensatz zur "Statik" eines Programmcodes zum Ausdruck zu bringen.

Der Prozeßbegriff läßt sich sowohl auf die Benutzer- wie auf die Systemebene anwenden. In den meisten modernen Rechensystemen gibt es eine größere Anzahl von nebeneinander existierenden Benutzer- und Systemprozessen. Jeder Prozeß benötigt nun für seine Ausführung bestimmte Ressourcen (wie etwa Speicherplatz, CPU-Zeit und Zugriff auf E/A-Geräte). Wenn nun Prozesse simultan auf das gleiche Betriebsmittel zugreifen wollen, ergibt sich von selbst die effiziente Koordination derartiger Aktivitäten als eine Grundaufgabe der Systemprogrammierung.

Prozesse befinden sich in unterschiedlichen **Zuständen**, die sich durch die jeweilige Art der Aktivität charakterisieren lassen. Solche Zustände sind beispielsweise

Zustand	Bedeutung
running	Anweisungen des Prozesses werden gerade ausgeführt
ready	Prozeß ist bereit zur Ausführung, wartet aber noch auf freien Prozessor
waiting	Prozeß wartet auf das Eintreten eines Ereignisses (z. B. darauf, daß ein von ihm belegtes Betriebsmittel fertig wird)
blocked	Prozeß wartet auf ein fremdbelegtes Betriebsmittel
new	Prozeß wird erzeugt (kreiert)
killed	Prozeß wird (vorzeitig) abgebrochen
terminated	Prozeß ist beendet (alle Instruktionen sind abgearbeitet)

Wichtig ist in diesem Zusammenhang, daß zu jedem Zeitpunkt auf einem beliebigen Prozessor immer nur ein Prozeß laufen kann, wohingegen sich viele Prozesse gleichzeitig im ready- oder waiting-Status befinden können.

Die Übergänge zwischen den einzelnen Zuständen kann man sich leicht anhand eines Diagramms verdeutlichen. Ein Beispiel für ein solches **Prozeßzustandsdiagramm** zeigt Abbildung 2.1 (ohne Zustände blocked und killed):

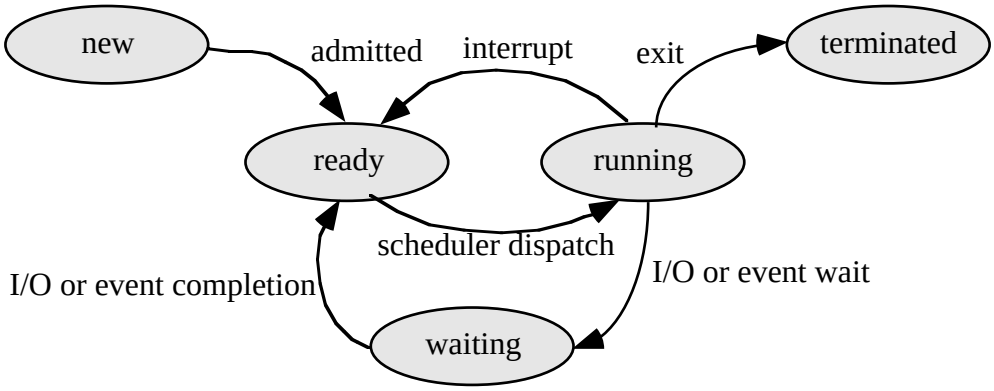


Abb. 2.1: Prozeßzustände

Im Zusammenhang mit Prozessen ist das Betriebssystem vor allem zuständig für die Erzeugung und Beendigung von Benutzer- und Systemprozessen, die Aufteilung von Speicherplatz und CPU-Zeit, die Bereitstellung von Mechanismen zur Synchronisation und Kommunikation unter Prozessen sowie zum Vorgehen in Deadlock-Situationen.

Dazu müssen Prozesse eine Repräsentation innerhalb des Betriebssystems besitzen. Diese erfolgt über sogenannte **Prozeßkontrollblöcke** (PCB), die alle für das Betriebssystem relevanten Informationen über einen Prozeß beinhalten. In einem PCB finden sich beispielsweise folgende Felder:

PCB-Feld	Bedeutung
Status:	beinhaltet einen der oben erläuterten Zustände
Programmzähler:	enthält die Adresse der nächsten auszuführenden Instruktion
CPU-Scheduling:	hier finden sich z. B. Prioritäten, Zeiger auf Warteschlangen und ähnliche Scheduling-Parameter
Speichermanagement:	enthält etwa die letzten Werte der Speicherregister, Seitentabellen und ähnliche Informationen für die Speicherverwaltung
E/A-Status:	Liste von zugeordneten E/A-Geräten, offenen Files usw.
Accounting:	benötigte CPU- und Realzeit, Zeitbeschränkungen, Prozeßnummern u. ä.

2.2 Scheduling

Ganz grob lassen sich Betriebssysteme aufgrund ihrer Arbeitsweise in Singletasking-, Multiprogramming- und Multitasking-Systeme einteilen. Beim **Singletasking** enthält der Hauptspeicher den Betriebssystemkern und darauf aufsetzend den Kommandointerpreter (Command Interpreter CI). Ein Prozeß wird bei seinem Aufruf exklusiv in den Speicher eingelesen. Bei der Implementierung ist besonders im Auge zu behalten, daß der Command Interpreter beim Einlesen des Prozesses u. U. überschrieben werden kann.

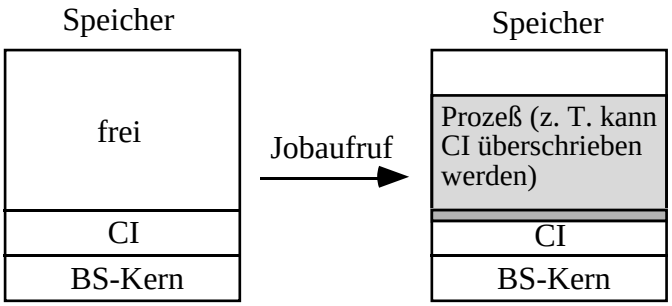


Abb. 2.2: Prinzip des Singletasking (Achtung: CI teilweise überschrieben!!)

Das **Multiprogramming** und **Multitasking** unterscheidet sich davon insofern, als jetzt neben dem Betriebssystemkern und dem Command Interpreter mehrere Prozesse gleichzeitig im Speicher koexistieren können:

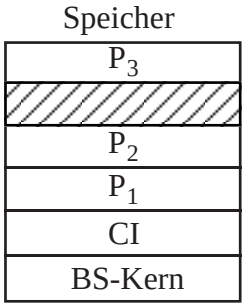


Abb. 2.3: Prinzip des Multitasking

Das Betriebssystem verwaltet also mehrere Prozesse gleichzeitig und regelt insbesondere den Zugriff auf die CPU. Dieses sogenannte **Scheduling** ist entscheidend für einen sorgsamen Umgang mit der verfügbaren CPU-Zeit. Scheduling findet auf verschiedenen zeitlichen Ebenen statt: Der Langzeit- (oder Job-)Scheduler ist dabei zuständig für die Auswahl der Prozesse, die in den Speicher geladen werden, während der Kurzzeit- (oder CPU-)Scheduler entscheidet, welcher der (bereits geladenen) Prozesse im ready-Status wann auf die CPU zugreifen darf. Gute Scheduling-Strategien legen Wert auf eine ausgewogene Mischung ("Job-Mix") zwischen rechenintensiven und E/A-intensiven Prozessen, um die Geschwindigkeitsdiskrepanz zwischen den "langsamen" E/A-Geräten und der "schnellen" CPU auszugleichen.

Der Unterschied zwischen Multiprogramming und Multitasking besteht darin, daß beim Multiprogramming der Benutzer während seiner CPU-Nutzung nicht unterbrochen werden kann (was natürlich bei rechenintensiven Prozessen zu einer Blockade der CPU führen kann), während beim Multitasking (auch "**Time-Sharing**" genannt) ein Prozeß nicht nur bei einem Interrupt unterbrochen werden kann, sondern z. B. auch nach Ablauf einer bestimmten Zeitspanne. Dies bietet mehreren Benutzern gleichzeitig die Möglichkeit zu interaktivem Betrieb, wobei jeder einzelne die Illusion haben kann, die CPU arbeite nur für ihn. Ein Beispiel hierfür ist das Round-Robin-Verfahren, auf das wir in Abschnitt 5.2.5 genauer eingehen werden.

2.3 Interprozeßkommunikation

Sobald man die Koexistenz mehrerer Prozesse erlaubt, muß man auch Möglichkeiten für eine Kommunikation zwischen diesen Prozessen vorsehen. Dabei gibt es zwei unterschiedliche Ansätze. Beim **message-passing** wird zunächst über das Betriebssystem eine Verbindung zwischen Prozeß A und Prozeß B aufgebaut. Typische Systemaufrufe in diesem Zusammenhang sind etwa `gethostid`, `open_connection`, `accept_connection` oder `close_connection`. Die so eingerichtete Verbindung kann zu zwei oder mehreren Prozessen gehören, unterschiedliche Kapazität aufweisen, unterschiedliche Nachrichtenlängen zulassen, uni- oder bidirektional sein und auf verschiedenartigste Weise implementiert werden. Mit Hilfe von `send`- und `receive`-Befehlen werden dann auf dieser Verbindung Nachrichten ausgetauscht.

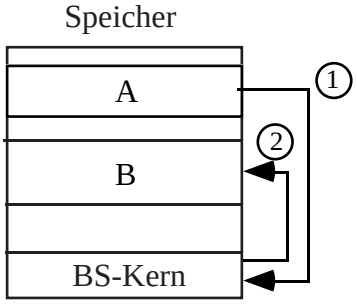


Abb. 2.4: Prinzip des message-passing

Der andere Ansatz (**shared-memory**) beruht darauf, die exklusive Nutzung von Speicherbereichen durch einzelne Prozesse zeitweise aufzuheben. Prozeß A speichert dabei eine Nachricht an Prozeß B ohne Umweg über den Betriebssystemkern in einem Speicherbereich, auf den auch B Zugriff hat.

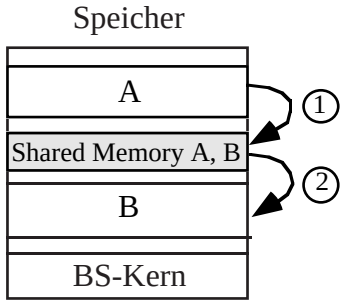


Abb. 2.5: Prinzip des shared-memory

Dieses Vorgehen hat den Vorteil, vor allem bei großen Datenmengen schneller zu sein als das message-passing. Allerdings können sehr leicht Synchronisations- bzw. Konsistenzprobleme auftauchen, wenn etwa B bereits liest, während A noch schreibt. Mechanismen zur Umgehung solcher Konflikte werden wir im Abschnitt über das "wechselseitige Ausschlußproblem" ausführlich kennenlernen.

3 Koordination und Synchronisation nebenläufiger Prozesse

In aller Regel arbeiten Prozesse heute nicht mehr isoliert, sondern kooperieren mit anderen Prozessen. Ihre Ausführung erfolgt also quasi-gleichzeitig, z. B. ineinander verzahnt ("interleaved"). Bei der Kooperation mehrerer Prozesse kann es leicht dazu kommen, daß Prozeß B auf Prozeß A angewiesen ist. Beispielsweise erzeugt A irgendwelche Resultate, deren anschließende Ausgabe auf einem Endgerät durch B veranlaßt wird. Abstrakter ausgedrückt heißt dies: Prozeß A "erzeugt" etwas, das Prozeß B "verbraucht". Das "Erzeuger-Verbraucher-Problem" wird uns als erstes "klassisches" Beispiel für Fragen der Koordination und Synchronisation nebenläufiger Prozesse dienen.

3.1 Erzeuger-Verbraucher-Problem

Gegeben sei ein Prozeß E, der "Erzeuger", und ein Prozeß V, der "Verbraucher". Ferner sei zwischen E und V ein Lager mit endlicher Kapazität `MAX` (etwa in Form eines Pufferspeichers) eingerichtet. Der Erzeuger kann Produkte im Zwischenlager ablegen, der Verbraucher kann Produkte daraus entnehmen. Dabei setzen wir voraus, daß die produzierten bzw. konsumierten Einheiten jeweils gerade in einen Pufferplatz passen.

Für das Zusammenspiel von Erzeuger und Verbraucher gelten folgende Randbedingungen:

1. Wenn das Lager voll ist, kann E nichts ablegen.
2. Wenn das Lager leer ist, kann V nichts entnehmen.
3. Der Lagerbestand kann nicht gleichzeitig von E und V verändert werden.

Bedingung 3, die "allgemeine Konsistenzbedingung" des Problems, kann bei einfachem Drauflosprogrammieren sehr leicht unter den Tisch fallen. Dies passiert beispielsweise, wenn wir das Erzeuger-Verbraucher-Problem durch Verwendung zweier Routinen ERZEUGER und VERBRAUCHER angehen, die beide auf eine gemeinsame Variable S, den aktuellen Lagerbestand, zugreifen und folgende Gestalt haben:

ERZEUGER:

```
repeat
  erzeuge Element;
  while S = MAX do noop;
  lege Element in Puffer ab;
  S := S + 1;
until FALSE;
```

VERBRAUCHER:

```
repeat
  while S = 0 do noop;
  entnimm Element aus Puffer;
  S := S - 1;
  verbrauche Element;
until FALSE;
```

Beide Routinen laufen unabhängig voneinander. Daher kann es bei unkontrolliertem Zugriff auf den Lagerbestand `S` zu Inkonsistenzen kommen, nämlich dann, wenn E nach Überprüfen der Bedingung "`S < MAX`" ein Produkt ablegt, aber nicht mehr dazu kommt, den Lagerbestand zu korrigieren, weil in diesem Moment V auf den Lagerbestand `S` zugreift, eine Einheit entnimmt, das "alte" `S` um 1 verringert und erst dann wieder E das Feld überläßt. E führt dann seine angefangene Routine zu Ende und erhöht seinerseits das "alte" (und inzwischen leider veraltete) `S` um 1 (woher sollte E auch wissen, daß V in der Zwischenzeit zugeschlagen hat ...?):

Zeit ↓

	Erzeugerprozeß	Wert von S	Verbraucherprozeß
ZE 1	read(S);	17	
ZE 2		17	read(S);
ZE 3	if S < MAX		
ZE 4	then		
ZE 5	lege ab;		
ZE 6			if S > 0
ZE 7			then
ZE 8			entnimm;
ZE 9		16	S := S -1
ZE10	S := S + 1;	18	

Obwohl sich also am Lagerbestand de facto nichts geändert hat, ist das "neue" `S` größer als das alte. Die Korrektur des Lagerbestandes durch V wurde nicht berücksichtigt, was zu einem "lost update" führt.

3.1.1 Einfache Lösung mittels Ringpuffer

Eine Lösung des Erzeuger-Verbraucher-Problems, die diese Schwierigkeiten umgeht, verwendet einen Ringpuffer. Dieser wird mittels einer modulo-Funktion **`mod`** realisiert und kann im richtigen Leben z. B. in Form eines ringförmigen Diamagazins angetroffen werden. Zunächst sei die Lösung auf einen Erzeuger- und einen Verbraucherprozeß beschränkt; später werden wir auch den Fall mehrerer Erzeuger und Verbraucher einbeziehen.

Die Idee ist einfach: Der Erzeuger legt sein Produkt im "ersten freien" Pufferplatz ab, der Verbraucher entnimmt ein Produkt aus dem "ältesten belegten". Beide vereinbaren, daß der Puffer höchstens `MAX-1` belegte Komponenten haben soll.

Dann benötigen wir zwei Variable:

`in` zeigt auf den ersten freien Puffererplatz (`mod MAX`)
`out` zeigt auf den ersten belegten Pufferplatz (`mod MAX`)

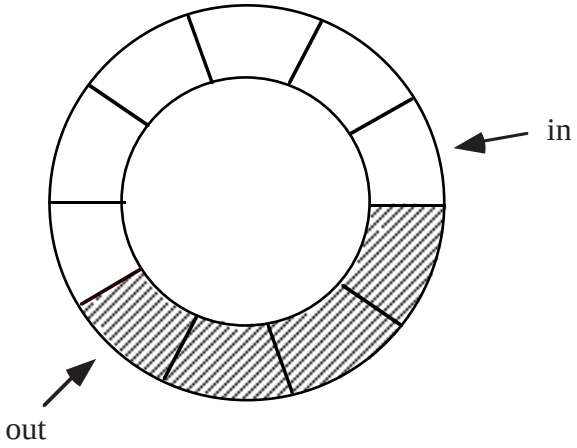


Abb. 3.1: Lösung des Erzeuger-Verbraucher-Problems mittels Ringpuffer

Der Puffer ist mithin genau dann leer, wenn `in = out` gilt, und genau dann voll, wenn `in+1 mod MAX = out` ist (man beachte, daß höchstens `MAX-1` Pufferplätze belegt werden dürfen).

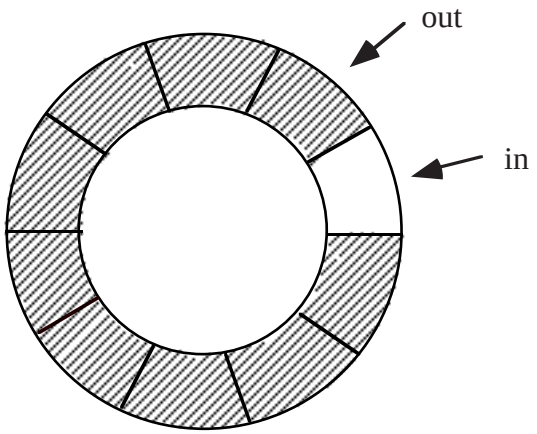


Abb. 3.2: Vollständig belegter Ringpuffer

ERZEUGER und VERBRAUCHER lassen sich folgendermaßen implementieren:

ERZEUGER:

```
repeat
  produziere ein Element und lege es in nextp ab;
                                        (* Zwischenlager *)
  while in+1 mod MAX = out do noop;     (* Puffer voll *)
  buffer[in] := nextp;                  (* ablegen *)
  in := in+1 mod MAX;                   (* Zeigerkorrektur *)
until FALSE;
```

VERBRAUCHER:

```
repeat
  while in = out do noop;                (*Speicher leer*)
  nextc := buffer[out];                  (*entnehmen*)
  out := out + 1 mod MAX;                (*Zeigerkorrektur*)
  verbrauche das in nextc enthaltene Element;
until FALSE;
```

3.1.2 Allgemeiner Lösungsansatz und neue Schwierigkeiten

Kann man diese Lösung in einfacher Weise auf den Fall mehrerer Erzeuger- und mehrerer Verbraucherprozesse verallgemeinern, indem man den Ringpuffer als allen gemeinsames Lager der Kapazität `MAX` ansieht und den Zugriff darauf über eine globale Zählvariable `counter` realisiert? Setzt man den Zähler zu Beginn auf 0 (bzw. einen beliebigen Anfangsfüllstand zwischen 0 und `MAX`), so lauten die vorgeschlagenen Algorithmen:

ERZEUGER:

```
repeat
  produziere ein Element und lege es in nextp ab;
  while counter = MAX do noop;
  buffer[in] := nextp;
  in := in + 1 mod MAX;
  counter := counter + 1;
until FALSE;
```

VERBRAUCHER:

```
repeat
  while counter = 0 do noop;
  nextc := buffer[out];
  out := out + 1 mod MAX;
  counter := counter - 1;
  verbrauche das in nextc enthaltene Element;
until FALSE;
```

Diese Lösung ist prinzipiell in Ordnung, krankt allerdings (wie auch die Lösung des einfachen Falles) daran, daß Probleme mit dem Füllgrad auftreten können. Der Grund dafür liegt im Befehl "`counter := counter ± 1`". Eine typische maschinensprachliche Implementation dieses Befehls läuft nach folgendem Schema ab:

```
register1 := counter;
register1 := register1 ± 1;
counter := register1;
```

wobei `register1` ein lokales CPU-Register ist. Der eine programmiersprachliche Befehl besteht also aus mehreren maschinensprachlichen Anweisungen. Inkrementiert nun ein Prozeß den Zähler, während ihn der andere gleichzeitig dekrementiert, so stellt sich dies auf der Maschinenebene als sequentielle Verzahnung der entsprechenden Anweisungen dar, die unter widrigen Umständen z. B. folgendes Aussehen haben (wobei wir zu Beginn `counter = 5` voraussetzen):

Taktzyklus	Prozeß	Anweisung	Resultat
1	E	register1 := counter	register1 = 5

2	E	register1 := register1+1	register1 = 6
3	V	register2 := counter	register2 = 5
4	V	register2 := register2 - 1	register2 = 4
5	E	counter := register1	counter = 6
6	V	counter := register2	counter = 4

Es wurde also je eine Einheit erzeugt und verbraucht, dennoch ändert sich der Füllstand.
Der Grund für solche Konsistenzprobleme ist stets darin zu suchen, daß gemeinsame Datenbestände von mehreren Prozessen gleichzeitig manipuliert werden. Ziel der folgenden Überlegungen wird sein, solche unkontrollierten simultanen Manipulationen zu verhindern.

3.2 Das Problem des wechselseitigen Ausschlusses

Gegeben seien n Prozesse P_0, P_1, ..., P_{n-1}. Bei jedem dieser Prozesse läßt sich der Programmcode in kritische und unkritische Bereiche aufteilen. Ein **kritischer Bereich** ist dabei definiert als Phase, in der ein Prozeß gemeinsame (**globale**) Daten benutzt. Die anderen Phasen des Prozesses nennt man analog dazu **unkritische Bereiche**:

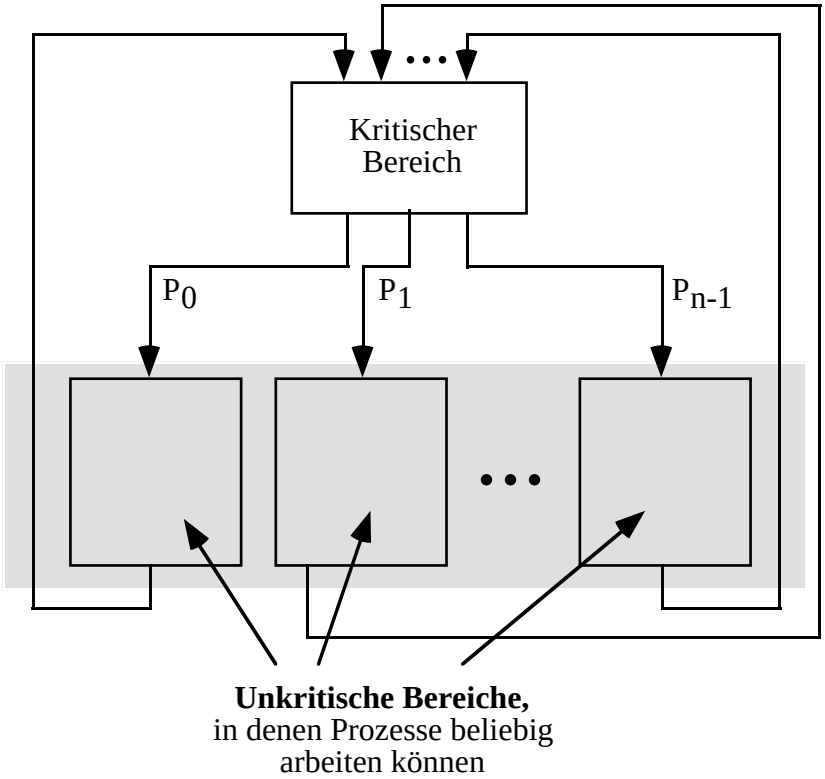


Abb. 3.3: Kritische und unkritische Bereiche

Befinden sich nun zwei Prozesse gleichzeitig in ihren jeweiligen kritischen Bereichen, so ist das Ergebnis ihrer Operationen (wie eben exemplarisch dargestellt) möglicherweise nicht mehr determiniert, sondern von zufälligen Umständen (etwa der Scheduling-Strategie) abhängig. Verhindern läßt sich dies dadurch, daß man jeweils nur einem einzigen Prozeß den Zugang zu seinem kritischen Bereich gestattet. Umgekehrt heißt dies, daß kein anderer Prozeß mehr seinen kritischen Bereich betreten darf, sobald ein Prozeß sich in seinem kritischen Bereich aufhält. Dies ist eine

hinreichende Bedingung dafür, daß Inkonsistenzen wie die beschriebenen ausgeschlossen werden.

Das **wechselseitige Ausschlußproblem** besteht nun in der Erstellung von Algorithmen, die dies gewährleisten. Dazu stehen zwei Mechanismen zur Verfügung: eine "Sperre", die es für jeden Prozeß vor dem Betreten des kritischen Bereichs zu überwinden gilt, und ein "Freigabemechanismus", der das Verlassen des kritischen Bereichs signalisiert.
Ganz so einfach ist die Lösung freilich nicht, denn sehr schnell stellt sich heraus, daß "Sperre" und "Freigabe" selbst wieder kritische Bereiche sind. Damit die "Sperre" funktioniert, muß sie unteilbare (**atomare**) Operationen enthalten. Dies können z. B. einfache Assemblerbefehle (wie etwa einfache Lese-/Schreibzugriffe oder Test-and-Set-Anweisungen) sein. Aus solchen einfachen atomaren Operationen lassen sich dann komplexe "atomare Bereiche" (Makros) konstruieren. Dies ist allein schon deshalb zweckmäßig, da die Programmierung ohne Makros (also einzig unter Verwendung von elementaren atomaren Operationen) sehr schnell unübersichtlich werden kann.

Eine korrekte Lösung des wechselseitigen Ausschlußproblems muß folgende drei Bedingungen erfüllen:

1. Mutual exclusion
Zu jedem Zeitpunkt darf sich höchstens ein Prozeß in seinem kritischen Bereich aufhalten.

2. Progress
Befindet sich kein Prozeß in seinem kritischen Bereich, und gibt es andererseits Prozesse, die ihren kritischen Bereich betreten möchten, so hängt die Entscheidung, welcher Prozeß als nächster seinen kritischen Bereich betreten darf, nur von diesen Kandidaten ab und fällt in endlicher Zeit. Die Entscheidung ist also unabhängig von der Ausführungsgeschwindigkeit der Prozesse (insbesondere hat es keinen Einfluß, wenn ein Prozeß in seinem unkritischen Bereich "stirbt").

3. Bounded waiting
Nachdem ein Prozeß sein Interesse am Betreten des kritischen Bereiches kundgetan hat, kann es vorkommen, daß zwischendurch noch andere Prozesse die Erlaubnis zum Betreten ihres kritischen Bereiches erhalten, bevor besagter Prozeß an die Reihe kommt. Es muß aber möglich sein, eine endliche obere Schranke für die Zeit anzugeben, die unser Prozeß warten muß, bis er dran ist.

Es sei ausdrücklich darauf hingewiesen, daß die dritte Bedingung den Ausschluß von Prioritätsregelungen beinhaltet und außerdem impliziert, daß jeder Prozeß seinen kritischen Bereich in endlicher Zeit wieder verläßt (also nicht etwa darin stirbt oder in einer Endlosschleife landet). Setzt man letzteres voraus, so läßt sich bounded waiting am leichtesten durch Angabe einer Schranke für die Anzahl der Prozesse nachweisen, die unserem Prozeß vorgezogen werden dürfen.

3.2.1 Zwei untaugliche Versuche und eine Lösung

Die folgenden Überlegungen beschränken sich zunächst auf Algorithmen, die für zwei gleichzeitig arbeitende Prozesse P_i und P_j angewendet werden können. Eine naheliegende Idee besteht darin, "einfache Lese- und Schreibzugriffe" als unteilbare Operationen realisieren. Beispielsweise können Befehle wie `turn := j` oder `choosing[i] := FALSE` atomar sein. Damit ließe sich z. B. ein Schalter `turn` heranziehen, der anzeigt, welcher der beiden Prozesse mit dem Betreten seines kritischen Bereichs an der Reihe ist. Nachdem ein Prozeß seinen kritischen Bereich verlassen hat, zeigt der Schalter auf den anderen der beiden Prozesse.

Dieser Ansatz führt zu folgendem Algorithmus für Prozeß P_i (analog P_j):

```
repeat

  Unkritischer Bereich;

  while (turn ≠ i) do noop;

  Kritischer Bereich;

  turn := j;

until FALSE;
```

Diese Lösung ist allerdings nicht korrekt. Falls nämlich "`turn = j`" gilt und P_j in seinem unkritischen Bereich stirbt, kann P_i nie mehr in seinen kritischen Bereich eintreten.

Dieses Problem läßt sich dadurch vermeiden, daß die Sperre erst dann ausgelöst wird, wenn wirklich ein Prozeß in seinen kritischen Bereich möchte. So etwas kann man mittels einer booleschen Variablen `flag[i]` in jedem Prozeß P_i realisieren. `flag[i]` zeigt an, daß P_i in seinen kritischen Bereich möchte, und kann von P_j abgefragt werden. Initialisiert wird `flag[i]`mit `FALSE`; der Algorithmus für P_i lautet:

```
flag[i] := FALSE;

repeat

  Unkritischer Bereich;

  flag[i] := TRUE;       (* A *)
  while flag[j] do noop; (* B *)

  Kritischer Bereich;

  flag[i] := FALSE;

until FALSE;
```

Auch diese Lösung ist leider nicht haltbar, da es hier ebenfalls zu zeitlichen Abläufen kommen kann, die zu einem kompletten Stillstand des Systems führen. Einfachstes Beispiel:

Zeit	P_i	P_j
1	`flag[i] := TRUE;`	
2		`flag[j] := TRUE;`
3		`while flag[i] do noop;`
4	`while flag[j] do noop;`	

An dem erreichten Zustand wird sich in alle Ewigkeit nichts mehr ändern. Man kann sich leicht selbst davon überzeugen, daß auch ein Vertauschen der Reihenfolge von A und B die Situation nicht verbessert.

Eine Kombination der beiden Algorithmen zugrundeliegenden Ideen aber führt zu einer Lösung des wechselseitigen Ausschlußproblems, die alle drei Bedingungen erfüllt:

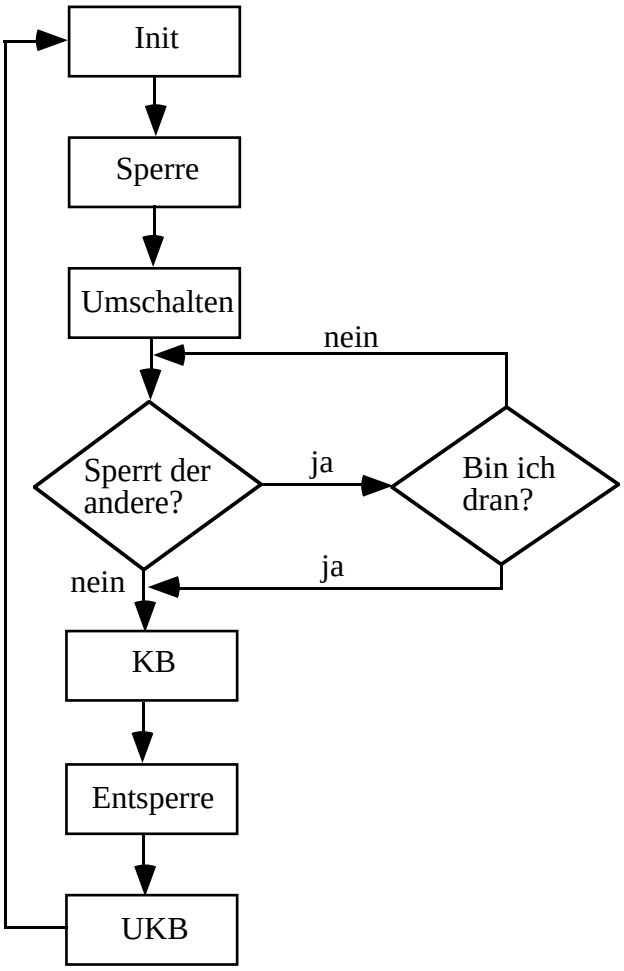


Abb. 3.4: Korrekte Lösung für das Problem des wechselseitigen Ausschlusses

Oder etwas formaler (für Prozeß P_i):

```
repeat

  flag[i] := TRUE;
  turn := j;
  while (flag[j] and turn = j) do noop;

  Kritischer Bereich;

  flag[i] := FALSE;

  Unkritischer Bereich;

until FALSE;
```

Warum ist diese Lösung korrekt?

1. Bedingung: Mutual exclusion (höchstens ein Prozeß im Kritischer Bereich)

P_i kann genau dann seinen kritischen Bereich betreten, wenn die Bedingung (`flag[j] and turn = j`) nicht erfüllt ist. Also ist `flag[j] = FALSE` oder `turn = i`, d. h. entweder möchte P_j nicht oder P_j ist nicht dran - jedenfalls kann P_j seine `while`-Schleife nicht überwunden haben und befindet sich mithin nicht in seinem kritischen Bereich.

2. Progress (Eintritt nach endlicher Zeit)

1. Fall: Beide Prozessen möchten in den KB.
   Dann hängt die Entscheidung nur von der Bedingung `turn = i` bzw. `turn = j ab.` Dies ist auf jeden Fall für `i` oder `j` erfüllt ⇒ einer der beiden Prozesse betritt den KB nach endlicher Zeit.
2. Fall: Nur P_i möchte in den KB (P_j analog).
   P_i setzt `flag[i] := TRUE` und `turn := j.` Wegen der UND-Verknüpfung in der `while`-Schleife und da `flag[j] = FALSE` ist, folgt: P_i betritt KB nach endlicher Zeit.

3. Bounded waiting (nur endlich oft Vortritt für andere Prozesse)

P_i kann maximal einmal überholt werden. Sobald nämlich P_i in der `while`-Schleife angekommen ist, muß er nur dann warten, wenn `flag[j] = TRUE` und

1. P_j noch vor `turn := i` ist oder
2. P_j `turn := i` passiert hat, bevor P_i `turn := j` gesetzt hat.

Da P_j nur im UKB sterben darf, wird er im 1. Fall nach endlicher Zeit `turn := i` bzw. im 2. Fall `flag[j] := FALSE` setzen. Von da an ist die Schleifenbedingung von P_i aber nicht mehr erfüllt, und sobald P_i diese Bedingung abgefragt hat, kann er in seinen kritischen Bereich eintreten.

Eine Erweiterung dieses Algorithmus' für n > 2 Prozesse P_1, P_2, ..., P_n stellt sich als keineswegs trivial heraus, wenn weiterhin nur einfache Lese-/Schreiboperationen in atomarer Implementierung zur Verfügung stehen. Es sei hier nur auf den Algorithmus von Eisenberg und McGuire verwiesen, ohne näher darauf einzugehen.

3.2.2 Der Bakery-Algorithmus

Einen wesentlichen Fortschritt stellt der sogenannte "Bakery-Algorithmus" dar. Im Alltag kann man dem hier realisierten Prinzip des öfteren begegnen, beispielsweise in der Zahnarztpraxis, auf dem Einwohnermeldeamt oder eben (vereinfacht) in Bäkkereien.

Der Algorithmus beruht auf der Ausgabe von Nummern für wartende Kunden. Solange keiner wartet, steht der Ticketzähler auf Null. Kommt der erste Interessent an, so zieht er eine Nummer ≥ 0. Später ankommende Kunden lösen weitere Tickets mit jeweils höheren Nummern. Es ist allerdings nicht ausgeschlossen, daß die gleiche Nummer an mehrere gleichzeitig ankommende Kunden vergeben wird. Die Bedienung erfolgt dann nach folgender **Strategie**:
Die kleinste Nummer ist jeweils als nächste dran. Falls sie an mehrere Kunden vergeben wurde, kommt ("Tie-break") derjenige dran, dessen Name im Alphabet am weitesten vorne steht (bzw. dessen Prozeßnummer am niedrigsten ist).

Als kritischen Bereich kann man leicht das Programmstück ausmachen, das die Ziehung der Nummern regelt. Im Algorithmus verwendet man hier ein Variablenfeld `choosing[i]`, das für jedes i mit `FALSE` vorbesetzt ist. Setzt Prozeß P_i seinen Feldwert `choosing := TRUE`, so signalisiert er damit, daß er eine Nummer ziehen will (sprich auf gemeinsame Daten zugreift). Der Algorithmus stellt dann sicher, daß diese gemeinsame Daten während des Zugriffs von P_i nicht von anderen Prozessen verändert werden, da die anderen ihrerseits vorher `choosing[i] = TRUE` abfragen müssen. Nach erfolgter Nummernziehung setzt P_i dann wieder seinen Feldeintrag `choosing[i] = FALSE`.

Vereinbart man nun noch, daß

$$(a,b) < (c,d) \iff \text{entweder } a < c \text{ oder } (a = c \text{ und } b < d)$$

gilt, so lautet der Algorithmus (für den i-ten von n Prozessen) wie folgt:

```
repeat

  choosing[i] := TRUE;
  number[i] := max(number[0], number[1], ..., number[n-1]) + 1;
  choosing[i] := FALSE;
  for j := 0 to n-1 do
    begin
    while choosing[j] do noop;
    while number[j] <> 0 and (number[j], j) < (number[i], i)
      do noop
    end;

  Kritischer Bereich;

  number[i] := 0;

  Unkritischer Bereich;

until FALSE;
```

Die Korrektheit dieser Lösung soll hier nicht im Detail nachgewiesen werden. Daß jeweils höchstens einer der Prozesse im kritischen Bereich ist, wird durch die oben

dargelegte Strategie (einschließlich der Tie-break-Regelung) gewährleistet. Da die Abarbeitung der Schlange im wesentlichen nach FIFO geschieht, ist auch progress und bounded waiting gewährleistet.
Zwei kleine Schönheitsfehler könnte man dieser Lösung ankreiden: Einmal ist der Algorithmus nicht ganz fair, da Prozesse mit kleinen Nummern einen winzigen Vorteil erhalten. Außerdem könnte man sich Szenarien ausdenken, in denen der Algorithmus bei Verwendung einer beschränkten Ganzzahl-Arithmetik versagt.

3.2.3 Enqueue und Dequeue

Eine Variante des Bakery-Algorithmus verwendet die atomaren Operationen `enqueue` und `dequeue` zur Verwaltung einer FIFO-Warteschlange von n Prozessen P_1, ..., P_n. Die Warteschlange kann man sich als verkettete Liste von Prozeßnummern vorstellen, die in einem Array `F[0..n]` verwaltet wird. `F[0]` ist dabei die Nummer des Prozesses, der aktuell in seinem kritischen Bereich ist, `F[F[0]]` ist die Nummer des "ältesten" (d. h. des am längsten wartenden) Prozesses usw. Wartet kein Prozeß, so ist `F[0]` = `0`. Die atomaren Operationen lauten dann:

```
enqueue(i):    {F[p] := i; p := i}

dequeue(i):    {if p ≠ i then F[0] := F[i]
                         else (p := 0; F[0] := 0)}
```

`enqueue(i)` ordnet also den Prozeß P_i in die Warteschlange an letzter Position ein, während `dequeue(j)` den ältesten Prozeß P_j aus der Warteschlange eliminiert. `p` ist dabei jeweils ein Zeiger auf den Platz im Array `F`, dessen Nummer der Nummer des bislang jüngsten Prozesses entspricht. Die Bedingung "`p ≠ i`" überprüft, ob sich hinter P_i noch weitere Prozesse in der Warteschlange befinden.

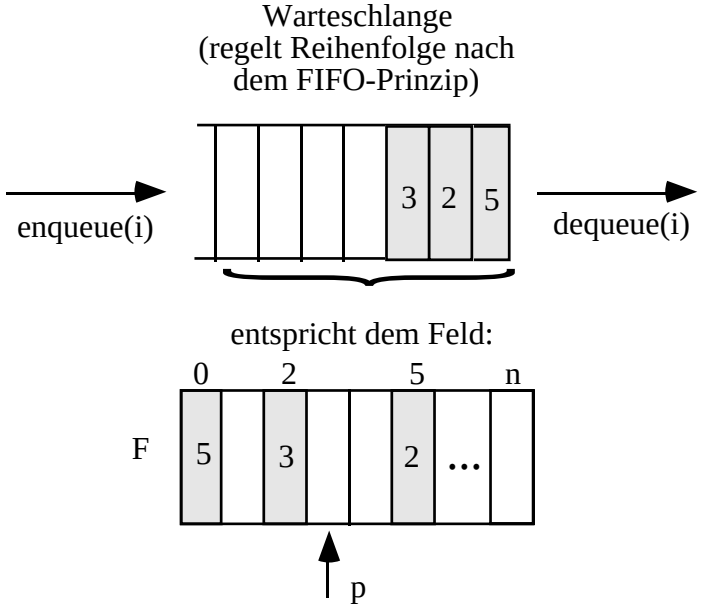


Abb. 3.5: Arbeitsweise von enqueue und dequeue

Der Algorithmus für P_i hat dann folgende Gestalt:

```
repeat
```

```
  Unkritischer Bereich;

  enqueue(i);
  while F[0] ≠ i do noop;

  Kritischer Bereich;

  dequeue(i);
until FALSE;
```

Sind `enqueue` oder `dequeue` nicht atomar, so kann es leicht zu Schwierigkeiten kommen, wie wir an einem Beispiel zeigen wollen. Sei hierfür `enqueue` teilbar und `p` zu 0 initialisiert. Wir betrachten die Prozesse P_i und P_k:

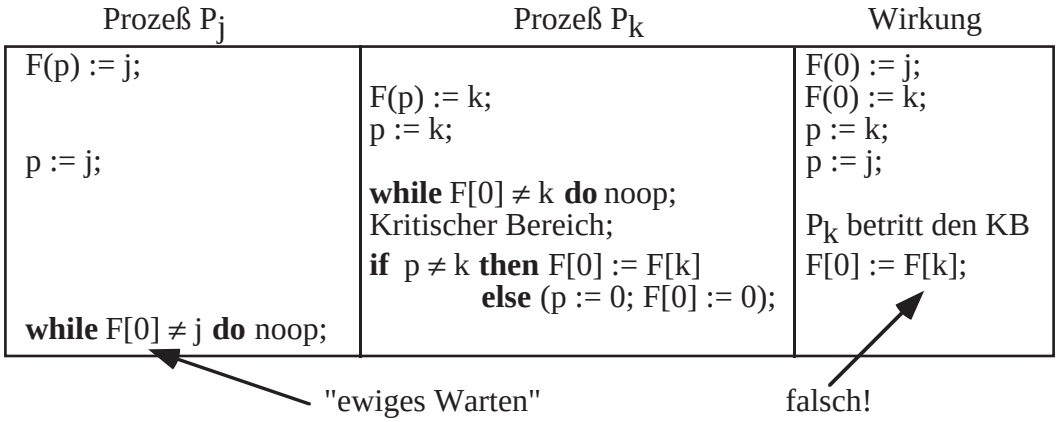


Abb. 3.6: Problematik bei teilbaren enqueue- und dequeue-Operationen

3.2.4 Synchronisationsmechanismen mit atomaren Operationen

Interrupt

Am einfachsten lassen sich atomare Operationen über die Interrupt-Steuerung realisieren. Das Programmstück, das atomar sein soll, wird dabei eingeleitet von einem Befehl, der einen Interrupt unmöglich macht, und schließt mit einem Befehl, der Interrupts wieder ermöglicht:

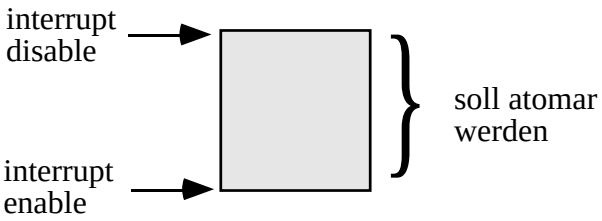


Abb. 3.7: Kritische Bereiche und Interrupts

Dieses Verfahren stellt eine Lösung dar, da wir sicher sein können, daß ein Prozeß, der in seinem kritischen Bereich ist, nicht gestört werden kann. Es ist allerdings nur für Einprozessorsysteme brauchbar.

Test-and-Set

Eine von vielen Computersystemen hardwaremäßig angebotene Möglichkeit besteht darin, eine boolesche Variable auf ihren Inhalt hin zu testen und diesen gegebenenfalls zu modifizieren, und zwar innerhalb einer unteilbaren Operation. Eine derartige Test-and-Set-Instruktion hat typischerweise folgendes Aussehen:

```
function Test-and-Set(var target: boolean): boolean;
  begin
  Test-and-Set := target;
  target := TRUE;
  end;
```

Diese ganze Operation erfolgt atomar. Das bedeutet, daß im Falle des gleichzeitigen Aufrufs zweier Test-and-Set-Operationen auf verschiedenen CPUs letztlich eine (zeitlich) sequentielle Abarbeitung in beliebiger Reihenfolge stattfindet.

Der Test-and-Set-Mechanismus kann sehr gut zur Lösung des wechselseitigen Ausschlußproblems verwendet werden. Hierzu deklariert man eine globale boolesche Variable `lock`, initialisiert sie einmalig mit `FALSE` und verwendet für jeden Prozeß den nachfolgenden Algorithmus:

```
repeat

  while Test-and-Set(lock) do noop;

  Kritischer Bereich;

  lock := FALSE;

  Unkritischer Bereich;

until FALSE;
```

Es ist gewährleistet, daß immer höchstens ein Prozeß in seinem kritischen Bereich ist. Allerdings ist die bounded-waiting-Bedingung nicht erfüllt, da ein Prozeß beliebig oft Pech haben kann, weil er die Variable `lock` jedesmal erst abfragt, nachdem ihm schon wieder ein anderer Prozeß zuvorgekommen ist. Um dies zu vermeiden, bedarf es eines komplizierteren Algorithmus´, der die Prozesse in zyklischer Reihenfolge testet und den wir hier für Prozeß P_i ohne nähere Erläuterung angeben:

```
var j: 0..n-1;
    key: boolean;

repeat
  waiting[i] := TRUE;
  key := TRUE;
  while waiting[i] and key do key := Test-and-Set(lock);
  waiting[i] := FALSE;
  Kritischer Bereich;

  j := i+1 mod n;
  while (j <> i) and not waiting[j] do j := j+1 mod n;
  if j = i then lock := FALSE
    else waiting[j] := FALSE;

  Unkritischer Bereich;
```

```
until FALSE;
```

Swap

Ein anderer oftmals implementierter Mechanismus erlaubt das atomare Vertauschen des Inhalts zweier boolescher Variablen. Diese Swap-Operation sieht folgendermaßen aus:

```
procedure Swap (var a, b: boolean);
var temp: boolean;

begin
temp := a;
a := b;
b := temp;
end;
```

Man nutzt sie ähnlich wie "Test-and-Set" zum atomaren "Aufschließen" eines globalen "Schlosses" `lock` mit einem lokalen "Schlüssel" `key`, beides boolesche Variable. `lock` wird zu Beginn auf `FALSE` gesetzt, um anzuzeigen, daß sich kein Prozeß in seinem kritischen Bereich befindet. Ein Prozeß P_i, der in seinen kritischen Bereich möchte, setzt seinen `key` auf `TRUE` und beginnt, den Inhalt von `key` und `lock` mittels `swap` zu vertauschen, um festzustellen, ob der Zugang zum kritischen Bereich gerade möglich ist. Solange ein anderer Prozeß P_k in seinem kritischen Bereich ist, bleibt `lock = TRUE`, und die Vertauschungsaktionen von P_i ändern nichts am Status quo. Verläßt P_k seinen kritischen Bereich, so setzt er `lock` auf `FALSE`. Bei der nächsten Vertauschungsaktion merkt P_i dann, daß er in seinen kritischen Bereich kann, und hat zugleich nach außen bekanntgegeben, daß für andere wartende Prozesse der Zugang jetzt wieder gesperrt ist:

```
repeat

  key := TRUE;
  repeat
    Swap(lock, key);
  until key = FALSE;

  Kritischer Bereich;

  lock := FALSE;

  Unkritischer Bereich;

until FALSE;
```

Die grundsätzliche Idee ist praktisch dieselbe wie beim Test-and-Set-Vorschlag. Daher verwundert es nicht, daß auch der Swap-Algorithmus kein bounded waiting garantiert. Übrigens ist global jeweils nur bekannt, daß ein Prozeß im kritischen Bereich ist, man kann aber nicht feststellen, welcher es ist.

3.3 Semaphore

3.3.1 Das Konzept eines Semaphors

Die bisher vorgestellten Software-Lösungen für das wechselseitige Ausschlußproblem basieren auf elementaren atomaren Operationen (wie Lese-/Schreibzugriffen, Test-and-Set, Swap) und werden bei der Übertragung auf komplexere Aufgabenstellungen sehr schnell unübersichtlich und schwer zu verifizieren. Daher liegt es nahe, höhersprachliche Konstrukte zu entwickeln, die ein größeres Maß an Übersichtlichkeit ermöglichen und dadurch weniger schwer zu verstehen und vor allem weniger fehleranfällig sind. Realisiert werden sie ihrerseits dann durch einfache atomare Konzepte, wie wir sie bisher kennengelernt haben.

Ein erstes wichtiges Beispiel eines solchen komplexeren Hilfsmittels finden wir im Konzept der **Semaphoren**. Veranschaulichen kann man sich einen Semaphor beispielsweise anhand eines Aachener Parkhauses, das an seinem Eingang einen Zähler besitzt. Am frühen Morgen wird der Zähler initialisiert und zeigt die Anzahl der verfügbaren Parkplätze an. Sobald ein Auto in das Parkhaus hineinfährt, wird der Zähler am Eingang um eins verringert. Verläßt ein Wagen das Parkhaus, so wird der Zähler wieder um eins erhöht. Steht der Zähler auf 0, so muß ein ankommendes Fahrzeug solange um den Block fahren, bis ein Auto das Parkhaus verläßt und dem Zähler signalisiert hat, daß jetzt wieder ein freier Platz zur Verfügung steht. Das Initialisieren, Inkrementieren und Dekrementieren des Zählers erfolgt dabei stets atomar.

Eine Abstraktion dieses Mechanismus führt zum Semaphorkonzept. Ein Semaphor `S` ist demnach eine Integervariable, die nur durch drei atomare Operationen namens `init`, `wait` und `signal` verändert werden kann. In den meisten Fällen ist mit einer Semaphor außerdem eine "**assoziierte Warteschlange**" verknüpft, um auszuschließen, daß Prozesse über längere Zeit in einer Abfrageschleife verweilen und somit unnötigerweise die CPU belegen (bzw. in der automobilistischen Variante ständig um den Block fahren, anstatt den Motor abzustellen und vor dem Eingang auf das Freiwerden des nächsten Parkplatzes zu warten). Assoziierte Warteschlangen verhindern ein derartiges "**busy waiting**", indem sie die Prozeßnummern in der Reihenfolge, in der sie dran sind, festhalten, woraufhin sich die wartenden Prozesse "schlafenlegen" können und daher einstweilen keine CPU-Zeit mehr beanspruchen. Jedesmal, wenn der kritische Bereich dann frei wird, läßt sich der Warteschlange entnehmen, welcher Prozeß als nächster an die Reihe kommt und demzufolge zu wecken ist.

Nun zur Definition der drei atomaren Operationen:

init(S, Anfangswert)

Die Wirkung dieser Operation ist `S := Anfangswert.` Hierdurch wird also der Semaphor `S` initialisiert. Als Anfangswert nimmt man sehr oft die Zahl von Prozessen, die sich gleichzeitig in einem Bereich aufhalten dürfen (in unserem Beispiel etwa die Anzahl der anfangs freien Parkplätze). Beim wechselseitigen Ausschlußproblem ist die Anzahl der Prozesse im kritischen Bereich auf höchstens 1 begrenzt, daher initialisiert man hier mittels `init(S,1)`. Einen derartigen Semaphor, der nur die Werte 0 oder 1 annehmen kann, nennt man übrigens **binär** im Gegensatz zu einem **Zählsemaphor**, dessen Wertebereich nicht nach oben beschränkt ist.

wait(S) (historisch auch P(S) von niederländisch "proberen")

Die Wirkung dieser Instruktion entspricht

```
{while S <= 0 do noop;
 S := S-1;}
```

Man muß sich dabei unbedingt ins Gedächtnis rufen, daß diese ganze Operation atomar erfolgt. Ist mit `S` eine Warteschlange assoziiert, so lautet der atomar implementierte Algorithmus:

```
{S := S-1;
 if S < 0 then (ordne Prozeß in Warteschlange an Position -S
                ein);}
```

signal(S) (oder auch V(S) von niederl. "verhogen" = erhöhen)

Diese letzte Semaphoroperation bewirkt

```
{S := S+1}
```

bzw. mit assoziierter Warteschlange

```
{if S ≤ 0 then (wecke den ältesten wartenden Prozeß auf und
                schicke ihn in den kritischen Bereich;}
```

Beides ist wiederum atomar zu realisieren.

Hat man einen Semaphor `S` samt der drei angegebenen Operationen zur Verfügung, so kann man folgenden prinzipiellen Lösungsalgorithmus für das wechselseitige Ausschlußproblem angeben:

globale Initialisierung:

```
init(S,1);
```

Algorithmus für Prozeß P_i :

```
repeat
  wait(S);
  Kritischer Bereich;
  signal(S);
  Unkritischer Bereich;
until FALSE;
```

in direkter Analogie zum Parkhausbeispiel (wenn man von der etwas unrealistischen Voraussetzung ausgeht, das Parkhaus bestehe aus genau einem Parkplatz).

Semaphore lassen sich auch anderweitig einsetzen, etwa im Bereich der Ablaufsteuerung: Hat man zwei Prozesse X und Y und will gewährleisten, daß Y vor X ausgeführt wird, so läßt sich dies einfach erreichen, indem man einen Semaphor `a` benutzt, ihn mit 0 initialisiert sowie am Anfang von X den Befehl `wait(a)` und am Ende von Y den Befehl `signal(a)` einfügt. Diese "umgekehrte Semaphornutzung" bewirkt, daß X solange warten muß, bis Y fertig ist.

Vor allem aber verwendet man Semaphoren zur Lösung von Koordinationsaufgaben wie dem bereits erwähnten Erzeuger-Verbraucher-Problem, den Reader-Writer-Problemen, dem Fünf-Philosophen-Problem und einer ganzen Reihe verwandter Aufgaben.

3.3.2 Erzeuger-Verbraucher-Problem

Betrachten wir zunächst noch einmal das Erzeuger-Verbraucher-Problem für den Fall von n Erzeugern, m Verbrauchern und einem Zwischenlager der Größe `MAX`. Die Erzeugerprozesse produzieren und legen im Lager ab, solange der dortige Bestand weniger als `MAX` beträgt. Die Verbraucherprozesse entnehmen ihrerseits Elemente aus dem Lager, solange der Bestand größer als 0 ist, und konsumieren. Teilt man die jeweiligen Prozesse auf, so ergibt sich beim Erzeuger das Produzieren und beim Verbraucher das Konsumieren als unkritischer Bereich, während das Ablegen bzw. Entnehmen als kritischer Bereich anzusehen ist, da Erzeuger wie Verbraucher dabei auf gemeinsame Daten zugreifen müssen.

Neu an der jetzt vorgestellten Lösung ist nun, daß der exklusive Zugriff auf das Lager mittels eines Semaphors `s` realisiert werden soll. Initialisiert wird `s` durch `init(s,1)`. Dann sieht der Sperrmechanismus zur Sicherung des exklusiven Zugriffs schematisch so aus:

```
...
wait(s);
Kritischer Bereich;
signal(s);
...
```

Außerdem lassen sich die beiden Nebenbedingungen (daß zum einen in ein volles Lager nichts gelegt und zum anderen aus einem leeren Lager nichts entnommen werden kann) über zwei zusätzliche Semaphoren `f` und `c` verwirklichen. Dabei bedeutet

`f` = 0 <=> Lager ist leer
`c` = 0 <=> Lager ist voll

Die Initialisierung geschieht mittels `init(f,0)` und `init(c,MAX)`. Während der gesamten Laufzeit soll dann `f+c` invariant bleiben, d. h. Inkrementierung von `f` bedingt Dekrementierung von `c` und vice versa. Die Algorithmen lauten :

ERZEUGER:

```
repeat
  produziere ein Element und lege es in nextp ab;
  wait(c);
  wait(s);
  füge nextp in den Puffer ein;
  signal(s);
  signal(f);
until FALSE;
```

VERBRAUCHER:

```
repeat
  wait(f);
  wait(s);
  entferne Element aus dem Puffer und lege es unter nextc ab;
  signal(s);
  signal(c);
  verbrauche das Element in nextc;
until FALSE;
```

Bemerkenswert ist, daß eine Vertauschung der "`signal`"-Instruktionen nichts an der Korrektheit ändert, wohl aber ein Vertauschen der "`wait`"-Befehle.

3.3.3 Reader-Writer-Probleme

Das nächste klassische Problem taucht vor allem im Zusammenhang mit der Verwaltung von Datenbanken auf. Hierbei haben zwei Arten von Prozessen Zugriff auf ein gemeinsames Datenobjekt (z. B. ein File): Prozesse der ersten Sorte, die "Writer", machen Updates der Daten. Dabei muß sichergestellt werden, daß ihr Schreibzugriff exklusiv erfolgt, d. h. kein anderer Prozeß (egal ob Writer oder Reader) darf während des Updates aktiv sein. Die zweite Art von Prozessen (die "Reader") stellen Anfragen, die simultan stattfinden können. Man kann sich das beispielsweise an einem Zugfahrplan veranschaulichen, der am Bahnhof aushängt: Ein Fahrplanwechsel (also ein Update) wird exklusiv durchgeführt (d. h. ein einzelner Bahnbeamter wechselt den aushängenden Fahrplan in einem Augenblick, in dem ihn niemand konsultiert), während eine prinzipiell beliebige Anzahl von Interessenten gleichzeitig im aushängenden Plan nach Abfahrtszeiten, Gleisnummern etc. suchen kann.

Man unterscheidet nun drei Varianten des Reader-/Writer-Problems (nach ihrem "Erfinder" P. J. Courtois auch "**Courtois-Probleme**" genannt), die sich hinsichtlich ihrer Prioritätsregelungen unterscheiden:

1. R/W-Problem:
Kein Reader muß auf eine Eintrittserlaubnis warten, es sei denn, es ist gerade ein Writer in seinem kritischen Bereich. Hier kann es sehr schnell passieren, daß die Reader das System monopolisieren, d. h. ein Writer, der gerne ein Update machen möchte, kann von einem Verschwörerkreis aus Readern auf ewig vom Zugang abgehalten werden ("**starvation problem**").

2. R/W-Problem:
Man kann versuchen, dieser starvation zu begegnen, indem man neu hinzukommenden Lesewilligen den Zutritt untersagt, sobald ein Writer wartet. Jetzt können jedoch die Writer das Monopol ausüben, so daß keinem Reader mehr der Zutritt in seinen kritischen Bereich gelingt.

3. R/W-Problem:
Hier wird Fairneß dadurch herzustellen versucht, daß man Lese- und Schreibphasen alternieren läßt: Ist gerade Lesephase und meldet sich ein Writer an, so werden keine neuen Reader mehr zugelassen. Sobald alle Reader fertig sind, darf der Writer zugreifen. Ist andererseits gerade eine Schreibphase und meldet sich ein Reader an, so wird das Ende dieser Schreibphase abgewartet und dann eine Lesephase gestartet.

Im folgenden soll eine Lösung für das erste R/W-Problem angegeben werden, die Semaphore und Zählvariable verwendet, und zwar:

Semaphor `s`: "Schreibphase"
Semaphor `w`: "Lesephase"
Zählvariable `n`: zählt Anzahl der gleichzeitig aktiven Leser

Der Semaphor `s` wird durch `init(s,1)` initialisiert und wird sowohl von Reader- als auch von Writer-Prozessen verwendet, um Writern einen exklusiven Zugriff zu ermöglichen. Der Semaphor `w` (ebenfalls mit `init(w,1)`) soll sicherstellen, daß ein Update von `n` ungestört erfolgen kann. Zu klären ist schließlich noch die Initialisierung von `n`: naheliegenderweise setzt man `n` zu Beginn auf 0. Dann ergeben sich folgende Algorithmen:

WRITER:

```
repeat
  wait(s);
  schreibe;
  signal(s);
until FALSE;
```

READER:

```
repeat
  wait(w);
  n := n + 1;
  if n = 1 then wait(s);
  signal(w);
  lies;
  wait(w);
  n := n - 1;
  if n = 0 then signal(s);
  signal(w);
until FALSE;
```

Wenn sich hierbei ein Writer in seinem kritischen Bereich aufhält und m Reader warten, so wartet einer davon in der zu `s` assoziierten Warteschlange, während m-1 in der zu `w` gehörenden Schlange gelandet sind.

3.3.4 Das Fünf-Philosophen-Problem

Dies ist ein weiterer Klassiker, dessen Berühmtheit allerdings weniger auf etwaige praktische Relevanz zurückzuführen ist, sondern eher darauf, daß man an diesem Problem besonders schön das Zusammenspiel von Prozessen und die Gefahr einer Verklemmung darstellen kann. Hier zunächst eine anschauliche Darstellung des Sachverhalts.

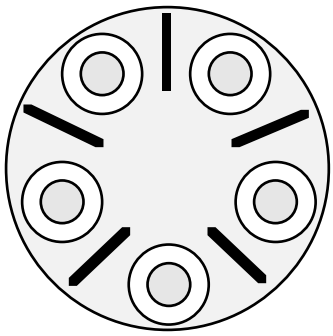


Abb. 3.8: Das Fünf-Philosophen-Problem

Philosophen verbringen (zugegebenermaßen leicht abstrakt dargestellt) ihr Leben mit Denken und Essen. Betrachten wir also fünf östliche Exemplare davon, die sich um einen runden Tisch versammelt haben. Vor jedem von ihnen befindet sich ein Teller voll Reis, und zwischen zwei Tellern liegt jeweils eines der aus Chinarestaurants hinreichend bekannten Eßstäbchen. Essen kann auch ein Philosoph nur unter Zuhilfenahme zweier dieser Stäbchen.
Von Zeit zu Zeit nun wird einer unserer Helden hungrig und greift hintereinander nach den beiden Stäbchen, die rechts und links von seinem Teller liegen. Natürlich kann er ein Stäbchen nur dann in die Hand nehmen, wenn es gerade nicht vom betreffenden Nachbarn benutzt wird. Ist es dem hungrigen Philosophen geglückt, beide Stäbchen an sich zu bringen, so ißt er, ohne sie jemals loszulassen, bis er satt ist, woraufhin er sie an ihren ursprünglichen Platz zurücklegt und von neuem ins Grübeln verfällt.

Abstrakt gesehen haben wir also fünf Prozesse ("Philosophen") P_0, ..., P_4 vor uns, die "kreisförmig" angeordnet sind. Ferner sind noch fünf Betriebsmittel ("Stäbchen") ρ_0 ...ρ_4 gegeben. Jeder Prozeß P_i braucht zeitweise zwei Betriebsmittel ρ_i ("linkes Stäbchen") und $\rho_{(i+1) \bmod 5}$ ("rechtes Stäbchen") gleichzeitig.

Schematisch haben die Prozesse folgende Struktur:

```
repeat
  Unkritischer Bereich ("Denken")
  Kritischer Bereich   ("Essen": Pi braucht  ρ[i] und
                                             ρ[(i+1) mod 5])
until FALSE;
```

Naive Lösung

Am naheliegendsten ist wohl, jedes Stäbchen durch einen Semaphor ρ_0 ... ρ_4 zu repräsentieren, der jeweils durch `init`(ρ_i,1) initialisiert wird. Für den Prozeß P_i steht dann folgender Algorithmus:

```
repeat
  wait(ρ[i]);
  wait(ρ[i+1 mod 5];
  Essen;
```

```
  signal(ρ[i]);
  signal(ρ[i+1 mod 5];
  Denken;
until FALSE;
```

Aber: Bei dieser Lösung sind Verklemmungen (Deadlocks) nicht ausgeschlossen. Eine Verklemmung tritt ein, wenn alle Prozesse annähernd gleichzeitig in ihren kritischen Bereich wollen. Jeder Philosoph greift dann (wahre Philosophen sind in aller Regel Linkshänder!) zunächst nach seinem linken Stäbchen und wartet mit diesem in der Hand darauf, daß er das rechte Stäbchen aufnehmen kann. Da dies jeder der fünf Philosophen tut, steht zu befürchten, daß sie bis in alle Ewigkeit so dasitzen und sich gegenseitig blockieren.

Lösungsmöglichkeiten zur Vermeidung von Deadlocks

Man kann das Eintreten eines Deadlocks durch die Einführung von Zwischenzuständen verhindern:

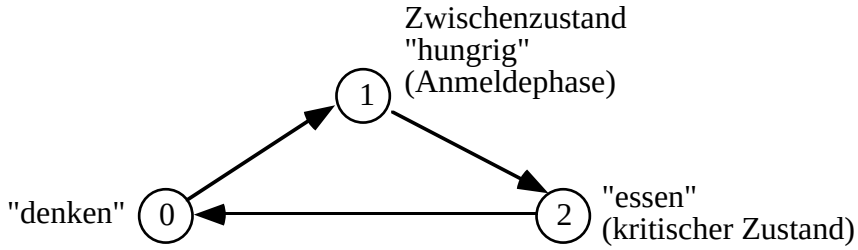


Abb. 3.9: Die drei (einzig) möglichen Zustände eines Philosophen

Der Zustand "essen" bleibt weiterhin kritisch, ebenso symbolisiert der Zustand "denken" auch hier den unkritischen Bereich. Neu ist ein Zustand "hungrig", den ein Philosoph einnimmt, wenn er sich zum Essen sozusagen "anmeldet".

Die Lösung basiert dann auf dem Semaphor `s` (mit `init(s,1)`) sowie den Semaphoren h_0, ..., h_4 (jeweils mit `init(`h_i`,0)`). Dabei ist $h_i = 1$ gleichbedeutend damit, daß beide Stäbchen für P_i frei sind und P_i obendrein hungrig ist. Die Zustände der Philosophen werden mittels der Kontrollvariablen c_i festgehalten, wobei c_i die Werte 0, 1 oder 2 (für "denken", "hungrig" bzw. "essen") annehmen kann.

Außerdem brauchen wir noch eine Testprozedur, die folgendermaßen aussieht:

```
procedure test(k)

if(c[(k-1) mod 5] ≠ 2          (* linker Nachbar ißt nicht *)
     and c[k] = 1              (* ich bin hungrig *)
     and c[(k+1) mod 5] ≠ 2)   (* rechter Nachbar ißt nicht *)
then
  begin
  c[k] := 2;
  signal(h[k]);
  end;
```

Der Algorithmus für den Philosophen P_i lautet damit:

```
repeat

  Denken;
```

```
  wait(s);
  c[i] := 1;              (* hungrig werden*)
  test(i);                (* Test, ob Stäbchen frei *)
  signal(s);

  wait(h[i]);

  Essen;

  wait(s);
  c[i] := 0;              (* zurück zum Denken *)
  test(i+1 mod 5);        (* Test, ob Nachbarn essen können *)
  test(i-1 mod 5);
  signal(s);

until FALSE;
```

Bei dieser Lösung ist das Eintreten eines Deadlocks ausgeschlossen. Allerdings kann es immer noch vorkommen, daß einzelne Philosophen verhungern, d. h. es ist möglich, daß ein Prozeß auf unbestimmte Zeit blockiert wird. Betrachten wir z. B. den hungrigen Prozeß P_2. P_1 und P_3 können sich dergestalt gegen ihn verschwören, daß zunächst P_1 mit den Stäbchen ρ_1 und ρ_2 ißt. Kurz bevor P_1 sein Mahl beendet, beginnt P_3, mit den Stäbchen ρ_3 und ρ_4 zu essen. In ähnlicher Weise kurzzeitig überlappend startet dann wieder P_1 seine Eßphase usw. bis zum bitteren Ende von P_2.

Eine andere Möglichkeit, gegen die Gefahr eines Deadlocks vorzugehen, besteht in der Einführung einer Asymmetrie unter den Prozessen, indem man einen Philosophen (etwa P_4) einfach zum Rechtshänder umpolt. Für die Algorithmen bedeutet das:

P_0 ... P_3 (Linkshänder)	P_4 (Rechtshänder)
wait(ρ_i)	wait(ρ_0)
wait(ρ_{i+1})	wait(ρ_4)
Essen	Essen
signal(ρ_i)	signal(ρ_0)
signal(ρ_{i+1})	signal(ρ_4)

Warum ist dieser Vorschlag deadlockfrei? Wie wir später noch ausführlich sehen werden, wenn wir zur eigentlichen Besprechung von Deadlocks kommen, ist für eine Verklemmung die Existenz einer zyklischen Kette charakteristisch. Dabei steht ein Pfeil vom Stäbchen zum Philosophen dafür, daß der Philosoph das Stäbchen "hat", während ein Pfeil vom Philosophen zum Stäbchen symbolisiert, daß der Philosoph das Stäbchen "will":

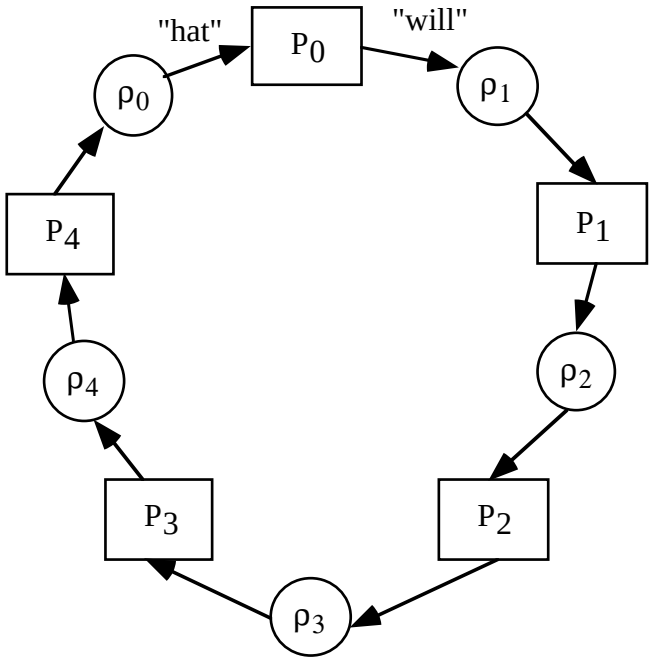


Abb. 3.10: Zyklisches Warten der Philosophen auf die Stäbchen

Auf unseren Fall übertragen liegt ein Deadlock vor, wenn gilt:

P_0 hat ρ_0 und will ρ_1
P_1 hat ρ_1 und will ρ_2
.....
P_4 hat ρ_4 und will ρ_0

Dies kann nur passieren, wenn alle Philosophen Linkshänder sind. Ist ein Rechtshänder (hier: P_4) dabei, so gehört ein Stäbchen (hier: ρ_4) zu keiner zyklischen Kette, und damit ist kein Deadlock mehr möglich.

In vielen Fällen, insbesondere für die Darstellung komplexerer Sachverhalte, sind die bisher vorgestellten Konzepte einschließlich der Semaphoren noch zu unhandlich, unübersichtlich und daher fehleranfällig. Aus diesem Grund greift man oft entweder auf abstraktere Notationen (wie Petrinetze) oder höhersprachliche Konstrukte (wie Regionen und Monitore) zurück, wie sie im folgenden dargestellt werden.

3.4 Petrinetze

3.4.1 Einführung

Petrinetze sind ein graphisches und mathematisches Hilfsmittel, das es erlaubt, viele der bei der Informationsübertragung und -verarbeitung auftretenden Erscheinungen in einheitlicher Weise zu beschreiben. Besonders geeignet sind Petrinetze zur Modellierung von nebenläufigen, asynchronen, verteilten, parallelen, nicht-deterministischen und/oder stochastischen Systemen. Die graphische Art der Darstellung ermöglicht eine Veranschaulichung komplexer Abläufe ähnlich wie dies etwa Flußdiagramme tun. Außerdem erlauben sie die formale (mathematische) Analyse qualitativer (Lebendigkeit, Erreichbarkeit, Verklemmungen) wie quantitativer (Verzögerung, Durchsatz) Systemeigenschaften.

Eingeführt wurden die Petrinetze von C. A. Petri 1962 im Rahmen seiner Dissertation. Seither hat man, um den sehr unterschiedlichen Anforderungen und Zielsetzungen in den verschiedensten Anwendungsbereichen gerecht werden zu können, eine Reihe neuer Petrinetzkonzepte entwickelt, die teilweise deutlich von der ursprünglichen Idee abweichen. Einfache Petrinetzkonzepte erlauben die Anwendung kraftvoller mathematischer Verfahren zur Analyse des modellierten Systems, während komplexere Konzepte zwar das betreffenden System wesentlich detaillierter nachbilden können, dafür aber mathematisch sehr schnell unzugänglich werden. Hierin liegt auch die maßgebliche Schwäche aller Petrinetze: Mit ihrer Hilfe entwikkelte Modelle tendieren schnell dahin, zu umfangreich für eine sinnvolle Analyse zu werden, obwohl das mit ihnen beschriebene System eigentlich nicht übermäßig groß erscheint. Daher ist es bei ihrer Verwendung oft notwendig, spezielle Modifikationen oder Einschränkungen vorauszusetzen, um eine geeignete Anpassung an die jeweilige Anwendung zu erreichen. Erfolgreich angewendet wurden und werden Petri-Netze bislang beispielsweise in der Leistungsbewertung und auf dem Gebiet der Kommunikationsprotokolle, zunehmend auch für die Modellierung und Analyse von verteilten Software- und Datenbank-Systemen, nebenläufigen und parallelen Programmen sowie einer Fülle weiterer Gebiete.

3.4.2 Grundbegriffe

Machen wir uns zunächst einmal mit den grundsätzlichen Eigenschaften von Petrinetzen und der hierbei verwendeten Terminologie vertraut.

Definition 1: Ein **Netz** sei definiert als ein endlicher Graph $X = (\{S \cup T\}, F)$, bestehend aus

einer endlichen Menge $S = \{s_1, s_2, ..., s_m\}$ von **Stellen**,

einer endlichen Menge $T = \{t_1, t_2, ..., t_n\}$ von **Transitionen** und

einer Menge $F \subseteq (S \times T) \cup (T \times S)$ von **Kanten**.

In der graphischen Darstellung eines Petrinetzes erscheinen die Stellen stets als Kreise, die Transitionen als Rechtecke und die Kanten als Pfeile. Ergänzend zur Definition 1 ist festzuhalten, daß die Mengen S und T als disjunkt vorausgesetzt werden.

Wir betrachten nun eine beliebige Stelle bzw. Transition und definieren ihren "Vor-" und "Nachbereich" wie folgt:

Definition 2: Für ein beliebiges $x \in S \cup T$ sei

$$\bullet x := \{y | (y, x) \in F\}$$

$$x\bullet := \{y | (x, y) \in F\}$$

$$^{-}x := \{(y, x) | (y, x) \in F\}$$

$$x^{-} := \{(x, y) | (x, y) \in F\}$$

Die Menge $\bullet x$ nennt man **Vorbereich** einer Stelle $x \in S$ bzw. einer Transition $x \in T$. Analog heißt $x\bullet$ **Nachbereich** einer Stelle bzw. Transition x.
Für eine Transition $t \in T$ heißen die Elemente der Menge $\bullet t$ **Eingangsstellen** und die Elemente der Menge $t\bullet$ **Ausgangsstellen** der Transition. Weiterhin bezeichnet man

die Kanten $(s,t) \in {}^{-}t$ als **Eingangskanten** und die Kanten $(t,s) \in t^{-}$ als **Ausgangskanten** der Transition t.

Durch Angabe der Elemente von S, T und F ist ein Netz vollständig beschrieben. Zur Veranschaulichung ein Beispiel:

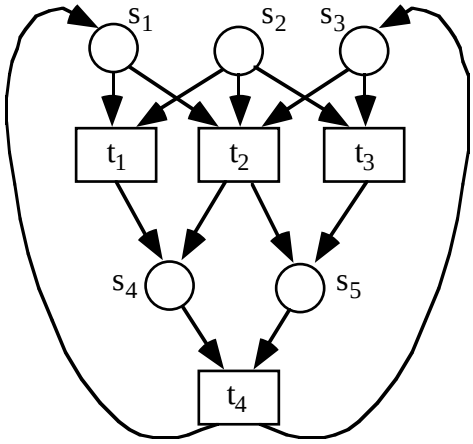


Abb. 3.11: Petrinetz

Stellen lassen sich in einem Petrinetz je nach Anwendung meist als Zustände, bereitgestellte Daten, Signale, Bedingungen oder Speicher für Objekte interpretieren, während Transitionen üblicherweise für Ereignisse, Vorgänge, Berechnungsschritte, Jobs oder auch Prozessoren stehen.

Ein sehr gebräuchliches Petrinetz-Konzept sind nun die Stellen-/Transitions-Systeme gemäß

Definition 3: Ein **Stellen-/Transitions-System (S/T-System)** ist ein 6-Tupel $Y = (S,T,F,K,W,M_0)$, bestehend aus einem Netz (S,T,F) sowie den Abbildungen

$K : S \to \mathbf{N} \cup \{\infty\}$,
$W : F \to \mathbf{N}$ und
$M : S \to \mathbf{N}$.

Durch K wird jeder Stelle eine **Kapazität** zugeordnet (im einfachsten Fall beträgt diese ∞, was keine Einschränkung zur Folge hat). Die Funktion M ordnet jeder Stelle eine Anzahl sogenannter Marken (token) zu und wird daher als **Markierung** von Y bezeichnet. Dabei kann eine Stelle nicht mehr Marken aufnehmen als ihre Kapazität erlaubt, das heißt es gilt $M(s) \le K(s) \quad \forall\, s \in S$.
M_0 stellt die Markierung dar, wie sie zu Beginn des durch das Petrinetz modellierten Prozesses vorliegt, und heißt daher **Anfangsmarkierung.**
Die Abbildung W schließlich ordnet jeder Kante ein **Kantengewicht** zu, welches im einfachsten Fall 1 beträgt. Kantengewichte von 1 werden der Übersichtlichkeit halber in der Regel nicht explizit angegeben, umgekehrt haben Kanten, für die kein Gewicht angegeben ist, automatisch Gewicht 1.

Die Marken werden graphisch durch schwarze Punkte innerhalb der kreisförmigen Stellen symbolisiert. Mit ihrer Hilfe kann der Gesamtzustand eines modellierten Sy-

stems dargestellt werden. Wenn die Stellen beispielsweise einen Speicher für ein spezielles Objekt darstellen, kann die Anzahl der Marken die Anzahl der vorhandenen Objekte repräsentieren. Die Kapazität einer Stelle entspricht dann der Speicherkapazität für die betreffenden Objekte.

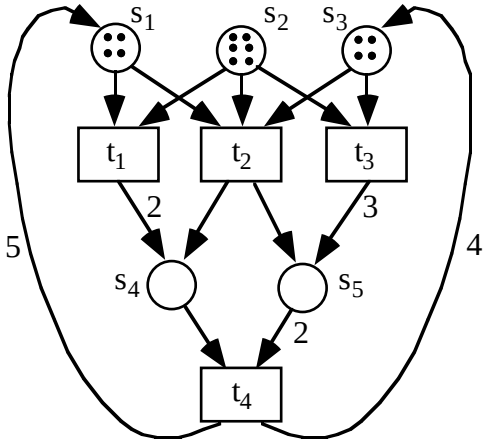


Abb. 3.12: Petrinetz mit Token und Kantengewichten

In unserem Beispiel ordnen wir jeder Stelle der Einfachheit halber die Kapazität ∞ zu, verwenden aber unterschiedliche Kantengewichte. Die gegebene Anfangsmarkierung läßt sich durch $M_0 = \{(s_1,4),(s_2,6),(s_3,4),(s_4,0),(s_5,0)\}$ beschreiben.

Nachdem wir im ersten Schritt zunächst die Topologie des zugrundeliegenden Netzes beschrieben und als zweites dann die zur Darstellung eines Systemzustandes erforderlichen Begriffe eingeführt haben, ermöglichen wir nun das Modellieren der Dynamik eines Systems durch die Definition einer "**Schaltregel**":

Definition 4: Man kann jeder Markierung M eine Menge $T_{akt}(M)$ zuordnen mit

$$T_{akt}(M) = \{\ t \in T \mid (M(s) \geq W(s,t)\ \forall\ s \in \bullet t) \wedge (M(s) \leq K(s) + W(t,s)\ \forall s \in t \bullet)\ \}$$

Die in $T_{akt}(M)$ enthaltenen Transitionen heißen **aktiviert** unter der Markierung M.. Aktivierte Transitionen t können **schalten** (oder "**feuern**"). Aus der Markierung M ergibt sich durch Feuern von t die **unmittelbare Folgemarkierung** M' wie folgt:

$$M'(s) = M(s) - W(s,t) \quad \text{falls } s \in {\bullet t} \setminus t^{\bullet}$$
$$M'(s) = M(s) + W(t,s) \quad \text{falls } s \in t^{\bullet} \setminus {\bullet t}$$
$$M'(s) = M(s) - W(s,t) + W(t,s) \quad \text{falls } s \in t^{\bullet} \cap {\bullet t}$$
$$M'(s) = M(s) \quad \text{sonst}$$

Eine Transition ist also genau dann aktiviert, wenn in allen Eingangsstellen der Transition die erforderliche Anzahl von Marken vorhanden ist und die Kapazität keiner der Ausgangsstellen durch das Feuern überschritten wird. Dabei wird die Anzahl der für die Aktivierung erforderlichen Marken durch die jeweiligen Gewichte der Eingangskanten beschrieben. Beim Feuern wird dann die durch die jeweiligen Eingangskantengewichte festgelegte Anzahl von Marken aus den Eingangsstellen der Transition entfernt und dafür die durch die Ausgangskantengewichte beschriebene Markenzahl den Ausgangsstellen hinzugefügt. Für diesen Vorgang schreibt man auch $M\ [t > M'$ und meint damit, daß t unter der Markierung M zur Markierung M' schaltet.

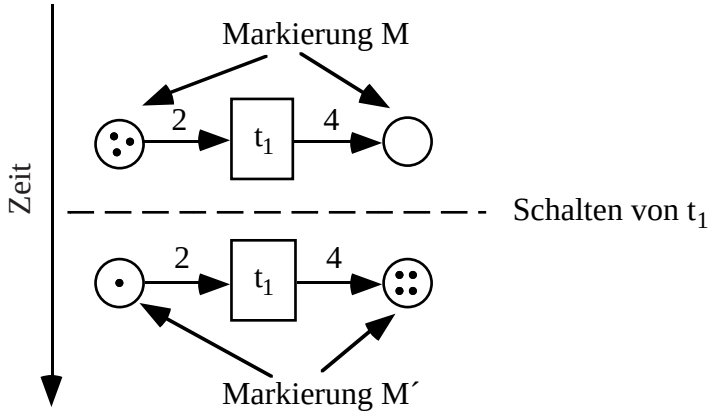


Abb. 3.13: Übergang von einer Markierung M zu einer Markierung M´

Zur Analyse eines Petrinetzes Y gibt es zwei wichtige Hilfsmittel: die **Erreichbarkeitsmenge** E_Y und den **Erreichbarkeitsgraphen** G_Y. Es sei dazu $T = \{t_1, t_2, ...\}$ die Menge der einfachen Transitionen und $T^* = \bigcup_{n=0}^{\infty} T^n$. Eine Markierung M´ ist von M aus **erreichbar,** wenn gilt:

$$\exists\ t_1 t_2 t_3 .... t_n \in T^*, n \in \mathbf{N}_0 : M[t_1>M_1[t_2>M_2 ..... [t_n>M'$$

Man schreibt dann auch $M[\omega>M'$ mit $\omega \in T^*$. M´ wird auch als **mittelbare Folgemarkierung** von M bezeichnet.
Die **Erreichbarkeitsmenge** E_Y ist definiert durch:

$$E_Y = E_Y(M_0) := \{M' \mid \exists\ \omega \in T^* : M_0[\omega > M'\}$$

Die **Erreichbarkeitsmenge** enthält also die Anfangsmarkierung M_0 (da auch die leere Folge $\in T^*$) und alle Markierungen des Netzes, die von der Anfangsmarkierung M_0 aus erreicht werden können.

Der **Erreichbarkeitsgraph** verbindet über eine Kante alle Markierungen M, M´ $\in E_Y$, für die gilt: $\exists\ t_i \in T : M[t_i>M'$ (d. h. M´ ist unmittelbare Folgemarkierung von M). Als Kantenbeschriftung wird die Transition angegeben, deren Schalten die entsprechende Änderung der Markierung des Netzes verursacht. Anhand des Erreichbarkeitsgraphen können nun verschiedene Eigenschaften des Petrinetzes - und damit

des modellierten Systems - ermittelt werden. Wichtige Begriffe in diesem Zusammenhang sollen hier nur informell definiert werden. Ist eine Markierung $M \in E_Y$ nicht von jeder anderen Markierung aus erreichbar (über das Schalten mindestens einer beliebigen Folge von Transitionen), spricht man von einer **teilweisen Verklemmung**. Wenn es sogar für eine Markierung $M \in E_Y$ gar keine Nachfolgemarkierung M´ mit M [t>M´ gibt, dann spricht man von einer Verklemmung oder einem **Deadlock**. Ist eine Transition unter keiner Markierung $M \in E_Y$ aktiviert, so spricht man von einer **toten Transition**. Die Transition $t_1 \in T$ steht im **Konflikt** mit $t_2 \in T$ unter einer Markierung M, wenn $\{t_1, t_2\} \subseteq T_{akt}(M)$, aber $t_1 \notin T_{akt}(M')$ mit $M[t_2>M'$.

3.4.3 Erzeuger-Verbraucher-Problem

Die Anschaulichkeit, die man für eigentlich komplexe Abläufe durch die Modellierung als Petrinetz erreichen kann, soll am Beispiel des Erzeuger-Verbraucher-Problems demonstriert werden. Dieses Problem als Petrinetz modelliert, sieht folgendermaßen aus:

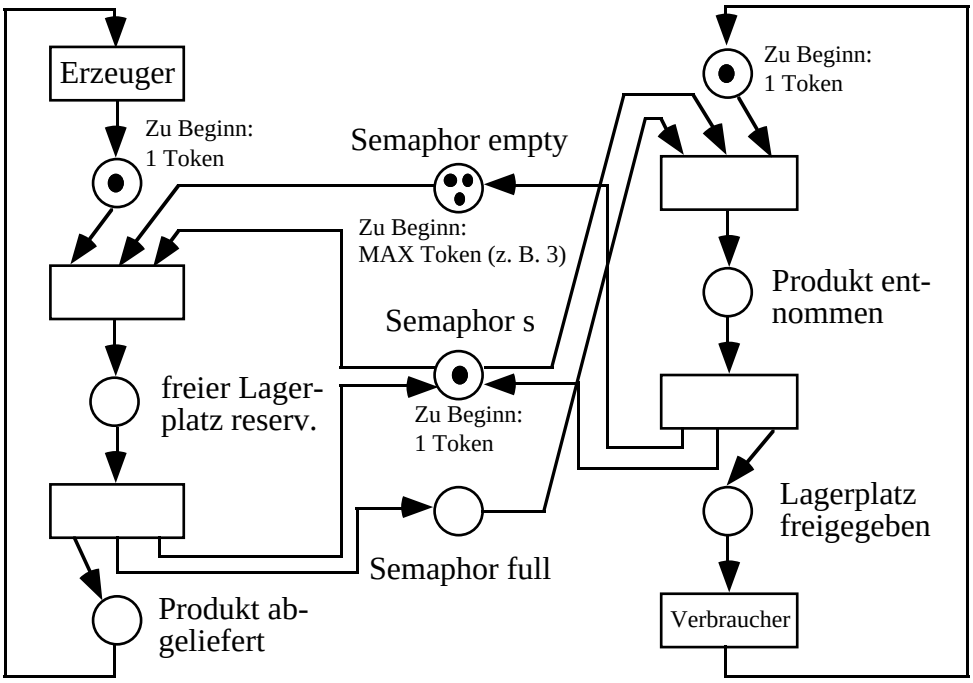


Abb. 3.14: Das Erzeuger-Verbraucher-Problem als Petrinetz

Zu Beginn liegen ein Token in `s`, MAX Token in `empty` und kein Token in `full`. Was dann passiert, mag der geneigte Leser am besten selbst (unter Zuhilfenahme diverser Pfennigstücke, Reißnägel o. ä.) ausprobieren. Sehr schnell und sehr schön wird dabei deutlich werden, daß die Dynamik dieses Petrinetzes genau der Lösung des Erzeuger-Verbraucher-Problems entspricht, wie sie unter 3.3.2 algorithmisch dargestellt wurde.

3.5 Bedingte kritische Regionen

Ein signifikanter Nachteil der Semaphoren besteht darin, daß sie bei falscher Verwendung immer noch fehleranfällig sind. Insbesondere kann ein versehentliches Vertauschen von `signal`- und `wait`-Befehlen schnell fatale Folgen zeitigen. Um solche Fehler auszuschließen, wurden eine Reihe von High-Level-Konstrukten eingeführt, von denen wir als erstes das Konzept der **Kritischen Regionen** betrachten werden. Dabei müssen wir voraussetzen, daß sich ein Prozeß zunächst einmal aus einer Reihe lokaler Daten und einem sequentiellen Programm zusammensetzt, das auf diesen Daten operiert. Der Zugriff auf diese lokalen Daten ist dabei wohlgemerkt nur dem zugehörigen sequentiellen Programm möglich, darüberhinaus aber greifen mehrere Prozesse noch auf gemeinsame globale Daten zu.

Beim Konzept der kritischen Regionen sind solche globalen Variablen `v` vom Typ `T` folgendermaßen zu deklarieren:

```
var v: shared  T;
```

Damit wird angedeutet, daß `v` von mehreren Prozessen manipuliert werden kann. Ein Prozeß kann allerdings auf die Variable `v` nur innerhalb eines Statements `region` zugreifen, das folgendermaßen aussieht:

```
region v do S;
```

In dieser Form ist das Konstrukt allerdings noch wenig hilfreich für Synchronisationen. Deshalb verwendet man meistens **bedingte kritische Regionen**:

```
region v when B do S;
```

Was bedeutet dies? Zunächst einmal wird durch `region v` sichergestellt, daß kein anderer Prozeß auf die Variable `v` zugreifen kann, solange der ursprüngliche Prozeß mit der Abarbeitung von `S` beschäftigt ist. Der Ausdruck `B` ist eine boolesche Bedingung, die den Zutritt zum kritischen Bereich einschränkt. Versucht nämlich ein Prozeß, seinen kritischen Bereich zu betreten (und darin die globalen Variablen `v` zu manipulieren), so wird zunächst seine Bedingung `B` ausgewertet. Lautet das Ergebnis dieser Auswertung `TRUE`, so darf der betreffende Prozeß in seinen kritischen Bereich, und `S` wird ausgeführt. Ist `B` aber `FALSE`, so läßt der Prozeß seinen kritischen Bereich in Ruhe (und geht beispielsweise "schlafen"). Später wird überprüft, ob einer der (von schlafendem Prozeß zu Prozeß u. U. verschiedenen) Ausdrücke `B` inzwischen `TRUE` geworden ist. Ist dies der Fall, so wird einer der entsprechenden schlafenden Prozesse geweckt und betritt seinen kritischen Bereich.
Es sei nochmals darauf hingewiesen, daß `region` die atomare Ausführung des gesamten Bereiches S sicherstellt. Dabei kann an der Stelle einer einzelnen Variablen v natürlich auch eine ganze Liste von Variablen stehen, die dann eben allesamt dem Zugriff anderer Prozesse entzogen bleiben, solange `S` nicht ganz ausgeführt ist.

Betrachten wir als Beispiel das Erzeuger-Verbraucher-Problem mit einem ringförmigen Zwischenlager. Dieser Puffer werde realisiert durch ein Feld `pool[0..n]`. Darüberhinaus verwenden wir noch eine Zählvariable `counter`, die uns den Lagerstand angibt (`0 ≤ counter ≤ MAX`). Außerdem müssen wir noch eine Variable `buffer` definieren, in der die globalen Variablen eingekapselt sind:

```
var buffer: shared record
                    pool: array[0..n-1] of items;
```

```
    counter, in, out: integer;
end;
```

Dann können wir die Algorithmen folgendermaßen angeben:

ERZEUGER:

```
region buffer when counter < MAX do
  begin
  pool[in] := nextp;
  in := (in + 1) mod MAX;
  counter := counter + 1;
  end;
```

VERBRAUCHER:

```
region buffer when counter > 0 do
  begin
  nextc := pool[out];
  out := (out + 1) mod MAX;
  counter := counter - 1;
  end;
```

Zum Schluß dieses Abschnitts wollen wir uns noch überlegen, wie man das Konstrukt einer bedingten kritischen Region implementieren kann. Eine mögliche Lösung greift auf Semaphore mit assoziierten Warteschlangen zurück, in welchen Prozesse unterkommen, die auf das Eintreffen von "`B = TRUE`" warten müssen. Dabei ist dieses Warten seinerseits noch zweistufig zu behandeln, so daß insgesamt mit jeder globalen Variablen folgende Variable assoziiert sind:

```
var  mutex, firstdelay, seconddelay: semaphore;
     firstcount, secondcount: integer;
```

Die Idee dieser Implementierung ist folgende: Unter den Prozessen, die in ihren kritischen Bereich wollen, kursiert ein (durch `mutex` geschütztes) Ticket, dessen Besitz einem Prozeß erlaubt, exklusiv diverse Aktivitäten (wie sie gleich näher beschrieben werden) auszuüben. Es ist also unter den eintrittswilligen Prozessen immer nur einer aktiv, alle anderen warten in einer der drei Warteschlangen `mutex` (außerhalb), `firstdelay` oder `seconddelay` (innerhalb). Bekommt ein Prozeß zum erstenmal das Ticket, so überprüft er seine Bedingung `B`. Ist sie erfüllt, darf er in seinen kritischen Bereich, ist sie `FALSE`, so reiht er sich in die erste Schlange (`firstdelay`) ein. Das Ticket wird unter gewissen Voraussetzungen (s. u.) dem nächsten Neuankömmling zur Verfügung gestellt. Auf diese Weise können sich eine Anzahl von Prozessen mit unerfüllten Bedingungen in der ersten Schlange versammeln. Jedesmal wenn nun ein Prozeß erfolgreich war und seinen kritischen Bereich wieder verläßt, passiert folgendes: Zunächst werden alle in der ersten Schlange wartenden Prozesse am Ende der zweiten Schlange (`seconddelay`) eingereiht, so daß die erste Schlange nun leer ist. Dann überprüfen der Reihe nach die ältesten der Prozesse in der zweiten Schlange von neuem ihre Bedingungen. Sind diese immer noch `FALSE`, so reihen sich die Prozesse wieder in der ersten Schlange ein. Sobald sich bei dieser Überprüfung ein Prozeß mit `B = TRUE` findet, darf dieser nun in seinen kritischen Bereich, arbeitet sein `S` ab, verläßt seinen kritischen Bereich und veranlaßt wieder, daß alle Prozesse in der ersten Schlange sich zur zweiten begeben und daraufhin alle Prozesse der zweiten Schlange wieder ihre Bedingungen überprüfen usw. Findet sich bei einer

solchen Überprüfung kein Prozeß mit Bedingung `TRUE`, so wird das Ticket dem nächsten Neuankömmling zur Verfügung gestellt.

Die Umsetzung dieser Idee in einen Algorithmus sieht folgendermaßen aus:

```
wait(mutex);
while not B do
  begin
  firstcount := firstcount + 1;
  if secondcount > 0
    then signal(seconddelay)
    else signal(mutex);
  wait(firstdelay);
  firstcount := firstcount - 1;
  secondcount := secondcount + 1;
  if firstcount > 0
    then signal(firstdelay)
    else signal(seconddelay);
  wait(seconddelay);
  secondcount := secondcount - 1;
  end;

Kritischer Bereich;

if firstcount > 0
  then signal(firstdelay)
  else if secondcount > 0
    then signal(seconddelay)
    else signal(mutex);
```

Dabei warten Neuankömmlinge in der mit `mutex` assoziierten Schlange, bis sie erstmals aufgerufen werden. Die "erste Schlange" entspricht dem Semaphor `firstdelay`, analog die zweite Schlange dem Semaphor `seconddelay`. Die Anzahl der in diesen beiden Schlangen wartenden Prozesse wird von `firstcount` bzw. `secondcount` aufgezeichnet.

Zum Abschluß betrachten wir ein Beispiel zu diesem Algorithmus. Nehmen wir an, fünf Prozesse 1, 2, ..., 5 scheitern jeweils an ihrem `B`.

firstdelay	1	2	3	4	5
seconddelay:					

Dann verläßt ein weiterer Prozeß seinen kritischen Bereich. Auf sein Signal hin wandern alle Prozesse in die zweite Schlange:

firstdelay:					
seconddelay:	1	2	3	4	5

und überprüfen dann der Reihe nach ihre Bedingungen. Prozeß 3 findet als erster `B = TRUE`, betritt seinen kritischen Bereich

firstdelay:	1	2
seconddelay:	4	5

verläßt ihn und signalisiert. Daraufhin wandern wieder alle in der ersten Schlange wartenden Prozesse in die zweite

firstdelay:				
seconddelay:	4	5	1	2

und das Spiel beginnt von vorne.

3.6 Monitore

3.6.1 Zum Konzept eines Monitors

Ein weiteres High-Level-Konstrukt zur Koordination bzw. Synchronisation von Prozessen begegnet uns in Gestalt des Monitorkonzepts. Ein **Monitor** ist dabei ein Objekt, das sich im wesentlichen aus einer Menge von Prozeduren und Daten zusammensetzt. Entscheidend ist, daß zu gegebener Zeit der Monitor (bzw. eine der in ihm enthaltenen Prozeduren) stets nur von höchstens einem Prozeß genutzt werden darf. Ruft also ein Prozeß eine Monitorprozedur auf und erhält die Erlaubnis, sie abzuarbeiten, so läuft die Prozedur atomar ab. Man könnte einen Monitor beispielsweise mit einem Formel-I-Rennwagen vergleichen. Auch hier handelt es sich um ein High-Level-Konstrukt, das seinen Benutzerprozessen (i. e. den Herren Schumacher, Berger, Hill usw.) eine größere Anzahl u. U. ausgefeilter Funktionalitäten (vorwärtsfahren, funken, Krach machen etc.) zur Verfügung stellt, aber stets nur von höchstens einem Prozeß benutzt werden kann.

Die Syntax eines Monitors hat i. d. R. folgende Gestalt:

```
monitor name
  begin
  Variablendeklarationen;
  procedure P1;
  procedure P2;
  ...
  Vorbesetzung der Variablen;
  end;
```

Da sich immer nur höchstens ein Prozeß innerhalb des Monitors aufhalten darf, braucht sich der Programmierer keine Gedanken um die explizite Implementierung von Sperren vor einem kritischen Bereich machen.

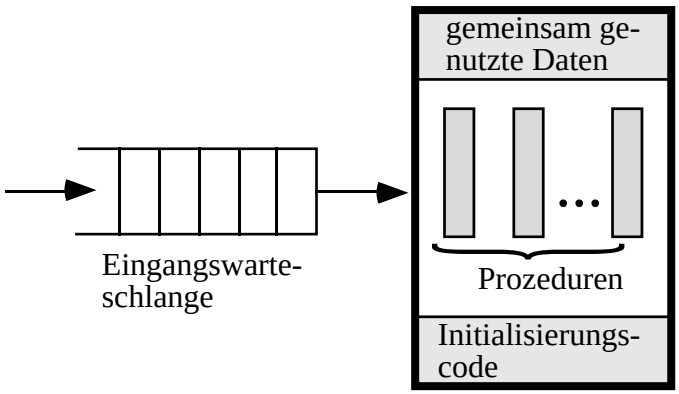


Abb. 3.15: Schematische Sicht eines Monitors

Indes sind Monitore in dieser Gestalt noch nicht zur Synchronisation von Prozessen geeignet. Es fehlt noch (ähnlich wie bei den bedingten kritischen Regionen) eine Möglichkeit, die Abarbeitung der Monitorprozeduren vom Eintreten gewisser Bedingungen abhängig zu machen. Hierzu dient das Konzept der **Bedingungsvariablen** (vom Typ "`condition`"). Diese Variablen werden ähnlich gehandhabt wie Semaphore, insbesondere gibt es wie bei diesen `wait`- und `signal`-Operationen.

Sei also beispielsweise `cond` eine Variable vom Typ `condition`. Der Befehl `wait(cond)` in einer Monitorprozedur (eine andere mögliche Syntax wäre `cond.wait`) veranlaßt einen Prozeß, sich schlafenzulegen (also sich hinten in die zu `cond` assoziierte Warteschlange einzureihen). `signal(cond)` (bzw. `cond.signal`) weckt analog dazu den ältesten Prozeß der zugehörigen Schlange. Im Unterschied zum Semaphorkonzept hat ein Signal jedoch **keine Wirkung**, wenn im Augenblick des "Absendens" kein anderer Prozeß darauf wartet. Man tut in diesem Fall einfach so, als sei kein Signal gegeben worden (der aufmerksame Leser wird deswegen auch bei allen folgenden Beispielen feststellen, daß jedem `wait`-Befehl immer noch eine Abfrage vorgeschaltet wird, ob ein solches "verpufftes" Signal gesendet wurde). Man kann dies auch dahingehend interpretieren, daß `signal(cond)` das Eintreffen einer Bedingung `cond` signalisiert, während `wait(cond)` das Warten von Prozessen aufgrund des Nichterfülltseins von `cond` realisiert.

Eine Synchronisation mittels `wait` und `signal` schafft natürlich gewisse Komplikationen, weil ja stets nur ein einziger Prozeß den Monitor nutzen darf. Was passiert aber, wenn ein solcher Prozeß A während seiner Monitorbenutzung ein "`signal`" von sich gibt, auf das ein weiterer Prozeß B ja wartet, um seinerseits fortschreiten zu können? Offensichtlich muß sich entweder A sofort schlafenlegen oder B darf nicht geweckt werden, denn nur so läßt sich gewährleisten, daß keine zwei Prozesse gleichzeitig aktiv sind.

Auf unser Rennwagenbeispiel läßt sich dies folgendermaßen übertragen: Zum Testen eines neuen Boliden werden zwei Fahrer (A und B) herangezogen, die vertragsgemäß beide zehn Runden zu absolvieren haben. Am Morgen kommt einer der beiden (A) an und fährt los. Kurze Zeit später erscheint auch der andere Fahrer (B), stellt sich an den Straßenrand und wartet. In der siebten Runde nun bekommt Fahrer A plötzlich rasende Kopfschmerzen und signalisiert der Box, daß er keine Lust mehr hat weiterzufahren. Für solche Situationen können zwei verschiedene Vorgehensweisen (= Semantiken) festgelegt worden sein: Entweder unterbricht Fahrer A seine Fahrt, Fahrer B besteigt den Wagen (= Monitor) und fährt seine zehn Runden, steigt

wieder aus, und Fahrer A fährt danach noch vertragsgemäß seine drei Runden zu Ende. Oder aber Fahrer A zeigt sich mannhaft, nimmt eine Aspirin und seine Kopfschmerzen nicht zur Kenntnis und fährt jedenfalls seine zehn Runden zu Ende, bevor er das Auto verläßt und Fahrer B weiterfahren kann.

Ein Ausweg aus diesem Problem besteht darin, den Befehl `signal(cond)` nur als die allerletzte Operation einer Monitorprozedur zu gestatten (da der fraglich Prozeß A daraufhin den Monitor sowieso verläßt, gibt es kein Problem). Dies hätte zur Folge, daß in der Formel 1 Kopfschmerzen per Definition nur am Ende der 10. Runde auftreten können. Weil dies allerdings nicht immer möglich ist (z. B. wenn mehr als ein "`signal`" zu geben ist), bleibt nichts anderes übrig als eine ausdrückliche semantische Festlegung zu treffen. Wenn Prozeß A "`signal(cond)`" gibt und Prozeß B auf `cond` wartet, so muß man sich zwischen folgenden möglichen Semantiken entscheiden: Entweder legt sich A schlafen und wartet solange, bis B den Monitor wieder verlassen hat, oder A fährt mit der Monitorbenutzung fort, wobei B weiterhin wartet. Beide Möglichkeiten haben ihre Vorzüge. Da A sich bereits im Monitor befindet, scheint der zweite Vorschlag vernünftiger zu sein, hat aber den Nachteil, daß zu dem Zeitpunkt, zu dem B schließlich Zutritt zum Monitor erhält, die ursprünglich abgewartete Bedingung möglicherweise bereits wieder veraltet ist.

3.6.2 Exklusive Vergabe eines Betriebsmittels

Im folgenden wollen wir anhand einiger (z. T. bereits besprochener) Beispiele die Nutzung von Monitoren demonstrieren.

Beginnen wir mit dem Monitor EXKL, der den exklusiven Zugriff auf ein gemeinsam genutztes Betriebsmittel regeln soll:

```
monitor EXKL;

begin

var busy: boolean;
var frei: condition;

procedure belegen;
  begin
    if busy then wait(frei);
    busy := TRUE;
  end;

procedure freigeben;
  begin
    busy := FALSE;
    signal(frei);
  end;

busy := FALSE                 (* Vorbesetzung! *)

end;
```

Die Nutzung erfolgt über die Routine

```
EXKL.belegen;
Kritischer Bereich;
EXKL.freigeben;
```

3.6.3 Erzeuger-Verbraucher-Problem

Zweites Beispiel ist ein Monitor namens LAGER zur Lösung des Erzeuger-Verbraucher-Problems. Wir setzen voraus, daß der gemeinsame Puffer die Kapazität `MAX` besitzt. Ferner verwenden wir die "Zählvariablen" `lastpointer` und `count` sowie die "Bedingungsvariablen" `nonempty` und `nonfull`.

```
monitor LAGER;

begin

var buffer: array[0..MAX-1] of items;
var lastpointer, count: integer;
var nonempty, nonfull: condition;

procedure Einfügen(x);
  begin
    if count = MAX then wait(nonfull);
    lastpointer := lastpointer + 1 mod MAX;
    count := count + 1;
    buffer[lastpointer] := x;
    signal(nonempty);
  end;

procedure Entnehmen(x);
  begin
    if count = 0 then wait(nonempty);
    x := buffer[(lastpointer - count + 1) mod MAX];
                                  (* älteste Einheit! *)
    count := count - 1;
    signal(nonfull);
  end;

count := 0;
lastpointer := 0;             (* Vorbesetzung *)

end;
```

Die Routinen von Erzeuger und Verbraucher werden damit sehr übersichtlich:

ERZEUGER:

```
repeat
  erzeuge x;
  LAGER.Einfügen(x);
until FALSE;
```

VERBRAUCHER:

```
repeat
  LAGER.Entnehmen(x);
  verbrauche x;
until FALSE;
```

3.6.4 Drittes Reader-Writer-Problem

Als letztes Beispiel für einen Monitor betrachten wir die "faire" dritte Variante des Reader-Writer-Problems: Ein ankommender Reader erhält Priorität vor später ankommenden Writern. Verläßt ein Writer den kritischen Bereich, so erhalten evtl. wartende Reader Zutritt, bevor der nächste Writer Einlaß erhält.
Der Monitor RW besteht aus vier Prozeduren, nämlich `startread`, `endread`, `startwrite` und `endwrite`. Um die Reader zu zählen, verwenden wir eine Variable `readercount`. `busywrite` zeigt an, ob ein Writer aktiv ist, außerdem tauchen noch die Bedingungsvariablen `okread` und `okwrite` auf. Mittels `wait(okread)` können Reader "schlafengelegt" werden; sie werden dann in eine Warteschlange eingereiht, deren Zustand sich mittels `okread.nonempty` (analog dazu `okwrite.nonempty`) abfragen läßt.

```
monitor RW;

begin

var readercount: integer;
var busywrite: boolean;
var okread, okwrite: condition;

procedure startread;
  begin
    if (busywrite or okwrite.nonempty) then wait(okread);
    readercount := readercount + 1;
    signal(okread);
  end;

procedure endread;
  begin
    readercount := readercount - 1;
    if readercount = 0 then signal(okwrite);
  end;

procedure startwrite;
  begin
    if (busywrite or (readercount > 0)) then wait(okwrite);
    busywrite := TRUE;
  end;

procedure endwrite;
  begin
    busywrite := FALSE;
    if okread.nonempty then signal(okread)
                       else signal(okwrite);
  end;

readercount := 0;
busywrite := FALSE;

end;
```

Auch hier noch die zugehörigen Routinen:

READER:

```
repeat
  RW.startread;
  Lesen;
  RW.endread;

  andere Operationen;

until FALSE;
```

WRITER:

```
repeat
  RW.startwrite
  Schreiben;
  RW.endwrite;

  andere Operationen;

until FALSE;
```

Zur Veranschaulichung zeigen wir hier noch an einem Beispiel, wie bei Verwendung dieser Algorithmen ein typischer Ablauf aussehen kann:

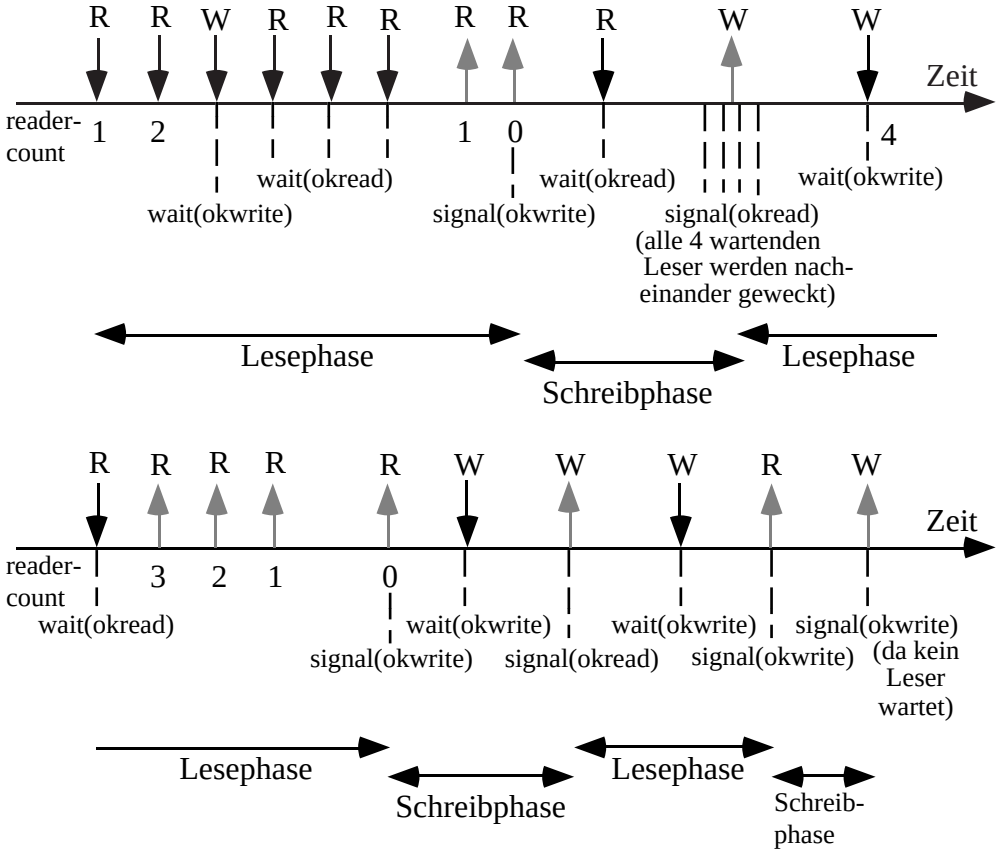


Abb. 3.16: Ablaufbeispiel für das 3. Reader/Writer-Problem

3.7 Datenbankorganisation

3.7.1 Transaktionen

Wir haben jetzt eine Vielzahl von Möglichkeiten kennengelernt, die Atomarität von Zugriffen auf kritische Bereiche sicherzustellen. Als ein Gebiet, auf dem derartige Mechanismen auch wirklich zum Einsatz kommen, greifen wir die Organisation von Datenbanken heraus.

Die Hauptaufgaben einer Datenbank bestehen in der Sicherung, Wiedergewinnung und Konsistenzerhaltung großer Datenbestände. Dabei bedienen sie sich sogenannter **Transaktionen**. Hierunter verstehen wir eine Menge von Operationen, die Lese- und Schreibzugriffe auf Datenbestände durchführen.

Als Beispiel für eine Transaktion soll die Buchung eines Fluges im Reisebüro dienen. Eine solche Buchung setzt sich zusammen aus einer Vielzahl von Zugriffen auf Informationen in einer zentralen Datenbank, beispielsweise aus

- Anfrage Flugnummer
- Anfrage Platzzahl in verschiedenen Sitz- oder Preisklassen
- mögliche Tarifermäßigungen
- Rückfragen
- ....
- Buchung

Ein wichtiges Anliegen muß nun sein, die Atomarität einer solchen (Buchungs-) Transaktion zu gewährleisten. Andernfalls wäre es leicht möglich, daß beispielsweise die Anfragen zweier verschiedener Reisebüros sich ineinander verschachteln, was bei ungünstigen Konstellationen dazu führen kann, daß das eine Reisebüro herausfindet, daß ein bestimmter Flug noch frei ist, mit dem Buchen dann aber solange braucht, daß ein anderes Reisebüro in der Zwischenzeit die letzten freien Plätze diese Fluges belegt, ohne daß das erste Reisebüro dies merkt. Konsistenz der Daten wäre hier nicht gewährleistet.

3.7.2 Einfache Recovery-Methoden

Transaktionen bestehen allgemein aus einer Reihe von Schreib- und Lese-Operationen und werden - wenn die Transaktion korrekt zu Ende ging - durch eine **commit**- bzw. andernfalls (also beim Auftreten eines Fehlers) eine **abort**-Operation abgeschlossen. Tritt ein abort auf, so muß ein sog. "**Rollback**" durchgeführt werden, um einen konsistenten Zustand der Daten wiederherzustellen. Eine derartige Aktion bezeichnet man auch als "**Recovery Operation**".

Gewöhnlich verwendet jedes Rechnersystem verschiedene Speichertypen. Insbesondere ist hier zu unterscheiden zwischen

- flüchtigen Speichern (**volatile storage**): normaler Speicher, z. B. Arbeitsspeicher
- nichtflüchtigen Speichern (**nonvolatile storage**): Hintergrundspeicher, Platte, Band
- stabilen Speichern (**stable storage**): CD-ROM.

Während in flüchtigen Speichern liegende Informationen leicht (z. B. bei Stromausfall) verlorengehen können, ist dies bei nichtflüchtigen Speichern kaum je und bei stabilen Speichern grundsätzlich überhaupt nicht möglich. Dieser gewaltige Vorteil nichtflüchtiger Speicher wird damit bezahlt, daß nichtflüchtige Speicher im Zugriff um ein Vielfaches langsamer sind als flüchtige.

Recovery-Operationen beruhen nun auf dem Zusammenspiel von flüchtigen und nichtflüchtigen Speichern. Eine einfache Methode basiert dabei auf dem Führen eines "**Logbuches**", das aktuelle Daten in einem nichtflüchtigen Speicher enthält. Jeder Eintrag beschreibt dabei eine einzelne Operation innerhalb einer Transaktion und enthält folgende Informationen:

- Transaktionsname

- Namen der Daten, die neu geschrieben wurden
- alter Wert des Datums
- neuer Wert des Datums.

Wenn nun eine Transaktion T_i startet, so wird die Eintragung "T_i starts" vorgenommen. Sodann wird jede write-Operation ins Logbuch eingetragen. Endet die Transaktion korrekt, so lautet der abschließende Eintrag "T_i commits".

Mit Hilfe des Logbuches lassen sich fehlerhafte Transaktionen leicht bereinigen. Hierzu stehen zwei idempotente Prozeduren zur Verfügung:

- `undo`(T_i): durch T_i evtl. überschriebene Daten werden auf ihre alten Werte zurückgesetzt
- `redo`(T_i): alle durch T_i aktualiserte Daten werden auf ihre neuen Werte gesetzt.

Die jeweiligen alten bzw. neuen Werte lassen sich dabei dem Logbuch entnehmen. Die Idempotenz von `undo` und `redo` stellt sicher, daß es auf das Resultat keinen Einfluß hat, ob man ein und dieselbe Prozedur einmal oder mehrmals hintereinander durchführt.

Wenn nun eine Transaktion T_i abgestürzt ist, so kann man mittels `undo`(T_i) leicht den Zustand der Daten vor Beginn von T_i wiederherstellen. Kompliziertere Verhältnisse liegen vor, wenn ein Systemfehler aufgetreten ist. In diesem Fall ist für jede einzelne Transaktion zu entscheiden, ob sie mittels `undo` oder `redo` zu behandeln ist. Dabei verfährt man folgendermaßen:

- Enthält das Logbuch den Eintrag "T_i starts", aber nicht den Eintrag "T_i commits", so wird die Transaktion T_i mittels `undo`(T_i) komplett rückgängig gemacht.
- Enthält das Logbuch dagegen den Eintrag "T_i starts" und den Eintrag "T_i commits", so gewinnen wir einen zuverlässigen Zustand der Daten durch die Operation `redo`(T_i).

Diese Vorgehensweise ist beim Auftreten eines Systemfehlers auf alle bislang durchgeführten Transaktionen anzuwenden. Dies kostet zum einen Zeit, zum anderen aber wird es des öfteren vorkommen, daß durch die Operation `redo` Daten einen Wert erhalten, der im folgenden bereits wieder veraltet ist und durch eine der nächsten Transaktionen überschrieben wird. Dadurch wird zwar die Konsistenz nicht beeinträchtigt, dennoch handelt es sich um unnötigen Overhead.

Um bei Systemfehlern die Anzahl der evtl. zu wiederholenden Operationen zu begrenzen, führt man "**checkpoints**" ein. Dabei werden zu bestimmten Zeitpunkten alle derzeit in flüchtigen Speichern gehaltenen Logbuch-Einträge sowie alle in flüchtigen Speichern gehaltenen Daten auf einen stabilen Speicher übertragen. Von diesem Checkpoint an ist dann die gesamte "Vergangenheit" als korrekt anerkannt. Dies impliziert, daß ein Checkpoint erst dann gesetzt werden kann, wenn alle noch laufenden Transaktionen korrekt zu Ende gebracht worden sind.

Vergleichbare Vorgehensweisen findet man auch im Bereich der Datenkommunikation. Will man dort etwa ein sehr langes File übertragen, wobei Fehler ausgeschlossen sein sollen, so geschieht dies entweder dadurch, daß man das gesamte File auf einmal überträgt (und beim Auftreten eines Fehlers eben von vorne anfängt) oder indem man das File durch Checkpoints in Kapitel aufteilt (und bei einem eventuellen Fehler nur bis zum letzten korrekt verbuchten Checkpoint zurücksetzt).

Konkret sieht dies nun bei Datenbanken folgendermaßen aus: Jeder Checkpoint wird ins Logbuch eingetragen. Tritt ein Systemfehler auf, so wird die erste Transaktion nach dem letzten Checkpoint ausfindig gemacht (genauer gesagt die erste Transaktion, deren Start nach dem letzten Checkpoint ins Logbuch eingetragen wurde). Diese und alle später begonnenen Transaktionen werden dann daraufhin überprüft, ob sie jeweils einen Eintrag "T_i commits" im Logbuch stehen haben (und deshalb mittels `redo`(T_i) zu behandeln sind) oder nicht (in welchem Falle ein `undo`(T_i) durchzuführen ist).

3.7.3 Serialisierbarkeit und potentielle Konflikte

Gegeben seien Transaktionen T_1, T_2, ..., T_n. Diese können ineinander verzahnt ausgeführt werden, man bezeichnet diese Ausführungsreihenfolge als **Schedule**. In einem Schedule können also z. B. zunächst einige Operationen der Transaktion T_1 ausgeführt werden, dann einige, die zu T_3 gehören, dann wieder einige von T_1, danach welche von T_2 usw. Ein Schedule heißt **seriell**, wenn er der Ausführung einer beliebigen Permutation der Transaktionen T_1, T_2, ..., T_n entspricht (d. h. die Transaktionen werden nicht ineinander verzahnt, sondern jede einzelne auf einmal ausgeführt, dafür aber in einer beliebigen Reihenfolge).

Betrachten wir als nächstes ein ganz einfaches Beispiel für ein solches serielles Schedule:

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nicht-serielle Schedules sind nun nicht notwendigerweise inkorrekt, selbst dann nicht, wenn sie Lese- und Schreiboperationen auf gemeinsamen Datenbeständen ausführen. Dies sieht man leicht an folgendem Beispiel 2:

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Bezeichne nun WM(i) die "Write-Menge" von Transaktion T_i (d. h. die Menge der Daten, auf die eine write-Operation im Rahmen von T_i zugreift), und analog dazu RM(i) die "Read-Menge" von T_i. Für zwei Transaktionen T_i und T_j liegt ein potentieller Konfliktfall dann vor, wenn gilt:

$$(WM(i) \cap WM(j)) \cup (WM(i) \cap RM(j)) \cup (RM(i) \cap WM(j)) \neq \emptyset$$

Ist nämlich $WM(i) \cap WM(j) \neq \emptyset$, so kommt es darauf an, welche der beiden Transaktionen zuletzt schrieb; die write-Operation der anderen Transaktion geht verloren:

T_i	T_j
write(C)	
(geht verloren!)	write(C)

Ist $WM(i) \cap RM(j) \neq \emptyset$ bzw. $RM(i) \cap WM(j) \neq \emptyset$, so können in einer Transaktion unmittelbar hintereinander ausgeführte Leseoperationen unterschiedliche Ergebnisse liefern:

T_i	T_j	
	read(D)	(alter Wert)
write(D)		
	read(D)	(neuer Wert)

In diesem Fall ist nicht klar, welcher Wert von D der "richtige" ist.

Betrachten wir unser Beispiel 2, so stellen wir fest, daß auch hier beispielsweise $RM(T_0) \cap WM(T_1) \neq \emptyset$ gilt. Dies zeigt, daß in Einzelfällen die Konsistenz erhalten bleiben kann, auch wenn streng genommen ein Konfliktrisiko besteht.

Manchmal kann man durch **Swapping** (d. h. sukzessives Vertauschen) von zeitlich benachbarten und nicht konfligierenden Daten (hier: read(B), write(B) von T_0 mit read(A), write(A) von T_1) das in Konflikt stehende Schedule in ein serielles umgewandelt werden. Derartige Schedules heißen **konfliktserialisierbar**.

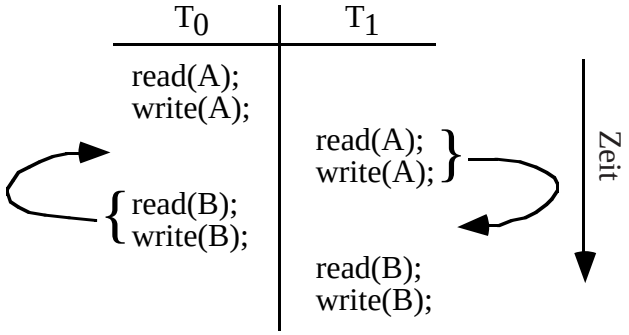


Abb. 3.17: Konfliktserialisierbarer Schedule (die angedeuteten Pfeile erfordern natürlich mehrere einzelne Swap-Operationen

Die Atomarität der einzelnen Transaktionen läßt sich dabei mittels einer wait/signal-Konstruktion leicht gewährleisten. Für allgemeine Zwecke ist dies aber oft zu umständlich. Daher lernen wir im folgenden zwei effizientere Protokolle zur Gewährleistung der Serialisierbarkeit kennen.

3.7.4 Zwei-Phasen-Sperrprotokoll

Sperrprotokolle beruhen darauf, daß vor dem Zugriff auf Daten, die auch von anderen genutzt werden können, atomare Sperren gesetzt werden (lock), die nach dem Zugriff wieder freigegeben werden (unlock). Das Design solcher Protokolle orientiert sich vor allem an der Granularität der Sperren, also an der Frage, ob man eine geringe Anzahl von Sperren, die dann eben größere "Gebiete" innerhalb der Datenbank abschließen und daher evtl. Zugriffsbegrenzungen zur Folge haben, oder eine hohe Anzahl von Sperren samt dem zugehörigen Aufwand für deren Verwaltung vorzieht.

Ein häufig verwendetes Sperrprotokoll soll hier näher betrachtet werden, und zwar das sog. "**Zwei-Phasen-Sperrprotokoll**" (Two-Phase-Locking). Der Name rührt daher, daß sich das Vorgehen dieses Protokolls in eine "up"- und eine "down"-Phase unterteilen läßt. Grundsätzlich gilt dabei, daß eine Transaktion erst dann Sperren anfordert, wenn sie wirklich benötigt werden. Die erste Phase ("up") ist dabei charakterisiert durch eine zunehmende Anzahl von Sperren. Neue Sperren können nur in der ersten Phase angefordert werden. In der zweiten (oder "down"-) Phase werden dann alle aufgebauten Sperren (mehr oder weniger) gleichzeitig wieder freigegeben.

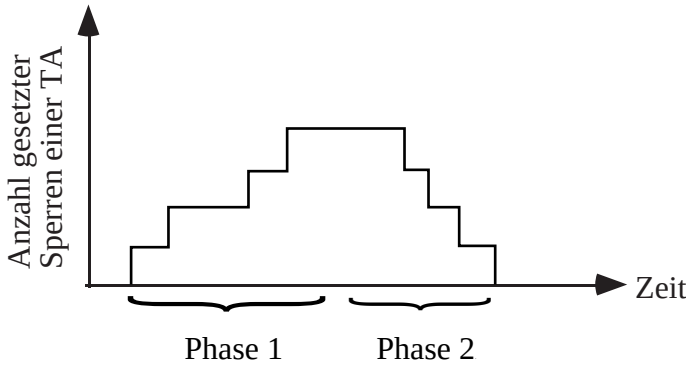


Abb. 3.18: Prinzip des Zwei-Phasen-Sperrens

Hierbei ist also ausgeschlossen, daß beispielsweise zunächst Sperren angefordert werden, dann einige davon wieder freigegeben werden und daraufhin neue Sperren aufgebaut werden usw.:

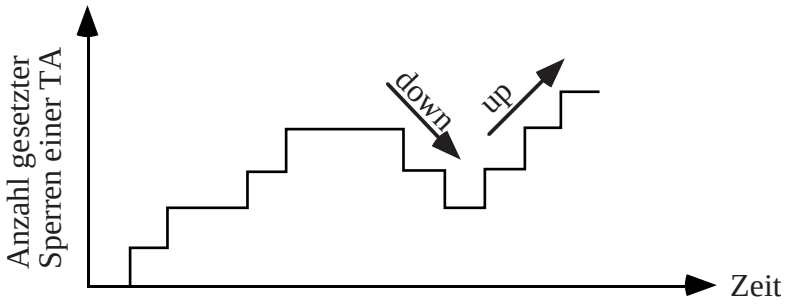


Abb. 3.19: Unzulässige Abfolge von up- und down-Phasen

Warum ist es sinnvoll, ein Vorgehen wie in der letzten Skizze auszuschließen? Der Grund ist einfach: In der "down"-Phase werden Datenbankeinträge verändert und danach freigegeben. Es ist aber möglich, daß eine Transaktion nicht korrekt beendet wird, weil sie etwa (z. B. aufgrund eines Deadlocks) zusätzlich benötigte Sperren nicht bekommt. Dann wären einige Objekte geändert, andere aber nicht, und die Konsistenz der Daten wäre nicht gewährleistet.

3.7.5 Timestamp-Mechanismen

Im eben beschriebenen Sperrprotokoll wird die Reihenfolge zweier konfligierender Transaktionen durch die Zuteilung der ersten Sperre bestimmt, die beide Transaktionen gemeinsam anfordern. Daneben ist jedoch auch denkbar, eine zeitliche Anordnung vorzunehmen. Dabei wäre jede Transaktion zu Beginn mit einem "Zeitwert" (**Timestamp**) zu versehen. Daraufhin hätte dann stets die "älteste" Transaktion (d. h. die Transaktion mit dem kleinsten Timestamp) Vorrang. Sollten zwei Transaktionen zufällig einen identischen Zeitwert aufweisen, so läßt sich eine eindeutige Entscheidung z. B. über die jeweiligen Prozeßnummern treffen.
Die Zeitwerte selbst kann man entweder über eine zentrale oder über lokale Uhren zuteilen. Ungenauigkeiten der lokalen Uhren führen höchstens zu einer gewissen Benachteiligung von Transaktionen mit nachgehenden Uhren.

Beim Start einer Transaktion T_i wird ihr ein eindeutiger Zeitwert $TS(T_i)$ zugewiesen. Serialisierbarkeit ist dann gleichbedeutend damit, daß T_i vor T_j ausgeführt wird, sofern $TS(T_i) < TS(T_j)$ ist. Dieses Vorgehen kann noch auf Read- bzw. Write-Operationen beschränkt werden, die T_i und T_j betreffen.

Für die Implementierung eines derartigen Schemas werden jedem Datum Q zwei Zeitwerte zugeschrieben:
- W-timestamp(Q) = größter Zeitwert einer Transaktion, die eine erfolgreiche write-Operation auf Q ausgeführt hat,
- R-timestamp(Q) = größter Zeitwert einer Transaktion, die Q erfolgreich gelesen hat.

Diese beiden Timestamps werden jedesmal aktualisiert, wenn eine neue `read`(Q)- oder `write`(Q)-Operation durchgeführt worden ist.
Das Timestamp-Protokoll stellt sicher, daß konfligierende Lese- und Schreiboperationen in der Reihenfolge der entsprechenden Timestamps ausgeführt werden. Es operiert dabei folgendermaßen:

Sei zunächst T_i eine Transaktion, welche `read`(Q) ausführen will:

a) Falls $TS(T_i) < W\text{-timestamp}(Q)$ gilt, so heißt dies, daß eine "spätere" Transaktion bereits erfolgreich geschrieben hat. Q ist also bereits geändert, und es bleibt nichts anderes übrig, als `read`(Q) zu verwerfen und die Transaktion T_i neu zu starten (mit entsprechend höherem neuen Timestamp).
b) Ist $TS(T_i) \geq W\text{-timestamp}(Q)$, dann wird die `read`(Q)-Operation durchgeführt und anschließend der R-timestamp(Q) auf das Maximum des bisherigen R-timestamps(Q) und $TS(T_i)$ gesetzt.

Betrachten wir nun eine Transaktion T_i, die die Operation `write`(Q) ausführen will:
a) Ist $TS(T_i) < R\text{-timestamp}(Q)$, so wurde Q bereits früher gelesen. Würde T_i seine `write`-Operation jetzt durchführen, so hätte eine andere Transaktion einen "alten" Wert gelesen. Um dies zu verhindern, muß `write`(Q) verworfen werden. Die Transaktion T_i wird zurückgesetzt und neugestartet.
b) Im Fall $TS(T_i) < W\text{-timestamp}(Q)$ würde ein Schreibvorgang ausgeführt, der bereits zum Zeitpunkt seiner Ausführung veraltet wäre. Sinnvollerweise verwirft man hier `write`(Q) und setzt die Transaktion T_i zurück.
c) Im verbleibenden Fall $TS(T_i) \geq \max(R\text{-timestamp}(Q), W\text{-timestamp}(Q))$ gibt es keine Probleme, daher wird die `write`-Operation ausgeführt.

Es sei darauf hingewiesen, daß jede zurückgesetzte Transaktion einen neuen (höheren) Timestamp erhält und mit diesem ausgerüstet nochmals gestartet wird. Das so durchgeführte Timestamp-Protokoll gewährleistet die Serialisierbarkeit konfligierender Transaktionen. Darüberhinaus bleibt festzuhalten, daß es (im Gegensatz zum Two-Phase-Locking-Protokoll) sogar Deadlockfreiheit garantiert.

3.7.6 Leistungsanalyse eines Sperrprotokolls

Zur Abrundung dieses Abschnitts betrachten wir nun noch ein einfaches Modell eines Sperrprotokolls und fragen uns, was ein solches Protokoll "kann". Die Antwort darauf wird in einer exemplarischen Leistungsanalyse bestehen, die auf gewisse mathematische Hilfsmittel zurückgreifen wird.

Betrachten wir also das sogenannte "Simple 2 PL" (Simple-Two-Phase-Locking-Protocol), also ein Zweiphasensperrprotokoll in seiner einfachsten Variante. Hierbei sperrt eine Transaktion bei Bedarf jeweils ein "Granule", also einen Bereich bestimmter Größe. Bleibt die Anforderung einer solchen Sperre erfolglos, weil der betreffende Granule bereits durch eine andere Transaktion gesperrt ist, so setzen wir die Transaktion sofort zurück und geben alle von ihr gesetzten Sperren frei. Dies ist natürlich keine besonders schlaue Strategie (z. B. wäre es möglicherweise viel effizienter, mit dem Konfliktpartner eine "Einigung" über das Granule herbeizuführen), aber sie wird uns die Modellierung leichter machen.

Kann es hierbei zu einem Deadlock kommen? Eine Deadlocksituation könnte nur auftreten, wenn eine Transaktion T_1 eine von T_2 gehaltene Sperre anfordert, T_2 ihrerseits eine von T_1 gehaltene Sperre benötigt, und keine der beiden Transaktionen bereit ist, gehaltene Sperren zurückzugeben. Eine solche Situation ist beim Simple 2 PL offensichtlich ausgeschlossen. Es handelt sich also um ein deadlockvermeidendes, aber sehr pessimistisch konzipiertes Protokoll.

Modellbildung

Um dieses Modell mit mathematischen Methoden analysieren zu können, sind zunächst einige vereinfachende Annahmen zu treffen. Insbesondere müssen wir modellieren, wie oft Transaktionen auf unsere Datenbank zugreifen und wie deren Sperrverhalten ist (d. h. wieviele und welche Sperren für welche Zeiträume angefordert werden).

Wie kommen Anforderungen von Sperren an?

Betrachten wir eine Datenbank, die aus M gleichartigen Granules besteht. Die Ankünfte der einzelnen Transaktionen sollen möglichst zufällig sein, da wir annehmen können, daß es sich um eine sehr große Datenbank handelt, die von vielen "Kunden" benutzt wird, wobei letztere sich nicht gegenseitig abstimmen.

Mathematisches Standardmodell für die beschriebene Art und Weise, in der Transaktionen ankommen, ist der **Poisson-Prozeß** (vgl. Ross, S. 1- 29). Auf ihn soll daher zunächst eingegangen werden.

Poisson-Prozesse

Stochastische Prozesse

Ein **stochastischer Prozeß** $\{X(t), t \in T\}$ ist eine Familie von Zufallsvariablen, d. h. für jedes t aus der Indexmenge T stellt X(t) eine Zufallsvariable dar. t läßt sich dabei meist als Zeit interpretieren, und X(t) ist dann nichts anderes als der Zustand des Prozesses zum Zeitpunkt t. Beispielsweise kann X(t) den Saldo eines Bankkontos, die Position eines Luftmoleküls im Raum, die Anzahl von Zuhörern in einer Vorlesung oder die Anzahl von Telefonanrufen, die seit dem frühen Morgen in einem Sekretariat eingegangen sind, darstellen (und zwar jeweils zum Zeitpunkt t). Für unsere Zwecke nehmen wir an, daß die Menge T ein Zeitintervall darstellt, der stochastische Prozeß also "kontinuierlich" ist.

Die Menge aller möglichen Werte, die die Zufallsvariable X(t) annehmen kann, nennen wir **Zustandsraum** des Prozesses. Stellt X(t) beispielsweise die Anzahl von Zuhörern in einer Vorlesung oder die Anzahl von Anrufen im Sekretariat dar, so ist der Zustandsraum sinnvollerweise die Menge {0, 1, 2, ...} aller natürlichen Zahlen. Einen derartigen Prozeß, dessen Zustandsraum die Menge der natürlichen Zahlen ist, nennt man auch **Zählprozeß**.

Betrachtet man nun einen kontinuierlichen Zählprozeß nur zu bestimmten Zeiten $t_0 < t_1 < ... < t_n$, so sind oftmals die Zuwächse $X(t_1) - X(t_0)$, $X(t_2) - X(t_1)$ etc. von Inter-

esse. Man sagt, der Prozeß besitze **unabhängige Inkremente**, wenn alle diese Zufallsvariablen $X(t_1) - X(t_0)$, $X(t_2) - X(t_1)$, ... $X(t_n) - X(t_{n-1})$ stochastisch unabhängig voneinander sind. Besitzt zusätzlich noch die Zufallsvariable $X(t_1+s) - X(t_0+s)$ dieselbe Verteilung wie $X(t_1) - X(t_0)$ für alle $t_0, t_1 \in T$ und $s > 0$, so hat der Prozeß sogar **stationäre unabhängige Inkremente**.

Nehmen wir zur Veranschaulichung die Anzahl in einem Sekretariat ankommender Telefonanrufe als Beispiel für einen Zählprozeß. Dieser Zählprozeß hat dann unabhängige Inkremente, wenn die Anzahl der Anrufe, die in einem bestimmten Zeitintervall ankommen, nicht davon abhängt, wieviele Anrufe in früheren oder späteren Zeitintervallen ankommen. Die Inkremente sind stationär dann, wenn (unabhängig von der Tageszeit) in gleichlangen Zeitintervallen gleiches Ankunftsverhalten herrscht (insbesondere etwa gleichviele Anrufe ankommen).

Poisson-Prozesse

Mit den eingeführten Begriffen können wir nun versuchen, einen Prozeß zu charakterisieren, der das "völlig zufällige" Eintreten von Ereignissen in kleinen Zeitintervallen beschreibt. Betrachten wir also die Wahrscheinlichkeit dafür, daß ein Ereignis innerhalb des Zeitintervalls [t, t+h] eintritt. Sinnvolle Annahmen für diese Wahrscheinlichkeit sind:

- Sie soll **proportional zur Intervall-Länge h** sein: Je länger ich warte, desto wahrscheinlicher tritt das Ereignis ein.

- Sie soll **proportional zu einer "Intensitätsrate"** λ sein, die die durchschnittliche Anzahl von Ereignissen pro Zeiteinheit darstellt: Je größer die Intensität, desto größer die Wahrscheinlichkeit.

- Sie soll **unabhängig von der Zeit t** sein: Es ist egal, wann ich auf das Eintreten eines Ereignisses zu warten beginne.

- Sie soll ferner **unabhängig von der Zahl und den Zeitpunkten bisheriger (und zukünftiger) Ereignisse** sein: "Vor/nach mir die Sintflut".

- Außerdem soll die Anzahl, wenn man genügend kleine Zeitintervalle betrachtet, immer nur um 1 wachsen können: Die Ereignisse sollen also **selten** genug auftreten, um auszuschließen, daß zwei Ereignissen gleichzeitig (oder "quasi-gleichzeitig") vorkommen.

Eine mathematische Formulierung dieser Kritierien für das "völlig zufällige" Eintreten von Ereignissen führt zu folgender Definition:

Definition: Ein Zählprozeß {N(t), t ≥ 0} heißt **Poisson-Prozeß** mit **Rate** λ > 0, wenn gilt:

(i) N(0) = 0
(ii) {N(t), t ≥ 0} hat stationäre, unabhängige Inkremente
(iii) P{N(h) ≥ 2} = o(h)
(iv) P{N(h) ≥ 1} = λh + o(h)

Dabei ist eine Funktion f von der Ordnung o(h), wenn gilt: $\lim_{h\to 0} f(h)/h = 0$, d. h. wenn f(h) "schneller gegen Null geht" als h.

Ein derartiger Poisson-Prozeß gehorcht der sogenannten **Poisson-Verteilung**. Damit läßt sich die Wahrscheinlichkeit dafür, daß im Intervall [0,t] genau i Ereignisse eintreten, darstellen als

$$P\{N(t) = i\} = \frac{(\lambda t)^i}{i!} \cdot e^{-\lambda t} .$$

Insbesondere ergibt sich hieraus der Mittelwert (Erwartungswert) der Ankünfte in dem betrachteten Intervall [0,t] zu λt. Er ist also proportional sowohl zur Intervallänge als auch zur Intensitätsrate, wie wir es oben ja gefordert haben.

Die Poisson-Verteilung liefert übrigens oft ausgezeichnete Approximationen, wenn man Prozesse betrachtet, die aus sehr vielen voneinander unabhängigen Teilkomponenten zusammengesetzt sind.

Wie lange benutzt eine Transaktion eine neue Sperre?

Nachdem wir in den Poisson-Prozessen ein geeignetes Modell für das Ankunftsverhalten von Sperrenanforderungen gefunden haben, ist nun noch zu klären, wie lange eine Transaktion nach der Anforderung einer neuen Sperre rechnen kann, bis sie die nächste Sperre anfordern muß.

Versuchen wir wieder, ein möglichst einfaches Modell zu beschreiben. Am allereinfachsten ist es sicherlich, wenn wir fordern, daß das Eintreten des nächsten Ereignisses unabhängig von der Vergangenheit geschehen soll. Ein solches Modell nennen wir "**memoryless**".

Diese Bedingung, daß die **Zwischenankunftszeit** *z*, also der zeitliche Abstand zwischen zwei Ereignissen, von der Vergangenheit unabhängig sein soll, wird von der (negativen) **Exponentialverteilung** erfüllt:

$$P\{z \le t\} = 1 - e^{-\lambda t}$$

Betrachtet man ein sehr kleines t, fragt also nach der Wahrscheinlichkeit dafür, daß der Abstand zwischen zwei Ereignissen winzig ist, so erhält man P{z≤t} ≈ 1-1 = 0: Daß also zwei Ereignisse fast gleichzeitig stattfinden, kommt so gut wie nicht vor. Sieht man sich das ganze für sehr großes t an, so ergibt sich eine Wahrscheinlichkeit P{z≤t} ≈ 1 - 0 = 1, d. h. wenn ich genügend lange warte, passiert "fast sicher" irgendwann etwas. Der Parameter λ übernimmt dabei im wesentlichen wieder die Rolle einer Intensität: mit wachsendem λ treten Ereignisse tendenziell früher ein.

Erfüllt die Exponentialverteilung tatsächlich die memoryless-Eigenschaft? Um dies zu überprüfen, berechnen wir die Wahrscheinlichkeit dafür, daß wir auf das Eintreten des nächsten Ereignisses nochmals eine Zeit von t_2 warten müssen, nachdem wir schon eine Zeit von t_1 gewartet haben. Mit Hilfe von bedingten Wahrscheinlichkeiten ergibt sich dafür

$$P\{z \geq t_2 + t_1 | z \geq t_1\} = \frac{e^{-\lambda(t_2+t_1)}}{e^{-\lambda t_1}} = e^{-\lambda t_2} = P\{z \geq t_2\}.$$

Anders ausgedrückt: Wie lange ich von einem bestimmten Zeitpunkt an auf das Eintreffen des nächsten Ereignisses warten muß, hat nichts damit zu tun, wie lange ich darauf schon gewartet habe.

Poissonprozesse und Exponentialverteilung hängen eng zusammen. Es läßt sich nämlich zeigen, daß ein Poisson-Prozeß mit Rate λ Zwischenankunftszeiten aufweist, die exponentialverteilt mit Parameter λ sind. Die **mittlere Zwischenankunftszeit** ergibt sich dann zu $1/\lambda$.

Zurück zum Modell

Wir nehmen also an, daß die Rechenzeit einer Transaktion pro erfolgreich erhaltener Sperre exponentialverteilt sei mit Parameter μ´, also im Mittel $1/\mu'$ beträgt.
Weiterhin nehmen wir an, daß die Wahrscheinlichkeit dafür, daß die Transaktion nach Bearbeitung der aktuellen Sperre terminiert, gleich Θ sei. Ist die Transaktion hingegen noch nicht beendet, so wählt sie ein festes der M Granules als neue Sperre mit Wahrscheinlichkeit

$$(1-\Theta)\cdot\frac{1}{M}.$$

Als Konsequenz ergibt sich, daß die Zahl der Sperren geometrisch verteilt mit Parameter Θ und Mittelwert $1/\Theta$ ist. Im Mittel dauert eine erfolgreiche Transaktion demnach

$$\frac{1}{\mu'}\cdot\frac{1}{\Theta} =: \frac{1}{\mu}$$

Nun betrachten wir eine Transaktion T_1, die zur Zeit t aktiv ist. Das bedeutet, daß T_1 einige Granules gesperrt hat, aber noch nicht fertig ist. Neben T_1 seien auch die Transaktionen T_2, T_3, ..., T_k aktiv zur Zeit t. Zu einem Konflikt kommt es, wenn T_1 nun auf ein Granule zugreifen will, das bereits von einer der anderen aktiven Transaktionen belegt ist.
Bezeichne $\gamma_k(t,dt)$ die Wahrscheinlichkeit dafür, daß T_1 im Intervall [t,t+dt] einen Konflikt erleidet unter der Voraussetzung, daß insgesamt k Transaktionen zur Zeit t aktiv sind. Außerdem bezeichne L_i die Anzahl der Sperren von T_i zur Zeit t. Dann erhält man näherungsweise

$$\gamma_k(t,dt) = \underbrace{\mu' dt}_{\substack{\text{Rate, mit der } T_1 \\ \text{mit der alten Sperre} \\ \text{"fertig" wird}}} \cdot \underbrace{(1-\Theta)}_{\substack{\text{Wahrscheinlichkeit,} \\ \text{daß ein zusätzliche} \\ \text{Sperre gebraucht wird}}} \cdot \underbrace{\frac{L_2 + L_3 + \ldots + L_k}{M}}_{\substack{\text{Wahrscheinlichkeit, daß besetzte} \\ \text{Sperre getroffen wird}}}$$

Wegen

$$\frac{L_2 + L_3 + ... + L_k}{M} \approx (k-1) \cdot \frac{1}{M} \cdot \underbrace{\frac{1}{\Theta}}_{\substack{\text{mittlere Zahl von} \\ \text{Granules pro TA}}}$$

erhält man daraus

$$\gamma_k(t, dt) \approx \frac{\mu'(1-\Theta)}{\Theta} \frac{(k-1)}{M} dt$$

Durch gewichtete Summation über alle k erhält man aus dieser bedingten Konfliktwahrscheinlichkeit die **unbedingte Konfliktwahrscheinlichkeit γ(t,dt)**:

$$\begin{aligned}
\gamma(t, dt) &= \sum_{k=1}^{\infty} \gamma_k(t, dt) \cdot P(k \text{ Transaktionen gleichzeitig aktiv}) \\
&= \sum_{k=1}^{\infty} \frac{\mu'(1-\Theta)}{\Theta} \cdot \frac{k-1}{M} dt \cdot P(k \text{ TAs gleichzeitig aktiv}) \\
&= \frac{\mu'(1-\Theta)}{\Theta} \frac{1}{M} dt \left[\sum_{k=1}^{\infty} k \cdot P(k \text{ TAs}) - \sum_{k=1}^{\infty} P(k \text{ TAs}) \right] \\
&= \frac{\mu'(1-\Theta)}{\Theta} \frac{1}{M} dt \left[\bar{k} - (1 - P(k=0)) \right]
\end{aligned}$$

mit $\bar{k}$ = mittlere Anzahl aktiver Transaktionen. Somit erhalten wir als **stationäre** Konflikt**rate** (also im Limes für $t \to \infty$)

$$\gamma = \frac{\mu'(1-\Theta)}{\Theta \cdot M} \cdot \left[\bar{k} - 1 + P(k=0) \right].$$

Sei f(T) die Wahrscheinlichkeit dafür, daß eine allgemeine Transaktion T Zeiteinheiten läuft, ohne von einem Konflikt betroffen zu sein. Da das Eintreten von Konflikten einen Poisson-Prozeß mit Rate γ darstellt, erhält man:

$$f(T) = e^{-\gamma \cdot T}$$

und damit

$$\begin{aligned}
p &:= P(\text{allgemeine Transaktion ist ohne Restart erfolgreich}) \\
&= \int_0^{\infty} f(h) \cdot \underbrace{P(h \le \text{TA - Zeit} \le h + dh}_{\substack{= \mu \cdot e^{-\mu h}, \text{ da TA-Zeit exponentialverteilt} \\ \text{mit Parameter } \mu}}) dh \\
&= \mu \int_0^{\infty} e^{-\gamma h} \cdot e^{-\mu h} dh = \frac{\mu}{\mu + \gamma}
\end{aligned}$$

Einsetzen ergibt schließlich

$$p = \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M}(\bar{k} - 1 + P(k=0))}$$

Hierbei handelt es sich um eine implizite Gleichung, da p auch noch in $\bar{k}$ steckt.

Als leicht vereinfachte untere Schranke für die Wahrscheinlichkeit, daß eine Transaktion erfolgreich ist, ohne einen Restart durchführen zu müssen, erhalten wir also

$$p \geq \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \cdot \bar{k}}$$

Verbesserung der unteren Schranke mit Little´s Result

Um bessere untere Schranken für p zu gewinnen, benötigen wir Formeln für $\bar{k}$ und P(k=0).
Ein Weg hierzu führt über "**Little´s Result**". Dieses besagt, daß in sehr allgemeinen Wartesystemen die mittlere Anzahl der im System befindlichen "Kunden" gleich dem Produkt aus Ankunftsrate und mittlerer Systemzeit (= Wartezeit + Bedienzeit) pro Kunde ist. In unserem Fall heißt das:

$$\underbrace{\bar{k}}_{\substack{\text{mittlere Trans-}\\ \text{aktionszahl}}} = \underbrace{\lambda}_{\text{Ankunftsrate}} \cdot \underbrace{\bar{T}}_{\substack{\text{mittlere}\\ \text{TA-Dauer}}}$$

Sei nun $\bar{T}_i$ die mittlere Zeit, die eine Transaktion benötigt, wenn sie genau im i-ten Versuch erfolgreich ist. Dann erhalten wir für die mittlere Transaktionsdauer

$$\bar{T} = \sum_{i=1}^{\infty} \bar{T}_i \cdot P(i-1 \text{ Restarts})$$

$$= \sum_{i=1}^{\infty} \bar{T}_i \cdot \underbrace{(1-p)^{i-1}}_{\substack{(i-1)-\text{mal}\\ \text{erfolglos}}} \cdot \underbrace{p}_{\substack{i-\text{ter}\\ \text{Versuch o. k.}}}$$

Nun ist aber

$$\bar{T}_1 = \frac{1}{\mu}$$

sowie

$\bar{T}_i \leq i \cdot \bar{T}_1$, da erfolglose Versuche i. a. kürzer sind als erfolgreiche

$$\Rightarrow \quad \overline{T} = \sum_{i=1}^{\infty} \overline{T}_i \cdot (1-p)^{i-1} \cdot p \le \frac{1}{\mu} \underbrace{\sum_{i=1}^{\infty} i(1-p)^{i-1} p}_{=1/p}$$

$$= \frac{1}{\mu \cdot p}$$

Setzt man dies via Little´s Result in unser bisheriges Ergebnis ein, so erhält man

$$p \ge \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{p} \cdot \frac{1}{\mu}} \overset{\text{Rechnung}}{\geq} 1 - \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu} \qquad (*)$$

Simple Two Phase Locking

Wenn beim Simple 2 PL eine Transaktion startet, dann braucht sie mehrere Sperren. Immer dann, wenn eine der angeforderten Sperren belegt ist, setzt die Transaktion zurück und beginnt von neuem. (Dadurch stellen genaugenommen die Transaktionsankünfte keine Poisson-Prozesse mehr dar. Für unsere näherungsweisen Berechnungen soll dies aber keine Rolle spielen.)

Im allgemeinen macht eine Transaktion somit mehrere Versuche (an deren Ende jeweils ein Konflikt steht), bis letztlich ein Versuch doch erfolgreich ist. In unserer bisherigen Darstellung haben wir einmal die Konfliktrate γ einer Transaktion (pro Zeiteinheit) abgeschätzt. Ferner haben wir eine untere Schranke für die Wahrscheinlichkeit p gefunden, daß eine Transaktion ungestört durchkommt.

Ist nun M ›› 1, dann können wir das Transaktionssystem als sogenanntes **M/G/∞-System** interpretieren. Diese Notation deutet an, daß wir ein System vor uns haben, bei dem Kunden gemäß einem Poisson-Prozeß ankommen (das "**M**" steht für "memoryless") und dann eine allgemeine Bedienung erfahren ("**G**" wie "general"), d. h. es werden keine speziellen Annahmen über die Abfertigungszeit gemacht. Außerdem sollen hierfür ∞ viele Bediener zur Verfügung stehen.

In unserem Fall liegt dann ein Ankunftsprozeß der Rate λ vor, die Bedienrate beträgt kμp, ferner ist $P(k=0) = e^{-\lambda/\mu p}$.

Dann erhält man aus (*)

$$p \ge \frac{1}{1 + \frac{1-\Theta}{\Theta^2 M}\left(\frac{\lambda}{\mu p} + e^{-\lambda/\mu p} - 1\right)}$$

Rechnung ergibt

$$1 \ge \frac{1 - \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu}}{1 + \frac{1-\Theta}{\Theta^2 M}\left(e^{-\lambda/\mu p} - 1\right)}$$

und schließlich

$$p \geq \frac{-\frac{\lambda}{\mu}}{\log\left(1-\frac{\lambda}{\mu}\right)} \qquad (**)$$

falls $\frac{\lambda}{\mu} < 1$ (Stabilitätsbedingung) und $\frac{1}{\Theta} \leq \sqrt{M}$ gilt.

Ist dagegen $\frac{1}{\Theta}$ groß, z. B. $\frac{1}{\Theta} \approx M$ (d. h. eine Transaktion braucht praktisch alle Sperren), dann war die bisherige Abschätzung der Sperrenzahl

$$L_2 + L_3 + ... + L_k \approx \frac{k-1}{\Theta} \approx (k-1)M$$

zu pessimistisch, da es ja nicht mehr als M Sperren gibt. Schlimmstenfalls erhält man hier

$$\begin{aligned} & L_2 + L_3 + ... + L_k \leq M - 1 \\ \Rightarrow \quad & \gamma_k(t, dt) \leq \mu'(1-\Theta)(1-\frac{1}{M})dt \\ \Rightarrow \quad & p = \frac{\mu}{\mu+\gamma} \geq \frac{\mu'\Theta}{\mu'\Theta + \mu'(1-\Theta)\frac{M-1}{M}} \\ & = \frac{M\Theta}{M\Theta + (1-\Theta)(M-1)} \end{aligned} \qquad (***)$$

als untere Schranke, die nicht mehr von k abhängig ist.

Beispielergebnisse:

Abschließend wollen wir die gefundenen Abschätzungen noch graphisch veranschaulichen.
Betrachten wir als erstes den Fall $1/\Theta \leq \sqrt{M}$ (kleine Transaktion) und $\rho = \lambda/\mu < 1$. Hierfür erhielten wir die Abschätzung (**)

$$p_L (= P(\text{TA kommt durch})) = \frac{-\rho}{\log(1-\rho)}$$

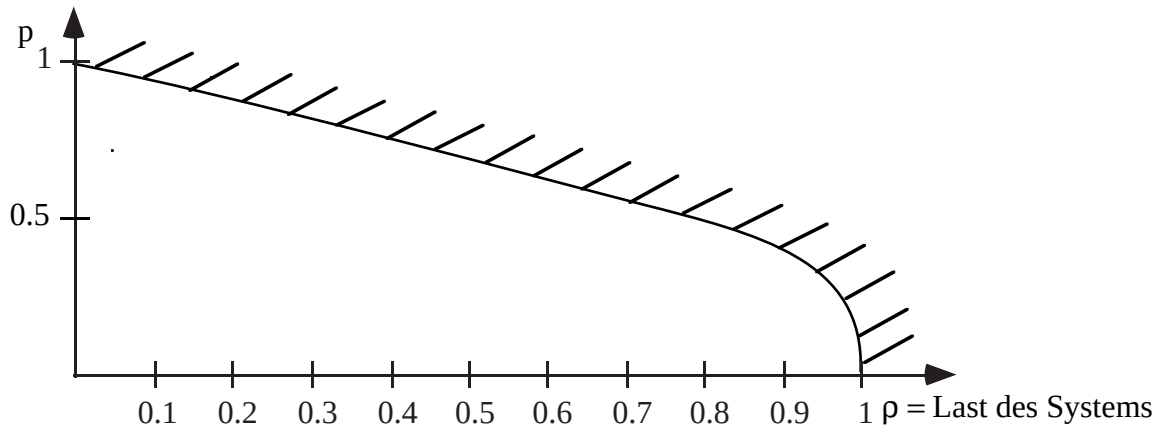


Abb.3.20: Leistungsverhalten von Simple 2 PL bei kleinen Transaktionen

Aus dieser Kurve läßt sich ersehen, daß im Falle kleiner Transaktionen bei nicht zu hoher Last das Simple 2 PL gar nicht einmal so schlecht funktioniert, z. B. beträgt bei einer Last $\rho = 0{,}5$ die Chance auf ein Durchkommen ohne Störung immerhin etwa 75% (d. h. 3 von 4 Fällen sind im Mittel erfolgreich).

Bei großen Transaktionen ($1/\Theta > \sqrt{M}$) wird das Leistungsverhalten deutlich schlechter. Zur Abschätzung können wir die Ungleichung (*) und (***) verwenden, d. h. wir haben für die Wahrscheinlichkeit p_L daß eine Transaktion störungsfrei verläuft, die beiden unteren Schranken

$$p_{L_1} = 1 - \frac{1-\Theta}{\Theta^2 M} \cdot \frac{\lambda}{\mu}$$

und

$$p_{L_2} = \frac{M\Theta}{M\Theta + (1-\Theta)(M-1)}$$

Nehmen wir als Beispiel ein System mit M = 1000 Granules und einer mittleren Zahl von einer Transaktion angeforderten Sperren $1/\Theta = 40 > \sqrt{1000}$, so erhalten wir folgendes Bild:

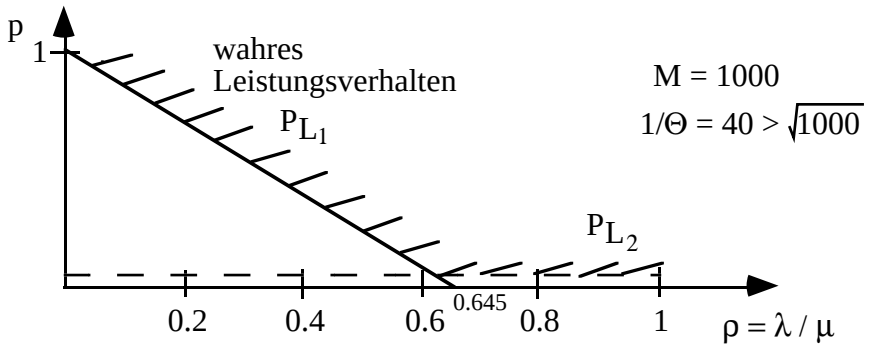


Abb. 3.21: Leistungsverhalten

Ab ca. 60% erweist sich also (zumindest von der unteren Schranke her) das Simple 2 PL als so gut wie unbrauchbar.

4 Deadlocks

Anfang dieses Jahrhunderts sah sich der Gesetzgeber des US-Bundesstaates Kansas gezwungen, eine Vorfahrtsregelung an Eisenbahnkreuzungen einzuführen. Zu diesem Behufe erließ er folgende Bestimmung: "Nähern sich zwei Züge einer Kreuzung, so hat jeder von ihnen zum vollständigen Stillstand zu kommen. Ferner darf keiner von beiden weiterfahren, solange der andere noch dasteht." (nachzulesen in "The Columbus Chicken Statute, and More Bonehead Legislation" von D. Hyman, S. Greene Press, Lexington, 1985).

Dies ist ein geradezu klassisches Beispiel für einen Systemzustand, in dem Prozesse (in diesem Fall die beiden Züge) auf Ereignisse (wie z. B. die Zuteilung von Betriebsmitteln) warten, die niemals eintreten werden. Einen solchen Zustand bezeichnet man als **Deadlock** oder **Verklemmung**. Eine wichtige Aufgabe bei der Entwicklung von Betriebssystemen, die Multiprogramming erlauben, besteht nun darin, Regelungen für den Umgang mit derartigen Situationen vorzusehen.

4.1 Systemmodell und notwendige Bedingungen

Betrachten wir zunächst kurz das zugrundeliegende Systemmodell. Wir nehmen an, das System bestehe aus einer Anzahl n von Prozessen P_1, ..., P_n ($n \geq 2$) und einer Anzahl verschiedener Betriebsmittel B_1, ..., B_m. Von jedem Betriebsmittel liegen evtl. mehrere identische Exemplare vor.

Die Prozesse fordern nun im Lauf ihrer Abarbeitung verschiedentlich Betriebsmittel an. Ist ein Exemplar des angeforderten Betriebsmittels gerade verfügbar, so wird es dem Prozeß zugeteilt, woraufhin dieser in der Abarbeitung fortfährt und das Betriebsmittel wieder freigibt, sobald er es nicht mehr benötigt. Sind aber sämtliche Exemplare des angeforderten Betriebsmittels gerade belegt, so muß der Prozeß solange warten, bis eines davon wieder frei wird.

Derartige Anforderungen und Zuteilungen von Betriebsmitteln lassen sich am besten graphisch charakterisieren. Eine Möglichkeit hierzu bietet der "**Request-Allocation-Graph**":

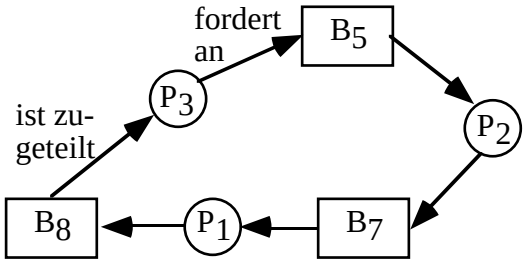


Abb. 4.1: Beispiel eines "Request-Allocation-Graphen"

Ein Pfeil von einem Betriebsmittel zu einem Prozeß bedeutet dabei, daß das Betriebsmittel gerade von dem Prozeß belegt ist. Zeigt ein Pfeil von einem Prozeß zu einem Betriebsmittel, so symbolisiert dies, daß der Prozeß gerne auf das Betriebsmittel zugreifen möchte. In unserem Beispiel belegt also der Prozeß P_3 gerade das Betriebsmittel B_8, möchte aber gerne noch auf B_5 zugreifen.

Dieser Graph läßt sich dadurch vereinfachen, daß man die Betriebsmittel wegläßt; man erhält so einen "**Wait-for-Graphen**":

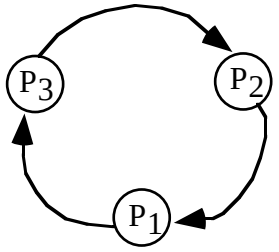


Abb. 4.2: Beispiel eines "Wait-for-Graphen"

Hier bedeutet der Pfeil von P_3 nach P_2 einfach, daß P_3 ein Betriebsmittel (nämlich B_5, aber dies explizit anzugeben ist jetzt nicht mehr notwendig!) anfordert.

Deadlocks und Systemzustände können (für zwei Betriebsmittel) mit sogenannten **Prozeßfortschrittsdiagrammen** veranschaulicht werden. Dazu ein Beispiel:

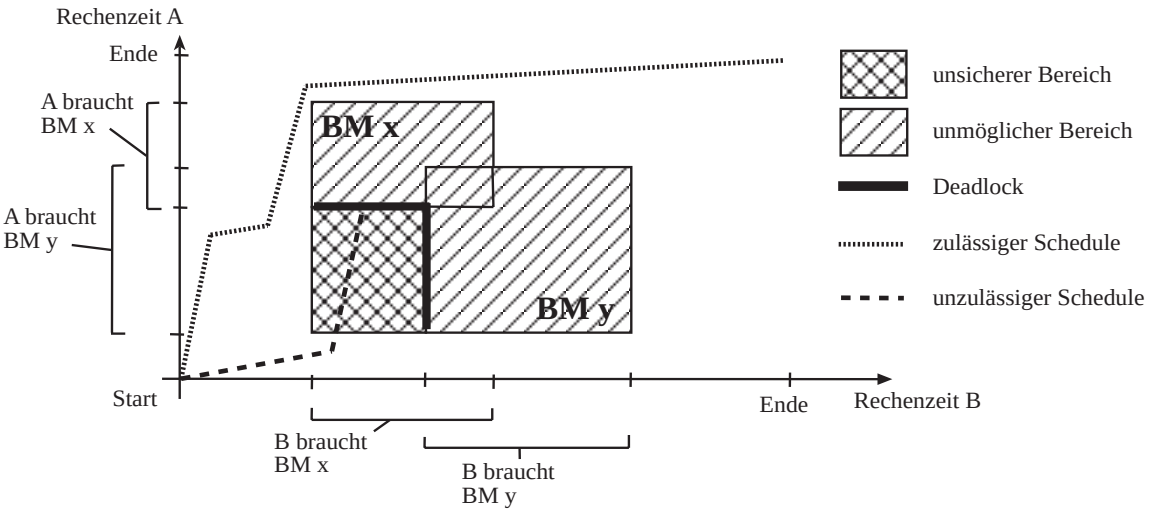


Abb. 4.3: Prozeßfortschrittsdiagramm

Zwei Prozesse A und B konkurrieren um zwei Betriebsmittel x und y. Ein **Schedule** gibt an, wie die Prozesse innerhalb ihrer Rechenzeit voranschreiten. Schraffierte Flächen geben an, in welchen Zeiträumen Betriebsmittel von beiden Prozessen beansprucht werden. Diese Bereiche können durch einen Schedule nicht erreicht werden **(unmöglicher Bereich)**. Gelangt ein Schedule in einen **unsicheren Bereich** (karierte Fläche), so erreicht er bei weiterem Fortschritt zwangsläufig einen **Deadlock-Zustand**.

Das Vorliegen eines Deadlocks läßt sich nun einfach durch folgende vier Bedingungen charakterisieren, die allesamt gleichzeitig erfüllt sein müssen:

A. Circular Wait
Es muß im Request-Allocation-Graphen (bzw. im Wait-for-Graphen) eine geschlossene Kette vorliegen. Genauer gesagt muß eine Menge $\{P_1, ... P_k\}$ wartender Prozesse existieren derart, daß P_1 auf ein Betriebsmittel wartet, das gerade von P_2 belegt wird, P_2 seinerseits auf eines wartet, das von P_3 belegt wird usw. bis schließlich P_k auf ein Betriebsmittel wartet, das von P_1 belegt wird.

B. Exclusive Use
Keines der fraglichen Betriebsmittel kann von mehreren Prozessen gleichzeitig benutzt werden (kein "Betriebsmittel-Sharing").

C. Hold and Wait
Prozesse können ein neues Betriebsmittel anfordern, ohne dabei die bisher von ihnen belegten zurückzugeben.

D. No Preemption
Betriebsmittel können von den Prozessen nur freiwillig zurückgegeben werden, ein Entzug von außen (durch "höhere Gewalt") ist nicht möglich.

Natürlich muß man im Ernstfall, wenn also einmal ein Deadlock-Zustand erreicht ist, bei der letzten Bedingung ansetzen, d. h. von außen entweder einen Prozeß "killen" oder ihm einige Betriebsmittel wegnehmen.

4.2 Gegenmaßnahmen

Prinzipiell gibt es drei verschiedene Möglichkeiten, mit der Deadlock-Problematik umzugehen. Man könnte erstens sicherstellen, daß das System niemals in einen Deadlock-Zustand geraten kann. Dies erreicht man entweder durch "**Deadlock-Prevention**" (d. h. Ausschluß einer der vier notwendigen Bedingungen) oder durch "**Deadlock-Avoidance**" (hierunter versteht man Verfahren, die Deadlocks durch Zusatzinformationen vermeiden; beispielsweise kann man vor jeder "riskanten" Operation erst einmal einen Test machen und die Operation nur dann zulassen, wenn sie "ungefährlich" ist).
Eine zweite Möglichkeit besteht darin, Deadlocks zuzulassen. In diesem Fall sind Mechanismen nötig, die in der Lage sind, das Vorliegen eines Deadlock-Zustands zu erkennen und ihn daraufhin zu beseitigen.
Drittens schließlich gibt es die Möglichkeit, die Problematik zu ignorieren. Hier nimmt man also an, daß Deadlocks so gut wie nie auftreten. Interessanterweise verfahren die meisten Betriebssysteme (einschließlich beispielsweise UNIX) nach dieser Methode.

4.2.1 Deadlock-Prevention

Wie bereits erwähnt, lassen sich Deadlocks einfach dadurch ausschließen, daß man eine der Bedingungen A - D unmöglich macht. Entsprechend gibt es folgende vier Ansätze:

- **Betriebsmittel-Sharing**, um Bedingung B zu vereiteln. Leider geht dies häufig aus praktischen Gründen nicht (z. B. den Drucker zu teilen).
- **Preemption**, d. h. Eingriffe von außen sind möglich. Dies hat leicht den Ruch einer Gewaltmaßnahme.

- **"Alles oder nichts"**: Ein Prozeß geht nicht nach dem Prinzip "hold and wait" vor, sondern fordert alle Betriebsmittel gleichzeitig an. (Dieses Verfahren besticht offensichtlich durch seine Ineffizienz. Man beginnt den Bau eines Hauses ja in der Regel auch nicht erst dann, wenn man endlich die Tapeten eingekauft hat.)

- **Unmöglichkeit eines Zyklus**. Dies wird algorithmisch häufig durch Einführung einer Betriebsmittel-Hierarchie folgendermaßen verwirklicht:

  - Teile Betriebsmittel in Klassen K_1, ..., K_h ein (mit $h \geq 1$).
  - Wenn ein Prozeß P_i aktuell Betriebsmittel der Klasse K_r (und niedrigerer Klassen) belegt, dann darf er "nur nach oben" anfordern, d. h. er darf nur auf Betriebsmittel der Klassen K_{r+1}, ..., K_h warten.
  - Sollte er dagegen weitere Betriebsmittel einer niedrigeren Klasse K_i (mit $i \leq r$) benötigen, dann muß er zunächst alle derzeit belegten Betriebsmittel der Klassen K_i, K_{i+1}, ..., K_r freigeben.

Daß dieses Schema einen Zyklus vermeidet, läßt sich leicht nachweisen. Nehmen wir hierzu an, es existiere ein Zyklus (P_0, P_1, ..., P_n) aufeinander wartender Prozesse. Bezeichne zu jedem dieser Prozesse $F(P_i)$ die höchste Klasse der von ihm angeforderten Betriebsmittel. Dann gilt: $F(P_0) < F(P_1) < F(P_2) < ... < F(P_n) < F(P_0)$, da jeder Prozeß ja nur Betriebsmittel einer höheren als den bisher belegten Klassen anfordern darf. $F(P_0) < F(P_0)$ ist aber ein Widerspruch.

Wichtig ist, daß auch bei Anforderung eines zusätzlichen Betriebsmittels einer bereits gehaltenen Klasse die "alten" Betriebsmittel dieser Klasse erst zurückgegeben werden müssen.

Läßt man in dem angegebenen Schema nur eine einzige Klasse zu ($h = 1$), so ist dies äquivalent zur sequentiellen Nutzung der Betriebsmittel, da in diesem Fall vor einer Neuanforderung erst alle bisher belegten Betriebsmittel zurückzugeben sind.

4.2.2 Deadlock-Avoidance

Die bislang vorgestellten Mechanismen verhindern das Eintreten eines Deadlocks, indem sie sicherstellen, daß nicht alle vier notwendigen Bedingungen gleichzeitig erfüllt sein können. Sie erreichen dies durch Einschränkungen bei der Anforderung von Betriebsmitteln. Leider wird dabei die Deadlockfreiheit durch Ineffizienz bei der Betriebsmittelnutzung erkauft, was letztlich auf Kosten des Systemdurchsatzes geht.

Eine alternative Vorgehensweise vor Vermeidung von Deadlocks nutzt zusätzliche Informationen um zu prüfen, ob Zustandsübergänge aufgrund von Betriebsmittel-Zuteilungen "sicher" sind. Es soll also ausgeschlossen werden, daß ein System, das sich nachgewiesenermaßen in einem gefahrlosen Zustand befindet, durch die Zuteilung eines Betriebsmittels in einen Zustand gerät, der evtl. in einen Deadlock einmünden könnte.

Was ein "sicherer Zustand" ist, darüber gehen die Meinungen teilweise auseinander. Wir halten uns an folgende Festlegung:

Ein System, bestehend aus den Prozessen P_1, ..., P_n, befindet sich genau dann in einem **sicheren Zustand**, wenn eine Reihenfolge von Prozeßausführungen (P_{i_1}, P_{i_2}, ..., P_{i_n}) existiert, so daß für alle j der Prozeß P_{i_j} weitergeführt werden kann, wenn P_{i_j}

seine maximalen Anforderungen stellt und wenn alle vorherigen Prozesse P_{i_1}, ... $P_{i_{j-1}}$ ihre Betriebsmittel zurückgegeben haben.

Existiert keine solche Folge von Prozessen, so ist das System in einem "unsicheren" Zustand. Der Startzustand ist natürlich automatisch "sicher".

Ein sicherer Zustand läßt sich also nach dieser Definition dadurch charakterisieren, daß es eine Möglichkeit gibt, alle Anforderungen auch im ungünstigsten Fall zu befriedigen. Betrachten wir zur Verdeutlichung folgendes Beispiel: Es gebe ein Betriebsmittel, davon aber gleich 12 Exemplare. Die drei Prozesse P_1, P_2 und P_3 haben folgende maximale Anforderungen: P_1: 10, P_2: 4 und P_3: 9 Exemplare des Betriebsmittels. Die aktuelle Zuteilung betrage P_1: 5, P_2: 2 und P_3: 2. Es gilt nun, eine Reihenfolge für die drei Prozesse zu finden, die die oben genannte Bedingung erfüllt.

Durch einfaches Ausprobieren stellt man fest, daß die Reihenfolge P_2, P_1, P_3 folgende Zustandsübergänge impliziert:

Schritt	Prozeß P_1	Prozeß P_2	Prozeß P_3	freie Betriebsmittel	ausgeführte Aktion
1	5	2	2	3	2 BM an P_2
2	5	4	2	1	4 BM von P_2
3	5	0	2	5	5 BM an P_1
4	10	0	2	0	10 BM von P_1
5	0	0	2	10	7 BM an P_3
6	0	0	9	3	9 BM von P_3
7	0	0	0	12	

Abb. 4.4: Beispiel einer möglichen Betriebsmittelanforderungsreihenfolge für 3 Prozesse

Folglich befindet sich das System in einem sicheren Zustand, kann also nicht in einen Deadlock geraten. Nehmen wir jetzt an, die aktuelle Zuteilung von P_3 betrage 3 statt 2. Dann erhalten wir:

Schritt	Prozeß P_1	Prozeß P_2	Prozeß P_3	freie Betriebsmittel	ausgeführte Aktion
1	5	2	3	2	2 BM an P_2
2	5	4	3	0	4 BM von P_2
3	5	0	3	4	

Deadlock, da P_1 5 BM braucht und P_3 6 BM benötigt

Abb. 4.5: Betriebsmittelanforderungsreihenfolge, die zu einem Deadlock führt

Da auch keine andere Reihenfolge der Prozeßabarbeitung möglich ist, stellt also der Zustand $P_1 = 5$, $P_2 = 2$ und $P_3 = 3$ einen unsicheren Zustand dar. Er kann zu einem Deadlock führen, nämlich dann, wenn P_1 und P_3 ihre Maximalanforderungen nutzen.

Deadlock-Avoidance-Algorithmen machen sich dieses Konzept nun dadurch zunutze, daß sie vor einer geplanten Betriebsmittelzuteilung prüfen, ob sich das System dadurch evtl. in einen unsicheren Zustand manövriert. Wird durch diesen Test festgestellt, daß der Folgezustand unsicher wäre, dann wird die geplante Zuteilung verworfen, andernfalls wird sie ausgeführt.

4.2.3 Der Banker´s Algorithmus

Ein bekannter Algorithmus zur Deadlock-Avoidance ist der "**Banker´s Algorithmus**". Er verfährt analog zu einem vorsichtigen Bankier, der sichergehen will, daß er die monetären Forderungen seiner Kunden mittels der verliehenen (und zu gegebener Zeit zurückkommenden) und der im Tresor verbliebenen Geldmenge befriedigen kann.

Man kann den Banker´s Algorithmus für zwei unterschiedliche Aufgaben heranziehen: Einmal dient er zur Deadlock-Avoidance (nämlich dann, wenn er auf **maximalen Anforderungen** beruht), ferner kann er auch zur Erkennung von Deadlocks eingesetzt werden (und basiert dann auf dem **aktuellen Zustand**).

Für die Implementierung des Algorithmus gehen wir von n Prozessen P_1, P_2, ..., P_n und m Betriebsmitteltypen B_1, B_2, ..., B_m aus. Seien ferner

$Q_{j,s}^{max}(k)$:= zur Zeit k von P_j maximal zusätzlich anforderbare Betriebsmittel vom Typ B_s

$Q_{j,s}(k)$:= zur Zeit k von P_j aktuell angeforderte Betriebsmittel vom Typ B_s ("re**Q**uest")

$H_{j,s}(k)$:= zur Zeit k von P_j ge**h**altene Betriebsmittel vom Typ B_s

$V_s(k)$:= zur Zeit k **v**erfügbare Betriebsmittel vom Typ B_s

Der Einfachheit halber betrachten wir ab sofort Q_j^{max}, Q_j, H_j und V als m-dimensionale Vektoren, entsprechend gilt dann das "≤-Zeichen" jeweils für alle Einzelkomponenten.

Wann gilt nun ein Zustand als unsicher? Im Fall der Deadlock-Avoidance muß man einen Zustand dann als unsicher bezeichnen, wenn es eine Teilmenge D von Prozessen gibt, so daß für keinen Prozeß aus D seine maximale Anforderung erfüllbar ist, selbst dann nicht, wenn alle nicht in D enthaltenen Prozesse ihre gehaltenen Betriebsmittel zurückgeben.

Formal ausgedrückt ist ein Zustand zum Zeitpunkt k genau dann unsicher, wenn gilt:

$$\exists\, D \subset \{P_1, P_2, ..., P_n\} \text{ mit } Q_j^{max}(k) > V(k) + \sum_{r \notin D} H_r(k) \quad \forall\, j \in D$$

Um den Banker´s Algorithmus zur Erkennung von Deadlocks einsetzen zu können, genügt es, Q_j^{max} durch Q_j zu ersetzen.

Doch zurück zur Deadlock-Avoidance. Der Banker´s Algorithmus dafür besteht aus zwei Teilen: der Sicherheitsprüfung und der Betriebsmittelzuteilung an einen geeigneten Prozeß. Er lautet folgendermaßen:

Teil 1 (Sicherheitsprüfung)

(1) Durchsuche P_1, ..., P_n nach dem ersten unmarkierten Prozeß P_i mit
$Q_i^{max}(k) \leq V(k)$!
Existiert kein solches i, dann gehe nach (3)!

(2) Setze $V(k) := V(k) + H_i(k)$ [d. h. P_i "arbeitet" mit den Betriebsmitteln und gibt die gehaltenen danach zurück];
"markiere" P_i;
zurück nach (1)

(3) Wurden alle P_i markiert, dann ist das System bezüglich des Betriebsmittelvektors B sicher, andernfalls ist es unsicher.

Teil 2 (Betriebsmittel-Zuteilung an einen Prozeß P_j)

(A) Ist $Q_j(k) \leq V(k)$, dann gehe nach (B);
ansonsten muß P_j warten.

(B) Setze (probehalber)
$V(k) := V(k) - Q_j(k)$
$H_j(k) := H_j(k) + Q_j(k)$
$Q_j^{max}(k) := Q_j^{max}(k) - Q_j(k)$

(C) Prüfe (mit Teil 1), ob der neue Zustand sicher ist.
wenn ja: o. k.
wenn nein: Zuteilung verwerfen (restore)

Exemplarisch wollen wir diesen Algorithmus auf unser Beispiel von vorhin anwenden.

	P_1	P_2	P_3	ausgeführte Aktion
$Q_j(k)$				1.) Prüfe P_1: $Q_1^{max}(k) = 5 > 3 = V(k)$ => geht nicht
$H_j(k)$	5	2	2	2.) Prüfe P_2: $Q_2^{max}(k) = 2 <= 3 = V(k)$ =>
$Q_j^{max}(k)$	5	2	7	$V(k) = V(k) + H_2(k) = 3 + 2 = 5$
V(k)		3		P_2 markieren
V(k)		5		1.) Prüfe P_1: $Q_1^{max}(k) = 5 >= 5 = V(k)$ $V(k) = V(k) + H_1(k) = 5 + 5 = 10$ P_1 markieren
V(k)		10		1.) Prüfe P_1: $Q_3^{max}(k) = 7 < 10 = V(k)$ $V(k) = V(k) + H_3(k) = 10 + 2 = 12$ P_3 markieren
V(k)		12		

Abb. 4.6: Banker´s Algorithmus für die korrekte BM-Anforderung

	P_1	P_2	P_3	ausgeführte Aktion
$Q_j(k)$				1.) Prüfe P_1: $Q_1^{max}(k) = 5 > 2 = V(k)$ => geht nicht
$H_j(k)$	5	2	3	2.) Prüfe P_2: $Q_2^{max}(k) = 2 <= 2 = V(k)$ =>
$Q_j^{max}(k)$	5	2	6	$V(k) = V(k) + H_2(k) = 2 + 2 = 4$
V(k)		2		P_2 markieren
V(k)		4		1.) Prüfe P_1: $Q_1^{max}(k) = 5 >= 4 = V(k)$ => geht nicht 2.) Prüfe P_3: $Q_3^{max}(k) = 6 >= 4 = V(k)$ => geht nicht => D = {P_1, P_3} nicht bearbeitbar; Zustand unsicher

Abb. 4.7: Banker´s Algorithmus für die unkorrekte BM-Anforderung

Abschließend noch eine Bemerkung zur Laufzeit des Algorithmus. In Prozeßtests läßt sich ermitteln, daß diese im besten Fall linear in der Anzahl der Prozesse *n* wächst, während sie im "worst case" $O(n^2)$ beträgt. Letzteres läßt sich leicht plausibel machen, wenn man davon ausgeht, daß im ersten Schritt alle *n* unmarkierten Prozesse zu testen sind, da erst der *n*-te paßt. Im nächsten Schritt kann es dann im ungünstigsten Fall zu *n*-1 Tests kommen, danach zu *n*-2 usw. In der Summe ergibt dies insgesamt

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \quad \text{Tests.}$$

Eine Verbesserung (aber nur für den Fall, daß $m = 1$ ist, also lediglich ein Betriebsmitteltyp vorliegt) liefert das **Verfahren von Holt**. Es beruht darauf, daß zunächst die Betriebsmittelanforderungen der Größe nach vorsortiert werden. Daraufhin werden die Prozesse der Reihe nach von der kleinsten bis zur größten Anforderung abgetestet. In diesem Fall wächst die Laufzeit für den Banker´s Algorithmus nur noch linear in n. Berücksichtigt man noch, daß ein schnelles Verfahren für die Vorsortierung (wie z. B. Heapsort) $O(n \log n)$ aufweist, so ergibt das Verfahren von Holt insgesamt eine Laufzeitabhängigkeit der Größenordnung $O(n \log n)$, was für große n eine enorme Verbesserung darstellen kann.

Banker´s Algorithmus	Banker´s Algorithmus in Kombination mit dem Verfahren von Holt
Laufzeit im "worst case": $\approx n^2/2 + n/2$ ==> $O(n^2)$	Laufzeit im "worst case": a) Sortierung z. B. mit Heap-Sort = $O(n \log n)$ b) lineare Testzeit = $O(n)$ ==> $O(n \log n)$
Beispiel: n = 1000 ==> $\approx 1000^2/2 + 1000/2$ $\approx 10^6 / 2$ = 500.000 Operationen	==> $\approx 1000 \log 1000$ ≈ 10000 Operationen

Abb. 4.8: Laufzeit des Banker´s Algorithmus für m = 1 mit und ohne das Verfahren von Holt

Natürlich verbraucht der Banker´s Algorithmus Rechenzeit. Daher ist die Frage, wie oft man ihn sinnvollerweise aufrufen sollte, durchaus ernstzunehmen. Es gibt darauf nur einige heuristische Antworten:

Häufige Aufrufe sind zu befürworten, wenn zu befürchten steht, daß Deadlocks häufig auftreten. Der Grund liegt darin, daß nach dem Eintreten eines Deadlocks, der oft eigentlich nur einige wenige Prozesse betrifft, ein Risiko dafür besteht, daß sich andere Prozesse ebenfalls verklemmen (und es zu mehreren Deadlock-Ketten kommt).

Eine andere Möglichkeit besteht darin zu warten, bis die Systemleistung offensichtlich stark heruntergeht, und erst danach zu prüfen. Dieses Vorgehen beruht auf der Annahme, daß Deadlocks nicht eintreten, solange die Systemleistung (wie etwa der Durchsatz von Jobs pro Zeiteinheit) genügend hoch ist.

Drittens kann man eine Prüfung in periodischen Abständen durchführen. Prüft man hierbei zu selten, so besteht allerdings die Gefahr, daß das System u. U. lange stillsteht, prüft man zu häufig, so wird evtl. unnötig Rechenzeit verbraten.
Schließlich besteht eine letzte Alternative darin, die Gefahr eines Deadlocks einfach zu ignorieren und erst zu reagieren (dann freilich mit Brachialgewalt), wenn wirklich gar nichts mehr geht. Diese Strategie ist zwar schlecht, aber einfach zu realisieren.

Dabei stellt sich natürlich abschließend noch die Frage, was zu tun ist, wenn wirklich einmal ein Deadlock eingetreten ist. Grundsätzlich sind hier zwei Vorgehensweisen

möglich. Zum einen kann man einfach hergehen und sämtliche verklemmte Prozesse killen und neustarten. Eine etwas differenziertere Methode besteht darin, der Reihe nach einzelne der betroffenen Prozesse zu killen und darauf zu hoffen, daß sich von irgendeinem Punkt an der Deadlock löst, da eine der vier notwendigen Bedingungen durchbrochen wird.

5 CPU-Scheduling

5.1 Grundlegendes

In einem Einprozessorsystem, das Multiprogramming erlaubt, läuft in der Regel einer der Prozesse auf der CPU (und befindet sich folglich im Zustand running), während verschiedene andere Prozesse im ready-Status darauf warten, bis der genannte Prozeß die CPU freigibt. Ist dies der Fall, so stellt sich natürlich die Frage, welcher der lauffähigen Prozesse vom Betriebssystem in den running-Zustand versetzt werden soll, also als nächster drankommt.

Eine Antwort darauf kann von verschiedenen Kriterien abhängig gemacht werden. Einmal kann man versuchen, möglichst "fair" vorzugehen (also z. B. keinen Prozeß zu lange warten zu lassen oder keinem Prozeß zu viel CPU-Zeit zuzuteilen). Man kann den Prozessen verschieden hohe Wichtigkeit zuordnen, was auf eine Prioritätenregelung hinausläuft. Oder man kann danach trachten, das Leistungsverhalten der CPU in den Mittelpunkt zu stellen, um beispielsweise einen möglichst hohen Durchsatz, hohe CPU-Auslastung, eine minimale mittlere Wartezeit (= Zeit, bis ein Prozeß drankommt) oder eine minimale mittlere Antwortzeit (d. h. Wartezeit + Bedienzeit) zu gewährleisten. Im allgemeinen wird man sich nicht auf ein einziges Kriterium versteifen, sondern verschiedene Aspekte in einer (hoffentlich gelungenen) Mischung betrachten. Aufgrund dieser Mischung muß man sich dann für eine Scheduling-Strategie entscheiden, deren mehrere im folgenden vorgestellt werden sollen.

5.2 Scheduling-Strategien

5.2.1 FIFO und LIFO

Bei weitem am einfachsten ist die Strategie "First-In-First-Out" (FIFO) bzw. "First-Come-First-Served" (FCFS). Nach diesem Schema erhält der Prozeß, der als erster CPU-Zeit angefordert hat (und also schon am längsten wartet), als nächster Zugriff auf die CPU. Daher läßt sich die Prozeßreihenfolge ganz einfach mittels einer FIFO-Warteschlange verwalten.
Gravierende Nachteile dieser Strategie liegen in der Tatsache begründet, daß "Langzeitjobs" in einem gewissen Sinne bevorzugt werden, d. h. die Werte für mittlere Wartezeiten, Antwortzeiten oder etwa die Anzahl wartender Kunden fallen bei FIFO oft schlechter aus als bei anderen Strategien.

Machen wir uns das an einem Beispiel deutlich, indem wir drei Prozesse mit Rechenzeit $P_1 = 24$, $P_2 = 3$ und $P_3 = 3$ betrachten. Nehmen wir an, daß P_2 unmittelbar nach Beendigung von P_1 beginnt (analog P_3, d. h. wir vernachlässigen evtl. auftretende Umschaltzeiten), so läßt sich der FIFO-Schedule graphisch folgendermaßen darstellen:

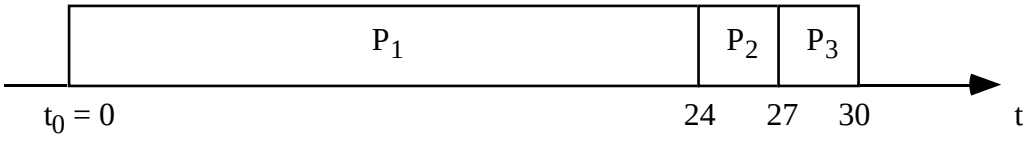


Abb. 5.1: Gantt-Chart für das FIFO-Beispiel

Eine solche graphische Darstellung wird für Scheduling-Fragen oft verwendet; man bezeichnet sie als "**Gantt-Chart**".

Die Berechnung der mittleren Wartezeit $\bar{t}$ ist nicht schwer und ergibt

$$\bar{t}_{FIFO} = \frac{0 + 24 + 27}{3} = \frac{51}{3} = 17$$

Vergleichen wir dieses Ergebnis sofort mit einer anderen, ähnlich einfachen Strategie, nämlich "**Last-In-First-Out**" (LIFO). Es ist unschwer zu erraten, daß die Vorgehensweise hierbei darin besteht, denjenigen Prozeß als ersten auf die CPU zugreifen zu lassen, der als letzter seinen Anspruch geltend gemacht hat. In diesem Fall erhalten wir folgende Gantt-Chart:

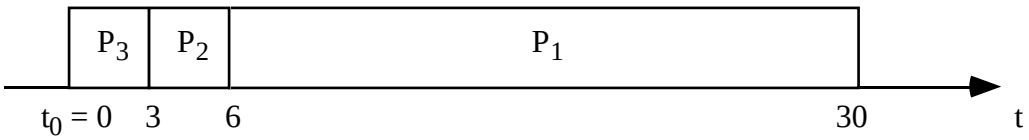


Abb. 5.2: Gantt-Chart für das LIFO-Beispiel

Kurze Rechnung ergibt

$$\bar{t}_{LIFO} = \frac{0 + 3 + 6}{3} = 3,$$

also eine deutlich kürzere mittlere Wartezeit für unser Beispiel. Im allgemeinen aber (und das heißt auf lange Sicht und für beliebige Schedules) sind die mittleren Wartezeiten von FIFO und LIFO gleich. Man kann sogar weitergehen und feststellen, daß im Schnitt die mittleren Wartezeiten aller beliebigen Strategien für beliebige Schedules gleich sind, solange wir Strategien betrachten, die ihre Auswahl nicht aufgrund der Jobdauern treffen und bei denen die Abarbeitung "**work conserving**" ist, d. h. solange das Umschalten zwischen Prozessen nur vernachlässigbar geringer Zeit bedarf.

Ein Kriterium, für das sich auch bei langfristiger allgemeiner Betrachtung ein Unterschied zwischen FIFO und LIFO feststellen läßt, ist die **Varianz** (oder Streuung) der Wartezeiten. Hierzu eine kurze Bemerkung: In der Wahrscheinlichkeitsrechnung betrachtet man oft die Ergebnisse zufälliger Messungen, die in der Regel um einen Mittelwert schwanken. Um diese Schwankung zu charakterisieren, berechnet man die durchschnittliche quadratische Abweichung der Zufallsereignisse vom Mittelwert und nennt dies die Varianz V der Zufallsgröße. Nimmt also eine Meßgröße bei verschiedenen Messungen nur Werte an, die nahe an ihrem Mittelwert liegen, so ist die Varianz klein, während eine große Varianz vorliegt, wenn die einzelnen Messungen ziemlich unterschiedliche Ergebnisse erbringen. Eine genauere Einführung findet sich in jedem Buch über Wahrscheinlichkeitsrechnung, z. B. in Fisz, S. 93.

Beispielsweise ist im obigen Beispiel

$$V_{FIFO} = \frac{1}{3}\left((0-17)^2 + (24-17)^2 + (27-17)^2\right) = 146$$

und

$$V_{LIFO} = \frac{1}{3}\left((0-3)^2 + (3-3)^2 + (6-3)^2\right) = 6$$

Allgemein kann man nun sagen, daß die Varianz der Wartezeit bei FIFO echt kleiner ist als die Varianz der Wartezeit bei LIFO. Der Unterschied verstärkt sich dabei noch mit wachsender CPU-Auslastung.

5.2.2 SJF und SRPT

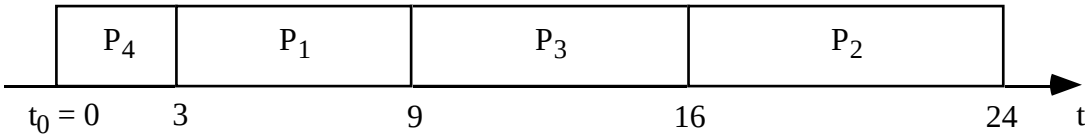
Ein allgemeines Problem bei FIFO bzw. LIFO besteht in der Benachteiligung, die "kurzlaufende" Jobs durch "Langläufer" (also Jobs, die die CPU für lange Zeit benötigen) erfahren. Im täglichen Leben trifft man auf dieses Problem beispielsweise auch an Supermarktkassen. Die pragmatische Lösung in diesem Fall bestand in der Einführung von Schnellkassen, die nur Kunden abfertigen, die maximal $10 \pm \varepsilon$ Artikel kaufen möchten.

Eine Verallgemeinerung dieser Idee führt zur Strategie "**Shortest-Job-First**" (SJF, auch "**Shortest-Processing-Time-First**" SPT genannt). Dabei muß man voraussetzen, daß die Dauer, die ein Job für seine Bearbeitung benötigt, im vorhinein bekannt ist. Unter dieser Voraussetzung bedient man dann einfach zu jedem Zeitpunkt den aktuell vorliegenden kürzesten Job.

Man kann zeigen, daß die mittlere Wartezeit bei SJF kleiner oder gleich der mittleren Wartezeit jeder anderen nichtunterbrechenden Strategie ist. Eine "nichtunterbrechende" (**non-preemptive**) Strategie läßt sich hierbei dadurch charakterisieren, daß ein Prozeß, dessen Bearbeitung einmal begonnen wurde, dann auch an einem Stück bis zu seiner Terminierung fortgeführt wird. Strategien, die die Unterbrechung eines einmal begonnenen Jobs erlauben, heißen dementsprechend "unterbrechend" bzw. "**preemptiv**". Offensichtlich ist die Menge aller nonpreemptiven Strategien eine Untermenge aller preemptiven.

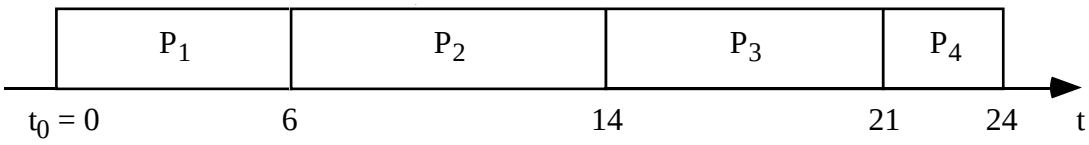
FIFO, LIFO und SJF sind allesamt nonpreemptive Strategien. Es läßt sich leicht an einem Beispiel demonstrieren, daß bezüglich der mittleren Wartezeit SJF in dieser Klasse optimal ist. Hierzu seien vier Prozesse (P_1 bis P_4) mit den Dauern P_1: 6, P_2: 8, P_3: 7 und P_4: 3 gegeben.

Shortest Job First:



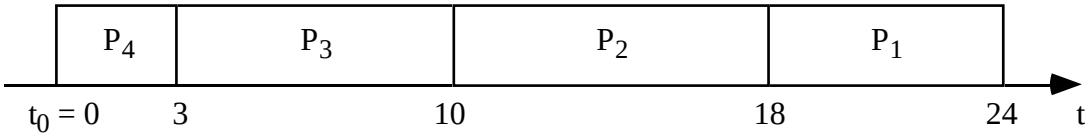
Mittlere Wartezeit: (3 + 9 + 16) / 4 = 7

FIFO:



Mittlere Wartezeit: (6 + 14 + 21) / 4 = 41/4 = 10,25

LIFO:



Mittlere Wartezeit: (3 + 10 + 18) / 4 = 31/4 = 7,75

Abb. 5.3: Die Strategien SJF, FIFO und LIFO im Vergleich

Optimal in der Klasse aller preemptiven Strategien ist eine Variante von SJF, nämlich die Strategie "**Shortest-Remaining-Processing-Time-First**" (SRPT). Sie unterscheidet sich von SJF insofern, als ein Prozeß, der sich gerade in Bearbeitung befindet, (evtl. sogar mehrmals) unterbrochen werden kann, sobald ein neuer Prozeß ankommt, dessen Bearbeitungszeit noch kürzer ist als die Restzeit, die der gerade rechnende Job noch benötigt.

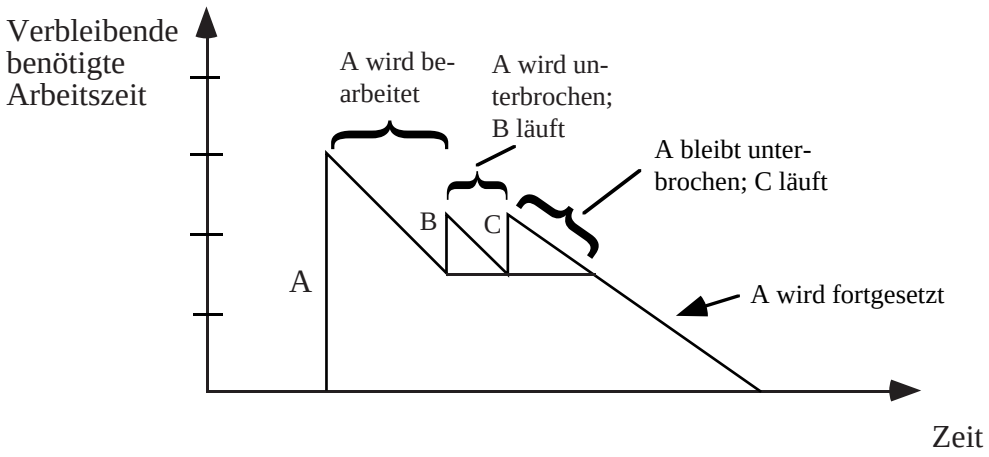


Abb. 5.4: Die SRPT-Strategie

Es läßt sich leicht zeigen, daß SRPT optimal bezüglich der mittleren Wartezeit unter allen Strategien ist, die "work conserving" sind (also vernachlässigbare Umschaltzeiten aufweisen).

Little´s Result erlaubt weiterhin die Feststellung, daß SRPT dann auch optimal bezüglich der mittleren Zahl von wartenden Prozessen ist:

$$\underbrace{\overline{N}}_{\substack{\text{mittlere Zahl von}\\ \text{(wartenden) Pro-}\\ \text{zessen im System}}} = \underbrace{\lambda}_{\substack{\text{Ankunfts-}\\ \text{rate}}} \cdot \underbrace{\overline{S}}_{\substack{\text{mittlere}\\ \text{Systemzeit}\\ \text{(Wartezeit)}}}$$

5.2.3 SEPT und SERPT

Freilich trifft man im Rahmen der Realisierung von SJF bzw. SRPT auf ein grundlegendes Problem: Woher weiß man a priori, wie lange ein Job dauern wird? Dieser Frage kann man auf zwei Arten begegnen. Erster Ansatz: Man erwartet eine Angabe der Prozeßdauer durch den Nutzer (etwa in der Art "Mein Job dauert 2 Minuten"). Der Nutzer wird sich in der Regel bemühen, eine möglichst zutreffende Angabe zu machen, denn nennt er eine zu kurze Zeit, so kann es ihm passieren, daß sein Job vor Terminierung abgewürgt wird. Ist die Zeit, die er nennt, zu lang, so beginnt die Abarbeitung seines Jobs u. U. unnötig spät (da es kürzere gibt, deren Bearbeitung vorgezogen wird).

Ein anderer Ansatz versucht, die zu erwartende Prozeßdauer aus Erfahrungen der Vergangenheit möglichst zutreffend zu schätzen. Hierbei nimmt man an, daß die einzelnen Jobs von verschiedenen Kundenquellen kreiert werden. Weiß man nun, wie lange in der Vergangenheit Jobs der verschiedenen Quellen gedauert haben, so kann man daraus mehr oder weniger gut auf das Verhalten in der Zukunft schließen. Dies führt zu Strategien wie SEPT ("**Shortest-Expected-Processing-Time-First**") bzw. SERPT ("**Shortest-Expected-Remaining-Processing-Time-First**").

Einfache Schätzverfahren verwenden hierbei etwa den Mittelwert aus zurückliegenden "Fenstern" (also z. B. den Durchschnitt der letzten n Jobs der betreffenden Quelle) oder beispielsweise 80 % des Maximalwerts der letzten n Jobs o. ä.

Eine andere Idee, um adaptiv aus der Vergangenheit zu "lernen", verwendet das "**exponential averaging**". Sei hierzu τ_n die Schätzung für den n-ten Job einer bestimmten Kundenquelle, sowie t_n die tatsächliche Dauer dieses Jobs. Dann erhält man eine Schätzung für den (n+1)-ten Job nach der Formel

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$$

Der Parameter α liegt dabei zwischen 0 und 1 und beeinflußt die Art des Lernens aus der Vergangenheit. Dies sieht man am besten, wenn man die beiden Extremfälle betrachtet:

- $\alpha = 0$: $\tau_{n+1} = \tau_n = ... = \tau_0$: Die Schätzung bleibt immer dieselbe, Lernen findet nicht statt ("stures Verhalten").
- $\alpha = 1$: $\tau_{n+1} = t_n$: hier wird jeweils nur der allerletzte Zustand für die Schätzung herangezogen, die Vergangenheit davor interessiert nicht ("völlig hektisches Verhalten").

Bessere Kompromisse verwenden ein α echt zwischen 0 und 1. Dies bewirkt letztlich ein "exponentielles Abklingen" der Vergangenheit. Führt man die Rekursion nämlich explizit aus, so erhält man schließlich

$$
\begin{aligned}
\tau_{n+1} &= \alpha \cdot t_n + (1-\alpha) \cdot \tau_n \\
&= \alpha \cdot t_n + (1-\alpha) \cdot \left(\alpha \cdot t_{n-1} + (1-\alpha) \cdot \tau_{n-1}\right) \\
&= \alpha \cdot t_n + (1-\alpha) \cdot \alpha \cdot t_{n-1} + (1-\alpha)^2 \cdot \tau_{n-1} \\
&= \ldots \\
&= \alpha \cdot t_n + (1-\alpha) \cdot \alpha \cdot t_{n-1} + \ldots + (1-\alpha)^n \cdot t_0 + (1-\alpha)^{n+1} \cdot \tau_0
\end{aligned}
$$

Je größer dabei α ist, desto schneller "vergißt" man die Vergangenheit. Wählt man also α zu groß, so wirken sich kurzfristige Schwankungen sehr stark aus, und es findet kaum eine "Glättung" statt. Ist α andererseits zu klein, so erfolgt die Korrektur bei wirklichen Trendveränderungen möglicherweise viel zu langsam.

Exkurs: Digitale Sprachübertragung

Exponential Averaging bei Scheduling-Strategien versucht, der Kurve der tatsächlich benötigten CPU-Zeit von Prozessen eines Kunden zu folgen. Ganz ähnlich muß beim **Digitalisieren von Sprachsignalen** der **Frequenzkurve** mit **Abtastwerten** gefolgt werden.

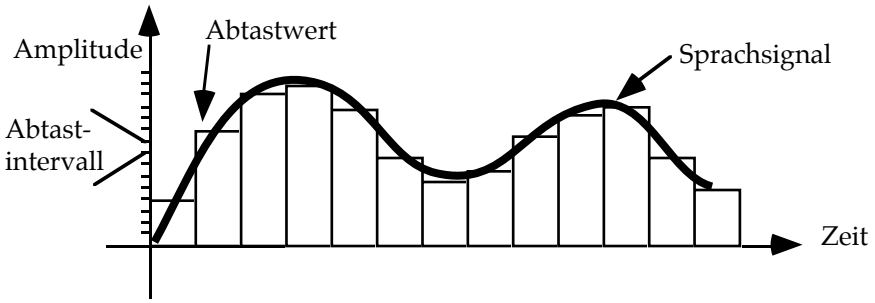


Abb. 5.5: Digitalisieren von Sprachsignalen

Wie oft ein Signal abgetastet werden muß, geht aus dem Nyquist-Theorem hervor:

Nyquist-Theorem: Vorausgesetzt, daß ein Signalwert exakt digitalisiert werden kann, läßt sich die Originalkurve aus den digitalen Signalen genau dann wiederherstellen, wenn die **Abtastrate (Abtastfrequenz)** mindestens doppelt so hoch ist wie die maximale im Signal vorkommende Frequenz.

Bei Telefonen liegt die (Audio-)Signalfrequenz üblicherweise zwischen 300 und 3400 Hertz. Eine Abtastfrequenz von ungefähr 6,8 kHz wäre demnach bei exakter Abtastung ausreichend.

PCM (Pulse Code Modulation), ein bekanntes Verfahren zur Sprachkodierung, tastet mit einer Frequenz von 8 kHz ab und kodiert jeden Abtastwert mit 8 Bit (nicht exakt, jedoch für das menschliche Ohr ausreichend). Es müssen also alle 125 µs 8 Bit übertragen werden, was einer Übertragungsrate von 64 kBit/s entspricht (man beachte, daß z. B. ein ISDN-B-Kanal genau 64 kBit/s Kapazität aufweist).

Um diese doch recht hohe Datenrate zu senken, können z. B. die Abtastintervalle vergrößert werden (vgl. Abbildung 5.5). Eine andere Möglichkeit besteht in der sogenannten **Delta-Modulation**. Dabei wird nicht jeder Abtastwert, sondern nur der

Startwert und danach jeweils die Änderung (**Delta**) zum vorherigen Wert übertragen (solche Strategien finden übrigens auch bei Videokompressions-Verfahren wie z. B. MPEG Anwendung). Der Signalverlauf wird also beim Sender und Empfänger "nachgeführt".

Um dieses Verfahren anwenden zu können, müssen zwei Dinge festgelegt werden:

- Die "Größe der Treppenstufe", d. h. die Änderung, die ±Delta verursacht, und
- die Anzahl von Bits, mit denen Delta kodiert wird, d. h. wieviele "Treppenstufen" auf einmal "genommen" werden können.

Beispielsweise funktioniert dies bei Kodierung mit einem Bit wie folgt: Übertrage entweder 0 (alter Wert + Delta) oder 1 (alter Wert - Delta), je nachdem welcher dieser Werte näher am aktuellen Abtastwert liegt. Die Einstellung der Treppenstufe ist jedoch schwierig. Ein kleines Delta erkennt zwar kleine Amplitudenunterschiede, kann jedoch nicht schnell auf große Schwankungen reagieren (entspricht analog einem kleinen α bei Exponential Averaging). Bei einem großen Delta-Wert (großes α bei Exponential Averaging) ist dies genau umgekehrt. Dazu einige Beispiele:

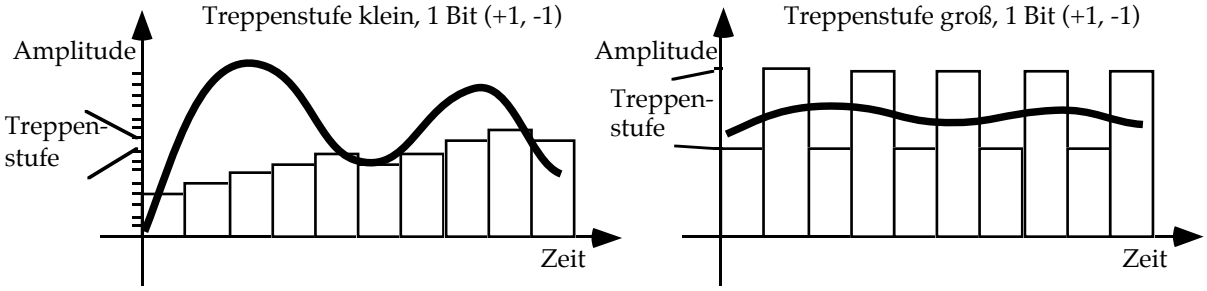


Abb. 5.6: 1-Bit-Delta

Offensichtlich ist ein "1-Bit-Delta" sehr häufig nicht in der Lage, dem Kurvenverlauf zu folgen. Alternativ können 2-, 3- oder 4-Bit-Deltas verwendet werden.

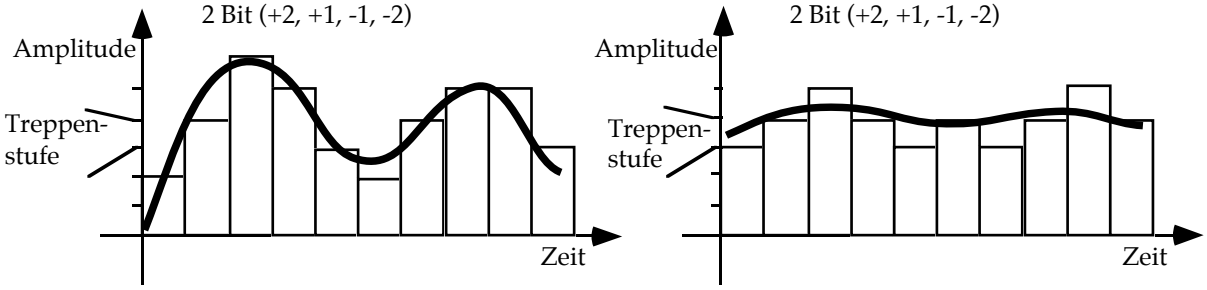


Abb. 5.7: 2-Bit-Delta

Solche Kodierungen erhöhen andererseits auch wieder die Datenrate.

5.2.4 Priority Scheduling

Beim Priority-Scheduling wird jedem Job eine Priorität zugeordnet, die Abarbeitung erfolgt dann nach dem Schema "**Highest-Priority-First**" (HPF). Eine verfeinerte Variante ordnet jeden Job in eine von mehreren Prioritätsklassen ein. Vorrang hat dann jeweils die höchste nichtleere Prioritätsklasse, wobei innerhalb der Klasse dann z. B. nach FIFO verfahren wird. Dieses Schema kann sowohl nicht-preemptiv als auch

preemptiv implementiert werden. In letzterem Fall wird ein Job mit niedriger Priorität unterbrochen, sobald ein Job höherer Priorität ankommt.

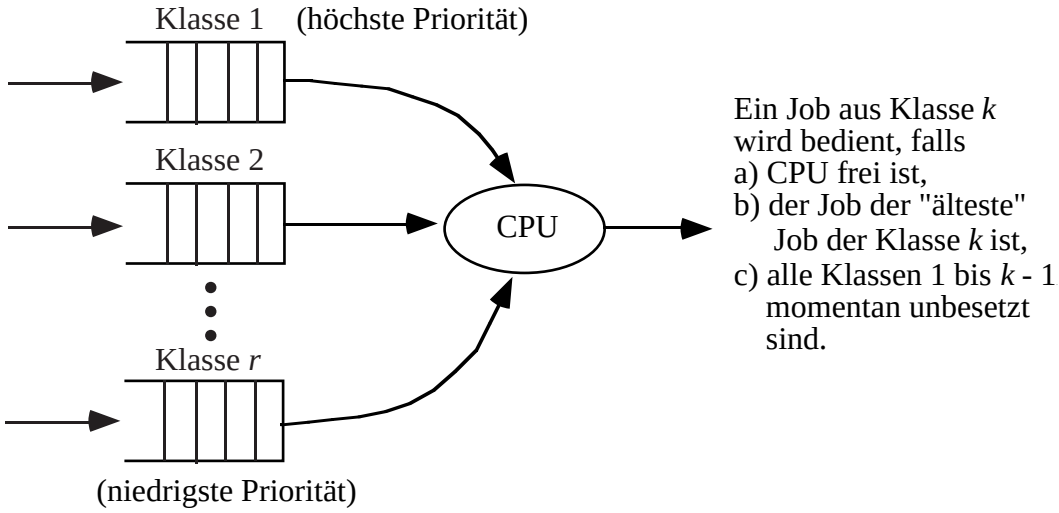


Abb. 5.8: Priority Scheduling

Dieses Vorgehen ist natürlich insofern problematisch, als die höheren Klassen ohne weiteres die niedrigeren monopolisieren können, wobei dann im Extremfall ein Job, der sich in der niedrigsten Klasse befindet, ewig bis zu seiner Abarbeitung warten muß.

5.2.5 RR und PS

Eine weitere Scheduling-Variante ist das **Round-Robin-Verfahren** (RR). Hierbei wird in gewisser Weise ein Ausgleich zwischen "Langläufern" und "Kurzläufern" unter den Jobs versucht, denn die Grundidee beim Round Robin versucht die Antwortzeit eines Jobs näherungsweise proportional zu seiner Dauer zu gestalten.

Hierzu vergibt der Scheduler der Reihe nach an jeden Prozeß ein **Quantum** Q (auch Zeitscheibe genannt). Wird ein Prozeß innerhalb von Q fertig, dann ist alles wunderbar. Schafft er dies nicht, so unterbricht er seine Arbeit und reiht sich ganz normal an die letzte Stelle einer FIFO-Schlange ein, und der nächste Prozeß erhält nun seine Zeitscheibe Q.

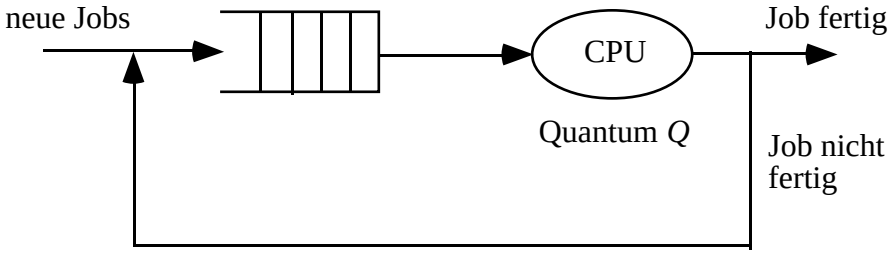


Abb. 5.6: Round-Robin

Für den Extremfall $Q \to 0$ (also für sehr kleine Quanten) und wenn Jobunterbrechungen nichts "kosten", erhält man daraus näherungsweise die Strategie "**Processor-Sharing**" (PS). Hierbei erhält dann jeder der n aktuell im System befindlichen Prozessoren genau 1/n der CPU-Leistung. Natürlich ist zu bedenken, daß in der Praxis bei zu kleinen Quanten der Verwaltungsoverhead schnell zu groß wird. Betrachtet man auf der anderen Seite den Fall $Q \to \infty$ (also sehr große Quanten), so geht das Schema in das ganz gewöhnliche FIFO-Scheduling über.

5.2.6 Multilevel Feedback Queueing

Ein letztes Scheduling-Verfahren, das hier vorgestellt werden soll, ist das "**Multilevel-Feedback-Queueing**". Diese Strategie arbeitet sowohl mit Prioritätsklassen ("Multilevel") als auch mit Zeitscheiben.
Die Idee ist dabei folgende: Ein neu ankommender Job wird in die höchste Prioritätsklasse eingereiht. Die Abarbeitung erfolgt dann mit einer Round-Robin-Strategie, wobei die einzelnen Prioritätsklassen unterschiedlich große Quanten erhalten: Klassen hoher Priorität haben kleine Quanten, Klassen niedrigerer Priorität entsprechend größere. War ein Job in seiner Prioritätsklasse dran, wurde allerdings nicht innerhalb der entsprechenden Zeitscheibe fertig, so wandert er in die nächstniedrigere Klasse.

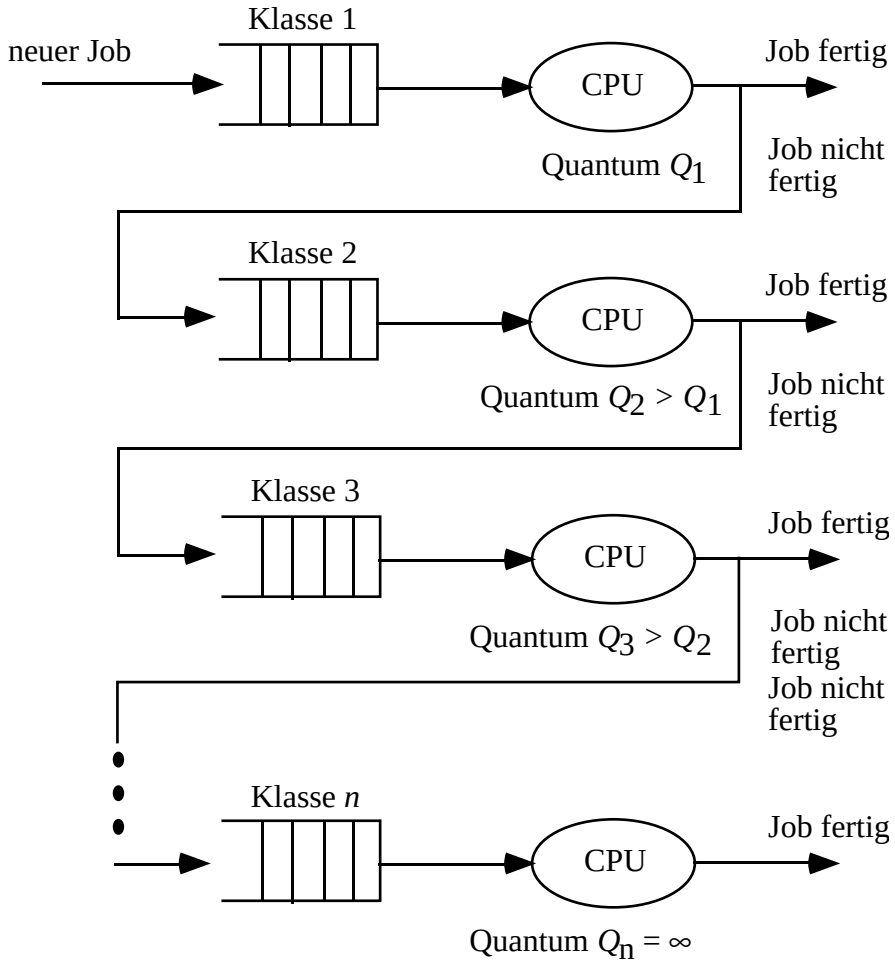


Abb. 5.7: Multilevel-Feedback-Queueing

Bedient wird bei freier CPU stets der älteste Job der höchsten Klasse. Dies hat zur Folge, daß extrem kurze Jobs immer sofort erledigt werden können. Längere Jobs wandern in die nächste Klasse und können dadurch von Kurzläufern überholt werden. Die Strategie nimmt natürlich in Kauf, daß sehr lange Jobs u. U. auch sehr, sehr lange im System bleiben, bis sie endlich abgearbeitet werden können. Im Hochlastfall kann es entsprechend wiederum zu einer Monopolisierung des Rechensystems durch Kurzläufer kommen.

Speicherverwaltung

6 Hauptspeicherverwaltung

In den vorangegangenen Abschnitten wurde davon ausgegangen, daß die CPU durch verschiedene Prozesse gleichzeitig genutzt werden kann. Durch diese Eigenschaft wird zum einen die Leistung des Rechners bzw. Systems erhöht, zum anderen die Antwort- bzw. Bearbeitungszeit von Prozessen gesenkt. Dieser Vorteil hat jedoch auch seinen Preis - und zwar müssen verschiedene Prozesse im Speicher verwaltet werden, d. h. der Speicher muß gemeinsam genutzt werden.

6.1 Speicherorganisation

Ein Prozeßsystem enthält viele gleichzeitig aktive Prozesse, die alle unterschiedliche Anforderungen, insbesondere unterschiedlichen Speicherbedarf, haben. Jeder Prozeß hat den Wunsch, schnell bedient zu werden, und das beinhaltet unter anderem den schnellen Speicherzugriff.

Die Probleme bei der Realisierung eines schnellen Speicherzugriffs für viele Prozesse liegen darin, daß

- schnelle Speicher sehr teuer sind (viel teurer als langsame Hintergrundspeicher wie z. B. Magnetbandgeräte),
- die Speicherausnutzung schlecht ist, wenn die Programme vollständig in schnellen kleinen Speichern gehalten werden, und
- die Speicherstruktur für den Programmierer transparent sein soll.

Lösen lassen sich diese Probleme durch eine Speicherhierarchie in Kombination mit dem Konzept des "virtuellen Speichers".

6.1.1 Speicherhierarchie

Im einfachsten Fall sind zwei Hierarchieebenen vorhanden, nämlich der Hauptspeicher und der Hintergrundspeicher. Der **Hauptspeicher** ist von der Kapazität her begrenzt, wohingegen der **Hintergrundspeicher** "beliebig" groß ist. Schaltet man zwischen Hauptspeicher und CPU noch einen schnellen Pufferspeicher (**Cache**), so ergibt sich folgendes Bild:

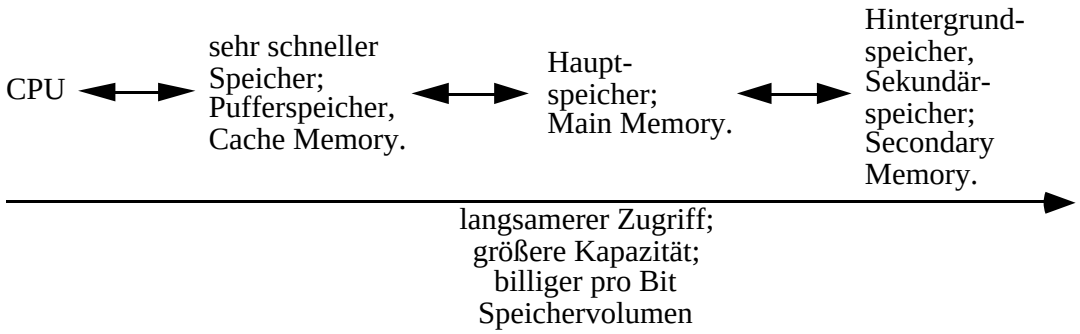


Abb. 6.1: Hierarchie-Level (HL) eines Speichers

Bei einer solchen hierarchischen Organisation werden aktuell benötigte Daten direkt verfügbar gehalten, die anderen Daten werden in den Hintergrund verdrängt und nur bei Bedarf "nach vorne" geholt.

6.1.2 Virtueller Speicher

Das Prinzip des virtuellen Speichers besteht darin, nur Teile des Programms im Hauptspeicher zu halten und den (beliebig großen) Rest im Hintergrundspeicher zu verwalten. Es ist also "virtuell" mehr Hauptspeicher vorhanden als reell. Man benötigt für solch einen virtuellen Speicher drei Bestandteile:

a) dem physikalischen Adreßraum eines Programms,
b) dem logischen Adreßraum eines Programms und
c) einem Tauschmechanismus zwischen a) und b).

In älteren Computersystemen organisierte der Programmierer die Zuteilung (Laden, Verdrängen) von Speicherblöcken selbst. Heute ist dies eine Aufgabe, die vom Betriebssystem automatisch übernommen wird.

Probleme der Speicherzuteilung treten an unterschiedlichsten Stellen auf. Zunächst ist es wichtig zu wissen, welche Speichereinheiten zu ersetzen sind. Dies können feste Einheiten wie zum Beispiel eine Seite sein, variable Größen wie Segmente und auch Mischformen von beidem. Hinsichtlich der Speicherkapazität, die im Hauptspeicher pro Programm reserviert werden soll, kann sowohl eine feste als auch eine variable Zuteilung vorgenommen werden. Weitere Probleme bestehen darin, wann Seiten oder Segmente nachgeladen werden sollen, wohin das Nachladen erfolgt und auch, was dafür ersetzt werden soll.

Die Beschäftigung mit diesen Fragen führt uns zu den Konzepten von **Segmentierung** und **Paging** (d. h. Seitenersetzung). Diese beiden Möglichkeiten können auch kombiniert werden, was in der Praxis jedoch keine große Bedeutung besitzt. Eine weitere Möglichkeit der Speicherverwaltung sind Buddy-Systeme, die im weiteren Sinne einen Spezialfall der Segmentierung darstellen.

6.2 Segmentierung

Bei Fragen der Speicherverwaltung ist immer im Auge zu behalten, daß die logische Nutzersicht auf den Speicher stets eine andere ist, als die momentane Struktur des physikalischen Speichers. Aus diesem Grund sollte die oft komplexe Struktur des physikalischen Speichers vor dem Benutzer verborgen werden können ("Transparenz"). Das Betriebssystem überträgt dann die Nutzersicht auf die physikalische Speicherbelegung, d. h. es muß eine Abbildung zwischen logischem und physikalischem Speicher erfolgen. "Segmentierung" ist ein Speicherverwaltungsschema, das versucht, die Nutzersicht möglichst direkt auf den Speicher zu übertragen.

6.2.1 Das Prinzip der Segmentierung

Der logische Adreßraum ist in diesem Zusammenhang eine Sammlung von Segmenten, wobei wir unter einem **Segment** eine als logische Einheit betrachtete Informationsmenge verstehen, die aus einem oder mehreren Unterprogrammen oder aus Daten zu einem Programm bestehen kann.

Jedes Segment besitzt einen Namen und eine Länge. Adressen bestehen aus dem **Segmentnamen** und dem sogenannten **Offset** innerhalb des Segments (Abstand zwischen dem Segmentanfang und einer beliebigen Stelle im Segment). Der Einfachheit halber können Segmente auch durchnumeriert werden. Dieses Verfahren macht die Benennung durch Namen überflüssig. In einem solchen Fall besteht eine logische Adresse aus dem geordneten Paar (Segment-Nummer, Offset).

Für jeden Prozeß lädt das Betriebssystem die Segmente, die für das Fortschreiten benötigt werden, in den Hauptspeicher. Falls nicht alle Segmente der einzelnen Prozesse gleichzeitig im Hauptspeicher gehalten werden können, wird ggf. ein Nachladen bzw. Verdrängen von Segmenten erforderlich. Als Konsequenz erhält man im Hauptspeicher Belegtbereiche (mit den Segmenten) und Lücken. Benachbarte Lücken werden zu einer großen vereinigt. Das Betriebssystem muß eine Liste der Segmente und Lücken mit ihren jeweiligen Größen verwalten.

6.2.2 Segmentierungsstrategien

Wo werden nun neue Segmente im Speicher abgelegt? Zuerst wird ein Lücke gesucht, die mindestens so groß wie das Segment ist. Dort wird das Segment dann linksbündig plaziert. Bezüglich der Lückenauswahl gibt es unterschiedliche Strategien:

- **First Fit (FF):**
  Plaziere das Segment in die erste passende Lücke.
- **Best Fit (BF):**
  Plaziere das Segment in die kleinste passende Lücke.
- **Worst Fit (WF):**
  Plaziere das Segment in die größte passende (Lücke.
- **Rotating First Fit (RFF):**
  Wie First Fit, jedoch wird von der Position der vorherigen Plazierung (hier: Ende der benutzten Lücke) ausgehend die nächste passende Lücke gesucht.

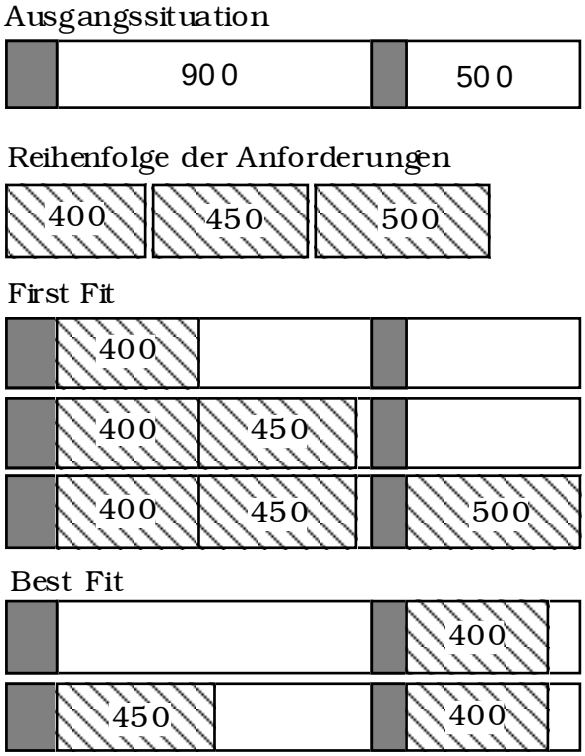
Der Nachteil von First Fit ist der hohe Verschnitt durch die Bildung vieler kleiner Lücken am Anfang des Speichers. Bei Best Fit wirkt sich nachteilig aus, daß in manchen Fällen der verbleibende Rest einer Plazierung extrem klein und später praktisch nicht mehr verwendbar ist. Durch Simulationen konnte gezeigt werden,

daß Rotating First Fit in der Regel am besten abschneidet. Allgemein gilt (unter Definition von ">" als "besser als") die folgende Relation:

RFF > FF > BF > WF

Man kann jedoch für alle Verfahren Szenarien erstellen, in denen jede der Strategien im Einzelfall optimal oder besonders schlecht ist. Dies soll an den folgenden Beispielen - ohne Berücksichtigung der Strategie Worst Fit - erläutert werden.

Beispiel zur Unbrauchbarkeit von Best Fit
Als Ausgangssituation stehen zwei Lücken der Größen 900 und 500 zur Verfügung. Es sollen Segmente der Größen 400, 450 und 500 plaziert werden. Insgesamt ist also genügend Speicherplatz vorhanden, um die Anforderungen zu erfüllen. Jedoch ermöglichen nur die Verfahren First Fit bzw. Rotating First Fit (von vorne begonnen) eine Plazierung der Anforderungen.



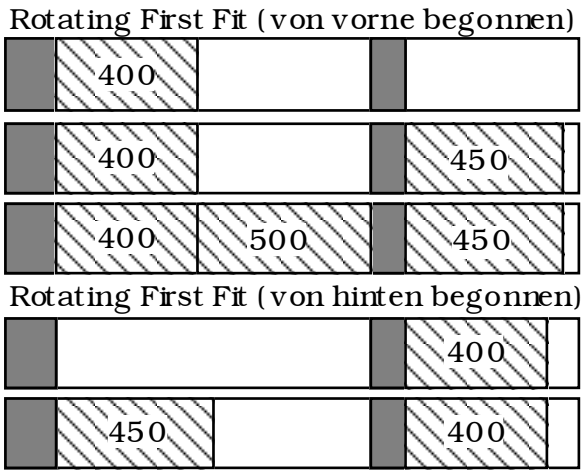


Abb. 6.2: Beispiel 1 zur Segmentierung

Abbildung 6.2 stellt diesen Sachverhalt noch einmal graphisch dar. Während First Fit und Rotating First Fit (von vorne begonnen) die Möglichkeit bieten, alle drei Segmente zu speichern, ist dies bei Best Fit und Rotating First Fit (von der letzten Lücke aus begonnen) nicht der Fall.

Beispiel zur Unbrauchbarkeit von First Fit und Rotating First Fit
In diesem Fall stehen als Ausgangssituation Speicherbereiche der Größe 700, 500 und 800 zur Verfügung. Es treten Anforderungen der Größen 400, 600 und 800 auf. Jedoch kann lediglich das Best Fit-Verfahren die Anforderungen plazieren. In diesem Fall erweist sich also Best Fit als die bessere Alternative.

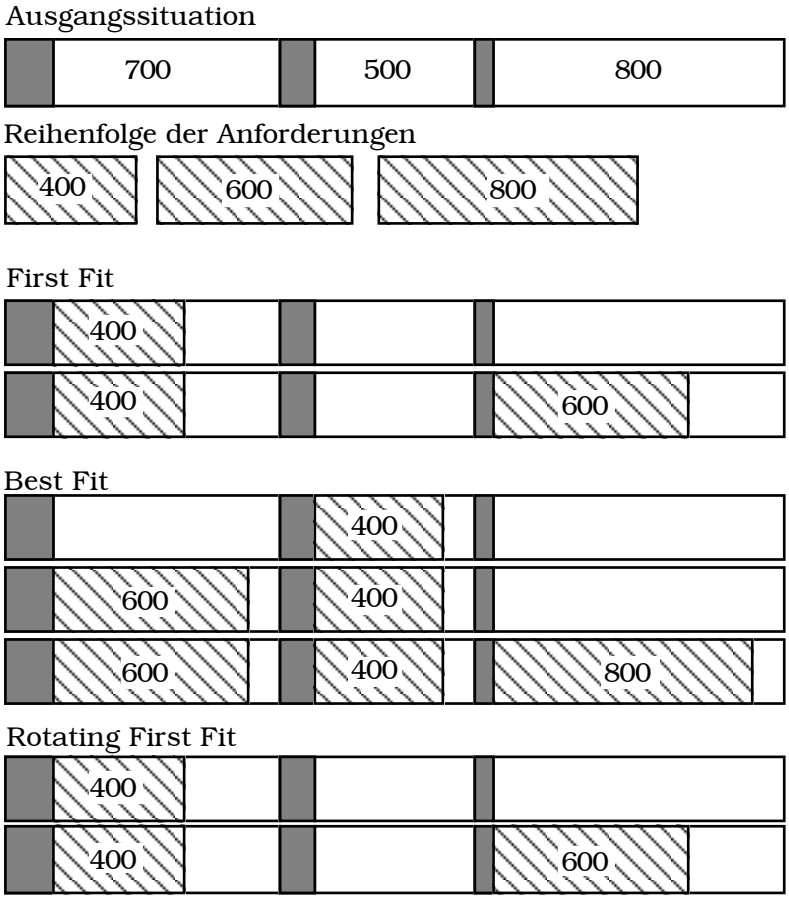


Abb. 6.3: Beispiel 2 zur Segmentierung

First Fit und auch Rotating First Fit führen beide zu einem ungenügenden Ergebnis, da bei ausreichendem Speicherplatz jeweils Teile von Segmenten verschenkt werden.

Im Zusammenhang mit der Segmentierung ergibt sich die Frage, wie weit man einen segmentierten Speicher füllen darf. Ein zu niedriger Füllgrad bewirkt zwar ein einfaches Plazieren, jedoch ist der Speicher schlecht ausgelastet. Hingegen gewährleistet ein hoher Füllgrad eine gute Speicherauslastung, nur ist aufgrund der vielen kleineren Lücken ("externe Fragmentierung") und dem damit verbundenen Suchaufwand die Plazierung von neuen Segmenten sehr aufwendig.

Eine Alternative besteht darin, nachträglich Daten im Speicher umzuordnen ("**Garbage Collection**"). Durch die Segmentierung kann die Situation eintreten, daß die größte Lücke nicht mehr ausreicht, um ein Segment zu speichern, jedoch der Platz in der Summe der Lücken vorhanden ist. Dann kann entweder das Segment auf mehrere Lücken verteilt werden oder man schiebt die Segmente zusammen. Muß allerdings häufiger zusammengeschoben werden, kann man davon ausgehen, daß der Speicher überlastet ist. Dann ist es besser, einen Teil der Prozesse abzubrechen.

6.3 Buddy-Systeme

Eine andere Speicheraufteilung sind die sogenannten Buddy-Systeme. Die Idee der **Buddy-Systeme** - Buddy kann sinngemäß mit "Kumpel" übersetzt werden - ist es, eine Organisationsform bereitzustellen, welche i. a. weniger Verschnitt bzgl. der Lückenstruktur als Segmente hat und auch weniger starr ist als z. B. das Paging-Konzept.

6.3.1 Einfache Buddy-Systeme

Bei einem Buddy-System läßt man nur Anforderungen in Größe einer 2er-Potenz zu. Sei die Gesamtgröße des physikalischen Speichers 2^{max}. Bei einer Anforderung teilt man den Speicher solange in kleinere 2er-Potenzen auf, bis die gewünschte Größe verfügbar ist. Für jede 2er-Potenz gibt es eine **Liste** L_{pot}, welche die momentan freien Speicherbereiche bzgl. der Größe 2^{pot} angibt.

Die beiden wesentlichen Aufgaben des Buddy-Systems sind die Plazierung von Segmenten sowie die Freigabe von nicht mehr benötigtem Speicherplatz. Bei der Plazierung eines Segments der Größe 2^{pot} wird in der entsprechenden Liste L_{pot} geprüft, ob dort freier Speicherplatz vorhanden ist. Folgende Fälle sind zu unterscheiden:

$L_{pot} \neq ø$: Plaziere Segment und lösche entsprechenden Speicherplatz aus der Liste.
$L_{pot} = ø$: Überprüfe die nächsthöhere Liste L_{pot+1}, dann L_{pot+2} usw. so lange, bis eine nichtleere Liste gefunden wurde. Zerlege einen dieser Listen-Kandidaten solange in zwei Hälften (Buddies), bis eine passende Buddy-Größe entsteht. Dort wird dann das Segment abgelegt. Sind allerdings alle Listen L_{max} leer, so ist eine Plazierung des Segments unmöglich.

Abbildung 6.4 stellt ein Beispiel für Buddy-Systeme dar. Sei max = 18. Es werde eine Anforderung der Größe 2^{16} betrachtet.

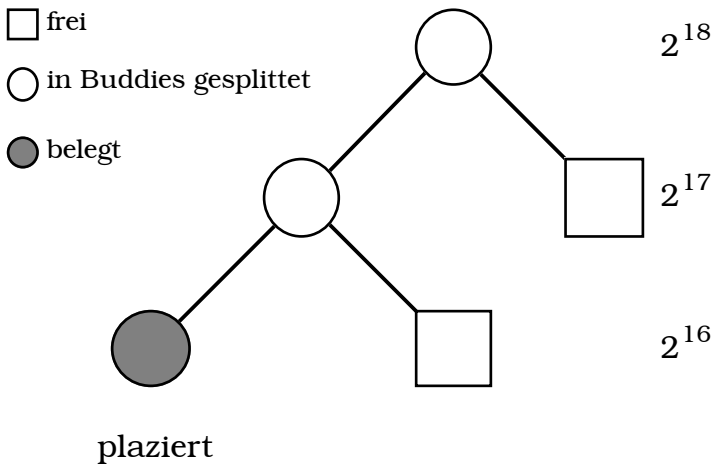


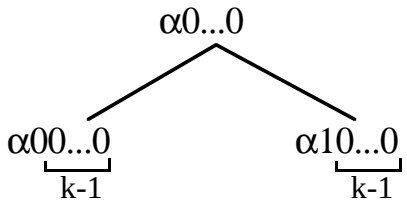
Abb. 6.4: Beispiel eines Buddy-Systems

Bei der Freigabe eines Segments der Größe 2^{pot} ist nachzusehen, ob der zugehörige Buddy (mit gemeinsamer Wurzel) ebenfalls frei ist. Falls ja, so vereinige beide wieder zu einem Bereich der Größe 2^{pot+1} und iteriere dies solange, bis keine Buddies mehr vereinigt werden können.

Die Anfangsadressen der jeweiligen Buddies bzw. die Adresse des wiedervereinigten Blocks berechnen sich äußerst einfach. Angenommen, es gäbe einen Speicherbereich $[0:2^{max}-1]$ (binär [0…0:1…1]). Ein Speicherbereich wird durch ein 2-Tupel gekennzeichnet, bestehend aus Anfangsadresse und Länge. Ein Speicherbereich der Größe 2^k hat folgendes Aussehen: [α0…0 : α1…1], wobei die Anzahl der Nullen bzw. Einsen k beträgt. Die **Vorsilbe** α charakterisiert den Speicherbereich.

$$\underbrace{\alpha 0 \ldots 0}_{k}$$

Bei Zerlegung entstehen folgende Buddies:



Beide neuentstandenen Buddies unterscheiden sich nur in einem Bit. Das Verfahren erzeugt eine baumartige Struktur (die Zahlen an den Knoten entsprechen jeweils der Vorsilbe α):

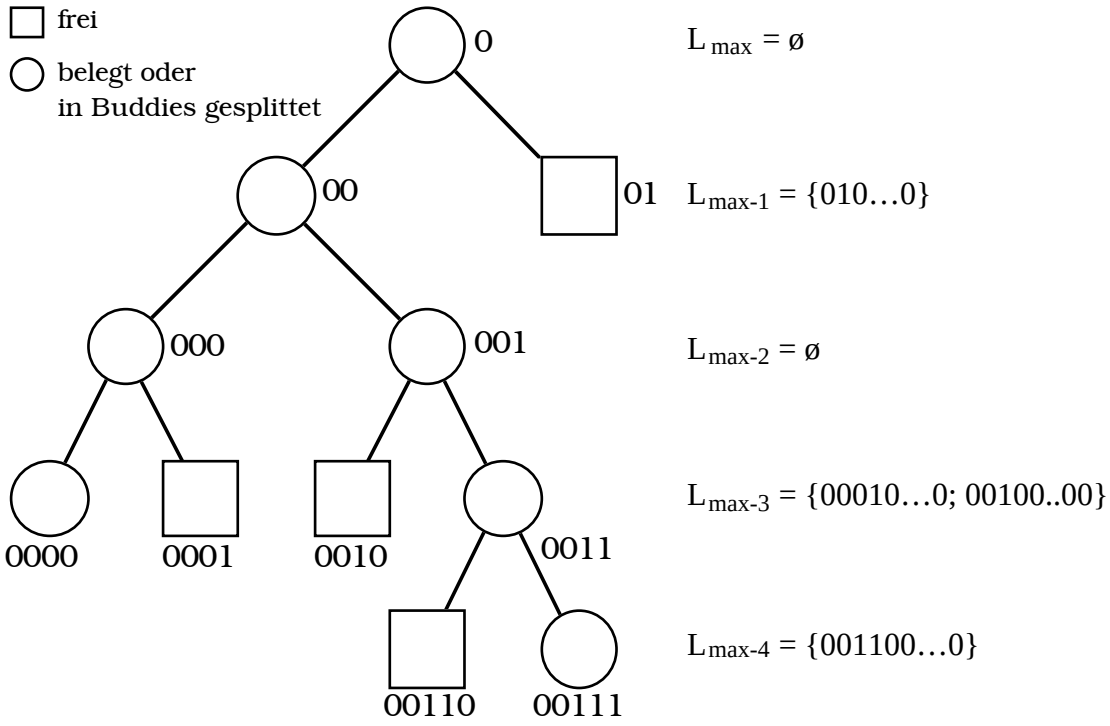


Abb. 6.5: Listenangabe in einem Buddy-System

Es kann vorkommen, daß benachbarte Speicherbereiche frei, jedoch keine Buddies sind. Dann lassen sich diese Bereiche nicht zu einem großen zusammenfassen.

6.3.2 Gewichtete Buddy-Systeme

Um eine feinere Aufteilung des Speichers zu erreichen, können **gewichtete Buddies** verwendet werden. Dabei ist ein Block der Größe 2^{r+2} z. B. im Verhältnis 1:3 in Blöcke der Größe 2^r bzw. $3 \cdot 2^r$ zu zerlegen. Der Buddy der Größe $3 \cdot 2^r$ wird im Verhältnis 2:1 in Buddies der Größen 2^{r+1} und 2^r zerlegt. Der Vorteil des Verfahrens liegt in der mehrstufigen Aufteilung der einzelnen Buddy-Größen. Nachteilig ist der erhöhte Aufwand, insbesondere ist die Adreßberechnung wesentlich komplizierter.

Beispielhaft wird in Abbildung 6.6 die Aufteilung eines Speicherbereichs der Größe 2^4 mit gewichteten Buddies gezeigt.

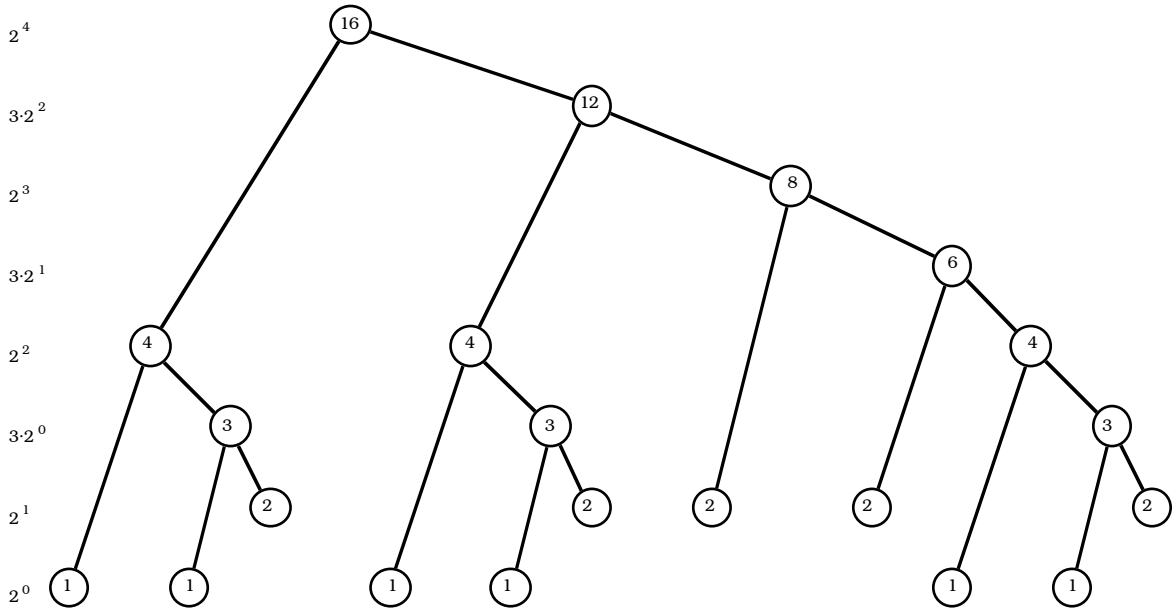


Abb. 6.6: Speicherbereich mit gewichteten Buddies

Die Vorgehensweise bei der Zuweisung von Blöcken in gewichteten Buddy-Systemen ist in Abbildung 6.7 graphisch dargestellt. Der Algorithmus gibt an, wie die Zuweisung eines Blocks der Größe u mit Hilfe eines gewichteten Buddy-Systems vorgenommen wird. Sei dabei $u \in \{2^0,2^1,\ldots,2^m\} \cup \{3\cdot2^0,\ldots,3\cdot2^{m-2}\}$, und seien L_1, L_2, ..., L_{2m} die Listen der zulässigen Blockgrößen, wobei einer hohen Blockgröße ein hoher Listenindex entspricht.

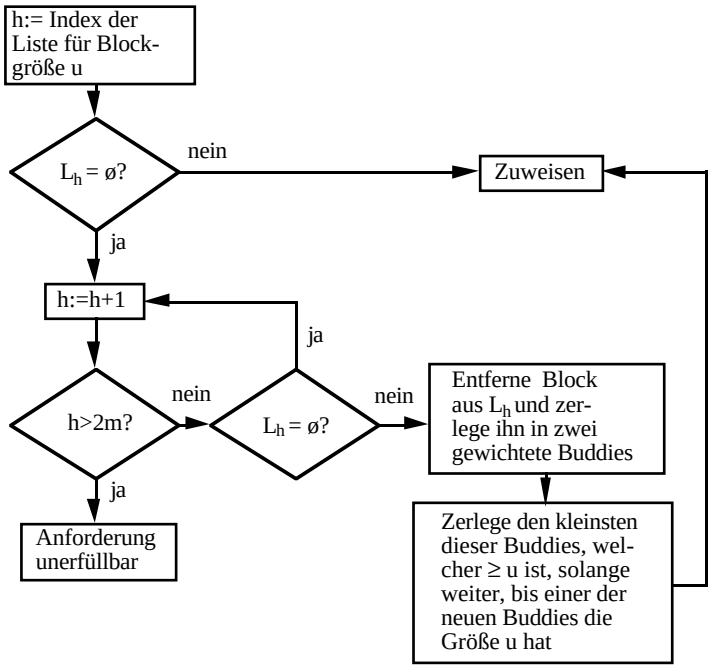


Abb. 6.7: Die Funktionsweise von gewichteten Buddy-Systemen

6.4 Demand-Paging

6.4.1 Prinzip des Pagings

Paging hat unter den Speicherverwaltungsmethoden heute einen so hohen Stellenwert, daß wir ihm ein eigenes Kapitel widmen wollen. Im Gegensatz zu Segmentierung und Buddysystemen werden beim **Paging** ausschließlich Speicherblöcke einheitlicher Größe verwendet; man bezeichnet sie als "Seiten" oder "**Pages**". Dies bedeutet zunächst, daß der logische Adreßraum von Programmen, die auf einem Hintergrundspeicher liegen, in Seiten unterteilt wird. Analog dazu erfolgt eine Aufteilung des physikalischen Adreßraums (Hauptspeicher) in "Rahmen" ("**Frames**"), die jeweils genau eine Seite aufnehmen können. Zur Ausführung des Programms werden dann alle (oder auch nur einige der) im Hintergrund gespeicherten Seiten in die Rahmen des Hauptspeichers geladen.

Der Vorteil dieses Verfahren liegt in der Vermeidung der Speicherzerstückelung (**externe Fragmentierung**). Als Nachteil muß man hingegen einen eventuellen Verschnitt innerhalb einer Seite hinnehmen, da die Größe eines Programmoduls im allgemeinen kein ganzzahliges Vielfaches einer Seite ist (**interne Fragmentierung**).

Es stellt sich die Frage, wann eine Seite aus dem Hintergrund in den Hauptspeicher gebracht werden soll. Die folgenden Strategien kommen in Betracht:

- **Demand-Paging**:
  Fehlt im Hauptspeicher eine Seite (liegt also ein "**Seitenfehler**" vor), so wird diese nachgeladen. Als Problem verbleibt, welche Seite dafür verdrängt werden soll (Replacement Strategy).

- **Demand-Prepaging**:
  Arbeitet wie Demand-Paging, jedoch werden im Fall eines Seitenfehlers gleich mehrere Seiten geladen und verdrängt.

- **Look Ahead**:
  Nachladeoperationen werden bei dieser Strategie auch durchgeführt, ohne daß unbedingt ein Seitenfehler vorliegt.

Bezüglich der Anzahl der Seitenfehler ist Demand-Paging optimal, da diese Strategie am ökonomischsten mit Seitentransporten umgeht. Hat man jedoch andere Zielgrößen, wie zum Beispiel Kosten (12 Seiten auf einmal nachzuladen, kann billiger sein, als das Nachladen von 12 Seiten hintereinander), so kann je nach den vorliegenden Umständen auch Demand-Prepaging oder Look Ahead optimal sein.

6.4.2 Demand-Paging-Strategien

Im Laufe der Abarbeitung eines Programms werden immer wieder Zugriffe auf den Hauptspeicher notwendig. Die Frage, die nun behandelt werden soll, lautet, wie man (bei bekanntem Programmablauf) eine besonders hohe Trefferquote ("**hit ratio**") bzgl. der im Hauptspeicher vorhandenen Seiten erreichen kann. Anders ausgedrückt geht es darum, beim Nachladen jeweils die "unwichtigsten" Seiten zu verdrängen.

Wir betrachten dazu
- den logischen Adreßraum N mit n Seiten: N = {0, ..., n-1} sowie
- den physikalischen Adreßraum M mit m Seitenrahmen: M = {0, ..., m-1}.

Im allgemeinen gilt dabei n >> m, das heißt der logische Adreßraum ist i. d. R. wesentlich größer als der physikalische. Die Ausführung eines Programms wird durch seine Folge von Seitenzugriffen charakterisiert. Diese Folge nennen wir den **Reference String** und bezeichnen ihn mit ω.

Sei $\omega = r_1 r_2 \ldots r_T \in N^T$ der Referenzstring einer Programmausführung. Dann bezeichnet r_1 den ersten Seitenzugriff des Programms und r_T den letzten. Wird auf eine Seite mehrmals hintereinander zugegriffen, so sind entsprechend mehrere r_i hintereinander identisch.

Sei weiter $S_t := \{i \mid i \in N \wedge i \text{ belegt Seitenrahmen in } M\}$ der Speicherzustand, d. h. die Menge der aktuell im Hauptspeicher befindlichen Seiten.

Dann läßt sich ein **Seitenaustauschalgorithmus** A als endlicher Automat ohne Ausgabe folgendermaßen beschreiben:

$A = (N, \{S_t\} \times Q, q_0, g_A)$ mit
N = Eingabemenge,
Q = Menge der Kontrollzustände,
$\{S_t\} \times Q$ = Speicherzustand und Kontrollzustand,
q_0 = Startzustand und
g_A = Überführungsfunktion.

Die Überführungsfunktion lautet formal:

$$g_A : N \times (\{S_t\} \times Q) \to \{S_t\} \times Q$$
$$(r_i, S, q) \mapsto (S', q')$$

D. h. wenn r_i die nächste benötigte Seite und der Systemzustand (S, q) ist, dann gehe zu einem neuen Systemzustand über, der eventuell einen Seitenaustausch beinhaltet.

Die verschiedenen (Demand-Paging-) Strategien unterscheiden sich voneinander durch die Struktur der Kontrollzustände und damit durch unterschiedliche Wahl der im Bedarfsfall zu ersetzenden Seite (Ersetzungsstrategie). Wir wollen als nächstes einige solche Strategien näher betrachten.

FIFO (First In First Out)

Die FIFO-Strategie ordnet die Seiten im Hauptspeicher nach ihrem "Alter" an. Im Bedarfsfall wird die älteste Seite verdrängt.

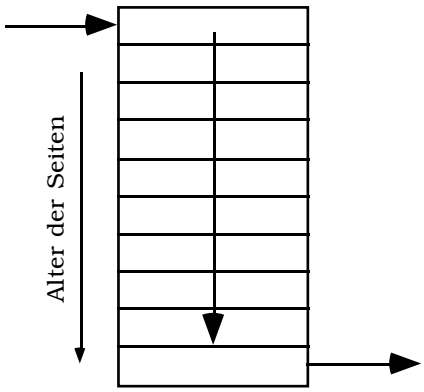


Abb. 6.8: Funktionsweise von FIFO

Angenommen, der Hauptspeicher faßt m Seitenrahmen. Sei der Kontrollzustand $q = (q_1, ..., q_m)$ mit $q_i \in \{0,\ldots,n-1\}$, wobei q_1 die Nummer der jüngsten und q_m die Nummer der ältesten Seite im Hauptspeicher ist. Die Übergangsfunktion g_{FIFO} für die FIFO-Strategie sieht dann wie folgt aus:

$$g_{FIFO} : N \times (\{S_t\} \times Q) \to \{S_t\} \times Q$$
$$(r_i, S, q) \mapsto (S', q')$$

derart, daß

falls $r_i \in S$
$\Rightarrow$ (kein Seitenfehler!) $S' = S$, $q' = q$

falls $r_i \notin S \wedge |S| < m$ und $S = (q_1, ..., q_k)$
$\Rightarrow S' = S \cup \{r_i\}$; $q' = (r_i, q_1, \ldots, q_k)$

falls $r_i \notin S \wedge |S| = m$
$\Rightarrow S' = S \cup \{r_i\} \setminus \{q_m\}$, $q'=(r_i,q_1,\ldots,q_{m-1})$

LRU (Least Recently Used)

Die LRU-Strategie arbeitet wie die FIFO-Strategie, jedoch wird bei jedem Zugriff auf eine Seite, die bereits im Hauptspeicher ist, diese "verjüngt". Im Bedarfsfall muß die Seite mit der maximalen **Rückwärtsdistanz**, also die am längsten nicht mehr benutzte Seite, ersetzt werden. Realisierbar ist eine solche Strategie am einfachsten mit einer verketteten Liste, da ansonsten ein hoher Aufwand an Umspeicherungen entstehen kann.

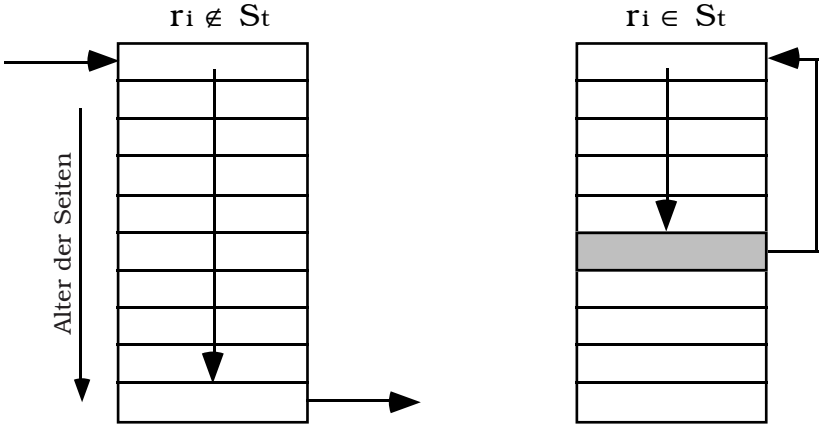


Abb. 6.9: Funktionsweise von LRU

SECOND CHANCE

Second Chance stellt einen Kompromiß zwischen FIFO und LRU dar. Prinzipiell arbeitet die Strategie wie FIFO, jedoch existiert ein zusätzliches Bit für jede Seite (**Use Bit**), welches bei einer nochmaligen Benutzung der Seite gesetzt wird. Ist eine neue Seite zu laden, so verdrängt die Strategie die älteste Seite mit nichtgesetztem Use Bit. Die Use Bits sind zu löschen, wenn

- alle Bits gesetzt sind, da dann daraus keine Informationen mehr zu gewinnen sind, und
- beim Eintreten eines Seitenfehlers.

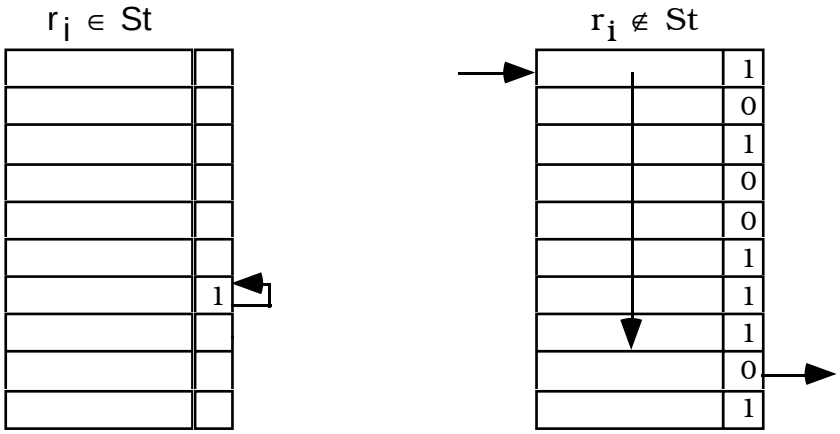


Abb. 6.10: Funktionsweise von Second Chance

Mit einem zusätzlichen Bit (**Modify Bit**) kann diese Strategie noch verfeinert werden. Das Modify Bit wird gesetzt, falls eine Seite im Hauptspeicher geändert wurde. Dies impliziert, daß die Seite nicht nur verdrängt, sondern auch zurück auf den Hintergrundspeicher geschrieben werden muß. Dadurch verursacht das Verdrängen solcher Seiten höhere Kosten. Definiert man für jede Kombination (Use Bit, Modify Bit) eine Klasse, so sollte zuerst das älteste Element der Klasse (0, 0) verdrängt werden. Ist diese Klasse leer, so werden die Klassen (0, 1), (1, 0) und (1, 1) in dieser Reihenfolge untersucht, und die älteste Seite der ersten nichtleeren Klasse wird aus dem Speicher geworfen.

Der Nachteil von FIFO und Second Chance liegt in der Gefahr, daß alte Seiten ausgelagert werden, auch wenn sie oft in Gebrauch sind. Andererseits sind diese beiden Strategien einfach zu realisieren. Weiter ist anzumerken, daß sich, mit Ausnahme von theoretischen Ansätzen, für jedes Verfahren ein Beispiel konstruieren läßt, bei dem es äußerst schlecht dasteht.

LFU (Least Frequently Used)

Die **LFU-Strategie** tauscht im Bedarfsfall die Seite mit der niedrigsten Nutzungshäufigkeit aus. Dabei gibt es mehrere Varianten, die Nutzungshäufigkeit zu messen, beispielsweise betrachtet man die Zugriffe auf die fragliche Seite

- seit Beginn des Referenzstrings,
- innerhalb der letzten h Zugriffe (**Fenster** der Größe h) oder
- seit dem letzten Seitenfehler.

CLIMB (Aufstieg bei Bewährung)

Bei der **Climb-Strategie** steigt eine Seite bei jedem Aufruf eine Position höher, wenn sie bereits im Speicher vorhanden ist. Sie tauscht also ihre Position mit der vor ihr stehenden Seite. Ist die Seite nicht im Speicher vorhanden, so wird die Seite auf der untersten Position verdrängt und die neue Seite dorthin geladen.

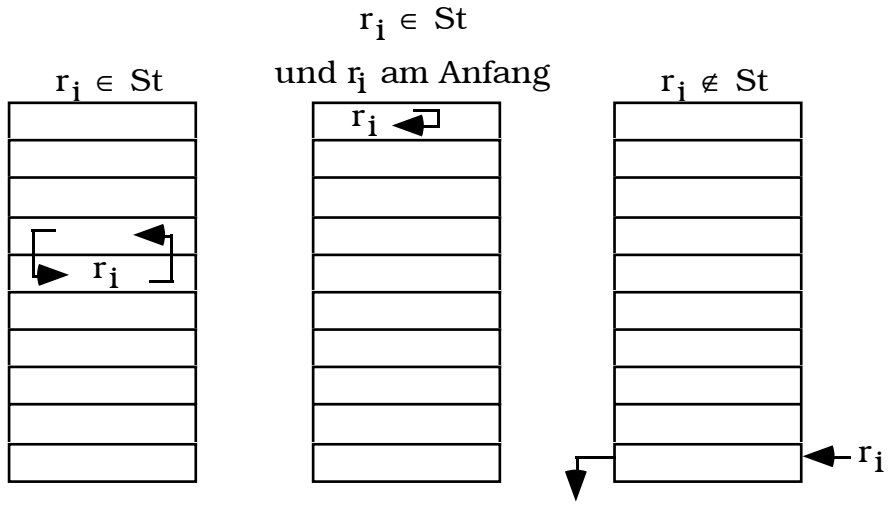


Abb. 6.11: Funktionsweise von CLIMB

RANDOM

Im Falle eines Seitenfehlers wird die Seite, die aus dem Hauptspeicher verdrängt wird, ausgewürfelt (also zufällig bestimmt).

OPT (Optimalstrategie)

Die **optimale Strategie** arbeitet nach dem Prinzip, die Seite mit dem größten **Vorwärtsabstand** auszutauschen. Die Seite also, die am längsten nicht mehr gebraucht werden wird, ist Tauschkandidat. OPT verursacht die wenigsten Seitenfehler unter allen Demand-Paging-Algorithmen, ist aber nur in Ausnahmefällen realisierbar. Man kann das Verfahren allerdings approximieren, indem man die Seite als Tauschkandidaten wählt, deren erwarteter Vorwärtsabstand am höchsten ist. Dieser Erwartungswert kann z. B. aufgrund von Ergebnissen des zurückliegenden Teils des Referenzstrings geschätzt werden. OPT eignet sich sehr gut dafür, die Güte der vorherigen Verfahren zu bestimmen ("Strategie X ist max x% schlechter als OPT").

6.5 Nicht-Demand-Paging

Nicht-Demand-Paging-Algorithmen nehmen bei Bedarf mehrere (und nicht nur erzwungene) Seitentransporte vor. Die Strategien nutzen die Tatsache aus, daß Programme meist "geographisch lokal" arbeiten: Wenn auf eine Seite r_i zugegriffen wird, ist die Chance, daß die "Nachbarn" r_i+1 (rechter Nachbar) und r_i-1 (linker Nachbar) ebenfalls referiert werden, sehr groß. Gute Beispiele hierfür sind sequentielle Schleifendurchläufe oder die Ausführung von einfachem Programmcode.

Unter den Nicht-Demand-Paging-Algorithmen ist der OBL-Algorithmus (One Block Look-Ahead) am bekanntesten. OBL arbeitet, solange kein Seitenfehler auftritt, im wesentlichen wie LRU. Es existieren mehrere Varianten dieser Strategie, von denen zwei hier vorgestellt werden sollen.

6.5.1 OBL als Demand-Prepaging-Version

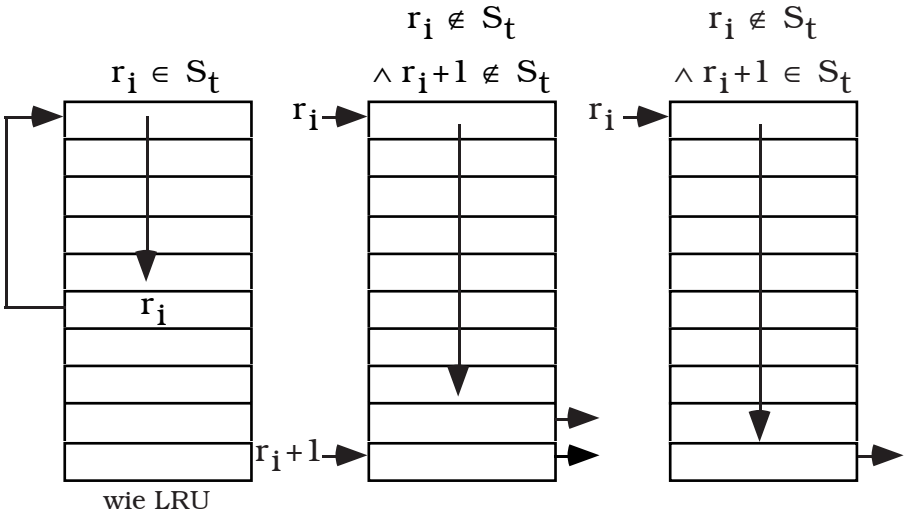


Abb. 6.12: Funktionsweise von OBL als Demand-Prepaging-Version

Falls r_i bereits im Speicher vorhanden ist, verhält sich das Verfahren analog zu LRU. Unterschiede treten auf, wenn aufgrund eines Seitenfehlers r_i nachzuladen ist. In diesem Fall wird unterschieden, ob die auf r_i folgende Seite r_i+1 bereits im Speicher vorhanden ist oder nicht. Ist sie nicht vorhanden, so ist sie an die letzte Stelle im Speicher zu laden. Ist die Seite r_i+1 allerdings vorhanden, so ist die Vorgehensweise wie bei FIFO.

6.5.2 OBL als Look-Ahead-Variante

Die Look-Ahead-Variante ist analog zur Demand-Prepaging-Version bis auf den Fall, daß die Seite r_i bereits im Speicher ist. Dann ist zu unterscheiden, ob die nachfolgende Seite r_i+1 bereits im Speicher ist oder nicht. Falls nein, so wird sie zusätzlich an die letzte Position geladen. Ist sie bereits vorhanden, so verhält sich die Variante wie die LRU-Strategie.

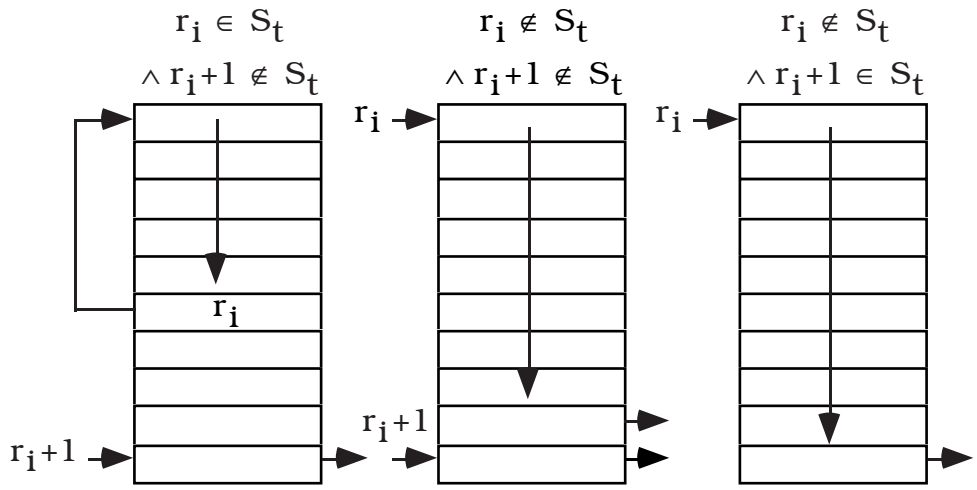


Abb. 6.13: Funktionsweise von OBL als Look-Ahead-Version

6.6 Diskussion der Paging-Algorithmen

An dieser Stelle sollen beide Arten des Pagings gegenübergestellt werden. Als Zielgröße betrachten wir dabei im folgenden die Kosten.

6.6.1 Kosten von Paging-Algorithmen

Eine Möglichkeit der Bewertung von Algorithmen sind ihre Kosten. Unter **Kosten** versteht man den Aufwand für die Speicherzustandsänderung.

Im weiteren Verlauf nehmen wir an, daß die Verdrängungskosten proportional zu den Ladekosten sind. Ferner sind alle Kosten für das Laden und ggf. das Verdrängen auf eins zu normieren.

Sei S_t der Speicherzustand zum Zeitpunkt t (d.h. der Zustand nach dem t-ten Speicherzugriff). S_t berechnet sich zu

$$S_t = S_{t-1} \cup X_t \setminus Y_t.$$

Dabei bezeichnet X_t die Menge der neugeladenen und Y_t die Menge der verdrängten Seiten.

Bei Anwendung eines Algorithmus A auf den Referenzstring $\omega = r_1 r_2 \ldots r_T$ und bei Verwendung von m Hauptspeicherseiten entstehen folgende Kosten:

a) Gesamtzahl der Seitentransporte (Seitenfehler)

$$\alpha(A, m, \omega) := \sum_{t=1}^{T} |X_t|$$

b) Kosten für das Laden von i Seiten =: h(i) mit
h: $N_0 \rightarrow N_0$ mit h(0) := 0
h(1) := 1 (normiert)
h(i) ≥ h(i-1)

c) Gesamtkosten C für den Referenzstring ω:

$$C(A, m, \omega) := \sum_{t=1}^{T} h(|X_t|)$$

Wie man sich leicht überlegen kann, gibt es für jeden Nicht-Demand-Paging-Algorithmus einen Demand-Paging-Algorithmus, der bei beliebigem Referenzstring höchstens ebensoviele Seitenfehler macht wie besagter Nicht-Demand-Paging-Algorithmus.

Daß Nicht-Demand-Paging-Algorithmen dennoch von Bedeutung sind, läßt sich damit begründen, daß die Gesamtkosten bei einem Nicht-Demand-Paging-Algorithmus in Abhängigkeit vom Referenzstring ω und der Speichergröße m geringer sein können als beim entsprechenden Demand-Paging.

Zu jedem Algorithmus A existiert ein DPA (Demand-Paging-Algorithmus) A* mit:

$$\alpha(A^*, m, \omega) \leq \alpha(A, m, \omega) \quad \forall m, \omega$$

d. h. DPA ist optimal bezüglich der Zahl der Seitentransporte.

Die Aussage, daß zu jedem Algorithmus A ein DPA A* existiert mit:

$$C(A^*, m, \omega) \leq C(A, m, \omega) \quad \forall m, \omega,$$

(d. h. DPA ist optimal bezüglich der Kosten) gilt jedoch nur, wenn das gemeinsame Nachladen von k Seiten nicht billiger ist als das einzelne Laden von k Seiten (also $h(k) \geq k \cdot h(1) = k$ für $k \geq 0$).

Die folgenden Betrachtungen beziehen sich auf Demand-Paging-Algorithmen. Wir stellen die Frage, wie sich die Seitenfehlerzahl und die Gesamtkosten für einen Referenzstring ω in Abhängigkeit von der Seitenzahl m verändern.

Zunächst liegt die Vermutung nahe, daß sich die Zahl der Seitenfehler (und damit bei DPAs die Kosten) mit wachsendem m verringert. Auch die Trefferquote (hit ratio) sollte sich verbessern. Jedoch ist diese Vermutung für einige Strategien und bei bestimmten "pathologischen" Referenzstrings überraschenderweise falsch! Die Vermutung gilt z. B. nicht für FIFO, SECOND CHANCE und CLIMB, ist aber für LRU, OPT und LIFO gültig. Dieses Phänomen soll am Beispiel von FIFO veranschaulicht werden.

6.6.2 FIFO-Anomalie

Es gibt m, ω mit: $C(FIFO, m+1, \omega) > C(FIFO, m, \omega)$

$\alpha(FIFO, m+1, \omega) > \alpha(FIFO, m, \omega)$

Beispiel: Sei m = 3; ω_1 = 2 3 0 1; ω_2 = 2 0 3 1 4 2 5 3 0 4 1 5; $\omega = \omega_1 \cdot \omega_2^k$

ω	**2 3 0 1**	**2 0 3 1 4 2 5 3 0 4 1 5**	**2 0 3 1 4 2 5 3 …**
3 Seiten	2 3 0 1	2 3 4 5 0 1	2 3 4 …
	2 3 0	1 2 3 4 5 0	1 2 3 …
	2 3	0 1 2 3 4 5	0 1 2 …
Seitenfehler	x x x x	x x x x x x	x x x
4 Seiten	2 3 0 1	4 2 5 3 0 4 1 5	2 0 3 1 4 2 5 …
	2 3 0	1 4 2 5 3 0 4 1	5 2 0 3 1 4 2 …
	2 3	0 1 4 2 5 3 0 4	1 5 2 0 3 1 4 …
	2	3 0 1 4 2 5 3 0	4 1 5 2 0 3 1 …
Seitenfehler	x x x x	x x x x x x x x	x x x x x x x …

Daraus kann für dieses Beispiel gefolgert werden:

$$\frac{C(FIFO,3,\omega=\omega_1\cdot\omega_2^k)}{C(FIFO,4,\omega=\omega_1\cdot\omega_2^k)} \xrightarrow{(k\to\infty)} \frac{1}{2}$$

Die Vergrößerung des Speichers führt bei diesem speziellen ω zu einer Verdopplung der Seitenfehler und der Kosten. Der Grund für diese sogenannte **FIFO-Anomalie** liegt darin, daß bei FIFO die Seiten unabhängig von ihrer Aktualität altern.

Interessant ist nun die Frage, für welche Algorithmen keine Beispiele der obigen Art konstruiert werden können. Die Antwort darauf führt uns zum Begriff des "Stack-Algorithmus".

6.6.3 Stack-Algorithmen

Sei S(A, m, ω) = S(m, ω) die Menge der Seitennummern, welche am Ende der Abarbeitung eines Referenzstrings ω durch einen DPA unter Verwendung von m Rahmen im Speicher stehen. Ein Algorithmus heißt **Stackalgorithmus** genau dann, wenn

$$S(m,\omega) \subseteq S(m+1,\omega) \quad \forall m,\omega$$

erfüllt ist.

Daraus kann man folgern, daß für jeden Stackalgorithmus gilt:
a) Mit wachsender Speichergröße sinkt die Fehlerrate ab. Ebenso sinken die Nachladekosten.
b) Als Konsequenz daraus ergibt sich, daß FIFO kein Stackalgorithmus ist (ebenso wie CLIMB und einige andere mehr).

LRU, LIFO und OPT sind Stackalgorithmen. Bei der Strategie LRU besteht dabei der Stack aus der Menge der zuletzt benötigten Seiten.

6.6.4 Prioritätsalgorithmen

Ein Demand-Paging-Algorithmus heißt **Prioritätsalgorithmus** genau dann, wenn es für alle ω (und unabhängig von m) eine Folge $\pi_1, \pi_2, \ldots, \pi_{T-1}$ von sogenannten Prioritätslisten gibt, für die gilt:

1) π_i ist eine geordnete Liste der in $r_1, r_2, \ldots, r_i$ vorkommenden Seitennummern.
2) Ist $\omega = r_1 r_2 \ldots r_t$ und $|S(m, \omega)| = m$ und $r_{t+1} \notin S(m, \omega)$ (d. h. es muß eine Seite ersetzt werden), dann bestimmt sich die zu verdrängende Seite durch die Liste π_t wie folgt:

$$S(m, \omega, r_{t+1}) = S(m, \omega) \cup \{r_{t+1}\} \setminus \min_{\pi_t} S(m, \omega)$$

Dabei ist $\min_{\pi_t} S(m, \omega)$ das Element niedrigster Priorität in S(m, ω) bzgl. der durch π_t gegebenen Anordnung.

Kurz gefaßt heißt dies, daß Prioritätsalgorithmen das Austauschverhalten unabhängig von m durch Prioritätslisten bestimmen.

Die folgende Tabelle gibt Beispiele für Prioritätsalgorithmen und die dabei verwendeten Prioritätslistenanordnungen.

Algorithmus	**Prioritätslistenanordnung**
LRU	wachsende Rückwärtsdistanz
OPT	wachsende Vorwärtsdistanz
LIFO	wachsende Zeit des Eintritts in den Hauptspeicher
LFU	abnehmende Häufigkeit der Benutzung

Es gilt folgender Satz:

Satz: a) A ist Prioritätsalgorithmus ⇒ A ist Stackalgorithmus

b) Zu jedem Stackalgorithmus A existiert eine Folge von Prioritätslisten, d. h. A ist Stackalgorithmus ⇒ A ist Prioritätsalgorithmus

Auf einen Beweis wollen wir hier verzichten.
Fallstrick:

Warum ist FIFO kein Prioritätsalgorithmus? Man könnte doch beispielsweise als Prioritätsliste die Ordnung der Seiten nach ihrem Alter im Hauptspeicher hernehmen und damit folgern:

$$\text{FIFO} = \text{Prioritätsalg. (?)} \Rightarrow \text{FIFO} = \text{Stackalg. (?)} \Rightarrow \text{Anomalie unmöglich ?}$$

im Widerspruch zum oben Gesagtem. Der Argumentationsfehler hierbei liegt in der vorgeschlagenen Prioritätsliste, da diese von m nicht unabhängig ist!

7 Speicherzuteilung bei Multiprogramming

7.1 Grundlegende Bemerkungen

Die bisher vorgestellten Ergebnisse beschränkten sich auf den Einprogrammbetrieb. Die Verhältnisse verkomplizieren sich spürbar beim Mehrprogrammbetrieb (Multiprogramming). Andererseits ist Multiprogramming wünschenswert, da der Einprogrammbetrieb aufgrund der Geschwindigkeitsdiskrepanz zwischen der CPU und den Endgeräten sehr ineffizient sein kann.

Beim Multiprogramming sind mehrere Programme gleichzeitig aktiv und greifen auf eine gemeinsame CPU zu. Wie soll man in diesem Fall den gemeinsamen Hauptspeicher für die verschiedenen Prozesse verwalten? Geht man beispielsweise von n gleichzeitig aktiven Prozessen aus, so stellt sich die Frage, wieviele der insgesamt verfügbaren Rahmen jeder einzelne Prozeß erhalten soll. Daß die naheliegende Idee, jedem Prozeß gleichviele Rahmen (z. B. m Stück) zuzuteilen, sich nicht immer als optimal herausstellen wird, ist unmittelbar einsichtig: Manche Prozesse werden i. d. R. nicht alle zugeteilten Seiten brauchen, während andere mit ihrem Anteil hinten und vorne nicht auskommen.

Ein eng damit zusammenhängendes Problem (da n und m ja voneinander abhängen) ist die Frage nach dem optimalen Multiprogramminggrad n. Wählt man n zu klein, so verschwendet man offensichtlich Systemressourcen. Zu großes n führt andererseits dazu, daß die Zahl der Rahmen, die man einem einzelnen Prozeß zuteilen kann, sinkt. Dadurch wächst die Zahl der Seitenfehler, bis (im sogenannten "**Thrashing**"-Bereich) der Systemdurchsatz indiskutabel wird.

Man kann folgende schematische Abhängigkeit des Systemdurchsatzes D vom Multiprogramminggrad feststellen:

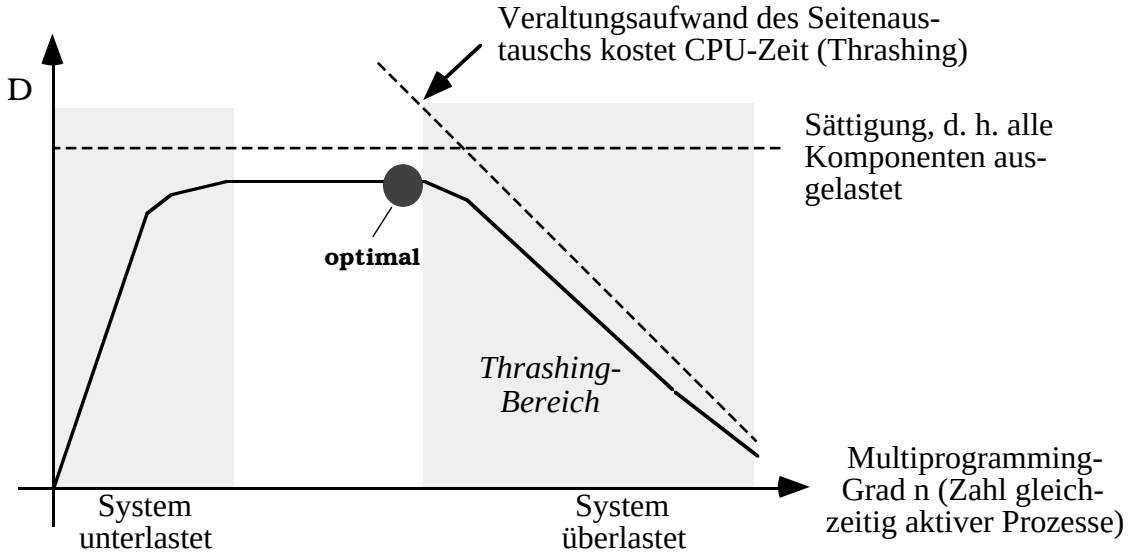


Abb. 7.1: Thrashing durch steigenden Multiprogramminggrad

Aufteilung des Speichers auf aktive Programme

Wie soll man also sinnvollerweise den Speicher auf die zur Zeit aktiven Prozesse aufteilen? Wünschenswert ist sicherlich, die Größe der zugewiesenen Bereiche in Abhängigkeit von der Zeit festzulegen. Daneben ergeben sich eine Reihe von Detailfragen:

1. Wieviele Rahmen weist man einem Prozeß zu?
2. Wann soll ein Prozeß neu aufgenommen werden?
3. Wann ist ein aktiver Prozeß stillzulegen (zu deaktivieren)?
4. Wann ändert sich der Speicherbereich?

Vorläufig können wir auf diese Fragen folgende allgemeine Antworten geben:

ad 1.: Man soll einem Programm so viele Rahmen geben, wie es "braucht", also die Anzahl momentan aktiver Seiten. Dabei nennen wir eine Seite "momentan aktiv", wenn die Wahrscheinlichkeit hoch ist, daß sie in naher Zukunft benötigt wird.

ad 2.: Ein neuer Prozeß kann dann gestartet werden, wenn genügend Speicherplatz für seine aktiven Seiten verfügbar ist.

ad 3.: Ein Prozeß ist dann stillzulegen, wenn ein Seitenfehler auftritt und alle Tauschkandidaten momentan aktiv sind.

ad 4.: Wenn die Zahl der aktiven Seiten eines Prozesses ansteigt und andere Prozesse Seiten gespeichert haben, die momentan nicht aktiv sind, so dürfen diese inaktiven Seiten von anderen Prozessen genutzt werden.

Lifetime-Funktion

Die Lifetime-Funktion L(m) gibt die mittlere Zeit zwischen aufeinanderfolgenden Seitenfehlern in Abhängigkeit von der zugeordneten Rahmenzahl m an. Gewöhnlich (d. h. einige seltene Anomalien ausgenommen) steigt L mit wachsendem m monoton an. Je mehr Rahmen ein Prozeß also im Hauptspeicher zur Verfügung hat, desto seltener werden die Seitenfehler, und desto mehr Zeit vergeht daher im Mittel zwischen zwei derselben. Die typische Gestalt einer Lifetime-Funktion hängt dabei sehr von den Prozessen ab, aber in den meisten Fällen hat L näherungsweise folgende Form (die sich mit einiger Fantasie als "S" interpretieren läßt):

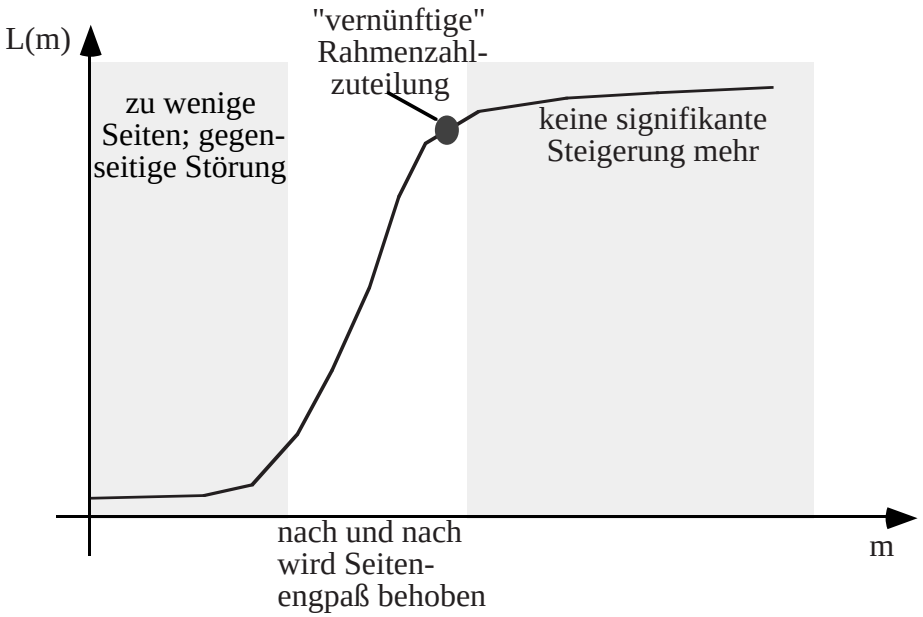


Abb. 7.2: Lifetime-Funktion

Stehen einem Prozeß nur wenige Rahmen zu, so kommt es zu häufigen Seitenfehlern. Jeder weitere dem Prozeß zugeteilte Rahmen bringt eine spürbare Verlängerung der Lifetime mit sich. Dies geht aber i. d. R. nicht ewig so weiter, stattdessen mündet die Kurve in einen Sättigungsbereich, in dem weitere Rahmenzuteilungen kaum mehr Einfluß auf die Lifetime haben. Steht einem Prozeß eine große Anzahl von Seiten zur Verfügung, kann es trotzdem zum Fehlen einer seltenen "exotischen" Seite kommen, was in einem erneuten Seitenfehler resultiert.

7.2 Working Set

Zur Schätzung der Lifetime-Funktion ist der sogenannte "**Working Set**" von Bedeutung. Hierunter versteht man die Menge der in einem (noch präziser zu definierenden) Intervall der unmittelbaren Vergangenheit benötigten Seiten. Dieser Ansatz beruht auf folgenden (empirisch motivierten) Annahmen:

- Die jüngere Vergangenheit ist eng mit der unmittelbaren Zukunft korreliert.
- Ein Prozeß, der bislang viele aktive Seiten hatte, wird sich darin mit hoher Wahrscheinlichkeit nicht ändern.
- Über weite Bereiche verhalten Prozesse sich "lokal": Seiten, die erst vor kurzem referenziert wurden, sind in der Zukunft wahrscheinlicher als solche, deren letzter Zugriff schon länger zurückliegt.

Diese Annahmen sind allerdings nicht allgemeingültig, Ausnahmen durchaus vorstellbar, beispielsweise dann, wenn ein Prozeß in einen völlig neuen Abschnitt eintritt (also einen "Phasenwechsel" durchmacht).

Betrachten wir nun einen Prozeß mit Referenzstring $\omega = r_1 r_2 r_3 ... r_t ... r_T$. Der Working Set W(t, h) dieses Prozesses zur Zeit t unter einem Rückwärtsfenster der Größe h ist dann definiert als

$$W(t,h) := \bigcup_{i=t-h+1}^{t} r_i$$

W(t, h) ist also die Menge der Seiten, die bei den letzten h Zugriffen mindestens einmal referenziert wurden.

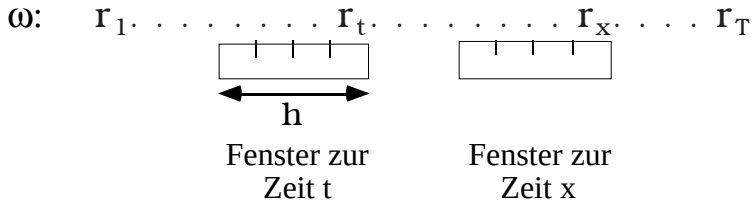


Abb. 7.3: Working-Set

Sei w(t, h) die Mächtigkeit von W(t, h). Trivial ist, daß w(t, h) mit wachsendem h steigt. "Lokale" Prozesse haben jedoch einen relativ kleinen Working Set, was bei nicht-lokalen Prozessen nicht der Fall ist. Jedenfalls stellt sich der Einfluß des Parameters h in jedem Fall als entscheidend heraus: Ist h zu klein gewählt, dann sind nicht alle aktiven Seiten im Working Set, ist h zu groß, dann befinden sich viele inaktive Seiten darin. Über die als nächstes erläuterte Working-Set-Strategie hat h direkten Einfluß auf die mittlere Anzahl m der zugeteilten Seiten pro Prozeß und damit auch auf den Multiprogramminggrad n.

Die Working-Set-Strategie:

1. h "gut" einstellen.

2. Jedem Prozeß so viele Rahmen zuteilen, wie seinem aktuellen Working Set W(t, h) entsprechen. Wird zusätzlicher Speicherplatz frei, so kann ggf. ein neuer Prozeß aktiviert werden. Umgekehrt muß ein Prozeß stillgelegt werden, wenn die Summe der Größen aller Working Sets momentan zu hoch ist.

3. Bei Seitenfehlern sind solche Seiten Tauschkandidaten, die aktuell zu keinem Working Set gehören. Der entsprechende Prozeß "stiehlt" sich diese Seite und gewinnt temporär eine neue Seite. Die Speicherzuteilung ist also variabel.

4. Ist kein Tauschkandidat verfügbar (jede Seite also im Working Set eines Prozesses), so kann man davon ausgehen, daß ein Austausch schädlich ist (aufgrund der Überlastung des Systems). Ein Seitentausch würde nur eine aktive Seite treffen und damit weitere Seitenfehler nach sich ziehen. Besser ist es, in diesem Fall den anfordernden Prozeß stillzulegen und erst dann wieder zu reaktivieren, wenn sich die Verhältnisse gebessert haben.

7.3 Einstellung der Fenstergröße h

Zur Beantwortung der Frage, welche Kriterien man heranziehen kann, um die Fenstergröße h optimal zu wählen, lassen sich drei Vorschläge machen:

7.3.1 Das "Knie-Kriterium"

Ein erstes Kriterium lautet folgendermaßen:

Wähle h so, daß $W(t,h) \cong m_{opt}$, wobei m_{opt} die Seitenzahl ist, welche zum "**primären Knie**" gehört. Das primäre Knie der Lifetime-Funktion ist der Kurvenwert, der von einer Gerade (durch den Ursprung), die sich L(m) von oben nähert, berührt (aber nicht geschnitten) wird. h heißt Abzisse des primären Knies genau dann, wenn

$$\frac{L(h)}{h} \geq \frac{L(h^*)}{h^*}$$

für alle h* gilt.

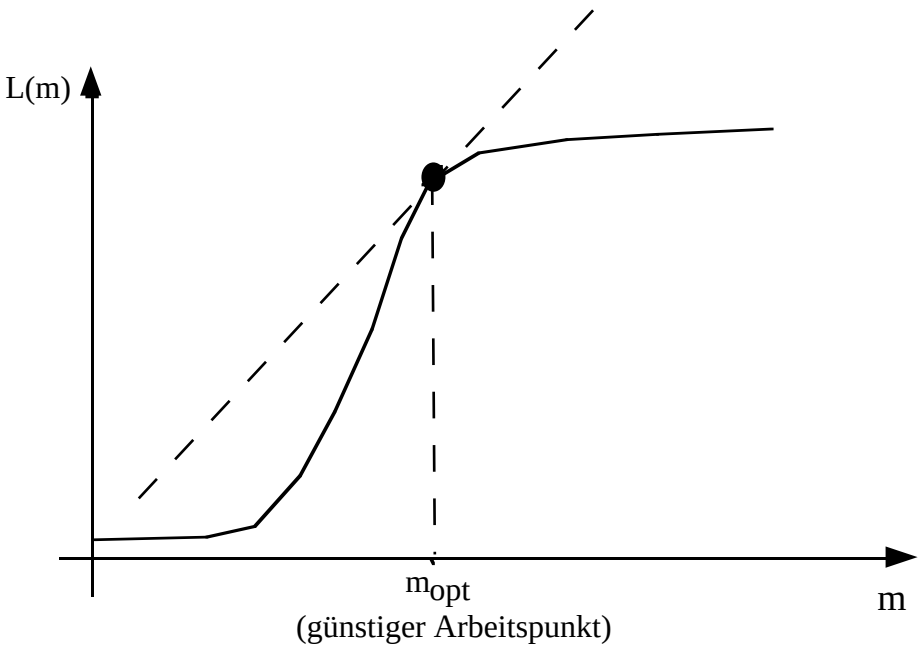


Abb. 7.4: Das Knie-Kriterium

7.3.2 Das "L=S - Kriterium"

Stelle h so ein, daß in etwa "Lifetime = Swaptime" gilt. Dieses Kriterium ergibt sich aus folgenden Überlegungen:

Der Gesamtdurchsatz D durch das System wird beschränkt durch:
a) Die mittlere Jobrate 1/T, wobei T die mittlere Rechenzeit/Job ist.
b) Die Kapazität der Paging-Station, die die Seitenfehler bearbeitet.

Sei a(m) die Seitenfehlerrate eines Jobs bei m zugeteilten Seiten und b die Bedienrate der Pagingstation (d. h. die Bediendauer eines Seitenfehlers beträgt $1/b$ Zeiteinheiten). Dann belegt jeder Job die Pagingstation für $T \cdot a(m) / b$ ZE und der Durchsatz wird durch $b/(T \cdot a(m))$ beschränkt.

Somit ergibt sich

$$D \le \frac{1}{T} \cdot \min\left(1, \frac{b}{a(m)}\right).$$

Mit

$$a(m) = \frac{1}{L(m)} = \frac{1}{\text{Lifetime}}$$

und

$$b = \frac{1}{S} = \frac{1}{\text{Swaptime}} \quad \left(= \frac{1}{\text{Bediendauer}}\right)$$

ergibt sich schließlich

$$D \le \frac{1}{T} \cdot \min\left(1, \frac{L(m)}{S}\right).$$

Dies ist gut erfüllt, wenn $L(m) \approx S$ gilt.

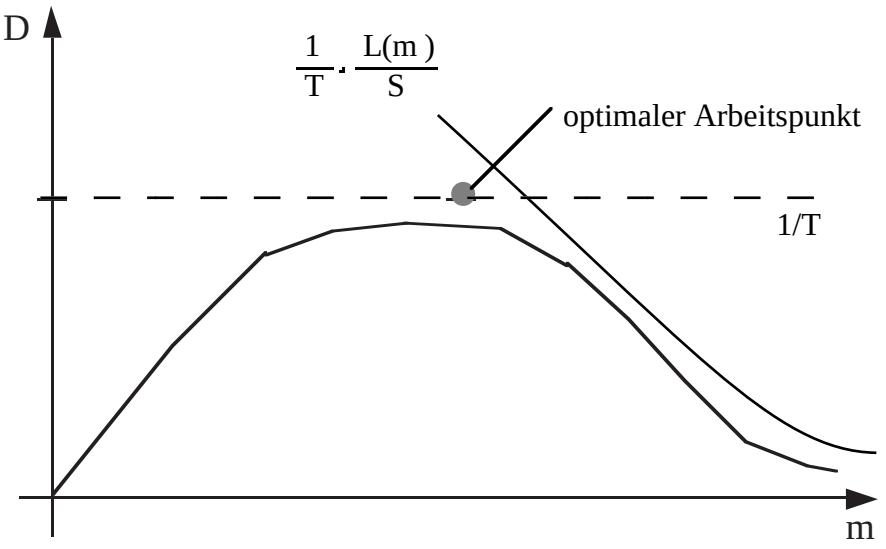


Abb. 7.5: Das "L = S -Kriterium"

Eine leichte Verbesserung erreicht man noch, wenn $L(m) \approx c \cdot S$ gewählt wird, wobei c ein Erfahrungswert ist und etwa 1,3 oder 1,5 betragen sollte.

7.3.3 Das "50%-Kriterium"

Das 50%-Kriterium besagt: Wähle h so, daß die Paging-Station zu ungefähr 50% ausgelastet ist. Man beachte, daß dieses Kriterium fast identisch zum "L = S - Kriterium" ist.

Nachweis:

Die Paging-Station kann als M/M/1-Warteschlangensystem angegeben werden. Das erste M bedeutet dabei, daß die Zeiten zwischen zwei Ankünften an der Paging-Station "memoryless", d. h. negativ exponentialverteilt sind, das zweite M bedeutet analoges für die Bedienzeitverteilung und die 1 steht für "eine Bedienstation".

Für M/M/1 gilt nun:

$$\text{Auslastung} = \rho = \frac{\lambda}{\mu} < 1$$

wobei λ der Parameter der Ankunftsverteilung und μ der Parameter der Bedienverteilung ist. Die mittlere Kundenzahl $\overline{N}$ berechnet sich bei M/M/1-Systemen wie folgt:

$$\overline{N} = \frac{\rho}{1 - \rho}$$

$\overline{N} = 1$ wird für $\rho = 1/2$ erreicht, also bei 50% Auslastung der Paging-Station. $\overline{N} = 1$ bedeutet jedoch auch, daß $L(m) = S$ ist. Realistisch gesehen, sind die Bedienzeiten der Paging-Station jedoch keineswegs "memoryless", sondern nahezu konstant. Das Modell für diesen Fall lautet M/D/1, wobei D für "deterministische Bedienung" steht.

Für die mittlere Kundenzahl $\overline{N}$ in M/D/1-Systemen gilt:

$$\overline{N} = \frac{\rho}{1 - \rho} - \frac{\rho^2}{2(1-\rho)}$$

also

$$\overline{N} = 1 \Leftrightarrow \rho = 2 \pm \sqrt{2} \approx 0,58$$

Der Fall $\rho = 2 + \sqrt{2}$ kommt hierbei nicht in Frage, da die Auslastung ρ nie über 1 steigen kann. Die optimale Auslastung der Paging-Station, die sich mit der Verbesserung ergibt, liegt also bei ≈ 58%.

Ist die Größe des Working Sets beschränkt, so stellt sich die Frage, nach welchen Kriterien freiwerdende Seiten ersetzt werden. Eine **"einfache" Working-Set-Strategie** wäre, eine freie Seite beliebig auszuwählen. **WS-LRU** dagegen nutzt die bei fester Speicherzuteilung bewährte Strategie: Aus der Gesamtheit der freiwerdenden Seiten wird diejenige verdrängt, die am längsten nicht genutzt wurde. Im Vergleich zu solchen **Fixed Space Strategies** zeigt sich, daß **Variable Space Strategies** wie WS-LRU i. a. deutlich bessere Ergebnisse bringen als selbst "Fixed"-OPT.

Ein Problem der Working-Set-Strategie liegt darin, daß sie aufwendig ist, da der aktuelle Working Set dauernd berechnet werden muß.

Vereinfachung:
Steuere zugeteilte Rahmen pro Prozeß bzw. Multiprogramminggrad über "Page Fault Frequency" (PFF), d. h. die Seitenfehlerhäufigkeit des Programms:

a) Wenn PFF zu hoch, dann gib diesem Programm mehr Seiten. Ist dieses nicht möglich, da bereits alle Seiten belegt, so lagere das Programm aus.

b) Wenn PFF zu niedrig, dann gib Seite eines entsprechenden Programms frei und starte evtl. zusätzliches Programm.

7.4 Die optimale Strategie VOPT

Gibt es nun (ähnlich wie im fixen Fall) eine optimale variable Strategie? Die Antwort lautet: Ja, und sie hat sogar einen Namen: **Variable OPT (VOPT)**.
Zunächst sollte jedoch die Bedeutung von ”optimal” in diesem Zusammenhang definiert werden:

Eine Variable Space Strategy heißt **optimal** genau dann, wenn die Strategie die Zahl der Seitenfehler bei gegebener mittlerer Zahl zugeteilter Seiten minimiert.

Betrachtet man nun einen Referenzstring zur Zeit t, so ergibt sich der Working Set WS(t,h) (wie schon bekannt) aus dem Rückwärtsfenster:

$$WS(t,h) = RF(t,h) = \bigcup_{j=t-h+1}^{t} r_j$$

Für VOPT wird dagegen das Vorwärtsfenster benutzt:

$$VF(t,h) = \bigcup_{j=t+1}^{t+h} r_j$$

Ist zusätzlich M_t die Menge der Seitenrahmen, die einem Programm zur Zeit t zugeordnet sind, so lautet die Strategie VOPT wie folgt:

Sei r_t die zuletzt referierte Seite. r_t wird gehalten, falls r_t im Vorwärtsfenster VF(t,h) enthalten ist. Ist r_t darin nicht enthalten, so wird die Seite sofort verdrängt, d. h. sie ist weder in M_{t+1}, M_{t+2}, ... noch in M_{t+h} enthalten.

Zwei direkte Folgerungen daraus sind:
1. VF(t, h) = RF(t+h, h)
2. VOPT agiert bezüglich der Seitenfehler genauso wie WS. WS hält jedoch überflüssige Seiten noch h Zeiteinheiten länger als VOPT.

Der Nachweis, daß VOPT und WS genau dieselben Seitenfehler produzieren, ist einfach. Seien r_t und r_{t+u} (u > 1) zwei aufeinanderfolgende Zugriffe auf dieselbe Seite. Dann gilt mit einer Fallunterscheidung:

a) u > h: VOPT wirft die Seite sofort raus (vor r_{t+1}). WS entfernt die Seite auch, jedoch erst zum Zeitpunkt r_{t+h} (t+h < t+u), denn erst dann ist die Seite nicht mehr im Rückwärtsfenster.

b) u ≤ h: Keine der Strategien lagert die Seite aus.

Im Fall a) müssen also beide Strategien die Seite erneut laden, während sie im Fall b) jeweils erhalten bleibt.

Trotzdem gibt es einen wichtigen Unterschied zwischen den Strategien (wie der schon eben angesprochene Fall a) vermuten läßt): Die mittlere zugeteilte Seitenzahl ist bei VOPT geringer als bei WS. Deutlich wird dies insbesondere bei **Phasenwechseln** von Programmen, d. h. in den Zeitbereichen, in denen Programme von einem lokalen Bereich in einen anderen wechseln. Während sich bei VOPT die Zahl der zugeteilten Seitenrahmen verringert, überschätzt WS die Anzahl wirklich benötigter Rahmen.

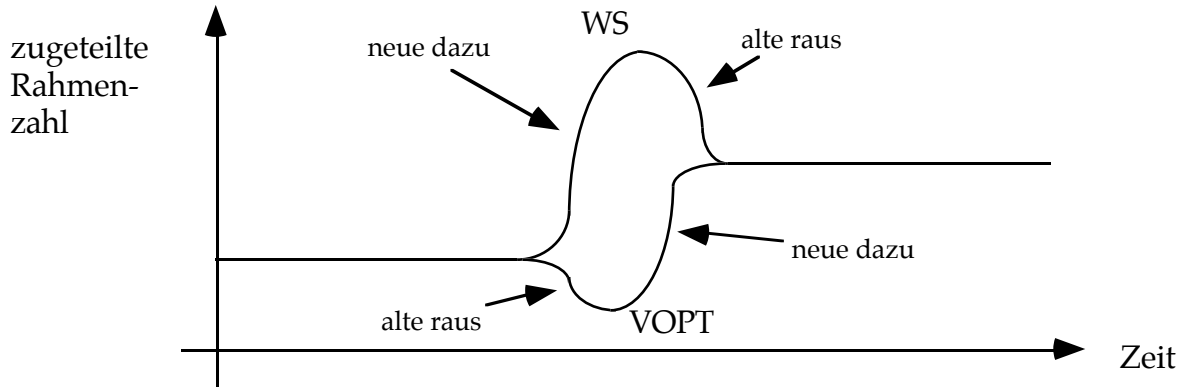


Abb. 7.6: Verhalten von WS bzw. VOPT bei Phasenwechseln

Bisher bezogen sich die Kosten, mit der eine Strategie beurteilt wurde, ausschließlich auf die Zahl der Seitenfehler. Wie gerade gezeigt, ist aber auch die Einbeziehung von Kosten, die die Anzahl der gehaltenen Seiten beschreiben, sinnvoll. Es soll nun gelten:

Kosten ≙ Seitenfehlerkosten + Seitenhaltekosten

Bei Fixed Space Strategies war diese Berücksichtigung übrigens nicht nötig, da in diesem Fall jedem Programm eine feste Anzahl von Seiten zugeteilt war, durch deren Freigabe kein weiteres Programm lauffähig wurde.

Es gilt nun folgender Satz:

Satz: Bezüglich der neuen Kostenfunktion ist VOPT der optimale Paging-Algorithmus, falls die Fenstergröße $h = \frac{R}{U}$ gewählt wird. Dabei entspricht R den Kosten der Bearbeitung eines Seitenfehlers und U den Kosten für das Halten einer Seite pro Zeiteinheit.

Beweis:

Sei $\omega = r_1\ r_2\ ...\ r_t$ ein beliebiger Referenzstring, der M Seiten benötigt. Die Kosten C dieses Referenzstrings ergeben sich dann zu:

$$C = \underbrace{M \cdot R}_{\substack{\text{Kosten, die durch} \\ \text{den Erstzugriff} \\ \text{entstehen}}} + \underbrace{\sum_{i=1}^{t} c_i}_{\substack{\text{Kosten, die nach} \\ \text{Zugriff } r_i \text{ bis zum} \\ \text{nächsten Zugriff auf} \\ \text{diese Seite entstehen}}}$$

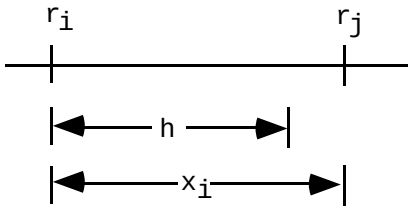
Die Kosten c_i berechnen sich dabei folgendermaßen:
Seien r_i und r_j zwei aufeinanderfolgende Zugriffe auf dieselbe Seite. Definiert man mithin $x_i := j - i$, so ergeben sich folgende zwei Fälle:

a) r_i wird zwischen i und j nicht aus dem Speicher geworfen
$\Rightarrow c_i = x_i \cdot U$,
d. h. die Seite wird x_i Zugriffe lang gehalten und es entstehen entsprechende Kosten.

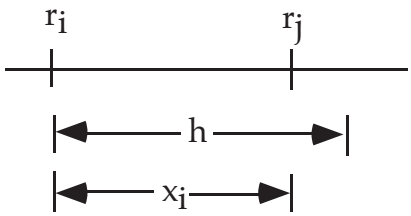
b) r_i wird zwischen i und j (im Schritt z_i, $0 \le z_i < j$) aus dem Speicher geworfen
$\Rightarrow c_i = z_i \cdot U + R$,
d. h. die Seite wird z_i Zugriffe lang gehalten, danach entfernt und anschließend mit Kosten R neu geladen (für VOPT gilt immer $z_i = 0$).

Gegeben sei nun ein Fenster der Größe $h = \frac{R}{U}$ für VOPT.

Für Fall a) muß offensichtlich gelten $h = \frac{R}{U} < x_i$.



VOPT wirft r_i raus und lädt sie wieder nach $\Rightarrow c_i(\text{VOPT}) = U \cdot 0 + R = R$.
Vergleicht man VOPT mit einer beliebigen anderen Strategie A^*, welche die Seite hält, so ergibt sich $c_i(A^*) = U \cdot x_i > U \cdot h = R = c_i(\text{VOPT})$, d. h. VOPT ist im Fall a) besser als jede andere Strategie A^*.
Für Fall b) gilt $h = \frac{R}{U} \ge x_i$.



VOPT hält also $r_i \Rightarrow c_i(\text{VOPT}) = U \cdot x_i$.

Vergleicht man VOPT wieder mit einer beliebigen anderen Strategie A^{**}, welche die Seite (im Schritt $z_i > i$) auslagert, so ergibt sich

$$c_i(A^{**}) = U \cdot z_i + R > R = U \cdot h \geq U \cdot x_i = c_i(\text{VOPT}),$$

d. h. VOPT ist auch im Fall b) besser als jede andere Stragegie A^{**}, q.e.d.

8 Das Dateisystem

Da das Dateisystem für Otto Normaluser derjenige Teil des Betriebssystems ist, mit dem er am häufigsten Kontakt hat, darf ein Abriß über seine Funktionsweise hier nicht fehlen. Im folgenden Abschnitt werden wir daher kurz, aber bündig auf die wichtigsten Aspekte eingehen.

8.1 Allgemeines zum Datei-Konzept

Alle Rechneranwendungen sind darauf angewiesen, Informationen über einen längeren Zeitraum und unabhängig von einzelnen Prozessen speichern zu können. Hierzu stehen verschiedene Möglichkeiten zur Verfügung. Um davon abstrahieren zu können, führt man die **Datei** als logische Speichereinheit ein, die vom Betriebssystem auf einen nichtflüchtigen Speicher (z. B. eine Festplatte) abgebildet wird. Die Datei selbst kann man definieren als **eine mit Namen versehene Sammlung zusammengehöriger Informationen, welche auf einem Hintergrundspeicher liegt**. Die Art der in einer Datei gespeicherten Informationen reicht dabei von Quell-, Objekt- und ausführbaren Programmen über numerische Daten, Texte und Bilder bis hin zu einkommenssteuererklärungsrelevanten Daten, wobei natürlich eine Datei je nach ihrem Typ unterschiedliche Strukturen aufweisen kann.

Einer Datei sind stets verschiedene **Attribute** zugeordnet. Sicherlich am wichtigsten ist der **Dateiname**, der dem Nutzer ein Bezugnehmen auf die Datei erlaubt. Oft wird an den Namen noch eine **Typinformation** (als "**extension**") angehängt, anhand derer das Betriebssystem erkennen kann, um welche Art von Datei es sich handelt. Typische Dateinamen lauten damit etwa "prog.c" oder "filesys.doc". Weitere Attribute können beispielsweise die **Größe** einer Datei, ihre **Position** im Speicher, Informationen über **Zugriffsrechte** oder **Zeit, Datum** und **Benutzeridentitäten** sein. Derartige Attribute werden allesamt im Rahmen einer **Verzeichnisstruktur** auf der Festplatte gespeichert.

Zur vollständigen Charakterisierung einer Datei als abstrakten Datentypus sind noch die Operationen auf Dateien anzugeben. Unverzichtbar sind hier das **Erzeugen** und **Löschen** von Dateien, **Lese-** und **Schreiboperationen** sowie die **Umpositionierung** von Zeigern für Lese- oder Schreibzugriffe innerhalb einer Datei. Die Möglichkeit, neue Informationen an das Dateiende anzuhängen, Dateien umzubenennen, zu kopieren usw. werden durch von Betriebssystem zu Betriebssystem variierende Mechanismen realisiert.

Die meisten derartigen Operationen machen es erforderlich, daß das Betriebssystem zunächst im entsprechenden Verzeichnis nach dem Dateinamen sucht. Diese Suche wird dann überflüssig, wenn der Benutzer gezwungen wird, vor dem Zugriff auf eine Datei diese zu **öffnen** (und sie analog nach dem Ende der Benutzung wieder zu **schließen**). Das System kann dann eine Liste geöffneter Dateien (**open-file table**) führen und damit eine aufgerufene Datei wesentlich schneller auffinden.

Der Zugriff selbst kann dann auf unterschiedliche Weise geregelt sein. Am weitesten verbreitet und auch am einfachsten ist der **sequentielle Zugriff**, bei dem eine Eintragung nach der anderen abgearbeitet wird. Der **direkte Zugriff** ermöglicht es dagegen, in der Datei an beliebigen Stellen zu lesen bzw. zu schreiben. Dies ist günstiger, wenn man unmittelbaren Zugang zu großen Mengen von Informationen wünscht,

also etwa in Datenbanken. Viele, aber nicht alle Betriebssysteme, unterstützen beide Zugriffsweisen.

8.2 Verzeichnisstruktur

Da das Dateisystem eines Rechners einen beträchtlichen Umfang erreichen kann, wird man versuchen, es irgendwie sinnvoll zu organisieren. Gewöhnlich geht man dazu in zwei Schritten vor: Zunächst unterteilt man das Dateisystem in verschiedene **Partitionen**. Teilt man eine Festplatte in mehrere Partitionen auf, dann kann man die so entstandenen Bereiche als getrennte Speicher behandeln. Partitionen können sich aber auch über mehrere Festplatten erstrecken und dienen dann zur logischen Gruppierung der Festplatten.

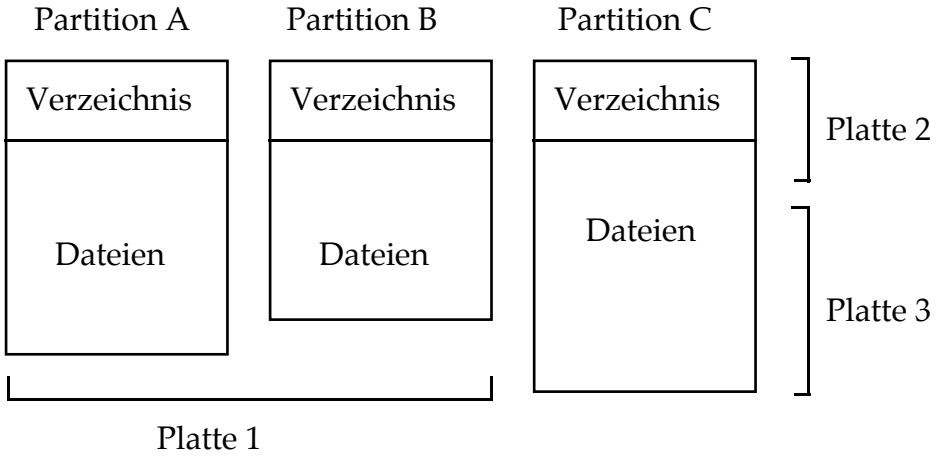


Abb. 8.1: Partionen und Verzeichnisse

Jede Partition enthält dann ihrerseits Informationen über die in ihr enthaltenen Dateien. Diese Informationen stehen in **Verzeichnissen** (directories) und umfassen die Attribute aller Dateien, die in der betreffenden Partition gespeichert sind (insbesondere natürlich deren Namen). Das Verzeichnis selbst kann auf vielerlei Weise strukturiert werden (einige Schemata werden wir sofort kennenlernen), wobei die auf einem Verzeichnis möglichen Operationen zu berücksichtigen sind, beispielsweise das **Suchen** nach einer Datei, ihr **Erzeugen**, **Löschen** und **Umbenennen** sowie das **Auflisten** des Verzeichnisses. Nützlich ist auch, Zugriffsmöglichkeiten auf das gesamte Dateisystem vorzusehen.

8.2.1 Single-Level-Verzeichnis

Dies ist die einfachste Verzeichnisstruktur: Alle Dateien sind in ein und demselben Verzeichnis enthalten:

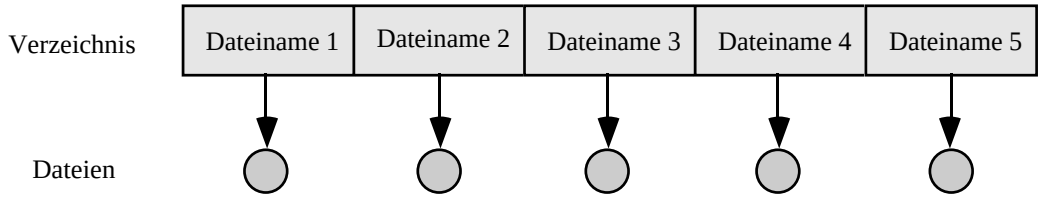


Abb. 8.2: Single-Level-Verzeichnis

Sobald aber eine größere Menge an Dateien vorliegt oder mehrere Benutzer beteiligt sind, werden schnell die beschränkten Möglichkeiten dieses Ansatzes deutlich. Beispielsweise müssen alle Dateinamen eindeutig sein. Dies kann nicht nur im Fall mehrerer Benutzer problematisch werden, sondern auch aufgrund der Tatsache, daß viele Betriebssysteme die Länge der verwendbaren Dateinamen beschränken.

8.2.2 Two-Level-Verzeichnis

Um die Verwirrung über die Dateinamen, wie sie unter mehreren Benutzern leicht auftritt, in den Griff zu bekommen, weist man am einfachsten jedem Benutzer ein eigenes Verzeichnis zu und gelangt so zu einer Verzeichnisstruktur, die zwei Ebenen umfaßt:

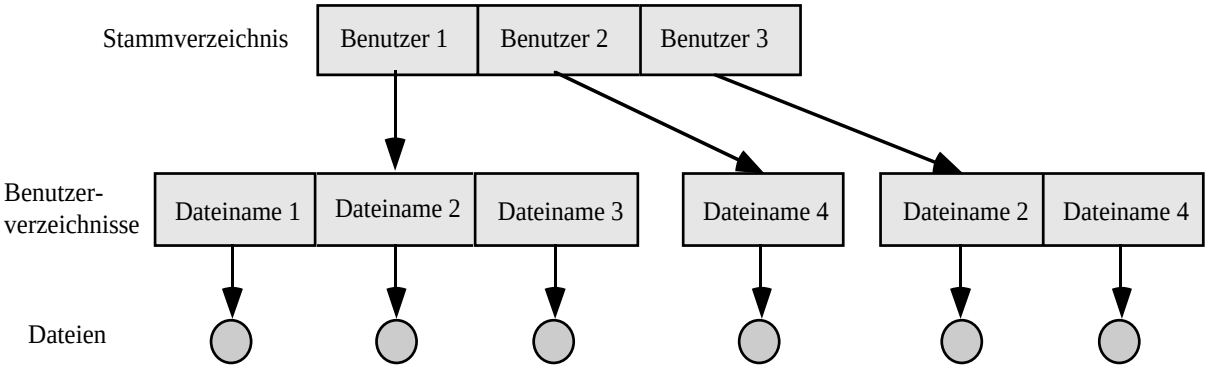


Abb. 8.3: Two-Level-Verzeichnis

Greift ein Benutzer auf eine seiner Dateien zu, so wird nur sein eigenes Verzeichnis danach durchsucht. Daher können jetzt zwei Benutzer gleich benannte Dateien speichern. Andererseits kann ein Benutzer aber auch auf Dateien eines anderen zugreifen, indem er statt des einfachen Dateinamens einen **Pfad** angibt, der aus Benutzer- und Dateiname besteht (z. B. "/Benutzer3/Dateiname2").

8.2.3 Verzeichnisbäume

Diese Idee läßt sich natürlich sofort verallgemeinern und führt zu einer baumartigen Verzeichnisstruktur. Jeder Benutzer kann damit sein Verzeichnis in beliebige Unterverzeichnisse aufteilen und entsprechend seine Dateien strukturiert ablegen. Der Verzeichnisbaum hat ein **Wurzelverzeichnis** (root directory), und jeder Dateiname entspricht dann einem eindeutigen Pfad von der Wurzel durch alle Unterverzeichnisse hin zur betreffenden Datei.

Bei der tagtäglichen Arbeit am Rechner befindet sich ein Benutzer stets in einem **aktuellen Verzeichnis**, das sinnvollerweise möglichst viele der Dateien enthält, die für ihn gerade von Interesse sind. Gibt er nämlich einen einfachen Dateinamen an, so wird nur das aktuelle Verzeichnis daraufhin durchsucht. Um eine Datei außerhalb des aktuellen Verzeichnisses anzusprechen, muß ihr gesamter Pfad angegeben oder das aktuelle Verzeichnis gewechselt werden. Pfade kann man dabei auf zwei Weisen angeben: der **absolute Pfad** beginnt bei der Wurzel und geht abwärts bis zur betreffenden Datei, während der **relative Pfad** vom aktuellen Verzeichnis ausgeht.

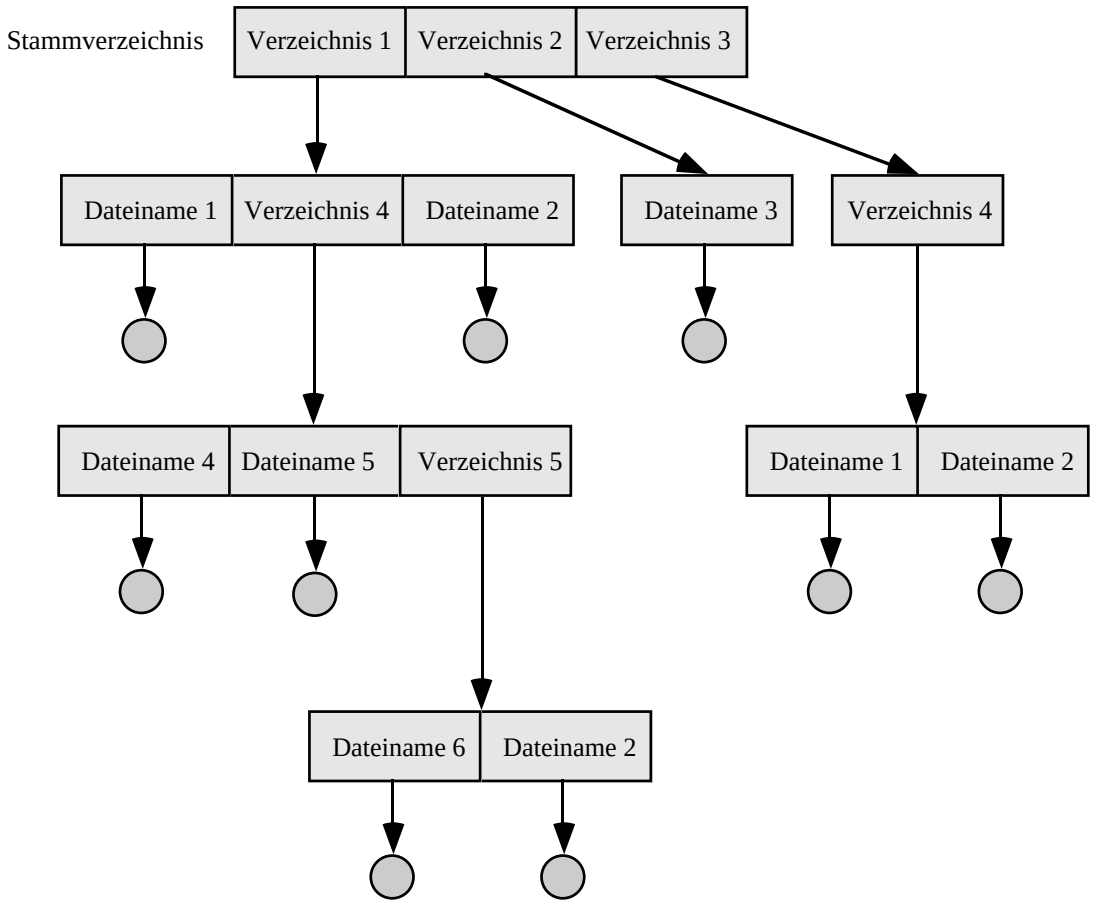


Abb. 8.4: Verzeichnisbaum

Interessant ist die Frage, wie das Löschen eines Verzeichnisses vonstatten gehen soll. Befinden sich keine Dateien oder Unterverzeichnisse darin, so kann man einfach den entsprechenden Eintrag im darüberliegenden Verzeichnis löschen. Ist das Verzeichnis aber nicht leer, so gibt es zwei Möglichkeiten: Bei manchen Betriebssystemen kann man ein Verzeichnis erst dann entfernen, wenn es ganz leer ist (was leicht zu einem beträchtlichen Aufwand führen kann). Gefährlicher, aber effizienter ist es, beim Löschen eines Verzeichnisses automatisch alle darin enthaltenen Dateien und Unterverzeichnisse mit zu entfernen.

8.2.4 Verzeichnisse mit azyklischem und allgemeinem Graphen

Wenngleich die eben besprochene Baumstruktur einem Benutzer leicht den Zugriff auf Dateien eines anderen Benutzers ermöglicht, können in speziellen Situationen andere Strukturen noch effektiver sein. Wenn zum Beispiel zwei Leute an einem gemeinsamen Projekt sitzen und die dafür relevanten Dateien zusammen in einem Unterverzeichnis abgelegt sind, so ist es vorstellbar, daß jeder der beiden dieses Unterverzeichnis in seinem eigenen Verzeichnisbaum haben möchte. Ein solches gemeinsam genutztes Verzeichnis erscheint also im Dateisystem an zwei (oder mehr) Stellen, existiert aber in Wirklichkeit nur einmal. Die beiden Programmierer arbeiten also nicht etwa an zwei unabhängigen Exemplaren derselben Dateien, sondern wenn einer Änderungen ausführt, so sind diese sofort auch dem anderen ersichtlich.

Dies ist allerdings bei Baumstrukturen nicht möglich, wohl aber, wenn das Dateisystem als (gerichteter) **azyklischer Graph** aufgebaut ist:

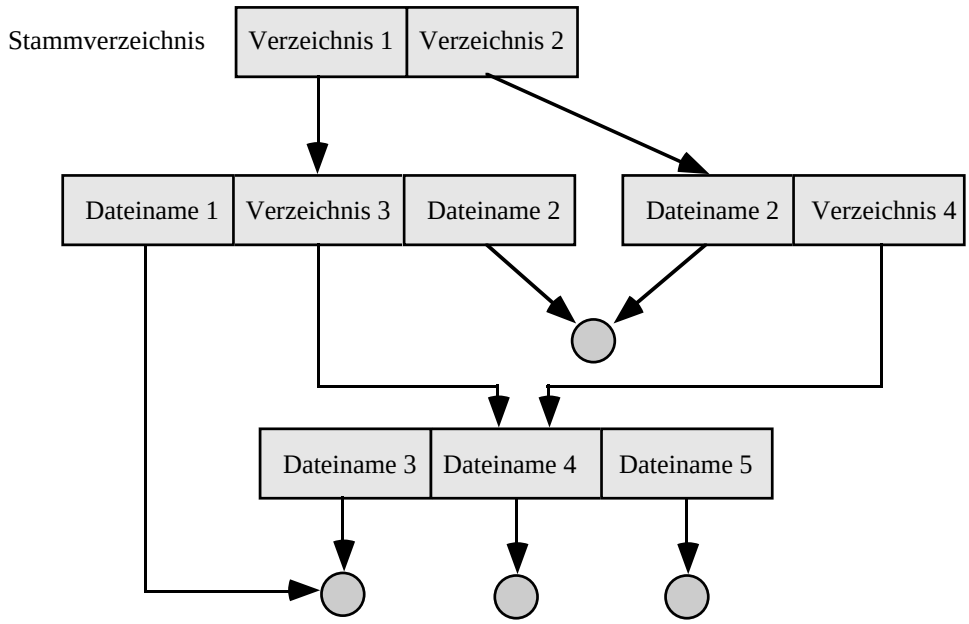


Abb. 8.5: Verzeichnisstruktur als azyklischer Graph

Eine solche Struktur ist flexibler als ein Baum, aber auch komplexer. Insbesondere gibt es keine eindeutigen absoluten Pfade mehr, auch das Löschen eines Verzeichnisses wird komplizierter. Ein schwerwiegendes Problem ist aber auch die Frage, wie man sicherstellen will, daß tatsächlich keine Zyklen vorkommen. Nur dann nämlich bleiben die Algorithmen zur Durchsuchung des Gesamtverzeichnisses nach einer Datei halbwegs einfach. Sobald Zyklen auftreten, kann es vorkommen, daß ein schlecht gebauter Suchalgorithmus in eine unendliche Schleife läuft; auch das Löschen eines Verzeichnisses wird in einem allgemeinen Graphen noch ein Stückchen komplizierter.

8.3 Zur Implementierung eines Dateisystems

Nach den bisher eher theoretischen Betrachtungen wenden wir uns nun langsam Fragen der Implementierung von Dateisystemen zu. Wir wissen bereits, daß sich das Dateisystem auf einem Hintergrundspeicher befindet, der große Mengen von Daten auf Dauer abspeichern kann. Festplatten bieten sich hierfür an, da sie zum einen modifizierte Daten sofort an der Stelle speichern können, an der die ursprünglichen standen, und da sie außerdem den direkten Zugriff auf beliebige Stellen einer Datei ermöglichen. Um den Datentransport zwischen Hauptspeicher und Platte effizient zu gestalten, werden Daten blockweise übertragen. Jeder **Block** besteht dabei aus einem oder mehreren **Sektoren**, deren Größe heutzutage je nach System zwischen 32 und 4096 Bytes schwanken kann.

Das Design eines Dateisystems stellt einen vor zwei unterschiedliche Fragen: Wie soll das Dateisystem für den Benutzer aussehen, und welche Algorithmen und Strukturen sind notwendig, um dieses "logische" Dateisystem auf den physikalischen Speicher abzubilden?

Normalerweise ist ein Dateisystem aus mehreren Schichten aufgebaut, wobei jede Schicht die von den unterhalb liegenden Schichten angebotenen Dienste dazu nutzt, ihrerseits höheren Schichten Dienste zur Verfügung zu stellen.

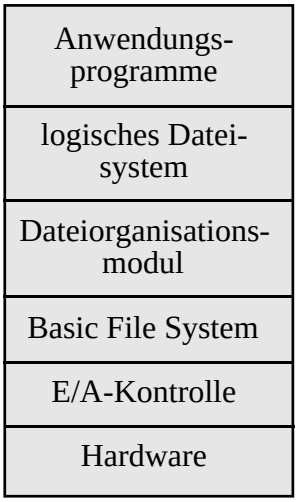


Abb. 8.6: Schichten eines Dateisystems

Die unterste Schicht eines Dateisystems, die **E/A-Kontrolle**, besteht im wesentlichen aus **Treibern**, die die technische Seite der Datenübertragung zwischen Hauptspeicher und Plattensystem regeln. Das **Basic File-System** veranlaßt die Treiber, physikalische Blöcke auf die Festplatte zu schreiben bzw. von dort zu lesen. In der nächsthöheren Ebene findet sich das **Dateiorganisationsmodul**, das die Verbindung zwischen den logischen Dateiblöcken und den physikalischen Blöcken herstellt und insbesondere dazu in der Lage ist, je nach verwendeter Speicherbelegungsstrategie (mehr dazu in Kürze!) logische Blockadressen in physikalische zu übersetzen. Das **logische Dateisystem** schließlich verwendet die oben erläuterte Verzeichnisstruktur dazu, das Dateiorganisationsmodul mit den nötigen Informationen zu einem gegebenen (symbolischen) Dateinamen zu versorgen. Um beispielsweise eine neue Datei zu erzeugen, ruft ein Anwendungsprogramm das logische Dateisystem auf. Dieses liest das entsprechende Verzeichnis in den Speicher, erledigt die Neueintragung und schreibt die Modifikation auf die Festplatte zurück. Auch der E/A-Zugriff auf eine Datei erfolgt über das logische Dateisystem. Damit dieses aber nicht bei jedem Zugriff das gesamte Verzeichnis nach der fraglichen Datei durchsuchen muß, verwendet man meist die bereits erwähnte Open-File-Tabelle, die alle notwendigen Informationen über die derzeit offenen Dateien enthält.

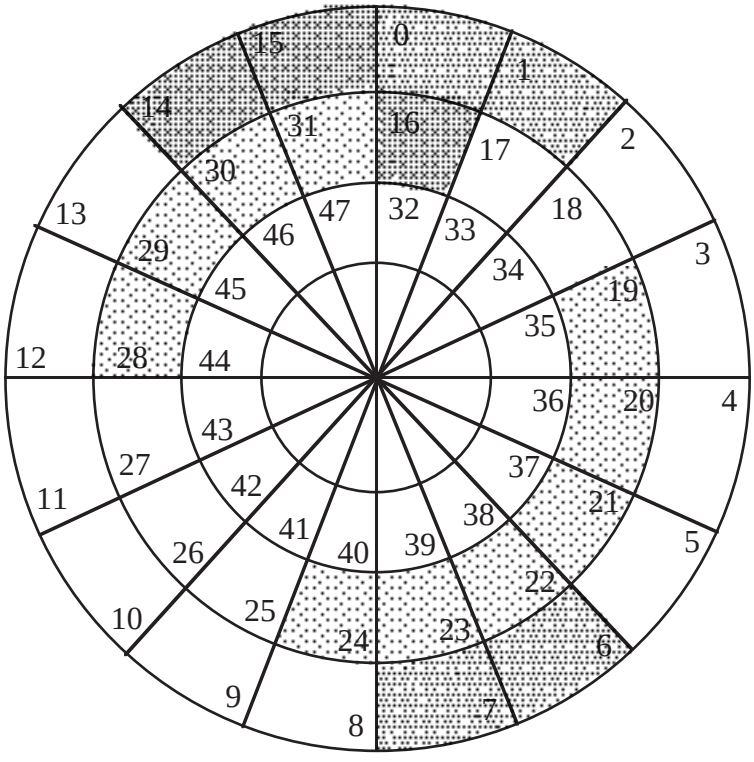
Daß eine Datei geöffnet werden muß, bevor man auf sie zugreifen kann, wissen wir schon. Analog dazu ist ein (zusätzliches) Dateisystem zu **mounten**, bevor es den Prozessen zur Verfügung steht. Hierzu muß das Betriebssystem die Stelle innerhalb der bestehenden Dateistruktur erfahren, an der das neue Dateisystem "einzuhängen" ist.

8.4 Belegungsstrategien

Die Möglichkeit auf Festplatten direkt und sequentiell zugreifen zu können, gibt uns ein gewisses Maß an Flexibilität für die Implementierung der Dateispeicherung. Die Grundfrage lautet, in welcher Weise der Speicherplatz, den die Platte zur Verfügung stellt, auf die Dateien aufgeteilt wird, um einerseits den Speicher gut auszunutzen und andererseits einen schnellen Zugriff auf die Dateien sicherzustellen. Weitverbreitet sind vor allem drei Strategien: die zusammenhängende, die verkettete und die indizierte Belegung.

8.4.1 Zusammenhängende Belegung

Bei dieser Methode belegt jede Datei eine Reihe zusammenhängender Blöcke auf der Festplatte. Die Belegung ist festgelegt, sobald die Adresse des ersten Blocks auf der Platte sowie die Länge der Datei (in Blöcken) bekannt ist. Der Zugriff auf eine zusammenhängend abgelegte Datei ist sowohl sequentiell wie direkt auf einfache Weise möglich. Beim sequentiellen Zugriff genügt es, sich an die Adresse des zuletzt referenzierten Blockes zu erinnern. Will man direkt auf den (logischen) Block *i* der Datei zugreifen und beginnt diese beim Block *b* der Festplatte, so greift man einfach auf den (physikalischen) Block *b+i* zu.



Verzeichnis

Datei	Start	Länge
Name 1	0	2
Name 2	14	3
Name 3	19	6
Name 4	28	4
Name 5	6	2

Abb. 8.7: Zusammenhängende Belegung eines Datenträgers

Ein Hauptproblem bei diesem Vorgehen ist die Frage, wie man genügend Speicherplatz für eine neue Datei ausfindig macht. Derartige Fragen haben wir bereits früher im Abschnitt über dynamische Speicherbelegung diskutiert. Damals stellten wir fest, daß "First-fit" und "Best-fit" die am meisten verbreiteten Strategien für dieses

Problem sind. Natürlich laufen diese Algorithmen auf "**externe Fragmentierung**" des Speicherplatzes hinaus: Durch Ablegen und Entfernen der Dateien wird im Laufe der Zeit der freie Speicherplatz in kleine Stücke zerbrochen, so daß irgendwann u. U. selbst das größte freie Stück für eine neu zu speichernde Datei nicht mehr ausreicht.

Ein weiteres ernsthaftes Problem ist die Frage, wieviel Platz eine Datei denn überhaupt braucht. Denn um ihr genügend Platz zuweisen zu können, muß die Größe einer Datei zum Zeitpunkt ihrer Erzeugung bekannt sein. Reservieren wir ihr zuwenig Platz, dann ist es u. U. nicht möglich, die Datei später zu erweitern. Dies führt dann entweder zum Abbruch des entsprechenden Prozesses mit allen Konsequenzen (insbesondere wird der Benutzer beim Neustart auf Nummer sicher gehen und den Speicherbedarf deutlich überschätzen, was im Endeffekt auf eine mehr oder weniger heftige Speicherplatzverschwendung hinausläuft) oder dazu, daß für die erweiterte Datei ein neues, größeres "Loch" gesucht und die bisherigen Daten dorthin kopiert werden müssen. In diesem Fall arbeitet das System trotz des aufgetretenen Problems wenigstens weiter, jedoch unter einem gewissen Geschwindigkeitsverlust.

Selbst wenn der totale Speicherbedarf für eine Datei im vorhinein bekannt ist, kann die Vorwegbelegung ineffizient sein. Hat man nämlich eine Datei, die langsam über einen größeren Zeitraum (z. B. ein Monat oder ein Jahr) stetig anwächst, so ist es nicht unbedingt sinnvoll, den Endbedarf von Anfang an zu reservieren, denn dann bleiben große Mengen an Speicherplatz über lange Zeit ungenutzt (ein Effekt, den man als "**interne Fragmentierung**" bezeichnet).

8.4.2 Verkettete Belegung

Die Methode der verketteten Belegung behebt die aufgetretenen Probleme der zusammenhängenden Belegung. Eine Datei ist hier als verkettete Liste von Blöcken realisiert, wobei diese Blöcke willkürlich über die ganze Platte verstreut sein können. Das Verzeichnis enthält dann einen Zeiger auf den ersten und einen auf den letzten Block der Datei, und jeder Block enthält einen Zeiger auf den nächsten Block der Datei.

Die Erzeugung einer neuen Datei besteht einfach in einer neuen Eintragung ins Verzeichnis, wobei die Zeiger zu `nil` (was einer leeren Datei entspricht) und die Dateilänge zu 0 initialisiert werden. Eine Schreiboperation kann dann bewirken, daß ein freier Plattenblock gefunden werden muß und mittels eines Zeigers ans Ende der bisherigen Datei angehängt wird. Beim Lesen der Datei folgt man einfach den Zeigern von Block zu Block.

Bei der verketteten Belegung kommt es weder zu externer Fragmentierung noch zu Verschwendung von Speicherplatz. Jeder freie Block auf der Platte kann genutzt werden, und es ist auch nicht nötig, die Länge einer Datei am Anfang verbindlich anzugeben, vielmehr kann eine Datei weiter und weiter wachsen, solange freie Blöcke auf der Platte vorhanden sind.

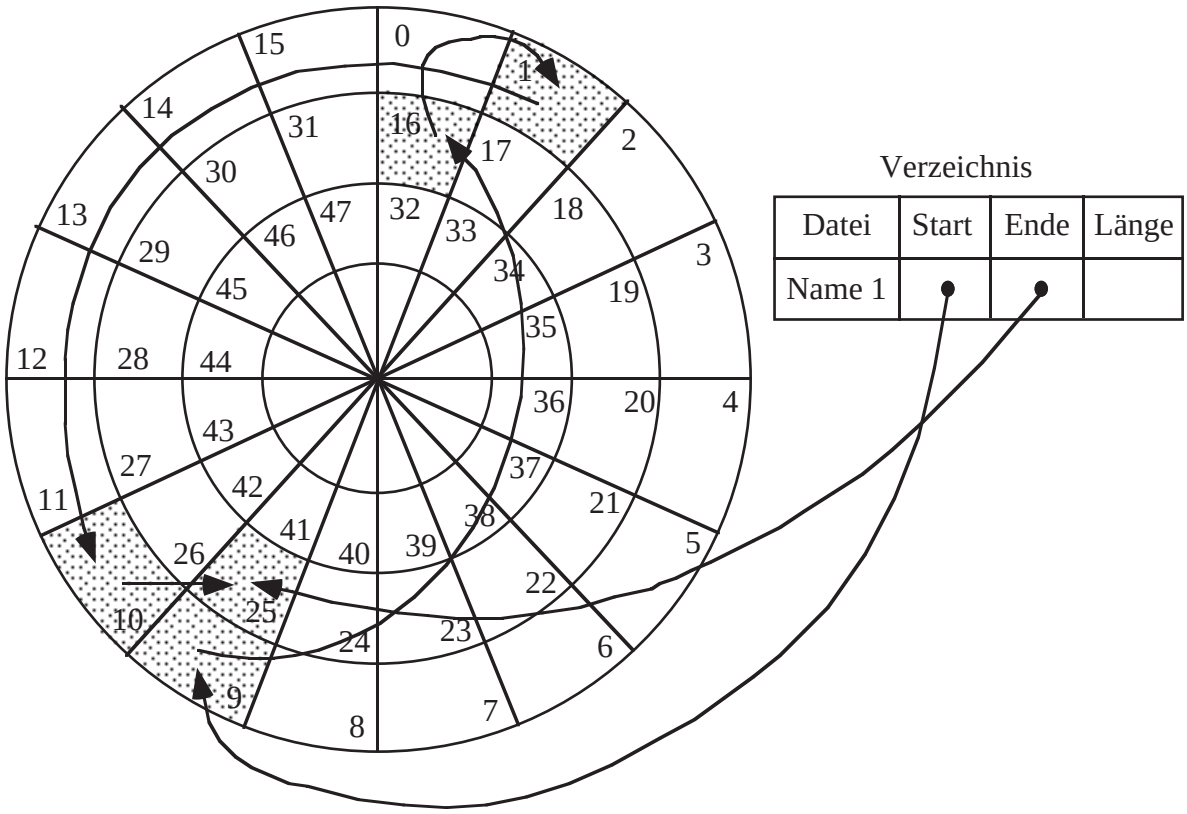


Abb. 8.8: Verkettete Belegung auf dem Datenträger

Allerdings hat diese Strategie auch Nachteile. Vor allem ist sie nur für sequentielle Zugriffe geeignet. Will man nämlich den *i*-ten Block einer Datei finden, so ist man gezwungen, am Beginn der Datei zu starten und den Zeigern solange zu folgen, bis man am fraglichen Block angekommen ist. Ein weiterer Nachteil ist, daß die Zeiger selbst Speicherplatz verbrauchen, den man sonst anderweitig nutzen könnte. Dieses Problem löst man üblicherweise dadurch, daß man mehrere Blöcke zu einem **Cluster** zusammenfaßt und diesen als Speichereinheit anstelle der Blöcke verwendet. Man braucht weniger Platz für die Zeiger, die Abbildung der logischen auf die physikalischen Blöcke bleibt einfach, und dennoch erhöht sich der Systemdurchsatz. Freilich geht dies auf Kosten einer höheren internen Fragmentierung, da Platz verschwendet wird, sobald ein Cluster nur teilweise voll ist.

Ein letzter Nachteil dieses Verfahrens wird offensichtlich, wenn man die Frage nach der Zuverlässigkeit stellt. Da eine Datei - durch Zeiger zusammengehalten - über die ganze Platte verstreut gespeichert ist, hat der Verlust oder die Beschädigung eines einzigen Zeigers schnell fatale Auswirkungen, ebenso wie die versehentliche Verwendung eines falschen Zeigers aufgrund etwa eines Software-Fehlers. Sich dagegen zu sichern bringt oft großen Overhead mit sich.

Eine wichtige Variante beruht auf der Verwendung einer Dateibelegungstabelle (**File-Allocation Table, FAT**). Hierbei befindet sich am Anfang jeder Partition eine Tabelle, in der für jeden Block der Partition ein Feld vorgesehen ist. Diese Tabelle funktioniert ähnlich wie eine verkettete Liste: Ein Dateieintrag im Directory enthält die Nummer *m* des ersten (physikalischen) Blocks, mit dem die gespeicherte Datei beginnt. An *m*-ter Stelle der Tabelle ist dann die Nummer des folgenden Blocks angegeben. Diese Kette geht weiter bis zum letzten Block der Datei, der in der Tabelle einen speziellen EOF-Wert zugeordnet bekommt. Freie Blöcke werden in der Tabelle durch den Wert 0 gekennzeichnet.

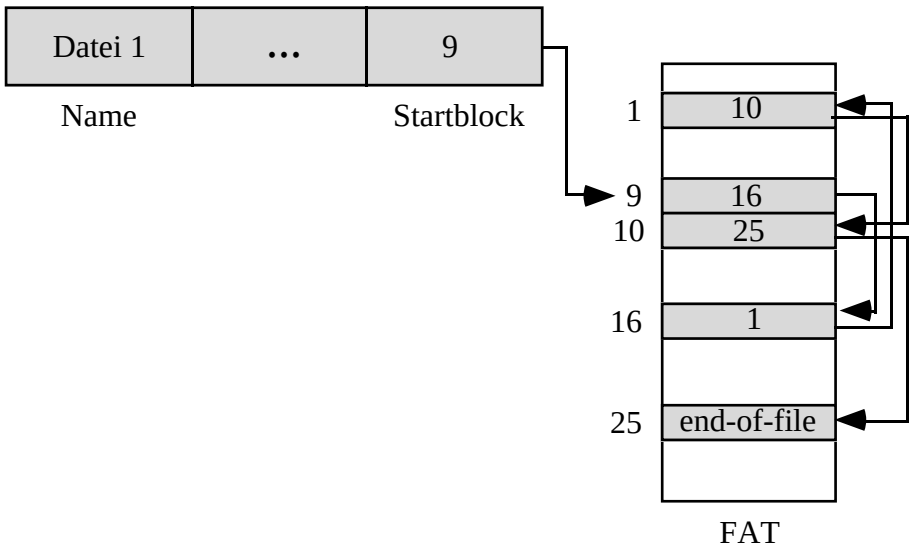


Abb. 8.9: File-Allocation Table

Zu bedenken ist allerdings, daß die Verwendung einer FAT für die Platte erhöhten Suchaufwand mit sich bringt, da der Lese-/Schreibkopf ständig zwischen der Tabelle und den entsprechenden Blöcken hin- und herspringen muß. Andererseits wird der direkte Zugriff dadurch wesentlich vereinfacht, da die Plazierung eines willkürlich herausgegriffenen Blocks schnell anhand der FAT festgestellt werden kann. Eine weitere Schwierigkeit besteht darin, daß ein Verlust der FAT zum Verlust der gesamten Datei führt.

8.4.3 Indizierte Belegung

Wir haben gesehen, daß die verkettete Belegung die Schwierigkeiten der zusammenhängenden Belegung in den Griff bekommt, aber nicht für einen effizienten direkten Zugriff geeignet ist. Die indizierte Belegung löst dieses Problem dadurch, daß sie alle Zeiger an einem Platz, dem **Indexblock**, zusammenfaßt. Jede Datei hat ihren eigenen Indexblock, der aus lauter Blockadressen auf der Platte besteht. Dabei zeigt der *i*-te Eintrag im Indexblock auf den *i*-ten (physikalischen) Block der gespeicherten Datei. Das Directory enthält dann nur noch die Adresse des Indexblocks. Um den *i*-ten Dateiblock zu lesen, schaut man dann an der *i*-ten Stelle des Indexblocks nach und kann dann direkt auf die gewünschte Stelle in der Datei zugreifen.

Somit unterstützt indizierte Belegung einen direkten Zugriff, ohne unter externer Fragmentierung zu leiden, da jeder freie Block auf der Platte zur Speicherung verwendet werden kann. Was "verschwendet" wird, ist Speicherplatz für den Indexblock, insbesondere wenn ein kompletter Block reserviert werden muß, obwohl die zugehörige Datei relativ kurz ist.

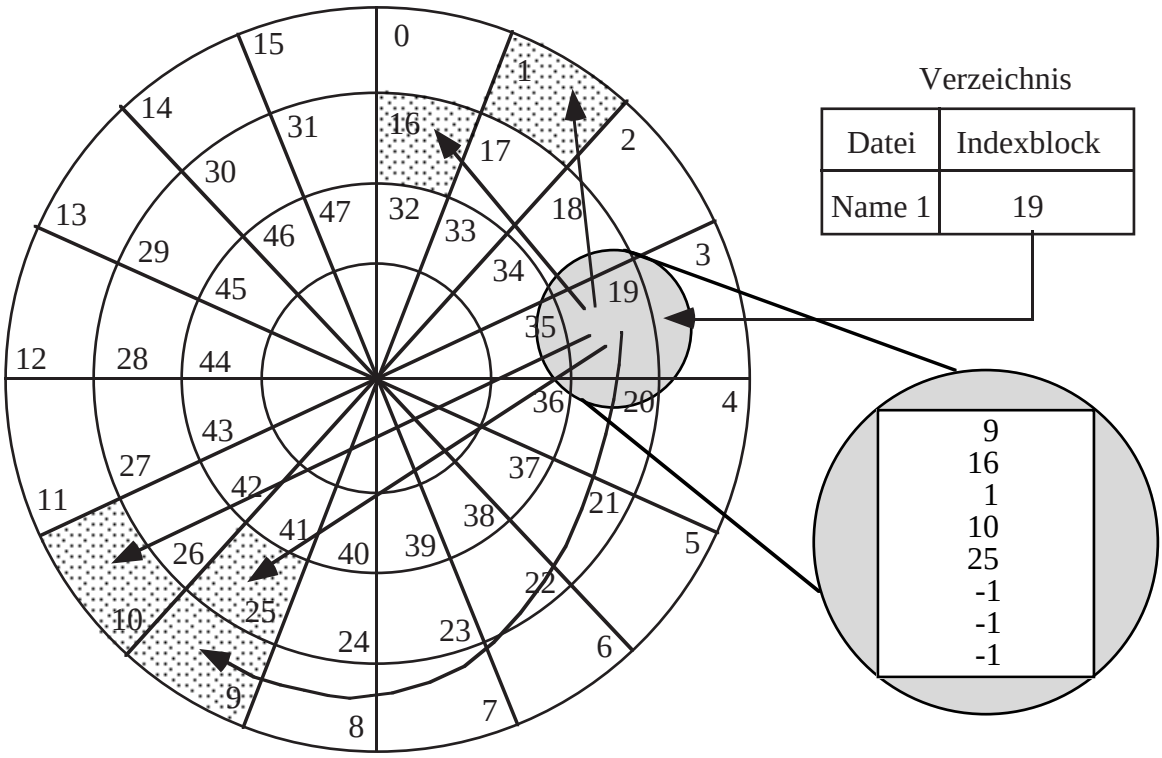


Abb. 8.10: Indizierte Belegung auf dem Datenträger

Wie groß sollte also ein Indexblock sein? Da jede Datei einen Indexblock besitzt, erscheint es ratsam, diese nicht zu groß zu machen. Zu kleine Indexblöcke wiederum können u. U. nicht alle für eine Datei benötigten Zeiger aufnehmen, so daß für die Speicherung großer Dateien Extramechanismen vorzusehen sind: Beispielsweise könnte man eine Verkettung von Indexblöcken vorsehen. Üblicherweise würde dann einer Datei ein Indexblock zugeordnet, an dessen letzter Stelle entweder ein `nil`-Zeiger (kleine Datei) oder ein Zeiger auf einen weiteren Indexblock (große Datei) zu finden ist. Oder man führt eine neue Ebene von Indexblöcken ein, deren Einträge auf Indexblöcke zeigen, die ihrerseits dann auf Dateiblöcke zeigen. Dieses Schema ist hierarchisch beliebig weit ausbaufähig. Eine dritte Möglichkeit kombiniert die beiden vorherigen Ansätze, indem sie einen Indexblock aufteilt in einen Bereich, der Adressen von Dateiblöcken enthält, und einen kleineren Bereich, in dem die Adressen weiterer Indexblöcke stehen können.

8.5 Speicherplatzverwaltung

Da auf einer Platte Speicherplatz nur in begrenztem Ausmaß vorhanden ist, sieht man sich dazu gezwungen, diesen - so es geht - wiederzuverwenden. Um über den freien Speicherplatz auf dem laufenden zu sein, bedient man sich einer Tabelle, der sogenannten "**free-space list**", in der alle Plattenblöcke verzeichnet stehen, die nicht durch eine Datei oder ein Verzeichnis belegt sind. Bei der Erzeugung einer Datei sucht man dann in dieser Tabelle nach entsprechend viel Speicherplatz, weist diesen der neuen Datei zu und wirft ihn zugleich aus der free-space list heraus. Umgekehrt wird anläßlich eines Dateilöschens der von ihr bisher belegte Platz in die Tabelle aufgenommen.

Trotz ihres Namens wird die free-space list nicht immer als Liste implementiert, sondern zum Beispiel als **Bit-Vektor**. Hierbei wird jeder Block durch ein Bit repräsen-

tiert. Ist der Block frei, so wird das Bit auf 1 gesetzt, andernfalls auf 0. Das ist relativ einfach, zudem ist es auf effiziente Weise möglich, den ersten freien Block oder *n* freie Blöcke hintereinander ausfindig zu machen. Man kann aber auch alle freien Blöcke auf der Platte zu einer **verketteten Liste** zusammenfügen und einen Zeiger auf ihren ersten Block an einer besonderen Stelle auf der Platte anbringen. Dieser erste Block enthält dann einen Zeiger auf den nächsten freien Block usw. Eine Modifikation davon besteht in der **Gruppierung** freier Blöcke. Hierbei sind im ersten freien Block *n* Adressen gespeichert. Die ersten *n-1* dieser Adressen zeigen auf tatsächlich freie Blöcke, während die letzte Adresse auf einen Block zeigt, der seinerseits *n-1* Adressen freier Blöcke und eine weitere Blockadresse enthält usw. Diese Implementierung ist deshalb so wichtig, da mit ihrer Hilfe auf einen Blick die Adressen einer großen Anzahl freier Blöcke herausgefunden werden können. Schließlich kann man auch noch ausnutzen, daß (insbesondere bei der zusammenhängenden Belegung oder beim Clustern) in der Regel eine größere Anzahl zusammenhängender Blöcke gleichzeitig frei wird, so daß es genügt, die Adresse des ersten derartigen Blocks parat zu haben und dann durch einfaches **Zählen** die Anzahl *n* freier zusammenhängender Blöcke festzustellen, anstatt etwa eine Liste von *n* einzelnen Blöcken zu verwalten.

8.6 Implementierung von Verzeichnissen

Da die Auswahl von Algorithmen zur Verzeichnisverwaltung großen Einfluß auf Effizienz, Leistungsfähigkeit und Zuverlässigkeit des Dateisystems haben kann, soll abschließend noch kurz hierauf eingegangen werden.

8.6.1 Lineare Liste

Am einfachsten implementiert man ein Verzeichnis als **lineare Liste**. Um einen bestimmten Eintrag zu finden, muß man diese Liste dann der Länge nach durchsuchen, was sich ziemlich zeitaufwendig gestalten kann. Bei der Erzeugung einer neuen Datei ist zunächst die Liste daraufhin zu durchsuchen, ob der vorgesehene Dateiname noch nicht verwendet worden ist. Dann hängt man den Zeiger auf die neue Datei einfach am Schluß der Liste an. Beim Löschen einer Datei wird ihr Eintrag aus der Liste entfernt. Den so gewonnenen freien Platz kann man auf dreierlei Weise nutzen: Entweder man markiert ihn als "frei", oder man trägt ihn in eine Liste freier Verzeichnisplätze ein. Dritte Alternative ist es, den letzten Eintrag des Verzeichnisses an die freigewordene Stelle zu kopieren und dadurch die Liste zu verkürzen.

Der wirkliche Nachteil dabei ist die Linearität der Suche nach einer bestimmten Datei, weil Informationen aus dem Verzeichnis sehr oft gebraucht werden und eine langsame Implementierung an dieser Stelle sich schmerzlich bemerkbar machen kann. Viele Betriebssysteme implementieren daher einen Software-Cache, um die zuletzt aufgerufenen Einträge aus dem Verzeichnis zu speichern. Man kann die durchschnittliche Suchzeit auch durch Verwendung einer sortierten Liste verringern, aber der Implementierungsaufwand und die Komplexität von Such- und Verwaltungsalgorithmen steigt dadurch natürlich beträchtlich.

8.6.2 Hash-Tabelle

Ein anderer Ansatz verwendet ein Hash-Verfahren zur Implementierung von Verzeichnissen. Unter einem **Hash** versteht man dabei eine Abbildungsfunktion

$$h: \quad \{\text{Schlüssel}\} \rightarrow \{\text{Speicherpositionen in der Hash - Tabelle}\}$$

Die Abbildung *h* sollte leicht zu berechnen sein und den verfügbaren Speicherraum möglichst gleichmäßig ausnutzen. Injektivität wird nicht verlangt, d. h. unter Umständen ergibt die Anwendung von *h* auf zwei verschiedene Schlüssel dasselbe Resultat.
In unserem Fall wandelt *h* jeden Dateinamen (= Schlüssel) in eine Position innerhalb einer **Hash-Tabelle** um. Der Tabelleneintrag an der ermittelten Position ist dann ein Zeiger auf die Stelle innerhalb des immer noch als lineare Liste vorliegenden Verzeichnisses, an der die fragliche Datei (samt ihren Attributen) zu finden ist.

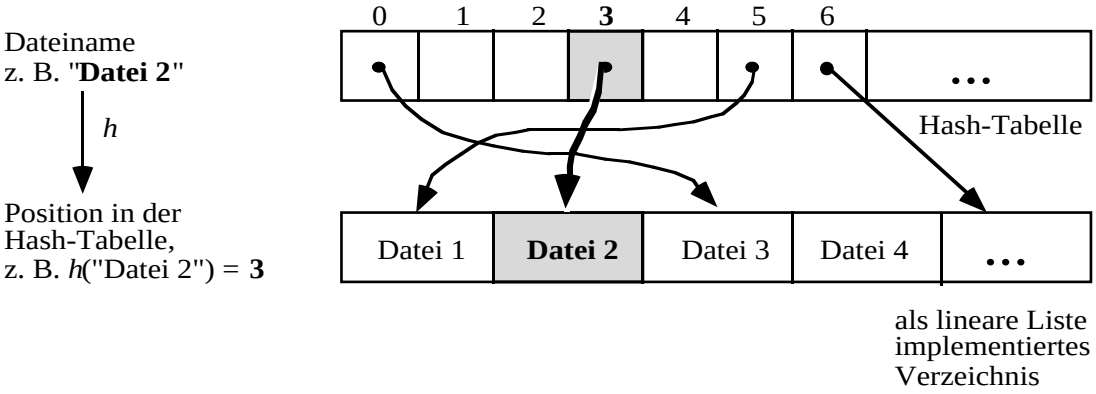


Abb. 8.11: Beispiel einer Hash-Tabelle

Dadurch wird die Suchzeit bei geschicktem Vorgehen enorm gedrosselt. Auch das Einfügen und Entfernen von Einträgen ist relativ problemlos, wenngleich Vorkehrungen für den Fall einer Kollision (bei der zwei unterschiedlichen Dateinamen über die Hash-Funktion die gleiche Tabellenposition zugeordnet wird) getroffen werden müssen. Nachteilig macht sich bei diesem Ansatz bemerkbar, daß die Hash-Tabelle nur einen begrenzten Umfang besitzt und die Hash-Funktion obendrein von dieser Tabellengröße abhängt, d. h. wenn sich die bislang verwendete Tabelle als zu klein erweist und man auf eine größere umsteigt, so ist auch die Hash-Funktion komplett neu zu berechnen.

Ergänzendes

9 Binden und Laden von Programmen bzw. Programm-Modulen

Programme werden i. a. nicht an einem Stück programmiert, sondern bestehen aus einzelnen Modulen, die untereinander Beziehungen aufweisen. Um aus diesen Modulen nun ein lauffähiges Programm zu erzeugen, müssen sie in einer geeigneten Weise miteinander verknüpft werden. Diese Aufgabe übernimmt gewöhnlich ein Programm namens **Binder** oder **Linker** (nicht politisch zu interpretieren!).
Die Aufgabe des Binders ist es, die Module in der gewünschten Reihenfolge zusammenzufassen. Die zunächst symbolischen Bezüge zwischen den Symbolen müssen dabei konkretisiert bzw. aufgelöst werden.
Hat der Binder seine Arbeit beendet, muß das Programm zur Ausführung in den Hauptspeicher gebracht werden. Diese Aufgabe wird vom **Lader** übernommen.

Zunächst stellt sich die Frage, in welcher Programmphase gebunden werden soll. Denkbar wären folgende Möglichkeiten:

Programmphase	**Binden günstig?**
Programmierung	Nein, da die Module dann nicht mehr unabhängig sind.
Wenn Quellcode komplett	Nein, da gleiche Module bei Wiederverwendung mehrfach übersetzt werden müßten.
Übersetzung	Nein, aus demselben Grund.
Wenn Module übersetzt vorliegen (Linkage Editor)	Günstig, da die Möglichkeiten zur Zusammensetzung flexibel bleiben und jetzt genügend Zeit vorhanden ist.
Beim Laden (Linkage Loader)	Gebräuchlich, jedoch mit dem Nachteil, daß man vor dem Start eines Programms das Binden abwarten muß.
Ausführung	Nein, obwohl sehr flexibel würde die Geschwindigkeit der Programme zu sehr leiden.

Abb. 9.1: Bindungsmöglichkeiten in den verschiedenen Programmphasen

Die folgende Graphik soll die Arbeitsweise des Linkage Editors veranschaulichen.

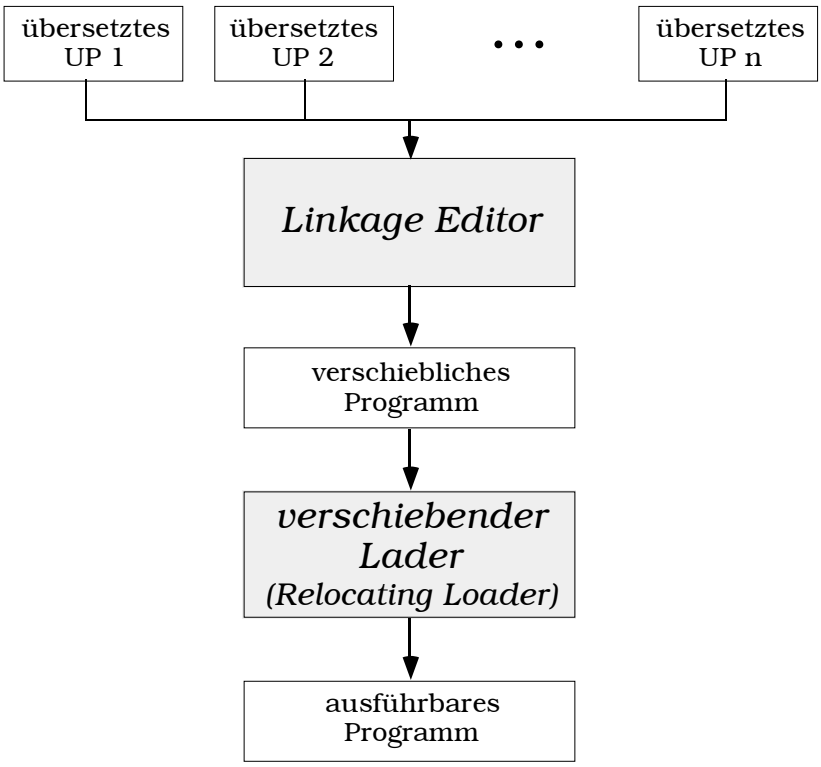


Abb. 9.2: Prinzip des Linkage Editors

Anhand eines vereinfachten Beispiels wollen wir die Arbeitsweise eines Linkage Editors studieren. Dabei lehnen wir uns an den Binder eines IBM-Großrechners an. Die durchzuführenden Schritte sind aber prinzipiell in allen Bindern nahezu identisch.
Als **Eingabe** erhält der **Linkage Editor** die zu bindenden Objektmodule - dies sind bereits einzeln übersetzte Unterprogramme - sowie eventuelle Steuerkommandos. Die **Ausgabe** besteht dann aus einem lauffähigen Programm, also einem zusammengebundenen Objekt, welches durch einen **Relocating Loader** geladen und ausführbar gemacht wird.
Der Aufbau eines Objektmoduls sieht beispielsweise folgendermaßen aus:

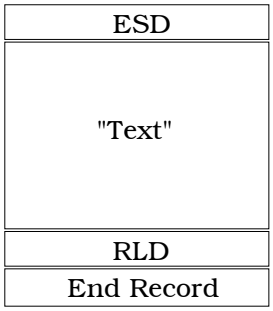


Abb. 9.3: Prinzipieller Aufbau eines Objektmoduls

Das **ESD (External Symbol Dictionary)** ist eine Liste der im folgenden Teil "Text" vorkommenden externen Symbolen. Im **Relocation Dictionary (RLD)** stehen diejenigen Bereiche im Text, die besonders behandelt werden müssen.

Ein **externes Symbol** ist entweder ein externer Name oder eine externe Referenz. Unter einem **externen Namen** versteht man den Namen, unter dem das Modul von aus-

sen erreichbar ist. Dies kann zum einen der **Name des Unterprogramms (Name of Control Section)** sein oder ein Label, welches eine symbolisch markierte **Einsprungstelle (Entry Point)** darstellt. Eine **externe Referenz** ist ein Name, der einen Aufruf nach außen bewirkt, also einen Bezug zu allen externen Namen eines anderen Moduls herstellt.

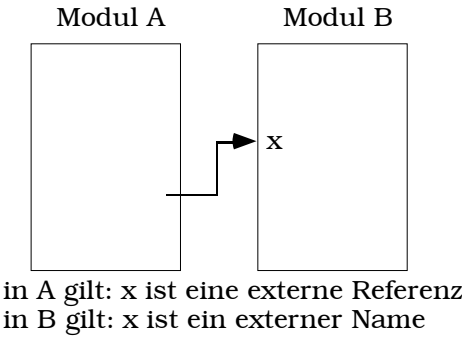


Abb. 9.4: Beispiel einer externen Referenz

Die Aufgabe des Binders ist es, die eben beschriebenen Bezüge konkret herzustellen. Wenden wir uns nun dem Aufbau der ESD-Einträge und des Relocation Dictionary zu. Der ESD-Eintrag sieht bei unserem Beispielbinder wie folgt aus:

Name	Typ	XXXXX	XXXXX

Die beiden letzten Felder sind dabei vom Typ des Eintrags abhängig. Die drei wichtigsten Typen sollen kurz vorgestellt werden:

a) **SD (Section definition)**
SD ist ein direkter Verweis auf ein Unterprogramm. Der entsprechende ESD-Eintrag lautet:

Name	SD	Anfangsadresse	Länge

b) **LD (Label Definition)**

Name	LD	Anfangsadresse	Pointer

Ein ESD-Eintrag vom Typ Label Definition verweist auf einen Entry Point. Die Anfangsadresse ist relativ zum Modulanfang. Der Pointer verweist auf die Zeile des ESD-Eintrags derjenigen Control Section (i. a. Unterprogramm), welche den betreffenden Namen enthält.

c) **ER (External Reference)**

Externes Symbol	ER	leer	leer

Mit diesem Eintrag wird lediglich gekennzeichnet, daß das verwendete Symbol auf ein extern definiertes Bezug nimmt.

Das Relocation Dictionary (RLD) enthält alle Adressen, bei denen beim Binden bzw. Laden noch Anpassungen vorgenommen werden müssen. Dies ist beispielsweise bei Adreßbefehlen der Fall. Eines RLD-Eintrag hat den folgenden Aufbau:

Relocation Pointer	Position Pointer	Flag	Address

Der "Relocation Pointer" zeigt auf den entsprechenden ESD-Eintrag. Der "Position Pointer" verweist auf den ESD-Eintrag der Control Section, in der die betreffende Konstante definiert ist. Das "Flag" zeigt die Art eines Befehls (siehe weiter unten) an, und in "Address" steht der relative Abstand der Konstanten zum Programmbeginn. Befehle, die eine Verschiebung erfordern, sind beispielsweise

Befehl	*Bedeutung*	*Wirkung*
DC A(X)	Define Constant Address	Inhalt dieser Zelle ist zur Ausführungszeit die aktuelle Adresse von X ,wobei X ein lokaler Name ist.
DC V(X)	Define Constant Variable	Inhalt ist die aktuelle Adresse von X, jedoch ist X hier ein externes Symbol.

Die verschiedenen Einträge sollen nun an einem Beispiel erläutert werden. Dazu betrachten wir die zwei Module CSECT A und CSECT B.

Beispielmodul CSECT A

A	SD	0	500	ESD von A
BILL	LD	200	1	
B	ER			

CSECT A ENTRY BILL	*Adr. rel .zu A*
BILL	200
DC V(B) /* vorläufig besetzt mit 0 */	300
END	499

3	1	V	300	RLD von A

Beispielmodul CSECT B

B	SD	0	300	ESD von B
BILL	ER			

CSECT B	*Adr. rel .zu B*
JOE	100
DC A(JOE) /* vorläufig besetzt mit 100 */	200
DC V(BILL) /* vorläufig besetzt mit 0 */	204
END	299

1	1	A	200	RLD von B
2	1	V	204	

Der Binder (Linkage Editor) erhält nun als Eingabe die Textmodule und jeweils die zugehörigen ESD und RLD. Daraus produziert er ein gemeinsames Modul mit Adressen, die relativ zueinander richtig positioniert sind. Ferner wird eine **Composite ESD** aus allen Modulen erstellt.

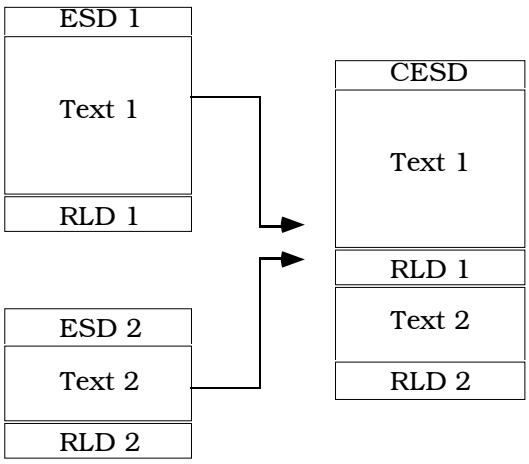


Abb. 9.5: Prinzipieller Aufbau eines Composite ESD

Das gebundene Modul hat folgendes Aussehen:

A	SD	0	500	CESD
BILL	LD	200	1	
B	SD	500	300	

CSECT A ENTRY BILL				*relative Adresse*
BILL				200
DC V(B)				300
END				499
3	1	V	300	RLD
CSECT B				500
JOE				600
DC A(JOE)				700
DC V(BILL)				704
END				799
3	3	A	700	RLD
2	3	V	704	

Bis jetzt wurde ein Modul erzeugt, das lediglich relative Adressen enthält. Dieses Modul muß noch an eine freie Stelle im Hauptspeicher geladen werden. Bei diesem Ladevorgang sind die relativen Adressen in die endgültigen absoluten Adressen umzurechnen. Diese Aufgaben übernimmt der Lader.
Man unterscheidet zwischen absoluten und verschiebenden Ladern. Der **absolute Lader** transportiert das Programm unverändert an eine fest vorgegebene Adresse im Hauptspeicher.
Der **verschiebende Lader** (relocating Loader) bestimmt zunächst einen ausreichend großen freien Speicher und plaziert das Programm dann an dieser Stelle. Die Adressen im Programm müssen mit Hilfe der Basisadresse zuzüglich der relativen Adresse berechnet werden.
Wir wenden jetzt den Vorgang des verschiebendes Ladens auf unser gebundenes Beispielmodul an. Dabei gehen CESD und RLD verloren. Im Beispiel ist ein zusammenhängender Speicherbereich von mindestens 800 Byte zu finden. Wir gehen davon aus, daß ein solcher Bereich ab der Speicherzelle 2000 gefunden wird und der Relocating Loader das gebundene Modul dorthin lädt. Nach dem Laden sind alle Bezüge zwischen den Modulen, die noch offen waren, aufgelöst. Das Plazierungsergebnis sieht wie folgt aus:

	abs. Adresse
CSECT A ENTRY BILL	
BILL	2200
DC V(B) /* = 2500 */	2300
END	
CSECT B	2500
JOE	2600
DC A(JOE) /* = 2600 */	2700
DC V(BILL) /* = 2200 */	2704
END	2799

10 Schutz und Sicherheit

10.1 Schutzmechanismen

Nicht jeder Benutzer oder Prozeß, darf in einem Multiprogramming-System uneingeschränkt über alle Ressourcen eines Rechners verfügen. So müssen z. B. Dateien anderer Benutzer, deren Speicherbereiche etc. vor unerlaubtem Zugriff geschützt werden. Solche Maßnahmen können mehrere Gründe haben. Daten könnten z. B. unerlaubt gelesen, geändert oder gelöscht werden oder ein Prozeß könnte sich Prioritäten oder Speicherbereiche zuordnen, die ihm gar nicht zustehen. Man unterscheidet i. a. zwischen der Politik des Schutzes (Strategie, nach der Rechte usw. verteilt werden) und den Mechanismen, mit denen diese Politik in die Tat umgesetzt wird. Da die Politik je nach System und Benutzer sehr unterschiedlich sein kann, sollen hier nur die Mechanismen betrachtet werden.

10.1.1 Schutzbereiche

Abstrakt kann ein Computer als eine Sammlung von Prozessen und Objekten definiert werden. Prozesse wurden bereits vorgestellt. **Objekte** sind die Teile des Rechners, die von Prozessen genutzt werden können. Dazu gehören zum einen Hardwarekomponenten, z. B. Speicher, CPU, Festplatte, andererseits aber auch Software wie z. B. Programme oder andere Dateien. Jedes Objekt ist durch einen eindeutigen Namen gekennzeichnet, und auf ihm können objektabhängige **Operationen** ausgeführt werden. Beispielsweise kann ein Speicher an einer bestimmten Stelle gelesen oder ein Programm gestartet werden. Damit ein Prozeß keine unerlaubten Operationen auf einem Objekt durchführen kann, muß das System zuerst einmal Kenntnis darüber haben, was ein Prozeß darf und was nicht. Dazu wird jedem Prozeß (oder Benutzer oder beiden) ein sogenannter Schutzbereich zugeordnet. Ein **Schutzbereich** ist eine Menge von geordneten Paaren, bestehend aus Objektname und einer Menge von **Rechten**, gewisse Operationen ausführen zu dürfen. Ein Element des Schutzbereiches könnte z. B. das Paar (Datei D, {lesen, schreiben}) sein. Um die Anzahl der so verwalteten Daten zu verringern, können Teile von Schutzbereichen oder auch ganze Schutzbereiche von mehreren Prozessen gemeinsam genutzt werden.

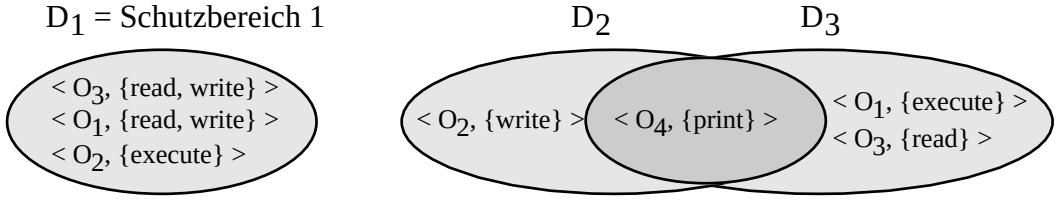


Abb. 10.1: Schutzbereiche

Zuordnungen von Schutzbereichen zu Prozessen können **statisch** oder **dynamisch** sein. Im statischen Fall werden sämtliche Elemente des Schutzbereiches bei der Erzeugung eines neuen Prozesses für seine gesamte Lebensdauer festgelegt. Bei der dynamischen Vergabe ist eine Änderung des Schutzbereiches zur Laufzeit eines Prozesses möglich. Beide Lösungen haben Nachteile. Die statische Zuteilung von Rechten erfordert eine Berücksichtigung aller möglichen Operationen auf allen möglichen Objekten, auch wenn diese von dem Prozeß gar nicht beansprucht werden (man denke an ein großes Dateisystem, bei dem für jeden Prozeß und jede Datei zuerst

einmal Zugriffsrechte vergeben werden). Ein dynamische Zuordnung könnte (als Extrem) eine Vergabe von Rechten nur für den Zeitraum einer Objekt-Operation ermöglichen. In jedem Fall ist die dynamische Methode jedoch wesentlich aufwendiger zu implementieren.

10.1.2 Zugriffsmatrizen

Eine Möglichkeit, Schutzbereiche zu speichern, sind **Zugriffsmatrizen**. Jede Zeile einer Zugriffsmatrix entspricht einem Schutzbereich. Die Spalten der Matrix werden den einzelnen Objekten, die zu schützen sind, zugeordnet. Jedes Element der Matrix enthält eine Menge von Rechten.

Objekt Schutzbereich	Datei1	Datei2	Datei3	Drucker
S1	lesen		lesen	drucken
S2		ausführen	lesen	drucken
S3	lesen, schreiben		lesen	
S4		ausführen	lesen, schreiben	drucken

Diese Art der Darstellung kann sowohl für statische als auch für dynamische Schutzbereiche benutzt werden. Die Zugriffsmatrix kann als Objekt in sich selbst enthalten sein.
Unter gewissen Umständen kann es für einen Prozeß wünschenswert sein, seinen Schutzbereich zu wechseln. Dazu ist nur eine einfache Erweiterung der Zugriffsmatrix notwendig. An das Ende der Matrix werden sämtliche Schutzbereiche angehängt. Das Recht, eine Operation “switch” auszuführen, bedeutet dann, daß ein Prozeß aus dem aktuellen Schutzbereich in den Objekt-Schutzbereich wechseln darf.

Objekt Schutzbereich	...	Drucker	S1	S2	S3	S4
S1	...	drucken				
S2	...	drucken	switch		switch	
S3	...			switch		
S4	...	drucken				

Beispielsweise kann in dem angegebenen Beispiel ein Prozeß aus S2 in S1 wechseln, nicht jedoch umgekehrt. Elemente der Matrix können noch zusätzliche Werte enthalten, die Änderungen der Zugriffsmatrix erlauben.

- Ist eine Operation eines Schutzbereichs mit Kopierrecht ausgezeichnet, so kann ein Prozeß dieses Schutzbereiches die Operation jedem anderen Schutzbereich zugänglich machen (Beispiel: Wäre die Operation drucken für S4 mit Kopierrecht gekennzeichnet, so könnte ein Prozeß aus S4 dem Bereich S3 Druckrechte zuordnen). Diese Weitergabe von Rechten kann mehrere Ausprägungen haben: Die eigenen Rechte können gelöscht werden oder nicht. Außerdem wäre eine Übergabe des Kopierrechts denkbar.

- Hat ein Element der Matrix als Zusatz den Wert “Eigentümer”, so dürfen beliebige Rechte in der entsprechenden Spalte gesetzt werden.

- Die Angabe “Kontrolle” ist nur in Spalten der Objekte vom Typ Schutzbereich möglich. Ist dieser Wert beispielweise für S2 und Objekt S3 gesetzt, so kann ein Prozeß aus S2 jedes Matrixelement der Zeile S3 löschen, d. h. anderen Prozessen u. U. Rechte entziehen.

10.1.3 Implementierung von Zugriffsmatrizen

Die einfachste Möglichkeit, Zugriffsmatrizen zu realisieren, besteht darin, nur eine einzige Matrix in einer **globalen Tabelle** zu speichern. Die Tabelle besteht aus geordneten Einträgen der Form (Schutzbereich, Objekt, Rechte). Immer wenn aus einem Schutzbereich S eine Operation m auf ein Objekt O aufgerufen wird, kontrolliert das System, ob ein Eintrag (S, O, R) existiert, bei dem m in R enthalten ist. Ist dies der Fall, so wird m erlaubt, sonst nicht.
Die Zugriffsmatrix kann jedoch auch **durch Zugriffslisten für Objekte** implementiert sein. Jedem Objekt werden in einer geordneten Liste Tupel der Form (Schutzbereich, Rechte) angehängt. Der Vorteil dieser Darstellungsart liegt insbesondere darin, daß Schutzbereiche ohne Rechte nicht in die Liste übernommen werden. In einem zusätzlichen Element **default-Rechte**, können zusätzlich Rechte gespeichert werden, die für alle Schutzbereiche gelten. Wird diese Menge zuerst geprüft, kann die Suche nach dem Tupel (S, R) in der Liste u. U. komplett eingespart werden (Effizienzsteigerung).
In umgekehrter Weise können auch **Capability-Listen für Schutzbereiche** benutzt werden. Diese enthalten die Objekte zusammen mit den erlaubten Operationen. Analog zu den Zugriffslisten für Objekte können Einträge ohne zulässige Operationen weggelassen werden. Objekte werden in der Liste durch ihre physikalische Namen bzw. Adresse repräsentiert, die sogenannte **Capability**. Ein Operationsaufruf enthält diese Capability des Objekts als Parameter und wird immer erlaubt, da die Capability nur bekannt ist, wenn das entsprechende Recht vorhanden ist. Da den Capabilities eine wichtige Rolle zukommt, müssen sie besonders geschützt werden. Insbesondere dürfen sie nur vom Betriebssystem geändert werden.
Ein Kompromiß zwischen den beiden zuletzt vorgestellten Verfahren stellt der **Schloß-Schlüssel-Mechanismus** dar. Jedes Objekt hat eine Liste eindeutiger Bitmuster (Schlösser). Listen von Bitmustern gehören ebenfalls zu jedem Schutzbereich (Schlüssel). Ein Prozeß darf eine Operation ausführen, wenn sein Schutzbereich den Schlüssel zu einem Schloß des Objekts enthält.

10.1.4 Entzug von Zugriffsrechten

Sollen zu einem Objekt gehörende Rechte entzogen werden, so stellen sich einige Fragen:

- Sollen die Rechte **sofort** oder **verzögert** gelöscht werden?

- Betrifft der Entzug **alle Schutzbereiche** oder **nur einen Teil der Schutzbereiche**?
- Können für ein Objekt **Teile der Rechte** entfernt werden oder nur **alle gemeinsam**?

- Ist der Entzug **dauerhaft** oder nur **zeitweise**?

Sind diese Fragen geklärt, muß ein Mechanismus entwickelt werden, der eine entsprechende Entfernung durchführt. In Zugriffslisten für Objekte ist die Entfernung der Rechte einfach: Die Liste des entsprechenden Objektes wird durchsucht, und die Rechte werden gelöscht. In Capability-Listen liegt das Problem darin, daß Objekte

nicht nur einfach, sondern in mehreren Listen auftauchen können. Man muß also alle Capabilities finden, um die Rechte zu entziehen. Zur Vereinfachung gibt es verschiedene Möglichkeiten:

- Periodisch werden alle Capabilities gelöscht und bei einem Zugriff neu gebildet.
- Für jedes Objekt gibt es Zeiger, die auf zugehörige Capabilities verweisen.
- Capabilities sprechen ein Objekt nicht mehr direkt, sondern über ein gemeinsames, zwischengeschaltetes Element an. Das Löschen dieses Elements entspricht dem Löschen aller Capabilities, da bei jedem Zugriff auf das Objekt ein Zeiger-Fehler auftritt.
- Jeder Capability wird bei ihrer Erzeugung der Schlüssel des Objekts übergeben (Bitmuster). Um erfolgreich auf das Objekt zugreifen zu können, müssen die Schlüssel von Objekt und Capability übereinstimmen. Sollen die Rechte entzogen werden, so wird einfach der Schlüssel des Objekts geändert.

10.2 Das Problem der Sicherheit

Die angesprochen Schutzmechanismen bewahren Daten und Ressourcen vor unberechtigtem Zugriff von Prozessen. Sie schützen aber keineswegs vor unberechtigtem Zugriff durch Menschen. Ein Benutzer, der beispielsweise das Paßwort des Systemverwalters kennt, kann sich ohne Probleme als dieser ausgeben und u. U. beträchtlichen Schaden anrichten. Auch das Übertragen von vertraulichen Daten über ungesicherte Netze (wie z. B. das Internet) kann gefährlich sein, wenn die Leitung abgehört wird.

10.2.1 Authentifizierung

Wie kann ein System nun kontrollieren, wer aktuell den Rechner nutzt? Eine Möglichkeit ist es, zu Beginn die **Authentizität** eines Benutzers zu prüfen. Dies geschieht i. a. über ein Paßwort. Es können jedoch auch andere Verfahren verwendet werden, wie z. B. Code-Karten oder (noch komplizierter) Vergleich von Fingerabdrücken.
Paßwörter bieten, sofern sie geeignet gewählt (also nicht unbedingt den Namen der Freundin, dafür aber möglichst lang und mit Sonderzeichen) und gehandhabt (sprich nicht mit einem Aufkleber an der Tastatur vermerkt sind) werden, eine recht gute Möglichkeit zur Authentifizierung. Dazu benutzt der Rechner eine Funktion f, die aus einer Zeichenkette x sehr schnell einen Funktionswert f(x) berechnen kann. Die umgekehrte Berechnung, d. h. von einem Funktionswert f(x) auf x zu schließen, sollte unmöglich sein. Für jeden Benutzer speichert das System nur den Funktionswert seines Paßworts und vergleicht diesen Wert bei jeder Authentifizierung mit dem Wert, der aus der Eingabe des Paßworts berechnet wird. Eine Übereinstimmung besagt, daß der Benutzer erfolgreich identifiziert wurde. In Netzwerken, bei denen Paßwörter teilweise unverschlüsselt übertragen werden, ist diese Sicherheit jedoch nicht mehr gegeben. Hier werden kryptographische Verfahren notwendig, auf die später noch etwas genauer eingegangen wird.

10.2.2 Viren und ähnliches Gewürm

In vielen Systemen stehen einige Programme mehreren Benutzern gemeinsam zur Verfügung, z. B. Editoren, Compiler etc. Diese Programme können als **“Trojanische**

Pferde" mißbraucht werden. Ein Benutzer, der ein gemeinsames Programm aufruft öffnet diesem praktisch die Tore zu allen seinen Rechten. Ein Compiler könnte z. B. eine Routine enthalten, die alle eigentlich nur von dem Benutzer lesbaren Dateien in ein anderes, für alle sichtbares Verzeichnis kopiert.
Ein anderes Problem besteht darin, daß Programmierer eines Betriebssystems sogenannte **"Trap Doors"** in den Code einbauen können. Zum Beispiel könnte ein bestimmtes Paßwort einem Benutzer damit Zugriff zu allen Daten ermöglichen. Dies ist besonders dann gefährlich, wenn es um sensitive Daten geht wie z. B. Kontostände in den Rechnern einer Bank etc.
Innerhalb von Computer-Netzwerken können **"Worms"** eine Gefahr darstellen. Ein Worm ist ein Programm, welches die Eigenschaft vieler Betriebssysteme nutzt, daß ein Prozeß in mehrere gleichartige aufgesplittet werden kann. Ist ein solcher Prozeß auf einem System erfolgreich gestartet, versucht es Zugriff auf andere Rechner zu bekommen, um dort eine Kopie zu starten. Da in manchen Betriebssystemen beispielsweise die Möglichkeit besteht, die Abfragen von Paßwörtern bei entferntem Einloggen abzuschalten, hat ein solches Programm gute Chancen, sich im Netz auszubreiten und Schaden anzurichten.
Eine andere Form sich ausbreitender Gefahren sind **Viren**. Ein Virus ist kein eigenständiges, sondern nur Teil eines "infizierten" Programms. Die Verteilung dieser Programme erfolgt über öffentlich zugängliche Software-Datenbanken oder einfach durch den Austausch von Disketten.
Vor all diesen Problemen kann man sich nie perfekt schützen. Ein vorsichtiger Umgang mit der benutzten Software kann jedoch erheblich zur Verringerung beitragen. Außerdem kann das Betriebssystem durch **Monitoring** mögliche Gefahrenquellen ausmachen. Beispielsweise kann das System von Zeit zu Zeit nach bekannten Viren durchsucht werden oder wichtige Verzeichnisse auf ihre Konsistenz prüfen.

10.2.3 Kryptographie

Kryptographie beschäftigt sich mit der Verschlüsselung von Informationen **(Klartext)**. Unter Benutzung kryptographischer Verfahren sind **chiffrierte (kodierte)** Daten für unberechtigte Leser nicht zu entziffern. Zum Lesen berechtigte Benutzer besitzen hingegen einen **Schlüssel**, mit dem der chiffrierte Klartext **dechiffriert (dekodiert)** werden kann. Im allgemeinen werden für den hier beschriebenen Vorgang drei Dinge benötigt: ein Kodier-Algorithmus K, ein Dekodier-Algorithmus D und ein geheimer Schlüssel s. Ein für die Übertragung einer Nachricht m geeignetes Kryptographie-Verfahren muß folgenden Bedingungen genügen:

1. $D(s, K(s, m)) = m$

2. D und K sind effizient berechenbar

3. Die Sicherheit ist nur von der Geheimhaltung des Schlüssels s, nicht jedoch von den Algorithmen D und K abhängig.

In unsicheren Netzwerken benutzt man für die Übertragung der Daten einen Mechanismus, der **öffentliche und private Schlüssel** benutzt. Eine mit dem öffentlichen Schlüssel (den dürfen alle kennen) kodierte Nachricht kann nur mit dem privaten Schlüssel (den darf nur der Empfänger kennen) dekodiert werden. Um eine Nachricht kodiert zu verschicken, muß also nur ein öffentlicher Schlüssel des Empfängers angefordert werden. Das Problem ist dabei: Wie findet man für sich geeignete Schlüssel? Eine Antwort liefert die Zahlentheorie mit folgendem Algorithmus:

1. Wähle zwei große Primzahlen p und q ($> 10^{100}$, schwer zu finden und aus diesem Grund auch sehr begehrt).

2. Berechne $p \cdot q$ sowie $z = (p-1) \cdot (q-1)$.

3. Wähle ein d teilerfremd zu z.

4. Finde ein e mit $e \cdot d = 1 \bmod z$.

Dann ist das Paar (e, n) der öffentliche und (d, n) der private Schlüssel. Kodiert wird eine Nachricht m mit $c = m^e \bmod n$, dekodiert mit $m = c^d \bmod n$. Um diese Kodierung zu knacken, müßte n in seine Primfaktoren zerlegt werden, was mit den heutigen Methoden (Mathematiker + Computer) und bei genügend großem n praktisch unmöglich ist.

11 Verteilte Systeme

Betrachtet man nicht einen isolierten Rechner, sondern gestaltet dessen Möglichkeiten durch die Verbindung zu anderen Rechnern komfortabler, so entsteht ein Verteiltes System. Dieser Begriff ist inzwischen Gegenstand zahlreicher Bücher. So verwendet etwa Tanenbaum in seinem Werk "Moderne Betriebssysteme" etwa die Hälfte des Buches zur Beschreibung Verteilter Systeme; das Nachfolgebuch des gleichen Autors bekam dann sogar den Titel "Verteilte Betriebssysteme". Auch wenn man sich real existierende Rechnernetze und Informationssysteme anschaut, ist die wachsende Bedeutung der Verteilten Systeme nicht zu leugnen. Allgemein kann man sagen, daß sich netzspezifische Fächer oder auch die Vorlesung "Datenkommunikation" mit den Schichten 1 - 4 (bezogen auf das in Kapitel 1.5.4 eingeführte OSI-Modell) beschäftigen, während unter dem Stichwort "Verteilte Systeme" anwendungsnah die Aufgaben der oberen Schichten 5 - 7 des OSI-Modells untersucht werden.

11.1 Einführung in Verteilte Systeme

Innerhalb dieses Abschnitts soll auf die Grundlagen Verteilter Systeme eingegangen werden. Ausgangspunkt dabei ist eine Begriffsbestimmung. Es schließt sich eine Diskussion an, ob Verteilte Systeme sinnvoll im Einsatz sind. Schließlich wird auf das kommerzielle Produkt, das für einen Zusammenschluß einzelner Rechner zu einem Verteilten System notwendig ist, eingegangen: die Verteilungsplattform.

11.1.1 Funktionalität und Anforderungen

In der Literatur gibt es eine Reihe verschiedener Ansätze zur Klärung dessen, was man unter einem Verteilten System genau verstehen soll. Wir wollen hier auf die Ausführungen in Spaniol et al., S. 86, zurückgreifen.

Ein **Verteiltes System** ist demnach ein System mit räumlich verteilten Komponenten, die keinen gemeinsamen Speicher benutzen und einer dezentralen Administration unterstellt sind. Zur Ausführung gemeinsamer Ziele ist eine Kooperation der Komponenten möglich. Werden von diesen Komponenten Dienste angeboten und angebotene Dienste genutzt, so entsteht ein **Client/Server-System**, im Falle einer zusätzlichen zentralen Dienstvermittlung ein sogenanntes Dienstvermittlungs- oder **Tradingsystem**.

Das Konzept der Verteilten Systeme ist relativ jung. Erst in der Mitte der 80er Jahre wurden die Voraussetzungen dafür geschaffen, dieses Prinzip einführen zu können. Als grundlegend für die Entwicklung bzw. Installation eines Verteilten Systems muß man dabei festhalten:

- **Leistungsexplosion bei Halbleiterchips**
  Die Rechenleistung bei Mikroprozessoren hat sich im letzten Jahrzehnt ca. alle zwei Jahre verdoppelt, die Kapazität von Halbleiterspeichern alle drei Jahre vervierfacht. Stetig wachsende Leistung bei schrumpfenden Preisen und Abmessungen bilden die Grundlage dafür, daß immer mehr Rechner immer komplexere Software ausführen können.
- **Schnelle Rechnernetze**
  Die Bereitstellung schneller lokaler Datennetze senkt die Zugriffszeiten auf externe Rechner und bildet so die ökonomische Voraussetzung dafür, Computer

(PCs wie Workstations) miteinander zu verbinden. Die Einführung der Ethernet-Technik in den siebziger Jahren kann als Wegbereiter für verteilte Softwaresysteme angesehen werden, heute erfolgt auch die Nutzung von FDDI, DQDB und zunehmend ATM.

- **Konzepte der Softwaretechnik**
  In den letzten drei Jahrzehnten waren erhebliche Fortschritte im Bereich der Softwaretechnik zu verzeichnen. Die Akzeptanz von programmiersprachlichen Konzepten wie Prozedur, Modul und Schnittstelle schuf die Voraussetzungen für die grundlegenden Mechanismen Verteilter Systeme. Konsequenzen waren der Remote Procedure Call (RPC) und die objektorientierte Modellierung Verteilter Systeme.

- **Autonomie der Rechnerorganisation**
  Die Abkehr von streng hierarchisch aufgebauten Organisationsformen in Unternehmen führt ganz allgemein zu einer Dezentralisierung und schafft flache Führungsstrukturen. Dadurch eröffnen sich weite Anwendungsfelder für Verteilte Systeme.

Verteilte Systeme besitzen eine Reihe von Vor- und Nachteilen. Auch wenn es beim heutigen Stand der Technik möglich ist, Verteilte Systeme zu installieren, so ist doch zu überlegen, warum beispielsweise ein bestehendes zentrales System durch ein Verteiltes System ersetzt werden sollte.

11.1.2 Vor- und Nachteile Verteilter Systeme

Das noch vor einigen Jahren typische Rechenzentrum eines Großunternehmens ist heute kaum noch anzutreffen. Statt dessen verfügen die meisten Unternehmen über Verteilte Systeme. Diese Beobachtung aus der Wirtschaft läßt bereits vermuten, daß letztlich wohl doch die Vorteile überwiegen. Daher wollen wir im folgenden zunächst auf die wichtigsten Vorteile von Verteilten Systemen eingehen (ausführlichere Überlegungen hierzu finden sich in Popien et al.).

- **Stetige Kapazitätsanpassung**
  Verteilte Systeme ermöglichen die stetige Anpassung der Größe eines Systems. Den neu hinzukommenden Anforderungen an das Computersystem eines expandierenden Unternehmens kann durch Erweiterungen bestehender Komponenten zeitgemäß entsprochen werden.

- **Integrierbarkeit bestehender Lösungen**
  Existierende Systeme können von neu hinzukommenden Systemkomponenten genutzt werden, ohne daß ein System gleicher Funktionaliät neu entwickelt werden muß.

- **Risikominimierung**
  Eine sukzessive Systemerweiterung minimiert das Risiko der Überlastung einzelner Systemkomponenten, da hierbei stets auf die gleichmäßige Auslastung sowohl bestehender als auch neu hinzukommender Module geachtet werden kann.

- **Flexibilität und Anpaßbarkeit**
  Die überschaubare, organisatorische Verwaltung der Kapazität eines Verteilten Systems erlaubt kostengünstige Realisierungen. Das System ist flexibel und anpaßbar.

- **Autonomie**
  Der Eigentümer einer Ressource hat die Möglichkeit, das Management dieser Komponente selbst zu übernehmen. In jedem Fall steht es ihm frei, bei Bedarf einzugreifen, um seine eigenen Interessen wahrzunehmen. Ferner sind die einzelnen Bestandteile eines Verteilten Systems weitestgehend autonom. Im Falle eines Fehlers oder sogar Ausfalls einer Systemkomponente können die übrigen Einheiten im Idealfall unbeeinflußt weiterarbeiten und ggf. den Störfall überbrücken.

Es gibt zahlreiche weitere Vorteile Verteilter Systeme. Diese reichen von besserer Bearbeitung einzelner Prozesse durch parallele Abarbeitung bzw. Nebenläufigkeit bis hin zu Möglichkeiten, Mobilität in Verteilten Systeme zu erhalten.

Bei der Diskussion der Nachteile - über die man sich auch streiten kann, da sie vielleicht auch nur eine Frage der Forschung und Entwicklung auf diesem Gebiet sind - seien folgende Punkte genannt:

- **Softwaredefizit**
  Für den Zeitraum des Übergangs zu Verteilten Systemen droht ein Softwaredefizit. Die Realisierung eines Verteilten Systems erfordert komplexere Softwarelösungen als die eines zentralen Systems. In dieser Hinsicht stehen die Verteilten Systeme erst am Anfang der Konzeptentwicklung.

- **Sicherheitsbedenken**
  Die neu hinzukommenden Netzwerkkomponenten können vollkommen neuartige Fehler verursachen. Unabhängig davon sind Verteilte Systeme auch aus Sicht des Datenschutzes bedenklich, da vernetzte Daten generell einen einfacher den Zugriff ermöglichen, als dies bei separater Datenhaltung der Fall ist.

- **Leistungsfähigkeit**
  Ein Prozessorpool bedingt im Mittel eine kleinere Wartezeit bis zur Abarbeitung einer Aufgabe, als dies bei mehreren einzelnen Komponenten mit insgesamt der gleichen Prozessorleistung der Fall ist. Um annähernd gleiche Größen zu erhalten, ist ein komfortables Leistungsmanagement bei Verteilten Systemen erforderlich.

11.1.3 Verteilungsplattformen

Verteilte Systeme zeichnen sich durch eine Reihe von Eigenschaften aus, die zentrale Systeme nicht benötigen bzw. über die sie gar nicht erst verfügen. Eine der wichtigsten Eigenschaften ist in diesem Zusammenhang die sogenannte Transparenz.

Transparenz ist eine zentrale Forderung, die sich aus dem Wunsch ergibt, den Umgang mit verteilten Anwendungen so weit wie möglich zu erleichtern. Sie umfaßt das Verbergen von Implementierungsdetails und die sogenannte

Verteilungstransparenz. Diese verbirgt ihrerseits die Komplexität eines Verteilten Systems. Dabei werden interne Vorgänge durch sogenannte Transparenzfunktionen vor dem Betrachter verborgen und der Nutzer entlastet.

Es gibt viele Ausprägungen von Verteilungstransparenz. An dieser Stelle seien nur die wichtigsten Stichworte genannt (Näheres wiederum in Popien et al.):

- Zugriffstransparenz,
- Ortstransparenz,
- Replikationstransparenz,
- Abarbeitungstransparenz,
- Migrationstransparenz,
- Ausfalltransparenz,
- Ressourcentransparenz,
- Verbundtransparenz und
- Gruppentransparenz.

Zur Realisierung verteilter Zugriffe bedarf es einer geeigneten Softwareinfrastruktur. Diese heißt **Verteilungsplattform** oder auch **Netzbetriebssystem** bzw. **Middleware** und dient zur Unterstützung der Interaktion zwischen den auf potentiell heterogenen Systemen ablaufenden Anwendungskomponenten. Die Verteilungsplattform wird dem lokalen Betriebssystem hinzugefügt oder übernimmt selbst Aufgaben eines Betriebssystems (siehe Abbildung 11.1).

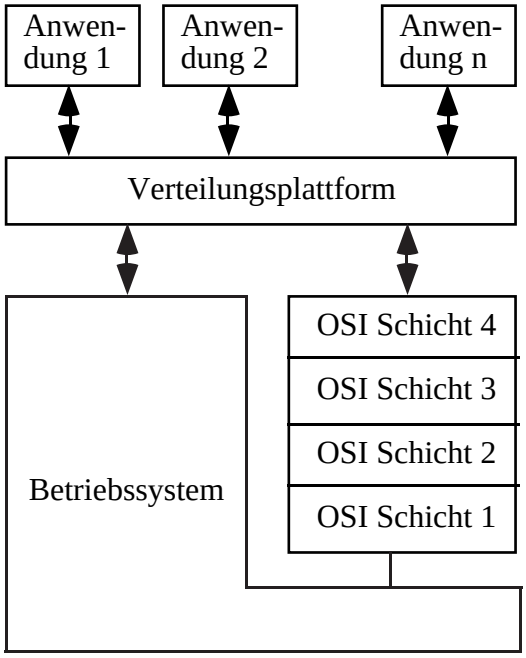


Abb. 11.1: Die Schnittstellen einer Verteilungsplattform

Mittels einer solchen Verteilungsplattform läßt sich die Verteilung transparent halten, d. h. Anwendungen werden von komplexen Details interner Vorgänge abgeschirmt.
Einer Verteilungsplattform können viele individuelle Systeme zugrundeliegen, auf denen sie aufsetzt. Gleichzeitig kann eine Vielzahl von Anwendungen auf die Verteilungsplattform zugreifen.

Kommerziell werden heutzutage verschiedene Verteilungsplattformen vertrieben. Die wichtigsten dieser Plattformen sind Distributed Computing Environment (DCE), Common Object Request Broker Architecture (CORBA) sowie ANSAware.

11.2 Strukturen in Verteilten Systemen

In diesem Abschnitts soll auf das den Verteilten Systemen zugrundeliegende Prinzip des Client/Server-Modells eingegangen und damit zusammenhängende Probleme angesprochen werden.

11.2.1 Das Client/Server-Modell

Warum ein neues Modell? Nach der Vorstellung des ISO/OSI-Modells im Eingangskapitel liegt doch eigentlich die Vermutung nahe, daß damit alle mit Kommunikation zusammenhängenden Fragen angemessen behandelt werden können. Die Antwort auf die Frage liegt im Verwaltungsaufwand der sieben Schichten, der so hoch ist, daß nach einem anderen (einfacheren) Modell gesucht wurde.

Die Idee des Client/Server-Modells besteht darin, ein Betriebssystem als eine Menge kooperierender Prozesse - sogenannter **Server** - zu strukturieren. Diese stellen den Nutzern - sogenannten **Clients** - Dienste bereit. Auf einem Rechner können sowohl Clients als auch Server (ggf. auch gleichzeitig) laufen. Das Client/Server-Modell basiert auf einem einfachen, verbindungslosen Anfrage/Antwort-Protokoll (siehe Abbildung 11.2) Der Client sendet dabei eine Anfrage und erhält seine Antwort vom Server.

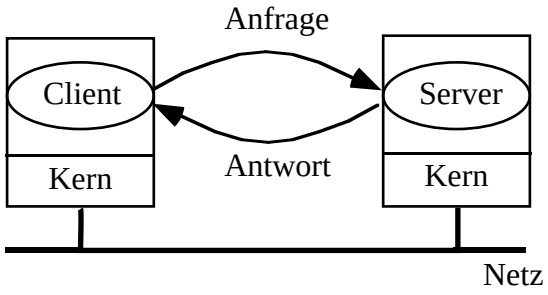


Abb. 11.2: Das Prinzip des Client/Server-Modells

Diese einfache Methode ist sehr effizient, da der Protokollstapel klein gehalten wird. Basierend auf dieser einfachen Struktur ist es lediglich notwendig, daß die Verteilungsplattform zwei Systemaufrufe anbietet. Ein Aufruf der Syntax `send(a,&mp)` verschickt die Nachricht, die durch `mp` referenziert wird, an einen Prozeß, der durch `a` identifiziert wird. Der Aufrufer wird dabei solange blockiert, bis die Nachricht vollständig verschickt ist. Demgegenüber wird der Aufrufer von `receive(a,&mp)` blockiert, bis eine Nachricht für ihn angekommen ist. Die Nachricht wird in den durch `mp` angegebenen Puffer kopiert. Der Parameter `a` gibt dabei die Adresse des Empfängers an.
Bei der Realisierung eines Client/Server-Modells ist die Synchronisation der `send`- und `receive`-Aufrufe von besonderer Bedeutung. Nur durch ein geeignetes Zusammenspiel dieser beiden Operationen kann das System auch ordnungsgemäß arbeiten.

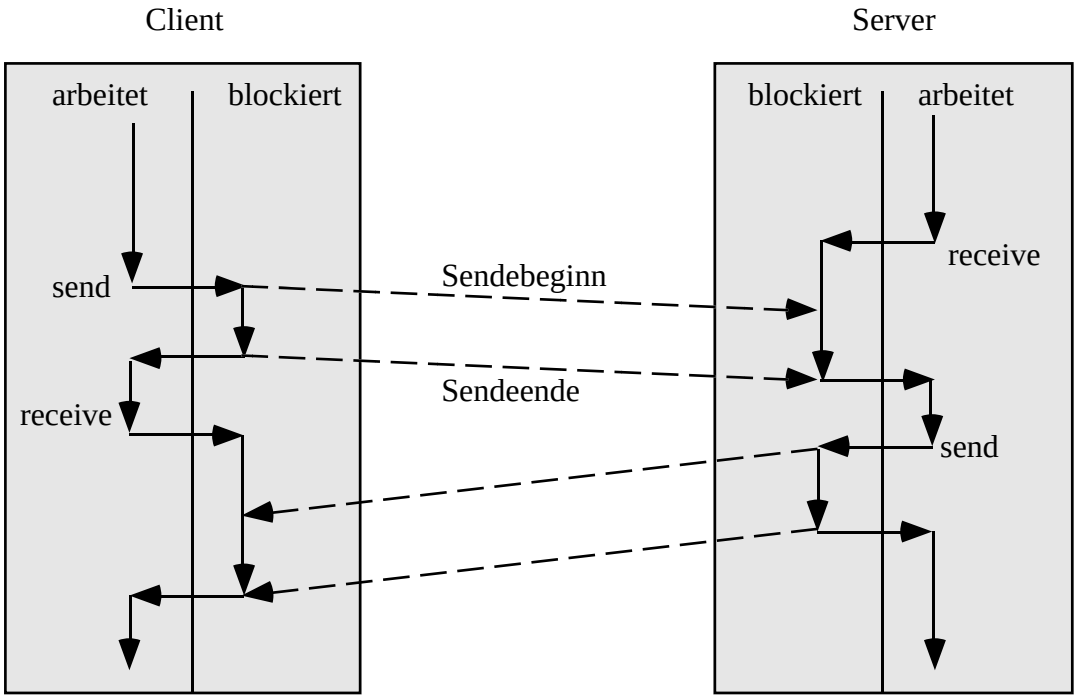


Abb. 11.3: Die Wirkungsweise des Client/Server-Modells

Zur Veranschaulichung der Wirkungsweise ist in Abbildung 11.3 ein Client sowie ein Server dargestellt. Beide Komponenten können sich in je zwei Zuständen befinden: blockiert und nicht blockiert, d. h. arbeitend. Der Zustand "blockiert" bedeutet dabei, daß der Prozessor der Komponente nicht arbeitet, sondern auf ein Ereignis wartet, das eine Modifikation des Systemzustands bedingt. In der Regel wird dieses Ereignis bei einem `receive`-Aufruf das Ankommen einer Nachricht sein. Bei einem `send`-Primitiv erfolgt solange eine Blockierung, bis ein Nachrichtenpaket vollständig verschickt ist.

Beim Client/Server-Modell gibt es verschiedene Probleme, die genauer untersucht werden müssen. Zunächst wollen wir uns um die Adressierung kümmern.

Adressierung

Im Normalfall ist die Adresse eines anzusprechenden Servers bekannt. Die Adresse ist eine Konstante, die der Client (woher auch immer - siehe folgende Fälle) kennt, zum Beispiel kann sie in einer Datei abgelegt sein. Ist diese Voraussetzung jedoch nicht erfüllt, so ergibt sich das Adressierungsproblem.

Die einfachste Form einer Adressierung ist die Angabe von Prozeß und Zielrechner, die sogenannte "**machine.process-Adressierung**". Diese Alternative zur absoluten Adressierung spaltet den Namen eines Prozesses in zwei Teile auf: eben den Prozeß und den Rechner, auf dem dieser Prozeß läuft. In welcher Reihenfolge diese beiden Komponenten des Namens angegeben werden, ist Vereinbarungssache: Läuft ein Prozeß mit der Nummer 4 auf einem Rechner 21415, so könnte als Bezeichnung gleichermaßen 4@21415 oder 21415.4 verwendet werden. Die Funktionsweise ist denkbar einfach: Nach dem Stellen einer Anfrage an einen Server erfolgt die Antwort zurück an den Client. Nachteil dieser Adressierung ist das Nichtvorhandensein von Ortstransparenz, da dem Nutzer nicht verborgen bleibt, wo er arbeitet.

Ein zweiter Ansatz sieht ein spezielles **Lokalisierungspaket** vor, das der Sender an alle anderen Rechner schickt. Das Lokalisierungspaket enthält die Adresse des Zielrechnerprozesses oder eine Bezeichnung des gesuchten Prozesses, d. h. Dienstes. Insbesondere in broadcastfähigen LANs ist diese Adressierung realisierbar. Dazu geht man in vier Schritten vor:

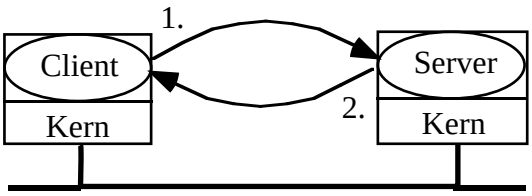
1. Broadcasten des Lokalisierungspakets
2. "Ich-bin-hier"-Antwort des Servers
3. Anfrage an den Server
4. Antwort des Servers.

Dieses Vorgehen sichert zwar die Ortstransparenz, verursacht in manchen Netzen durch den Broadcastaufruf jedoch zusätzliche Last. Aus diesem Grunde gibt es eine dritte Alternative, die gerade in Verteilungsplattformen der Normalfall ist: die Einführung eines **Name-Servers** oder **Traders**.

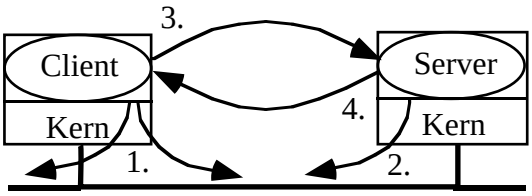
Diese ausgesprochen komfortable Realisierung läuft prinzipiell in folgenden vier Schritten ab:

1. Anfrage des Clients an den Trader nach der Adresse eines gewünschten Servers
2. Übermittlung der gewünschten Adresse vom Trader an den Client
3. Anfrage des Clients an die benannte Adresse eines Servers
4. Antwort des Servers an den Client

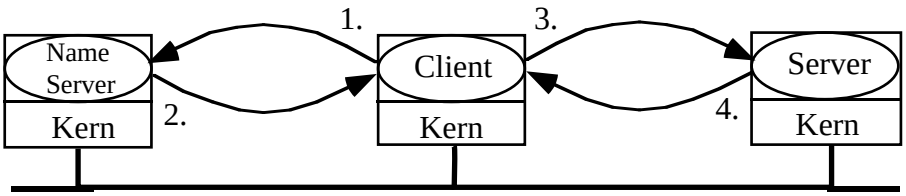
Das Vorhandensein eines solchen Traders kann dem Client auch bei unvollständigen Angaben die Vermittlung eines Dienstes ermöglichen. Diese Realisierung dieser Dienstvermittlung erfordert jedoch einige neue Mechanismen (vgl. Spaniol et al.). Abbildung 11.4 stellt diese drei Alternativen noch einmal gegenüber.



a) machine.process-Adressierung



b) Lokalisierungspaket



c) Name-Server-Konzept

Abb. 11.4: Adressierungsarten

Blockierung

In Analogie zu den in Abbildung 11.2 eingeführten Mechanismen unterscheidet man zwischen blockierenden und nichtblockierenden Primitiven. Es ist die Aufgabe des Systementwicklers, zwischen diesen beiden Möglichkeiten zu unterscheiden.

Bei **blockierenden Primitiven** wird der Prozeß während der Sendung einer Nachricht blockiert, d. h. Anweisungen werden erst dann weiter abgearbeitet, wenn die Nachricht vollständig abgesendet ist. Analog endet die Blockierung beim Empfangen erst, nachdem die Nachricht angekommen und kopiert ist. Die in Abbildung 11.3 vorgestellte Implementierung basiert auf der Verwendung von blockierenden Primitiven.

Bei **nichtblockierenden Primitiven** wird die zu sendende Nachricht zunächst nur in einen Puffer des Betriebssystemkerns kopiert. Direkt im Anschluß daran, also noch vor der eigentlichen Sendung, erfolgt die Entblockierung. Der sendende Prozeß kann parallel zur Nachrichtenübertragung mit seiner Ausführung fortfahren. Dieser Geschwindigkeitsvorteil wird jedoch dadurch erkauft, daß der Sender nicht weiß, wann die Übertragung beendet ist und wann er den Puffer wieder nutzen kann. Ein zusätzlicher Nachteil ist, daß ein einmal beschriebener Puffer nicht mehr verändert werden kann.

Pufferung

Auch bezüglich der Pufferung gibt es zwei Möglichkeiten, Primitive zu realisieren. Bislang wurde immer von **ungepufferten Primitiven** ausgegangen. `receive(a,&mp)` informiert den Kern seiner Maschine, daß der aufrufende Prozeß die Adresse `a` abhören will und bereit ist, Nachrichten von dort zu empfangen. Einen Puffer stellt er an der Adresse `&mp` bereit. Problematisch bei dieser Herangehensweise kann sein, daß im Falle des früheren Abschickens eines `send` und späteren Aufrufens von `receive` der Kern bei einer ankommenden Nachricht nicht weiß, ob einer seiner Prozesse die Adresse dieser Nachricht benutzen wird und wohin er ggf. die Nachricht kopieren soll. In diesem Fall kann der Verlust einer Nachricht nicht ausgeschlossen werden. Auch wenn mehrere Clients dieselbe Adresse benutzen, sind Probleme nicht ausgeschlossen.

Eine alternative Implementierung sind die **puffernden Primitive**. Ein empfangender Kern speichert die eintreffenden Nachrichten für einen bestimmten Zeitraum zwischen. Wird kein passendes `receive` aufgerufen, so werden die Nachrichten nach einen Timeout gelöscht. Es müssen Puffer vom Kern bereitgestellt und verwaltet werden. Eine konzeptionell einfache Lösung sieht dafür die Definition einer Datenstruktur "Mailbox" vor.

Zuverlässigkeit

Client/Server-Systeme lassen sich hinsichtlich ihrer Zuverlässigkeit in drei Klassen einteilen. Die erste Klasse verfährt nach dem Prinzip der gelben Post: Abschicken, der Rest ist egal - in der Regel kommt schon alles an. Eine sicherere Möglichkeit besteht darin, daß der empfangende Kern eine Bestätigung an den Sender schickt. Und die dritte Möglichkeit geht schließlich davon aus, daß die Antwort des Servers an den Client gleichzeitig auch die Bestätigung der Anfrage ist, also nur noch der Client bestätigt, daß er die Antwort vom Server erhalten hat.

11.2.2 Der Remote Procedure Call

Das Client/Server-Modell bietet einen brauchbaren Weg, ein Verteiltes Betriebssystem zu strukturieren. Trotzdem hat es eine extreme Schwachstelle: Das Basisparadigma aller Kommunikation ist die Ein- und Ausgabe von Daten, der explizite Aufruf von Kommunikationsprimitiven wie `send` und `receive` dient jedoch zum Austausch von Daten. Wollen wir unser Ziel erreichen, verteilte Berechnungen genauso aussehen zu lassen wie zentrale Berechnungen, dann benötigen wir komfortablere Mechanismen.

Der Vorschlag von Birell und Nelson aus dem Jahre 1984 besteht darin, daß ein Programm ein Unterprogramm aufruft, welches sich auf einem anderen Rechner befindet. Diese Methode ist als "entfernter Unterprogrammaufruf" (**Remote Procedure Call**, RPC) bezeichnet worden. Ruft ein Rechner A ein Unterprogramm auf einem Rechner B auf, so wird der aufrufende Prozeß auf A ausgesetzt (suspendiert), und die Ausführung des aufgerufenen Unterprogramms findet auf Rechner B statt. Dabei werden Parameter ausgetauscht, der Nachrichtenaustausch bleibt für den Benutzer jedoch unsichtbar.
Sei `read` ein entferntes Unterprogramm. Dann ist nicht dieses `read` in der Bibliothek eines Clients enthalten, sondern die Bibliothek enthält einen sogenannten **Client-Stub**, an den sich der Client mit seiner Anfrage wendet und von dem er letztlich

auch die Antwort bekommt. Ein **Stub** wird zuweilen auch als Stellvertreterprozedur bezeichnet. Er ist ein Prozeß, der die Aufgabe besitzt, aus einer Anfrage eine Nachricht zu generieren, die an den Server geschickt werden kann. Der Aufruf eines Unterprogramms im Client-Stub löst eine sogenannte Ausnahmebehandlung aus: zunächst werden die Parameter des Aufrufs in eine Nachricht verpackt. Das `send`-Primitiv beauftragt dann den Kern, die Nachricht an den Server zu schicken. Im Anschluß daran ruft der Client-Stub ein `receive` auf und blockiert sich so lange, bis eine Antwort eintrifft.

Nun zur Server-Seite: trifft die Nachricht vom Client ein, so übergibt der Kern diese an den sogenannten **Server-Stub**, der an den aktuellen Server gebunden ist. Im allgemeinen hat der Server-Stub gerade ein `receive` aufgerufen und wartet auf ankommende Nachrichten. Der Server-Stub packt dann die in der Nachricht enthaltenen Parameter aus und ruft das Unterprogramm lokal auf. Parameter und Rücksprungadresse werden abgelegt. Es folgt die Ausführung und die Rückgabe der Ergebnisse an den Server-Stub. Erhält der Server-Stub die Kontrolle zurück, so verpackt er die Ergebnisse wieder zu einer Nachricht, die er mittels `send` an den Client schickt. Ggf. ruft er danach wieder ein `receive` auf, um für den Empfang der nächsten Nachricht bereit zu sein.

Erreicht die Antwort des Servers den Client, wird sie in einen entsprechenden Puffer kopiert und der Client-Stub wird entblockiert. Der Client-Stub prüft die Nachricht und packt sie aus. Dann schiebt er sie auf den Stack. Wenn der Aufrufer von `read` die Kontrolle zurückerhält, so ist ihm nur bekannt, daß die Daten vorliegen, nicht aber, daß ein Aufruf entfernt ausgeführt wurde. Diese Transparenz zeichnet den RPC aus.

Auf diese Art und Weise können entfernte Dienste durch einfache, d. h. lokale Unterprogrammaufrufe ausgeführt werden. Dies erfolgt ohne explizite Anwendung der Kommunikationsprimitive. Alle Details sind durch zwei Bibliotheksroutinen - den Client- und den Server-Stub - verborgen. Eine Zusammenfassung dieser Schritte ist in Abbildung 11.5 dargestellt.

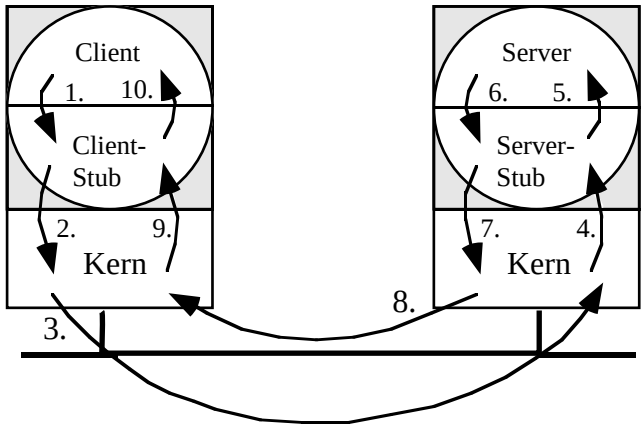


Abb. 11.5: Das Prinzip des RPCs

Es gibt verschiedene Probleme, die bei der Realisierung eines RPCs zu lösen sind. Eines der wichtigsten ist das sogenannte **Parameter Marshalling**, die Parameterübergabe. Hierunter versteht man das Verpacken einer Nachricht. Zum Beispiel würde aus dem einfachen Aufruf `x=produkt(2,y)` eine Nachricht der Form (`produkt,2,y`) oder (`y,2,produkt`) gemacht werden. Was in diesem Falle noch recht unkritisch aussieht, kann sich schnell als Problem herausstellen: unter-

schiedliche Formate (INTEL-Format, SPARC-Format) können unterschiedliche Interpretationen der Nachricht bedingen.

Ein weiteres Problem, das bei Verteilungsplattformen gelöst sein muß, ist die Thematik des **dynamischen Bindens**: In einem Verteilten System ist es möglich, daß sich aufgrund bestimmter Ereignisse die Adresse eines Servers bzw. einer Schnittstelle ändert. Dies hätte zur Folge, daß zahlreiche Programme (die mit Konstanten als Adresse arbeiten) neu geschrieben und übersetzt werden müßten. Dieses Problem wird durch Einsatz sogenannter "dynamischer Binder" gelöst.

Zu Beginn einer Server-Ausführung erfolgt dabei der Aufruf eines sogenannten `initialize()`. Dieser bewirkt das Exportieren der Schnittstelle des Servers. Der Server sendet hierzu eine Nachricht an ein Programm, das Binder genannt wird, und gibt damit seine Existenz bekannt. Dieser Vorgang wird auch als Registrierung des Servers bezeichnet. Dabei übergibt der Server dem Binder seinen Namen, die Versionsnummer (bzw. einen eindeutigen Bezeichner) und ein sogenanntes Handle, das zur Lokalisierung des Servers dient, z. B. eine Ethernet- oder IP-Adresse. Möchte der Server irgendwann keinen Dienst mehr anbieten, so läßt er sich einfach entregistrieren.

Mit diesen Voraussetzungen auf Server-Seite ist dynamisches Binden möglich: Ruft der Client zum ersten Mal ein entferntes Unterprogramm auf, so erkennt der zugehörige Stub, daß der Client noch nicht an einen entsprechenden Server gebunden ist. Dann sendet der Client-Stub eine Nachricht an den Binder, um z. B. eine bestimmte Version einer bestimmten Schnittstelle zu importieren. Der Binder überprüft, ob ein oder mehrere Server eine Schnittstelle mit diesem Namen und der entsprechenden Versionsnummer exportieren. Ist das nicht der Fall, so schlägt die Operation fehl. Existiert jedoch ein passender Server, so liefert der Binder das Handle und den eindeutigen Bezeichner an den Client-Stub zurück. Der Client-Stub benutzt das Handle als Adresse, an die er die Anfragenachricht sendet. Die Nachricht enthält die Parameter und den eindeutigen Bezeichner, mit dem der richtige Server ausgewählt wird, falls mehrere existieren.

Der Vorteil des dynamischen Bindens liegt in der hohen Flexibilität. Der Binder kann seine Server in regelmäßigen Abständen überprüfen und ggf. entregistrieren, Lastausgleiche steuern, wenn mehrere Server den gleichen Dienst anbieten (Load Balancing) und Sicherheitsmechanismen integrieren (z. B. Authentifikation). Der Preis für diese zusätzliche Funktionalität ist der Aufwand für das Im- und Exportieren von Schnittstellen. Außerdem kann der Binder in großen Systemen auch zu einem Engpaß werden, so daß mehrere Binder benötigt werden, die durch ihr zu synchronisierendes Verhalten eine hohe Netzlast bedingen können.

Schlußbemerkungen

Dieses Buch gab einen Überblick über die Konzepte und die Funktionsweise moderner Betriebssysteme. Ein geschichtlicher Rückblick zeigte, wie sich Betriebssysteme von manuell gesteuerten Einbenutzer-Einprozessor-Systemen bis hin zu heutigen parallelen bzw. verteilten Betriebssystemen entwickelt haben.
Eine zentrale Aufgabe des Betriebssystems stellt die Verwaltung der laufenden Programme (Prozesse) dar. Heutige Systeme erlauben mehreren Prozessen, gleichzeitig aktiv zu sein. Die unkontrollierte Änderung von Daten durch mehrere Prozesse kann dabei zur Inkonsistenz dieser Daten führen. Außerdem können durch den Zugriff auf gemeinsame Ressourcen Verklemmungen (Deadlocks) entstehen. Es wurden Mechanismen und Algorithmen vorgestellt, mit denen ein Betriebssystem solche Prozesse synchronisieren und Deadlocksituationen erkennen bzw. vermeiden kann. Durch die Angabe geeigneter Schedulingstrategien wurde außerdem gezeigt, welche Abarbeitungsreihenfolgen für Prozesse besonders günstig sind.
Programme müssen sich zur Laufzeit im Hauptspeicher befinden. Dieser verfügt jedoch normalerweise nicht über die Kapazität, die Gesamtheit der Programme gleichzeitig aufzunehmen. Aus diesem Grund werden Strategien benötigt, die sich mit einer effizienten Organisation und Zuteilung von Hauptspeicherplatz beschäftigen. Von den wichtigsten dieser Strategien, Seitenersetzung und Segmentierung, wurden verschiedene Varianten zusammen mit ihren Vor- und Nachteilen vorgestellt.
Ergänzend zu diesen Konzepten wurden außerdem grundlegende Prinzipien zur Verwaltung von Dateien, Methoden zum Binden und Laden von Programmen, Schutz- und Sicherheitsmechanismen sowie einführende Anmerkungen zum immer mehr an Bedeutung gewinnenden Bereich der Verteilten Systeme präsentiert.

Literatur

Silberschatz, A.; Galvin, P. B.: Operating System Concepts.
Reading (Addison-Wesley) 41994.

Tanenbaum, A. S.: Modern Operating Systems.
Englewood Cliffs (Prentice-Hall) 1992.

Tanenbaum, A. S.: Verteilte Betriebssysteme.
München (Prentice Hall) 1995.

Fisz, Marek: Wahrscheinlichkeitsrechnung und mathematische Statistik.
Berlin (Deutscher Verlag der Wissenschaften) 1989.

Ross, S. M.: Applied Probability Models with Optimization Applications.
San Francisco (Holden Day) 1970

Spaniol, O.; Popien, C.; Meyer, B.: Dienste und Dienstvermittlung in Client/Server-Systemen. Bonn (TAT) 1994.

Popien, C.; Schürmann, G.; Weiß, K.-H.: Das ODP-Referenzmodell für die offene Verarbeitung in Verteilten Systemen. Stuttgart (Teubner) 1995.

Index