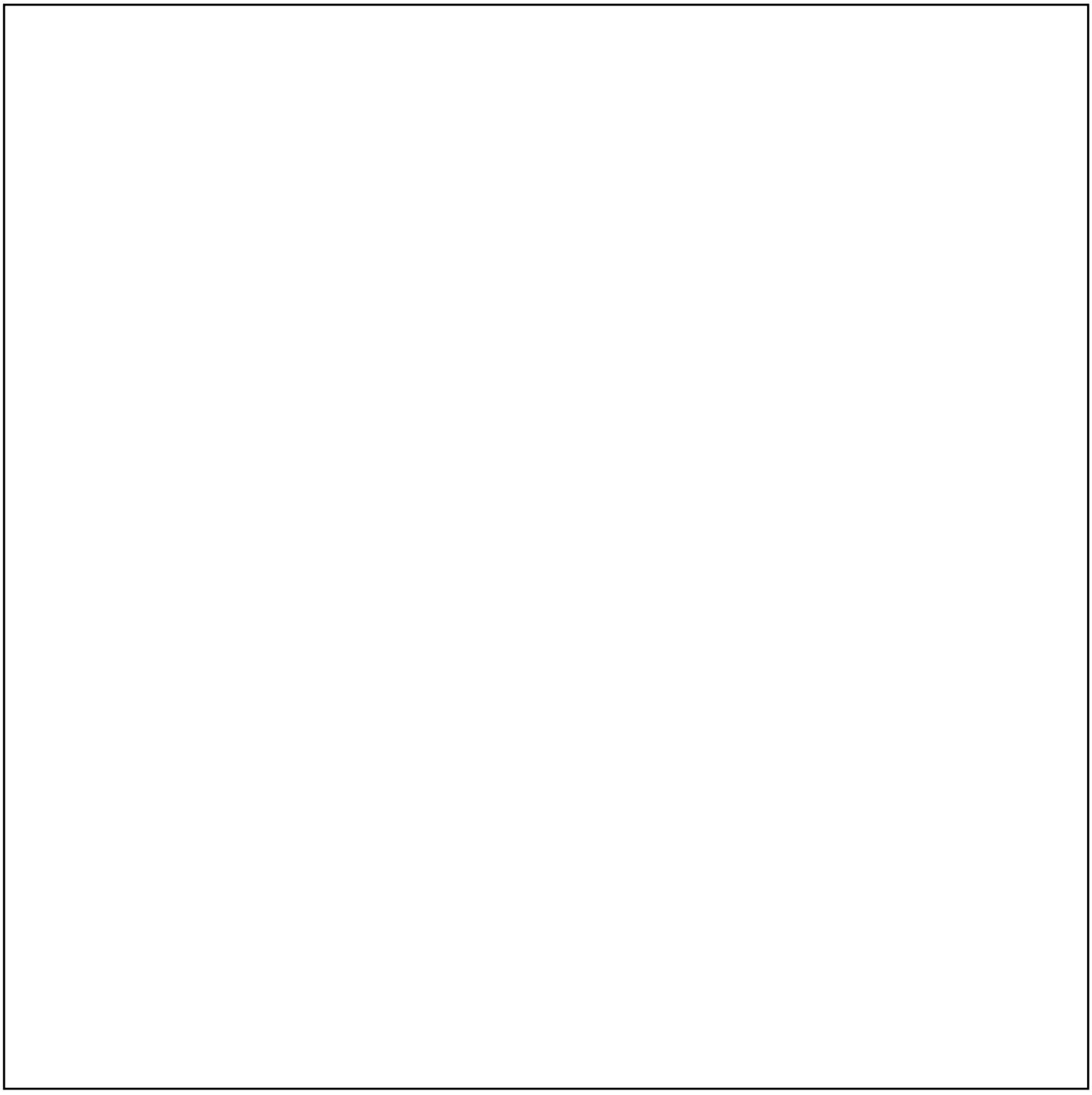
FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 1:
Einführung

Autor: Udo Kelter



Inhalt

Kurseinheit 1

Einführung

Rechner dienen normalerweise dazu, für Benutzer Anwendungsprogramme auszuführen, z. B. ein Buchhaltungssprogramm, ein Datenbankmanagementsystem, einen Übersetzer, eine Anlagensteuerung, ein Spiel oder irgendein Programm, das eine aus Sicht eines Benutzers sinnvolle Aufgabe erfüllt. Die Hardware des Rechners, z. B. die CPU, der Hauptspeicher oder die Ein/Ausgabe-Geräte, ist zur Ausführung derartiger Programme nicht fähig, man benötigt hierzu u. a. ein Betriebssystem, das eine Umgebung bietet, in der die Anwendungsprogramme laufen können. Ein **Betriebssystem** für einen Rechner ist eine Menge von Programmen, die es ermöglichen, den Rechner zu betreiben und Anwendungsprogramme auf ihm auszuführen.

Ganz grob gesagt kann man sich ein Computersystem also in drei Schichten vorstellen, siehe Abbildung 1.1, eine feinere Aufteilung werden wir natürlich später noch vornehmen. Das Betriebssystem steht zwischen Hardware und den Anwendungsprogrammen der Benutzer und kontrolliert und koordiniert das Zusammenspiel.

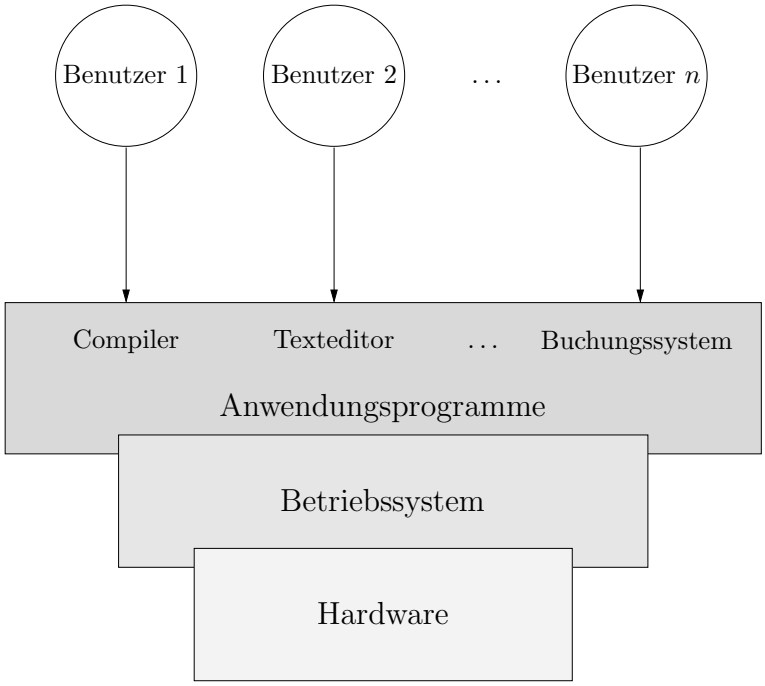


Abbildung 1.1: Rolle des Betriebssystems.

Die obige Definition des Begriffs Betriebssystem ist bewusst sehr allgemein und

unscharf; die Gründe werden im Folgenden klar werden. Tatsächlich lassen sich Dutzende verschiedener Definitionen in Lehrbüchern, Standards, Prospekten usw. finden; beispielhaft wollen wir hier zwei Definitionen aus Standards zitieren:

- DIN 44300: Die Programme eines digitalen Rechners, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.
- ANS (American National Standard): Software which controls the execution of programs and which may provide scheduling, debugging, input/output control, accounting, compilation, storage assignment, data management and related services.

Die Uneinheitlichkeit des Begriffs Betriebssystem ist nicht weiter erstaunlich, wenn man folgendes bedenkt: Die Aufgaben eines Betriebssystems und daraus folgend seine Architektur und Funktionalität hängen offensichtlich stark davon ab, was man mit dem gesamten Rechner anfangen möchte und in welcher Form man ihn benutzen möchte. Hier zunächst einige Beispiele:

- Betrieb eines Flugbuchungssystems für 2000 Reisebüros
- Lohnabrechnung in einer kleineren Firma
- mathematische Berechnungen für einen Statiker
- Textverarbeitung für einen Lehrbuchautor
- Bürokommunikation in einer Firma mit 400 Mitarbeitern
- Steuerung eines Flugzeugs
- Software-Entwicklung
- Fahrzeugsimulation
- Simulation der Luftströmung in einem Windkanal

Die zugehörigen Anwenderprogramme nennen wir im Folgenden **Applikationen**. Man erkennt unschwer, dass die unterschiedlichen Applikationen sowohl qualitativ als auch quantitativ sehr unterschiedliche Anforderungen an den Rechner und insbesondere an das Betriebssystem stellen. So muss z. B. in manchen Fällen der Rechner von mehreren Benutzern parallel benutzbar sein, in anderen Fällen müssen maximale Antwortzeiten garantiert eingehalten werden. Die quantitativen Anforderungen unterscheiden sich oft um mehrere Größenordnungen: die Menge der zu verwaltenden Daten mag nur wenige Megabyte oder viele Terabyte groß sein, Zahl und Heterogenität der benötigten Ein-/Ausgabegeräte und Speichergeräte können stark variieren, die Zahl der parallelen Benutzer kann zwischen 2 und mehreren 1000 schwanken, die erforderliche Rechenleistung mag von einem einfachen, preiswerten Mikroprozessor oder nur von mehreren parallel arbeitenden Höchstleistungsprozessoren erbracht werden können.

Auf der anderen Seite hängt ein Betriebssystem natürlich auch von der vorhandenen Hardware ab. Verteilte oder zentrale Rechner, Rechner mit einem oder mehreren Prozessoren, mit Kanal- oder Bus-Architektur, mit graphischen oder alphanumerischen Terminals, mit oder ohne Schutzfunktionen in der Hardware erfordern bzw.

erlauben jeweils andere Betriebssysteme. Generell kann man sagen, dass die Entwicklungen von Hardware und Betriebssystemen sich gegenseitig stark beeinflusst haben: Viele Betriebssystemfunktionen lassen sich nicht ohne direkte Unterstützung durch die Hardware effizient genug realisieren; umgekehrt haben neue Geräte und Rechnerarchitekturen zu neuen Anforderungen an und Konzepten in Betriebssystemen geführt.

Zu guter Letzt haben die absoluten und relativen Kosten einzelner Komponenten eines Rechners einen großen Einfluss auf die wirtschaftlichen Optimierungsziele und damit, wie wir noch sehen werden, einen enormen Einfluss auf die Funktionalität und Architektur von Betriebssystemen.

Es kann daher nicht *das eine* Betriebssystem geben, das für alle Arten von Anwendungsfällen und Rechnerarchitekturen gleich gut geeignet ist. Genauso wie es sehr unterschiedliche Rechenanlagen gibt, gibt es auch sehr unterschiedliche Betriebssysteme.

Wir wollen es aber natürlich nicht bei dieser vagen Vorstellung von einem Betriebssystem belassen, sondern in der Kurseinheit 1 von verschiedenen Standpunkten aus betrachten, was ein Betriebssystem leisten soll, also welche Anforderungen es erfüllen soll, und was es ist, also aus welchen Komponenten es besteht. Ferner soll diese Kurseinheit einen ersten Eindruck davon vermitteln, wie ein Betriebssystem und die Hardware zusammen funktionieren.

1.1 Aufgaben eines Betriebssystems

In diesem Abschnitt werden wir uns der Frage, was ein Betriebssystem ist, sozusagen von *außen* nähern, d. h. wir werden vor allem die Aufgaben, die ein Betriebssystem aus Sicht der Benutzer bzw. Applikationen übernimmt, betrachten.

1.1.1 Klassische Aufgaben von Betriebssystemen

Eine ganze Reihe von *klassischen* Aufgaben werden von fast allen Betriebssystemen erfüllt, wenn auch oft in ganz unterschiedlicher Weise.

Gerätesteuerung. Eine der wichtigen Aufgaben von Betriebssystemen ist das Verbergen der Besonderheit der Hardware-Geräte von Benutzern und Applikationsprogrammieren. Die Applikationen sollen so weit wie möglich unabhängig von der Hardware sein. Mit anderen Worten müssen alle Dienste, die bzw. deren Realisierung abhängig von der Hardware sind, so weit wie möglich im Betriebssystem lokalisiert werden. Eine klassische Aufgabe eines Betriebssystems besteht daher darin, die im Rechner vorhandenen Geräte (Ein-/Ausgabe-, Speicher- und Kommunikations-Geräte) zu steuern. Aus diesem Grunde werden Betriebssysteme oft als **Steuerprogramme** (**control program**) bezeichnet.

Schutz. Oft wird ein Rechner von mehreren Anwendern benutzt, von denen jeder eigene, private Daten im Rechner speichert. Es ist dann Aufgabe des Betriebssystems, die vorhandenen Speicher in diesem Sinne zu verwalten und die Daten einzelner Benutzer vor unbefugten Zugriffen zu schützen.

Entdeckung und Behandlung von Fehlern. In einem Rechner können die verschiedensten Fehler eintreten: z. B. Division durch 0 oder Benutzung illegaler Adressen bei der Ausführung von Applikationen, Defekte in Leitungen, Datenträgern oder Laufwerken, Papierstau im Drucker usw. Bei jedem Fehler muss das Betriebssystem geeignet reagieren.

Mehrprogrammbetrieb. In vielen Fällen soll ein Rechner gleichzeitig von mehreren Benutzern benutzt werden, wobei jeder Benutzer eine eigene Applikation ausführen lässt. Das Betriebssystem muss dann die parallele Ausführung mehrerer Applikationen ermöglichen. Man spricht hier auch von parallelen Prozessen (multiprogramming). Informell ausgedrückt ist ein **Prozess** ein in Ausführung befindliches Programm.

Realisierung unterschiedlicher Betriebsarten. Eine Betriebsart oder auch Programmausführungsart bestimmt, wie das Betriebssystem Aufträge abarbeitet, speziell hinsichtlich der Kommunikation der ausgeführten Programme mit ihrer Umwelt. Beispiele sind Dialog-, Stapel- und Echtzeitbetrieb.

Prozess-Synchronisation und -Kommunikation. Oft ist es sinnvoll, eine bestimmte Aufgabe durch mehrere parallel ausgeführte Programme zu lösen. Die zusammengehörigen parallelen Prozesse müssen dann i. d. R. Nachrichten austauschen und sich bei der Benutzung gemeinsamer Daten oder anderer Ressourcen synchronisieren können.

Betriebsmittel bzw. Ressourcenverwaltung. Die elementaren, durch die Hardware realisierten Ressourcen eines Rechners sind vor allem:

- die CPU, die zum Ausführen von Anwendungsprogrammen benötigt wird,
- Eingabe- und Ausgabegeräte wie z. B. Bildschirm, Tastatur, Maus, Laufwerke, Drucker,
- Hauptspeicher und insbesondere darin gespeicherte Programme und Daten,
- Kommunikationsverbindungen.

Diese Ressourcen werden von den Anwendungsprogrammen sowie vom Betriebssystem selbst benötigt. Das Betriebssystem muss die Ressourcen verwalten und bei konkurrierenden Anforderungen entscheiden, wie und in welcher Reihenfolge die Anforderungen bedient werden sollen. Hierbei ist anzustreben, alle Anforderungen möglichst rasch, vollständig und fair zu erfüllen, die vorhandenen Ressourcen voll auszunutzen und zugleich den Verwaltungsaufwand zu minimieren. Soweit möglich sollen nur einmal im Rechner vorhandene Ressourcen wie Drucker, Datenfernverbindungen etc. für alle Benutzer verfügbar gemacht werden.

Kommandosprache. Selbst wenn ein ausführbares Programm schon in einem Rechner vorhanden, also gespeichert ist, ist es ein separater Schritt, den Rechner zu beauftragen, dieses Programm auch zu starten und auszuführen und danach die Ausführung zu überwachen und ggf. abzubrechen oder zu beeinflussen. Hierzu bietet

jedes Betriebssystem eine oder mehrere textuelle oder graphische Kommandosprachen an. Aus Sicht eines Endanwenders ist die Kommandosprache oft der präsenteste Teil eines Betriebssystems und hat einen ganz wesentlichen Einfluss auf die Benutzbarkeit und Benutzungsfreundlichkeit des Betriebssystems, während sie aus Sicht eines Implementierers nur einen kleinen Teil des Systems ausmacht[1].

Oft will man nicht nur ein einzelnes Programm ausführen, sondern immer wieder eine bestimmte Folge von Programmen. Hierzu sollte eine Kommandosprache sogenannte **Kommandoprozeduren** anbieten. In einer Kommandoprozedur können die gewünschten Aufrufe textuell notiert werden, die hierbei benutzte Syntax ist manchmal ähnlich wie die von Programmiersprachen, oft aber auch grundlegend verschieden; die Prozedur kann dann mit einem einzigen Kommando ausgeführt werden.

Administration. Beim Betrieb (!) eines Rechners fallen diverse Aufgaben an, die zunächst einmal nicht unmittelbar zur Ausführung von Applikationen beitragen. Beispiele für solche Aufgaben sind Datensicherung, Systemgenerierung, Systemkonfigurierung, Leistungsüberwachung,sonstige Administrationsaufgaben, einfache Textverarbeitung, Modulverwaltung, Formatierung usw. Diese Aufgaben treten oft mehr oder weniger identisch auch als Anwendung auf, so dass hier meist keine eindeutige Trennung zwischen den Applikationsprogrammen und dem Betriebssystem möglich ist.

Abschließend sei noch einmal betont, dass die Funktionen, die einzelne Betriebssysteme zur Lösung der oben genannten Aufgaben und Probleme enthalten, sehr unterschiedlich bzgl. ihres Leistungsumfangs und ihrer konkreten Details sein können; die Festlegung von Art und Umfang dieser Funktionen wird vom intendierten Anwendungsbereich und Annahmen über die Leistungsfähigkeit der unterliegenden Hardware und deren Kosten abhängen.

1.1.2 Standard-Bibliotheken und Dienstprogramme

Betrachten wir noch einmal die obigen Beispiele von Anwendungen. Es stellt sich hier die Frage, wie sich die skizzierten Aufgaben und daraus abgeleitete Teilaufgaben auf die Komponenten eines Rechnersystems gemäß Abbildung 1.1 verteilen. Für viele erforderliche Funktionen oder **Dienste** des *gesamten* Systems gilt, dass die Komponente, in der der Dienst realisiert wird, anhand von Kosten/Nutzen-Abschätzungen und anderen Zweckmäßigkeitserwägungen zu bestimmen ist und nicht automatisch eindeutig bestimmt ist. Manche (Teil-) Aufgaben können u. U. nach Abwägung von Kosten und Leistungsanforderungen günstiger in der Hardware oder in den *unteren Schichten* der Anwendungsprogramme erfüllt werden anstatt im Betriebssystem. Nehmen wir z. B. an, in einer Applikation würden Fließkommazahlen multipliziert. Es ist nun keineswegs eindeutig, auf welcher Ebene der Dienst *Multiplikation von Fließkommazahlen* realisiert wird: er könnte realisiert sein

[1] Umso unverständlicher ist, wie wenig Aufmerksamkeit den Kommandosprachen (speziell im Vergleich zu Programmiersprachen) in der Vergangenheit zuteil geworden ist. Speziell die Kommandosprachen konventioneller Großrechner sind schlecht bis katastrophal. Merkliche Fortschritte in diesem Bereich sind erst durch Betriebssysteme für Mini-Computer und PCs zu verzeichnen gewesen.

- in der Hardware durch einen entsprechenden Prozessor
- im Betriebssystem durch ein entsprechendes Unterprogramm
- auf der Ebene der Anwendungsprogramme durch ein entsprechendes Unterprogramm in der Bibliothek, die zum Compiler der benutzten Sprache gehört.

Ein anderes Beispiel ist eine indexsequentielle Zugriffsmethode für Dateien, bei der Datensätze als Speichereinheiten auf Basis von Blöcken realisiert werden. Man kann die Zugriffsmethode innerhalb des Betriebssystems oder durch die Applikationen realisieren. Das Modul, das die Operationen der Zugriffsmethode implementiert, kann sogar in beiden Fällen das gleiche sein, d. h. es ist bzgl. der Funktionalität völlig gleichgültig, ob man dieses Modul zum Betriebssystem oder zu den Applikationen zählt.

Völlig beliebig ist die Lokalisierung einzelner Dienste natürlich nicht. Zunächst einmal gelten einige pragmatische Richtlinien:

- Leistungsanforderungen können eine Realisierung in tieferen Schichten erzwingen, also in der Hardware.

- Häufige Teilaufgaben von Applikationen sollten nicht jedesmal erneut in den Applikationen realisiert werden, sondern *herausfaktorisiert* und in eine nächsttiefere Schicht verlagert werden. Diese Funktionen können dann *mitten im Betriebssystem* oder in einer **Standard-Bibliothek**, also sozusagen in einer Ebene zwischen den Applikationen und dem *Kern* des Betriebssystems, realisiert werden; wo man diese Grenze zieht, ist aber eher Geschmackssache.

  Was häufig oder selten ist, hängt durchaus vom intendierten Anwendungsgebiet des Rechners ab. Generell sollte das Betriebssystem nur elementare, im Anwendungsgebiet breit anwendbare und nicht auf spezielle Anwendungen zugeschnittene Funktionen enthalten.

- Wie schon oben erwähnt, unterscheiden sich die Administrationsaufgaben z. T. praktisch nicht von *normalen* Anwendungen. Diese Aufgaben werden deshalb üblicherweise von sogenannten **Dienstprogrammen** (**utilities**) übernommen. Dienstprogramme werden genau wie *normale* Applikationen aufgerufen und unterscheiden sich von diesen eigentlich nur dadurch, dass sie eine ganz allgemein auftretende, nicht anwendungsspezifische Aufgabe lösen. Obwohl sie also eher neben den Benutzeranwendungen stehen, stellen sie vor allem im Hinblick auf die Administration von Betriebssystemen einen ganz wesentlichen Teil des Betriebssystems dar.

Der verbliebene *Kern* des Betriebssystems umfasst die Programme, die beim laufenden System im Hauptspeicher geladen sein müssen, um die im laufenden Betrieb potentiell jederzeit auftretenden Aufgaben behandeln zu können. Sie unterscheiden sich von Bibliotheksfunktionen typischerweise auch dadurch, dass sie besonders geschützt werden müssen. Wir werden in diesem Kurs die Standard-Bibliotheken und die Dienstprogramme als Teil des Betriebssystems auffassen. Wir können nunmehr den Kasten *Betriebssystem* in unserer ersten groben Skizze von einem Rechnersystem bereits etwas verfeinern, siehe Abbildung 1.2.

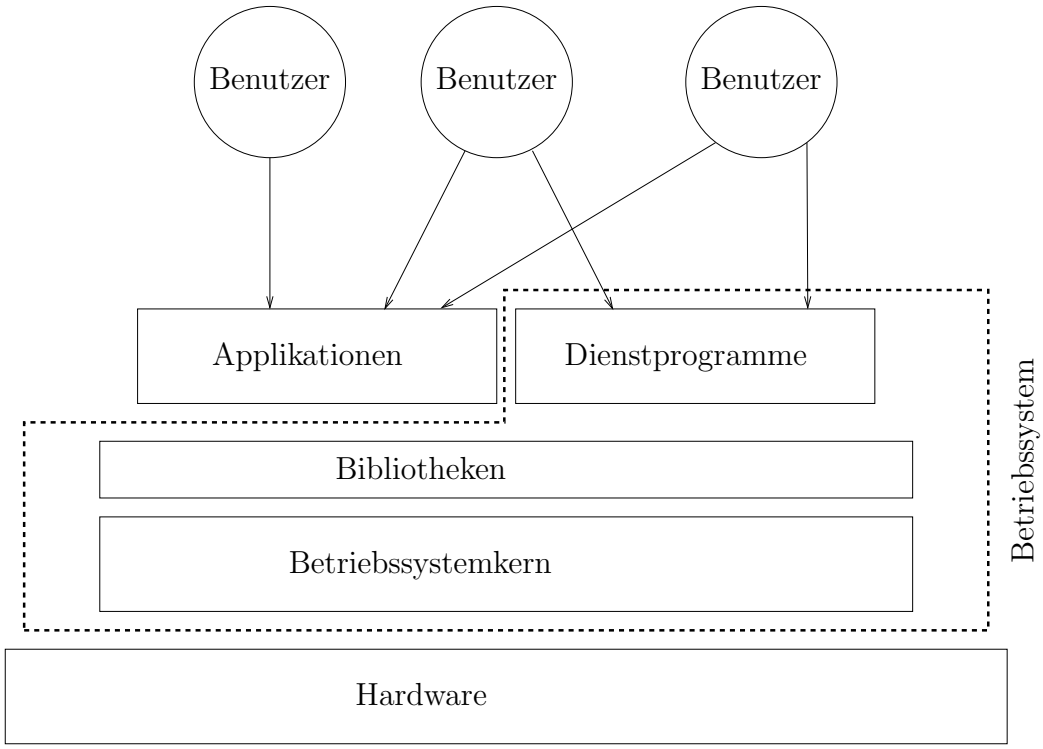


Abbildung 1.2: Dienstprogramme, Bibliotheken und Betriebssystemkern.

1.1.3 Systemsoftware vs. Betriebssystem

Es gibt eine Reihe von Aufgaben bzw. zugehörige Programme, die relativ hardwarenah und applikationsunabhängig sind, die aber meist nicht (auch nicht in diesem Text) als Aufgabe bzw. Teil des Betriebssystems angesehen werden. Dies hat meist historische Gründe: diese Problembereiche haben sich zwar in engem Kontakt mit, aber dennoch unabhängig von Betriebssystemen entwickelt sowohl als Wissenschaft wie auch als Menge auf dem Markt erhältlicher Produkte und ein so großes Eigengewicht, dass sie ohnehin separat behandelt werden müssen. Die zugehörige Software nennt man **Systemsoftware**. Die Begriffe Systemsoftware und Betriebssystem werden oft synonym benutzt. Wir fassen hier Systemsoftware als den allgemeineren Begriff auf, das Betriebssystem ist ein Teil der Systemsoftware. Wir beschreiben i. F. einige dieser an das Betriebssystem *angrenzenden* Systeme, um von diesen Seiten her den Begriff Betriebssystem weiter einzugrenzen.

Übersetzer. Eine Applikation ist üblicherweise in einer höheren Programmiersprache wie Pascal geschrieben, die unabhängig von dem konkreten Rechner ist. Um die Applikation auf einem konkreten Rechner ausführen zu können, müssen mehrere Schritte durchlaufen werden:

- Quellprogramme sind i. d. R. nicht monolithisch, sondern bestehen aus selbständig übersetzbaren Quellprogramm-Moduln. Ein Quellprogramm-Modul wird zunächst von einem Compiler oder Assemblierer[2] in ein **Bindemodul** (oft auch **Objekt-Modul** genannt) übersetzt.

[2]Anstelle von Übersetzern können auch Interpreter auftreten. Dieser Unterschied ist aber hier nicht wesentlich.

- Aus einem oder mehreren Bindemoduln, darunter üblicherweise Moduln aus Standard-Bibliotheken, erzeugt dann ein **Binder** ein **Lademodul**.
- Das Lademodul wird vom **Lader** in den Speicher des Rechners geladen. Danach wird es gestartet und ausgeführt.

Hieraus ergibt sich unmittelbar die Frage, ob Compiler, Binder und Lader auch zum Betriebssystem gehören, denn sie sind ja offenbar zur Ausführung der Applikationen erforderlich. Es gibt keine absolut gültige Antwort auf diese Frage. Üblicherweise zählt man Compiler nicht zum Betriebssystem, Binder manchmal, Lader immer. In manchen Rechnern existieren überhaupt keine Übersetzer oder Binder; die entsprechenden Aufgaben müssen dann auf anderen Rechnern (ggf. eines ganz anderen Typs) erledigt werden (mit Hilfe sogenannter Cross-Compiler). In anderen Systemen können Programme dynamisch gebunden werden; das Binden ist hier also Teil der vom Betriebssystem überwachten Programmausführung. Insgesamt gibt es keine ganz scharfe Grenze zwischen dem Betriebssystem und dem **Programmentwicklungssystem**, also der Menge aller zur Entwicklung von Programmen benötigten Programme. Dies zeigt sich auch deutlich an Testhilfeprogrammen (debugger): diese erlauben es typischerweise, den Arbeitsspeicher oder die Register eines in Ausführung befindlichen Programms zu inspizieren und zu verändern. Dies ist natürlich nur mit Hilfe des Betriebssystemkerns realisierbar.

Die für die Programmentwicklung und die Administration eines Rechners benötigten Dienstprogramme überschneiden sich sehr stark. Deshalb kann man die Menge der Dienstprogramme auch als einen eigenen, selbständigen Teil der Systemsoftware ansehen, siehe z. B. [Ca82]).

Datenbankmanagementsysteme. Eine der klassischen Aufgaben von Betriebssystemen ist die Verwaltung von Sekundärspeichern. Dies geschieht in der Regel durch Dateisysteme, d. h. den Benutzern werden Dateien als Einheiten zur permanenten Speicherung von Daten zur Verfügung gestellt. Dateien bieten aber nur relativ elementare Möglichkeiten zur Strukturierung der Daten. Für große Anwendungsbereiche, vor allem betrieblich/administrative Standardaufgaben, bieten Datenbankmanagementsysteme (DBMS) wesentlich bessere Möglichkeiten zur Datenstrukturierung; durch die relativ mächtigen Abfragesprachen, Reportgeneratoren und andere zugehörige Werkzeuge wird außerdem die Entwicklung von Applikationen wesentlich vereinfacht.

DBMS arbeiten teilweise auf Basis von Dateisystemen (vor allem, wenn diese bereits satzorientierte Zugriffsmethoden zur Verfügung stellen), teilweise direkt auf Sekundärspeichern, umgehen also z. T. Dienste des Betriebssystems, da eine eigene Verwaltung der Sekundärspeicher besser auf die Merkmale ihrer Benutzung eingehen kann. Insofern muss das Betriebssystem die teilweise Verwaltung von Speichergeräten durch Applikationen ermöglichen. Ähnliches gilt für die Verwaltung von Hauptspeicherpuffern oder Datenkommunikationsverbindungen (auf zentrale Datenbanken wird oft über Netzwerke zugegriffen).

Es sind schon einige sogenannte Datenbank-Betriebssysteme entwickelt worden, die speziell auf die Bedürfnisse von DBMS hin optimiert worden sind und Basisfunktionen anbieten, die die Konstruktion von DBMS erleichtern. Allerdings sind auch *normale* Betriebssysteme in letzter Zeit zunehmend besser in der Lage, die

Konstruktion von DBMS zu unterstützen; Ursachen sind Verbesserungen der Funktionalität der Betriebssysteme und Leistungssteigerungen der Hardware.

Datenfernübertragung und Rechnernetze. Eine weitere Standardaufgabe von Betriebssystemen ist die Verwaltung der an den Rechner angeschlossenen Terminals und anderer Ein-/Ausgabegeräte. Sofern diese in größerer Entfernung vom Rechner aufgestellt sind, treten viele eigene Probleme auf: die wichtigsten sind der Schutz vor Übertragungsfehlern infolge von magnetischen oder elektrischen Störfeldern und vor unautorisiertem Abhören der übertragenen Daten. Gegenmaßnahmen sind Protokolle, durch die Übertragungsfehler entdeckt und behoben werden, bzw. Verschlüsselungen. Weitere Protokolle regeln z. B. den Auf- und Abbau von Verbindungen. Die gleichen Protokolle werden zur Kommunikation zwischen Rechnern benutzt. Dieser umfassende Themenbereich wird ausführlich in Unger [Un02] behandelt. Aspekte verteilter Systeme behandelt zudem [HILS05]. Wir gehen daher hier nicht weiter auf dieses Thema ein.

Fenster- bzw. Graphik-Systeme. Während in den 1970er Jahren noch alphanumerische Terminals üblich waren, haben sich in den 1980er Jahren graphikfähige Workstations bzw. Terminals auf breiter Front durchgesetzt. Diese bieten weitaus umfangreichere Möglichkeiten zur Gestaltung von Benutzungsschnittstellen; andererseits ist aber auch der Realisierungsaufwand wesentlich größer als bei Benutzungsschnittstellen auf Basis von alphanumerischen Terminals. Es stellte sich außerdem heraus, dass die graphischen Benutzungsschnittstellen, also sozusagen die graphischen Kommandosprachen, von verschiedenen Applikationen meist sehr inhomogen sind, was für die Benutzer sehr störend ist, weil es den Lernaufwand erhöht. Zur Reduzierung des Programmieraufwands und zur konstruktiven Vereinheitlichung der Benutzungsschnittstellen wurden mehrere sogenannte Fenstersysteme geschaffen. Diese bieten (Bibliotheks-) Funktionen zum Aufbau von Fenstern und anderen graphischen Objekten an, darüber hinaus aber auch Funktionen für komplexere Dialoge, z. B. geschachtelte Menüs. Wenn man so will, werden hier abstraktere, problemnähere Geräte simuliert, ähnlich wie Dateien problemnähere Speichermedien als „nackte“ Magnetplatten sind. Fenstersysteme sind außerdem oft verteilt, so dass sich noch Bezüge zu Rechnernetzen ergeben.

Obwohl Fenstersysteme ein ganz wesentlicher Teil von graphikfähigen Rechnern sind und der Rechner ohne das Fenstersystem praktisch nicht benutzbar ist, zählt man Fenstersysteme nicht zum Betriebssystem.

1.2 Architektur von Betriebssystemen

Nachdem wir uns in Abschnitt 1.1 vor allem aus Sicht eines Benutzers (bzw. Applikationsprogrammierers) damit beschäftigt hatten, was ein Betriebssystem leisten kann oder sollte, nehmen wir nun einen anderen Standpunkt ein: wir wollen jetzt einen ersten Eindruck davon gewinnen, wie ein Betriebssystemkern *von innen* aussieht. Wir werden uns zunächst ein genaueres Bild von der Schnittstelle nach *oben*, also zu den Applikationen hin, und *nach unten*, also zur Hardware hin, verschaffen und danach einige sehr einfache, grundlegende Lösungsansätze für die wichtigsten

Aufgaben von Betriebssystemen vorstellen. Diese bilden aber immerhin eine ausreichende Basis, um danach weitergehende Aufgaben beschreiben zu können.

1.2.1 Ein Ebenenmodell für Rechner

Es erweist sich oft als hilfreich, komplexe Systeme als eine Hierarchie von *Ebenen* bzw. *virtuellen Maschinen* anzusehen. Jede virtuelle Maschine bietet nach oben eine gewisse Menge von *Diensten*, also Objekte und Operationen bzw. Sprachen, an. Man konzentriert sich hierbei auf die funktionalen Aspekte des Systems; von nichtfunktionalen Aspekten wie Performance oder Parallelität wird hier normalerweise abstrahiert. Jede Schnittstelle nach oben wird realisiert innerhalb einer virtuellen Maschine unter Benutzung der Dienste der nächsttieferen virtuellen Maschine.

Ein Ebenenmodell, welches alle wichtigen Sprachen in einem Rechner umfasst, ist in Abbildung 1.3 angegeben. Wir gehen i. F. nur auf die beiden Ebenen oberhalb und unterhalb des Betriebssystems näher ein.

höhere Programmiersprachen
Assembler-Sprachen
Betriebssystem-Ebene
konventionelle Maschinenebene
Mikroprogramm-Ebene
digitale Logikebene

Abbildung 1.3: Ebenenmodell eines Rechners.

Digitale Logik. Auf der Ebene der digitalen Logik betrachtet man einen Rechner als eine Menge von Gattern, die Boole'sche Funktionen realisieren. Neben den rein digitalen Komponenten besteht ein Rechner auch noch aus Wandlern, die Informationen zwischen analoger und digitaler Darstellung umwandeln können; Beispiele sind entsprechende Wandler in Bildschirmen, Modems, Tongeneratoren, Druckern, Mikrophonen, Messwertgebern, Tastaturen, Anzeigelampen usw.; hierauf gehen wir nicht näher ein.

Mikroprogramm-Ebene. Die oben erwähnten Gatter sind in einer Weise zusammengeschaltet, dass elementare Funktionen realisiert werden; ein Beispiel für einen solchen *Mikrobefehl* ist der Transport des Inhalts eines Registers in ein anderes. Weiter sind Mechanismen vorhanden, durch die die Ausführung von Mikrobefehlen mit Hilfe von Mikroprogrammen ausgelöst und kontrolliert werden kann.

Konventionelle Maschinenebene. Jeder Maschinenbefehl eines Prozessors wird durch ein Mikroprogramm realisiert. Jeder Prozessor definiert eine Menge von Maschinenbefehlen. Häufig in Kleinrechnern auftretende Zentralprozessoren (CPUs) sind z. B. Intel Pentium und PowerPC. Die konventionelle Maschinenebene ist die

tiefste Ebene, die normalerweise für Programmierer eines Rechners erreichbar ist, die beiden untersten Ebenen sind normalerweise durch die Hardware und Firmware realisiert, nur bei den sehr seltenen mikro-programmierbaren Rechnern sind die Mikroprogramme austauschbar. Neben der CPU (bzw. mehreren CPUs) treten in einem Rechner i. A. weitere periphere Prozessoren auf, z. B. in Hardware-Schnittstellen; diese Prozessoren können auch in Geräten untergebracht sein. Die Prozessoren kommunizieren untereinander; wichtigstes Mittel hierzu sind *Unterbrechungen*; dies sind Signale, durch die die peripheren Prozessoren die CPU bei dem Programm, das sie gerade ausführt, unterbrechen können. Hierauf gehen wir in Abschnitt 1.2.3 noch genauer ein.

Betriebssystemebene. Die Betriebssystemebene ist eine Erweiterung der konventionellen Maschinenebene, d. h. es kommen bestimmte Operationen hinzu, sogenannte **Systemaufrufe**[3] (**system calls**). Viele dieser neuen Operationen arbeiten auch mit neuen Typen von Objekten oder Ressourcen, z. B. Dateien oder Prozessen, die auf der konventionellen Maschinenebene noch nicht existieren und die im Betriebssystem simuliert werden. Typische Systemaufrufe sind, siehe [POSIX88]:

- Operationen mit Dateien und Dateiverzeichnissen z. B. anlegen, löschen, lesen, schreiben, Zustand abfragen usw.
- Operationen mit Prozessen z. B. Prozess erzeugen (incl. Programm laden), starten, beenden, steuern, überwachen, austauschen von Nachrichten zwischen Prozessen
- E/A-Operationen mit Geräten wie Terminals oder Bandgeräten
- Operationen zur Hauptspeicherverwaltung
- Setzen und Abfragen von Systemparametern
- allgemeine Zustandsabfragen z. B. Systemuhr, aktive Prozesse, usw.[4]

Die Menge der Systemaufrufe nennt man auch (**Applikations-**) **Programmierschnittstelle** (**application programming interface**, **API**). Zu jedem Betriebssystem gehört eine Spezifikation seines API. Diese wird allerdings normalerweise nicht auf dieser Ebene, sondern auf der Ebene der Assemblerprogramme oder sogar der höheren Programmiersprachen formuliert sein.

Unterbrechungen und gewisse Zeitverzögerungseffekte sind auf der Betriebssystemebene nicht mehr sichtbar. Die Syntax von Programmen auf dieser Ebene ist

[3] Die Bezeichung System*aufruf* ist üblich wohl eher aus historischen Gründen, aber leider nicht besonders glücklich: es handelt sich hier um eine *Operation*, die durch eine Spezifikation beschrieben wird. Man kann also einen Systemaufruf aufrufen, ausführen oder abbrechen, was alles etwas seltsam klingt. Besser wäre daher die Bezeichnung Systemoperation oder -funktion. In [POSIX88] wird die Bezeichnung *primitive* benutzt, was aber auch nicht anschaulicher ist.

[4] Beim Vergleich mit der Liste der Aufgaben eines Betriebssystems in Abschnitt 1.1.1 fällt auf, dass Operationen für eine ganze Reihe dort genannter Aufgaben hier fehlen. Ursache hierfür ist, dass diese Aufgaben die interne Funktionsweise des Betriebssystems betreffen; diese stellen keinen Dienst dar, der von einer Applikation aufgerufen werden könnte. Man kann jedoch oft Parameter für diese Funktionsbereiche einstellen (z. B. mit welcher Priorität welche Klassen von Aufträgen bearbeitet werden sollen). Zur Übergabe solcher Parameter und für zugehörige Abfragen kann es dann wieder Operationen geben; oft werden diese Werte aber auch nur einmal beim Starten des Betriebssystems aus bestimmten Dateien eingelesen und können im laufenden Betrieb nicht mehr verändert werden.

immer noch binär, also für Programmierer nicht gerade angenehm zu lesen (in einem eines Programms beispielsweise; ein Dump ist eine druckbare, meist hexadezimale Darstellung des Inhalts des Hauptspeichers).

Assemblersprache. Auf der Assembler-Ebene sind deshalb Maschinenprogramme in einer besser lesbaren Form dargestellt: es werden lesbare Namen für Maschinenfehle (z. B. MOVE, ADD usw.) und für Register (R0, R1, ...) sowie symbolische Namen für Hauptspeicheradressen verwandt. Jeder Maschinenbefehl wird durch genau einen Assembler-Befehl dargestellt; jede Assembler-Sprache bezieht sich also auf genau einen Prozessor. Hinzu kommen Makros und technische Hilfsmittel zum modularen Programmieren, siehe auch Abschnitt 1.1.3. Systemaufrufe werden i. A. durch Makros in einer lesbaren Form dargestellt.

Höhere Programmiersprachen. Im Gegensatz zu Assembler-Sprachen sind höhere Programmiersprachen weitestgehend unabhängig von konkreten Prozessoren und Betriebssystemen. Vom Prozessor und dem Betriebssystem hängt aber natürlich die Codeerzeugung im Compiler ab: bei der gleichen Sprache braucht man also für jeden Prozessor und jedes Betriebssystem einen eigenen Compiler. Für uns ist speziell interessant, welche *Teile* einer Sprache also vor allem welche Arten von Anweisungen zu Abhängigkeiten vom Betriebssystem führen. Typischerweise fallen diese in zwei Bereiche: Ein-/Ausgabe und Parallelität.

Viele Sprachen enthalten spezielle Anweisungen für die Ein-/Ausgabe, teilweise noch einmal getrennt für Dateien und *Standard* -Geräte wie Tastatur, Bildschirm oder Drucker. Bei den Geräten liegt diesen Anweisungen oft ein relativ stark vereinfachtes Modell der Funktionsweise der Geräte zugrunde: meist werden einfach Byte-Folgen an das Gerät ausgegeben; spezielle Effekte wie Hell/Dunkel-Darstellung oder Zeilenvorschübe werden durch geräteabhängige Sonderzeichen im ausgegebenen Text bewirkt. Diese Sonderzeichen muss der Applikationsprogrammierer kennen, die Programmiersprache unterstützt oder kennt derartige Details nicht. Bei den Dateien wird oft eine der gängigen, in Betriebssystemen realisierten Zugriffsmethoden unterstellt. Zeichen- oder satzorientierte Zugriffsmethoden z. B. werden in Kurseinheit 2 vorgestellt.

Viele Sprachen enthalten gar keine Anweisungen für die Ein-/Ausgabe. Stattdessen gehört zu jedem Compiler eine Bibliothek von E/A-Funktionen, die wie normale, getrennt übersetzte Unterprogramme aufgerufen werden, also keine spezielle Syntax für E/A-Anweisungen in der Sprachdefinition erfordern[5]. Dieser Ansatz ist flexibler als fest definierte E/A-Anweisungen; je nach den vorhandenen E/A-Geräten braucht man nur diese Bibliothek anzupassen; bei neuartigen E/A-Geräten braucht man nur die Bibliothek passend zu erweitern, z. B. sind graphische Bildschirme fast immer nur über die E/A-Bibliotheken eines Fenstersystems benutzbar. Weder Sprachdefinition noch Compiler müssen angepasst werden. Noch wichtiger ist, dass der Compiler überhaupt nicht mehr bzgl. der Ein-/Ausgabe vom Betriebssystem abhängt,

[5]Diese E/A-Funktionen sind i. A. *nicht* identisch mit den Systemaufrufen für die Ein-/Ausgabe. Die Systemaufrufe sind i. d. R. wesentlich primitiver. So mag für die Ausgabe auf einen Bildschirm nur ein Systemaufruf zur Verfügung stehen, der Texte ausgibt. Andererseits mag es eine Bibliotheksfunktion geben, die den Wert einer Variablen vom Typ real ausgibt. Dieser Wert muss zunächst einmal in einen Text konvertiert werden; erst das Ergebnis der Konversion kann dann vom dem Systemaufruf verarbeitet werden.

also leicht auf ein neues Betriebssystem portiert werden kann. Außerdem kann eine einzige E/A-Bibliothek von verschiedenen Sprachen aus benutzt werden, vorausgesetzt, einige technische Probleme bei der Parameterübergabe werden gelöst. Man wird solche E/A-Bibliotheken daher eher als Teil des Betriebssystems oder der System-Software denn als Teil der Sprache bzw. des Compilers auffassen, siehe Abbildung 1.2. Die Flexibilität von E/A-Bibliotheken hat allerdings auch einen Nachteil: da sie nicht Teil der Sprachdefinition sind, sind sie oft auch nicht standardisiert, so dass Programme u. U. nicht mehr zwischen verschiedenen Betriebssystemen portabel sind.

Einige wenige höhere Programmiersprachen haben Konstrukte, die die parallele Ausführung von Programmteilen ermöglichen; ggf. können solche parallelen Progammzweige sogar Nachrichten austauschen oder sich synchronisieren. Man kann derartige parallele Programmzweige entweder im Laufzeitsystem der Sprache oder durch eigene Betriebssystem-Prozesse realisieren. Die zweite Lösung ist in den meisten Betriebssystemen nicht praktikabel, weil der Aufwand zum Erzeugen und Synchronisieren von Prozessen um mehrere Größenordnungen zu hoch ist. Die umfangreiche Funktionalität von Betriebssystem-Prozessen z. B. eigener Adressraum, eigene geöffnete Dateien usw. verursacht viel Aufwand, weswegen man die Prozesse auch *schwergewichtig* nennt. Die umfangreiche Funktionalität wird außerdem für solche parallelen Programmzweige nicht benötigt. Der Trend geht aber dahin, in Betriebssystemen zur Befriedigung derartiger Anforderungen zusätzlich *leichtgewichtige* Prozesse (Threads) anzubieten, siehe dazu auch Kurseinheit 3.

Kommandosprachen. Im obigen Ebenenmodell wurden Kommandosprachen nicht erwähnt; tatsächlich liegen Kommandosprachen neben der oben eingeführten Hierarchie. Vom Sprachniveau her liegen sie im Bereich von Assemblersprachen bis zu höheren Programmiersprachen. Sie können aber auch in Verbindung mit Maschinenprogrammen benutzt werden.

Ein anderes Einordnungsmerkmal ist die Ebene, in der sie realisiert werden. Der Kommandointerpreter war in alten Betriebssystemen Teil des Kerns; vielfach waren sogar wichtige Funktionen des Betriebssystemkerns *nur* über die Kommandosprache aufrufbar und nicht über das API. In modernen Betriebssystemen werden Kommandointerpreter als ganz normale Applikationen ausgeführt: wenn ein Kommando, dass ein bestimmtes Programm ausgeführt werden soll, eingegeben worden ist, ruft der Kommandointerpreter, der auch *Shell* genannt wird, eine API-Operation auf, die einen neuen Prozess anlegt und das auszuführende Programm in den zugeordneten Speicherbereich lädt und startet; danach wartet die Shell auf das Ende des von ihm gestarteten Prozesses. Dieser Ansatz ist wesentlich flexibler und bietet sehr große Vorteile: erstens wird der Betriebssystemkern kleiner, zweitens ist die Kommandosprache leicht auszutauschen, wenn ein Benutzer eine eigene Kommandosprache haben und den zugehörigen Interpreter implementieren möchte oder wenn für neue E/A-Geräte eine neue Kommandosprache benötigt wird, z. B. bei der Umstellung von alphanumerischen auf graphische Terminals.

1.2.2 Ein einfaches Betriebssystem

Um nun endlich eine ungefähre Vorstellung davon zu bekommen, wie ein Betriebssystemkern aussieht, stellen wir in diesem Abschnitt die Struktur eines besonders

einfachen Betriebssystems vor, nämlich CP/M (Control Program for Microcomputers), das in den frühen 1980er Jahren populär war. CP/M war für Kleinstrechner gedacht, die einen sehr einfachen Zentralprozessor (8-Bit-Prozessoren wie Intel 8080 oder Zilog Z80) hatten und an denen nur genau ein Benutzer mit genau einem Programm arbeitete. Von der langen Liste von Aufgaben eines Betriebssystems aus Abschnitt 1.1.1 bleibt hier i. W. nur noch die Datei- und Geräteverwaltung übrig; das soll uns aber jetzt nicht weiter stören, wir werden komplexere Systeme später kennenlernen. Der Hauptspeicher war bei CP/M mit der Größe 64 Kbyte wie in Abbildung 1.4 aufgeteilt.

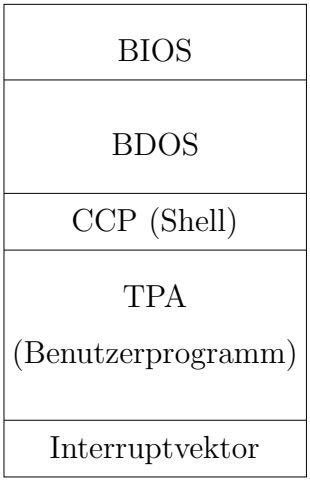


Abbildung 1.4: Struktur des Hauptspeichers bei CP/M.

An der Spitze des Hauptspeichers steht das BIOS (Basic Input Output System). Dieses enthält alle *hardwareabhängigen* Teile des Betriebssystems, insb. die sogenannten (**Geräte-**) **Treiber**. Ein Gerätetreiber ist die Software, die mit dem **Controller** eines Geräts kommuniziert. Zu jedem Gerät bzw. jeder Hardware-Schnittstelle gehört ein passender Treiber. Der Controller besteht aus einem oder auch mehreren Chips, die das Gerät kontrollieren. Die Funktionsweisen von Treibern und Controllern werden in Kurseinheit 2 erläutert.

Unterhalb des BIOS liegt das Betriebssystem BDOS (Basic Disc Operating System). BDOS ist eine Komponente, die Dateien auf Disketten realisiert. Hierzu bietet das BDOS eine Reihe von Systemaufrufen an. Details, wie man die Sektoren einer Diskette oder Magnetplatte beim Anlegen und Löschen von Dateien verwaltet, werden wir in Kurseinheit 2 kennenlernen. Außerdem bietet das BDOS Systemaufrufe für die Konsole, einen Drucker und je ein zeichenorientiertes Ein- bzw. Ausgabegerät, z. B. ein Modem, an.

Alle Systemaufrufe (also Schnittstellen zu den Diensten, die das BDOS anbietet) sind *hardwareunabhängig*, d. h. Applikationen können auf verschiedenen Rechnern, die eine eigene Implementierung von CP/M haben, unverändert laufen. Die Implementierung von CP/M muss demnach hardwareabhängig sein. Andererseits hängen viele Detailaufgaben überhaupt nicht von der Hardware ab, z. B. die Syntaxüberprüfung von Dateinamen, ebenso viele weitere Funktionen. Bei allen Betriebssystemen sind viele Funktionen also hardwareunabhängig. Um also *das gleiche Betriebssystem* (gemeint ist hier das gleiche API) leicht auf Hardware-Plattformen

portieren zu können, teilt man den Kern in hardwareabhängige bzw. -unabhängige Teile. BDOS gehört zum hardwareunabhängigen Teil von CP/M.

Der CCP (Console Command Processor) ist ein Interpreter (Shell), der in einer Endlosschleife Kommandos von der Konsole liest, analysiert und dann mit Hilfe der Funktionen, die das BDOS anbietet, ausführt. Nur wenige Kommandos werden direkt im CCP implementiert. Die meisten Kommandos sind nichts anderes als Namen von Dienstprogrammen, die wie normale Applikationen ausgeführt werden. Hierbei wird die Datei, die das ausführbare Programm enthält, in die TPA (Transient Program Area) kopiert, die man auch als *Benutzerbereich* bezeichnen kann, und es wird an den Anfang dieses Programms gesprungen. Die Applikation beendet sich selbst, indem sie in das CCP zurückkehrt.

Das BDOS wird nicht nur vom CCP benutzt, sondern auch von den Applikationen: wenn diese z. B. einen Text auf die Konsole ausgeben wollen, benutzen sie entsprechende Systemaufrufe.

1.2.3 Unterbrechungen

Das Betriebssystem tut von sich nichts – es wartet normalerweise auf Aufforderungen *von außen*, etwas zu tun. Diese Aufforderungen stammen vor allem von den Applikationen (Software) oder der Hardware. Jede Aufforderung wird durch eine Unterbrechung signalisiert. Hardware kann zu jeder Zeit eine Unterbrechung auslösen und die CPU erfährt sie, indem ein Signal gesendet wird. Software kann Unterbrechungen (sogenannte Traps) auslösen, wenn z. B. ein Systemaufruf ausgeführt wird.

Betrachten wir hierzu ein Beispiel: Nehmen wir an, die CPU führt gerade das Programm der Shell aus, das Betriebssystem *wartet*. Als nächstes ruft die Shell einen Systemaufruf auf, durch den eine Zeile auf die Konsole ausgegeben wird, das Betriebssystem wird also jetzt aktiviert. Es analysiert den Systemaufruf (Details folgen später) und stellt das erste auszugebende Zeichen fest. Dieses Zeichen wird nun an den Treiber für den Bildschirm übergeben und der Treiber wird gestartet. Im einfachsten Fall braucht der Treiber jetzt nur das Zeichen und ggf. seine Position oder andere Anweisungen in ein spezielles Register oder an eine feste Hauptspeicheradresse zu schreiben. Der Bildschirm-Controller bemerkt dies und gibt das Zeichen auf dem Bildschirm aus; dies dauert aber eine ganze Weile, in der die CPU Hunderte oder Tausende von Befehlen ausführen kann bzw. muss, wenn sie sich nicht selbst anhalten kann.

Wir haben nun die ganz typische Situation, dass das Betriebssystem auf die Beendigung einer E/A-Operation warten muss. Der genaue Zeitpunkt, wann die E/A-Operation beendet sein wird, ist nicht vorab bekannt. Ein denkbarer *synchroner* Weg ist, die CPU in einer Warteschleife *Däumchen drehen* zu lassen und immer wieder ein geeignetes Statusregister abzufragen (*polling*). Dies führt jedoch zu sehr vielen Speicherzugriffen und belastet die internen Busse; manche Prozessoren haben deshalb einen Maschinenbefehl, durch den sie sich stillegen können. Man kann sich aber auch eine sinnvollere Beschäftigung für die CPU vorstellen, z. B. den Anwendungsprozess, der die Ein-/Ausgabe verursacht hat, oder bei einem System mit parallelen Prozessen irgendeinen anderen ausführbereiten Prozess auszuführen.

Es muss also einen *asynchronen* Weg geben, durch den das E/A-Gerät der CPU mitteilt, dass der E/A-Auftrag erfolgreich oder nicht erfolgreich ausgeführt wurde.

Hierzu, aber auch zu anderen Zwecken, dienen **Unterbrechungen** (**interrupts**).

Der Unterbrechungsmechanismus ist je nach Prozessor und Rechnerarchitektur sehr verschieden. Oft ist ein **Unterbrechungsregister** PSW (Program Status Word) vorhanden; jedes Bit dieses Registers zeigt eine Unterbrechung an. Geräte bzw. Geräte-Controller können diese Bits setzen. Der Prozessor prüft nun zwischen (oder sogar während) der Abarbeitung von Maschinenbefehlen, ob ein Bit des Unterbrechungsregisters gesetzt ist. Falls ja, wird die normale Programmausführung unterbrochen: statt dessen wird eine zu der Unterbrechung gehörige **Unterbrechungsroutine** (**interrupt handler**) ausgeführt. Dies funktioniert am einfachsten mit einem Array von Adressen von Unterbrechungsroutinen, dem sogenannten **Unterbrechungsvektor** (**interrupt vector**), der an einer festen Adresse im Hauptspeicher steht: die Nummer des Bits im Unterbrechungsregister wird als Index im Unterbrechungsvektor verwendet und das Unterprogramm mit dieser Adresse wird aufgerufen, siehe Abbildung 1.5. Das Setzen eines Bits im Unterbrechungsregister bewirkt also einen asynchronen Unterprogrammaufruf.

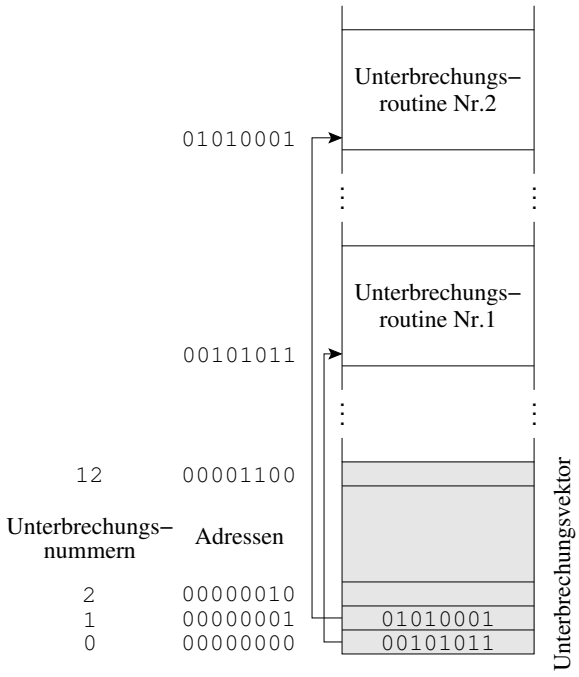


Abbildung 1.5: Jeder Eintrag im Unterbrechungsvektor enthält die Adresse einer Unterbrechungsroutine.

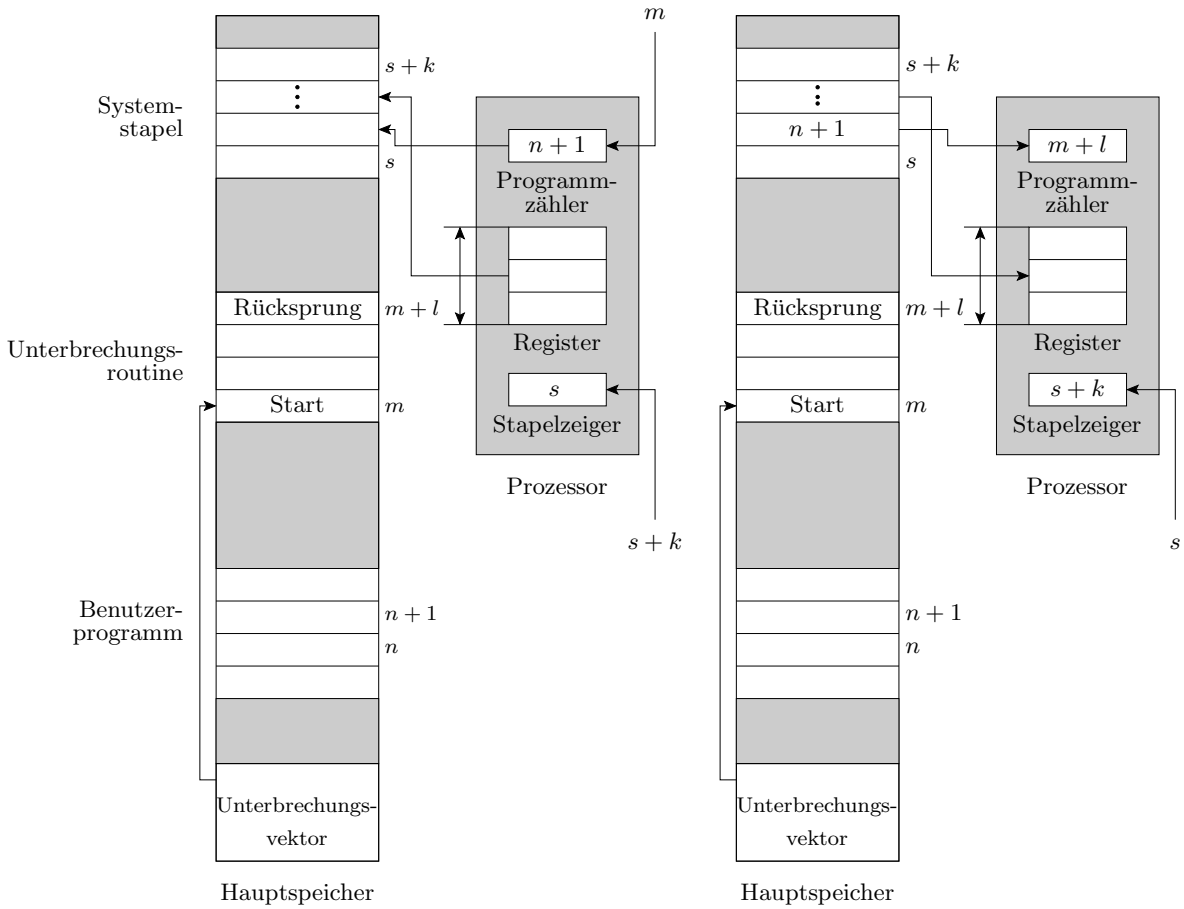
Wie bei jedem Unterprogrammaufruf müssen die bisherigen Inhalte von Registern sowie andere Teile des Programmzustandes, soweit sie durch das Unterprogramm verändert werden, zunächst gerettet werden, typischerweise auf einem Systemstapel (stack). Dies kann automatisch im Rahmen der Unterbrechungserkennung oder explizit durch die Unterbrechungsroutine erfolgen. Insgesamt läuft eine Unterbrechung, die durch ein E/A-Gerät ausgelöst ist, also in folgenden Schritten ab:

1. Der Controller des E/A-Geräts setzt ein Bit z. B. im Unterbrechungsregister.
2. Die Ausführung des laufenden Programms wird unterbrochen, d. h. die CPU beendet die Ausführung des aktuellen Befehls, bevor er auf die Unterbrechung

reagiert.

3. Die CPU analysiert die aufgetretene Unterbrechung.
4. Der Zustand des unterbrochenen Programms wird gerettet.
5. Die Unterbrechungsroutine wird ausgeführt.
6. Der Zustand des unterbrochenen Programms wird wiederhergestellt, und es wird fortgesetzt.

Abbildung 1.6 (a) zeigt entsprechend Schritt 4 die Rettung des aktuellen Benutzerprogramms, das nach dem Befehl an Stelle n unterbrochen wird. Die Inhalte



(a) Unterbrechung nach dem Befehl an Stelle n (b) Rückkehr von der Unterbrechung

Abbildung 1.6: Änderungen im Systemstapel und in den Registern bei einer Unterbrechung.

sämtlicher Register und die Adresse des nächsten Befehls $n+1$, insgesamt k Wörter, werden auf dem Systemstapel abgelegt. Nach Beendigung der Unterbrechungsroutine wird der Zustand des unterbrochenen Programms wiederhergestellt (Schritt 6), siehe Abbildung 1.6 (b), und die Ausführung wird fortgesetzt.

Tatsächlich treten noch einige weitere Komplikationen auf, denn während der Abarbeitung einer Unterbrechungsroutine können erneut Unterbrechungen eintreten, außerdem können mehrere Unterbrechungen zugleich eintreten. Hierzu kann

man weniger wichtige Unterbrechungen *maskieren* und damit sozusagen abschalten; in Schritt 3 oben können ebenfalls schon Prioritäten berücksichtigt werden. Diese Details sind aber für uns nicht mehr wesentlich.

Wir haben Unterbrechungen oben zunächst durch asynchrone E/A-Vorgänge motiviert. Tatsächlich ist der Mechanismus aber auch noch auf andere Probleme anwendbar:

- Realisierung von Systemaufrufen: hierauf gehen wir im Abschnitt 1.2.5 ausführlicher ein.
- Realisierung von Zeitscheiben: In allen Systemen, in denen mehrere Programme gleichzeitig ausgeführt werden, muss ein Prozess, dem die CPU zugeteilt ist, nach einer gewissen Zeit wieder unterbrochen werden, um z. B. zu prüfen, ob der Prozess die ihm zustehende Rechenzeit ausgeschöpft hat oder ob jetzt ein anderer Prozess ausgeführt werden soll, auf Details solcher Prozesswechsel gehen wir in Kurseinheit 3 näher ein. Hierzu benötigt man einen **Zeitgeber** (**timer**), der mit einem Zeitwert geladen werden kann, dann rückwärts läuft und, wenn 0 erreicht wird, eine Unterbrechung erzeugt, das Prinzip wurde schon vor langer Zeit in Eieruhren realisiert. Das Zeitintervall nennt man eine **Zeitscheibe**, die Unterbrechung am Ende auch **Zeitscheibenablauf**.
- Behandlung von Programmfehlern: bei einer Division durch 0, Überlauf, ungültigem Operationscode, unzulässiger Adresse oder sonstigen Laufzeitfehlern eines Programms kann die CPU selbst eine Unterbrechung (**Trap**) auslösen. Das Benutzerprogramm wird nach einer solchen Unterbrechung normalerweise nicht fortgesetzt, sondern abgebrochen; das Betriebssystem erzeugt einen Speicherabzug (Dump), den man für die spätere Fehlersuche verwenden kann, und übergibt die Kontrolle wieder an den Kommandointerpreter. Die Laufzeitsysteme von manchen Programmiersprachen erlauben es aber auch, solche Fehler abzufangen: hierzu teilt das Laufzeitsystem dem Betriebssystem mit, dass es z. B. bei einer Division durch 0 an eine bestimmte Adresse im Laufzeitsystem springen soll.

Der Unterbrechungsmechanismus ist in der Tat absolut zentral für die Funktion (und das Verständnis) von Betriebssystemen, deshalb fassen wir die wichtigsten Eigenschaften noch einmal zusammen:

- Der Betriebssystemkern wird nur durch Unterbrechungen aktiviert.
- Die Unterbrechungen können durch Hardware, z. B. asynchrone E/A-Operationen, oder Software z. B. Programmfehler oder Systemaufrufe ausgelöst werden, diese Unterbrechung ist synchron.
- Die normale Programmausführung wird bei einer Unterbrechung unterbrochen und ggf. nach Beendigung der Unterbrechungsroutine wieder fortgesetzt, ohne dass das unterbrochene Programm hiervon etwas bemerken kann.
- Aus Sicht eines laufenden Programms können jederzeit asynchrone Unterbrechungen eintreten und die Ausführung des Programms für eine unbestimmte Zeit anhalten.

1.2.4 Speicherschutz

Im Beispielsystem CP/M können im Benutzerbereich laufende Programme auf beliebige Adressen zugreifen und z. B. absichtlich oder versehentlich das Betriebssystem verändern oder gar funktionsuntüchtig machen. Dies entspricht nicht gerade der Erwartung, dass das Betriebssystem Fehlverhalten der Programme abfangen soll. Das mindeste, was hier erforderlich ist, sind Schutzmaßnahmen, die schreibende (oder sogar lesende) Zugriffe des Benutzerprogramms auf den im Hauptspeicher befindlichen Betriebssystemkern verhindern. Dies ist nur möglich, wenn die Hardware **Speicherschutzmechanismen** anbietet. Derartige Mechanismen werden in Kurseinheit 4 ausführlicher vorgestellt werden.

Ein Beispiel für einen solchen Mechanismus ist ein **Grenzregister**: es enthält die Adresse, an der der Benutzerbereich im Hauptspeicher beginnt. Wir nehmen ab jetzt an, dass das Betriebssystem den Bereich ab Adresse 0 belegt, das Benutzerprogramm also die *hohen* Adressen. Greift nun das Benutzerprogramm auf eine Adresse zu, die kleiner als der Wert im Grenzregister ist, wird eine Unterbrechung ausgelöst, und die Kontrolle wird vom Betriebssystem übernommen.

1.2.5 Der Supervisor Call

Unabhängig davon, wie nun der Zugriff auf Adressen, an denen der Betriebssystemkern geladen ist, im Detail verhindert wird, entsteht hieraus ein Problem: eine Applikation kann nicht mehr einfach die zu den Systemaufrufen gehörenden Routinen im Betriebssystemkern aufrufen, denn hierzu müsste sie einen Zugriff (Unterprogramm-Sprung) an eine Adresse im Betriebssystemkern ausführen, was wir durch die Einführung von Schutzmechanismen gerade verhindert haben. Dieses Problem lässt sich mit Hilfe des Unterbrechungsmechanismus lösen: man führt einen zusätzlichen Maschinenbefehl ein, den **supervisor call** (**SVC**), der auch **Trap** genannt wird. Ein Trap ist eine Software-Unterbrechung, die entweder durch eine Programmfehler (Division durch 0 oder Zugriff auf die unzulässigen Adressen) oder durch einen Systemaufruf verursacht. Der Befehl SVC löst eine Unterbrechung aus, die vom Betriebssystem so interpretiert wird, dass das auslösende Programm einen Systemaufruf durchführen will. Die Systemaufrufe bilden also die Schnittstelle zwischen dem Betriebssystem und den Benutzerprogrammen.

Der Mechanismus, wie ein Systemaufruf gestartet wird, ist meistens sehr stark von der Hardware abhängig und muss oft in Assembler geschrieben werden. Es gibt auch eine Bibliotheksfunktion, die einen Systemaufruf aus der Programmiersprache C und anderen Sprachen zulässt. Unter UNIX existiert eine Eins-zu-Eins-Relation zwischen den Systemaufrufen (z. B. *read*) und den entsprechenden Bibliotheksfunktionen (z. B. *read*), die die Systemaufrufe dann durchführen. Das heißt, dass für jeden Systemaufruf ziemlich genau eine Bibliotheksfunktion existiert, die von den Benutzerprogrammen meistens genutzt wird. In POSIX unter Windows hingegen gibt es eine Menge von ein paar tausenden Bibliotheksfunktionen, die **Win32 API** (**Application Program Interface**) genannt wird. Die Programmierer sollen sie benutzen, wenn sie Systemaufrufe nutzen wollen. Von den Win32-API-Funktionen sind viele Systemaufrufe, aber viele arbeiten vollständig im Benutzerbereich. Man kann leider unmöglich entscheiden, was ein echter Systemaufruf ist und was im Benutzerbereich bearbeitet werden kann. Es kann passieren, dass eine Funktion in einer

Windows-Version Systemaufruf ist und in einer anderen Version nicht.

Es gibt Dutzende von Systemaufrufen, die jeweils verschiedene Parameterlisten haben können, andererseits nur einen einzigen Maschinenbefehl SVC. Man geht deshalb z. B. folgendermaßen vor:

Jeder Systemaufruf bekommt eine *eindeutige Nummer*. Diese Nummer kann z. B. in einem Register oder in einem Parameterfeld des Maschinenbefehls SVC gespeichert werden. Nach der durch den SVC ausgelösten Unterbrechung findet das Betriebssystem diese Nummer an der festgelegten Stelle und benutzt sie wieder als Index für eine Tabelle, die die Adressen der Unterprogramme enthält, die zu den Systemaufrufen gehören, vergleiche Abschnitt 1.2.3.

Für die Parameter des Systemaufrufs kann man analog weitere Register verwenden, allerdings reichen die Register bei größeren Parametern z. B. 2000 Byte, die einen kompletten auszugebenen Bildschirminhalt beinhalten, nicht aus. Deshalb legt man im Hauptspeicher zunächst eine Datenstruktur, meist einen einfachen Record, den sog. **Parameterblock**, an, der alle erforderlichen Parameterwerte enthält. Die Adresse dieses Hauptspeicherbereichs wird in ein Register geladen und auf diese Weise dem Betriebssystemkern übermittelt. Auf der Sprachebene Assembler sieht dies typischerweise so aus, dass für jeden Systemaufruf ein Makro vorhanden ist, welches die gleichen Parameter wie der Systemaufruf hat. Dieses Makro generiert außer dem SVC i. W. Assembler-Anweisungen, die einen Speicherbereich passender Größe anlegen, der ggf. schon entsprechende Konstanten enthält, und das Parameterregister mit der Adresse dieses Blocks laden.

1.2.6 System- und Benutzermodus

Abschließend kommen wir noch einmal auf das Thema Schutz zurück. Durch die Speicherschutzmechanismen werden zunächst einmal das Betriebssystem selbst und zugehörige Datenstrukturen (z. B. E/A-Puffer) vor Zugriffen geschützt. Allerdings werden auf diese Weise keine *speziellen* Maschinenbefehle geschützt, z. B. das Setzen des Grenzregisters, das Setzen von Bits im Unterbrechungsregister, der HALT-Befehl, der den Rechner anhält, und vor allem E/A-Maschinenbefehle z. B. Lesen und Schreiben von Registern, die Schnittstellen repräsentieren, Befehle, die Treiber aktivieren oder steuern und ähnliches.

Wenn Applikationen diese Maschinenbefehle ausführen dürften, könnten sie z. B. absichtlich oder unabsichtlich den Speicherschutz unterlaufen oder das Funktionieren des Betriebssystemkerns empfindlich stören. Applikationen werden daher in den meisten Systemen daran gehindert, solche Maschinenbefehle auszuführen. Die CPU hat hierfür zwei oder mehr Ausführungsmodi. Im **Systemmodus** (auch **Monitormodus** oder **privilegierter Modus** genannt) sind alle Maschinenbefehle erlaubt. Im Systemmodus werden außerdem alle Speicherschutzmechanismen abgeschaltet. Im **Benutzermodus** sind nur die *normalen*, nicht sicherheitskritischen Befehle erlaubt; die restlichen Befehle nennt man auch **privilegierte** Befehle. Bei einem Versuch, im Benutzermodus einen privilegierten Befehl auszuführen, erzeugt die CPU eine Unterbrechung.

Eine Unterbrechung führt nun immer zur Umschaltung in den Systemmodus. Die Unterbrechungsroutine kann zum Schluss einen (selbstverständlich privilegierten) Befehl ausführen, durch den der Ausführungsmodus wieder umgeschaltet wird.

Durch die Speicherschutzmechanismen und die Ausführungsmodi erreicht man

einen weitgehenden Schutz des Betriebssystems und der Hardware-Schnittstellen vor Zugriffen durch Applikationen. Applikationen können dann nur noch durch Systemaufrufe Ein- oder Ausgaben veranlassen oder sonstige Dienste des Betriebssystems in Anspruch nehmen. Dies ermöglicht eine strikte Kontrolle über den Zugang zu allen kritischen Ressourcen.

1.2.7 Mehrprogrammbetrieb

Wir waren bei unseren bisherigen Betrachtungen von einem sehr einfachen System ausgegangen, in dem nur eine einzige Applikation geladen ist und ausgeführt wird (*uni-programming*). Wie schon in Abschnitt 1.1 erwähnt, ist aber eine der wesentlichsten Anforderungen an einen Rechner, mehrere (Benutzer-) Programme gleichzeitig auszuführen. Auf Basis der Mechanismen, die wir bis jetzt kennengelernt haben, ist es nur noch ein kleiner Schritt, sich vorzustellen, wie dies im Prinzip funktionieren könnte: Statt nur eines Programms lädt man mehrere Programme in den Benutzerbereich des Hauptspeichers und teilt den Programmen abwechselnd Zeitscheiben zu. Dies nennt man **Mehrprogrammbetrieb** (**Multiprogramming**, **Multitasking**). Zu den wesentlichen Problemen, die jetzt zu lösen sind, gehört die Aufteilung des Hauptspeichers und die Zuteilung der CPU an die Programme; diese Probleme werden in den Kurseinheiten 3 und 4 ausführlich behandelt, für jetzt soll die Auskunft genügen, dass die Probleme lösbar sind.

Parallele Prozesse. Jedes der ausgeführten Programme bildet einen **Prozess**. Die Prozesse laufen **parallel** insofern, als sie zeitlich überlappen. Wir nehmen an, dass nur eine CPU real vorhanden ist; daher kann natürlich immer nur ein einziger Prozess zu einem Zeitpunkt ablaufen, die Prozesse laufen also nicht *echt* parallel, denn hierzu müsste für jeden Prozess eine reale CPU zur Verfügung stehen. Dennoch hat jeder Prozess, da er die Wartezeiten, in denen die CPU andere Prozesse bedient, nicht bemerken kann, den Eindruck, eine eigene, private CPU zur Verfügung zu haben. Mit anderen Worten simuliert das Betriebssystem einen **virtuellen Prozessor** für jeden Prozess, also für jedes ausgeführte Programm. Dies kann als eine der wichtigsten Leistungen eines Betriebssystems überhaupt angesehen werden. Für den Mehrprogrammbetrieb gibt es einige völlig verschiedene Motivationen, auf die wir später näher eingehen werden, siehe Abschnitt 1.3.

Bemerkenswert an den parallelen Prozessen ist, dass der relative Ausführungsfortschritt der einzelnen Prozesse zueinander unvorhersehbar ist: jeder Prozess kann im Prinzip jederzeit unterbrochen werden, wenn z. B. seine Zeitscheibe abläuft oder wenn eine E/A-Operation beendet wird. Wegen der Zufälligkeit der Ausführungsdauer von E/A-Operationen oder auch wegen Ungenauigkeiten des Zeitgebers für die Zeitscheiben ist die Verzahnung von unsynchronisierten parallelen Prozessen nicht reproduzierbar. Dies gilt auch dann, wenn tatsächlich mehrere reale Prozessoren zur Verfügung stehen. Dies führt zu außerordentlich harten Problemen, wenn mehrere parallele Prozesse gemeinsame Variablen oder andere Ressourcen koordiniert benutzen sollen. Lösungen für derartige Probleme werden in Kurseinheit 5 vorgestellt werden.

Kommunizierende parallele Prozesse. Betriebssysteme kommunizieren parallel mit vielen Schnittstellen, Geräten oder indirekt mit Instanzen in ihrer *Umwelt*;

Betriebssysteme sind daher inhärent parallele Systeme. Es hat sich herausgestellt, dass parallele Systeme meist sehr viel klarer strukturiert werden können, wenn man sie als System kommunizierender paralleler Prozesse auffasst. Für die Kommunikation zwischen Prozessen gibt es verschiedene Mechanismen, z. B. je eine Operation zum Senden einer Nachricht und zum Warten auf die Ankunft einer Nachricht; Details sind im Moment nicht wichtig. Wesentlich ist hier, dass nicht nur Applikationen parallel ausgeführt werden können, sondern dass man das Betriebssystem selbst auch als System paralleler Prozesse realisieren kann.

Erhöhung der Auslastung von Betriebsmitteln. Der Geschwindkeitsunterschied zwischen der CPU und den E/A-Operationen bei langsamen E/A-Geräten wie Tastaturen, Bildschirmen oder Druckern ist extrem groß. Es ist sehr unwahrscheinlich, dass beispielsweise ein Benutzer, der mit einem Texteditor arbeitet, mehr als etwa 5 Anschläge pro Sekunde (im Durchschnitt!) schafft. Wenn wir annehmen, dass jeder Tastendruck sofort von der CPU verarbeitet wird, dann wäre also alle 200 Millisekunden ein Tastendruck zu verarbeiten (insb. auf den Bildschirm auszugeben). Eine CPU, die heute typischerweise zwischen 1 und 100 Millionen Maschinenbefehle pro Sekunde ausführen kann, wird hierzu aber nur wenige Millisekunden benötigen, sagen wir 10. Dann würde die CPU von diesem Prozess nur zu 5 % ausgelastet und würde 95 % der Zeit damit verbringen, auf Tastatureingaben bzw. Bildschirmausgaben zu warten.

Während dieser Wartezeiten können natürlich andere Prozesse weitergeführt werden. Wenn man den Aufwand für die Umschaltung zwischen den Prozessen und sonstige Engpässe im System einmal vernachlässigt, dann könnte die CPU im Idealfall 20 Editoren bedienen, ohne dass die einzelnen Prozesse aus Sicht ihrer Benutzer langsamer laufen würden. Man kann den Idealfall zwar nicht realisieren, es ist aber in der Praxis immerhin erreichbar, dass bei einer mittleren Auslastung der CPU (und anderer Betriebsmittel) jeder einzelne der parallelen Prozesse nicht wesentlich langsamer ausgeführt wird, als wenn ihm der ganze Rechner alleine zur Verfügung stehen würde.

Im obigen Beispiel traten sehr langsame E/A-Geräte auf. Andere E/A-Geräte, z. B. Magnetplatten oder Magnetbänder, haben wesentlich höhere Übertragungsraten, wenn z. B. gerade ein Sektor einer Magnetplatte gelesen wird. Würde jedes gelesene Zeichen einzeln von der CPU verarbeitet, dann wäre selbst eine relativ schnelle CPU hiermit voll beschäftigt, also nicht für die Ausführung von Applikationen verfügbar. Deshalb wird bei schnellen E/A-Geräten so verfahren, dass diese selbständig ganze Blöcke in den Hauptspeicher schreiben, so dass nur noch nach Beendigung der kompletten Übertragung eine Unterbrechung erzeugt werden muss und die CPU zwischendurch nicht mehr belastet wird. Dies bedingt allerdings, dass die Geräte (genauer gesagt die Controller der Geräte) direkten Zugriff zum Hauptspeicher haben; deshalb heißen solche Geräte auch **DMA**- (**direct memory access**) **Geräte** (bzw. -Controller). Da ein großer Teil der Ein-/Ausgaben von schnellen Geräten verursacht wird, ist DMA tatsächlich eine wesentliche Voraussetzung dafür, mehrere Prozesse mit einer einzigen CPU ohne wesentliche Verschlechterung der Antwortzeiten bedienen zu können.

1.2.8 Laden des Betriebssystems

Wir waren bisher immer davon ausgegangen, dass sich der Betriebssystemkern schon im Hauptspeicher befindet; der dazu gehörige Lader lädt dann bei Bedarf weitere Programme in den Hauptspeicher. Wie aber ist der Lader in den Speicher gekommen?

Um den Lader incl. der erforderlichen Treiber in den Hauptspeicher zu laden, braucht man offenbar einen weiteren Lader, einen sogenannten **Urlader**. Leider verschiebt sich das Problem zunächst nur auf den Urlader. Der Unterschied zwischen einem normalen Lader und einem Urlader ist aber, dass der Urlader sehr einfach sein kann: er muss nur ein ganz bestimmtes Programm von einer ganz bestimmten Adresse z. B. den ersten 10 Sektoren einer Platte an eine ganz bestimmte Adresse im Hauptspeicher kopieren; deshalb kann der Urlader deutlich kleiner als der Lader in Bytes gemessen sein. Gegebenenfalls kann man auch einen Ururlader einsetzen; irgendwann muss aber diese Kette beendet werden.

In älteren Systemen wurde das Problem so gelöst, dass zunächst der Urlader von Lochkarten oder einem Band in den Hauptspeicher kopiert und dann gestartet wurde. Der hierzu erforderliche *Ururlader* war ein einziger Maschinenbefehl, den man manuell laden und ausführen konnte oder der durch Drücken einer reset-Taste ausgeführt wurde.

In neueren Systemen verwendet man ein **ROM** (**read only memory**); diese Halbleiterspeicher sind nicht flüchtig und nicht beschreibbar. Im ROM sind außer dem Urlader mittlerweile oft größere Teile des Betriebssystemkerns enthalten, wodurch die Ladezeit des Betriebssystems nach dem Einschalten eines Rechners verkürzt wird. Theoretisch könnte man den kompletten Betriebssystemkern im ROM speichern; dies hätte aber den Nachteil, dass am Kern nichts mehr verändert werden kann (außer durch Austausch des ROMs).

Für die (Um-) Konfigurierung von Betriebssystemen ist es sehr vorteilhaft, wenn man wahlweise verschiedene Betriebssysteme laden kann. Hierfür muss es möglich sein, dem Urlader das zu ladende Betriebssystem anzugeben, z. B. in Form eines Geräteidentifizierers (die Betriebssysteme müssen dann auf verschiedenen Geräten gespeichert sein). Hierzu bieten manche Systeme einen einfachen interaktiven Monitor an, der im ROM gehalten wird und der entsprechende Kommandos ausführen kann. Letztlich sind diese Details aber sehr abhängig vom Rechnertyp, so dass wir hier nicht näher darauf eingehen wollen.

1.3 Betriebsarten

Optimierungsziele beim Mehrprogrammbetrieb. Wir haben im vorigen Abschnitt einen ersten Eindruck davon gewonnen, wie mehrere Programme parallel von einer einzigen CPU ausgeführt werden können. Wir wenden uns nun der Frage zu, warum dies denn sonderlich wünschenswert sein sollte und was hierbei vielleicht noch zu beachten sein mag. In technischer Hinsicht bietet der Mehrprogrammbetrieb den Vorteil, u. U. die Auslastung von Betriebsmitteln zu verbessern oder ein paralleles System als System kommunizierender Prozesse realisieren zu können. Wir werden aber sehen, dass der Mehrprogrammbetrieb nicht einseitig unter technischen

Aspekten betrachtet werden darf, sondern nur in einem weiteren Kontext sinnvoll behandelt werden kann, nämlich im Rahmen der Frage, wie ein Rechner ingesamt betrieben werden soll. Diese Frage kann nur *von außen* beantwortet werden, wir kehren jetzt also zu dem Standpunkt zurück, den wir schon in Abschnitt 1.1 hatten, also der Sicht der Anwender bzw. Betreiber auf Rechner und Betriebssystem.

Wir können folgende Fälle bzgl. der Zahl der Anwender und der parallel ausgeführten Programme auf einem Rechner unterscheiden:

1. Es gibt nur einen Anwender auf dem Rechner. Dieser will
   (a) immer nur eine Applikation zu einer Zeit ausführen.
   (b) i. A. mehrere Applikationen zugleich ausführen, z. B. einen Text-Editor für Quellprogramme, einen Compiler und ein Modulverwaltungssystem, die auf zusammenhängenden Daten arbeiten.
2. Es gibt mehrere Anwender, die auf gemeinsamen Daten arbeiten z. B. ein Buchungssystem einer Bank benutzen oder in sonstiger Weise kooperieren und die deshalb gemeinsam und gleichzeitig einen Rechner benutzen müssen. Wir nehmen (zu Recht) an, dass die Bank reich ist und einen leistungsfähigen Rechner gekauft hat, der problemlos alle Anwender bedienen kann.
3. Es gibt mehrere Anwender, die am liebsten jeweils einen eigenen Rechner hätten, ihn sich aber nicht leisten können und deshalb einen Rechner gemeinsam benutzen. Wie wir schon in Abschnitt 1.2.7 gesehen haben, kann durch den Mehrprogrammbetrieb die Auslastung der CPU und anderer Ressourcen verbessert werden. Um die Sache auf den Punkt zu bringen, nehmen wir weiter an, dass die Leistung des Rechners so gerade noch ausreicht, die Anforderungen der Anwender zu erfüllen, so dass es darauf ankommt, die vorhandenen Ressourcen so gut wie möglich auszuschöpfen[6].

Im Fall 1 (a) kommt man mit einem sehr einfachen Betriebssystem aus, wie wir es in Abschnitt 1.2 beschrieben haben. Die Fälle 1 (b) und 2 sind aus Sicht des Betriebssystems nicht sonderlich verschieden: entscheidend ist, dass mehrere Prozesse gleichzeitig laufen. Ob diese Prozesse mit einem einzigen oder mehreren Benutzern kommunizieren, ist insgesamt eher nebensächlich, es hat natürlich einige Auswirkungen auf die E/A-Verwaltung. Wir fassen deshalb den Fall 1 (b) als Sonderfall von Fall 2 auf. Die beiden Fälle zeigen allerdings, dass eine Unterscheidung von Betriebssystemen nach der Anzahl der gleichzeitigen *Benutzer* nicht sinnvoll ist; wesentlich ist die Anzahl der gleichzeitig ausgeführten *Programme*.

Die Fälle 2 und 3 zeigen, dass es zwei völlig unterschiedliche Motivationen für den Mehrprogrammbetrieb gibt: Kooperation von Benutzern und Geldmangel. Es

[6] Tatsächlich war dies der Normalfall! Am Anfang der 1970er Jahre kostete ein Rechner, der weniger leistete als ein PC der 100-MHZ-Klasse, etwa 5.000.000 DM, war in Dutzenden solider Stahlgestelle untergebracht, die eine ganze Etage füllten, brauchte etwa 100 kW Strom und ggf. sogar Kühlwasser und musste von besonders geschultem und gut bezahltem Personal bedient werden. Eine Sekunde (!) Rechenzeit kostete damals in der Größenordnung von 1 DM. Es gibt auch heute noch (Super-) Rechner, die Kosten in dieser Größenordnung verursachen. Allerdings sind deren Rechenleistungen nur für extrem hohe Anforderungen erforderlich, z. B. bei der Wettervorhersage, Strömungssimulationen und ähnlichem. Für die breite Masse der *Allerweltsaufgaben* sind heute weitaus billigere Rechner völlig ausreichend, und man kann es sich leisten, die Rechner immer schlechter auszulasten und dafür den Benutzungskomfort zu erhöhen.

können natürlich auch beide Fälle gleichzeitig eintreten. Technisch gesehen unterscheiden sich diese beiden Fälle vor allem durch den Grad, in dem die Anwender den Rechner auslasten (10 bzw. 90 %). Dies hat ganz gravierende Auswirkungen auf die Optimierungsziele bei der Realisierung paralleler Prozesse und auf die allgemeine Organisation des Rechnerbetriebs: Im ersten Fall wird man den Benutzungskomfort, insbesondere die Bedienungsgeschwindigkeit, erhöhen, im zweiten Fall die Ausnutzung aller Ressourcen des Rechners optimieren wollen. Leider sind beide Ziele gegensätzlich, d. h. im ersten Fall muss man eine schlechtere Ausnutzung der Ressourcen in Kauf nehmen, im zweiten Fall schlechtere Antwortzeiten und Benutzungskomfort. Die unterschiedlichen Optimierungsziele führen zu zwei völlig verschiedenen Betriebsarten, nämlich Dialogbetrieb und Stapelbetrieb.

Betriebsarten. Wir haben uns bisher wenig um die Details gekümmert, wie ein Benutzer Programme startet und wie die Programme anschließend ablaufen, speziell in welcher Weise sie mit dem Benutzer oder ihrer sonstigen *Umwelt* interagieren. Technisch gesehen handelt es sich hier vor allem um die Fragen,

1. wie Arbeitsaufträge dem System übergeben werden,
2. wie Ein- und Ausgaben bei der Programmausführung verarbeitet werden und
3. mit welcher Priorität Ressourcen zugeteilt werden.

Es gibt in dieser Hinsicht einige völlig verschiedene Formen, sogenannte **Betriebsarten**. Wir stellen die üblichen Betriebsarten vor allem *aus der Sicht der Anwendungen* vor. Die Betriebsarten haben allerdings auch Auswirkungen auf die erforderlichen Strukturen in Betriebssystemen; hierauf werden wir später eingehen.

Interaktiver Betrieb. Für einen interaktiven (oder on-line oder Dialog-) Betrieb benötigt man geeignete Ein- und Ausgabegeräte, z. B. eine Tastatur und einen Bildschirm, durch die das Betriebssystem bzw. ein laufendes Programm und ein Benutzer kommunizieren können.

Ein *Rechenauftrag* an den Rechner besteht hier aus einer *Sitzung*, in deren Verlauf der Benutzer die auszuführenden Programme in einer Kommandosprache aufruft. Bei interaktiven Mehrbenutzer-Systemen (sogenannte **time-sharing** Systeme) muss sich der Benutzer anfangs identifizieren und am Ende die Sitzung explizit beenden, d. h. Beginn und Ende der Sitzung sind deutlich erkennbar. Bei Einbenutzer-Systemen entfällt die An- bzw. Abmeldung.

Vom Benutzer aufgerufene Programme werden immer *sofort* gestartet. Die Ausgaben des Programms und die Eingaben des Benutzers werden immer sofort weitergeleitet. Bei manchen Systemen kann der Benutzer erst nach einer abgeschlossenen Ausgabe neu eingeben, bei anderen Systemen kann der Benutzer jederzeit also sozusagen *blind* eingeben, weil er die Ausgaben auf vorherige Kommandos noch nicht kennt; normalerweise wartet der Benutzer aber nach einer Eingabe immer auf die zugehörige Ausgabe. Typische Beispiele interaktiv ausgeführter Programme sind fast alle heute verfügbaren Editoren für Texte, Bilder oder andere Arten von Dokumenten, darüber hinaus Spiele, Tabellenkalkulationsprogramme, manche Datenbankabfragesysteme usw. Manche Programme können nur interaktiv sinnvoll ausgeführt werden, weil sie ihre Ausgaben auf ein interaktives Medium wie einen Bildschirm

beziehen; andere Programme kann man sowohl interaktiv wie im Stapel ausführen (s. u.).

Das Betriebssystem muss dafür sorgen, dass die Antwortzeit, also die reale Zeit zwischen einer Eingabe und der darauf folgenden Ausgabe, möglichst niedrig ist, denn so lange muss der Benutzer warten. Ein wesentliches Merkmal interaktiv ausgeführter Programme, z. B. eines Editors, ist jedoch, dass die Rate der vom Benutzer eingegebenen bzw. vom Editor ausgegebenen Daten extrem klein ist im Vergleich zur Verarbeitungskapazität einer CPU. Insgesamt werden die Ressourcen eines Rechners im Dialogbetrieb von einem Prozess allein relativ schlecht ausgenutzt.

Stapelbetrieb. Im Stapelbetrieb (auch: **off-line** Betrieb) erstellt ein Benutzer zunächst eine Beschreibung des Auftrags, den man auch **Stapeljob** (**batch job**) nennt. In einem Stapeljob wird normalerweise nicht nur ein einziges Programm ausgeführt, sondern mehrere, die z. B. schrittweise auf den vorherigen Ergebnissen aufbauen. Die Abfolge dieser Programme wird normalerweise durch eine Kommandoprozedur beschrieben. Außerdem müssen darin sämtliche benötigten Eingaben enthalten sein, also schon vor dem Start der Programme vorliegen. Zur Beschreibung des Stapeljobs gehören auch gewisse organisatorische Angaben, z. B. wieviel CPU-Zeit der Job maximal verbrauchen darf.

Die Auftragsbeschreibung wird als nächstes als Inhalt eines Datenträgers (Lochkartenstapel, Diskette, Band o. ä.) der Bedienmannschaft übergeben und von dieser in den Rechner übertragen oder vom Benutzer interaktiv dem System als Inhalt einer Datei übergeben. Nachdem der Auftrag dem System übergeben wurde, kann es längere Zeit dauern, bis mit der Bearbeitung des Auftrags begonnen wird.

Der Auftraggeber hat i. A. keinen Kontakt (mehr) zum Rechner, wenn dieser den Stapeljob schließlich ausführt. Es ist also keine direkte Kommunikation zwischen den ausgeführten Programmen und dem Benutzer, der den Start der Programme veranlasst hat, möglich. Die im Rahmen des Stapeljobs ausgeführten Programme verarbeiten ihre Eingabedaten und erzeugen dabei in der Regel Ausgabedaten, die z. B. ausgedruckt oder in einer Datei gespeichert werden, oder verändern den Inhalt von vorhandenen Dateien oder Datenbanken.

Die Bezeichnung *Stapel* rührt daher, dass typischerweise ein Stapel solcher Aufträge vorliegt, die nach und nach ausgeführt werden. Stapelverarbeitung tritt in fast jedem Büro auf: Links auf dem Schreibtisch liegt ein Stapel unerledigter Fälle, in der Mitte wird gerade ein Fall oder vielleicht mehrere parallel bearbeitet, rechts liegen die bearbeiteten Fälle.

Beispiele off-line ausführbarer Programme sind Compiler, manche Textformatierer, große Datenbankauswertungen, viele mathematische Programme usw. Für die Stapelausführung eignen sich vor allem Programme, die umfangreiche Datenmengen verarbeiten oder sehr rechenintensiv sind, die aber nicht zeitkritisch sind, d. h. deren Ausführung eine Zeitlang ohne Probleme unterbrochen werden kann.

Statt off-line kann man ein Programm natürlich auch interaktiv ausführen, auch wenn es nichts auf den Bildschirm ausgibt; das hat den Vorteil, dass es sofort gestartet wird und man die Ergebnisse, wenn sie zusätzlich auf den Bildschirm ausgegeben werden, schon zwischendurch überprüfen und ggf. die Ausführung des Jobs abbrechen kann. Die Stapelausführung ist aus Sicht eines Benutzers also ziemlich unattraktiv. Der direkte Nutzen des Stapelbetriebs liegt in den besseren Möglichkeiten für das Betriebssystem, die Ressourcen eines Rechners möglichst hoch auszulasten:

Wegen der größeren Spielräume bei den Antwortzeiten kann das Betriebssystem die Reihenfolge, nach der die wartenden Aufträge vom Stapel genommen werden, relativ frei bestimmen und ggf. durch weitere, feinere Steuerungsmaßnahmen die von den laufenden Aufträgen ausgehende Last für CPU, Ein-/Ausgabeberäte usw. steuern. Der Stapelbetrieb ist also allein durch den oben erwähnten Geldmangel motiviert.

Festzuhalten bleibt, dass Stapelausführung eigentlich nur in Betriebssystemen sinnvoll ist, die die gleichzeitige Ausführung mehrerer Programme erlauben. Außerdem werden Stapeljobs und interaktiv ausgeführte Programme nach gänzlich verschiedenen Kriterien gesteuert, was oft unterschiedliche interne Strukturen in Betriebssystemen erfordern kann.

Hintergrundausführung. Eine Zwischenform zwischen Dialog- und Stapelbetrieb stellt die Hintergrundausführung von Programmen dar. Hier wird ein Programm oder eine Kommandoprozedur interaktiv gestartet und dann *im Hintergrund* ausgeführt, während die interaktiven Programme *im Vordergrund* weiterlaufen.

In mancher Hinsicht kann man einen Hintergrundjob wie einen Stapeljob behandeln, in vielen Fällen liefe dies aber den Absichten der Benutzer zuwider. Wie schon früher erwähnt ist es oft sinnvoll, eine Aufgabe nicht durch einen einzigen Prozess, sondern durch mehrere kommunizierende Prozesse lösen zu lassen. Dies gilt auch dann, wenn ein Benutzer das gesamte System kommunizierender Prozesse interaktiv steuert oder Daten interaktiv eingibt. In diesem Fall mag z. B. nur ein einziger Prozess mit den Ein-/Ausgabegeräten direkt verbunden, also im obigen Sinne interaktiv sein, während alle anderen Prozesse *im Hintergrund* laufen. Viele Fenstersysteme sind als ein solches System kommunizierender Prozesse organisiert; ebenso implementiert man Datenbankmanagementsysteme oft mit Hilfe eines oder mehrerer im Hintergrund laufender *Serverprozesse*.

Der entscheidende Unterschied zwischen derartigen im Hintergrund laufenden Programmen und Stapeljobs liegt also darin, dass Hintergrundprozesse i. W. wie interaktive Prozesse behandelt werden müssen, denn sie stellen ja einen Teil des Systems kommunizierender Prozesse dar. Sie müssen z. B. sofort gestartet werden und können nicht einfach zur Optimierung der Auslastung des Rechners zeitweise stillgelegt werden. Außerdem ist bei Beendigung des Hintergrundprozesses der Dialogprozess, der diesen Hintergrundprozess gestartet hatte, zu benachrichtigen, während man bei einem Stapeljob z. B. ein Protokoll ausdruckt.

Realzeitbetrieb. Bei manchen Anwendungen müssen Ein-/Ausgaben nicht nur *relativ schnell* wie bei der interaktiven Ausführung, sondern innerhalb von harten Zeitgrenzen verarbeitet werden. Beispielsweise möge ein Messwertgeber laufend Eingabedaten mit einer gewissen Rate liefern; Eingabedaten, die nicht rechtzeitig weiterverarbeitet werden, werden durch die folgenden überschrieben und sind damit verloren.

Absolute Zeitgrenzen setzen natürlich voraus, dass der gewählte Algorithmus, das sich daraus ergebende geladene Programm und die Hardware ausreichend schnell und leistungsfähig sind. Die Rolle, die das Betriebssystem in diesem Zusammenhang spielt, liegt vor allem darin, die Ausführung des zeitkritischen Programms nicht zu verzögern, also erforderliche Ressourcen verfügbar zu machen oder nicht selbst Ressourcen (z. B. CPU-Zeit) zu verbrauchen.

Man könnte meinen, Realzeitanwendungen würden nur einfach besonders hohe Priorität bei der Zuteilung der von ihnen benötigten Ressourcen erfordern. Leider ist dies nicht so einfach. Viele Betriebssysteme, die primär für Dialog- oder Stapel-Ausführung entworfen wurden, sind für Realzeitanwendungen überhaupt nicht geeignet, weil die Struktur des Kerns dieser Betriebssysteme auf der Annahme beruht, dass kurz- oder auch längerfristige Unterbrechungen der Ausführung der Anwenderprogramme keinen ernsthaften Schaden anrichten außer einigen Beschwerden bei der Betriebsleitung. Das Betriebssystem räumt sich daher selbst Priorität bei der Abarbeitung von Unterbrechungen ein. Realzeitanwendungen erfordern daher einen besonders konstruierten Betriebssystemkern.

Teilhaberbetrieb. Hierbei handelt es sich um eine Abart des Dialogbetriebs. Der Unterschied liegt darin, dass hier *mehrere* Benutzer über je ein eigenes Terminal *mit einem einzigen Prozess kommunizieren* (also *Teilhaber* an diesem Prozess sind), während wir oben beim Dialogbetrieb stillschweigend vorausgesetzt hatten, dass jeder Benutzer mit einem eigenen Prozess kommuniziert. Beispiele für den Teilhaberbetrieb sind transaktionsorientierte Systeme wie Flugbuchungssysteme, Verkaufskassensysteme, Bankenterminalsysteme usw., typischerweise auf Basis eines Datenbankmanagementsystems. Die Aufgabe der Steuerung und Überwachung paralleler Aktivitäten wird hier teilweise von einem **Transaktionsmonitor** übernommen, liegt also nicht mehr alleine in der Verantwortung des Betriebssystems. Das Besondere aus Sicht des Betriebssystems liegt hier darin, dass ein einzelnes ablaufendes Programm parallel über mehrere Schnittstellen mit mehreren Benutzern kommuniziert und hierbei unterstützt werden muss.

Statt des Teilhaberbetriebs kann man im Prinzip auch ein System kommunizierender Prozesse verwenden, z. B. je einen interaktiven Prozess für jede Schnittstelle und einen Hintergrundprozess mit dem eigentlichen Datenbankmanagementsystem. Diese Lösung verursacht allerdings erheblich mehr Aufwand und ist in Systemen, in denen Prozesswechsel oder Interprozesskommunikation *teuer* sind, nicht angebracht. Bei älteren Rechenanlagen und deren Betriebssystemen war dies der Fall, deshalb findet sich der Teilhaberbetrieb vor allem dort. In moderneren Rechnern sind die Leistungsreserven i. d. R. bei nicht allzu anspruchsvollen Anwendungen so hoch und die Interprozesskommunikation so leistungsfähig, dass eine Lösung als System kommunizierender Prozesse realisierbar ist, wodurch eine spezielle Betriebsart nicht mehr erforderlich ist. Wir werden den Teilhaberbetrieb deshalb i. F. auch nicht weiter behandeln.

Betriebsarten eines Betriebssystems. Es stellt sich nun die Frage, ob ein Betriebssystem nur eine oder mehrere dieser Betriebsarten anbieten sollte, oder ob man sogar die Betriebssysteme nach der angebotenen Betriebsart klassifizieren kann. Tatsächlich gibt es manchmal gute Gründe, nur eine Betriebsart anzubieten:

- Betriebssysteme für PCs erlauben i. d. R. nur den Dialogbetrieb (manchmal zusätzlich einen Hintergrundprozess). Diese Betriebssysteme sind vergleichsweise primitiv und erlauben keine parallele Ausführung von mehreren Programmen; wie wir oben gesehen haben, ist dann auch die Stapelausführung nicht mehr sonderlich sinnvoll.

- Bei Anlagensteuerungen und anderen Realzeitanwendungen wird ohnehin meist ein relativ kleiner Rechner exklusiv für eine Aufgabe eingesetzt, so dass schon von daher nur eine Ausführungsart benötigt wird.

Die Beschränkung auf eine einzige Ausführungsart ist andererseits nicht zwingend: die meisten heute vorhandenen Betriebssysteme, speziell solche für *Allzweckrechner*, bieten gleichzeitig Dialogbetrieb und Stapelbetrieb oder Hintergrundausführung an; diese Betriebsarten sind sozusagen Standard. Die interaktiven Prozesse werden dann typischerweise mit höherer Priorität bedient als die Stapeljobs bzw. Hintergrundprozesse.

1.4 Zusammenfassung

Ziel der ersten Kurseinheit war, Ihnen eine erste Vorstellung von dem Aufgabenbereich und der Funktionsweise eines Betriebssystems zu vermitteln. Es gibt keine universell gültige Menge von Aufgaben eines Betriebssystems; die Aufgaben und Funktionen hängen ab von dem vorgesehenen Anwendungsbereich des Rechners, der zugrundeliegenden Hardware und den Kosten des Rechners (in Relation zu den finanziellen Möglichkeiten seiner Benutzer). Der große Einfluss der Kosten auf die von einem Betriebssystem anzubietenden Funktionen zeigte sich am deutlichsten am Stapelbetrieb, der in erster Linie der Kostenminimierung dient.

Wir haben auch gesehen, dass Betriebssysteme neben Datenbankmanagementsystemen, Datenübertragungssystemen und Fenster- oder Graphik-Systemen nur ein Teil der Systemsoftware sind und dass die Grenzen zwischen diesen Bereichen oft nicht ganz scharf sind.

Die wichtigsten und häufigsten Aufgabengebiete bzw. Dienstleistungen eines Betriebssystems sind: Gerätesteuerung, Schutz, Fehlerbehandlung, Realisierung von parallelen Prozessen, Prozess-Synchronisation bzw. -Kommunikation, Betriebsmittelverwaltung, Realisierung einer Kommandosprache und Administration. Applikationen können die Dienste des Betriebssystems über die (Applikations-) Programmierschnittstelle in Anspruch nehmen.

Das gesamte Betriebssystem besteht aus Dienstprogrammen, Bibliotheken und dem Betriebssystemkern; der Kern selbst besteht aus einem hardwareabhängigen und einem hardwareunabhängigen Teil. Die Funktion des Betriebssystemkerns basiert ganz wesentlich auf Unterbrechungen, die z. B. von Geräten erzeugt werden können und einen asynchronen Aufruf einer Unterbrechungsroutine verursachen. Durch einen Zeitgeber können regelmäßige Unterbrechungen erzeugt werden, die das Betriebssystem dazu benutzen kann, die CPU abwechselnd verschiedenen Prozessen zuzuteilen und damit mehrere Programme scheinbar parallel auszuführen.

Ob einzelne Funktionen in einem Betriebssystem realisiert werden können, hängt vielfach davon ab, ob die vorhandene Hardware dies unterstützt. Erste Beispiele, die wir in dieser Kurseinheit kennengelernt haben, waren: Speicherschutzmechanismen wie das Grenzregister, supervisor call, verschiedene CPU-Modi und der Zeitgeber.

Literatur

[Ba90] D. W. Barron. *Computer operating systeme – for micros, minis and mainframes*. Chapman and Hall, 1990.

[Ca82] P. Calingaert. *Operating systems elements – a user perspective*. Prentice Hall, 1982.

[De90] H. M. Deitel. *An introduction to operating systems*. Addison–Wesley, 1990.

[HILS05] J. Haake, C. Icking, B. Landgraf und T. Schümmer. *Verteilte Systeme, Kurs 1678*. FernUniversität Hagen, 2005.

[POSIX88] *Portable Operating System Interfase for Computer Environments (IEEE Standard P1003.1) Draft 12.3*. IEEE, 1988/05.

[SiGa98] A. Silberschatz, and P.B. Galvin. *Operating system concepts – Fifth Edition*. Addison–Wesley, 1998.

[St03] W. Stallings. *Betriebsysteme: Prinzipien und Umsetzung 4., überarbeitete Auflage*. Pearson Studium, 2003

[Ta02] A. S. Tanenbaum. *Moderne Betriebssysteme: 2., überarbeitete Auflage*. Pearson Studium, 2002.

[Un02] C. Unger. *Kommunikations- und Rechnernetze, Kurs 1690*. FernUniversität Hagen, 2002.

Glossar

Applikation (*application, user program*) Anwendungsprogramm, das ein Problem eines Anwenders / Benutzers löst.

Benutzermodus (*user mode*) Ausführungsmodus der CPU, der für Applikationen gedacht ist und in dem privilegierte Befehle unzulässig sind.

Betriebsart Methode, wie ein Rechner Aufträge entgegennimmt und abarbeitet.

Betriebsmittel (*resource*) wird zur Ausführung von Applikationen benötigt und von Betriebssystem verwaltet; Beispiele: CPU, Geräte, Hauptspeicher; kann physisch oder logisch sein.

Bindemodul (*object module*) Ergebnis der Übersetzung eines Quellprogramm-Moduls.

Binder (*binder, linkage editor*) Dienstprogramm, welches mehrere Bindemoduln zu einem Binde- oder Lademodul vereinigt und dabei modulübergreifende Referenzen herstellt.

Dienstprogramm (*utility*) Teil des Betriebssystems; erfüllt Standard-Aufgaben bei der Administration des Rechners, Textverarbeitung usw.

direct memory access (*DMA*) Fähigkeit von Geräten (bzw. Controllern), selbständig Blöcke in den oder aus dem Hauptspeicher zu übertragen.

Fenstersystem (*window system*) Menge von Programmen, die die Programmierung von graphischen Benutzungsschnittstellen unterstützen.

Gerätetreiber (*device driver*) Programm, das von der CPU ausgeführt wird und ein bestimmtes Gerät steuert.

Grenzregister (*fence register*) Register, das die Grenze zwischen dem Benutzerbereich und dem Betriebssystem-Bereich im Hauptspeicher enthält; wird zum Speicherschutz benutzt.

Hintergrundprozess (*background process*) Prozess, der im Dialogbetrieb gestartet wurde und parallel zu dem Dialogprozess ohne interaktive Ein-/Ausgaben läuft.

interaktiver Betrieb (*interactive processing mode*) Betriebsart, bei der ein Prozess über geeignete Ein-/Ausgabegeräte (meist Tastatur und Bildschirm) während seiner Laufzeit mit einem Benutzer kommuniziert.

Kern (*kernel, nucleus*) Teil des Betriebssystems, der zur Laufzeit immer im Hauptspeicher verfügbar sein muss und der insb. die Systemaufrufe realisiert.

Kommandosprache (*command language*) Sprache, in der Benutzer ihre Aufträge an das Betriebssystem formulieren; erlaubt insb. den Aufruf von Programmen.

Lademodul (*load module*) Maschinenprogramm, das von einem Lader in den Hauptspeicher geladen werden kann.

Mehrprogrammbetrieb (*multi-programming*) Die Fähigkeit eines Betriebssystems, mehrere Programme auf einem einzigen physisch vorhandenen Prozessor zeitlich überlappend auszuführen.

Programmierschnittstelle (*application programming interface*) Menge der Systemaufrufe eines Betriebssystems.

Prozess (*process*) Programm in Ausführung.

Quellprogramm-Modul (*source module*) selbständig übersetzbarer Teil eines Programms in einer höheren Programmiersprache oder Assembler.

Realzeitbetrieb, Echtzeitbetrieb (*real time processing*) Betriebsart, bei der das Betriebssystem die Ausführung zeitkritischer Prozesse ermöglicht; erfordert eine spezielle Unterbrechungsverarbeitung.

Standard-Bibliothek (*standard library*) Teil des Betriebssystems; enthält *systemnahe* Funktionen, die von vielen Applikationen benötigt werden.

Stapelbetrieb (*batch processing*) Betriebsart, bei der ein Auftrag ohne direkte Kommunikation mit dem Auftraggeber abgewickelt wird.

Stapeljob (*batch job*) Auftrag an einen Rechner, der im Stapelbetrieb ausgeführt werden soll.

Supervisor call, SVC Maschinenbefehl, der eine spezielle Unterbrechnung erzeugt, durch die ein Anwenderprogramm indirekt einen Systemaufruf aufrufen kann.

Systemaufruf (*system call*) vom Betriebbssystem angebotene, von Applikationen aufrufbare Operation.

Systemmodus (*system mode, monitor mode, supervisor mode*) Ausführungsmodus der CPU, in dem privilegierte Befehle zulässig sind.

Systemsoftware (*system software*) allgemeine Bezeichnung für systemnahe Software, auf der Applikationen aufbauen.

Teilhaberbetrieb Betriebsart, bei der mehrere Terminals interaktiv mit einem einzigen Prozess kommunizieren.

Time-sharing System Mehrbenutzersystem, das gleichzeitig mehrere interaktive Prozesse ausführen kann.

Trap Software-Unterbrechung, die durch Fehler wie z. B. Division durch 0 oder Zugriffe auf unzulässige Adressen oder Systemaufrufe hervorgerufen wird.

Unterbrechung (*interrupt*) von Schnittstellen oder Geräten erzeugbarer asynchroner Unterprogrammaufruf, durch den die CPU bei der normalen Ausführung eines Programms unterbrochen wird, um eine Unterbrechungsroutine auszuführen.

Unterbrechungsregister (*interrupt register*) Register, in dem jedes gesetzte Bit das Eintreten einer Unterbrechung anzeigt.

Unterbrechungsvektor (*interrupt vector*) Tabelle von Startadressen von Unterbrechungsroutinen.

Urlader (*bootstrap loader*) Programm, das den Betriebssystemkern lädt.

Zeitscheibe (*time slice*) Zeitdauer, in der eine Applikation ausgeführt wird, bis ein Zeitgeber eine Unterbrechung erzeugt, nach der der Applikation der Prozessor u. U. entzogen wird.

Zugriffsmethode (*access method*) Menge von Operationen, durch die der Inhalt einer Datei in eine Menge kleinerer Speichereinheiten (z. B. Sätze oder Zeichen) zerlegt wird; Beispiele: sequentielle und index-sequentielle Zugriffsmethode.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 2:
Geräteverwaltung und Dateisysteme

Autoren: Udo Kelter, Christian Icking, Lihong Ma

Inhalt

Kurseinheit 2

Geräteverwaltung und Dateisysteme

2.1 E/A-Geräte

2.1.1 Gerätetypen

Die Hauptkomponenten eines Rechners sind normalerweise der (Applikations-) Prozessor, der Hauptspeicher und verschiedene Ein-/Ausgabe-Geräte (**E/A-Geräte**). Der Hauptspeicher wird gewöhnlich durch einen schnelleren, kleineren *Cache* ergänzt. Er ist ein Zwischenspeicher von Datenbewegungen zwischen Hauptspeicher und den Prozessorregistern mit dem Ziel der Leistungssteigerung. Wenn der Prozessor ein Wort aus dem Hauptspeicher lesen will, wird es zuerst im Cache gesucht. Wenn das Wort schon im Cache vorliegt, dann wird es direkt an den Prozessor übergeben, sonst wird ein Block des Hauptspeichers, der aus einer festen Anzahl von Wörtern besteht und in dem das gesuchte Wort liegt, in den Cache kopiert und das Wort an den Prozessor transferiert.

Im Laufe der Zeit ist eine große Zahl von Typen von E/A-Geräten mit unterschiedlichsten Eigenschaften entwickelt worden; die Entwicklung in diesem Bereich geht weiter rasant voran. Mit der Entwicklung neuer Gerätetypen oder der Verbesserung der Eigenschaften vorhandener Typen ändern sich auch ständig die Randbedingungen und Probleme beim Einsatz und der Verwaltung der Geräte. Dennoch lassen sich einige relativ konstante prinzipielle Aufgaben und Lösungsansätze ausmachen.

Beginnen wir mit dem Sinn und Zweck der E/A-Geräte in einem Rechner. Dieser liegt in zwei Bereichen:

1. permanente Speicherung großer Datenmengen: der Hauptspeicher eines Rechners ist hierzu i. d. R. aus mehreren Gründen nicht geeignet:
   - Er ist zu teuer oder zumindest wesentlich teurer als Speicherplatz auf E/A-Geräten.
   - Er ist zu klein: bei vielen Prozessoren können nur wenige Megabyte Hauptspeicher adressiert werden, während die Programme und Benutzerdaten wesentlich größer sind.
   - Der Inhalt des Hauptspeichers geht bei gängigen Technologien verloren, sobald die Stromversorgung ausfällt bzw. der Rechner ausgeschaltet wird.

E/A-Geräte, die der permanenten Speicherung großer Datenmengen dienen, nennt man auch **Sekundärspeicher**.

Der Hauptspeicher wird dementsprechend auch **Primärspeicher** genannt. Ein Sekundärspeicher besteht normalerweise aus einem **Datenträger** (auch **Medium** genannt) und einem **Laufwerk**. Bei manchen Sekundärspeichern sind die Datenträger austauschbar, bei anderen sind sie fest im Laufwerk eingebaut. Laufwerke sind üblicherweise fest im Rechner eingebaut bzw. mit diesem verbunden. Beispiele für Datenträger sind: Magnetplatten, Magnetbänder (Spulen bzw. Kassetten), optische Platten, optische Bänder, Disketten, Lochkarten, Lochstreifen und andere, siehe eine Speicherhierache in Abbildung 2.1[1].

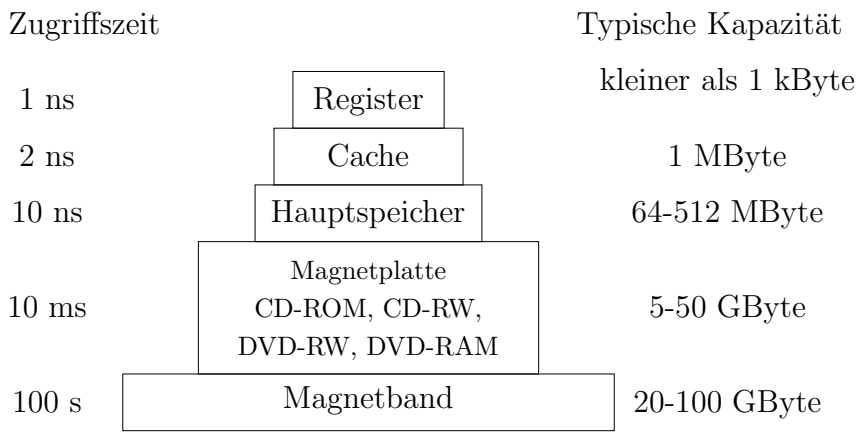


Abbildung 2.1: Jede Schicht arbeitet als Cache für die nächsttiefere Schicht. Die Kosten nehmen von oben nach unten ab, dagegen steigen die Kapazität und die Zugriffszeiten.

Zur Beurteilung verschiedener Sekundärspeicher ist es hilfreich, drei verschiedene Motivationen für die permanente Speicherung von Daten zu unterscheiden:

- Arbeitsdaten: diese Daten werden beim täglichen Betrieb eines Rechners benutzt und müssen möglichst schnell verfügbar sein.
- Sicherungskopien (back-up): dies sind Kopien der Arbeitsdaten; sie werden nur deshalb angefertigt, weil die Arbeitsdaten aus verschiedenen Gründen beschädigt werden könnten und man dann wenigstens eine Sicherungskopie der Arbeitsdaten hat (auch wenn diese vielleicht veraltet ist). Auf den Sicherungskopien wird normalerweise nicht gearbeitet, sie werden nur im Bedarfsfall zurückkopiert. Deshalb braucht der Zugriff zu Sicherungskopien nicht besonders schnell zu sein. Da man u. U. von den gleichen Arbeitsdaten zu verschiedenen Zeitpunkten wiederholt Sicherungskopien anfertigt, kann der Umfang der Sicherungsdaten sehr groß werden; da diese Daten ansonsten zu nichts nütze sind, ist es hier besonders wichtig, die Kosten für die Speicherung der Daten gering zu halten.
- Archivierte Daten: dies sind ebenfalls Kopien von Arbeitsdaten, sie weisen aber eine Reihe von Unterschieden zu Sicherungskopien auf:

[1]Wir verwenden die Abkürzungen *s*, *ms*, μs und *ns* für Sekunde, Millisekunde, Mikrosekunde und Nanosekunde, wobei 1 s = 10^3 ms, 1 ms = $10^3 \mu$s und 1μs = 10^3 ns.

Aus gesetzlichen oder anderen Gründen will man oft Daten über sehr lange Zeiträume (Jahrzehnte) verwahren. Daher kommt der Langzeitstabilität des Speichermediums hier eine größere Bedeutung zu.

Nachdem Arbeitsdaten archiviert worden sind, werden sie oft nicht mehr weiter bearbeitet und können als Arbeitsdaten gelöscht werden.

2. Kommunikation mit menschlichen Benutzern oder anderen Rechnern:

   Durch **Ausgabegeräte** werden im Rechner gespeicherte Informationen in eine visuelle, akustische oder sonstige Form umgewandelt. Beispiele sind: alphanumerische oder graphische Bildschirme von Terminals (Kathodenstrahlröhren, LEDs oder LCDs), Zifferndisplays, Anzeigelampen, Hupen, Stimm- oder Tongeneratoren, Drucker, Fotosatzsysteme, Stanzer, Plotter, Stellmotoren usw.

   Durch **Eingabegeräte** werden umgekehrt Informationen aus der Umwelt aufgenommen und in eine digitalisierte Form umgewandelt. Beispiele sind: Scanner, Klarschriftleser, Strichcode-Leser, Tastaturen von Terminals, Mäuse, Lichtgriffel, Messgeräte oder Mikrophone mit Analog-Digital-Wandler, usw.

   Kommunikationsgeräte können Daten eingeben und ausgeben. Im Gegensatz zu den oben genannten Ein-/Ausgabegeräten ist hier das Hauptziel, ausgegebene Daten über ein Netzwerk zu einem anderen Rechner zu übertragen und dort inhaltlich unverändert wieder einzugeben. Für die Übertragung werden die Daten in elektrische (analoge oder digitale) Signale, Licht, Töne oder Radiowellen umgeformt. Beispiele für Kommunikationsgeräte sind: Modems (Modulatoren/Demodulatoren), ISDN (Integrated Services Digital Network), ADSL (Asymmetric Digital Subscriber Line), HFC (Hybrid Fiber Coaxial Cabel), Akustik-Koppler, Ethernet-Controller, X.25-Controller usw.

Die **Übertragungseinheit** zwischen Hauptspeicher und Gerät ist bei Speichergeräten meist ein Block und bei Geräten zur externen Ein-/Ausgabe und Kommunikation meist ein Zeichen. Man spricht daher oft auch von **Block-** bzw. **Zeichen-Geräten**. Blockorientierte Geräte z. B. Festplatten und Bänder speichern Informationen in Blöcken, die in der Regel eine feste Größe haben. Übertragungen erfolgen jeweils blockweise. Ein **Block** ist eine Folge von Zeichen (meist 512, 1024 oder 2048 Zeichen). Zeichenorientierte Geräte übertragen Daten byteweise ohne Blockstruktur. Maus, Terminals, Drucker, Kommunikationsanschlüsse sind zeichenorientierte Geräte.

Aus Platzgründen können hier nicht alle o. g. Gerätetypen diskutiert werden; viele sind auch nur für spezielle Anwendungsbereiche interessant. Wir werden uns daher im Folgenden auf die gängigsten Gerätetypen konzentrieren, und zwar Sekundärspeicher mit Platten oder Bändern als Datenträger. Deren Eigenschaften werden später genauer vorgestellt.

2.1.2 Controller

Viele Geräte werden nicht direkt mit der CPU verbunden, sondern über einen **Controller** (auch **Geräte-Steuereinheit** oder **Adapter** genannt). Ein Controller ist ein Chip oder mehrere Chips auf einer Platin, die das Gerät kontrolliert. Er bekommt Befehle vom Betriebssystem und führt sie aus. An einen Controller können oft mehrere Geräte angeschlossen werden. Die Schnittstellen zwischen Controller und Gerät

sind oft international standardisiert; hierdurch können die gleichen Geräte an verschiedene Rechner angeschlossen werden, sofern ein passender Controller verfügbar ist. Die Struktur der Kommunikation zwischen den Rechnerkomponenten ist dann in etwa wie in Abbildung 2.2 angegeben.

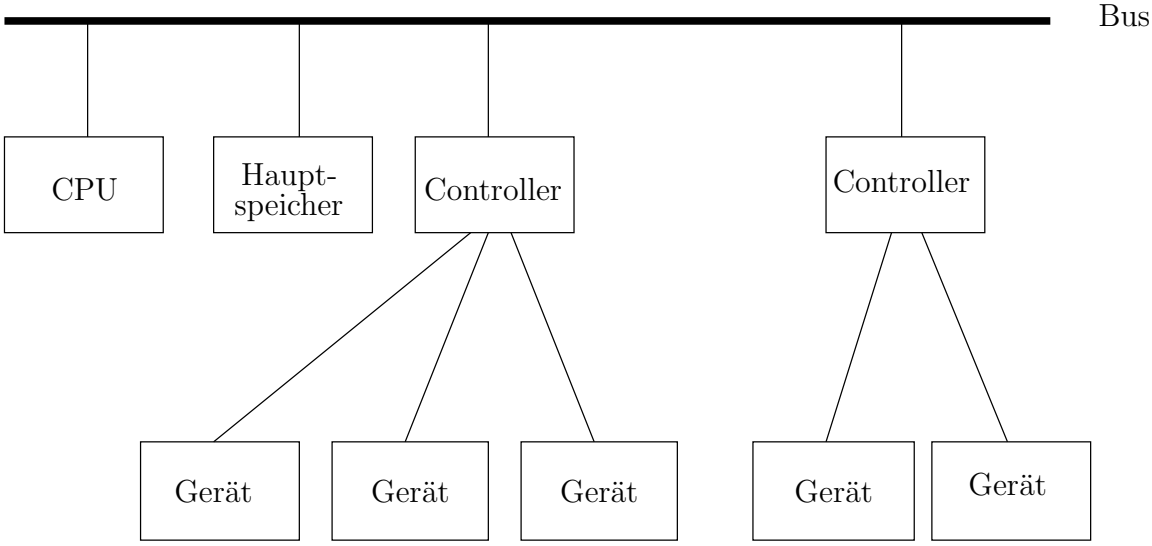


Abbildung 2.2: Struktur der Hardware-Komponenten eines Rechners.

Das Kommunikationssystem zwischen CPU[2], Hauptspeicher und Controllern ist meist ein Bus-System. Früher waren Kanäle üblicher. Der Controller hat normalerweise die Form einer Steckkarte und ist mit CPU und Hauptspeicher zusammen im gleichen Gehäuse untergebracht. Die Geräte haben oft eigene Gehäuse und werden durch vielpolige Kabel mit dem Controller verbunden.

Der Grund für die Einführung von Controllern liegt vor allem darin, dass andernfalls die CPU mit sehr vielen elementaren Aufgaben bei der Steuerung der Geräte belastet würde, z. B. bei der Steuerung der Köpfe eines Plattenlaufwerks oder durch die Berechnung von Prüfsummen zur Fehlererkennung bei Datenübertragungen. Durch diese Aufgaben würden die Applikationen, die die CPU ausführen soll, erheblich verlangsamt. Diese Aufgaben werden in den Controllern durch eigene Prozessoren (die meist langsamer und billiger, manchmal aber auch vom gleichen Typ wie der CPU-Prozessor sind) ausgeführt. Der Controller simuliert für die CPU ein abstrakteres, fehlerfreieres Gerät als das oder die an den Controller angeschlossenen Geräte. Der Trend geht dahin, immer mehr „Intelligenz“ in die Controller zu verlagern, um dadurch die CPU zu entlasten.

Die entscheidende Konsequenz für die Arbeitsweise von Betriebssystemen hieraus ist, dass diese fast immer mit Controllern und nicht mit den Geräten selbst kommunizieren. Dies hat noch den zusätzlichen Vorteil, dass man neuartige Geräte mit Hilfe eines Controllers, der einen bereits bekannten Gerätetyp simuliert, an einem Rechner betreiben kann, ohne das Betriebssystem zu ändern.

[2]Unter CPU verstehen wir hier nicht nur den eigentlichen Prozessor, sondern auch alle damit „direkt“ zusammenhängenden, oft auf der gleichen Platine untergebrachten Bauteile, z. B. Caches, memory management unit usw.

CPU und Controller benutzen spezielle Register, reservierte Worte im Hauptspeicher oder andere von der speziellen Hardware abhängige Mittel, um miteinander zu kommunizieren. Die das Betriebssystem ausführende CPU übermittelt den Controllern Aufträge und ggf. Parameter, die Controller geben Abschlussmeldungen, Fehlercodes, gelesene Daten u. ä. zurück.

2.1.3 Ein Schichtenmodell für E/A-Software

Durch die in einem Rechner vorhandenen Geräte sind wir prinzipiell in der Lage, Daten permanent zu speichern, einzugeben und auszugeben, allerdings nur auf dem relativ primitiven *Sprach*-Niveau, auf dem die CPU mit den Geräten bzw. Controllern kommuniziert.

Auf der anderen Seite haben wir die Benutzer eines Rechners, die Anwenderprogramme schreiben und die Daten ein- oder ausgeben oder speichern möchten, allerdings auf einem wesentlich höheren Sprachniveau, z. B. in höheren Programmiersprachen, wo von allen aus Sicht der Benutzer unwichtigen Besonderheiten der Geräte und lästigen Detailproblemen bei deren Steuerung abstrahiert wird.

Die in einem Rechnersystem auftretenden Sprachebenen wurden bereits in Abschnitt 1.2.1 erläutert. Wir wiederholen die wichtigsten Punkte noch einmal kurz.

Die konventionelle Maschinenebene ist charakterisiert durch den Befehlsvorrat des Prozessors und die Befehle, die die Geräte bzw. Controller ausführen können. Auf dieser Ebene sind noch der reale Hauptspeicher, Unterbrechungen (Interrupts) und ähnliche Details der *nackten* Maschine sichtbar.

Die Betriebssystemebene ist i. W. eine Erweiterung der konventionellen Maschinenebene; zusätzlich vorhanden sind insb. die Systemaufrufe, z. B. Operationen mit Dateien. Einen Systemaufruf, der Ein- oder Ausgaben veranlasst, nennen wir auch **E/A-Systemaufruf** (im Gegensatz etwa zu einem Systemaufruf für die Prozessverwaltung). Da wir es in dieser Kurseinheit nur mit E/A-Systemaufrufen zu tun haben werden, werden wir dies nicht immer betonen, sondern zur Sprachvereinfachung E/A-Systemaufrufe kurz als **Systemaufrufe** bezeichnen.

Die Ebene *Assemblersprache* ist die unterste Ebene, auf der normalerweise noch programmiert wird. Diese und alle höheren Sprachschichten interessieren hier nicht weiter. Durch Assembler oder Compiler wird die Funktionalität der Betriebssystemfunktionen, die mit Geräten zusammenhängen, nicht erweitert, sondern i. W. nur in syntaktisch besserer Form verfügbar gemacht, d. h. alle diesbezüglichen Funktionen müssen bereits in der Betriebssystem-Schicht realisiert werden. Den Teil des Betriebssystems, der diese Funktionen realisiert, nennen wir hier **E/A-Software**.

Geräteunabhängige E/A-Software und Geräte-Treiber. Man ist generell bestrebt, möglichst große Teile der E/A-Software geräteunabhängig zu halten. Es sollte möglich sein, Programme zu schreiben, die mit Dateien auf unterschiedlichen Geräten arbeiten, ohne dass zuvor das Programm für den jeweiligen Gerätetyp geändert und neu übersetzt werden muss. Beispielsweise ist die Verwaltung verschieden großer Platten prinzipiell nicht sonderlich verschieden, lediglich einige *technische Daten* der Platten müssen berücksichtigt werden. Erhebliche Unterschiede können die Platten bzw. ihre Controller hingegen in den Details ihrer Steuerung, Art der Datenübergabe, Verwendung von Registern usw. aufweisen.

Das Betriebssystem und speziell die E/A-Software ist deshalb intern in weitere drei Schichten strukturiert, wie in Abbildung 2.3 gezeigt, siehe auch das Beispiel über CP/M in Abschnitt 1.2.2. Jede Schicht hat eine genau bestimmte Funktionalität und wohl definierte Schnittstelle zu den angrenzenden Schichten.

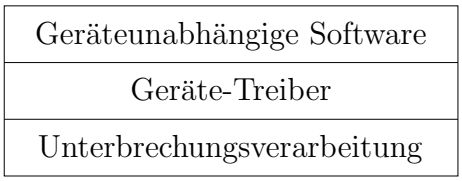


Abbildung 2.3: Schichten in der E/A-Software.

Alle geräteabhängigen Funktionen werden in die (**Geräte-**) **Treiber** verlagert. Für jedes reale Gerät (genauer gesagt für jeden Controller) ist ein eigener Treiber vorhanden, der vom Hersteller des Gerätes programmiert und mit dem Gerät zusammenausgeliefert wird. Gerätetreiber kommunizieren direkt mit deren Controllern über den Bus. Die Treibersoftware ist übrigens oft zeitkritisch, d. h. die absoluten Antwortzeiten müssen innerhalb gewisser Grenzen liegen. Die naheliegende Funktion eines Gerätetreibers ist das Entgegennehmen von Lese- und Schreibbefehlen aus höher liegenden Schichten, die Überprüfung der Korrektheit der Eingabeparameter und die Übersetzung solcher Parameter zu konkreten physischen Werten, z. B. ein Plattentreiber muss eine Blockadresse in eine Spur-, Sektor- und Zylindernummer umrechnen können. Normalerweise ist ein Gerätetreiber ein Teil des Betriebssystemkerns, siehe Abbildung 2.4, damit er auf die Hardware und auf die Register des Controllers zugreifen kann.

Typische Funktionen, die in der geräteunabhängigen E/A-Software realisiert werden, sind: Verwalten der Geräte, Realisierung von externen Namen für die Geräte, Simulation einheitlicher Sektorgrößen bei Platten, Verwaltung von Puffern im Hauptspeicher, Verwaltung von freien Sektoren auf Platten und anderes. Aus Effizienzgründen kann es notwendig sein, einzelne Funktionen in die Treiber zu verlagern, d. h. die Grenze zwischen geräteabhängiger und geräteunabhängiger Software ist nicht ganz eindeutig.

Die Unterbrechungsverarbeitung verarbeitet alle Arten von Unterbrechungen im Rechner. Auf die Details dieser Schicht kann hier nicht näher eingegangen werden. Ausführlichere Beschreibungen finden sich in [Ha98] und [Ta02].

2.1.3.1 Verwaltung von E/A-Systemaufrufen

Wenn in einem Rechner mehrere Applikationen parallel ausgeführt werden, so werden diese unabhängig voneinander Ein- oder Ausgaben durchführen wollen und hierzu passende Systemaufrufe aufrufen. Die von den Applikationen aufgerufenen und vom Betriebssystem exportierten Systemaufrufe sollen normalerweise **synchron** arbeiten, d. h. ein Systemaufruf ist nach Rückkehr zur aufrufenden Applikation fertig ausgeführt[3].

[3]Eine Ausnahme hiervon stellen manche Ausgaben dar, bei denen es sinnvoll sein kann, bereits vor ihrer Beendigung die aufrufende Applikation weiter auszuführen. Die Ausgabe wird dann asynchron zur Applikation unter der Kontrolle des Betriebssystems zu Ende ausgeführt. Aller-

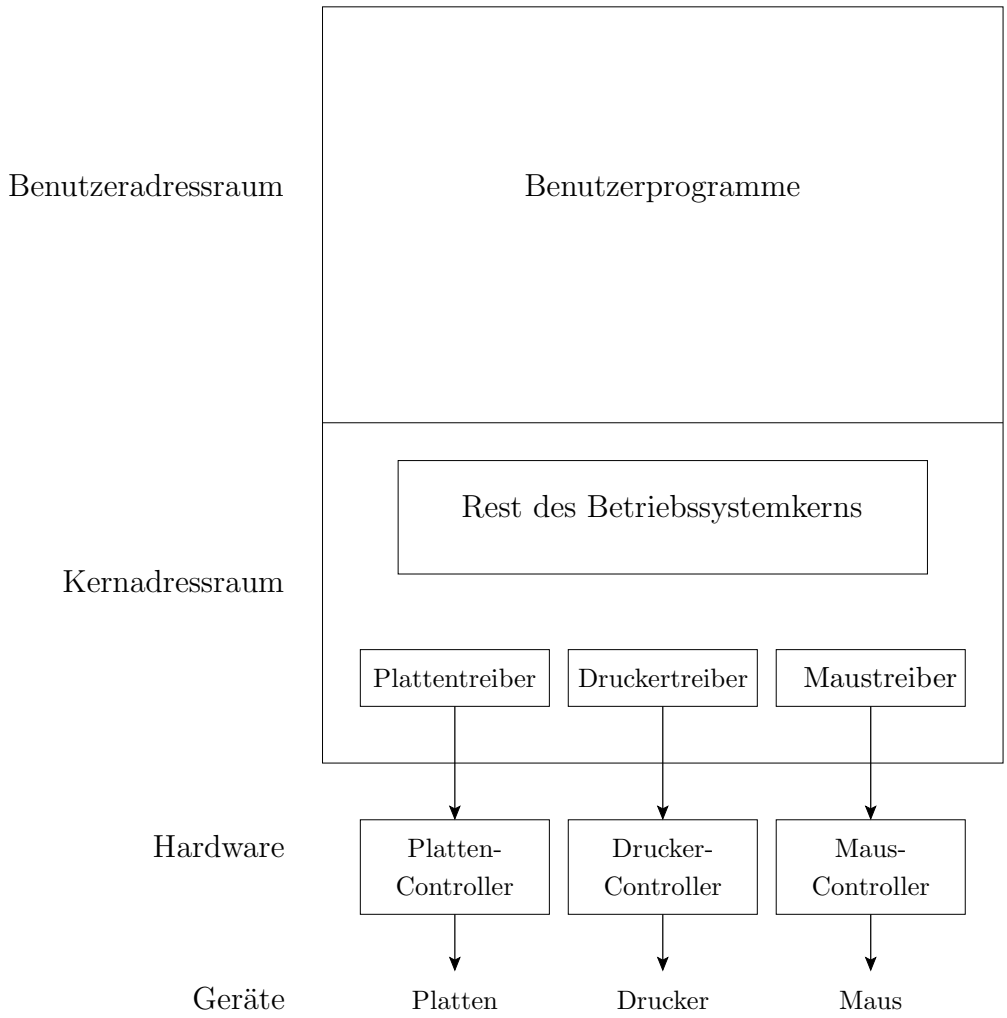


Abbildung 2.4: Positionierung der Gerätetreiber im Hauptspeicher.

Wie wir schon in Kurseinheit 1 gesehen haben, sind E/A-Operationen auf Maschinenebene jedoch **asynchron**. *Eine Ein-/Ausgabe* besteht auf dieser Ebene i. W. aus folgenden Phasen:

1. aus einer *Startphase*, in der die CPU dem Gerät einen **E/A-Auftrag** übermittelt. Hierzu ruft die CPU eine oder mehrere E/A-Operationen mit dem angesprochenen Gerät auf, wobei die erforderlichen Parameter übergeben werden und wodurch letztlich das Gerät gestartet und ein Datentransport zwischen Gerät und Puffern bzw. Registern initiiert wird.

2. aus einer *Wartephase*, in der auf das Ende des Transports gewartet wird.

3. aus einer *Nachbearbeitungsphase*, in der z. B. eventuell aufgetretene Fehler analysiert, Puffer reorganisiert und ähnliche Aufgaben erledigt werden. Das Ende der Wartephase kann z. B. durch eine Unterbrechung angezeigt werden; man

dings können bei solchen Systemaufrufen keine Fehleranzeigen übergeben werden, denn der Fehler kann auftreten, nachdem der Systemaufruf aus Sicht der Applikation schon beendet ist. Die Fehler müssen daher auf andere Weise zu einem späteren Zeitpunkt abgefragt werden; für diese Abfrage ist es unvermeidlich, auf das Ende der betroffenen Ausgaben zu warten.

kann auch wiederholt zu späteren Zeitpunkten explizit (mit Hilfe weiterer E/A-Operationen) prüfen, ob die angestoßene Ein-/Ausgabe schon beendet worden ist.

Während des Wartens wird man den Prozess, der die E/A-Operation veranlasst hat, zunächst in einen Wartezustand *blockiert* versetzen und die CPU einem anderen ausführungsbereiten Prozess zuteilen, bis auch dieser eine Ein- oder Ausgabe veranlasst und ebenfalls in einen Wartezustand versetzt wird. Im allgemeinen führen die Applikationen also *mehrere überlappende Ein-/Ausgaben* durch.

Mehrere Ein-/Ausgaben können für das gleiche Gerät gelten; bspw. können mehrere Prozesse Sektoren auf der gleichen Platte lesen oder schreiben. Daher muss für jedes Gerät eine **E/A-Auftragsliste** geführt werden. Bei Geräten, die immer nur eine Ein-/Ausgabe zugleich ausführen können (z. B. ein Bildschirm), kann immer nur ein Auftrag initiiert sein, alle übrigen müssen warten. Wenn das Gerät *fertig* meldet, ist automatisch klar, welcher Auftrag nun beendet ist. Bei Geräten, die mehrere Aufträge parallel behandeln können und bei denen die Aufträge nicht notwendigerweise in der Reihenfolge beendet werden, in der sie angekommen sind (z. B. bei manchen Platten/Controllern), muss bei jeder Fertigmeldung des Geräts zugleich angegeben werden, welcher Auftrag gemeint ist (siehe auch Abschnitt 2.2.4.1).

Außer den wartenden bzw. schon initiierten E/A-Aufträgen muss das Betriebssystem einige weitere Angaben über den Zustand jedes Geräts verwalten. Ein Terminal kann z. B. ausgeschaltet sein, ein Drucker kann einen Papierstau haben oder gerade mit dem Ausdrucken einer Seite beschäftigt sein oder eine Kommunikationsverbindung kann gerade gestört sein. Die **Gerätezustandstabelle** enthält für jedes angeschlossene Gerät alle derartigen Angaben sowie die E/A-Auftragsliste.

Bei jeder Unterbrechung, die durch ein E/A-Gerät verursacht worden ist, wird zunächst der zugehörige Eintrag in der Gerätezustandstabelle lokalisiert und die dort vorhandenen Angaben zum Zustand werden ggf. aktualisiert. Sofern ein früher initiierter E/A-Auftrag nunmehr beendet ist, wird der entsprechende Eintrag aus der E/A-Auftragsliste entfernt und ggf. wird der bisher blockierte Prozess in den Zustand *ausführungsbereit* versetzt. Außerdem kann ggf. ein bisher wartender E/A-Auftrag nunmehr initiiert werden.

2.1.3.2 Virtuelle Geräte

Kehren wir wieder zu den schon erwähnten Entwicklern von Programmen bzw. den späteren Benutzern dieser Programme zurück. Diese wünschen sich in vielen Fällen Geräte mit Eigenschaften und Fähigkeiten, die die real existierenden Geräte gar nicht aufweisen. Hierzu zwei Beispiele:

- **Dateisysteme**: Magnetplatten und ähnliche Speichergeräte sind - selbst in idealisierter Form - normalerweise keine Geräte, mit denen Entwickler von Programmen gerne umgehen würden (oder können). Stattdessen möchten sie aus ihrer Sicht zusammengehörige Mengen von Daten in **Dateien** speichern können. Dateien ersetzen daher aus Sicht der Benutzer die realen Speichergeräte vollständig. Man kann also Dateien als virtuelle Geräte ansehen, die aus Benutzersicht gleichrangig neben realen Geräten wie einem Bandgerät oder Drucker stehen.

- **Spooling**: Drucker und ähnliche Geräte stehen normalerweise nur in sehr geringer Anzahl zur Verfügung, werden andererseits von sehr vielen parallelen Prozessen benutzt. Zwei parallele Prozesse können natürlich nicht überlappend denselben Drucker benutzen: wenn zwei Prozesse gleichzeitig auf demselben Drucker drucken würden, würden die beiden Ausdrucke vermischt werden. Drucker sind exklusiv zu benutzende Betriebsmittel, Prozesse müssen daher nacheinander den Drucker benutzen. Zur Lösung dieses Problems verwendet man eine Technik namens *Spooling*, bei der das Betriebssystem für jeden Prozess einen eigenen privaten Drucker simuliert. Hierbei werden die Daten, die ein Prozess drucken will, zunächst in einer Datei gesammelt. Anschließend wird die Datei in eine Warteschlange (*Spooling-Verzeichnis*) eingefügt. Ein eigener zyklischer Prozess, der auch *Dämon* (*daemon*) oder Hintergrundprozess genannt wird, entnimmt dann immer wieder eine auszudruckende Datei aus dem Spooling-Verzeichnis und gibt sie direkt auf dem Drucker aus.

Wir gehen im Folgenden davon aus, dass oberhalb der Betriebssystemschicht nur noch **virtuelle Geräte** benutzt werden, während das Betriebssystem selbst reale Geräte benutzt. Die wichtigsten Merkmale von virtuellen Geräten (incl. Dateien) im Gegensatz zu realen Geräten sind:

- virtuelle Geräte können eine wesentlich andere Funktionalität (bzw. Spezifikation) als reale Geräte haben;
- die Operationen auf virtuellen Geräten arbeiten synchron;
- es kann erheblich mehr virtuelle als reale Geräte geben.

Es ist hilfreich, reale und virtuelle Geräte wie abstrakte Datenobjekte anzusehen: Jedes Gerät hat einen bestimmten Typ. Für jeden Gerätetyp gibt es eine **Spezifikation**, die die möglichen Operationen auf dem Gerät beschreibt. Beispiele für die Spezifikation virtueller Gerätetypen sind der Datentyp file in Pascal oder index-sequentielle Dateien, siehe Abschnitt 2.3.4.1; man nennt diese Datentypen auch **Zugriffsmethoden** für Dateien. Die Spezifikation eines realen Geräts (bzw. eines Controllers) ist durch die Hardware gegeben; derartige Gerätespezifikationen sind oft (durch DIN, ECMA, IEEE, ISO oder andere Gremien) standardisiert.

2.1.4 Entwurfsziele für E/A-Software

Hauptaufgaben der E/A-Software sind die Verwaltung der Geräte, Realisation von virtuellen Geräten und Zugriffsmethoden und die Abwicklung der E/A-Systemaufrufe. Darüber hinaus soll die E/A-Software, speziell die Schnittstelle zu den Benutzerprogrammen, folgende *Eigenschaften* aufweisen:

Geräteunabhängigkeit von Applikationen: Ein Benutzerprogramm soll, soweit es sinnvoll möglich ist, mit verschiedenen (realen oder virtuellen) Geräten arbeiten können. Beispielsweise sollte ein Benutzerprogramm, das mit Dateien arbeitet, nicht berücksichtigen müssen, ob die Dateien auf einer Diskette, einer Magnetplatte oder einer optischen Platte gespeichert sind.

Der Schlüssel zur Realisierung der Geräte-Unabhängigkeit liegt in der Definition von virtuellen Gerätetypen (s. o.). Die Grundidee ist die gleiche wie bei abstrakten Datentypen: der virtuelle Gerätetyp hat eine spezifizierte Schnittstelle. Diese Schnittstelle kann mehrere Implementierungen haben. In den Implementierungen können unterschiedliche reale Gerätetypen eingesetzt werden. Beispiele für solche Schnittstellen mit mehreren Implementierungen sind:

- sequentielle Dateien: diese können sowohl auf Magnetplatten wie auf Magnetbändern verschiedenen Typs realisiert werden.
- *einfache* alphanumerische Terminals: spezifiziert sind diese i. d. R. durch einen Gerätetyp eines größeren Herstellers, ein Beispiel für einen derartigen *de-facto Standard* ist VT100 von DEC.

Sofern von einem Gerätetyp mehrere äquivalente und austauschbare Exemplare in einem Rechner vorhanden sind, z. B. Drucker oder Bandlaufwerke, so sollte die aktuelle Zuordnung transparent (d. h. unsichtbar) für das Benutzerprogramm sein. Das Benutzerprogramm gibt nur den Befehl aus, zu drucken oder z. B. das Band mit Etikett XYZ zu montieren. Das Betriebssystem stellt dann fest, welcher Drucker bzw. welches Bandlaufwerk frei ist und wählt eines aus. Im Falle der Bandlaufwerke würde z. B. eine Meldung auf der Operateur-Konsole erscheinen *Bitte Band XYZ in Bandlaufwerk B einlegen.*

Einheitliche Namen: Die Geräteunabhängigkeit erfordert, dass in den Benutzerprogrammen selbst keine Bezeichnungen für Geräte explizit oder implizit z. B. durch Operationsnamen angegeben werden. Stattdessen müssen die Dateien bzw. Geräte, mit denen ein Benutzerprogramm arbeiten soll, als Parameter angegeben werden. Die Namen können entweder beim Aufruf eines Programms mittels eines Kommandos in der Kommando-Sprache des Rechners übergeben werden oder als Parameter eines Systemaufrufs.

Die Namen von Dateien und Geräten sollten einen einheitlichen und gemeinsamen Namensraum bilden. Aus Sicht der Benutzer, die normalerweise nur mit Dateien arbeiten, stellt sich diese Forderung so dar, dass Geräte durch Dateien repräsentiert werden sollen.

Kodierungsunabhängigkeit: Geräte benutzen oft verschiedene Zeichen-Codierungen. Auf der Schnittstelle zu den Applikationen sollte eine für alle Geräte einheitliche Codierung verwendet werden z. B. nach ISO 8859/1 Latin 1. Die notwendigen Übersetzungen von und nach gerätespezifischen Codierungen müssen automatisch vorgenommen werden, z. B. im Geräte-Treiber.

Wechselseitiger Ausschluss: Dateien und viele andere Geräte sind Betriebsmittel, die von verschiedenen Prozessen nur nacheinander exklusiv benutzt werden können. Die E/A-Software muss daher den wechselseitigen Ausschluss bei der Benutzung dieser Geräte ermöglichen. Ein Folgeproblem des wechselseitigen Ausschlusses sind Deadlocks. Viele Betriebssysteme bieten aus pragmatischen Kosten/Nutzenabwägungen heraus für dieses Problem keine oder nur eine rudimentäre Lösung an. Den wechselseitigen Ausschluss und Deadlocks werden wir in Kurseinheit 5 ausführlicher behandeln.

Fehlerbehandlung: Auftretende Fehler sollten auf einer möglichst *tiefen* Schicht (d. h. möglichst Hardware-nah) behandelt werden und nur, wenn unvermeidlich, an höhere Schichten weitergemeldet werden. Beispiele:

- Ein Übertragungsfehler kann durch einen fehlerkorrigierenden Code schon im Controller korrigiert werden.
- Wenn ein Sektor einer Platte als defekt bekannt ist, dann kann bereits im Controller veranlasst werden, dass statt dieses defekten Sektors immer ein bestimmter Ersatzsektor benutzt wird.
- Wenn ein Sektor einer Diskette nicht lesbar ist und dies durch ein Staubkorn auf der Diskette verursacht wird, kann die E/A-Software das Problem durch mehrere Leseversuche lösen.

2.2 Plattenspeicher

In diesem Abschnitt werden die wesentlichen Eigenschaften von Plattenspeichern vorgestellt. Ein Plattenspeicher besteht aus einem Plattenlaufwerk z. B. Floppy-Laufwerke und einer darin montierten Festplatte. Das Hauptmerkmal ist es, dass ein Lese- und Schreibvorgang ungefähr gleich schnell ist, deshalb können sie ideal als Hintergrundspeicher eingesetzt werden.

Die bei weitem wichtigste Form von Plattenspeichern sind derzeit Magnetplattenspeicher; diese werden deshalb detailliert diskutiert. Disketten und optische Platten funktionieren im Prinzip recht ähnlich und werden deshalb anschließend nur kurz vorgestellt.

2.2.1 Magnetplatten

Eine **Magnetplatte** besteht aus mehreren (meist etwa 2 – 10) Scheiben aus einem festen Material, die in ihrer Mitte durch eine Achse fest miteinander verbunden sind und gemeinsam um diese Achse rotieren. Mit (*Magnet-*) Platte wird manchmal auch eine einzelne Scheibe bezeichnet und dementsprechend mit *Plattenstapel* die Achse und die darauf befindlichen Scheiben. Oft wird mit *Platte* auch das Plattenlaufwerk zusammen mit einer Platte bezeichnet. Wir nehmen im Folgenden an, dass die Achse senkrecht steht und die Scheiben somit waagerecht übereinander liegen.

Jede Scheibe ist auf beiden Seiten, die man auch **Oberflächen** nennt, mit einer magnetisierbaren Beschichtung (ähnlich wie ein Tonband) versehen. Die untere Seite der untersten Scheibe und die obere Seite der obersten Scheibe bleiben meist unbenutzt. Die Scheiben haben einen gewissen Abstand voneinander, der ausreicht, um einen **Arm** des Plattenlaufwerks zwischen je zwei Scheiben einfahren zu können. Am Ende eines Arms sind zwei **Lese-/Schreibköpfe** angebracht, von denen jeweils einer zu der oben und einer zu der unten angrenzenden Oberfläche gerichtet ist. Alle Arme sind starr miteinander verbunden. Die Arme können horizontal durch einen Stellmotor verschoben werden. Abbildung 2.5 zeigt schematisch einen senkrechten Schnitt durch die Achse der Magnetplatte und die Arme des Laufwerks.

Wenn ein Arm nicht bewegt wird, dann markiert ein darauf befestigter Lese-/Schreibkopf auf der sich drehenden Plattenoberfläche eine kreisförmige **Spur**, auf

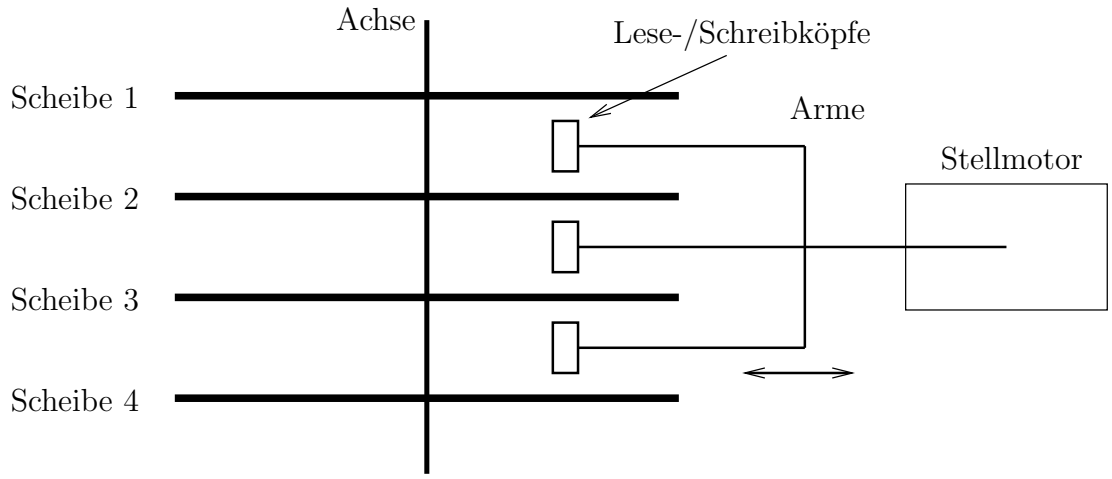


Abbildung 2.5: Aufbau einer Magnetplatte.

der Daten gelesen bzw. geschrieben werden können. Die Spuren, die einer bestimmten Position der Arme entsprechen und die somit senkrecht übereinander liegen, nennt man einen **Zylinder**.

Von den Eigenschaften der Oberflächen, Lese-/Schreibköpfe, Magnetisierungstechnik usw. hängen nun folgende Größen ab:

- die erforderliche Breite einer Spur;
- der erforderliche Abstand zwischen zwei Spuren;
- die maximale Dichte, mit der in einer Spur geschrieben werden kann.

Hieraus und aus dem nutzbaren Teil des Durchmessers der Platte ergibt sich die Zahl der Spuren der Platte. Derzeit übliche Platten haben mehrere tausend Spuren und können bis zu mehreren hunderttausend Byte pro Spur schreiben.

Die Menge der Daten, die auf einer Spur untergebracht werden können, ist damit noch viel zu groß, um als kleinste Einheit der Verwaltung des Inhalts einer Platte sinnvoll zu sein. Deshalb unterteilt man die Spuren noch einmal in gleich große **Sektoren**. Zwischen je zwei Sektoren bleibt aus technischen Gründen ein kleines Stück der Spur unbenutzt. In einem Sektor der Platte sind nicht nur die Nutzdaten, sondern auch Hilfsdaten für die Fehlerkorrektur und für organisatorische Zwecke gespeichert. Genauer gesagt, ein Sektor besteht aus drei Feldern: Präambel, Nutzdaten und Fehlerkorrektur. Die Präambel beginnt mit einem bestimmten Bitmuster, damit die Hardware den Beginn eines Sektors erkennen kann. Sie speichert außerdem die Nummern des Zylinders und des Sektors. Im Fehlerkorrekturfeld stehen redundante Informationen, die zur Wiederherstellung von Lesefehlern benötigt werden können. Die einzelnen Sektoren einer Spur können *unabhängig voneinander* gelesen und geschrieben werden, ein Sektor selbst aber immer nur *komplett*.

Die Zahl der Sektoren wird so gewählt, dass pro Sektor etwa 0,5-4 kByte Nutzdaten gespeichert werden können. Den Nutzinhalt eines Sektors nennen wir einen **Block**.

Sektoradressen. Man kann eine Platte als einen dreidimensionalen Array auffassen[4]. Ein Sektor wird innerhalb einer Platte identifiziert durch eine **Sektoradresse**; diese besteht aus

- der Nummer z der Spur (bzw. des Zylinders)
- der Nummer o der Oberfläche
- der Nummer s (bzw. Position) des Sektors innerhalb der Spur

Diese drei Nummern werden oft zu einer einzigen Nummer, die wir **Sektornummer** nennen, z. B. anhand der folgenden Formel kombiniert: wenn S die Zahl der Sektoren pro Spur und O die Zahl der Oberflächen ist, dann entspricht die Sektoradresse (z, o, s) dem Sektor mit Nummer $(z * O + o) * S + s$. Wir nehmen an, dass alle Nummerierungen mit 0 beginnen, also $0 \leq z < Z$, $0 \leq o < O$ und $0 \leq s < S$.

Operationen. Das Betriebssystem kann (über den Controller) einzelne Sektoren der Platte lesen oder schreiben; dies sind die beiden einzigen uns hier interessierenden Operationen. Die das Betriebssystem ausführende CPU übermittelt die Sektoradresse oder -nummer, Lese-/Schreibanzeige, Adresse von Pufferbereichen und ggf. weitere Angaben dem Controller über die durch die Hardware vorhandenen Kommunikationsmöglichkeiten.

Fest- und Wechselplatten. Die Magnetplatten-Technologie hat in den letzten Jahren rasante Fortschritte gemacht, vor allem durch die Steigerung der Schreib- und Spurdichte. Inzwischen sind Platten erhältlich, die mehrere Hundert Gigabyte speichern können. Die hohen Schreibdichten erfordern einen völlig staubfreien Zustand der Oberflächen. Dies kann nur dadurch erreicht werden, dass Platte und wesentliche Teile des Laufwerks in einem hermetisch verschlossenen Gehäuse untergebracht werden, d. h. die Platte ist fest im Laufwerk eingebaut (**Festplatte**) und ein Wechsel des Datenträgers im Laufwerk ist nicht möglich.

Früher übliche **Wechselplatten** sind nahezu ausgestorben. Hauptmotivation war seinerzeit die Kosteneinsparung, da so für mehrere Platten nur ein Laufwerk und die dazugehörige Steuereinheit benötigt wurden. Nachteilig war natürlich, dass immer nur eine Platte zugreifbar war; hinzu kam das nicht unerhebliche Risiko von Beschädigungen beim Montieren und Demontieren der Platte. Durch den Preisverfall für Festplatten ist diese Motivation für Wechselplatten verschwunden.

Eine andere Motivation für Wechselplatten existiert aber auch heute noch: Transport von großen Datenmengen zwischen Rechnern bzw. Sicherungskopien, die im Schadensfall sehr schnell installierbar sind. Hierfür ist heute der Austausch kompletter Plattenlaufwerke naheliegend. Wegen des immer kleineren Durchmessers der Platten (heute sind 3,5 Zoll üblich und 1 Zoll verfügbar; Mitte der 80er Jahre waren noch 8 – 10 Zoll üblich) und der geringeren Bauhöhe sind die kompletten Laufwerke

[4]Diese Vorstellung ist nur solange zulässig, wie alle Spuren die gleiche Anzahl von Sektoren enthalten. Dies ist bei Platten in der Praxis immer der Fall, nicht hingegen bei Disketten. So könnte ein Controller in verschiedenen Spuren verschieden große Sektoren anlegen. Auf manchen Diskettenlaufwerken werden auf den äußeren Spuren, die ja einen größeren Umfang haben, mehr Sektoren geschrieben als auf den inneren. Dies kann allerdings auf den äußeren Spuren eine Herabsetzung der Drehzahl erfordern, was bei einer Platte viel zu lange dauern würde.

inzwischen so leicht und handlich, dass man mit ihnen wie mit früheren Wechselplatten umgehen kann. Über USB- oder Firewire-Schnittstellen angeschlossene Festplatten lassen sich sogar im laufenden Betrieb anschließen oder entfernen.

Partitionen. Die Kapazitäten moderner Festplatten werden immer größer, und es ist manchmal sinnvoll, mit einem so großen Festplattenspeicher nicht in einem Stück zu arbeiten, z. B. manche Dateisysteme für die Unterstützung größerer Festplatten nicht ausgelegt sind. Einige Dateisysteme unterstützten zwar größere Festplatten, aber ein übermäßiger Verwaltungsaufwand ist die Folge.

Zu diesem Zweck kann man die Festplatten in Partitionen aufteilen, auf jede Partition kann dann wie auf eine separate Festplatte zugegriffen werden. Dies wird durch das Hinzufügen von Partitionstabellen ermöglicht. Bei den modernen Computern steht im Sektor 0 der **Master Boot Record** (MBR), der Bootcode und die Partitionstabelle enthält, siehe Abbildung 2.6. Jeder Eintrag in der Partitionstabelle enthält mehrere wichtige Angaben über die Partition, z. B. den Startsektor und die Größe der Partition, also wo die Partition auf der Festplatte beginnt und endet. Damit überhaupt von der Festplatte gebootet werden kann, muss eine der Partitionen als *aktiv* gekennzeichnet werden. Das aktiv-Flag wird von Bootloadern verwendet, d. h. das Betriebssystem wird von der Partition gestartet, die als aktiv markiert ist. In einem PC hat die Partitionstabelle Platz für vier Hauptpartitionen mit möglichen Unterpartitionen.

Abbildung 2.6: Partitionen einer Festplatte.

Bevor eine Festplatte benutzt werden kann, muss für jede Partition getrennt die Operation **High-Level-Formatierung** ausgeführt werden, d. h. ein Dateisystem wird angelegt. Die Formatierung legt einen *Bootsektor* oder *Bootblock*, den freien Speicherbereich für Administration, das Wurzelverzeichnis (root directory) und ein leeres Dateisystem an. Es wird noch zusätzlich ein Code in die Partitionstabelle geschrieben und damit gekennzeichnet, welches Dateisystem in der jeweiligen Partition verwendet wird. Der Bootsektor enthält u. a. den Urlader des Betriebsystems.

Sobald der Rechner eingeschaltet wird, wird das BIOS gestartet, das den Master Boot Record einliest und Bootcode ausführt. Das Bootprogramm überprüft, welche Partition aktiv ist. Danach wird der Bootsektor der entsprechenden Partition gelesen und ausgeführt. Der Urlader im Bootsektor durchsucht das Wurzelverzeichnis nach dem Betriebssystem oder einem größeren Bootlader. Anschließend wird das gefundene Programm in den Hauptspeicher geladen und ausgeführt.

2.2.2 Disketten

Disketten unterscheiden sich von Magnetplatten im Prinzip nur wenig, allerdings in ihren konkreten Leistungsmerkmalen erheblich: sie haben eine wesentlich kleinere Kapazität und sind erheblich langsamer als Platten, dem steht gegenüber, dass Laufwerk und Datenträger sehr preiswert sind:

- es ist nur eine Scheibe vorhanden. Die Scheiben sind aus flexiblem Material.
- Schreib- und Spurdichte sind erheblich geringer, damit auch die Empfindlichkeit gegenüber Staub. Disketten sind daher stets auswechselbare Datenträger.
- Die Diskette wird normalerweise nicht dauernd im Laufwerk gedreht, sondern nur, wenn auf sie zugegriffen wird. Der Motor muss also eigens gestartet werden. Außerdem dreht sich die Diskette im Vergleich zu Platten wesentlich langsamer, ebenso dauert das Positionieren auf eine gewünschte Spur länger.

2.2.3 Optische Platten

Optische Platten und ihre Laufwerke wurden abgeleitet aus Video-Platten bzw. CDs und DVDs. Die Plattenoberfläche wird durch einen Laserstrahl abgetastet und ggf. auch beschrieben. Pro Oberfläche können etwa 300 – 6000 MByte gespeichert werden. Optische Platten sind stets auswechselbare Datenträger und es ist nur eine Scheibe vorhanden; hierin ähneln sie Disketten. Manchmal sind beide Seiten der Scheibe benutzbar, aber das Laufwerk arbeitet nur auf einer Seite, d. h. die Platte muss ggf. im Laufwerk gedreht werden. Es gibt inzwischen auch **Plattenwechsler**; diese enthalten ein Laufwerk und ein Magazin, in dem etwa 3 bis 100 Platten verwahrt werden können. Durch spezielle Befehle kann eine der Platten in das Laufwerk eingelegt bzw. an seinen Platz im Magazin zurückgebracht werden.

Bzgl. des Schreibens von Daten sind drei Arten von optischen Platten zu unterscheiden:

1. nicht beschreibbare Platten (CD-ROM, Compact Disc-Read Only Memory): diese Platten werden wie CDs hergestellt und kommen als Distributionsmedium für Software oder große Datenmengen in Frage.
2. einmal beschreibbare Platten (CD-R, CD-Recordable und DVD-R): wie bei einem Stück Papier kann eine zunächst freie Stelle einmal beschrieben werden; danach kann diese Stelle nicht mehr verändert, sondern nur noch durchgestrichen werden. Diese Platten können vor allem für die Archivierung von Daten eingesetzt werden.
3. beliebig oft beschreibbare Platten (**magneto-optische** Platten; CD-RW, CD-ReWritable und DVD-RW, Digital Versatile Disk-RW): bei diesen Platten kann der Inhalt gelöscht und wieder neu beschrieben werden.

Aus Sicht des Betriebssystems können optische Platten beim Lesen im Wesentlichen wie Magnet-Platten behandelt werden. Das Schreiben einmal beschreibbarer Platten erfordert spezielle Maßnahmen, auf die wir hier nicht näher eingehen.

2.2.4 Datenübertragung zwischen Platte und Hauptspeicher

In diesem Abschnitt gehen wir auf einige weitere Details bei der Übertragung von Daten zwischen Platte und Hauptspeicher ein, die wir oben zurückgestellt haben.

Die Übertragung zwischen Platte und Hauptspeicher findet in zwei Etappen statt:

1. Übertragung zwischen Platte und Controller;
2. Übertragung zwischen Controller und Hauptspeicher.

Pufferung im Hauptspeicher. Beim Lesen eines Blocks wird insgesamt der Nutzinhalt eines Sektors der Platte in einen **Puffer im Hauptspeicher** übertragen, beim Schreiben umgekehrt.

Die Verwendung eines Puffers im Hauptspeicher hat verschiedene Gründe. Wenn ein Benutzerprozess einen Datenblock von einer Platte in einen Datenbereich in seinem Adressraum lesen will, führt er einen *read*-Systemaufruf aus und blockiert dann und wartet, bis die relative langsame Übertragung abgeschlossen ist. Danach wird der Block verarbeitet und der nächste Block gelesen, nun blockiert der Prozess wieder. Dieser Vorgang ist sehr ineffizient. Ein anderer Grund ist, während Übertragung stattfindet, muss der Prozess aus irgendeinem Grund seinen Adressraum aufgeben, siehe später die Kurseinheit 4, also gibt es eine Konflikt. Um solche Probleme zu lösen, wird ein *Puffer im Kernadressraum* angelegt. Die eingelesenen Daten werden zuerst an den Puffer übertragen. Wenn die Übertragung fertig ist, kopiert der Prozess den Block in einem Schritt in den Benutzeradressraum und fordert einen weiteren Block an. Dieses voraussehende Einlesen ist wesentlich effizienter, wenn der Block irgendwann einmal benötigt werden wird. Man kann die Effizienz noch erhöhen, indem zwei Puffer (*Double Buffering*) im Kernadressraum reserviert werden. Während das Betriebssystem ein Puffer leert bzw. füllt, überträgt ein Prozess Daten an den bzw. von dem zweiten Puffer.

Pufferung im Controller. Bei der Übertragung zwischen Platte und Controller, also wenn der Lese-/Schreibkopf über einen zu übertragenden Sektor fährt, müssen die Daten mit einer fest vorgegebenen sehr hohen Transferrate (Übertragungsrate) (mehrere MByte/sek) zum oder vom Controller übertragen werden. Dies erfordert, die Daten–also den Inhalt eines Sektors–komplett in einem **Puffer im Controller** zwischenzuspeichern. Eine direkte Weiterleitung der Daten in den Hauptspeicher kann daran scheitern, dass das für die Übertragung erforderliche Bus-System nicht frei ist; es kann von der CPU oder Geräten mit höherer Priorität stark belegt sein, so dass der Übertragungsversuch wiederholt werden müsste. Allerdings ist dieser Fall bei den meisten Rechnern unwahrscheinlich, weil die Übertragungsrate des Bussystems erheblich höher ist als die der Platte. Andererseits können z. B. Fehlerprüfungen erst nach vollständigem Lesen des Sektorinhalts im Controller durchgeführt werden; im Fehlerfall wäre die Übertragung sinnlos gewesen.

In einem separaten Schritt müssen die Daten zwischen Hauptspeicher und Puffer im Controller zeichenweise übertragen werden. Diese Übertragung kann entweder die CPU oder der Controller durchführen. Bei Controllern, die diese Fähigkeit haben, spricht man von **direct memory access** (**DMA**) Controller.

Es kann sein, dass der Controller immer gleich mehrere Sektorinhalte zwischen Platte und Puffern überträgt, z. B. die gleichzeitig von verschiedenen Köpfen überquerten Sektoren; hierbei sind diverse Varianten denkbar, die wir hier nicht alle durchgehen wollen. Auf jeden Fall wäre die Übertragungseinheit nicht ein Block, sondern mehrere Blöcke. Dies ist für unsere folgenden Betrachtungen aber unwesentlich, solange die einzelnen Blöcke im Hauptspeicher wieder identifiziert werden können. Wesentlich für die spätere Verwaltung von Platten ist, dass sich eine Platte aus Sicht des Betriebssystems als Menge einzeln adressierbarer Sektoren darstellt.

Übertragung per DMA. Der DMA-Controller kann unabhängig von der CPU auf das Bus-System zugreifen. Er enthält mehrere Register, die vom Prozessor gelesen und geschrieben werden können, um Lese/Schreib-Befehle zu übergeben. Zusätzlich gibt der Prozessor ein Kommando an den Controller der Festplatte ab, damit dieser die Daten von der Platte in seinen Puffer einliest und die Fehlerprüfung durchführt. Sobald die gültigen Daten im Puffer vorliegen, kann die Übertragung per DMA beginnen.

Der DMA-Controller schickt dem Controller der Festplatte einen Lesebefehl mit einer Adresse über das Bus-System, damit der Controller weiß, wohin er im Hauptspeicher z. B. das nächste Zeichen schreiben soll. Wenn das Schreiben beendet ist, schickt der Controller der Festplatte ein Signal über den Bus zum DMA-Controller. Dieser Ablauf wird wiederholt, bis die Datenübertragung abgeschlossen ist. Danach erzeugt der DMA-Controller eine Unterbrechung und teilt der CPU mit, dass die Daten jetzt im Hauptspeicher vorliegen.

Überlappung von Übertragungen. Nach der Beauftragung durch die CPU führt der Controller den Zugriff auf die Platte durch und informiert die CPU über den Abschluss des Zugriffs durch eine Unterbrechung. Während dieser Wartezeit wird der Prozess, für den der Plattenzugriff durchgeführt wird, in den Zustand *blockiert* überführt und ein anderer Prozess wird ausgeführt. Auch dieser kann u. U. einen Zugriff zu derselben Platte auslösen und wird blockiert und so weiter.

Insgesamt kann man nicht davon ausgehen, dass ein Controller einen Auftrag der CPU sofort ausführen kann, denn er kann noch dabei sein, einen anderen Auftrag abzuarbeiten. Der Auftrag muss deshalb warten. Die Warteschlange kann durch den Controller oder die CPU verwaltet werden, im letzteren Fall muss der Controller signalisieren, dass er im Moment keine weiteren Aufträge annehmen kann. Auf Strategien, nach denen der nächste abzuarbeitende Auftrag aus der Warteschlange ausgewählt werden kann, kommen wir später zurück.

Wir gehen im Folgenden davon aus, dass der Controller die Warteschlange verwaltet und die CPU von dieser Arbeit entlastet wird.

Transport zwischen Platte und Controller. Wenn ein Übertragungsauftrag aus der Warteschlange ausgewählt worden ist, läuft der Transport zwischen Platte und Controller in folgenden Schritten ab:

1. Die Lese-/Schreibköpfe werden auf die gesuchte Spur positioniert. Die hierfür benötigte Zeit nennt man **Suchzeit** oder auch **Positionierzeit**. Die Zeit für eine konkrete Positionierung hängt von der zu überwindenden Distanz ab. Heute übliche Platten benötigen dazu ca. 1 – 10 Millisekunden.

2. Es wird gewartet, bis sich die Platte soweit weitergedreht hat, dass der gesuchte Sektor bei den Köpfen erscheint. Die hierfür benötigte Zeit nennt man **Latenzzeit**. Diese hängt von der Rotationsgeschwindigkeit der Platte (heute üblicherweise 4500 – 10800 U/min) ab. Im Durchschnitt muss 1/2 Umdrehung abgewartet werden, was ca. 2,8 – 7 Millisekunden ausmacht. Manche Platten haben doppelte, einander gegenüber angeordnete Arme; hierdurch halbiert sich die Latenzzeit.
3. Während der Kopf über den Sektor gleitet, werden die gelesenen bzw. geschriebenen Daten vom bzw. zum Controller übertragen. Die hierfür benötigte Zeit nennt man **Übertragungszeit**. Diese hängt i. W. von der Rotationsgeschwindigkeit der Platte und der Schreibdichte ab und liegt in der Größenordnung von 1 Millisekunde.

Die Summe der drei oben genannten Zeiten bezeichnen wir als **Zugriffszeit**. Weitere Aufgaben beim Transport von Sektorinhalten sind das Berechnen von Prüfsummen zur Fehlerkorrektur und die Auswertung diverser Steuersignale. Moderne Controller sind aber so schnell, dass die hierfür erforderlichen Zeiten vernachlässigbar sind.

Da die Zugriffszeit zu einem beliebigen Sektor der Platte, also die Summe aus Suchzeit, Latenzzeit und Übertragungszeit, innerhalb vertretbarer Grenzen um einen Mittelwert schwankt, nennt man Platten auch Speicher mit **Direktzugriff**.

2.2.4.1 Maßnahmen zur Beschleunigung der Zugriffszeiten

Um die Suchzeit und möglichst auch die Latenzzeit zu verkürzen oder das Suchen ganz zu vermeiden, können Maßnahmen sowohl im Betriebssystem wie im Controller ergriffen werden. Im Folgenden werden vor allem diejenigen Optimierungen, die typischerweise im Controller realisiert werden, vorgestellt. Auf einige im Betriebssystem realisierte Maßnahmen werden wir später genauer eingehen.

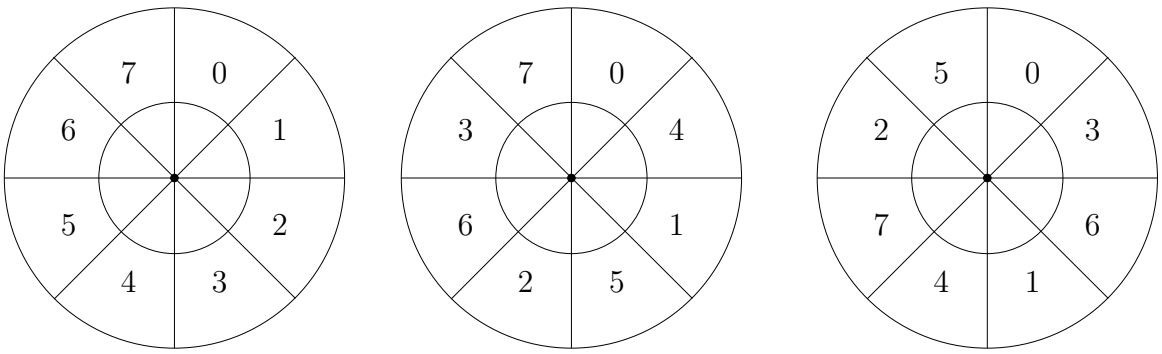
Verwaltung der Warteschlange der Übertragungsaufträge. Die Suchzeit ist in etwa linear zu der überbrückenden Distanz. Daher sollten Aufträge in Zylindern, die der aktuellen Position der Köpfe nahe liegen, bevorzugt werden. Als jeweils nächster sollte also derjenige Übertragungsauftrag ausgeführt werden, bei dem die kleinste Suchzeit auftritt (shortest-seek-time-first, **SSTF**).

Ein Problem bei SSTF ist, dass die Köpfe sich u. U. aus einem engen Bereich von Zylindern nicht mehr hinausbewegen, wenn dort laufend neue Aufträge abzuarbeiten sind, und dass Aufträge bzgl. weiter entfernt liegender Zylinder übermäßig lange warten müssen. Im schlimmsten Fall können manche Aufträge verhungern, d. h. sie werden nicht bearbeitet, wenn die neuen Aufträge zu jeder Zeit ankommen können. Abhilfe schafft die **SCAN**-Strategie, bei der die Köpfe immer abwechselnd nach außen und innen wandern, solange in der jeweiligen Richtung Aufträge vorliegen. Ähnlich funktionieren viele Aufzüge, deshalb wird der Algorithmus auch als Aufzugalgorithmus bezeichnet. Die mittlere Suchzeit ist bei SCAN zwar höher als bei SSTF, die mittlere Ausführungszeit eines Übertragungsauftrags incl. der Wartezeit bis zum Beginn der Ausführung ist hingegen kürzer.

Diese Optimierungen finden weitestgehend im Controller statt, d. h. das Betriebssystem ist nicht darin involviert und daher auch nicht abhängig davon. Vorteil hiervon ist, dass Platten bzw. Controller austauschbar sind, ohne dass das Betriebssystem geändert werden müsste.

Interleaving. Angenommen, man muss eine Folge von Blöcken, die z. B. zu einer Datei gehören, nacheinander in die Sektoren einer noch leeren Spur schreiben. Ein naheliegender Gedanke ist, den i-ten Block in den i-ten Sektor zu schreiben. Dies könnte aber zu folgendem unerwünschten Effekt führen: Beim späteren Lesen der Blöcke würde zuerst der 1. Block in den Puffer des Controllers übertragen. Von dort muss er über den Bus zum Hauptspeicher übertragen werden. Wenn die hierfür benötigte Zeit deutlich länger als die Zeit ist, in der der Lese-/Schreibkopf die Lücke zwischen zwei Sektoren überquert, dann befindet sich der Kopf schon über oder hinter dem 2. Sektor, wenn anschließend der 2. Sektor gelesen werden soll. Nun muss nahezu eine volle Umdrehung der Platte abgewartet werden. Dieser Effekt wiederholt sich bei allen folgenden Sektoren.

Um Abhilfe zu schaffen, überspringt man beim Schreiben jeweils einen oder mehrere Sektoren. Wieviele Sektoren übersprungen werden sollen, hängt vom Rechner ab; diese Anzahl, die man **interleave factor** nennt, muss beim Formatieren der Platte festgelegt werden. Bei 8 Sektoren in einer Spur und einem übersprungenen Sektor, also einem interleave factor 1, würde man z. B. 8 aufeinanderfolgende Blöcke in Position 0, 4, 1, 5, 2, 6, 3, 7 schreiben, siehe Abbildung 2.7.



nicht interleaved interleaved mit Faktor 1 interleaved mit Faktor 2

Abbildung 2.7: Interleaving.

Das Interleaving wird am besten im Controller durchgeführt. Das Betriebssystem schreibt also aus seiner Sicht in durchlaufend nummerierte Sektoren, der Controller setzt die Sektoradressen bzw. Sektornummern gemäß dem interleave factor um.

Puffer. Man kann im Controller Pufferbereiche für eine größere Anzahl von Blöcken vorsehen.

Diese können im Prinzip in gleicher Weise wie ein E/A-Puffer im Hauptspeicher benutzt werden. Dies ist nur dann sinnvoll, wenn kein großer E/A-Puffer im Hauptspeicher vorhanden ist, z. B. bei Rechnern, in denen der Hauptspeicher nicht sehr groß z. B. wegen Adresslängenbeschränkungen ausgebaut werden kann.

Daneben können diese Puffer dazu verwendet werden, um Sektoren im voraus zu lesen, d. h. wenn Sektor i in einer Spur gelesen wird, werden gleich noch die folgenden k Sektoren gelesen. Wenn k groß genug ist, kann auf das Interleaving verzichtet werden, so dass eine ganze Spur in einer einzigen Plattenumdrehung gelesen werden kann.

Vergabe von Sektoren. Da Dateien sehr häufig ganz gelesen werden, sollte man die zu einer Datei gehörigen Sektoren möglichst innerhalb einer Spur bzw. innerhalb eines Zylinders anordnen. Ist kein Zylinder mit ausreichend freiem Platz verfügbar, sollten die benutzten Zylinder möglichst nahe beieinander liegen.

2.3 Dateisysteme

2.3.1 Einführung

Wie schon in Abschnitt 2.1.3 erwähnt ist es normalerweise aus diversen Gründen nicht sinnvoll, Benutzer direkt auf einem Plattenspeicher d. h. auf der konventionellen Maschinenebene arbeiten zu lassen. Stattdessen möchten die Benutzer mit Dateien arbeiten. Eine Datei ist eine Sammlung von Informationen, die auf einem sekundären Speicher gespeichert sind. Ein **Dateisystem** ist eine Menge von Dateien und Verzeichnissen, die incl. der erforderlichen Hilfsdaten auf einem physischen oder logischen Datenträger gespeichert sind. Ein Dateisystem soll die Transparenz schaffen, dass dem Besitzer der Dateien die Details verborgen bleiben, er braucht nicht zu wissen, wo und wie die Dateien gespeichert sind und wie Datenträger arbeiten. Den Teil der E/A-Software, der die Operationen auf Dateisystemen realisiert, nennen wir auch **Dateiverwaltungssystem**. Grundlegende Merkmale von Dateisystemen sind aus der Sicht von Benutzern:

- man kann auf jede Datei direkt z. B. durch Angabe ihres Namens zugreifen;
- innerhalb gewisser Grenzen können beliebig viele Dateien unterschiedlicher Größe erzeugt werden;
- Dateien sind intern zusätzlich strukturiert, z. B. in Sätze. Diese Strukturen bzw. die damit möglichen Operationen nennt man **Zugriffsmethoden für Sätze bzw. Zeichen**.

Zur Realisierung von Dateisystemen kommen aufgrund dieser Eigenschaften nur Direktzugriffsspeicher in Frage. Derzeit sind dies im Endeffekt Plattenspeicher; hiervon werden wir auch im Folgenden ausgehen. Damit ist aber nicht ausgeschlossen, dass in Zukunft ganz andere Direktzugriffsspeicher hinzukommen, z. B. batteriegepufferte Halbleiterspeicher, bei denen sich u. U. neue Möglichkeiten zur Realisierung von konventionellen Dateisystemen wie auch Möglichkeiten zur Realisierung von Dateisystemen mit neuartigen Eigenschaften ergeben werden.

In diesem Abschnitt befassen wir uns mit folgenden Hauptthemen:

- Verwaltung der Menge aller Dateien eines Dateisystems und Realisierung von Dateinamen, siehe Abschnitt 2.3.2.

- Realisierung einzelner Dateien als Menge von Sektoren auf einer Platte und Verwaltung unbenutzter Sektoren, siehe Abschnitt 2.3.3.
- Realisierung von benutzerorientierten Zugriffsmethoden, siehe Abschnitt 2.3.4.

Dateisysteme und Platten. Die E/A-Software verwaltet i. A. mehrere physische oder logische Platten.

Bei den meisten Betriebssystemen muss eine einzelne Datei immer ganz auf einer Platte liegen; dies werden wir auch im Folgenden annehmen. Die Größe von Dateien ist dann begrenzt durch die Größe der Platten. Manche Betriebssysteme erlauben deshalb Dateien, die auf mehr als einer Platte gespeichert sind.

Bei manchen Betriebssystemen enthalten die Namen von Dateien den Namen der Platte, auf der die Datei gespeichert ist. Statt des Namens der Platte muss manchmal vor allem bei PC-Betriebssystemen und Disketten der Name des Laufwerks angegeben werden, in dem sich die Platte befindet. Der Benutzer muss also wissen, auf welcher Platte eine gesuchte Datei gespeichert ist. Andere Betriebssysteme erfordern dies nicht.

Zugriffsschutz. Einen wichtigen Aspekt der Dateiverwaltung, nämlich den Zugriffsschutz, behandeln wir erst in Kurseinheit 6.

2.3.2 Dateiverzeichnisse

Informationen über die Dateien in einem Dateisystem werden in **Dateiverzeichnissen** (**Katalogen**, **Ordnern**, **directories**) verwaltet. Verzeichnisse sind meistens wieder Dateien, die aber als Verzeichnis speziell gekennzeichnet sind, so dass der Inhalt durch die Operationen, die auf *normalen* Dateien arbeiten, nicht verändert werden darf.

In manchen älteren oder auf kleinere Rechner ausgerichteten Betriebssystemen wurde ein einziges Verzeichnis auf jedem Datenträger oder nur ein Verzeichnis pro Benutzer benutzt. Standard sind heute allerdings **geschachtelte Verzeichnisse** (auch **hierarchische Dateisysteme** genannt), siehe Abbildung 2.8.

Ein Verzeichnis enthält zu jeder Datei des Verzeichnisses den Namen der Datei, z. B. *prog.c* für ein C-Programm, *prog.java* für ein Java-Programm, *info.pdf* für eine PDF-Datei. Außerdem gibt es eine Liste von zusätzlichen Informationen, den sogenannten **Attributen** der Datei:

- *Aktuelle Größe der Datei* in Byte, Wörtern oder Blöcken.
- *Besitzer*, der die Kontrolle über diese Datei hat. Der Besitzer kann anderen Benutzern Zugriffsrechte erteilen, verweigern oder ändern.
- *Zugriffsrechte*, die zeigen, wer auf die Datei zugreifen darf und wer nicht.
- *Datum der Erzeugung der Datei*, die zum ersten mal im Verzeichnis gespeichert worden ist.
- *Datum des letzten lesenden Zugriffs.*
- *Datum des letzten schreibenden Zugriffs.*
- *Typ der Datei* (anwendbare Zugriffsmethode).

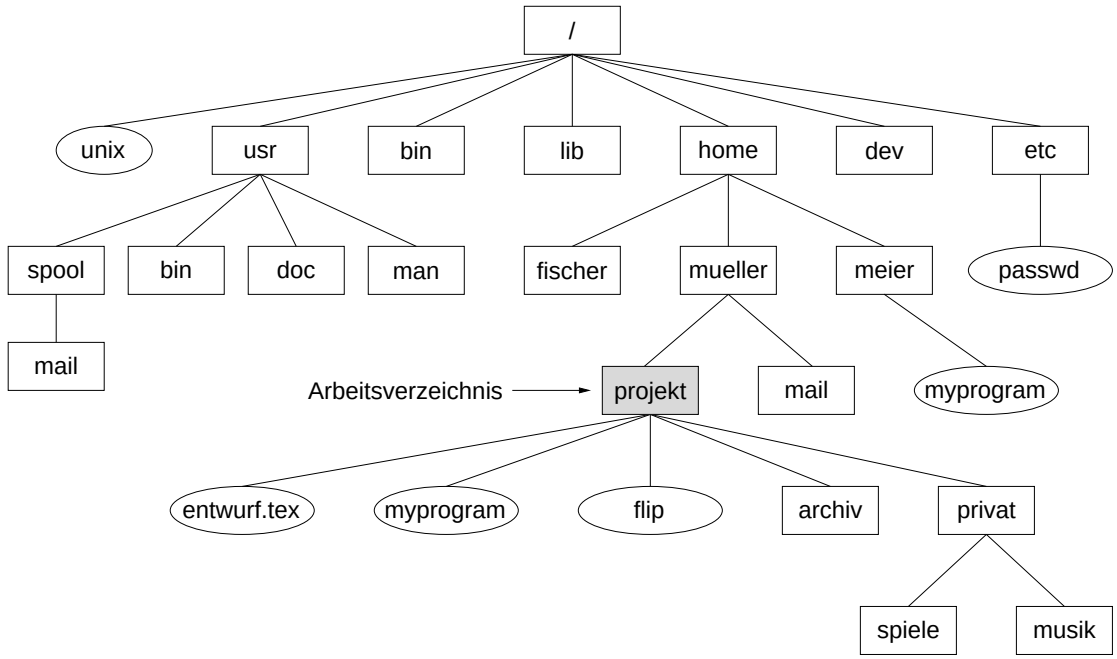


Abbildung 2.8: Ein Ausschnitt aus einer Verzeichnisstruktur unter UNIX.

Die in einem Verzeichnis auftretenden Namen von Dateien sind zunächst nur *lokale Namen*, d. h. dieser Name identifiziert die Datei nur innerhalb dieses Dateiverzeichnisses.

Die Struktur der Verzeichnisse ist normalerweise ein Baum. Das **Wurzelverzeichnis** bildet den Einstiegspunkt in das ganze Dateisystem. Von dort aus kann man zu anderen Verzeichnissen oder Dateien durch Angabe von deren lokalen Namen **navigieren**. In einem vorhandenen Verzeichnis können beliebige weitere Verzeichnisse angelegt werden.

Zur Identifikation von Dateien (incl. Verzeichnissen) verwendet man (**absolute**) **Pfadnamen**. Diese bestehen aus der Konkatenation der auf dem Weg zu der Datei auftretenden lokalen Namen, wobei die einzelnen Namen durch ein Trennzeichen wie `/` (slash bei UNIX) oder `\` (backslash bei Windows) getrennt werden, z. B. der vollständige Name der Datei `myprogram` von Benutzer Meier ist `/home/meier/myprogram`, während `/home/mueller/projekt/myprogram` der Name der Datei `myprogram` von Benutzer Müller ist.

Beispielsweise bezeichnet `/usr/spool/mail` in vielen UNIX-Systemen ein Verzeichnis, in dem die Mailboxen für die E-Mails aller Benutzer enthalten sind. In diesem Fall tritt im Wurzelverzeichnis „`/`“ ein Eintrag mit Namen `usr` auf, im Verzeichnis `/usr` ein Eintrag mit Namen `spool` und im Verzeichnis `/usr/spool` ein Eintrag mit Namen `mail`.

Da die Pfadnamen in der Praxis recht lang werden, andererseits aber Benutzer fast immer lokal in wenigen Verzeichnissen und Unterverzeichnissen arbeiten, werden üblicherweise auch **relative Pfadnamen** unterstützt. Hierzu wird ein Verzeichnis als **Arbeitsverzeichnis** festgelegt (mittels eines entsprechenden Kommandos *cd*); dieses dient als Ausgangspunkt für die Navigation bei relativen Pfadnamen (vgl. Kurseinheit 7), z. B. bei jeder Anmeldung ist das Arbeitsverzeichnis für Müller das Verzeichnis `mueller`. Mit dem Kommando `cd projekt` kann Müller in das Arbeitsverzeichnis `projekt` gehen, und er kann dort einfach seine Datei durch `myprogram`

ansprechen.

2.3.3 Zugriffsmethoden für Seiten

Eine erste Frage beim Entwurf von Dateisystemen ist, wie denn Dateien – die ja verschiedene Größen haben und wachsen oder schrumpfen können – auf Plateninhalte abgebildet werden. Dieses Problem ist ganz ähnlich dem Problem, einen realen Hauptspeicher auf mehrere Prozesse zu verteilen, womit wir uns in Kurseinheit 4 befassen werden.

Um den Inhalt einer Datei den Sektoren einer Platte zuordnen zu können, wird er in **(Datei-) Seiten** aufgeteilt. Wie dies geschieht, werden wir später diskutieren. Eine Datei entspricht somit einer Folge von Seiten.

Die Seiten einer Datei müssen nun auf die Sektoren der Platte bzw. die darin gespeicherten Blöcke abgebildet werden. Es liegt nahe, die Größe von Seiten und Blöcken identisch zu wählen, also je eine Seite in einem Block zu speichern. Man kann aber auch die Seiten der Dateien z. B. doppelt oder halb so groß wie die Blöcke wählen. Dies führt zu einigen Detailproblemen, die wir hier nicht diskutieren wollen. Wir nehmen deshalb im Folgenden an, dass Seiten und Blöcke gleich groß sind. Man beachte aber, dass *Seite* und *Block* dennoch verschiedene Begriffe sind: Ein Block ist der Inhalt eines Sektors, der zwischen Hauptspeicher und Datenträger übertragen werden kann; dieser Begriff ist völlig unabhängig davon, ob auf dem Datenträger ein Dateisystem realisiert wird oder nicht. Eine Seite ist ein Teil einer Datei und hat eine Position innerhalb der Datei.

Zur Strukturierung und Vereinfachung der weiteren Diskussion teilen wir bzgl. der Operationen auf Dateien die geräteunabhängige E/A-Software in Abbildung 2.3 in zwei weitere Schichten auf, siehe Abbildung 2.9.

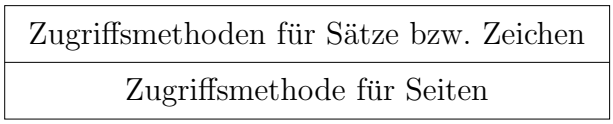


Abbildung 2.9: Zugriffsmethoden.

Die Zugriffsmethode für Seiten realisiert eine Datei als eine *Folge von Seiten*, die von 0 an fortlaufend durchnummeriert sind. Eine Seite ist hier ein Byte-Feld fester Größe und hat keine darüberhinausgehende interne Struktur. Die wesentlichen Operationen, die auf dieser Ebene bereitgestellt werden, sind:

- Erzeugen, Kopieren und Löschen einer Datei
- Öffnen und Schließen einer Datei
- Anlegen von n zusätzlichen bzw. Freigeben der letzten n Seiten
- Angeben der Nummer der letzten angelegten Seite
- *direktes* Lesen bzw. Schreiben der Seite Nr. i
- Kopieren einer Seite
- Vertauschen der Seiten Nr. i und j

Eine Zugriffsmethode für Seiten ist in realen Betriebssystemen nicht immer explizit vorhanden oder für nichtprivilegierte Benutzer benutzbar. Sie ermöglicht es uns aber, Fragen wie die Realisierung von Sätzen auf Seiten und die Verwaltung freier Sektoren getrennt zu diskutieren. Außerdem wird sie oft von Datenbanksystemen benutzt, die aus Performance-Gründen das Dateisystem nicht benutzen und stattdessen *direkt auf der Platte* arbeiten wollen. Vorteile bietet auch die Implementierung von Zugriffsmethoden für Sätze bzw. Zeichen auf Basis einer Zugriffsmethode für Seiten beim Kopieren von Dateien: Dateien können dann seitenweise kopiert werden. Würde eine Zugriffsmethode für Sätze bzw. Zeichen in ihren Datenstrukturen Sektoradressen benutzen, müssten diese Sektoradressen in der Kopie korrigiert werden.

Unterhalb der Zugriffsmethode für Seiten besteht eine Datei aus mehreren Sektoren, die die Dateiseiten enthalten, und zugehörigen Verwaltungsdaten, die in anderen Sektoren gespeichert werden. Vielfach sind die Verwaltungsdaten zentral für alle Dateien innerhalb eines Dateisystems zusammengefasst. Die wesentlichen Aspekte bei der Realisierung einer Zugriffsmethode für Seiten, die wir im Folgenden diskutieren wollen, sind:

- Verwaltung der Sektoren einer Datei
- Strategie bei Auswahl von Sektoren beim Anlegen zusätzlicher Seiten
- Verwaltung der freien Sektoren in einem Dateisystem

2.3.3.1 Verwaltung der Sektoren einer Datei

Hierfür werden im Folgenden drei wichtige Techniken vorgestellt.

File allocation table. Eine **file allocation table** (**FAT**) ist eine *zentrale* Datenstruktur, die Informationen über *alle* Dateien eines Dateisystems sowie über die freien Blöcke enthält. Sie besteht aus einem Array, in dem jeder Sektor der Platte durch einen Eintrag repräsentiert wird. Wir nehmen hier an, dass alle Sektoren der Platte von 0 an fortlaufend durchnummeriert sind. Die zu einer Datei gehörigen Sektoren werden in der FAT linear verkettet, d. h. wenn z. B. eine Datei `spiel` in den Sektoren 3, 27 und 19 gespeichert ist, siehe Abbildung 2.10, dann enthält

- Eintrag 3 der FAT die Angabe „27“, also die Nummer des Folgesektors,
- Eintrag 27 die Angabe „19“ und
- Eintrag 19 die Angabe „nil“, die anzeigt, dass dies der letzte Block der Datei ist.

Freie bzw. defekte Blöcke werden in der FAT als solche markiert.

Die FAT wird in einem oder mehreren fest vereinbarten Sektoren der Platte gespeichert. Bei Starten des Rechners wird sie in einen Puffer im Hauptspeicher übertragen und bleibt dort permanent.

Sequentielles Verarbeiten einer Datei wird durch die FAT sehr gut unterstützt. Direktzugriff zum i-ten Sektor einer Datei erfordert das Durchlaufen der linearen Liste; die hierfür erforderliche Zeit übersteigt aber nur bei einer langsamen CPU und großen Dateien die Zeit für einen Plattenzugriff.

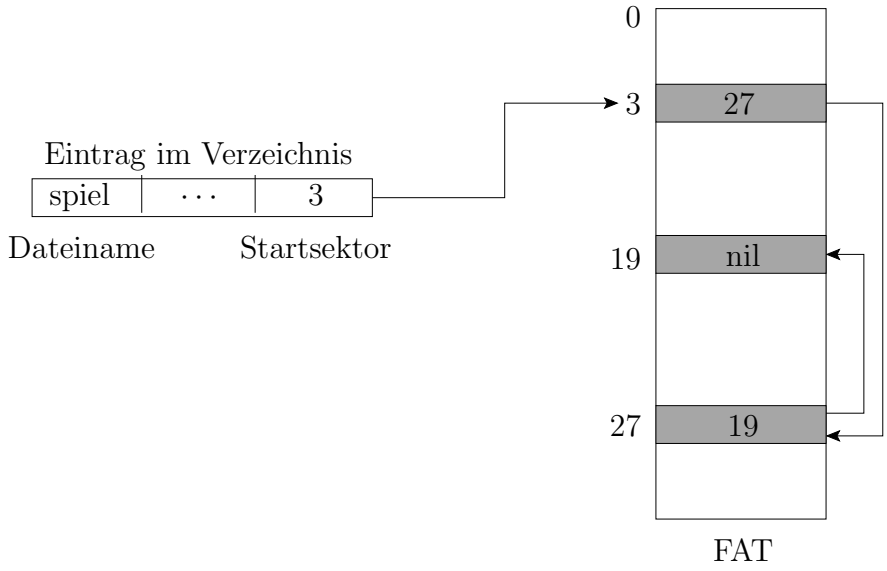


Abbildung 2.10: File allocation table.

Der große Nachteil der FAT ist, dass sich die gesamte Tabelle zu jeder Laufzeit des Rechners im Hauptspeicher befinden muss. Bei einer Platte mit z. B. 20 GByte und Blockgröße 1 kByte benötigt man schon $20 \cdot 2^{20}$, also mehr als 20 Millionen Einträge. Obwohl ein einzelner Eintrag der FAT nur etwa 4 Byte benötigt, beträgt der Platzbedarf im Hauptspeicher schon 80 MByte.

In Dateiverzeichnissen kann die Menge der Sektoren, die zu einer Datei gehören, durch die Nummer des ersten Sektors angegeben werden, siehe auch Abbildung 2.10.

Sektoradresstabellen (i-nodes). Die Alternative zu der *zentralen* Verwaltung aller Dateien in einer FAT ist, *dezentral* für *jede Datei* eine eigene **Sektoradresstabelle** zu halten, die in Position i die Adresse des Sektors speichert, der die i-te Seite dieser Datei enthält. Statt Sektoradressen kann man auch Sektornummern verwenden. Sektornummern benötigen nur geringfügig weniger Platz, insb. wenn für die 3 Nummern, aus denen eine Sektoradresse besteht, immer nur die gerade ausreichende Zahl von Bits benutzt wird. Der Unterschied fällt wegen der Aufrundung auf ganze Bytes u. U. gar nicht ins Gewicht. Bei Sektoradressen kann man leichter kontrollieren, wann eine Spur verlassen wird, und man kann die Sektoren einer Spur leichter als zyklisch hintereinanderliegend behandeln.

Die Sektoradresstabelle kann allerdings bei großen Dateien selbst wiederum relativ groß werden. Die Sektoradresstabellen aller Dateien müssen natürlich auch auf der Platte gespeichert und die für sie verwendeten Sektoren müssen verwaltet werden. Wir unterscheiden daher zwischen **Dateiseiten-Sektoren** (kurz: **D-Sektoren**), die Seiten von Dateien enthalten, und anderen Sektoren, die z. B. Teile von Sektoradresstabellen enthalten.

Die Verwaltung der Sektoradresstabellen im Betriebssystem UNIX, die anschließend vorgestellt wird, ist auf folgende Randbedingungen ausgerichtet:

- die meisten Dateien sind relativ klein (unter 1 kByte) bzw. nur wenige Seiten groß;
- die maximale zulässige Dateigröße sollte wenigstens 100 Mbyte betragen;

- Dateien von über 10 MByte sind sehr selten;
- oft wird nur der Anfang der Datei gelesen.

Sei X die Anzahl der Sektoradressen, die in einem Block gespeichert werden können. Bei 4 Byte langen Sektoradressen und 1 kByte langen Blöcken ist $X = \frac{1\text{ kByte}}{4\text{ Byte}} = \frac{2^{10}}{2^2} = 2^8 = 256$. Die Sektoradresstabelle einer Datei wird nun in 4 Abschnitte der Länge 10, X, X^2 und X^3 aufgeteilt, also z. B. bei $X = 256$ der Länge 10, 256, 65.536 und 16.777.216. Sektoradresstabellen mit mehr als 16.843.018 Einträgen, die einer Datei von rund 16 GByte entsprechen, sind nicht möglich.

Zu *jeder Datei* existiert ein sogenannter **i-node** (**Index-Node**); das ist eine Tabelle, die Angaben über die Datei enthält. Unter anderem stehen dort, siehe Abbildung 2.11:

- 10 **direkte** Sektoradressen. Dies sind Adressen von D-Sektoren, und zwar die Einträge 0 - 9 der Sektoradresstabelle. Wenn also der i-node einer Datei im Hauptspeicher vorhanden ist, dann kann auf die ersten 10 Seiten *direkt*[5] zugegriffen werden.

- eine **indirekte** Sektoradresse. Der hier angegebene Sektor enthält X direkte Sektoradressen, entsprechend den Einträgen 10 bis $9 + X$ der Sektoradresstabelle.

- eine **doppelt indirekte** Sektoradresse. Der hier angegebene Sektor enthält X indirekte Sektoradressen, entsprechend den Einträgen $10 + X$ bis $9 + X + X^2$ der Sektoradresstabelle.

- eine **dreifach indirekte** Sektoradresse. Der hier angegebene Sektor enthält X doppelt indirekte Sektoradressen, entsprechend den Einträgen $10 + X + X^2$ bis $9 + X + X^2 + X^3$ der Sektoradresstabelle.

Es werden natürlich nur so viele Sektoren reserviert, wie tatsächlich für eine Datei benötigt werden. Bei einer Datei mit 6 Seiten haben bspw. die letzten 4 direkten und alle indirekten Sektoradressen im i-node den Wert *undefiniert*, d. h. es wird dort ein Wert angegeben, der keine gültige Sektoradresse darstellt und der vom System so interpretiert wird, dass für diese Seite kein Sektor angelegt ist. Bei einer Datei mit 30 Seiten sind alle 10 direkten Sektoradressen definiert, ebenfalls die einfach indirekte Adresse; der dort angegebene Sektor enthält 20 definierte und (bei $X = 256$) 236 undefinierte Sektoradressen.

Jeder i-node enthält auch einige Attribute. Die Attribute umfassen Schutzinformationen der Datei, Besitzer, die Zeiten der Erzeugung, des letzten Zugriffs und der letzten Veränderung, die Größe und die Anzahl der Links. Durch Links (auch *hard links* genannt) kann eine Datei in mehreren Verzeichnissen vorkommen. Erst beim Löschen des letzten Links wird der Inhalt wirklich gelöscht.

In Abbildung 2.12 ist ein klassisches UNIX-Dateisystem innerhalb einer Partition einer Festplatte dargestellt. Block 0 ist der Bootblock, dann folgen:

[5] *Direkt* bedeutet hier, dass auf D-Seiten zugegriffen werden kann, ohne vorher Sektoren mit Verwaltungsdaten lesen zu müssen. Natürlich kann auch zu Sektoren, die durch eine *indirekte* Adresse angegeben werden, *direkt* (im üblichen Sinn) zugegriffen werden.

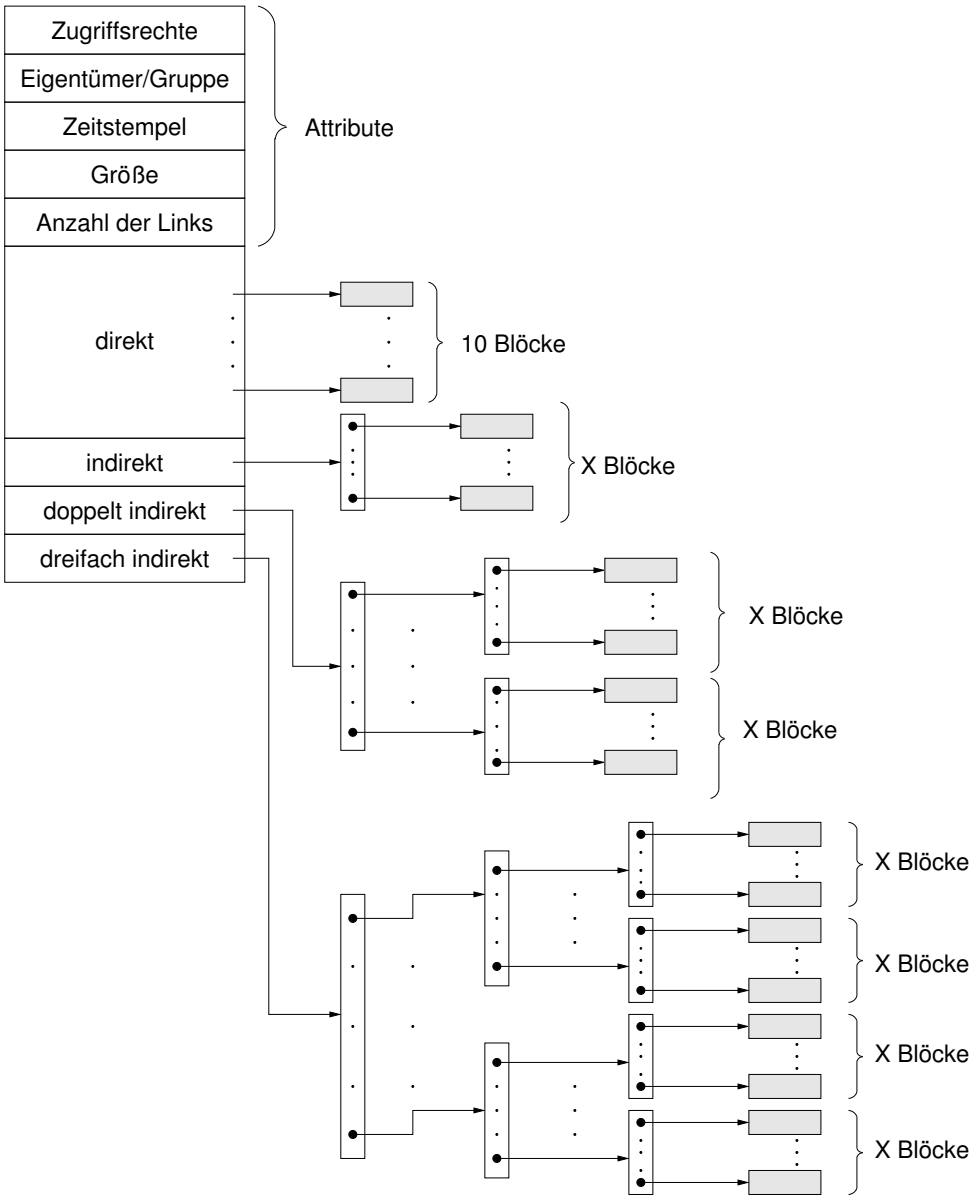


Abbildung 2.11: Ein i-node für eine Datei sowie deren Daten unter UNIX.

- Superblock (Block 1), der wichtige Informationen des Dateisystems (die Größe des Dateisystems, die Größe der i-node-Liste, die Anzahl der freien i-nodes, die Liste und die Anzahl der freien Blöcke, siehe Abschnitt 2.3.3.2) enthält und als Erstes in den Hauptspeicher geladen wird, wenn der Rechner gebootet oder auf das Dateisystem zum ersten Mal zugegriffen wird.

- Die i-nodes-Liste, in der die i-nodes von 1 bis zu einem Maximum nummeriert sind. Jeder i-node hat eine feste Länge und beschreibt genau eine Datei.

- Datenblöcke, alle Dateien und Verzeichnisse werden hier gespeichert.

Jedes Verzeichnis wird genau wie eine normale Datei behandelt. In einem Verzeichnis gibt es eine Liste von Verzeichniseinträgen und jeder Eintrag enthält einen Dateinamen und die Nummer des i-nodes dieser Datei, siehe Abbildung 2.13. Da die Attribute einer Datei in ihrem i-node gespeichert werden, sind die Verzeich-

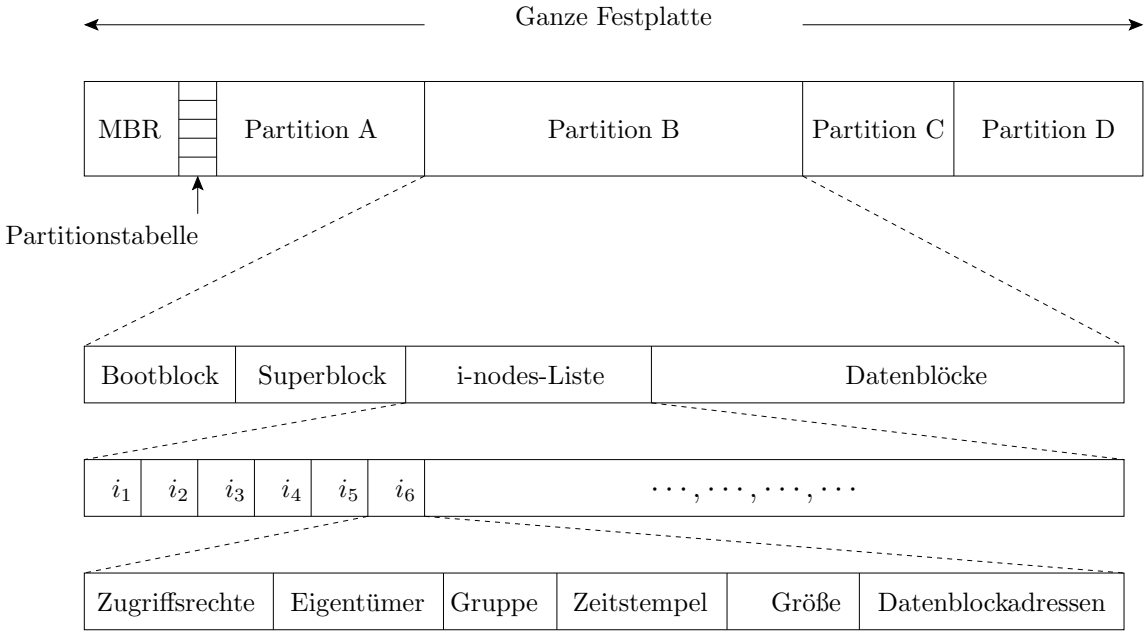


Abbildung 2.12: Realisierung eines klassischen UNIX-Dateisystems auf Festplatten.

niseinträge hier kürzer als bei FAT-Dateisystemen, bei denen die Speicherung der Attribute in den Verzeichniseinträgen erfolgt.

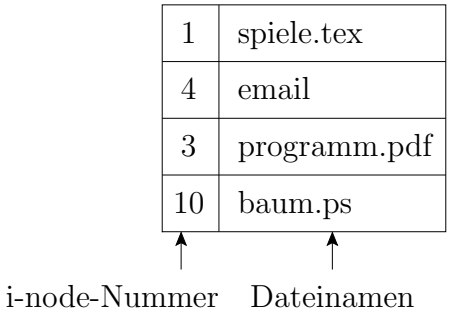
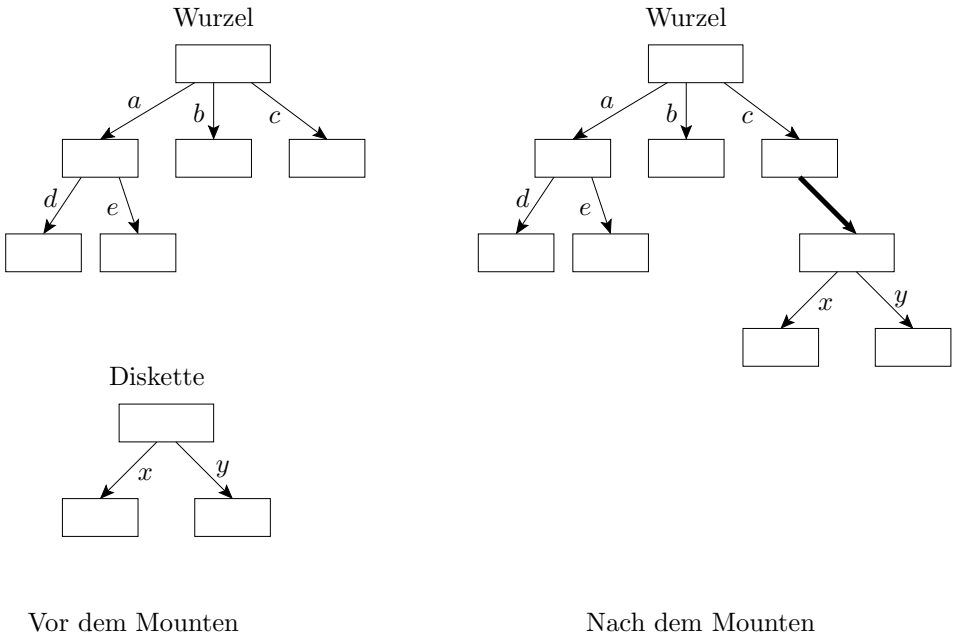


Abbildung 2.13: Ein UNIX-Verzeichnis, jeder Eintrag bezieht sich auf einen i-node.

Wir wollen z. B. die Datei `/home/meier/myprogram` in Abbildung 2.8 finden. Das Dateisystem lokalisiert zuerst das Wurzelverzeichnis. Im Wurzelverzeichnis vergleicht das System jeden Eintrag mit dem gesuchten Dateiname `home` und findet die Nummer des i-nodes `/home`. Unter Verwendung der Nummer des i-nodes werden Speicherblöcke des i-nodes auf der Platte gefunden und in den Hauptspeicher geholt. Danach werden aus dem i-node die Plattenblöcke extrahiert, damit die Suche nach dem Name `meier` ermöglicht wird. Wenn die Nummer des i-nodes `/home/meier` gefunden wird, kann dieser i-node wieder in den Hauptspeicher geholt und die Plattenblöcke gelesen werden. Schließlich wird die Nummer des i-nodes `/home/meier/myprogram` verwendet und diese Datei mit Plattenblöcke lokalisiert.

In den meisten Dateisystemen ist jede einzelne Partition, Platte, CD-ROM oder Diskette in sich abgeschlossen, sie enthält ein eigenes Wurzelverzeichnis und alle von der Wurzel erreichbaren Unterverzeichnisse und Dateien müssen sich auf demselben Speichermedium befinden. Für den Zugang zu allen Festplatten und entfernbaren Medien bietet UNIX das Konzept des *mounting*. UNIX erlaubt keine Bezie-

hung von Datenträgern mit Laufwerksbuchstaben wie bei Windows. Statt dessen bewirkt der *mount*-Systemaufruf, dass das Dateisystem der anderen Datenträger in die Verzeichnishierarchie des Computers aufgenommen wird, und zwar an jeder Stelle, die das Programm wünscht, siehe Abbildung 2.14.



Vor dem Mounten Nach dem Mounten

Abbildung 2.14: Vor dem Mounten konnten die Dateien auf der Diskette nicht benutzt werden. Nach dem Mounten gehören sie zur Dateiverzeichnishierarchie.

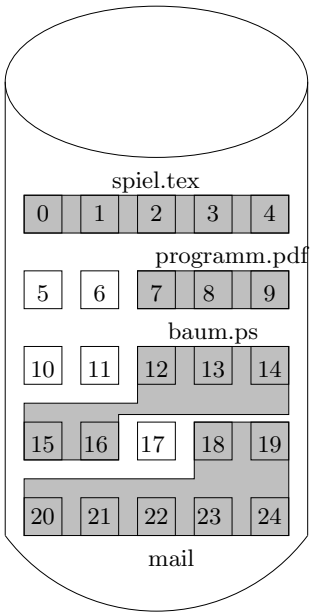
Sektorfolgen. Wie schon in Abschnitt 2.3.3 sollte man die Sektoren, die zu einer Datei gehören, zur Vermeidung von Plattenarmbewegungen möglichst innerhalb einer Spur bzw. eines Zylinders anordnen, am besten sogar als Folge hintereinanderliegender Sektoren, d. h. Sektoren mit fortlaufenden Nummern; vergleiche Abschnitt 2.2.4.1 über interleaving. Bei großen Platten, die nicht allzu voll sind, können erfahrungsgemäß fast alle kleinen bis mittelgroßen Dateien in 1 oder 2 Sektorfolgen untergebracht werden, die Länge der Sektorfolgen liegt typischerweise im Bereich von 5 bis 20.

Unter diesen Annahmen ist es deutlich platzsparender, statt einer Sektoradressstabelle eine Sektorfolgen-Tabelle zu verwenden, die zu jeder Sektorfolge die Adresse des ersten Sektors und die Länge der Folge enthält, siehe Abbildung 2.15. Innerhalb der Sektorfolgen liegen die Dateiseiten mit aufsteigend sortierten Nummern, d. h. wenn der erste Sektor der Folge mit der Länge n die Seite Nr. k enthält, dann folgen die Seiten mit Nummer $k+1$, $k+2$, ..., $k+n-1$.

2.3.3.2 Verwaltung freier Sektoren

Für die Speicherung der Menge der freien Sektoren sind zwei Verfahren üblich:

1. eine Tabelle mit Sektoradressen oder Sektorfolgen: man kann hier die freien Sektoren als eine weitere Datei ansehen. Die Sektoren mit den Adressen der freien Sektoren bzw. Sektorfolgen werden hier allerdings linear verkettet, da eine sequentielle Zugriffsmöglichkeit ausreicht.



Verzeichnis

Dateien	Start	Länge
baum.ps	12	5
mail	18	7
programm.pdf	7	3
spiel.tex	0	5

Abbildung 2.15: Realisierung des Dateisystems durch Sektorfolgen der Festplatte.

2. ein Bitfeld, in dem für jeden Sektor angegeben wird, ob er frei ist oder nicht. Verwendet man eine FAT, so kann man das Bit, das anzeigt, ob ein Sektor frei ist, in den Einträgen der FAT unterbringen, sofern dort noch Platz ist.

Das Bitfeld benötigt meist deutlich weniger Platz. Beide Verfahren benötigen allerdings meist mehr als einen Block zur Speicherung der Hilfsdaten. Bei der (unsortierten) Sektoradresstabelle, auf der man i. W. einen Keller (stack) von Adressen realisiert, reicht es völlig aus, wenn nur ein einziger, nämlich der letzte dieser Blöcke im Hauptspeicher gehalten wird; die anderen Blöcke brauchen nur auf Platte vorhanden zu sein. Das Bitfeld sollte stets komplett im Hauptspeicher gehalten werden, denn durch Sektorfreigaben kann jederzeit an beliebiger Stelle im Bitfeld eine Änderung erforderlich sein.

2.3.3.3 Maßnahmen zur Beschleunigung von Zugriffen

E/A-Puffer. Wie schon oben erwähnt werden Sektornutzinhalte beim Lesen zunächst ohnehin in einen Puffer im Hauptspeicher übertragen. Wenn der Prozess, der diesen Sektor gelesen hat, einen anderen Sektor liest, könnte man wieder den gleichen Puffer verwenden, da es relativ unwahrscheinlich ist, dass derselbe Prozess später noch einmal den gleichen Sektor liest. Allerdings kann ein anderer Prozess diesen Sektorinhalt vielleicht anschließend benötigen. Dies trifft besonders für Sektoren zu, die zu häufig benutzten Dateien, die z. B. Systemdaten oder häufig benutzte Progamme enthalten, gehören. Aufgrund dieser Beobachtung wird oft ein größerer Bereich des Hauptspeichers (einige 100 oder 1000 kByte) als E/A-Puffer reserviert. In manchen Betriebssystemen wird dieser Bereich statisch fixiert, in anderen passt er sich an die jeweilige Last auf dem Rechner an. Für das Ersetzen von Sektorinhalten in diesen Puffern gelten die gleichen Überlegungen wie für das Ersetzen von Seiten eines virtuellen Speichers; dieses Thema werden wir in Kurseinheit 4 behandeln, so dass wir hier auf diese Strategien nicht näher eingehen.

Asynchrones Schreiben. Beim Schreiben einer Seite ist zu unterscheiden, ob das Benutzerprogramm sicher sein möchte, dass sich nach Beendigung der von ihm aufgerufenen Schreiboperation die Daten tatsächlich auf der Platte, also einem permanenten Speicher, befinden. Falls ja, dann muss die Beendigung der Übertragung zur Platte abgewartet werden (synchrones Schreiben). Falls nein, dann können die Daten zunächst in den E/A-Puffer geschrieben werden und von dort aus asynchron zur Platte übertragen werden. Die Schreiboperation führt also nicht mehr zu einer Blockierung des aufrufenden Prozesses. Die Schreiboperation für Seiten sollte also einen Parameter haben, durch den zwischen beiden Alternativen gewählt werden kann.

Optimierung von Sektorfolgen. Wie schon oben erwähnt sollten Dateien möglichst nicht in verstreuten Sektoren, sondern in nahe beieinanderliegenden Sektorfolgen angelegt werden. Um dies zu erreichen, sind verschiedene Maßnahmen denkbar:

1. Wenn eine Datei hinten um einen Sektor verlängert wird, dann kann die letzte Sektorfolge verlängert werden, wenn der Sektor dahinter frei ist. Ein naheliegender Gedanke ist daher, derartige Sektoren möglichst freizuhalten.

   Diese Idee wird in manchen Betriebssystemen (z. B. in BS2000) so realisiert, dass dann, wenn eine Datei um einen Sektor verlängert wird, immer gleich k Sektoren reserviert werden. $k-1$ Sektoren bleiben zunächst unbenutzt und werden bei den folgenden $k-1$ Sektoranforderungen aufgebraucht. Die Zahl k wird vom Eigentümer der Datei festgelegt. Hier sollte ein zusätzliches Kommando verfügbar sein, mit dem die unbenutzten reservierten Seiten freigegeben werden können.

2. Eine optimale Anlage der Sektorfolgen ist - vor allem bei relativ voller Platte – sehr aufwändig, weil u. U. sogar einzelne Felder verschoben werden müssen. Andererseits müssen Anforderungen nach freien Sektoren sehr schnell bearbeitet werden.

   Eine Lösung des Problems besteht darin, bei der Vergabe eher eine schnelle Entscheidung zu treffen. Der dadurch eintretende *Schaden* in Form von ungünstig verteilten Sektoren wird durch eine **Reorganisation der Platte** wieder behoben.

   Die Reorganisation kann durch einen Systemverwalter von Hand gestartet werden. Eleganter ist ein mit niedriger Priorität laufender Hintergrundprozess, der laufend das Dateisystem auf derartige Mängel hin untersucht und diese dann behebt.

2.3.4 Zugriffsmethoden für Zeichen und Sätze

Hauptmerkmal dieser Zugriffsmethoden ist, dass die Applikationen Sätze bzw. Zeichen lesen und schreiben, nicht Seiten. Diese Zugriffsmethoden simulieren sozusagen die Speichereinheit Satz bzw. Zeichen auf der Speichereinheit Seite. Aus Sicht der Benutzer besteht eine Datei nunmehr aus einer Menge von Sätzen bzw. Zeichen und einer Zugriffsstruktur. Eine **Zugriffsstruktur** bestimmt, wie einzelne Speichereinheiten identifiziert und lokalisiert werden können und wie die Menge der Speichereinheiten der Datei verändert werden kann. Eine Zugriffsstruktur kann man auch als generischen abstrakten Datentyp ansehen, worin der Parametertyp die Speicherein-

heit ist. Eine konkrete Zugriffsmethode ist also ein abstrakter Datentyp, der aus einer generischen Zugriffsstruktur durch Wahl einer bestimmten Speichereinheit entsteht.

2.3.4.1 Zugriffsstrukturen

Bevor wir die Zugriffsmethoden diskutieren, wollen wir die darin auftretenden Zugriffsstrukturen vorstellen. Unter einer **Speichereinheit** verstehen wir im Folgenden einen Satz oder ein Zeichen.

Sequentielle Zugriffsstruktur: Diese Zugriffsstruktur entspricht einer einfach verketteten Liste von Speichereinheiten. Die wesentlichen Operationen sind:

- beim Schreiben von Dateien:
  - Erzeugen einer leeren Datei (incl. Öffnen)
  - Anhängen einer Speichereinheit (aus einem Puffer)
  - Schließen der Datei
- beim Lesen von Dateien:
  - Öffnen einer Datei
  - Abfrage, ob Dateiende erreicht
  - Kopieren der nächsten Speichereinheit in einen Puffer
  - Schließen der Datei

Nicht möglich sind der direkte Zugriff zur i-ten Speichereinheit der Datei oder das Einfügen einer Speichereinheit zwischen zwei vorhandenen Speichereinheiten.

Statische Direktzugriffsstruktur: Diese Zugriffsstruktur entspricht einem hinten erweiterbaren Array. Die wesentlichen Operationen sind die der sequentiellen Zugriffsstruktur sowie zusätzlich Operationen, die *direkt*, d. h. in ungefähr konstanter Zeit, eine durch ihre *Nummer* identifizierte Speichereinheit lesen oder schreiben. Nicht möglich ist das Einfügen einer Speichereinheit zwischen zwei vorhandenen Speichereinheiten. Zugriffsmethoden für Seiten realisieren i. W. eine statische Direktzugriffsstruktur.

Dynamische Direktzugriffsstruktur bzw. Indexsequentielle Direktzugriffsstruktur: Wir nehmen hier eine linear geordnete Menge von *Schlüsselwerten* für Sätze an, bei dieser Zugriffsstruktur treten nur Sätze als Speichereinheit auf. Ein Schlüsselwert identifiziert entweder keinen oder genau einen Satz in der Datei.

Die wesentlichen Operationen sind die der sequentiellen Zugriffsstruktur, wobei die Sätze in der Reihenfolge ihrer Schlüssel gelesen werden und wobei die Operation *Schreiben des nächsten Satzes* entfällt, sowie zusätzlich Operationen, die *direkt*, d. h. in ungefähr konstanter Zeit, einen Satz mit einem vorgegebenen Schlüsselwert erzeugen, löschen, lesen oder schreiben.

2.3.4.2 Zugriffsmethoden für Zeichen

Bei Zugriffsmethoden für Zeichen kann man eine Datei als einen linearen Adressraum wie bei einem Array ansehen. Eine Zugriffsmethode für Zeichen realisiert eine statische Direktzugriffsstruktur. Die Realisierung auf Basis einer Zugriffsmethode für Seiten ist offensichtlich.

Die letzte Seite der Datei wird i. A. nicht vollständig genutzt. Um das letzte Zeichen der Datei erkennen zu können, wird die Länge der Datei in Bytes (und nicht etwa in Blöcken) innerhalb der Dateiattribute im Dateiverzeichnis gespeichert.

In vielen Betriebssystemen für PCs und in UNIX wird standardmäßig nur eine Zugriffsmethode für Zeichen auf der Ebene *Betriebssystem* angeboten. Zugriffsmethoden für Sätze werden nicht angeboten und insb. nicht innerhalb des Betriebssystems benutzt. Sie können auf Basis der Zugriffsmethode für Zeichen implementiert werden (s. u.) und stehen dann normalerweise in Form einer Bibliothek zu Verfügung.

2.3.4.3 Zugriffsmethoden für Sätze

Die Varianten der Zugriffsmethoden für Sätze unterscheiden sich in folgenden Punkten:

- Inhalt und Struktur eines Satzes: meist ein Text (bzw. Byte-Feld), manchmal ein beliebiger Datentyp.
- feste/variable Satzlänge, d. h. alle Sätze der Datei haben gleiche/unterschiedliche Längen.

Die zugehörigen Angaben, z. B. die feste Satzlänge oder die maximale Satzlänge bei variabler Satzlänge, werden innerhalb der Dateidattribute gespeichert.

Die Realisierung einer Zugriffsmethode für feste Satzlänge ist relativ einfach und wird hier nicht weiter diskutiert, zumal solche Zugriffsmethoden inzwischen weniger bedeutend sind. Bei Zugriffsmethoden für variable Satzlänge sind nur die sequentielle Zugriffsstruktur und die dynamische Direktzugriffsstruktur üblich. Die Realisierung der statischen Direktzugriffsstruktur wird daher hier nicht diskutiert. Manche Implementierungen von Zugriffsmethoden für Sätze lassen sich auf Basis eines (simulierten) linearen Adressraums, wie er durch eine Zugriffsmethode für Zeichen realisiert wird, am leichtesten erklären bzw. realisieren.

Satzlängenkodierungen Bei allen Zugriffsmethoden für variable Satzlänge steht man (unabhängig von der Zugriffsstruktur) vor dem Problem, dass die Länge jedes einzelnen Satzes in irgendeiner Form erkennbar sein muss. Hierfür sind 2 Verfahren üblich:

1. **Satzende-Markierung** : dies ist ein Sonderzeichen oder eine Folge von Sonderzeichen, die das Satzende anzeigt. Üblich sind die Druckersteuerzeichen ‘carriage return’ und ‘line feed’; dies ist bei Ausgabe der Datei auf einen Drucker oder ein einfaches Terminal vorteilhaft. Die Sonderzeichenfolge darf natürlich nicht mehr innerhalb des Satzes auftreten, sondern muss dort umkodiert werden.
2. **Längenfeld** vor dem Satz: Typischerweise werden 2 Byte zur Darstellung der Länge verwendet.

Realisierung von Zugriffsmethoden für variable Satzlänge und sequentielle Zugriffsstruktur Auf Basis einer Zugriffsmethode für Zeichen ist die Realisierung sehr einfach: man konkateniert einfach alle Sätze (incl. Satzende-Markierung bzw. Längenfeld).

Die Realisierung auf Basis einer Zugriffsmethode für Seiten, bei der nur so viele ganze Sätze in einem Block gespeichert werden, wie in diesen hineinpassen, ist ziemlich kompliziert (und wird noch komplizierter, wenn die maximale Satzlänge S größer als die Blocklänge B ist). Daher ist es einfacher, zunächst über der Zugriffsmethode für Seiten eine Zugriffsmethode für Zeichen zu realisieren.

Realisierung von Zugriffsmethoden für variable Satzlänge und dynamische Direktzugriffsstruktur Eine wesentliche Anforderung ist hier, dass die Sätze einer Datei effizient sequentiell gelesen werden können. Dies impliziert, dass jeder zur Datei gehörige Sektor nur einmal zum Hauptspeicher übertragen und dann sofort komplett verarbeitet werden muss. Deshalb müssen die Sätze so auf die Seiten der Datei verteilt werden, dass jede Seite die Sätze mit Schlüsselwerten aus einem bestimmten Intervall enthält. Beim Einfügen von Sätzen mitten in der Datei kann nun ein Überlauf einer Seite auftreten, beim Löschen analog ein Unterlauf.

Hauptprobleme bei der Realisierung einer dynamischen Direktzugriffsstruktur sind Behandlung von Über- und Unterläufen sowie die Wahl der Zugriffspfade, durch die zu einem gegebenen Schlüsselwert der zugehörige Satz gefunden wird. Die gängigste Datenstruktur zur Lösung der Probleme sind B*-Bäume; diese werden in den Kursen zu Datenstrukturen und Datenbanksystemen vorgestellt, so dass wir hier darauf verzichten können.

Da die hier diskutierten Zugriffsmethoden auf alle Fälle Seitengrenzen berücksichtigen und kennen müssen, sollten sie auf Basis einer Zugriffsmethode für Seiten realisiert werden. Ist nur eine Zugriffsmethode für Zeichen vorhanden, dann kann auf dieser leicht eine Zugriffsmethode für Seiten realisiert werden.

2.4 Magnetbänder

Magnetbänder spielen seit jeher eine große Rolle bei der Sicherung und Archivierung von Daten. Grund ist der im Vergleich zu Direktzugriffsspeichern sehr günstige Preis pro Megabyte Speicherplatz; die schlechten Zugriffszeiten spielen dabei keine Rolle.

Eine weitere wichtige Rolle spielen Bänder als Distributionsmedium für Daten bzw. Software. Da sie relativ kompakt und unempfindlich sind, eignen sie sich zum Versand per Post. Häufig wird auch das Betriebssystem eines Rechners auf einem Magnetband ausgeliefert. Daher enthält praktisch jeder größere Rechner ein Magnetbandlaufwerk.

2.4.1 Hardware-Eigenschaften

Eine schon sehr alte Form von Magnetbändern sind **Spulen** mit 1/2 Zoll breitem Band. Auf diesem Band sind 9 parallele Spuren vorhanden, die parallel geschrieben bzw. gelesen werden können. Übliche Schreibdichten sind 800, 1600, 3200 und 6250 Bits pro Zoll (bpi). Eine der Spuren zeichnet ein Parity-Bit zur Fehlererkennung

auf. Die maximale Kapazität der Bänder liegt je nach Länge (bis zu 730 m) und Schreibdichte zwischen 20 und 150 MByte.

Auch **Magnetbandkassetten** sind schon lange bekannt und weit verbreitet. In *konventionellen* Kassetten ist im Vergleich zu Spulen das Magnetband meist schmaler (z. B. 1/4 Zoll), kürzer (bis zu 190 m) und dichter beschreibbar und hat mehr Spuren. Die maximale Kapazität der Kassetten liegt je nach Typ zwischen 60 und ca. 500 MByte. Im Unterschied zu Spulenbändern werden die Spuren einzeln, also nicht parallel beschrieben. Die Spuren werden abwechselnd in verschiedenen Richtungen benutzt.

Eine weitere Klasse von Magnetbandkassetten basieren auf Aufzeichnungstechnologien aus dem Video-Bereich (VHS und 8mm-Format). Die Daten werden hier im Schrägspur-Verfahren aufgezeichnet. Die Kapazität dieser Kassetten reicht bis zu 14 GByte.

Bänder sind immer wechselbare Datenträger.

Blöcke sind die Einheit, die zwischen Band und Hauptspeicher übertragen wird. Auch Bänder werden üblicherweise von Controllern gesteuert; diese enthalten wiederum meist große Puffer.

Der Inhalt des Bands ist bei den Varianten der Bänder verschieden aufgebaut. Bei Spulenbändern werden die Bits eines Bytes sowie ein Parity-Bit in die parallelen Spuren geschrieben. Ein Block (also z. B. 1 kByte Nutzinformation) entspricht daher einem Abschnitt der Bandoberfläche. Zwischen je zwei Abschnitten muss aus technischen Gründen eine unbenutzte **Lücke** (gap) bleiben. Bei Kassetten bilden die *waagerechten* bzw. *schrägen* Spurstücke auf dem Band logisch gesehen eine einzige Spur, in die bitseriell geschrieben wird. Ein Block entspricht einem Abschnitt in dieser logischen Spur, ähnlich wie ein Sektor auf einer Plattenspur.

Die Größe der Blöcke kann relativ frei gewählt werden. Für kleine Blöcke spricht, dass der Datenverlust bei Defekten bzw. der Korrekturaufwand geringer ist. Je kürzer allerdings die Blöcke sind, desto größer ist der Anteil der Lücken am Band und umso geringer die Netto-Kapazität des Bands.

Es ist nicht (wie bei Platten) erforderlich, dass alle Blöcke auf dem Band gleich lang sind. Eine variable Länge kann für manche Anwendungsfälle vorteilhaft sein.

Viele Bandgeräte können Blöcke sowohl vorwärts wie rückwärts schreiben. *Direkt zugreifbar* sind nur diejenigen Blöcke, die sich in der Nähe des Lese-/Schreibkopfes befinden. Auf weiter entfernte Blöcke kann nur zugegriffen werden, nachdem das Band dorthin umgespult worden ist. Bei variabel langen Blöcken muss sogar das ganze Band gelesen werden. Bänder sind daher keine Speicher mit Direktzugriff.

Operationen. Die wesentlichen Operationen auf Bändern, die dem Betriebssystem zur Verfügung stehen, sind:

- lies/schreibe den nächsten/vorigen Block
- positioniere n Blöcke vorwärts/rückwärts
- spule zum Anfang zurück

2.4.2 Verwaltung der Datenträger

Wie schon oben erwähnt sind Magnetbänder wechselbare Datenträger. Dies führt zu gewissen Problemen, z. B. bei der Verwaltung der Bänder und beim Zugriffsschutz. Allerdings hängen diese Probleme davon ab, ob sich z. B. die Bandlaufwerke in einem abgeschlossenen Raum unter der Kontrolle von Operateuren befinden oder direkt an einem Benutzerarbeitsplatz, ob der Rechner eine größere Anzahl anonymer Benutzer bedient oder nur eine kleine, geschlossene Gruppe von Benutzern usw. Daher sind die in verschiedenen Betriebssystemen vorzufindenden Maßnahmen zur Verwaltung von Bändern recht unterschiedlich.

Etiketten. Um zu verhindern, dass ein Benutzer A ein Band montieren und dann lesen und/oder verändern könnte, welches eigentlich einem Benutzer B gehört, werden in vielen Betriebssystemen **Etiketten** benutzt. Dies sind spezielle Blöcke, die an den Anfang des Magnetbands geschrieben werden und entsprechende Angaben enthalten. Die Etiketten werden nur vom Betriebssystem geschrieben und können über die Funktionen, die das Betriebssystem den Benutzern anbietet, nicht beliebig verändert werden. Bänder ohne Etikett werden überhaupt nicht verarbeitet, damit diese Kontrollen nicht umgehbar sind.

Repräsentation von Bändern. Manche Betriebssysteme erlauben es, ein Verzeichnis der vorhandenen Bänder bzw. Kassetten zu führen. Dies geschieht am einfachsten dadurch, dass man Bänder durch Dateien repräsentiert, da Standard-Angaben für Dateien wie Zugriffsrechte, Datum der letzten Änderung, Satzlänge u. a. auch bei Bändern sinnvoll sind.

Handhabung von Bändern. Bei Rechnern mit zentral (z. B. in einem Rechnerraum) aufbewahrten Datenträgern ist ein Betriebssystem-Befehl erforderlich, der das Einlegen eines Bandes bewirkt. Bei mehreren gleichwertigen Laufwerken kann das Betriebssystem eines aussuchen und dann bei manuell bedienten Geräten eine entsprechende Meldung auf die Operateurskonsole schreiben bzw. bei elektronisch gesteuerten Geräten eine Kassette aus dem Lager entnehmen und in das Laufwerk einlegen (derartige automatische Datenträger-Wechsler existieren auch für optische Platten).

2.4.3 Dateien auf Magnetbändern

Normalerweise werden Magnetbänder nur sequentiell beschrieben bzw. gelesen. Daher besteht der Inhalt des Magnetbands entweder aus einer einzigen Datei oder einer Folge von Dateien, die allerdings nur sequentiell gelesen oder geschrieben werden können. Hierarchische Verzeichnisse wie bei Platten sind nicht üblich.

Sofern bei der Datensicherung mehrere Dateien auf ein Band geschrieben werden sollen, fügt man diese zunächst durch ein Backup-Programm zu einer einzigen Datei zusammen (ähnlich wie eine Bibliothek) und komprimiert sie dabei oft gleichzeitig. Ähnlich wie ein Dateiverzeichnis gruppiert ein derartiges Backup-Programm eine Menge von Dateien (die aber nicht unbedingt aus einem einzigen Verzeichnis des Dateisystems stammen müssen). Der wesentliche Unterschied ist allerdings, dass auf

einzelne Dateien nicht mehr direkt mit Hilfe eines Pfadnamens zugegriffen werden kann.

2.5 Zusammenfassung

Eine der wichtigsten Leistungen eines Rechners besteht in der permanenten Speicherung von Daten. Hierzu werden Speichergeräte benutzt. Bei den heute verfügbaren Technologien dominieren Magnetplatten, Disketten und optische Platten als Direktzugriffspeicher und verschiedene Sorten von Bändern als Back-up- und Archivierungsmedium. Wir haben die Gemeinsamkeiten und Unterschiede zwischen verschiedenen Arten von Speichergeräten und die sich daraus für die Verwaltung der Geräte und Datenträger ergebenden Aufgaben kennengelernt.

Aus Sicht der Benutzer sind Dateien die verfügbaren Speichermedien in einem Rechner. Die Dateien in einem Dateisystem sind üblicherweise in hierarchischen Verzeichnissen organisiert. Benutzerprogramme greifen auf eine Datei über die Operationen einer Zugriffsmethode zu. Zugriffsmethoden realisieren entweder Seiten, Zeichen oder Sätze als Speichereinheiten und realisieren eine Zugriffsstruktur wie die sequentielle oder index-sequentielle. Dateien sind virtuelle Geräte; generell werden oberhalb der Betriebssystemebene nur noch virtuelle Geräte über die Systemaufrufe zur Verfügung gestellt; das Betriebssystem selbst arbeitet mit realen Geräten. Wir haben diverse Unterschiede zwischen realen und virtuellen Geräten kennengelernt.

Die Hauptaufgaben bei der Realisierung von Dateisystemen auf Platten sind die Verwaltung aller organisatorischen Angaben zu den Dateien (in Dateiattributen) sowie die Zuordnung von Sektoren zu Dateien und die Verwaltung der freien Sektoren einer Platte. Hierfür haben wir drei Techniken kennengelernt (FAT, Sektoradresstabellen, Sektorfolgen). Weiterhin haben wir diverse Maßnahmen kennengelernt, durch die der Zugriff auf die Sektoren einer Datei beschleunigt wird.

Die E/A-Software ist in Schichten gegliedert. Man unterscheidet zwischen geräteabhängigen Teilen (Treiber) und geräteunabhängigen Teilen. Bei den Zugriffsmethoden unterscheidet man zusätzlich Schichten, in denen Seiten und Zeichen bzw. Sätze als Speichereinheiten realisiert werden.

Beim Entwurf der E/A-Software müssen einige Prinzipien und Entwurfsziele beachtet werden. Wichigstes Ziel ist die Geräteunabhängigkeit von Benutzerprogrammen. Weitere Ziele sind einheitliche Namensräume für alle (virtuellen) Geräte, Kodierungsunabhängigkeit, Behandlung von parallelen Zugriffen und möglichst frühe Fehlerbehebung.

Literatur

[Ha98] J.P. Hayes. *Computer architecture and organization (3rd edition)*. McGraw–Hill, 1998.

[SiGaGa02] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts: sixth Edition*. John Wiley & Sons, Inc. 2002 .

[Ta02] A. S. Tanenbaum. *Moderne Betriebssysteme: 2., überarbeitete Auflage*. Pearson Studium, 2002.

[Ta06] A. S. Tanenbaum. *Computerarchitektur: Strukturen-Konzepte-Grundlagen 5. Auflage*. Pearson Studium, 2006.

Glossar

Adapter s. Controller.

Ausgabegerät wandelt im Rechner gespeicherte Informationen in eine visuelle, akustische oder sonstige Form um (z. B. Bildschirm, Hupe, Drucker usw.).

Block (*block, physical record*) Nutzinhalt eines Sektors einer Platte bzw. eines Bands.

Block-Gerät (*block device*) Geräte, bei denen die Übertragungseinheit zwischen Hauptspeicher und Gerät bzw. Controller ein Block ist.

Controller dient dem Anschluss von E/A-Geräten an einen Rechner.

Datei (*data set, file*) virtuelles Speichermedium (aus Sicht von Benutzerprogrammen); realisiert innerhalb eines Dateisystems.

Dateiattributes (*file attributes*) Beschreibung einer Datei (z. B. Dateiname, Dateigröße, Zugriffsrechte, Speicherungsort u. ä.); enthalten in einem Dateiverzeichnis.

Dateiseite von einer Zugriffsmethode für Seiten realisierte Speichereinheit; eine Datei besteht aus mehreren Seiten; eine Seite ist meist so groß wie ein Block und wird in einem Sektor gespeichert.

Dateisystem Menge von Dateien und Verzeichnissen sowie erforderlicher Hilfsdaten, welche auf einem (logischen) Datenträger gespeichert werden.

Dateiverwaltungssystem Teil der E/A-Software, der Operationen mit Dateien und der Dateisysteme auf Datenträgern realisiert.

Dateiverzeichnis (*directory*) enthält eine Menge von Dateiattributen.

Datenträger (*medium*) speichert Daten in maschinenlesbarer Form, z. B. magnetisierte Folie, gelochte Papierstreifen u. ä.

E/A-Software Teil des Betriebssystems, der alle mit der Verwaltung und Benutzung von E/A-Geräten zusammenhängenden Funktionen realisiert.

Eingabegerät (*input device*) dient der Aufnahme und Umwandlung von Informationen aus der Umwelt in eine digitalisierte Form (z. B. Tastatur, Maus usw.).

file allocation table (FAT) Array zum Verwalten der Sektoren einer Platte oder Diskette.

Geräte-Treiber (*device driver*) Programm, das von der CPU ausgeführt wird und ein bestimmtes Gerät steuert.

i-node Realisierung einer Datei im Betriebssystem UNIX; enthält u. a. Sektoradressen der Dateiseiten.

interleaving Umnummerierung der Sektoren einer Spur zur Beschleunigung des sequentiellen Lesens oder Schreibens von Sektoren in der Spur.

Katalog Dateiverzeichnis.

Kommunikationsgerät E/A-Gerät, das zum Datenaustausch zwischen verschiedenen Rechnern dient.

Latenzzeit (*latency time*) Zeit, bis ein gesuchter Sektor in einer Spur bei dem Lese-/Schreibkopf erscheint.

Laufwerk (*drive*) Komponente eines Rechners, in die Datenträger montiert werden können; ermöglicht Lesen/Schreiben auf dem Datenträger.

logische Platte Intervall von Zylindern eines Plattenlaufwerks; ein physisches Plattenlaufwerk kann in mehrere logische Platten unterteilt werden.

Medium s. Datenträger.

Pfadname (*path name*) Name einer Datei innerhalb eines hierarchischen Dateisystems; besteht aus einer Folge von lokalen, innerhalb von einzelnen Dateiverzeichnissen gültigen Dateinamen; Ausgangspunkt ist bei **absoluten Pfadnamen** das Wurzelverzeichnis, bei **relativen Pfadnamen** das aktuelle Arbeitsverzeichnis.

SCAN Auswahlkriterium für die Abarbeitung von wartenden Übertragungsaufträgen einer Platte.

Sektor (*sector*) Teil einer Spur einer Platte; identifiziert durch die Nummer der Oberfläche, der Spur und des Sektors innerhalb der Spur.

Sektoradresstabelle Tabelle zur (dezentralen) Verwaltung der Sektoren einer Datei.

Sekundärspeicher (*secondary memory*) zur permanenten Speicherung großer Datenmengen dienendes E/A-Gerät.

shortest-seek-time-first (SSTF) Auswahlkriterium für die Abarbeitung von wartenden Übertragungsaufträgen einer Platte.

Spooling Simulation von exklusiv benutzbaren E/A-Geräten wie Drucker, Lochstreifenleser etc.

Spur (*track*) eine kreisförmige Linie auf einer Plattenoberfläche, auf der ein Lese-/Schreibkopf Daten liest bzw. schreibt.

Suchzeit (*seek time*) Zeit zum Positionieren des Lese-/Schreibkopfes eines Plattenlaufwerks auf eine andere Spur.

Treiber s. Geräte-Treiber.

virtuelles Gerät (*virtual device*) durch das Betriebssystem simuliertes Gerät; der Typ (bzw. die Funktionalität) eines virtuellen Geräts ist oft eine Abstraktion/Idealisierung einer Menge ähnlicher realer Geräte.

Wurzelverzeichnis (*root directory*) Wurzel eines Dateisystems - Ausgangspunkt für alle absoluten Pfadnamen beim Zugriff auf Dateien.

Zeichen-Gerät (*character device*) Gerät, bei dem die Übertragungseinheit zwischen Hauptspeicher und Gerät bzw. Controller ein Zeichen ist.

Zugriffsmethode (*access method*) Menge von Operationen, durch die der Inhalt einer Datei in eine Menge kleinerer Speichereinheiten (Dateiseiten, Sätze oder Zeichen) zerlegt wird und durch die eine Zugriffsstruktur für diese Menge von Speichereinheiten realisiert wird. Beispiele: sequentielle und index-sequentielle Zugriffsmethode.

Zylinder (*cylinder*) Menge der Spuren auf den Oberflächen einer Platte mit einem gegebenen Radius.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 3:
Prozess- und Prozessorverwaltung

Autoren: Stefan Wohlfeil, Rolf Klein, Christian Icking, Lihong Ma

Inhalt

Kurseinheit 3

Prozess- und Prozessorverwaltung

3.1 Einführung

Wie schon in Kurseinheit 1 erklärt wurde, sind schnelle Prozessoren mit der Abarbeitung eines einzigen Programms in der Regel unterfordert. Wir gehen daher im Folgenden davon aus, dass der Rechner im Mehrprogrammbetrieb (*Multiprogramming*, *Multitasting*) betrieben wird. Während ein Programm z. B. auf Eingaben des Benutzers wartet, kann der Prozessor mit der Bearbeitung eines anderen Programms weitermachen. Oder der Prozessor kann regelmäßig jeweils ein Stück eines Programms bearbeiten und dann mit einem anderen Programm weitermachen. Man sagt, dass der Prozessor zwischen verschiedenen Programmen hin- und hergeschaltet wird.

Es sollen also mehrere Programme quasi gleichzeitig (als *parallele Prozesse*) abgearbeitet werden. Die Kontrolle dieser parallelen Aktivitäten ist nicht einfach. In dieser Kurseinheit wollen wir uns damit befassen, wie das Modell der Prozesse hilft, diese Parallelität in Systemen einfacher zu beschreiben und zu beherrschen.

Außerdem werden wir vorstellen, wie es möglich ist, dass der Prozessor zwischen zwei Prozessen hin- und hergeschaltet werden kann. Dann stellen wir eine Reihe von Strategien vor, wie das Betriebssystem aus der großen Zahl von bereiten Prozessen einen Prozess auswählen kann, der als nächster den Prozessor zugeteilt bekommt. Die Auswahl einer solchen Strategie bestimmt das Verhalten des Rechnersystems in entscheidendem Maße.

3.2 Programme und Prozesse

Es ist ausgesprochen schwierig, den Begriff Prozess so allgemein zu definieren, dass er auf alle Rechner und Betriebssysteme sinnvoll anwendbar ist. Wir werden daher keine formale Definition angeben, sondern wollen den Begriff informell beschreiben.

Aus dem bisher Gesagten ist schon klar, dass die Begriffe Programm und Prozess irgendwie zusammengehören. Wir wollen uns deshalb zunächst kurz mit dem Begriff Programm genauer befassen. Dann werden wir die Unterschiede zwischen einem Programm und einem Prozess aufzeigen.

Programme. Programme können in sich parallel oder sequentiell sein. Wir beschränken uns auf die Klasse der sequentiellen Programme, wie sie in den meisten

gängigen Programmiersprachen formuliert werden können. Sequentiell heißt in diesem Zusammenhang, dass während der Ausführung des Programms zu jedem Zeitpunkt höchstens eine Anweisung des Programms ausgeführt wird. Das Programm ist eine Formulierung eines Algorithmus, d. h. es beschreibt in endlicher Form, welche Operationen in welcher Reihenfolge ausgeführt werden sollen. Der Programmtext ist die vollständige Beschreibung des Programms.

Prozess. Wird ein Programm von einem Prozessor abgearbeitet, so reicht der Programmtext zur vollständigen Beschreibung nicht mehr aus. Der Prozessor hat bereits eine Reihe von Operationen ausgeführt. Er kennt die nächste auszuführende Operation (Programmzähler), und seine Register enthalten berechnete Daten. Während der Programmtext normalerweise auf sekundären Speichern (Festplatten, Bänder) steht, befindet sich ein Programm zur Ausführung im Hauptspeicher. Falls das Programm Dateien bearbeitet, so hat es diese evtl. geöffnet und schon teilweise gelesen. Damit können wir jetzt den Begriff Prozess wie folgt beschreiben:

> Ein ablaufendes Programm bezeichnet man als **Prozess**, inklusive des aktuellen Werts des Befehlszählers, der Registerinhalte und der Belegungen der Variablen.

Unterschiede. Wichtig an dieser Festlegung ist der kleine Unterschied zwischen den Begriffen Programm und Prozess. Ein Programm an sich ist etwas statisches, während ein Prozess etwas dynamisches ist. Tanenbaum [Ta02] hat dies durch eine Analogie sehr schön gezeigt. Ein Mensch will einen Kuchen backen. Er hat ein Kochbuch, in dem das Rezept für die Zubereitung steht. Außerdem hat er alle Zutaten. Das Rezept entspricht in dieser Analogie dem *Programm*, der Mensch entspricht dem *Prozessor*, die Zutaten den Eingabedaten und der Vorgang des Kuchenbackens entspricht dem *Prozess*.

Es ist anhand dieser Analogie auch leicht einzusehen, dass mehrere Prozesse dasselbe Programm ausführen können (Mehrere Menschen backen nach demselben Rezept). Trotzdem sind diese Prozesse natürlich nicht gleich, da sie z. B. verschieden weit in ihrer Ausführung vorangekommen sein können. Außerdem sieht man leicht ein, dass ein Prozessor (Mensch) mehreren Prozessen (Apfelkuchen backen, Zeitung lesen, spülen) abwechselnd zugeteilt werden kann.[1]

3.3 Prozesszustände und Prozessumschaltung

In diesem Abschnitt werden wir die schon erwähnten Merkmale präzisieren, die einen Prozess von einem Programm unterscheiden.

Speicherbereich. Wie schon gesagt, muss ein Prozess im Hauptspeicher des Rechners sein[2], damit er vom Prozessor ausgeführt werden kann. Wie das genauer realisiert werden kann, ist Thema der Kurseinheit 4 (Hauptspeicherverwaltung). Im Moment gehen wir einfach davon aus, dass das Betriebssystem dem Prozess (wie

[1] Wer oder was in dieser Analogie dann das Betriebssystem ist, das die Zuordnung übernimmt, bleibt der Phantasie der Leserinnen und Leser überlassen.

[2] Zumindest muss der Teil, der die nächsten auszuführenden Instruktionen für den Prozessor enthält, im Hauptspeicher sein.

auch immer) einen *logischen Speicherbereich* zuteilt. Man spricht in diesem Fall auch vom *logischem Adressraum* des Prozesses. In diesem Bereich kann der Prozessor den Prozess ausführen.

Kennzeichnend für einen Prozess ist nun, womit dieser seinen Speicherbereich belegt. Der logische Speicherbereich eines Prozesses enthält:

Programmsegment Es enthält den vom Prozessor ausführbaren Code des Programms.

Stacksegment Hier steht der Programmstack, auf den bei jedem Prozeduraufruf ein Aktivierungsblock mit den Parametern, lokalen Variablen und der Rücksprungadresse kommt.

Datensegment Hier stehen die Daten, die das Programm bearbeiten soll oder die es erzeugt hat.

Während der Abarbeitung des Prozesses ändert sich das Programmsegment nicht[3], jedoch ändern sich Stack- und Datensegment mit dem Fortschritt der Abarbeitung. Im Moment ist für unsere Betrachtungen allerdings nur wichtig, dass

1. das Betriebssystem dem Prozess zu seinem Beginn einen logischen Speicherbereich zur Verfügung stellt.
2. nach dem Ende des Prozesses dieser Speicherbereich wieder freigegeben wird.

Was der Prozess zwischen seiner Entstehung und seinem Verschwinden in seinem Speicherbereich macht, ist in diesem Kapitel nicht weiter wichtig.

Prozesszustände. Zwischen seinem Entstehen und Verschwinden durchläuft ein Prozess verschiedene Zustände. Manchmal kommt er in der Ausführung seines Programms vorwärts, dann muss er evtl. auf Benutzereingaben warten. Wenn in einem Rechner nur ein Prozessor vorhanden ist, so kann dieser auch nur einen Prozess zu einem bestimmten Zeitpunkt abarbeiten. Weitere im System vorhandene Prozesse müssen warten. Dabei unterscheidet man Prozesse, die nur darauf warten, dass der Prozessor frei wird, und Prozesse, die z. B. auf Benutzereingaben warten. Letztere können in ihrer Ausführung ja selbst dann nicht fortfahren, wenn sie den Prozessor zugeteilt bekämen. Wir unterscheiden also folgende **Prozesszustände**:

erzeugt: Der Prozess ist gerade neu entstanden, d. h. das Betriebssystem erstellt die Datenstrukturen, die für die Verwaltung des Prozesses verwendet werden, und weist ihm Adressraum im Hauptspeicher zu.

bereit: Der Prozess ist rechenbereit, hat aber den Prozessor nicht zugeteilt bekommen. Er wartet nur darauf, dass der Prozessor für ihn frei wird.

rechnend: Der Prozessor ist dem Prozess zugeteilt und arbeitet die Anweisungen des zugehörigen Programms ab.

blockiert: Der Prozess wartet auf irgendein Ereignis (z. B. Benutzereingabe).

beendet: Das Ende der Ausführung des zugehörigen Programms ist erreicht.

[3] Von Programmen, die selbstmodifizierenden Code enthalten, sehen wir hier einmal ab.

Auf einem Rechner mit einem Prozessor kann immer nur ein Prozess im Zustand *rechnend* sein. Dagegen können mehrere Prozesse in den Zuständen *erzeugt*, *bereit*, *blockiert* und *beendet* sein. Diese Prozesse werden dann in Listen verwaltet.

Dieses Modell der Prozesse ist auch auf Mehrprozessor-Rechnern anwendbar. Ein Unterschied ist dann, dass auch mehrere Prozesse im Zustand *rechnend* sein können. Jedoch kann auch hier auf einem Prozessor immer nur ein Prozess ausgeführt werden.

Prozesserzeugung. Es gibt vier häufige Ereignisse, die das Erzeugen eines neuen Prozesses verursachen:

1. *Bereitstellung von Diensten durch das Betriebssystems.* Wenn ein Rechner gestartet wird, werden normalerweise mehrere Prozesse erzeugt. Einige davon interagieren mit Benutzern und laufen im Vordergrund. Andere laufen im Hintergrund, z. B. das Empfangen von E-Mail und der Prozess, der die Drucksteuerung ausführt. In UNIX kann man das Kommando *top* ausführen, um die laufenden Prozesse und ihre Ressourcenverbrauch anzuzeigen. Unter Windows 95/98/ME kann man die Tastenkombination CTRL-ALT-DEL benutzen, und unter Windows 2000 wird dazu der Taskmanager benutzt.

2. *Systemaufruf zum Erzeugen eines neuen Prozess durch existierende Prozesse.* Ein bestehender Prozess kann mehrere neue Prozesse erzeugen, um die Arbeit zu verteilen. In UNIX existiert nur ein Systemaufruf zum Erzeugen eines neuen Prozesses: `fork`. Unter Windows gibt es den entsprechenden `CreateProcess`. Nach einer Prozesserzeugung haben die beiden Prozesse, also *Vaterprozess* und *Kindprozess*, je einen eigenen getrennten Adressraum.

3. *Benutzeranfrage.* Wenn ein Benutzer ein Kommando im einem Fenster eingibt oder ein Doppelklicken eines Icons startet, wird ein neuer Prozess erzeugt und lässt das gewählte Programm darin laufen.

4. *Ein neuer Batch-Job zum Starten.*

Prozessbeendigung. Nachdem ein Prozess gestartet hat, wird ein Programm ausgeführt, danach terminiert er irgendwann. Gründe für das Terminieren eines Prozesses sind:

1. *Normale Beendigung.* Der Prozess hat seine Arbeit erfolgreich beendet, anschließend wird ein Systemaufruf z. B. unter UNIX `exit` und unter Windows `ExitProcess` ausgeführt.

2. *Fehler durch den Prozess selbst*, z. B. der Prozess versucht auf ungültige Speicheradressen zuzugreifen oder die Division durch Null auszuführen.

3. *Fehler*, den der Prozess festgestellt hat, z. B. Zugriff auf eine Datei, die nicht existiert oder nicht geöffnet ist.

4. *Beendigung durch einen anderen Prozess.* Ein Systemaufruf z. B. unter UNIX `kill` und unter Windows `TerminateProcess` durch einen anderen Prozess beendet den laufenden Prozess.

Prozesshierachien. Ein Prozess kann einen oder mehrere Kinderprozesse erzeugen, die Kinderprozesse können wieder Kinderprozesse erzeugen. Aus einem Prozess und seinen Nachkommen entsteht eine Prozessfamilie, die eine Prozesshierachie bildet. In UNIX gibt es zwei spezielle Prozesse *Prozess 0* und *Prozess 1*. Wenn das System gebootet wird, wird *Prozess 0* erzeugt, der die Echtzeituhr einrichtet, das Wurzelverzeichnis mountet und einen Prozess 1 erzeugt, der auch als *init*-Prozess bezeichnet wird. Alle weitere Prozesse im System stammen von *init* ab. Der Prozess *init* stellt zuerst fest, wie viele Terminals es geben soll. Dann spaltet er für jedes Terminal einen neuen Prozess, der wartet, bis sich jemand anmeldet. Nachdem ein Benutzer korrekt eingeloggt hat, führt der Login-Prozess eine Shell aus, um Kommandoeingaben einzunehmen. Diese Kommandoeingaben können wiederum Prozesse starten, siehe Abbildung 3.1. Somit gehören alle Prozesse im gesamten System zu einem einzigen Baum mit `init` an seiner Wurzel. Unter Windows hingegen gibt es kein Konzept einer Prozesshierachie, es wird keine Beziehung zwischen Vater und Kindern erzwungen. Alle Prozesse sind gleichwertig.

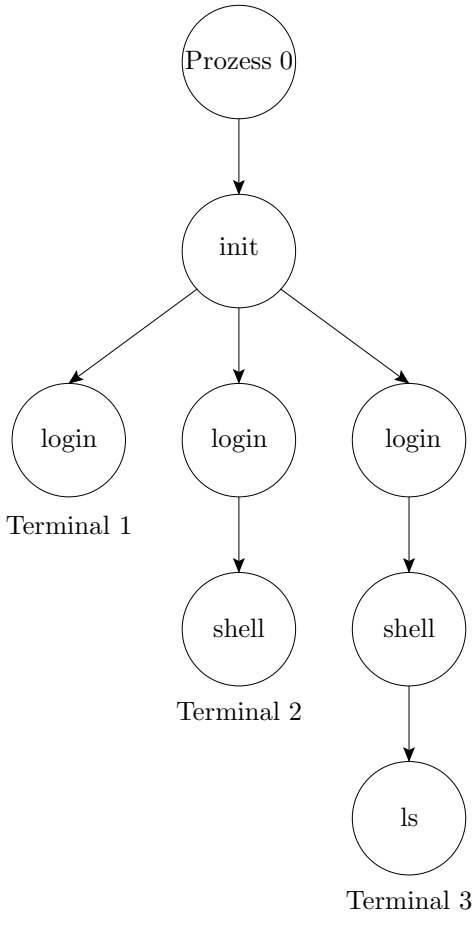


Abbildung 3.1: UNIX/Linux-Prozessbaum.

3.3.1 Zustandsübergänge

Zwischen den Prozesszuständen sind nun verschiedene Übergänge möglich. In Abbildung 3.2 sind noch einmal die fünf Zustände und die möglichen Übergänge in einem Zustandsübergangsdiagramm dargestellt.

Wenn ein neuer Prozess erzeugt wird, so tritt dieser neue Prozess kurz in den Zustand *erzeugt* ein (1). Damit das zugehörige Programm tatsächlich ausgeführt werden kann, müssen dem Prozess einige Ressourcen zugeteilt werden. Die wichtigste davon ist der Hauptspeicher, in dem der Prozess ablaufen soll. Hat das System einen logischen Speicherbereich (siehe Kurseinheit 4) für den Prozess festgelegt, so geht der Prozess in den Zustand *bereit* über (2). Der Prozess wartet nun darauf, dass er den Prozessor zugeteilt bekommt, damit er abgearbeitet werden kann.

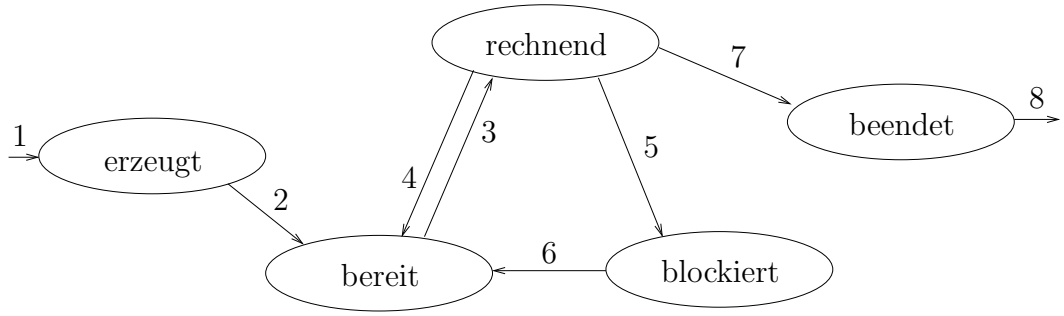


Abbildung 3.2: Prozesszustandsübergangsdiagramm.

Nachdem der bereite Prozess den Prozessor zugeteilt bekommt, geht er in den Zustand *rechnend* über (3). In diesem Zustand bleibt der Prozess dann eine Weile. Wartet der Prozess dann auf ein Ereignis (z. B. Ein-/Ausgaben), so geht er in den Zustand *blockiert* (5). Für jeden Ereignistyp existiert eine eigene Liste (Warteschlange), in der die Prozesse einsortiert werden, die auf ein Ereignis dieses Typs warten. Mögliche Ereignistypen sind: Tastendruck, Mausklick, Plattenzugriff, Bereitmeldung vom Drucker, Nachrichten von einem anderen Rechner eines Rechnernetzes usw. In Abbildung 3.3 sind die einzelnen Warteschlangen dargestellt.

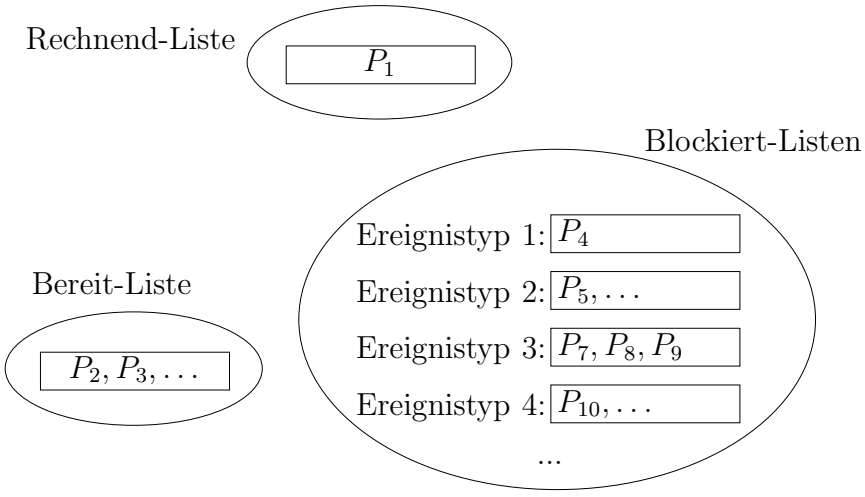


Abbildung 3.3: Listen von Prozessen in den jeweiligen Zuständen.

In diesem Zustand bleibt der Prozess, bis das Ereignis eingetreten ist. Wenn das Ereignis eintritt, wird in der zugehörigen Liste der blockierten Prozesse ein Prozess, der auf dieses Ereignis wartet, herausgesucht. Dieser Prozess geht in den Zustand *bereit* über (6).

Übungsaufgabe 3.1 Erklären Sie, warum kein direkter Übergang vom Zustand *blockiert* in den Zustand *rechnend* vorgesehen ist.

Der direkte Übergang vom Zustand *rechnend* in den Zustand *bereit* (4) kann zwei Ursachen haben. Zum einen kann ein Prozess von sich aus entscheiden, dass er den Prozessor an einen anderen Prozess abgeben will. Dazu ruft er eine Systemfunktion auf, das Betriebssystem erhält die Kontrolle und kann einen anderen Prozess in den Zustand *rechnend* versetzen. Betriebssysteme, bei denen nur der Prozess selbst entscheidet, ob er den Prozessor abgeben will oder nicht, nennt man **non-preemptive**. Die zweite Möglichkeit ist, dass das Betriebssystem einem laufenden Prozess den Prozessor entzieht. Wie das genau vor sich geht, wird später erklärt. Solche Systeme nennt man **preemptive**.

Terminiert der Prozess, so geht er in den Zustand *beendet* (7). Der von ihm belegte Speicherbereich sowie weitere vom Prozess gebundene Ressourcen werden freigegeben. Dann verschwindet der Prozess aus dem System (8).

3.3.2 Der Prozesskontrollblock

In diesem Abschnitt wollen wir uns etwas genauer damit befassen, welche Informationen über Prozesse das Betriebssystem haben muss.

Identifikation. Das Betriebssystem muss als erstes wissen, welche Prozesse im System überhaupt existieren. Jeder Prozess muss eindeutig identifizierbar sein (*Prozessnummer, Prozess-ID*). Sonst könnte es nicht entscheiden, welcher Prozess als nächster den Prozessor zugeteilt bekommen soll. Außerdem muss es von jedem Prozess wissen, in welchem Zustand er sich befindet.

Hauptspeicher. Damit das Betriebssystem unterschiedliche Prozesse voreinander schützen kann, muss es die Speicherbereiche, in denen ein Prozess abläuft, kennen. Dann kann es Speicherzugriffe eines Prozesses in den Speicherbereich eines anderen Prozesses erkennen und verhindern (*Grenzregister*).

In Kurseinheit 4 (Hauptspeicherverwaltung) wird noch vorgestellt, dass ein Prozess nicht immer komplett im Hauptspeicher vorliegen muss, damit er abgearbeitet werden kann. Man unterscheidet zwischen dem logischen und dem physischen Speicherbereich eines Prozesses. An dieser Stelle ist aber nur wichtig, dass das Betriebssystem neben der Schutzinformation weitere Informationen braucht, um die „Speicherverhältnisse“ eines Prozesses zu kennen.

Prozessorzustand. Wenn ein Prozess wieder in den Zustand *rechnend* übergehen soll, so muss der Prozess die Abarbeitung seines Programms an genau der Stelle fortsetzen, an der er vorher stand. Das Betriebssystem muss also den Programmzähler wieder restaurieren. Außerdem müssen alle Prozessorregister wieder ihre alten Werte erhalten. Hatte der Prozess vorher eine Datei teilweise gelesen, so muss auch dieser Zustand (welche Datei war geöffnet, wo stand der Dateizeiger) wiederhergestellt werden.

Buchhaltung. Neben den bisher genannten Daten verwaltet das Betriebssystem dort weitere Informationen. Zum Beispiel wird dort auch Buch darüber geführt,

wie lange ein Prozess bisher gerechnet hat, wieviel Haupt- bzw. Plattenspeicher er belegt usw. Diese Information kann man dazu benutzen, den Benutzern ihre verbrauchten Ressourcen in Rechnung zu stellen. Dazu muss dann natürlich auch bei jedem Prozess vermerkt sein, welchem Benutzer er zugeordnet ist.

Interne Verwaltungsinformationen. Außerdem merkt sich das Betriebssystem im Prozesskontrollblock Daten zu den Prozessen, anhand derer es später entscheiden kann, welchem Prozess es den Prozessor als nächsten zuteilen will (z. B. Prozessprioritäten). Oder es speichert dort auch einen Zeiger auf den nächsten Prozess, der in demselben Zustand ist. So kann das Betriebssystem dann einfach eine Liste aller bereiten Prozesse verwalten.

Prozesskontrollblock Alle diese Daten, die das Betriebssystem über einen Prozess verwalten muss, bezeichnet man als **Prozesskontrollblock** oder kurz *PCB* vom englischen Begriff process control block. Im Prozesskontrollblock stehen also folgende Informationen:

- Eine eindeutige Prozessnummer und der Prozesszustand.
- Informationen über den Speicherbereich, in dem der Prozess abläuft.
- Der Programmzählerstand, an dem der Prozess zuletzt unterbrochen wurde.
- Die Inhalte der Prozessorregister zum Zeitpunkt der Unterbrechung.
- Informationen über alle geöffneten Dateien des Prozesses.
- Abrechnungs- und Statistik-Informationen.
- Interne Verwaltungsinformationen für das Betriebssystem.

Diese Aufzählung ist nicht vollständig. Je nach Betriebssystem stehen im Prozesskontrollblock mehr oder weniger Informationen. Das Programm, der Stack, die Daten und der Prozesskontrollblock eines Prozesses werden zusammen als **Prozessabbild** bezeichnet.

3.3.3 Prozessumschaltung

Der Teil des Betriebssystems, der für das Umschalten des Prozessors zwischen den Prozessen zuständig ist, nennt man **Dispatcher**. Seine Aufgaben sind:

- Anhalten des rechnenden Prozesses. Der Prozessor ist dem Prozess entzogen und im Moment dem Betriebssystem zugeordnet.
- Sichern der Informationen über den bisher rechnenden Prozess in dessen Prozesskontrollblock.
- Aus dem Kontrollblock des nächsten Prozesses dessen alten Zustand wiederherstellen. Der nächste Prozess stammt aus der Menge der Prozesse im Zustand *bereit*.
- Den Prozessor an den neuen rechnenden Prozess übergeben, damit dieser in den Zustand *rechnend* kommt.

Bei heute üblichen Systemen schaltet das Betriebssystem den Prozessor mehrmals pro Sekunde zwischen verschiedenen Prozessen hin und her. Es ist klar, dass der Dispatcher deshalb so schnell wie möglich arbeiten muss. Da der Dispatcher auch direkt auf die Prozessorregister zugreifen muss, ist er in der Regel in der Maschinensprache (Assembler) programmiert. Aus Effizienzgründen legt man auch fest, dass der nächste Prozess, der den Prozessor zugeteilt bekommen soll, immer an einer festgelegten Adresse steht. Welcher Prozess als nächster den Prozessor erhalten soll, wird vom Scheduler festgelegt, siehe dazu Abschnitt Abschnitt 3.4.

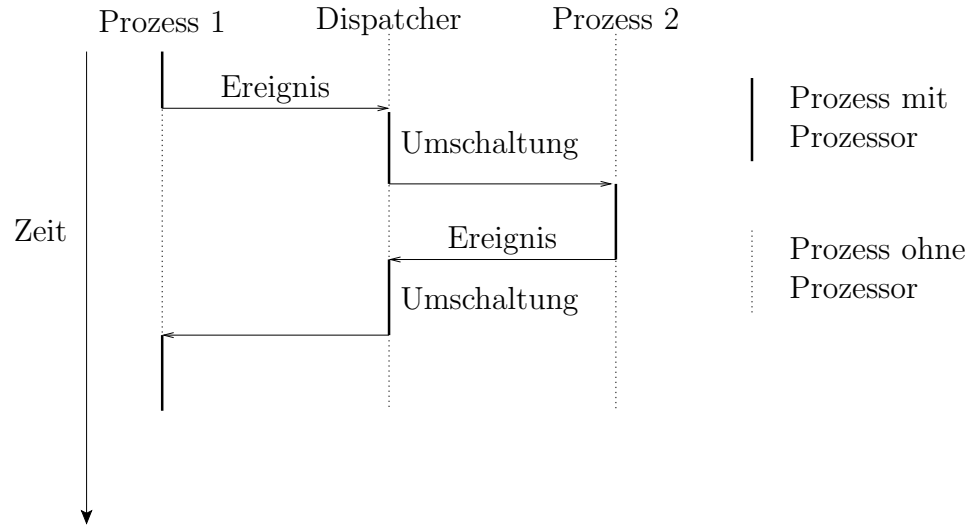


Abbildung 3.4: Ablauf der Prozessorumschaltung.

In Systemen, die nach der non-preemptive-Prozessumschaltung arbeiten, wird der Dispatcher vom vorher rechnenden Prozess aufgerufen. Bei Systemen mit preemptive-Prozessumschaltung ist zunächst nicht klar, wie einem rechnenden Prozess der Prozessor gegen dessen Willen entzogen werden kann.

Wie kann das Betriebssystem also einem rechnenden Prozess den Prozessor entziehen, obwohl es den Prozessor selbst gar nicht hat? Die Lösung dieses Problems bietet die Hardware. Moderne Rechner haben einen *timer* (Zeitschalter), der jeweils nach einem bestimmten Intervall (z. B. 100 ms) eine Unterbrechung (Interrupt) erzeugt. Wie schon in Kurseinheit 1 angesprochen, wird dann eine Unterbrechungsroutine des Betriebssystems aufgerufen. Jetzt hat das Betriebssystem also den Prozessor und kann dann den Dispatcher aufrufen und dadurch einem anderen Prozess den Prozessor zuteilen.

Eine Prozessumschaltung findet immer dann statt, wenn

- ein Prozess eine Ein- oder Ausgabe durchführen will oder auf ein anderes Ereignis warten muss. Der Prozess geht dann vom Zustand *rechnend* in den Zustand *blockiert* über.
- ein Prozess den Prozessor freiwillig abgeben will oder die Zeitscheibe abgelaufen ist, siehe dazu auch Abschnitt Abschnitt 3.4.3.
- ein Prozess mit der Abarbeitung seines Programms fertig ist und daher in den Zustand *beendet* übergeht.

- ein neuer Prozess erzeugt wurde.

- ein Ereignis eintritt, auf das ein Prozess im Zustand *blockiert* wartet. Der bisher blockierte Prozess wechselt in den Zustand *bereit* und bewirbt sich mit um den Prozessor.

3.3.4 Ursachen für Zustandswechsel und fällige Aktionen

Nachdem nun die Realisierung der Zustandsübergänge zwischen den Zuständen *rechnend* und *bereit* erklärt wurde, sollen nun auch die Aktionen des Dispatchers bei den anderen Zustandsübergängen erklärt werden.

Wir fassen die Gründe für den Übergang und die vom Betriebssystem dann durchgeführten Aktionen in Tabellen zusammen. Die Aktionen sind nicht in einer konkreten Programmiersprache formuliert, sondern in Umgangssprache. Einige der Aktionen lassen sich nur in Maschinensprache formulieren (Registerinhalte sichern), während man andere Aktionen auch in höheren Programmiersprachen beschreiben kann.

Übergang nach *erzeugt* **und weiter nach** *bereit*

Ursachen	Aktionen
• Der Benutzer hat ein neues Programm gestartet. • Ein laufender Prozess hat einen neuen Prozess erzeugt.	• Der neue Prozess bekommt eine eindeutige Identifikation (*Prozessnummer, Prozess-ID*). • Dem Prozess wird vom Betriebssystem ein logischer Adressraum zugeordnet. • Weitere, vom Prozess benötigte Ressourcen werden bereitgestellt. • Der Prozess wird in die Bereit-Liste eingetragen.

Übergang von *bereit* **nach** *rechnend*

Ursachen	Aktionen
• Der Scheduler hat den Prozess ausgewählt.	• Aus dem Prozesskontrollblock des Prozesses wird sein alter Zustand wiederhergestellt. • Der Prozess wird aus der Bereit-Liste entfernt und in die Rechnend-Liste eingetragen. • Der Prozess erhält den Prozessor und wird ausgeführt.

Übergang von *rechnend* **nach** *bereit*

Ursachen	Aktionen
• Der Prozess gibt den Prozessor freiwillig ab. • Die Zeitscheibe (siehe Abschnitt 3.4.3) des Prozesses ist abgelaufen und das Betriebssystem entzieht dem Prozess den Prozessor. • Ein Ereignis ist eingetreten, z. B. eine Benutzereingabe oder ein neuer Prozess wurde erzeugt.	• Sichern des Prozesszustands des rechnenden Prozesses in seinem Prozesskontrollblock. • Entfernen des Prozesses aus der Rechnend-Liste und Eintragen in die Bereit-Liste. • Aufruf des Schedulers, damit dieser einen neuen Prozess auswählt, der den Prozessor erhalten soll.

Übergang von *rechnend* **nach** *blockiert*

Ursachen	Aktionen
• Der rechnende Prozess wartet auf ein Ereignis.	• Sichern des Prozesszustands in seinem Prozesskontrollblock. • Entfernen des Prozesses aus der Rechnend-Liste. • Prozess wird in die entsprechende Blockiert-Liste eingetragen. • Aufruf des Schedulers

Übergang von *blockiert* **nach** *bereit*

Ursachen	Aktionen
• Das Ereignis ist eingetreten, auf das der Prozess wartete.	• Sichern des Zustands des rechnenden Prozesses in dessen Prozesskontrollblock. • Suchen des Prozesses, der auf das Ereignis gewartet hat, Entfernen dieses Prozesses aus der Blockiert-Liste und Eintragen in die Bereit-Liste. • Entweder weiterrechnen des bisher rechnenden Prozesses oder bisher rechnenden Prozess in Bereit-Liste Eintragen und Scheduler aufrufen.

Übergang von *rechnend* **nach** *beendet* **und aus dem System**

Ursachen	Aktionen
• Der Prozess ist am Ende seines Programms angekommen.	• Entfernen des Prozesses aus der Rechnend-Liste. • Vom Prozess belegte Ressourcen freigeben. • Aufruf des Schedulers.

3.4 Scheduling

In den bisherigen Abschnitten wurde dargestellt, wie das Betriebssystem den Prozessor zwischen zwei Prozessen umschalten kann. Die Frage, die sich als nächstes stellt, ist: Welchem Prozess soll das Betriebssystem denn als nächstes den Prozessor zuteilen? Auf einem großen Rechner mit mehreren Benutzern hat das Betriebssystem die Qual der Wahl, welcher der vielen Prozesse aus dem Zustand *bereit* als nächster den Prozessor zugeteilt bekommen soll.

Der Teil des Betriebssystems, der diese Aufgabe erfüllt, wird **Scheduler** genannt. Man spricht jedoch nicht nur im Zusammenhang mit der Prozessorzuteilung von Scheduling. Dieser Begriff wird immer dann verwendet, wenn ein knappes Betriebsmittel (in unserem Fall der Prozessor) einer Reihe von Bewerbern (in unserem Fall den Prozessen) gegenüber steht, und eine Auswahl unter den Bewerbern getroffen werden muss.

Ein von-Neumann-Rechner kann ein Programm, bzw. einen Prozess nur dann abarbeiten, wenn das Programm im Hauptspeicher des Rechners steht. Bisher sind wir stillschweigend davon ausgegangen, dass alle Prozesse bereits im Hauptspeicher stehen. In der Realität ist das natürlich nicht der Fall. Programme sind auf Sekundärspeichern, i. A. Festplatten gespeichert. Wenn ein Benutzer also einen neuen Prozess startet, der ein bestimmtes Programm ausführen soll, so muss zunächst das Programm vom Sekundärspeicher in den Hauptspeicher geladen werden. In Abhängigkeit von der gewählten Hauptspeicherverwaltung (siehe Kurseinheit 4) gibt es hier verschiedene Möglichkeiten. Wenn ein Programm immer komplett im Hauptspeicher stehen muss, stellt sich wiederum ein Scheduling-Problem.

Welcher Prozess soll als nächstes in den Hauptspeicher geladen werden? Ein **Long-Term Scheduler** trifft diese Auswahl. Zur Zeit der Großrechner (60er und 70er Jahre) war dies der Operateur an der Konsole, der die Bänder in den Rechner einlegte. Da man in moderneren Betriebssystemen darauf verzichtet, immer das komplette Programm im Hauptspeicher zu haben, stellt sich dieses Scheduling-Problem in dieser Form heute nicht mehr.

Im Unterschied dazu bezeichnet man den Scheduler, der aus der Menge der Prozesse im Zustand *bereit*[4] den nächsten rechnenden Prozess auswählt, als **Short-Term Scheduler**.

In diesem Abschnitt werden einige Strategien vorgestellt, nach denen diese Auswahl getroffen werden kann. Um die verschiedenen Strategien besser vergleichen und bewerten zu können, stellen wir zunächst einige Bewertungsmaßstäbe vor. Die Vor- und Nachteile der einzelnen Maßstäbe werden dann im Anschluss an die Strategien diskutiert.

3.4.1 Qualitätsmaßstäbe von Scheduling-Strategien

Prozessorauslastung. Früher waren Rechner sehr teuer und mit das teuerste Bauteil war der Prozessor. Man optimierte die Scheduler also daraufhin, den Prozessor möglichst gut auszulasten, um möglichst viel der teuren Rechenzeit zu nutzen.

[4] Diese Prozesse, bzw. die entscheidenden Teile des Prozesses befinden sich schon im Hauptspeicher.

Heute ist dieses Kriterium (hohe **Prozessorauslastung**) nicht mehr ganz so wichtig. Prozessoren stehen in großer Zahl und preiswert zur Verfügung.

Es werden heute z. B. spezielle Rechner, sog. X-Displays hergestellt, deren Prozessor ausschließlich die Aufgabe hat, das graphische Fenstersystem auf diesem Rechner anzuzeigen. Die eigentliche Rechenarbeit wird auf einem anderen Rechner im Netz ausgeführt. Der Prozessor im X-Display wird auch dann nicht zu dieser Rechenarbeit herangezogen, wenn er nicht mit dem Anzeigen der Fenster beschäftigt ist.

Antwortzeit. Wenn man als einer von vielen Benutzern an einem Rechner sitzt, wünscht man sich möglichst sofortige Reaktionen des Rechners auf eigene Eingaben. Eine kurze **Antwortzeit**, die das Zeitintervall zwischen Eingabe des Benutzers und Reaktion des Systems ist, für interaktive Benutzer ist also das Ziel des Scheduling-Algorithmus. Da heute immer mehr Aufgaben interaktiv am Rechner erledigt werden, ist dieses Kriterium recht wichtig.

Durchlaufzeit. Hat man jedoch kein interaktives Programm, sondern sogenannte *batch-Programme* laufen, die z. B. Compiler, große Simulationen wie die Wettervorhersage oder auch längere Berechnungen aus dem Maschinenbau sein können, dann will man das fertige Ergebnis möglichst schnell bekommen. Die minimale **Durchlaufzeit**, die das Zeitintervall zwischen Start und Ende des Prozesses ist, ist hier das anzustrebende Ziel.

Durchsatz. Aus kaufmännischer Sicht betrachtet, möchte man, dass der Rechner möglichst viel leistet, d. h. möglichst viel Arbeit in vorgegebener Zeit erledigt. Dieses Kriterium nennt man **Durchsatz**. Das Ziel des Schedulers ist also ein möglichst hoher Durchsatz, d. h. möglichst viele Aufträge sollen in einem bestimmten Zeitintervall bearbeitet werden. In diesem Fall ist der Begriff Auftrag mit einem Prozess gleichzusetzen.

Fairness. Als einer unter vielen Benutzern will man natürlich auch, dass es bei der Verteilung der Prozessorkapazität gerecht zu geht. Niemand will u. U. ewig auf die Bearbeitung seines Auftrags warten, nur weil ständig jemand kommt und sich vordrängelt.

Die folgende Liste zeigt noch einmal die bisher genannten Qualitätsmerkmale von Scheduling-Algorithmen:

- Maximale Effizienz, d. h. hohe Prozessorauslastung
- Minimale Antwortzeiten
- Minimale Durchlaufzeiten
- Maximaler Durchsatz
- Fairness, d. h. gerechte Verteilung des Prozessors

Alle genannten Kriterien (außer der Fairness) ändern ihre Werte im Laufe der Zeit. Die Prozessorauslastung ist manchmal recht niedrig, dann wiederum sehr hoch. Was soll also in diesem Zusammenhang der Begriff *maximale Effizienz* bedeuten?

In der Regel wird man versuchen, die *Durchschnittswerte* der Kriterien zu optimieren. Die durchschnittliche Antwortzeit soll minimal werden, die durchschnittliche Auslastung maximal. Bei zeitkritischen Systemen will man allerdings nicht eine minimale durchschnittliche Antwortzeit, sondern man will höchstens einen bestimmten Zeitraum warten. Das *Maximum* der Antwortzeit soll also minimiert werden.

Es gibt Vorschläge, bei interaktiven Systemen die *Abweichung* von der durchschnittlichen Antwortzeit zu minimieren. Dadurch erhält man ein System, bei dem man die Antwortzeit gut vorhersagen kann. Für unerfahrene Benutzer ist dies u. U. eine große Hilfe, da sie sich sonst oft fragen: Was macht der Rechner so lange? Habe ich etwas falsch gemacht?

Optimale Scheduling-Algorithmen. Einen optimalen Scheduling-Algorithmus, der alle genannten Qualitätsmerkmale erfüllt, kann es aber nicht geben. Sorgt ein Scheduler für minimale Antwortzeiten, indem er alle batch-Programme nur dann laufen lässt, wenn kein interaktiver Benutzer am Rechner ist, so verfehlt er das Ziel der minimalen Durchlaufzeit. Benutzer, die auf das Ende ihres batch-jobs warten, werden benachteiligt und müssen vielleicht länger als nötig warten. Rechenzeit, die einer Benutzergruppe zugeschlagen wird, muss einer anderen Benutzergruppe weggenommen werden.

CPU burst. Bei der Betrachtung der Scheduling-Algorithmen ist nun nicht die komplette Rechenzeit eines Prozesses von Bedeutung. Während ihrer Abarbeitung geben viele Prozesse den Prozessor von sich aus ab, z. B. indem sie blockieren. Daher ist für das Scheduling nur die Zeit interessant, die ein Prozess den Prozessor am Stück behalten will. Diese Zeit wird **CPU burst** genannt. Interaktive Programme, die ständig Benutzereingaben erfordern, haben einen niedrigen CPU burst. Nach relativ kurzer Rechenzeit gehen sie in den Zustand *blockiert* über und geben dadurch den Prozessor wieder frei. Programme, die nur lange Berechungen ausführen, z. B. ein lineares Gleichungssystem mit 1000 Unbekannten zu lösen, haben dagegen einen großen CPU burst.

Wir gehen im Folgenden davon aus, dass die Prozesse ihre angegebene **Bedienzeit**, das ist die Zeit, in der sie den Prozessor brauchen, an einem Stück haben wollen, d. h. Bedienzeit = CPU burst. Mit **Wartezeit** bezeichnen wir die Zeit zwischen dem Eintreffen im System und der Prozessorzuteilung. Außerdem zählt die Zeit im Zustand *bereit* mit zur Gesamtwartezeit, da der Prozess auch dann auf den Prozessor wartet. Die **Durchlaufzeit** ist die Summe aus Wartezeiten und Bedienzeiten eines Prozesses.

3.4.2 Non-preemptive Scheduling

3.4.2.1 First Come First Served (FCFS)

Eine einfache Scheduling-Strategie ist es, die Prozesse in der Reihenfolge ihrer Ankunft im System abzuarbeiten.

Zur Beschreibung dieses und auch der folgenden Verfahren soll immer wieder das folgende Beispiel herangezogen werden:

Gegeben sind die Prozesse P_1, P_2, P_3, P_4 und P_5. Sie treffen in dieser Reihenfolge zur selben Zeit im System ein. Die Bedienzeit der Prozesse ist in der folgenden

Tabelle angegeben:

Prozess	P_1	P_2	P_3	P_4	P_5
Bedienzeit	22	2	3	5	8

Beim First Come First Served Scheduling werden die Prozesse in der Reihenfolge: P_1, P_2, P_3, P_4, P_5 abgearbeitet.

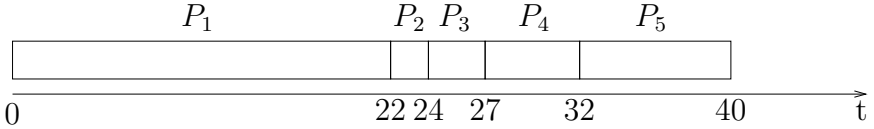


Abbildung 3.5: Ablauf bei First Come First Served Scheduling

Die Antwortzeiten[5] der Prozesse sind:

$$\begin{aligned} R_1 &= 0 + 22 = 22 \\ R_2 &= 22 + 2 = 24 \\ R_3 &= 24 + 3 = 27 \\ R_4 &= 27 + 5 = 32 \\ R_5 &= 32 + 8 = 40 \end{aligned}$$

Die mittlere Antwortzeit beträgt also:

$$R_\varnothing = \frac{22 + 24 + 27 + 32 + 40}{5} = \frac{145}{5} = 29$$

Ein kurzer Prozess hinter einem *Langläufer* muss nicht nur absolut, sondern auch relativ zu seiner eigenen Rechenzeit länger warten, als ein *Langläufer* hinter einem kurzen Prozess. Für interaktive Benutzer sind die dabei entstehenden Antwortzeiten nicht zumutbar.

Dieser Scheduling-Algorithmus hat den Vorteil, dass er sehr leicht zu implementieren ist. Der Scheduler braucht alle bereiten Prozesse nur in einer normalen Warteschlange (queue) nach dem FIFO Prinzip zu verwalten. Die Komplexität der Operationen Einfügen und Entfernen eines Elementes aus dieser Struktur ist unabhängig von der Zahl der Prozesse, also konstant.

Die Vor- und Nachteile des First Come First Served Scheduling sind:

Vorteile

+ Einfach zu implementieren.
+ geringer Verwaltungsaufwand, da die Operationen in konstanter Zeit arbeiten.
+ Fairness, da alle Prozesse der Reihe nach den Prozessor erhalten.

Nachteile

– kurz laufende Prozesse müssen u. U. sehr lange warten, daher für Dialogsysteme *nicht* geeignet.

[5] In unserem Fall sind die Antwortzeiten auch gleichzeitig die Durchlaufzeiten.

3.4.2.2 Shortest Job First (SJF)

Eine andere Scheduling-Strategie ist es, immer dem Prozess mit der kürzesten Bedienzeit den Prozessor zuzuordnen. Haben mehrere Prozesse dieselbe Bedienzeit, so kann man für diese Prozesse die First Come First Served Strategie anwenden.

In unserem oben schon genannten Beispiel ergibt sich folgende Ausführungsreihenfolge: P_2, P_3, P_4, P_5, P_1.

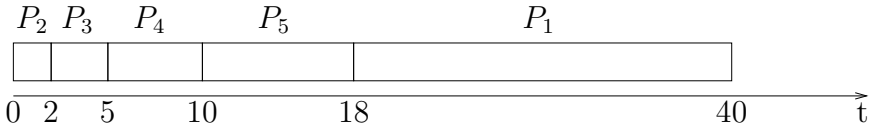


Abbildung 3.6: Ablauf bei Shortest Job First

Die Antwortzeiten der Prozesse sind:

$$\begin{aligned} R_1 &= 18 + 22 = 40 \\ R_2 &= 0 + 2 = 2 \\ R_3 &= 2 + 3 = 5 \\ R_4 &= 5 + 5 = 10 \\ R_5 &= 10 + 8 = 18 \end{aligned}$$

Daraus ergibt sich die mittlere Antwortzeit zu:

$$R_{\varnothing} = \frac{2 + 5 + 10 + 18 + 40}{5} = \frac{75}{5} = 15$$

Man kann beweisen, dass diese Strategie immer die mittlere Antwortzeit minimiert. Die Bedienzeit des ersten Prozesses (P_2) kommt bei allen folgenden Prozessen zu deren Wartezeit hinzu, d. h. in die durchschnittliche Antwortzeit geht diese Bedienzeit mit dem Faktor n (in unserem Fall 5) ein. Einmal taucht die Bedienzeit in der Antwortzeit des Prozesses selbst auf und dann in der Wartezeit der folgenden $n-1$ (hier 4) Prozesse.

Die zweite Bedienzeit (P_3) geht nur noch bei $n-1$ Prozessen, die dritte (P_4) nur noch bei $n-2$ Prozessen, ..., die letzte Bedienzeit nur noch bei einem Prozess in die Antwortzeit ein. Die mittlere Antwortzeit ist daher minimal, wenn die kleinsten Bedienzeiten bei den meisten Prozessen und umgekehrt, die größten Bedienzeiten bei den wenigsten Prozessen in die Antwortzeit eingeht.

Ein großer Nachteil dieser Strategie tritt auf, wenn z. B. alle 3 Zeiteinheiten ein neuer Prozess mit der Bedienzeit 3 in das System eintritt. Ein bereits wartender Prozess mit größerer Bedienzeit, wird *niemals* den Prozessor erhalten, da ja immer auch ein Prozess mit kürzerer Bedienzeit in der Warteschlange steht. Man sagt, dass der Prozess mit der längeren Bedienzeit dann **verhungert**. In der englischen Literatur wird auch der Begriff **starvation** benutzt.

Um diese Strategie zu implementieren, kann man z. B. eine sortierte Liste für die bereiten Prozesse vorsehen. Das Entnehmen des nächsten Prozesses ist dann in konstanter Zeit möglich. Neu eintreffende Prozesse müssen dagegen einsortiert werden, was zu einer linearen Zeitkomplexität führt. An Stelle der linearen Liste kann man allerdings auch einen Heap benutzen. Das Einfügen eines Prozesses und auch das Entfernen sind dann in $O(\log n)$ möglich.

Das größte Problem bei der Implementierung dieser Scheduling-Strategie liegt darin, die Bedienzeit der Prozesse zu kennen. Früher musste man als Benutzer diesen Wert selbst schätzen und bei der Abgabe des Programms mitteilen. Beim Long-Term Scheduling ist diese Strategie also relativ leicht einsetzbar. Die Benutzer können bei immer wieder auftretenden Problemen, z. B. Lösung eines Gleichungssystems mit 200 Unbekannten, die Bedienzeit dieses Prozesses sehr gut vorhersagen. Bei unbekannten Programmen oder neuen Problemgrößen, bleibt aber auch dem erfahrenen Benutzer dann keine andere Möglichkeit, als zu raten.

Beim Short-Term Scheduling ist das Verfahren nicht so ohne weiteres anwendbar. Es gibt keine Möglichkeit, den Prozessen im Zustand *bereit* anzusehen, wie lang deren nächster CPU burst sein wird. Daher versucht man diesen Wert möglichst gut vorherzusagen. Anhaltspunkt für diese Vorhersage ist das bisherige Verhalten des Prozesses. Dabei geht man davon aus, dass ein Prozess, der bisher immer nur kurze CPU bursts hatte, auch in Zukunft mit großer Wahrscheinlichkeit nur kurze CPU bursts haben wird. Ein Prozess mit großen CPU bursts hat auch in Zukunft wahrscheinlich wieder große CPU bursts.

Man benutzt dann die folgende Formel:

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

Dabei ist τ_n die Schätzung des n-ten CPU bursts und t_n ist die tatsächliche Länge des n-ten CPU bursts. $0 \leq \alpha \leq 1$ ist ein Gewichtungsfaktor. Für $\alpha = 1$ ist nur der tatsächliche letzte CPU burst für die neue Schätzung maßgebend. Bei $\alpha = 0$ geht nur die letzte Schätzung in die neue Schätzung ein.

Vorteile	**Nachteile**
+ Garantiert minimale durchschnittliche Antwortzeit.	– Langläufer müssen eventuell ewig warten.
+ Kurz laufende Prozesse werden *nicht* benachteiligt.	– Die Dauer des nächsten CPU bursts ist nicht bekannt und kann nur geraten bzw. geschätzt werden.
+ Algorithmus ist relativ leicht zu implementieren, falls die Dauer des nächsten CPU bursts bekannt ist.	– größerer Verwaltungsaufwand $O(\log n)$.

Übungsaufgabe 3.2 Gegeben sind die folgenden CPU bursts eines Prozesses: 4, 4, 4, 4, 6, 8, 10, 12, 12, 12, 12. Bestimmen Sie für $\alpha = 1/2$ und $\alpha = 3/4$ die Folgeschätzwerte, wenn die erste Schätzung 6 war.

3.4.2.3 Priority Scheduling

Die Idee des *Priority Scheduling* ist, dass man die Prozesse in unterschiedlich wichtige Klassen einteilt. Zuerst bekommen die Prozesse der wichtigsten Klasse (Prozesse mit der größten **Priorität**) den Prozessor zugeteilt, dann erst die der weniger wichtigen Klassen.

Prioritäten können dabei *intern*, d. h. vom Betriebssystem selbst, oder *extern*, d. h. von außen, vorgegeben werden. Interne Prioritäten können z. B. vom Hauptspeicherbedarf, Plattenspeicherbedarf, der Anzahl der geöffneten Dateien, der bisher verbrauchten Rechenzeit usw. abhängen.

Externe Prioritäten sind häufig mit der Person des Benutzers verbunden. In einem Unternehmen hätten die Prozesse des Vorstands höchste Priorität, dann die Prozesse der Abteilungsleiter, Gruppenleiter, ... bis zuletzt die Prozesse der einfachen Mitarbeiter kommen. Eine andere Möglichkeit die Prioritäten zu vergeben liegt in der Bezahlung. Wenn die Benutzer für ihre Prozesse bezahlen müssen, so kann man die Preise nach Prioritäten staffeln. Wer es eilig hat, bekommt eine hohe Priorität für seine Prozesse und bezahlt dafür einen höheren Preis.

Die schon besprochene Shortest Job First Strategie kann man auch als eine Prioritäts-Strategie interpretieren. Je kleiner die Bedienzeit eines Prozesses ist, desto höher ist seine Priorität und umgekehrt.

In der Regel werden die Prioritäten durch natürliche Zahlen dargestellt, meist durch Zahlen aus einem festen Intervall. Einige Betriebssysteme verwenden kleine Zahlen für niedrige Priorität und große Zahlen für hohe Priorität. Es gibt aber auch Betriebssysteme, bei denen es gerade umgekehrt ist (kleine Zahl entspricht hoher Priorität). Im Kurstext sollen kleine Zahlen einer hohen Priorität entsprechen.

Wenn man die Prioritäten *statisch* festlegt, d. h. jeder Prozess bekommt eine Priorität zugeordnet und behält diese Priorität bis zu seinem Ende, hat man Probleme, den Prozessor gerecht zu verteilen. Prozesse mit niedriger Priorität müssen u. U. ewig warten, falls ständig neue Prozesse mit hoher Priorität im System ankommen. Man kann dies verhindern, indem die Priorität der Prozesse *dynamisch* angepasst wird, siehe Feedback Scheduling.

Bei der Implementierung dieser Strategie braucht man wieder eine sortierte Liste, bzw. einen Heap. Das Einfügen und Entfernen eines Prozesses hat daher im Fall der sortierten Liste die Komplexität $O(n)$ und $O(1)$, bzw. beim Heap jeweils $O(\log n)$.

Übungsaufgabe 3.3 Erklären Sie, wie man einen Heap anstelle der sortierten Liste benutzt. Beschreiben Sie, wie der Prozess mit der größten Priorität entnommen wird, und wie ein neuer Prozess eingefügt wird.

Vorteile	Nachteile
+ *Wichtige* Aufgaben werden schnell erledigt	– Bei statischer Prioritätsvergabe *nicht* fair, da Prozesse dann aushungern können.
	– Wie die Prioritäten festgelegt werden sollen, ist nicht so ohne weiteres klar.

Non-preemptive Scheduling-Strategien sind für den Dialogbetrieb nicht so gut geeignet. Ein Benutzer, bzw. ein Programm kann die Rechnerbenutzung für alle anderen unmöglich machen, indem es den Prozessor einfach nicht mehr abgibt. Die anderen Benutzer können dem nur tatenlos zusehen.

Daher benutzt man im Dialogbetrieb nur Scheduling Verfahren, die solchen *unfairen* Programmen den Prozessor auch wegnehmen können.

3.4.3 Preemptive Scheduling

3.4.3.1 Round Robin

Eine für den interaktiven Betrieb (Dialogbetrieb) besser geeignete Scheduling-Strategie ist *Round Robin*. Die bereiten Prozesse werden, wie bei First Come First Served, in einer Warteschlange verwaltet. Der erste Prozess dieser Schlange bekommt als nächster den Prozessor zugeteilt.

Der rechnende Prozess darf den Prozessor jedoch nur für eine bestimmte Zeit behalten. Diese Zeit nennt man **Zeitscheibe** (engl. **time slice**) oder auch **Quantum**. Hat der rechnende Prozess den Prozessor nach dem Ablauf seiner Zeitscheibe noch nicht freigegeben, so wird er vom Betriebssystem unterbrochen und kommt dann an das *Ende* der Warteschlange. Dort werden auch neu entstandene Prozesse eingereiht.

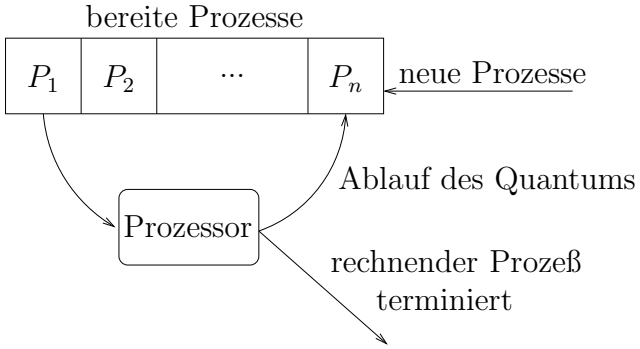


Abbildung 3.7: Schema bei Round Robin Scheduling.

Die Größe der Zeitscheibe ist ein sorgfältig zu wählender Parameter dieser Strategie. Jeder Prozesswechsel dauert ja auch eine gewisse Zeit. Wählt man das Quantum q zu klein, z. B. nur so groß wie die Prozesswechselzeit, passiert folgendes: Ein Prozess rechnet q Zeiteinheiten, und dann findet der Prozesswechsel statt, der wiederum q Zeiteinheiten benötigt. Es sind $2q$ Zeiteinheiten vergangen, in denen der Prozessor aber nur q Zeiteinheiten für die Benutzerprozesse zur Verfügung stand. Die andere Hälfte der Zeit war der Prozessor mit der Verwaltungsarbeit des Betriebssystems (Prozessumschaltung) beschäftigt. Die Prozessorauslastung ist, was die eigentlich zu leistende Arbeit betrifft, zu gering.

Wählt man das Quantum allerdings zu groß, ändert sich zwar das Verhältnis echter Arbeit zu Verwaltungsaufwand zugunsten der echten Arbeit, aber die Wartezeiten für die interaktiven Benutzer werden unzumutbar groß. Falls ein Prozess gerade am Ende der Warteschlange steht und alle Prozesse vorher das größere Quantum komplett ausnutzen, können schnell Wartezeiten im Bereich von mehreren Sekunden auftreten. Man muss also für das Quantum einen geeigneten (mittleren) Wert finden. Eine sinnvolle Größe für das Quantum ist ein Wert, der etwas größer als die für eine übliche Interaktion erforderliche Zeit ist, typischerweise in Millisekunden-Größenordnungen.

Abbildung 3.8 zeigt den zeitlichen Ablauf unserer 5 Beispielprozesse, wenn das Scheduling nach der Round Robin Strategie vorgenommen wird. Das gewählte Zeitquantum beträgt 3 Zeiteinheiten. Die Prozessumschaltzeit ist 0 Zeiteinheiten.

Wenn ein Prozess sein Quantum nicht bis zum Ende ausnutzt, findet natürlich sofort ein Prozesswechsel statt. Wenn dagegen nur noch ein rechenbereiter Prozess

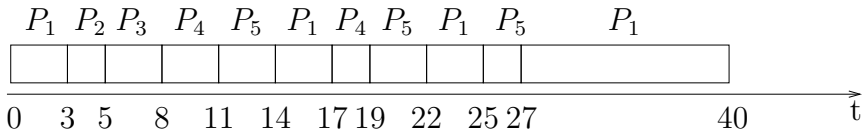


Abbildung 3.8: Ablauf bei Round Robin.

vorhanden ist, braucht dieser nicht nach jedem Ablauf des Quantums unterbrochen zu werden, nur damit er den Prozessor sofort wieder erhält. Der genaue Ablauf ist in folgender Tabelle noch einmal dargestellt:

t	Warteschlange	Restzeit					Kommentar
		P_1	P_2	P_3	P_4	P_5	
0	P_1, P_2, P_3, P_4, P_5	22	2	3	5	8	
3	P_2, P_3, P_4, P_5, P_1	19	2	3	5	8	
5	P_3, P_4, P_5, P_1	19	-	3	5	8	P_2 fertig
8	P_4, P_5, P_1	19	-	-	5	8	P_3 fertig
11	P_5, P_1, P_4	19	-	-	2	8	
14	P_1, P_4, P_5	19	-	-	2	5	
17	P_4, P_5, P_1	16	-	-	2	5	
19	P_5, P_1	16	-	-	-	5	P_4 fertig
22	P_1, P_5	16	-	-	-	2	
25	P_5, P_1	13	-	-	-	2	
27	P_1	13	-	-	-	-	P_5 fertig
40		-	-	-	-	-	P_1 fertig

Die Antwortzeiten der einzelnen Prozesse sind:

$$R_1 = 40$$
$$R_2 = 5$$
$$R_3 = 8$$
$$R_4 = 19$$
$$R_5 = 27$$

Die mittlere Antwortzeit liegt also bei:

$$R_\oslash = \frac{5 + 8 + 19 + 27 + 40}{5} = \frac{99}{5} = 19{,}8$$

Verglichen mit der Strategie First Come First Served, hat sich die mittlere Antwortzeit erheblich verbessert. Die Benachteiligung kurzer Prozesse findet bei diesem Verfahren nicht statt. Langlaufende Prozesse haben bei dieser Scheduling-Strategie sehr große Durchlaufzeiten, da kurzen Rechenzeiten (q) immer wieder lange Wartezeiten (Anzahl der Prozesse $\times$ q) gegenüber stehen.

Zur Implementierung des Verfahrens reicht wieder eine *normale* Warteschlange aus. Die Zeitkomplexität für das Einfügen und Entfernen eines Prozesses ist $O(1)$.

Da alle Prozesse gleich behandelt werden (jeder bekommt sein Quantum des Prozessors), erfüllt diese Strategie das Qualitätsmerkmal der Fairness besonders gut.

Vorteile	Nachteile
+ Fairness, jeder Prozess bekommt seinen Anteil am Prozessor.	– Langlaufende Prozesse müssen bis zu ihrem Ende recht lange warten.
+ Kurz laufende Prozesse werden *nicht* benachteiligt.	
+ Einfach zu implementieren.	
+ Wenig Verwaltungsaufwand.	

3.4.3.2 Shortest Remaining Time First

Die vorhin besprochene Shortest Job First Strategie lässt sich einfach zu einer preemptive Strategie erweitern, bei der auch rechnende Prozesse vom System unterbrochen werden können. Die Idee dabei ist, dass jedesmal, wenn ein neuer Prozess eintrifft, der rechnende Prozess angehalten wird. Die noch verbleibende Bedienzeit dieses Prozesses wird zu seiner neuen Bedienzeit. Dann wird wieder unter allen bereiten Prozessen, der Prozess mit der kleinsten Bedienzeit ausgewählt.

Anhand des folgenden Beispiels soll diese Strategie erklärt werden. Gegeben sind wiederum 5 Prozesse P_1, P_2, P_3, P_4 und P_5. Ihre Ankunftzeit im System und ihre Bedienzeit ist durch folgende Tabelle gegeben:

Prozess	P_1	P_2	P_3	P_4	P_5
Bedienzeit	22	2	3	5	8
Ankunftzeit	0	0	4	4	4

Eine non-preemptive Shortest Job First Strategie führt die Prozesse in der Reihenfolge P_2, P_1, P_3, P_4 und P_5 aus. Nachdem zum Zeitpunkt 0 nur die Prozesse P_1 und P_2 im System sind, wird der kürzere der beiden gestartet. Zum Zeitpunkt 2 ist P_2 beendet und nur noch P_1 im System. Also wird jetzt P_1 rechnend. Die Antwortzeiten[6] der einzelnen Prozesse sind:

$$\begin{aligned} R_1 &= (\ 2-0) + 22 = 24 \\ R_2 &= (\ 0-0) + \ 2 = \ 2 \\ R_3 &= (24-4) + \ 3 = 23 \\ R_4 &= (27-4) + \ 5 = 28 \\ R_5 &= (32-4) + \ 8 = 36 \end{aligned}$$

Die durchschnittliche Antwortzeit beträgt also:

$$R_\oslash = \frac{24+2+23+28+36}{5} = \frac{113}{5} = 22{,}6$$

Bei einer Shortest Remaining Time First Strategie wird der zum Zeitpunkt 4 rechnende Prozess (P_1) dann unterbrochen, da neue Prozesse im System eintreffen. P_1 hat dann noch eine verbleibende Bedienzeit von $22 - 2 = 20$ Zeiteinheiten. Da dies die größte Bedienzeit ist, erhalten zuerst die anderen Prozesse den Prozessor. Der zeitliche Ablauf ist also wie folgt:

Die mittlere Antwortzeit ist also:

$$R_\oslash = \frac{40+2+3+8+16}{5} = \frac{69}{5} = 13{,}8$$

[6]Beachten Sie, dass P_3, P_4 und P_5 erst zum Zeitpunkt 4 im System eintreffen. Ihre Wartezeit ist also gleich der Differenz aus Startzeitpunkt und 4.

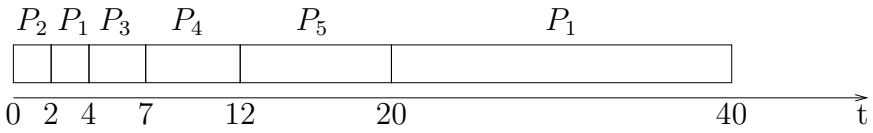


Abbildung 3.9: Ablauf bei preemptive Shortest Job First.

3.4.3.3 Priority Scheduling

Auch beim Priority Scheduling kann man sich eine preemptive Version analog zur Shortest Job First Strategie überlegen. Immer wenn ein neuer Prozess im System eintrifft, wird der rechnende Prozess unterbrochen, und danach erhält der Prozess mit der höchsten Priorität den Prozessor. Das kann dann entweder der bisher rechnende Prozess sein, falls der neue Prozess eine kleinere Priorität hatte, oder es ist der neue Prozess, der dann eine höhere Priorität als der bisher rechnende Prozess hat.

Übungsaufgabe 3.4 Gegeben sind die Prozesse $P_1, \ldots, P_5$. Sie treffen in dieser Reihenfolge zur selben Zeit im System ein. Die Prozesse haben folgende Bearbeitungszeiten: 15, 7, 1, 4, und 8. Geben sie für jeden der folgenden Scheduling Algorithmen die mittlere Antwortzeit R an.

- First Come First Served
- Shortest Job First
- Round Robin mit $Q = 4$.

3.4.4 Kombinationen der Scheduling-Strategien

Die bisher betrachteten Scheduling-Strategien hatten jeweils ihre Vor- und Nachteile. Durch gleichzeitige Anwendung verschiedener Strategien versucht man nun, die unterschiedlichen Vorteile verschiedener Strategien zu vereinigen und die Nachteile zu unterbinden.

3.4.4.1 Feedback Scheduling

Ein Nachteil des Round Robin Verfahrens ist die absolute Gleichbehandlung *aller* Prozesse. Dadurch müssen auch Prozesse, die ihr Quantum nicht komplett ausnutzen, sondern ständig wegen Ein-/Ausgaben blockieren, lange auf die nächste Prozessorzuteilung warten. Bei solchen Prozessen ist es aber besser, ihnen den Prozessor so schnell wie möglich wieder zu geben. Dann kann der Prozess schnell seine nächste Ein-/Ausgabe starten, er blockiert wieder, und die Ein-/Ausgabe kann parallel zur Abarbeitung der anderen Prozesse stattfinden. Siehe dazu auch Abschnitt 2.2.4 unter dem Stichwort *DMA*.

Man fasst Verfahren, die die Vergangenheit eines Prozesses bei der Auswahl des nächsten Prozesses berücksichtigen, unter dem Begriff **Feedback Scheduling** zusammen.

Beim Priority Scheduling kann man die Priorität der einzelnen Prozesse z. B. bei jeder Prozessumschaltung neu berechnen. Prozesse, die nicht *rechnend* waren, erhalten eine höhere Priorität. Ein Prozess kann jetzt nicht mehr ewig warten müssen, da

seine Priorität beim Warten ja ständig steigt. Irgendwann hat auch dieser Prozess dann die höchste Priorität und bekommt den Prozessor zugeteilt. In der Literatur ist dieses Verfahren der Prioritätsanpassung unter dem Stichwort **aging** bekannt.

Nimmt man nun noch das Round Robin Verfahren hinzu, so kann man folgende Strategie (rechenzeitabhängiges Feedback Scheduling) realisieren:

Neue Prozesse erhalten zunächst eine *hohe* Priorität und eine *kleine* Zeitscheibe. Hat der Prozess sein Quantum komplett genutzt, so erhält er eine niedrigere Priorität, aber sein Quantum verdoppelt sich. So müssen sehr lang laufende Prozesse nicht ständig ihren Zustand wechseln. Stattdessen kommen sie seltener an den Prozessor und dürfen ihn dafür dann etwas länger behalten. Kurz laufende Prozesse haben eine höhere Priorität und werden dadurch von den Langläufern nicht behindert.

Der Verwaltungsaufwand ist bei diesem Verfahren jedoch recht groß. Bei *jedem* Prozesswechsel *muss* die ganze Liste der bereiten Prozesse durchgesehen werden, damit die neuen Prioritäten bzw. Zeitscheiben vergeben werden können. Das hat zur Folge, dass die Zeitkomplexität der Operationen linear, d. h. $O(n)$ ist.

Vorteile	**Nachteile**
+ Prozesse werden nicht ein für allemal kategorisiert, sondern je nach ihrem Verhalten beurteilt.	– Höherer Verwaltungsaufwand, da Prozesse ständig neu beurteilt werden müssen.
+ Größere Gerechtigkeit, da kein Verhungern von Prozessen.	

Übungsaufgabe 3.5 Um keinen Prozess länger als 500ms warten zu lassen, programmiert ein Systementwickler das Round Robin Verfahren mit dynamischer Zeitscheibengröße. Bei n bereiten Prozessen wird die Zeitscheibe Q auf $500\text{ms}/n$ gesetzt. Was halten Sie von dieser Scheduling-Strategie?

3.4.4.2 Multiple Queues

Ein häufig benutztes Verfahren besteht darin, die Prozesse wie beim Priority Scheduling in verschiedene Klassen einzuteilen. Für jede der Klassen ist dann ein eigener Scheduler zuständig. Ein mögliche Einteilung könnte z. B. so aussehen:

1. Die Klasse der Systemprozesse. Sie hat die höchste Priorität 1. Innerhalb der Klasse werden die Prozesse nach dem First Come First Served Verfahren ausgewählt.

2. Die Klasse der Dialogprozesse. Sie hat mittlere Priorität 2. Innerhalb der Klasse werden die Prozesse nach dem Round Robin Verfahren ausgewählt.

3. Die Klasse der Hintergrundprozesse (batch jobs). Sie hat niedrigste Priorität($\geq$ 3). Innerhalb der Klasse kommt das rechenzeitabhängige Feedback Scheduling zur Anwendung.

Das „Gesamtscheduling“ kann nun so aussehen, dass Dialogprozesse erst dann *rechnend* werden können, wenn kein Prozess aus der Klasse der Systemprozesse mehr *bereit* ist. Und erst wenn weder System-, noch Dialogprozesse bereit sind, kann ein Hintergrundprozess den Prozessor erhalten.

Ein andere Möglichkeit des Gesamtscheduling ist es, die Prozessorzeit auf die verschiedenen Klassen zu verteilen. Man kann 60% der Zeit für Systemprozesse, 30% für die Dialogprozesse und 10% für die Hintergrundprozesse reservieren. Die Scheduler der einzelnen Klassen können dann diese Zeit an „ihre“ Prozesse vergeben. Wenn in einer Klasse keine rechenbereiten Prozesse mehr sind, wird die Zeit dieser Klasse natürlich den anderen Klassen hinzugeschlagen und verfällt nicht einfach.

Bei der gezeigten Klasseneinteilung (in System-, Dialog-, Hintergrundprozesse) wird jeder Prozess einmal in eine Klasse eingeordnet und bleibt dann für immer in dieser Klasse. Man kann auch hier Feedback Strategien anwenden, um Prozesse auch zwischen den Klassen wechseln zu lassen.

Ein Prozess, der bisher ständig Benutzereingaben verlangte, kann auf einmal eine lange Berechnung durchführen. Gehörte er also vorher zu den interaktiven Prozessen, so ist jetzt ein Hintergrundprozess aus ihm geworden. Dementsprechend sollte dieser Prozess beim Scheduling jetzt auch anders behandelt werden.

Insgesamt ergeben sich bei diesem Verfahren viele mögliche Scheduling Strategien, je nachdem:

- wieviele Klassen man definiert,
- welche Scheduling-Strategie man für die einzelnen Klassen vorsieht,
- nach welcher Strategie man die Klasse des nächsten Prozesses bestimmt,
- ob und wie Prozesse ihre Klasse ändern können,
- in welche Klasse neue Prozesse zuerst kommen.

Die Vor- und Nachteile dieser Strategie kann man also wie folgt zusammenfassen:

Vorteile	**Nachteile**
+ Sehr gute Differenzierung zwischen Prozessen.	– Hoher Verwaltungsaufwand für das Scheduling.
+ Aufgrund der vielen Parameter sehr vielseitig einstellbar.	– Aufgrund der vielen Parameter ist es sehr schwierig, eine gute Einstellung zu finden.

3.4.5 Auswahl einer Scheduling-Strategie

Wie kann man nun aus der Menge der möglichen Scheduling-Strategien, die optimale für ein konkretes System auswählen? In Abschnitt 3.4.1 wurden bereits einige Qualitätsmaßstäbe für die Scheduling-Strategien vorgestellt. Anhand dieser Maßstäbe kann man die verschiedenen Strategien vergleichen. Dazu gibt es verschiedene Vorgehensweisen.

Mathematischer Ansatz. Anhand von realen Daten misst man den CPU burst und die Ankunftzeiten von Prozessen. Aus diesen gemessenen Werten versucht man, die statistische Verteilung dieser Werte zu bestimmen. Man kann dann Formeln für die Wahrscheinlichkeit bestimmter Ankunftszeiten oder CPU bursts angeben. Aus diesen statistischen Verteilungen und Formeln kann man für die meisten Verfahren dann den durchschnittlichen Durchsatz, die Prozessorauslastung und die Wartezeit

berechnen. So kann man die Strategie bestimmen, die bei dieser Verteilung z. B. die kleinsten Wartezeiten erreicht.

Oder man nimmt ein Beispiel von Prozessen mit typischen Werten der einzelnen Parameter (siehe das Beispiel beim First Come First Served Scheduling) und wendet die verschiedenen Strategien auf dieses Beispiel an. Dann kann man direkt die Zahlenwerte für die Qualitätsmerkmale ausrechnen.

Wenn man dieses Verfahren automatisieren will, dann schreibt man ein Programm, das eine Simulation der Strategien durchführt. Für größere und dadurch realistischere Beispiele kann man das Verhalten der verschiedenen Strategien durchspielen und sich die Qualitätsmerkmale ausrechnen.

Interpretationsprobleme. Alle bisher genannten Verfahren liefern entweder Zahlenwerte oder eine Formel. Wie diese Werte jedoch zu interpretieren und bewerten sind, bleibt ein Problem. Man kann zwar die Verfahren relativ zueinander bewerten, aber eine absolute Beurteilung ist nicht möglich. Wie will man z. B. beurteilen, ob eine durchschnittliche Antwortzeit von 500 ms tolerierbar ist oder nicht? Vielleicht ist die Antwortzeit ausgerechnet bei den Prozessen, die dem Benutzer wichtig sind, wesentlich schlechter. Außerdem ist es schwierig, sich von den Werten wirklich eine konkrete Vorstellung zu machen. (Empfindet man z. B. 500 ms als lang?)

Ein weiteres Problem ist, dass man nur von einem Modell der Wirklichkeit ausgeht. Man hat bestimmte Annahmen gemacht, die nicht exakt mit der Realität übereinstimmen müssen.

Realer Test. Diese Probleme umgeht man, indem man die verschiedenen Strategien in einem existierenden Betriebssystem implementiert und so unter den realistischsten Bedingungen (normaler täglicher Betrieb) testen kann.

Auch hier hat man aber das Problem, dass die Benutzer ihr Verhalten an das neue System anpassen werden. Während der Tests verhalten sich die Benutzer aber noch wie vorher gewohnt. Wenn man als Benutzer aber einmal weiß, dass interaktive Prozesse bei der Prozessorvergabe stark bevorteilt werden, so wird man versuchen, auch die eigenen batch jobs wie interaktive Prozesse aussehen zu lassen. Man könnte z. B. einfach einige unsinnige Ausgaben machen, die man später einfach ignoriert. Das System könnte aber aufgrund der Ausgaben annehmen, dass es sich um einen interaktiven Prozess handelt und ihn bei der Vergabe des Prozessors bevorzugen.

Außerdem wird wahrscheinlich jeder Benutzer des Systems eine andere Rangfolge unter den Qualitätsmaßstäben haben. Dem einen sind optimale Durchlaufzeiten das Wichtigste, während andere Benutzer mehr Wert auf kurze Antwortzeiten legen. Allen Benutzern ist in der Regel zumindest der Wunsch nach schnellster Abarbeitung der eigenen Prozesse gemeinsam.

Zusammenfassend kann man also festhalten, dass es nicht einfach ist, für ein gegebenes System eine gute Scheduling-Strategie auszuwählen.

3.5 Vorteile und Probleme des allgemeinen Prozessmodells

Allgemeine Verwendbarkeit. Das bisher vorgestellte Modell der Prozesse ist sehr allgemein gehalten. Mit diesem Modell kann man parallele Aktivitäten auf einem Rechner sehr einfach beschreiben. Anhand eines Beispiels lässt sich das einfach nachvollziehen. Wenn ein Programmierer ein Programm erstellt, so braucht er dazu verschiedene Hilfsprogramme.

- Einen Editor, mit dem er den Programmtext schreibt.
- Einen Compiler, der das Programm in Maschinensprache übersetzt.
- Ein Programm, in dem er das Handbuch zum Betriebssystem ansehen kann.
- Einen Editor, mit dem er an der Dokumentation des Programms schreibt.
- Einen Debugger, unter dessen Kontrolle er den Ablauf seines Programms beobachten kann. So können Fehler im Programm einfacher gefunden werden.

Für jedes dieser Programme startet der Entwickler einen eigenen Prozess. Es ist z. B. sehr praktisch, wenn die Editor-Prozesse ständig laufen. Man kann jederzeit weiterschreiben, auch wenn gleichzeitig das Programm neu übersetzt wird. Zwischen den Tastendrücken im Editor wird der Prozessor dem Compiler-Prozess zugeteilt und kann dort weiter arbeiten. Wenn zwischendurch ein elektronischer Brief ankommt, so kann man einfach einen neuen Prozess starten, in dem der Brief gelesen wird.

Hoher Verwaltungsaufwand. Ein Prozess ist völlig getrennt von anderen Prozessen, er hat seinen eigenen Speicherbereich, eigene geöffnete Dateien, usw. Die Datenmenge, die das Betriebssystem im Prozesskontrollblock verwalten muss, ist recht groß.

Der Speicherbereich, der einem Prozess zugeordnet ist besteht in der Regel aus drei Segmenten, siehe dazu auch Abschnitt 3.3 auf Seite 74:

- Programmsegment, mit dem ausführbaren Programmcode.
- Stacksegment, mit dem Stapel der bei Prozeduraufrufen entsteht.
- Datensegment, mit den Daten, die vom Programm bearbeitet oder erzeugt werden.

Wenn der Entwickler zwei Editor-Prozesse laufen lässt, in denen er jeweils eine eigene Datei bearbeitet, so steht für jeden Prozess ein Editor-Programmsegment im Speicher. Da dieser Bereich immer gleich ist, kann man eine gemeinsame Benutzung dieses Speicherbereiches für verschiedene Editor-Prozesse vorsehen.

Es dauert trotzdem recht lang, bis ein neuer Prozess erzeugt ist.

Aufwändige Kommunikation. Da jeder Prozess komplett getrennt von allen anderen Prozessen abläuft, gibt es zunächst keine Möglichkeit der direkten Kommunikation zwischen den Prozessen. Indirekt, über den Inhalt von Dateien, können

Informationen von einem Prozess zu einem anderen übertragen werden. Im obigen Beispiel sichert der Entwickler sein Programm, das er im Editor schreibt. Der Compiler-Prozess liest und übersetzt dann diese gesicherte Datei.

Für die Synchronisation ist der Anwender selbst verantwortlich. Wenn er während eines Compiler-Laufs die Datei im Editor ändert und abspeichert, kann er dadurch den Compiler-Prozess stören. Der Compiler könnte dann zur Hälfte die alte Version und zur Hälfte die neue Version der Datei lesen und übersetzen. Es ist klar, dass dabei die merkwürdigsten Fehlermeldungen des Compilers auftreten können. Weder der Compiler-Prozess, noch der Editor-Prozess können allein feststellen, dass dieser Fehler aufgetreten ist.

Daher bietet ein Betriebssystem weitere Möglichkeiten der Prozesskommunikation. In Kurseinheit 5 wird dieses Thema genauer behandelt.

Parallelität innerhalb von Prozessen. Ein Grund für die Erstellung des Prozessmodells war, dass man die Parallelität innerhalb einen Systems einfach beschreiben wollte. Parallelität tritt jetzt aber nur auf der Ebene der Prozesse selbst auf. Wenn man Parallelität innerhalb von Prozessen modellieren will, muss man weitere Anstrengungen unternehmen.

Die Probleme, die man mit dem bisherigen Modell der Prozesse hat, sind also:

- Recht großer Aufwand für Erzeugung neuer Prozesse.
- Prozesse belegen viele Betriebssystem-Ressourcen.
- Kommunikation zwischen den Prozessen nur indirekt oder über das Betriebssystem möglich.
- Parallelität innerhalb von Prozessen ist nur aufwändig zu beschreiben.

3.6 Leichtgewichtige Prozesse

Es hat sich als sehr nützlich erwiesen, wenn nicht für alle Aufgaben eigene Prozesse vorhanden sind. Statt dessen werden mehrere Ressourcen von verschiedenen Prozessen gemeinsam benutzt; z. B. ein gemeinsamer globaler Datenbereich. Man spricht in diesem Fall von **leichtgewichtigen Prozessen** (engl. **lightweight process**) oder auch **Threads**.

Parallele Aktivitäten innerhalb eines Prozesses lassen sich dann dadurch einfach beschreiben. Ein Prozess kann aus mehreren Threads bestehen. Ein Thread ist durch seinen eigenen

- Registersatz
- Programmzähler
- Stackbereich

definiert. Das Programm- und Datensegment teilen sich die Threads. Für die Synchronisation der Zugriffe auf die gemeinsamen Bereiche sind die Threads selbst zuständig. Abbildung 3.10 zeigt die Speichereinteilung bei einem Prozess, der aus mehreren Threads besteht. Das Betriebssystem hat hier keine Möglichkeit, den Zugriff eines Threads in z. B. den Stackbereich eines anderen Threads zu erkennen. Da

diese Threads aber aus demselben Prozess stammen, und somit auch zu demselben Benutzer gehören, kann man davon ausgehen, dass solche Zugriffe kein Sicherheitsrisiko sind.

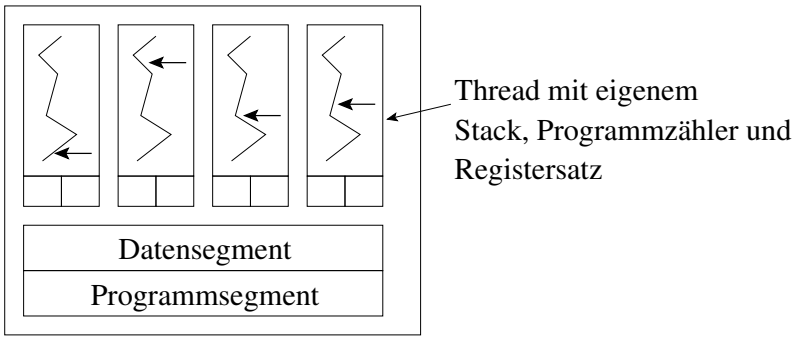


Abbildung 3.10: Prozess mit mehreren Threads.

Vorteile von Threads.

- *Erreichbarkeit*: Wenn ein Prozess für eine bestimmte Aktion z. B. eine Anfrage nach einer Ressource blockiert würde, so muss nur ein Thread blockiert werden, während die anderen Threads weiterarbeiten können; siehe Abschnitt 3.6.1.

- *Teilen der gemeinsamen Ressourcen*: Es ist möglich, dass verschiedene Prozesse auf die gleichen Ressourcen z. B. Datenspeicher, Prozessressourcen zugreifen.

- *Mehr Effizienz*: Threads haben gegenüber herkömmlichen Prozessen den Vorteil, dass sie wesentlich effizienter arbeiten können, weil bei einem Wechsel der rechnenden Threads nur die Register ausgetauscht werden. Auch im Vergleich zum Erzeugen von weiteren echten Prozessen ist es wesentlich günstiger neue Threads zu kreieren. Es ist natürlich ebenfalls effizienter, was den Speicherverbrauch anbetrifft, wenn der Speicher nicht dupliziert wird.

3.6.1 Realisierungen von Threads

Man unterscheidet zwei Hauptarten von Thread-Realisierungen: **Benutzer-Threads** und **Kernel-Threads**.

Benutzer-Threads. Diese Realisierung ist die einfachere. Alle Thread-Informationen liegen im Benutzeradressraum, der Kern hat keinerlei Kenntnis davon, ob ein Prozess mehrere Threads verwendet oder nicht; siehe Abbildung 3.11. Um seine Threads zu verwalten, braucht jeder Prozess seinen eigenen privaten *Thread-Kontrollblock*, der analog zum Prozesskontrollblock ist. Er behandelt das Scheduling seiner Threads selbst mit Hilfe der entsprechenden Thread-Library-Funktionen, die auch für das Erzeugen und Beenden von Threads unabhängig vom Kern zur Verfügung stehen.

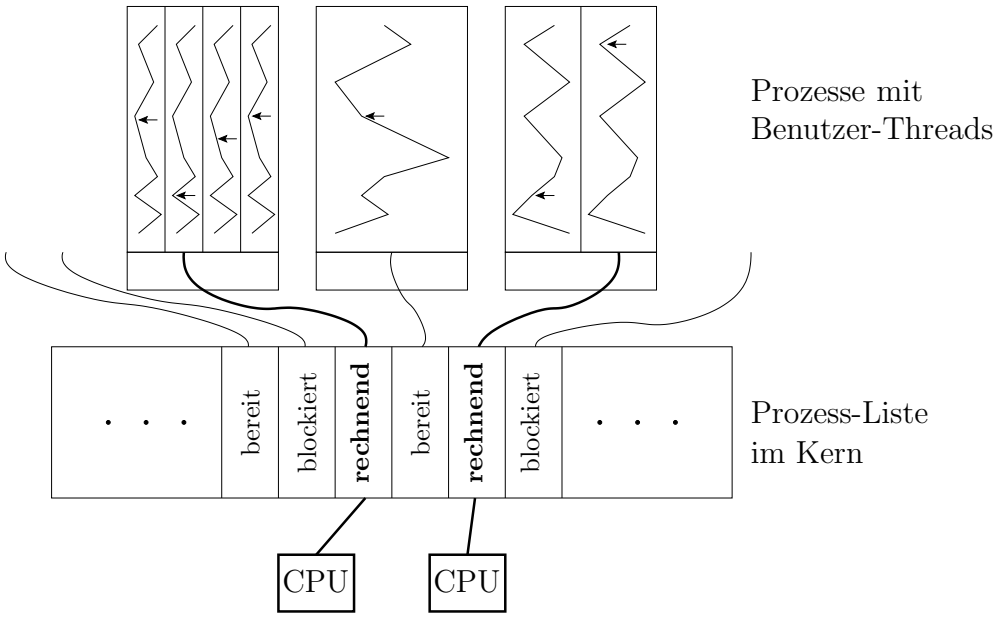


Abbildung 3.11: Zweiprozessor-System mit Benutzer-Threads.

Kernel-Threads. Der Kern verwaltet die Threads ähnlich wie er schon Prozesse verwaltet. Es gibt eine Tabelle von Prozessen und Threads, die alle am Scheduling des Betriebssystem-Kerns teilnehmen; siehe Abbildung 3.12. Diese Realisierung ist erheblich aufwändiger, weil viele Systemaufrufe, die Prozesse betreffen, für Threads neu implementiert werden müssen.

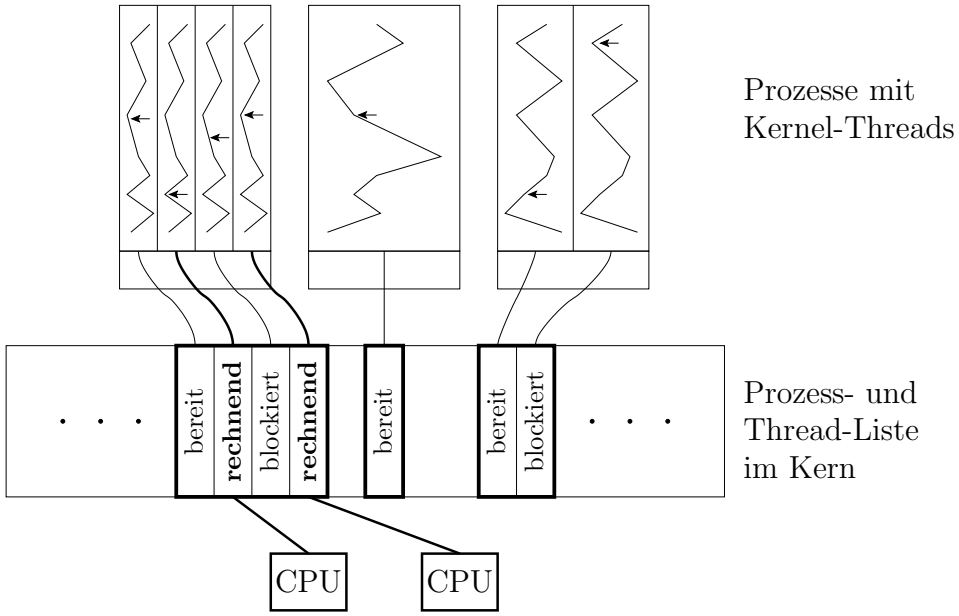


Abbildung 3.12: Zweiprozessor-System mit Kernel-Threads.

Vergleich der Thread-Realisierungen. Benutzer- und Kernel-Threads lassen sich am besten in einer direkten Gegenüberstellung vergleichen, die Vor- und Nachteile liegen mal auf der einen und mal auf der anderen Seite.

	Benutzer-Threads	**Kernel-Threads**
+	Einfache Realisierung durch Bereitstellung einer Thread-Library auf Benutzerebene. Die Library lässt sich relativ leicht auf andere Betriebssysteme portieren.	
−		Aufwändige Realisierung durch Neuimplementierung von Systemaufrufen.
−	Zusammengehörige Benutzer-Threads können auch in Multiprozessor-Rechnern insgesamt nur einen Prozessor benutzen und daher nur nacheinander abgearbeitet werden.	
+		Zusammengehörige Kernel-Threads können in Multiprozessor-Rechnern auf mehreren Prozessoren parallel ablaufen.
−	Wenn ein Benutzer-Thread blockiert, dann ist der ganze Prozess mit allen zugehörigen Threads blockiert.	
+		Ein blockierender Kernel-Thread eines Prozesses beeinflusst nicht die anderen Threads des Prozesses.
+	Das Erzeugen und Umschalten von zusammengehörigen Threads regelt der Prozess selbst. Erzeugung und Wechsel sind schneller, weil mit weniger Verwaltungsaufwand belastet, und flexibler, weil der Prozess seine eigene Zuteilungsstrategie realisieren kann.	
−		Bei Erzeugung und Umschaltung von Kernel-Threads ist der Kern beteiligt und macht in etwa dasselbe wie bei normaler Prozesserzeugung bzw. normalem Prozesswechsel, daraus resultiert ein gewisser Verwaltungsaufwand.
−	Ein Benutzer-Thread kann nicht ohne besonders trickige Programmierung von seinem eigenen Prozess zu einer Unterbrechung gezwungen werden, was also ungefähr einem nonpreemptiven System entspricht.	
+		Die ausgeklügelte Scheduling-Strategie des Betriebssystems wird mitbenutzt, dadurch wird eine faire Behandlung aller Threads garantiert.
+	Ein Prozess mit sehr vielen Benutzer-	− Ein Prozess mit sehr vielen Kernel-
−	Threads wird das Gesamtsystem kaum stärker belasten als einer mit wenigen. Andererseits wird ein solcher Prozess möglicherweise weniger CPU-Zeit bekommen als er sinnvollerweise bekommen müsste.	+ Threads hat starke Auswirkungen auf die Leistung des Gesamtsystems. Andererseits kann es natürlich auch erwünscht sein, dass das Gesamtsystem seine Leistung hauptsächlich auf diesen einen Prozess und dessen Threads konzentriert.

Realisierungen von Threads in aktuellen Betriebssystemen sind oft keine reinen Kernel-Threads, sondern Mischformen. Sie unterscheiden sich auch in der Art, wie Unterbrechungen bei Programmfehlern oder wie geöffnete Dateien behandelt werden oder was ein `fork`-Systemaufruf in einem Prozess mit mehreren Threads genau bewirkt. Dasselbe Programm wird sich deshalb auf verschiedenen Systemen durchaus unterschiedlich verhalten.

3.6.2 Anwendungsgebiete für Threads

Mehrprozessor-Maschinen. Wenn in einem Rechner mehr als ein Prozessor vorhanden ist, so kann der Programmierer mit Hilfe von Threads sein Programm „parallelisieren“ und dadurch wesentlich beschleunigen. Verschiedene Threads können jeweils einem eigenen Prozessor zugewiesen werden und dann parallel abgearbeitet werden.

Gerätetreiber für langsame Geräte. Ein Gerätetreiber für z. B. ein Diskettenlaufwerk ist ein Prozess, der immer eine Reihe von Anfragen zu erfüllen hat. Er läuft auf dem Prozessor im Controller des Laufwerks. Verschiedene Prozesse des Rechners wollen jeweils bestimmte Daten lesen. Jede dieser Anforderungen zerfällt im Prinzip in drei Teile.

1. Zunächst muss der Treiber den Auftrag in eine Liste eintragen, sowie die übergebenen Parameter analysieren und interpretieren. Hier findet dann auch die Berechung der Sektoradresse statt.

2. In der zweiten Phase findet die eigentliche Datenübertragung statt. Während dieser Phase ist der Treiberprozess blockiert, da er auf das Ende der Übertragung wartet. Es wäre an dieser Stelle aber schon möglich, mit der Bearbeitung des nächsten Auftrags zu beginnen und dort schon einmal die Parameter zu analysieren.

3. Am Ende kontrolliert der Treiber, ob alle Daten richtig übertragen wurden (Checksummen) und meldet dem Rechner, dass der Auftrag erledigt wurde.

Für jede Anfrage sind also dieselben Schritte durchzuführen. Es bietet sich an, für jede Anfrage einen eigenen Thread zu erzeugen. Der Prozessor im Controller kann dann schon die nächste Anfrage (Thread) bearbeiten, falls der Lese-/Schreibkopf für die vorherige Anfrage positioniert wird (d. h. der vorherige Thread ist im Zustand blockiert).

Verteilte Systeme. Auch hier hat man oft bestimmte Server (Dateiserver, Druckserver), die ihre Dienste den verschiedensten Clients (Anwendern) anbieten. Beispielsweise kann ein Client an einen Dateiserver eine Anfrage der Art: „Gib mir die ersten 1000 Bytes der Datei mit dem Namen xyz“ stellen. Es ist klar, dass auch hier verschiedene Anfragen vom Server soweit wie möglich parallel bearbeitet werden sollen. Threads sind eine einfache Möglichkeit, dies zu programmieren.

3.7 Zusammenfassung

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie:

- den Unterschied zwischen Programm und Prozess erklären können.
- die Prozesszustände und die möglichen Zustandsübergänge mitsamt ihren Ursachen verstanden haben.
- den Zweck und Inhalt eines PCBs beschreiben können.
- die Aufgaben des Dispatchers und des Schedulers angeben können.
- die vorgestellten Scheduling Strategien verstanden haben und die Arbeitsweise an Beispielen demonstrieren können.
- den Unterschied zwischen Threads und Prozessen kennen und jeweilige Anwendungsgebiete dieser Modelle angeben können.

Literatur

[NiBuPr96] B. Nichols, D. Buttlar, J. Proulx Farrell. *Pthreads Programming.* O'Reilly 1996.

[SiGaGa02] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts: sixth Edition.* John Wiley & Sons, Inc. 2002 .

[Ta02] A. S. Tanenbaum. *Moderne Betriebssysteme: zweite, überarbeitete Auflage.* Pearson Studium, 2002.

Lösungen der Übungsaufgaben

Übungsaufgabe 3.1, Seite 79: Wenn das Ereignis eintritt, auf das ein Prozess im Zustand blockiert wartet, ist in der Regel ein anderer Prozess im Zustand rechnend. Der bisher blockierte Prozess kann also den Prozessor nicht sofort erhalten (in den Zustand rechnend wechseln), da ein anderer Prozess den Prozessor hat. Der blockierte Prozess wechselt daher in den Zustand bereit und kann bei der nächsten Prozessorvergabe mit berücksichtigt werden.

Übungsaufgabe 3.2, Seite 89: Zur Berechnung der Werte wird die Formel

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

benutzt. Aus der ersten Schätzung von 6 und dem ersten CPU burst von 4 ergibt sich die nächste Schätzung zu:

$$\tau_{n+1} = \alpha \cdot 4 + (1-\alpha) \cdot 6$$

Die folgende Tabelle zeigt die geschätzten Werte. Die Zeiten wurden dabei auf zwei Stellen nach dem Komma gerundet.

CPU burst	4.00	4.00	4.00	4.00	6.00	8.00	10.00	12.00	12.00	12.00	12.00
$\alpha = 1/2$	6.00	5.00	4.50	4.25	4.12	5.06	6.53	8.27	10.13	11.07	11.53
$\alpha = 3/4$	6.00	4.50	4.13	4.03	4.01	5.50	7.38	9.34	11.34	11.83	11.96

Übungsaufgabe 3.3, Seite 90: Ein Heap ist ein fast vollständiger binärer Baum, bei dem für jeden inneren Knoten x_i gilt: $\mathrm{Key}(x_i) \leq \mathrm{Key}(x_{2i}) \wedge \mathrm{Key}(x_i) \leq \mathrm{Key}(x_{2i+1})$. Dabei sind x_{2i} und x_{2i+1} die Kinder des Knotens x_i. Heaps lassen sich sehr leicht in arrays speichern.

In unserem Fall ist der Schlüssel gleich der Prioritätszahl. Der Knoten mit der kleinsten Zahl, d. h. mit der höchsten Priorität steht in der Wurzel des Baums. Wenn dieser Knoten entfernt wird, so wird das letzte Element des Heaps zur Wurzel gemacht. Dann vertauscht man dieses Element solange mit dem kleineren seiner Kinder, bis es unten angekommen ist oder kein kleineres Kind mehr existiert.

Im schlimmsten Fall muss man dabei die komplette Höhe des Baumes durchlaufen. Bei n Knoten ist die Höhe $\log n$.

Fügt man ein neues Element in einen Heap ein, so stellt man es zuerst an das Ende. Dann vertauscht man dieses Element so lange mit seinem Vater, bis es größer als der Vater oder an der Wurzel angekommen ist. Auch hier muss man evtl. den Baum in voller Höhe durchlaufen.

Genaueres hierzu können Sie im Kurs 1663 (Datenstrukturen) unter dem Stichwort priority queues nachlesen.

Übungsaufgabe 3.4, Seite 94: Die Antwortzeit der Prozesse ist gleich der Wartezeit auf den Prozessor plus die Bearbeitungszeit.

- Die Antwortzeit des ersten Prozesses ist 15, die des Zweiten $15 + 7$, die des dritten $15 + 7 + 1$, usw. bis zum letzten Prozess mit der Antwortzeit $15 + 7 + 1 + 4 + 8$. Die Summe aller Antwortzeiten ist $15 + 22 + 23 + 27 + 35 = 122$ Damit ergibt sich die mittlere Antwortzeit zu $122/5 = 24{,}4$.

- Die Prozesse werden in der Reihenfolge P_3, P_4, P_2, P_5, P_1 abgearbeitet. Daraus ergeben sich folgende Antwortzeiten: $1, 1+4, 1+4+7, 1+4+7+8$ und $1+4+7+8+15$. Die mittlere Antwortzeit ist also $73/5 = 14{,}6$.
- Die folgende Tabelle zeigt den Ablauf beim Round Robin Scheduling mit Zeitquantum $Q = 4$.

t	Warteschlange	Restzeit					Kommentar
		P_1	P_2	P_3	P_4	P_5	
0	1, 2, 3, 4, 5	15	7	1	4	8	
4	2, 3, 4, 5, 1	11	7	1	4	8	
8	3, 4, 5, 1, 2	11	3	1	4	8	
9	4, 5, 1, 2	11	3	-	4	8	P_3 fertig
13	5, 1, 2	11	3	-	-	8	P_4 fertig
17	1, 2, 5	11	3	-	-	4	
21	2, 5, 1	7	3	-	-	4	
24	5, 1	7	-	-	-	4	P_2 fertig
28	1	7	-	-	-	-	P_4 fertig
35	-	-	-	-	-	-	P_1 fertig

Die Summe aller Antwortzeiten ist $9 + 13 + 24 + 28 + 35 = 109$. Daraus ergibt sich die mittlere Antwortzeit zu $109/5 = 21{,}8$.

Übungsaufgabe 3.5, Seite 95: Solange das System nicht überlastet ist, arbeitet diese Strategie wie geplant.

Wenn allerdings zu viele Prozesse im Zustand bereit sind, so wird das Quantum immer kleiner. Deshalb werden immer mehr Prozessumschaltungen erforderlich. Eine Prozessumschaltung braucht eine konstante, in der Regel kleine Zeit. Wenn das Quantum aber wesentlich kleiner als die Prozessumschaltzeit wird, ist der Rechner nur noch mit Prozessumschaltungen beschäftigt und kommt kaum noch dazu, die Benutzerprozesse zu bearbeiten.

Wenn neben dem rechnenden Prozess kein weiterer Prozess mehr bereit ist, so ist das Zeitquamtum dann als 500ms/0 definiert. Diesen Fall (Division durch 0) muss der Scheduling Algorithmus abfangen und kann dann die Zeitscheibe auf einen vernünftigen großen Wert setzen.

Glossar

Alterung (*aging*) heißt der Vorgang, bei dem die Priorität von Prozessen mit deren Alter(Wartezeit) steigt. Dadurch soll das Verhungern von Prozessen mit niedriger Priorität verhindert werden.

Antwortzeit (*response time*) Das Zeitintervall zwischen dem Start eines Auftrags (Prozess wird rechnend) und der Reaktion des Systems. Sie ist die Summe aus Wartezeit und Bedienzeit.

Bedienzeit (*service time*) Die zur Ausführung eines Auftrags nötige Prozessorzeit.

CPU burst Das Zeitintervall, für das ein Prozess den Prozessor ohne freiwillige Abgabe (auch durch I/O) behalten will.

Dispatcher (*dispatcher*) Der Teil des Betriebssystems, der die Umschaltung des Prozessors zwischen zwei Prozessen vornimmt.

Durchsatz (*throughput*) bezeichnet die „Leistung“ eines Systems. Sie ergibt sich als Quotient aus Arbeit, d. h. abgearbeiteten Prozessen, und dazu benötigter Zeit.

Feedback Scheduling ist eine Scheduling-Strategie, bei der das bisherige Verhalten eines Prozesses in das Scheduling eingeht.

First Come First Served ist eine Scheduling-Strategie, bei der die Prozesse in der Reihenfolge ihrer Ankunft den Prozesser zugeteilt bekommen.

long-term scheduler Der Teil des Betriebssystems, der entscheidet, welcher Prozess als nächster in das System, bzw. den Hauptspeicher geladen wird.

non pre-emptive heißt eine Scheduling-Strategie, die einem rechnenden Prozess den Prozessor nicht entzieht. Nur wenn der Prozess selbst den Prozessor abgibt (warten auf I/O oder freiwillige Abgabe des Prozessors), findet eine Prozessumschaltung statt.

pre-emptive heißt eine Scheduling-Strategie, bei der das Betriebssystem einem rechnenden Prozess den Prozessor entziehen kann.

Priorität (*priority*) sie bestimmt die „Wichtigkeit“ von Prozessen und dadurch auch die Reihenfolge, in der sie den Prozessor zugeteilt bekommen.

Programm (*program*) Beschreibung eines Algorithmus in einer für Mensch oder Maschine verständlichen Sprache.

Prozess (*process*) Ein in der Ausführung befindliches Programm.

Prozessabbild (*process image*) Ein Prozessabbild besteht aus dem Programm, dem Stack, den Daten und dem Prozesskontrollblock eines Prozesses.

Prozesskontrollblock (*process control block*) Eine Datenstruktur, in der das Betriebssystem wichtige Daten über einen Prozess speichert.

Prozessorauslastung (*CPU utilization*) Der Anteil der Zeit, in der an Benutzerprozessen (d. h. keinen systeminternen Verwaltungsprozessen) gerechnet wird, an der Gesamtzeit.

Prozessumschaltung (*context switch*) Der Vorgang, bei dem der Prozessor von einem Prozess zu einem anderen Prozess umgeschaltet wird. Diese Aktion wird von Dispatcher ausgeführt.

Prozesszustand (*process state*) Die Beschreibung der aktuellen Aktivität eines Prozesses (Erzeugt, Bereit, Rechnend, Blockiert, Beendet).

Quantum Siehe Zeitscheibe.

Round Robin ist eine pre-emptive Scheduling-Strategie, bei der die Prozesse nacheinander den Prozessor für ein bestimmtes Quantum zugeteilt bekommen.

Scheduler Ein Teil des Betriebssystems, der aus einer Menge von Bewerbern um ein Betriebsmittel einen auswählt, der das Betriebsmittel dann zugeteilt bekommt. Am häufigsten im Zusammenhang mit der Prozessorzuteilung benutzt.

short-term scheduler Der Teil des Betriebssystems, der aus der Liste der bereiten Prozesse, den Prozess auswählt, der als nächstes den Prozessor zugeteilt bekommt.

Shortest Job First Scheduling-Strategie, bei der immer der Prozess mit der kürzesten Bedienzeit als nächster den Prozessor erhält.

Thread Leichtgewichtiger Prozess.

verhungern (*starvation*) Ein Prozess verhungert, wenn er zwar bereit ist, aber trotzdem niemals den Prozessor erhält und rechnen kann.

Wartezeit (*waiting time*) Die Zeit, für die ein Auftrag im System ist, aber nicht bearbeitet wird. Es ist die Zeit, die ein Prozess in der Bereit-Liste verbringt.

Zeitscheibe (*time slice*) Die Zeit, für die ein Prozess den Prozessor maximal belegen darf, bevor er vom System unterbrochen wird.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 4:
Hauptspeicherverwaltung

Autor: Udo Kelter, Christian Icking, Lihong Ma

Inhalt

Kurseinheit 4

Hauptspeicherverwaltung

4.1 Einführung

In den Kurseinheiten 1 und 3 haben wir Möglichkeiten kennengelernt, wie auf Basis einer einzigen physischen CPU mehrere virtuelle CPUs simuliert und dadurch mehrere Programme *gleichzeitig* ausgeführt werden können. Nun erhebt sich natürlich die Frage, wo denn diese Programme und die zugehörigen Daten überhaupt stehen, denn wir haben auch nur einen einzigen physischen Hauptspeicher in unserem Rechner. Mit anderen Worten: wir müssen für jedes ausgeführte Programm nicht nur eine CPU, sondern auch einen Hauptspeicher simulieren.

Diese Kurseinheit wird dieses Thema behandeln. Wir werden dabei zunächst von einfachen Lösungen ausgehen und auf deren Basis komplexere, leistungsfähigere Lösungen entwickeln; in fast allen Fällen setzen die leistungsfähigeren Lösungen auch eine spezielle Hardware-Unterstützung voraus.

4.1.1 Namens- und Adressräume

Um ein genaueres Verständnis davon zu bekommen, was ein Hauptspeicher aus Sicht eines Programmes überhaupt ist, müssen wir noch einmal auf die Ebenenhierarchie in Kurseinheit 1 zurückkommen. Wir beschränken uns hier auf die Ebenen von der konventionellen Maschinenebene an aufwärts. Auf allen Ebenen treten Programme auf, in denen *Objekte*, also einzelne Bytes oder Worte, Programmvariablen oder komplexe Datenstrukturen, manipuliert werden. Hierbei werden die Objekte durch Namen oder Adressen identifiziert. Die Frage ist nun, wie die Namens- bzw. Adressräume, die man auf den verschiedenen Ebenen antrifft, aussehen und wie sie von den oberen Ebenen in die unteren transformiert werden.

Beginnen wir mit der konventionellen Maschinenebene. Auf dieser Ebene sind noch sehr viele Details der Hardware und der vorhandenen Konfiguration des Rechners sichtbar und spielen bei der Adressierung eine Rolle. Die hier bei verschiedenen Rechnertypen anzutreffenden Strukturen sind oft sehr unterschiedlich. Aus Platzgründen beschränken wir uns im Folgenden auf eine Skizzierung der Strukturen, wie sie in den heute am meisten verbreiteten Rechnern mit Bus-Architektur auftreten.

Die Hardware umfasst einen **physischen** oder **realen** (**Haupt-** oder **Arbeits-**) **Speicher**, der heute üblicherweise aus Halbleitern gebildet wird, die in einem oder mehreren Moduln angeordnet sind. Die Zahl und oft auch die Art der Moduln in

einem Rechner kann meist variiert werden, wodurch die Größe des Hauptspeichers des Rechners variiert werden kann. Der Speicher besteht aus einzeln adressierbaren Speicherzellen (Worten oder Bytes). Bytes sind üblicherweise 8 Bit groß, Worte meist deutlich größer. Der Unterschied spielt für uns aber keine Rolle; deswegen reden wir im Folgenden meist nur von Bytes. Entscheidend ist, dass jedes Byte einzeln durch eine **physische** (oder **reale**) **Adresse** (physical address[1]) identifiziert bzw. adressiert werden kann.

Eine physische Adresse ist dabei eine ganze Zahl, die in binärer Form mit n Stellen dargestellt wird. Wir nehmen an, dass die CPU zumindest dann, wenn sie im Systemmodus arbeitet, solche Adressen erzeugt und sie auf einem Adressbus zum physischen Hauptspeicher übertragen kann, wenn sie Speicherzellen lesen oder schreiben will. Für jede der n Stellen der Adresse ist eine eigene Busleitung und ein eigener Anschluss an der CPU und am Speicher vorhanden[2]. Die Zahl der Anschlüsse bestimmt somit auch die Größe (*Breite*) der Adressen. Wenn der Adressbus einer CPU n Stellen breit ist, können die Adressen 0 bis $2^n - 1$ übertragen werden. Also liegt die Breite dieses Busses die maximale Anzahl an adressierbaren Zelle fest, z. B. können über einen 16 Bit breiten Adressbus maximal 2^{16} Zellen angesprochen werden. Die Menge dieser Zahlen nennen wir den **physischen Adressraum**.

Wie wir oben schon gesehen haben, kann die Größe des realen Speichers variiert werden. Bei einigen älteren, technisch überholten Rechnertypen kann man den Speicher größer wählen als den physischen Adressraum. Dann ist immer nur ein Teil des realen Speichers direkt zugreifbar und es muss durch besondere Maschinenbefehle zwischen verschiedenen Teilen des realen Hauptspeichers umgeschaltet werden. Umgekehrt kann der physische Speicher *kleiner* sein als der physische Adressraum (bei der heute üblichen Adressbreite von 32 Bit ist dies der Normalfall). Es gibt also *syntaktisch korrekte* Adressen, zu denen in einem bestimmten Rechner keine Speicherzelle vorhanden ist. Der Versuch, eine solche Adresse für einen Zugriff zu benutzen, führt zu einer Unterbrechung (bus error) und zum Abbruch des Programms. Solche Adressen nennen wir **nicht realisierte** (physische) **Adressen**; andernfalls sprechen wir von **realisierten Adressen**.

Wie schon oben erwähnt, erzeugt die CPU reale Hauptspeicheradressen, wenn sie ein Programm ausführt, z. B. um den nächsten auszuführenden Befehl heranzuholen oder Operanden zu lesen bzw. zu schreiben. In den Maschinenbefehlen werden verschiedenste Arten von Adressoperanden benutzt, die sich auf diesen Adressraum beziehen. Bei direkten Adressen wird die Adresse direkt als ganzzahliger Wert angegeben. Des Weiteren treten indirekte, indizierte und andere Adressen auf. Die verfügbaren Adressierungsarten hängen stark von der CPU ab. Letztlich wird aber bei allen Adressierungsarten innerhalb der CPU ein ganzzahliger Wert bestimmt, eben die reale Adresse.

Auf der nächsthöheren Ebene, der Betriebssystemebene, liegen i. W. die gleichen Maschinenprogramme vor: im Vergleich zur konventionellen Maschinenebene hinzu-

[1] Meist wird das englische 'physical' in das deutsche 'physikalisch' (= die Physik betreffend) übersetzt. Die Übersetzung 'physisch' (= körperlich, stofflich) passt aber besser, denn physische Adressen ergeben sich vor allem aus der physisch vorhandenen Hardware, weniger aus den Gesetzen der Physik.)

[2] Hier nehmen wir zur Vereinfachung an, dass die gesamte Adresse auf einmal übertragen wird. Manche CPUs übertragen die Adresse stückweise und kommen dadurch mit einem schmaleren Bus aus.

gekommen sind die Systemaufrufe, weggefallen sind die privilegierten Befehle, was aber keinen Einfluss auf die Syntax der übrigen Maschinenbefehle und insb. auf die darin auftretenden Adressoperanden hat.

Die wesentliche Änderung liegt in der *Bedeutung* der Adressen: wir setzen voraus, dass die Programme auf dieser Ebene im Benutzermodus ausgeführt werden, wodurch die Adressen bei vielen Rechnern (wenn auch nicht bei allen) anders als im Systemmodus interpretiert werden. Aus diesem Grund bezeichnen wir diese vom Programm generierten Adressen als **logische Adressen**.

In historisch frühen Rechnerarchitekturen waren die logischen Adressen identisch mit den physischen Adressen, d. h. sie wurden unverändert zur Identifizierung einzelner Bytes im Hauptspeicher benutzt. Heute gilt dies aber nicht mehr: im Benutzermodus werden logische Adressen mit Hilfe spezieller Hardware, der sog. **MMU Memory Management Unit**, in physische Adressen umgesetzt. Physisch ist die MMU entweder in der CPU integriert oder dem Prozessor nahe angeordnet, logisch gesehen liegt sie zwischen der CPU und dem Hauptspeicher, siehe Abbildung 4.1. Eine logische Adresse wird also hier in eine physische Adresse umgeformt; wir nennen solche logischen Adressen auch **virtuelle Adressen**, wenn wir diesen Sachverhalt betonen wollen. Die Adresstransformation ist prozessspezifisch. Im Systemmodus werden die von der CPU erzeugten Adressen hingegen nicht transformiert, sondern direkt als physische Adressen interpretiert.

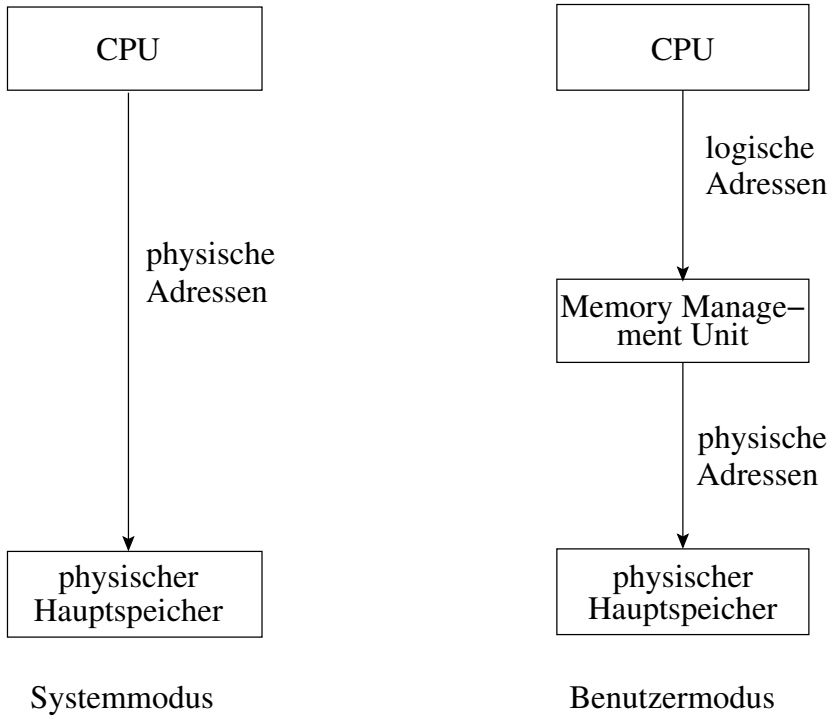


Abbildung 4.1: Die MMU bildet virtuelle Adressen auf physische Adressen des Hauptspeichers ab.

Logische Adressen sind auch auf den höheren Sprachebenen sichtbar. In Assembler-Programmen werden lediglich anstelle von logischen Adressen Bezeichner verwendet; die Semantik dieser Programme, insb. die Adressberechnungen, basiert jedoch unverändert auf logischen Adressen.

Bei höheren Programmiersprachen wird die Vorstellung, dass ein Programm auf einem linear adressierten Hauptspeicher arbeitet, verlassen; stattdessen treten Variablen verschiedener elementarer oder komplexer Typen auf, die durch Namen identifiziert werden. Logische Adressen treten in den meisten höheren Programmiersprachen aber dennoch auf, nämlich als *Zeiger* bzw. *Pointer*: der Wert einer Zeiger-Variablen ist nichts anderes als eine logische Adresse. Viele Sprachen erlauben sogar Adressrechnungen (i. W. Additionen) mit Zeiger-Variablen.

Logische Adressen treten auch in **Dumps** auf, die die meisten Betriebssysteme nach einem unkontrollierten Abbruch eines Benutzerprogramms erzeugen und mit denen ein Programmierer, der nicht in Assembler programmiert, wahrscheinlich unmittelbar nichts anfangen kann: ein Dump ist ein Abzug des logischen Arbeitsspeichers des beendeten Prozesses.

4.1.2 Übersetzer, Binder und Lader

Im Ebenenmodell von Kurseinheit 1 traten von oben nach unten drei Arten von Programmen auf: Programme in höheren Programmiersprachen, Assemblerprogramme und Maschinenprogramme mit oder ohne Systemaufrufe; für die auftretenden Adressen spielt dieser Unterschied keine Rolle. Wir kommen noch einmal auf die Schritte zurück, in denen i. A. Programme der oberen Schichten nach unten übersetzt[3] werden:

1. Quellprogramme in Assembler oder höheren Programmiersprachen sind i. d. R. nicht monolithisch, sondern bestehen aus selbständig übersetzbaren **Quellprogramm-Moduln**. Ein Quellprogramm-Modul wird zunächst vom Compiler oder Assemblierer in ein **Bindemodul** (oft auch **Objekt-Modul** genannt) übersetzt.

2. In jedem Bindemodul können sich externe Referenzen auf Adressen in anderen Moduln befinden. Dabei können die externen Referenzen der Bindemodule nur symbolisch dargestellt werden. Ein vollständiges, lauffähiges Programm muss also aus mehreren, i. A. unabhängig voneinander übersetzten Bindemoduln zusammengesetzt werden, dies besorgt der **Binder**. Der Binder erzeugt ein **Lademodul**, das die aneinander grenzende Verknüpfung aller Bindemoduln darstellt. Jede Referenz in einem Modul muss von einer symbolischen Adresse in eine Referenz auf eine Adresse im Gesamtlademodul umgewandelt werden. In Abbildung 4.2 enthält Modul A einen Prozeduraufruf von Modul B. Wenn diese Moduln im Lademodul zusammengefasst werden, wird die symbolische Referenz auf Modul B in eine spezifische Referenz auf die Adresse der Einsprungstelle von B innerhalb des Lademoduls umgewandelt.

3. Das Lademodul wird vom **Lader** in den logischen Hauptspeicher des Prozesses geladen; dies geschieht i. d. R. beim Start eines neuen Prozesses. Das Lademodul wird dabei in manchen Fällen einfach kopiert. In anderen Fällen aber unterscheidet sich das **geladene Programm** vom Lademodul.

[3] Anstelle von Übersetzern können auch Interpreter auftreten. Dieser Unterschied ist aber für die Art der Namens- bzw. Adressräume nicht wesentlich.

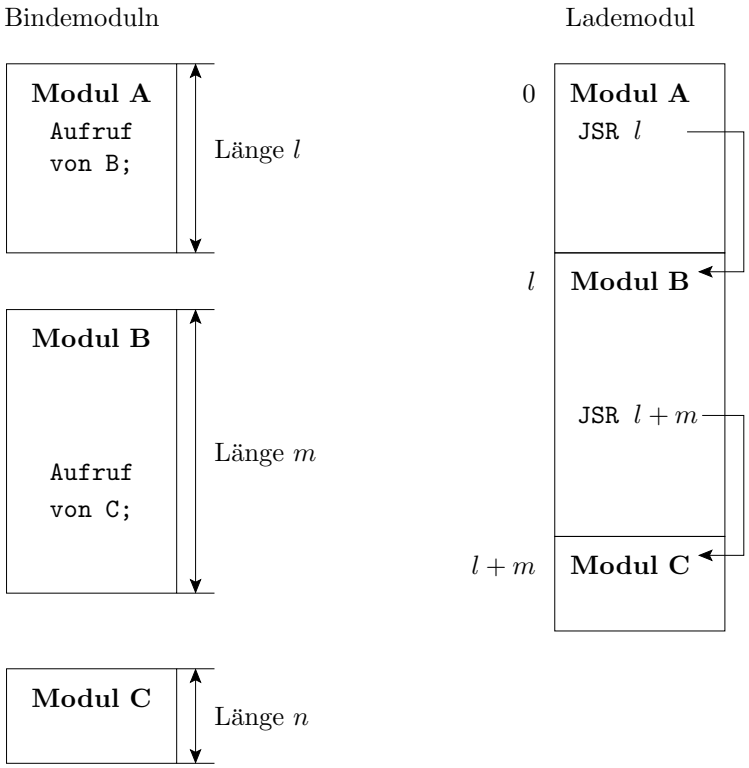


Abbildung 4.2: Der Binder löst die Referenzen auf. Im Lademodul sind die Aufrufe der Bindemoduln unter ihrem symbolischen Namen ersetzt durch JSR-Befehle (*jump to subroutine*) auf relative Adressen im Lademodul.

Es ist sehr wichtig, diese vier Erscheinungsformen eines Programms zu unterscheiden. Die Folge der Transformationsschritte ist noch einmal in Abbildung 4.3 veranschaulicht.

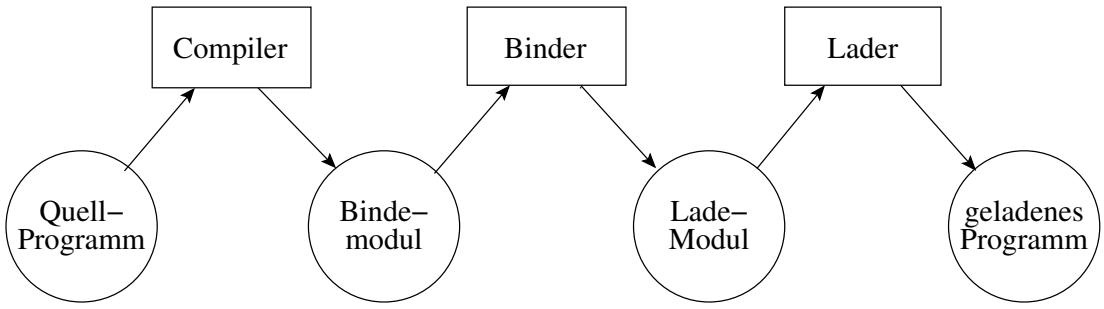


Abbildung 4.3: Die Schritte vom Quellprogramm bis zum ausführbaren Programm.

Die Menge der Adressen des logischen Hauptspeichers eines Prozesses ist bei vielen Systemen ein Intervall, das immer bei 0 anfängt; bei anderen Systemen kann es bspw. das Intervall [100000, 200000) sein. Die im geladenen Programm auftretenden Adressen (Sprungadressen oder Adressen von Daten) müssen offensichtlich im logischen Adressraum des Prozesses liegen. Bzgl. der Frage, wie der Lader dies erreicht, können wir nun zwei Fälle unterscheiden:

1. Wenn bereits der Binder die Adressen des logischen Hauptspeichers kennt, der bei der Ausführung des zu erzeugenden Lademoduls verfügbar sein wird, dann kann der Binder gleich passende Adressen erzeugen; man nennt diese Adressen

auch **absolute Adressen**. Der Lader braucht dann dieses Lademodul *nur noch in den logischen Hauptspeicher zu kopieren*. Die obige Voraussetzung ist vor allem dann erfüllt, wenn ein Programm gebunden und danach sofort geladen und gestartet werden soll; Binder und Lader werden dann oft zu einem **Bindelader** zusammengefasst.

Nachteil der Erzeugung absoluter Adressen in Lademoduln ist, dass das Lademodul nur noch an genau diesen Bereich logischer Adressen geladen werden kann. Viele Programme werden aber sowieso nur einmal geladen, z. B. bei der Entwicklung von Programmen, wo oft Zyklen der Form editieren - compilieren - binden - laden und ausführen durchlaufen werden.

2. Im anderen Fall muss es der Binder dem Lader ermöglichen, das Lademodul in einen *beliebigen* logischen Adressbereich zu laden. Hierzu nimmt der Binder einfach an, dass im logischen Hauptspeicher von 0 an nummerierte Adressen zur Verfügung stehen; man nennt diese Adressen auch **relative Adressen**. Der Lader muss dann beim Laden nur noch die Startadresse des verfügbaren Bereichs logischer Adressen aufaddieren, die im Programm auftretenden relativen Adressen also in absolute umformen. Da der Lader i. A. bei einzelnen Worten im Lademodul nicht erkennen kann, ob es sich hier um Maschinenbefehle oder um Daten (also Konstanten) handelt, muss der Binder geeignete Hilfsdaten erzeugen, durch die der Lader alle Maschinenbefehle findet, die umzusetzende Adressen enthalten. Diese Hilfsdaten werden üblicherweise zusammen mit dem *eigentlichen* Lademodul in einer Datei gespeichert und sind nicht mehr Teil des geladenen Programms (außer bei dynamischen Bindeladern; s. u.).

Betrachten wir als nächstes die Funktion des Binders und seine Eingaben. Der Binder plaziert die zu bindenden Moduln hintereinander in einen einzigen Raum relativer oder absoluter Adressen und legt dabei die Basisadresse jedes Bindemoduls fest. Die Lage eines Bindemoduls in den Lademoduln, die dieses Bindemodul enthalten, ist i. A. nicht fest oder vorab bekannt. Deshalb enthält das Bindemodul Relativadressen analog zu Relativadressen in einem Lademodul. Beim Binden werden diese Relativadressen innerhalb des Bindemoduls um die Basisadresse des Bindemoduls im Lademodul erhöht, siehe Abbildung 4.2. Darüber hinaus löst der Binder externe Referenzen auf, also Referenzen zu benannten Einsprungpunkten in fremden Moduln; hierauf gehen wir hier aber nicht näher ein. Der Binder kann entweder ein neues Bindemodul erzeugen oder ein Lademodul, manche Systeme machen gar keinen Unterschied hierzwischen.

Auf den höheren Sprachebenen treten nur noch symbolische Bezeichner auf, deshalb redet man hier üblicherweise von **Namensräumen**. Die Umsetzung von (Quellmodul-lokalen) Namensräumen in Räume von Relativadressen ist Sache von Assemblierern oder Compilern und interessiert uns hier nicht weiter.

Wenn man die Folge von Umsetzungen von oben nach unten durch unsere Hierarchie durchläuft, werden *abstraktere* Adressen bzw. Namen in *konkretere* umgesetzt. Bei allen Schritten spricht man auch von einer Bindung und von einem zugehörigen **Bindungszeitpunkt**. Man kann mehrere Bindungsschritte auf einmal machen; dann sagt man, dass die Bindung *früh* erfolgt.

Verschiebbare Maschinenprogramme. Wie wir oben gesehen haben, müssen beim Binden und/oder Laden Moduln in Maschinensprache innerhalb eines Adressraums verschoben werden. Die hierbei erforderlichen Änderungen an den Adressen, die in dem Modul auftreten, sind relativ aufwändig. Man hätte hier am liebsten **verschiebbare Maschinenprogramme** (**relocatable code**, **position independent code**), die beim Verschieben überhaupt nicht verändert werden müssen.

Viele CPU-Typen unterstützen die Lösung dieses Problems durch die Basisregister-Adressierung. Hierbei wird eine logische Adresse, die in einem Maschinenbefehl als Operand benötigt wird, nicht als direkte Adresse spezifiziert, sondern durch ein Paar

(Registernummer, Offset).

Die Adresse des Operanden ist dann der Inhalt des angegebenen Registers erhöht um den Offset, siehe Abbildung 4.4. Das angegebene Register nennt man **Basisregister**. Wenn man nun dafür sorgt, dass das Basisregister bei der Programmausführung die

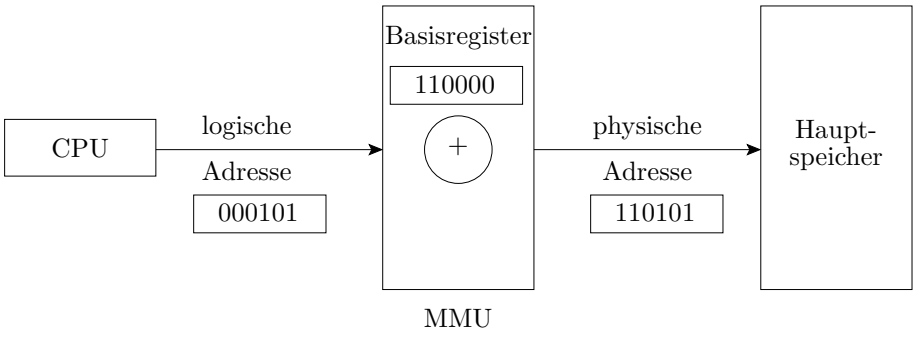


Abbildung 4.4: Hardwareunterstützung mit Hilfe des Basisregisters.

Basisadresse eines Moduls im logischen Adressraum des Prozesses enthält, der das Programm ausführt, kann man als Offset gerade die schon oben erwähnten Relativadressen für Referenzen innerhalb des Moduls verwenden. Der Witz an der Sache ist, dass das Bindemodul nun überhaupt *keine absoluten Adressen mehr enthält.* Es kann völlig unverändert Teil des Lademoduls und später Teil des gespeicherten Programms werden. Es kann sogar zur Programmlaufzeit innerhalb des Speichers verschoben werden. Voraussetzung ist aber immer, dass das Basisregister den richtigen Wert enthält.

Dynamisches Binden. Man mag übrigens einwenden, dass der Binder im Kontext von Betriebssystemen eigentlich ähnlich wie Details der Compilierung nicht besonders erwähnenswert ist, da aus Sicht des Betriebssystems nur die ladbaren Programme interessant sind, nicht hingegen deren Entstehung. Dies stimmt für die meisten älteren und weniger leistungsfähigen Betriebssysteme. Moderne Betriebssysteme unterstützen hingegen das *dynamische Binden* von Programmen. Hierbei wird das Binden zum Teil der Programmausführung. Ein Lademodul kann hier offene, d. h. noch nicht gebundene externe Referenzen enthalten. Werden diese Referenzen benutzt, wird die Ausführung des Programms unterbrochen, aus einer *Bindemodulbibliothek* wird ein passendes Modul nachgeladen und die offene Referenz wird gebunden. Das ausgeführte Programm wird also *inkrementell* gebunden; es werden nur die Moduln gebunden, die zur Ausführungszeit tatsächlich benötigt werden. Man

kann das dynamische Binden z. B. dadurch realisieren, dass ein *dynamischer Bindelader* als Bindemodul zu dem ladbaren Programm hinzugebunden wird und dass bei allen Aufrufen bisher noch offener Referenzen zunächst der Bindelader wie ein ganz normales Unterprogramm aufgerufen wird, der dann die fehlenden Bindemoduln inkrementell zu dem bereits vorhandenen hinzulädt und bindet. Alternativ kann man den dynamischen Bindelader als Teil des residenten Betriebssystemkerns anlegen und durch einen entsprechenden Systemaufruf verfügbar machen; hierfür sprechen u. a. die Probleme, die auftreten, wenn das Bindelader-Bindemodul versehentlich gelöscht wird.

Hauptvorteil des dynamischen Bindens ist, dass die geladenen Programme selbst unter Berücksichtigung der zum Binden erforderlichen Hilfsdaten i. A. kleiner werden, somit also weniger Hauptspeicherplatz benötigt wird. Hauptnachteil ist der etwas erhöhte Laufzeitaufwand, der durch das Binden entsteht. Außerdem hat das dynamische Binden Auswirkungen auf das Versionsmanagement von Programmen, auf die wir hier aber nicht näher eingehen wollen.

Eng mit dem dynamischen Binden hängt das gemeinsame Benutzen von Bindemoduln in mehreren parallel ausgeführten Programmen zusammen, was wir erst später behandeln werden.

4.2 Einfache zusammenhängende Speicherzuweisung

Nach den vorbereitenden Betrachtungen im vorigen Abschnitt muss also das Betriebssystem jedem Prozess einen logischen Hauptspeicher zur Verfügung stellen.

Wir fangen mit den einfachen Speicherverwaltungsverfahren an. Die erste Vereinfachung besteht darin, dass der gesamte logische Hauptspeicher eines Prozesses *zusammenhängend* in einem Stück im physischen Hauptspeicher realisiert wird; demzufolge bilden die realisierten logischen Adressen ein zusammenhängendes Intervall. Der logische Hauptspeicher muss sowohl das ausgeführte Programm wie auch die Daten enthalten.

Die nächste und noch drastischere Vereinfachung besteht darin, dass wir zu jedem Zeitpunkt *maximal einen* Benutzerprozess zulassen. Diese Annahmen treffen auf sehr alte Betriebssysteme zu (die erste Generation) und auf die heute millionenfach verbreiteten Betriebssysteme für PCs.

Die Belegung des physischen Speichers ist dann typischerweise wie in Abbildung 4.5 angegeben: Beginnend bei Adresse 0 steht entweder das vollständige Betriebssystem oder ein residenter *Kern*, zu dem auf alle Fälle der Lader gehört, während andere Teile des Betriebssystems nur bei Bedarf geladen werden. Dahinter liegt, beginnend bei Adresse a, das geladene Anwenderprogramm inkl. seiner Daten.

Der Binder kann hier absolute Adressen generieren, wenn der vom Betriebssystemkern belegte Teil immer gleich lang ist, also a Bytes.

Speicherschutz. Das Anwenderprogramm kann, wenn keine Vorkehrungen dagegen getroffen werden, auch Adressen $< a$ absichtlich oder unabsichtlich benutzen und somit in das Betriebssystem eingreifen oder gar dieses zerstören. Um dies zu verhindern, sind verschiedene Maßnahmen zum Speicherschutz möglich, die alle in irgendeiner Form durch die Hardware unterstützt werden müssen:

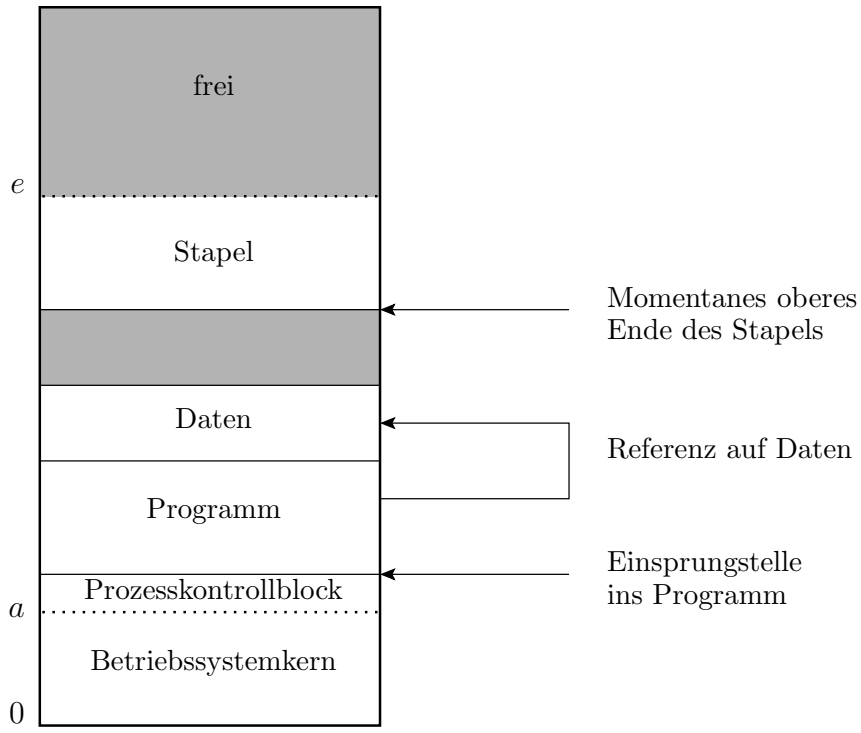


Abbildung 4.5: Einfache zusammenhängende Speicherzuweisung.

- Wenn logische Adressen unverändert als physische Adressen verwendet werden, kann man eine Unterbrechung erzeugen, wenn das Anwenderprogramm auf eine Adresse $< a$ zugreift. Hierzu ist ein „**Grenzregister**“ erforderlich, das die derzeitige Grenze zwischen dem geschützten, das Betriebssystem enthaltenden Bereich und dem ungeschützten Bereich enthält, also den Wert a. Den Wert a hart zu verdrahten, wäre zu unflexibel, denn die Größe des Kerns inkl. dort angelegter Puffer kann variieren. Das Grenzregister kann nur im Systemmodus gesetzt werden. Bei jedem Speicherzugriff muss nun durch eine Hardware-Funktion abgefragt werden, ob die Adresse, auf die zugegriffen werden soll, kleiner als der Wert des Grenzregisters ist. Falls nein, ist alles in Ordnung. Falls ja, wird eine Unterbrechung erzeugt, der auslösende Befehl nicht ausgeführt und das Betriebssystem übernimmt die Kontrolle.

- Man verwendet ein *internes, für das Anwenderprogramm unsichtbares Basisregister*, welches den Wert a enthält, und benutzt im Anwenderprogramm nur Relativadressen zur Basis a. Bei jedem Speicherzugriff im Benutzermodus wird der Wert des Basisregisters zu der angegebenen Adresse hinzuaddiert. Mit anderen Worten läuft das Anwenderprogramm in einem *virtuellen* Adressraum ab und kann überhaupt nicht auf physische Adressen $<$ a zugreifen. Im Systemmodus findet die Adressumrechnung natürlich nicht statt.

Durch den Schutz des Betriebssystems vor Zugriffen durch das Anwenderprogramm entsteht jetzt allerdings das Problem, dass das Anwenderprogramm keine Betriebssystemroutinen mehr aufrufen kann, z. B. um seine eigene Terminierung anzuzeigen, denn dies wären ja Sprünge an eine Adresse $< a$. Wie schon in Kurseinheit 1 dargestellt, löst man dieses Problem mit Hilfe von Systemaufrufen, durch

die eine Unterbrechung erzeugt wird, die zur Unterbrechung der Ausführung des Anwenderprogramms und zur Übergabe der Kontrolle an das Betriebssystem führt.

Swapping. Obwohl wir in diesem Abschnitt noch annehmen, dass immer nur ein einziges Anwenderprogramm im Speicher steht, können dennoch mehrere Programme zeitlich verzahnt ausgeführt werden. Der Aufwand für einen Prozesswechsel ist allerdings erschreckend hoch, denn der gesamte Hauptspeicherinhalt des bisher laufenden Anwenderprogramms muss auf einen Sekundärspeicher kopiert werden, neben den Registersätzen und sonstigen Daten, die zur Fortsetzung der Programmausführung benötigt werden, siehe Kurseinheit 3; umgekehrt muss der Hauptspeicherinhalt des anschließend auszuführenden Programms von einem Sekundärspeicher in den Hauptspeicher kopiert werden. Den ganzen Vorgang der Aus- und Einlagerung von Prozessen nennt man **swapping**, siehe Abbildung 4.6.

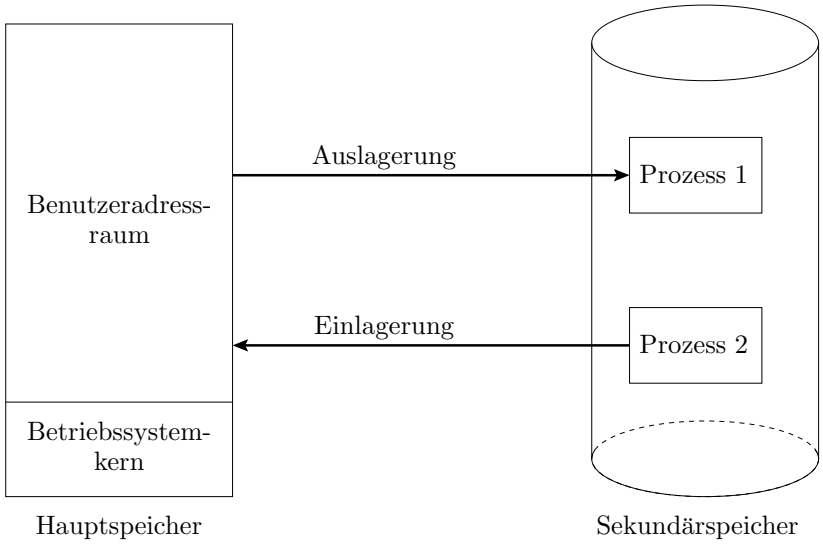


Abbildung 4.6: Prozess 1 wird ausgelagert, und anschlienßend wird Prozess 2 eingelagert.

Der Aufwand für das swapping kann reduziert werden, wenn das Betriebssystem weiß, welche Teile des reservierten Hauptspeichers *tatsächlich* benutzt werden; die unbenutzten Teile können dann beim swapping weggelassen werden. Zunächst einmal kann der maximal für den Prozess benötigte Platz kleiner sein als der verfügbare logische Hauptspeicher, so dass für den Prozess nur der Bereich von Adresse a bis e reserviert wird, siehe Abbildung 4.5.

Selbst hierin sind u. U. Teile unbenutzt. Benutzt sind auf alle Fälle das geladene Programm und statische Datenbereiche; deren Größe liegt bereits nach dem Binden fest. Hinzu kommen dynamisch angelegte Datenstrukturen, deren Auftreten normalerweise von Eingabedaten abhängt. Diese Datenstrukturen können irgendwo im restlichen reservierten Hauptspeicher angelegt werden, am besten aber direkt angrenzend an den statischen Teil. Damit das Betriebssystem weiß, welche Bereiche dynamisch belegt sind, überlässt man es am besten dem Betriebssystem, auch den reservierten Hauptspeicher zu verwalten. Das Anwenderprogramm muss also, wenn es z. B. einen zusätzlichen Bereich von 100 Bytes für dynamische Datenstrukturen benötigt, diesen Bereich vom Betriebssystem explizit durch eine Funktion namens

`getmain`, `malloc` o. ä. anfordern und umgekehrt später durch eine Funktion namens `free` o. ä. freigeben.

Obwohl swapping relativ lange dauert, wurde es in einigen frühen time-sharing Systemen benutzt; bei einem sehr kleinen Hauptspeicher hat man ohnehin nicht viele andere Alternativen.

Ein Problem beim swapping stellen asynchrone Ein- oder Ausgaben dar. Wenn ein Prozess, der noch auf die Beendigung einer E/A-Operation wartet, ausgelagert würde, würde die E/A-Operation u. U. auf Hauptspeicherbereiche schreiben, die im ausgelagerten Prozess als Puffer reserviert waren, die aber inzwischen vom neu eingelagerten Prozess für andere Zwecke benutzt werden, siehe auch Abschnitt 2.2.4. Daher müssen entweder alle E/A-Puffer im vom Betriebssystem kontrollierten Hauptspeicherbereich also im Kern liegen und die zu übertragenden Daten müssen von dort zum Benutzerbereich kopiert werden oder ein Prozess, der noch auf die Beendigung einer E/A-Operation wartet, darf nicht ausgelagert werden.

4.3 Mehrfache zusammenhängende Speicherzuweisung

Die in einem Rechner vorhandenen Speicher sind normalerweise groß genug, um mehrere Programme und deren Daten nebeneinander aufnehmen zu können. Im Idealfall kann die CPU, wenn der gerade ausgeführte Auftrag z. B. auf eine E/A-Operation warten muss, immer sofort und ohne swapping zu einem ausführungsbereiten Prozess wechseln und diesen ausführen. Das zeitraubende swapping wird entweder ganz überflüssig oder zumindest seltener.

Wir bleiben bei der im vorigen Abschnitt gemachten Annahme, dass jeder Prozess einen zusammenhängenden Abschnitt des physischen Speichers zugeteilt bekommt und dass der logische Hauptspeicher des Prozesses zusammenhängend ist, verallgemeinern aber auf den Fall mehrerer solcher Abschnitte im physischen Hauptspeicher (*multiple contiguous allocation*), die wir **(Speicher-) Segmente** nennen. Es ergibt sich jetzt sofort die Frage, wie der gesamte physische Speicher aufgeteilt werden soll. Zwei grundlegende Ansätze sind:

- Der physische Speicher wird beim Starten des Betriebssystems *fest* in Segmente unterschiedlicher Größe aufgeteilt. Anzahl und Größe der Segmente ändern sich zumindest im laufenden Betrieb nicht. Wenn ein Segment frei wird, wird aus der Menge der auf Ausführung wartenden Aufträge ein Auftrag ausgewählt, dessen Speicherplatzanforderungen von dem freien Segment erfüllt werden; das auszuführende Programm wird dann dorthin geladen und ausgeführt.
- Der physische Speicher wird *variabel* in Segmente aufgeteilt. Anzahl, Lage und Größe der Segmente ändern sich anhand der Speicherplatzanforderungen von zu startenden Prozessen.

Die bekanntesten Beispiele für die beiden Ansätze sind **MFT** bzw. **MVT** (multiprogramming with a fixed / variable number of tasks), beides Versionen des OS/360 der IBM. Wir verwenden der Einfachheit halber diese beiden Abkürzungen als Bezeichnungen für die beiden Ansätze.

Speicherschutz. Im Vergleich zur einfachen zusammenhängenden Speicherzuweisung müssen nun unzulässige Speicherzugriffe über beide Grenzen eines Segments hinweg abgewehrt werden. Nach *oben* kommen sowohl ein Grenzregister wie auch ein Basisregister unter Verwendung virtueller Adressen in Betracht; siehe Abbildung 4.7. Nach *unten* kommt nur ein zusätzliches Grenzregister in Betracht. Beide Register brauchen nur einmal im Rechner vorhanden zu sein und müssen vom Dispatcher, siehe Kurseinheit 3, bei jedem Prozesswechsel analog zu anderen Registern gerettet bzw. geladen werden.

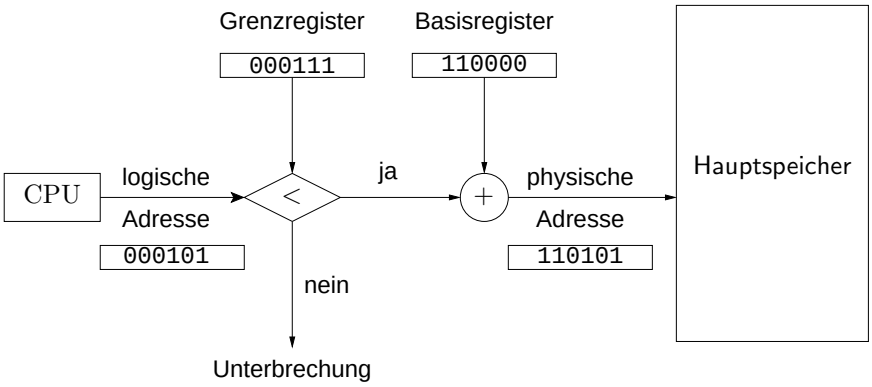


Abbildung 4.7: Speicherschutz durch Grenzregister und Basisregister.

Swapping. Swapping innerhalb eines Segments ist analog zur einfachen zusammenhängenden Speicherzuweisung möglich. Während des Ein- oder Auslagerns in einem Segment kann nun die CPU ein Programm in einem anderen Segment ausführen, vorausgesetzt, wir haben DMA-Controller und die CPU wird nicht selbst durch das swapping belastet; im Idealfall ist immer ein Prozess, der gerade in einem Segment eingelagert ist, bereit zur weiteren Ausführung, so dass die CPU nie auf Einlagerungen warten muss.

Eine generelle Frage ist, ob ein einzulagernder Prozess immer wieder in das gleiche Segment eingelagert werden muss, von dem er ausgelagert wurde. Dies ist im Prinzip nicht wünschenswert, weil sich so bei einem einzelnen Segment eine Warteschlange ausführungsbereiter Prozesse bilden kann, obwohl andere Segmente unbelegt sind.

Bei Verwendung eines Basisregisters und virtueller Adressen kann der Prozess in einem beliebigen Segment eingelagert werden, sofern dieses wenigstens so groß wie das vorher benutzte Segment ist. Einige lösbare technische Probleme entstehen, wenn z. B. der Stack oder eine andere dynamische Struktur am unteren Rand des verfügbaren Adressbereichs beginnt und *nach oben* wächst. Eine geeignete Anordnung der Daten innerhalb eines Segments vorausgesetzt, kann der effektiv von einem Prozess benutzte Speicherbereich nach dem Einlagern sogar größer werden als das Segment, in dem sich der Prozess vorher befand.

Ohne virtuelle Adressen ist ein Wechsel des benutzten Segments praktisch nicht möglich. Nachdem also einem Prozess ein bestimmtes Segment zugewiesen wurde, kann dieses während der Ausführungszeit des Prozesses nicht mehr gewechselt werden.

Fragmentierung. Unter Fragmentierung versteht man den Effekt, dass durch die Aufteilung des physischen Speichers *Reste*, also kleine Speicherbereiche, entstehen, die nicht oder nur schlecht nutzbar sind. Man unterscheidet zwei Arten der Fragmentierung:

1. *Interne Fragmentierung* entsteht dadurch, dass einem Prozess ein größerer Speicherbereich zugewiesen wird, als dieser eigentlich benötigt.
2. *Externe Fragmentierung* liegt vor, wenn zwar nicht zugewiesene Speicherbereiche vorhanden sind, diese aber einzeln zu klein sind, um die Speicherplatzanforderungen von auf Ausführung wartenden Prozessen zu erfüllen.

Sowohl MFT wie auch MVT beruhen beide darauf, dass zu Beginn eines auszuführenden Auftrags bzw. vor einem erneuten Einlagern dessen Speicherplatzanforderungen bekannt sind. Den Platzbedarf von Programmen, die nur statische Datenstrukturen benutzen, kann man recht genau abschätzen. In vielen Fällen hängt der Platzbedarf jedoch von den Eingabedaten oder anderen unvorhersehbaren Faktoren ab, so dass letztlich der Benutzer für jeden Auftrag eine Schätzung des maximalen Speicherplatzbedarfs angeben muss. Wenn im Verlauf der Programmausführung ein höherer Speicherplatzbedarf als geschätzt auftritt, muss das Programm *abgebrochen* werden. Man wird diesen Schätzwert also vorsichtshalber etwas höher wählen. Hierdurch wird einem Prozess ein größerer Hauptspeicherbereich zugewiesen, als dieser eigentlich aktuell benötigt, nach unserer obigen Definition liegt also interne Fragmentierung vor. Diese Art der Fragmentierung wird oft sogar eine der wichtigsten Ursachen für eine schlechte Ausnutzung des vorhandenen Speichers sein. Allerdings kann das Betriebssystem diesen Fehler nicht mit vertretbarem Aufwand oder gar nicht erkennen bzw. behandeln, der Benutzer müsste das Betriebssystem über den aktuellen Speicherplatzbedarf jederzeit auf dem laufenden halten, der benutzte Platz müsste zusammenhängend sein usw. Es handelt sich somit um einen inhärenten Mangel der zusammenhängenden Speicherplatzvergabe.

Weitere Aspekte der Fragmentierung werden weiter unten im Zusammenhang mit Scheduling-Problemen diskutiert werden.

4.3.1 MFT

Wenn bei MFT die Segmentgrößen ungünstig gewählt werden, kann sehr viel interne Fragmentierung auftreten. Nehmen wir z. B. an, wir hätten 1000 kB Platz für Anwenderprogramme, dieser Platz sei in 4 Segmente von 100, 200, 200 und 500 kB aufgeteilt und es seien 6 Aufträge mit je 150 kB Speicherbedarf auszuführen, siehe Abbildung 4.8. Obwohl die Aufträge insgesamt nur 900 kB benötigen und 1000 kB vorhanden sind, können nur 3 Aufträge mit insgesamt 450 kB gleichzeitig ausgeführt werden. 550 kB bleiben unbenutzt, davon 100 kB wegen externer und 450 kB wegen interner Fragmentierung.

Leider ist es fast unmöglich, *günstige* Segmentgrößen zu finden. Die Segmente liegen relativ lange (in der Größenordnung von Tagen oder Wochen) fest und können nur z. B. durch einen Operateureingriff vor dem erneuten Starten des Systems verändert werden. Für einen so langen Zeitraum lässt sich die Mischung der auftretenden Speicheranforderungen normalerweise kaum abschätzen. Darüber hinaus muss wenigstens ein großes Segment vorhanden sein, das den größten Auftrag,

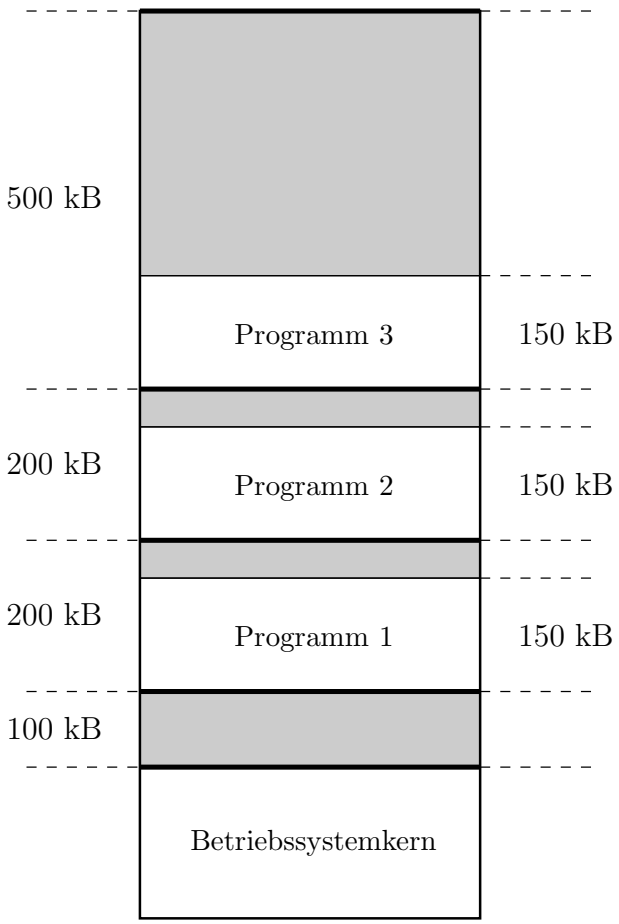


Abbildung 4.8: MFT: Der Speicher wird in vier Segmente von 100, 200, 200 und 500 kB aufgeteilt. Interne Fragmentierung tritt auf.

den ein Anwender ausführen können muss, aufnehmen kann, auch wenn so große Aufträge sehr selten sind.

Die interne Fragmentierung kann durch geschicktes Scheduling verringert werden. Zunächst einmal sollte klar sein, dass einem Auftrag das kleinste ausreichende Segment zugewiesen wird, sofern mehrere zur Auswahl stehen, also kein unnötig großes (**best-available-fit**).

Der nächste Schritt besteht darin, einem Auftrag nur ein Segment zuzuweisen, das *nicht wesentlich* (z. B. maximal die Hälfte) größer als benötigt ist (**best-fit-only**). Ein Auftrag muss dann u. U. zunächst warten, obwohl ein ausreichend großes Segment frei ist. In unserem obigen Beispiel haben wir dem dritten 150-kB-Auftrag das 500-kB-Segment zugewiesen. Wenn nun dieser Auftrag das Segment längere Zeit belegt und inzwischen ein 400-kB-Auftrag ankommt, muss dieser warten. Deshalb ist es vielleicht besser, den kleinen Auftrag zunächst warten zu lassen; wenn nun kein großer Auftrag kommt, war das Warten leider nutzlos. Man erkennt, dass zwar die interne Fragmentierung verringert, möglicherweise aber die externe Fragmentierung stark erhöht wird. Diese Strategie führt praktisch zu einer Klassifizierung aller Aufträge nach ihrer Größe und unabhängigen Schedulern für jede Größenklasse. Der Systemdurchsatz wird hierdurch verbessert, wenn immer genug Aufträge in jeder Größenklasse vorhanden sind, andererseits kann die Wartezeit kleiner Aufträge auf diese Weise unnötig verlängert werden.

Eine falsche bzw. ungünstige Zuordnung eines Segments zu einem Prozess ist nur dann ein Problem, wenn kein Segmentwechsel möglich ist (beim swapping). Systeme, die mit Basisregistern und virtuellen Adressen arbeiten, können das Segment wechseln und ermöglichen deshalb wesentlich flexiblere Reaktionen auf wechselnde Speicherplatzanforderungen der Prozesse.

4.3.2 MVT

Die erheblichen Probleme, die bei MFT infolge der festen Aufteilung des Speichers entstehen, treten bei MVT im Prinzip nicht auf. So kommt es prinzipiell nicht zu interner Fragmentierung.

Es kann jedoch bei MVT zu erheblicher externer Fragmentierung kommen. Dies hängt vor allem davon ab, wie geschickt aus dem freien Speicherplatz Stücke *herausgeschnitten* werden. Dieses Problem wollen wir zunächst in einem etwas allgemeineren Kontext behandeln.

Wir stehen bei MVT vor folgendem allgemeinen Problem: Wir müssen einen von $0 \ldots n$ adressierbaren Array von Speicherzellen verwalten. In diesem Array sind i. A. schon einige Segmente also geschlossene Intervalle der Form $[a, b]$ belegt. Die verbleibenden unbelegten Segmente nennen wir **Lücken**, siehe Abbildung 4.9. Es sind nun zwei verschiedene Ereignisse möglich:

- Ein neues Segment der Länge x wird angefordert. Wir müssen nun eine Lücke finden, deren Länge $\geq x$ ist. Wenn die Lücke genau die Größe x hat, weisen wir diese Lücke der Anforderung zu und sind fertig. Andernfalls teilen wir von der Lücke vorne (oder hinten) ein Stück der Länge x ab und weisen dieses der Anforderung zu; der Rest bildet eine neue, kleinere Lücke. Wenn keine ausreichend große Lücke vorhanden ist, muss mit der Erfüllung der Anforderungen solange gewartet werden, bis durch Freigabe bisher belegter Segmente eine ausreichend große Lücke entstanden ist.

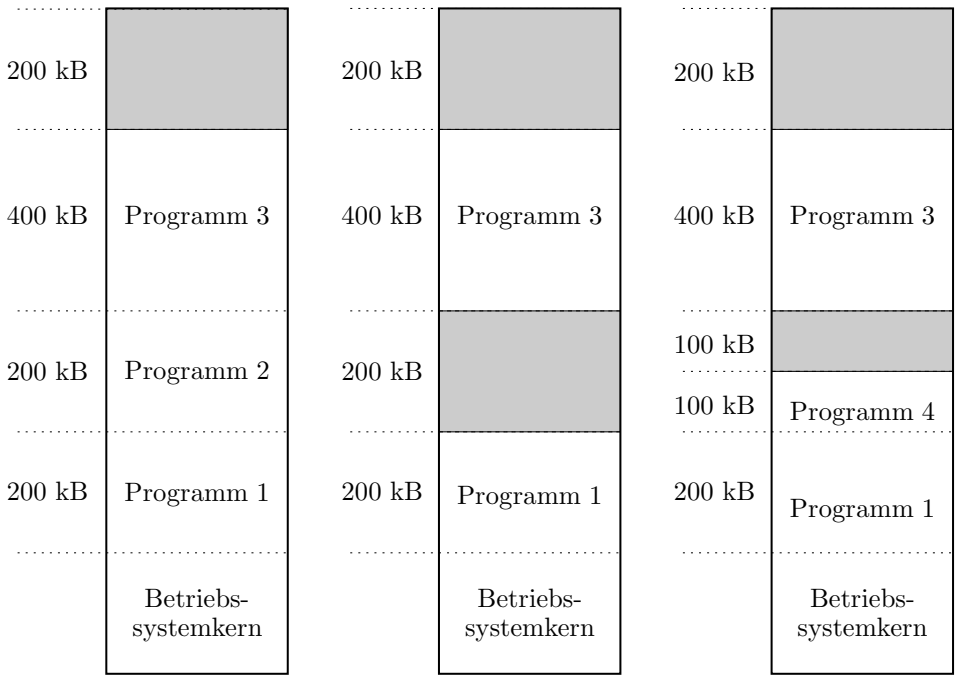
- Ein belegtes Segment wird freigegeben. Aus dem belegten Segment wird jetzt eine Lücke. Wenn diese vorne oder hinten an eine bereits vorhandene Lücke angrenzt, werden die Lücken vereinigt.

Eine Speicherplatzanforderung kann i. A., wenn mehrere ausreichend große Lücken vorhanden sind, auf verschiedene Arten erfüllt werden. Die Entscheidung zwischen den Alternativen treffen wir anhand einer *Speicherplatzvergabestrategie*. Folgende Eigenschaften sollen dabei optimiert werden:

- Die Fragmentierung soll möglichst gering sein. Das Problem ist, dass die Folge der Anforderungen und Freigaben nicht vorab bekannt ist.

- Der Verwaltungsaufwand soll möglichst gering sein.

Das Problem der Freispeicherverwaltung trat übrigens schon bei zwei früheren Gelegenheiten auf: Man legt eine Datei auf einer Festplatte möglichst als durchgehende Folge von Sektoren an, siehe Abschnitt 2.3.3. Statt Bytes verwaltet man hier also Sektoren. Das zweite Beispiel war die Anordnung von Sätzen variabler Länge innerhalb eines Blocks, siehe Abschnitt 2.3.4. In beiden Fällen war das Problem aber weitaus weniger akut als bei MVT: Statt einer einzigen Sektorfolge für die ganze



(a) Bevor Programm 2 terminiert (b) Programm 2 terminiert (c) Programm 4 eingelagert

Abbildung 4.9: MVT: Externe Fragmentierung tritt auf.

Datei kann man notfalls mehrere Sektorfolgen verwenden; dies führt dann lediglich zu zusätzlichen Wartezeiten, reduziert also *nur* die Performance des Dateisystems und kann bei gelegentlichen Reorganisationen des Dateisystems behoben werden. Bei der Anordnung eines Satzes in einem Block kann notfalls der gesamte Blockinhalt reorganisiert werden, denn ein Block ist i. A. relativ klein. In beiden Fällen ist eine optimale Speicherplatzvergabe und die Minimierung der externen Fragmentierung also wünschenswert, bringt aber letztlich nur geringe Performance-Vorteile, so dass nur Speicherplatzvergabestrategien in Frage kommen, die sehr wenig Aufwand verursachen.

Außer in Betriebssystemen tritt das Problem der Freispeicherverwaltung übrigens auch noch in Laufzeitsystemen von Compilern und in Datenbanksystemen auf, allerdings unter wiederum anderen Randbedingungen. Ziel ist auch dort möglichst wenig Fragmentierung; belegte Bereiche sind jedoch leicht verschiebbar, so dass dort Speicherkompaktifizierung wichtig ist.

Bei MVT ist das Vergabeproblem hingegen ganz akut: Bei zu hoher externer Fragmentierung wird die Zahl der parallel (ohne swapping) ausführbaren Prozesse verkleinert, wodurch die gesamte Systemleistung deutlich reduziert werden kann. Aus diesem Grund wird das Speicherplatzvergabeproblem im Zusammenhang mit MVT besonders intensiv behandelt; einige der im Folgenden diskutierten Lösungen sind daher auch nicht sinnvoll auf andere Fälle, wo dieses Problem auftritt, übertragbar.

Im Folgenden werden wir einige Speicherplatzvergabestrategien vorstellen. Sofern nichts anderes erwähnt wird, werden wir dabei stets folgende Annahmen machen:

- Man verwendet als Segmentgröße nur ein ganzes Vielfaches einer sinnvollen

Einheit, z. B. 1 kB. Der angeforderte Platz wird also auf das nächstgrößere ganze Vielfache dieser Einheit aufgerundet. Dadurch bleibt nun ein Bruchteil einer solchen Einheit im Durchschnitt die Hälfte unbenutzt, d. h. es tritt in der letzten für ein Segment vergebenen Einheit interne Fragmentierung auf, die aber insgesamt vernachlässigbar ist. Der entscheidende Vorteil dieser Aufrundung ist, dass sehr kleine Lücken, mit denen man ohnehin nichts mehr anfangen kann, weil normale Speicherplatzanforderungen wesentlich größer sind, und die nur Verwaltungsaufwand verursachen, vermieden werden.

- Wenn bei einer Anforderung nur ein Teil einer Lücke belegt werden muss, dann wird der zu belegende Teil an den oberen oder unteren Rand der Lücke gelegt, und natürlich nicht in die Mitte, wodurch 2 kleinere neue Lücken entstehen würden.

First Fit. Bei first fit wird der Speicher *am Anfang beginnend* durchsucht und die erste verfügbare, ausreichend große Lücke gewählt, siehe Abbildung 4.10, in der die Zuteilung eines Blocks der Größe 16 MByte dargestellt ist. *Erste* bezieht sich dabei auf die Lage der Lücken, d. h. man stellt sich am besten vor, die Lücken seien in einer nach Anfangsadressen aufsteigend sortierten Liste organisiert, die bei der Suche nach einer ausreichend großen Lücke abgesucht wird.

First fit führt dazu, dass bei den kleinen Adressen besonders viele kleine Lücken entstehen (wieso?), so dass die Suche nach großen Lücken lange dauert. Abhilfe schafft das Next Fit.

Next Fit. Dieses funktioniert wie first fit mit der Änderung, dass immer hinter der (ehemaligen) Lücke weitergesucht wird, an der die letzte Zuteilung erfolgte, siehe Abbildung 4.10. Wenn man an das Ende der Liste kommt, fängt man vorne wieder an. Die Suche wird erfolglos abgebrochen, wenn man den Ausgangspunkt wieder erreicht.

Best Fit. Hier wird die (bzw. eine) kleinste ausreichend große Lücke gewählt, siehe Abbildung 4.10. Für best fit muss man die Lücken nach ihrer Größe sortieren, entweder linear oder z. B. in einem binären Baum. Um freigegebene Segmente effizient zusammenlegen zu können, muss man außerdem bei jedem belegten oder freien Segment einen Verweis auf das davor- und das dahinterliegende Segment haben.

Best fit tendiert dazu, sehr viele kleine Lücken zu produzieren, die schlecht oder gar nicht mehr zugewiesen werden können.

Worst Fit. Hier wird, auf den ersten Blick überraschend, die größte vorhandene Lücke gewählt, siehe Abbildung 4.10. Motiviert wird diese Wahl dadurch, dass dann (im Gegensatz zu best fit) auch die Restlücke relativ groß ist und gut weiterverwendet werden kann.

Buddy. Das Buddy-Verfahren unterscheidet sich wesentlich von allen bisher vorgestellten Verfahren. Hauptziel ist hier, die Verwaltung der Segmente zu vereinfachen. Der zugewiesene Speicherplatz wird immer auf die nächsthöhere 2er-Potenz aufgerundet. Bei 100 angeforderten Bytes würden also 128 zugeteilt. Das Buddy-Verfahren führt daher zu einer nicht unerheblichen internen Fragmentierung.

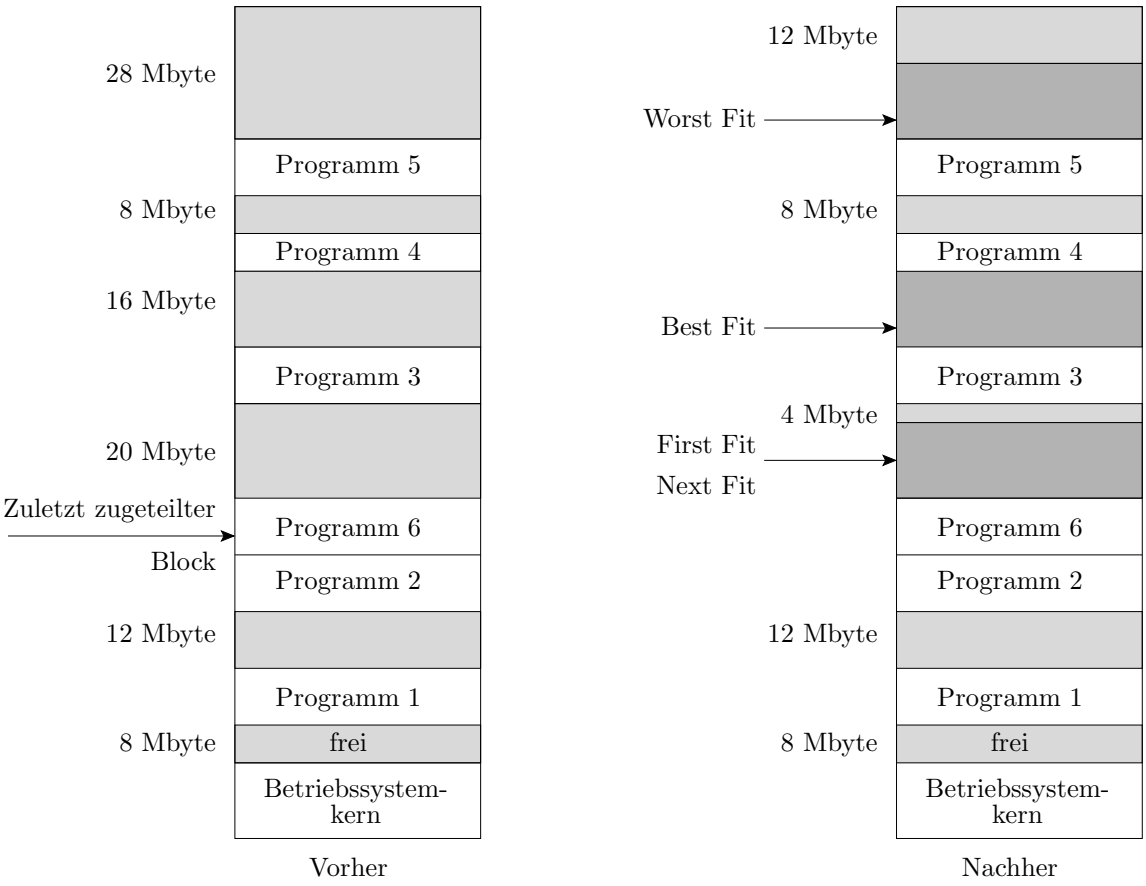


Abbildung 4.10: MVT: Vier verschiedene Speicherplatzvergabestrategien bei der Anforderung eines 16 Mbyte großen Speicherblocks.

Wir nehmen an, die Gesamtgröße des Speichers sei ebenfalls eine 2er-Potenz 2^k. Dieser Speicher wird halbiert in eine *untere* und *obere* Hälfte, die man *buddies* (*Kumpel*) voneinander nennt, siehe Abbildung 4.11. Jede Hälfte kann erneut in zwei zueinandergehörige buddies geteilt werden usw. Man stellt sich die entstehenden Teile am besten als binären Baum vor: der gesamte Speicher auf Ebene 0, dessen Hälften der Größe 2^{k-1} auf Ebene 1, deren Hälften der Größe 2^{k-2} auf Ebene 2 usw. Die freien Segmente der Größe 2^l fasst man in einer Liste L_l zusammen. Abbildung 4.12 zeigt eine Binärbaum der Partnerzuteilung direkt nach der Anforderung D.

Ein angefordertes Segment der Größe 2^x kann dann sofort in Liste L_x gefunden werden. Ist diese Liste leer, wird ein buddy aus Liste L_{x+1} halbiert, wodurch 2 buddies der Größe 2^x entstehen. Ist auch L_{x+1} leer, teilt man einen Buddy der Größe 2^{x+2} usw.

Bei Freigabe eines Segments wird dieses wieder mit seinem buddy zusammengelegt, sofern auch dieser frei ist. Auch die Freigabe kann sich zu den größeren Buddies hin wiederholen. Man kann übrigens den Buddy eines Segments anhand der Adresse und der Größe dieses Segments bestimmen (wie?), es sind hierfür keine weiteren Hilfsdaten erforderlich.

Beim Buddy-Verfahren sind die Anforderungs- und Freigabe-Operationen sehr effizient implementierbar. Andererseits tritt erhebliche interne und u. U. dadurch verursacht externe Fragmentierung auf.

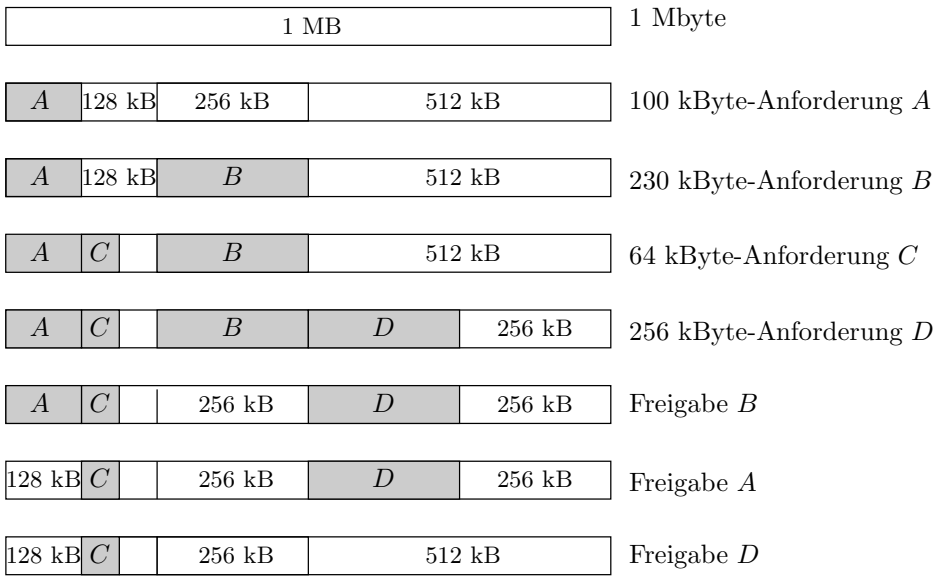


Abbildung 4.11: Das Buddy-Verfahren zur Speicherzuteilung.

Vergleich der Verfahren. Welches dieser Verfahren das Beste ist, ist nicht ganz leicht zu sagen. Man kann für jedes Verfahren leicht Fälle finden, wo gerade dieses Verfahren am besten abschneidet. Die First-Fit-Strategie ist nicht nur die einfachste, sondern in der Regel auch die beste und schnellste.

Die Next-Fit-Strategie führt zu etwas schlechteren Ergebnissen als First Fit. Beim ihr kommt es nämlich häufiger vor, dass ein Block am Ende des freien Speichers zugeteilt wird. Dadurch zerfällt der größte freie Speicherblock, der sich in der Regel am Ende des Speichers befindet, schnell in kleine Fragmente.

Die Best-Fit-Strategie ist trotz ihres Namens meist die schlechteste Wahl. Da sie nach dem kleinstmöglich passenden Speicherblock sucht, wird das zurück bleibende Fragment so klein wie möglich. Obwohl dann durch jede Speicheranforderung immer nur die kleinste Speichermenge verschwendet wird, ist der Hauptspeicher innerhalb kürzester Zeit mit Speicherblöcken übersät, die zu klein sind, um noch irgendwelche Speicherzuteilungsanforderungen erfüllen zu können.

4.3.3 Kompaktifizierung

Wenn die Inhalte von Segmenten verschiebbar sind, kann man in dem Fall, wo die externe Fragmentierung so hoch wird, dass deswegen Speicheranforderungen nicht mehr erfüllt werden können, den gesamten physischen Speicher reorganisieren bzw. kompaktifizieren, in Laufzeitsystemen nennt man dies auch garbage collection. Der Aufwand zum Verschieben größerer Segmente ist natürlich beträchtlich und muss gegen die erzielten Vorteile z. B. höherer Parallelitätsgrad beim multi-programming abgewogen werden.

Im einfachsten Fall werden alle belegten Bereiche nach *vorne* geschoben, wobei ihre Reihenfolge unverändert bleibt. Der gesamte freie Speicher ist dann *hinten* in einer einzigen Lücke verfügbar. Diese *Holzhammermethode* verursacht aber meist unvertretbar viel Aufwand. Man kommt leicht auf bessere Methoden, wenn man beachtet, dass i. d. R. gar nicht so viel freier Speicher benötigt wird und dieser auch nicht unbedingt hinten liegen muss:

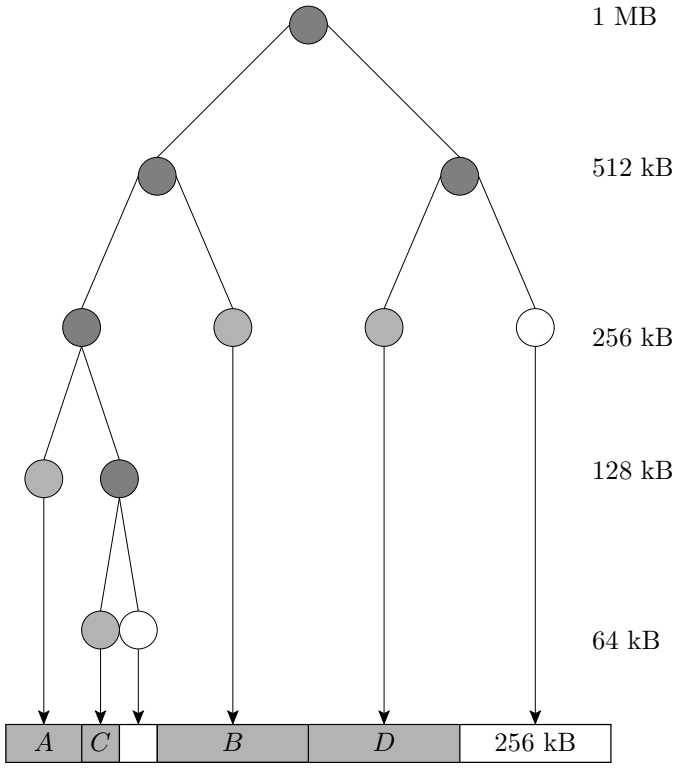


Abbildung 4.12: Baumstruktur des Buddy-Verfahrens.

- Man schiebt nur so lange nach vorne, bis eine ausreichend große Lücke entsteht.
- Man schiebt hinten stehende Segmente in vorne liegende Lücken, wenn diese nur *unwesentlich* größer sind.
- Man sucht einen Teil des Speichers, der nur zu einem geringen Teil belegt ist und verschiebt die dort stehenden Segmente in Lücken in anderen Teilen des Speichers.

Die Auswahl einer optimalen Strategie ist schwierig und muss die konkreten Umstände und Randbedingungen berücksichtigen.

Fast umsonst lässt sich der Speicher kompaktifizieren, wenn das swapping zum Verschieben von Segmenten benutzt wird.

4.4 Nichtzusammenhängende Speicherzuweisung

Wir bleiben in diesem Abschnitt bei der Annahme, dass der physische Speicher in Segmente variabler, von den Applikationen vorgegebener Größe geteilt wird. Im Unterschied zum vorigen Abschnitt gehen wir aber nun davon aus, dass einem Prozess *mehrere* verstreut liegende Segmente des physischen Speichers zugewiesen werden.

Trennung von Programm und Daten. Ein ganz anderer Ansatz zur Verminderung des Fragmentierungsproblems ist, das ausgeführte *Programm* und die *Daten* in getrennten Segmenten zu speichern. Die angeforderten Segmente sind jetzt kleiner; Anforderungen können dadurch eher befriedigt werden. Wir verlassen hier die Vorstellung, dass einem Prozess ein einziger durchgehender Abschnitt des physischen

Speichers zugeteilt wird. Eine ganz wesentliche Frage ist nun, inwieweit dies im logischen Hauptspeicher eines Prozesses sichtbar wird und inwieweit Binder und/oder Lader hiervon betroffen sind. Ohne virtuelle Adressierung ist die Trennung natürlich direkt sichtbar. Der logische Hauptspeicher eines Prozesses ist also nicht mehr durchgehend adressiert, sondern besteht aus **logischen Segmenten**, analog zu den physischen Segmenten, die wir bisher immer als Teil des realen Speichers aufgefasst haben. Virtuelle Adressierung werden wir unten genauer untersuchen.

Speicherschutz. Offensichtlich funktionieren die bisherigen Speicherschutzmechanismen jetzt nicht mehr. Zugriffe sind nur innerhalb der beiden Segmente erlaubt. Dies erfordert für jedes Segment einen eigenen Satz von Schutzregistern. Im Prinzip kann man für die untere Grenze ein Grenzregister oder ein Basisregister einsetzen. Grenzregister haben den Nachteil, dass relativ komplizierte Adressvergleiche durchgeführt werden müssen.

Basisregister erlauben eine wesentlich elegantere Lösung des Problems: man verwendet das erste Bit der virtuellen Adresse für die Unterscheidung, um welches der beiden logischen Segmente es sich handelt. Mit dem nun identifizierten Segment verfährt man wie üblich. Aus Sicht der Anwenderprogramme wird also der logische Adressraum in zwei Hälften geteilt, wobei das Programm in der unteren Hälfte liegt und die Daten in der oberen Hälfte (oder umgekehrt). Ein Vorteil gegenüber der nicht virtuellen Adressierung ist, dass das Datensegment immer die gleiche logische Adresse hat, diese Adresse also nicht erst vom Betriebssystem abgefragt werden muss.

Mehrere Segmente. Man kann statt zwei auch mehr Segmente unterscheiden, z. B. 16, und die virtuellen Adressen in einen oberen Teil (z. B. 4 Bit), die **Segmentnummer**, und einen unteren Teil, die **Relativadresse** innerhalb eines Segments, teilen. Dies kann z. B. sinnvoll sein, wenn das Anwenderprogramm eine Vielzahl von kleineren Datenbereichen dynamisch anlegt und jeweils einzeln Platz dafür vom Betriebssystem anfordert; die Daten können dann auf mehrere Segmente verteilt werden. Man kann die logischen Segmente auch dazu benutzen, Moduln eines Programms darin zu laden.

Wesentlich ist hier, dass aus Sicht der Anwenderprogramme weiterhin nur *ein einziger* logischer Adressraum vorhanden ist, dessen logische Segmentierung normalerweise nicht sichtbar ist[4]. Sichtbar wird die logische Segmentierung bei längeren Operanden, die von einem einzigen Maschinenbefehl verarbeitet werden: diese Operanden dürfen i. A. nicht über Segmentgrenzen hinweg gespeichert werden. Sichtbar wird die Segmentierung auch durch *Löcher*, also nicht realisierte logische Adressen, im logischen Adressraum: Die Menge aller logischen Adressen ist aufgeteilt in gleichgroße Abschnitte der Größe 2^r, wobei r die Länge der Relativadressen ist. Jeder dieser Abschnitte wird durch eine Segmentnummer identifiziert. Wenn einem solchen Abschnitt ein physisches Segment, das kleiner als 2^r ist, zugeordnet wird, dann sind die hinteren Adressen in diesem Abschnitt nicht realisiert, d. h. Zugriffe zu diesen Adressen würden zum Programmabbruch führen, obwohl dahinter wieder andere Adressen kommen, die realisiert sind.

[4]Hiervon zu unterscheiden ist die zweidimensionale Adressierung, die wir weiter unten in Abschnitt 4.6.1 behandeln werden.

Wiedereintrittsfähige Programme (reentrant code). Die Trennung von Programm und Daten ermöglicht in manchen Fällen erhebliche Speicherplatzeinsparungen. Programme wie Editoren, Compiler, Dienstprogramme usw. werden oft von vielen Prozessen *gleichzeitig* ausgeführt; dabei belegt das Programm oft den größten Teil des Speicherplatzes, der von einem Prozess benötigt wird. Anstatt nun für jeden Prozess eine eigene Kopie des Programms zu speichern, liegt es nahe, nur eine einzige Kopie für alle Prozesse zu speichern, wodurch der insgesamt benötigte Speicherplatz oft deutlich reduziert wird.

Dies bedingt zunächst einmal eine Trennung von Programm und Daten in eigene Segmente. Außerdem muss das Programm **wiedereintrittsfähig** (**reentrant**) sein: Insgesamt darf das Programmsegment bei der Ausführung überhaupt nicht verändert werden. Daher darf es z. B. keine statischen Variablen enthalten, denn jeder Prozess braucht ja ein privates Exemplar solcher Variablen. Dies muss einerseits der Compiler oder Assemblerprogrammierer beachten, andererseits kann man dies dadurch kontrollieren, dass man das Programmsegment schreibschützt.

Für einen Compiler bzw. Binder ist es i. A. kein Problem, wiedereintrittsfähige Programme zu erzeugen. In vielen Systemen ist dies daher der Normalfall.

4.5 Paging

4.5.1 Seitenorientierte Speicherzuweisung

Wie wir in den vorigen Abschnitten gesehen haben, verursacht die zusammenhängende Speicherzuweisung viele Probleme, die zur Senkung der gesamten Systemleistung führen können und aufwändige Speicherverwaltungsalgorithmen erforderlich machen. Das Hauptproblem liegt dabei darin, dass

1. die logischen Hauptspeicher *variabel* groß sind und dass
2. je ein logischer Hauptspeicher als ein *durchgehender* Abschnitt des physischen Hauptspeichers realisiert wird.

Alles wäre viel einfacher, wenn alle zu verwaltenden Speicherabschnitte gleich groß wären. An (1) können wir in diesem Sinne nicht viel ändern. Wir können aber bei (2) ansetzen, wenn wir zunächst einmal nicht an den Aufwand denken. Wir erinnern uns, dass wir schon einmal bei Zugriffsmethoden für Zeichen, siehe Abschnitt 2.3.4.2, einen Array von Bytes auf Basis von größeren Verwaltungseinheiten realisiert haben, in diesem Fall von Dateiseiten, die in Sektoren gespeichert werden.

Wir wenden nun die Idee einer seitenorientierten Speicherverwaltung ganz analog auf unser Problem an, um einen mit Adresse 0 beginnenden durchgehend adressierten logischen Hauptspeicher für einen Prozess zu realisieren.

Wir unterteilen den logischen Hauptspeicher in Einheiten einer bestimmten Größe, die **Seiten** (**page**) genannt werden. Die Benutzerprogramme bemerken von dieser Aufteilung nichts! Die logische Adresse wird in eine *Seitennummer* und einen *Offset* geteilt, siehe Abbildung 4.13. Wenn eine Seite p Speicherzellen (Bytes oder Worte) groß ist, dann liegt die logische Adresse v in Seite mit Seitennummer s und Offset d, wobei

$$s = v \text{ div } p \qquad \text{und} \qquad d = v \bmod p.$$

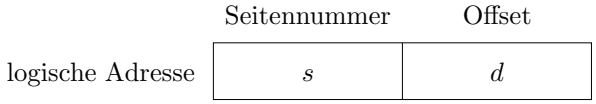


Abbildung 4.13: Eine logische Adresse wird in Seitennummer und Offset zerteilt.

Die entsprechenden Einheiten im physischen Hauptspeicher heißen **Seitenrahmen** (**page frame**). Seiten und Seitenrahmen sind gleich groß, ein Seitenrahmen kann also genau eine Seite eines logischen Hauptspeichers aufnehmen. Wir müssen uns jetzt noch in geeigneter Form merken, in welchen Seitenrahmen die Seiten des logischen Hauptspeichers stehen. Hierzu verwendet man im einfachsten Fall für jeden logischen Hauptspeicher (also für jeden Prozess) eine **Seitentabelle ST**, die zu jeder Seite s die Nummer des Seitenrahmens ST[s] enthält, in dem die Seite s gerade gespeichert ist, siehe Abbildung 4.14.

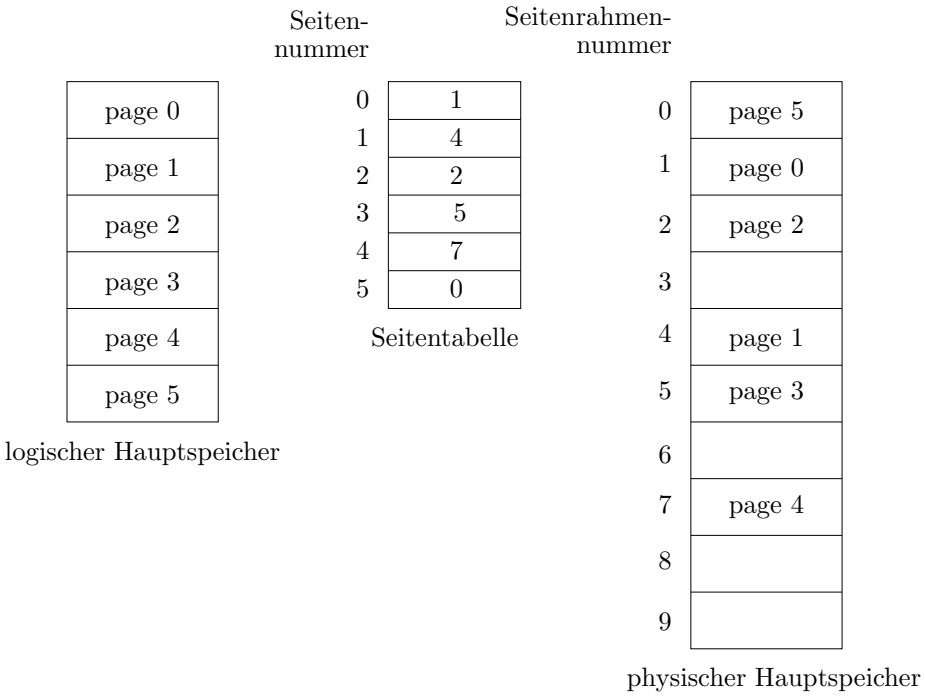


Abbildung 4.14: Ein Beispiel zur Umsetzung von virtuellen in physische Adressen.

Die Adressen müssen natürlich bei *jedem* Hauptspeicherzugriff des Prozessors wie oben beschrieben umgesetzt werden. Aus Performance-Gründen ist klar, dass dies nicht durch Software, sondern nur durch die Hardware durchgeführt werden kann. Der Prozessor arbeitet also zumindest aus der Sicht von Applikationsprogrammen mit einem bei 0 beginnend linear adressierten Hauptspeicher; diese Adressen sind offensichtlich *virtuell*, denn intern wird auf ganz andere physische Adressen zugegriffen. Eine virtuelle Adresse v wird nach folgender Formel in eine physische Adresse r umgeformt, hierbei unterstellen wir einen linear adressierten physischen Adressraum:

$$r = \mathrm{ST}[s] * p + d.$$

Damit diese Umformung möglichst einfach ist, wählt man $p = 2^k$; typische Werte für k sind $9, \ldots, 12$, also $p = 512, \ldots, 4096$. Wenn man $v = \sum_{i=0}^{n-1} a_i 2^i$ als Binärzahl mit

n Stellen darstellt, dann ist die Seitennummer $s = v \text{ div } p = a_{n-1} \cdots a_k$ durch die vorderen $n - k$ Stellen und der Offset $d = v \bmod p = a_{k-1} \cdots a_0$ durch die hinteren k Stellen dargestellt, siehe Abbildung 4.15. Die Seitennummer wird nun als Index für die Seitentabelle benutzt. Die dort gefundene *Seitenrahmennummer* braucht nur noch mit $a_{k-1} \cdots a_0$ konkateniert zu werden, um die gesuchte physische Adresse zu ergeben. Der ganze Vorgang der seitenorientierten Adressumsetzung wird **paging** genannt und ist in Abbildung 4.15 dargestellt.

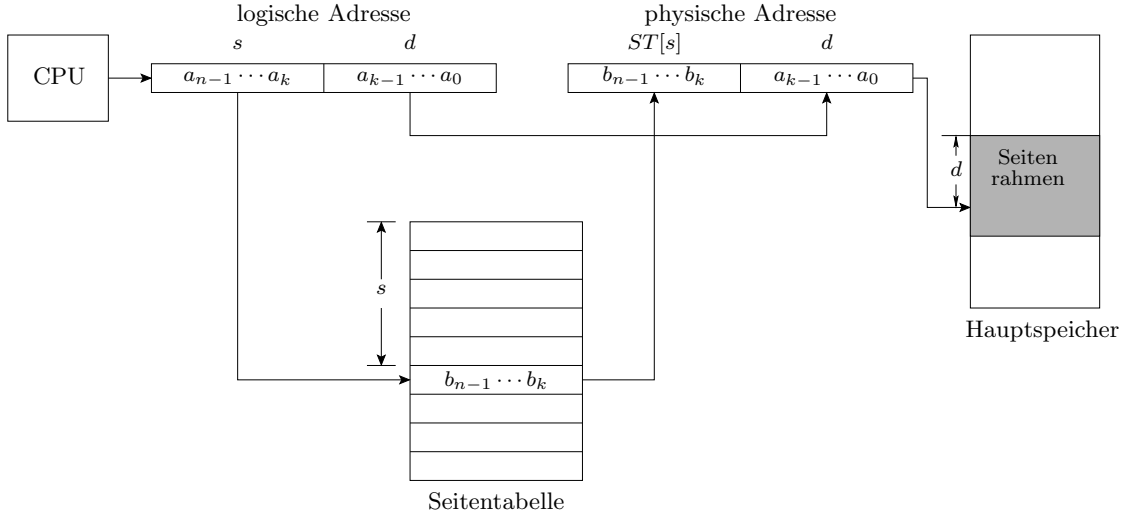


Abbildung 4.15: Die Umsetzung von virtuellen in physische Adressen durch MMU bei der seitenorientierten Speicherverwaltung.

In Abbildung 4.15 sind die virtuelle und die physische Adresse gleich lang mit n Bit. Sie können aber auch ohne weiteres verschieden lang sein. Dies ist sogar vielfach so gemacht worden, um z. B. Computer mit 16-Bit-Prozessoren, die keine längeren Adressen als 16 Bit verarbeiten können, mit größeren Hauptspeichern ausrüsten zu können, z. B. mit 20 Bit langen physischen Adressen. Umgekehrt arbeiten heute die meisten Prozessoren mit 32 Bit langen Adressen, während viele Rechner höchstens auf etwa 64 MB, entsprechend 26 Bit langen physischen Adressen, aufgerüstet werden können.

Man erkennt leicht, dass eine seitenorientierte Speicherverwaltung außerordentlich flexibel sein kann:

- Beim swapping brauchen nicht unbedingt alle Seiten eines Prozesses ausgelagert zu werden, sondern nur so viele, wie freie Seitenrahmen benötigt werden. Wenn wir einmal annehmen, dass eine Seite genau einen Block groß ist (oder ein ganzzahliges Vielfaches davon), dann benötigen wir i. W. wieder eine Tabelle, die zu jeder Seite die Adresse des Sektors angibt, in dem diese Seite gespeichert ist. Statt einer solchen Tabelle könnten im Prinzip auch andere Verfahren zur Verwaltung der Sektoren von Dateien übernommen werden, siehe Kurseinheit 2; aus Gründen, die wir erst später kennenlernen werden, ist eine Tabelle aber besser geeignet.

- Ein existierender logischer Hauptspeicher kann problemlos hinten verlängert werden, indem man weitere Seitenrahmen hinzunimmt und die Seitentabelle entsprechend korrigiert.

- Ein Prozess muss nicht unbedingt einen durchgehenden logischen Hauptspeicher haben. So könnten z. B. die Seiten 0–10 und 20–25 vorhanden sein, während die Seiten 11–19 einfach fehlen. Auf den ersten Blick mag dies nicht sonderlich sinnvoll erscheinen; wir werden aber gleich Probleme im Kontext gemeinsam benutzter Bindemoduln kennenlernen, wo diese Fähigkeit sehr nützlich ist.

Ein wichtiger Vorteil des Verfahren paging ist, dass es externe Fragmentierung nicht gibt und interne Fragmentierung nur aus einem Teil der letzten Seite eines Prozesses besteht.

4.5.2 Zusätzliche Funktionen

Die seitenorientierte Speicherverwaltung ermöglicht gegenüber der zusammenhängenden Speicherzuweisung völlig neue Leistungen der Speicherverwaltung:

Speicherschutz. Wir können jetzt jede einzelne Seite individuell schützen. Wenn z. B. Programme und Daten auf verschiedenen Seiten stehen, kann man die Programmseiten gegen Schreiben schützen, während auf den Datenseiten Schreiben erlaubt ist. Hierzu muss die Seitentabelle ein weiters *Protection-Bit* oder *Schutzbit* enthalten, welches durch die Hardware bei jedem Zugriff geprüft wird. Um Software vor unerlaubtem Kopieren zu schützen, kann man bei einigen Systemen einzelne Seiten zusätzlich gegen Lesen schützen; solche Seiten dürfen nur noch ausgeführt werden, müssen also Maschinenprogramme ohne Daten enthalten. Es kann drei Bits benutzt werden, jeweils eines für das Recht, die Seite zu lesen, zu überschreiben und auszuführen

Shared Memory. Wir haben oben schon die gemeinsame Benutzung von Programmsegmenten durch mehrere Prozesse kennengelernt. Durch die seitenorientierte Verwaltung des Speichers ist es prinzipiell möglich, einzelne Seiten (auch Datenseiten!) in mehreren logischen Hauptspeichern gemeinsam zu benutzen (**shared memory**). Wenn ein Prozess auf eine solche Seite schreibt, erscheint der modifizierte Inhalt auch im Hauptspeicher des anderen Prozesses, sie wird dort sozusagen *eingeblendet*. Dies ist eine außerordentlich effiziente Art, Daten zwischen Prozessen auszutauschen, weil die Daten im physischen Hauptspeicher nicht mehr kopiert werden müssen. Shared memory macht allerdings die Verwaltung der Seitentabellen und Seitenrahmen kompliziert. Die Benutzung von shared memory ist auch programmiertechnisch nicht ganz einfach, denn die Prozesse müssen einander kennen, also z. B. in einer Prozesshierarchie in Beziehung zueinander stehen, und sich bei der Benutzung der gemeinsamen Speicherbereiche synchronisieren, siehe später Kurseinheit 5.

Gemeinsam benutzte Bindemoduln (shared libraries). Wir konnten bisher nur vollständige geladene Programme in mehreren Prozessen gleichzeitig ausführen. Dies ist insofern nicht ganz zufriedenstellend, als Programme oft größtenteils aus Bibliotheksmoduln (Laufzeitbibliothek, Standardroutinen) bestehen und nur wenige individuelle Moduln enthalten. Wenn man nun annimmt, dass jedes Bindemodul innerhalb eines Lademoduls auf einer neuen Seite anfängt, wobei natürlich viele Seiten schlecht gefüllt wären, also in gewissem Umfang interne Fragmentierung

auftreten würde, und dies mit der Idee gemeinsam benutzter Programmsegmente kombiniert, dann liegt es nahe, das Binden von Programmen völlig anders zu gestalten: die benötigten Bindemoduln werden nur noch ein einziges Mal für alle parallel laufenden Prozesse in den physischen Hauptspeicher geladen und in die logischen Hauptspeicher eingeblendet, in denen sie benötigt werden. Es werden also viel *kleinere* Programmteile mehrfach benutzbar, wodurch der Grad der Mehrfachbenutzung geladener Programmseiten erheblich steigt und die erhöhte interne Fragmentierung meist mehr als kompensiert wird.

Wenn Bindemoduln verschiebbar sind, können sie an beliebige Adressen in verschiedenen logischen Hauptspeichern eingeblendet werden. Wenn nicht (weil z. B. absolute Adressen auftreten), muss das Bindemodul bei allen logischen Hauptspeichern an der gleichen Adresse eingeblendet werden. Dies stellt aber kein unlösbares Problem dar, wenn der von dem Prozessor adressierbare Bereich groß genug ist: man reserviert dann einfach einen großen Teil des logischen Hauptspeichers, z. B. die obere Hälfte, für gemeinsam benutzte Software. Hauptnachteil ist hier, dass dieser Teil des Hauptspeichers vom Systemverwalter kontrolliert werden muss, so dass von Anwendern entwickelte Software i. d. R. nicht hier eingefügt werden kann. Man benötigt ferner einen speziellen *absoluten* Lader, der Bindemoduln in bestimmte Seiten lädt und Adresskorrekturen wie sonst beim Binden vornimmt.

Der Bindevorgang muss hier natürlich erheblich anders funktionieren als bei konventionellen Bindern, die ladbare Programme erzeugen. Offene externe Referenzen (z. B. Unterprogrammaufrufe) in den gemeinsam benutzbaren Moduln können nicht mehr dadurch gebunden werden, dass an der aufrufenden Stelle die Adresse des aufgerufenen Unterprogramms eingesetzt wird, denn die mehrfach benutzten Bindemoduln dürfen nicht verändert werden. Stattdessen müssen die Bindungsstrukturen in den Datenbereichen dargestellt werden; die Adressen von externen Referenzen müssen dann immer erst in Register geladen werden.

4.5.3 Implementierungsaspekte

Realisierung der Adressumsetzung. Bei der oben vorgestellten Adressumrechnung von virtuellen in physische Adressen ist zunächst einmal zu klären, wie denn die Seitentabelle realisiert wird.

Es liegt nahe, die Seitentabelle einfach in einem reservierten Teil des physischen Hauptspeichers unterzubringen und bei jeder Adressumrechnung auf diese Tabelle zuzugreifen. Jeder Zugriff zu einer virtuellen Adresse würde dann allerdings *zwei Zugriffe* zu physischen Speicherzellen auslösen, was die Leistung der CPU in etwa halbieren würde, da Zugriffe zum Hauptspeicher sehr lange dauern im Vergleich zu Zugriffen zu Registern oder sonstigen Vorgängen innerhalb der CPU.

Die Seitentabelle des gerade ausgeführten Prozesses muss also in viel schnelleren und damit auch sehr teuren Speichern untergebracht werden. Die Idee, eine feste Anzahl von Registern hierfür vorzusehen, führt auch nicht weiter: aus Kostengründen sind maximal einige Dutzend oder Hundert solcher Register möglich, so dass selbst bei recht großen Seiten (z. B. 4 kB) die maximale Größe eines virtuellen Speichers bei wenigen MB liegen würde. Die Idee ist auch deshalb nicht sehr attraktiv, weil Programme normalerweise nur auf wenigen Seiten arbeiten und deshalb immer nur sehr wenige der teuren Register tatsächlich benutzt werden würden.

Besser wäre es, nur die *aktuell* benutzten Einträge der Seitentabelle in schnel-

len Speichern zu halten und die *komplette* Seitentabelle im langsameren Hauptspeicher. Deshalb werden in den meisten Rechnern schnelle **Assoziativspeicher** (**TLB, Translation Lookaside Buffer**) eingesetzt, siehe Abbildung 4.16. Diese enthalten etwa 8–16 Einträge der Form

(Seitennummer, zugehörige Seitenrahmennummer).

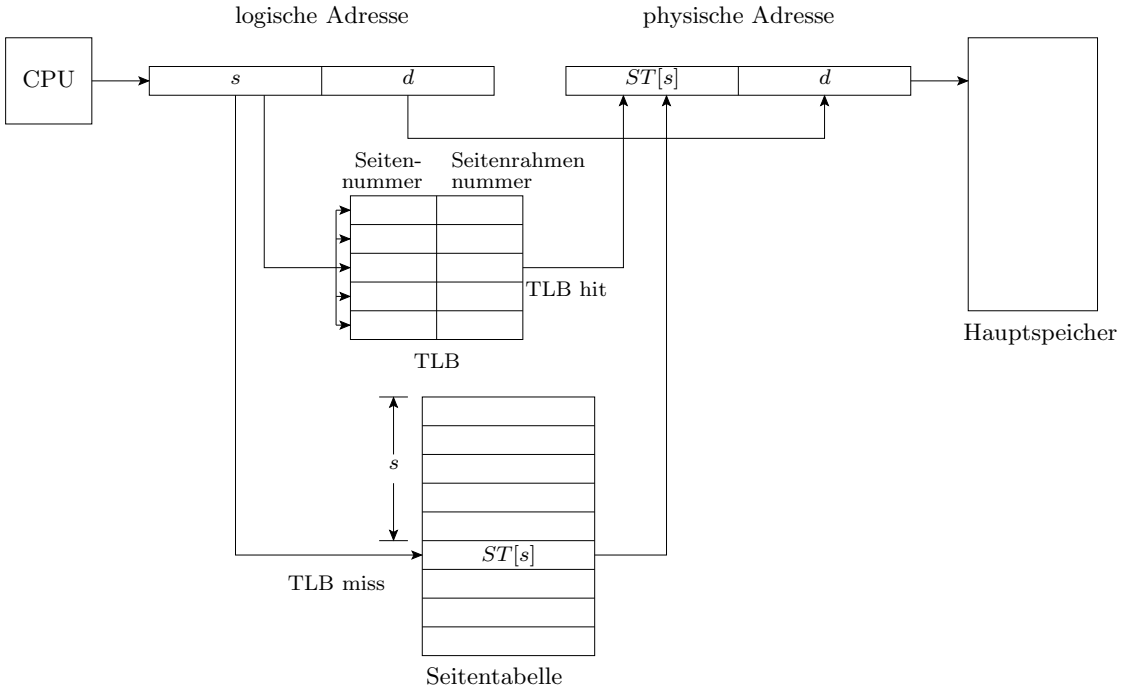


Abbildung 4.16: Die Umsetzung von virtuellen in physische Adressen mit Hilfe von TLB.

Man beachte, dass hier die Seitennummer explizit gespeichert ist, während sie in der Seitentabelle implizit durch die Tabellenposition gegeben ist. Wenn dem Assoziativspeicher jetzt eine Seitennummer präsentiert wird, wird diese parallel in allen Einträgen mit den dort vorhandenen Seitennummern verglichen, d. h. für jeden Eintrag sind eigene Schaltkreise vorhanden, die den Vergleich durchführen. Wenn die gesuchte Seitennummer vorhanden ist (*TLB hit*), wird die zugehörige Seitenrahmennummer ausgegeben. Der ganze Vorgang ist typischerweise um den Faktor 10 schneller als ein Hauptspeicherzugriff. Da außerdem schon bei relativ wenigen Plätzen im Assoziativspeicher (8–16) eine Trefferquote von 80–90 % erzielt wird, ist die durchschnittliche Gesamtverzögerung durch die Adressumsetzung erträglich. Wenn der Assoziativspeicher die gesuchte Seitennummer nicht enthält (*TLB miss*), wird die zugehörige Seitenrahmennummer aus der Seitentabelle geholt und in den Assoziativspeicher eingetragen. Hierbei muss i. d. R. ein schon vorhandener Eintrag verdrängt werden. Wenn zum Beispiel die Suche nach einer Seitennummer in TLB nur 10 ns beträgt und der Zugriff auf den Hauptspeicher 100 ns, dann beträgt der Zugriff auf eine Seite 110 ns, wenn ihre Seitennummer im TLB ist. Bei einem TLB miss dagegen gibt es zwei Zugriffe auf den Hauptspeicher, und das ganze dauert insgesamt 210 ns. Wenn im TLB eine Trefferquote 80 % erzielt werden kann, ist die durchschnittliche Zugriffszeit (Gesamtverzögerung) auf eine Seite

$$0.80 \cdot 110 + 0.20 \cdot 210 = 130 \text{ ns}.$$

Diese durchschnittliche Verzögerung ist nur 30 % langsamer als die Zugriffszeit auf den Hauptspeicher.

Speicherung der Seitentabelle. Bei einem logischen Adressraum von z. B. 2^{32} Adressen (Adresslänge 32 Bit) und einer Seitengröße von 2^{12} Bytes hat die Seitentabelle maximal $\frac{2^{32}}{2^{12}} = 2^{20}$ Einträge. Diese Tabelle kann außerdem z. B. infolge von gemeinsam benutzten Bindemoduln stark gestreut belegt sein. Aus diesem Grund wird in manchen Systemen die Seitentabelle durch *mehrstufige* Seitentabellen realisiert. Die Seitentabellen werden nicht im realen Speicher, sondern auch im virtuellen Speicher abgelegt. Das bedeutet, dass Seitentabellen genauso wie alle anderen Seiten dem Paging unterliegen. Der Zweck der mehrstufige Seitentabellen ist, dass nicht mehr alle Seitentabellen gleichzeitig im Speicher gehalten werden müssen. Besonders die Tabellen, die nicht gebraucht werden, sollten nicht im Speicher liegen. Bei einem zweistufigen Seitentabellen–Verfahren gibt es eine erste Stufe der Seitentabelle, bei der jeder Eintrag die Adresse oder die Seitenrahmennummer einer Seitentabelle der zweiten Stufe enthält. Wenn die Länge der Seitentabelle der ersten Stufe p_1 ist und wenn die Länge einer Seitentabelle der zweiten Stufe maximal p_2 beträgt, kann ein Prozess aus bis zu $p_1 \cdot p_2$ Seiten bestehen. *In der Regel darf eine Seitentabelle maximal so lang sein wie eine Seite.*

Betrachten wir beispielweise noch einmal den 32 Bit langen logischen Adressraum mit einer 4 kByte–Seitengröße. Der Adressraum besteht aus also $\frac{2^{32} \cdot 4 \text{ Byte}}{4 \cdot 2^{10} \text{ Byte}} = 2^{20}$ Seiten. Wenn jede dieser Seiten durch einen 4 Byte großen Seitentabelleneintrag verwaltet wird, entsteht eine Seitentabelle mit 2^{20} Einträgen, die $2^{22} = 2^{20} \cdot 4$ Byte braucht. Diese Seitentabelle belegt wieder 2^{10} Seiten. Sie kann im virtuellen Speicher gespeichert und mittels einer Hauptseitentabelle, die 2^{10} Einträge enthält und 4 kByte Hauptspeicher belegt, referenziert werden, siehe Abbildung 4.17. Also 32 wird nun in 3 Teile von 10, 10 und 12 Bits geteilt, Die ersten Bits werden als Index für die erste Stufige der Seitentabelle benutzt. Die zweiten Bits wird als Index für die ausgewählte Seitentabelle der zweiten Stufe benutzt, um die Seitenrahmennummer der gesuchten Seite selbst zu finden. Man vergleiche dies mit den indirekten Sektoradressen in UNIX, siehe Abschnitt 2.3.3.1.

Seitenrahmentabelle. Das Betriebssystem muss eine weitere, globale Tabelle verwalten, in der Informationen über den Zustand aller Seitenrahmen enthalten sind: die **Seitenrahmentabelle**. Zu jedem Seitenrahmen müssen wir uns merken, ob er eine Seite enthält und, wenn ja, welche Seite welchen Prozesses. Bei shared memory muss u. U. sogar eine Menge von Paaren

(Prozessidentifizierer, Seitennummer)

für einen Seitenrahmen gespeichert werden.

Adressumsetzung im Systemmodus. Das Betriebssystem muss unbedingt fähig sein, mit physischen Adressen zu arbeiten; beispielsweise muss das Betriebssystem bei Seitenein- oder -auslagerungen die Seitentabellen, die ja nicht Teil irgendeines virtuellen Hauptspeichers sind, modifizieren können. Im Systemmodus werden Adressen daher nicht als virtuelle, sondern physische Adressen behandelt. Dies bedingt natürlich, dass das Betriebssystem virtuelle Adressen *von Hand* in physische

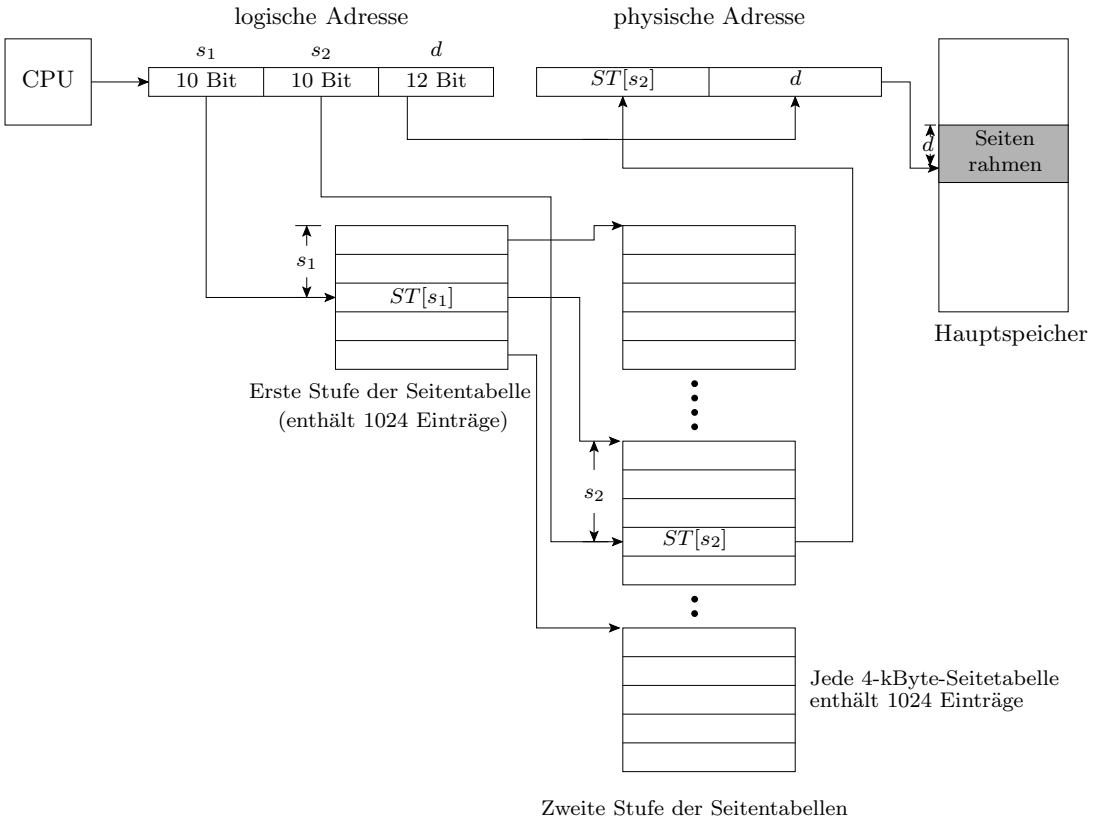


Abbildung 4.17: Zweistufige Seitentabellen.

umrechnen muss. Dies ist allerdings nur selten erforderlich, so dass es kein wesentliches Performance-Problem darstellt. Nützlich ist in diesem Zusammenhang aber, wenn der Prozessor es gestattet, die Adressumsetzung ein- und auszuschalten.

4.6 Virtuelle Hauptspeicher

4.6.1 Demand Paging

Wir waren bisher davon ausgegangen, dass ein Prozess nur dann weiter ausgeführt wird, also die CPU zugeteilt bekommt, wenn sich *alle* seine Seiten im physischen Hauptspeicher befinden. Wenn man jetzt noch einmal an die Zugriffsmethoden für Zeichen auf Basis von Seiten denkt und sich die bisher vorgestellten Mechanismen noch einmal in Ruhe ansieht, bemerkt man, dass man durch eine relativ kleine Erweiterung die Flexibilität noch einmal erheblich steigern kann: eine zeichenorientierte Datei ist auch eine mit 0 beginnend adressierte Folge von Speicherzellen. Beim Direktzugriff zur i-ten Speicherzelle (in POSIX mit der Operation `lseek`) wird der entsprechende Block in den Puffer geholt, sofern er noch nicht dort ist, so dass die gesuchte Speicherstelle anschließend verarbeitet werden kann. Diese Idee, dass der Inhalt einer Datei zunächst einmal auf einem Hintergrundspeicher steht und erst bei Bedarf in den physischen Hauptspeicher geholt wird, können wir auf die Verwaltung der logischen Hauptsspeicher übertragen.

Für jeden logischen Hauptspeicher brauchen wir also Platz auf einem Hintergrundspeicher (i. A. einer Festplatte), dem sog. **Swap-Bereich**, der beim Start des

Betriebssystems reserviert wird. Eine mögliche Implementierung ist nun, jeden Prozess einen Teil des Swap-Bereichs zu geben, der so groß ist wie der gesamte Hauptspeicherbedarf des Prozesses. Damit handelt man sich aber wieder dieselben Probleme ein wie bei der statischen Verwaltung des Hauptspeichers: Prozesse können zur Laufzeit immer mehr Hauptspeicher anfordern, so dass der reservierte Bereich nicht mehr groß genug ist. Deshalb reserviert man besser keine zusammenhängenden Blöcke im Swap-Bereich, sondern auch nur einzelne Seiten: Nur wenn eine Seite ausgelagert werden muss, wird sie auf der Festplatte untergebracht, wobei die Adresse jetzt in einer Zuordnungstabelle verwaltet werden muss. Diese Tabelle kann ziemlich groß werden, weil sie einen Eintrag für jede ausgelagerte Seite speichert. Der Eintrag wird aber sofort gelöscht, wenn die Seite wieder eingelagert wird.

Beim Zugriff auf eine Seite können nun zwei Fälle eintreten:

1. Die Seite befindet sich im Hauptspeicher: Wir können wie oben beschrieben die physische Adresse ausrechnen.

2. Die Seite ist nicht im Hauptspeicher: man bezeichnet dies als **Seitenfehler** (**page fault**). Wir müssen die Abarbeitung des Befehls zunächst einmal abbrechen, es findet eine Unterbrechung, ein so genannter *Trap*, statt, denn jetzt muss zuerst die Seite vom Hintergrundspeicher in einen freien Seitenrahmen eingelagert werden. Dies dauert aber so lange, dass man u. U. einen Prozesswechsel vornimmt, wenn ein anderer Prozess ausführungsbereit ist. Schließlich wird der unterbrochene Prozess fortgesetzt und der abgebrochene Befehl erneut ausgeführt, diesmal ohne Seitenfehler.

Wir haben im Fall 2 stillschweigend angenommen, dass noch ein freier Seitenrahmen vorhanden ist. Was aber, wenn alle Seitenrahmen schon belegt sind? Wir müssen dann offensichtlich einen Seitenrahmen freimachen. Nach welchen Kriterien wir das *Opfer* aussuchen, lassen wir zunächst einmal offen. Die ausgesuchte Seite muss auf den Hintergrundspeicher ausgelagert werden, damit wir sie später bei Bedarf wieder einlagern können. Genaugenommen gilt dies aber nur in dem Fall, dass diese Seite seit dem letzten Zeitpunkt, an dem sie eingelagert worden war, verändert worden ist. Um dies erkennen und unnötige Seitenauslagerungen vermeiden zu können, verwendet man üblicherweise ein sog. **dirty bit** für jeden Seitenrahmen. Nach dem Einlagern einer Seite wird dieses Bit auf 0 gesetzt. Jeder schreibende Zugriff zu einer Seite setzt als Seiteneffekt (durch die Hardware) das Bit auf 1. Die in einem Seitenrahmen enthaltene Seite braucht also nur dann ausgelagert zu werden, wenn das dirty bit auf 1 gesetzt ist.

Den oben beschriebenen Vorgang des Ein- und Auslagerns von Seiten nennt man **demand paging**, weil Seiten immer erst bei Bedarf eingelagert werden, nie auf Vorrat. Im Gegensatz dazu werden beim **prepaging** Seiten eingelagert, die aktuell noch nicht, vermutlich aber in Zukunft benötigt werden. Das prepaging erweist sich wegen des damit einhergehenden Verdrängens schon vorhandener Seiten normalerweise als nachteilig und wird nur in Ausnahmefällen angewandt.

Demand paging hat bemerkenswerte Auswirkungen auf unsere Vorstellung von der Natur des logischen Hauptspeichers eines Prozesses: es ist theoretisch ausreichend, wenn jeweils nur die von einem Befehl betroffenen Speicherzellen im physischen Speicher vorhanden sind. Diese Zahl (maximal etwa 8) hängt i. W. nur von

der Zahl der Operanden der Prozessorbefehle, von deren Größe und von den Adressierungsarten ab, nicht aber von der Größe des Programms oder der Datenbereiche. Theoretisch können wir beliebig große Programme mit sehr kleinen physischen Hauptspeichern ausführen![5] Wir können jetzt vor allem aber *logische Adressräume realisieren, die größer als der physische Speicher sind!* Deshalb spricht man hier von **virtuellen Hauptspeichern**. Zum Vergleich: bei der zusammenhängenden Speicherzuweisung traten zwar auch schon virtuelle *Adressen* auf; der logische Hauptspeicher eines Prozesses war aber immer *vollständig* im physischen Hauptspeicher realisiert, wenn der Prozess ausgeführt wurde.

Demand paging kostet natürlich viel Zeit; man wird sich fragen, ob die ganze Idee überhaupt sinnvoll ist und ob man nicht so oder so das komplette Programm inkl. aller Daten im Hauptspeicher haben muss. Dies ist zum Glück nicht der Fall. Ursache ist die *Lokalität des Zugriffsverhaltens* von Prozessen. So werden überwiegend hintereinander stehende Instruktionen oder Schleifen, die nur wenige Seiten umfassen, ausgeführt und es wird relativ selten zu *entfernten* Programmteilen gesprungen. Verschachtelte Datenstrukturen werden meist linearisiert und dadurch auf eine minimale Zahl von Seiten konzentriert. Manche Programmteile z. B. Initialisierungen und Spezialfälle werden nur sehr selten durchlaufen. Insgesamt kommt man im zeitlichen Mittel mit einem Bruchteil aller Seiten des logischen Hauptspeichers aus, ohne dass Seitenfehler zu häufig werden. Das Konzept des virtuellen Speichers basiert ganz wesentlich auf dieser **Lokalitätsannahme**.

Zusammengefasst haben virtuelle Speicher für den Anwender, für Entwickler von Anwenderprogrammen und für Entwickler von Betriebssystemen ganz erhebliche Vorteile:

- Es müssen nicht alle Programme und Daten aller laufenden Prozesse im physischen Hauptspeicher gehalten werden, sondern wegen der Lokalität des Zugriffsverhaltens in der Regel nur ein Bruchteil davon. Umgekehrt gesehen kann mit einem gegebenen physischen Speicher ein höherer Parallelitätsgrad und damit eine insgesamt bessere Rechnerausnutzung erzielt werden.

- Es lassen sich beliebig große virtuelle Adressräume realisieren bzw. benutzen. Begrenzt sind die Adressräume nur durch die Länge der Adressen, mit denen der Prozessor arbeitet (heute meist 32 Bit). Realisiert werden kann natürlich nur so viel virtueller Hauptspeicher, wie Platz auf dem Hintergrundspeicher vorhanden ist.

- Der virtuelle Speicher muss nicht durchgehend benutzt werden. Anwendungsprogrammierer können z. B. im Prinzip beliebig großzügige Platzreserven zwischen dynamisch angelegten Datenstrukturen anlegen.

Paging bzw. virtuelle Speicher erfordern zwar eine Reihe von Voraussetzungen in der Hardware, sind aber heute bei allen Rechnern mit Ausnahme der untersten Leistungsklasse (einfache PCs, Spielcomputer) Standard. Eine seitenorientierte Speicherverwaltung ohne virtuellen Hauptspeicher wäre zwar auch denkbar, ist aber

[5] Praktisch natürlich nicht, weil dies wegen der häufigen Seitenfehler zu langsam wäre. Außerdem begrenzt die Länge der Adressen und (in der Praxis am ehesten bemerkbar) die Größe des Swap-Bereichs auf der Festplatte die mögliche Größe der virtuellen Hauptspeicher.

nicht sehr sinnvoll und offenbar nie realisiert worden, da der zusätzliche Nutzen durch den virtuellen Hauptspeicher die zusätzlichen Kosten z. B. zusätzlich erforderliche Hardware-Unterstützung bei weitem überwiegt. Die schon oben erwähnten Vorteile einer seitenorientierten Speicherverwaltung werden durch virtuelle Hauptspeicher noch wesentlich verstärkt.

Memory-mapped IO. Memory-mapped IO ist eine Funktion, die zwar schon bei einer seitenorientierten Speicherverwaltung realisierbar wäre, aber erst im Zusammenhang mit virtuellen Hauptspeichern sinnvoll ist.

Die Idee ist denkbar einfach: man *blendet* Seiten einer Datei in den virtuellen Hauptspeicher ein. Beispielsweise könnte man die 7. Seite einer Datei D in die 134. Seite eines virtuellen Hauptspeichers V einblenden, wir setzen hier generell voraus, dass Dateiseiten und Hauptspeicherseiten gleich groß sind. Der zu V gehörige Prozess sieht dann auf der 134. Seite die Daten, die in der 7. Seite der Datei stehen. Verändert der Prozess diese Daten, dann verändert er implizit auch den Inhalt der Datei! Aufeinanderfolgende Seiten von V brauchen nicht notwendigerweise aufeinanderfolgende Seiten von D zu enthalten; die Datei braucht auch nicht komplett eingeblendet zu werden. Seiten einer Datei können sogar mehrfach gleichzeitig oder zu unterschiedlichen Zeitpunkten an verschiedenen Stellen von V eingeblendet werden.

Das memory-mapped IO bietet in bestimmten Fällen insb. bei nichtsequentieller Verarbeitung einer Datei erhebliche Vorteile gegenüber den klassischen E/A-Verfahren: bei diesen kann das Anwendungsprogramm immer nur auf einen sehr kleinen Teil der Datei direkt zugreifen, nämlich den Satz oder das Zeichen, auf das der Dateizeiger gerade zeigt. Größere Abschnitte der Datei müssen von der Applikation immer erst explizit in den eigenen Hauptspeicher bzw. später zurück in die Datei *kopiert* werden. Diese Seiten existieren dann einmal in der Datei, vielleicht noch einmal in den Dateisystem-Puffern, dann im physischen Hauptspeicher als Teil des logischen Hauptspeichers des Anwenderprozesses und zum Schluss noch einmal im Swap-Bereich auf der Festplatte.

Seiten einer Datei können beim memory-mapped IO sogar in mehrere virtuelle Adressräume analog zu shared memory eingeblendet werden; siehe Abbildung 4.18. Dies kann wieder zur Interprozesskommunikation benutzt werden und hat gegenüber dem klassischen shared-memory-Konzept große Vorteile: dort existiert der Kommunikationsbereich nur als Teil der Adressräume der Prozesse, und die kommunizierenden Prozesse müssen einander kennen, z. B. in der Prozesshierarchie miteinander verbunden sein. Dateien existieren dagegen persistent und haben global eindeutige Namen.

Applikationen können zur Laufzeit mit Hilfe entsprechender Systemaufrufe Dateiseiten einblenden oder solche Einblendungen wieder aufheben, analog zum Öffnen und Schließen von Dateien, sowie geänderte Seiten auf die Datei zurückschreiben, also permanent machen. Für das Einblenden von Dateiseiten in *mehrere* virtuelle Hauptspeicher benötigt man einige zusätzliche Steuerungsfunktionen:

- Wenn die Seiten nur zum Lesen eingeblendet werden sollen, müssen sie gegen (versehentliches) Schreiben durch diesen oder andere Prozesse geschützt werden. Dies kann analog zu Dateien durch Unterscheiden eines Lese- und Schreibmodus und durch Sperren geschehen.

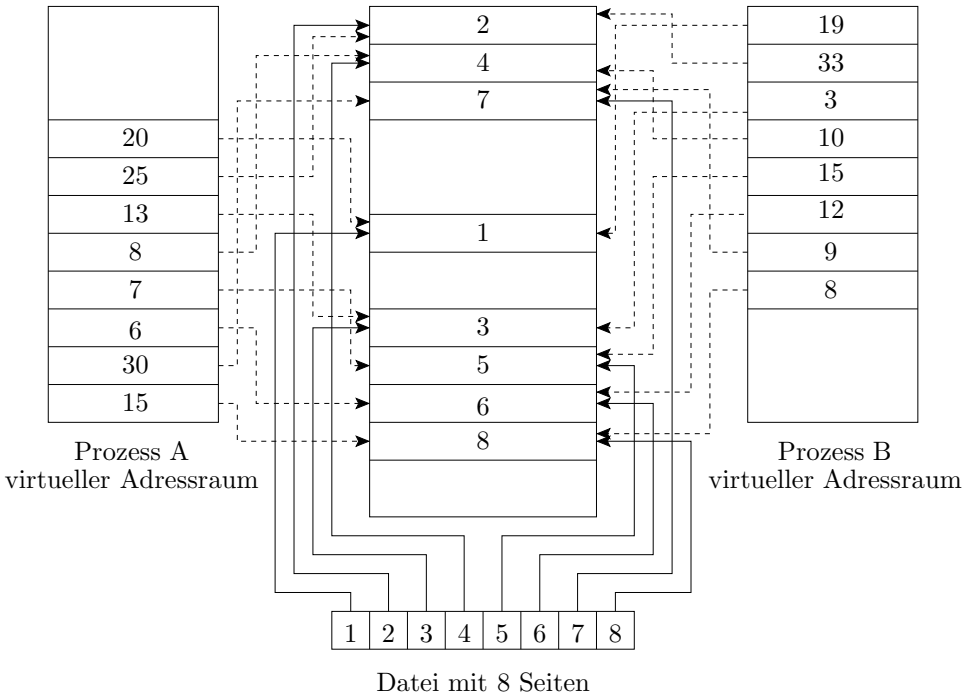


Abbildung 4.18: Zwei Prozesse bearbeiten eine Datei, deren Seiten werden in die virtuellen Adressräume der Prozesse eingeblendet.

- Oft soll der Inhalt der Datei nicht verändert werden, die Applikation will aber trotzdem Seiteninhalte verändern können. Dies kann durch das Kopieren beim Schreiben (**copy on write**) erzielt werden: sobald eine nur zum Lesen eingeblendete Seite beschrieben wird, wird sie wie eine normale Seite des virtuellen Speichers behandelt, d. h. bei der nächsten Auslagerung nicht mehr in die Datei, sondern in den Swap-Bereich geschrieben. Wenn die Applikation diese Seite auf Dauer retten möchte, muss sie sie wie üblich in eine Datei kopieren.

Das memory-mapped IO ersetzt nur Zugriffsmethoden für Zeichen, keine Zugriffsmethoden für Sätze. Implementierungen von Zugriffsmethoden für Sätze können aber das memory-mapped IO ausnutzen.

Zweidimensionale logische Adressräume. Obwohl shared memory, gemeinsam benutzte Bindemoduln und memory-mapped IO erst in letzter Zeit in kommerziellen Betriebssystemen auftreten, sind diese Techniken keineswegs neu. Sie traten alle schon vor rund 30 Jahren im Betriebssystem Multics [Or72] auf, das in vieler Hinsicht ein Meilenstein in der Geschichte der Betriebssysteme war. Da es eine sehr spezielle Hardware benötigte (Rechner des Typs GE 645), hat es sich jedoch nicht sehr verbreitet.

In Multics arbeitet ein Prozess gleichzeitig mit mehreren bei 0 beginnenden linearen Adressräumen von 2^{16} Worten. Die Adressräume werden Segmente genannt und sind für Anwenderprogramme sichtbar. Ein Multics-Segment entspricht einer Datei, die ab Adresse 0 in das Segment eingeblendet wird. Jeder Datei ist eine feste, 18 Bit lange Segmentnummer zugeordnet. Eine logische Adresse besteht somit aus einer Segmentnummer und einer relativen Adresse im Segment. Man könnte die Konkatenierung der beiden Zahlen als eine einzige Zahl auffassen - tatsächlich werden die

beiden Zahlen sogar in dieser Form gehandhabt - und hätte dann wieder nur einen Adressraum. Der Unterschied zu der eindimensionalen Adressierung liegt vor allem in einigen Maßnahmen, die das dynamische Binden von in Segmenten enthaltenen Bindemoduln unterstützen, z. B. Vermeiden direkter Adressen, vgl. Abschnitt 4.5.2, und in damit integrierten Schutzmaßnahmen.

4.6.2 Seitenauslagerungsstrategien

Nachdem wir jetzt die Grundidee der seitenorientierten Speicherverwaltung und virtueller Speicher erklärt haben, bleiben eine ganze Reihe von Details zu untersuchen:

- Wie vermeidet man zu häufige Seitenfehler?
- Wie sucht man *Opfer* bei der Seitenauslagerung aus?
- Welche Zusammenhänge bestehen zwischen der Seitenauslagerungsstrategie und dem Scheduling?

Kosten eines Seitenfehlers. Es ist intuitiv klar, dass Seitenfehler Aufwand verursachen, der das gesamte System belastet, sowie die Ausführung des betroffenen Prozesses verlangsamen und deshalb vermieden werden sollten. Wir wollen uns zunächst einmal eine Vorstellung von der Größenordnung dieser *Kosten* eines Seitenfehlers verschaffen, um auf dieser Basis diskutieren zu können, wieviele Seitenfehler denn auftreten dürfen, ohne dass sie merklich stören.

Ein Seitenfehler macht zumindest die Einlagerung der fehlenden Seite erforderlich, eventuell zusätzlich die Auslagerung einer anderen Seite. Ein Festplattenzugriff dauert i. A. deutlich mehr als 10^5 CPU-Instruktionen. Wenn also die Ausführungsgeschwindigkeit eines Prozesses nicht wesentlich durch die Wartezeiten bei Seiteneinlagerungen verringert werden soll (z. B. nicht mehr als 10 %), dann darf maximal 1 Seitenfehler auf 10^6 Instruktionen eintreten.

Bei einem Seitenfehler ist außerdem ein Prozesswechsel angebracht, für den die CPU typischerweise mehrere tausend Instruktionen ausführen muss. Wenn dieser unproduktive Anteil der Arbeit der CPU nicht mehr als 10 % ausmachen soll, müssen aus Sicht der CPU-Belastung Seitenfehler insgesamt (über alle Prozesse hinweg) mit einer wesentlich kleineren Wahrscheinlichkeit als 10^{-4} auftreten.

Diese überschlägigen Rechnungen zeigen, dass Seitenfehler nur sehr selten auftreten dürfen.

Minimierung der Seitenfehlerrate. Was können wir nun tun, um Seitenfehler unwahrscheinlich zu machen?

1. Wir lassen natürlich keine Seitenrahmen unbenutzt.
2. Wir können den einzelnen Prozessen mehr Seitenrahmen zuweisen, weil dann meist - leider nicht immer - weniger Seitenfehler auftreten. Da wir die Zahl der physisch vorhandenen Seitenrahmen als fest vorgegeben annehmen, muss also notgedrungen die Zahl der parallelen Prozesse und damit u. U. die Gesamtauslastung des Systems sinken; dies ist nicht unbedingt wünschenswert. Wir stehen also vor dem Problem, jedem einzelnen Prozess im System eine günstige Anzahl von Seitenrahmen zuzuweisen. Eine (Seitenrahmen-) **Zuweisungsstrategie** legt fest, wann einem Prozess wieviele Seitenrahmen zugewiesen

bzw. entzogen werden. Eine sehr einfache Strategie ist, die vorhandenen Seitenrahmen gleichmäßig auf die im System vorhandenen Prozesse zu verteilen. Wie wir noch sehen werden, ist diese Strategie ziemlich schlecht. Bessere Strategien berücksichtigen sowohl das Zugriffsverhalten jedes einzelnen Prozesses, also *lokale* Kriterien, wie auch die Gesamtbelastung des Rechnersystems, also *globale* Kriterien.

3. Wenn wir nun davon ausgehen, dass einem Prozess über einen gewissen Zeitraum hinweg eine feste maximale Anzahl von Seitenrahmen zugewiesen ist, sollten diese Seitenrahmen möglichst geschickt ausgenutzt werden. Beim demand paging haben wir keine Wahl, welche Seite wir bei einem Seitenfehler einlagern. Wir haben aber die Wahl, welche Seite wir auslagern, d. h. hier besteht ein Gestaltungsspielraum. Eine **Seitenauslagerungsstrategie** legt fest, welche Seite eines Prozesses ausgelagert wird. Anlass für eine Seitenauslagerung kann nicht nur ein Seitenfehler sein, sondern auch der Entzug von Seitenrahmen. In beiden Fällen sind die gleichen Auslagerungsstrategien anwendbar.

Die optimale Seitenauslagerungsstrategie. Wir wollen uns zunächst mit der Seitenauslagerung befassen. Nehmen wir also an, dass einem Prozess eine bestimmte Zahl von Seitenrahmen zugewiesen ist, dass alle Seitenrahmen belegt sind und dass ein Seitenfehler eintritt. Wenn wir nun eine Seite auslagern, auf die nie wieder zugegriffen wird, haben wir offenbar eine günstige Wahl getroffen. Wenn wir eine Seite wählen, die gleich als nächste wieder benötigt wird, haben wir offenbar eine schlechte Wahl getroffen. Man kann nun zeigen [Be66, CoD73] und es ist auch intuitiv einleuchtend, dass die **optimale Seitenauslagerungsstrategie** darin besteht, die Seite auszulagern, die erst am weitesten in der Zukunft wieder benutzt werden wird; siehe Abbildung 4.19. Alle Seiten, die nie wieder benutzt werden, haben dabei den Abstand unendlich. Leider ist diese Strategie nicht implementierbar, denn der Rechner kann nicht hellsehen. Die Entscheidung, welche Seite auszulagern ist, kann nur auf Basis des derzeitigen Zustands des Systems und dem bisherigen Verhalten des Prozesses erfolgen. Die optimale Strategie ist aber insofern nützlich, als sie zeigt, dass es unser Ziel sein muss, das zukünftige Zugriffsverhalten des Prozesses möglichst gut abzuschätzen. Außerdem ist sie als Vergleichsmaßstab für andere Strategien nützlich.

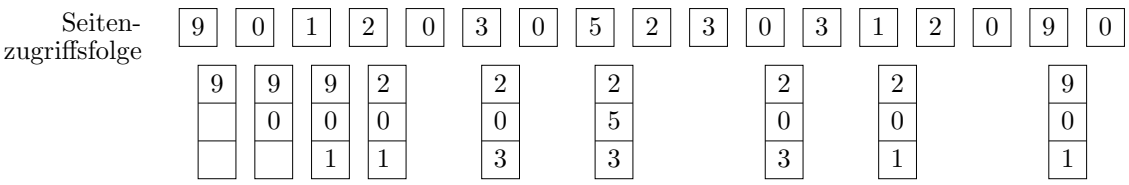


Abbildung 4.19: Die optimale Seitenauslagerungsstrategie mit drei zugeteilten Seitenrahmen.

LRU (least recently used). Geht man von der schon früher erwähnten Lokalitätsannahme aus, dann sollten eigentlich Seiten, auf die in der jüngsten Vergangenheit oft zugegriffen worden ist, auch in der näheren Zukunft oft benutzt werden.

Umgekehrt ist bei Seiten, die schon lange nicht mehr benutzt wurden, eine baldige erneute Benutzung weniger wahrscheinlich; dies legt es nahe, solche Seiten bevorzugt auszulagern. Bei der **LRU-Strategie** wird daher die Seite ausgelagert, die am längsten von allen eingelagerten Seiten nicht mehr benutzt wurde; siehe Abbildung 4.20.

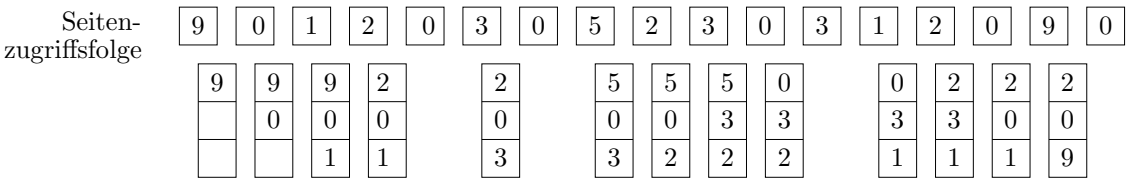


Abbildung 4.20: Die LRU-Strategie mit drei zugeteilten Seitenrahmen.

LRU ist eine sehr gute Strategie. Das Hauptproblem bei LRU ist die effiziente Implementierung, insb. die laufende Generierung von Daten, mit deren Hilfe man erkennen kann, welches die am längsten unbenutzte Seite ist. Wie schon erwähnt wurde, darf der Gesamtaufwand für die Umsetzung einer virtuellen in eine physische Adresse höchstens einen Bruchteil der Dauer eines Speicherzugriffs ausmachen, worunter natürlich auch die Sammlung von Informationen zur Realisierung einer Seitenauslagerungsstrategie fällt. Hinzu kommt, dass die Zustandsinformation beim swapping nicht verlorengehen darf, also das swapping komplizierter wird. LRU ist daher nicht ohne eine relativ aufwändige Hardware-Unterstützung implementierbar. Man kann z. B. Uhren mit einer sehr hohen Auflösung benutzen und bei jedem Zugriff zu einer Seite eine Zeitmarke für diese Seite neu setzen. Problematisch ist dann die Suche nach der kleinsten Zeitmarke und die Behandlung von Überläufen der Uhr. Man kann LRU auch mit Hilfe einer doppelt verketteten Liste, die alle eingelagerten Seiten repräsentiert, implementieren. Die Liste ist nach dem Zeitpunkt des letzten Zugriffs zu jeder Seite sortiert. Nach einem Zugriff rückt der betroffene Eintrag der Liste immer an die Spitze. Hierzu wird der Eintrag ggf. von hinten nach vorne verschoben oder ganz neu eingefügt. Bei einer Auslagerung wird der letzte Eintrag aus der Liste entfernt. Diese Algorithmen müssen in der Hardware realisiert werden.

Zugriffsbits. Wegen des erheblichen Aufwands werden oft effizientere Algorithmen verwendet, die LRU nicht exakt, aber doch annähernd implementieren. Ein Ansatz ist die Überlegung, dass es nicht auf den exakten Zeitpunkt des letzten Zugriffs zu einer Seite ankommt, sondern dass man sich nur für größere Beobachtungsperioden (z. B. 1000 Zugriffe) merkt, ob währenddessen zu der Seite wenigstens einmal zugegriffen wurde. Innerhalb dieser Intervalle ist dann die Reihenfolge der Zugriffe nicht mehr unterscheidbar.

Die Implementierung dieser Idee erfordert ein zusätzliches **Zugriffsbit** (**Referenced-Bit**, R-Bit) pro Seitenrahmen, das bei *jedem* Zugriff zu dieser Seite auf 1 gesetzt wird. Zum Vergleich: das dirty bit wird nur bei *schreibenden* Zugriffen gesetzt; siehe Abbildung 4.21. Zu Beginn einer Beobachtungsperiode werden alle Zugriffsbits auf 0 gesetzt. Am Ende einer Beobachtungsperiode werden die Zugriffsbits gesichert, am besten in einem Schieberegister pro Seite. Bei einem z. B. 8 Bit langen Schieberegister kann man dann die letzten 8 Beobachtungsperioden überblicken. Abbildung 4.22 zeigt ein Beispiel, in dem nach einer Beobachtungsperiode die Zugriffsbits der Seiten 1 bis 6 die Werte 1, 0, 1, 0, 1 und 1 haben,

caching	R-Bit	dirty bit	protect	present	Seitenrahmennummer

Das R-Bit wird bei jedem Zugriff auf 1 gesetzt.
Das dirty bit wird bei jeder Schreib-Operation auf 1 gesetzt. Bei Auslagerung muss der Seitenrahmen zurück auf die Festplatte geschrieben werden, wenn das dirty bit 1 ist.
Die protect-Bits regeln den Zugriff auf die Seite.
Das present-Bit zeigt an, ob die Seite momentan im Hauptspeicher liegt.

Abbildung 4.21: Ein typischer Seitentabelleneintrag.

d. h. in der Periode wurden Seiten 1, 3, 5 und 6 zugegriffen. Nachdem die sechs Schieberegister der Seiten um ein Bit nach rechts verschoben und die Zugriffsbits von links eingefügt wurden, ergeben sich die neuen Anzahlen des Zugriffs auf die Seiten in den letzten 8 Perioden.

Zugriffsbits für die Seiten 1 bis 6 am Ende einer Periode

1	0	1	0	1	1

Seite	Inhalte der Schieberegister am Ende einer Periode	Seite	Inhalte der Schieberegister nach der Aktualisierung
1	0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 0
2	0 0 1 0 1 1 0 0	2	0 0 0 1 0 1 1 0
3	1 1 0 0 0 0 0 0	3	1 1 1 0 0 0 0 0
4	0 1 0 1 1 0 1 0	4	0 0 1 0 1 1 0 1
5	1 0 0 0 1 1 1 1	5	1 1 0 0 0 1 1 1
6	1 1 1 1 1 1 1 1	6	1 1 1 1 1 1 1 1

Abbildung 4.22: Am Ende einer Periode werden die Schieberegister aktualisiert, die Werte zeigen die Zugriffshäufigkeit der Seiten.

FIFO (first-in-first-out). Eine relativ leicht implementierbare Strategie ist **FIFO**. Hier wird die Seite ersetzt, die sich am längsten im Speicher befindet. Man kann eine FIFO-Schlange von alle Seiten im Hauptspeicher verwalten. Am Anfang der Schlange steht die älteste Seite und am Ende die Seite, die zuletzt eingelagert wurde. FIFO führt allerdings zu einer deutlich höheren Seitenfehlerrate als LRU. FIFO hat die absonderliche Eigenschaft, dass u. U. bei mehr verfügbaren Seitenrahmen mehr Seitenfehler auftreten.

Besonders nachteilig ist, dass FIFO laufend intensiv benutzte Seiten auslagert. Diese Fehlentscheidung führt dann natürlich nach sehr kurzer Zeit zu einem erneuten Seitenfehler, durch den die Seite gleich wieder eingelagert wird.

Second Chance. Man erweitert deshalb die FIFO-Strategie oft dahingehend, dass Seiten nicht sofort ausgelagert werden, sondern noch eine Weile im Speicher bleiben und ihnen eine *letzte Chance* gegeben wird. Eines von mehreren solchen Verfahren arbeitet mit einem Zugriffsbit pro Seite: wenn das Zugriffsbit der gemäß FIFO auszulagernden Seite gesetzt (R-Bit = 1) ist, wird es zurückgesetzt (R-Bit = 0) und kommt die Seite ans Ende der FIFO-Schlange und die nunmehr erste Seite in der Schlange ist das neue Opfer. Wenn die Zugriffsbits aller Seiten gesetzt sind, wird die ursprünglich als Opfer vorgesehene Seite dann schließlich doch ausgelagert. Ein anderes Verfahren basiert auf der Idee, bei einem Seitenfehler zunächst die fehlende Seite einzulagern und erst danach die auszulagernde Seite asynchron auf den Swap-Bereich zu schreiben. Hierzu werden einige Seitenrahmen als Ausgabepuffer behandelt; die in diesem Puffer stehenden Seiten sind schon oder werden gerade auf den Swap-Bereich geschrieben; das dirty bit dieser Seiten ist dann zurückgesetzt. Bei einem späteren Seitenfehler können diese Seitenrahmen sofort für eine Einlagerung benutzt werden. Falls nun bei FIFO eine gerade intensiv benutzte Seite zur Auslagerung bestimmt wurde, kann sich diese Seite noch im Ausgabepuffer befinden; in diesem Fall wird eine erneute Einlagerung vermieden.

Der Clock-Algorithmus. Die Second-Chance-Strategie ist eine modifizierte Version von FIFO und verbessert die Leistung enorm, aber sie ist ineffizient, weil sie ständig Seiten in der Liste verschiebt. Der Clock-Algorithmus ist eine andere Implementierung von Second Chance, wobei die Schlange durch eine zirkuläre Liste ersetzt wird und ein Uhrzeiger auf eine Seite zeigt. Jede Seite in der Liste wird mit einem Zugriffsbit versehen. Wenn eine Seite in einen Rahmen eingelagert wird, wird das Zugriffsbit auf 0 gesetzt. Wenn die Seite später benutzt wird, wird das Bit auf 1 gesetzt. Wenn eine Seite ausgelagert werden soll, rückt der Uhrzeiger vor, um einen Kandidaten zu finden. Wenn er auf eine Seite stößt, deren Zugriffsbit auf 1 steht, wird es auf 0 zurückgesetzt, und der Zeiger rückt weiter vor. Dieser Vorgang wird so lange wiederholt, bis eine Seite mit R-Bit = 0 gefunden wird; siehe Abbildung 4.23.

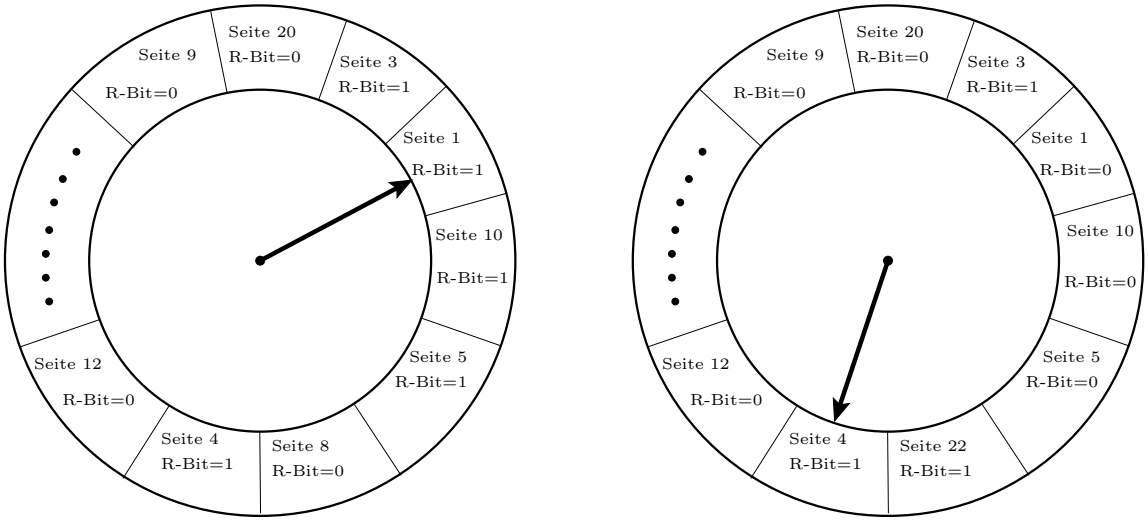
Zusammenfassend kann man sagen, dass Auswahl und Implementierung einer Seitenauslagerungsstrategie eine sehr anspruchsvolle Aufgabe ist und stark von der verfügbaren Hardware-Unterstützung, aber auch von der Art des zu realisierenden Betriebssystems abhängen. Schließlich bestehen noch Zusammenhänge mit dem Scheduling und der Speicherzuweisung; diese Themen behandeln wir jetzt anschließend.

4.6.3 Zuweisungsstrategien

Wir kommen jetzt auf die oben offengelassene Frage zurück, wieviele Seitenrahmen einem Prozess zugewiesen werden sollten. Wir betrachten hierzu zunächst noch einmal das typische Zugriffsverhalten von Prozessen und ziehen daraus dann einige Konsequenzen.

4.6.3.1 Die Arbeitsmengenstrategie

Lokalitäten. Prozesse greifen nicht völlig zufällig auf einzelne Seiten zu, sondern haben meist - zumindest für kurze Zeiträume - Schwerpunkte. Dies hat mehrere



Zustand vor der Auslagerung

Zustand nach der Auslagerung

Abbildung 4.23: Seite 22 soll eingelagert werden. Der Clock-Algorithmus wählt Seite 8 zur Auslagerung aus.

Ursachen: Programme enthalten oft Schleifen. Ein Beispiel ist eine Anweisung, die die Elemente eines Arrays A in einer Variablen B aufsummiert:

```
for i:=1 to 100 do B:=B+A[i]
```

Die Seiten, die den in der Schleife durchlaufenen Code enthalten, werden dann eine Zeitlang sehr häufig benutzt. Komplexe Datenstrukturen werden üblicherweise kompakt gespeichert; die Elemente eines Records oder Arrays werden direkt hintereinander im Speicher angeordnet. Werden nun alle Elemente nacheinander verarbeitet (wie im obigen Beispiel), dann treten während dieser Zeit sehr viele Zugriffe zu den (wenigen) Seiten auf, die diese Datenstrukturen enthalten.

Eine Menge von Seiten, die über einen Zeitraum wesentlich häufiger als andere Seiten benutzt werden, nennt man eine **Lokalität** des Prozesses. Man kann Lokalitäten sehr gut graphisch veranschaulichen (siehe [SiGG00], p. 327), indem man in einer Matrix Zugriffe zu einzelnen Adressen oder die Werte der oben erwähnten Zugriffsbits über die Zeit hinweg aufträgt, wobei ein Zugriff durch einen schwarzen Punkt dargestellt wird. Bei den meisten beobachteten Prozessen treten große weiße Bereiche auf und relativ klar umrissene graue bis schwarze Bereiche. Ein Programm verweilt aus dieser globalen Sichtweise immer abwechselnd in verschiedenen Lokalitäten.

Die Arbeitsmengenstrategie. Ganz offensichtlich sollten alle Seiten der Lokalität, in der sich ein Prozess gerade befindet, geladen sein. Die Frage ist jetzt natürlich, wie man denn im laufenden System erkennt, welche Seiten zur aktuellen Lokalität gehören. Die übliche Antwort ist recht einfach: es sind alle Seiten, die *in letzter Zeit* benutzt worden sind. Diese Menge bezeichnet man als die **Arbeitsmenge** (**working set**) des Prozesses.

Die **Arbeitsmengenstrategie** besteht nun darin, einem Prozess zu einem bestimmten Zeitpunkt so viele Seitenrahmen zuzuweisen, wie dessen aktuelle Arbeits-

menge groß ist. Die Arbeitsmengenstrategie ist eine Zuweisungsstrategie:

- Bei jedem Seitenfehler wird einem Prozess ein weiterer Seitenrahmen zugewiesen[6].
- Wenn eine Seite aus der Arbeitsmenge herausfällt, also zu lange nicht mehr benutzt wurde, wird dem Prozess ein Seitenrahmen entzogen. Welche Seite nun ausgelagert wird, kann im Prinzip von einer beliebigen Seitenauslagerungsstrategie bestimmt werden; sinnvoll ist aber im Prinzip nur LRU, in der Praxis können sich Abweichungen ergeben, weil LRU nur approximiert wird; wir gehen i. F. von LRU aus.

Es ist recht instruktiv, die Arbeitsmengenstrategie und die Strategie, einem Prozess eine *feste* Menge von Seitenrahmen zuzuweisen, was wir oben bei der Diskussion der Ersetzungsstrategien angenommen hatten, bzgl. der Ursachen für Seitenauslagerungen zu vergleichen:

- Bei einer festen Zahl von zugeteilten Seitenrahmen ist die Ursache einer Seitenauslagerung immer eine Seiteneinlagerung, die ihrerseits durch einen *Seitenfehler* verursacht wurde.
- Bei der Arbeitsmengenstrategie ist die Ursache einer Seitenauslagerung immer das *Herausfallen einer Seite aus der Arbeitsmenge*[7]. Es können also Seiten eines Prozesses ausgelagert werden, ohne dass dieser Prozess einen Seitenfehler verursacht hätte und ohne dass für diesen Prozess eine neue Seite eingelagert wird.

Die Arbeitsmenge approximiert die aktuelle Lokalität lediglich; Abweichungen ergeben sich aus folgenden prinzipiellen Ursachen:

- Die Arbeitsmenge enthält auch Seiten, die in letzter Zeit nur ein einziges Mal benutzt worden sind und die eigentlich ohne größeres Risiko wieder ausgelagert werden können.
- Beim Übergang von einer Lokalität zu einer anderen enthält die Arbeitsmenge noch eine Zeitlang Seiten aus der alten Lokalität, die erst nach und nach aus der Arbeitsmenge herausfallen. Umgekehrt ist die neue Lokalität noch nicht vollständig aufgebaut.

Die letzte Zeit bezeichnet man auch als *das* **Fenster**, sozusagen der Ausschnitt aus der gesamten Geschichte des Prozesses, der für die Bestimmung der Arbeitsmenge betrachtet wird. Wählt man das Fenster zu groß, wird die Arbeitsmenge unnötig groß. Wählt man es zu klein, wird u. U. nicht die vollständige Arbeitsmenge erfasst. Nach [MaD74] sollte man die Fenstergröße in der Größenordnung von 10000 Speicherzugriffen wählen.

[6] Offensichtlich kann man diese Strategie nicht beliebig lange durchhalten, weil z. B. die Zahl der Seitenrahmen begrenzt ist oder weil diesem Prozess im Vergleich zu anderen Prozessen zu viele Seitenrahmen zugewiesen werden würden (s. u. Bemerkungen zum Scheduling). Man muss also die Arbeitsmengenstrategie in dieser Hinsicht modifizieren und mit anderen Strategien kombinieren, was wir hier aber nicht detailliert behandeln werden.

[7] Tatsächlich wird die Seitenauslagerung i. A. nicht unmittelbar in dem Moment, wo eine Seite aus der Arbeitsmenge fällt, stattfinden, sondern erst beim nächsten Prozesswechsel, Seitenfehler eines anderen Prozesses o. ä., also wenn der frei werdende Seitenrahmen tatsächlich gebraucht wird.

Realisierungsprobleme. Probleme bereitet nur das Entziehen von Seitenrahmen. Theoretisch muss bei *jedem Speicherzugriff* geprüft werden, ob in diesem Moment irgendeine Seite *hinten* aus dem Fenster herausfällt, also in den letzten Δ Zugriffen nicht mehr benutzt wurde, wenn Δ die Fenstergröße ist. Mit vertretbarem Aufwand ist eine exakte Überprüfung nicht möglich; hier gelten i. W. die gleichen Überlegungen wie bei der Realisierung von LRU. Aus Effizienzgründen kann man daher die Arbeitsmenge eines Prozesses nur approximativ bestimmen, z. B. mit Hilfe von Zugriffsbits.

Beeinflussung der Arbeitsmenge durch die Programmstruktur. Ein Entwickler von Anwenderprogrammen braucht normalerweise keine Rücksicht darauf zu nehmen, wie ein virtueller Speicher realisiert ist; man kann ihn wie einen physischen Speicher behandeln. Man kann aber relativ häufig zwischen funktional äquivalenten Programmstrukturen wählen, die zu völlig verschiedenen Arbeitsmengen führen, z. B. indem Datenstrukturen mehr oder weniger verstreut angelegt werden.

4.6.3.2 Kontrolle der Seitenfehlerrate

Beim Arbeitsmengen-Modell werden einem Prozess Seitenrahmen entsprechend der Größe der Arbeitsmenge zugeteilt; hieraus schließen wir, dass das eigentliche Ziel, eine geringe Seitenfehlerrate, erreicht werden wird. Man kann aber auch bei dem eigentlichen Ziel ansetzen und die Seitenfehlerrate messen und dann, wenn sie zu hoch ist, dem Prozess mehr Seiten zuteilen. Wenn sie umgekehrt sehr niedrig ist, vor allem im Vergleich zu parallel laufenden Prozessen, kann man daraus schließen, dass dem Prozess wohl zu viele Seitenrahmen zugeteilt worden sind und ihm Seitenrahmen entziehen. Dies nennt man **Seitenfehlerhäufigkeitsstrategie** (bzw. -verfahren).

Ein Verfahren zur Realisierung der Seitenfehlerhäufigkeitstrategie ist der **Seitenfehler-Frequenz-Algorithmus**(**Page Fault Frequency**). Er benutzt das Zugriffsbit einer Seite. Wenn ein Seitenfehler auftritt, registriert das Betriebssystem die Zeit t, die seit dem letzten von diesem Prozess verursachten Seitenfehler vergangen ist. Man legt einen Grenzwert T fest. Wenn $t \leq T$ ist, wird dem Prozess noch eine Seite mehr zugeteilt. Wenn $t > T$ ist, werden alle Seiten mit R-Bit = 0 ausgelagert und zugleich wird das Zugriffsbit der verbliebenen Seiten auf 0 gesetzt.

Dieses Verfahren funktioniert vor allem in den Übergangsphasen bei Lokalitätswechseln nicht gut. Die rasche Aufeinanderfolge von Seitenfehlern während des Wechsels von einer Lokalität zu einer neuen führt dazu, dass die Anzahl der zugeteilten Seitenrahmen des Prozesses stark anwächst, bevor die Seiten der alten Lokalität entfernt werden. Diese plötzliche Häufung von Speicheranfragen kann die erneute Aus- und Einlagerung des Prozesses mit den entsprechenden unerwünschten Overheads verursachen.

4.6.4 Scheduling

Wir haben das Problem der Speicherplatzzuweisung bisher aus Sicht eines einzelnen Prozesses behandelt; wichtigstes Ergebnis war, dass einem Prozess wenigstens so viele Seitenrahmen zugewiesen werden müssen, wie die aktuelle Lokalität Seiten enthält, und dass man die Lokalität hinreichend genau durch die Arbeitsmenge approximieren kann. Weist man weniger Seitenrahmen zu, steigt die Seitenfehlerrate

zu hoch an; dies gilt *unabhängig von der Seitenauslagerungsstrategie*. Da diese die zugeteilten Seitenrahmen i. A. nicht optimal ausnutzen, muss also eher noch eine gewisse Reserve zugeteilt werden. Wenig sinnvoll ist es, einem Prozess eine feste Menge von Seitenrahmen zuzuteilen.

Für die globale Steuerung der Menge der ausgeführten Prozesse folgt daraus, dass man nur so viele Prozesse parallel laufen lassen darf, dass die Summe der Größen ihrer Arbeitsmengen kleiner als der verfügbare physische Hauptspeicher bleibt. Wie wir ebenfalls gesehen haben, können die einzelnen Arbeitsmengen und damit die erwähnte Summe erheblich schwanken, speziell kann die Summe größer werden als der physische Speicher. In diesem Fall muss die Zahl der aktiven Prozesse *verringert* werden, indem einige Prozesse, z. B. solche mit geringer Priorität, für eine Weile völlig stillgelegt und sämtliche von ihnen belegten Seitenrahmen freigegeben werden. Andernfalls kommt es zu einer **Systemüberlastung** (**thrashing**, auch **Seitenflattern** genannt), das System produziert dann sozusagen nur noch innere Reibung, nämlich Seitenfehler. In einem solchen Zustand ist die CPU-Auslastung sehr gering, denn die CPU muss ja ständig auf einzulagernde Seiten warten. In einigen frühen Systemen mit demand paging reagierten die Scheduler dann völlig falsch: um die CPU-Auslastung zu erhöhen, erhöhten sie die Zahl der arbeitenden Prozesse noch mehr.

Interaktive Prozesse können natürlich nicht oder nur unter großem Protest der betroffenen Benutzer stillgelegt werden.

Eine weitere Frage ist, ob nach einem Prozesswechsel die Seiten des bisher aktiven Prozesses durch den nunmehr aktiven Prozess aus dem Speicher verdrängt werden sollen. Gemäß den Annahmen des Arbeitsmengen-Modells wäre es jedenfalls sinnlos, den passivierten Prozess weiter auszuführen, wenn nicht vorher alle Seiten der Arbeitsmenge wieder geladen worden sind. Wenn der Prozess also auf ein kurzfristig eintretendes Ereignis wartet, sollten seine Seiten nicht verdrängt werden. Bei längerfristigem Warten z. B. bei einer Denkpause eines interaktiven Benutzers können die Seiten hingegen verdrängt werden. Vor Fortsetzung des Prozesses müssen alle Seiten der Arbeitsmenge des Prozesses wieder eingelagert werden; dies ist üblicherweise der einzige Fall von prepaging.

4.6.5 Benutzergesteuerte Speicherverwaltung

Wir hatten oben schon erwähnt, dass Anwendungsprogrammierer durch eine mehr oder weniger geschickte Wahl der Programm- oder Datenstrukturen die Größe der Arbeitsmenge und damit ggf. die Seitenfehlerrate beeinflussen können. Man kann noch einen Schritt weiter gehen und dem Benutzer Möglichkeiten geben, dem Betriebssystem gezielt Hinweise auf das Zugriffsverhalten des Programms zu geben, so dass die Speicherzuweisung und Seitenersetzung daraufhin optimiert werden kann. Beispiele hierfür sind:

- Wenn eine größere Datei mit memory-mapped IO in den virtuellen Speicher eingeblendet wird und dann i. W. nur einmal sequentiell durchlaufen werden soll, können diese Seiten sehr schnell wieder ausgelagert werden, am besten schon beim nächsten Seitenfehler. Diese Strategie wäre normalerweise also bei lokalem Zugriffsverhalten sehr schlecht.

- Wichtige Datenstrukturen, z. B. Puffer eines Datenbanksystems, werden u. U. überdurchschnittlich häufig benutzt; die entsprechenden Seiten sollten daher bevorzugt im Hauptspeicher gehalten werden.

Derartige *Beratungsfunktionen* werden erst seit kurzem in Ansätzen in kommerziellen Betriebssystemen eingeführt.

4.7 Zusammenfassung

Wir haben uns in dieser Kurseinheit mit der zentralen Frage befasst, wie der vorhandene physische Hauptspeicher eines Rechners zu verwalten ist. Wesentlichstes Ziel dabei war, für parallele Prozesse mehrere logische Hauptspeicher zu simulieren. Wir haben eine ganze Reihe von Speicherzuweisungsverfahren kennengelernt, die sich hinsichtlich ihrer Leistungsfähigkeit wesentlich unterscheiden; aus Performance-Gründen müssen allerdings bei den leistungsfähigeren Verfahren zentrale Funktionen in der Hardware realisiert sein.

Die Aufteilung und Verwaltung des physischen Hauptspeichers muss gemeinsam mit dem Speicherschutz gesehen werden; die konkreten Mechanismen behandeln fast immer beide Problembereiche gemeinsam. Tendenziell bieten daher leistungsfähigere Verwaltungsverfahren auch besseren Schutz.

Bei der Diskussion dieses Themas muss strikt getrennt werden zwischen den verschiedenen Formen, in denen ein Programm auftritt (Modul in einer höheren Programmiersprache bzw. Assembler, Bindemodul, Lademodul, geladenes Programm), den darin auftretenden Namen bzw. Adressen von Objekten und der unterschiedlichen Interpretation von Adressen im System- bzw. Benutzermodus (physische bzw. reale und logische Adressen). In diesem Zusammenhang spielt es außerdem eine Rolle, ob Programme statisch oder dynamisch gebunden werden, ob sie verschiebbar und ob sie gemeinsam benutzbar sind.

Man kann die Speicherzuweisungsverfahren grob danach klassifizieren, ob sie einem Prozess einen einzigen zusammenhängenden Abschnitt des physischen Hauptspeichers zuweisen oder mehrere solche Abschnitte; bei der zweiten Gruppe von Verfahren muss, da der logische Hauptspeicher fast immer ein geschlossenes Intervall von Adressen hat, ein mehr oder weniger komplizierter Hardware-Mechanismus vorhanden sein, der logische in physische Adressen umsetzt.

Bei der zusammenhängenden Speicherzuweisung wird i. A. der Rest des physischen Hauptspeichers, der nicht für den Betriebssystemkern benötigt wird, in ein oder mehrere Segmente aufgeteilt, die jeweils als Ganze einzelnen Prozessen zugewiesen werden. Man unterscheidet hier weiter Verfahren, die den physischen Hauptspeicher in feste bzw. variabel große Segmente aufteilen (MFT bzw. MVT). MVT ist wesentlich flexibler; als wichtigstes Detailproblem tritt hier die Verwaltung der variabel großen Segmente auf. Hierzu gibt es mehrere Verfahren (u. a. first fit, best fit, buddy).

Die Verfahren, die den physischen Hauptspeicher nichtzusammenhängend zuweisen, können wiederum in zwei Gruppen klassifiziert werden: bei der ersten Gruppe werden nur wenige, variabel große Segmente zugewiesen, die der Applikation in Form von logischen Segmenten mehr oder weniger sichtbar sind. Bei der zweiten Gruppe wird der physische Hauptspeicher für Applikationen praktisch unsichtbar in viele

Stücke gleicher Größe aufgeteilt, nämlich Seitenrahmen, der logische Hauptspeicher analog in Seiten.

Die seitenorientierte Hauptspeicherverwaltung ist die Basis für virtuelle Hauptspeicher: bei diesen ist der logische Hauptspeicher eines Prozesses nur teilweise im physischen Hauptspeicher realisiert. Bei einem Zugriff zu einer nicht realisierten Seite tritt ein Seitenfehler ein und die fehlende Seite wird eingelagert (demand paging). Seitenfehler verursachen erheblichen Aufwand und dürfen nicht zu häufig werden. Aus diesem Grund müssen einem Prozess zumindest so viele Seitenrahmen zugewiesen werden, dass alle Seiten der Lokalität, in der sich der Prozess befindet, realisiert sind. Diese Menge wird mit Hilfe der Arbeitsmenge approximiert, die wiederum mit Zugriffsbits annähernd bestimmt wird. Alternativ kann man die Seitenfehlerrate eines Prozesses überwachen und anhand dieser die Zahl der zugewiesenen Seitenrahmen bestimmen. Insgesamt dürfen nur so viele Prozesse parallel ausgeführt werden (also in den Zustand *bereit* überführt werden und Zeitscheiben zugeteilt bekommen), dass die Seitenfehlerrate ingesamt gering bleibt. Bei der Auslagerung von Seiten wählt man am besten die LRU-Strategie; diese lässt sich jedoch aus Aufwandsgründen nur approximativ z. B. mit Hilfe von Zugriffsbits realisieren.

Literatur

[Be66] L.A. Belady. *A study of replacement algorithms for virtual storage computers.* IBM J. Research and Development, 1966.

[CoD73] E.G. Coffman, and P.J. Denning. *Operating systems theory.* Prentice Hall, 1973.

[MaD74] S.E. Madnick, and J.J. Donovan. *Operating systems.* McGraw–Hill, 1974.

[Or72] E.I. Organick. *The Multics system: an examination of its structure.* MIT Press, Cambridge MA, 1972.

[SiGG00] A. Silberschatz, P.B. Galvin, and G. Gagne. *Applied operating system concepts, First Edition.* John Wiley & Sons, Inc., 2000.

Glossar

absolute Adresse (*absolute address*) in einem Lademodul auftretende Adresse, die bereits identisch mit der logischen Adresse nach dem Laden ist.

Adresse (*address*) Identifizierer einer Speicherzelle eines (physischen oder logischen) Hauptspeichers; tritt als direkte Adresse auch in Adressoperanden von Maschinenbefehlen auf.

Adressoperand Parameter eines Maschinenbefehls, dessen Auswertung eine Adresse ergibt; die Adresse kann direkt angegeben sein oder z. B. mit Hilfe des Inhalts von Registern berechnet werden.

Adressraum (*address space*) Menge syntaktisch zulässiger Adressen (auch wenn zu einzelnen Adressen keine Speicherzelle vorhanden ist); i. W. gegeben durch die Länge der Adressfelder in Maschinenbefehlen.

Arbeitsmenge (*working set*) Menge der Seiten des virtuellen Speichers eines Prozesses, zu denen „in letzter Zeit“ zugegriffen worden ist; approximiert die Menge der „aktuell benötigten“ Seiten.

Arbeitsspeicher s. Hauptspeicher.

Basisregister (*base register*) Register, das die Basisadresse eines Speicherabschnitts enthält. Im Kontext der zusammenhängenden Speichervergabe: wird bei der Umsetzung der logischen Adressen in reale Adressen benutzt.

best available fit Strategie bei der Zuteilung von Speicherabschnitten bei MFT; zugewiesen wird das kleinste verfügbare ausreichend große Segment.

best fit only Strategie bei der Zuteilung von Speicherabschnitten bei MFT; zugewiesen werden nur Segmente, deren Größe nicht zu stark über dem angeforderten Speicherplatz liegt.

Bindemodul (*object module*) Ergebnis der Übersetzung eines Quellprogramm-Moduls.

Binder (*binder, linkage editor*) Dienstprogramm, welches mehrere Bindemoduln zu einem Binde- oder Lademodul vereinigt und dabei offene modulübergreifende Referenzen auflöst.

Bindelader (*linkage editor and loader*) kombinierter Binder und Lader, der üblicherweise kein Lademodul erzeugt, sondern direkt ein geladenes Programm.

demand paging Verfahren, bei dem eine Seite erst eingelagert wird, wenn ein Zugriff auf diese Seite einen Seitenfehler verursacht.

direkte Adresse (*direct address*) in einem Maschinenprogramm direkt gespeicherte Adresse (z. B. als Operand eines Maschinenbefehls).

dirty bit Bit, welches anzeigt, ob auf einer Seite in einem bestimmten Beobachtungszeitraum geschrieben wurde.

dynamischer Bindelader inkrementell arbeitender Bindelader, der zu einem laufenden Programm dynamisch Moduln hinzulädt und bindet.

externe Fragmentierung (*external fragmentation*) Vorhandensein kleiner, nicht zugewiesener und nicht mehr zuweisbarer Speicherabschnitte.

Fragmentierung (*fragmentation*) Vorhandensein von unbenutzten kleinen Stücken eines Speichers.

gemeinsam benutzte Bindemoduln (*shared library*) Bindemoduln, welche Teil der dynamisch gebundenen Programme mehrerer paralleler Prozesse sind; erfordern dynamischen Bindelader, der offene Referenzen aus Bibliotheken gemeinsam benutzter Bindemoduln befriedigt.

Grenzregister (*fence register*) Register, das eine Grenze des einem Prozess zugewiesenen Segments enthält und für Speicherschutzmaßnahmen benutzt wird.

Hauptspeicher (*main memory*) Menge von einzeln adressierbaren Speicherzellen (Bytes oder Worte), die die CPU als Ganze direkt lesen oder schreiben kann.

interne Fragmentierung (*internal fragmentation*) Zuweisung eines Speicherabschnitts, der größer als angefordert ist, wodurch Teile des zugewiesenen Speicherplatzes unbenutzt bleiben.

Kompaktifizierung (*compaction*) Reorganisation eines nach MVT aufgeteilten Speichers mit dem Ziel, kleine Lücken zu großen zusammenzulegen.

Lademodul (*load module*) Maschinenprogramm, das von einem Lader in den Hauptspeicher eines Prozesses geladen werden kann.

logische Adresse (*logical address*) im Benutzermodus interpretierte Adresse.

logischer (Haupt- oder Arbeits-) Speicher (*logical main memory*) aus Sicht von Maschinenprogrammen im Benutzermodus vorhandener Hauptspeicher; oft prozessspezifisch.

Lokalität (*locality*) Menge von Seiten, zu denen ein Prozess während eines relativ kurzen Abschnitts seiner Gesamtausführungszeit zugreift.

LRU (least recently used) Seitenauslagerungsstrategie, nach der die am längsten unbenutzte Seite ausgelagert wird.

memory-mapped IO Einblenden von Seiten einer Datei in die Seiten eines virtuellen Hauptspeichers.

MFT Speicherplatzvergabestrategie (multiprogramming with a fixed number of tasks).

MVT Speicherplatzvergabestrategie (multiprogramming with a variable number of tasks).

Objekt-Modul s. Bindemodul.

paging seitenorientierte Umsetzung von logischen (virtuellen) Adressen in reale Adressen.

physische Adresse (*real address, physical address*) im Systemmodus interpretierte Adresse; Adresse einer Speicherzelle (Byte oder Wort) im physischen Speicher.

physischer (Haupt- oder Arbeits-) Speicher (*physical/real main memory*) im Rechner durch Speichermoduln realisierter Speicher.

prepaging Einlagerung von Seiten, ohne dass ein Zugriff zu diesen Seiten (mit Seitenfehler) stattgefunden hätte.

Quellprogramm-Modul (*source module*) selbständig übersetzbarer Teil eines Programms in einer höheren Programmiersprache oder Assembler.

reale Adresse siehe physische Adresse.

realisierte Adresse physische oder logische Adresse, zu der eine Speicherzelle im realen Hauptspeicher vorhanden ist.

relative Adresse (*relative address*) in einem Binde- oder Lademodul auftretende Adresse, die relativ zum Anfang des Binde- oder Lademoduls berechnet ist.

Seite (*page*) Abschnitt eines logischen Hauptspeichers, der in einem Seitenrahmen gespeichert werden kann.

Seitenflattern (*thrashing*) Zustand eines Rechners, in dem dieser fast nur noch Seiten ein- und auslagert und keine produktive Arbeit mehr leistet, weil der reale Hauptspeicher zu klein ist, um die Lokalitäten, in denen sich die bereiten bzw. aktiven Prozesse gerade befinden, aufzunehmen.

Seitenrahmen (*page frame*) Abschnitt des physischen Speichers, in dem eine Seite gespeichert werden kann.

Segment (*segment*) zusammenhängender Abschnitt des Speichers (wenn keine weitere Angabe: des physischen Speichers).

Speicher (*memory*) in dieser Kurseinheit: Hauptspeicher.

shared memory Seiten, die Teil des virtuellen Hauptspeichers mehrerer Prozesse sind.

swapping Aus- bzw. Einlagern vollständiger Prozesse (inkl. Inhalt des zugehörigen Hauptpeichers) auf einen bzw. von einem Sekundärspeicher.

Swap-Bereich Ein reservierter Bereich auf der Festplatte, um die ausgelagerten Seiten aufzunehmen.

verschiebbares Maschinenprogramm (*relocatable / position independent code*) Maschinenprogramm, das beim Laden bzw. Verschieben in einem Speicher nicht verändert werden muss (darf z. B. keine direkten Adressen enthalten).

virtuelle Adresse (*virtual address*) logische Adresse, bei der besonders betont werden soll, dass sie nicht identisch mit der zugehörigen physischen Adresse ist.

virtueller (Haupt-) Speicher (*virtual storage*) Mit Hilfe von demand paging simulierter logischer Hauptspeicher; hat virtuelle Adressen; kann größer sein als der physische Hauptspeicher.

wiedereintrittsfähige Programme (*reentrant code*) Lademoduln, die als gemeinsam benutzter Hauptpeicherbereich in den Hauptspeicher von mehreren Prozessen eingeblendet und dann von allen Prozessen parallel ausgeführt werden können; dürfen z. B. keine Datenbereiche enthalten; sind oft auch verschiebbar.

working set s. Arbeitsmenge.

Zugriffsbit (*referenced bit*) Bit, welches anzeigt, ob eine Seite in einem bestimmten Beobachtungszeitraum zugegriffen wurde.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 5:
Prozesskommunikation

Autoren: Lutz Richter, Udo Kelter, Christian Icking, Lihong Ma

Inhalt

Kurseinheit 5

Prozesskommunikation

Die auf einem Rechnersystem gemeinsam ablaufenden Programme müssen die verschiedenen Betriebsmittel der Hard- und Software miteinander geteilt benutzen, wenn eine effiziente Ausnutzung der verschiedenen Komponenten des Rechnersystems erreicht werden soll. Betrachtet man etwa ein einzelnes Programm, so wird dieses kaum während seiner gesamten Laufzeit eine bestimmte Hardware-Komponente exklusiv benutzen. Beispielsweise kann ein Ein-/Ausgabe-Kanal die Bedienanforderungen eines einzelnen Programms sehr viel schneller abwickeln, als diese Anforderungen von dem betrachteten Programm generiert werden. Es ist daher im Hinblick auf die Auslastung dieses E/A-Kanals zweckmäßig, dass mehrere Programme während eines gewissen Zeitabschnitts diesen Kanal wechselseitig für ihre Ein- bzw. Ausgabe benutzen. Bezüglich der Übertragungskapazität dieses E/A-Kanals wird durch eine solche Aufteilung der Übertragungsvorgänge eine höhere Auslastung erreicht werden.

Die simultane Benutzung von Rechnerkomponenten durch mehrere Programme ist aber nicht auf die Hardware beschränkt. Werden nämlich zeitlich überlappt mehrere Programme aus einer höheren Programmiersprache in die Maschinensprache übersetzt, so ist es zweckmäßig, wenn nicht für jedes der zu übersetzenden Programme eine eigene Kopie des Übersetzungsprogramms in einem privaten Speicherbereich enthalten ist. Vielmehr ist es wirtschaftlicher im Hinblick auf das Betriebsmittel Speicher, wenn nur eine einzige Kopie des Übersetzungsprogramms im Speicher resident gehalten wird und alle den Übersetzungsvorgang simultan ausführenden Programme auf diese eine Kopie des Übersetzungsprogramms zugreifen.[1]

Es ist naheliegend, dass bei der wechselseitigen Benutzung von Hard- und Software-Betriebsmitteln, die über einen hinreichend großen Zeitabschnitt betrachtet simultan erscheint, entsprechende Voraussetzungen erfüllt sein müssen, damit durch eine derartige zeitlich verzahnte Benutzung keine Konflikte hervorgerufen werden. Die Prozesse, die bestimmte Betriebsmittel gleichzeitig benutzen, und die Prozesse, die diese zur simultanen Benutzung vorgesehenen Betriebsmittel verwalten, müssen miteinander *kommunizieren*.

Hard- und Software-Betriebsmittel haben bestimmte gemeinsame Eigenschaften. Eine Datei bspw. kann zu einer Zeit nur die Anforderungen von einem Pro-

[1] Ein Programm, das mit einer einzigen Kopie im Speicher wechselseitig von mehreren anderen Programmen aufgerufen werden kann, muss gewisse Eigenschaften hinsichtlich seines Aufbaus besitzen. Man nennt solche Programme wiedereintrittsinvariant (reentrant).

zess bedienen. Obwohl es sich hierbei um ein Software-Betriebsmittel handelt, erfolgt die gemeinsame Benutzung dieses Betriebsmittels in der gleichen Weise wie bei Hardware-Betriebsmitteln. Man unterscheidet daher **physische Betriebsmittel** (Geräte, Hardware-Komponenten) und **logische Betriebsmittel**. Unter logischen Betriebsmitteln verstehen wir mithin Betriebsmittel, die durch die Software simuliert werden. Aus Sicht der Anwender verhalten sie sich genauso wie physische Betriebsmittel. Um etwa mehreren Prozessen die gemeinsame Benutzung einer einzigen physischen Platte zu ermöglichen, teilt man das physische Laufwerk in mehrere logische Platten auf. Wenn ein Prozess eine solche logische Platte benutzt, so entsteht der Eindruck, als stehe ihm die gesamte Platte zur Verfügung. Bei der Kommunikation eines Prozesses mit dem das Betriebsmittel verwaltenden Prozess wird dabei in der Regel nicht unterschieden, ob es sich um ein physisches oder logisches Betriebsmittel handelt.

5.1 Konkurrente Prozesse

Konkurrente Prozesse beschreiben die Vorgänge mehrerer simultaner oder paralleler Aktivitäten. Beispiele hierfür sind zeitlich überlappt ausgeführte Ein-/Ausgabe-Operationen mit Abläufen von Programmen im Prozessor sowie das gleichzeitige Vorhandensein mehrerer Benutzerprogramme im Hauptspeicher. Die Planung und Steuerung konkurrenter Prozesse führt zu Problemen wie

- die Umschaltung eines Ablaufs auf einen anderen
- die Sicherung einer Aktivität gegen die unerwünschten Effekte einer anderen Aktivität
- die Synchronisierung voneinander abhängiger konkurrenter Prozesse.

Konkurrente Prozesse erfordern die gemeinsame Benutzung von Betriebsmitteln und Informationen. Hierfür gibt es mehrere Gründe:

1. Kosten: es ist unwirtschaftlich, für sämtliche Benutzer eines Rechnersystems separate und exklusive Betriebsmittel vorzusehen;
2. gemeinsame Programmteile: es ist nützlich, wenn verschiedene Anwender Programme anderer Benutzer verwenden können;
3. gemeinsame Daten: vielfach ist es notwendig, dass für unterschiedliche Anwendungen gemeinsame Datenbasen zur Verfügung stehen;
4. Redundanz: eine einzige Kopie eines bestimmten Dienstleistungsprogramms z. B. eines Übersetzers oder Datenübertragungsprogramms sollte vorzugsweise mehreren Anwendern gleichzeitig zugänglich sein (Wiedereintrittsinvarianz).

Ein Betriebssystem muss **determiniert** sein in dem Sinne, dass unter der Kontrolle dieses Betriebssystems ablaufende Programme jeweils identische Resultate haben sollten, unabhängig davon, in welcher Umgebung, d. h. zusammen mit welchen anderen Programmen, diese Programme ablaufen. Andererseits sind die tatsächlichen Vorgänge **nicht-determiniert** insofern, als es möglich sein muss, auf Ereignisse zu reagieren unabhängig von der Reihenfolge ihres Auftretens. Solche Ereignisse

sind z. B. Betriebsmittelanforderungen, Laufzeit-Fehler in den Anwenderprogrammen und asynchrone Unterbrechungen, die von E/A-Geräten erzeugt werden. Ein Betriebssystem muss also in der Lage sein, eine beliebige Folge solcher Ereignisse determiniert hinsichtlich ihrer Konsequenzen zu behandeln.

5.1.1 Disjunkte und überlappende Prozesse

Wenn konkurrente Prozesse keine gemeinsamen Daten verändern, so sind sie **disjunkt** oder voneinander unabhängig. Disjunkte Prozesse können jedoch auch auf gewisse Daten gemeinsam zugreifen, Voraussetzung ist jedoch, dass diese Daten während der gesamten Laufzeit dieser konkurrenten Prozesse nicht verändert werden.

Betrachten wir in Abbildung 5.1 zwei konkurrente Prozesse.

Prozess 1	Prozess 2
$X_1 := \max(a_1, b_1);$	$X_2 := \max(a_2, b_2);$
$Y_1 := \max(c_1, d_1);$	$Y_2 := \max(c_2, d_2);$
$Z_1 := X_1 + Y_1;$	$Z_2 := X_2 - Y_2;$

Abbildung 5.1: Zwei disjunkte Prozesse ohne gemeinsam benutzte Daten.

Sie können beliebig nebeneinander ablaufen, da sie nur disjunkte Mengen von Objekten verarbeiten.

Die in Abbildung 5.2 dargestellten konkurrenten Prozesse werden aber noch immer disjunkt sein, obwohl sie gemeinsam benutzte Daten enthalten.

Prozess 1	Prozess 2
$X_1 := \max(a_1, b_2);$	$X_2 := \min(a_1, b_1);$
$Y_1 := \max(a_2, b_1);$	$Y_2 := \min(a_2, b_2);$
$Z_1 := X_1 + Y_1;$	$Z_2 := X_2 - Y_2;$

Abbildung 5.2: Zwei disjunkte Prozesse mit gemeinsam benutzten Daten.

Da die gemeinsam benutzten Daten a_1, b_1, a_2 und b_2 zwar gleichzeitig benutzt, aber von keinem der beiden konkurrenten Prozesse verändert werden, sind die in diesem Beispiel angegebenen Abläufe noch immer disjunkt.

Komplizierter werden die Verhältnisse, wenn die konkurrenten Prozesse gemeinsame Daten enthalten, die verändert werden. Wir sprechen dann von **überlappenden Prozessen**.

Sehr offenkundig werden die Probleme am Beispiel in Abbildung 5.3 sichtbar. Wir unterstellen, dass die einzelnen Anweisungen in diesem Beispiel atomar ausgeführt werden, d. h. aus Sicht der Variablen ohne Überlappung.

Das Ergebnis, das Prozess 1 bzw. Prozess 2 in Abbildung 5.3 druckt, hängt offensichtlich von der Reihenfolge der Ausführung der Anweisungen ab. Welches Ergebnis bei welcher Anweisungsreihenfolge gedruckt wird, gibt Abbildung 5.4 an.

$a := 10;$

Prozess 1	Prozess 2
(1.1) $a := a * a;$	(2.1) $a := a/2;$
(1.2) PRINT(a);	(2.2) PRINT(a);

Abbildung 5.3: Zwei überlappende Prozesse.

Anweisungsreihenfolge	Ergebnis Prozess 1	Ergebnis Prozess 2
1.1 - 1.2 - 2.1 - 2.2	100	50
1.1 - 2.1 - 1.2 - 2.2	50	50
1.1 - 2.1 - 2.2 - 1.2	50	50
2.1 - 2.2 - 1.1 - 1.2	25	5
2.1 - 1.1 - 1.2 - 2.2	25	25
2.1 - 1.1 - 2.2 - 2.1	25	25

Abbildung 5.4: Die Ergebnisse sind abhängig von der Reihenfolge der Prozesse.

Die unterschiedlichen Ergebnisse hängen von den relativen Geschwindigkeiten der überlappenden Prozesse ab.

Da Hardware-Prozessoren im allgemeinen die Verarbeitung von Programmen mit nicht präzis definierten Geschwindigkeiten vornehmen, können aus diesem Phänomen für die Steuerungsaufgaben von Betriebssystemen schwierige Probleme entstehen.

Wenn Abläufe konkurrenter Prozesse von der relativen Geschwindigkeit ihrer Ausführung abhängen, nennen wir diese Abläufe **zeitkritisch**.

Die Frage zeitkritischer Abläufe soll an einem weiteren Beispiel in Abbildung 5.5 illustriert werden. Wir betrachten zwei konkurrente Prozesse, von denen der eine, Beobachter genannt, auftretende Ereignisse beobachten und zählen soll. Der andere Prozess, der Berichterstatter, druckt periodisch die Zahl der seit dem letzten Ausdruck gezählten Ereignisse und setzt danach den Zähler auf 0 zurück.

Beobachter	Berichterstatter

`repeat`
`  beobachte Ereignis;`
`  ZAEHLER := ZAEHLER + 1;`
`until Ende;`
	`repeat`
	`  PRINT(ZAEHLER);`
	`  ZAEHLER := 0;`
	`until Ende;`

Abbildung 5.5: Beobachter- und Berichterstatter-Prozesse.

Beobachter und Berichterstatter sind überlappende Prozesse, da beide die Variable `ZAEHLER` benutzen. Hierdurch entsteht die Unzulänglichkeit, dass die Variable `ZAEHLER` nicht zu jedem Zeitpunkt die Anzahl der beobachteten Ereignisse angibt, falls nämlich der Berichterstatter-Prozess die Variable `ZAEHLER` löscht, bevor sie der Beobachter-Prozess inkrementieren konnte. Hierdurch kann der Stand der Variable `ZAEHLER` beliebig von der Anzahl tatsächlich beobachteter Ereignisse abweichen. Es kann sogar der Fall eintreten, dass der Beobachter-Prozess überhaupt nicht dazu

kommt, irgendwelche Ereignisse zu zählen, falls nämlich der Berichterstatter den `ZAEHLER` immer löscht, nachdem der Beobachter diesen inkrementiert hat. Diese Situationen, in denen zwei oder mehrere Prozesse quasi im Wettbewerb einen gemeinsamen Speicher lesen oder schreiben und das Ergebnis davon abhängt, wer wann genau läuft, werden **Race Conditions** genannt.

5.1.2 Kritische Abschnitte und gegenseitiger Ausschluss

Wie die letzten Beispiele des vorausgehenden Abschnitts deutlich gezeigt haben, ist es unzweckmäßig, während der Veränderung gemeinsam benutzter Variablen durch einen Prozess anderen Prozessen Zugriff zu diesen Variablen zu erlauben. Das Ergebnis ist unter diesen Umständen nicht-determiniert.

In konkurrenten Prozessen werden mehrere Abschnitte unterschieden:

- **unkritische Abschnitte**, in denen nicht auf von mehreren Prozessen gemeinsam benutzte Daten zugegriffen wird, und
- **kritische Abschnitte** (**critical sections**), die lesend und/oder schreibend gemeinsame Daten verarbeiten, die von mehreren konkurrenten Prozessen benutzt werden.

Kritische Abschnitte in konkurrenten Prozessen müssen eindeutig gegen die unkritischen Abschnitte abgegrenzt sein. Ein Prozess kann nacheinander mehrere verschiedene kritische Abschnitte durchlaufen, allerdings dürfen diese nicht ineinander geschachtelt sein.

Damit kritische Abschnitte determinierte Resultate zur Folge haben, müssen solche Abschnitte ohne Störung durch die anderen konkurrenten Prozesse in einer Sequenz ausgeführt werden. Innerhalb eines kritischen Abschnitts greift ein Prozess also exklusiv auf gemeinsam benutzte Betriebsmittel zu.

Die Notwendigkeit kritischer Abschnitte in konkurrenten Prozessen ist ein logisches Problem, das immer bei gemeinsam benutzten Daten auftritt. Das Problem lässt sich weder durch zusätzliche Hardware noch durch zusätzliche Software lösen.

Daten in Prozessen sind ebenfalls Betriebsmittel, die von diesen Prozessen benutzt werden. In Abschnitt 5.1.1 hatten wir disjunkte Prozesse betrachtet, die dieses Attribut dadurch erhielten, dass sie keine gemeinsamen Daten benutzten oder gemeinsam benutzte Daten während der gesamten Laufzeit dieser konkurrenten Prozesse nicht verändert wurden. Allgemeiner kann man sagen, dass Betriebsmittel und damit auch Daten in **gemeinsam benutzbare (shareable)** und **nicht-gemeinsam benutzbare (non shareable) Betriebsmittel** unterteilt werden können.

Nicht gemeinsam benutzbare Betriebsmittel können von konkurrenten Prozessen nur in *kritischen Abschnitten* belegt werden. Beispiele nicht-gemeinsam benutzbarer Betriebsmittel sind

- die meisten peripheren Geräte sowie
- veränderliche Daten.

Zu den gemeinsam benutzbaren Betriebsmitteln gehören beispielweise

- Prozessoren (CPUs)
- nur lesbare Daten (read-only files).

Damit die Resultate von Abläufen konkurrenter Prozesse determiniert bleiben, müssen deren kritische Abschnitte *exklusiv* ausgeführt werden, d. h. *ein* kritischer Abschnitt zu *einer* Zeit. Das Betreten und Verlassen eines kritischen Abschnitts muss zwischen den Prozessen abgestimmt (synchronisiert) sein. Die Lösung dieses Problems bezeichnet man als **wechselseitigen (gegenseitigen) Ausschluss (*mutual exclusion*)**. Eine abstrakte Form des wechselseitigen Ausschlusses sieht folgendermaßen aus:

```
/* nicht kritischer Abschnitt */

enter_critical_region;
/* kritscher Abschnitt */
exit_critical_region

/* nicht kritischer Abschnitt */
```

Die zwei Funktionen `enter_critical_region` und `exit_critical_region` regeln das Betreten und Verlassen des kritischen Abschnitts. Jeder Prozess, der versucht, in den kritischen Abschnitt einzutreten, muss zuerst `enter_critical_region` ausführen, und danach dann `exit_critical_region`, damit andere Prozesse in ihren kritischen Abschnitt eintreten können.

5.2 Synchronisation

Die Aufgabe der Realisierung des wechselseitigen Ausschlusses, also die Realisierung von `enter_critical_region` und `exit_critical_region`, ist die zentrale Frage der Synchronisation konkurrenter Prozesse.

Wir betrachten daher im Folgenden sequentielle, zyklische Prozesse, die konkurrent zueinander ablaufen und hinsichtlich ihrer kritischen Abschnitte miteinander synchronisiert werden müssen. Zyklisch nennen wir diese Prozesse deshalb, weil sie am Ende ihres Ablaufs die gleiche Aufgabe sofort oder später erneut wieder abwickeln sollen.

Beispiel eines solchen zyklisch-sequentiellen Prozesses ist etwa der schreibende Zugriff zu einer gemeinsam genutzten Datei. Sämtliche Aufgaben der Vor- und Nachbereitung des Zugriffs zu dieser Datei im Auftrag mehrerer Auftraggeber können als unkritisch und damit auch als beliebig parallel ausführbar betrachtet werden. Anders verhält es sich jedoch mit dem Vorgang der Veränderung der Datei selbst. Falls nämlich dieser Zugriff gleichzeitig vorgenommen würde, wäre der resultierende Zustand der Datei unbestimmt. Es muss also sichergestellt sein, dass zu einer Zeit nur ein Prozess die Datei verändert.

5.2.1 Synchronisationsvariablen

Betrachten wir zwei Prozesse, die jeder aus einem oder mehreren unkritischen parallel ausführbaren Abschnitten sowie aus jeweils genau einem exklusiven Abschnitt bestehen. Der wechselseitige Ausschluss wird über die globale Synchronisationsvariable s gesteuert, die mit 1 initialisiert sei.

$s := 1;$

Prozess 1	Prozess 2
`repeat`	`repeat`
`  while` $s <> 2$ `do begin`	`  while` $s <> 1$ `do begin`
`    kritischer Abschnitt 1;`	`    kritischer Abschnitt 2;`
$s := 2;$	$s := 1;$
`    unkritischer Abschnitt;`	`    unkritischer Abschnitt;`
`  end;`	`  end;`
`until false;`	`until false;`

Abbildung 5.6: Zwei Prozesse mit einer globalen Synchronisationsvariable s.

Eine mögliche Lösung dieser Aufgabe ist in Abbildung 5.6 angegeben. Obwohl die gestellte Synchronisationsaufgabe offensichtlich richtig gelöst ist, wirft sie eine Reihe von Fragen auf:

(1) Die kritischen Abschnitte können nur in der abwechselnden Reihenfolge 1, 2, 1, 2, 1, ... ausgeführt werden, d. h. ein Prozess kann nicht schneller als der andere ablaufen, es ist kein Überrunden möglich.

(2) Ein Prozess hindert sich selbst daran, ein zweites Mal hintereinander den kritischen Abschnitt zu betreten. Er muss solange warten, bis der andere Prozess seinen kritischen Abschnitt durchlaufen hat und die gemeinsame Variable wieder zurückgesetzt hat. Geschieht dies nie, so wartet der andere Prozess ewig. Also behindert das Stoppen des einen Prozesses den anderen Prozess, d. h. letzterer wird in seinem Ablauf ad infinitum blockiert, obwohl dies offensichtlich gar nicht notwendig wäre.

Eine Abhilfe bezüglich der genannten Restriktionen schafft die in Abbilfung 5.7 angegebene Lösung. In Zeile 2 wird der Wunsch angemeldet, in den kritischen Bereich eintreten zu können. In Zeile 3 wartet der Prozess so lange, wie der andere Prozess noch in seinen kritischen Bereich eintreten möchte oder schon eingetreten ist, was durch $s_2 = 0$ angezeigt ist. Sobald der andere Prozess $s_2 = 1$ setzt, kann das Warten beendet werden. Zeile 5 zeigt an, dass der kritische Bereich verlassen worden ist; ggf. kann nun das Warten von Prozess 2 in Zeile 3 beendet werden.

Bei dieser Lösung besteht zwar nicht die o. g. Einschränkung (1) und das Stoppen eines Prozesses behindert den anderen Prozess auch nur dann, wenn der gestoppte Prozess sich gerade in seinem kritischen Abschnitt befand, dafür birgt aber diese Variante eine andere Gefahr in sich. Wenn nämlich beide Prozesse gleichzeitig auf die `while`-Anweisung stoßen, dann sind offensichtlich sowohl s_1 als auch s_2 gleich Null, und keiner der beiden Prozesse kann ohne Eingriff von außen diese Anweisung

$s_1 := 1;\ s_2 := 1;$

Prozess 1		Prozess 2
`repeat`	1	`repeat`
$s_1 := 0;$	2	$s_2 := 0;$
`while` $s_2 = 0$ `do begin end;`	3	`while` $s_1 = 0$ `do begin end;`
`kritischer Abschnitt 1;`	4	`kritischer Abschnitt 2;`
$s_1 := 1;$	5	$s_2 := 1;$
`unkritischer Abschnitt;`	6	`unkritischer Abschnitt;`
`until false;`	7	`until false;`

Abbildung 5.7: Zwei Prozesse mit zwei globalen Synchronisationsvariablen.

jemals wieder verlassen. Diesen Zustand bezeichnet man als gegenseitige Blockade oder **Systemverklemmung** (**Deadlock**), siehe Abschnitt 5.3.

5.2.2 Anforderungen an einen wechselseitigen Ausschluss und eine erste Lösung

Im Laufe der Zeit sind eine ganze Reihe von Lösungen des Problems des wechselseitigen Ausschlusses für zwei oder beliebig viele Prozesse vorgeschlagen worden, die z. T. wiederum andere Mängel hatten. Eine wichtige Lehre aus diesen Lösungsversuchen ist daher folgende Liste von **Anforderungen**, die ein Algorithmus zur Realisierung des wechselseitigen Ausschlusses erfüllen muss:

1. Zu einer Zeit befindet sich höchstens ein Prozess in seinem kritischen Abschnitt (*mutual exclusion*).

2. Falls mehrere Prozesse gleichzeitig versuchen, den kritischen Abschnitt zu betreten, so wird innerhalb einer endlichen Zeit eine Entscheidung getroffen, welcher Prozess zuerst in den kritischen Abschnitt eintritt.

3. Wenn ein Prozess in den kritischen Abschnitt eintreten möchte, darf er nicht beliebig oft von anderen Prozessen, die ebenfalls in den kritischen Abschnitt eintreten wollen, überholt werden, also im Extremfall verhungern. Generell sollte die Wartezeit dieses Prozesses gemessen in der Zahl der Durchläufe anderer Prozesse durch ihren kritischen Abschnitt *fair* sein (fairness condition).

4. Kein Prozess darf außerhalb eines kritischen Abschnitts einen anderen Prozess blockieren. Wenn ein Prozess im unkritischen Abschnitt gestoppt wird, so beeinflusst dies nicht den weiteren Ablauf der anderen Prozesse.

5. Die Lösung darf nicht auf irgendwelchen Annahmen über die relative oder absolute Ausführungsgeschwindigkeit der Prozesse beruhen.

Der erste Algorithmus für n Prozesse, der alle fünf o. g. Anforderungen erfüllt, wurde in [EiM72] vorgestellt. Wir nehmen an, dass die Prozesse von 0 bis $n-1$ nummeriert sind. Als zentrale Datenstruktur benötigt der Algorithmus:

```
var zustand : array [0..n-1] of (aussen, angemeldet, innen)
    turn    : 0.. n-1
```

- `zustand[i]` zeigt den Zustand von Prozess i an; die Werte haben in etwa folgende Bedeutung.
  - `aussen`: im unkritischen Bereich;
  - `angemeldet`: der Prozess möchte in den kritischen Bereich eintreten;
  - `innen`: im kritischen Bereich.
  - Der initiale Wert für alle i ist `aussen`.
- Der Wert `turn` zeigt an, welcher Prozess sich gerade im kritischen Bereich befindet, sofern sich überhaupt ein Prozess gerade im kritischen Bereich befindet.
  - Der initiale Wert von `turn` ist beliebig.

Prozess i führt das Programm nach Eisenberg und McGuire aus.

```
var j : 0..n;

repeat
   /* enter_critical_region */
   repeat
      zustand[i] := angemeldet;
      j := turn;
      while j <> i do begin
         if zustand[j] = aussen
            then j := (j + 1) mod n
            else j := turn;
      end;
      zustand[i] := innen;
      j := 0;
      while (j < n) and (j = i or zustand[j] <> innen)
      do j := j + 1;
   until (j >= n) and (turn = i or zustand[turn] = aussen)
   turn := i;

   kritischer Abschnitt von Prozess i;

   /* exit_critical_region */
   j := (turn + 1) mod n;
   while (zustand[j] = aussen) do
      j := (j + 1) mod n;
   turn := j;
   zustand[i] = aussen;

   unkritischer Abschnitt von Prozess i;

until false;
```

Wir wollen nun zeigen, dass diese Lösung alle fünf Anforderungen erfüllt. Dabei wird sich zeigen, dass diese Lösung den Eintritt in den kritischen Abschnitt immer

der Reihe nach gemäß der Nummer der Prozesse erlaubt, wobei auf $n-1$ wieder 0 folgt. Wenn also die Prozesse i, j und k in den kritischen Abschnitt eintreten wollen und folgendermaßen in der zyklischen Reihenfolge angeordnet sind:

$$\texttt{turn}, \ldots, j, \ldots, i, \ldots, k, \ldots, (n + \texttt{turn} - 1) \bmod n$$

dann wird Prozess j vor Prozess i und Prozess k nach Prozess i in den kritischen Abschnitt eintreten können. Zur Sprachvereinfachung sagen wir, dass Prozess j **vor** Prozess i und Prozess k **hinter** Prozess i liegt.

Die erste `while`-Schleife sucht ausgehend vom Prozess mit der Nummer `turn` bis zum Prozess i nach einem Prozess im Zustand `angemeldet` oder `innen`. Falls ein solcher Prozess gefunden wird, wird die Schleife durch die Zuweisung `j:=turn` erneut gestartet; Prozess i kann also keinen vor ihm stehenden Prozess überholen. Beim erneuten Durchlauf durch die Schleife kann sich der Wert von `turn` geändert haben, weil ein Prozess seinen kritischen Abschnitt verlassen hat. Der neue Wert von `turn` kann aber wegen der `while`-Schleife im Epilog nur zwischen dem alten Wert von `turn` und i liegen.

Nachdem Prozess i die erste `while`-Schleife durchlaufen hat, kann es noch andere Prozesse hinter Prozess i geben, die den kritischen Abschnitt gerade betreten oder schon darin sind, weil sie die `while`-Schleife schon früher durchlaufen haben. Es kann also hinter der ersten `while`-Schleife mehrere Prozesse geben, deren Zustand auf `innen` gesetzt ist.

Die zweite `while`-Schleife sucht nun nach einem *anderen* Prozess im Zustand `innen`. Falls ein solcher Prozess k gefunden wird, ist anschließend $j = k < n$ und die folgende `until`-Bedingung wird falsch. Prozess i kann nur dann in den kritischen Abschnitt eintreten, wenn $j = n$ ist. Es ist unmöglich, dass ein anderer Prozess k gleichzeitig $j = n$ hat, weil `zustand[i]=innen` während der gesamten Dauer der zweiten `while`-Schleife gilt, also speziell beim Test `zustand[k]<>innen`. Daher kann höchstens ein Prozess in den kritischen Abschnitt eintreten, Anforderung 1 ist also erfüllt.

Falls die zweite `while`-Schleife erfolglos, also mit $j < n$ endet, muss die innere `repeat`-Schleife wiederholt werden. Prozess i setzt dabei seinen Zustand auf `angemeldet` zurück. Wenn Prozess j vor Prozess i liegt, kann ihn Prozess i wegen der ersten `while`-Schleife nicht mehr überholen. Prozess j durchläuft irgendwann seinen kritischen Abschnitt, setzt danach `zustand[j]` auf `aussen` zurück und liegt danach wegen der Änderung von `turn` hinter Prozess i. Wenn Prozess j hinter Prozess i liegt, setzt er irgendwann `zustand[j]` auf `angemeldet` zurück und kann wegen der ersten `while`-Schleife nicht mehr `zustand[j]` auf `innen` setzen. Insgesamt können also die Bedingungen, die zur erfolglosen Durchführung der `while`-Schleife führen, nicht beliebig lange gelten. Daher ist Anforderung 2 erfüllt.

Bei den vorgenannten Überlegungen ist auch gezeigt worden, dass Prozesse, die ihre Absicht anmelden, in den kritischen Abschnitt einzutreten, in der zyklischen Reihenfolge ihrer Nummern bedient werden. Im ungünstigsten Fall muss ein Prozess darauf warten, dass $n-1$ andere Prozesse ihren kritischen Abschnitt durchlaufen. Dies ist der maximal erreichbare Grad von Fairness, d. h. Anforderung 3 ist optimal erfüllt.

Wenn ein Prozess k im unkritischen Abschnitt gestoppt wird, dann bleibt `zustand[k] = aussen`. Dieser Prozess behindert daher keine anderen Prozesse beim

Eintreten in den kritischen Abschnitt. Schließlich haben wir bei unseren obigen Überlegungen keine Annahmen bzgl. der relativen und/oder absoluten Ausführungsgeschwindigkeit der Prozesse gemacht. Anforderungen 4 und 5 sind daher auch erfüllt.

5.2.3 Semaphore

Die bisher beschriebenen Synchronisationsverfahren waren in mancherlei Hinsicht schwerfällig und auch undurchsichtig. Wirklich wünschenswert wäre eine übersichtliche Lösung, die effizient arbeitet und für Ein- wie für Multiprozessorsysteme geeignet ist.

Wenn sich etwa ein Prozess in seinem kritischen Abschnitt befindet, so sollte kein anderer Prozess ständig nach Erlaubnis zum Eintritt in seinen kritischen Abschnitt fragen. Dieses Prinzip, das man **aktives Warten** (**busy wait**) nennt, ist unwirtschaftlich, da es unproduktive Prozessorzeit benötigt. Vielmehr sollte es möglich sein, dass ein Prozess, der in den kritischen Abschnitt eintreten möchte, *deaktiviert* und wieder *aufgeweckt* wird, wenn der derzeit aktive Prozess seinen kritischen Abschnitt verlässt. Wir werden aber sehen, dass aktives Warten sich zumindest bei mehreren Prozessoren nicht ganz vermeiden lässt.

Die Hauptschwierigkeit der bisher beschriebenen Synchronisationsverfahren bestand darin, dass nicht gewährleistet war, dass die für den Zugriff zu den Synchronisationsvariablen erforderliche Folge von Einzelschritten *ungestört nacheinander* und *nicht unterbrochen* durch andere Prozesse ablief.

Ein weitaus übersichtlicheres und softwaretechnisch einfacher zu implementierendes Synchronisationsverfahren hat E. W. Dijkstra [Di68] vorgeschlagen. Anstelle expliziter Synchronisationsvariablen wird ein sogenannter **Semaphor**[2] benutzt. Ein Semaphor kann als eine Variable betrachtet werden, die einen ganzzahligen Wert hat. Mit Hilfe dieses Semaphors wird eine bestimmte kritische Ressource überwacht. Initialisiert man s mit k, so werden höchstens k gleichzeitige Zugriffe auf diese Ressource zugelassen, in den allermeisten Fällen wird $k = 1$ sein. Zur Benutzung dieser Semaphore stehen zwei Operationen `down` und `up` (Verallgemeinerung von *sleep* und *wakeup*) zur Verfügung, die als *unteilbare* (*atomare*) *Elementaroperationen* betrachtet werden, d. h., vor Abschluss dieser Operationen sind diese nicht unterbrechbar.

1. Die **down-Operation** `down`(s) angewendet auf den Semaphor s bedeutet, dass der gegenwärtige Wert des Semaphors um 1 vermindert wird. Wird der resultierende Wert negativ, blockiert der Prozess, der die Operation `down` ausführt.

2. Die **up-Operation** `up`(s) bedeutet, dass der Wert des Semaphors s um 1 erhöht wird. Ist der resultierende Wert nicht positiv, wird ein durch eine `down`-Operation blockierter Prozess freigegeben, also geht dieser Prozess vom Zustand *blockiert* zum Zustand *bereit* über.

Sei s der Semaphor eines bestimmten kritischen Abschnitts, so wird `down`(s) vor Beginn und `up`(s) nach Abschluss des kritischen Abschnitts ausgeführt.

[2]Semaphore (griech. Zeichenträger) werden verwendet zur Signalisierung mittels Flaggen, Leuchtzeichen oder sonstiger Zeichen. Vor Einführung der Telegraphie wurden Semaphore zur Übertragung von Nachrichten zwischen entfernten Punkten benutzt (z. B. im Schiffsverkehr).

```
var s : semaphor;

s := k; (* Initialisierung *)

Prozess i für i= 1,2, ..., n

repeat
   down(s);
   kritischer Abschnitt i;
   up(s);
   unkritischer Abschnitt;
until false;
```

Man unterscheidet zwei Typen von Semaphoren. Semaphore, die mit 1 initialisiert werden, bezeichnet man als **binäre Semaphore**, und Semaphore, deren Werte auch größere Zahlen sein können, werden **allgemeine Semaphore** genannt.

Implementierung von Semaphoren. Zur Implementierung von Semaphoren benutzen wir eine Warteschlange, sie nimmt die Prozesse auf, die aufgrund des Semaphors warten. Die Prozesse in der Warteschlange können z. B. nach dem FIFO-Prinzip ausgewählt werden.

```
down(s):
    s : = s - 1;
    if s < 0
        then begin
            füge den Prozess in die Warteschlange ein;
            versetze den Prozess in den blockierten Zustand;
        end;

up(s):
    s : = s + 1;
    if s <= 0   (* Die Warteschlange ist nicht leer gewesen *)
        then begin
            wähle einen Prozess aus der Warteschlange aus;
            versetze ihn in den bereiten Zustand;
        end;
```

Zu jedem gegebenen Zeitpunkt kann der Wert s wie folgt interpretiert werden:

- $s > 0$: s ist die Anzahl der Prozesse, die noch die Operation `down` ohne Blockierung ausführen können.
- $s \leq 0$: $|s|$ ist die Anzahl der Prozesse, die sich in der Warteschlange des Semaphors s befinden.

Die Operationen `down` und `up` sollen die kritische Abschnitte schützen, aber sie haben *selbst* kritische Abschnitte. Was zunächst wie ein Widerspruch aussieht, ist aber unvermeidlich. Der Vorteil von Semaphoren ist, dass die Operationen nur sehr kurze Zeit benötigen und die Implementierung in die Hardware verlagert werden kann.

Bei einem *Einprozessorsystem* können die kritischen Abschnitte von Semaphoren einfach durch Sperren von Interrupts realisiert werden. Dieser Ansatz ist aber gefährlich, weil ein Benutzerprozess die Macht bekommt, die Unterbrechungen abzuschalten. Und er funktioniert auch nicht bei einem *Multiprozessorsystem* mit gemeinsamem Speicher, siehe Abbildung 5.8. Nur die CPU, die das Sperren ausgeführt

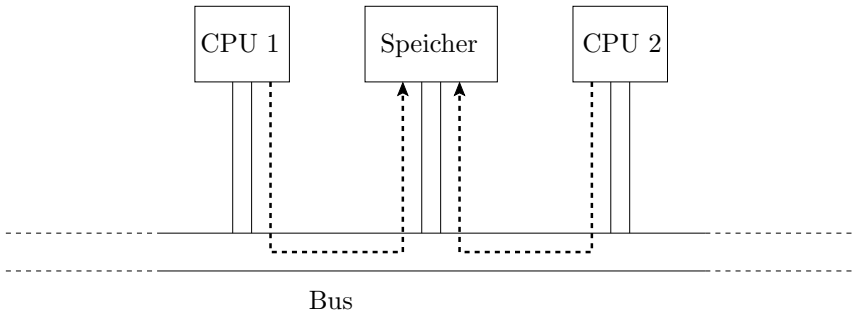


Abbildung 5.8: Wenn CPU 1 die Unterbrechungen abschaltet, kann CPU 2 trotzdem über den Bus auf den Speicher zugreifen.

hat, wird durch die Abschaltung der Unterbrechungen betroffen, die anderen CPUs können weiter laufen und auf den gemeinsam genutzten Speicher zugreifen. Ein Ansatz, der auch für Multiprozessorsysteme funktioniert, ist, die atomaren Operationen mit Hilfe eines speziellen Prozessorbefehls, der `TSL`-Anweisung (Test and Set Lock), zu realisieren. Um die `TSL`-Anweisung zu verwenden, wird eine globale Variable `lock` zur Koordinierung des Zugriffs auf den gemeinsam genutzten Speicher benutzt. Die Anweisung `TSL RX LOCK` liest den Inhalt der Speicherzelle `lock` ins Register `RX` und ersetzt ihn durch einen anderen Wert (nicht Null). Diese Lies- und Schreib-Operationen sind garantiert unteilbar, da der Bus gesperrt wird, wenn eine CPU die `TSL`-Anweisung ausführt. Für die Dauer der `TSL`-Anweisung wird so allen CPUs der Zugriff auf den Speicher verboten.

Jeder Semaphor `s` wird durch eine `lock` Variable geschützt. Bevor ein Prozess den Wert von `s` um 1 vermindert oder erhöht, startet er zuerst die Operation `enter_region`. Nachdem er den kritischen Abschnitt verlassen hat, führt er `leave_region` aus; siehe den folgenden Assembler-Code.

```
enter_region(s.lock):
    TSL REGISTER LOCK (* kopiere Sperrvariable lock ins Register
                         und setzt lock auf 1 *)
    CMP REGISTER, #0  (* vergleiche der Wert im Register mit 0 *)
    JNE enter_region  (* wenn der Wert nicht 0 ist, war die Sperre
                         schon gesetzt, gehe zurück zum Anfang *)
    RET

leave_region(s.lock):
    MOVE LOCK, #0     (* speichere 0 in lock *)
    RET               (* Rücksprung *)
```

Man beachte, dass `enter_region` ein aktives Warten ist. Trotzdem werden Semaphore oft so realisiert, da der kritische Abschnitt extrem kurz ist. Nachdem ein

Prozess `down` erfolgreich ausführen konnte, tritt er entweder in den kritischen Abschnitt ein oder stellt sich in die Warteschlange, und dabei passiert natürlich kein aktives Warten mehr.

5.2.3.1 Das Erzeuger/Verbraucher-Problem

Einige Anwendungsbeispiele sollen den Gebrauch der Semaphor-Operationen näher erläutern.

Zwei zyklische Prozesse *Erzeuger und Verbraucher* produzieren bzw. konsumieren eine Ware. Zwischen Erzeuger und Verbraucher ist eine Einweg-Kommunikation vorgesehen, wobei der Austausch der Ware über einen Puffer stattfindet. Die vom Erzeuger produzierte Ware wird im Puffer abgelegt und vom Verbraucherprozess dort abgeholt, siehe Abbildung 5.9, dieses Problem ist für viele Vorgänge in Betriebssystemen charakteristisch, z. B. Ein-/Ausgabevorgänge, Verwaltung von Warteschlangen und Realisierung von Spooling-Technik, siehe auch Kurseinheit 2.

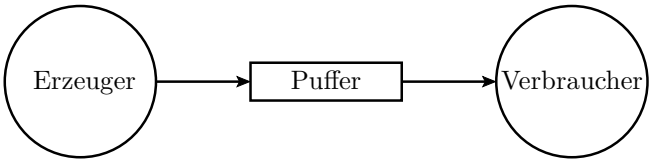


Abbildung 5.9: Das Erzeuger/Verbraucher-Problem.

Der einfachste Fall dieser Aufgabenstellung geht von einem *unbegrenzt* großen Puffer aus, siehe Abbildung 5.10. Der Semaphor `inhalt`, der mit 0 initialisiert wird, soll die Anzahl der Elemente im Puffer beschreiben.

```
var inhalt : semaphor;
inhalt := 0;
```

Erzeuger-Prozess	Verbraucher-Prozess

`repeat`
`  Erzeuge Ware;`
`  Bringe Ware nach Puffer;`
`  up(inhalt);`
`until false;`
	`repeat`
	`  down(inhalt);`
	`  Hole Ware aus Puffer;`
	`  Verbrauche Ware;`
	`until false;`

Abbildung 5.10: Einfache Erzeuger- und Verbraucher-Prozesse mit einem unendlich großen Puffer.

Der Verbraucher kann also Ware nur dann aus dem Puffer entnehmen, wenn dieser nicht leer ist. Allerdings kann der gleichzeitige Zugriff von Erzeuger und Verbraucher zum Puffer zu Störungen führen. Um das Problem zu lösen, kann ein zusätzlicher binärer Semaphor `zustand` eingefügt werden, der nur einem einzigen Prozess den Zugriff auf den Puffer zulässt. Der Wert 1 des Semaphors `zustand` zeigt an, dass keiner der beiden Prozesse gerade auf den Puffer zugreift, siehe Abbildung 5.11.

```
var inhalt, zustand : semaphor;
inhalt := 0; zustand := 1;
```

Erzeuger-Prozess

```
repeat
  Erzeuge Ware;
  down(zustand);
  Bringe Ware nach Puffer;
  up(zustand);
  up(inhalt);
until false;
```

Verbraucher-Prozess

```
repeat
  down(inhalt);
  down(zustand);
  Hole Ware aus Puffer;
  up(zustand);
  Verbrauche Ware;
until false;
```

Abbildung 5.11: Erzeuger- und Verbraucher-Prozesse mit Semaphoren. Nur ein Prozess darf auf den Puffer zugreifen.

Übungsaufgabe 5.1 Können die beiden `up`-Operationen im Erzeuger-Prozess bzw. die beiden `down`-Operationen im Verbraucher-Prozess gemäß Abbildung 5.11 vertauscht werden?

Man kann nun den allgemeinen Semaphor `inhalt` in Abbildung 5.11 durch den binären Semaphor `verzoegern` ersetzen, indem der Verbraucher-Prozess nur dann Ware aus dem Puffer entnehmen darf, wenn `verzoegern` den Wert 1 hat, der Puffer sich also nicht in einem kritischen Zustand leer befindet, siehe Abbildung 5.12, wobei `anzahl` die Füllung des Puffers zählt. Solange der Puffer nicht leer ist, kann der Verbraucher mit beliebiger Geschwindigkeit verbrauchen.

```
var zustand, verzoegern : semaphor;
    anzahl : integer;
zustand := 1; verzoegern := 0; anzahl := 0;
```

Erzeuger-Prozess

```
repeat
   Erzeuge Ware;
   down(zustand);
   Bringe Ware nach Puffer;
   anzahl := anzahl + 1;
   if anzahl = 1
      then up(verzoegern);
   up(zustand);
until false;
```

Verbraucher-Prozess

```
alt: integer;
repeat
   down(verzoegern);
   repeat
      down(zustand);
      Hole Ware aus Puffer;
      anzahl := anzahl - 1;
      alt := anzahl;
      up(zustand);
      Verbrauche Ware;
   until alt = 0;
until false;
```

Abbildung 5.12: Erzeuger- und Verbraucher-Prozesse mit binären Semaphoren.

Für praktische Anwendungen ist der realistische Fall eines in seiner Kapazität beschränkten Puffers interessant, siehe die folgende Übung.

Übungsaufgabe 5.2 Man erweitere die Lösung des Erzeuger-Verbraucher-Problems in Abbildung 5.12 in der Weise, dass die Kapazität des Puffers auf n Elemente beschränkt ist. Zur Lösung verwende man die beiden allgemeinen Semaphore `leer` und `voll` sowie den binären Semaphor `zustand`.

1. Geben Sie die vollständige algorithmische Beschreibung für den Erzeuger- und für den Verbraucher-Prozess an!

2. In welchem Sinne ist die erhaltene Lösung symmetrisch?

5.2.3.2 Das Philosophen-Problem

Bereits in den Beispielen mit expliziten Synchronisationsvariablen war unterschieden worden, ob die Synchronisationsvariablen nur von einem Prozess verändert werden oder ob mehrere Prozesse wechselseitig diese Synchronisationsvariablen manipulieren konnten. Im Hinblick auf die Synchronisation mittels Semaphore soll nun dieser Unterschied noch deutlicher herausgestellt werden.

Wir betrachten hierzu das ebenfalls auf Dijkstra zurückgehende Beispiel der **dinierenden Philosophen**. $n \geq 2$ Philosophen sitzen an einem runden Tisch. Jeder dieser n Philosphen durchläuft zyklisch die drei Zustände `Denken` (0), `Hungrig` (1) und `Essen` (2).

Um essen zu können, braucht jeder Philosoph gleichzeitig eine *links* und eine *rechts* von seinem Teller liegende Gabel. Den n Philosophen stehen aber nur insgesamt n Gabeln zur Verfügung, zwei an dem runden Tisch aneinander angrenzende Teller sind durch genau eine Gabel getrennt. Wenn also zwei benachbarte Philosophen gleichzeitig hungrig werden und dann essen wollen, so wird es Schwierigkeiten geben[3]. Betrachten wir jeden Philosophen als zyklisch sequentiellen Prozess, so wird es gerade darauf ankommen, das gleichzeitige Essen zweier benachbarter Philosophen oder auch nur den Versuch dazu zu vermeiden.

```
var rechts_i, links_i : semaphor;
rechts_i := 1; links_i := 1;
```

Philosoph i-Prozess

```
repeat
   denken;                /* Philosoph denkt */
   hungrig;               /* Philosoph hat Hunger */
   down(links_i);         /* nimm linke Gabel */
   down(rechts_i);        /* nimm rechte Gabel */
   essen;                 /* Philosoph ißt */
   up(rechts_i);          /* lege linke Gabel zurück */
   up(links_i);           /* lege rechte Gabel zurück */
until false;
```

Abbildung 5.13: Keine Lösung für das Problem der dinierenden Philosophen.

[3]Auch wenn sie nacheinander hungrig werden, gibt es Probleme, und zwar hygienische, von denen hier jedoch abstrahiert wird.

Die einfachste Form einer Lösung dieses Problems wäre die Einführung von Semaphoren für jede Gabel. Jeder Philosoph hätte dann genau einen Semaphor `links` und einen weiteren `rechts`. Nummeriert man die Philosphen im Gegenuhrzeigersinn von 0 bis $n-1$ durch, so entspricht offensichtlich der Gabel `rechts_i` die Gabel `links_j`, $j = (i+1) \bmod n$, für $i = 0, 1, \ldots, n-1$, des rechten Nachbarn. Für den i-ten Philosophen könnte eine Lösung wie in Abbildung 5.13 angegeben lauten.

Bei dieser Variante ist allerdings die Gefahr der gegenseitigen Blockade nicht ausgeschlossen, falls alle Philosophen gleichzeitig essen wollen und jeder seine linke Gabel nimmt, aber die rechte Gabel nie erhalten kann, da ja sein rechter Nachbar ebenfalls auf seine rechte Gabel wartet usw.

Um dieser Gefahr zu entgehen, führen wir für jeden Philosophen eine Statusvariable c_i ein, die die Werte 0 (`denken`), 1 (`hungrig`) oder 2 (`essen`) annimmt. Dann muss für $i = 0, \ldots, n-1$ offensichtlich verhindert werden, dass

$$\Big(c_i = 2 \quad \text{und} \quad c_{(i+1) \bmod n} = 1\Big) \quad \text{oder} \quad \Big(c_i = 2 \quad \text{und} \quad c_{n+(i-1) \bmod n} = 1\Big)$$

gleichzeitig gilt. Damit der Übergang von $c_i = 1$ nach 2 vollzogen werden kann, muss die Bedingung

$$c_i = 1 \qquad \text{und} \qquad c_{(i+1) \bmod n} \neq 2 \qquad \text{und} \qquad c_{n+(i-1) \bmod n} \neq 2 \tag{5.1}$$

erfüllt sein.

Zur allgemeinen Lösung benutzt man zwei Arten von Semaphoren und n Statusvariable.

1. Ein *gemeinsamen Semaphor* `ausschluss`, der mit 1 initialisiert wird. Bevor ein Philosoph versucht, die Gabel zu nehmen oder zurückzulegen, führt er eine *down*-Operation auf `ausschluss` aus. Nachdem er die Gabel genommen oder zurückgelegt hat, führt er eine *up*-Operation auf `ausschluss` aus. Es kann also immer nur ein Philosoph gleichzeitig Gabeln nehmen oder zurücklegen.
2. *n privaten Semaphore* `privat`$[i]$, deren Wert zu Anfang mit 0 initialisiert werden. Das Ziel des Semaphors *privat*$[i]$ ist, den i-ten hungrigen Philosoph zu blockieren, d. h. warten zu lassen, falls eine seiner beiden Gabeln nicht frei ist.
3. Der Statusvektor c_i für den i-ten Philosoph, dessen Wert zu Anfang auf `denken` gesetzt wird, wird benutzt, um zu verfolgen, ob er gerade hungrig ist oder denkt oder isst. Ein Philosoph kann nur in den Zustand von Essen übergehen, wenn seine beiden Nachbarn nicht beim Essen sind.

Folgender Prozess beschreibt den Philosophen i:

```
repeat
   denken;
   Gabel_nehmen(i);    (* nimm zwei Gabeln *)
   essen;
   Gabel_weglegen(i); (* lege beide Gabeln zurück *)
until false;
```

Die Prozedur `Gabel_nehmen(i)`:

```
down(ausschluss);     (* tritt in den kritischen Abschnitt ein *)
c[i] := hungrig;      (* merke, dass Philosoph i hungerig ist *)
teste(i);             (* nimm beide Gabeln, wenn sie frei sind *)
up(ausschluss);       (* verlasse den kritischen Abschnitt *)
down(privat[i]);      (* schlafe ein, wenn nicht
                          beide Gabeln vorhanden sind *)
```

Die Prozedur `Gabel_weglegen(i)`:

```
down(ausschluss);     (* tritt in den kritischen Abschnitt ein *)
c[i] := denken;       (* Philosoph ist mit Essen fertig *)
teste(links(i));      (* evtl. Nachbarn aufwecken *)
teste(rechts(i));     (* evtl. Nachbarn aufwecken *)
up(ausschluss);       (* verlasse den kritischen Abschnitt *)
```

Zum Schluss die Prozedur `teste(i)`:

```
if (c[i] = hungrig and
    c[links(i)]  <> essen and (* link(i) = n+(i-1) mod n *)
    c[rechts(i)] <> essen)    (* rechts(i) = i+1 mod n *)
   then begin
          c[i] := essen;
          up(private[i]);  (* wecke Philosoph auf *)
   end;
```

Die Funktion `teste` prüft dabei genau, ob die Bedingung 5.1 erfüllt ist. Wenn die Testbedingung erfüllt ist, setzt sie das betreffende Element c_i auf 2 (Zustand `essen`) und führt auf dem entsprechenden privaten Semaphor `privat`$[i]$ eine `up`-Operation aus, um den Philosoph zu wecken.

Dieses Beispiel benutzt den gemeinsamen Semaphor `ausschluss`, um sicherzustellen, dass die erwähnte Testfunktion tatsächlich exklusiv benutzt wird. Die privaten Semaphore `privat`$[i]$ sichern in Verbindung mit der Testfunktion den Übergang zum Zustand `essen` nur zu einem solchen Zeitpunkt, zu dem eine gegenseitige Blockade aufgrund des Zustands des Nachbarn nicht möglich ist.

Die Unterscheidung in private und gemeinsame Semaphore hilft häufig, eine Aufgabe hinsichtlich der erforderlichen Synchronisationen durchsichtiger zu machen. Das Problem der dinierenden Philosophen zeigt, wie die Prozesse um den exklusiven Zugriff auf eine begrenzte Anzahl von Ressourcen konkurrieren.

5.2.3.3 Das Leser-Schreiber-Problem

Ein weiteres klassisches Problem ist das Leser-Schreiber-Problem. Es gibt einen von mehreren Prozessen gemeinsam genutzten Datenbereich. Dieser Datenbereich kann eine Datei, ein Hauptspeicherblock oder eine Reihe von Prozessorregister sein. Es gibt mehrere Prozesse, die den Datenbereich nur lesen und einige, die in den Dateibereich nur schreiben. Man kann sich vorstellen, dass mehrere Prozesse z. B. eine

Datenbank eines Reservierungssystems einer Fluggesellschaft auslesen, und ein Prozess die Datenbank aktualisiert. Wir betrachten zuerst das Leser-Schreiber-Problem, das die folgenden Bedingungen erfüllen muss:

1. Beliebige viele Leser können die Datei gleichzeitig lesen.
2. Nur ein Schreiber kann immer jeweils in die Datei schreiben.
3. Während ein Schreiber in die Datei schreibt, kann sie nicht von einem Leser gelesen werden.

Eine Lösung mit der Priorität für die Leser zeigt das folgende Programm.

```
var readcount : integer;(* zählt die Anzahl der Leser *)
    db : semaphor;      (* Kontrolle zum Zugriff auf
                           den Datenbereich *)
    readsem : semaphor; (* Kontrolle für den Zugriff auf readcount *)

    (* Initialisierung *)

    readcount := 0; db := 1; readsem := 1;



Leser-Prozess:

    repeat
       down(readsem);  (* sichert den exklusiven Zugriff
                          auf readcount *)
       readcount := readcount + 1; (* ein Leser kommt dazu *)
       if (readcount = 1) then
          down(db);    (* ab jetzt behalten die Leser
                          die Kontrolle über den Zugriff
                          auf den Datenbereich *)
       up(readsem);    (* gibt den exklusiven Zugriff auf readcount
                          frei, damit die anderen auch lesen können *)
       Lies Daten;
       down(readsem);  (* sichert den exklusiven Zugriff *)
       readcount := readcount - 1; (* ein Leser weniger *)
       if (readcount = 0) then
         up(db);       (* beim letzten Leser wird der exklusive
                          den Zugriff freigegeben. Ein Schreiber
                          darf jetzt schreiben *)
       up(readsem);    (* gibt den exklusiven Zugriff frei *)

    until false
```

```
Schreiber-Prozess:

    repeat
       down(db);        (* sichert den exklusiven Zugriff auf
                           den Datenbereich *)
       Schreibe Daten;
       up(db);          (* gibt den exklusiven Zugriff auf
                           Datenbereich wieder frei *)
    until false
```

Der erste Leser führt eine `down`-Operation auf `db` aus, ab jetzt behalten die Leser die Kontrolle über den Datenbereich, solange mindesten ein Leser liest. Die nachfolgenden Leser brauchen nur die Anzahl der Leser zu erhöhen. Wenn ein Leser fertig ist, erniedrigt er den Zähler `readcount`, der letzte Leser muss das exklusive Zugriffsrecht auf den Datenbereich freigeben, damit der Schreiber auf den Datenbereich zugreifen darf.

Diese Lösung hat aber den Nachteil, dass der Schreiber "verhungern" kann, wenn ständig neue Leser ankommen und er somit den Zugriff auf den Datenbereich nicht erhält. Die nächste Lösung verhindert dies dadurch, dass keine neuen Leser auf den Datenbereich zugreifen dürfen, sobald mindestens ein Schreiber angekündigt hat, schreiben zu wollen. Der Schreiber braucht nur auf Leser zu warten, die noch fertig werden müssen, und nicht auf nach ihm ankommende Leser. Dafür brauchen wir zusätzlich noch drei Semaphore und eine Variable.

1. Eine Variable `writecount` wird gebraucht, um die Anzahl der Schreiber zu zählen. Sobald sie nicht 0 ist, wird kein neuen Leser zugelassen.

2. Ein Semaphor `writesem` kontrolliert den Zugriff auf `writecount`.

3. Durch einen Semaphor `mutex1` wird die Priorität der Schreiber über die Leser realisiert. Er sperrt alle Leser solange, bis alle Schreiber fertig sind.

4. Durch einen Semaphor `mutex2` wird erreicht, dass sich alle Leser nacheinander um den Eintritt in den kritischen Abschnitt bewerben. Bei einer aktuellen Wettbewerbssituation zwischen mehreren Lesern und einem Schreiber wird durch `mutex2` sichergestellt, dass der Schreiber im ungünstigen Fall einem Leser den Vortritt lassen muss. Es darf daher nur *ein* Leser aufgrund von `mutex1` warten, während sich eventuell vorhandene zusätzliche Leser in die Warteschlange des Semaphors `mutex2` einreihen, bevor sie aufgrund von `readsem` warten. Wenn also ein Leser kommt, muss er sich zunächst in die Warteschlange von `mutex2` stellen und wartet darauf, dass er sich in die Warteschlange von `mutex1` stellen kann. Es gibt höchstens einen Leser in der Warteschlange von `mutex1`. Wenn sich aber der erste Schreiber meldet, verhindert er zunächst, dass weitere Leser in die Warteschlange von `mutex1` gelangen, und er braucht nur auf den Leser in der Schlange von `mutex1` zu warten.

```
var readcount, writecount : integer;
    db, readsem, writesem, mutex1, mutex2 : semaphor;

    readcount := 0; writecount := 0;
    db : = 1; readsem := 1; writesem := 1; mutex1 := 1; mutex2 := 1;


Leser-Prozess:
    repeat
       down (mutex2);      (* nimmt Leser auf *)
            down(mutex1); (* stellt sicher, dass nur ein Leser auf
                             mutex1 warten muss *)
                 down(readsem);
                 readcount := readcount + 1;
                 if (readcount = 1) then down(db);
                 up(readsem);
            up(mutex1);
       up(mutex2);
       Lesen Daten;
       down(readsem);
            readcount := readcount - 1;
            if (readcount = 0) then up(db);
       up(readsem);
    until false


Schreiber-Prozess:

    repeat
       down(writesem);       (* sichert den exklusiven Zugriff
                                auf writecount *)
       writecount := writecount + 1; (* ein Schreiber mehr *)
       if (writecount = 1) then
          down(mutex1);    (* keine Leser dürfen mehr zugreifen *)
       up(writesem);        (* gibt den exklusiven Zugriff auf
                               writecount frei *)
       down(db);            (* sichert den exklusiven Zugriff
                               auf den Datenbereich *)
       Schreiben die Daten;
       up(db);              (* gibt den exklusiven Zugriff frei *)
       down(writesem);
       writecount := writecount - 1;
       if (writecount = 0) then
          up(mutex1);        (* die Leser können wieder lesen *)
       up(writesem);         (* gibt den exklusiven Zugriff
                                auf Datenbereich frei *)
    until false
```

5.2.4 Nachrichtenaustausch

Bei den bisher betrachteten Verfahren wurde die Synchronisation durch Zugriff auf *gemeinsame Daten* (Synchronisationsvariable, Semaphore) geregelt.

Eine andere Koordinationsmöglichkeit besteht in dem *Austausch von Nachrichten*. Ein Prozess **Sender** erzeugt Nachrichten fester Länge, die ein zweiter Prozess **Empfänger** konsumiert. Der Austausch dieser Nachrichten erfolgt über einen Puffer der Kapazität K, d. h. der Puffer kann K Nachrichten aufnehmen. Die Größe des Puffers bestimmt, wie weit der Sender dem Empfänger vorauseilen darf. Bezeichnet s die Anzahl der vom Sender gesendeten Nachrichten und e die Anzahl der vom Empfänger empfangenen Nachrichten, so gilt:

1. $0 \leq e \leq s$
2. $0 \leq s - e \leq K$.

Der Nachrichtenaustausch beschreibt in äquivalenter Weise das bereits bisher beschriebene Synchronisationsproblem. Der Sender kann dann eine Nachrichten in den Puffer hinlegen, wenn noch Plätze im Puffer sind. Der Empfänger kann an einer bestimmten Stelle seines Ablaufs - etwa dort, wo eine Nachricht empfangen/eingelesen werden soll - nicht fortfahren, wenn die Nachricht noch nicht im Puffer abgelegt worden ist. Er wird also warten müssen, bis der Sender durch ein Signal dem Empfänger mitteilt, dass die Nachricht zur Verfügung steht. Die **wait-Operation** des Empfängerprozesses ist mit der `down`-Operation vergleichbar, wie das **signal** der `up`-Operation bei Semaphoren entspricht. Da der Senderprozess dem Empfängerprozess um bis zu K Nachrichten voraus sein darf, kann der Empfängerprozess an der fraglichen Stelle seines Ablaufs fortfahren, ohne zu warten, wenn die zu empfangende Nachricht bereits eingetroffen ist.

Man kann aber umgekehrt auch dem Empfängerprozess eine Vorgabe $L \geq 0$ einräumen, in dem z. B. das betrachtete System mit gefülltem Puffer gestartet wird, die es dem Empfängerprozess erlaubt, bis zu L-mal ohne Warten seinen Ablauf fortzusetzen.

Ein Beispiel für die Anwendung des Nachrichtenaustauschs in der Praxis ist der sogenannte **Ringpuffer**, dessen letzter Platz dem ersten vorausgeht, siehe Abbildung 5.14.

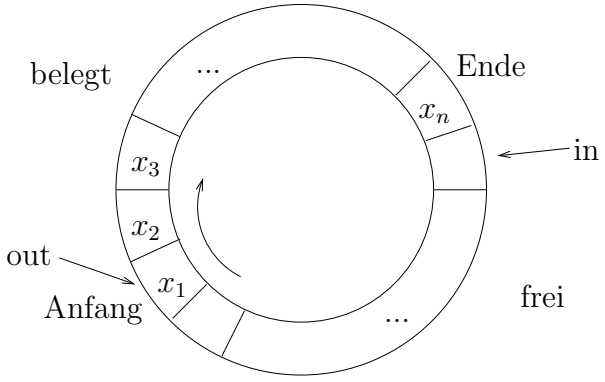


Abbildung 5.14: Ringpuffer.

Der Puffer habe K Plätze und wird in einer festgelegten Richtung belegt und nach Entnahme der Nachrichten wieder freigegeben. Der Zeiger `in` markiert den ersten freien Platz und `out` bezeichnet den Beginn des belegten Bereichs. Zur Implementierung der Kommunikationsaufgabe werden Variablen und drei Semaphore benutzt:

- `in`: erster freier Platz,
- `out`: erster belegter Platz,
- `g`: Anzahl der belegten Plätze,
- `ausschluss`: Semaphor zum Schutz von `in`, `out` und `g`,
- `leer`: Semaphor, um die Anzahl der noch freien Plätze zu verfolgen.
- `voll`: Semaphor, um zu garantieren, dass der Empfänger nicht auf den leeren Puffer zugreift.

Sender- und Empfängerprozess können dann wie in Abbildung 5.15 angegeben beschrieben werden.

```
var puffer : array [0..K − 1] of integer;
var in, out, g : integer;
var ausschluss, leer, voll : semaphor;
in := 0; out := 0; g := 0;
ausschluss :=1; leer:= K; voll:=0;
```

Sender-Prozess

```
repeat
   Erzeuge zu sendende
   Nachricht;
   down(leer);
   down(ausschluss);
   puffer[in] := zu
      sendende Nachricht;
   in := (in + 1) mod K;
   g := g + 1;
   up(ausschluss);
   up(voll);
until false;
```

Empfänger-Prozess

```
repeat
   down(voll);
   down(ausschluss);
   Empfangene Nachricht :=
       puffer[out];
   out := (out + 1) mod K;
   g := g − 1;
   Verbrauche empfangene
   Nachricht;
   up(ausschluss);
   up(leer);
until false;
```

Abbildung 5.15: Der Ablauf von Sender- und Empfänger-Prozessen.

Eine verallgemeinerte Form der Kommunikation und damit auch der Synchronisation von Prozessen mittels Nachrichtenaustausch ist das sog. **Briefkasten-Prinzip (mailbox system)**. Wenn Prozess A mit Prozess B kommunizieren will, so wird ein Briefkasten eingerichtet, der die beiden Prozesse miteinander verbindet und zum Nachrichtenaustausch benutzt werden kann. Prozess A legt im Briefkasten eine Nachricht ab, die Prozess B zu einem späteren Zeitpunkt abholt, der ausschließlich von B bestimmt wird. Man unterscheidet **Einweg-Kommunikation**, d. h. Prozess

A erzeugt nur Nachrichten und Prozess B vebraucht nur Nachrichten, siehe Abbildung 5.15, und **Zweiweg-Kommunikation**. Bei letzterer wird vom Senderprozess A eine Bestätigung (acknowledgment) durch den Empfängerprozess B erwartet, dass dieser auch die gesendete(n) Nachricht(en) korrekt und vollständig empfangen hat. Um Probleme - insbesondere bei mehreren aufeinanderfolgenden Nachrichten - zu vermeiden, wird in der Regel der gleiche Briefkasten sowohl für die Nachricht als auch für die Bestätigung derselben verwendet. Wenn eine Nachricht gesendet wurde, so wird der dafür verwendete Briefkasten so lange reserviert, bis die entsprechende Bestätigung empfangen wurde.

Übungsaufgabe 5.3

1. In je einer zyklischen Sende- und Empfangsprozedur modelliere man einen Nachrichtenübertragungsvorgang, wobei die vom Sender gesendete Nachricht jeweils erst vom Empfänger bestätigt werden soll, bevor die nächste Nachricht gesendet wird. Mögliche auftretende Fehler sollen bei der Lösung unberücksichtigt bleiben. Mit insgesamt vier Semaphoren soll eine für Sender und Empfänger symmetrische Lösung angegeben werden. Wie müssen die Semaphore initialisiert werden, damit vermieden wird, dass der Sender Nachrichten sendet, bevor der Empfänger dazu bereit ist?

2. Man schreibe einen Algorithmus, der simultan Eingabe, Verarbeitung und Ausgabe einer Datenmenge t unter Benutzung dreier Puffer A, B und C wie folgt realisiert:

   ```
   Schritt 1: Eingabe (A)
   Schritt 2: Verarbeitung (A), Eingabe (B)
   Schritt 3: Ausgabe (A), Verarbeitung (B), Eingabe (C)
   Schritt 4: Ausgabe (B), Verarbeitung (C), Eingabe (A)
   usw.
   ```

   Überlappende Eingabe, Verarbeitung und Ausgabe der entsprechenden Puffer sollen durch die Anweisungen

   `Eingabe (`A`) // Verarbeitung (`B`) // Ausgabe (`C`)` ausgedrückt werden. Die Boole'sche Variable `weiter` ist wahr, wenn die Datenmenge noch weitere Elemente enthält. Die anzugebende Lösung sollte auch für den Fall brauchbar sein, dass die Datenmenge t nur 0, 1 oder 2 Elemente enthält.

3. Um welchen maximalen Faktor G kann die Ausführungszeit bei dieser dreifachen Pufferung reduziert werden, wenn man sie mit dem sequentiellen Ablauf vergleicht? Zusätzlicher Aufwand zur Verwaltung des dreifachen Puffers bleibe außer Betracht.

5.2.5 Monitore

Obwohl Nachrichtenpuffer (bzw. Briefkästen) eine übersichtliche Form für die Kommunikation zwischen Prozessen darstellen, ist jedoch die Implementierung kritischer Abschnitte mit diesem Prinzip nicht ganz einfach. Hauptnachteile aus softwaretechnischer Sicht sind, dass Semaphore globale Variablen sind und dass zueinandergehörige `down`- und `up`-Operationen beliebig im Quelltext verstreut sein können, es ist schwierig, den Überblick zu wahren. Synchronisation kann auch dadurch herbeigeführt werden, indem ein *spezieller Prozess* damit beauftragt wird, den Zutritt zu kritischen Abschnitten zu kontrollieren. Ein Prozess, der einen kritischen Abschnitt zu betreten wünscht, sendet eine Nachricht an den genannten speziellen Prozess und wartet auf dessen Bestätigung zur Erlaubnis, den gewünschten kritischen Abschnitt auch ausführen zu dürfen. Der spezielle Prozess achtet darauf, dass diese Erlaubnis *nur einem* Anforderer zu einer Zeit gewährt wird. Diese Überlegung führt zu dem Konzept des Monitors[4] [Di71, Ho74, Br75].

Unter einem **Monitor** versteht man eine Menge von Prozeduren, Variablen, und Datenstrukturen, die als Betriebsmittel betrachtet und mehreren Prozessen zugänglich sind. Die gemeinsam genutzten Betriebsmittel können geschützt werden, indem sie im Monitor platziert werden. Auf die internen Variablen und Datenstrukturen können *nur* die Prozeduren des Monitors und *keine externe* Prozeduren direkt zugreifen. Es kann jeweils nur *ein* Prozess zu einer Zeit im Monitor aktiv sein. Ein Prozess betritt den Monitor, indem er eine seiner Prozeduren aufruft.

Ein Monitor folgt syntaktisch dem Schema

```
monitor monitorname
     Datendeklarationen; (* Gemeinsame Daten *)
     procedure prozedurname1(parameter) { prozedurkörper }
     procedure prozedurname2(parameter) { prozedurkörper }
     .
     .
     .
     procedure prozedurnameN(parameter) { prozedurkörper }
     begin
       Initialisierung;
     end monitor;
```

Ein Monitor unterstützt die Synchronisation durch die Verwendung von *Bedingungsvariablen*, die im Monitor enthalten sind und auf die nur von Methoden innerhalb des Monitors zugegriffen werden kann. Zu jeder Bedingungsvariable gehört eine Warteschlange von Prozessen, die darauf warten, dass die Variable wahr wird. Zwei Funktionen wirken auf die Bedingungsvariablen:

1. `wait`(c): Der aufrufende Prozess wird blockiert, wenn die Bedingung c nicht erfüllt ist. Der Prozess wird in die Warteschlange der Bedingungsvariable c aufgenommen. Unter den Prozessen, die den Monitor betreten wollen, wird einer ausgewählt, der in den Zustand bereit geht. Existiert kein solcher Prozess, dann wird der Monitor freigegeben.

[4] Dieser Monitor hat nichts mit einem Bildschirm zu tun.

2. `signal`(c): Wenn in einer Monitorprozedur die Bedingung c wahr wird, dann wird die `signal`-Operation aufgerufen. Dadurch soll bei nichtleerer Schlange der Bedingungsvariablen wenigstens ein Prozess aktiviert werden. Da nur ein Prozess zu einem Zeitpunkt in einem Monitor rechnen darf, schlägt Hoare vor, dass der signalisierende Prozess den Monitor an einen signalisierten Prozess abgibt, der seine Berechnung im Monitor fortsetzen kann. Deshalb muss der signalisierende Prozess den Monitor verlassen und sich wieder um erneuten Zutritt bewerben.

Das Konzept eines Monitors ist vergleichbar mit einem Raum, siehe Abbildung 5.16, zu dem es nur einen einzigen bewachten Eingang gibt, so dass sich immer jeweils nur ein Prozess im Raum befinden kann. Andere Prozesse, die versuchen, in den Raum einzutreten, gelangen in eine Warteschlange für blockierte Prozesse, die auf die Verfügbarkeit des Monitors warten. Sobald ein Prozess in den Raum eingetreten ist, kann er sich selbst auf der Grundlage der Bedingung c blockieren, indem er die Funktion `wait`(c) aufruft. Er gelangt dann in eine Warteschlange für Prozesse, die auf die Änderung der Bedingung warten, um wieder in den Monitor einzutreten. Wenn ein Prozess, der im Monitor läuft, eine Änderung der Bedingungsvariablen c bemerkt, gibt er das Signal `signal`(c) aus, das die entsprechende Warteschlange darüber informiert.

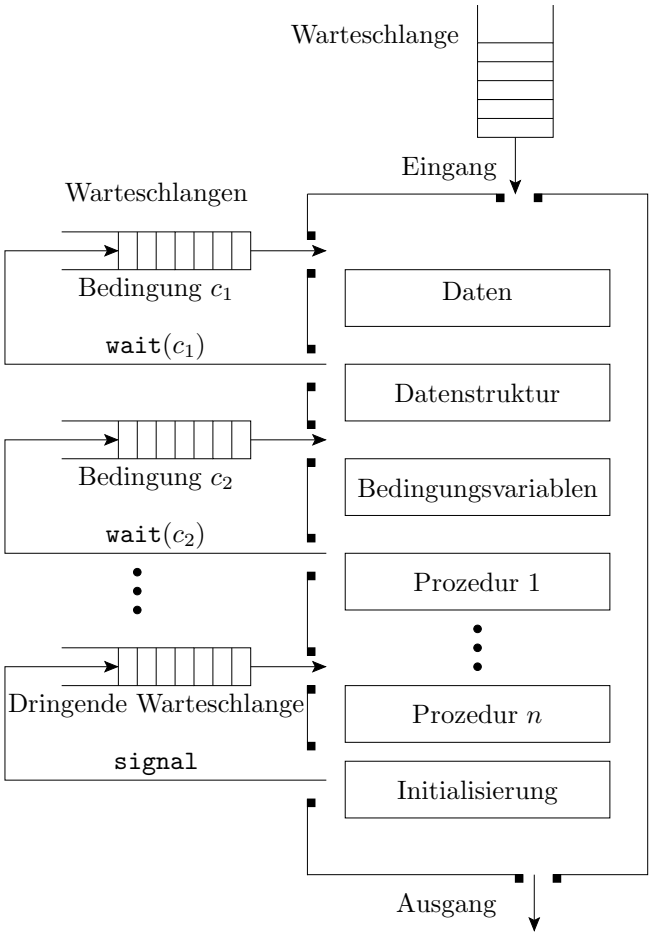


Abbildung 5.16: Aufbau eines Monitors.

Die Implementierung der `down`- und `up`-Operationen aus Abschnitt 5.2.3 lässt sich mit Hilfe eines Monitors schreiben, siehe Abbildung 5.17.

```
monitor Semaphor
  sem_positiv : condition;
  s : integer;

  procedure down();
    begin
      s := s - 1;
      if s < 0 then wait(sem_positiv);
    end;

  procedure up();
    begin
      s := s + 1;
      if s <= 0 then signal(sem_positiv);
    end;

 begin
   s := n;   (* Initialisierung *)
 end monitor;
```

Abbildung 5.17: Realisierung von Semaphor durch ein Monitor.

Die Benutzung des Monitors durch Prozesse erfolgt einfach durch Aufrufe der Form `Semaphor.down()` bzw. `Semaphor.up()`.

Als einfaches Beispiel für einen Monitor wählen wir das Problem des Nachrichtenaustausches mit einem Puffer fester Größe. Der Monitor `sendrecieve` kontrolliert die Zugriffsfunktionen `insert` und `remove` des Puffers, der die Nachrichten speichert. Der Monitor hat zwei Bedingungsvariablen: `notfull` und `notempty`. Wenn noch Platz im Puffer vorhanden ist, gilt `notfull`. Wenn es mindestens eine Nachricht im Puffer gibt, gilt `notempty`, siehe Abbildung 5.18.

Ein Sender kann nur mit Helfe von `insert` eine Nachricht im Puffer ablegen, während ein Empfänger mittels `remove` eine Nachricht aus dem Puffer holen kann. Weder Sender noch Empfänger haben direkten Zugriff auf den Puffer, siehe Abbildung 5.19.

Bei Einsatz von Monitoren sorgt die Monitorstruktur selbst für die Durchsetzung des wechselseitigen Ausschlusses. Im Fall von Semaphoren liegt die Verantwortung für den wechselseitigen Ausschluss und die Synchronisierung beim Programmierer. Der Vorteil, den Monitore im Vergleich zu Semaphoren haben, besteht darin, dass sämtliche Synchronisierungsfunktionen auf den Monitor beschränkt sind. Daher ist es einfacher, zu überprüfen, ob die Synchronisation korrekt vorgenommen wurde, und Fehler festzustellen. Wie auch schon bei Semaphoren ist auch beim Monitoring genau darauf zu achten, dass keine Deadlocks entstehen können, siehe Abschnitt 5.3.

Weitere insbesondere sprachliche Konzepte wie z. B. **Guarded Commands** und **Guarded Regions** erleichtern die Formulierung von Kommunikations- und Synchronisationsaufgaben, siehe [Br78, Di75]. Guarded Commands versetzen einen Pro-

```
monitor sendrecieve;
  buffer : array [0..N-1] of news;
  in, out : integer;
  count : integer;
  notfull, notempty : condition;

  procedure insert(news x);

  begin
     if (count = N) then
        wait(notfull);      (* Puffer ist voll *)
     buffer[in] := x;
     in := (in + 1) mod N;
     count := count + 1;
     signal(notempty);      (* eine Nachricht mehr* )
  end

  procedure remove(news x);

  begin
     if (count = 0) then
        wait(notempty);     (* Puffer ist leer *)
     x := buffer[out];
     out := (out +1) mod N;
     count := count - 1;
     signal(notfull);
  end

  begin
    in := 0; out := 0; count := 0; (* Initialisierung *)
  end monitor;
```

Abbildung 5.18: Nachrichtenaustausch durch Monitor.

zess in die Lage, eine beliebige Wahl zur Ausführung des zeitlich nächsten Ablaufs zu treffen. Diese Auswahl wird lediglich durch eine Inspektion der Zustandsvariablen für die alternativen Ablaufteile bestimmt. Während eine Guarded Region ähnlich wie ein Monitor einen Prozessablauf blockieren kann, ist dies jedoch bei Guarded Commands nicht möglich. Für weitere Einzelheiten sei an dieser Stelle auf die angegebene Originalliteratur verwiesen.

```
procedure sender();
begin
   repeat
      Erzeugen Nachricht x;
      sendrecieve.insert(x);  (* Aufruf Monitor-Prozedur *)
   until false;
end

procedure reciever();

begin
   repeat
      sendrecieve.remove(x);  (* Aufruf Monitor-Prozedur *)
      Verbrauchen Nachricht x;
   until false
end
```

Abbildung 5.19: Aufruf von Monitor-Prozeduren.

5.3 Deadlocks – Systemverklemmungen

In modernen Betriebssystemen treten zahlreiche konkurrent ablaufende Aufgaben auf, die sich in vielfältiger Weise gegenseitig beeinflussen. Beispielsweise bewerben sich zeitlich verzahnt mehrere Prozesse um verschiedene Betriebsmittel, z. B. Prozessor, Hauptspeicher, E/A-Einheiten, Dateien, Übersetzungsprogramme u.s.w., die, nach der Reihenfolge der Anforderung zugeteilt, häufig zu Zuständen des Gesamtsystems führen können, so dass alle bzw. ein Teil der am System beteiligten Prozesse sich gegenseitig blockieren.

Ein Prozess muss manchmal auf ein gewisses Ereignis warten. Wenn das Ereignis auftritt, kann der Prozess seinen Ablauf fortsetzen. Wenn jedoch das Ereignis nicht auftritt, ist der Prozess unter Umständen beliebig lange blockiert.

Wir bezeichnen solche Zustände des Systems als inkonsistent - das System ist verklemmt, es befindet sich in einem **Deadlock** (auch **Interlock** genannt).

Deadlocks treten nicht nur in Rechnersystemen, sondern ganz allgemein in dynamischen Systemen auf. Zwei weitere Beispiele sollen zur Verdeutlichung dienen.

1. Verkehrsstau: Die Vorfahrtsregelung in einem Kreisverkehr sehe die Regelung *Rechts vor Links* vor. Durch diese Regelung können beliebig viele Fahrzeuge in den Kreisverkehr einfahren. Das wird allerdings nur solange möglich sein, als auch entsprechend viele Fahrzeuge den Kreisverkehr wieder verlassen. Geschieht dies nicht, so kommt es zu einem Verkehrsstau, siehe Abbildung 5.20, der nur durch eine Umkehr der oben angegebenen Regel wieder aufgelöst werden kann.

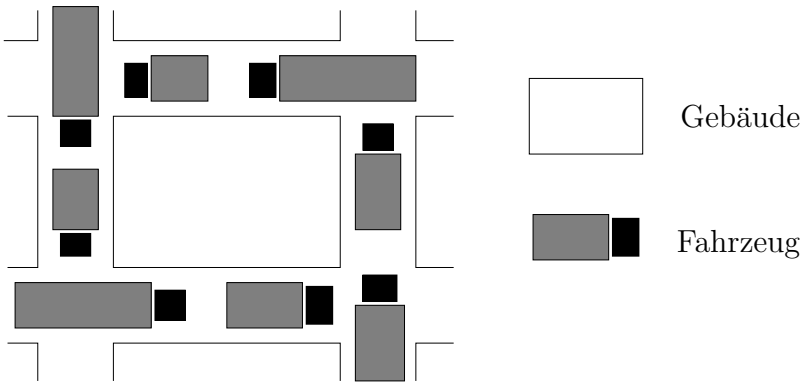


Abbildung 5.20: Verkehrsstau.

2. Buchausleihe: Zwei Studenten A und B haben je ein Buch a bzw. b entliehen. Beide Studenten finden beim Studium ihrer Bücher a bzw. b Referenzen auf die Bücher b bzw. a. Jeder behält das ursprünglich entliehene Buch, um darauf zu warten, zunächst die angegebene Referenz einsehen zu können bevor er in seinem zuerst entliehenen Buch weiterliest.

Student A	Student B
entleiht Buch a	entleiht Buch b
liest	liest
benötigt Buch b	benötigt Buch a
wartet	wartet

Beide Beispiele sind charakteristisch für den Umstand, dass die bestehende Verklemmung ohne einen radikalen Eingriff von außen nicht aufgelöst werden kann.

Bei den angegebenen Beispielen – und auch den bereits früher angeführten Fällen – konnte es zu einer Systemverklemmung nur deshalb kommen, weil gleichzeitig jede der folgenden vier **Deadlock-Bedingungen** erfüllt war:

1. *Wechselseitiger Ausschluss*: Die Prozesse fordern und erhalten exklusive Kontrolle über die benötigten Betriebsmittel. Also kann zu einem Zeitpunkt immer jeweils nur ein Prozess ein Betriebsmittel nutzen.

2. *Nichtunterbrechbarkeit (no preemption)*: Die Betriebsmittel können temporär nicht zurückgegeben werden, sondern bleiben dem Prozess bis zum Ende der Anforderung zugeordnet.

3. *Hold-and-wait-Bedingung*: Die Prozesse belegen (exklusiv) bereits zugewiesene Betriebsmittel, während sie noch auf zusätzliche Betriebsmittel warten.

4. *Zyklische Warte-Bedingung (circular wait)*: Es besteht eine geschlossene Kette aus Prozessen und von ihnen belegten bzw. angeforderten Betriebsmitteln in der Weise, dass jeder Prozess ein oder mehrere Betriebsmittel belegt, die vom nächsten Prozess in der Kette benötigt werden, siehe Abbildung 5.21.

Wichtig ist, dass wirklich alle vier Bedingungen für einen Deadlock notwendig erfüllt sein müssen: Sobald nur eine der Bedingungen nicht erfüllt ist, kommt es nicht zum Deadlock.

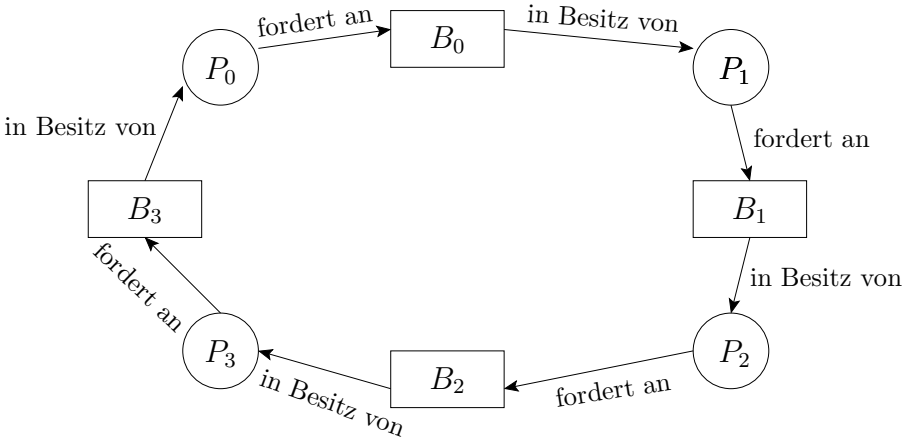


Abbildung 5.21: Prozess i fordert Betriebsmittel i, aber Prozess $i+1$ besitzt Betriebsmittel i mit $0 \leq i \leq 3$ und $P_4 = P_0$.

Die vier Bedingungen sollen am Beispiel des Verkehrsstaus illustriert werden. Betriebsmittel ist der Platz, den ein Fahrzeug im Kreis belegt. Bedingung 1 gilt, weil zu einer Zeit nur ein Fahrzeug an einer Stelle stehen (fahren) kann. Ein belegtes Straßenstück kann nicht vorübergehend einem anderen Fahrzeug zur Verfügung gestellt werden, ohne dass das auf diesem Stück stehende Fahrzeug weiterfährt, also ist Bedingung 2 erfüllt. Solange ein Auto nicht freie Fahrt erhält, wird das nachfolgende oder in den Kreis einfahrende Fahrzeug warten müssen, d. h. Bedingung 3 gilt. Bedingung 4 ergibt sich schließlich aus der zyklischen Anordnung des Kreisverkehrs.

Bei der Behandlung inkonsistenter Systemzustände werden drei prinzipiell verschiedene Vorgehensweisen angewendet:

1. *Erkennung und Beseitigung von Systemverklemmungen (deadlock detection)*: Die benutzten Methoden versuchen, möglicherweise aufgetretene Deadlocks festzustellen und dann durch einen Eingriff in das System aufzulösen. Zunächst versucht man, die an einem Deadlock beteiligten Prozesse und Betriebsmittel herauszufinden, dann werden die Prozesse abgebrochen und die zur Beseitigung des Deadlocks erforderlichen Betriebsmittel freigegeben. Der Nachteil dieser Vorgehensweise ist der vorzeitige Abbruch einzelner Prozesse, die später auch nicht mehr an der abgebrochenen Stelle fortgesetzt werden können.
2. *Vermeidung von Systemverklemmungen (deadlock avoidance)*: Die mit diesem Ziel arbeitenden Verfahren benutzen Informationen über zukünftige Betriebsmittelanforderungen und versuchen, nur solche Betriebsmittelvergaben zuzulassen, bei denen nicht die Gefahr einer späteren Systemverklemmung besteht.
3. *Verhinderung von Systemverklemmungen (deadlock prevention)* Diese Strategie versucht, zumindest eine der vier Deadlock-Bedingungen niemals erfüllen zu lassen.

Natürlich sind die Verfahren zur Vermeidung und Verhinderung[5] von Systemverklemmungen eindeutig den Verfahren zur Erkennung und Beseitigung überlegen.

[5]Die Begriffe *avoidance* und *prevention* werden in der englischsprachigen Literatur über Betriebssysteme in dem genannten Sinne einheitlich verwendet: Leider gilt das nicht für die deutschen Übersetzungen, wo manchmal die beiden Begriffe *Vermeidung* und *Verhinderung* vertauscht werden.

Wir werden allerdings die Schwierigkeit bei den einzelnen Methoden gleich kennen lernen.

5.3.1 Erkennung und Beseitigung eines Deadlocks

Hier werden zunächst nach Möglichkeit alle Anforderungen zu benötigen Betriebsmitteln erfüllt. Das Betriebssystem führt regelmäßig einen Algorithmus zur Erkennung von Deadlocks aus, der das zyklische Warten (Bedingung 4) erkennt.

Wir nehmen an, dass es n Prozesse $P_1, \ldots, P_n$ und m Klassen von Betriebsmitteln gibt. Zu jedem Zeitpunkt wird der aktuelle *Zustand* zur Belegung der Betriebsmittel von Prozessen durch die folgenden Informationen definiert.

1. Der Betriebsmittelvektor $B = (b_1, \ldots, b_m)$ gibt die Anzahl der Betriebsmittel an, die von jeder Klasse insgesamt verfügbar sind.

2. Der Betriebsmittelrestvektor $R = (r_1, \ldots, r_n)$ gibt die Anzahl Betriebsmittel an, die von jeder Klasse noch frei sind.

3. Die Belegungsmatrix

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{pmatrix}$$

   gibt an, wie viele Betriebsmittel aktuell von den Prozessen belegt sind. Hier ist also c_{ij} die Anzahl der Betriebsmittel der Klasse j, die Prozess i belegt.

4. Die Anforderungsmatrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

   gibt an, wie viele Betriebsmittel zur Zeit angefordert werden. Hier ist also a_{ij} die Anzahl der Betriebsmittel der Klasse j, die Prozess i gerne haben möchte.

Es gilt für $j = 1, \ldots, m$ immer

$$r_j = b_j - \sum_{i=1}^{n} c_{ij}.$$

Der Algorithmus zur Erkennung von Deadlocks ist in Abbildung 5.22 dargestellt. Er sucht nach einem Prozess, dessen Anforderungen durch die momentan verfügbaren Betriebsmittel erfüllt werden können. Im vorderen Teil des Algorithmus werden zunächst die Variablen initialisiert. Der wesentliche Teil des Algorithmus ist die `repeat`-Schleife. Bei jedem Durchlauf werden alle Prozesse, die potentiell noch im Deadlock enthalten sind, daraufhin untersucht, ob ihre Restanforderung durch die noch vorhandenen Betriebsmittel erfüllt werden kann. Falls ein solcher Prozess gefunden wird, wird dieser als beendbar markiert, und die von ihm belegten Betriebsmittel können freigegeben werden. Die `repeat`-Schleife wird so lange wiederholt,

```
var n, m, i, j : integer;
    r : array[1..m] of integer;
    c, a : array[1..n,1..m] of integer;
    beendbar : array[1..n] of boolean;
    noch_einer_beendbar, deadlock : boolean;

for i := 1 to n do begin
   beendbar[i] := false;
end;

repeat
   noch_einer_beendbar := false;
   for i := 1 to n do begin
      if not beendbar[i] then begin
         beendbar[i] := true;
         for j := 1 to m do
            if a[i,j] > r[j] then beendbar[i] := false;
         if beendbar[i] then begin
            noch_einer_beendbar := true;
            for j := 1 to m do
               r[j] := r[j] + c[i,j];
         end;
      end;
   end;
until (not noch_einer_beendbar);

deadlock := false;
for i := 1 to n do
   if not beendbar[i] then deadlock := true;
```

Abbildung 5.22: Algorithmus zur Erkennung von Deadlocks.

wie weitere beendbare Prozesse gefunden werden. Nur wenn alle Prozesse als beendbar markiert sind, dann liegt keine Verklemmung vor, ansonsten bilden die nicht beendbaren Prozesse einen Deadlock-Zyklus.

Was kann man nun tun, wenn ein Deadlock erkannt wurde? Die folgenden Ansätze für eine manuelle oder teilautomatische Lösung sind möglich, wobei keiner wirklich in jeder Beziehung zufriedenstellend und praktikabel ist.

1. Abbruch aller verklemmten Prozesse. Das ist eine weit verbreitete Lösung.

2. Schrittweiser Abbruch von verklemmten Prozessen, bis kein Deadlock mehr existiert. Die Reihenfolge der abzubrechenden Prozesse sollte so festgelegt werden, dass die Kosten minimiert werden.

3. Alle Prozesse schreiben in regelmäßigen Abständen (an so genannten Checkpoints) ihren Zustand in eine Datei. So können (hoffentlich) alle verklemmten Prozesse an einen Checkpoint zurück- und von dort aus weitergeführt werden.

4. Schrittweiser Entzug von Betriebsmitteln, bis kein Deadlock mehr da ist, und Zurückführung der betroffenen Prozessen an Checkpoints, Weiterführung von dort aus.

5.3.2 Vermeidung von Deadlocks

Beschäftigen wir nun mit der Frage, ob man Deadlocks zuverlässig vermeiden kann, indem Ressourcen nur dann zugeteilt werden, wenn dies garantiert sich ist. Solche Verfahren benötigen dann zusätzliche Informationen von Prozessen darüber, welche Betriebsmittel zukünftig noch angefordert werden. Bei jeder Anforderung muss nun feststellt werden, ob der Prozess blockiert wird, obwohl das Betriebsmittel zur Verfügung steht, nur um eine möglich zukünftige Verklemmung zu vermeiden. Es geht also darum, das Scheduling sicher zu gestalten.

Der **Bankier-Algorithmus** wurde von Dijkstra 1965 veröffentlicht und ist eine Erweiterung des Algorithmus zur Erkennung von Deadlocks, siehe Abschnitt 5.3.1. Er basiert auf dem Konzept der sicheren Zustände zur Belegung von Betriebsmitteln. Zu jedem Zeitpunkt wird der Zustand durch verfügbare Restbetriebsmittel R und die Belegungsmatrix C definiert. Außerdem muss jeder Prozess seine *maximalen Anforderungen* an allen Betriebsmitteln vorab bekannt geben, diese sind in der maximalen Anforderungsmatrix

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1m} \\ m_{21} & m_{22} & \cdots & m_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ m_{n1} & m_{n2} & \cdots & m_{nm} \end{pmatrix}$$

gespeichert. Hier ist also m_{ij} die maximale Anzahl der Betriebsmittel der Klasse j, die Prozess i belegen kann.

Ein Zustand ist sicher, wenn es mindestens eine Ausführungsreihenfolge der Prozesse gibt, die nicht zu einer Verklemmung führt, selbst wenn alle Prozesse sofort ihre maximale Anzahl an Betriebsmitteln anfordern. Das System ist in einem sicheren Zustand gestartet. Wenn ein Prozess Betriebsmittel anfordert, wird getestet, ob der Zustand durch diese Zuteilung unsicher würde. Falls ja, bleibt das System im sicheren Zustand und blockiert den Prozess so lange, bis es sicher ist, die Anforderung zu gewähren.

Der Bankier-Algorithmus ist fast gleich zu dem Algorithmus zur Erkennung von Deadlocks in Abbildung 5.22, nur haben wir anstelle von der Anforderungsmatrix A jetzt die maximale Anforderungsmatrix M und anstelle des Tests `a[i,j] > r[j]` den Test `m[i,j] - c[i,j] > r[j]`.

Leider haben wir auch dadurch keine endgültige Lösung des Deadlocksproblems gefunden, denn die maximalen Anforderungen eines Prozesses sind kaum im Voraus verfügbar.

5.3.3 Verhinderung von Deadlocks

Nun wollen wir von vornherein keine Deadlocks zulassen, und zwar so, dass zu jedem Zeitpunkt zumindest eine der vier Deadlock-Bedingungen nicht erfüllt ist.

Eine Aufweichung der ersten Deadlock-Bedingung ist im Allgemeinen nicht möglich, denn für viele Anforderungen ist der wechselseitige Ausschluss absolut erforderlich.

Die Wiederherstellung der Unterberechbarkeit (Bedingung 2) könnte durch Ressourcenentzug realisiert werden: Wenn ein Prozess zusätzliche Betriebsmittel anfordert, muss er alle bisher reservierten Betriebsmittel zunächst freigeben und dann zusammen mit den zusätzlichen Betriebsmitteln erneut anfordern. Diese Methode garantiert trivialerweise die Deadlockfreiheit. Aber sie ist nicht immer praktikabel, z. B. man kann nicht einem Prozess ein Ausgabegerät entziehen, auf das er gerade etwas ausgibt.

Ein Ansatz, um ein Eintreten der Hold-and-wait-Bedingung zu verhindern, ist, dass jeder Prozess alle von ihm benötigten Ressourcen gleichzeitig im Voraus anfordern muss. Wenn sie verfügbar sind, dann werden sie ihm zugeteilt, sonst blockiert der Prozess. Die Nachteile des Ansatzes sind, dass er nicht effizient ist und man nicht immer im Voraus sehen kann, wie viele Ressourcen benötigt werden.

Ein Eintreten der zyklischen Warte-Bedingung kann dadurch beseitigt werden, dass eine lineare Ordnung aller Betriebsmittel festgelegt wird. Die Ordnung wollen wir hier mit *wichtiger als* bezeichnen, sowie der Regel, dass dann, wenn schon Betriebsmittel reserviert sind, keine zusätzlichen Betriebsmittel, die wichtiger als die schon reservierten sind, angefordert werden können. Es können nur noch weniger wichtige Betriebsmittel nachgefordert werden. Wenn ein Prozess P also auf die Freigabe von weniger wichtigen Betriebsmitteln wartet, können die Prozesse, die diese Betriebsmittel gerade belegen, höchstens auf noch weniger wichtige Betriebsmittel warten, aber nie auf ein von P belegtes Betriebsmittel. Zyklisches Warten ist somit ausgeschlossen. Der Nachteil des Ansatzes ist, dass man eine *vernünftige* solche Ordnung für *alle* Ressourcen kaum finden kann.

5.3.4 Eine übergreifende Strategie

Wie wir in den vorangegangenen Abschnitten gesehen haben, ist es für Betriebssysteme gar nicht leicht, Deadlocks sicher zu vermeiden oder zu verhindern bzw. wenigstens zu erkennen und sinnvoll automatisch zu behandeln. Deshalb ist in den am weitesten verbreiteten Systemen wie Windows und Linux die so genannte *Vogel-Strauß-Strategie* „implementiert“: Das Problem wird ignoriert. Man setzt darauf, dass die Wahrscheinlichkeit für das Eintreten eines Deadlocks extrem gering ist und dass der Benutzer im Notfall selbst eingreift und z. B. Prozesse beendet oder den Rechner neu startet.

Eine wirkliche, kombinierte Strategie, die die Vor- und Nachteile der genannten Verfahren berücksichtigt, hat Silberschatz [SiG98] vorgeschlagen. Die Ressourcen werden nach ihrer Wichtigkeit in vier Klassen eingeteilt, und Verklemmungen zwischen einzelnen Klassen werden durch diese Ordnung wie in Abschnitt 5.3.3 beschrie-

ben verhindert. Innerhalb jeder Klasse wird das hier am besten geeignete Verfahren angewendet. Die vier Klassen sind:

1. *Interne Ressourcen* wie Prozesskontrollblöcke und Ein-/Ausgabe-Kanäle: Verhinderung durch lineare Ordnung (Abschnitt 5.3.3), die sich hier leicht festlegen lässt.

2. *Hauptspeicher* für Anwenderprozesse: Verhinderung durch Ressourcenentzug (Abschnitt 5.3.3), ein Prozess kann in den Swap-Bereich ausgelagert werden und macht dadurch Hauptspeicherressourcen für andere Prozesse frei.

3. *Prozessressourcen* wie Dateien, Laufwerke und andere Geräte: Die Vermeidungsstrategie (Abschnitt 5.3.2) ist hier sinnvoll, weil man durchaus verlangen kann, dass Prozesse im Voraus festlegen, welche Geräte sie benötigen werden.

4. Der *Swap-Bereich* für die Auslagerung von Prozessen: Verhinderung durch Vorausanforderung (Abschnitt 5.3.3), die maximalen Speicheranfoederungen sollen zum Prozessbeginn bekannt gegeben werden.

5.4 Zusammenfassung

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie in der Lage sein:

- die Notwendigkeit von Prozess-Kommunikation als zentrale Aufgabe von Betriebssystemen anhand praktischer Beispiele zu erläutern
- die bei konkurrenten Prozessen auftretenden Fragen differenziert aufzulisten
- disjunkte und überlappende Prozesse hinsichtlich ihrer Konsequenzen zu unterscheiden
- kritische Abschnitte in konkurrenten Prozessen von unkritischen Abschnitten abzugrenzen und das Problem des gegenseitigen Ausschlusses zu formulieren
- Aufgaben der Synchronisation mit Hilfe expliziter Synchronisationsvariablen zu lösen
- Semaphore von expliziten Synchronisationsvariablen zu unterscheiden
- das Erzeuger-Verbraucher-Problem algorithmisch zu beschreiben
- Kommunikations-Probleme durch Nachrichtenaustausch zu formulieren
- das Prinzip der Synchronisation mittels Monitore anschaulich zu erläutern
- die vier Deadlock-Bedingungen anhand von Beispielen zu demonstrieren
- Verfahren zur Deadlock-Erkennung von solchen zur Deadlock-Vermeidung und Deadlock-Verhinderung zu unterscheiden.

Literatur

[Br75] P. Brinch Hansen. *The Programming Language Concurrent Pascal*. IEEE Trans. on Software Engineering, SE-1: p. 199-207, 1975.

[Br78] P. Brinch Hansen. *Distributed processes: a concurrent programming concept*. Communications of the ACM 21:11, p. 934-941, 1978.

[CoES71] E.G. Coffman, M.J. Elphick, and A. Shoshani. *System Deadlocks*. ACM Computing Surveys 3, p. 67-78, 1971.

[Di68] E.W. Dijkstra. *Cooperating sequential processes*. In F. Genuys, *Programming Languages*, Academic Press, New York, 1968.

[Di71] E.W. Dijkstra. *Hierarchical ordering of sequential processes*. Acta Informatica 1, p. 115-138, 1971.

[Di75] E.W. Dijkstra. *Guarded commands, nondeterminacy, and formal derivation of programs*. Communications of the ACM 18:8, p. 453-457, 1975.

[EiM72] M.A. Eisenberg, and M.R. McGuire. *Further comments on Dijkstra's concurrent programming control problem*. Communications of the ACM 15:11, p. 999, 1972/11.

[Ha72] A.N. Habermann. *Synchronisation of communicating processes*. Communications of the ACM 15:3, p. 171-176, 1972.

[Ho74] C.A.R. Hoare. *Monitors: an operating system structuring concept*. Communications of the ACM 17:10, p. 549-557, 1974.

[Pa71] S.S. Patil. *Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes*. Memo 57, Project MAC, MIT, Cambridge, Mass., 1971.

[Pr75] L. Presser. *Multiprogramming coordination*. ACM Computing Surveys 7, p. 21-44, 1975.

[SiG98] A. Silberschatz, P.B. Galvin. *Operating System Concepts, Fifth Edition*. Addison-Wesley Publishing Company, 1998.

Lösungen

Übungsaufgabe 5.1, Seite 171:

1. Wir betrachten den folgenden Fall, dass `inhalt = 0` ist. Der Verbraucher-Prozess versucht aber, die Waren zu verbrauchen, er führt eine `down`-Operation aus, nun ist `inhalt = -1`. Nachdem der Erzeuger ein Element in den Puffer gelegt hat, führt er eine `up`-Operation aus. Der wartende Prozess wird in den Zustand bereit versetzt, aber zu diesem Zeitpunkt ist `inhalt = 0`. Also stimmt `inhalt` nicht mehr mit der Anzahl der Elemente im Puffer überein. Wenn der Erzeuger-Prozess weitere Elemente erzeugt und der Verbraucher-Prozess aber nicht zur Ausführung kommt, dann zeigt `inhalt` immer eins weniger als die Anzahl der Elemente im Puffer.

2. Die Vertauschung der `up`-Operationen im Erzeuger-Prozess ist für den Ablauf ohne Bedeutung, jedoch führt die Vertauschung der `down`-Operationen im Verbraucher-Prozess zur gegenseitigen Blockade.

Übungsaufgabe 5.2, Seite 172:

1.
```
var voll, leer, zustand: semaphor;
voll := 0; leer := n; zustand := 1;
```

Erzeuger-Prozess
```
repeat
  Erzeuge Ware;
  down(leer);
  down(zustand);
  Bringe Ware nach Puffer;
  up(zustand);
  up(voll);
until false;
```

Verbraucher-Prozess
```
repeat
  down(voll);
  down(zustand);
  Hole Ware aus Puffer;
  up(zustand);
  up(leer);
  Verbrauche Ware;
until false;
```

2. Die Lösung ist bezüglich der beiden Semaphore `leer` und `voll` völlig symmetrisch. Die leeren Positionen des Puffers werden durch `leer` und die vollen durch `voll` beschrieben. Der Erzeuger-Prozess produziert `voll` und verbraucht `leer` und der Verbraucher-Prozess produziert `leer` und verbraucht `voll`.

Übungsaufgabe 5.3, Seite 180:

1.
```
var nachricht, antwort erlaubnis, empfang : semaphor;
nachricht := 0; antwort := 0; erlaubnis := 0; empfang := 0;
```

Sender-Prozess

`repeat`
`  down(erlaubnis);`
`  Senden der Nachricht;`
`  up(nachricht);`
`  down(antwort);`
`  Empfang der Bestätigung;`
`  up(empfang);`
`until false;`

Empfänger-Prozess

`repeat`
`  up(erlaubnis);`
`  down(nachricht);`
`  Empfang der Nachricht;`
`  Senden der Bestätigung;`
`  up(antwort);`
`  down(empfang);`
`until false;`

   Der Semaphor `erlaubnis` signalisiert die Empfangsbereitschaft des Empfängers; mittels der Semaphoren `nachricht` und `antwort` wird die Übertragung der Nachricht bzw. der Empfangsbestätigung synchronisiert und der Semaphor `empfang` soll den korrekten Empfang der Empfangsbestätigung durch den Sender bestätigen. Alle Semphore werden mit 0 initialisiert.

2. Die Variablen `letztes`, `laufendes` und `naechstes` kennzeichnen aufeinanderfolgende Elemente der Datenmenge t, die sich in den 3 Phasen Eingabe, Verarbeitung und Ausgabe befinden:

```
while weiter do begin
   Eingabe(t.naechstes);
   while weiter do begin
      t.laufendes := t.naechstes;
      Verarbeitung(t.laufendes)//Eingabe(t.naechstes);
      while weiter do begin
         t.letztes := t.laufendes;
         t.laufendes := t.naechstes;
         Ausgabe(t.letztes)//Verarbeitung(t.laufendes)
            //Eingabe(t.naechstes);
      end;
      Ausgabe(t.laufendes);
   end;
   Verarbeitung(t.naechstes);
   Ausgabe(t.naechstes);
end;
```

3. Seien E, V und P die für die Eingabe, Verarbeitung bzw. Ausgabe benötigten Zeiten, dann gilt

$$G = \frac{E + V + P}{\max(E, V, P)} \leq 3.$$

Glossar

Briefkasten-Prinzip (*mailbox system*) Form der Kommunikation konkurrenter Prozesse mittels Nachrichtenaustausch.

Erkennung einer Systemverklemmung (*detection of deadlocks*) Feststellung von Systemverklemmungen und Auflösung derselben durch Eingriff in das System.

Erzeuger-Verbraucher-Problem (*producer/consumer problem*) klassische Kommunikationsform zweier konkurrenter Prozesse, bei denen der eine etwas produziert, das der andere Prozess konsumiert.

wechselseitiger Ausschluss (*mutual exclusion*) exklusive Ausführung kritischer Abschnitte in konkurrenten Prozessen.

kritischer Abschnitt (*critical section*) Abschnitte, die lesend und/oder schreibend gemeinsame Daten, die von mehreren konkurrenten Prozessen benutzt werden, verarbeiten.

konkurrente Prozesse (*concurrent processes*) Abläufe simultaner oder paralleler Prozesse.

Monitor (*monitor*) Menge von Prozeduren und Datenstrukturen zur Synchronisation paralleler Prozesse, die als Betriebsmittel betrachtet mehreren Prozessen zugänglich sind, aber nur von einem Prozess zu einer Zeit benutzt werden können.

`down`-Operation (`down` *operation*) Semaphor-Operation, die vor Eintritt in einen kritischen Abschnitt ausgeführt wird.

Wettbewerbssituation (*race condition*) Situationen, in denen mehrere Prozesse einen gemeinsamen Speicher lesen oder schreiben und das Ergebnis davon abhängt, wer wann genau abläuft.

Ringpuffer (*ring buffer*) zyklischer Puffer für die Kommunikation über Nachrichtenaustausch.

Semaphor (*semaphor*) spezielle Synchronisationsvariable, die nur über `down`- bzw. `up`-Operationen getestet bzw. verändert werden darf.

`signal`-Operation (`signal` *operation*) auf Monitore anwendbare Operation; entspricht der `up`-Operation bei Semaphoren.

Systemverklemmung (*deadlock, interlock*) Systemzustand, in dem einer oder mehrere Prozesse blockiert sind und ohne Eingriff von außen ihren Ablauf nicht mehr fortsetzen können.

`up`-Operation (`up` *operation*) Semaphor-Operation, die nach Verlassen eines kritischen Abschnitts ausgeführt wird.

Verhinderung von Systemverklemmungen (*deadlock prevention*) durch nicht Erfüllung zumindest eine der vier Deadlock-Bedingungen kommt überhaupt keine Systemverklemmung vor.

Vermeidung von Systemverklemmungen (*deadlock avoidance*) durch Benutzung von Information über zukünftige Betriebsmittelanforderungen werden Systemverklemmungen gar nicht erst zugelassen.

`wait`**-Operation** (`wait` *operation*) auf Monitore anwendbare Operation; entspricht der `down`-Operation bei Semaphoren.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 6:
Sicherheit

Autor: Udo Kelter

Inhalt

Kurseinheit 6

Sicherheit

6.1 Einführung

Ein Rechner hat in der Regel ein konkret umrissenes Aufgabenfeld, d. h. er soll für bestimmte Benutzer bestimmte Aufgaben erfüllen. Außer für diese geplanten Aufgaben kann ein Rechner aber auch für andere Zwecke eingesetzt werden; dies kann einfach nur unerwünscht sein, einen ernsthaften Schaden anrichten oder gar kriminell sein. Man erwartet daher von einem Rechner und insb. dessen Betriebssystem, dass sie *sicher* sind. Fast alle Betriebssysteme enthalten deshalb an unterschiedlichen Stellen Sicherheitsmechanismen. Deren Sinn und Wirksamkeit ist aber nur dann verständlich, wenn man die damit verfolgten Ziele und die Randbedingungen kennt. Wir beginnen deshalb mit einer etwas breiter angelegten Untersuchung des Problembereichs *Sicherheit* und stellen erst danach einige typische Sicherheitsmechanismen von Betriebssystemen vor.

6.1.1 Eine Taxonomie

Die ganz allgemeine Anforderung, ein Rechner möge sicher sein, ist ähnlich verschwommen und unscharf wie die Forderung nach hoher Benutzungsfreundlichkeit. Was diese generelle Anforderung konkret bedeutet, hängt von vielen Faktoren ab, u. a. von der Art der Benutzung des Rechners, seinem organisatorischen Umfeld, seiner physischen und logischen Architektur usw.; die Erfüllung dieser Forderung wiederum betrifft alle möglichen Bereiche von Hard- und Software, insbesondere auch das Betriebssystem. Unser erster Schritt besteht daher in der Klärung bzw. Festlegung einiger grundlegender Begriffe.

Schäden. Motiviert sind alle Sicherheits- bzw. Schutzmaßnahmen dadurch, dass gewisse Schäden oder Schädigungen eintreten können. Beispiele hierfür sind:

- *Verlust der Datenvertraulichkeit:* Personaldaten, betriebliche Daten oder andere geheimzuhaltende Daten werden von Personen gelesen, die hierzu nicht durch eine explizite Erlaubnis, dank ihrer Stellung o. ä. befugt sind. Der Schaden kann im Verlust der Vertraulichkeit von Information, wodurch die Information als solche wertlos werden kann, Verletzung intellektueller Eigentumsrechte an Daten oder Programmen, Verrat von Geheimnissen, Verletzung der Privatsphäre oder ähnlichem bestehen.

- *Verlust der Datenintegrität:* Daten werden auf unautorisierte Weise verändert, verfälscht oder gelöscht. Der direkte Schaden kann im völligen Verlust der Information oder in der Minderung ihrer Korrektheit oder Präzision bestehen. Der indirekte Schaden kann z. B. im Tod eines Patienten, der ein falsches Medikament bekommt, in der Zerstörung von Maschinen oder Anlagen, die von einem Rechner gesteuert werden, im Verlust von Geld oder anderen wirtschaftlichen Gütern usw. bestehen.

- *Verlust der Systemverfügbarkeit:* Ein Programm beinträchtigt ein *fremdes* gleichzeitig laufendes Progamm oder das Betriebssystem in seiner Funktion oder bringt es gar zum Abbruch. Solche *denial of service* Angriffe gehören zunehmend zum Alltag. Handelt es sich bei dem Computer zum Beispiel um einen Internet-Server, so kann dieser durch das Senden von Millionen Anfragen stillgelegt werden. Der Schaden kann in der verlorenen Rechenzeit, erhöhten Wartezeit anderer Benutzer, Nichtverfügbarkeit des Rechners o. ä. bestehen.

- *Finanzieller Verlust:* Rechenzeit oder an den Rechner angeschlossene Drucker, Datenfernverbindungen, Speichergeräte, Terminals oder andere Ressourcen werden von Personen benutzt, die hierfür nicht zugelassen sind und die die entstehenden Kosten nicht tragen. Der Schaden kann finanzielle Kosten der Rechenzeit oder des Verbrauchsmaterials sein oder in der Verschlechterung oder gar völligen Einstellung der Bedienung der autorisierten Benutzer bestehen.

- *Missbrauch:* Der Rechner wird für kriminelle Handlungen benutzt, z. B. betrügerische Geldüberweisungen, Ausstellung gefälschter Versicherungspolicen, usw.

Es ist sehr wichtig zu erkennen, dass die meisten der oben genannten Schäden auch dann eintreten können, wenn Information manuell verarbeitet wird. Oft werden die gleichen Informationen sowohl manuell wie auch durch Rechner verarbeitet. Beispielsweise können gewisse Personaldaten ausgedruckt und in dieser Form unzulässig weitergegeben werden. Die oben genannten Schäden können also nicht nur dann eintreten, wenn Informationen gerade in einem Rechner verarbeitet werden, sondern auch bei ihrer manuellen Verarbeitung.

Hieraus folgt unmittelbar, dass das Thema Sicherheit nicht allein auf Rechner oder noch enger auf das Betriebssystem eingeschränkt werden darf, sondern dass man insbesondere auch das organisatorische Umfeld mit einbeziehen muss.

Bedrohungen. In den obigen Beispielen traten die Schäden immer als Folge von Operationen mit Dateien oder Prozessen auf. Das Löschen, Kopieren oder Verändern einer Datei ist natürlich nicht prinzipiell schädlich genausowenig wie die entsprechenden Operationen in einer Kartei; erst durch die Ursache oder die veranlassende Person wird ein solcher Vorgang unerwünscht oder schädlich. Unerwünschte oder unerlaubte Vorgänge, speziell Operationen mit Rechnern und den darin gespeicherten Daten, bezeichnen wir als **Bedrohungen**. Beispiele für Bedrohungen sind:

- unzulässige Aktionen von zugelassenen Benutzern des Rechners, z. B. Überschreiten von Kompetenzen usw.; wir gehen hier auf die Motive nicht näher ein; eine ausführliche Darstellung dieses Themas findet sich z. B. in [Pk91]

- Fehlverhalten von Programmen, z. B. Programmierfehler
- Materialfehler bei Speichermedien, Speichergeräten etc.
- Fehler bei Datenübertragungen
- fehlerhafte Eingabe von Daten oder irrtümliche Kommandos
- Aktionen von Eindringlingen, z. B. Hacker, Spione, Viren, Würmer

Sicherheitsmaßnahmen. Sicherheitsmaßnahmen zielen darauf ab, Schäden zu verhindern, das Ausmaß von Schäden zu begrenzen oder eingetretene Schäden zu entdecken. Man beachte, dass eine bestimmte Art von Bedrohung zu mehreren Arten von Schäden führen kann und umgekehrt ein bestimmter Schaden durch verschiedene Arten von Bedrohungen verursacht werden kann.

Manche Sicherheitsmaßnahmen wirken generell gegen gewisse Arten von Bedrohungen, unabhängig von dem möglichen Schaden. Beispiele sind Zugangskontrollen zu verschließbaren Räumen / Schränken, Schlössern an Rechnern, Passwörtern, die das Eindringen in ein System schwierig machen.

Manche Sicherheitsmaßnahmen wirken generell gegen gewisse Arten von Schäden, unabhängig von den möglichen Ursachen des Schadens. So kann man sich teilweise gegen den Verlust von Daten schützen, indem man Kopien anfertigt und diese an einem sicheren Ort aufbewahrt.

Im allgemeinen kann aber jede Kombination von Schäden und Bedrohungen nur durch speziell darauf abgestimmte Sicherheitsmaßnahmen wirksam bekämpft werden.

Sicherheit im engeren Sinn. Wir haben bereits oben gesehen, dass manche Arten der Schäden von völlig unterschiedlichen Bedrohungen verursacht werden können. So kann ein Systemabsturz durch einen Sabotageakt eines Eindringlings oder durch einen Programmierfehler im Betriebssystem verursacht werden. Die Korrektheit von Daten kann durch eine unzulässige Aktion eines Benutzers oder durch einen Programmierfehler eines Applikationsprogramms verlorengehen. Wenn man den Begriff Sicherheit weit fasst, also alle möglichen Bedrohungen einschließt, dann gehören zu diesem Themenbereich u. a..:

- Korrektheit von Programmen, speziell des Betriebssystems, von Netzwerkprotokollen, von Anwendungsprogrammen
- Zuverlässigkeit von Programmen in atypischen Situationen
- Sicherstellung von ausreichender Performance bzw. von Realzeit-Eigenschaften
- Wiederherstellung von Daten nach Medienfehlern oder anderen Störungen
- generelle Sicherstellung der Integrität und Präzision von Daten
- sichere Anwendbarkeit von Benutzungsschnittstellen

Man erkennt unschwer, dass das Thema Sicherheit völlig ausufert und die gesamte Informatik als echte Teilmenge enthält. *Wir schränken hier deshalb den Begriff Sicherheit auf solche Bedrohungen ein, die eine vom Betreiber oder Benutzer des Rechners ungewollte oder unerlaubte Benutzung eines Rechners darstellen.*

Sicherheitsstrategien. Aus der Vielfalt der möglichen Schäden und Bedrohungen ergibt sich eine große Zahl denkbarer Sicherheitsmaßnahmen. Aus Aufwands- und Kostengründen kann aber in einem konkreten Fall immer nur eine *endliche* Menge von Sicherheitsmaßnahmen vorgesehen werden. Die Auswahl hängt ab von:

- dem Ausmaß und der Qualität der möglichen direkten und indirekten Schäden (der Medikamentenverteilungsplan einer Klinik ist anders zu behandeln als der Speiseplan der Kantine)
- der Wahrscheinlichkeit einzelner Bedrohungen
- den Kosten der Sicherheitsmaßnahmen

Eine konkrete Auswahl von Sicherheitsmaßnahmen und ihre organisatorische Einbettung nennt man **Sicherheitsstrategie** (**security policy**). Es ist wichtig, die folgenden Aspekte einer Sicherheitsstrategie strikt zu trennen (vgl. [ITSEC91, St91]):

Sicherheitsziele: Die Sicherheitsstrategie einer Organisation z. B. Betrieb, Behörde o. ä. hat das Ziel, bestimmte Sicherheitsziele zu erreichen, d. h. bestimmte identifizierbare Ressourcen, die sich in der Verfügungsgewalt dieser Organisation befinden, vor identifizierbaren Arten unerlaubter oder ungewollter Benutzung (also Bedrohungen) zu schützen.

Organisatorische Sicherheitsstrategie: Hierbei handelt es sich um eine Menge von Gesetzen, Vorschriften, Regeln und Praktiken, die regulieren, wie eine Organisation die Ressourcen, für die gewisse Sicherheitsziele zu erreichen sind, verwaltet, schützt und verteilt. Hierzu gehören im weiteren Sinne Maßnahmen zur Durchsetzung und Sicherstellung der sorgfältigen Anwendung der Regelungen: so fehlt oft bei Personen, die mit schutzbedürftigen Daten umgehen, jegliches Problembewusstsein und damit auch die Motivation zur Befolgung von Sicherheitsmaßnahmen.

Automatisierte Sicherheitsstrategie: Dies ist eine Menge von Sicherheitsmaßnahmen, Restriktionen und sonstigen Eigenschaften, durch die ein Rechner die Benutzung von Informationen oder anderer Ressourcen verhindert, wenn dadurch organisatorische Sicherheitsstrategien verletzt werden würden.

Man beachte, dass viele Sicherheitsziele besser und billiger durch organisatorische Maßnahmen als durch Sicherheitsmaßnahmen in Rechnern erreicht werden können. So schützt ein stabiles Gehäuse mit einem Schlüsselschalter die Daten in einem Rechner möglicherweise besser vor gewissen unbefugten Zugriffen als ein ausgeklügelter Passwort-Mechanismus.

Darüber hinaus sind manche Sicherheitsmaßnahmen in Rechnern nicht sonderlich sinnvoll, wenn sie nicht durch entsprechende organisatorische Maßnahmen begleitet werden. So macht es wenig Sinn, den Zugriff zu Dateien über Zugriffsrechte zu kontrollieren, wenn die Dateien auf Disketten oder anderen entnehmbaren Datenträgern gespeichert sind, die leicht auf einem anderen Rechner, auf dem die Sicherheitsmaßnahmen nicht realisiert sind, montiert und dort ohne Einschränkungen gelesen werden können.

Die Essenz dieser Überlegungen ist, dass es nicht angebracht ist, allein auf Sicherheitsmaßnahmen in Rechnern zu vertrauen oder umgekehrt für jede Bedrohung Sicherheitsmaßnahmen im Rechner vorzusehen. Dies betrifft insbesondere die Frage, welche Sicherheitsmechanismen in einem Betriebssystem vorhanden sein sollten. Die Sicherheitsmechanismen verursachen i. d. R. Aufwand. Da Betriebssysteme einer breiten Klasse von Anwendungen dienen sollen, wird man Sicherheitsmechanismen vermeiden, die stets Aufwand verursachen, aber nur in einer kleinen Zahl von Anwendungen zur Realisierung der automatisierten Sicherheitsstrategie benötigt werden. Bevor wir allerdings wünschenswerte Sicherheitsmechanismen in Betriebssystemen näher diskutieren können, müssen wir zunächst die spezielle Rolle, die das Betriebssystem bei der Realisierung einer automatisierten Sicherheitsstrategie spielen kann, näher bestimmen.

Primäre und sekundäre Bedrohungen. Wie wir schon oben gesehen haben, sind gewisse Bedrohungen unabhängig vom Einsatz von Rechnern in dem Sinne, dass sie bei beliebigen Rechnern und selbst bei einer manuellen Verarbeitung von Information analog auftreten. Solche Bedrohungen nennen wir **primär**. Die daraus resultierenden Sicherheitsziele nennen wir ebenfalls primär. Beispielsweise kann die Gefahr bestehen, dass eine Datei von einem böswilligen Eindringling gelöscht wird; deshalb muss sie im Rechner geschützt werden. Bei manueller Verarbeitung in einem Karteikasten bestünde analog die Gefahr, dass er gestohlen oder zerstört wird.

Nun entstehen leider durch den Einsatz des Rechners noch zusätzliche **sekundäre** Bedrohungen. Beispielsweise könnte es infolge einer unzureichenden Implementierung der Zugangskontrollen möglich sein, unter einer falschen Benutzeridentifizierung in ein System einzudringen. Dies ist als solches noch kein Schaden, ermöglicht es aber nun einem Eindringling, *richtige* Schäden anzurichten. Durch sekundäre Bedrohungen wird also indirekt die Erreichung primärer Sicherheitsziele gefährdet. Obwohl dieser Zusammenhang oft sehr indirekt ist, müssen die sekundären Bedrohungen mit der gleichen Aufmerksamkeit behandelt werden wie die primären. Beispiele sind:

- gegenseitige Störung parallel ablaufender Programme
- unzureichende Realisierung einer Sicherheitsstrategie infolge falscher Einstellung von Parametern von Sicherheitsfunktionen
- unzureichende Implementierung von Sicherheitsfunktionen, die *technologische Attacken* erlauben
- diverse Bedrohungen bei der Übertragung von Daten in Netzwerken

6.1.2 Realisierung einer automatisierten Sicherheitsstrategie

Ein Rechner im Sinne einer automatisierten Sicherheitsstrategie besteht aus folgenden *Schichten*:

- Applikationsprogramme
- Datenbankmanagementsysteme, Archivierungssysteme oder ähnliche Basissoftware

- Betriebssystem incl. eventuell zugehöriger Dienstprogramme
- Hardware

Sicherheitsfunktionen. Eine **Sicherheitsfunktion** ist eine Menge von zusammengehörigen Schnittstellen und/oder Eigenschaften, die eine Ebene der darüberliegenden Ebene zur Realisierung von Sicherheitsmaßnahmen anbietet[1]. Die Realisierung einer Sicherheitsfunktion durch konkrete Datenstrukturen und Algorithmen nennen wir **Sicherheitsmechanismus**. Ein Beispiel für eine typischerweise vom Betriebssystem angebotene Sicherheitsfunktion ist die Möglichkeit, es einzelnen Benutzern zu erlauben oder zu verbieten, einzelne Dateien zu lesen oder zu schreiben. Aus Sicht der Applikationen besteht diese Sicherheitsfunktion aus gewissen Schnittstellen, durch die die Zugriffsrechte verändert werden können, und der Zurückweisung von unberechtigten Lese- bzw. Schreiboperationen.

Sicherheitsmaßnahmen auf verschiedenen Ebenen. Man beachte, dass auf allen oben aufgeführten Ebenen schutzbedürftige Ressourcen realisiert sein können. Die Programme, die eine Ebene bilden, und die darin enthaltenen Sicherheitsfunktionen können nur solche Ressourcen schützen, die auf dieser Ebene identifizierbar sind, die also auf dieser Ebene realisiert werden oder die auf einer tieferen Ebene realisiert, aber auf dieser Ebene repräsentiert werden. Das Betriebssystem kann z. B. den Zugriff zu einer Datei direkt kontrollieren. Es kann indirekt einen Drucker vor unbefugter Benutzung schützen, wenn es weiß, dass dieser Drucker an einer bestimmten Schnittstelle angeschlossen ist und wenn durch organisatorische Maßnahmen verhindert wird, dass jemand den Drucker an eine andere Schnittstelle anschließt.

Das Betriebssystem bzw. irgendeine Ebene kann keine Ressourcen schützen, die erst auf einer höheren Ebene realisiert werden. So sind einzelne Felder von Datensätzen in einem Datenbanksystem für das Betriebssystem nicht identifizierbar und daher auch nicht schützbar. Ressourcen, die durch Applikationsprogramme realisiert werden, müssen durch Sicherheitsfunktionen in den Applikationsprogrammen selbst geschützt werden.

Hieraus folgt, dass zur Realisierung einer automatisierten Sicherheitsstrategie i. A. *Sicherheitsmaßnahmen auf mehreren Ebenen* benutzt werden müssen. Wenn Ressourcen der unteren Ebenen auf einer höheren Ebene repräsentiert werden, kann man zwischen beiden Ebenen wählen. Der Schutz von Ressourcen auf höheren Ebenen birgt jedoch die Gefahr des Unterlaufens der Sicherheitsfunktionen mit sich; hierauf gehen wir später noch genauer ein.

Administration. Sicherheitsfunktionen allein realisieren noch keine automatisierte Sicherheitsstrategie: wenn z. B. gemäß der Strategie gewissen Benutzern der Zugriff zu einer Datei erlaubt und allen anderen verboten sein soll, dann müssen die Rechte der Benutzer auch entsprechend gesetzt werden. Eine automatisierte Sicherheitsstrategie wird i. A. mit Hilfe einer oder mehrerer Sicherheitsfunktionen mit pas-

[1] Eine Sicherheitsfunktion ist nicht die Menge *aller* derartiger Schnittstellen bzw. Eigenschaften einer Ebene, sondern man bildet sinnvolle Mengen zusammengehöriger Schnittstellen bzw. Eigenschaften. Eine Ebene kann also mehrere, voneinander unabhängige Sicherheitsfunktionen anbieten. Beispiele für Sicherheitsfunktionen, die das Betriebssystem anbieten kann bzw. sollte, folgen später.

send eingestellten Parametern realisiert. Das Einstellen der Parameter nennt man auch **Administration** von Sicherheitsfunktionen; es erfolgt oft manuell.

Eine besondere Rolle kommt dabei dem sog. **Systemadministrator** zu. Dies ist ein Benutzer oder eine Benutzerrolle, die mehrere Benutzer spielen können, mit besonderen Rechten, z. B. dem Recht zur Verwaltung der eingetragenen Benutzer im System.

Sicherheitsfunktionen sollten möglichst generell anwendbar sein, d. h. durch Wahl geeigneter Parameter und Einstellungen eine möglichst große Menge von automatisierten Sicherheitsstrategien realisieren können.

Sichere Implementierung von Sicherheitsfunktionen. Die Sicherheitsfunktionen auf einer bestimmten Ebene wirken nur gegen Angriffe *von oben*, nicht hingegen gegen *seitliche* Angriffe oder Angriffe *von unten*. Beispielsweise kann man die Sicherheitsfunktionen eines Datenbanksystems dadurch unterlaufen, dass man direkt auf die Dateien zugreift, in denen die Datenbank gespeichert wird, vorausgesetzt kann man deren Format entschlüsseln. Ein anderes Beispiel ist das Betriebssystem selbst: der Schutz des oben erwähnten Druckers kann unterlaufen werden, wenn Benutzerprogramme (ggf. in Maschinensprache geschrieben) direkt und unter Umgehung des Betriebssystems auf die Schnittstelle zugreifen können, an die der Drucker angeschlossen ist. Ähnlich können Sicherheitsfunktionen des Dateisystems umgangen werden, wenn Benutzerprogramme direkt auf ein Speichermedium zugreifen können.

Im allgemeinen muss daher jede Ebene die sichere Implementierung von Sicherheitsfunktionen auf den höheren Ebenen unterstützen, indem ein Unterlaufen verhindert wird. Konkret bedeutet dies aus Sicht des Betriebssystems, dass

- die Hardware Sicherheitsfunktionen bieten muss, durch die gewährleistet wird, dass nur das Betriebssystem Zugriff auf in der Hardware realisierte schutzbedürftige Ressourcen (Schnittstellen, Geräte usw.) hat; dies geschieht meist durch die Unterscheidung zweier Betriebsmodi (privilegiert / nicht privilegiert) und diverse Speicherschutzmechanismen, siehe Kurseinheit 4.

- das Betriebssystem geeignete Funktionalitäten anbieten muss, die eine sichere Implementierung von Sicherheitsmechanismen in höheren Ebenen ermöglichen.

6.1.3 Subjekte und Objekte

Wir haben bisher Termini wie Benutzer, Ressource o. ä. benutzt, die in erster Linie auf Personen und Sachen der realen Welt, in etwa aber auch auf deren Repräsentation in einem Rechner anwendbar sind; dies war durchaus Absicht, denn die oben diskutierten Probleme waren davon weitgehend unabhängig. Da wir i. F. einen eher technischen, auf den Rechner bezogenen Standpunkt haben werden, ist es angebracht, auf technische Begriffe überzugehen.

Ziel aller Sicherheitsmechanismen ist letztlich, bestimmte Zugriffe oder Operationen zu verhindern, sofern sie von bestimmten Personen verursacht werden: Zugriffe werden veranlasst von *aktiven* Instanzen, die **Subjekte** genannt werden, und arbeiten auf *passiven* Instanzen, die **Objekte** genannt werden. Sodann muss sich ein

System in einer geeigneten Form merken, welche Subjekte welche Operationen auf welchen Objekten ausführen dürfen. Die hierzu angewandten konkreten Verfahren differieren beträchtlich, so dass wir vorerst nur etwas vage von *Rechten* oder *Berechtigungen* sprechen werden. Exaktere Definitionen folgen später.

Bei der Vergabe von Rechten werden in der Regel ähnliche oder zusammengehörige Operationen zu sog. **Zugriffsmodi** zusammengefasst. Beispielsweise wird man sequentielles Lesen einer Datei und direktes Lesen von Zeichen oder Sätzen nicht unterscheiden, sondern alle zugehörigen Operationen zu einem einzigen Modus namens *lesen* zusammenfassen.

Ein Objekt ist also eine identifizierbare zu schützende passive Einheit innerhalb der Rechteverwaltung des Betriebssystems. Ein Subjekt ist eine aktive Einheit innerhalb der Rechteverwaltung des Betriebssystems, die zu Objekten zugreift oder Ressourcen verbraucht.

Der Zusammenhang zwischen Begriffen der realen Welt und den technischen Begriffen stellt sich wie folgt dar:

Zunächst einmal können Menschen nicht direkt auf Daten oder sonstige Ressourcen in einem Rechner zugreifen. Sie können nur über irgendeine Schnittstelle des Rechners einen Prozess starten, der in ihrem Auftrag bestimmte Zugriffe im Rechner durchführt oder sogar weitere Prozesse startet. Benutzer müssen also im Rechner in geeigneter Form als Subjekte repräsentiert werden und jedem Prozess muss genau ein Subjekt des Typs *Benutzer* zugeordnet sein, in dessen Namen er arbeitet.

Repräsentationen von menschlichen Benutzern sind allerdings nicht die einzigen denkbaren Subjekte. Wenn über eine Schnittstelle irgendwelche Eingaben kommen, ist damit keineswegs gesagt, dass dahinter ein menschlicher Benutzer steht, es könnte z. B. auch ein anderer Rechner sein, dem ebenfalls gewisse Rechte als Rechner eingeräumt werden sollen, z. B. Post im Briefkasten abzulegen, oder ein Messwertgeber, der Messreihen in den Rechner überträgt.

Darüber hinaus ist es oft sinnvoll, Programmen Rechte dahingehend einzuräumen, dass ein Prozess die Rechte des Programms hat, das er ausführt. Hierdurch kann erreicht werden, dass Zugriffe zu bestimmten Objekten nur für bestimmte Programme möglich sind. In diesem Fall sind die Programme, die ja in einer Datei gespeichert werden, sowohl Objekt wie Subjekt. Ein typisches Beispiel hierfür ist ein Druckprogramm, das das *Schreib-Recht* für die Datei hat, die einen angeschlossenen Drucker repräsentiert; andere Subjekte haben nur das Recht, dieses Druckprogramm auszuführen, nicht aber, direkt in die Datei zu schreiben.

6.1.4 Zusammenfassung

Ziel dieser Kurseinheit ist, Sicherheitsfunktionen und -Mechanismen in Betriebssystemen vorzustellen. Um deren Motivation und Wirksamkeit diskutieren zu können, haben wir hier vorbereitend das Sicherheitsproblem ganz allgemein analysiert. Fassen wir noch einmal die wichtigsten Punkte dieser allgemeinen Analyse zusammen:

1. Das Sicherheitsproblem kann nahezu beliebig weit gefasst werden. In dieser Kurseinheit gehen wir nur auf solche Bedrohungen ein, die eine vom Betreiber oder Benutzer des Rechners unerlaubte oder ungewollte Benutzung eines Rechners darstellen (Sicherheit im engeren Sinn).

2. Das Sicherheitsproblem darf nicht auf den technischen Aspekt, also auf im Rechner realisierte Sicherheitsmaßnahmen, reduziert werden. Eine wirksame Sicherheitsstrategie muss neben technischen Maßnahmen auch Maßnahmen im organisatorischen oder psychologischen Bereich umfassen. Gegen viele Bedrohungen schützen Maßnahmen in den nichttechnischen Bereichen besser als technische Maßnahmen; in solchen Fällen ist es fraglich, ob in einem Rechner überhaupt Sicherheitsfunktionen gegen solche Bedrohungen realisiert werden sollten.

3. Sicherheitsfunktionen können nicht nur im Betriebssystem, sondern u. U. besser in darüberliegenden Schichten realisiert werden. Das Betriebssystem muss deshalb nicht alle denkbaren Sicherheitsfunktionen selbst enthalten, sondern die sichere Implementierung von spezielleren Sicherheitsfunktionen auf den höheren Ebenen ermöglichen.

4. Umgekehrt benötigt das Betriebssystem die Unterstützung der unter ihm liegenden Schicht, also der Hardware, um die von ihm angebotenen Sicherheitsfunktionen sicher implementieren zu können.

6.2 Sicherheitsfunktionen von Betriebssystemen

6.2.1 Anforderungen

Wir haben im vorigen Abschnitt das Sicherheitsproblem in einer ganzheitlichen Sicht behandelt; hieraus ergab sich, dass die Sicherheitsanforderungen und automatisierten Sicherheitsstrategien in konkreten Fällen sehr stark variieren können. Das eine Extrem sind unkritische Anwendungen auf PCs, bei denen nahezu keine Sicherheitsmaßnahmen ergriffen werden, das andere Extrem sind Anwendungen auf mehrbenutzerfähigen Rechnern in Bereichen wie Geheimdienst, Medizin oder Anlagensteuerung, in denen äußerst hohe Sicherheitsanforderungen vorliegen.

Wir stehen also wiederum vor der Frage, welche Sicherheitsfunktionen ein Betriebssystem dem Betreiber eines Rechners bzw. den Anwendungsprogrammen anbieten sollte. Wir wollen zunächst diese Frage einer recht globalen Kosten/Nutzen-Analyse unterwerfen.

Mit der Leistungsfähigkeit von Sicherheitsfunktionen in Betriebssystemen steigen leider auch mehrere Arten von Kosten:

1. der Aufwand für die Implementierung der Sicherheitsfunktionen, der sich letztlich in einem höheren Preis niederschlägt
2. der Laufzeit-Aufwand für die Sicherheitsfunktionen
3. der Administrationsaufwand

Den Entwicklungsaufwand und den damit zusammenhängenden Kostenaspekt wollen wir hier nicht diskutieren. Der Laufzeit-Aufwand stellt sich für die Benutzer eines Rechners als eine Leistungsminderung des Rechners dar, die aber bei der heutigen Leistungsstärke von Rechnern kaum noch relevant ist.

Wesentlich problematischer ist die Administration: das entscheidende Problem sind meistens weniger die administrativen Routinetätigkeiten, sondern der Erwerb

der notwendigen Kenntnisse zur adäquaten Handhabung der Administrationsfunktionen, d. h. es handelt sich hier eher um qualitative Kosten. Oft erfordert dies sehr gute Vorkenntnisse über das Betriebssystem und ausreichend Zeit zum Erlernen, Einüben und laufenden Aktualisieren des Wissens. Diese Voraussetzungen sind typischerweise bei hauptberuflichen Systemverwaltern von Großrechnern gegeben, nicht hingegen bei vielen Betreibern oder Benutzern von kleineren Rechnern.

Aus alledem folgt, dass es eine universell einsetzbare Menge von Sicherheitsfunktionen oder sogar nur eine universell gültige Menge von Anforderungen an Sicherheitsfunktionen eines Betriebssystems nicht geben kann.

6.2.2 Sicherheitsklassifikationen

Immer wieder sind Versuche unternommen worden, sinnvolle Klassifizierungen der Gesamtmenge der Sicherheitsfunktionen, die von einem (Betriebs-) System angeboten werden, zu entwickeln [TCSEC85, ITSEC91, BSI92]. Allerdings orientieren sich diese Klassifizierungen sehr stark an geheimdienstlichen / militärischen Anwendungen; die in den höheren Sicherheitsstufen geforderten Sicherheitsfunktionen scheinen nur bedingt auf andere sensitive Anwendungen anwendbar zu sein, etwa bei medizinischen, geschäftlichen oder personenbezogenen Daten. Dennoch werden diese Klassifizierungen wahrscheinlich dazu führen, dass die in den höheren Sicherheitsstufen geforderten Sicherheitsfunktionen in vielen kommerziellen Betriebssystemen implementiert werden. Die Nachfrage nach sicheren Systemen wird außerdem durch die zunehmend öffentlich geführte Diskussion über Computer-Sicherheit verstärkt.

Wenn man aus einem anderen Blickwinkel einmal untersucht, welche Sicherheitsfunktionen denn real existierende Betriebssysteme heute bieten, d. h. kommerziell vertriebene Betriebssysteme; Prototypen aus Forschungsinstitutionen lassen wir hier außer acht, dann ergibt sich folgendes Bild:

- Betriebssysteme für PCs haben meist keine nennenswerten Sicherheitsfunktionen.
- Betriebssysteme für Universalrechner (Mainframes, Workstations) haben meist konzeptionell überschaubare, noch relativ leicht administrierbare Sicherheitsfunktionen[2].
- Sicherheitsfunktionen, die hohen Anforderungen genügen, sind nur in wenigen speziellen, meist erst in letzter Zeit entwickelten Betriebssystemen vorhanden.

6.2.3 Funktionsbereiche

Dieser Abschnitt stellt nun eine Liste von typischen Bereichen (angelehnt an [ITSEC91, BSI92]) vor, in denen ein sicheres Betriebssystem Sicherheitsfunktionen

[2]Die Tendenz geht aber zu immer leistungsfähigeren Sicherheitsfunktionen. Diese Betriebssysteme stammen z. T. aus den 60er Jahren, in denen der räumliche Zugang zu Rechnern (Terminals, Ausgaberäume für Papierausgaben) üblicherweise strikt kontrolliert wurde, also durch organisatorische Maßnahmen bereits relativ viel Sicherheit gegeben war. Eine derartige räumliche Abschirmung ist heute meist unmöglich. Außerdem zeigten die inzwischen publik gewordenen Fälle von Computerkriminalität, dass auch eine räumliche Abschirmung Sicherheitsmechanismen im Rechner nicht überflüssig macht.

anbietet. Wir beschreiben diese Bereiche und Merkmale der Sicherheitsfunktionen informell. Die Beschreibung der Funktionen ist also wesentlich unpräziser und abstrakter als etwa Schnittstellenspezifikationen; ein so hoher Grad an Präzision wäre auch nicht sinnvoll, denn viele Details der Gestaltung der Schnittstellen sind für die zu erreichenden Ziele unerheblich.

Identifikation und Authentisierung. Hierunter fallen Funktionen, die erzwingen, dass sich jeder Benutzer dem Rechner gegenüber identifiziert, damit die Benutzung des Rechners durch unberechtigte Personen ganz verhindert werden kann. Üblicherweise wird hierzu zu Beginn einer Sitzung eine **login-Prozedur** durchlaufen, bei der ein Benutzer seine Identifikation angeben muss.

Ein Benutzer muss verifizieren, dass er tatsächlich der Benutzer mit der eingegebenen Identifikation ist, indem er sich durch weitere *Beweise* authentisiert. Derartige Beweise fallen in drei Klassen:

1. Wissen des Benutzers, z. B. Kenntnis eines Passworts,

2. Besitz des Benutzers, z. B. eines Schlüssels oder maschinenlesbaren Ausweises oder

3. Eigenschaften des Benutzers, z. B. Fingerabdruck, Klang der Stimme.

Zusätzlich müssen (Administrations-) Funktionen vorhanden sein, durch die Identifizierungen neu zugelassener Benutzer dem Betriebssystem bekannt gemacht werden bzw. durch die bisher vorhandene Benutzeridentifizierungen gelöscht werden können. Weiter muss die Authentisierungsinformation, die zur Überprüfung der Identität von Benutzern dient, änderbar sein.

Zugriffskontrollen. Die Zugriffskontrollen umfassen die *Rechteverwaltung* und *Rechteprüfung*. Diese Funktionen können den Informationsfluss zwischen Subjekten, den Zugang zu Information (Lesen), das Erzeugen oder Ändern von Information (Schreiben), die dabei benutzten Speicher und allgemein die Benutzung von Ressourcen kontrollieren bzw. einschränken. Hierzu gehören auch entsprechende Administrationsfunktionen. Bei der Verwaltung von Rechten können meist Benutzer zu Benutzergruppen und ggf. Objekte zu Objektgruppen zusammengefasst werden.

Rechte sollten so *feinkörnig* verwaltet werden, dass jedem Benutzer exakt die Rechte gewährt werden können, die dieser zur Erfüllung seiner Aufgabe benötigt, Prinzip des kleinstmöglichen Privilegs; *least privilege principle*, d. h. man sollte nicht wegen einer zu grobkörnigen Verwaltung der Rechte dazu gezwungen sein, Subjekten zusammen mit benötigten Rechten weitere Rechte zu gewähren, die diese eigentlich gar nicht benötigen.

Beweissicherung / Audit. In diesen Bereich fallen Funktionen, durch die sicherheitsrelevante Ereignisse protokolliert werden. Die Protokollierung erfolgreicher Ausübungen von Rechten dient dazu, später feststellen zu können, wie ein vorhandener Systemzustand (insb. vorhandene Rechte) entstanden ist, um so ggf. einen

Missbrauch von Rechten nachträglich entdecken zu können. Protokolle über Operationsaufrufe, die wegen fehlender Rechte zurückgewiesen wurden, können Hinweise auf Penetrationsversuche liefern.

Die Protokollierung kann i. d. R. parametrisiert werden bzgl. der Art der zu protokollierenden Ereignisse und des Umfangs der aufgezeichneten Information.

Hinzu kommen Funktionen zur Auswertung und Verwaltung der entstandenen Protokolle.

Initialisierung von Speicherbereichen. Speicherplatz im Hauptspeicher oder auf Sekundärspeichern wird i. A. wiederverwendet, also nacheinander zur Speicherung verschiedener Objekte benutzt. Wenn der alte Inhalt eines Speicherbereichs nach seiner Freigabe und vor seiner erneuten Zuweisung nicht gelöscht wird, kann ein unzulässiger Informationsfluss eintreten. Um dies zu vermeiden, müssen Sicherheitsfunktionen vorhanden sein, die Speicherbereiche vor ihrer Wiederverwendung, am besten aber direkt nach ihrer Freigabe initialisieren.

Gewährleistung der Funktionsfähigkeit. Eine denkbare Attacke auf einen Rechner oder wichtige Komponenten eines Rechnersystems besteht darin, einen Absturz zu provozieren oder durch Starten einer großen Zahl von Prozessen oder durch ähnliche Mittel den Rechner mehr oder weniger lahmzulegen und ihn dadurch für die berechtigten Benutzer unbrauchbar zu machen. Allerdings ist es für den Rechner in vielen Fällen nicht entscheidbar, ob eine Attacke oder eine *normale* Überlastsituation vorliegt.

Technologischen Attacken, bei denen Designfehler unzureichende Scheduling-Algorithmen, fehlende Prioritäten, Implementierungsfehler oder sonstige Mängel bei der Entwicklung oder der Installation des Betriebssystems ausgenutzt werden, wird am ehesten dadurch der Boden entzogen, dass diese Mängel beseitigt werden. Die Behandlung solcher Fehler gehört nach unserer früheren Definition nicht mehr zum engeren Themenbereich *Sicherheit*. Dennoch ist vorstellbar, dass zusätzliche Sicherheitsfunktionen ggf. *defensiver Code*[3] zur Sicherstellung der Funktionsfähigkeit des Systems eingeführt werden.

Datenübertragungssicherung. Die Übertragungswege bei Datenübertragungen liegen meist nicht mehr in der Kontrolle des Betreibers eines Rechners. Deshalb sind bei Datenübertragungen spezielle Sicherheitsmaßnahmen erforderlich. Hierzu gehören:

- Identifikation und Authentisierung von Kommunikationspartnern
- Zugriffskontrolle zu den Kommunikationseinrichtungen
- Sicherung der Vertraulichkeit der übertragenen Daten
- Sicherung der Integrität der übertragenen Daten

[3]Defensive Codes prüfen die Randbedingungen, die notwendig sind, damit Programme korrekt arbeiten. Es sollen alle denkbaren Fehlersituationen abgefangen werden, die eigentlich nicht notwendig wären, wenn alles richtig programmiert würde. Natürlich erhöhen die Defensive Codes Komplexität. Das Ziel der defensiven Codes ist, dass jede Operation so implementiert wird, dass sie sich SELBST vor FEHLERN schützt.

Diese Funktionsbereiche überschneiden sich z. T. mit den schon oben genannten; allerdings sind die konkreten Sicherheitsmechanismen für Datenübertragungen doch deutlich verschieden von jenen in Betriebssystemen. Die getrennte Behandlung der Sicherheit von Datenübertragungen ist also durchaus sinnvoll. Tatsächlich handelt es sich um ein sehr umfangreiches Gebiet, es existiert inzwischen sogar ein eigener Standard: ISO 7498-2-1988 Security Addendum, das man üblicherweise nicht dem Themenbereich *Betriebssysteme*, sondern eher dem Themenbereich *Rechnernetze* zuordnet. Aus diesem Grund werden wir es hier nicht weiter behandeln.

6.2.4 Implementierungen

Sicherheitsfunktionen eines Betriebssystems sind nur dann sinnvoll, wenn sie nicht umgangen werden können, siehe Abschnitt 6.1.2. Benutzerprozesse müssen daher daran gehindert werden, den Hauptspeicher anderer Benutzerprozesse oder des Betriebssystemkerns zu lesen oder zu schreiben, auf Hardware-Schnittstellen oder E/A-Geräte direkt zuzugreifen usw. Einige Mittel hierzu (Speicherschutzmechanismen, privilegierter / nichtprivilegierter Modus u. ä.) wurden bereits in früheren Kurseinheiten vorgestellt. Aus Platzgründen sollen hier keine weitergehenden und spezielleren Mittel vorgestellt werden; interessierte Leser seien auf [Bu00, Pc96, De82] verwiesen. Alle Schutzmechanismen basieren immer in irgendeiner Weise auf Funktionen, die in der Hardware realisiert werden und die von Benutzerprogrammen nicht umgangen werden können.

6.2.5 Spezifikationen

Wenn man bei einem konkreten Betriebssystem bzw. dessen Spezifikation herausfinden möchte, welche Sicherheitsfunktionen es eigentlich enthält und wie diese genau arbeiten, so wird man i. d. R. leider keine zentrale Stelle außer vielleicht in einem Tutorial finden, an der alles Wesentliche zusammengefasst ist. Tatsächlich sind die Sicherheitsfunktionen innerhalb der in Kurseinheit 1 vorgestellten Systemarchitektur bzw. den zugehörigen Schnittstellenspezifikationen sehr verstreut:

- Es gibt meist einige spezielle Systemaufrufe für Sicherheitsfunktionen, Beispiel in POSIX: Administration der Rechte mittels der Systemaufrufe `chmod` und `chgrp`.
- Zu einem solchen speziellen Systemaufruf gibt es meist ein Standard-Werkzeug, durch das i. W. der Systemaufruf eine Benutzungsschnittstelle erhält. Auf diese Werkzeuge gehen wir daher nicht weiter ein.
- Viele Details der Spezifikation von Systemaufrufen haben mit Sicherheitsfunktionen zu tun: die Semantik von vielen Systemaufrufen, z. B. von fast allen Operationen auf Dateien, erfordert eine Rechteprüfung oder hat Auswirkungen auf den Rechtezustand des Systems. Dementsprechend enthält z. B. die Spezifikation aller Systemaufrufe, die mit Dateien arbeiten, einen oder mehrere entsprechende Fehlercodes, die das Fehlen von Rechten, die zur Ausführung dieses Systemaufrufs erforderlich gewesen wären, anzeigen. Man mag geneigt sein, Fehlercodes als nebensächlich abzutun; tatsächlich aber spezifizieren diese Fehlercodes ganz wesentliche Teile der Sicherheitsfunktionen!

6.3 Benutzerverwaltung und -identifikation

Man kann ganz grob drei aufeinander aufbauende Stufen unterscheiden, in denen die Benutzung eines Systems eingeschränkt werden kann:

1. Man muss überhaupt als Benutzer zugelassen sein.
2. Ein Benutzer kann generellen Benutzungsbeschränkungen unterliegen; dies werden wir unter der Überschrift *Benutzerprofile* behandeln, siehe Abschnitt 6.4.
3. Der Zugriff eines Benutzers auf ein bestimmtes Objekt kann eingeschränkt werden; dies werden wir unter der Überschrift *Zugriffskontrollen* behandeln, siehe Abschnitt 6.5.

In diesem Abschnitt behandeln wir die erste, gröbste Form der Kontrolle. Sie besteht i. W. aus den folgenden Funktionsbereichen:

Benutzerverwaltung: Das System führt eine Liste der *Personen*, die das System benutzen dürfen; diese nennen wir im Folgenden kurz Benutzer.

Identifikation: Jede Person, die das System benutzen will, muss sich zunächst als ein registrierter Benutzer zu erkennen geben.

Authentisierung: Die Person muss ggf. durch weitere Angaben beweisen, dass sie wirklich der angegebene Benutzer ist.

6.3.1 Benutzerverwaltung

Für jeden zugelassenen Benutzer werden alle Angaben gespeichert, die für die systeminterne Verwaltung oder für den Betrieb des Rechners erforderlich sind, z. B.:

- Benutzername, dies ist eine Folge druckbarer Zeichen, meist mit Beschränkungen bzgl. Länge und Alphabet; ist nicht unbedingt der Vor- oder Nachname des Benutzers
- interner Benutzeridentifizierer
- Abrechnungskennzeichen
- Postanschrift, z. B. für Abrechnungen
- Telefonnummer
- Projektzugehörigkeit
- Datum der letzten Systembenutzung
- Authentisierungsdaten
- Benutzerprofil, siehe Abschnitt 6.4

Eingetragen werden fast alle Daten ausschließlich vom **Systemadministrator**; manche unkritische Eintragungen z. B. seine Telefonnummer können vom Benutzer selbst direkt geändert werden, oder werden als Seiteneffekt anderer Aktionen geändert, wenn z. B. der Benutzer sein Passwort ändert, werden implizit die Authentisierungsdaten geändert. Jeder registrierte Benutzer, der ein System benutzen

darf, bekommt z. B. unter UNIX vom Administrator eine eindeutige **UID** (**User IDentification**) zugewiesen. Eine UID ist ein Integer zwischen 0 und 65535. Jeder Prozess besitzt genau die UID des Benutzers, der ihn gestartet hat. Ein Kindprozess erbt die UID von seinem Vater. Ein Benutzer, der unter UNIX der **Superuser** (**root**) genannt wird, besitzt besondere Recht und kann viele Schutzmechanismen umgehen. Der Superuser hat die UID 0 und kann alle Dateien im System lesen, unabhängig davon, wem sie gehören oder wie sie geschützt sind. Prozess mit der UID 0 haben die Möglichkeit, eine kleine Anzahl geschützter Systemaufrufe durchzuführen, die für normale Benutzer gesperrt sind.

Benutzergruppen. Vielfach ist es praktischer, Rechte nicht für einzelne Benutzer, sondern für Gruppen von Benutzern zu vergeben. Die Zugriffskontrollen machen hiervon üblicherweise intensiven Gebrauch. Hierzu muss das Betriebssystem neben den Benutzern noch (Benutzer-) **Gruppen** verwalten. Jede Gruppe besitzt auch eine **GID**(**Group IDentification**). Zu jeder dem System bekannten Gruppe werden

- der Gruppenname (analog zum Benutzernamen)
- der interne Gruppenidentifizierer (analog zum internen Benutzeridentifizierer)
- die Liste der Mitglieder der Gruppe
- ggf. ein Verwalter der Gruppe

gespeichert.

In den meisten Betriebssystemen wird ein Benutzer dadurch als Mitglied einer Gruppe ausgewiesen, dass er vom Systemadministrator als Mitglied der Gruppe eingetragen wird. Stattdessen könnte man auch ein beliebiges Authentisierungsverfahren einsetzen.

Manchmal wird statt explizit eingerichteter, benannter Gruppen eine Gruppe durch einen *Benutzernamen* angegeben, der Sonderzeichen wie ‚*' für eine beliebige Folge von Zeichen oder ‚?' für ein beliebiges Zeichen enthält. Dies ist sozusagen der Name der Gruppe und zugleich ein Ausdruck, der die Menge der Mitglieder der Gruppe angibt. Die Gruppe ‚P12*' enthielte also alle Benutzer, deren Benutzernamen mit ‚P12' beginnt. Hierbei müssen die Benutzer- und Gruppennamen die organisatorische Hierarchie der Gruppen wiedergeben, also in etwa wie Abschnittsnummern in einem Inhaltsverzeichnis strukturiert werden. Dieses Verfahren ist einfacher realisierbar, weil Gruppen nicht mehr explizit eingerichtet werden müssen. Hauptnachteil ist aber, dass man die Benutzernamen z. B. den Nachnamen des Benutzers nicht mehr beliebig wählen kann, dass Umstrukturierungen der Gruppenhierarchie sehr schwierig sind, und dass überlappende Gruppen schlecht modelliert werden können.

Aktivierung von Gruppen. Sinn der Gruppen ist, wie schon angedeutet, einem Benutzer neben den Rechten, die er schon als Benutzer hat, weitere Rechte zu gewähren. Nun wird häufig der Fall eintreten, dass ein Benutzer Mitglied in mehreren Gruppen ist, weil er deren Rechte ab und zu braucht, aber eben nicht immer. Im einfachsten Fall kann ein Benutzer jederzeit die Rechte aller Gruppen ausnutzen, in denen er Mitglied ist, z. B. in der Berkeley-Version von UNIX. Das Vorhandensein der Rechte birgt andererseits die Gefahr in sich, dass sie bei einem fehlerhaften Kommando versehentlich ausgenutzt werden.

Damit sich der Benutzer vor Bedienungsfehlern schützen kann, ist es wünschenswert, die Mitgliedschaft in Gruppen nur bei Bedarf **aktivieren** zu können, so dass nur die Rechte von aktivierten Gruppen tatsächlich nutzbar sind. Die Frage ist nun, welche Gruppen für einen einzelnen Prozess aktiviert werden können:

- In UNIX System V bspw. ist immer genau eine vom Benutzer angegebene Gruppe aktiviert. Die aktivierte Gruppe kann geändert werden. Nachteilig ist hier, dass man u. U. sehr häufig die aktivierte Gruppe ändern muss. Außerdem muss man *künstliche* Gruppen einführen, wenn man die Rechte von zwei oder mehr vorhandenen Gruppen gleichzeitig benötigt.

- Es ist denkbar, eine beliebige, vom Benutzer angegebene Menge von Gruppen aktivierbar zu machen. Die Vielzahl der möglichen Kombinationen von Rechten ist allerdings dann schwer überschaubar.

- Bei den in [Ke90, Sa88] vorgestellten Konzepten wird *eine* Gruppe explizit aktiviert; dadurch werden alle „Obergruppen“ in einer im System verwalteten Hierarchie von Gruppen implizit mitaktiviert, die typischerweise die Organisationsstruktur eines Unternehmens oder Projekts widerspiegelt.

Die Möglichkeiten zur gleichzeitigen Aktivierung mehrerer Gruppen haben ganz erhebliche Auswirkungen darauf, wie der Systemadministrator Gruppen einrichten und mit Rechten versehen muss, um eine bestimmte automatisierte Sicherheitsstrategie zu realisieren.

Gegen die gleichzeitige Aktivierung mehrerer Gruppen spricht eigentlich das Prinzip *least privilege principle*, nur die minimal erforderlichen Rechte zu gewähren; wenn also die Rechte einer Gruppe immer einer Aufgabe entsprechen, dann braucht auch höchstens eine Gruppe aktiviert zu sein. Andererseits können dann, wenn die Aktivierung mehrerer Gruppen erlaubt ist, kleinere Mengen von zusammengehörenden Rechten gebildet werden.

Viele Systeme haben **vordefinierte Gruppen**, z. B. *Systembetreuung*, *Operateur* oder *alle_Benutzer*. In der Gruppe *alle_Benutzer* sind automatisch alle Benutzer Mitglied; ferner ist diese Gruppe i. d. R. stets automatisch aktiviert.

6.3.2 Identifikation

Ein Benutzer gibt zu Beginn einer interaktiven Sitzung oder eines Stapeljobs seinen Namen *Login-Name* und ggf. weitere Angaben dem Betriebssystem bekannt, z. B. durch Eingabe an einem Terminal oder durch ein *Job*- oder *login*-Kommando bei Stapeljobs. Das Betriebssystem prüft dann, ob ein Benutzer mit dem angegebenen Namen bekannt ist und, falls ja, wie sich dieser Benutzer authentisieren muss und verlangt dann ggf. weitere Angaben. Diesen ganzen Vorgang bezeichnen wir als **login-Prozedur**.

Nach erfolgreichem Durchlaufen der login-Prozedur wird für den Benutzer ein Prozess erzeugt, der dem Benutzer eine initiale Arbeitsumgebung zur Verfügung stellt; dies kann ein Kommandointerpreter sein, der die folgenden Kommandos des Benutzers interpretiert, oder eine Applikation. Das Betriebssystem speichert die Identifikation dieses Benutzers als Teil des Prozesskontrollblocks sowie jedes weiteren Prozesses, der von diesem Prozess gestartet wird. Hierdurch kann bei jedem

Zugriff eines Prozesses leicht herausgefunden werden, in wessen Namen dieser Zugriff ausgeführt wird; dies wird insb. für die Zugriffskontrollen benötigt.

6.3.3 Authentisierung

Man kann i. A. nicht davon ausgehen, dass die Benutzernamen geheim sind: sie erscheinen auf den Deckblättern von Papier-Ausgaben, werden für die elektronische Post benutzt oder von Befehlen ausgegeben, die die derzeit aktiven Benutzer im System anzeigen. Die Kenntnis eines Benutzernamens ist also i. A. kein hinreichender Beweis dafür, dass die von einem Benutzer angegebene Identität mit seiner tatsächlichen übereinstimmt. Im Rahmen der login-Prozedur kann daher die Vorlage zusätzlicher Beweise für die Echtheit der angegebenen Benutzeridentität verlangt werden.

Am häufigsten werden hierzu Passwörter benutzt. Die einfachsten Implementierungen halten eine zentrale Liste von (Login-Name, Passwort)-Paaren. Der eingegebene Login-Name wird in der Liste gesucht und das angegebene Passwort wird mit dem gespeicherten Passwort verglichen. Eine bessere Lösung, die auch in UNIX verwendet wird, ist, Passwörter in der Liste (Login-Name, Passwort) verschlüsselt zu speichern. Wenn ein Passwort eingetippt wird, wird es gleich verschlüsselt. Danach wird der Login-Name in der Liste gesucht. Nachdem der Login-Name gefunden wird, wird das verschlüsselte Passwort mit dem eingetippten und verschlüsseltem Passwort verglichen. Der Vorteil des Verfahrens ist, dass niemand, auch nicht der Superuser, Passwörter von Benutzern nachschlagen kann, weil sie nirgendwo im System in unverschlüsselter Form abgespeichert werden.

Es gibt einige Regeln, wie Passwörter gehandhabt werden sollten.

1. Passwörter sollten mindestens 8 Zeichen lang sein.
2. Passwörter sollten sowohl Klein- als auch Großbuchstaben enthalten.
3. Passwörter sollten wenigstens ein Sonderzeichen oder eine Zahl enthalten.
4. Passwörter sollten nicht in Wörterbüchern vorkommen oder Namen von Leuten sein.

Passwörter sollten regelmäßig durch den Benutzer gewechselt werden. Am sichersten ist eine Generierung durch das System, damit sie nicht zu leicht z. B. durch systematisches Ausprobieren zu erraten sind. Seltener sind Authentisierungsdialoge, Schlüssel, Ausweiskarten oder andere auf spezieller Hardware beruhende Verfahren, siehe [Pc96]. Auf die Vielzahl der Details wollen wir hier nicht näher eingehen.

Interne Benutzeridentifizierer. Die Benutzeridentifikation bildet eine Basis für diverse andere Sicherheitsfunktionen, insb. das Benutzerprofil, Zugriffskontrollen und Audit. Die Realisierung dieser Sicherheitsfunktionen erfordert, dass für jede Aktivität eines Prozesses bekannt ist, im Namen welchen Benutzers dieser Prozess läuft. Nun wäre es relativ unpraktisch, den Benutzernamen, also einen Text, bei dem Prozess oder für gewisse Arten der Zugriffskontrollen bei den Objekten zu speichern. Deshalb verwendet man hier besser interne Benutzeridentifizierer; dies sind typischerweise Nummern. Der Benutzername wird somit nur noch im Rahmen der

login-Prozedur benötigt und dabei in eine interne Benutzeridentifizierung umgesetzt; diese ist dann Merkmal des anschließend gestarteten Prozesses und vererbt sich auf alle Subprozesse dieses Prozesses. Die interne Benutzeridentifizierung kann natürlich bei Bedarf, z. B. für Auskunftsfunktionen, wieder in den Benutzernamen umgesetzt werden.

Interne Benutzeridentifizierer werden üblicherweise wiederverwendet, d. h. wenn ein Benutzer ausgetragen wird, wird sein Identifizierer freigegeben und später eventuell für einen neuen Benutzer wiederverwendet. Wenn nun irgendwo im System noch Berechtigungen für den ausgetragenen Benutzer vorhanden sind, erbt der neue Benutzer diese unkontrolliert. Diesen Effekt kann man ausschließen, indem interne Benutzeridentifizierer nicht wiederverwendet werden; dies hat den zusätzlichen Vorteil, dass ein interner Benutzeridentifizierer einen Benutzer auch über die Zeit hinweg eindeutig identifiziert, erfordert aber i. A. einen relativ großen Wertebereich für die Identifizierer (speziell in verteilten Systemen) und wird selten praktiziert. Alternativ kann man beim Austragen eines Benutzers das komplette System daraufhin untersuchen, ob noch Berechtigungen für diesen Benutzer existieren und diese dann löschen. Je nach der Methode, wie Berechtigungen im System verwaltet werden, kann dies aus Aufwandsgründen unmöglich sein.

6.3.4 Programme als Subjekte

Wir haben bisher zwei Sorten von Subjekten kennengelernt: Benutzer und Gruppen. Die Berechtigungen insb. bei den Zugriffskontrollen, die einem Subjekt zuteilbar sind, sind aber oft noch zu grobkörnig.

Betrachten wir hierzu folgendes Beispiel: man will allen Benutzern erlauben, Botschaften in einem Briefkasten abzulegen. Dieser ist als spezielle Datei realisiert, in der die Botschaften in einer ganz bestimmten Syntax hintereinander gespeichert sind, die die spätere Trennung der einzelnen Botschaften ermöglicht. Würde man nun allen Benutzern das Recht geben, in die Datei zu schreiben bzw. hinten Text anzuhängen, könnten sie die Syntaxregeln für die Datei verletzen. Die Überwachung einer speziellen Syntax innerhalb einer Datei ist andererseits ein zu spezielles Problem, als dass dafür ein allgemeiner Zugriffskontrollmechanismus verantwortlich sein könnte.

Das Problem ließe sich am einfachsten dadurch lösen, dass nur ein *spezielles Dienstprogramm* Botschaften in den Briefkasten einträgt. Dies lässt sich dadurch erzwingen, dass nur dieses Programm das Recht zum Schreiben der Briefkastendatei erhält und alle Benutzer das Recht zum Ausführen dieses Programms. Mit anderen Worten ist das *Programm Subjekt von Berechtigungen*. Auf diese Weise lassen sich nahezu beliebig feinkörnige Berechtigungen vergeben, wobei sogar z. T. die Semantik der Operationen auf Objekten mit in die Rechteprüfung einbezogen werden kann.

Programme als Subjekte verursachen andererseits gewisse Probleme. Während die Menge der Benutzer und Gruppen relativ klein ist und sich nur selten unter der Kontrolle des Systemadministrators ändert, ist die Zahl der Programme bzw. Dateien i. A. sehr groß, und jeder Benutzer kann praktisch jederzeit Programme erzeugen oder löschen. Die internen Identifizierer von Programmen müssten daher ziemlich lang sein. Der Pfadname der Datei, in der ein Programm steht, ist als Identifizierer nicht geeignet, denn Dateien können i. A. innerhalb von Verzeichnis-Bäumen verschoben werden. Dateien haben oft überhaupt keinen systemweit eindeutigen und

stabilen Identifizierer, der nicht wiederbenutzt wird. Viele diesbezügliche Probleme kann man durch Programmgruppen vermeiden, die genau wie Benutzergruppen explizit durch den Systemverwalter verwaltet werden. Hierdurch können auch Programme mit gleichen Rechten zu einer Gruppe zusammengefasst werden. Einzelne Programme braucht man dann gar nicht mehr als Subjekte zuzulassen, ggf. muss eine Gruppe mit einem einzigen Mitglied gebildet werden. Man kann auch *gemischte* Gruppen zulassen, in denen sowohl Benutzer wie Programme Mitglied sind.

Das SETUID-Bit Programme als Subjekte treten nur in wenigen (neueren) Systemen auf. In POSIX wird ein ganz anderer Weg begangen, Programmen Rechte zu gewähren. Aufgerufene Programme werden normalerweise in einem eigenen Prozess ausgeführt, der die aktiven Subjekte des aufrufenden Prozesses erbt. Von dieser Regel wird abgewichen, wenn bei dem Programm das sog. **SETUID-Bit**, auch **s-Bit** genannt, gesetzt ist. In diesem Fall wird das Programm *im Namen des Besitzers* des Programms ausgeführt, d. h. aktiviertes Subjekt ist der Besitzer des Programms, nicht der zum aufrufenden Prozess gehörige Benutzer.

Ein Beispiel der Anwendung sind Spezialprogramme, die mit Ein-/Ausgabe-Geräten verknüpft sind. Zum Beispiel kann der Besitzer des Druckerprogramms, welches die Druckaufträge in einer Warteschlange verwaltet, entweder der *Superuser* (*root*) oder ein spezieller Benutzer, *daemon*, sein. Nur der Besitzer, sagen wir *daemon*, darf das Programm lesen und schreiben, damit niemand anderes das Programm manipulieren und so vielleicht den Drucker direkt ansprechen kann. Die Warteschlange wiederum ist auch nur für *daemon* schreib- und lesbar. Ausführbar ist das Druckerprogramm aber für alle Benutzer. Wenn das SETUID-Bit des Druckerprogramms, das *daemon* gehört, gesetzt ist, dann kann es von jedem Benutzer ausgeführt werden, und sie erhalten dadurch die Fähigkeiten von *daemon*, aber nur für die Ausführung dieses Programms, d. h. sie dürfen Druckaufträge in die Warteschlange stellen.

Viele sensible UNIX-Programme gehören *root*, haben aber das SETUID-Bit gesetzt, z. B. das Programm, das es Benutzern ermöglicht, ihr Passwort zu ändern. Die Datei, in der die Passwörter aller Benutzer gespeichert sind, kann nur von *root* gelesen und beschrieben werden. Um allen Benutzern zu ermöglichen, ihr eigenes Passwort zu ändern, gibt es das Programm `/usr/bin/passwd` mit folgenden Zugriffsrechten, man beachte das kleine `s`:

```
-rwsr-xr-x  1 root shadow    /usr/bin/passwd
```

Wer sein Passwort ändern möchte, startet dieses Programm. Der Prozess erhält durch das s-Bit die Rechte des Besitzers, also *root*, und kann also die Passwortdatei ändern, das Programm wird aber nur die Änderung des *eigenen* Passworts erlauben. Das Programm muss also so sorgfältig geschrieben sein, dass dadurch keine andere, unerlaubte Aktion gestartet werden kann.

Dieses Konzept hat allerdings gravierende Nachteile: der aufgerufene Prozess und alle von ihm gestarteten Prozesse (!) haben nicht nur z. B. das Recht zum Schreiben des einen Briefkastens, sondern *alle* Rechte des Besitzers dieses Programms. Wenn es z. B. gelingt, von diesem Prozess aus einen weiteren Prozess, z. B. einen Kommandointerpreter oder z. B. einen Texteditor, zu starten, dann sind schwerwiegende Einbrüche möglich. Dies ist besonders fatal, wenn der Besitzer des Programms der *super user* ist, für den alle Zugriffskontrollen abgeschaltet sind, was in UNIX bei

vielen Dienstprogrammen der Fall ist, in denen sich eine Reihe berühmter Sicherheitslücken finden.

Es gibt noch eine Reihe weiterer Konzepte, wie die Rechte eines Kindprozesses im Vergleich zum Elternprozess vermehrt werden können; hierauf können wir aber aus Platzgründen nicht näher eingehen.

Aktive Subjekte eines Prozesses. Die aktiven Subjekte eines Prozesses sind diejenigen Subjekte, deren Rechte der Prozess ausnutzen kann. Nach unseren bisherigen Diskussionen sind dies:

- der Benutzer, für den ein Prozess läuft
- alle explizit oder implizit aktivierten Gruppen, in denen der Benutzer, für den ein Prozess läuft, natürlich Mitglied sein muss
- ggf. das ausgeführte Programm oder Gruppen, in denen das ausgeführte Programm Mitglied ist.

6.4 Benutzerprofile

Die generellen *Benutzungsbeschränkungen* bzw. Rechte eines *einzelnen Benutzers* bezeichnet man als **Benutzerprofil**. Diese können z. B. betreffen:

- Einschränkungen bzgl. der Wahl der Kommandosprache bzw. der auszuführenden Applikation: sofern mehrere Kommandosprachen oder Einstellungen einer Kommandosprache verfügbar sind, kann hierdurch die Menge der Kommandos, die ein Benutzer aufrufen kann, eingeschränkt werden.

  Oft kann auch erzwungen werden, dass zu Beginn einer Sitzung automatisch eine bestimmte Applikation aufgerufen wird und an deren Ende die Sitzung automatisch beendet wird.
- Quota: Der Verbrauch von Rechenzeit, Druckerpapier, Plattenplatz etc. kann oft quantitativ eingeschränkt werden, ggf. sogar individuell für jeden einzelnen Prozess. In vielen an der Stapelverarbeitung orientierten Systemen muss für jeden Job schon zu Beginn angegeben werden, wieviele Ressourcen er maximal benötigen wird. Diese Angaben werden beim Scheduling der Jobs ausgenutzt. Beim Überschreiten der Grenzen wird der Prozess abgebrochen.
- Die Ausübung gewisser sicherheitskritischer Funktionen kann generell verboten werden, z. B.:
  - die Benutzung der Break-Taste des Terminals
  - das Starten von Subprozessen
  - die Änderung des eigenen Passworts[4]

  Ggf. werden solche Funktionen auch anhand von standardisierten Rollen zusammengefasst, z. B. Gäste / normale Benutzer / Operateur / Systemadministrator. Jede Rolle ist mit einer bestimmten Menge von **Privilegien** verbunden.

[4]Das Verbot der Änderung des eigenen Passworts ist nur dann sinnvoll, wenn eine Gruppe von Personen unter der gleichen *anonymen* Benutzeridentifizierung arbeitet. Allerdings ist aus Sicherheitsgründen von solchen anonymen Benutzeridentifizierungen eher abzuraten.

- Ort und Zeit der Benutzung des Rechners können eingeschränkt werden. Die Benutzungszeit kann z. B. auf die normale Bürozeit eingeschränkt werden. Der Ort, von dem aus ein Rechner benutzt werden kann, kann nur indirekt über die benutzte Schnittstelle eingeschränkt werden. Hierzu zählen nicht nur Terminal-Schnittstellen, sondern auch Netzwerk-Schnittstellen: so kann die Benutzung des Rechners über ein Netzwerk ganz verboten werden.

In modernen Betriebssystemen, die Kommandointerpreter und Kommandos als Werkzeuge realisieren, braucht man für viele der oben aufgeführten Beschränkungen keine eigenen Konzepte und Schnittstellen in den Betriebssystemkern einzuführen: so kann die Wahl der Kommandosprache dadurch eingeschränkt werden, dass ein entsprechender Kommandointerpreter gestartet wird. Zeitliche oder örtliche Beschränkungen können wiederum durch spezielle Kommandointerpreter realisiert werden. Die Benutzung einzelner Kommandos kann einzelnen Subjekten durch Zugriffskontrollen (s. u.) erlaubt bzw. verboten werden, indem das Recht gewährt oder nicht gewährt wird, den Inhalt der Datei, die das ausführbare Kommando enthält, zu laden und auszuführen. Auf diese Weise kann man vermeiden, den Kern des Betriebssystems unnötig aufzublähen. Außerdem sind die Einschränkungen so leichter auf spezielle Bedürfnisse anzupassen oder zu erweitern. Nachteile sind die Gefahr von Implementierungsfehlern, z. B. wenn man durch Benutzung des Break-Signals zu einem mächtigeren Kommandointerpreter kommen kann, die eventuell fehlende Übersicht über das vollständige Benutzerprofil einzelner Benutzer (was aber durch entsprechende Werkzeuge hergestellt werden könnte) und u. U. der geringere Zwang zur Anwendung der Sicherheitsmaßnahmen. Kein wesentlicher Unterschied besteht in der erforderlichen Administration.

6.5 Zugriffskontrollen

Zugriffskontrollen ermöglichen es, einzelnen *Subjekten* den Zugriff auf einzelne *Objekte* zu erlauben bzw. zu verbieten. Die wichtigsten Funktionen im Zusammenhang mit Zugriffskontrollen wurden schon in Abschnitt 6.2.3 kurz vorgestellt: Rechteverwaltung und Rechteprüfung.

6.5.1 Einführung

Ziele der Zugriffskontrollen. Die Zugriffskontrollen sollen letztlich einen Beitrag zur Realisierung einer automatisierten Sicherheitsstrategie leisten. Ehe wir uns mit Details der Funktionsweise der Zugriffskontrollen befassen, wollen wir uns erst eine grobe Vorstellung davon verschaffen, worin denn dieser Beitrag besteht.

Die Zugriffskontrollen haben das Ziel, vor allem folgende Arten von Schäden zu verhindern:

- *Verlust der Vertraulichkeit bzw. Integrität von Information*: dies stellt sich innerhalb des Rechners so dar, dass zu schützende Information, die in Objekten gespeichert ist, von unbefugten Subjekten gelesen bzw. verändert oder zerstört wird.

- *unzulässige Benutzung von Ressourcen*: Sofern Ressourcen durch Objekte repräsentiert werden, kann der Zugriff auf diese Ressourcen ganz verhindert

oder auf bestimmte Arten der Nutzung *Operationen* eingeschränkt werden, hierbei handelt es sich eher um *qualitative* Einschränkungen; quantitative Beschränkungen wie z. B. Quota zählen üblicherweise nicht zu den Zugriffskontrollen.

Subjekte und Objekte. Bei einer abstrakten, konzeptionellen Betrachtungsweise der Zugriffskontrollen spielt es keine große Rolle, um welche konkreten Subjekte und Objekte es sich handelt. *Objekte* sind alle, auf das mittels Zugriffskontrolle zugegriffen kann. Beispiele sind normale Dateien, ausführbare Dateien, Verzeichnisse, ganze Bäume von Verzeichnissen, Bänder, Disketten oder andere Medien, Schnittstellen, Uhren, Seiten, Datenstrukten usw. sein. Jedes Objekt hat einen eindeutigen Namen, über den es referenziert wird und eine endliche Menge von Operationen, die von Prozessen auf diesem Objekt ausgeführt werden können. Z. B. für eine Datei sind die Operationen `read` und `write` sinnvoll; bei einem Semaphor sind `down` und `up` sinnvoll. *Subjekte* sind Einheiten, die auf Objekte zugreifen können. Sie können Benutzer, Prozesse, Schnittstellen usw. sein.

Sobald man jedoch an die Implementierung der Konzepte geht, ergeben sich aus der Art und den Merkmalen der Subjekte und Objekte wichtige Randbedingungen, die auf die Anwendbarkeit der Konzepte und auf Details ihrer Realisierung großen Einfluss haben. Deshalb werden wir schon bei der Diskussion der Konzepte deren Anwendungsbereich im Auge behalten. Es zeigt sich, dass eine Unterscheidung zwischen *persistenten* und *transienten* Subjekten bzw. Objekten hierfür nützlich ist; diese werden wir deshalb zunächst vorstellen.

Der persistente Rechtezustand. Wir nehmen zunächst einmal an, dass keine Prozesse ablaufen: Rechte existieren unabhängig von Prozessen, auch wenn sie nur von Prozessen ausgeübt werden können. Rechte sind also ein Teil des *persistenten*, d. h. Systemabschaltungen überdauernden Zustands eines Rechners. Wir nennen die Menge solcher Rechte den **persistenten Rechtezustand**. Der persistente Rechtezustand macht direkt oder indirekt Aussagen darüber, welche Subjekte auf welchen Objekten welche Operationen ggf. mit welchen Parametern ausführen dürfen oder nicht ausführen dürfen. Insofern beruht er begrifflich auf Objekten insb. deren *Datenmodell* und Subjekten. Der persistente Rechtezustand ist bei den verschiedenen Zugriffskontrollverfahren völlig verschieden aufgebaut, so dass wir erst später auf Details seiner Struktur eingehen werden.

Man erkennt sofort, dass nur solche Subjekte und Objekte im Rahmen des persistenten Rechtezustands Sinn machen, die ebenfalls persistent sind; Prozesse sind z. B. keine persistenten Subjekte. In vielen Systemen sind nur Dateien persistente Objekte, weil alle Geräte, Medien, Schnittstellen oder sonstige schutzbedürftige Ressourcen als Dateien repräsentiert werden[5].

Einen Überblick über den Zusammenhang zwischen Rechteverwaltung und Rechteprüfung einerseits und Funktionsblöcken wie der Benutzerverwaltung andererseits gibt Abbildung 6.1 in Form einer Grobstruktur an.

[5]Dies ist nicht nur im Sinne der Geräteunabhängigkeit von Applikationen vorteilhaft, siehe Kurseinheit 2, sondern auch wegen der vereinfachten und homogeneren Zugriffskontrollen.

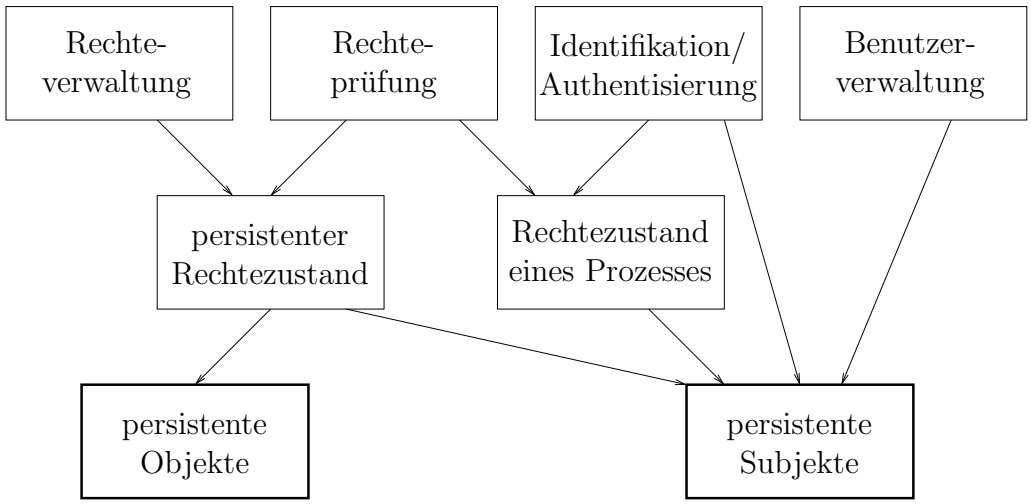


Abbildung 6.1: Ein Funktionsschema für Zugriffskontrollen, die Kästen stellen Funktionsbereiche oder Systemzustände dar, nicht unbedingt Moduln.

Die Basis bilden die Subjekte und Objekte, die im System verwaltet und insb. als Teil des Systemzustands persistent gespeichert werden. Die Subjekte und ihre Attribute werden von den *Benutzerverwaltungsfunktionen* verwaltet, z. B. eingetragen, abgefragt, gelöscht, geändert usw.. Eine analoge Komponente zur Verwaltung der Objekte ist nicht extra aufgeführt.

Die wichtigste Aufgabe der *Rechteverwaltung* ist, den *persistenten Rechtezustand* zu verwalten. Die *Rechteprüfungsfunktionen* benutzen den persistenten Rechtezustand, um die Zulässigkeit von Zugriffen eines Prozesses zu prüfen.

Zugriffe zu Objekten werden nur von Prozessen veranlasst, nicht von menschlichen Benutzern oder anderen externen Subjekten. Daher muss eindeutig festliegen, für welche Subjekte ein Prozess aktiv ist; dies geschieht u. a. innerhalb der login-Prozedur. Die Korrektheit aller Rechteprüfungen beruht somit ganz wesentlich auf der korrekten Zuordnung von Benutzern zu Subjekten mittels der Identifikations- und Authentisierungsfunktionen im Rahmen der login-Prozedur. Wir bezeichnen die Menge der aktiven Subjekte des Prozesses, siehe Abschnitt 6.3.4, und ggf. zusätzlich Attribute der Subjekte, wie z. B. den Grad ihrer Vertrauenswürdigkeit, als den **Rechtezustand dieses Prozesses**.

Die Rechteprüfungen benutzen sowohl den persistenten Rechtezustand wie auch den Rechtezustand des Prozesses, dessen Zugriffe gerade kontrolliert werden.

Der Rechtezustand eines Prozesses wird naheliegenderweise im Prozesskontrollblock gespeichert. Er wird vom Elternprozess gesetzt, also entweder vom login-Prozess oder von einem Benutzer-Prozess; die Details ergeben sich aus der Semantik der Systemaufrufe, die Prozesse starten. Ein Prozess kann seinen Rechtezustand, z. B. die Menge seiner aktiven Subjekte, nur an unkritischen Stellen ändern, denn sonst könnte der Prozess eine falsche Identität vortäuschen. Ein Beispiel für eine solche Änderung ist ein Systemaufruf, durch den eine Benutzergruppe aktiviert oder deaktiviert werden kann. Ebenfalls Teil des Prozessdeskriptors ist oft eine Voreinstellung (Default-ACL, siehe Abschnitt 6.5.2.3) für die Rechtefestlegungen der Objekte, die von diesem Prozess erzeugt werden.

Der transiente Rechtezustand. Nachdem im Rahmen der Rechteprüfung einem Prozess ein bestimmter Zugriff erlaubt wurde, ist es sehr häufig praktisch, dies als Entstehung eines neuen *transienten* Rechts aufzufassen, bei dem der Prozess Subjekt ist. Ein typisches Beispiel ist die Art, wie Rechte an Dateien überprüft werden: Beim Öffnen einer Datei werden die Rechte der aktiven Subjekte auf Basis des persistenten Rechtezustands überprüft. Wenn ausreichende Rechte vorhanden sind, wird die Datei im gewünschten Modus geöffnet; hierbei wird intern ein *Dateikontrollblock* angelegt, in dem u. a. die erlaubten Zugriffsmodi vermerkt sind. Nachdem die Datei einmal geöffnet ist, werden bei jedem einzelnen Lesen eines Zeichens oder Satzes die Rechte nur noch auf Basis der Daten im Dateikontrollblock geprüft, nicht mehr auf Basis des persistenten Rechtezustands.

Wesentlich ist hier, dass die Rechteprüfungen nicht auf Basis des persistenten Rechtezustands stattfinden, sondern auf Basis eines **transienten Rechtezustands**, der bzgl. dieser Datei im Dateikontrollblock verwaltet wird. Der transiente Rechtezustand stellt eine Teilmenge der Rechte, die durch den persistenten Rechtezustand gegeben sind, in einer effizienter benutzbaren Form dar, insofern kann man ihn als Cache ansehen.

Ein Prozess ist in diesem Sinne ein **transientes Subjekt**, das bei Beendigung des Prozesses oder bei einem Systemstillstand automatisch verschwindet. Die Endlichkeit der Lebenserwartung ist allerdings nicht das einzige Unterscheidungsmerkmal zwischen den beiden Sorten von Subjekten, denn auch persistente Subjekte können explizit gelöscht werden, wenngleich dies um Größenordnungen seltener geschieht als bei Prozessen. Wichtiger ist, dass Prozesse Kindprozesse erzeugen können, ein ähnliches Konzept gibt es für persistente Subjekte nicht, und dass dabei meist die aktuell vorhandenen Rechte des Elternprozesses an seinen Kindprozess *selektiv* weitergegeben werden sollen. Nehmen wir z. B. an, ein Prozess hat mehrere Dateien erzeugt und möchte eine von diesen von einem Dienstprogramm, das als Kindprozess abläuft, verarbeiten lassen. Dann wäre es unpraktisch, wenn alle Rechteprüfungen im Kindprozess erneut durchgeführt werden müssten.

Ferner arbeiten Prozesse häufig mit *transienten Objekten*, z. B. irgendwelchen vom Betriebssystem verwalteten Datenstrukturen im Hauptspeicher. Auch hier ist die Endlichkeit der Lebensdauer nicht das entscheidende Merkmal, eine temporäre Datei existiert u. U. auch nur für wenige Sekunden. Wesentlich sind hier die Adressierung bzw. die Identifizierer der Objekte: Diese Objekte werden typischerweise durch prozesslokale Referenzen identifiziert, siehe Abschnitt 6.5.2.4.

Unter dem transienten Rechtezustand verstehen wir somit alle Rechte, in die transiente Subjekte oder Objekte involviert sind. Offensichtlich überdauert der transiente Systemzustand eine Systemabschaltung nicht. Im Gegensatz zum persistenten Rechtezustand wird der transiente Rechtezustand nicht explizit, also z. B. durch Kommandos des Benutzers oder Systemverwalters, verändert, sondern implizit durch das Starten von Prozessen, Öffnen oder Schließen von Dateien usw.

Zugriffskontrollmodelle. Wir haben bisher erst einige globale Aspekte der Zugriffskontrollen betrachtet; völlig offen blieb bisher, wie sie denn im Detail funktionieren. Die festzulegenden Details betreffen folgende Punkte:

- Welche Struktur, welchen *Datentyp* haben der persistente und der transiente Rechtezustand? Wie können sie sich ändern?

- Wie ist der Rechtezustand eines Prozesses aufgebaut? Wie wird er initialisiert bzw. verändert?
- Wie wird über die Zulässigkeit eines Zugriffs entschieden?

Die Summe der Antworten auf diese Fragen nennen wir ein **Zugriffskontrollmodell**.

Wie hängen nun ein Zugriffskontrollmodell und eine automatisierte Sicherheitsstrategie zusammen? Man vergleicht ein Zugriffskontrollmodell am besten mit einer Programmiersprache: der konkrete persistente Rechtezustand ist ein *Programm* in dieser Sprache, zusammen mit dessen Semantik ist hieraus ableitbar, welche Subjekte derzeit wie auf welche Objekte zugreifen können. Mit anderen Worten besteht eine automatisierte Sicherheitsstrategie aus

- dem persistenten Rechtezustand und seiner Administration durch Automatismen innerhalb des Zugriffskontrollmodells und/oder durch automatisierte Verfahren auf der Ebene der Applikationen
- den vom System durchgeführten Prüfungen und anderen Auswirkungen der *Semantik* des Zugriffskontrollmodells

Ähnlich wie eine Programmiersprache ist ein Zugriffskontrollmodell normalerweise nur für eine bestimmte Klasse von Problemen geeignet, in diesem Fall Sicherheitsprobleme, weil nur die Probleme in dieser Klasse *bequem*, *elegant* oder überhaupt gelöst werden können.

Klassen von Zugriffskontrollmodellen. Man unterscheidet traditionell zwei Hauptgruppen von Zugriffskontrollmodellen: diskretionäre und informationsflussorientierte bzw. markenbasierte. Diese werden in den beiden folgenden Abschnitten ausführlicher vorgestellt werden. Daneben gibt es noch weitere Zugriffskontrollmodelle, die nicht in diese beiden Klassen passen; diese werden wir anschließend ganz kurz behandeln.

Wir wollen zunächst zwischen zwei ganz prinzipiellen Strategien für Zugriffskontrollen unterscheiden:

1. **offene Systeme** (bzw. explizites Verbieten): verboten ist nur, was explizit verboten ist; alles andere ist automatisch erlaubt. Ein Beispiel hierfür sind PC-Betriebssysteme, in denen Festplatten oder Partitionen von Festplatten nur dann geschützt sind, wenn explizit ein Passwort für sie eingerichtet wird.
2. **geschlossene Systeme** (bzw. explizites Erlauben): erlaubt ist nur, was explizit erlaubt ist; alles andere ist automatisch verboten. Alle *ausgewachsenen* Betriebssysteme sind geschlossene Systeme.

Beide Ansätze sind bei den üblichen diskretionären Zugriffskontrollen, siehe Abschnitt 6.5.2, mit einer zweiwertigen Logik insofern äquivalent, als man im Prinzip jede Berechtigungssituation auf beide Arten darstellen kann[6]. Der Hauptunterschied liegt in der Wahrscheinlichkeit der Entdeckung von Administrationsfehlern: Wenn einem Subjekt versehentlich *zu wenig* Rechte gewährt worden sind, wird dies

[6]Wir werden später noch eine dreiwertige Logik kennenlernen, wo dies nicht mehr gilt.

sehr schnell zu einer Beschwerde und zu einer Korrektur des Administrationsfehlers führen. Im umgekehrten Fall ist damit nicht zu rechnen. Daher ist nur der zweite Ansatz akzeptabel; wir setzen ihn daher i. F. stets voraus.

6.5.2 Diskretionäre Zugriffskontrollen

Diskretionäre Zugriffskontrollen (**discretionary access controls**, **DAC**) sind Zugriffskontrollen, durch die der Zugriff zu Objekten auf Basis der Identität von Subjekten und/oder Gruppen, zu denen Subjekte gehören, eingeschränkt wird; die Zugriffskontrollen sind diskretionär (= im Ermessen liegend) insofern, als bestimmte Subjekte, die sog. **Besitzer** eines Objekts, darüber entscheiden können, ob und wie andere Subjekte auf dieses Objekt zugreifen dürfen.

Fast alle Dateiverwaltungssysteme haben diskretionäre Zugriffskontrollen.

Man erkennt leicht folgendes: *Der Schutz der Vertraulichkeit bzw. Integrität von Information kann durch diskretionäre Zugriffskontrollen allein nicht garantiert werden*: Ein Subjekt s_1, welches ein Objekt o_1 lesen darf, kann bei praktisch allen diskretionären Zugriffskontrollmodellen ohne weiteres den Inhalt von o_1 in ein anderes Objekt o_2 kopieren und dort einem anderen Subjekt s_2 zugänglich machen, welches die in o_1 enthaltene Information nicht lesen darf. s_1 kann also Sicherheitsziele verletzen. Die Wirksamkeit von diskretionären Zugriffskontrollen hängt also ganz entscheidend davon ab, dass die Besitzer von Objekten Rechte entsprechend der zu realisierenden *organisatorischen* Sicherheitsstrategie vergeben und dass ebenfalls Lese- bzw. Schreibrechte nur im Sinne der organisatorischen Sicherheitsstrategie ausgenutzt werden; dies entzieht sich aber jeder Kontrolle durch das System; das Betriebssystem kontrolliert hier keinerlei Informationsflüsse.

Trotz dieser Schwächen sind diskretionäre Zugriffskontrollen sehr verbreitet und beliebt: Vorteile sind die einfache Implementierbarkeit und die sehr flexible und dezentrale Rechteverwaltung. Eine sehr häufige Anwendung besteht z. B. darin, sich selbst das Schreibrecht für eigene Dateien zu entziehen, um sie nicht versehentlich zu verändern, obwohl hier eigentlich keine Bedrohung im Sinne von Abschnitt 6.1.1 vorliegt.

6.5.2.1 Zugriffsmodi

Um verschiedene Schutzmechanismen untersuchen zu können, betrachten wir das Konzept der *Schutzdomäne*. Eine Domäne ist eine Menge von (Objekt, Rechte)-Paaren. Jedes Paar spezifiziert ein Objekt und eine Teilmenge der Operationen, die auf einem Objekt ausführbar sind. Die Teilmenge der Operationen hängt vom Typ des Objekts ab. So sind z. B. auf Text-Dateien und Dateiverzeichnissen unterschiedliche Operationen möglich. Eine Domäne entspricht oft einem einzelnen Benutzer und gibt an, was der Benutzer tun darf und was nicht, siehe Abbildung 6.2. Jeder Prozess läuft zu jedem Zeitpunkt in einer Schutzdomäne, d. h. es gibt eine Menge von Objekten, auf die er zugreifen kann. Für jedes dieser Objekte besitzt er eine Menge von Rechten. Prozesse können während der Ausführung auch von Domäne zu Domäne wechseln. Die Domänen müssen nicht disjunkt sein, z. B. in Abbildung 6.2 liegt das Paar (Drucker 1, `w`) in Domane 1 und 2, also kann ein Prozess in Domane 1 oder in Domäne 2 das Objekt Drucker 1 schreiben. In UNIX wird die Domäne eines Prozesses durch seine UID und seine GID definiert. Für jedes gegebene (UID,GID)-Paar

ist es möglich, eine vollständige Liste aller Objekte, z. B. Dateien oder E-/A-Geräte, die durch Spezialdateien repräsentiert werden, zu erstellen, auf die zugegriffen werden kann. Wenn ein Prozess einen Systemaufruf ausführt, dann wechselt er vom Benutzermodus in den Systemmodus. Der Systemmodus hat eine andere Menge von Zugriffen auf andere Objekte als der Benutzermodus. Daher verursacht der Systemaufruf einen Domänenwechsel.

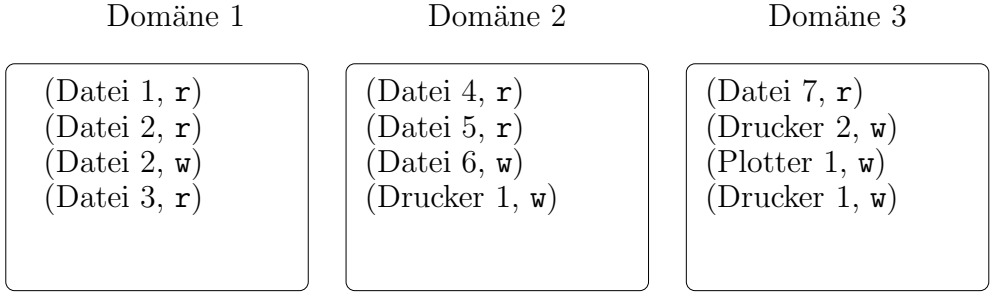


Abbildung 6.2: Drei Schutzdomänen. Es ist möglich, dass das gleiche Objekt z. B. Drucker 1 in zwei Domänen enthalten sein kann.

Die Zahl der verschiedenen Operationen inkl. eventueller Parameter ist i. A. so hoch, dass es unpraktisch wäre, jede einzelne Operation zu kontrollieren. Stattdessen fasst man die Operationen zu sinnvollen Gruppen zusammen und nennt eine solche Gruppe einen **Zugriffsmodus** oder kurz **Modus** (manchmal auch **Zugriffstyp**). Die Menge der Modi ist einheitlich für alle Objekttypen; ggf. ist einem Modus bei einem Objekttyp eben keine einzige Operation zugeordnet. Übliche Modi sind:

`r` **(read):**	lesen den Inhalt eines Objekts
`w` **(write):**	schreiben den Inhalt eines Objekts
`a` **(append):**	hängen an den Inhalt eines Objekts an, nur sinnvoll bei Objekten, deren Typ eine Sequenz von Zeichen, Bytes, Nachrichten oder anderem ist
`x` **(execute):**	laden und ausführen ein Objekt, das ein ausführbares oder interpretierbares Programm enthält
`n` **(navigate):**	navigieren über ein Objekt hinweg
`d` **(delete):**	löschen ein Objekt
`o` **(owner):**	verändern die Zugriffsrechte eines Objekts (s. u.)

Es gibt in manchen Systemen noch weitere Modi, meist aber weniger. Eine große Zahl von Modi erlaubt eine feinere Kontrolle von Rechten. Eine zu hohe Zahl von Modi macht aber für die Benutzer die Administration der Rechte zu aufwändig und unüberschaubar.

Die zu den Modi gehörigen Operationsmengen sind i. d. R. disjunkt. Überschneidungen zwischen Modi, z. B. `w` impliziert `r` und `a`, erhöhen scheinbar den Komfort bei der Administration der Rechte; tatsächlich führen sie aber eher zu Komplikationen, da die Rücknahme von Rechten unklarer und komplizierter werden kann.

6.5.2.2 Der abstrakte Rechtezustand

Zu jedem Zeitpunkt muss immer eindeutig definiert sein, ob der Zugriff eines Subjekts s zu einem Objekt o mittels einer Operation aus einem Modus m erlaubt ist oder nicht. Den persistenten oder transienten Rechtezustand bei diskretionären Zugriffskontrollen kann man sich daher abstrakt und unter Vernachlässigung aller Implementationsaspekte als eine Abbildung

$$\texttt{zugriff} : \mathcal{S} \times \mathcal{O} \times \mathcal{M} \to \mathcal{W}$$

vorstellen, wobei

$\mathcal{S}$ die aktuell vorhandene Menge der Subjekte des Systems,

$\mathcal{O}$ die aktuell vorhandene Menge der Objekte des Systems,

$\mathcal{M}$ die Menge der Modi (diese Menge ist statisch) und

$\mathcal{W}$ die Menge der Werte von Rechtefestlegungen, also $\mathcal{W} = \{\text{erlaubt}, \text{verboten}\}$

ist. Ein Quadrupel (s, o, m, w) aus $\mathcal{S} \times \mathcal{O} \times \mathcal{M} \to \mathcal{W}$ nennen wir eine **Rechtefestlegung**. Wenn $\text{zugriff}(s, o, m) = w$ gilt, dann sagen wir, dass die Rechtefestlegung (s, o, m, w) im System gültig ist.

Eine äquivalente, häufig benutzte Form der Darstellung der Abbildung `zugriff` ist eine **Zugriffskontrollmatrix**; jede Zeile entspricht einem Subjekt, jede Spalte einem Objekt. Der Eintrag für Subjekt s und Objekt o enthält die Menge derjenigen Modi, für die der Zugriff erlaubt ist, also:

$$\text{matrix}(s, o) = \{m \mid \text{zugriff}(s, o, m) = \text{erlaubt}\}$$

Tabelle 6.1 zeigt ein Beispiel für eine derartige Matrix.

	o_1	o_2	o_3	o_4
s_1	owrx	owrx		
s_2			owrx	x
s_3	r	r	r	or

Tabelle 6.1: Eine Zugriffskontrollmatrix.

Sowohl die Abbildung `zugriff` als auch die Zugriffskontrollmatrix sind abstrakte Darstellungen des aktuellen Zustands der Rechtefestlegungen. Eine direkte Implementierung in Form einer Tabelle oder Matrix scheidet natürlich aus: der Zugriff ist normalerweise in den weitaus meisten Fällen nicht erlaubt; daher wären fast alle Einträge in der Matrix leer. Hinzu kommt, dass sich die Menge der Subjekte und Objekte ständig ändern kann; hier müssen die unterschiedlichen Gegebenheiten beim persistenten und transienten Rechtezustand berücksichtigt werden.

Die Implementierungen des Rechtezustands bzw. der diskretionären Zugriffskontrollen kann man grob in zwei Gruppen teilen:

- solche, die die Matrix in *Spalten* aufteilen, also alle zu einem Objekt gehörigen Rechtefestlegungen zusammenfassen, und diese bei dem Objekt speichern; wir nennen sie deshalb **granulatorientiert**[7]. Beispiele sind:

[7] Die Bezeichnung *objektorientiert* wäre naheliegender, sie wird aber im Kontext von Datenbanksystemen und Programmiersprachen bzw. -systemen mit einer anderen Bedeutung so weit verbreitet benutzt, dass sie irritieren würde.

  - Zugriffskontrollisten
  - Schutzbits

  Wie wir später sehen werden, eignen sich die granulatorientierten Verfahren vor allem zur Implementierung des persistenten Rechtezustands.

- solche, die die Matrix in Zeilen aufteilen, also alle zu einem Subjekt gehörigen Rechtefestlegungen zusammenfassen, und diese bei dem Subjekt speichern; wir nennen sie deshalb **subjektorientiert**. Beispiele sind:
  - Profile
  - Capabilities

  Wie wir später sehen werden, eignen sich die subjektorientierten Verfahren vor allem zur Implementierung des transienten Rechtezustands.

6.5.2.3 Granulatorientierte Implementierungen

Bei granulatorientierten Implementierungen kann man leicht herausfinden, welche Subjekte Rechte an einem gegebenen Objekt haben; man kann aber i. A. schlecht herausfinden, welche Rechte ein gegebenes Subjekt hat. Ein Vorteil dieses Ansatzes ist, dass man die zu einer Datei gehörigen Rechtefestlegungen auf dem gleichen Datenträger speichern kann wie die Datei selbst. Daher werden für Dateien bzw. persistente Objekte fast immer granulatorientierte Implementierungen verwendet.

Zugriffskontrollisten. Zu jedem Objekt wird eine Liste geführt. Die Liste enthält alle Domänen, die auf das Objekt zugreifen können. Jede Domäne gehört genau zu einem Objekt. Für jede Domäne wird angegeben, wie der Zugriff erfolgen darf. Diese Liste wird **Zugriffskontrolliste**(**access control list, ACL**) genannt. ACL ist eine Folge von Records der Form:

```
ACL_eintrag = record
   s : domänenidentifikation;
   w : array [ menge_der_modi ] of boolean;
end;
```

Jeder Eintrag gibt für jeden zu einer bestimmten Domäne gehörenden Prozess an, für welche Modi der Zugriff erlaubt ist, siehe Abbildung 6.3, wobei wir die übliche zweiwertige Rechtelogik annehmen und `true` mit *erlaubt* gleichsetzen; bei einer drei- oder mehrwertigen Rechtelogik muss statt `boolean` ein geeigneter Aufzählungstyp verwendet werden. Da *verboten* der Default-Wert ist, enthält die ACL nur solche Einträge, in denen wenigstens für einen Modus m gilt `w[m] = true`. Mit anderen Worten kann die ACL bei Änderungen von Rechtefestlegungen wachsen oder schrumpfen.

Es gibt zwei verschiedene Arten, eine ACL auszuwerten:

1. Man fasst die ACL als *Sequenz* auf und durchläuft diese Sequenz so lange, bis man auf einen Eintrag für ein aktives Subjekt des Prozesses stößt. Genau dieser Eintrag und kein anderer wird zur Bestimmung der Rechte des Prozesses verwendet. Wenn also ein Recht bei diesem Eintrag nicht gewährt wird, in

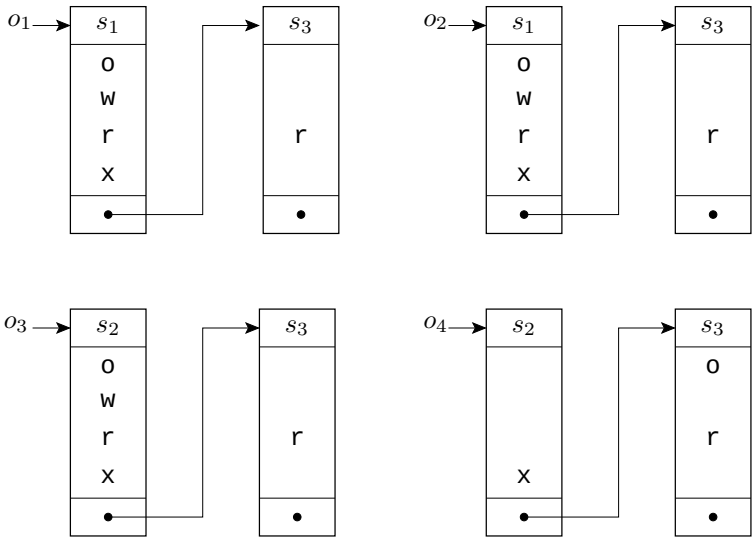


Abbildung 6.3: Zugriffskontrollliste für Objekte in Tabelle 6.1, wobei `o,r,...` genau `w[o]=true`, `w[r]=true`, ... bedeuten.

einem späteren Eintrag jedoch einem anderen aktiven Subjekt gewährt wird, dann hat der Prozess dieses Recht letztlich *nicht*. Die Rechte der aktiven Subjekte werden hier also nicht *addiert*. Die Reihenfolge der Einträge in der ACL ist relevant für die Rechte eines Prozesses.

2. Man fasst die ACL als *Menge* auf, sucht alle Einträge für aktive Subjekte des Prozesses und kombiniert die in diesen Einträgen vorgefundenen Rechtewerte. Allerdings gibt es für den Rechtewert *verboten* nun zwei Interpretationen:

   (a) explizit verboten

   (b) nicht explizit erlaubt

   Im ersten Fall darf der Zugriff dem Prozess nicht erlaubt werden, wenn er für wenigstens ein aktives Subjekt *verboten* ist. Im zweiten Fall wird der Zugriff dem Prozess erlaubt, wenn er für wenigstens ein aktives Subjekt *erlaubt* ist; die Rechte der aktiven Subjekte werden hier also *addiert*. Man erkennt leicht, dass die erste Interpretation i. A. zu streng ist und dass man die zweite Interpretation auf alle Fälle benötigt.

   Man kann nun aber einem Subjekt nicht mehr generell einen Zugriff verbieten, genauer gesagt jedem Prozess, bei dem dieses Subjekt aktiv ist. Um dieses Problem zu lösen, muss man einen dritten Rechtewert *explizit verboten* einführen, der wie oben angegeben zu interpretieren ist.

Benannte ACLs. Man kann ACLs auch als eigenständige Einheiten auffassen, ihnen einen Namen geben und mehreren Objekten die gleiche benannte ACL zuweisen. Dies hat zwei Vorteile:

1. ACLs können recht lang werden und man spart auf diese Weise Platz, sofern - was häufig zutrifft - mehrere Objekte die gleiche ACL haben.

2. Bei gleichförmigen Änderungen der Rechte an den Objekten muss nur die benannte ACL geändert werden und nicht die ACL jedes einzelnen Objekts. Eine nicht gleichförmige Änderung der Rechte an den Objekten mit einer gemeinsamen ACL wird allerdings komplizierter.

Benannte ACLs führen sozusagen einen Indirektionsschritt bei der Bildung des Rechtezustands ein: eine ACL gehört nicht mehr direkt zu einem Objekt, sondern erst indirekt über den ACL-Namen. Dies ist zugleich der Hauptnachteil von benannten ACLs:

- Sie erfordern zusätzliche Werkzeuge und Systemaufrufe zur Verwaltung und Handhabung der benannten ACLs.
- Bei der Festlegung der ACL eines Objekts sind oft zusätzliche Benutzerkommandos nötig.
- Änderungen am Rechtezustand durch Änderung der benannten ACL oder der Menge der zur ACL gehörigen Objekte können leichter zu ungewollten Effekten führen.

Schutzbits. Schutzbits kann man als eine stark eingeschränkte Form von ACLs auffassen. Sie treten z. B. im Dateisystem von UNIX auf. Angegeben werden dort zwei Subjektidentifizierer und drei Gruppen zu je drei Bits, als ein Beispiel siehe das folgende Verzeichnis:

```
 4  -rwxr-xr--   1 mueller  bteam   1730 Jul 15 10:53  myprogram
60  -rw-r-----   1 fischer  bteam  29710 Mär 20  1996  flip
 8  -rw-------   2 mueller  bteam   4038 Jul 27 18:46  entwurf.tex
 2  drwxrwxr--   2 mueller  bteam    512 Jul 27 16:43  archiv
 2  drwx------   4 mueller  bteam    512 Mai 12 09:01  privat
```

Ganz rechts stehen die *Namen* der fünf in diesem Verzeichnis enthaltenen Objekte; nach ihnen ist die Ausgabe alphabetisch sortiert. In der vierten Spalte sind die Besitzer der Objekte angegeben, z. B. in dem obigen Verzeichnis ist `mueller` der Besitzer der Dateien `myprogram`, `entwurf.tex`, der Verzeichnisse `archiv` und `privat`; er hat als einziges Subjekt das owner-Recht, wenn man einmal vom *super user* absieht, für den alle Rechteprüfungen ausgeschaltet sind. Das zweite Subjekt ist immer eine Gruppe, die in der fünften Spalte steht, z. B. `bteam`. Die Benutzer `mueller` und `fischer` müssen nicht unbedingt Mitglieder von `bteam` sein.

Jede Gruppe von Bits gibt für ein bestimmtes Subjekt an, ob dieses das Lese-, Schreib- und Ausführungs- bzw. Navigationsrecht für das Objekt hat, siehe Tabelle 6.2. Die erste Bitgruppe gilt für den Besitzer, die zweite für die angegebene Gruppe und die dritte für die Menge aller übrigen Benutzer, die man als eine implizit vorhandene Gruppe auffassen kann. Man kann die Schutzbits also als eine ACL mit drei Einträgen auffassen; diese wird als *Sequenz* ausgewertet.

Schutzbits sind zwar leichter implementierbar als allgemeine ACLs, realisieren aber das allgemeine DAC-Modell nicht vollständig und sind in der Praxis nicht zufriedenstellend.

Binär	Symbolisch	Erlaubte Dateizugriffe
111000000	rwx------	Eigentümer darf lesen, schreiben, ausführen
111111000	rwxrwx---	Eigentümer, Gruppe dürfen lesen, schreiben, ausführen
110100100	rw-r--r--	Alle dürfen lesen, nur Eigentümer darf schreiben
111101101	rwxr-xr-x	Eigentümer darf alles, andere dürfen lesen und ausführen
000000000	---------	Keiner hat Zugriff

Tabelle 6.2: Beispiele für Dateizugriffsschutzbits.

Modifikation von ACLs. Der oben definierte abstrakte Rechtezustand nahm auf eine Menge von Modi Bezug, die u. a. den Modus `o` (owner) enthielt. Man fasst hier die zu einem Objekt gehörenden Rechtefestlegungen also, etwas lax ausgedrückt, die ACL als Teil des Objekts und Änderungen an diesen als eine Operation auf dem Objekt auf. Das owner-Recht berechtigt zu solchen Änderungen. Ein Subjekt mit owner-Recht für ein Objekt nennen wir **Besitzer** des Objekts.

Der owner-Modus ist nur bei granulatorientierten Implementierungen sinnvoll: Bei subjektorientierten. Implementierungen sind die mit dem owner-Modus zusammenhängenden Änderungsoperationen nicht ohne weiteres implementierbar.

In vielen Systemen hat jedes Objekt immer genau einen (exklusiven) Besitzer, der auch die Kosten des Objekts tragen muss. In anderen Systemen kann ein Objekt mehrere Besitzer haben; es muss aber immer wenigstens einen Besitzer haben, weil sonst keine Änderungen an der ACL mehr möglich wären.

Wieder andere Besitzer-Konzepte basieren auf der Philosophie, dass ein Rechner und seine Objekte einem Unternehmen gehören und dass alle Berechtigungen zunächst einmal zentral vom Systemadministrator im Auftrag des Unternehmens vergeben werden. Da dieser sich nicht um alle Kleinigkeiten kümmern kann, **delegiert** er Teile seiner Rechte an andere Benutzer, behält aber trotzdem seine Rechte und hat im Zweifelsfall sogar Vorrang. Er kann z. B. das Besitzerrecht für einen ganzen Dateibaum, der die Daten eines Projekts enthält, an den Leiter dieses Projekts delegieren. Der Projektleiter mag nun seinerseits die Verantwortung für einen Teil dieser Objekte an einen Teilprojektleiter weiterdelegieren. Im allgemeinen entstehen so **Delegationsketten**. Für ein Objekt können mehrere Delegationsketten vorliegen, sofern nicht-exklusive Delegationen erlaubt sind. Das obige abstrakte Modell des Rechtezustands muss auf alle Fälle passend erweitert werden, wenn man das Delegieren von Rechten unterstützen will. Delegationsketten sind bisher in Betriebssystemen nur sehr selten realisiert worden, häufig dagegen in Datenbanksystemen.

6.5.2.4 Subjektorientierte Implementierungen

Bei subjektorientierten Implementierungen kann man leicht herausfinden, welche Rechte ein Subjekt hat, und die Rechte eines Subjekts kopieren; man kann aber nur sehr schwer herausfinden, welche Rechte für ein Objekt vorhanden sind oder beispielsweise allen oder einzelnen Subjekten Rechte an diesem Objekt entziehen.

Profile. Wenn die Zugriffsmatrix zeilenweise zerlegt wird, dann wird jedem Prozess eine Liste von Objekten zugeordnet, auf die er zugreifen kann. Dabei wird

angegeben, welche Operationen auf jedem Objekt erlaubt sind. Also wird ein Benutzerprofile definiert. Ein Profil ist analog zu einer Zugriffskontrolliste eine Folge von Records der Form:

```
profileintrag = record
   o : objektidentifikation;
   w : array [ menge_der_modi ] of boolean;
end;
```

Jeder Eintrag gibt die Rechte für ein bestimmtes Objekt an. Einträge, siehe Abbildung 6.4, in denen nur der Rechtewert *verboten* auftritt, werden aus dem Profil entfernt, d. h. der default-Wert ist auch hier *verboten*.

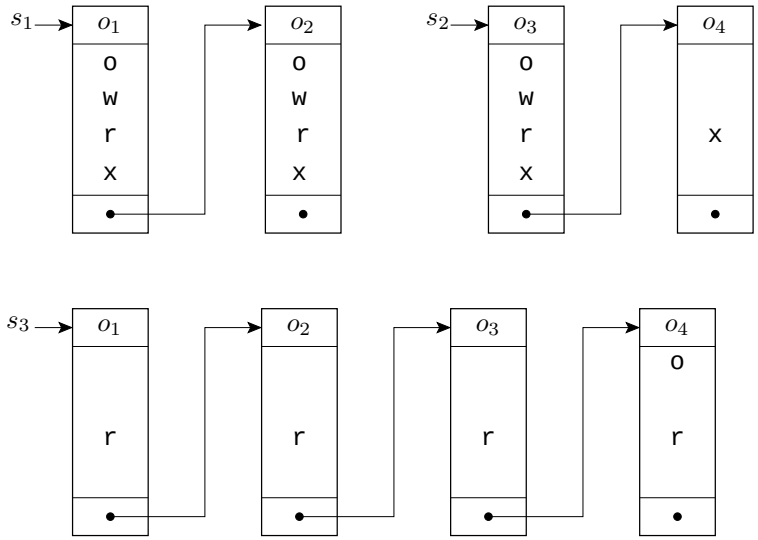


Abbildung 6.4: Profile für Subjekte in Tabelle 6.1.

Profile sind zur Realisierung des persistenten Rechtezustands wesentlich schlechter geeignet als ACLs:

- Die Zahl der Objekte ist üblicherweise sehr hoch, jedenfalls wesentlich höher als die Zahl der Subjekte. Die Objektidentifizierer sind daher relativ lang, die Suche nach einem Objektidentifizierer im Profil dauert relativ lange.

- An einem Objekt haben i. d. R. nur wenige Subjekte Rechte, insb. wenn man geeignete Gruppen bildet. Hingegen hat ein Subjekt meist Rechte an vielen Objekten. Dabei ist es für die Mehrzahl der dynamisch erzeugten Objekte nicht ohne weiteres möglich, Gruppierungen analog zu Benutzergruppen zu finden bzw. zu verwalten, die den Zustand wesentlich vereinfachen würden.

- Das Erzeugen von Objekten, an denen viele Subjekte Rechte haben sollen, und das Löschen von Objekten, an denen viele Subjekte Rechte haben, ist sehr aufwändig.

Profile sind also nur dann brauchbar, wenn die Zahl der Objekte, an denen ein Subjekt Rechte hat, klein ist. Wenn die Menge dieser Objekte statisch ist, kann

man das Profil sogar als einen einzigen Record bzw. Array realisieren und sich die Speicherung der Objektidentifizierer sparen: dies ist z. B. bei den Ressourcen der Fall, die durch Benutzerprofile kontrolliert werden, und bei der Parameterliste des Programms, das in einem Prozess ausgeführt wird. Aus diesen Überlegungen heraus werden subjektorientierte Implementierungen oft bei der Realisierung des transienten Rechtezustands angewandt.

Capabilities. Eine Capability ist aus Sicht der Zugriffskontrollen im Prinzip ein Profileintrag. Eine Menge von Capabilities ist also in bezug auf den dargestellten Rechtezustand äquivalent zu einem Profil. Der entscheidende Unterschied liegt in der *Adressierung* der Objekte bzw. im Typ der Objektidentifikation innerhalb des Profileintrags. Eine Capability ist vergleichbar mit einer *opaken* Variablen oder einem *geschützten Pointer*: der *Besitz* der Capability ermöglicht überhaupt erst Zugriff auf das zugehörige Objekt, man spricht deshalb oft von einem *Capability-Ticket*. Ohne die Capability ist das Objekt für den Prozess überhaupt nicht sichtbar. Das heißt, um ein Objekt `o` zugreifen zu können, muss der Prozess die Capability für `o` als Parameter angeben. Die Capability ermöglicht also direkten Zugriff auf das Objekt, ohne Suche nach einem Objektidentifizierer in einer langen Liste von Profileinträgen.

Ein Capability-Ticket legt für einen Benutzer zulässige Objekte und Operationen fest. Jeder Benutzer verfügt über eine Reihe von Tickets und ist berechtigt, diese anderen zu verleihen und zu geben. Da Tickets im System verteilt sein können, stellen sie im Vergleich zu Zugriffskontrollliste ein größeres Sicherheitsproblem dar. Die Tickets müssen durch Hard- oder Software vor Manipulation durch Benutzerprogramme geschützt werden. Es gibt drei bekannten Schutzmethoden. Die erste Möglichkeit ist eine Hardwarearchitektur, in der jedes Speicherwort ein zusätzliches Bit hat, das angibt, ob das Wort eine Capability enthält oder nicht. Das Bit kann nur von Programmen geändert werden, die im Systemmodus laufen. Die zweite Möglichkeit ist, eine Capability innerhalb des Betriebssystems zu halten, wo sie für Benutzer nicht zugänglich ist. Allerdings muss dann in den Benutzerprogrammen so etwas wie ein Zeiger auf eine Capability oder eine Nummer einer Capability benutzt werden. Ein Prozess kann sagen: *Lies 2 kByte von der Datei, auf die Capability 5 zeigt*. Die Adressierung ist ähnlich der Adressierung der Blöcke in i-node. Die dritte Methode ist, eine Capability im Benutzeradressraum zu gespeichern. Man kann dann sie durch Verschlüsselung vor Veränderungen schützen.

Ein gutes Beispiel für eine Capability der zweiten Art ist ein Dateikontrollblock: dieser wird beim erfolgreichen Öffnen einer Datei angelegt und enthält einen Eintrag darüber, in welchem Modus (read, write, append) eine Datei geöffnet wurde, sowie diverse andere, hier nicht interessierende Daten wie Adressen der zugehörigen Sektoren, Hauptspeicherpuffer usw.. Wenn die Datei zum Lesen geöffnet wurde und der Prozess versucht, in die Datei zu schreiben, dann wird dieser Versuch zurückgewiesen. Der Dateikontrollblock repräsentiert also eine Menge von Rechtefestlegungen: in diesen ist das Subjekt stets der Prozess, der die Datei geöffnet hat, und das Objekt die stets geöffnete Datei und nicht etwa die Datei als persistentes Objekt! Für jeden Zugriffsmodus ergibt sich aus dem Eintrag im Dateikontrollblock, ob der zugehörige Wert *erlaubt* bzw. *verboten* ist.

Beim Öffnen der Datei wird natürlich deren ACL nach einem der schon oben beschriebenen Verfahren ausgewertet und der Dateikontrollblock, also die Capability, nur dann erzeugt, wenn die Auswertung der ACL positiv verlief. Bei späteren

Zugriffen zur geöffneten Datei wird deren ACL *nicht* mehr erneut ausgewertet.

Man erkennt an diesem Beispiel auch, dass sich Capabilities leicht kopieren lassen und dass die Capability Teil eines Mechanismus zur direkten Adressierung von Objekten ist. Es ist daher etwas irreführend, wenn in manchen Quellen eine Zeile in der Zugriffskontrollmatrix als *Capability-Liste* bezeichnet wird (analog zu einer ACL), denn dies suggeriert, dass diese Liste bei jedem Zugriff zu einem Objekt durchsucht werden müsste. Die Bezeichnung *Capability-Liste* ist aber in solchen Systemen gerechtfertigt, in denen Mengen von Capabilities in Form einer Liste, deren Ordnung der Einträge in Wirklichkeit keine Rolle spielt, permanent in Dateien gespeichert werden können, damit sie später von Subjekten (also Prozessen) ausgenutzt werden können.

6.5.3 Informationsflusskontrollen

Eine prinzipielle Schwäche diskretionärer Zugriffskontrollen wurde schon oben erwähnt: sie ermöglichen *keine Kontrolle* des Informationsflusses und ihre Wirksamkeit hängt wesentlich davon ab, dass Subjekte ihre Rechte nicht missbrauchen, sondern nur so einsetzen, dass die Sicherheitsziele der organisatorischen Sicherheitsstrategie nicht verletzt werden. Diese Schwäche führte zur Entwicklung einer anderen Klasse von Zugriffskontrollmodellen, die es erlauben, den Informationsfluss innerhalb eines Rechners direkt zu kontrollieren, und die wir **Informationsflusskontrollen** (**mandatory access controls**) nennen. Die Bezeichungen für diese Modelle sind nicht einheitlich; andere Bezeichungen sind **globale Zugriffskontrollen** und **Sicherheitsklassenmodell** (**multi-level security**).

Diese Modelle wurden in erster Linie aufgrund von Anforderungen im Bereich militärischer oder geheimdienstlicher Anwendungen entwickelt; sie sind teilweise im zivilen Bereich anwendbar, allerdings ist seit einigen Jahren eine heftige Debatte im Gange, wie breit denn dieser Anwendungsbereich eigentlich ist und ob nicht noch ganz andere Modelle benötigt werden, übrigens auch im militärisch / geheimdienstlichen Bereich. Unabhängig von der Frage der Anwendbarkeit sind aber die Informationsflusskontrollen in zweierlei Hinsicht interessant:

1. Nur für diese Modelle lässt sich mit Hilfe mathematischer Modelle *beweisen*, dass sie die zunächst informell definierten Sicherheitssziele Vertraulichkeit und Integrität von Information realisieren. Grundlegende Arbeiten waren [BeL73, Bi77]. Aus Platzgründen können wir diese Theorien hier nicht vorstellen; hierzu sei auf [Pk91, Pc96, De82] verwiesen.

2. Informationsflusskontrollen werden inzwischen in den höheren Klassifikationsstufen für sichere Systeme gefordert [TCSEC85, ITSEC91]. Bei Rechnerbeschaffungen im öffentlichen bzw. militärischen Bereich werden z. T. Rechner dieser höheren Sicherheitsstufen vorgeschrieben. Dies hat dazu geführt, dass in einigen kommerziell erhältlichen Betriebssystemen Informationsflusskontrollen vorhanden sind; die Tendenz ist steigend.

Informationsflusskontrollen werden übrigens fast immer ergänzend zu diskretionären Zugriffskontrollen benutzt; dies nehmen wir auch i. F. an.

Grundidee. Wie schon oben erwähnt, liegt das zentrale Problem bei diskretionären Zugriffskontrollen darin, dass nur der Zugriff zu Datenbehältern kontrolliert wird, nicht hingegen zu der darin enthaltenen Information: ein Prozess kann ohne weiteres z. B. den Inhalt eines *geheimen* Behälters in einen öffentlich zugänglichen Behälter kopieren; die Zugriffsbeschränkungen der von einem Prozess *erzeugten* Daten hängt in keiner Weise von den Zugriffsbeschränkungen der *gelesenen* Daten ab. Exakt an diesem Punkt setzen die Informationsflusskontrollverfahren an: die zentrale Idee ist, Zugriffsbeschränkungen der von einem Prozess gelesenen Daten auf die erzeugten Daten zu *vererben*.

Wir beschränken uns hier auf solche Verfahren, bei denen diese Grundidee noch vereinfacht bzw. vergröbert wird: es wird nicht überprüft, ob *tatsächlich Information* von einem Objekt über einen Prozess in ein anderes Objekt fließt, sondern nur überprüft, ob überhaupt irgendwelche Daten fließen und damit *potentiell* Information fließt. Wenn z. B. ein Prozess eine Datei D_1 zum Lesen geöffnet hat und später eine Datei D_2 zum Schreiben, dann gilt bereits alle Information als von D_1 nach D_2 übertragen, selbst wenn der Prozess überhaupt keine Daten direkt oder in verarbeiteter Form von D_1 nach D_2 kopiert hat. Man könnte diese Verfahren daher auch als *Datenflusskontrollen* bezeichnen.

6.5.3.1 Modelle auf Basis von Sicherheitsklassen

Es sind nun verschiedene Verfahren denkbar, durch die Zugriffsbeschränkungen vererbt werden können. Die wichtigsten bisher realisierten Informationsflusskontrollverfahren beruhen auf der Annahme, dass man den Grad der Vertraulichkeit und der Integrität bzw. Vertrauenswürdigkeit von Daten in Form von (**Sicherheits-**) **Klassen**[8] (oder -Stufen) klassifizieren kann und dass man i. W. unzulässige Informationsflüsse zwischen verschiedenen Klassen verhindern muss.

Ein typisches Beispiel von **Vertraulichkeitsklassen** ist {offen (unklassifiziert), vertraulich, geheim, streng geheim}. Zwei Beispiele von Integritätsklassen für einen Text mit mathematischen Formeln sind {formeln_ng, formeln_ok} und {deutsch_ng, deutsch_ok}, die ausdrücken sollen, dass in diesem Text die Formeln bzw. das Deutsch noch nicht geprüft bzw. in Ordnung sind.

Labels. Jedes Objekt und jedes Subjekt wird nun genau einer Vertraulichkeitsklasse und genau einer Integritätsklasse zugeordnet. Diese Klassifizierungen bezeichnet man auch als **Labels** (bzw. **Marken**) der Subjekte bzw. Objekte. Die Labels drücken folgendes aus:

- bei einem Objekt: der derzeitige Inhalt ist (potentiell) so vertraulich bzw. wenigstens so korrekt wie durch die entsprechende Klasse angegeben.
- bei einem Subjekt:
  1. die von diesem Subjekt erzeugten Daten sind (potentiell) so vertraulich bzw. wenigstens so korrekt wie durch die entsprechende Klasse angegeben.
  2. dieses Subjekt ist berechtigt,

[8] Man beachte den Unterschied zwischen den Begriffen Sicherheitsklasse und Sicherheitsstufe: Sicherheitsklassen klassifizieren die Vertraulichkeit oder Integrität von Daten, Sicherheitsstufen klassifizieren die Sicherheitsfunktionen eines Rechners bzw. Betriebssystems.

- Daten dieser und kleinerer Vertraulichkeitsklassen zu lesen und
- Daten dieser und kleinerer Integritätsklassen zu erzeugen bzw. zu verändern.

Das Subjekt ist *nicht* berechtigt, Daten in anderen als den vorstehend angegebenen Klassen zu lesen oder zu verändern.

Man beachte, dass für lesende Zugriffe nur die Vertraulichkeitsklassen relevant sind, für schreibende Zugriffe nur die Integritätsklassen.

Wir nehmen an, dass nur Benutzer klassifiziert werden, für Subjekte vom Typ Gruppe oder Programm macht dies nicht viel Sinn, und dass ein *Prozess beim Start das gleiche Vertraulichkeits- und Integritätslabel erhält wie der Benutzer, für den er läuft.* Der Benutzer kann auch beim login *kleinere* Labels als das maximal für ihn zulässige wählen und damit innerhalb dieses Prozesses auf einen Teil seiner Rechte verzichten. Objekte, die der Prozess erzeugt, erhalten die beiden Labels des Prozesses.

Zugriffsregeln. Ziel ist es nun, zu verhindern, dass ein Subjekt Daten aus anderen als den explizit zugelassenen Klassen lesen bzw. schreiben kann. Wir betrachten zunächst einmal die Vertraulichkeit näher. Um das gesteckte Ziel zu erreichen, müssen folgende Regeln eingehalten werden:

einfache Geheimhaltungsbedingung: Ein Prozess darf ein Objekt nur lesen, wenn es der gleichen oder einer niedrigeren Klasse angehört als der Prozess.

***-Eigenschaft:** Ein Prozess darf ein Objekt nur schreiben, wenn es der gleichen oder einer höheren Klasse angehört als der Prozess.

Also dürfen Prozesse nach unten lesen und nach oben schreiben, aber nicht umgekehrt, siehe Abbildung 6.5. Ohne die *-Eigenschaft könnte ein Prozess den Inhalt eines geheimen Objekts in ein offenes Objekt kopieren und so die geheimen Daten unberechtigten Subjekten zugänglich machen. Die *-Eigenschaft verhindert, dass die Informationen von einer höheren Klasse auf eine niedrige fließen.

Ruhe-Prinzip: Ein Prozess kann die Vertraulichkeitsklasse eines Objekts nicht verändern.

Das Ruhe-Prinzip gilt so nur für zerstörbare Objekte, z. B. Dateien, die, nachdem die enthaltenen Daten nicht mehr benötigt werden, völlig aufgelöst werden und nicht mehr existieren.

Permanente Objekte wie Plattensektoren oder Hauptspeicherbereiche können dagegen nicht zerstört werden, sondern werden nacheinander für verschiedene Zwecke benutzt. Sie wechseln also durch erneute Belegung und spätere Freigabe zwischen den Zuständen *benutzt* und *unbenutzt*. Das Ruhe-Prinzip gilt für solche Objekte nur im Zustand *benutzt*. Nach der Freigabe eines Objekts enthält dieses noch die vorher darin vorhandenen Daten. Diese müssen am besten sofort, spätestens vor einer erneuten Belegung des Objekts gelöscht werden, d. h. durch Daten überschrieben werden, die keine Rückschlüsse auf den früheren Inhalt erlauben. Solange die alten Daten nicht gelöscht sind, darf das Objekt nicht zugreifbar sein.

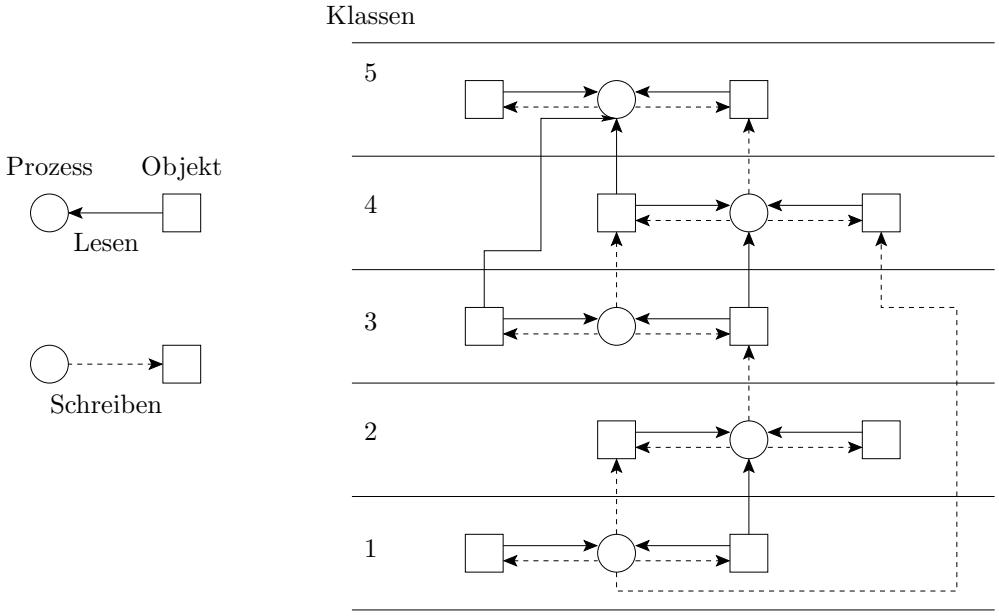


Abbildung 6.5: Multilevel-Sicherheit. Ein durchgezogener Pfeil von einem Objekt auf einen Prozess bedeutet, dass der Prozess das Objekt lesen darf. Ein gestrichelter Pfeil von einem Prozess auf ein Objekt bedeutet, dass der Prozess auf das Objekt schreiben darf.

Kommunikationsregel: Sofern Prozesse miteinander kommunizieren können, darf ein Prozess P_1 nur dann einem Prozess P_2 Daten senden, wenn die Vertraulichkeitsklasse von P_1 nicht höher d. h. gleich oder niedriger als die von P_2 ist.

Für die Integritätssicherung von Daten gelten Regeln, die analog zur einfachen Geheimhaltungsbedingung, zur *-Eigenschaft, zum Ruhe-Prinzip und zur Kommunikationsregel gebildet werden. Das Problem der Geheimhaltung des Inhalts von wiederbenutzbaren Objekten nach deren Freigabe entfällt hier.

Gleitende Labels. Das oben vorgestellte Schema ist recht unflexibel, wenn man mit Objekten unterschiedlicher Klassifizierung arbeiten muss. Lästig ist zunächst einmal, dass ein Prozess nur Objekte mit der (hohen) Vertraulichkeitsklasse des Benutzers erzeugt. Dies kann man zwar dadurch vermeiden, dass man z. B. beim login ein niedrigeres Label für den Prozess wählt, verliert aber dann das Recht, mit diesem Prozess Objekte höherer Klassen zu lesen. Wenn man also versehentlich ein zu niedriges Label für einen Prozess gewählt hat, muss man ihn ggf. abbrechen und mit einem höheren Label mit der Arbeit wieder ganz von vorne beginnen.

Ein anderes Problem ist, dass ein Prozess, der vertrauliche Daten in ein Objekt hineinschreiben will, dies nicht direkt tun kann, wenn hierfür das Vertraulichkeitslabel des Objekts angehoben werden müsste, was durch das Ruhe-Prinzip verhindert wird.

Die Lösung dieser Probleme besteht in einer Erweiterung des ursprünglichen Modells, wonach Labels von Prozessen oder Objekten automatisch bei Vertraulichkeit erhöht bzw. bei Integrität gesenkt werden, sofern dies wegen der *-Eigenschaft erforderlich ist. Für Vertraulichkeitslabels gelten folgende Regeln:

- Ein Prozess darf ein Objekt genau dann lesen, wenn dessen Vertraulichkeitslabel nicht höher als das Vertraulichkeitslabel des Benutzers ist, für den der Prozess läuft.

- Wenn ein Prozess mit Vertraulichkeitslabel L_1 ein Objekt mit Vertraulichkeitslabel L_2 liest und wenn L_2 höher als L_1 ist, dann wird das Vertraulichkeitslabel des Prozesses automatisch auf das kleinste Label angehoben, das nicht niedriger als das vorherige Vertraulichkeitslabel des Prozesses und das Vertraulichkeitslabel des gelesenen Objekts ist.

- Wenn ein Prozess mit Vertraulichkeitslabel L_1 ein Objekt mit Vertraulichkeitslabel L_2 schreibt und wenn L_1 höher als L_2 ist, dann wird das Vertraulichkeitslabel des Objekts automatisch auf das kleinste Label angehoben, das nicht niedriger als das Vertraulichkeitslabel des Prozesses und das vorherige Vertraulichkeitslabel des geschriebenen Objekts ist. Man beachte, dass hier das Ruhe-Prinzip verletzt wird, allerdings offensichtlich ohne die Gefahr, dass die Vertraulichkeit von Information verletzt wird.

Für gleitende Integritätslabels gelten analoge Regeln.

Bei gleitenden Labels kann versehentlich ein Objekt geheimer oder weniger integer gemacht werden, indem sein Inhalt unverändert oder nur marginal verändert zurückgeschrieben wird. Deshalb muss das Gleiten von Labels abschaltbar sein.

Vertrauenswürdiges Ändern von Labels. Wir haben oben schon das Problem erwähnt, dass die Daten in einem System tendenziell immer geheimer werden. Insbesondere kann leicht der Fall eintreten, dass der Inhalt von Objekten weniger geheim ist, als es das Label des Objekts ausdrückt. Eine Ursache hierfür ist, dass bereits potentielle Informationsflüsse zur Anhebung von Labels führen können (bei gleitenden Labels), während ein tatsächlicher Informationsfluss gar nicht stattgefunden haben muss. Eine andere Ursache ist, dass die geheimen Anteile manuell aus dem Objektinhalt entfernt worden sein können, das Betriebssystem dies aber nicht erkennen kann. Für Integritätslabels gelten diese Überlegungen analog.

Um nun zu erreichen, dass die Labels eines Objekts wieder mit dessen tatsächlichem Inhalt übereinstimmen, benötigt man eine Möglichkeit, manuell Vertraulichkeitslabels herabzusetzen bzw. Integritätslabels heraufzusetzen. Das Recht zu solchen Änderungen (**trusted downgrade** bzw. **upgrade**) wird man nur speziellen, besonders vertrauenswürdigen Personen bzw. Subjekten gewähren (z. B. sog. Sicherheitsbeauftragten), die z. B. durch eine spezielle Rolle im System repräsentiert sein können. Man beachte, dass solche Änderungen die *automatisierte Sicherheitsstrategie verletzen(!), nicht hingegen die organisatorische Sicherheitsstrategie*, denn man unterstellt hier, dass die vertrauenswürdigen Personen die organisatorische Sicherheitsstrategie nicht verletzen.

Eine etwas überraschende Konsequenz hieraus ist, dass auch die Informationsflussmodelle zumindest teilweise darauf bauen müssen, dass die Benutzer ihre Rechte nicht missbrauchen; dies hatten wir als Hauptmangel der diskretionären Zugriffskontrollen erkannt und es war ein Hauptmotiv für die Erfindung der Informationsflusskontrollen. Im Unterschied zu den diskretionären Zugriffskontrollen kann hier

allerdings die Zahl der Benutzer, die vertrauenswürdig sein müssen, sehr klein gehalten werden, d. h. in dieser Hinsicht ist doch ein wesentlicher Fortschritt erzielt worden.

6.5.3.2 Andere Informationsflusskontrollen

Die Modelle auf Basis von Sicherheitsklassen sind die ältesten und am weitesten verbreiteten Informationsflusskontrollmodelle. Wie schon früher erwähnt, wird allerdings vielfach bezweifelt, ob sie sehr breit anwendbar sind, siehe [BiB89, BrN89, McMN90]. Es wurden daher in letzter Zeit mehrere neuartige Zugriffskontrollmodelle vorgeschlagen, die stärker an konkreten Anwendungsbereichen (Justiz, Medizin usw.) orientiert sind oder die versuchen, das Prinizip der informationellen Selbstbestimmung im Zugriffskontrollmodell zu verankern. Diese Modelle sind allerdings derzeit eher noch Forschungsgegenstand.

6.6 Zusammenfassung

Diese Kurseinheit behandelte den Problemkreis Sicherheit. Zunächst stellten wir fest, dass Sicherheit nicht isoliert auf den Rechner oder das Betriebssystem behandelt werden kann, sondern dass das gesamte organisatorische Umfeld mit einbezogen werden muss. Es ist daher zu trennen zwischen einer organisatorischen Sicherheitsstrategie und dem Anteil von dieser, der automatisiert (speziell durch das Betriebssystem) realisiert werden kann. Ferner sahen wir, dass die verschiedenen Schichten eines Rechnersystems (Hardware, Betriebssystem, ggf. Datenbanksystem, Applikation) bzgl. der Realisierung der automatisierten Sicherheitsstrategie zusammenwirken müssen. Vor allem müssen tiefere Schichten den darüberliegenden ausreichende Mittel, also Sicherheitsfunktionen, zur Verfügung stellen, damit diese speziellere Sicherheitsstrategien realisieren können.

Die Sicherheitsfunktionen, die die höheren Schichten von einem Betriebssystem erwarten, variieren sehr stark abhängig von den Sicherheitszielen der konkreten Anwendung, der Art der benutzten Rechner und vielen anderen Faktoren. Eine gewisse Orientierung liefern seit neuem entwickelte Sicherheitsklassifikationen. Typischerweise erwartet man von einem Betriebssystem mehr oder weniger leistungsstarke Sicherheitsfunktionen in folgenden Bereichen: Identifikation, Authentisierung, Rechteverwaltung, Rechteprüfung, Beweissicherung, Speicherschutz und Datenübertragungssicherung.

Basis oben aufgezählten Funktionen sind Speicherschutzverfahren, Schutz von Schnittstellen durch Betriebsmodi der CPU usw., die durch die Hardware unterstützt sind und die zunächst den Betriebssystemkern und die Anwendungen untereinander schützen.

Ein grobes Raster, durch das die Aktionen eines Benutzers generell eingeschränkt werden können, sind Benutzerprofile. Im allgemeinen benötigt man jedoch feinkörnigere Kontrollen, bei denen für einzelne Subjekte und Objekte festgelegt werden kann, welche Zugriffe erlaubt sind.

Üblich sind vor allem diskretionäre Zugriffskontrollen, bei denen die Entscheidung über die Rechte an einem Objekt bei dem sog. Besitzer des Objekts liegen. Die wichtigsten Implementierungsformen sind Zugriffskontrollisten (vor allem für

den persistenten Rechtezustand) und Profile bzw. Capability-Listen (vor allem für den transienten Rechtezustand).

Diskretionäre Zugriffskontrollen können keine unerwünschten Informationsflüsse verhindern. Dies kann durch Informationsflusskontrollen verhindert werden. Die wichtigste Form von Informationsflusskontrollen sind solche auf Basis von Vertraulichkeits- und Integritätslabels von Subjekten und Objekten, bei denen unerwünschte Informationsflüsse infolge der *-Eigenschaft verhindert werden.

Literatur

[BeL73] D.E. Bell, and J. La Padula. *Secure computer systems: mathematical foundations*. MITRE Corp. MTR-2547, vol. 1, Bedford MA, 1973.

[Bi77] K.J. Biba. *Integrity considerations for secure computer systems*. ESD-TR-76-372, MITRE Corp. MTR-3153, Bedford, MA, 1977.

[BiB89] J. Biskup, and H.H. Brüggemann. *The personal model of data - towards a privacy oriented information system*. In *Proc. 5th Intl. Conf. Data Engineering*, Los Angeles, IEEE Computer Society Press, pp. 348-355, 1989.

[Bu00] J. Buchmann. *Einführung in die Kryptographie, dritte erweiterte Auflage*. Springer, 2000.

[BrN89] D.F.C. Brewer, and M.J. Nash. *The chinese wall security policy*. In *Proc. IEEE Symp. on Security and Privace*, Oakland, California, pp. 206-214, 1988.

[BSI92] Bundesamt für die Sicherheit in der Informationstechnik. *Kriterien für die Bewertung der Sicherheit von Systemen in der Informationstechnik*, 1992. http://www.bsi.de/literat/kriterie.htm

[De82] D.E. Denning. *Cryptography and data security*. Addison–Wesley, Reading MA, 1982.

[ITSEC91] Information Technology Security Evaluation Criteria (ITSEC). *Harmonised Criteria of France, Germany, the Nederlands, the United Kingdom*, 1991. http://www.bsi.de/zertifiz/itkrit/itsec-en.pdf

[Ke90] U. Kelter. *Group paradigms in discretionary access controls for object management systems*. In F. Long, *Software Engineering Environments*, Proc. Ada Europe International Workshop on Environments, LNCS 467, Springer-Verlag, 1990.

[McMN90] J. McCollum, J.R. Messing, and L. Notargiacomo. *Beyond the pale of MAC and DAC – defining new forms of access control*. In *Proc. IEEE Symp. on Security and Privacy*, Oakland CA, pp. 190-200, 1990.

[Pc96] Charles P. Pfleeger. *Security in Computing, 2. ed.*. Prentice Hall, 1996.

[Pk91] K. Pommerening. *Datenschutz und Datensicherheit*. B.I. Wissenschaftsverlag, Mannheim, 1991.

[Sa88] R. Sandhu. *Transformation of access rights*. In *Proc. IEEE Symp. on Security and Privacy*, Oakland CA, pp. 259-268, 1988.

[St91] D.F. Sterne. *On the buzzword „security policy“*. In *Proc. 1991 IEEE Symposium on Research in Security and Privacy*, Oakland CA, pp. 219-230, 1991.

[TCSEC85] *Trusted computer systems evaluation criteria*. US Department of Defense, DOD 5200.28-STD, 1985.

Glossar

Administration (*administration*) in dieser Kurseinheit: Einstellen von Parametern von Sicherheitsfunktionen, Verwalten der Benutzer und Gruppen.

Administrator (*administrator*) Benutzer oder Benutzerrolle mit besonderen Rechten, die zur Administration des Systems erforderlich sind.

Authentisierung (*authentication*) Kontrollen, durch die die angegebene Identität von Subjekten (Benutzern, Prozessen) nachgewiesen werden soll.

Automatisierte Sicherheitsstrategie (*automated security policy*) Menge von Restriktionen und sonstigen Eigenschaften, durch die ein Rechner die Benutzung von Informationen oder anderer Ressourcen verhindert, wenn hierbei organisatorische Sicherheitsstrategien verletzt werden würden.

Bedrohung (*security threat*) unerwünschter oder unerlaubter Vorgang in einem Rechner, der potentiell zu einem Schaden führt.

Benutzer (*user*) Person, die einen Rechner benutzen darf.

Benutzerprofil (*user profile*) allgemeine Beschränkungen der Rechte eines Benutzers.

Benutzerverwaltung (*user administration*) Funktionen, durch die die Menge der Benutzer eines Systems bestimmt werden kann und durch die relevante Informationen über diese Benutzer verwaltet werden.

Besitzer (*owner*) Subjekt, welches über die Rechte an einem Objekt entscheiden kann (s. diskretionäre Zugriffskontrollen).

Beweissicherung (*audit*) Protokollierung sicherheitsrelevanter Ereignisse.

Capability geschützter Pointer auf ein Objekt, der zugleich Angaben über die erlaubten Operationen auf dem Objekt beinhaltet.

diskretionäre Zugriffskontrolle (*discretionary access control, DAC*) Zugriffskontrollen, durch die der Zugriff zu Objekten auf Basis der Identität von Subjekten und/oder Gruppen, zu denen Subjekte gehören, eingeschränkt wird; die Zugriffskontrollen sind diskretionär (= im Ermessen liegend) insofern, als bestimmte Subjekte, die sog. Besitzer eines Objekts, darüber entscheiden können, ob und wie andere Subjekte auf dieses Objekt zugreifen dürfen.

Identifikation (*identification*) Vorgang zu Beginn einer Sitzung bzw. eines Stapeljobs, bei dem sich der Benutzer dem Rechner gegenüber identifiziert.

login-Prozedur Dialog zwischen Benutzer und Rechner zu Beginn einer Sitzung, durch den sich der Benutzer identifiziert und authentisiert.

Objekt (*object*) identifizierbare zu schützende Einheit.

organisatorische Sicherheitsstrategie (*organizational security policy*) Menge von Gesetzen, Vorschriften, Regeln und Praktiken, die regulieren, wie eine Organisation die Ressourcen, für die gewisse Sicherheitsziele zu erreichen sind, verwaltet, schützt und verteilt.

Passwort, Kennwort (*password*) Wort, durch das sich ein Benutzer gegenüber dem System authentisiert (z. B. beim Anmelden oder beim Zugriff zu geschützten Ressourcen).

persistenter Rechtezustand Menge der Rechtefestlegungen, die eine Systemabschaltung überdauert.

Rechtefestlegung (*access right determination, access rule*) Festlegung durch einen Rechtewert w, ob ein Subjekt s zu einem Objekt o im Modus m zugreifen darf; dargestellt durch ein Quadrupel (s, o, m, w).

Rechteverwaltung (*access rights administration*) Funktionen, durch die dem Betriebssystem mitgeteilt werden kann, ob und wie Subjekte auf Objekte zugreifen dürfen.

Rechteprüfung Funktionen, durch die bei Zugriffen zu Objekten geprüft wird, ob der Prozess hierzu berechtigt ist.

Sicherheitsfunktion (*security function*) Menge von zusammengehörigen Schnittstellen und/oder Eigenschaften, die eine Ebene der darüberliegenden Ebene zur Realisierung von Sicherheitsmaßnahmen anbietet.

Sicherheitsklassifikation (*security class*) Hierarchie von Sicherheitsstufen, von denen jede eine Menge von Sicherheitsfunktionen spezifiziert.

Sicherheitsmechanismus (*security mechanism*) Realisierung einer Sicherheitsfunktion durch konkrete Datenstrukturen und Algorithmen.

Sicherheitsziel (*security policy objective*) [Aussage über die] Absicht, eine identifizierbare Ressource vor unerlaubter Benutzung zu schützen.

Subjekt (*subject*) aktive Einheit in einem System, die zu Objekten zugreift oder Ressourcen verbraucht.

transienter Rechtezustand Rechte von transienten Subjekten (also Prozessen) an Objekten, insb. transienten Objekten wie Hauptspeicherstrukturen.

vertrauenswürdiges Subjekt (*trusted subject*) Subjekt, das Vertraulichkeitslabels herabsenken oder Integritätslabels anheben kann, das also die automatisierte Sicherheitsstrategie durchbrechen darf, dem aber dahingehend vertraut wird, dass es die organisatorische Sicherheitsstrategie nicht durchbricht.

Zugriffskontrolle (*access control*) Rechteverwaltung und Rechteprüfung.

Zugriffskontrolliste (*access control list, ACL*) Liste von Subjekten und zugehörigen Modi; bezieht sich auf ein Objekt und gibt an, dass die aufgeführten Subjekte in den angegebenen Modi zugreifen dürfen.

Zugriffskontrollmodell (*access control model*) Festlegung des persistenten Rechtezustands, des Rechtezustands von Prozessen und von Regeln zur Bestimmung der Zulässigkeit von Zugriffen.

Zugriffskontrollmatrix (*access control matrix*) matrixförmige Darstellung davon, welche Subjekte (repräsentiert durch Zeilen) auf welche Objekte (repräsentiert durch Spalten) zugreifen dürfen; ggf. Angabe der zulässigen Zugriffsmodi.

Zugriffsmodus (*access mode, access type*) Name für eine Menge von Operationen, die auf Objekten ausführbar sind

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Kurseinheit 7:
Kommandosprachen

Autoren: Udo Kelter, Christian Icking, Lihong Ma

Inhalt

Kurseinheit 7

Kommandosprachen

Eine **Kommandosprache** (**Auftragskontrollsprache**, **command language**, **job control language**) ist eine Sprache, in der ein Benutzer Aufträge an einen Rechner formulieren kann oder besser gesagt, muss, wenn er den Rechner dazu bringen will, etwas für ihn zu tun. Die Kommandosprache ist oft der Teil des Betriebssystems, mit dem die Benutzer am intensivsten Kontakt haben und der am ehesten mit einem Betriebssystem assoziiert wird.

Eine ganz wesentliche Aufgabe einer Kommandosprache besteht darin, es den Benutzern eines Rechners zu erlauben, Programme zu laden, hierdurch einen Prozess zu kreieren und diesen zu starten. Dies ist jedoch i. A. nicht allein mit Hilfe der Kommandosprache, also architektonisch gesehen durch den Kommandointerpreter möglich, sondern mit Hilfe entsprechender Systemaufrufe von jedem beliebigen Prozess aus. Bevor wir daher auf Kommandosprachen näher eingehen, befassen wir uns zunächst allgemeiner mit dem Starten von Prozessen und den dabei entstehenden Prozesshierarchien.

7.1 Das Starten von Prozessen

Der Begriff Prozess wurde in den früheren Kurseinheiten 1, 3 und 5 bereits von verschiedenen Seiten aus präzisiert und diskutiert. Relativ offen blieb bisher noch, wie Prozesse in mehrprozessfähigen Betriebssystemen überhaupt entstehen und wie sie mit ihrer Umgebung kommunizieren bzw. mit dieser *Kontakt aufnehmen*. Tatsächlich hängen die meisten einschlägigen Details hierbei wie auch bei der Verwaltung laufender Prozesse ziemlich stark vom einzelnen Betriebssystem ab. Wir stellen daher nur die wichtigsten, häufig auftretenden Funktionen vor und illustrieren diese am Beispiel von konkreten Betriebssystemen.

In älteren Betriebssystemen wurde das Starten von Prozessen als ein hoheitlicher Akt angesehen, der dem Kommandointerpreter vorbehalten war; gemeine Benutzerprozesse konnten keine Prozesse starten. Zu den Zeiten, als zum Laden eines Programms ein Operateur einen Kartenstapel einlegen oder ein Band montieren musste, war diese Anschauung wohl noch gerechtfertigt. In modernen Betriebssystemen kann jeder Prozess weitere Prozesse starten. Die Frage ist jetzt natürlich, wie man ein Programm schreibt, das einen anderen Prozess startet, welche Systemaufrufe das Betriebssystem hierzu anbieten muss und welche sonstigen programmiertechnischen Maßnahmen zum Starten eines Prozesses zu ergreifen sind. Wir können die Aufga-

be, einen Prozess zu starten, bzw. die daraus resultierenden Anforderungen an das Betriebssystem grob wie folgt zerlegen:

- Es muss natürlich möglich sein, einen neuen Prozess zu erzeugen und ein ladbares Programm in den Arbeitsspeicher dieses Prozesses zu laden, siehe Kurseinheit 4, d. h. also den schon früher beschriebenen Lader aufzurufen. Nachdem der Prozess gestartet ist, wird man ggf. später auf sein Ende warten wollen.
- Beim Aufruf eines Programms wird man ihm oft ähnlich wie beim Aufruf einer Prozedur Parameter mitgeben wollen. Analog zu Rückgabewerten von Funktionen sollte ein Prozess am Ende seiner Ausführung Informationen über den Erfolg der Ausführung an den Prozess, der ihn gestartet hatte, zurückgeben können.
- Man muss die Standard-Ein-/Ausgabegeräte des gestarteten Prozesses konkreten Geräten oder Dateien zuordnen können.

  Fast alle Programme erzeugen irgendwelche Ausgabedaten, die bei interaktiver Ausführung auf einem Bildschirm oder bei Stapelausführung auf einen Drucker ausgegeben werden sollen. Ein derartiges Gerät nennen wir **Standard-Ausgabegerät** des Prozesses. Im Unterschied zur Ausgabe in eine Datei braucht ein Standard-Ausgabegerät nicht eigens geöffnet zu werden. Analog zur Ausgabe benötigen viele Programme ein **Standard-Eingabegerät**. Bei interaktiver Ausführung z. B. die Tastatur bzw. bei Stapelausführung z. B. die Datei, aus der auch die Befehle des Stapeljobs gelesen werden.
- Ein Prozess läuft üblicherweise in einer *Umgebung*, die bestimmt, was ein Prozess von seiner Außenwelt wahrnehmen kann; beim Starten eines Prozesses muss man diesem eine initiale Umgebung zur Verfügung stellen können.

In diesem Abschnitt wollen wir nun einige Lösungsansätze für die oben aufgelisteten Aufgaben kennenlernen.

7.1.1 Systemaufrufe zum Starten und Steuern von Prozessen

Hierzu stellt das Betriebssystem Systemaufrufe zum

- Erzeugen und Starten eines Prozesses
- Warten auf das Ende eines gestarteten Prozesses
- Abfragen zur Menge der existierenden Prozesse
- Abbrechen eines Prozesses von außen
- Beenden eines Prozesses

zur Verfügung. Wenn ein Prozess einen anderen erzeugt hat, bezeichnen wir ihn als **Elternprozess** und den erzeugten Prozess als **Kindprozess**. Da der Kindprozess seinerseits Kindprozesse erzeugen kann, entsteht i. A. eine baumartige Struktur von *Vorfahren*.

Die Details der Prozessverwaltung und der zugehörigen Systemaufrufe sind von Betriebssystem zu Betriebssystem sehr verschieden. Die Systemaufrufe, vor allem die

Datenstrukturen ihrer Parameter, sind oft stark beeinflusst durch die Programmiersprache, in der das Betriebssystem geschrieben worden ist bzw. in der die Systemaufrufe eingebettet sind. In den folgenden Beispielen lassen wir alle zu speziellen Details aus und verwenden eine relativ *abstrakte*, funktionale Notation für die Syntax der Systemaufrufe, in der jeweils nur die Typen der Parameter bzw. Rückgabewerte angegeben sind.

Erzeugen und Starten eines Prozesses. Das Naheliegendste ist ein Systemaufruf der Form

```
process_create (name_ladb_prog, ... ) : pid
```

dem der Name einer Datei, die ein ladbares Programm enthält, sowie ggf. weitere Parameter übergeben werden und der einen Prozessidentifizierer `pid` (üblicherweise vom Typ Integer) zurückgibt. Dieser Systemaufruf erzeugt einen neuen Prozessdeskriptor und einen logischen Hauptspeicher für diesen Prozess, lädt das Programm in diesen Hauptspeicher und führt ggf. weitere Initialisierungen durch. Es ist sinnvoll, das Erzeugen und Starten eines Prozesses zu trennen, denn nach dem Erzeugen des Prozesses können zunächst noch Änderungen am Prozesszustand von außen vorgenommen werden, bevor der Prozess selbst gestartet wird. Zum Starten benötigt man dann eine weitere Funktion

```
process_start (pid, ... )
```

Ein Beispiel für eine Betriebssystemschnittstelle, in der dieser Ansatz verwirklicht ist, ist PCTE [PCTE97].

Eine völlig andere Vorgehensweise findet sich in POSIX[1]: dort werden Prozesse durch einen Systemaufruf

```
fork () : pid
```

erzeugt. Dieser parameterlose Systemaufruf erzeugt eine fast vollständige Kopie des aufrufenden Prozesses. Insb. wird der vollständige logische Hauptspeicher des aufrufenden Prozesses (!) und der Befehlszähler kopiert. Eltern- und Kindprozess führen also anschließend beide die Anweisung hinter dem `fork` aus. Der einzige Unterschied zwischen dem Eltern- und Kindprozess besteht im zurückgegebenen Prozess-Identifizierer: beim Kindprozess wird 0 zurückgegeben, beim Elternprozess der Prozessidentifizierer des Kindprozesses (immer ein Wert ungleich 0),[2] siehe Abbildung 7.1.

Nun ist es natürlich ziemlich sinnlos, zweimal das gleiche Programm mit den gleichen Daten parallel auszuführen. Daher ist der weitere Verlauf typischerweise so, dass sofort nach dem `fork` der zurückgegebene Prozessidentifizierer in beiden Prozessen mit 0 verglichen wird, denn beide führen ja das gleiche Programm aus; der Elternprozess führt dann das vorhandene Programm weiter aus, der Kindprozess lädt ein völlig neues Programm. Dies geschieht z. B. durch den Systemaufruf

[1] POSIX [POSIX88] ist eine Spezifikation von Systemaufrufen, die i. W. vom Betriebssystem UNIX stammen. POSIX ist die einzige Spezifikation von Systemaufrufen, die international durch die ISO standardisiert wurde. Die POSIX-Schnittstellen sind Teil der X/OPEN-Spezifikationen und auf einer großen Zahl von Rechnern implementiert worden.

[2] Außerdem gibt es Unterschiede bei den geöffneten Dateien beider Prozesse; hierauf gehen wir später genauer ein.

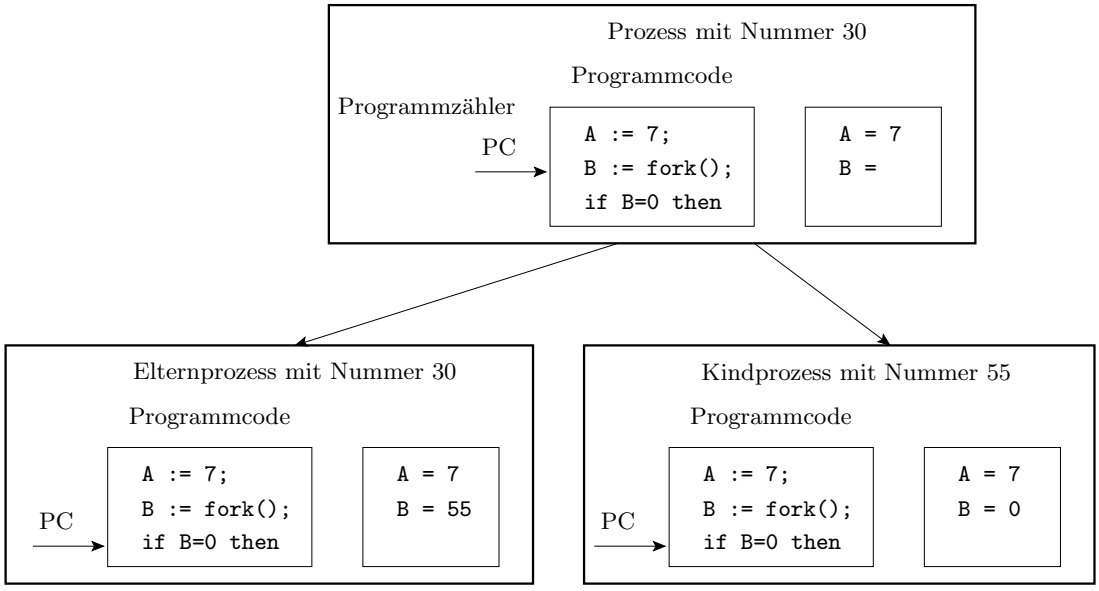


Abbildung 7.1: Modell der Prozesserzeugung in UNIX. Der Prozess mit Nummer 30 erzeugt einen Kindprozess durch `fork()`. Nach dem `fork`-Aufruf gibt es zwei Prozesse, den Elternprozess mit Nummer 30 und den Kindprozess mit Nummer 55. Die beiden Prozesse unterscheiden sich nur in dem bei `fork` zurückgegebenen Wert `B`.

```
exec (name_ladb_prog, ... )
```

der das in der angegebenen Datei enthaltene ladbare Programm in den Hauptspeicher des Prozesses lädt und dabei das bisher vorhandene Programm löscht. Im allgemeinsten Fall hat `exec` drei Parameter. Verschiedene Bibliotheksfunktionen, inclusive `execl`, `execv`, `execve`, `execvp`, erlauben es, die Parameter auf verschiedene Arten wegzulassen oder zu spezifizieren. Insgesamt lässt sich also mit `fork` und `exec` der gleiche Effekt erzielen wie mit `process_create` und `process_start`[3].

Warten auf das Ende eines Prozesses. Nachdem ein Prozess einen anderen Kindprozess erzeugt hat, wird er oft sofort anschließend oder später auf das Ende des Kindprozesses warten wollen (dies gilt insb. für Kommandointerpreter). Hierzu benötigt der Elternprozess einen geeigneten Systemaufruf. Als Beispiel hierfür sei der POSIX-Systemaufruf

```
wait (...) : pid
```

genannt, der es ermöglicht, auf das Ende eines Kindprozesses zu warten, siehe Abbildung 7.2. Zurückgegeben wird der Prozessidentifizierer des beendeten Kindprozesses

[3]Man kann (mit Recht) den Kopf darüber schütteln, warum in POSIX (bzw. UNIX) für einen neuen Prozess erst mit relativ großem Aufwand eine vollständige Kopie des logischen Hauptspeichers des Prozesses erzeugt wird, um diese dann anschließend sofort wieder zu löschen. Dieser Design-Fehler in POSIX ist wie so vieles in real existierenden Betriebssystemen nur historisch zu erklären: UNIX wurde in den 1960er und 70er Jahren entwickelt, seinerzeit auf 16-Bit-Prozessoren, somit bei 64 kB großen logischen Hauptspeichern. Die meisten Programme waren noch wesentlich kleiner. Unter diesen Voraussetzungen ist das Kopieren des Hauptspeichers eines Prozesses (i. W. die gleiche Funktion wie beim Swapping) kein sonderliches Problem. Leider treffen diese Voraussetzungen heute i. A. nicht mehr zu; wenn man heute die POSIX-Schnittstellen auf einem Betriebssystemkern implementieren will, muss man durch besondere Maßnahmen (z. B. copy-on-write-Techniken) die schlimmsten Folgen dieses Design-Fehlers in POSIX vermeiden.

und Zustandsinformation zu diesem, die u. a. einen Rückgabewert des Kindprozesses enthält.

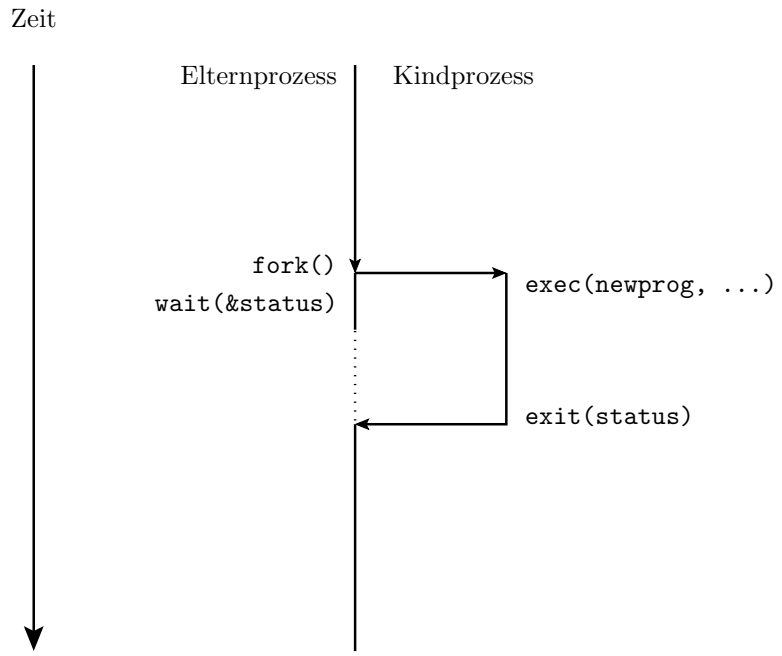


Abbildung 7.2: Prozesserzeugung und Synchronisation. Der mit `fork()` erzeugte Kindprozess führt mit `exec` ein anderes Programm als sein Elternprozess aus. Der Elternprozess wartet mit `wait` auf das Ende des Kindprozesses, dann findet er in der Variablen `status` einen bei `exit` vom Kindprozess angegebenen Statuswert.

Abfragen zur Menge der existierenden Prozesse. In diese Kategorie fallen diverse Systemaufrufe, durch die ein Prozess Informationen über sich selbst und andere Prozesse sowie über die Menge der existierenden Prozesse erhalten kann. Beispiele hierfür sind die Systemaufrufe

```
getuid () : uid
getgroups () : grouplist
```

die den Besitzer bzw. die Menge der aktivierten Gruppen eines Prozesses abfragen. Z. B. unter UNIX kann man mit dem Befehl `top` eine Liste aller existierenden Prozesse und ihres Ressourcenverbrauchs ausgeben lassen, die ständig aktualisiert wird.

Abbrechen eines Prozesses von außen. Oft ist es erforderlich, einen Prozess von außen abzubrechen, z. B. weil er eine Endlosschleife ausführt und sich nicht mehr selbst beendet oder weil die weitere Ausführung von einem Benutzer aus irgendwelchen Gründen als nicht mehr sinnvoll angesehen wird. Ein Beispiel (aus UNIX) für einen Systemaufruf, durch den ein anderer Prozess abgebrochen werden kann, ist

```
kill (pid, ... )
```

Beenden eines Prozesses. Wenn ein (Kind-) Prozess sein Programm abgearbeitet hat, muss er das Betriebssystem hierüber informieren und sich durch einen entsprechenden Systemaufruf selbst beenden. Ein Beispiel hierfür (wieder aus UNIX) ist

```
exit ( status )
```

In `status` kann ein Rückgabewert an den Elternprozess übergeben werden, siehe auch Abbildung 7.2.

Eine interessante Frage ist, ob auch die Kindprozesse eines Prozesses, der sich selbst beendet hat oder der von außen beendet wurde, automatisch abgebrochen werden sollen oder nicht. Tatsächlich benötigt man i. A. beide Alternativen. Der Betriebssystemkern sollte also beide Varianten anbieten.

Als ein einfaches Beispiel zeigen wir, wie von einem laufenden Prozess namens `forkdemo` aus ein Kindprozess `beeper` erzeugt wird, der eine Folge von Pieptönen produziert, bis er von seinem Elternprozess `forkdemo` wieder beendet wird.

Das C-Programm `beeper` für den Kindprozess ist sehr einfach:

```
/* beeper.c, Endlosschleife mit Pieptönen       */
/* kompilieren: cc -o beeper beeper.c           */
/* starten mit: beeper                          */
/* beenden mit: kill oder Control-C             */

# include<stdio.h>

int main()
{
    while (!0) {
      printf ("%c",07);
    }
}
```

Das C-Programm `forkdemo` für den Elternprozess enthält eine Schleife, in der die Eingabe von 1, 2 oder 0 angefordert wird; Eingabe von 1 startet den Kindprozess, Eingabe von 2 beendet ihn wieder, und wenn man 0 eingibt, werden der Elternprozess und auch der Kindprozess beendet.

```
/* forkdemo.c, erzeugt und beendet Kindprozess beeper */
/* kompilieren: cc -o forkdemo forkdemo.c             */
/* starten mit: forkdemo                              */
/* beeper starten lassen mit 1, stoppen mit 2,        */
/* Programm beenden mit 0                             */

#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
```

```
int main(int argc, char*argv[])

{

int prozessnr, ergebnis, piep;
                        /* piep = 0, Piepton ist aus */
                        /* piep = 1, Piepton ist an */
char c ;

    piep = 0;
    prozessnr = -1;      /* kein Prozess gestartet */
    do {
         printf("1: Piepton starten\n");
         printf("2: Piepton stoppen\n");
         printf("0: Programm beenden\n");
         scanf("%s",&c);  /* Eingabe von Tastatur einlesen */
         printf("\n");
         printf("Eingegeben ist %c\n",c);

         if (c =='1' && piep == 0) {
           piep = 1;     /* Piepton wird nur einmal gestartet */
           prozessnr = fork(); /* neuen Prozess erzeugen: */
           if (prozessnr == 0) {/* im Kindprozess hat prozessnr */
                           /* den Wert 0, im Elternprozess */
                           /* die Prozessnummer des */
                           /* Kindprozesses */
             ergebnis = execvp( "beeper", argv);
                           /* 'beeper' starten */
             exit(0);
           }
         }/* if c = '1' und Piepton läuft */
         if (c == '2' && prozessnr != -1) {
           piep = 0;           /* Piepton wird abgeschaltet */
           ergebnis = kill( prozessnr,9); /* Kindprozess beenden */
         }
    } while (c != '0');
    if ( piep != 0) {
      ergebnis = kill(prozessnr,9);  /* nicht vergessen, den */
                                     / * Piepton abzuschalten */
    }
}
```

7.1.2 Die Umgebung eines Prozesses

Ein Elternprozess wird typischerweise einen Kindprozess deshalb gestartet haben, damit der Kindprozess irgendeine Aufgabe ggf. parallel zum Elternprozess erledigt, z. B. das Übersetzen eines Programms. Nun muss das Übersetzungsprogramm, nachdem es in den Hauptspeicher des Kindprozesses geladen worden ist, natürlich irgend-

woher wissen, welche Dateien es verarbeiten soll, d. h. es müssen an den Kindprozess gewisse Parameter übergeben werden. Die üblichen Parameterübergabemechanismen für Prozeduren in Programmiersprachen funktionieren hier aber nicht mehr, da der Hauptspeicher des Elternprozesses für den Kindprozess nicht zugreifbar ist, und sie sind, wie wir gleich sehen werden, auch nicht optimal angepasst an diese Aufgabe. Man benötigt daher andere Mittel, eine (Ausführungs-) **Umgebung** für den Kindprozess zu erzeugen. Der Begriff der Umgebung eines Prozesses variiert bei den Betriebssystemen erheblich. Typische Komponenten der Umgebung sind:

- beim Start übergebene Parameterwerte
- Umgebungsvariablen
- die offenen Dateien

Parameter. Parameter spielen die gleiche Rolle wie Parameter von Prozeduren in Programmiersprachen. Beispielsweise würde man einem Übersetzer als Parameter die Namen der Dateien übergeben, die das zu übersetzende Programm, das Bindemodul bzw. das Programmlisting incl. der Fehlermeldungen enthalten. Z. B. der POSIX-Systemaufruf `execve(name,argv,envp)` hat drei Parameter: den Namen der datei, die ausgeführt werden soll, einen Zeiger auf das Feld der Argumente und einen Zeiger auf das Feld der Umgebungsvariablen. In dem oben erwähnten POSIX-Systemaufruf `execve` werden die Parameter z. B. in Form eines Arrays von Zeichenketten übergeben. Es sind aber auch andere Mechanismen denkbar.

Umgebungsvariablen. Eine Umgebungsvariable besteht aus einem Namen und einem Wert vom Typ String. Parameter und Umgebungsvariablen haben beide letztlich einen (kurzen) Text als Wert. Die Unterscheidung zwischen ihnen ist ähnlich motiviert wie die Unterscheidung zwischen Parametern und globalen Variablen in Programmen: die Parameter stellen die Ein- bzw. Ausgabewerte der Funktion (im mathematischen Sinn), als die man das aufgerufene Programm auffassen kann, dar. Wiederholte Aufrufe desselben Programms haben üblicherweise verschiedene Eingabeparameterwerte.

Umgebungsvariablen enthalten dagegen allgemeine Daten oder Einstellungen, deren Wert nur selten verändert wird oder vom Benutzerprozess überhaupt nicht verändert werden kann. Ein typisches Beispiel ist eine Umgebungsvariable, die die Identifizierung des Benutzers enthält, für den der Prozess läuft. Allerdings können Umgebungsvariablen auch von Benutzerprozessen eingerichtet und als Ersatz für Parameter benutzt werden[4]. Ebenso kann der Wert von solchen Umgebungsvariablen jederzeit geändert werden, d. h. ein Programm kann seine eigene Ablaufumgebung verändern.

Es ist klar, dass die Umgebungsvariablen eines Prozesses auch für dessen Kindprozesse verfügbar sein sollten. Hierfür gibt es zwei grundsätzlich verschiedene Vorgehensweisen:

[4]Globale Variablen sind in Programmen zu Recht *verboten*. Diese softwaretechnologische Regel folgt allerdings aus Problemen, die mit der Größe und Schachtelungstiefe von Programmen zusammenhängen, und ist wegen der typischerweise ganz anderen Gegebenheiten bei geschachtelten Prozessen nicht ohne weiteres auf diese anwendbar.

1. Die Umgebungsvariablen sind global und nur einmal im System vorhanden. Wenn also ein Kindprozess Werte von Umgebungsvariablen ändert, dann werden diese Änderungen auch in der Umgebung seines Elternprozesses sichtbar. Ein Beispiel für diesen Ansatz ist MS-DOS, obgleich MS-DOS nur bedingt als Beispiel geeignet ist, weil es keine echten parallelen Prozesse hat. In einem System, in dem verschiedene Benutzer gleichzeitig arbeiten, sind globale Umgebungsvariablen natürlich völlig inakzeptabel. Selbst benutzerspezifische Umgebungsvariablen wären i. A. zu restriktiv, wenn ein Benutzer mehrere parallele Prozesse gleichzeitig ausführen kann, z. B. im Hintergrund oder in verschiedenen Fenstern auf einem Graphikbildschirm. Bei einem ein-Benutzer-ein-Prozess-System wie MS-DOS treten die Probleme global gültiger Umgebungsvariablen nicht so akut auf.

2. Die Umgebungsvariablen sind prozess-spezifisch. Wie wir schon gesehen haben, ist dies in einem wirklichen Mehrbenutzersystem die einzig akzeptable Alternative. Die Umgebungsvariablen des Elternprozesses werden für jeden Kindprozess bei dessen Erzeugung kopiert. Der Kindprozess kann anschließend die Werte dieser Umgebungsvariablen beliebig verändern, ohne dass dies irgendwelche Auswirkungen auf die Werte der Variablen des Elternprozesses hätte. Am Ende des Kindprozesses werden seine Umgebungsvariablen gelöscht, d. h. deren Werte sind verloren und können auch nicht zur Übergabe von Rückgabewerten oder sonstiger Informationen an den Elternprozess verwendet werden.

Offene Dateien. Die Aufgabe, die ein Elternprozess an einen Kindprozess delegiert, kann u. a. darin bestehen, den Inhalt einer Datei, die der Elternprozess bereits geöffnet hat, weiterzuverarbeiten. Nun wäre es denkbar, dass der Elternprozess die Datei schließt, der Kindprozess die Datei wieder öffnet, ggf. die richtige Stelle darin sucht und die Datei dann weiterverarbeitet. Am Ende des Kindprozesses müsste der ganze Vorgang wieder umgekehrt verlaufen. Dies wäre relativ umständlich und ineffizient. Die einfachere Lösung ist, offene Dateien vom Elternprozess auf den Kindprozess zu vererben.

Die Details hängen hier wiederum stark davon ab, wie Dateien in einem Betriebssystem geöffnet und wie eine der geöffneten Dateien in Dateioperationen (wie z. B. lesen oder schreiben) identifiziert wird. Beim Öffnen einer Datei muss das Betriebssystem zunächst einen **Dateikontrollblock** für eine *offene* Datei anlegen, in dem alle notwendigen Informationen über die geöffnete Datei enthalten sind, z. B. Öffnungsmodus (lesen, schreiben, anhängen), aktuelle Position des Dateizeigers beim sequentiellen Verarbeiten usw.. Man verwechsle einen Dateikontrollblock nicht mit einer Dateiattribute! Eine Dateiattribute enthält Informationen über eine Datei und ist Teil des Dateisystems. Ein Dateikontrollblock enthält Informationen über *eine geöffnete Datei*, also den Sachverhalt, dass ein Prozess auf einer Datei arbeitet.

Im allgemeinen können sogar mehrere Prozesse über einen Dateikontrollblock auf einer Datei arbeiten. Daher können Dateikontrollblöcke auch nicht im logischen Adressraum eines der Prozesse liegen. Stattdessen werden alle vorhandenen Dateikontrollblöcke vom Betriebssystem in einer zentralen Datenstruktur verwaltet.

Ein Prozess kann mehrere Dateien öffnen und gleichzeitig mit ihnen arbeiten; hierbei muss er offene Dateien bzw. die entstandenen Dateikontrollblöcke identifizieren können. Zu diesem Zweck werden die offenen Dateien eines Prozesses in vielen

Betriebssystemen von 0 an durchnummeriert, in anderen erhalten sie symbolische Namen, worauf wir nicht näher eingehen. Diese Nummern kann das Betriebssystem sehr einfach mit Hilfe einer prozessspezifischen Umsetztabelle (also einem Array) in die Adressen der Dateikontrollblöcke umsetzen; diese Tabelle ist Teil des Prozesskontrollblocks des jeweiligen Prozesses. Die Zusammenhänge sind noch einmal in Abbildung 7.3 zusammengefasst. Diese nur innerhalb eines Prozesses gültigen Nummern nennen wir **Dateiidentifizierer**[5]. Jeder Dateiidentifizierer eines Prozes-

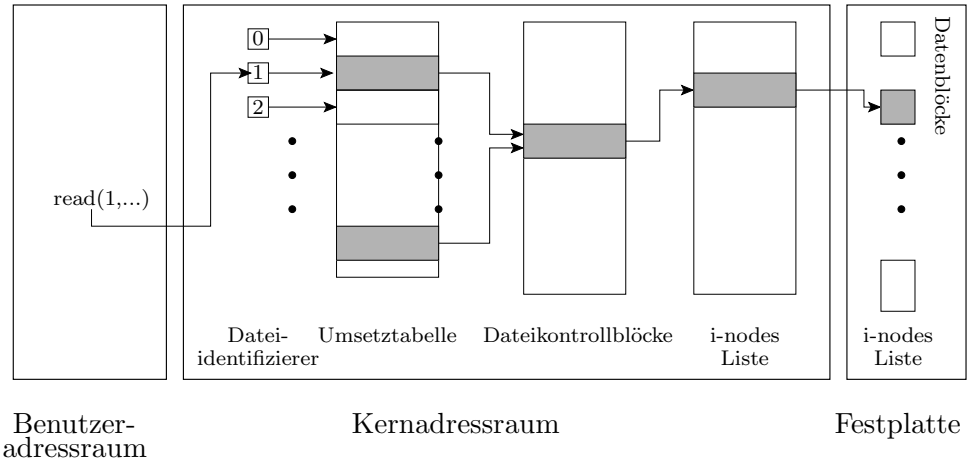


Abbildung 7.3: Dateikontrollblöcke.

ses identifiziert genau einen Dateikontrollblock. Umgekehrt können sich aber mehrere Dateiidentifizierer eines Prozesses auf den gleichen Dateikontrollblock beziehen. Wenn *eine Datei* geschlossen wird, wird zunächst nur der Dateiidentifizierer und damit ein Verweis auf den Dateikontrollblock gelöscht, nicht der Dateikontrollblock selbst. Dieser wird erst gelöscht, wenn der letzte Dateiidentifizierer, der auf ihn verweist, gelöscht worden ist. Eine Datei kann auch mehrfach von verschiedenen Prozessen geöffnet werden. Dann existieren für die gleiche Datei mehrere unabhängige Dateikontrollblöcke.

Das Vererben der offenen Dateien eines Prozesses auf einen Kindprozess kann nun so realisiert werden, dass die Tabelle kopiert wird, die die Dateiidentifizierer in die Adressen der Dateikontrollblöcke umsetzt. Der Kindprozess kann dann mit den gleichen Dateiidentifizierern auf die gleichen offenen Dateien zugreifen wie der Elternprozess.

Beim Vererben einer offenen Datei darf deren Dateikontrollblock selbst nicht kopiert werden. Die beiden entstehenden Dateikontrollblöcke hätten unabhängige Dateizeiger, d. h. der Kindprozess würde nur seinen eigenen Dateizeiger verändern

[5]Genaugenommen werden hier Dateien nur indirekt über den Dateikontrollblock identifiziert. Wie in Abbildung 7.3 ersichtlich ist, wird sogar der Dateikontrollblock nur indirekt identifiziert, denn die Nummer, die im Benutzerprogramm benutzt wird, muss zunächst innerhalb des Betriebssystems in die Adresse des Dateikontrollblocks umgesetzt werden. Die Nummer ist also eigentlich ein *Dateiidentifizierer-Identifizierer*. Diese Unterscheidung bringt aber nicht viel ein, so dass wir diese begriffliche Unschärfe hier in Kauf nehmen.

In POSIX werden die Dateiidentifizierer *file descriptor* genannt; diese Bezeichnung ist aber ziemlich irreführend, denn die Nummer identifiziert zwar (wenn auch indirekt) eine Datei, sie *ist* aber keine Datenstruktur, die Informationen über die Datei enthält (das ist z. B. in UNIX der i-node). Der Dateikontrollblock wird in POSIX *open file description* genannt.

und der Elternprozess wüsste nicht, welches Stück der Datei vom Kindprozess verarbeitet worden ist bzw. wieviel Ausgaben der Kindprozess in einer Ausgabedatei erzeugt hat.

Der Elternprozess will i. A. allerdings nicht alle offenen Dateien, sondern nur eine Teilmenge davon an den Kindprozess vererben. Es sind diverse Techniken denkbar, durch die der Elternprozess dem Betriebssystem die Menge der zu vererbenden offenen Dateien angeben kann. In POSIX ist jedem Dateiidentifizierer eine Anzeige (das *close-on-exec flag*) zugeordnet, die angibt, ob dieser Dateiidentifizierer und die dazugehörige offene Datei an einen Kindprozess vererbt werden soll.

Standard-Ein-/Ausgabegeräte. Ein Programm soll typischerweise irgendwelche Ausgabedaten erzeugen und hierzu irgendwelche Eingabedaten benutzen. Hierzu kann man natürlich Dateien benutzen, die die Ein- bzw. Ausgabedaten enthalten. Daneben kommen aber auch die E/A-Geräte eines Rechners hierfür in Frage.

Interaktive Programme erhalten Eingaben von Tastaturen, Mäusen oder ähnlichen Eingabegeräten und schreiben Ausgaben auf Bildschirme oder andere Ausgabegeräte. Hier wäre es völlig unsinnig zu verlangen, dass das Programm für seine Ein-/Ausgaben zunächst einmal Dateien benutzt.

In einem Stapeljob kann man die Datei, die die Kommandoprozedur des Stapeljobs enthält, als Ersatz für eine Tastatur betrachten, d. h. ein direkt hinter dem Kommando folgender Abschnitt in dieser Datei kann von dem gestarteten Prozess so eingelesen werden, als ob diese Daten über eine Tastatur eingegeben worden wären[6]. Das Protokoll, das bei einem Stapeljob erstellt wird, wird auf einen Drucker zumindest ogisch ausgegeben, denn tatsächlich wird das Protokoll infolge des Spoolings wahrscheinlich zuerst in eine Datei geschrieben werden.

In praktisch allen Betriebssystemen findet sich daher das Konzept der **Standard-Ein-/Ausgabegeräte**: Jeder Prozess hat ein ihm zugeordnetes Standard-Eingabegerät und (wenigstens) ein Standard-Ausgabegerät. Standard-Ein-/Ausgabegeräte unterscheiden sich von Dateien in zwei Punkten:

1. Aus Sicht eines Prozesses sind seine Standard-Ein-/Ausgabegeräte automatisch vorhanden, d. h. im Gegensatz zu Dateien müssen sie nicht explizit geöffnet oder geschlossen werden. Die Zuordnung der Standard-Ein-/Ausgabegeräte, die ja nur virtuelle, prozesslokal vorhandene Geräte sind, zu realen Geräten ist außerhalb und vor Beginn des Prozesses vorgenommen worden.

2. Der *Datentyp* von Dateien ist durch deren Zugriffsmethode gegeben, d. h. konzeptionell liest und schreibt man Zeichen oder Sätze in einer Datei. Der Datentyp von realen E/A-Geräten weicht davon ganz erheblich ab: also wenn

[6] Es entsteht nun das Problem, das Ende der Eingabedaten, die eine *logische* Datei innerhalb der Stapeljob-Datei bilden, zu erkennen; dahinter wird ja i. A. ein weiteres Kommando folgen, das als solches erkannt werden muss. Eine denkbare Lösung besteht darin, bestimmte Zeilen (wie '/EOF', '.') oder Zeichen ('<control>-d') als Anzeige des Dateiendes zu interpretieren: sobald eine solche Anzeige erreicht wird, gilt die Eingabe als beendet. Die Dateiende-Anzeige darf natürlich nicht im Text auftreten. Eine andere Lösung besteht darin, prinzipiell alle Zeilen, die mit einem speziellen Zeichen anfangen (z. B. '/'), als Kommando zu interpretieren. Sobald eine Kommandozeile erreicht wird, ist das Ende der Eingabedatei implizit angezeigt.

wir einmal Details entsprechender Systemaufrufe außer acht lassen, konzeptionell liest man eine Zeile oder den Inhalt eines Formulars auf dem Bildschirm, löscht den Bildschirm oder gibt Texte z. B. in heller / dunkler oder normaler / invertierter Darstellung auf einem Bildschirm aus, man druckt Zeilen auf dem Papier, erzeugt einen Zeilen- oder Seitenvorschub im Drucker usw., man steuert vielleicht zusätzlich die Datenübertragung zwischen Rechner und E/A-Gerät bei Datenfernübertragung oder schaltet z. B. das lokale Echo während der Eingabe eines Kennworts ab. Die Details hängen von diversen technischen Merkmalen der Geräte und der Datenübertragungsfunktionen ab.

Diese *semantischen* Unterschiede haben bei älteren Betriebssystemen dazu geführt, dass völlig verschiedene Systemaufrufe für Standard-E/A-Geräte bzw. Dateien vorhanden sind. Die Parameterlisten der Systemaufrufe für Terminal-Ein-/Ausgaben enthalten z. B. diverse Parameter, die mit der Steuerung des Terminals und der Datenübertragung zum bzw. vom Terminal zu tun haben; diese Parameter machen für Dateien keinen Sinn. Umgekehrt werden in diesen Betriebssystemen meist satzorientierte Zugriffsmethoden für Dateien verwendet; die entsprechenden Lese/Schreiboperationen sind nicht immer ohne weiteres auf einem Terminal implementierbar.

Umlenkung von Standard-E/A-Geräten. Unterschiedliche Systemaufrufe für Standard-E/A-Geräte bzw. Dateien haben aber aus Sicht eines Anwendungsprogrammierers einen ganz entscheidenden Nachteil: sehr viele Programme arbeiten mit einfachen sequentiellen Folgen von ein- bzw. ausgegebenen Sätzen oder Zeichen. Aus Sicht eines Anwenders besteht kein signifikanter Unterschied dazwischen, ob man Eingabezeilen am Bildschirm direkt in das Programm eingibt oder erst in eine Datei schreibt und dann diese Datei verarbeiten lässt. Analog gilt dies für Ausgaben. Deshalb will man sehr häufig das gleiche Programm einmal mit interaktiv ein-/ausgegebenen Daten ausführen, ein andermal mit Ein-/Ausgabedaten in Dateien. Wenn die Systemaufrufe für Standard-E/A-Geräte bzw. Dateien verschieden sind, ist dies aber nicht möglich.

Dieses Problem führt zu der Forderung, dass ein und dasselbe ladbare Programm sowohl mit realen Geräten wie auch mit Dateien als Standard-E/A-Geräten lauffähig sein soll. Was bei einer konkreten Ausführung des Programms verwendet werden soll, muss natürlich von außen vorgegeben werden. Die Fähigkeit, Dateien statt realer Ein-/Ausgabegeräte zu benutzen, nennt man **Umlenkung** von Standard-E/A-Geräten. Diese Forderung impliziert fast direkt die neue Forderung, dass das Lesen und Schreiben von Daten auf Standard-E/A-Geräten und Dateien mit *exakt gleichen Systemaufrufen* durchgeführt wird. Dies bedeutet, dass ein Programm die Standard-E/A-Geräte wie Dateien behandeln muss, wir erinnern uns aus Kurseinheit 2, dass reale Geräte ohnehin als Dateien repräsentiert werden sollten.

Offensichtlich kann man nur dann mit den gleichen Systemaufrufen auf Standard-E/A-Geräten und Dateien arbeiten, wenn diese den gleichen *Datentyp* haben. Dieser gemeinsame Datentyp kann nur die Merkmale enthalten, die sowohl in Dateien wie auch in Standard-E/A-Geräten realisierbar sind, Hell/Dunkel-Darstellung auf dem Bildschirm kann offenbar nicht Teil dieses Datentyps sein. Dies bedeutet natürlich, dass auch Applikationen auf Basis dieses relativ primitiven Datentyps geschrieben sein müssen, wenn ihre Ein-/Ausgaben umlenkbar sein sollen.

Ein derartiger gemeinsamer Datentyp, der in POSIX und MS-DOS realisiert wird, sind Folgen von Zeichen, die gelesen bzw. geschrieben werden. Bei Dateien haben wir diesen Datentyp schon als zeichenorientierte Zugriffsmethode kennengelernt, allerdings ohne Direktzugriff, siehe Kurseinheit 4. Bei Standard-E/A-Geräten bedingt dies, alle Steueranweisungen als Zeichenfolgen mit spezieller Bedeutung in den ausgegebenen Text einzubetten, wenn z. B. spezielle Fähigkeiten eines Terminals ausgenutzt werden sollen[7].

Als Beispiel betrachten wir das Programm `sort` unter UNIX. Das Standard Ein- und Ausgabemedium in UNIX sind die beiden Dateien `stdin` (standard-input) für die Eingabe und `stdout` (standard-output) für die Ausgabe. Mit den Symbolen "<" und ">" wird `stdin` und `stdout` umgelenkt. `sort` liest `stdin`, sortiert zeilenweise und schreibt die sortierten Zeilen nach `stdout`. Das Kommando

```
sort < eindatei
```

bewirkt, dass `sort` seine Daten nicht von der Terminaltastatur erwartet, sondern aus der Datei `eindatei` liest. Mit `sort > ausdatei` wird die Ausgabe auf `ausdatei` gelenkt. Falls `ausdatei` schon existiert, wird sie überschrieben.

Mit der Kombination von "<" und ">" liest

```
sort < eindatei > ausdatei
```

von `eindatei` und schreibt nach `ausdatei`.

Den Standard-E/A-Geräten werden nun als *Konvention* einfach feste Dateiidentifizierer zugewiesen, beispielsweise in MS-DOS und POSIX

0 : Standard-Eingabegerät
1 : Standard-Ausgabegerät

In POSIX wird zusätzlich zwischen *normalen* Ausgaben und Fehlermeldungen unterschieden; die Fehlermeldungen werden auf die Datei mit Dateiidentifizierer 2 geschrieben.

Standard-E/A-Geräte müssen natürlich *außerhalb der Applikation* geöffnet werden, also z. B. durch den Kommandointerpreter und von dort als offene Dateien vererbt worden sein. Ein wesentliches Hilfsmittel, das ein Elternprozess nunmehr benötigt, wenn er die Standard-Ein-/Ausgaben eines zu startenden Kindprozesses umlenken will, ist ein Systemaufruf wie `dup2 (di_alt, di_neu)` (aus POSIX), der den Dateiidentifizierer `di_alt` kopiert, sodass anschließend `di_neu` auf den gleichen Dateikontrollblock wie `di_alt` verweist. So kann z. B. eine bereits zum Lesen geöffnete Datei, die im Elternprozess den Identifizierer 7 hat, den zusätzlichen Identifizierer 0 bekommen, der dann an den Kindprozess vererbt wird.

[7] Die spezielleren Eigenschaften und Leistungsmerkmale von Terminals sind dem Betriebssystem daher unbekannt. Dies ist einerseits ein Vorteil, weil der Betriebssystemkern dadurch kleiner bleibt und nicht speziellen Code für jedes anschließbare Terminal enthalten muss und weil somit neue Terminalvarianten ohne Änderung des Betriebssystemkerns betrieben werden können. Andererseits kann man es als Nachteil ansehen, dass die genaue Kenntnis der Terminal-Eigenschaften in die Verantwortung der Applikationen (bzw. deren Programmierer) verschoben wird. Teilweise lässt sich dieses Problem durch parametrisierbare Treiber lösen. Letztlich ist es aber unvermeidbar, dass ein Programm, das spezielle Terminaleigenschaften ausnutzt, nicht ohne weiteres mit beliebigen anderen Terminals betrieben werden kann.

7.2 Generelle Merkmale von Kommandosprachen

Viele UNIX-Systeme haben eine graphische Benutzungsschnittstelle, wie sie durch den Macintosh und später Windows populär wurden. Trotzdem bevorzugen echte Programmierer nach wie vor eine Kommandozeilen-Schnittstelle, *Shell*. Sie ist viel *schneller* zu benutzen, *mächtiger* und einfach *erweiterbar*.

Es gibt viele, oft sehr unterschiedliche Kommandosprachen. Dies hat mehrere Ursachen:

- Eine Kommandosprache, vor allem ihre Semantik, hängt i. A. nicht unerheblich von den im Betriebssystemkern realisierten Funktionen ab. Einfluss auf die Kommandosprache hat z. B., wie Programme ausgeführt werden, wie Prozesse geschachtelt werden können, wie Parameter an Programme übergeben und ggf. Fehlercodes zurückgegeben werden usw. Mit anderen Worten ist nicht jede Kommandosprache auf jedem Betriebssystemkern implementierbar.

- Kommandosprachen für interaktiven Betrieb hängen stark von den Hardware-Eigenschaften der benutzten realen E/A-Geräte ab, u. a. vom Vorhandensein von Funktionstasten, vom Zeichenvorrat, von den Graphikfähigkeiten des Bildschirms und von dem Übertragungsmodus zwischen Terminal und Rechner (block- oder zeichenorientiert). Textuelle und *graphische Sprachen* unterscheiden sich ganz erheblich.

- Kommandosprachen sind auch deutlich beeinflusst von der Architektur des Betriebssystems.

Aufgrund dieser Vielfalt wird in dieser Kurseinheit keine spezielle Kommandosprache detailliert vorgestellt. Es wird angenommen, dass die Leser wenigstens eine Kommandosprache kennen und damit bereits ein intuitives Verständnis der grundlegenden Begriffe haben. Wir beschränken uns hier darauf, eine Übersicht über die wichtigsten Merkmale und Implementierungsaspekte von Kommandosprachen zu geben.

Kommandos und Kommandoprozeduren. Ein **Kommando** ist eine Aufforderung an das Betriebssystem, einen bestimmten Dienst zu verrichten. Der typischste Effekt eines Kommandos ist, dass ein Programm als eigener Prozess ausgeführt wird. Daneben gibt es Kommandos, die z. B. den Kommandointerpreter selbst oder den Betriebssystemkern steuern, also nicht zum Start eines Prozesses führen. Textuelle Kommandos haben üblicherweise die Form

```
<kommandoverb> <Parameterliste>
```

Das Wort *Kommando* wird meist mit zwei verschiedenen Bedeutungen benutzt:

1. im oben definierten Sinn, z. B. in Sätzen wie *der Benutzer gibt ein Kommando*

2. als Synonym zu Kommandoverb, z. B. in Sätzen wie *das Kopier-Kommando hat 2 Parameter* oder *die Menge der Kommandos kann erweitert werden.*

Was gemeint ist, geht normalerweise aus dem Kontext hervor. Unter UNIX ist das allgemeine Format einer Kommandozeile etwa dieses:

`kommandoverb option argumente`

Die folgende Regeln gelten meistens:

- Das Kommando wird in Kleinbuchstaben eingegeben.
- Mit Optionen kann man die Wirkung eines Kommandos verändern.
- Die Argumente sind z. B. Namen von Dateien, die mit dem Kommando bearbeitet werden sollen. Bei den meisten UNIX-Kommandos kann man gleich mehrere, durch Leerzeichen getrennte Argumente angeben.

Man kann ein Kommando auch ohne Optionen und Argumente aufrufen, zum Beispiel listet das Kommando `ls` die Dateinamen im aktuellen Verzeichnis auf.

Mit dem Kommando `ls -l` kann man eine ganze Zeile von Informationen zu jeder Datei erhalten, z. B. Dateinamen, -größe, -eigentümer usw. Die Option `-l` verändert die normale Ausgabe von `ls` zum langen Format. Man kann die Information zu einer ganz bestimmten Datei anfordern, indem man ihren Namen als Argument angibt, z. B. `ls -l file1`.

Eine **Kommandoprozedur** enthält im einfachsten Fall eine Folge von Kommandos, die bei Ausführung der Kommandoprozedur der Reihe nach ausgeführt werden. Im allgemeinen will man jedoch auf bestimmte Ereignisse reagieren können. Hierzu ein typisches Beispiel: eine Kommandoprozedur enthalte Kommandos zum Übersetzen eines Quellprogramms, zum anschließenden Binden und zum Ausführen des gebundenen Programms. Wenn nun beim Übersetzen ein gravierender Fehler auftritt, macht das Binden und Ausführen keinen Sinn mehr, sollte also unterbleiben. Dementsprechend sind meist bedingte Verzweigungen in der Kommandoprozedur programmierbar. Im allgemeinen sind sogar beliebig komplexe Ablaufstrukturen denkbar.

Als ein Beispiel betrachten wir das Kommando `make` unter UNIX. Ein etwas umfangreicheres Programm besteht aus mehreren Moduln, die jeweils in separaten Dateien abgelegt sind. Je komplexer das Programm wird, um so mehr Dateien müssen verwaltet werden. Wird der Inhalt einer Datei verändert, ist zu entscheiden, welche Übersetzungs- und Binde-Vorgänge anschließend durchzuführen sind, damit das läuffähige Programm dem *neuesten Stand* entspricht. Durch die Änderung einer Datei können davon Dutzende anderer Quelldateien betroffen sein, die neu zu übersetzen und zu binden sind. Das Kommando `make` dient dazu, nach einer Änderung nur die tatsächlich notwendigen Aktionen durchzuführen, um das gesamte Programm auf den neuesten Stand zu bringen. Dafür benötigt `make` eine Beschreibung der Abhänigkeiten zwischen den Dateien, aus denen das Programm entsteht, und die Definition der auszuführenden Aktion. Diese Angaben werden in einer Datei abgelegt, die üblicherweise den Namen `Makefile` trägt. Der Kommando-Aufruf `make` genügt dann, um nach einer Änderung das Programm mit einer minimalen Anzahl notwendiger Aktionen neu zu erzeugen. Das Kommando liest im aktuellen Verzeichnis die Datei `Makefile` und erhält so Kenntnis über die Abhängigkeiten zwischen den Dateien. Es prüft das Modifikationsdatum jeder Datei, ermittelt damit, welche Dateien geändert wurden, leitet aus den im `Makefile` beschriebenen Abhängigkeiten die notwendigen Aktionen ab und führt sie aus, siehe Abbildung 7.4.

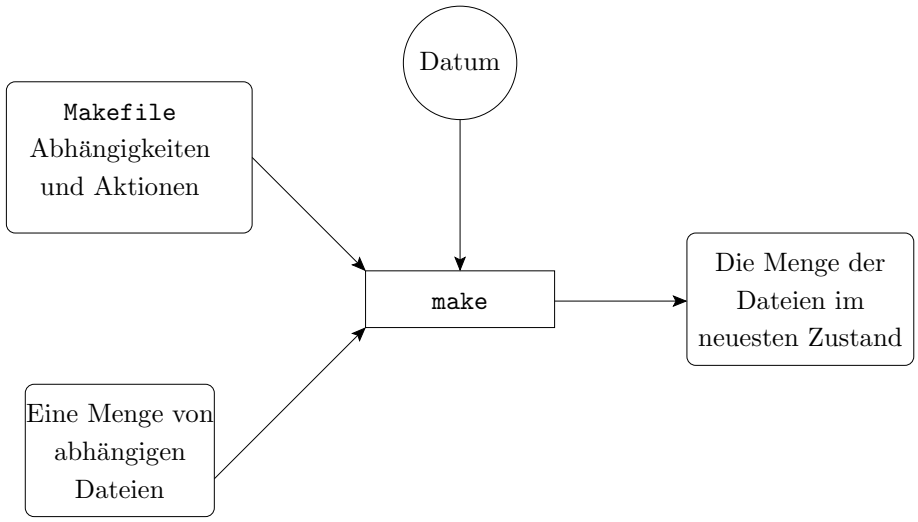


Abbildung 7.4: Funktionsprinzip des Kommandos `make`.

Ein einfaches `Makefile` für das Programm `forkdemo` von Seite 248 sieht so aus:

```
all: forkdemo beeper
forkdemo: forkdemo.c
        cc -o forkdemo forkdemo.c
beeper: beeper.c
        cc -o beeper beeper.c
```

Die Zeilen mit den Kommandos müssen durch ein Tabulatorzeichen und nicht etwa durch Leerzeichen eingerückt sein. Das `Makefile` ist folgendermaßen zu verstehen:

Das ausführbare Programm `forkdemo`, das Ziel oder Target, hat den Namen `forkdemo`. Es hängt von dem Modul im Quellcode `forkdemo.c` ab. Es entsteht durch den Aufruf

`cc -o forkdemo forkdemo.c`

Entsprechendes gilt für das Programm `beeper`. Durch `make forkdemo` bzw. `make beeper` wird der Compileraufruf gestartet, wenn das Änderungsdatum der Quelldatei neuer ist als das der Zieldatei. Falls nichts zu tun ist, weil die Programme schon nach der letzten Änderung der Quelldatei erzeugt wurden, dann antwortet `make` mit `nothing to be done`.

Die Zeile `all: forkdemo beeper` am Anfang dient dazu, die Ziele anzugeben, die überprüft und ggfs. erzeugt werden sollen, wenn das Kommando `make` ohne Argumente ausgeführt wird.

Wenn man jetzt nur die Datei `beeper.c` ändert, dann wird durch den Aufruf `make` das Programm `beeper` neu compiliert. Ein `Makefile` ist ähnlich aufgebaut wie ein Backrezept: erst werden die Zutaten aufgelistet, dann folgen die Anweisungen. Man beginnt mit dem Ziel und geht rückwärts bis zu den Quellen. In diesem einfachen Fall übrigens entspricht unser `Makefile` exakt den üblichen Standard-Regeln von `make` in UNIX, man hätte also die Zeilen mit `forkdemo` und `beeper` ganz weglassen können (nicht aber die Zeile mit `all`), um dasselbe Resultat zu erzielen.

Im Vergleich mit konventionellen Programmiersprachen entspricht ein Kommando in etwa einem einfachen Statement, z. B. einer Zuweisung oder einem Prozedur-

aufruf, das Kommandoverb einem Statement-Typ oder einem Prozedurnamen und eine Kommandoprozedur dem Rumpf einer Prozedur oder einem Hauptprogramm. Programmiersprachen und Kommandosprachen weisen teilweise erhebliche Unterschiede auf, die sich aus der Semantik der Sprache und den Randbedingungen bzgl. der Benutzung und Realisierung ergeben und von denen wir einige noch bei passender Gelegenheit diskutieren werden. An dieser Stelle sei bereits erwähnt, dass Kommandos auch innerhalb von Kommandoprozeduren praktisch immer interpretiert, also nicht im konventionellen Sinn übersetzt werden. Vor der eigentlichen Ausführung werden sie zunächst oft textuell umgeformt, wobei von Makro-Assemblierern bekannte Techniken verwendet werden. In einigen Fällen sind auch allgemein verwendbare Textprozessoren in die Kommandosprache dergestalt integriert, dass Kommandos teilweise von Textprozessoren generiert werden.

Einfluss der Betriebsart. Wir hatten bereits in Kurseinheit 1 mehrere Betriebsarten kennengelernt; wichtig waren vor allem der interaktive Betrieb und der Stapelbetrieb. Diese beiden Betriebsarten unterscheiden sich erheblich darin, wie Kommandos an das Betriebssystem übergeben werden:

- Im interaktiven Betrieb wird immer genau ein Kommando vom Benutzer editiert, dann an das Betriebssystem übergeben und von diesem ausgeführt. Das Kommando kann textuell oder graphisch sein, auf diesen Unterschied gehen wir später genauer ein. Wichtig ist hier vor allem, dass die Eingabe und ggf. das Korrigieren des Kommandos gut unterstützt wird und dass dem Benutzer sofort anschließend sinnvolle Informationen über den Erfolg und den erzielten Effekt des Kommandos gegeben werden.

- Im Stapelbetrieb wird immer eine Kommandoprozedur an das Betriebssystem übergeben, z. B. als Inhalt einer Textdatei, graphische Sprachen eignen sich nicht zur Notation von Kommandoprozeduren. Es sollte natürlich auch möglich sein, Kommandoprozeduren interaktiv aufzurufen, um z. B. Kommandosequenzen zusammenfassen zu können.

Manche älteren Betriebssysteme haben völlig unterschiedliche Kommandosprachen für Dialog- und Stapelbetrieb. Dies ist eine Zumutung für die Benutzer und technisch nicht erforderlich. Manche Kommandos zur Ablaufsteuerung von Stapeljobs oder Kommandoprozeduren sind im Dialogbetrieb natürlich sinnlos und sollten dann einfach ignoriert oder zurückgewiesen werden; dies gilt analog für den umgekehrten Fall.

Anforderungen an Kommandosprachen. Selbst bei vergleichbaren Randbedingungen sind die Kommandosprachen oft sehr verschieden und von sehr unterschiedlicher *Güte*. Es stellt sich natürlich die Frage, was denn die Qualitätsmerkmale einer Kommandosprache sind und welche Anforderungen gestellt werden sollten. Aus schon oben erwähnten Gründen können nicht alle Kommandosprachen gleich sein, weil sie sich an bestimmte Gegebenheiten anpassen müssen. Andererseits gibt es eine Reihe von relativ abstrakten Anforderungen, die unabhängig von diesen Gegebenheiten gültig sind:

- Eine Kommandosprache sollte natürlich die vollständige und effiziente Ausnutzung der Leistungen eines Rechners ermöglichen.

- Die Kommandosprachen sollten, so weit wie möglich und sinnvoll, unabhängig von der Betriebsart sein.

- Kommandosprachen sollten *benutzungsfreundlich* sein[8]. Sie sollten leicht erlernbar und bequem handhabbar sein. Die Sprachkonzepte sollten konsistent und klar sein. Bei interaktiven Systemen sollten softwareergonomische Standards eingehalten werden, z. B. DIN 66234, Teil 8: Grundsätze der Dialoggestaltung, und es sollte ein Hilfesystem vorhanden sein.

- Kommandosprachen sollten *an individuelle Benutzerbedürfnisse adaptierbar* sein. Dies betrifft einerseits zumeist kleinere, manchmal nur kurzfristige Änderungen, die ein Benutzer für den Eigengebrauch vornimmt, um so eine *private Variante* der Kommandosprache benutzen zu können. Dies betrifft andererseits - speziell bei Universalrechnern - die Gesamtmenge aller Benutzer aller Rechner mit einem bestimmten Betriebssystem: hier ist von einer sehr großen Bandbreite der Vertrautheit der Benutzer mit einem Rechner auszugehen. Die Extremfälle sind gelegentliche Benutzer ohne allgemeine EDV-Erfahrung und professionelle Benutzer Entwickler, Systemverwalter usw.. Die Bandbreite ist so groß, dass man i. A. nicht einfach eine sehr komplexe Sprache für professionelle Benutzer vorsehen kann und hieraus eine bestimmte Teilmenge von elementaren Kommandos als Sprache für unerfahrene Benutzer: eine zufriedenstellende Lösung lässt sich meist nur erzielen, wenn man völlig verschiedene Kommandosprachen, die verschiedenen Funktionsumfang haben können, für bestimmte Typen von Benutzern zur Verfügung stellt.

Leider muss man sagen, dass die Betriebssysteme für konventionelle Großrechner, die in den 1960er und 70er Jahren entwickelt wurden und innerhalb derer in vielen Bereichen wie Prozesssteuerung, Hauptspeicherverwaltung, Datenschutz u. a. wegweisende Konzepte entwickelt wurden, schlechte bis absolut katastrophale Kommandosprachen hatten, vor allem im Hinblick auf die Benutzungsfreundlichkeit. Diese Sprachen waren so schwer erlernbar und benutzbar, dass man selbst für relativ triviale Aufgaben Spezialisten heranziehen musste, und *normale* Benutzer sich tunlichst darauf beschränken sollten, einige wenige unbedingt erforderliche Kommandos auf Lochkarten zu kopieren und nichts daran zu ändern. Über die Ursachen für diesen desolaten Zustand kann man viel spekulieren, was hier unterbleiben soll. Man kann aber sicher sagen, dass einer der Gründe für den enormen Erfolg, den PCs und UNIX-Systeme seit Ende der 70er Jahre haben, deren viel bessere Kommandosprachen bzw. Benutzungsschnittstellen sind.

Architektonische Aspekte. Die Forderung nach benutzertypspezifischen Kommandosprachen, die sich aus Überlegungen zur Ergonomie ergab, führt mehr oder weniger direkt zu der *architektonischen* Forderung, dass *der Kommandointerpreter*

[8]Diese Forderung ist nicht sonderlich spezifisch für Betriebssysteme, sondern für alle Software-Systeme mehr oder weniger gleichartig zutreffend.

eines Betriebssystems auswechselbar sein sollte. Damit ist gemeint, dass ein ausreichend qualifizierter Entwickler den (bzw. einen) Interpreter für eine Standard-Kommandosprache durch einen selbstgeschriebenen Interpreter für eine völlig neue Kommandosprache ersetzen bzw. den neuen Interpreter als Alternative verfügbar machen kann. Dies bestärkt ein weiteres Mal die Forderung, die schon in früheren Kurseinheiten mehrfach erwähnt wurde, dass Kommandointerpreter ganz normale Applikationen und kein integraler Bestandteil des Betriebssystemkerns sein sollten. Dies ist auch in softwaretechnischer Hinsicht ein Vorteil, denn hierdurch wird die Betriebssystem-Software besser modularisiert.

Eine weitere Begründung für diese Forderung ergibt sich daraus, dass interaktive Kommandointerpreter mehr oder weniger abhängig von den E/A-Geräten, insb. den Eigenschaften der Terminals, und ggf. der damit zusammenhängenden Graphiksoftware sind. Wenn nun neue Typen von E/A-Geräten verfügbar werden, so können diese i. A. gar nicht oder nicht optimal mit einem *fest eingebauten* Kommandointerpreter zusammenarbeiten. Dieser müsste also angepasst (d. h. erweitert) werden, was wiederum nur dem Hersteller des Betriebssystems möglich ist.

Die Forderung, dass Anwender eigene Kommandointerpreter schreiben und statt des Standard-Interpreters benutzen können, zieht mehrere wesentliche Forderungen an den Betriebssystemkern nach sich. Die wichtigste ist: *die komplette Funktionalität des Betriebssystemkerns muss über die Systemaufrufe verfügbar sein.* Diese Forderung war bei vielen älteren Betriebssystemen nicht erfüllt, d. h. viele wichtige Dienste des Betriebssystemkerns konnten nur über den Kommandointerpreter in Anspruch genommen werden, nicht hingegen innerhalb von Programmen. Unter diesen Umständen war es mehr oder weniger unmöglich, hinreichend mächtige Kommandointerpreter selbst zu entwickeln.

Das Konzept, Kommandointerpreter als normale Applikationen anzusehen, wurde vor allem durch UNIX populär. Aus UNIX (bzw. POSIX) stammt auch die inzwischen weit verbreitete Bezeichnung **Shell** (Schale, Muschel) für solche Kommandointerpreter: Diese Bezeichnung drückt bildlich aus, dass eine Shell eine Benutzungs*oberfläche* um den Betriebssystemkern bildet. Ebenfalls aus UNIX stammt die Bezeichnung (**Shell-**) **Skript** für eine Kommandoprozedur.

Betriebssysteme ohne Kommandosprache. Oft wird der Kommandosprache eine sehr prominente Rolle innerhalb eines Betriebssystems zugewiesen; Betriebssysteme werden bisweilen sogar als Interpreter von Kommandosprachen definiert. Dies führt uns zu der folgenden Frage, die sehr aufschlussreich für die Rolle einer Kommandosprache ist: Muss ein Betriebssystem eine Kommandosprache haben?

Die Antwort ist: im Prinzip nein. Ein Rechner ohne Kommandosprache kann offenbar durchaus *eine* bestimmte Aufgabe erfüllen, denn das Betriebssystem könnte, nachdem es selbst geladen ist, immer ein ganz bestimmtes Programm laden und ausführen. Allerdings könnte dieses Programm, wenn es keine Eingabedaten einlesen oder auf sonstige Weise *von außen* beeinflusst werden kann, immer nur eine einzige Aufgabe erfüllen. Manchmal ist dies der Fall: *Die Elektronik*, die einen Kopierer, einen Schreibautomaten, einen Heizkessel oder eine andere Maschine steuert, führt immer das gleiche Programm aus[9]. Wenn wir also einen Rechner für verschiedene

[9] Man kann sich jetzt trefflich darüber streiten, ob und inwieweit die Tasten, die man am Kopierer oder anderen Maschinen drücken kann, nicht doch eine Art von Kommandosprache darstellen.

Zwecke benutzen wollen und daher zu wechselnden Zeitpunkten verschiedene Programme auf ihm ausführen wollen, ohne jedesmal ein anderes Betriebssystem zu laden, sofern dies überhaupt möglich ist, z. B. durch Austausch eines Datenträgers, brauchen wir also doch eine Kommandosprache und insb. Schnittstellen, über die wir einzelne Kommandos in der Kommandosprache an den Rechner übergeben. Wir konzentrieren uns i. F. auf solche Rechner bzw. deren Betriebssysteme.

7.3 Kommandos

7.3.1 Phasen der Kommandoverarbeitung

Nach der textuellen oder ggf. halbgraphischen Eingabe eines Kommandos folgt, bevor das Kommando *wirklich* ausgeführt wird (also z. B. ein Kindprozess gestartet wird), zunächst eine Vorverarbeitungsphase, in der i. W. Teile des Kommandos textuell ersetzt werden, und zwar zunächst das Kommandoverb und danach die Parameter. Um die verschiedenen Funktionen eines Kommandointerpreters bei der Verarbeitung von Kommandos besser trennen zu können, unterscheiden wir i. F. mehrere Phasen der Verarbeitung eines Kommandos:

1. Eingabe
2. Vorverarbeitung des Kommandoverbs
3. Vorverarbeitung der Parameterliste
4. Ausführung des Kommandos

7.3.2 Ausführung von Kommandos

Nach der Eingabe und Vorverarbeitung eines Kommandos folgt die *eigentliche* Ausführung. Hierauf wollen wir zuerst eingehen, da die Art der Ausführung Einfluss auf die Konzepte zur Vorverarbeitung des Kommandos hat.

Die Mechanismen zum Starten eines Kindprozesses und zur Übergabe von Parametern wurden schon oben in Abschnitt 7.1.2 ausführlich besprochen. Auf die Stapelausführung gehen wir hier ebenfalls nicht weiter ein.

Interne und externe Kommandos. Man unterscheidet generell zwei Arten der Ausführung von Kommandos:

intern: Das Kommando wird innerhalb des Kommandointerpreters ausgeführt. Der Kommandointerpreter ruft hierzu, sofern erforderlich, Systemaufrufe auf.

als Kindprozess (extern): Das Kommando wird als eigenständiger Kindprozess des Prozesses, in dem der Kommandointerpreter ausgeführt wird, ausgeführt. Der Kommandointerpreter startet hierzu ein Programm, welches das Kommando (bei Kommandoprozeduren: den Kommandointerpreter mit der Kommandoprozedur als Eingabedatei) realisiert, und wartet das Ende dieses Prozesses ab.

Die zweite Alternative ist natürlich nur in solchen Betriebssystemen möglich, die parallele Prozesse, Prozesshierarchien und ggf. Interprozesskommunikation anbieten.

Meist sind beide Alternativen gangbar. Beispielsweise könnte ein Kommando `date`, welches die aktuelle Uhrzeit ausgibt, innerhalb des Kommandointerpreters in Form einer kleinen Prozedur realisiert sein, die i. W. den aktuellen Stand der Systemuhr mittels eines Systemaufrufs liest, das Ergebnis in eine alphanumerische Darstellung konvertiert und schließlich samt einiger Steuerzeichen auf das Standard-Ausgabegerät ausgibt. Die gleiche Prozedur könnte aber auch als selbständiges Programm übersetzt und ausgeführt werden. Nachteilig bei der externen Realisierung ist natürlich der viel höhere Aufwand, der i. W. durch das Erzeugen und Vernichten des Prozesses entsteht.

Ein wesentlicher Unterschied zwischen einer internen und externen Realisierung ist, dass bei einem Kindprozess ein völlig neuer Prozesszustand erzeugt wird, der mit Ausnahme von Initialisierungen z. B. von Umgebungsvariablen unabhängig vom Zustand des Prozesses des aufrufenden Kommandointerpreters ist. Der interne Zustand eines Kommandointerpreters, zu dem z. B. die Liste der bisher bearbeiteten Kommandos zählt, wird *nicht* auf einen Kindprozess vererbt. Kommandos, die

- den internen Zustand oder die Umgebung (vgl. Abschnitt 7.1.2) des Kommandointerpreterprozesses verändern oder
- die Teile des Zustands des Kommandointerpreterprozesses lesen, die nicht an den Kindprozess vererbt werden,

können daher nur intern ausgeführt werden: Beispiele sind die Kommandos `cd` und `history`. `cd` ändert das aktuelle Arbeitsverzeichnis, und `history` gibt die letzten aufgerufenen Kommandos aus.

Ausführung von Kommandoprozeduren. Kommandoprozeduren können sowohl intern wie extern ausgeführt werden.

Am einfachsten werden sie extern in einem Kindprozess des Kommandointerpreterprozesses ausgeführt, in dem wieder ein Kommandointerpreter ausgeführt wird, der die Zeilen der Datei, die die Kommandoprozedur enthält, durch Umlenkung des Standard-Eingabegeräts genauso wie interaktive Eingaben verarbeitet. Diese Realisierung ist einfach und elegant, erlaubt beliebiges Schachteln von Kommandoprozeduren und sogar Rekursion und erlaubt es zusätzlich, eine Kommandoprozedur mit einem anderen Kommandointerpreter auszuführen als demjenigen, von dem aus der Aufruf erfolgte, der gewünschte Interpreter kann z. B. durch ein spezielles Kommando in der ersten Zeile der Kommandoprozedur angegeben werden. Wenn die Kommandoprozedur als Hintergrundprozess oder im Stapel ausgeführt werden soll, ist auf alle Fälle die externe Ausführung erforderlich.

Da das Performance-Problem hier zweitrangig ist, könnte man meinen, allein mit der externen Ausführung auskommen zu können. Dies ist jedoch nicht der Fall: Wenn eine Kommandoprozedur intern ausgeführte Kommandos enthält, die im aufrufenden Interpreter wirksam werden sollen, ist die externe Ausführung unsinnig, denn diese Kommandos wirken dann auf den Kommandointerpreter des Kindprozesses, nicht auf den des Elternprozesses. Musterbeispiel für solche Kommandoprozeduren sind die Initialisierungsprozeduren der UNIX-Shells. Die C-Shell (eine der UNIX-Shells) beispielsweise führt, nachdem sie gestartet wurde und bevor sie das erste

Benutzerkommando einliest, die Kommandoprozedur in der Datei `.cshrc` aus. Diese Kommandoprozedur enthält üblicherweise Kommandos, durch die ein Benutzer den Suchpfad setzt und diverse Einstellungen an der Shell gemäß den persönlichen Präferenzen vornimmt; es handelt sich also durchweg um intern auszuführende Kommandos.

Um eine Kommandoprozedur wahlweise intern oder extern ausführen zu können, muss es hier zwei verschiedene Formen des Aufrufs geben. In UNIX wird im Normalfall extern ausgeführt. Wenn interne Ausführung gewünscht ist, dann muss hierzu das interne Kommando `source` mit dem Namen der Datei, die die Prozedur enthält, als Parameter benutzt werden.

7.3.3 Eingabe von Kommandos

Bei der Ausführung von Kommandoprozeduren ist die Eingabe einzelner Kommandos aus Sicht des Kommandointerpreters mehr oder weniger trivial: es muss nur ein entsprechender Teil der Datei, die die Kommandoprozedur enthält, gelesen werden. Meist enthält eine Zeile genau ein Kommando. Es sind aber auch *Fortsetzungszeilen* möglich und manchmal können mehrere Kommandos in eine Zeile geschrieben werden. Komplizierter werden die Verhältnisse bei der interaktiven Eingabe von Kommandos. Wie wir gleich sehen werden, kann man die Eingabe eines Kommandos oft eher als einen *Mikrodialog* zwischen Benutzer und Betriebssystem auffassen.

Textuelle und graphische Eingabe von Kommandos. Textuelle Kommandos sind Folgen von alphanumerischen Zeichen *die Kommandozeile*. Im interaktiven Betrieb werden sie über eine Tastatur eingegeben. Beispielsweise könnte der Text `rm a.out` bedeuten, dass eine Datei namens a.out gelöscht werden soll.

Graphische Kommandos erfordern ein graphikfähiges Ausgabemedium und *Zeigegeräte* wie Maus, Trackball o. ä. Hier sind - zumindest aus Benutzersicht - sehr vielfältige Kommandoformen denkbar: Drücken von Buttons oder Anklicken von Menü-Einträgen, Verschieben von Skalenreitern, Verschieben von Symbolen usw. Das Kommando zum Löschen unserer Datei a.out könnte beispielsweise darin bestehen, dass man mit einer Maus die Anzeigemarke auf einem Bildschirm über das Symbol für die Datei a.out bewegt, dann den linken Knopf der Maus drückt und festhält, dann die Maus so bewegt, dass die Anzeigemarke über das Symbol für einen Mülleimer wandert, wobei das Symbol für unsere Datei mitwandert, und schließlich den linken Knopf wieder loslässt.

Aus einer hardwarenahen Sicht ist bei den obigen Kommandos i. W. folgendes passiert:

Bei der Eingabe des textuellen Kommandos über eine Tastatur sendet die Tastatur einzelne Zeichen an eine Schnittstelle im Rechner[10]. Das Betriebssystem gibt das gleiche Zeichen sofort wieder auf den Bildschirm aus, dieses *Echo* wird meist schon im Treiber für die Schnittstelle realisiert. Die Eingabe eines speziellen Zeichens, z. B. des `carriage return` -Zeichens, zeigt das Ende des Editierens des Kommandos an. Das in einem Pufferbereich gespeicherte vollständige Kommando wird nun weiterverarbeitet.

[10] Auf Terminals, die im Block-Modus arbeiten können, gehen wir hier nicht näher ein.

Das Editieren eines Kommandos ist vergleichbar mit dem Editieren einer Zeile in einem beliebigen interaktiven Texteditor. Daher sollten auch die aus Text-Editoren bekannten Editierfunktionen verfügbar sein: Verschieben des Cursors innerhalb des Kommandos, Einfügen von Zeichen, Löschen von Zeichen oder Worten usw.[11]

Bei dem oben beschriebenen graphischen Kommando ist folgendes passiert: Die Maus sendet, wenn eine Taste heruntergedrückt oder losgelassen wird oder wenn die Maus in bestimmte Richtungen bewegt wird, entsprechende Zeichenfolgen, die meist mit dem sogenannten `escape` -Zeichen beginnen und die man deshalb **escape-Sequenzen** nennt, über eine Schnittstelle an den Rechner[12]. Der Rechner verarbeitet jede einzelne escape-Sequenz sofort und verändert insb. den Bildschirminhalt entsprechend, z. B. werden Menüs oder Formulare angezeigt, Fensterinhalte verschoben usw. Der Gesamteffekt einer einzelnen escape-Sequenz hängt stark vom logischen Bildschirminhalt und der aktuellen Stellung der Anzeigemarke ab. Die Verwaltung der entsprechenden Zustandsdaten ist recht komplex und macht spezielle softwaretechnische Strukturen erforderlich, worauf wir hier nicht näher eingehen wollen. Nachdem das Ende der Kommandoeingabe erkannt wird, wird das Kommando ausgeführt.

Die graphische Eingabe *eines Kommandos* besteht also aus einer größeren Zahl von Zwischenschritten auch aus Sicht des Benutzers, so dass auch hier eine Art von Mikrodialog vorliegt. Eine ähnliche dialogorientierte, sogenannte *halb-graphische* Kommandoeingabe mit Menüs ist übrigens auch in gewissem Umfang auf alphanumerischen Terminals unter Benutzung der Cursor-Tasten realisierbar.

Insgesamt erkennt man unschwer, dass es sehr verschiedene Formen geben kann, in der *das gleiche Kommando* gegeben werden kann. Die eigentliche Wirkung von Kommandos sollte daher unabhängig vom Kommandointerpreter im Betriebssystemkern oder einer nahe darüberliegenden Schicht realisiert und über einen Systemaufruf oder eine Bibliotheksfunktion zugänglich sein, während alle Aspekte der Auswahl von Kommandos, von Parametern oder von Optionen sowie der Interaktion mit einem Benutzer im Kommandointerpreter realisiert sein sollten.

Wir gehen i. F. von textuellen Kommandos aus. Aufgrund der vorstehenden Überlegungen gilt das Folgende aber auch analog für graphische Kommandos.

---

[11] Viele Kommandointerpreter (insb. in UNIX, z. B. die Bourne-Shell) sind hier absolut unbefriedigend, denn man kann immer nur hinten weiterschreiben oder löschen und nicht innerhalb der Zeile editieren (im Gegensatz zu moderneren, komfortableren Shells wie Korn-Shell `ksh` oder `zsh`, `bash`, `tcsh`). Dies hat wohl historische Ursachen: Diese Systeme wurden in den 1960er Jahren entwickelt, als – heute unvorstellbar – Schreibmaschinenterminals benutzt wurden, d. h. alle Ausgaben mussten ausgedruckt werden; das Editieren innerhalb einer Zeile ist unter diesen Umständen natürlich schwierig. Die – nach heutigen Maßstäben ebenfalls unvorstellbar – langsame Geschwindigkeit der Schreibmaschinen und der Lärm waren wohl auch Gründe, warum seinerzeit die meisten Kommandoverben in UNIX sehr kurz und kryptisch gewählt wurden. Dies ist eines von vielen Beispielen, wo heutige Systeme oder gar internationale Standards an sehr alter, nur noch im Museum zu besichtigender Technologie orientiert sind.

[12] Hierin unterscheidet sich eine Maus überhaupt nicht von einer Tastatur. Funktionstasten auf Tastaturen senden meist auch escape-Sequenzen. Bei manchen Rechnern kann man sogar über die Tastatur die gleichen escape-Sequenzen wie über die Maus eingeben, wobei man u. U. mehrere Tasten gleichzeitig drücken muss, d. h. man kann die Anzeigemarke auf einem Bildschirm auch ohne Maus bewegen!

Hilfen bei der interaktiven Eingabe von Kommandos. Die interaktive Eingabe von textuellen Kommandos kostet Zeit und erfordert vom Benutzer, alle erforderlichen Parameterwerte oder sonstigen Angaben präsent zu haben. Deshalb sollte die Eingabe von Kommandos möglichst effizient sein, zugleich aber auch so sicher, dass die Wahrscheinlichkeit von Verwechslungen oder Irrtümern gering ist. Ferner sollte ein Benutzer auch *während* der Eingabe des Kommandos Hilfen verlangen können. Einige Techniken zur Unterstützung der interaktiven Eingabe von Kommandos werden i. F. vorgestellt.

Abkürzungen von Kommandoverben oder Dateinamen: In manchen Systemen kann statt eines vollen Kommandoverbs oder Dateinamens ein ausreichend langes Präfix verwendet werden, das innerhalb der Menge der Namen aller Kommandos bzw. Dateien eindeutig ist.

Man kann noch einen Schritt weitergehen und an jeder beliebigen Stelle innerhalb eines Namens durch Eingabe eines Sonderzeichens (z. B. escape) bewirken, dass der Kommandointerpreter so viele Zeichen ergänzt, bis zur Entscheidung zwischen möglichen Alternativen wieder eine manuelle Eingabe des Benutzers erforderlich ist. Dies hat den weiteren Vorteil, dass der volle Name am Bildschirm erscheint und kontrolliert werden kann. Sinnvoll ist dies aber nur, wenn sich ein Benutzer bei Bedarf während der Eingabe des Kommandos die möglichen weiteren Eingaben anzeigen lassen kann[13]. Der normale Ablauf der Eingabe eines Kommandos wird hier also durch Ausgaben des Betriebssystems unterbrochen.

Statt beliebiger Präfixe von Kommandoverben sind auch Kurz- und Langformen denkbar.

Dateinamen können unter bestimmten Umständen auch durch Sonderzeichen abgekürzt werden; dies ist jedoch nicht allein für interaktive Eingaben interessant und wird deshalb später in allgemeinerer Form besprochen werden.

Wiederholung früherer Kommandos: Bei der interaktiven Eingabe eines Kommandos können einem Benutzer leicht Tippfehler unterlaufen, die eine erneute Eingabe des Kommandos erforderlich machen. Die erneute Eingabe eines ggf. modifizierten Kommandos sollte möglichst einfach sein. Manche Kommandointerpreter erlauben es, mit Hilfe der Cursortasten frühere Kommandos in die Kommandozeile zu holen und dort zu editieren. Andere Interpreter erlauben es, anhand einer Nummer auf frühere Kommandos Bezug zu nehmen, in der C-Shell kann man z. B. mit dem Kommando `!n` das n-te Kommando erneut aufrufen. Nützlich ist in diesem Zusammenhang ein weiteres Kommando, welches die zuletzt eingegebenen Kommandos noch einmal auflistet, diese können durch andere Bildschirmausgaben inzwischen verschwunden sein.

7.3.4 Vorverarbeitung des Kommandoverbs

Erweiterbare Kommandointerpreter. Ein Benutzer gibt deshalb Kommandos ein, damit der Rechner irgendeine Arbeit verrichtet. Es gibt nun aber drei verschiedenartige Mechanismen, wie ein Rechner Arbeit verrichten kann:

[13] In graphischen oder halbgraphischen Browsern für Dateisysteme ist dies automatisch der Fall, weil dort immer die Namen aller vorhandenen Kommandos bzw. Dateien angezeigt und i. W. ganze Namen angewählt werden.

1. durch Laden und Ausführen eines ladbaren Programms
2. durch Ausführen einer Kommandoprozedur
3. direkt durch den Kommandointerpreter

In den ersten beiden Fällen kommt zusätzlich die Hintergrundausführung als Alternative hinzu. Die Frage ist nun, *inwieweit dieser Unterschied in der Syntax der Kommandosprache sichtbar ist.*

In vielen älteren Betriebssystemen gibt es für das Laden und Ausführen eines Programms ein eigenes Kommando (mit Namen wie `exec` oder `run`)[14]. Analog kann es ein eigenes Kommando zum Ausführen von Kommandoprozeduren z. B. `do` geben. Ein Benutzer muss also immer genau wissen, in welcher Form ein gewisser Dienst im Kommandointerpreter, als ladbares Programm oder als Kommandoprozedur realisiert ist. Zu allem Überfluss kann die Art, wie Parameter übergeben werden, bei allen Arten verschieden sein.

Im Gegensatz dazu ist bei modernen Betriebssystemen die Syntax zum Aufruf von allen drei Arten von Diensten gleich, was natürlich viel angenehmer ist. Ausführbare Programme oder Kommandoprozeduren werden durch Nennung des (Pfad-) Namens der Datei, der sie enthält, aufgerufen. Das Betriebssystem entscheidet anhand eines Suffix des Dateinamens z. B. `.exe` und `.com` oder durch Inspektion des Dateiinhalts, ob es sich um ein ausführbares Programm oder eine Kommandoprozedur handelt und verfährt dementsprechend. In POSIX muss der Prozess außerdem das 'execute'-Recht für die Datei haben.

Ein weiterer Vorteil des zweiten Ansatzes ist, dass die Kommandosprache sehr leicht *erweiterbar* ist: man kann ein weiteres Kommandoverb ganz einfach hinzufügen oder ein vorhandenes ersetzen, indem man ein Programm oder eine Kommandoprozedur mit dem entsprechenden Namen schreibt. Der Kommandointerpreter wird hierbei natürlich gar nicht verändert; daher lassen sich die Kommandoverben zur Ablaufsteuerung und generell alle Kommandoverben, die innerhalb des Kommandoprozessors realisiert werden, auch *nicht* auf diese Weise ändern.

Wegen der erheblichen Vorteile gehen wir i. F. von *erweiterbaren* Kommandointerpretern aus.

Suchpfade. Aus organisatorischen Gründen wird man normalerweise die ladbaren Programme und Kommandoprozeduren in verschiedenen Verzeichnissen anordnen, z. B. je ein Verzeichnis für die Standard-Kommandos des Betriebssystems hier im Sinne von Kommandoverben, für die lokalen Erweiterungen, für jedes größere Programmpaket usw. Um diese Kommandos aufrufen zu können, muss eigentlich der *absolute* Pfadname angegeben werden, dies gilt übrigens auch für den Systemaufruf `execve` in POSIX, was wegen der Länge der Pfadnamen ziemlich unbequem ist. Man würde meist lieber nur den lokalen Namen der Dateien innerhalb der Verzeichnisse angeben.

Dieses Problem wird in den UNIX-Shells wie folgt gelöst: man gibt der Shell mittels einer Umgebungsvariablen eine Liste von Verzeichnissen, den sogenannten **Suchpfad**, an. Die Verzeichnisse werden in der angegebenen Reihenfolge nach einem vorgegebenen Kommandoverb durchsucht, das hier ein lokaler Dateiname sein muss.

[14] Im BS2000 beispielsweise kann man darüber hinaus das Laden (incl. Erzeugen eines Prozesses) und Starten der Ausführung trennen (Kommandos `load` und `resume`).

Tatsächlich werden diese Verzeichnisse nicht bei jedem Kommando durchsucht, was zu sehr vielen Plattenzugriffen führen würde, sondern nur beim Setzen des Suchpfads bzw. beim Aufruf eines spezielles Kommando hierfür `rehash`: dann wird eine Shell-interne Zuordnungstabelle aufgebaut, die jedem gefundenen lokalen Namen in einem Verzeichnis den vollständigen Pfadnamen zuordnet. Diese Pufferung hat den Effekt, dass eine neue ausführbare Datei, die in eines der Verzeichnisse eingefügt wird, nicht sofort als neues Kommando zur Verfügung steht, sondern erst nach erneutem Setzen des Suchpfads bzw. nach einem erneuten `rehash`.

Aliase. Wir haben oben schon gesehen, dass man die Menge der Kommandos durch ausführbare Programme und Kommandoprozeduren erweitern kann. Diese bedingen jedoch die relativ aufwändige Ausführung als Kindprozess und benötigen eine zusätzliche Datei zur Speicherung des Kommandos. Bei manchen sehr trivialen Erweiterungen bzw. Änderungen des Kommandovorrats scheint dieser Aufwand unnötig hoch. Beispiele sind das einfache Umbenennen eines Kommandos oder das Umsetzen des Defaultwerts einer Option für einen bestimmten Prozess.

Zur Lösung dieses Problems werden oft Alias-Konzepte eingeführt. Ein **Alias** ist ein neues Kommandoverb; tritt dieses Kommandoverb in einem Kommando auf, so wird es noch innerhalb der Shell, also bevor diese ein Programm startet und vor der Substitution durch vollständige Pfadnamen durch eine beliebige Zeichenkette ersetzt. Diese enthält nun links ein neues Kommandoverb und dahinter beliebige Parameter. Das neue Kommandoverb kann wiederum ein Alias sein, d. h. Aliase können geschachtelt werden. Hierbei können Zyklen auftreten; der Kommandointerpreter muss diese entdecken und dann die Bearbeitung des Kommandos abbrechen.

Aliase können jederzeit mit Hilfe eines intern auszuführenden Kommandos eingerichtet werden und sind nur innerhalb dieser Ausführung des Kommandointerpreters gültig. Z. B. kann man unter UNIX mit

```
alias dir=ls
```

das Kommando `dir` zur Verfügung stellen, das dann dieselbe Bedeutung wie `ls` hat, dies kann sinnvoll sein für Benutzer, die sich das gleichnamige DOS-Kommando nicht abgewöhnen können.

7.3.5 Verarbeitung der Parameter

Parameterübergabeverfahren. Ein Kommando ist vergleichbar mit einer Prozedur in einer algorithmischen Programmiersprache. Es wird identifiziert durch einen Namen, der bei textuellen Kommandos üblicherweise vorne also links steht. Daran anschließend folgen die Parameter. Diese werden durch Kommata, Leerzeichen oder andere Weise voneinander getrennt. Es gibt zwei Methoden, wie aktuelle Parameter formalen Parametern zugeordnet werden:

Stellungsparameter werden durch ihre Position innerhalb der Parameterliste identifiziert. Dies ist die in Programmiersprachen übliche Methode. Der i-te aktuelle Parameter wird dem i-ten formalen Parameter zugeordnet. Für jeden formalen Parameter muss ein aktueller angegeben werden, ggf. ist der Wert undefiniert, was durch die leere Zeichenkette ausgedrückt wird. Für kurze Parameterlisten ist diese Methode gut geeignet.

Kommandos haben häufig eine große Zahl von Optionen, die jeweils zu einem eigenen Parameter führen, dessen aktueller Wert normalerweise nicht angegeben wird, weil der Default-Wert verwendet wird. Wenn die Zahl der Parameter eine bestimmte Grenze $\sim$ 10 überschreitet, sind Stellungsparameter fehleranfällig und, wenn die meisten Parameter undefiniert bleiben, auch ziemlich lästig.

Namensparameter werden durch einen Namen identifiziert. Aktuelle Parameter werden den formalen Parametern anhand dieser Namen zugeordnet. Beim Aufruf des Kommandos sind also der Name des Parameters und sein aktueller Wert anzugeben, z. B. in der Form `option7=yes`. Die Namensparameter können in einer beliebigen Reihenfolge angegeben werden. Nicht aufgeführte Namensparameter haben automatisch den aktuellen Wert undefiniert bzw. einen vordefinierten Wert. Namensparameter sind vor allem in Makrosprachen verbreitet.

Stellungs- und Namensparameter können gleichzeitig verwendet werden. Die Stellungsparameter stehen z. B. im BS2000 immer vorne.

Mengen von Dateien als Parameter. Oft will man das gleiche Kommando mit mehreren Dateien ausführen. Anstatt für jede Datei ein eigenes Kommando ggf. mit Optionen einzugeben, möchte man lieber nur ein Kommando eingeben, in dem alle Dateinamen aufgezählt werden. Ein Beispiel hierfür ist das Kommando unter UNIX
`lpr -printername kap1.pr kap2.3.4.pr kap5.3.pr`
durch das die drei angegebenen Dateien auf einen Drucker mit Name `printername` gedruckt werden sollen. Man möchte auf diese Weise eine *beliebige* Anzahl von Dateien drucken können, d. h. die Parameterliste von `lpr` muss variabel lang sein können.

Die explizite Angabe der Dateinamen ist oft aufwändig und mit viel Tipparbeit verbunden. Daher kann man in den meisten Kommandointerpretern Mengen von Dateinamen mit Hilfe von meist stark vereinfachten regulären Ausdrücken angeben. In UNIX sind z. B. folgende Sonderzeichen sogenannte **wild card characters**, die normalerweise nicht in Dateinamen benutzt werden, vorgesehen:

`*`	steht für eine beliebige Folge von Zeichen. Hierbei gilt folgende Ausnahme: wenn der * am Anfang eines Worts steht, steht er für eine ein beliebige Folge von Zeichen, die nicht mit '.' beginnt.
`?`	steht für ein beliebiges einzelnes Zeichen, z. B. `h?p` steht sowohl für `hop` als auch `hip`, aber nicht für `help`.
`[ ]`	ein beliebiges der zwischen den eckigen Klammern angegebenen Zeichen, z. B. `[kK]apitel` würde auf `kapitel` und auf `Kapitel` passen.

Man beachte, dass die Bedeutung eines Dateinamens, der Sonderzeichen enthält, von den Namen der aktuell vorhandenen Dateien abhängt. Das Kommando
`lpr <optionen> *.pr`
ist also genau dann äquivalent zu dem vorigen, wenn es keine anderen Dateien mit Suffix `.pr` als die drei oben angegebenen Dateien im aktuellen Arbeitsverzeichnis gibt.

Das erste Kommando weicht in einem Punkt ganz wesentlich von dem klassischen Begriff von einer Prozedur bzw. ihrem Aufruf ab, zur Erinnerung: wir hatten bisher die Ähnlichkeit zwischen Kommandos und Prozeduraufrufen betont: die Zahl

der formalen Parameter der *Prozedur* `lpr` ist variabel[15]. Beim zweiten Kommando wird zwar auch eine Menge von Dateinamen spezifiziert, aber nur ein aktueller Parameterwert übergeben; dieser Wert spezifiziert eine Menge von Dateinamen.

Nehmen wir nun weiter an, das `lpr` Kommando sei extern realisiert, also durch ein ausführbares Programm oder eine extern auszuführende Kommandoprozedur. Eine interessante Frage ist nun, welche Parameter der Kommandointerpreter an das Programm bzw. die Kommandoprozedur übergibt.

Eine Alternative ist, immer nur *einen* Dateinamen zu übergeben. In beiden obigen Beispielen würde also das Druckprogramm oder die Druckprozedur für jede angegebene Datei einzeln aufgerufen. Dies entspricht zwar der konventionellen Vorstellung von einer festen Parameterliste, ist aber wegen des wiederholten Ladens des Programms ziemlich ineffizient und soll hier nicht weiter verfolgt werden. Wir nehmen also an, dass das aufgerufene Programm oder die Prozedur nur einmal geladen wird und eine variabel große Menge von Dateien verarbeiten kann.

Die entscheidende Frage ist nun, ob die Sonderzeichen im Kommandointerpreter oder in der Applikation expandiert werden bzw., anders gesagt, wieviel Vorverarbeitung innerhalb des Kommandointerpreters erfolgt.

Die UNIX-Shells sind Beispiele für Kommandointerpreter, die alle Sonderzeichen expandieren. Dem ausführbaren Programm bzw. der Kommandoprozedur wird also eine Liste von Namen existierender Dateien übergeben.

Dies erfordert, dass ein ausführbares Programm bzw. eine Kommandoprozedur eine solche Liste von Parametern überhaupt verarbeiten kann. In vielen klassischen Programmiersprachen ist dies nicht ohne weiteres möglich. In C, der Standard-Programmiersprache in UNIX, ist dies hingegen sehr einfach möglich: die Liste der Parameter wird dem Programm als ein Array von Strings übergeben (vgl. oben Abschnitt 7.1.2), der mit Hilfe diverser Operatoren in C verarbeitet werden kann. Das Hauptprogramm unseres Druckprogramms z. B. ist eine Schleife, in der die vorhandenen Parameter inspiziert und für jeden gefundenen Dateinamen der eigentliche Druckvorgang ausgeführt wird. Die Kommandosprachen der UNIX-Shells enthalten ebenfalls Kommandos, durch die Listen von Texten leicht verarbeitet werden können.

Sonderzeichen können natürlich auch von der Expansion in der Shell geschützt werden.

Ein Beispiel für einen Kommandointerpreter, der Sonderzeichen nicht expandiert, ist der in MS-DOS. Die Sonderzeichen müssen also im Applikationsprogamm expandiert werden. Hierzu können diese durch zwei Systemaufrufe den ersten bzw. nächsten Dateinamen zu einem Ausdruck holen, wenn sie wollen, sie können die Expansion aber auch selbst nach eigenen Regeln durchführen. Die Expansion der Sonderzeichen wird, wenn sie sich auf diese Systemaufrufe stützt, also vom Betriebssystemkern festgelegt und kann auch nicht durch Austausch des Kommandointerpreters verändert werden.

[15]Dies darf nicht mit Namensparametern verwechselt werden: Bei Namensparametern ist die Menge der formalen Parameter fest, lediglich die aktuellen Parameter, deren Wert undefiniert bleiben soll, werden nicht aufgezählt. Die obigen Dateinamen sind natürlich keine aktuellen Namensparameter.

Ein ganz wesentlicher Vorteil der Expansion von Sonderzeichen im Kommandointerpreter ist, dass dieses nicht erneut in jedem Programm realisiert werden muss und dass die Bedeutung der Sonderzeichen daher immer innerhalb eines Kommandointerpreters gleich ist. Bei Expansion der Sonderzeichen innerhalb der Programme bzw. Kommandoprozeduren haben diese i. A. keine einheitliche Bedeutung, was den Lernaufwand erhöht und leicht zu schwerwiegenden Fehlern führen kann. Hinzu kommt, dass der Zeitpunkt der Expansion der Sonderzeichen nicht einheitlich ist; dies kann zu einigen subtilen Fehlern führen.

Nicht anwendbar ist die Expansion von Sonderzeichen im Kommandointerpreter, wenn für das Kommando Namen von noch nicht existierenden Dateien benötigt werden. Das Kommando

```
copy *.v1 *.v2
```

das alle Dateien mit Suffix `.v1` auf entsprechende Dateien mit Suffix `.v2` kopieren soll, ist in UNIX *nicht* mit Hilfe eines Kopierprogramms realisierbar, selbst wenn dieses eine Liste von Dateinamen mit geradzahliger Länge entsprechend verarbeiten würde: die Shell behandelt beide * gleich und expandiert sie *vor* dem Aufruf des Kopierprogramms. Zu diesem Zeitpunkt existieren die neuen Namen der Dateien aber noch gar nicht. In UNIX muss ein entsprechendes Programm oder eine Shell-Prozedur mit anders gestalteten Parametern geschrieben werden, um den hier gewünschten Effekt zu erzielen. Beim Aufruf dieses Programms müssten die Parameter vor der Expansion durch die Shell geschützt werden. In MS-DOS funktioniert der obige Befehl wie erwartet, weil das Kopierprogramm den zweiten * völlig anders verarbeitet als den ersten.

Kommandosubstitution. Wir haben oben mit den Sonderzeichen in Dateinamen bereits eine Möglichkeit kennengelernt, wie zumindest bei der UNIX-Variante der Parameterteil eines Kommandos sozusagen textuell modifiziert wird, bevor z. B. ein Kindprozess mit den so erhaltenen Parametern gestartet wird. In diesem Abschnitt stellen wir ein weiteres Verfahren vor, mit dem ebenfalls in den UNIX-Shells der Kommandotext modifiziert werden kann: die **Kommandosubstitution**. Die Grundidee ist einfach: Innerhalb eines Kommandos erscheint ein weiteres Kommando, das einen Text auf das Standard-Ausgabegerät schreibt; dieses *innere* Kommando wird zunächst in einer als Kindprozess gestarteten Shell ausgeführt und der Aufruf dann durch den erzeugten Text ersetzt! Als Beispiel betrachten wir das folgende Kommando:

```
set prompt = \$username@‘hostname‘:
```

`prompt` ist eine Shell-Variable, die einen Text enthält, den die Shell auf den Bildschirm ausgibt, bevor sie ein Kommando einliest. Das obige Kommando setzt diesen Text auf den Benutzernamen gefolgt von ‘@’, dem Rechnernamen und ‘:’. `username` ist eine Shell-Variable, die den Namen des Benutzers enthält. Das eigentlich Interessante ist der Teil `‘hostname‘`: `hostname` ist ein Kommando, das den Namen des Rechners auf das Standard-Ausgabegerät schreibt. Die Notation ‘...‘ bedeutet, dass das enthaltene Kommando durch seine Ausgabe textuell ersetzt werden soll.

7.4 Variablen und Kontrollstrukturen

Wir hatten schon früher erwähnt, dass Sprachen für Kommandoprozeduren meist nur sehr eingeschränkte Kontrollstrukturen anbieten, dass sie aber eigentlich vollwertige algorithmische Sprachen sein sollten. Sie sollten ähnliche Kontrollstrukturen wie gängige imperative Programmiersprachen wie Ada, C oder Modula-2 anbieten. Kommandos sollten ähnlich wie Prozeduren in Programmiersprachen aufrufbar sein. Da alle Schleifenkonstrukte mit Variablen arbeiten, ist zusätzlich ein Variablenkonzept analog zu dem in Programmiersprachen erforderlich.

7.4.1 Variablen

Viele Kommandointerpreter erlauben es, Variablen zu definieren. Eine **Shell-Variable** ist ein Paar (Bezeichner, Wert). Der Bezeichner unterliegt üblicherweise den gleichen Restriktionen wie in Programmiersprachen, wie erstes Zeichen ein Buchstabe, beschränkte Länge usw. Neue Variablen können jederzeit deklariert werden. Meist ist nur ein einziger Wertebereich bzw. Datentyp möglich, nämlich Texte über einem bestimmten Alphabet. Sofern der Wert einer Variablen die alphanumerische Darstellung einer ganzen Zahl ist, kann oft auch mit der so dargestellten Zahl gerechnet werden. Wenn der Text durch Leerzeichen oder andere Trennzeichen in Worte geteilt ist, kann manchmal auch direkt auf das n-te Wort zugegriffen werden, etwa in der Form `variablenname[n]`. In diesem Fall kann man den Typ der Variablen eher als einen Array von Worten auffassen.

Der Wert von Variablen oder von Worten innerhalb einer Variablen kann natürlich jederzeit neu gesetzt werden. Dies geschieht meist durch ein spezielles Kommando z. B. `set variablenname = neuerwert`, das natürlich intern auszuführen ist.

Auf den ersten Blick wirken Shell-Variablen und Umgebungsvariablen sehr ähnlich, siehe Abschnitt 7.1.2. Der entscheidende Unterschied liegt darin, dass Shell-Variablen nur lokal im Kommandointerpreter existieren und nicht an Kindprozesse weitervererbt werden; ist die Vererbung gewünscht, müssen sie auf Umgebungsvariablen kopiert werden.

Der Wert einer Variablen kann in Kommandos textuell eingesetzt werden. Hierzu ist bei der Eingabe bzw. Aufschreibung des Kommandos der Name der Variablen zusammen mit einem Operator in der C-Shell durch Voranstellen eines `$` anzugeben. Wenn beispielsweise die Dateien zu einem größeren Anwendungssystem XY im Verzeichnis `/usr/local/XY` stehen, könnte man folgende Kommandofolge zum Starten des Systems benutzen:

```
set XY = /usr/local/XY
$XY/load $XY/fonts $XY/init
```

Das Kommando `/usr/local/XY/load` wird also mit den Parametern `/usr/local/XY/fonts` und `/usr/local/XY/init` aufgerufen.

Variablen können auch vordefiniert sein. Die C-Shell hat beispielsweise u. a. folgende vordefinierten Variablen:

`argv`	Liste der Parameterwerte, die beim Aufruf der Shell angegeben worden sind
`cwd`	absoluter Pfadname des aktuellen Arbeitsverzeichnisses

`filec` Schalter, mit dem die automatische Ergänzung von Dateinamen an- und abgeschaltet werden kann, siehe Abschnitt 7.3.3.

`home` *Heimverzeichnis* (home directory) eines Benutzers , dieses Verzeichnis ist nach dem Login das aktuelle Arbeitsverzeichnis

`path` der Suchpfad, eine Liste von Namen von Verzeichnissen, in denen nach ausführbaren Dateien gesucht werden soll.

7.4.2 Ablaufsteuerung

In Kommandosprachen finden sich fast alle Ablaufsteuerungskonstrukte wieder, die man schon aus konventionellen Programmiersprachen kennt. Oft werden sie aber wesentlich modifiziert, um den besonderen Bedürfnissen der *programmierten* Abgabe von Kommandos gerecht zu werden. Als Beispiel sei die folgende `for`-Schleife genannt, die man in der Bourne-Shell (ebenfalls eine der UNIX-Shells) programmieren kann:

```
for i in *
do
   cp $i $i.alt
done
```

Diese Prozedur kopiert jede Datei im aktuellen Verzeichnis auf eine Datei mit gleichem Namen verlängert um das Suffix `.alt`. Der * wird wie üblich in die Menge der Namen im aktuellen Verzeichnis expandiert. Hinter dem '`in`' könnte statt des * auch eine explizite Liste von Worten stehen.

Die UNIX-Shells bieten hier Möglichkeiten, die weit über die meisten anderen Kommandosprachen bzw. -Interpreter hinausgehen, weil sie durch die Kommandosubstitution eine enge Integration beliebiger textverarbeitender Programme in Kommandoprozeduren erlauben. Dies sei am folgenden Beispiel erläutert:

```
for i in `grep kap namensliste`
do
  cp $i $i.alt
done
```

Diese Kommandoprozedur kopiert Dateien wie im vorigen Beispiel, allerdings nur die Dateien, deren Name in der Datei `namensliste` steht und den String 'kap' enthält. Im Vergleich zum letzten Beispiel ist der *, der innerhalb der Shell expandiert wird, ersetzt worden durch ein ganz normales Kommando, das die Liste der Namen der zu verarbeitenden Dateien erzeugt, die dann im Rahmen der Kommandosubstitution in das `for`-Kommando eingesetzt werden. Im Beispiel verwenden wir das Kommando `grep`, welches die als zweiten Parameter angegebene Datei liest und auf das Standard-Ausgabegerät die Sätze schreibt, die das als ersten Parameter angegebene Textmuster enthalten. Durch die Kommandosubstitution kann man praktisch beliebige textverarbeitende Programme einsetzen, um z. B. Schleifen zu steuern oder Dateinamen zu generieren.

Das obige Beispiel zeigt auch, dass Textverarbeitungsfunktionen einen ganz wesentlichen Bestandteil der Kommandosprachen bilden, weil z. B. Dateinamen durch

solche Funktionen generiert werden.

7.4.3 Softwaretechnische Aspekte

Aus einer softwaretechnischen Sicht ist die Entwicklung von Kommandoprozeduren im Prinzip durchaus vergleichbar mit der Entwicklung von *normalen* Programmen. Daher sollten im Prinzip die Methoden der Softwaretechnik bei der Entwicklung von Kommandoprozeduren angewandt werden und es sollte eine Programmierumgebung für die Kommandosprache vorhanden sein, die Spezifikation, Codierung, Testen usw. von Kommandoprozeduren unterstützt. Diese prinzipiellen Überlegungen sind allerdings normalerweise mehr oder weniger gegenstandslos, weil die entwickelten Prozeduren relativ klein sind, typischerweise einige hundert Zeilen lang und mit nahezu linearer Ablaufstruktur.

Bei moderneren Kommandosprachen ist dies anders: diese sind z. T. wesentlich mächtiger als konventionelle Programmiersprachen und eignen sich zur Erstellung von größeren Systemen, speziell von Prototypen oder Anwendungen, die keine hohen Performance-Anforderungen haben, Kommandoprozeduren sind wegen der interpretativen Abarbeitung i. d. R. wesentlich langsamer als vergleichbare Programme in Programmiersprachen. Als Musterbeispiel sind hier die typischerweise in UNIX-Systemen vorhandenen Kommandosprachen zu nennen.

Bei UNIX kommt noch hinzu, dass die Kommandosprache, die Programmiersprache C und Textprozessor-Sprachen (wie `grep`, `awk` oder `sed`) tatsächlich einen **Sprachverbund** bilden (s. voriges Beispiel), d. h. einzelne Moduln von Anwendungssystemen sind in verschiedenen Sprachen geschrieben. Eine gemeinsame Benutzung von Programmiersprachen und Kommandosprachen zur Systementwicklung ist auch bei älteren Systemen möglich; allerdings geht dort die Integration der Sprachen nicht sonderlich weit, weil die Kommandosprachen zu primitiv sind.

Festzuhalten bleibt, dass moderne Kommandosprachen ein erhebliches Potential zur Entwicklung von Applikationen besitzen und dass ein Betriebssystem damit aus einer bestimmten Sicht eine Software-Entwicklungsumgebung für Kommandoprozeduren ist, vgl. auch die Bemerkungen zur Trennung zwischen dem Betriebssystem und dem Programmentwicklungssystem in Kurseinheit 1, Abschnitt 1.1.3.

7.5 Zusammenfassung

In dieser Kurseinheit haben wir uns mit der Frage befasst, wie die Benutzer eines Rechners diesem Arbeitsaufträge übergeben können. Hierzu benötigen sie eine Kommandosprache. Der Kommandointerpreter benötigt dann seinerseits Mittel zum Starten von Prozessen.

Das Starten eines Prozesses umfasst das Erzeugen eines Prozesskontrollblocks und eines logischen Hauptspeichers für den Prozess, das Laden des auszuführenden Programms in den Hauptspeicher, die Übergabe von Parametern, das Einrichten von Umgebungsvariablen und das Vererben offener Dateien. Der Betriebssystemkern stellt hierzu Systemaufrufe zur Verfügung, durch die Prozesse erzeugt, gestartet, kontrolliert und beendet werden können. Es erweist sich als sehr vorteilhaft, wenn die gleichen Systemaufrufe auf die Standard-E/A-Geräte eines Prozesses und auf Dateien anwendbar sind: in diesem Fall können die Ein- bzw. Ausgaben eines

Prozesses von den Standardgeräten auf Dateien umgelenkt werden.

Die Kommandosprachen verschiedener Rechner sind sehr unterschiedlich, weil sie sich mehr oder weniger an die Funktionalität des Betriebssystemkerns und Merkmale der Standard-E/A-Geräte anpassen müssen. Dennoch gibt es einige allgemeine Anforderungen: Kommandosprachen sollten unabhängig vom Betriebsmodus, benutzungsfreundlich und adaptierbar sein. Der Kommandointerpreter sollte nicht Teil des Betriebssystemkerns sein, sondern ein ganz normales Anwendungsprogramm.

Ein einzelnes Kommando wird in mehreren Phasen verarbeitet: zunächst wird es bei Kommandoprozeduren aus einer Datei eingelesen und bei interaktiver Eingabe durch einen Editiervorgang erstellt; alternativ hierzu werden bei graphischen Benutzungsschnittstellen graphische Operationen mit auf dem Bildschirm dargestellten Objekten vorgenommen. Danach wird zunächst das Kommandoverb vorverarbeitet, wobei Aliase und/oder Abkürzungen ersetzt werden. Anschließend werden die Parameter in ähnlicher Weise vorverarbeitet. Besonders interessant ist die Möglichkeit, mit Mengen von Dateien zu arbeiten. Nach Abschluss der Vorverarbeitung wird schließlich das *eigentliche* Kommando entweder direkt im Kommandointerpreter, durch Ausführen einer Kommandoprozedur oder durch Starten eines Prozesses ausgeführt.

Kommandosprachen enthalten neben den *elementaren* Kommandos auch noch Möglichkeiten zur Ablaufsteuerung und zum Umgang mit Variablen, die im Prinzip ähnlich wie in konventionellen Programmiersprachen, jedoch in vielen Details auf den besonderen Anwendungsbereich abgestimmt sind. Die Leistungsfähigkeit verschiedener Kommandosprachen variiert in dieser Hinsicht allerdings ganz erheblich. Besonders leistungsfähig ist das Konzept der Kommandosubstitution, durch das beliebige Textprozessoren oder in konventionellen Sprachen geschriebene Programme in Kommandoprozeduren eingebunden und dadurch sehr komplexe Systeme geschaffen werden können.

Literatur

[PCTE97] *Portable Common Tool Environment – Abstract Specification (Standard ECMA-149)*. European Computer Manufacturers Association, Geneva, 1997.

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-149.pdf

[POSIX88] *Portable Operating System Interface for Computer Environments (IEEE Standard P1003.1)*. Draft 12.3, IEEE, 1988/05.

Glossar

Alias (*alias*) Kommandoverb, das im Kommandointerpreter durch eine Zeichenkette (den aktuellen Wert des Alias) ersetzt wird.

Dateikontrollblock (*open file descriptor*) interne Datenstruktur des Betriebssystems, in dem alle erforderlichen Informationen über eine geöffnete Datei enthalten sind.

Dateiidentifizierer (*file handle*) Wert (üblicherweise vom Typ Integer), der einen Dateikontrollblock und damit indirekt eine Datei identifiziert.

Elternprozess (*parent process*) Wenn ein Prozess einen anderen erzeugt hat, bezeichnet man ihn als Elternprozess und den erzeugten Prozess als Kindprozess.

externes Kommando (*external command*) Kommando, das außerhalb des Kommandoprozessors ausgeführt wird, indem dieser einen Kindprozess startet.

internes Kommando (*internal command, Shell command*) Kommando, das durch den Kommandoprozessor selbst ausgeführt wird, ohne einen Kindprozess zu starten.

Kindprozess (*child process, subprocess*) s. Elternprozess.

Kommando (*command*) Aufforderung an das Betriebssystem, eine bestimmte Aktivität durchzuführen. Kann textuell oder graphisch sein. Falls textuell: besteht üblicherweise aus Kommandoverb und Parametern.

Kommandointerpreter (*command interpreter, Shell*) Teil des Betriebssystems, der Kommandos einliest, interpretiert und alles weiter Erforderliche zur Ausführung der Aktivität veranlasst. Sollte nicht Teil des Betriebssystemkerns sein.

Kommandoprozedur (*command procedure, Shell script*) Programm in einer Kommandosprache.

Kommandosubstitution (*command substitution*) textuelles Ersetzen eines innerhalb eines Kommandos erscheinenden weiteren Kommandos durch den Text, den es auf das Standard-Ausgabegerät schreibt.

Kommandoverb (*command verb*) erstes Wort in einem Kommando. Bezeichnet die auszuführende Aktivität (vergleichbar mit einem Prozedurnamen).

Namensparameter explizit benannter Parameter; Name muss beim Aufruf angegeben werden. Der aktuelle Wert wird einem formalen Parameter auf Basis gleicher Namen zugewiesen.

POSIX Spezifikation einer Menge von Systemaufrufen (abgeleitet vom Betriebssystem UNIX).

Prozessidentifizierer (*process identifier*) Wert (üblicherweise vom Typ Integer), der einen Prozess identifiziert. Wird als Parameter oder Rückgabewert in diversen Systemaufrufen zur Prozessverwaltung verwendet.

Shell Synonym zu Kommandointerpreter (aus UNIX).

Shell-Skript Synonym zu Kommandoprozedur (aus UNIX).

Shell-Variable (*Shell variable*) besteht aus Bezeichner (Name) und Wert vom Typ String. Existiert nur in der Shell und wird im Gegensatz zu Umgebungsvariablen nicht auf Kindprozesse vererbt.

Sonderzeichen (*wild card characters*) in Dateinamen üblicherweise nicht benutzte Zeichen (z. B. * oder ?), mit deren Hilfe Ausdrücke (ähnlich wie reguläre Ausdrücke) gebildet werden können, die Mengen von Dateinamen beschreiben.

Standard-E/A-Gerät virtuelles E/A-Gerät, von dem aus ein interaktiver Prozess die Benutzereingaben erhält bzw. auf das er die Ausgaben für den Benutzer schreibt. Bei Stapelausführung analog die Datei, die die Kommandoprozedur enthält bzw. das Protokoll. Wird in manchen Betriebssystemen als Datei ohne Namen behandelt, die innerhalb des Prozesses nicht geöffnet oder geschlossen werden muss.

Stellungsparameter Parameter, bei dem der aktuelle Wert dem formalen Parameter infolge gleicher Position in den Parameterlisten zugewiesen wird.

Suchpfad (*path*) Folge von Namen von Verzeichnissen, in denen nach lokalen Namen von ausführbaren Programmen gesucht wird, wenn ein solcher lokaler Name als Kommandoverb benutzt wird.

Umgebung (*environment*) Menge der Informationen, die einem Prozess von dem aufrufenden Prozess übergeben wird; umfasst Parameter, Umgebungsvariablen und offene Dateien.

Umgebungsvariable (*environment variable*) besteht aus Name und Wert vom Typ String; wird auf Kindprozesse vererbt.

Umlenkung (*redirection*) Ersetzen der Standard-E/A-Geräte durch Dateien für einen Prozess; wird vom aufrufenden Prozess veranlasst und vom Kindprozess nicht bemerkt.

FernUniversität
Gesamthochschule in Hagen

Betriebssysteme

Gesamtinhaltsverzeichnis und Index

Lehrgebiet Praktische Informatik VI

Inhalt

Index

Zur Unterscheidung von normalen Einträgen sind die Seitennummern von englischen Fachbegriffen *schräg* und von UNIX-Befehlen und -Bezeichnungen unterstrichen gedruckt.