Betriebssysteme (SoSe 2005)

Klaus Barthelmann

Inhaltsverzeichnis

Kapitel 1

Einleitung

Worum geht es in der Vorlesung?

1.1 Was ist ein Betriebssystem?

Ein *Betriebssystem* ist ein Programm, das zwischen den Benutzern und der Rechnerhardware vermittelt. Es soll den Benutzern eine Umgebung bieten, in der sie Programme *bequem* und *effizient* ausführen können. Es soll von der konkreten Hardware *abstrahieren* und häufiger gebrauchte *Dienste* anbieten. Das Betriebssystem *verwaltet* die vorhandenen Ressourcen (d. h. die Komponenten des Rechners, aber auch Rechenzeit und Speicherplatz), um eine gute Auslastung sicher zu stellen, aber auch um Konflikte zwischen Konkurrenten aufzulösen.

Eine Rechenmaschine lässt sich in Schichten untergliedern, nämlich die konkrete Architektur, die virtuelle Maschine und die Anwendungsebene, die in Abbildung 1.1 von unten nach oben eingetragen sind.

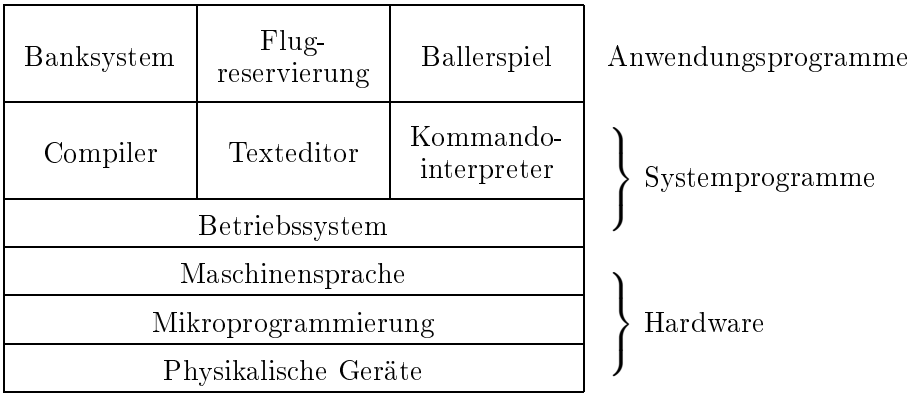


Abbildung 1.1: Aufbau in Schichten

Man sieht, dass das Betriebssystem (der *Kern*) nur einen Teil dessen ausmacht, was man normalerweise als „Betriebssystem“ kauft. Gerade der Kommandointerpreter (heutzutage normalerweise ein grafisches Fenstersystem), mit dem man am unmittelbarsten kommuniziert, gehört zu den Systemprogrammen.

Am besten versteht man, was ein Betriebssystem tut, wenn man die Entwicklungsgeschichte betrachtet. Verschiedene Bedürfnisse entstanden erst im Laufe der Zeit. Und was man an einem Betriebssystem hat, merkt man am besten, wenn man Dienste vermisst.

1.2 Geschichte

Die Entwicklung der Betriebssysteme ist natürlich eng mit der Verbesserung der Technologie und der Entstehung höherer Programmiersprachen verknüpft.

Die ersten Rechner waren sehr große „Taschenrechner“. Ein Programmierer brauchte hohes Spezialwissen und musste sich um alles kümmern. Die Eingabe geschah z. B. bitweise über Schalter. Lochstreifen und -karten waren schon ein Fortschritt. Dafür konnte man beim Rechnen zusehen. Rechner wurden reserviert wie große Forschungsgeräte heutzutage. Das führte zu einer schlechten Auslastung. Mit der Zeit entstanden Bibliotheken für nützliche Programme. Steuerungen (Driver) für Lochkartenleser, Magnetbänder und Drucker wurden geschrieben, ebenso Assembler, Binder (Linker) und Lader. Den Übergang markierten die ersten höheren Programmiersprachen. Für deren Übersetzer (Compiler) mussten häufig die Bänder gewechselt werden. Während der langen Einstellzeiten blieb der teure Rechner ungenutzt.

Zum einen wurden professionelle Betreiber (Operators) eingestellt. Diese waren auf Grund ihrer größeren Erfahrung geschickter. Außerdem konnten sie gleichartige Aufträge zusammenfassen (Batch). So wurde viel Zeit für Umrüstungen gespart. Die normalen Benutzer verloren allerdings den direkten Kontakt mit dem Rechner. Fehler mussten an Hand von Hexdumps gefunden werden. Trotzdem war die menschliche Bedienung immer noch sehr langsam. Die Geschichte der Betriebssysteme begann, als die Abfolge von Programmen automatisiert wurde. Ein rudimentärer Interpretierer für Steuerungskarten musste her. Dieser konnte Aufträge voneinander abgrenzen und (System-)Programme laden lassen. Natürlich gehörten Gerätesteuerungen dazu.

Aber der Geschwindigkeitsunterschied zwischen dem Rechner und seinen Peripheriegeräten führte immer noch zu einer geringen Auslastung der Rechner. Durch schnellere Peripheriegeräte (Magnetbänder statt Lochkarten) wurde der Abstand nur wenig gemindert, weil die Rechengeschwindigkeit ebenfalls zunahm. Aber immerhin. Lochkarten wurden an speziellen Stationen oder kleinen Rechnern auf Magnetbänder kopiert. Ebenso wurden

Magnetbänder ausgedruckt. Solche Stationen konnte man unabhängig vom Rechner betreiben und entsprechend vervielfachen. Für den Rechner bedeuteten die Magnetbänder keine Umstellung (ein Stück Geräteunabhängigkeit); nur die Steuerung musste angepasst werden.

Nun war es außerdem gut, die Magnetbänder vom Rechner abzukoppeln. *Pufferung* erlaubte es, die Magnetbänder vorauszulesen, damit die gewünschten Daten (hoffentlich!) verfügbar waren, bevor sie gebraucht wurden. Ebenso konnten Ausgaben, wenn sie schubweise auftraten, unabhängig vom Rechner auf das Magnetband geschrieben werden. Leider kann ein Puffer nur Unregelmäßigkeiten ausgleichen (und nur so lange er groß genug ist). Die meisten Aufträge haben aber einen Überhang entweder an Rechenzeit oder Ein-/Ausgabezeit. Festplatten erlaubten einen wahlfreien Zugriff auf den Inhalt. Dadurch wurde die Zuordnung zwischen Aufträgen und Magnetbändern aufgehoben; mehrere Aufträge konnten eingelesen werden. Spooling ist also eine radikalere Lösung, die Unregelmäßigkeiten der Auslastung über Auftragsgrenzen hinweg auszugleichen versucht. Der wahlfreie Zugriff erforderte ein Inhaltsverzeichnis auf der Platte.

Spooling erlaubt aber auch, mehrere Aufträge „gleichzeitig“ zu bearbeiten (Multiprogramming) und zwischendurch zu wechseln (Scheduling), wenn ein Auftrag zeitweilig auf die Erfüllung einer Anforderung warten muss. Hier beginnen interessante Aufgaben des Betriebssystems: Verwaltung des Hauptspeichers, Auswahl des nächsten zu bearbeitenden Auftrags, Überwachung ihrer Zustände, gegenseitiger Schutz. Das Zeitscheibenverfahren (Time-Sharing, Time-Slicing, Multitasking, Multiprocessing) geht noch weiter und unterteilt auch die Rechenzeit. Spätestens nach Ablauf eines festgelegten Zeitintervalls wird auf jeden Fall ein anderer Auftrag bearbeitet. Das Zeitscheibenverfahren erzeugt für jeden Auftrag die Illusion, den (verlangsamten) Rechner alleine zu benutzen. Mit so einem System kann man interaktiv (im Dialog) arbeiten. Wenn mehrere Benutzer gleichzeitig aktiv sind, wird ein strukturiertes Dateisystem auf der Festplatte erforderlich, das Verzeichnisse (Directories, Ordner, Folder) und benannte Dateien (Files) erlaubt. Weil der Hauptspeicher schnell zu klein wird, wird die Auslagerung (Swapping) von Teilen des Inhalts auf Platte nötig. Time-Sharing und Batching vertragen sich ganz gut, weil Menschen vergleichsweise langsam sind, dem Rechner also nur ein wenig „Freizeit“ kosten.

Neuere Trends sind *verteilte Systeme*, in denen selbstständige Rechner durch ein Netz miteinander verbunden sind. Vorteile sind die gemeinsame Benutzung von Ressourcen, eine Steigerung der Rechenleistung durch Kooperation, eine höhere Verfügbarkeit (durch Ausweichmöglichkeiten) und, für die einzelnen Benutzer, die Möglichkeit, miteinander zu kommunizieren. *Mobiles Rechnen* treibt dieses Konzept auf die Spitze, indem Rechenleistung zu jeder Zeit und an jedem Ort verfügbar gemacht werden soll. Die einzelnen Rechner schrumpfen zu einem Hilfsmittel, um auf das Netzwerk zuzugreifen.

Echtzeitsysteme werden zur Steuerung zeitkritischer Vorgänge eingesetzt.

Sie bieten eine geringe Antwortzeit durch Verzicht auf gute Auslastung.

1.3 Konzepte

Bei einem Betriebssystem sind im wesentlichen fünf Konzepte zu untersuchen. Abstrakte Schöpfungen sind der *Prozess* (die Ausführung eines Programms, also ein Vorgang, der Rechenzeit, Speicherplatz und andere Ressourcen verbraucht) und die *Datei* (ein Behälter für Daten, der unter einem Namen ansprechbar ist). Abstraktionen konkreter Gegebenheiten sind das *Gerät* und der *Speicher*. Dazu kommen noch die Vorkehrungen dafür, die Rechner in einem *Netzwerk* zu betreiben. Die Idee, jede benötigte Software beim Einschalten des Rechners aus dem Internet zu laden, hat sich allerdings noch nicht durchgesetzt. Bisher besitzen die meisten Computer eine eigene Festplatte.

Kapitel 2

Hardware und Systemstruktur

Beschäftigen wir uns zuerst mit der *Hardware*, der elektronischen Maschine, auf der das Betriebssystem läuft. (Entgegen dem Anschein enthält bereits die „nackte“ Maschine ein Programm, nämlich das Mikroprogramm, auch *Firmware* genannt.) Grundlage ist auch heute meist die von Neumannsche Architektur, die 1946/47 von Burks, Goldstine und von Neumann vorgeschlagen wurde. Ihre *physikalische Struktur* ist in Abbildung 2.1 dargestellt.

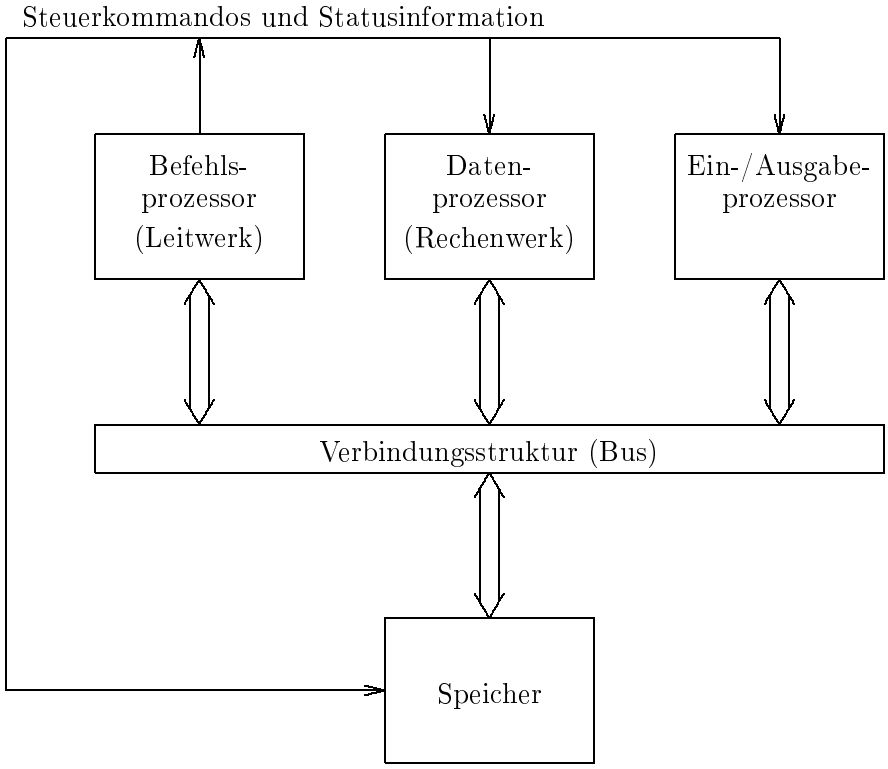


Abbildung 2.1: Architektur des klassischen Universalrechenautomaten

Natürlich kann es mehrere Busse (häufig mit unterschiedlichen Geschwindigkeiten und unterschiedlicher Breite) und mehrere Ein-/Ausgabeprozessoren (jedes Gerät wie Tastatur, Maus, Bildschirm, Festplatte, Diskettenlaufwerk, Ethernet-, serielle und parallele Schnittstelle besitzt seine eigene Steuerung) geben. Befehls- und Datenprozessor werden zur *zentralen Recheneinheit* (Central Processing Unit, CPU) zusammengefasst.

Neben der physikalischen Struktur wird eine Rechnerarchitektur durch ihr *Operationsprinzip* beschrieben. Hier besteht es darin, dass ein homogener Speicher sowohl Befehle als auch Daten enthält, die jeweils durch Bitvektoren dargestellt werden. Eine Operation wird auf den Inhalt einer durch ihre Adresse angesprochenen Speicherzelle angewandt, den sie gemäß des Zustands der Maschine interpretiert.

Da sowohl Befehle als auch Daten aus dem Speicher geholt werden, stellt der Speicher neben dem Bus (beide sind normalerweise langsamer als die CPU) einen „Flaschenhals“ dar. Linderung erhofft man sich von *Speicherverschränkung* (Storage Interleaving). Da in einem Speicherblock immer nur eine Zelle adressiert werden kann, ein Programm aber häufig fortlaufende Adressen anspricht, verteilt man aufeinander folgende Adressen auf verschiedene Speicherblöcke. Ergänzt wird dies durch die *Speicherhierarchie*. Innerhalb der zentralen Recheneinheit gibt es einen besonders schnellen Speicher (Cache), in dem Daten aus dem Hauptspeicher vor ihrer Verarbeitung zwischengelagert werden. Um den Zwischenspeicher zu füllen, werden Befehle und Daten auf Vorrat angefordert. Wenn die Vorhersage nicht zutrifft (z. B. weil das Programm einen Sprungbefehl ausführt), war die Arbeit umsonst. Die Speicherhierarchie wird nach der langsameren Seite hin durch den Hintergrundspeicher (Festplatte, Magnetband o. ä.) fortgesetzt.

Mehr zur Rechnerarchitektur und zur technischen Realisierung gibt es in der Vorlesung *Technische Informatik*.

2.1 Mikroprogrammierung und Maschinensprache

Normalerweise ist es zu aufwändig (komplizierte Schaltungen erschweren eine hohe Integrationsdichte) und zu teuer, die Maschinenbefehle direkt durch elektronische Bauteile interpretieren zu lassen. Daher schiebt man eine zusätzliche Ebene ein, die nicht außerhalb der zentralen Recheneinheit sichtbar ist. Das *Mikroprogramm* liegt in einem besonders schnellen, meist unveränderlichen Speicher und wird mit einer höheren Taktfrequenz ausgeführt.

Der Mikroprogrammspeicher enthält einen Interpretierer für die *Maschinensprache*, die Sprache, die die „nackte“ Maschine versteht. Er arbeitet in drei sich ständig wiederholenden Phasen: Holen einer Anweisung aus dem Hauptspeicher, Entschlüsselung, Ausführung. (Die Entschlüsselung kann darin bestehen, den Maschinenbefehl als Index in einen Sprungvektor zu benutzen, der zum ausführenden Mikroprogrammabschnitt führt.) Die zen-

trale Recheneinheit führt also ununterbrochen Befehle aus, selbst wenn sie nichts Nützliches leistet.

Ein einfaches Beispiel für eine Maschinensprache könnte etwa folgendermaßen aussehen. Jede Anweisung besteht aus einem 4 Bit langen *Operationscode* und einer 12 Bit langen Speicheradresse M. Alle Anweisungen wenden eine bestimmte Operation auf den Inhalt der angegebenen Speicherzelle an. Operationscodes sind:

	Symbol	Bedeutung
0001	LOAD	$M \rightarrow$ ACC
0010	STORE	ACC $\rightarrow M$
0011	ADD	ACC $+ M \rightarrow$ ACC
0100	SUBTRACT	ACC $- M \rightarrow$ ACC
0101	MULTIPLY	ACC $* M \rightarrow$ ACC
0110	DIVIDE	ACC$/M \rightarrow$ ACC
0111	JUMP	Sprung zur Speicherstelle M
1000	JUMPZERO	Sprung nach M, falls ACC leer
1001	JUMPMSB	Sprung nach M, falls das führende Bit von ACC gesetzt ist
1010	JUMPSUB	Sprung zum Unterprogramm bei M
1011	RETURN	Rückkehr vom Unterprogramm bei M

ACC ist der Akkumulator (ein spezielles Register).

Das Anweisungspaar JUMPSUB und RETURN erlaubt die typische Realisierung von (nichtrekursiven!) Prozeduraufrufen. Angenommen, das Hauptprogramm enthält an der Adresse h die Anweisung JUMPSUB p. Nach deren Ausführung enthält die Speicherzelle mit der Adresse p den Wert $h + 1$, und es steht die Anweisung mit Adresse $p + 1$ zur Ausführung an. Die Prozedur beendet sich später durch die Anweisung RETURN p. Danach steht die Anweisung mit Adresse $h + 1$ zur Ausführung an; es geht im Hauptprogramm weiter.

Die Realisierung von Prozeduraufrufen mit JUMPSUB und RETURN erlaubt keine Rekursion. Die Rücksprungadresse würde nämlich überschrieben, und alle Aufrufe bis auf den letzten gingen verloren. Tatsächlich werden Rücksprungadressen und lokale Variablen auf dem *Stapel* oder *Keller* verwaltet. Abbildung 2.2 zeigt den Ablauf bei einem Prozeduraufruf.

Am Anfang belegt das Hauptprogramm einen gewissen Speicherplatz für globale und lokale Variablen. Bei einem Prozeduraufruf bringt es die Parameter auf den Keller und lässt Platz für das Ergebnis. Die Größe des Ergebnisses (dessen Typ) ist bekannt. Die Prozedur selbst speichert zuerst die Rücksprungadresse. Den Platz findet sie wieder, da der Abstand vom Anfang fest ist. Danach nutzt sie den Stapel für ihre lokalen Variablen. Weitere Prozeduraufrufe sind ebenfalls möglich. Am Ende interessiert nur noch das Ergebnis, das das Hauptprogramm oben auf seinem Stapel findet. Die

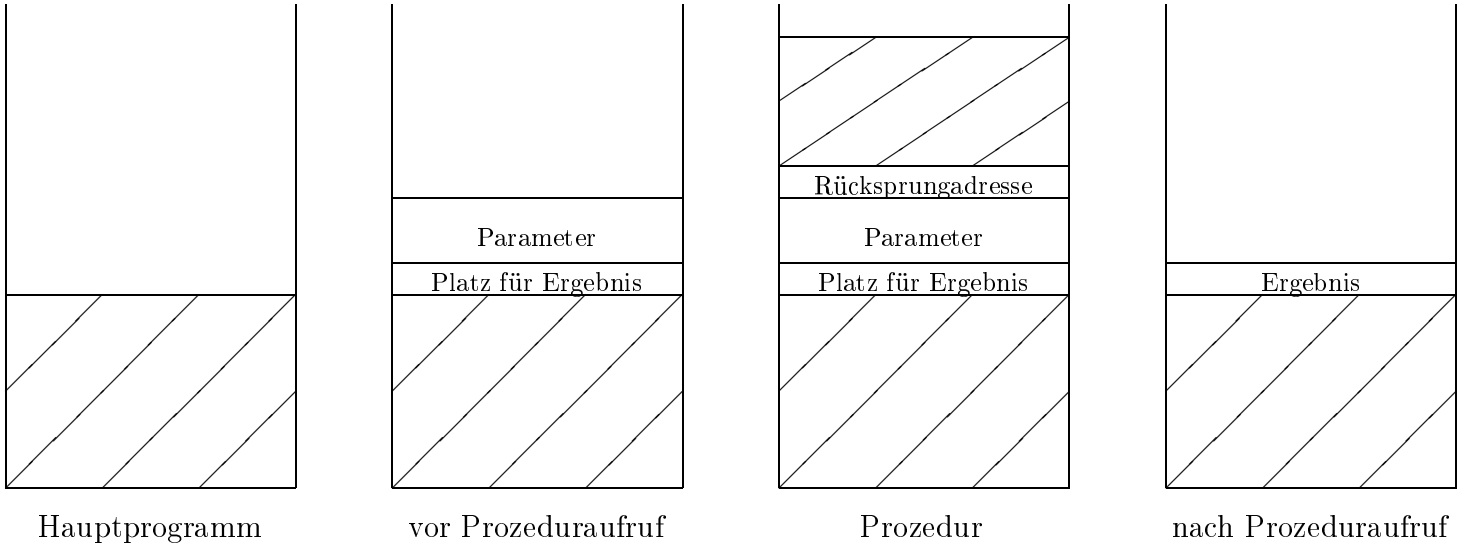


Abbildung 2.2: Stapel bei einem Prozeduraufruf

Position des obersten Kellerelements ist so wichtig, dass man dessen Adresse in einem speziellen Register speichert, dem Stapelzeiger.

Es spricht einiges dafür, die Position von Rücksprungadresse und Parametern zu vertauschen, weil die Parameter zu den lokalen Variablen gehören. Eine Erweiterung ist nötig, wenn Prozeduren geschachtelt werden dürfen. Dann müssen die Anfangsadressen der umgebenden Prozeduren ebenfalls abgelegt werden, damit deren lokale Variablen erreichbar sind. Für eine Sprache wie C reicht das angegebene Schema allerdings aus. Soweit unser kleiner Ausflug in die Vorlesung *Compilertechnik*.

Abbildung 2.3 zeigt ein Maschinenprogramm zur Berechnung von $y = 2^x$. Die erste Spalte gibt die Hauptspeicheradresse an.

Verschiedene Maschinenbefehle können unterschiedlich lange Ausführungszeiten haben. Die Taktfrequenz des Rechners muss sich nach dem langsamsten Maschinenbefehl richten. Während es einerseits wünschenswert ist, häufig gebrauchte Anweisungsfolgen zwecks höherer Ausführungsgeschwindigkeit als Maschinenbefehl zu implementieren, kann dadurch andererseits die durchschnittliche Ausführungszeit steigen. Messungen haben gezeigt, dass einfache Anweisungen wie LOAD und STORE am häufigsten auftreten. Das Prinzip *RISC* (Reduced Instruction Set Computer) lässt nur wenige und einfache Maschinenbefehle zu. Sie sollen innerhalb eines Maschinenzyklus ausführbar und fest verdrahtet (also nicht mikroprogrammiert) sein. Ein einheitliches Befehlsformat ermöglicht, die Anweisungen wie am Fließband überlappend auszuführen (Pipelining): Anweisung holen, entschlüsseln, Ope-

	Anweisung der Maschinensprache	
946	LOAD 958	
947	STORE 957	setze y auf 1
948	LOAD 956	
949	JUMPZERO 959	falls $x = 0$, Ende
950	SUBTRACT 958	
951	STORE 956	subtrahiere 1 von x
952	LOAD 957	
953	ADD 957	
954	STORE 957	verdopple y
955	JUMP 948	
956	x	
957		$= y$
958	1	
959		

Abbildung 2.3: Maschinenprogramm zur Potenzierung

randen holen, verknüpfen und speichern wären solche Phasen. Speicherzugriffe sollen durch eine hohe Anzahl von Registern (und z. B. die *Fenstertechnik* bei der Parameterübergabe an Prozeduren) verringert werden.

Die meisten Maschinensprachen kennen verschiedene *Adressierungsarten*. Bisher haben wir nur die *direkte* Adressierung kennengelernt. Andere Möglichkeiten sind: Bei der *unmittelbaren* Adressierung enthält die Anweisung den Operanden selbst, nicht seine Adresse. Bei der *indirekten* Adressierung enthält die Anweisung einen Zeiger auf den Operanden, d. h. die adressierte Speicherzelle enthält noch nicht den Operanden, sondern seine Adresse. Die *indizierte* Adressierung erfordert die Einführung eines Indexregisters. Dessen Inhalt wird zu jeder Adresse addiert, um die endgültige Speicherzelle des Operanden zu ermitteln. Indirekte und indizierte Adressierung lassen sich wiederum kombinieren.

2.2 Unterstützung des Betriebssystems

Die normale Arbeitsweise allein reicht nicht aus. Wenn z. B. mehrere Geräte zu betreuen sind, wäre es zeitaufwändig, ständig ihren Zustand abzufragen (Polling), um herauszufinden, ob Ein-/Ausgabeanforderungen vorliegen. Diese Abfragen müssten für Geräte mit unterschiedlichen Geschwindigkeiten (Tastatur, Festplatte) auch noch unterschiedlich häufig erfolgen. Die Rechenzeit kann besser verwendet werden, wenn die Geräte die zentrale Recheneinheit bei einer Anforderung *unterbrechen*. Die CPU beendet dann den gerade

angefangenen Maschinenbefehl, sichert den Inhalt einiger Register und bearbeitet die Unterbrechung. Während der Bearbeitung werden andere Unterbrechungen gesperrt, es sei denn, es ist eine Hierarchie vorgesehen, und die erneute Unterbrechung hat Vorrang. (Zum Intel 80386 gibt es einen eigenen Chip für die Unterbrechungsbehandlung, der solche Vorrangstufen einführt.) Unterbrechungen finden sehr häufig statt!

Es gibt verschiedene Arten von Unterbrechungen (allgemeiner: Ausnahmen). Von der CPU selbst ausgelöste Ausnahmen sind z. B. Division durch Null. Dazu gehören auch undefinierte Operationscodes (Traps), die vom Betriebssystem für spezifische Aufgaben genutzt werden können. Die eigentlichen Unterbrechungen werden durch einen speziellen Eingang am Prozessor ausgelöst. Durch Setzen von Registern lassen sich Unterbrechungen auch während des normalen Betriebs abschalten. Es gibt aber eigentlich immer einen NMI (Non Maskable Interrupt) als Sonderfall; der ist für Notfälle (z. B. Stromausfall, Speicherfehler) vorgesehen.

Eine Unterbrechung ruft ein (kurzes!) Maschinenprogramm auf. Dazu ist am Speicheranfang eine Sprungtabelle (der *Unterbrechungsvektor*) vorgesehen. Jeder Unterbrechung ist eine Nummer zugeordnet, die als Index in die Sprungtabelle verwendet wird, um die Adresse des zuständigen Maschinenprogramms zu finden. (Beim Intel 80386 teilt der Zusatzchip dem Prozessor die Nummer mit, wenn es sich um eine normale Unterbrechung handelt.) Die Behandlungsprozedur lädt am Ende die Register wieder mit den gesicherten Werten und lässt neue Unterbrechungen zu.

Bei der Datenübertragung von einem schnellen Gerät (Festplatte) wäre es zu aufwändig, für jedes angekommene Zeichen eine Unterbrechung auszulösen. Hier benutzt man *direkten Speicherzugriff* (Direct Memory Access, DMA). Die zentrale Recheneinheit löst den Vorgang aus, indem sie dem Gerät einen Speicherbereich zuweist. Das Gerät füllt den Speicherbereich selbstständig und unterbricht die zentrale Recheneinheit erst am Ende. Eine kleine Feinheit dabei ist, dass die zentrale Recheneinheit und das Gerät während der Übertragungszeit um den Speicherzugriff konkurrieren. Das Gerät erhält normalerweise Vorrang (das nennt man „Cycle Stealing“). Obwohl die zentrale Recheneinheit kaum zum Arbeiten kommt, spart man doch den Aufwand der Unterbrechungsbearbeitung.

Ein Betriebssystem, das auf einer Maschine wie der bisher beschriebenen laufen muss, ist arm dran. Jeder Benutzer kann auf die Hardware direkt zugreifen und so die Arbeit stören oder Sicherheitsmechanismen umgehen. Deshalb gibt es verschiedene *Betriebsarten* (vier beim Intel 80386), in denen unterschiedliche Befehle erlaubt und unterschiedliche Speicherbereiche zugänglich sind. Manche sind nur im *privilegierten* oder *Supervisor*-Modus zulässig. Der Zugriff auf Geräte ist z. B. nicht allgemein erlaubt. Die zentrale Recheneinheit geht vor allem bei der Ausnahmebehandlung in den privilegierten Modus über. Daher lösen *Systemaufrufe* häufig solche Ausnahmen aus. Der Vorteil ist, dass das Betriebssystem zuerst die Zulässigkeit der An-

forderung (und ggf. ihrer Parameter) prüfen kann, bevor es eine Aktion ausführt.

2.3 Struktur eines Betriebssystems

Betriebssysteme sind große Programme, die irgendwie strukturiert werden müssen. Im Laufe der Entwicklung hat man daher, entsprechend den jeweils gängigen Trends im Software-Engineering, verschiedene Ansätze ausprobiert.

Eine nahe liegende Gliederung setzt den Aufbau der Hardware (Mikroprogrammierung, Maschinensprache) fort: Das Betriebssystem ist in Schichten/Ringen angeordnet, wobei die oberen/äußeren, dem Benutzer zugewandten Schichten/Ringe auf den unteren/inneren, hardwarenahen aufbauen. Bei jedem Übergang nach unten/innen werden die Berechtigungen sorgfältig geprüft, da notwendigerweise die Privilegien zunehmen. Jede Schicht implementiert eine *virtuelle Maschine*, die nach oben hin immer abstrakter wird.

Ziemlich exotisch mutet das Betriebssystem VM an, das auf IBM-Großrechnern läuft. Es enthält nichts weiter als die Unterstützung des Multiprocessings, indem es für jedes Anwenderprogramm eine „nackte“ Maschine simuliert. Was man normalerweise unter einem Betriebssystem versteht – den Rechner leichter benutzbar zu machen –, wird erst dazu geladen; es können also mehrere verschiedene „Betriebssysteme“ gleichzeitig laufen, die jeweils nur einen Benutzer haben. Auf den IBM-Großrechnern kann das z. B. CMS sein.

Diese Lösung ist nicht ganz einfach. Neben dem Hauptspeicher (wo es üblich ist, wie wir noch sehen werden) muss auch der Plattenplatz virtuell verwaltet werden: Der vorhandene Platz wird auf „Minidisks“ aufgeteilt. Andere Geräte werden sowieso gemeinsam benutzt. Trickreich ist auch die Simulation des Supervisor-Modus: Während VM selbst natürlich privilegiert ist, sind es die darüber liegenden „Betriebssysteme“ wie CMS nicht. CMS muss aber, wie es sich für ein Betriebssystem gehört, in einem virtuellen Supervisor-Modus laufen. Tatsächlich löst jede privilegierte Anweisung auf der Hardware eine Ausnahme aus. Diese wird von VM bearbeitet, so dass die gewünschte Aktion doch noch ausgeführt wird.

Bei der Schichtstruktur liegen die Gerätesteuerungen meist ganz unten, worauf Prozess- und Hauptspeicherverwaltung aufbauen. Es kann aber praktisch sein, eine Gerätesteuerung hinzuzufügen oder auszutauschen (weil man am Rechner herumgebastelt hat), ohne das ganze Betriebssystem neu generieren zu müssen. Dann ist es besser, das Betriebssystem in die Bearbeiter für verschiedene eng umrissene Aufgaben zu gliedern. Diese Bearbeiter werden aus dem Kern entfernt und laufen teilweise als gewöhnliche Benutzerprozesse. Der höher privilegierte Kern enthält einerseits einen Mechanismus zur Vergabe der Aufträge, ermöglicht also die *Interprozesskommunikation* und Prüfung der Zuständigkeit. Andererseits können manche Teile nicht ausge-

lagert werden, die nur zur Unterstützung der Bearbeiter dienen. Immerhin erreicht man aber eine Trennung zwischen Mechanismus und Taktik. Ein Paradebeispiel für diesen Ansatz ist das Betriebssystem Mach – das sich doch nicht durchgesetzt hat, obwohl es als Grundlage von OSF/1 (des gemeinsamen Betriebssystems der Open Software Foundation) dienen sollte..

Der Aufbau entspricht der modernen objektorientierten Sicht. Neben der leichteren Wartbarkeit bringt diese Struktur auch eine höhere Robustheit gegenüber Ausfällen mit sich. In Abbildung 2.4 ist der Kern nur für die Kommunikation zwischen Auftraggeber und Auftragnehmer zuständig. Der Kunde richtet einen Befehl an den zuständigen Bearbeiter und wartet auf die Antwort.

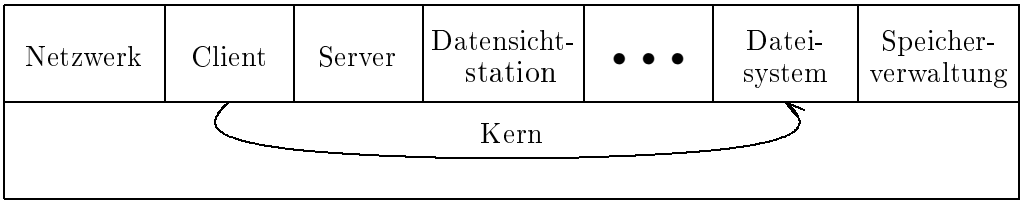


Abbildung 2.4: Auftraggeber-Auftragnehmer-Modell

Beide vorgestellten Gliederungen sehen die Verwaltung von Prozessen als zentral an, oft verbunden mit der Hauptspeicherverwaltung. Von der Hauptspeicherverwaltung führt ein natürlicher Übergang zur Festplattenverwaltung und weiter zum Dateisystem. In dieser Reihenfolge werden die Bestandteile besprochen. Gerätesteuerungen werden nur kurz gestreift.

Moderne Betriebssysteme entsprechen allerdings weder der einen noch der anderen Gliederung. Üblicherweise bestehen sie wie gehabt aus einem monolithischen Kern, der Prozesse, Hauptspeicher und das Dateisystem verwaltet. Die Gerätesteuerungen sind dagegen ausgelagert. Man kann die Treiber installieren, wenn man neue Hardware einbaut (Windows), oder die Treiber werden sogar dynamisch beim Zugriff auf ein Gerät geladen (Linux).

Kapitel 3

Prozesse

Wie gesagt, ist ein (sequentieller) *Prozess* oder *Auftrag* die Ausführung eines Programms, d. h. ein *Vorgang*, der Rechenzeit, Speicherplatz usw. erfordert. Mehrere Prozesse können durchaus das gleiche Programm ausführen. Die „gleichzeitige“ Bearbeitung mehrerer Prozesse ist nicht nur nötig, wenn sich mehrere Benutzer einen Rechner teilen können. Sie kann auch zu einer besseren Auslastung der Ressourcen führen. Das Prozesskonzept ist grundlegend für ein Betriebssystem.

3.1 Einführung in das Prozesskonzept

Im System laufen mehrere Prozesse nicht wirklich gleichzeitig ab. Die Illusion wird durch das *Zeitscheibenverfahren* hervorgerufen. Dabei werden den Prozessen abwechselnd kleine Zeitintervalle zugeteilt, in denen ihnen die zentrale Recheneinheit gehört und sie Fortschritte machen dürfen. Die Abbildung 3.1 skizziert, wann welcher Prozess bearbeitet wird. [1]

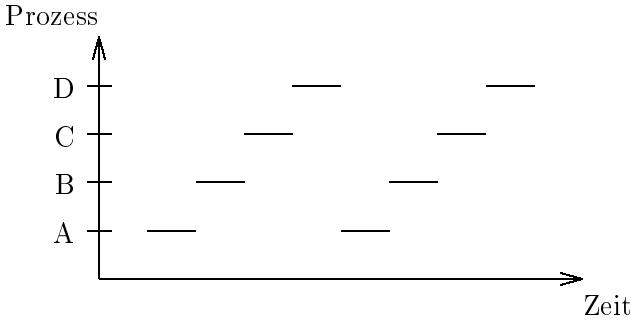


Abbildung 3.1: Rechenzeitzuteilung

[1] Die eingezeichneten Intervalle gehen davon aus, dass jeder Prozess sein Quantum immer voll ausschöpft. Häufig werden jedoch z. B. Ein- oder Ausgaben angefordert, wodurch die Ablösung vorzeitig herbeigeführt wird.

Die Prozesse merken nichts vom Wechsel; am Programmtext ist auch nicht zu ersehen, wo die „Pausen“ eingeschoben werden. Es ist ebenfalls nicht vorhersehbar, wie lange ein Programm insgesamt braucht (*Verweilzeit* im System). Einen Prozess zwischendurch immer wieder anzuhalten und fortzusetzen, ohne dessen Arbeit zu stören, erfordert natürlich Verwaltungsaufwand.

Prozesse können verschiedene Zustände einnehmen, siehe Abbildung 3.2. Ein Prozess *blockiert* z. B., wenn er eine Eingabe anfordert. Ein blockierter Prozess könnte selbst dann keine Fortschritte machen, wenn man ihm Rechenzeit gäbe. Das Blockieren ist der einzige Übergang, der durch die Initiative des aktiven Prozesses geschieht. Ein Prozess wird z. B. bei der erfolgten Eingabe *aufgeweckt*. Er wird dadurch *bereit*, d. h. wartet auf die Zuteilung von Rechenzeit. Der laufende Prozess, der seine Zeitscheibe ausgeschöpft hat, wird *verdrängt* von einem bereiten Prozess, der nach gewissen Kriterien auszuwählen ist. Diesen Austausch des laufenden Prozesses nimmt der *Dispatcher* vor; die zugrundeliegende Auswahl trifft der *Scheduler*.

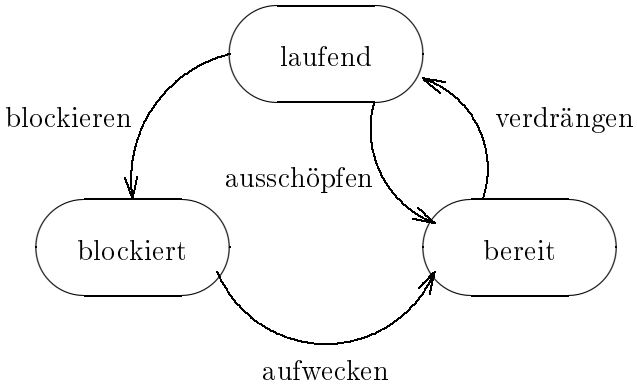


Abbildung 3.2: Prozesszustände

Es kann sinnvoll sein, einen Prozess durch äußere Einwirkung anzuhalten bzw. später wieder fortzusetzen. Dann kommen zwei weitere Zustände hinzu (siehe Abbildung 3.3).

3.2 Implementierung von Prozessen

Die Information über einen Prozess steht in seinem *Prozesskontrollblock* (PCB). Dieser beschreibt ihn hinreichend genau, um ihn nach mehreren Prozesswechseln ohne Schaden wieder aufsetzen zu können. Gespeichert werden unter vielen anderen folgende Daten:

- der Prozesszustand

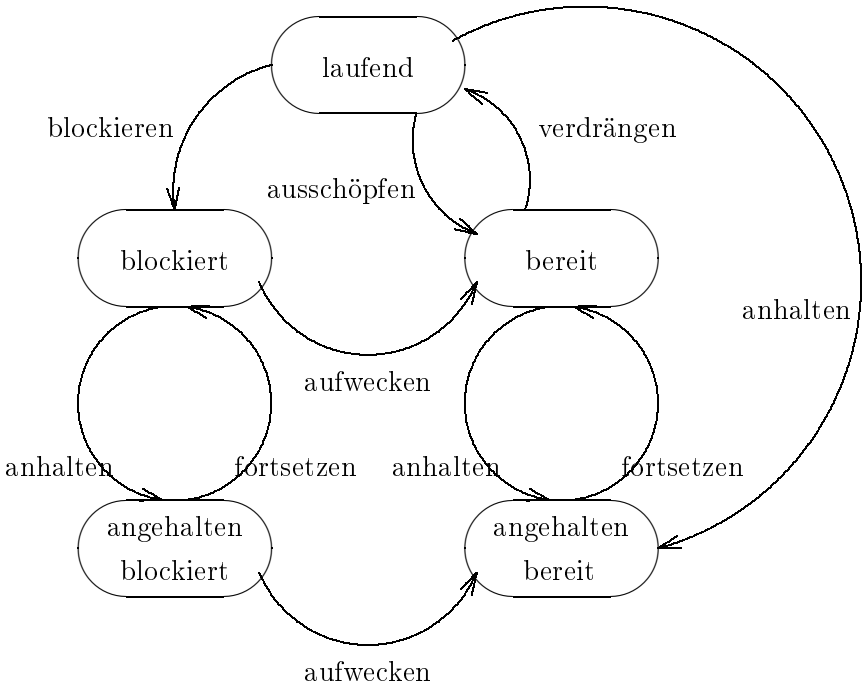


Abbildung 3.3: Weitere Prozesszustände

- der letzte Wert des Befehlszählers und des Stapelzeigers, sowie anderer Register (Akkumulator, Indexregister)
- Informationen für den Scheduler, z. B. eine Priorität
- die Speicherzuordnung, um die Speichersegmente wiederzufinden
- Verwaltungsinformation, z. B. Prozessnummer und verbrauchte Rechenzeit
- der Status aller geöffneten Dateien

Die Prozesskontrollblöcke werden üblicherweise als Tabelle (fester Größe) oder als Liste verwaltet. Eine Liste lässt sich in verschiedene Warteschlangen aufteilen: Eine für die bereiten Prozesse und je eine für jedes Gerät.

Der Prozesswechsel geschieht durch eine Unterbrechung. Beim Ausschöpfen und Verdrängen wird sie vom Zeitgeber ausgelöst, beim Aufwecken durch das entsprechende Ein-/Ausgabegerät. Nicht jede solche Unterbrechung muss tatsächlich zu einem Prozesswechsel führen. Der Zeitgeber unterbricht häufiger, um Buchführungsaktionen (z. B. Systemzeit fortschreiben) auszuführen.

Bei einem Prozesswechsel (auch *Kontextumschaltung* genannt) wird der Prozesskontrollblock des laufenden Prozesses auf den aktuellen Stand gebracht und danach der Zustand des ausgewählten Prozesses so wieder her-

gestellt, wie er bei dessen Verdrängung bestand und in dessen Prozesskontrollblock festgehalten wurde.

Im Folgenden werden die wichtigsten Systemaufrufe zur Prozessverwaltung unter UNIX beschrieben. An ihnen lassen sich einige Entwurfsentscheidungen ablesen. Jeder Prozess kann neue Kindprozesse erzeugen, mit denen er parallel läuft. Das Warten auf die Beendigung eines Kindprozesses muss explizit herbeigeführt werden.

fork erzeugt eine identische Kopie des laufenden Prozesses. Alle Variablenwerte stimmen überein, ja sogar die gleichen Dateien werden verwendet. Nur einen Unterschied gibt es, der beide Exemplare davor bewahrt, für immer synchron zu laufen: den Rückgabewert des Funktionsaufrufs. Im Elternprozess liefert *fork* die Kennung des gerade erzeugten Kindprozesses, im Kindprozess dagegen 0. Alle neuen Prozesse müssen durch *fork* erzeugt werden. Daraus folgt, dass es einen ersten Prozess geben muss, aus dem sich alle in einem Stammbaum (ähnlich der Dateihierarchie) ableiten. Dieser Prozess (*init*) wird beim *Booten* gestartet. Aus ihm entstehen zunächst die *login*-Prozesse (für jedes Terminal einer), die beim Anmelden wiederum den Kommandointerpreter starten. Der Kommandointerpreter kann seinerseits mehrere Prozesse gleichzeitig starten (Hintergrundprozesse).

exec (in verschiedenen Variationen) erlaubt einem Prozess, ein neues Programm auszuführen. Der Aufruf von *exec* ist somit die letzte Anweisung des alten Programms. Meist folgt *exec* direkt auf ein *fork*. Ein Prozess beendet sich mit *_exit*. *wait* wartet auf die Beendigung eines Kindprozesses. Der Funktionswert ist die Kennung des Kindes. Sie kann mit dem Wert von *fork* verglichen werden. Gezieltes Warten auf einen Prozess ist mit *waitpid* möglich. Die Systemaufrufe *getpid* und *getppid* liefern die Kennung des laufenden bzw. des Elternprozesses.

Da beim Ende des Kindprozesses ein wenig Information zum Elternprozess übertragen werden muss, wird der Kindprozess noch so lange aufbewahrt, bis der Elternprozess wait aufgerufen hat. Der Kindprozess wird zum „Zombie“; es gibt einen zusätzlichen Prozesszustand „beendet“. Abbildung 3.4 zeigt den Ablauf bei der Ausführung eines Kommandos.

Das Gerüst eines Kommandointerpreters in C99 zeigt Abbildung 3.5. Dabei werden viele der gerade vorgestellten Systemaufrufe verwendet. (Die meisten UNIX-Systemaufrufe werden in <unistd.h> bekannt gemacht. Durch die Benutzung von Typen wie pid_t aus <sys/types.h> oder Makros wie WIFEXITED und WEXITSTATUS aus <sys/wait.h> wird das Programm klarer und portabler. Typisch ist die Fehlerbehandlung nach *jedem* Systemaufruf.)

Ein ausgewachsener Kommandointerpreter wie `csh` (C-Shell) oder `bash` (Bourne Again Shell) bietet noch viel mehr Möglichkeiten, z. B.:

- Sogenannte Hintergrundprozesse, auf deren Ende nicht gewartet wird. (Hinter dem Kommando steht ein &.)

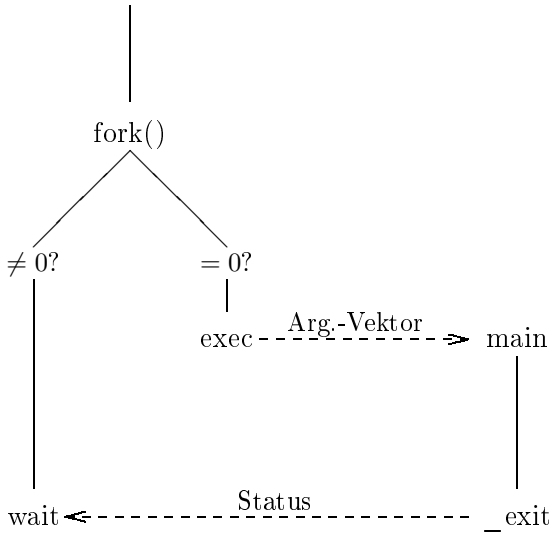


Abbildung 3.4: Ausführung eines Kommandos

- Job Control: Prozesse anhalten und fortsetzen (`fg` bzw. `bg`)
- Abfrage des Status und Verknüpfung von Kommandos mit && und ||.
- Ausdrücke und Kontrollstrukturen.
- Variablen und Verarbeitung der Ausgaben von Kommandos.

3.3 Scheduling

Wenn mehrere bereite Prozesse warten, muss eine Auswahl getroffen werden. Die Frage ist, wie. Es gibt widersprüchliche Anforderungen:

- *Fairness* (kein bereiter Prozess darf ewig warten müssen)
- hohe *Effizienz* (Anteil der nützlichen Zeit an der gesamten CPU-Zeit)
- geringe *Antwortzeit* (Zeit zwischen Ende der Eingabe und Beginn der Ausgabe) für interaktive Benutzer
- geringe *Verweilzeit* (Zeit zwischen Beginn der Eingabe und Ende der Ausgabe) bzw. geringe *Wartezeit* (Zeit im bereiten Zustand) für Stapelaufträge (Stapel = Batch)
- hoher *Durchsatz* (Anzahl der erledigten Aufträge pro Zeiteinheit)

Besonders interaktive Benutzer und Stapelaufträge stellen konträre Anforderungen.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

void execute(const char *command, char *const parameters[])
{
    pid_t pid;
    switch (pid = fork()) {
    case -1: // error
        perror("cannot fork");
        break;
    case 0: // child
        execvp(command, parameters);
        perror("cannot exec");
        exit(EXIT_FAILURE);
        break;
    default: // parent
        int stat_loc;
        if (waitpid(pid, &stat_loc, 0) == -1)
            perror("cannot wait");
        else if (WIFEXITED(stat_loc))
            printf("status: %d\n", WEXITSTATUS(stat_loc));
    }
}
```

Abbildung 3.5: Gerüst eines Kommandointerpreters

3.3.1 Wer zuerst kommt, mahlt zuerst (First-Come First-Served)

Die Prozesse werden in der Reihenfolge ihres Eintreffens bearbeitet und nicht verdrängt. Natürlich werden sie abgelöst, wenn sie blockieren. Von der Inbesitznahme der CPU einmal abgesehen, kann auch die Auslastung des Systems schwach sein, obwohl der Aufwand für die Prozessumschaltung entfällt. Es brauchen nur rechenintensive Prozesse vor ein-/ausgabeintensiven drankommen.

3.3.2 Reihum-Scheduling (Round-Robin)

Einer der einfachsten und verbreitetsten Algorithmen. Jeder Prozess erhält reihum seinen Anteil an der Rechenzeit, und zwar ein festgelegtes *Quantum*

(Zeitintervall). Wenn ein Prozess sein Quantum ausgeschöpft hat oder vorher blockiert oder terminiert, kommt der nächste an die Reihe.

Es genügt, eine zyklisch verkettete Liste der bereiten Prozesse und einen Zeiger auf den laufenden Prozess zu führen. Wenn der Scheduler (der durch eine Unterbrechung vom Zeitgeber aktiviert wurde) entscheidet, einen Prozesswechsel durchzuführen, genügt es, den Zeiger auf den Nachfolger in der Liste zu rücken. Außerdem fällt natürlich der Aufwand für die eigentliche Prozessumschaltung (siehe oben) an.

Je kleiner das Quantum, desto geringer die Antwortzeit, aber umso geringer auch die Effizienz (wegen des Verwaltungsaufwands für häufige Prozesswechsel).

3.3.3 Shortest Job First

Bei Stapelaufträgen lässt sich möglicherweise die Laufzeit schon vorher abschätzen. Weiterhin sollen alle Aufträge gleich wichtig sein.

Die Shortest-Job-First-Strategie bearbeitet den Auftrag mit der jeweils kürzesten Laufzeit bis zum Ende oder, in der verdrängenden (preemptive) Variante, bis ein Auftrag eintrifft, dessen Laufzeit geringer als die verbleibende Restlaufzeit ist. Wenn alle Aufträge schon zu Anfang vorliegen, dann führt diese Strategie nachweislich zur kürzesten durchschnittlichen Verweilzeit. Die verdrängende Variante minimiert die mittlere Wartezeit.

Im Dialogbetrieb kann man die Ausführung eines Kommandos jeweils als einen Auftrag ansehen. Dann minimiert die Shortest-Job-First-Strategie auch die mittlere Antwortzeit! Allerdings muss die Laufzeit eines Kommandos geschätzt werden. Hierzu wird meist das Verfahren der *Alterung* angewandt: Der Schätzwert S für die Laufzeit des nächsten Kommandos ergibt sich als gewichteter Mittelwert aus dem Schätzwert S und der tatsächlichen Laufzeit T des vorhergegangenen Kommandos.

$$S_{i+1} = aS_i + (1-a)T_i \quad (i \geq 1); \qquad S_1 = T_0$$

Am einfachsten setzt man $a := 1/2$.

3.3.4 Scheduling nach Prioritäten

Prozesse können unterschiedlich wichtig sein; wichtigere Prozesse sollten mehr Zeit zugeteilt bekommen. Das Reihum-Verfahren kann leicht angepasst werden, indem man zu jeder Prioritätsklasse eine eigene verkettete Liste führt. Innerhalb einer Prioritätsklasse erfolgt die Zeitzuteilung reihum; niedrigere Prioritätsklassen kommen jedoch erst zum Zug, wenn alle höheren leer sind.

Prioritäten können statisch oder dynamisch vergeben werden. Bei statischen Prioritäten besteht offensichtlich die Gefahr des „Verhungerns“, d. h. manche Prozesse (in den untersten Prioritätsklassen) kommen nie an die

Reihe. Bei bestimmten Anwendungen, z. B. einer Datenbank oder einem Betriebssystem, können sie aber angebracht sein, weil sichergestellt ist, dass die privilegierten Schichten ihre Vorrechte nicht ausnutzen. Bei einem Betriebssystem, das in viele Prozesse zerlegt ist, würde der Kern die höchste Priorität bekommen, gefolgt von den Steuerungen für die einzelnen Geräte.

Dynamische Prioritäten werden zur Laufzeit neu berechnet. Beispiel: Ein-/ausgabeintensive Prozesse sind die meiste Zeit blockiert und belegen Speicher. Da sie ihr Quantum selten ausschöpfen, ist es gerecht (und senkt die Verweilzeit), sie häufiger auszuwählen. Eine geeignete Formel ist $P := 1/f$, wobei f der vom letzten Quantum benutzte Teil ist. Je höher der Wert von P, desto höher die Priorität. Allerdings müssen hier niedrige Prioritäten ab und zu neu berechnet werden, da sonst Prozesse „verhungern“ können.

Auch die Länge eines Auftrags kann als statische oder dynamische Priorität angesehen werden. Wie bei Shortest Job First gibt es bei Prioritäten allgemein eine verdrängende (preemptive) und eine nichtverdrängende Variante.

Weiteres Beispiel: Unter allen angemeldeten Benutzern wird die Rechenzeit gleichmäßig aufgeteilt. Notiere

- Anzahl der Benutzer n
- vom betreffenden Benutzer seit der Anmeldung für alle Prozesse insgesamt verbrauchte Zeit R
- seit der Anmeldung des betreffenden Benutzers vergangene Zeit T

Der Benutzer mit der höchsten Priorität $\frac{T}{nR}$ erhält Rechenzeit, bis er von einem anderen überholt wird.

Ähnlich: Bei einem Realzeitsystem lassen sich drohende Zeitüberschreitungen erkennen. Der dringendste Fall wird bearbeitet.

3.3.5 Mehrfache Warteschlangen (Multilevel Feedback Queues)

Wie wir gesehen haben, verdienen interaktive Prozesse eine hohe Priorität und brauchen nur ein kleines Quantum; bei Stapelaufträgen ist es genau umgekehrt. Daher kombiniert man sinnvollerweise Priorität und Quantum, besonders wenn Prozesswechsel sehr aufwändig sind und man die Prozesse richtig einschätzen muss.

Man trifft z. B. folgende Zuordnung:

Priorität	Quantum
0	1
1	2
2	4
...	...
i	2^i

Die kleinste Zahl entspricht der höchsten Priorität. Ein Prozess beginnt ganz oben und sinkt jedesmal eine Klasse tiefer, wenn er sein Quantum vollständig ausschöpft.

Nicht vorgesehen ist, dass sich das Verhalten eines Prozesses von rechenintensiv nach interaktiv ändern kann. Man sollte die Priorität nicht gerade vom Drücken einer Taste abhängig machen.

Eine Variante: Es gibt spezielle Warteschlangen für

1. Datensichtstation
2. Platte
3. kurzes Quantum
4. langes Quantum

Wenn ein blockierter Prozess aufgeweckt wird, wandert er nach (1) oder (2). Wenn er sein Quantum ausschöpft, nach (3). Und wenn er das Quantum immer noch mehrmals hintereinander ausschöpft, nach (4).

3.3.6 Zweistufiges Scheduling

Ein blockierter Prozess belegt Speicher, ohne zu arbeiten. Man kann das Speicherabbild ebenso gut auf Platte auslagern (siehe Speicherverwaltung). Die Zeit für einen Prozesswechsel, der eine Einlagerung erfordert, ist relativ hoch.

Daher läuft zunächst nur ein lokaler Scheduler für Prozesse im Hauptspeicher. Dessen Auswahlkriterien sind wie gehabt. Von Zeit zu Zeit wird ein globaler Scheduler aktiviert, der Prozesse zwischen Platte und Hauptspeicher austauscht. Zu dessen Kriterien gehören:

- Zeit der Ein-/Auslagerung
- CPU-Zeit-Zuteilung
- Größe des Prozesses
- Priorität

3.4 Verklemmungen

Wenn mehrere Prozesse quasi-gleichzeitig ausgeführt werden, müssen sie sich zwangsläufig die vorhandenen Ressourcen teilen. Dabei kann es geschehen, dass sie miteinander konkurrieren und sich gegenseitig behindern. Es besteht sogar die Gefahr, dass sie sich in eine Situation manövrierten, in der es nicht mehr weitergeht (*Verklemmung*, *Deadlock*).

3.4.1 Betriebsmittel

Ein *Betriebsmittel* (eine *Ressource*) kann zu jedem Zeitpunkt nur von einem Prozess benutzt werden. Ein Betriebsmittel kann z. B. ein Gerät oder eine Information sein. Auch Prozesskontrollblöcke können zu den Betriebsmitteln zählen, wenn sie in einer Tabelle fester Größe verwaltet werden.

3.4.2 Charakterisierung von Verklemmungen

Eine Menge von Prozessen hat sich verklemmt, wenn jedes Element auf ein Ereignis wartet, das nur von einem anderen Element ausgelöst werden kann. Die Ereignisse sind meistens Freigaben von Betriebsmitteln.

Für eine Verklemmung sind vier Bedingungen notwendig:

- Bedingung des gegenseitigen Ausschlusses. Jedes Betriebsmittel ist entweder genau einem Prozess zugeordnet oder verfügbar.

- Wartebedingung. Prozesse belegen bereits zugewiesene Betriebsmittel, während sie zusätzliche Betriebsmittel anfordern.

- Bedingung der Nichtentziehbarkeit. Zuvor zugeordnete Betriebsmittel können einem Prozess nicht zwangsweise entzogen werden. Sie müssen explizit durch den Prozess freigegeben werden, der sie besitzt.

- Bedingung der geschlossenen Kette. Es existiert eine geschlossene Kette von zwei oder mehr Prozessen, in der jeder auf ein Betriebsmittel wartet, das vom nächsten Prozess in der Kette benutzt wird.

Zur Darstellung dienen gerichtete Graphen (*Betriebsmittelzuteilungsgraphen*): Prozesse (Kreise) und Betriebsmittel (Quadrate) sind Knoten. Eine Kante von einem Betriebsmittel zu einem Prozess bedeutet, dass der Prozess das Betriebsmittel momentan benutzt. Eine Kante von einem Prozess zu einem Betriebsmittel bedeutet, dass der Prozess blockiert ist und auf die Zuteilung des Betriebsmittels wartet. Ein Zyklus im Graphen zeigt eine Verklemmung an.

Vier Strategien zur Behandlung von Verklemmungen werden im folgenden behandelt.

3.4.3 Ignorieren des Problems

Der Aufwand zur Vermeidung von Verklemmungen erscheint z. B. in UNIX zu hoch. Das Problem tritt selten genug auf, so dass es, wie auch gelegentliche Systemabstürze, in Kauf genommen werden kann.

3.4.4 Entdecken und Beheben

Der Betriebsmittelzuteilungsgraph wird bei jeder Zuteilung auf den aktuellen Stand gebracht und auf Zyklen untersucht. Bei einer Verklemmung werden so lange Prozesse eliminiert, bis kein Zyklus mehr besteht. In Stapelsystemen kann ein Prozess einfach noch einmal gestartet werden, wenn alle Dateien in ihren ursprünglichen Zustand zurückversetzt wurden. Noch gröber ist ein Verfahren, das Blockierungen über eine gewisse Dauer als Verklemmungen interpretiert.

3.4.5 Verhindern

Wenn eine der vier oben angeführten notwendigen Bedingungen nicht erfüllt ist, kann keine Verklemmung eintreten.

Die Bedingung des gegenseitigen Ausschlusses kann durch Spooling beseitigt werden. Dabei werden alle Anforderungen auf der Platte zwischengespeichert. Ein Dämonprozess besitzt das Betriebsmittel und bearbeitet die Anforderungen. *Aber:* Nicht alle Betriebsmittel sind für Spooling geeignet, und der Plattenplatz ist auch ein Betriebsmittel.

Die Wartebedingung kann dadurch beseitigt werden, dass alle Prozesse ihre Anforderungen vor dem Start bekannt geben müssen. *Aber:* Nicht alle Prozesse kennen ihre Anforderungen. Die Betriebsmittel werden schlecht ausgelastet. Andere Möglichkeit: Vor jeder neuen Anforderung müssen alle benutzten Betriebsmittel frei gegeben werden. *Aber:* Die Freigabe ist meist nicht möglich. Anspruchsvolle Prozesse können verhungern.

Die gewaltsame Entziehung von Betriebsmitteln kann in Ausnahmefällen möglich sein, wenn der Operator manuell eingreift. Wäre es leicht, hätte man kein Problem.

Da man Prozesse nicht dazu zwingen kann, zu jeder Zeit immer nur ein Betriebsmittel zu benutzen, können Zyklen durch eine globale Nummerierung der Betriebsmittel vermieden werden. Ein Prozess darf ein Betriebsmittel nur dann anfordern, wenn dessen Nummer größer ist als alle benutzten. *Aber:* Es gibt sehr viele Betriebsmittel und keine allgemein gültige Anordnung.

3.4.6 Vermeiden

Hier werden die Betriebsmittelanforderungen zur Laufzeit geprüft und Verklemmungen durch sorgfältige Zuweisungen vermieden.

Prozesspfad

Betrachte zwei Prozesse mit jeweils mehreren *kritischen Abschnitten*. (Das sind Abschnitte, in denen sie sich nicht gleichzeitig befinden können, weil sie das gleiche Betriebsmittel benutzen.) Trage den Verlauf des ersten Prozesses waagrecht und den des zweiten senkrecht in ein Diagramm ein. Dann liefert

der gemeinsame Ablauf beider Prozesse eine Lebenslinie, die in waagrechten und senkrechten Abschnitten von links unten nach rechts oben verläuft. Die Bereiche, in denen sich beide Prozesse in entsprechenden kritischen Abschnitten befinden würden, sind verboten. Meist gibt es einen weiteren Bereich, nach dessen Betreten man dem verbotenen Bereich nicht mehr ausweichen kann, siehe Abbildung 3.6. Insgesamt muss man die Lebenslinie um den verbotenen Bereich herumführen.

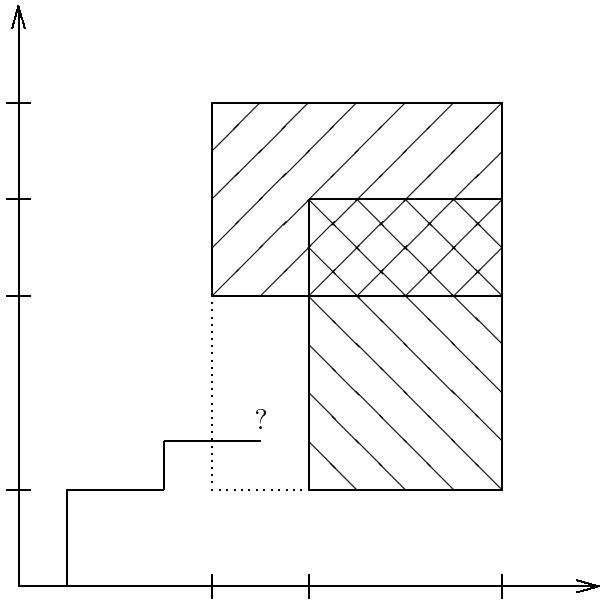


Abbildung 3.6: Ablaufpfade zweier Prozesse

Bankier-Algorithmus

Jeder Kunde (Prozess) hat einen Kreditrahmen (Maximalforderung), innerhalb dessen er sich bewegt. Bei jeder neuen Anforderung prüft der Bankier, ob es noch einen Weg gibt, alle Anforderungen zu erfüllen, d. h. einem Kunden das Ausschöpfen seines Kreditrahmens zu erlauben, bei dessen Rückzahlung den nächsten Kunden zufriedenzustellen usw. Wenn so ein Weg existiert, heißt der Zustand *sicher* und die Anforderung wird gewährt.

Der momentane Kredit und der Kreditrahmen werden für jedes Betriebsmittel geführt. Es ergeben sich zwei Matrizen mit den Kunden als Zeilen und den Betriebsmitteln als Spalten. Die Differenz aus jeweiliger Gesamtsumme und Spaltensumme ist das freie Kapital. Mit diesem Kapital muss ein Kunde seinen Kreditrahmen ausschöpfen können; wenn er seinen Kredit zurückzahlt, muss es für den nächsten reichen usw. *Aber:* Die Maximalforderungen sind selten bekannt. Die Anzahl der Prozesse schwankt. Sogar die Anzahl der Betriebsmittel kann schwanken.

Kapitel 4

Hauptspeicherverwaltung

Prozesse, die von der CPU bearbeitet werden, lagern im *Hauptspeicher*. Dieser ist meist wesentlich schneller als der *Hintergrundspeicher* (Festplatte, Magnetband o. ä.), daher relativ teuer und knapp. Bei der Multiprogrammierung müssen sich mehrere Prozesse den vorhandenen Platz teilen.

4.1 Speicherverwaltung ohne dynamische Ein- und Auslagerung

Die einfachste Lösung: Prozesse müssen mit dem vorhandenen Hauptspeicher auskommen. Beispiele waren MSDOS und die ersten Versionen von UNIX. Der Betriebssystemkern läuft ständig als unabhängiger Prozess, damit er nicht zu jedem gestarteten Programm hinzugebunden werden muss. Das dient außerdem der Modularisierung und dem Schutz vor unbeabsichtigten Übergriffen. Die anderen Prozesse (in MSDOS nur einer) teilen sich den restlichen Hauptspeicher (Abbildung 4.1). Neuerdings werden auch Bibliotheken (*Shared Libraries*) im Hauptspeicher gehalten und dynamisch hinzugebunden.

Folgende Gründe sprechen dafür, mehrere Prozesse gleichzeitig im Hauptspeicher zu halten:

- Prozesse sind häufig blockiert; dann sollte ein anderer Prozess laufen, um die Ressourcen besser zu nutzen.
- Mehrere Benutzer teilen sich einen Rechner, oder ein Benutzer möchte weitere Aufgaben erledigen, während der Rechner noch mit der Ausführung eines Kommandos beschäftigt ist.

Dabei ist es unwirtschaftlich, den Rechner ausschließlich dem laufenden Prozess zur Verfügung zu stellen, da jeder Wechsel ein Sichern des laufenden Prozesses und Laden des nächsten erfordert.

Ein Ausbau des Hauptspeichers kann somit eine spürbare Leistungssteigerung bewirken.

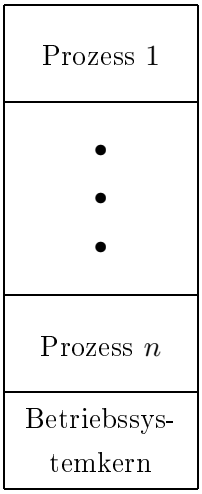


Abbildung 4.1: Speicheraufteilung

4.1 Beispiel. Ein Rechner hat 1 MB Hauptspeicher.[1] Das Betriebssystem und jeder Prozess benötigen 200 KB. Die Wahrscheinlichkeit p dafür, dass ein Prozess blockiert ist, sei 80% (durchaus realistisch beim Warten auf Benutzereingaben). Die CPU-Auslastung (Wahrscheinlichkeit dafür, dass nicht alle Prozesse gleichzeitig blockiert sind) beträgt $1 - p^4 \approx 59\%$. Bei 2 MB (mit 9 Prozessen) beträgt sie schon 87%. Mit einem zusätzlichen Megabyte Hauptspeicher scheint der Rechner also um 28% mehr zu leisten.

Die einfachste Möglichkeit, den Speicher aufzuteilen, besteht darin, feste Partitionen einzurichten. Bessere Verfahren werden später besprochen.

Zunächst taucht das Problem auf, dass Prozesse die ihnen zugeteilte Partition nicht vollständig ausfüllen. Das ist *interner Verschnitt*.

Weiterhin ist nicht klar, wie die Partitionen den Prozessen zugeteilt werden sollen. Gibt es für jede Partition eine Warteschlange von Prozessen, für die sie „gerade richtig“ wäre, besteht die Gefahr, dass große Partitionen leerstehen, weil sie nicht von kleinen Prozessen genutzt werden dürfen. Gibt es nur eine Warteschlange von Prozessen für alle Partitionen und wird beim Freiwerden einer Partition der nächste Prozess ausgewählt, für den sie ausreichend groß ist, steigt die Verschwendung, da Prozesse häufig in zu großen Partitionen laufen. Wird beim Freiwerden einer Partition der Prozess ausgewählt, der sie am besten ausfüllt, werden kleine Prozesse benachteiligt (und das will man absolut nicht).

Schließlich stellt sich hier und im folgenden die Frage, was beim Laden eines Programms geschieht. In der Maschinensprache werden konkrete

[1] Wir nehmen hier 1 MB = 1024 KB = 1048576 B an, obwohl diese Verwendung von „M“ der Norm widerspricht. Diesen (üblichen) Missbrauch setzen wir später fort mit 1 GB = 1024 MB und 1 TB = 1024 GB.

Adressen angesprochen. Das Programm soll aber in jeder ausreichend großen Partition laufen. Es gibt folgende Möglichkeiten:

1. *Verlagerung* (*Relokation*): Alle Adressen im Programm werden verändert. Der Binder liefert die Information dazu, welche Stellen angepasst werden müssen.

   Der entstehende Prozess verwendet absolute Speicheradressen, die aber aus Schutzgesichtspunkten schlecht sind (ein Prozess kann eine beliebige Stelle im Speicher adressieren). Abhilfe kann jedoch die Hardware schaffen, indem sie Speicherschutzschlüssel vergibt, die den Zugriff auf die Inhaber beschränken. Die Vergabe dieser Speicherschutzschlüssel erfordert privilegierte Maschinenbefehle.

2. *Relative Adressierung*: Anfangsadresse (Basis) und Länge des Prozesses werden in besonderen Registern gespeichert. Zu jeder angesprochenen Adresse wird zur Laufzeit der Inhalt des Basisregisters addiert (indizierte Adressierung). Das Längenregister erlaubt die Überprüfung der Zulässigkeit.

   Ein Vorteil ist, dass der Prozess zur Laufzeit im Speicher verschoben werden kann.

4.2 Ein- und Auslagerung (Swapping)

Wenn der Hauptspeicher nicht ausreicht, weil die Bearbeitung zu ineffizient (häufig sind alle Prozesse im Hauptspeicher blockiert) oder gar unfair (in einem Dialogsystem werden Benutzer ausgeschlossen, solange kein Hauptspeicher verfügbar ist) wird, müssen Prozesse zur Laufzeit durch andere ersetzt werden. Dazu dient die bereits besprochene *Speicherhierarchie*. Prozesse werden zwischen Primär- und Sekundärspeicher transportiert.

Gerne möchte man blockierte Prozesse auslagern. Wenn allerdings Daten durch DMA übertragen werden, kann der betroffene Speicherbereich nicht anderweitig benutzt werden. Um das zu vermeiden, kann das Betriebssystem Puffer zur Verfügung stellen, aus denen ein Prozess später die Daten holt.

Der Hauptspeicher wird in variable Partitionen entsprechend der Prozessgröße unterteilt. Dabei ergeben sich folgende Probleme:

- Mit der Zeit entstehen unbrauchbare Lücken zwischen den Partitionen (siehe Abbildung 4.2). Das ist *externer Verschnitt*. Wenn Prozesse im Hauptspeicher verschoben werden können, lässt sich der Speicher kompaktifizieren. Das ist allerdings zeitintensiv.

- Prozesse brauchen Speicherplatz für die Daten, die zur Laufzeit anfallen. Daher wird ihr Bedarf dynamisch wachsen. Wenn keine Lücke angrenzt, muss der Prozess in eine freie Partition ausreichender Größe

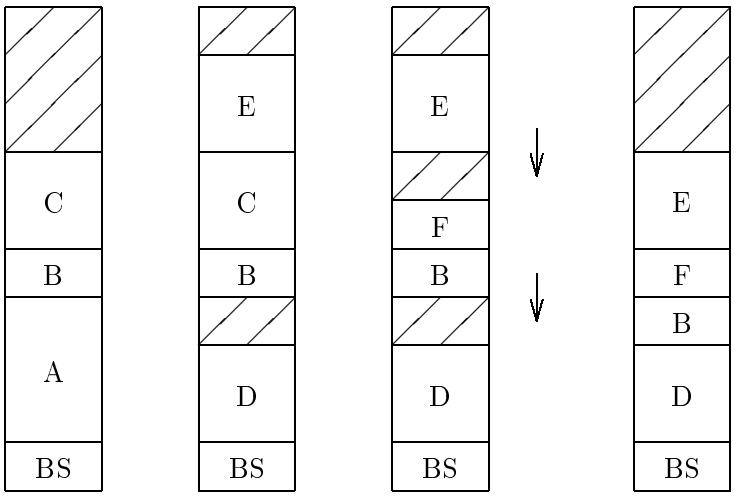


Abbildung 4.2: Kompaktifizierung

verschoben werden. Notfalls müssen andere Prozesse ausgelagert werden, um diese Partition zu schaffen. Wenn der Hintergrundspeicher nicht mehr ausreicht, muss der Prozess abgebrochen werden.

Da schon das Verschieben eines Prozesses aufwändig ist, sollte für jeden Prozess von Anfang an bzw. bei jeder Einlagerung eine Wachstumslücke vorgesehen werden. Diese wird praktischerweise zwischen Stapel und Halde gelegt, damit sie für den Teil zur Verfügung steht, der sie braucht. Es ergibt sich die Abbildung 4.3. Erst wenn sich Stapel und Halde berühren, muss das Betriebssystem eingreifen und die Partition des Prozesses vergrößern, den Prozess auslagern oder abbrechen. Unschön ist, dass die Wachstumslücke ebenfalls Platz auf dem Hintergrundspeicher beansprucht.

- Die freien Bereiche müssen verwaltet werden. Die im folgenden vorgestellten Verfahren sind auch für die Haldenverwaltung (`new`, `dispose` in Pascal; `malloc`, `realloc`, `free` in C) anwendbar.

4.2.1 Speicherverwaltung mit Bitvektoren

Der Speicher wird in gleich große Einheiten zerlegt. Der Bitvektor ist ein Inhaltsverzeichnis des Speichers mit einem Bit pro Einheit, das die Belegung der jeweiligen Einheit angibt. Wenn die Einheiten klein sind, wird der Bitvektor groß. Wenn die Einheiten aber groß sind, gibt es viel Verschnitt, da ein Prozess nur ganze Einheiten benutzen kann. Außerdem ist die Suche nach Lücken ausreichender Größe aufwändig.

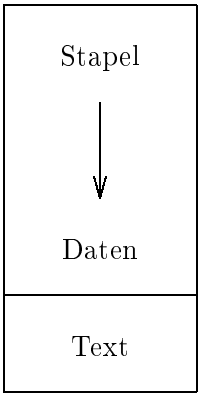
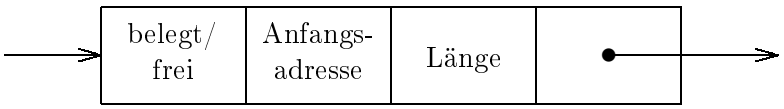


Abbildung 4.3: Aufteilung einer Partition

4.2.2 Speicherverwaltung mit verketteten Listen von Partitionen

Für jede Partition (variabler Größe) gibt es ein Element der Liste.



Die Liste ist nach den Anfangsadressen der Partitionen sortiert, damit beim Freiwerden leicht die Nachbarpartitionen bestimmt werden können. Falls ein oder beide Nachbarn auch frei sind, werden die Partitionen verschmolzen. Dazu ist es vorteilhaft, die Liste doppelt zu verketten.

Belegte Partitionen haben in der Liste offenbar keine Funktion. Für die freien Partitionen muss aber keine separate Liste geführt werden, denn die Information (Länge und Zeiger auf den Nachfolger) kann direkt in die Lücken geschrieben werden (wenn diese nicht zu klein sind).

Für die Auswahl einer angeforderten Partition gibt es viele Möglichkeiten:

- *First-Fit*: Die erste freie Partition ausreichender Größe wird gewählt. Der Speicher wird am Anfang zerstückelt.
- *Next-Fit*: Wie First-Fit, die Suche beginnt jedoch dort, wo die vorherige geendet hat. Scheint keine Vorteile zu bieten.
- *Best-Fit*: Unter allen freien Partitionen wird die kleinste gewählt, die noch genügend groß ist. Das Verfahren ist aber erstens langsam, da die gesamte Liste durchsucht werden muss, und erzeugt zweitens viele kleine Splitter, die garantiert nutzlos sind.
- *Worst-Fit*: Wählt immer die größte freie Partition. Das kann natürlich nicht überzeugen.

Suchzeit kann gespart werden, wenn die freien Partitionen nach ihrer Länge sortiert werden. Dann wird Best-Fit so schnell wie First-Fit. Der hohe Aufwand beim Verschmelzen wiegt aber schwerer.

4.2.3 Speicherverwaltung mit dem Buddy-System

Der Zugriff auf Speicherblöcke wird schneller, wenn diese in einem binären Baum gehalten werden. Bei der Anforderung von Speicherplatz wird die kleinste Partition ausreichender Größe gewählt und solange halbiert, bis die richtige Größe erreicht ist. Abbildung 4.4 zeigt die Anforderung von 100 KB bei einem verfügbaren Speicher von 2 MB.

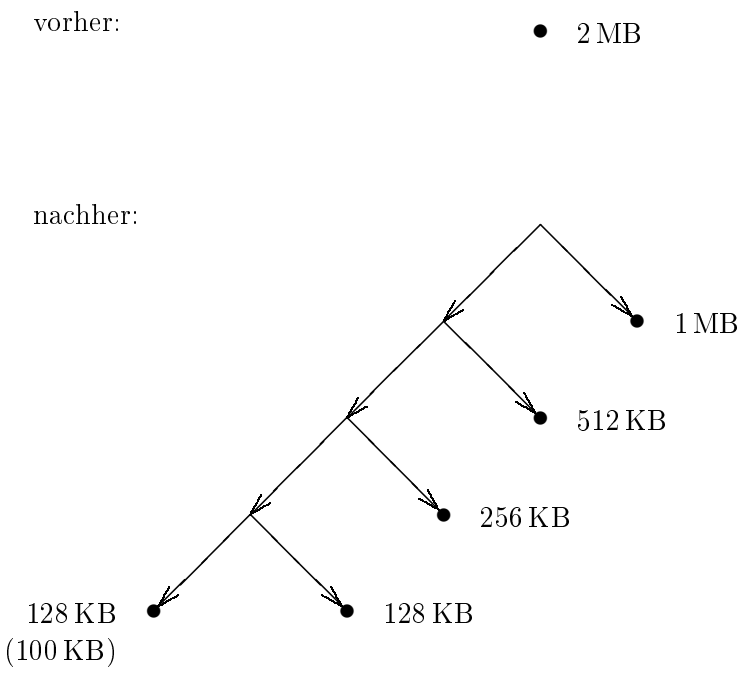


Abbildung 4.4: Speicherplatzanforderung im Buddy-System

Bei der Freigabe von Speicherplatz wird die Partition mit ihrem Partner (Buddy) verschmolzen, wenn dieser ebenfalls frei ist, und der Vorgang solange wie möglich wiederholt.

Der binäre Baum wird durch verkettete Listen für jede mögliche Größe (Zweierpotenz) implementiert. Das Verfahren ist sehr schnell, weil sowohl bei der Anforderung als auch bei der Freigabe sortierte Listen verwendet werden. Es entsteht aber viel interner Verschnitt.

4.2.4 Platzzuordnung für die Auslagerung

Meist wird der Platz für die Auslagerung auf den Hintergrundspeicher schon bei der Prozesserzeugung belegt. Falls nicht, entstehen die gleichen Probleme wie bei der Einlagerung, und die Verfahren für den Hauptspeicher sind anzuwenden.

4.3 Virtuelle Speicher

Bisher wurden nur ganze Prozesse ein- und ausgelagert. Falls ein Programm aber überhaupt zu groß für den Hauptspeicher ist, muss es in mehrere Teile zerlegt werden, die unabhängig voneinander lauffähig sind.

Bei der *Overlay*-Technik übernimmt der Programmierer die Aufteilung. Das Betriebssystem lagert die Teile bei Bedarf selbstständig ein und aus. Wenn die Aufteilung automatisiert wird, entsteht der *virtuelle Speicher*. Prozesse sprechen dann keine realen Hauptspeicheradressen an, sondern virtuelle, die von der *Speicherverwaltungseinheit* (Memory Management Unit, MMU) auf reale abgebildet werden. Teile von Programmen oder Daten können dann auf den Hintergrundspeicher ausgelagert werden, ohne dass der Prozess etwas davon merkt. Ein benötigter Block wird vom Hintergrundspeicher in den Hauptspeicher geladen und verdrängt eventuell einen anderen. Der Prozess wird währenddessen angehalten (blockiert). Die Größe eines Prozesses ist nur noch durch den Auslagerungsbereich auf dem Hintergrundspeicher beschränkt.

Es gibt zwei grundsätzliche Verfahren, die sich gut kombinieren lassen.

4.3.1 Segmentierung

Jeder Prozess bekommt eine Anzahl (statt bisher eines) von Segmenten variabler Größe. Dabei kann es sich um die bekannten Text-, Daten- oder Stapelsegmente handeln, aber auch um Dateien, die zwecks Leistungssteigerung im Hauptspeicher gehalten werden. Alle Segmente eines Prozesses stehen in einer Tabelle (siehe Abbildung 4.5), auf die dessen *Segmenttabellenbasisregister* verweist. Jeder Eintrag in dieser Tabelle enthält neben der Anfangsadresse die Länge L und den Schutzschlüssel S. Außerdem sind (nicht aufgeführt) das Anwesenheitsbit, das anzeigt, ob das Segment im Hauptspeicher liegt, und die Adresse des Segments auf dem Hintergrundspeicher vorhanden.

Eine virtuelle Adresse besteht aus der Segmentnummer, die als Index in die Tabelle dient, und einer Relativadresse, die schließlich zur Anfangsadresse des Segments addiert wird, um die reale Adresse zu erhalten. Die Relativadresse muss kleiner als die Länge L sein. Der Schutzschlüssel S enthält Zugriffsrechte (z. B. lesen, schreiben, ausführen, verlängern), die z. B. das Beschreiben eines Textsegments oder das Ausführen eines Datensegments verbieten.

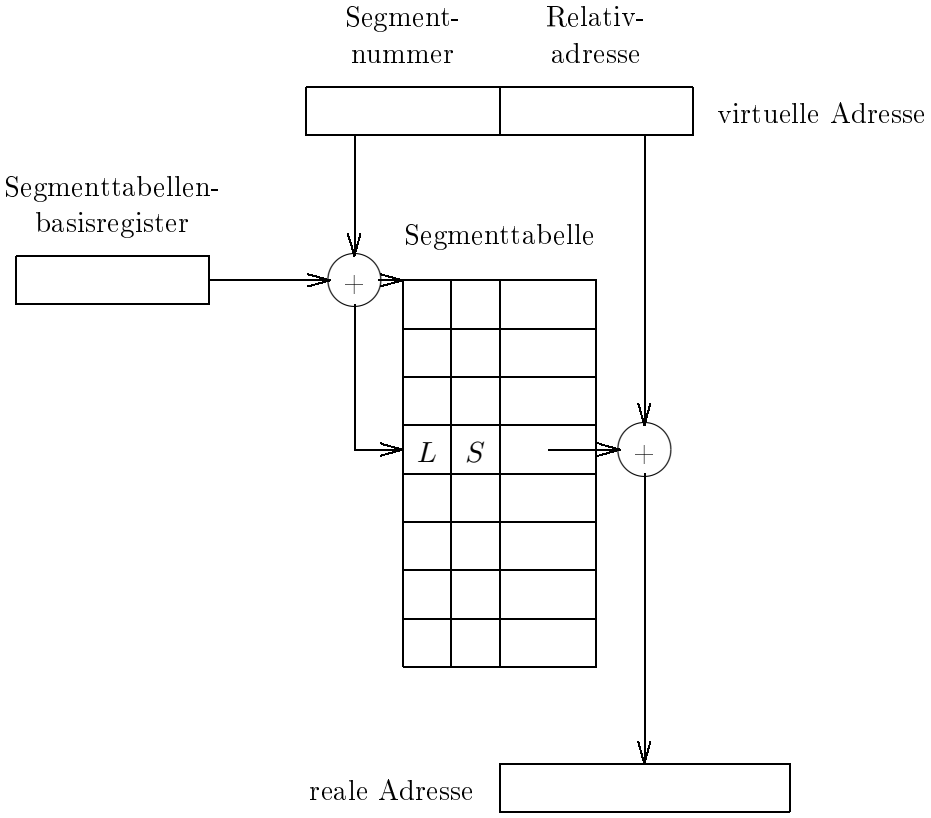


Abbildung 4.5: Segmentierung

Ganz nebenbei ergibt sich der Vorteil, dass mehrere Prozesse gemeinsame Segmente benutzen können. So kann die Interprozesskommunikation beschleunigt werden. (Das Kopieren der übermittelten Daten entfällt.) Zu beachten ist dabei, dass selbstbezügliche Segmente (also Daten mit Zeigern oder Text mit Sprüngen) überall die gleiche Nummer bekommen müssen. Mit der Auslagerung müssen alle betroffenen Prozesse einverstanden sein.

4.3.2 Seitenwechsel (Paging)

Der Hauptspeicher wird in Portionen (*Kacheln* oder *Rahmen*) gleicher Größe aufgeteilt. Ein Prozess erhält eine Anzahl von *Seiten* eben dieser Größe. Alle Seiten eines Prozesses stehen in einer Tabelle (siehe Abbildung 4.6), auf die dessen *Seitentabellenbasisregister* verweist. Jeder Eintrag in dieser Tabelle enthält neben der Kachelnummer das Anwesenheitsbit P. Letzteres gibt an, ob sich die angesprochene Seite im Hauptspeicher befindet. Wenn nicht, entsteht beim Zugriff ein *Seitenfehler*; es erfolgt eine Unterbrechung und der Prozess wird blockiert, während

- die Zugriffsberechtigung geprüft wird (falls vorgesehen),
- nötigenfalls eine andere Seite auf den Hintergrundspeicher ausgelagert wird, um eine Kachel freizumachen,
- die angesprochene Seite vom Hintergrundspeicher in diese Kachel geholt und
- die Tabelle angepasst wird.

Danach kann der Maschinenbefehl zu Ende gebracht (bzw. nochmals ausgeführt) und der Prozess fortgesetzt werden. Die Adresse der Seite auf dem Hintergrundspeicher steht ebenfalls in der Tabelle, obwohl sie nicht aufgeführt ist. Schutzbits sind ebenfalls möglich.

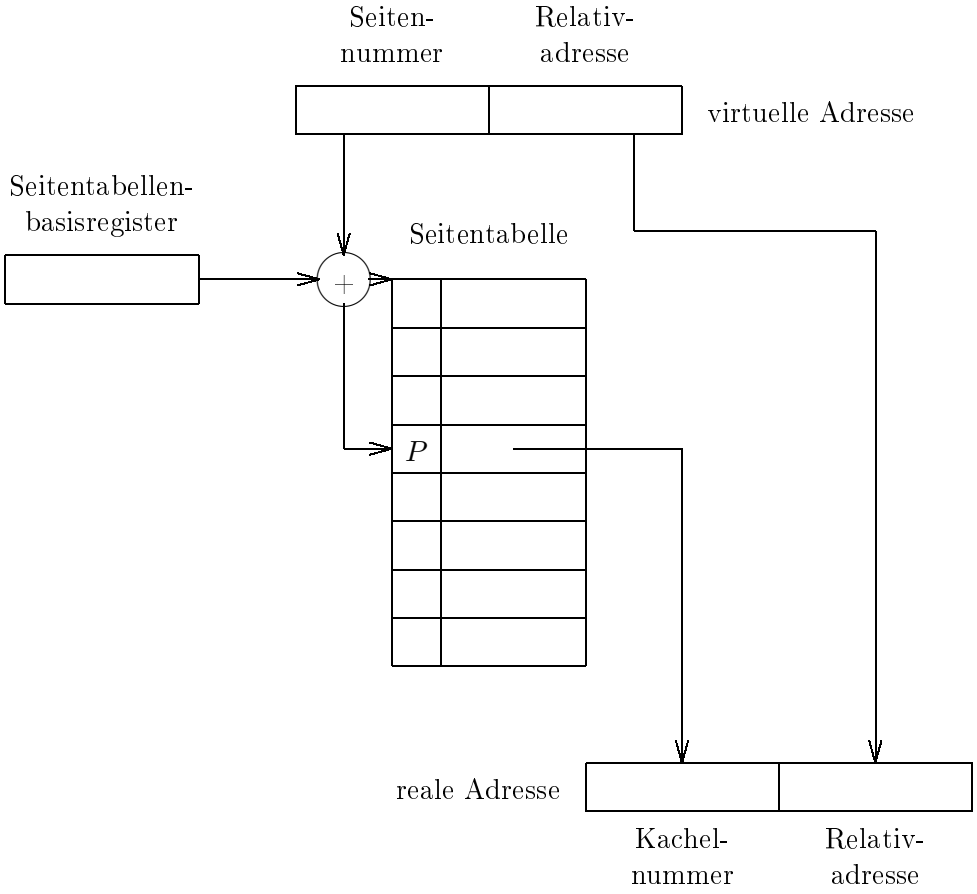


Abbildung 4.6: Seitenwechsel

Eine virtuelle Adresse besteht aus der Seitennummer, die als Index in die Tabelle dient, und einer Relativadresse, die schließlich zur Anfangsadresse

der Kachel addiert wird, um die reale Adresse zu erhalten. Die Kachelgröße wird normalerweise als Zweierpotenz gewählt, so dass statt einer Addition das Aneinanderhängen der Adressteile genügt.

Seiten sollten nicht zu groß gewählt werden, weil sonst die interne Fragmentierung hoch ist. Die letzte Seite eines Prozesses ist normalerweise nicht ausgefüllt. Zudem steigt die Wahrscheinlichkeit, dass häufig benutzte Seiten selten nachgefragte Daten enthalten. Bei kleinen Seiten dagegen wird die Seitentabelle lang (viele Seiten). Außerdem ist der Transport mehrerer kleiner Seiten von und zum Hintergrundspeicher aufwändiger als der einer großen, da bei einer Festplatte das Positionieren des Kopfes die meiste Zeit kostet.

Eine Seitentabelle zeigt Tabelle 4.1. Dabei wird ein virtueller Adressraum von 64 KB auf einen realen von 32 KB abgebildet. Die linke Tabelle enthält zu jeder Seitennummer die entsprechende Kachelnummer oder ein „X“, falls sich die Seite nicht im Hauptspeicher befindet.

Tabelle 4.1: Seitentabelle

Virtueller Adressraum	Zuordnung
0K–4K	2
4K–8K	1
8K–12K	6
12K–16K	0
16K–20K	4
20K–24K	3
24K–28K	X
28K–32K	X
32K–36K	X
36K–40K	5
40K–44K	X
44K–48K	7
48K–52K	X
52K–56K	X
56K–60K	X
60K–64K	X

Kachel (Rahmen)	Realer Adressraum
0	0K–4K
1	4K–8K
2	8K–12K
3	12K–16K
4	16K–20K
5	20K–24K
6	24K–28K
7	28K–32K

Das Betriebssystem führt in der *Kacheltabelle* Buch über alle Kacheln des Hauptspeichers. Ein Eintrag kann anzeigen, ob die Kachel belegt ist, und, wenn ja, von welcher Adresse auf dem Hintergrundspeicher die entsprechende Seite kam und welcher Eintrag einer Seitentabelle auf die Kachel verweist. Er wird außerdem Informationen für die Seitenersetzung enthalten, insbesondere ein Bit, das ggf. die Auslagerung des Inhalts verbieten kann. Bei der Einlagerung einer Seite wird die Hintergrundspeicheradresse einem Eintrag

der Seitentabelle entnommen und auf diesen verzeigert. Bei der Auslagerung kann er dann leicht auf den aktuellen Stand gebracht werden.

4.3.3 Seitenersetzungsalgorithmen

Bei einem Seitenfehler muss eine Seite eingelagert werden, aber dazu muss vorher meist Platz geschaffen werden. Zuerst wird eine Seite/Kachel ausgewählt und dann, falls sich der Inhalt geändert hat, auf den Hintergrundspeicher gesichert. Nach Möglichkeit sollte man jedoch keine Seite auslagern, die gleich wieder gebraucht wird. Da auch ein Betriebssystem nicht hellsehen kann, versucht man wie üblich von der Vergangenheit auf die Zukunft zu schließen. Zur geschickten Auswahl einer Seite werden die folgenden Algorithmen verwendet.

FIFO (First In First Out)

Bei einem Seitenfehler wird die am längsten anwesende Seite ausgelagert. Dazu sind die gefüllten Einträge der Kacheltabelle als Liste organisiert, die nach Alter geordnet ist. Es kann aber sehr häufig gebrauchte Seiten erwischen. Das Verfahren zeigt darüber hinaus eine erstaunliche Anomalie (von Belady entdeckt): Mehr verfügbare Kacheln können zu mehr Seitenfehlern führen! (Eine Kachel mehr kann im schlechtesten Fall die Anzahl der Seitenfehler fast verdoppeln!)

4.2 Beispiel. Es gibt fünf virtuelle Seiten, die in der Reihenfolge 012301401234 angesprochen werden. Bei drei Kacheln treten neun Seitenfehler auf, bei vier dagegen zehn!

LRU (Least Recently Used)

Besser ist es, die am längsten nicht benutzte Seite zu entfernen. Allerdings ist es zu aufwändig, eine entsprechende Liste zu führen. Die Hardware kann folgende Unterstützungen bieten:

- Die Maschinenbefehle werden mitgezählt; zu jeder Seite wird die Nummer des letzten Zugriffs gespeichert.
- Eine $n \times n$-Bitmatrix wird geführt (n sei die Anzahl der Kacheln). Bei einem Zugriff auf die k. Kachel wird die Summe der k. Zeile maximiert (alle Einträge werden auf Eins gesetzt) und danach werden alle Zeilensummen erniedrigt (die k. Spalte wird mit Nullen gefüllt).

NRU (Not Recently Used)

Da LRU meist zu aufwändig ist, muss es approximiert werden. Am einfachsten teilt man die anwesenden Seiten in Klassen ein:

Klasse 0: nicht modifiziert und in der letzten Zeiteinheit nicht angesprochen

Klasse 1: modifiziert und in der letzten Zeiteinheit nicht angesprochen

Klasse 2: nicht modifiziert und in der letzten Zeiteinheit angesprochen

Klasse 3: modifiziert und in der letzten Zeiteinheit angesprochen

Bei einem Seitenfehler wird irgendeine Seite aus der niedrigsten vorkommenden Klasse ausgelagert.

Voraussetzung ist, dass die Hardware die nötigen Informationen liefert. Pro Seite sind dies ein *Zugriffsbit*, das bei jedem Zugriff gesetzt wird, und ein *Modifikationsbit*, das bei jeder Veränderung gesetzt wird. Anfangs sind alle Bits Null. In festen Zeitabständen werden alle Zugriffsbits zurückgesetzt.

Eine Simulation dieser Bits durch Software ist darauf angewiesen, Seitenfehler zu provozieren, und zwar jeweils beim ersten lesenden und beim ersten schreibenden Zugriff.

FIFO und NRU

Durch Kombination mit NRU lassen sich die Nachteile von FIFO vermeiden. Die zwei Möglichkeiten sind:

- Lagere in der niedrigsten Klasse die älteste Seite aus.
- Lagere die älteste Seite aus, die in der vergangenen Zeiteinheit nicht angesprochen wurde. („Second Chance“: Falls das Zugriffsbit gesetzt ist, wird es nur zurückgesetzt und die Seite „verjüngt“, die Suche nach einem Auslagerungskandidaten geht aber weiter.)

NFU (Not Frequently Used)

Eine andere Möglichkeit ist es, die in einem gewissen Zeitintervall am wenigsten benutzte Seite auszulagern (LFU, Least Frequently Used). Da dies zu aufwändig ist, wertet man Zugriffe nicht einzeln. Für jede Kachel gibt es einen Zähler, zu dem in festen Zeitabständen das jeweilige Zugriffsbit addiert und danach zurückgesetzt wird. Bei einem Seitenfehler wird die Seite mit dem kleinsten Wert ausgelagert. Problematisch ist, dass „Rekorde“ bestehen bleiben. Also muss ein Verfahren der Alterung angewandt werden: Alle Bits werden in jedem Schritt nach rechts geschoben (und wandern schließlich aus dem Wort hinaus), das Zugriffsbit wird links angehängt (und wiegt am schwersten).

4.3.4 Segmentierung mit Seitenwechsel

Bei der Kombination beider Verfahren besteht jedes Segment aus einer variablen Anzahl von Seiten. Das Verfahren wird zweistufig, da jeder Eintrag in der Segmenttabelle auf den Anfang der zugehörigen Seitentabelle verweist. Eine virtuelle Adresse besteht jetzt aus drei Teilen (siehe Abbildung 4.7).

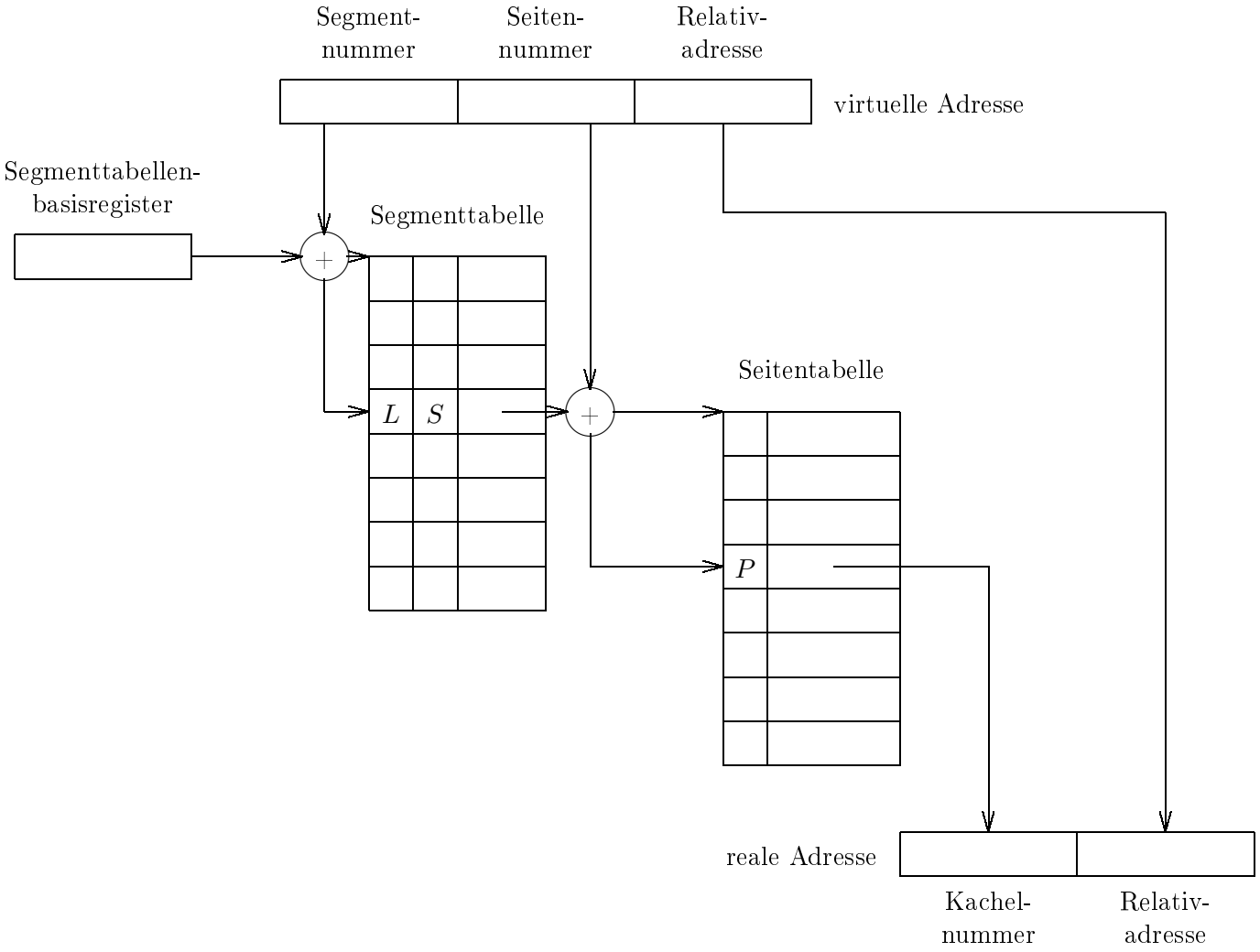


Abbildung 4.7: Segmentierung mit Seitenwechsel

Prinzipiell muss das Betriebssystem alle Tabellen verwalten. Einerseits sind sie normalerweise zu groß, um sie in Registern unterzubringen. Sie können sogar so groß werden, dass sie nicht mehr in den Hauptspeicher passen. Ein mehrstufiges Verfahren (mit einem Baum von Seitentabellen) erlaubt dann, untere Stufen (z. B. für die Wachstumslücke) auszulagern. Anderer-

seits werden die Tabellen auch zur Adressübersetzung gebraucht, wenn ein Prozess Daten mit dem Betriebssystem austauscht. Ein Prozess kennt nämlich die realen Adressen überhaupt nicht. Der Inhalt des Segmenttabellenbasisregisters steht im jeweiligen Prozesskontrollblock; es wird bei einem Prozesswechsel verändert. Die Kontextumschaltung geht daher schnell.

Allerdings braucht man jetzt immer drei Speicherzugriffe statt einen. Weil Speicherzugriffe sowieso einen Engpass darstellen, ist das untragbar. Abhilfe schafft ein (schneller) assoziativer Speicher, der *Translation Lookaside Buffer*. Er enthält einige Paare von virtuellen und realen Adressen (genauer: Segment- und Seitennummer gegenüber Kachelnummer), zusammen mit Verwaltungsinformation für die Seitenersetzung. Ein assoziativer Speicher ist inhaltsadressiert; eine gesuchte virtuelle Adresse wird gleichzeitig mit allen vorhandenen Paaren verglichen. Die entsprechende reale Adresse wird so, falls sie enthalten ist, ohne einen Hauptspeicherzugriff ermittelt. Assoziativspeicher ist teuer und daher knapp. Aber schon mit winzigen Speichern (z. B. 16 Einträge) lassen sich erfahrungsgemäß hervorragende Ergebnisse erzielen (Trefferquote über 90 %). Ein Eintrag im Translation Lookaside Buffer hat nicht genau das gleiche Format wie ein Eintrag in Segment- und Seitentabelle zusammengenommen, weil einerseits z. B. die Adresse einer ausgelagerten Seite auf dem Hintergrundspeicher nur bei einem Seitenfehler gebraucht wird (und da macht ein Hauptspeicherzugriff mehr nicht viel aus), andererseits Gültigkeit und Alter eines Eintrags von Interesse sind.

Die Umsetzung von virtuellen in reale Adressen wird zur Beschleunigung durch Hardware vorgenommen. Die besagte MMU, als Teil der zentralen Recheneinheit, enthält

- einen Teil der Tabellen, irgendwo zwischen dem Segmenttabellenbasisregister allein und allen Segment- und Seitentabellen für alle Prozesse,
- ggf. den Translation Lookaside Buffer,
- die Logik, um nötigenfalls in den Tabellen im Hauptspeicher nachzusehen sowie ggf. einen Seitenfehler auszulösen. (Falls der Assoziativspeicher groß genug ist, kann man das Nachsehen in den Tabellen dem Betriebssystem übertragen, sich also mehr Zeit lassen.)

Die Register und (soweit vorhanden) Tabellen in der MMU werden bei einem Prozesswechsel geladen. Das ist natürlich ein privilegierter Vorgang. Der Inhalt des Translation Lookaside Buffers wird gelöscht, damit kein Prozess die Seiten seines Vorgängers benutzt. Bei jedem Seitenfehler wird auch der Assoziativspeicher auf den neuen Stand gebracht. Effizienter ist es, die Daten mehrerer Prozesse in der MMU zu halten und über Prozessnummern zuzugreifen.

4.3.5 Fallstudie: Intel 80386

Der Intel 80386 unterstützt im Protected Mode sowohl Segmentierung als auch (optional) Seitenwechsel. Er besitzt ein globales und ein lokales Segmenttabellenbasisregister. Das erste, in Verbindung mit einem Längenregister, wird vom Betriebssystem nur einmal geladen und dann nicht mehr verändert. (Es ist vergleichbar mit der Anfangsadressen-/Längenkombination für den Unterbrechungsvektor.) Die globale Segmenttabelle enthält unter anderem Einträge für die lokalen Segmenttabellen und die Prozesskontrollblöcke. Das lokale (und damit eigentliche) Segmenttabellenbasisregister soll bei einem Prozesswechsel verändert werden. (Es ist vergleichbar mit dem Prozesszustandsregister, das auf den Prozesskontrollblock verweist.) Es enthält einen Index in die globale Segmenttabelle. Bei jeder Veränderung des lokalen Segmenttabellenbasisregisters wird der entsprechende Eintrag aus der globalen Segmenttabelle in unsichtbare Register kopiert, so dass anschließende Zugriffe schnell gehen. Sechs Segmentregister (für Stapel-, Daten-, Text- und drei weitere Segmente) enthalten Indexe wahlweise in die globale oder lokale Segmenttabelle. Auch für sie gibt es unsichtbare Register, die den entsprechenden Eintrag speichern. Ein Index besteht aus einer 13 Bit-Nummer, dem global/lokal-Indikator und der Privilegierungsstufe (Betriebsart). Daher kann ein Prozess bis zu 8192 verschiedene Segmente ansprechen. Ein Eintrag besteht aus einer 32 Bit-Basisadresse, einer 20 Bit-Länge (die wahlweise in Bytes oder Seiten zu interpretieren ist), Anwesenheitsbit, Privilegierungsstufe, Zugriffsrechte usw. Der virtuelle Adressraum umfasst somit wahlweise 4 GB oder 64 TB gegenüber dem „linearen“ Adressraum von 4 GB.

Falls Seitenwechsel eingeschaltet ist, werden die entstandenen 32 Bit-Adressen nochmals über ein zweistufiges Verfahren in 32 Bit-Adressen übersetzt. Ein Seitentabellenbasisregister soll bei einem Prozesswechsel verändert werden. Es verweist auf ein Seitenverzeichnis, dessen Einträge auf Seitentabellen verweisen, die ihrerseits (natürlich) Kachelnummern enthalten. Seiten sind 4 KB groß, umfassen also ein kleines Vielfaches der Größe eines Sektors auf der Festplatte. Kacheln haben daher 20 Bit-Nummern, und so gibt man auch jedem Seitenverzeichnis und jeder Seitentabelle 1024 Einträge zu je 4 B, damit diese genau in eine Kachel passen. Eine „lineare“ Adresse besteht schließlich aus dem 10 Bit-Index im Seitenverzeichnis, dem 10 Bit-Index in der Seitentabelle und dem 12 Bit-Index in der Kachel. Nur die oberen 20 Bits des Seitentabellenbasisregisters werden berücksichtigt und als Kachelnummer interpretiert. Jeder Eintrag in einem Seitenverzeichnis oder einer Seitentabelle wird also über die 20 Bit-Basisadresse, dem 10 Bit-Index und zwei Null-Bits adressiert. Er besteht aus einer 20 Bit-Kachelnummer, dem Anwesenheitsbit, einem primitiven Schutzschlüssel und Zusatzinformation für die Seitenersetzung. Auch Seitentabellen können folglich ausgelagert werden, aber nur, wenn sie ausschließlich auf ausgelagerte Seiten verweisen. Der Eintrag zu einer ausgelagerten Seite(ntabelle) enthält deren 29 Bit-Adresse auf

dem Hintergrundspeicher. Der normale Inhalt wurde überschrieben bis auf den primitiven Schutzschlüssel und das Anwesenheitsbit. Bei einem Seitenfehler nimmt ein spezielles Register die angesprochene „lineare“ Adresse auf. Der Translation Lookaside Buffer umfasst 32 Einträge. Er wird bei jedem Schreibzugriff auf das Seitentabellenbasisregister gelöscht, selbst wenn sich dessen Wert nicht ändert.

Ein Eintrag in einer Segmenttabelle enthält ein Bit, das bei jedem Zugriff gesetzt wird. Da Textsegmente immer schreibgeschützt sind, braucht man nur die Datensegmente anfangs schreibzuschützen, um NRU zu simulieren. Ein Eintrag in einer Seitentabelle enthält neben Zugriffs- und Modifikationsbit drei Bits, die vom Betriebssystem benutzt werden können. Das sind gute Voraussetzungen für NFU. Umfangreichere Daten müssen in der Kacheltabelle gespeichert werden. Die Kacheltabelle braucht mehr als ein Bit, um die Gültigkeit eines Eintrags anzuzeigen. Neben normalen Seiten kommen nämlich auch Seitenverzeichnisse und -tabellen vor, die besonders behandelt werden müssen.

4.3.6 Das Arbeitsmengen-Modell

Bisher wurde implizit davon ausgegangen, dass nur ein Prozess (der möglicherweise nicht in den Hauptspeicher passt) zu verwalten ist. Bei mehreren Prozessen sind weitere Überlegungen nötig.

Beim *Demand-Paging* wird eine Seite eingelagert, sobald ein Seitenfehler auftritt. Die notwendige Blockierung des Prozesses ist unproduktiv und obendrein häufig vorhersehbar. Zum Beispiel werden nach dem Laden eines Programms der Anfang des Programmtexts, die globalen Daten und ein kleiner Stapel sofort gebraucht.

Treten ständig sehr viele Seitenfehler auf, so sagt man, der Prozess „flattert“. Das kann z. B. daher kommen, dass bei LRU die Seiten bereiter Prozesse gute Auslagerungskandidaten sind, sie aber sofort wieder eingelagert werden, wenn die Prozesse laufen.

Normalerweise sprechen Prozesse in kleinen Zeiträumen nur einen kleinen Teil ihrer Seiten an, der sich auch nur langsam ändert (*räumliche und zeitliche Lokalität*). Dieser Teil heißt *Arbeitsmenge*. Beim *Prepaging* versucht das Betriebssystem, die Arbeitsmengen aller bereiten Prozesse einzulagern. Falls der Hauptspeicher nicht ausreicht, werden Prozesse angehalten.

Die Arbeitsmengen können geschätzt werden, indem man sich z. B. beim Alterungsverfahren die ersten n Bits (n geeignet) ansieht. Ein direkterer Weg ist die Messung der Seitenfehler pro Zeiteinheit (Page Fault Frequency, PFF). Ist die Rate zu hoch, braucht der Prozess weitere Kacheln, um seine Arbeitsmenge unterzubringen. Er bekommt die eingelagerte Seite zusätzlich. Ist die Rate zu klein, muss er Kacheln abgeben. Die Auswahl trifft die Seitenersetzungsstrategie.

Weil sich die Arbeitsmengen von Zeit zu Zeit abrupt ändern können, ist

eine *globale* Zuweisungsstrategie vorzuziehen: Wenn vor dem Einlagern einer Seite Platz geschaffen werden muss, wird die auszulagernde Seite unter allen vorhandenen ausgewählt. Trotzdem sollte keinem Prozess seine Arbeitsmenge entzogen werden. Eine *lokale* Zuweisungsstrategie (der Prozess, der den Seitenfehler hervorgerufen hat, muss die auszulagernde Seite abgeben) eignet sich nur, solange alle Prozesse ihre Arbeitsmengen besitzen.

Kapitel 5

Dateisystem

Eine *Datei* (File) ist ein Behälter für Daten, die unter einem Namen zusammengefasst sind. Dazu zählen Quelltexte, Programme und Datenbanken. UNIX und MSDOS unterscheiden diese Typen nicht, obwohl es Namenskonventionen gibt, die von manchen Anwendungsprogrammen erzwungen werden. Die üblichen Definitionen behaupten, dass Dateien als Folge von Datensätzen (Records) mit Feldern strukturiert sind. In UNIX und MSDOS sind Datensätze einfach Bytes. Eine wichtige Eigenschaft von Dateien ist ihre Dauerhaftigkeit (*Persistenz*); sie existieren auch nach Beendigung des erzeugenden Prozesses weiter, bis sie explizit gelöscht werden.

Der Zugriff auf eine Datei ist unabhängig von dem Gerät, auf dem sie sich befindet. Dateien stellen neben den Prozessen die wichtigste Abstraktion dar. In UNIX und vielen modernen Betriebssystemen wird außerdem *jedes* Gerät als Datei angesprochen. Dateien können wiederum in *Verzeichnissen* (Directories, Ordnern, Folders) zusammengefasst werden, so dass sich eine hierarchische Struktur ergibt. Verzeichnisse sind aber letztlich auch nur besondere Dateien.

5.1 Systemaufrufe

Das Bild, das das Dateisystem dem Benutzer bietet, ist in allen modernen Betriebssystemen ähnlich (nämlich baumförmig), auch wenn es unter verschiedenen Datei-Managern verborgen wird. Um die Aufgaben kennenzulernen, steigen wir daher gleich auf die Schnittstelle zum Betriebssystem hinunter. Die Aufrufe werden normalerweise auch von Anwenderprogrammen unter UNIX nicht direkt abgesetzt, sondern sind in bequemer zu handhabenden Bibliotheksfunktionen verborgen.

Es wäre sehr mühsam, wenn man für jede Datei die Folge aller Verzeichnisse (den *absoluten Pfadnamen*) angeben müsste, die vom Wurzelverzeichnis zu ihr führen. Daher ist jedem Prozess ein momentanes *Arbeitsverzeichnis* zugeordnet. *Relative Pfadnamen* beginnen nicht im Wurzelverzeichnis, sondern im momentanen Arbeitsverzeichnis. Das momentane Arbeitsverzeichnis

kann mit *chdir* gesetzt und mit *getcwd* abgefragt werden.

Jeder Prozess gehört seinem Eigentümer und zu einer Gruppe. Jede Datei, die ein Prozess anlegt (durch *creat*), erbt Eigentümer und Gruppe. Für jede Datei (und jedes Verzeichnis) führt UNIX Zugriffsrechte, die ebenfalls bei der Erzeugung angegeben werden: r = lesen, w = schreiben, x = ausführen (Dateien) bzw. durchlaufen (Verzeichnisse). Diese Rechte werden wiederum jeweils für den Eigentümer, für die Gruppe und für alle anderen Benutzer unabhängig geführt. Daneben gibt es das Zugriffsrecht s = setzen sowohl für Eigentümer als auch für Gruppe. Ein Prozess, der ein Programm in so einer Datei ausführt, läuft nicht unter seinem *wirklichen* Eigentümer (Gruppe), sondern unter dem Eigentümer (Gruppe) der Datei, der damit der *effektive* Eigentümer (Gruppe) wird. Diese und weitere Rechte kann der Eigentümer durch *chmod* ändern. Mit *access* kann ein Prozess testen, ob er die gewünschten Zugriffsrechte auf eine Datei besitzt. Durch *stat* kann ein Prozess Auskunft über weitere *Attribute* einer Datei bekommen, wie Länge und Datum der letzten Änderung. Mit *utime* kann das Datum geändert werden.

Dateien können durch *open* geöffnet werden, und zwar zum Lesen, zum Schreiben (der bisherige Inhalt geht verloren) oder zum Anhängen. Sinnvolle Kombinationen sind ebenfalls möglich. Der gelieferte Dateideskriptor ermöglicht den Zugriff auf den Inhalt. Dabei handelt es sich um einen Index in eine Tabelle fester Größe. Ein Eintrag enthält neben den Attributen die momentane *Position*. Der Dateideskriptor wird in Aufrufen von *read* (lesen), *write* (verändern), *lseek* (positionieren) und *close* (schließen) verwendet. Durch das Schließen wird der Dateideskriptor unbrauchbar. Eine Datei kann von einem Prozess mehrfach geöffnet werden, Einträge in der Deskriptortabelle können sogar kopiert werden (*dup*, *dup2*, allgemein *fcntl*). Dass die Deskriptortabelle bei der Prozesserzeugung kopiert wird, erlaubt es, die Ein- und/oder Ausgabe eines neuen Prozesses umzulenken.[1] Das Vorgehen zeigen die Abbildungen 5.1 und 5.2. Man könnte die übergebenen Dateien auch erst im Kindprozess öffnen (und müsste sie dann im Elternprozess nicht wieder schließen), würde aber Fehlersituationen später erkennen.

Verzeichnisse werden durch *mkdir* erzeugt und durch *rmdir* gelöscht, falls sie leer sind. Spezielle Systemaufrufe zum Bearbeiten gibt es nicht, wohl aber Bibliotheksfunktionen. Letztere beruhen darauf, dass Verzeichnisse eben auch Dateien sind. Wie man den Inhalt eines Verzeichnisses ausgeben kann, zeigt Abbildung 5.3.

Das Dateisystem in UNIX bildet keinen reinen Baum wie unter MSDOS, sondern einen gerichteten azyklischen Graphen (Directed Acyclic Graph, DAG). Ein oder mehrere Verzeichnisse können die gleiche Datei (Verzeichnisse sind hierbei ausnahmsweise nicht eingeschlossen) unter möglicherweise verschiedenen Namen enthalten (*hard Links*). Trotzdem kann eine Datei

[1] Ob ein Deskriptor bei einem `exec` geschlossen wird, hängt von einem Flag ab. Bei den drei Standarddeskriptoren ist es üblicherweise nicht gesetzt.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)

// Kommando: program <ifile >ofile
void ioredirection(const char *program, const char *ifile, const char *ofile)
{
    int ifd = open(ifile, O_RDONLY);
    if (ifd < 0) {
        perror(ifile);
        return;
    }
    int ofd = open(ofile, O_CREAT | O_WRONLY | O_TRUNC, CREATE_MODE);
        // creat(ofile, CREATE_MODE);
    if (ofd < 0) {
        perror(ofile);
        close(ifd);
        return;
    }
```

Abbildung 5.1: Erzeugung eines Kindprozesses mit umgelenkter Standard-Ein-/Ausgabe, Teil 1

aber nur einen Eigentümer haben! Ein zusätzlicher Pfad zu einer bestehenden Datei kann durch *link* angelegt werden. Sowohl auf Dateien als auch auf Verzeichnisse dürfen *unlink* (Pfad löschen) und *rename* (Pfad umbenennen) angewandt werden. Tatsächlich gibt es keinen Aufruf, der eine Datei explizit löscht! Die Löschung erfolgt implizit, sobald auf eine Datei nicht mehr zugegriffen werden kann.

Neben Namen für eine Datei, die alle gleichberechtigt sind, gibt es häufig *Verweise* (*symbolic* oder *soft Links*). Dies sind Dateien, die einen relativen Pfad, der an den aktuellen Pfad gehängt wird, oder einen absoluten Pfad, der den aktuellen Pfad ersetzt, enthalten. Bei jedem Zugriff wird folglich ein neuer Pfad erstellt, der auch ins Leere führen kann.

In UNIX gibt es keine Laufwerke; die Geräte werden durch *mount* in das Dateisystem *montiert*. Das Dateisystem auf einer bestimmten Festplatte oder Diskette wird dadurch zu einem Unterbaum (bzw. Unter-DAG; hard Links über Gerätegrenzen hinweg sind verboten) des Gesamtsystems. Auch

```
    switch (fork()) {
    case −1:
        perror("cannot fork");
        break;
    case 0: // child
        close(STDIN_FILENO);
        dup(ifd); // dup2(ifd, STDIN_FILENO);
        close(ifd);
        close(STDOUT_FILENO);
        dup(ofd); // dup2(ofd, STDOUT_FILENO);
        close(ofd);
        execl(program, program, NULL);
        perror("cannot exec");
        exit(EXIT_FAILURE);
        break;
    default: // parent
        close(ifd);
        close(ofd);
    }
}
```

Abbildung 5.2: Erzeugung eines Kindprozesses mit umgelenkter Standard-Ein-/Ausgabe, Teil 2

bewegliche Medien (z. B. Disketten) dürfen nicht einfach gewechselt werden, weil möglicherweise gepufferte Daten zurückgeschrieben werden müssen. Der Aufruf *umount* meldet ein Gerät ab und bringt dessen Dateisystem in einen konsistenten Zustand. Außerdem gibt es den Aufruf *sync*, um auch zwischendurch die gepufferten Daten zurückzuschreiben.

5.2 Entwurf des Dateisystems

Jetzt gehen wir auf Implementierungsgesichtspunkte ein. Da Medien mit sequentiellem Zugriff (wie Magnetbänder) eine primitive Struktur haben, konzentrieren wir uns auf Medien mit wahlfreiem Zugriff (wie CDs, Festplatten und Disketten). Letztere stellen wir uns zunächst wie Schallplatten mit einer fortlaufenden Rille vor; Einzelheiten folgen später.

5.2.1 Plattenverwaltung

Die Anforderungen ähneln denen bei der Hauptspeicherverwaltung, jedoch werden etwas andere Lösungen gewählt.

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

void directory(const char *dirname)
{
    DIR *dirp;
    if ((dirp = opendir(dirname)) == NULL) {
        perror(dirname);
        return;
    }
    struct dirent *direntp;
    while ((direntp = readdir(dirp)) != NULL)
        puts(direntp->d_name);
    closedir(dirp);
}
```

Abbildung 5.3: Anzeige eines Verzeichnisses

Die Abspeicherung einer Datei als fortlaufende Bytefolge ist sehr ungünstig, da Dateien wachsen können und das Verschieben noch aufwändiger als eine Verschiebung im Hauptspeicher ist. Außerdem sind Platten blockorientierte Geräte, so dass einzelne Bytes sowieso nicht adressiert werden können. Also wird eine Datei in Blöcke fester Länge aufgeteilt, die nicht fortlaufend abgelegt werden müssen.

Die Blockgröße darf nicht zu groß (z. B. eine Spur oder Zylinder, folgt später) gewählt werden, da sonst der Verschnitt zu hoch wird. Sie darf aber auch nicht zu klein (z. B. ein Sektor, ebenda) sein, da eine Datei dann aus sehr vielen Blöcken besteht, die Verwaltung aufwändig wird und beim Laden viele Plattenzugriffe nötig werden. Dennoch umfasst ein Block üblicherweise nur ganz wenige Sektoren.

Der freie Speicher kann als verkettete Liste der Plattenblöcke verwaltet werden. Um Plattenzugriffe zu sparen, nimmt ein freier Block neben der Verkettungsinformation so viele freie Blocknummern auf wie möglich. Alternativ kann ein Bitvektor (ein Bit pro Block) als Inhaltsverzeichnis dienen. Diese Speicherung ist kompakter, kann aber mehr Plattenzugriffe erfordern, wenn der Bitvektor nicht im Hauptspeicher gehalten werden kann.

5.2.2 Speicherung der Dateien

Eine Datei besteht aus einer Folge von Blöcken. Eine verkettete Liste ist aber ungeeignet, da eine Datei sonst sequentiell durchsucht werden müsste. (Außerdem müsste die Verkettungsinformation jedesmal entfernt werden, bevor

ein Benutzer die Daten zu sehen bekommt.) Daher wird die Verkettungsinformation zentralisiert.

File Allocation Table (FAT) aus MSDOS

Unmittelbar nach dem Bootsektor am Anfang einer Platte liegt der Vektor aus Abbildung 5.4. Den Anfangsblock einer Datei (im Beispiel der Dateien A,

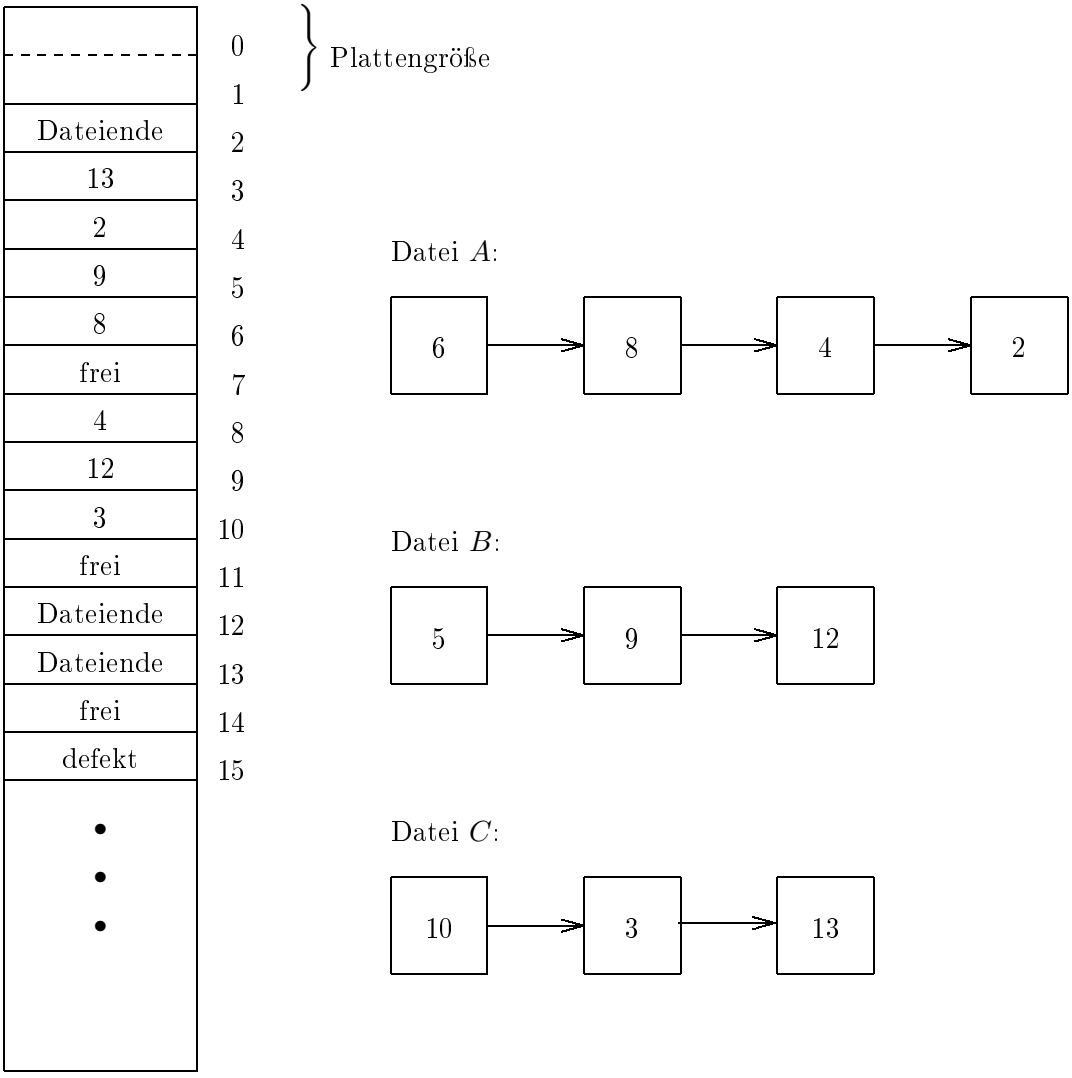


Abbildung 5.4: File Allocation Table

B und C) entnimmt man dem Verzeichnis, das den Dateinamen enthält. Den jeweiligen Fortsetzungsblock findet man in der FAT unter der Blocknummer. Die Verwaltung des freien Speichers ist bereits inbegriffen.

Bei großen Platten ergeben sich viele und lange Einträge, so dass die Tabelle zu lang wird, um sie im Hauptspeicher zu halten. Beispiel: Bei einer Platte von 500 MB mit Blöcken zu 1 KB sind 500 K Einträge zu je 3 B nötig, d. h. die FAT belegt selbst 1500 Blöcke. Dann kann die Suche eines Fortsetzungsblocks sehr lange dauern.

I-Node aus UNIX

Jede Datei hat ihre eigene Blockliste (i-Node, von „index“). Den i-Node zu einer Datei entnimmt man dem Verzeichnis, das den Dateinamen enthält. Die Struktur eines i-Node zeigt Abbildung 5.5. Sieben (MINIX) bis zwölf (BSD 4.2) Verweise auf Plattenblöcke stehen direkt im i-Node, so dass man bei kurzen Dateien direkt auf alle Blöcke zugreifen kann. Ist die Datei länger, werden für weitere Blöcke indirekte Verweise benutzt. Diese zeigen auf Plattenblöcke, die ihrerseits Verweise enthalten. Bis zu zwei (MINIX) oder drei (BSD 4.2) Indirektionsstufen sind vorgesehen. Die Plattenblöcke mit Verweisen werden natürlich nur angelegt, wenn sie gebraucht werden (aus dem Längenfeld des i-Node abzulesen).

Da der i-Node jeder geöffneten Datei im Hauptspeicher gehalten wird, braucht man für die Suche eines Fortsetzungsblocks maximal drei Plattenzugriffe. Die Verkettungsinformation braucht außerdem weniger Platz als eine FAT.

Bei Blöcken zu 1 KB und Blockadressen aus 4 B kann ein indirekter Block 256 Verweise aufnehmen. Wenn zehn direkte Verweise in den i-Node passen, reicht das bereits für Dateien bis zu einer Länge von 10 KB. Mit einfach indirekten Blöcken kommt man auf 266 KB. Zweifach indirekte Blöcke liefern zusätzliche 64 MB und dreifach indirekte Blöcke weitere 16 GB. Wird die Blockgröße auf 2 KB erhöht, darf *eine* Datei schon über 256 GB beanspruchen.

An *Attributen* stehen weiterhin im i-Node: Die Art der Datei (normale Datei, Verzeichnis, Verweis, Pipe, Socket, zeichen- oder blockorientiertes Gerät) und die Zugriffsrechte, Eigentümer und Gruppe, Zeit des letzten Zugriffs, der letzten Veränderung und der letzten Änderung des i-Node, Anzahl der Pfade zur Datei. Beim Öffnen einer Datei kommen im Hauptspeicher noch hinzu: Das Gerät, von dem der i-Node stammt, und die i-Node-Nummer, ein Zähler, wie oft die Datei geöffnet ist, und ein Kennzeichen, wenn ein anderes Gerät auf den i-Node montiert ist. I-Nodes werden erst gelöscht, wenn die Anzahl der Verweise Null wird *und* wenn sie von keinem Prozess geöffnet sind.

5.2.3 Struktur der Verzeichnisse

Verzeichnisse erlauben, Dateien über ihren Namen zu finden.

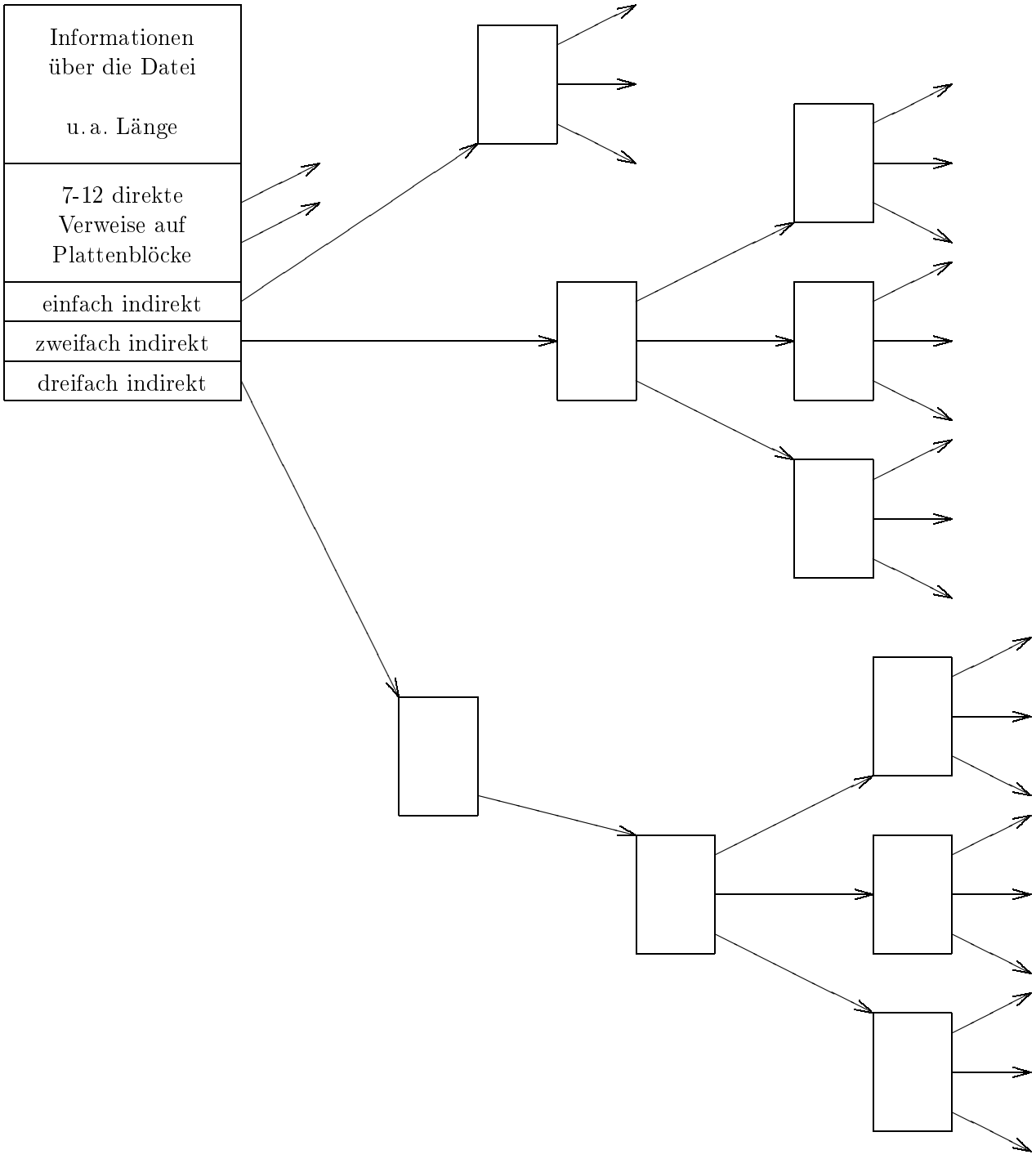


Abbildung 5.5: I-Node mit indirekten Blöcken

MSDOS

In MSDOS hat das Wurzelverzeichnis eine feste Länge. Alle anderen Verzeichnisse sind jedoch Dateien. Die Struktur eines Eintrags zeigt Abbildung 5.6.

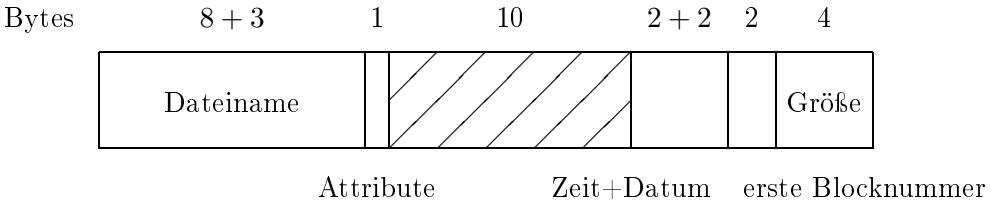


Abbildung 5.6: Verzeichniseintrag in MSDOS

Wie bekommt man lange Dateinamen (Windows) in diesem Schema unter? Man nutzt einen Fehler aus! Falls nämlich die illegale Attributkombination 0F (hex) benutzt wird, wird der Eintrag von DOS überlesen. Man bringt daher die Wndows-Dateinamen in Einträgen mit folgendem Aufbau unter: ein Byte enthält die laufende Nummer (Bits 1–5) und die Information, ob eine Fortsetzung existiert (Bit 6); die nächsten zehn Bytes enthalten fünf Unicode-Zeichen; das Attributbyte muss natürlich beibehalten werden; ein Byte enthält den Typ (0); ein Byte enthält die Prüfsumme für den dazugehörigen DOS-Dateinamen (so dass man hoffen kann, Abweichungen zu erkennen); die nächsten zwölf Bytes enthalten sechs Unicode-Zeichen; zwei Bytes enthalten die Clusternummer (0); die letzten vier Bytes enthalten noch einmal zwei Unicode-Zeichen. Die Namensteile werden meist in umgekehrter Reihenfolge im Verzeichnis abgelegt.

Dieses Format (VFAT unter Linux) bringt einige Probleme mit sich. Da die Länge des Wurzelverzeichnisses fest ist und diese unsichtbaren Einträge Platz wegnehmen, ist kaum vorherzusagen, wieviele Dateinamen noch hineinpassen. Wenn man unter DOS Dateien umbenennt oder löscht, bleiben die zusätzlichen Einträge unverändert. Wenn man dies unter Windows tut, entstehen mit der Zeit Lücken in den Verzeichnissen. Eine regelmäßige Defragmentierung des Dateisystems ist unerlässlich. Trotz allem dürfte dies das zur Zeit am weitesten verbreitete Dateisystem sein.

UNIX

Alle Verzeichnisse in UNIX sind Dateien. Die Struktur eines Eintrags zeigt Abbildung 5.7. Die Struktur in BSD 4.2 (und allen modernen UNIXen), wo Namen bis zur Länge von 256 Zeichen erlaubt sind, ist etwas komplizierter. Die i-Node-Nummer des Wurzelverzeichnisses ist fest (1 in MINIX, 2 in BSD 4.2).

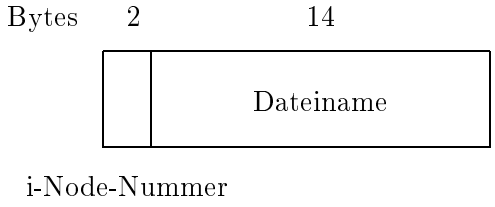


Abbildung 5.7: Verzeichniseintrag in UNIX

5.2.4 Einteilung der Platte

Dateien und Verzeichnisse werden zwar (scheinbar) wahllos auf der Platte verteilt, manche Daten verdienen aber einen besonderen Platz.

MSDOS

Die Einteilung einer Platte unter MSDOS zeigt Abbildung 5.8. Die FAT

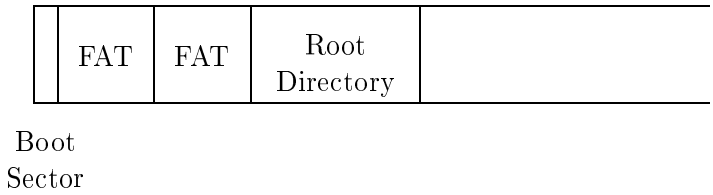


Abbildung 5.8: Platteneinteilung in MSDOS

wird zur Sicherheit mehrfach gehalten. Der Boot-Sektor enthält folgende Informationen: Sprungbefehl (3 B), Name (8 B), Anzahl der Bytes pro Sektor (2 B), Anzahl der Sektoren pro Cluster (1 B, entspricht den Blöcken bzw. Zonen in UNIX), Anzahl der reservierten Sektoren am Plattenanfang (2 B, normalerweise gleich 1), Anzahl der FATs (1 B), Anzahl der Einträge im Wurzelverzeichnis (2 B), Anzahl der verfügbaren Sektoren (2 B), Gerätekennung (1 B), Anzahl der Sektoren pro FAT (2 B), Anzahl der Sektoren pro Spur (2 B), Anzahl der Spuren pro Zylinder (2 B), Anzahl verborgener Sektoren (2 B, normalerweise gleich 0). Der Rest kann ein Programm aufnehmen. Am Ende steht eine magische Zahl.

UNIX

Die Einteilung einer Platte unter UNIX zeigt Abbildung 5.9. Zur Leistungssteigerung können mehrere Blöcke zu einer Zone zusammengefasst werden. Da die Platzverschwendung schnell unerträglich würde, wenn man Zonen als unteilbare Einheiten behandeln würde, erlaubt BSD 4.2 auch die Zuteilung einzelner Blöcke (Fragmente genannt). Eine Datei darf jedoch Fragmente

nur innerhalb einer Zone besitzen; eine Zone muss erst vollständig aufgefüllt sein, bevor Fragmente in einer neuen Zone angelegt werden. Falls in einer Zone kein Platz mehr ist, weil sie Fragmente von anderen Dateien enthält, werden die vorhandenen Fragmente in eine leere Zone kopiert und dort weitergemacht.

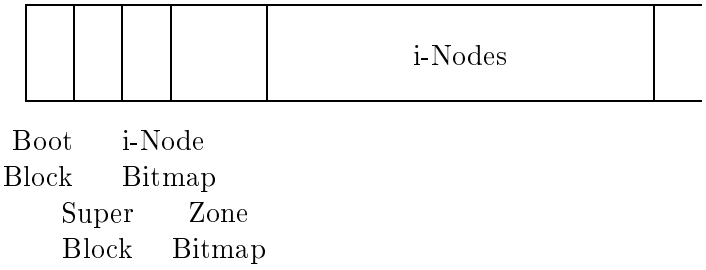


Abbildung 5.9: Platteneinteilung in UNIX

Der Inhalt des Boot-Blocks wird beim Systemstart ausgeführt. Der Super-Block enthält folgende Informationen: Die Anzahl der i-Nodes und der Zonen, die Größe der i-Node-Bitmap und der Zonen-Bitmap, die Adresse der ersten Datenzone, die Anzahl der Blöcke pro Zone (bzw. den Zweierlogarithmus davon), die maximale Dateigröße und eine magische Zahl. Die Super-Blöcke aller montierten Dateisysteme werden im Hauptspeicher gehalten. Dann kommen weitere Informationen hinzu: Zeiger auf die i-Node-Bitmap und die Zonen-Bitmap (ebenfalls im Hauptspeicher), das Gerät, von dem der Super-Block stammt, die i-Node-Nummer des Wurzelverzeichnisses auf dem montierten Dateisystem und der i-Node des Verzeichnisses, auf das montiert wurde, die Zeit der letzten Veränderung und ein Kennzeichen, ob das Dateisystem schreibgeschützt oder inkonsistent ist.

Ob ein Gerät montiert wurde, sieht man an der Information im i-Node der betroffenen Stelle. Daraufhin werden alle Super-Blöcke durchsucht, ob sie diese i-Node-Nummer enthalten.

In BSD 4.2 wurde zur Leistungssteigerung die Platte in Zylindergruppen eingeteilt. Jede Zylindergruppe enthält eine Kopie des Super-Blocks, da dieser sehr wichtige Informationen enthält, eigene i-Nodes und Bitmaps. So wird die Suchzeit zwischen i-Nodes und Datenblöcken bzw. mehreren Datenblöcken der gleichen Datei verringert. Damit Dateien annähernd zusammenhängend abgespeichert werden können, darf die Platte nicht ganz gefüllt werden (ca. 10% Reserve).

5.2.5 Leistungssteigerung des Dateisystems

Plattenzugriffe sind vergleichsweise langsam. Daher werden häufig gebrauchte Blöcke im Hauptspeicher (*Block Cache*) gehalten. Der Cache ähnelt den

Seiten im Hauptspeicher, ist jedoch nicht so zeitkritisch. Als Strategie ist LRU mit verketteten Listen möglich. Das Verfahren sollte aber modifiziert werden:

- Wenn der Block wahrscheinlich bald wieder gebraucht wird (vor allem halbvolle Blöcke beim Schreibzugriff, auch andere Datenblöcke, nicht aber i-Nodes, indirekte Blöcke, Verzeichnisse), wird er normal ans Ende der Liste gehängt, sonst an den Anfang.

- Wenn der Block wichtig für die Konsistenz des Dateisystems ist (i-Nodes, indirekte Blöcke, Verzeichnisse, nicht aber Datenblöcke), sollte er sofort auf die Platte gesichert werden. Dieses Verfahren heißt *Write Through* im Gegensatz zum normalen *Copy Back*. Datenblöcke werden in regelmäßigen Abständen durch den Systemaufruf *sync* gesichert.

5.2.6 Zuverlässigkeit des Dateisystems

Platten können ausfallen, daher sollte man in regelmäßigen Abständen Sicherungskopien der wichtigen Inhalte anfertigen.

Bei einem „Absturz“ des Rechners oder beim einfachen Ausschalten kann das Dateisystem inkonsistent werden. Leichte Fehler können mit dem UNIX-Kommando *fsck* (vergleiche *chkdsk* in MSDOS) repariert werden. Meist werden folgende Plausibilitätsprüfungen angestellt:

- Von jedem Block wird gezählt, wie oft er jeweils in den i-Nodes und in der Block-Bitmap vorkommt. Normalerweise muss jeder Block entweder in der ersten oder in der zweiten Liste genau einmal vorkommen. Folgende Fehler können auftreten:

  - Ein Block kommt in keiner der beiden Listen vor. Er fehlt der Verwaltung, wird also als frei markiert.
  - Ein Block kommt mehrfach in der Freiliste vor (bei Bitmaps unmöglich). Die Freiliste wird neu aufgebaut.
  - Ein Block kommt in beiden Listen vor. Er wird als belegt markiert.
  - Ein Block kommt in i-Nodes mehrfach vor. Es werden Kopien seines Inhalts angelegt und verteilt. Datenverlust ist wahrscheinlich.

- Von jedem i-Node wird gezählt, wie oft er in Verzeichnissen vorkommt. Normalerweise müsste der Wert mit dem Verweiszähler übereinstimmen, sonst wird der Verweiszähler angepasst.

- Unmögliche Block- oder i-Node-Nummern werden erkannt.

- Die Anzahl der Blöcke und i-Nodes wird mit der Information im Super-Block verglichen.

Bei heutigen großen Platten ist eine Reparatur des Dateisystems oft zu aufwändig und unzuverlässig. Daher verbreitet sich das *Journaling*, ein Transaktionskonzept wie bei Windows NT. Strukturelle Änderungen am Dateisystem (Dateien anlegen oder löschen zum Beispiel) werden nicht direkt ausgeführt, sondern zunächst in einem speziellen Bereich zwischengespeichert, dem *Journal* oder *Log*. Da fortlaufende Blöcke geschrieben werden, geht dies recht effizient. Sobald ein Eintrag erfolgreich abgeschlossen wurde, gilt die Änderung als verbindlich. Einträge, die es nicht soweit schaffen, werden beim nächsten Wiederhochfahren des Rechners rückgängig gemacht.

Im laufenden Betrieb werden die Journal-Einträge schrittweise abgearbeitet. (Eine Änderung erfordert üblicherweise mehrere Aktionen, z. B. einen Eintrag in einem Verzeichnis, Zuteilung eines i-Nodes und Zuteilung eines Blocks.) Wenn alles Notwendige ausgeführt wurde, wird der Journal-Eintrag gelöscht. Kommt es nicht so weit, werden beim nächsten Wiederhochfahren des Rechners alle fehlenden Schritte nachgeholt.

5.3 Verteilte Dateisysteme

Durch Montieren wurden die Datenträger hinter dem Dateisystem verborgen. Wenn Rechner vernetzt sind, sollten sich auch Geräte an verschiedenen Rechnern zum Aufbau eines gemeinsamen Dateisystems verwenden lassen. Zwei gängige Produkte sind Sun-NFS (Network File System) und AFS (Andrew File System; die Namensgeber der Carnegie-Mellon-Universität hießen beide Andrew mit Vornamen).

5.3.1 Sun-NFS

Jeder Rechner mit eigener Platte kann Teile seines lokalen Dateisystems exportieren. Jeder andere Rechner im Netz kann Teile der exportierten Dateisysteme in sein eigenes Dateisystem montieren. Er kann sein eigenes Dateisystem sogar ausschließlich aus „geliehenen“ Komponenten zusammenstellen. Er muss nur angeben, welches Verzeichnis er von welchem Rechner haben möchte. Dateisysteme, die der Exporteur (Server) möglicherweise in den exportierten Baum montiert hat, bekommt der Importeur (Client) allerdings nicht zu sehen. So erhält jeder Rechner im Netz sein individuelles, aber homogen wirkendes Dateisystem.

Beim Montieren eines Verzeichnisses gibt der Exporteur dem Importeur eine entsprechende Kennung (File Handle). Geräte- und i-Node-Nummer genügen dem Exporteur, um jeden Dateinamen eindeutig aufzulösen. Für den Importeur sind das nur nichtssagende Zahlen, die er sich merken muss. Der Exporteur merkt sich dagegen nichts von Bedeutung; NFS ist *gedächtnislos*. Der Vorteil wird bei einem Zusammenbruch deutlich: Sobald der Server wieder läuft, geht es für die Clients weiter wie vorher; stirbt ein Client, hat der Server keine Probleme mit seiner Buchführung. Bei jedem Dateizugriff muss

der Client alle erforderlichen Daten bereitstellen. Insbesondere gibt es keine offenen Dateien; jede Lese- oder Schreiboperation wird mit dem Dateinamen und der Position aufgerufen.

Alle Systemaufrufe, die das Dateisystem betreffen, werden von der VFS-Schicht (*Virtual File System*) bearbeitet. Sie arbeitet mit netzwerkweit eindeutigen v-Nodes. Anforderungen werden je nach Bedarf an den lokalen Bearbeiter oder an NFS weitergegeben. Umgekehrt werden auch NFS-Anforderungen, die von außerhalb kommen, von der VFS-Schicht bearbeitet. NFS läuft als eigener Prozess.

Sun-NFS ist als Sammlung von Sun-RPCs implementiert. Das macht die Sache einerseits relativ aufwändig, weil jede Anforderung auf dem Server vollständig abgearbeitet sein muss, bevor der Client eine Antwort bekommt. Jede Komponente eines Pfadnamens erfordert obendrein einen eigenen Prozedur-Fernaufruf, weil der Exporteur das Dateisystem des Importeurs nicht kennt. Daher werden Caches für bereits bekannte Daten eingesetzt. Die wiederum zerstören die Semantik der Dateioperationen vollends; Wirkungen werden erst nach einer gewissen Zeit (3 s für Dateien, 30 s für Verzeichnisse) beim Server sichtbar.

5.3.2 AFS

Andrew treibt das Cache-Prinzip auf die Spitze: So, wie man den Hauptspeicher als Cache für den Hintergrundspeicher ansehen könnte, lässt sich eine lokale Platte als Cache für ein globales Dateisystem verwenden. Der Cache hilft, Netzverkehr zu vermeiden. Er wirkt wie ein virtueller Speicher. Mehrere ausgezeichnete Server bieten den Clients ein netzwerkeinheitliches, *riesiges*, homogenes Dateisystem. Der Standort der Dateien kann sogar geändert werden, ohne dass die Clients etwas merken würden. Clients besitzen jedoch auch eine Platte, auf der sie ein lokales Dateisystem halten, in das das globale montiert wird. Außerdem reservieren sie einen Cache-Bereich, in dem lokale Kopien von ganzen Dateien liegen. Beim Öffnen einer Datei wird eine lokale Kopie angefertigt, beim Schließen wird der letzte Stand zum Server gebracht. Verändert werden nur die lokalen Kopien.

Lokale Kopien werden aufgehoben, selbst wenn die Dateien geschlossen sind. Wenn ein Client den Server bei jeder Gelegenheit fragen würde, ob die Kopie noch aktuell ist, wäre das zu aufwändig. Stattdessen merkt sich der Server, welche Clients lokale Kopien haben. Sobald er von einem Client eine neue Version bekommt, teilt er dies allen anderen mit. Stürzt ein Client ab, markiert er seine Cache-Einträge selbst als ungültig. Außer einem anfangs langsameren Zugriff hat er davon keine Nachteile.

Die wenigen externen Anfragen sind RPCs. Darin werden Dateien über netzwerkweit einheitliche *Fids* (File Identifier) angesprochen. Der wichtigste Teil davon ist eine Volume-Nummer. *Volumes* sind die kleinsten Verwaltungseinheiten. Sie bestehen aus einigen wenigen Verzeichnissen. Jeder Ser-

ver besitzt eine Kopie der globalen Volume-Datenbank.

5.4 Schutzmechanismen

Sicherheit umfasst sowohl *Zuverlässigkeit* als auch *Schutz*. Zuverlässigkeit wird durch Redundanz gesteigert. Dem Schutz dient die Vergabe von Zugriffsrechten. Die Schutzproblematik wird in den letzten Jahren immer drängender, weil Rechner häufig zur Verwaltung großer Datenmengen (in Datenbanken) eingesetzt werden, die nicht für die Allgemeinheit bestimmt sind. Wirksamer Schutz muss auf einer möglichst niedrigen Stufe ansetzen. Im Betriebssystem ist hauptsächlich das Dateisystem betroffen.

Schutz beginnt mit der Identifizierung der Benutzer. Dazu werden meist Passwörter verlangt. Ist ein Passwort leicht zu erraten, ist die erste Stufe schon umgangen. Kein gutes Betriebssystem wird, nicht einmal kurzzeitig, unverschlüsselte Passwörter verwenden. Vielmehr wird das eingegebene Passwort verschlüsselt und in dieser Form mit dem erwarteten Wert verglichen. Um systematisches Probieren mit Hilfe von Computern zu verhindern, sollte einerseits die Verschlüsselung künstlich langsam gehalten werden, was außerdem bei jedem Fehlversuch gesteigert werden kann. Andererseits sollten auch die verschlüsselten Passwörter nicht direkt zugänglich sein. Wichtig ist schließlich, dass durch das Passwort nur die beantragten Privilegien erreicht werden. Es wäre z. B. fatal, wenn ein Benutzer den Login-Prozess anhalten könnte, um nachträglich den Benutzernamen auszutauschen. Verschlüsselungstechniken sind im Übrigen die einzige Möglichkeit, das Problem der *Beglaubigung* (Authentication) zwischen vernetzten Rechnern zu lösen.

Sind die Benutzer bekannt, kann man ein allgemeines Modell aufstellen: die *Zugriffsmatrix*. Es gibt Subjekte (Prozesse), die man in *Domains* zusammenfassen kann. In UNIX ist ein Domain durch Eigentümer und Gruppe gegeben; alle Subjekte mit diesen Kennzeichen haben die gleichen Rechte. Subjekte können unter Umständen den Domain wechseln, indem sie zuverlässige Agenten beauftragen. Agenten in UNIX machen von den s-Rechten Gebrauch. Außerdem gibt es natürlich Objekte, zu denen Dateien, Betriebsmittel und die Subjekte zählen. Ein Objekt bietet eine Menge von anwendbaren Operationen an, gehört also zu einem abstrakten Datentyp. In der Zugriffsmatrix steht zu jedem Domain-Objekt-Paar die Menge der Rechte (eine Teilmenge der anwendbaren Operationen), die ein Subjekt des Domains am Objekt hat.

Die Zugriffsmatrix wird nicht so abgespeichert, weil sie sehr dünn besetzt ist. Stattdessen kann man

- die *Berechtigungen* (Capabilities) beim Domain speichern. Dabei handelt es sich um eine Liste von Objekten mit den dazugehörigen Rechten, also eine Zeile der Zugriffsmatrix. Dieses Schema macht Zugriffe sehr

effizient (in UNIX könnte man Deskriptoren geöffneter Dateien als Capability interpretieren), hat aber den Nachteil, dass einmal erteilte Berechtigungen nur schwer wieder entzogen werden können. Berechtigungen selbst werden als spezielle Objekte (mit Operationen wie Erteilen, Weitergeben, Entziehen) verwaltet, die vor unberechtigten Subjekten geschützt sind.

- die *Zugriffskontrollen* (Access Control List, ACL) beim Objekt speichern. Dabei handelt es sich um eine Liste von Domains mit den dazugehörigen Rechten, also eine Spalte der Zugriffsmatrix. Speicherplatz lässt sich sparen, indem eine Voreinstellung angegeben wird, die für alle nicht genannten Domains gilt. In UNIX wird das Schema zu den neun Zugriffsbits komprimiert.

Eine weitere Möglichkeit ist die Einführung von *Schlüsseln*, die sowohl an Domains als auch an Objekte vergeben werden. Ein Zugriff ist erlaubt, wenn Domain und Objekt einen Schlüssel gemeinsam haben.

Die Zugriffsmatrix ändert sich: Subjekte und Objekte können erzeugt und gelöscht werden, Rechte können erteilt und entzogen werden. Benutzerkommandos setzen gewisse Zugriffsrechte voraus und nehmen dann gewisse Änderungen vor. Gegeben endliche Mengen von Rechten und Kommandos, kann irgendwann das Subjekt s das Recht r am Objekt o erhalten? Diese Frage ist formal unentscheidbar! Man zeigt dies durch Simulation von Turing-Maschinen (auf der Hauptdiagonalen der Zugriffsmatrix). Folglich müssen Rechte und Kommandos sehr sorgfältig konstruiert werden; ein Korrektheitsbeweis kann im Einzelfall recht schwierig werden.

Kapitel 6

Netzwerke und verteilte Systeme

Neben die klassischen Aufgaben eines Betriebssystems (Prozesse, Speicher, Dateisystem und Geräte) tritt zunehmend die Vernetzung. In kommerziellen Anwendungen werden Rechner nicht mehr isoliert betrieben.

UNIX bietet eine Vielzahl von Mechanismen zur Interprozesskommunikation, insbesondere auch im Zusammenhang mit dem Internet. Das ist kein Wunder, denn das Internet entstand ursprünglich als Verbindung zwischen UNIX-Rechnern.

6.1 Das Referenzmodell für offene Systeme

Damit Rechner verschiedener Hersteller vernetzt werden können, muss ein unabhängiger Standard existieren. Die ISO (International Standards Organization) hat das Referenzmodell für offene Systeme OSI (Open Systems Interconnection) entwickelt. Es enthält sieben Schichten und zugehörige Protokolle (siehe Abbildung 6.1).

Konzeptionell findet der Datenaustausch zwischen den Kommunikationspartnern innerhalb einer Schicht statt. Dazu wird jeweils ein *Protokoll* verwendet (waagrechte, gestrichelte Linien). Die senkrechten Verbindungen stellen Schnittstellen dar. Jede Schicht baut auf der darunter liegenden auf, d. h. benutzt die Dienste der untergeordneten Schicht, um die eigenen Dienste zu implementieren. Tatsächlich erfordert eine Datenübertragung also, dass der Sender seine Daten durch die Schichten nach unten reicht, bis sie real übertragen werden können, wonach der Empfänger die Daten nach oben durch die Schichten wandern lässt. Beim Durchgang durch die unteren Schichten werden den Daten normalerweise auf der Senderseite zusätzliche Informationen hinzugefügt, die auf der Empfängerseite entfernt werden.

Die Schichten sind durchnumeriert. Ihnen werden folgende Aufgaben zugeordnet:

1. Die *Bitübertragungsschicht* (Physical Layer) steuert das Übertragungsmedium. Sie kümmert sich um die elektrische Darstellung der Bits

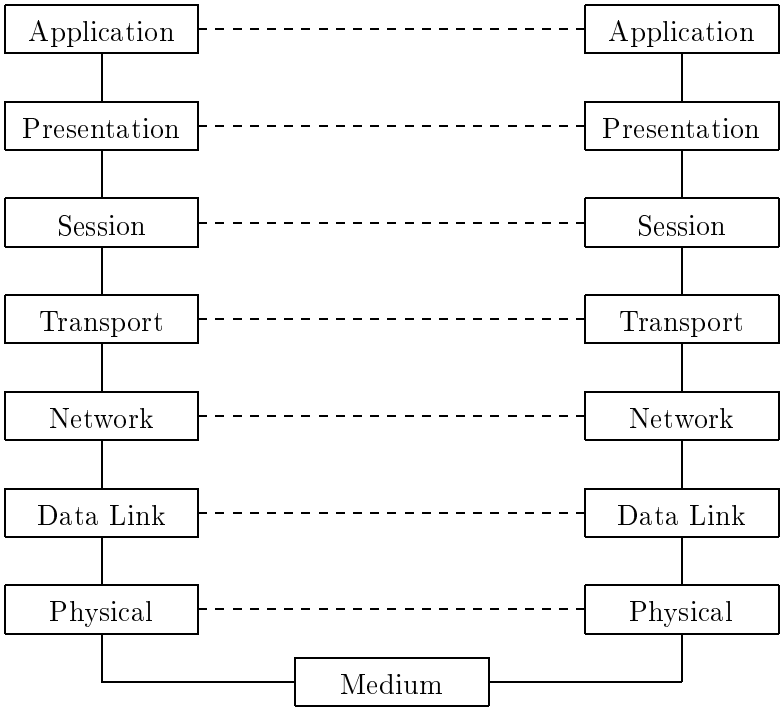


Abbildung 6.1: OSI-Referenzmodell

(Größe und Dauer der Impulse). Außerdem gehört hierzu die Normung der Stecker und Kabel.

Beispiele für Übertragungsmedien sind:

- Telefonleitung (analoge Signale im akustischen Bereich)
- ISDN (Integrated Services Digital Network)
- DSL (Digital Subscriber Line)
- Thin-Wire-, Thick-Wire- oder Twisted-Pair-Ethernet (für lokale Netze)
- Satellitenfunk
- RS-232-C, V.24 (für serielle Schnittstellen)

Beim Thin-Wire-Ethernet handelt es sich um ein Koaxialkabel, das an beiden Enden Abschlusswiderstände trägt. Ein Rechner hängt an einem T-Stück, an dem der Strang mit BNC-Steckern angeschlossen ist.

2. Die *Sicherungsschicht* (Data Link Layer) besteht aus zwei Teilen. Der untere enthält die Kollisionsbehandlung, wenn sich mehrere Sender ge-

genseitig stören. Der obere kümmert sich um die Erkennung und Behebung von Übertragungsfehlern und die Flussregelung (Ausgleich der Geschwindigkeitsunterschiede zwischen Sender und Empfänger).

Zur Kollisionsbehandlung werden folgende Zugangsprotokolle eingesetzt:

- CSMA/CD (Carrier Sense Multiple Access with Collision Detection) bei Ethernet: Die Rechner sind durch einen Bus verbunden, den sie ständig abhören. Falls nichts los ist, dürfen sie senden. Wenn zwei Sender annähernd gleichzeitig beginnen, stören sie sich gegenseitig (Kollision). Sie brechen dann sofort ab und warten eine zufällig gewählte Zeiteinheit, bevor sie einen neuen Versuch starten. Das System wird durch den Ausfall eines angeschlossenen Rechners nicht gestört. Jedoch kann die Wartezeit nicht begrenzt werden und wird bei hoher Last tatsächlich sehr groß.
- Token-Ring: Die Rechner sind zu einem Ring verbunden. Sie hören alle Kommunikationen mit und prüfen, ob die Nachricht für sie bestimmt ist. Falls ja, wird sie abgespeichert, falls nein, wird sie einfach weitergereicht. Auf dem Ring kreist eine Berechtigungsmarke (Token). Ein Rechner kann sie entfernen und eine Empfängeradresse samt Botschaft senden. Daran hängt er eine neue Marke an. Dieses System wird mit hoher Last besser fertig, reagiert aber empfindlich auf den Ausfall eines Rechners oder Übertragungsfehler, die den Token betreffen.
- Token-Bus: Die Ringstruktur wird durch einen linearen Bus ersetzt, damit Rechnerausfälle besser toleriert werden können. Allerdings muss eine künstliche Reihenfolge hergestellt werden, um das Kreisen der Berechtigungsmarke zu simulieren.

Die Erkennung und Behebung von Übertragungsfehlern und die Flussregelung führen zur *Fenstertechnik* (Sliding Window): Der Datenstrom wird in Blöcke (Frames) aufgeteilt, die mit Kontrollinformation versehen werden. Der Empfänger bestätigt den Erhalt jedes Blocks. Ein fehlerhafter Block wird wiederholt. Damit der Sender nicht jede Quittung abwarten muss, wird am Anfang vereinbart, dass der Sender solange weitermachen darf, wie die unbestätigten Blöcke eine bestimmte Zahl (das Fenster) nicht überschreiten.

3. Die *Vermittlungsschicht* (Network Layer) übernimmt die Wegewahl (*Routing*) für die einzelnen Pakete. Bei einem Datagrammdienst ist nicht viel mehr zu tun. Für den Aufbau von virtuellen Verbindungen (wie z. B. beim Protokoll X.25) sind weitere Funktionen zu übernehmen: Die Wegewahl erfolgt beim Verbindungsaufbau. Falls die Reihenfolge einzelner Pakete durcheinandergerät, ist dies zu korrigieren.

Mehrere virtuelle Verbindungen können über den gleichen Weg laufen (Multiplexing). Daher müssen die Pakete beim Empfänger auseinandersortiert werden. Außerdem ist eine eigene Flussregelung zu treffen.

Die Wegewahl kann statisch erfolgen: Für jedes Ziel steht in einer Tabelle der Weg. Besser ist aber ein dynamisches Verfahren. Das kann entweder zentral oder verteilt gesteuert werden. Eine zentrale Steuerung ist häufig weder möglich noch wünschenswert. Also bleibt nur eine Planung, die sich an lokalen Gegebenheiten (z. B. Länge einzelner Warteschlangen) orientiert. Dabei besteht die Gefahr, dass ein Paket in Kreis herum geschickt wird (Looping); um das zu verhindern, müssen alle passierten Zwischenstationen notiert werden.

4. Die *Transportschicht* (Transport Layer) ist die Ebene, auf der sich die Interprozesskommunikation abspielt. Die Benutzerdaten müssen an die konkreten Maschinengegebenheiten angepasst werden, u. a. sind sie ggf. in Pakete gewisser Größe aufzuteilen. Man unterscheidet verschiedene Qualitätsklassen der Dienste, auf die sich beide Partner zu Beginn der Kommunikation einigen müssen. Diese Schicht kann z. B. einen unterschiedlichen Umfang an Fehlererkennung und -behebung (durch zusätzliche Kontrollinformation) bieten. Sie kann auch mehrere parallele Verbindungen zur Leistungssteigerung aufbauen. Zu den Diensten zählen virtuelle Verbindungen (für die einiges zu tun ist, wenn sie auf einen Datagrammdienst in Ebene 3 aufsetzen), Datagramme und Broadcasting.

5. Die *Kommunikationssteuerung* (Session Layer) kümmert sich um die Synchronisation der Partner und steuert ihren Dialog, so dass z. B. beide abwechselnd senden und empfangen. Sie kann eine Kommunikation in logische Abschnitte unterteilen, indem sie Sicherungspunkte setzt, auf die bei einem Fehler zurückgegangen werden kann. Verschiedene Funktionseinheiten werden genormt, über die sich die Partner zu Beginn einer Kommunikation verständigen müssen.

6. Die *Darstellungsschicht* (Presentation Layer) ermöglicht das Aushandeln einer Übertragungssprache. Hierzu gehören die rechnerunabhängige Datenrepräsentation, Datenkompression und Verschlüsselung.

7. Die *Anwendungsschicht* (Application Layer) befasst sich mit konkreten Programmen. Bestimmte Funktionen, wie Terminalsteuerung, Datenübertragung, elekronische Post, sind so häufig, dass es sich lohnt, sie zu normen.

Die drei unteren Schichten dienen vornehmlich der Kommunikation zwischen benachbarten Netzknoten, während ab Schicht 4 die Endpunkte angesprochen sind. Viele Standards stammen von der CCITT (Comité Consultatif

International de Télégraphique et Téléphonique). Schicht 4 wird meist benutzt; die bekannten Protokolle TCP (Transmission Control Protocol), ein verbindungsorientiertes Protokoll, und UDP (User Datagram Protocol), ein Datagrammdienst, gehören hierher. Beide zählen zur Internet Protocol Suite des DARPA (Defense Advanced Research Project), zu der auch IP (Internet Protocol), ICMP (Internet Control Message Protocol) und [R]ARP ([Reverse] Address Resolution Protocol) gehören. Die zuletzt genannten Protokolle werden meist zur Schicht 3 gerechnet, obwohl z. B. IP keinerlei Fehlerbehandlung durchführt. Zu den drei oberen Schichten zählen Sun-RPC in Schicht 5, Sun-XDR in Schicht 6 und die Programme telnet, ftp, mail, rlogin, rsh, rcp etc. in Schicht 7. Eigentlich würde man eher sagen, dass RPC auf XDR aufbaut und nicht umgekehrt. Die Programme telnet, ftp usw. dagegen lassen die Schichten 5 und 6 ganz aus.

Das Referenzmodell erfasst die gängigen Protokolle nicht sehr gut. Ein weiteres Problem ist, dass es von einer Unterbrechungssteuerung ausgeht. Z. B. durchläuft ein bestätigter Dienst die Stationen Request (Ausgangspunkt), Indication (entsprechende Unterbrechung beim Partner), Response (Bestätigung), Confirm (entsprechende Unterbrechung zum Abschluss). Die Schichtstruktur führt nicht unbedingt zu einer effizienten Implementierung.

6.2 Interprozesskommunikation (in UNIX)

Prozesse besitzen keine gemeinsamen Variablen, so dass sie nicht unmittelbar Daten austauschen können. Wenn sie außerdem auf verschiedenen Rechnern laufen, können sie möglicherweise nicht einmal auf gemeinsame Dateien zugreifen. UNIX bietet, historisch bedingt, mehrere unterschiedlich mächtige Mechanismen zur Interprozesskommunikation (IPC), die durch Systemaufrufe erreichbar sind.

6.2.1 Pipes

Pipes sind Puffer einer bestimmten systemabhängigen Größe. Man kann auf sie schreiben und von ihnen lesen wie von einer Datei. Das Betriebssystem übernimmt die Synchronisation der beteiligten Prozesse durch Blockierung. *pipe* erzeugt eine namenlose Pipe, die nur dem aufrufenden Prozess bekannt ist. Dieser kann sie aber bei der Prozesserzeugung an seine Kinder vererben. *mkfifo* erzeugt eine Pipe, die unter einem Dateinamen ansprechbar ist. Abbildung 6.2 zeigt, ganz analog zur Ein-/Ausgabeumlenkung in den Abbildungen 5.1 und 5.2 auf den Seiten 45 und 46, wie man einen Kindprozess erzeugt, der durch eine Pipe verbunden ist.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

// Kommando: prog1 | prog2
void pipeline(const char *prog1, const char *prog2)
{
    int fd[2];
    if (pipe(fd) < 0) {
        perror("cannot create pipe");
        return;
    }
    switch (fork()) {
    case -1:
        perror("cannot fork");
        break;
    case 0: // child
        close(fd[1]);
        close(STDIN_FILENO);
        dup(fd[0]); // dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        execl(prog2, prog2, NULL);
        perror("cannot exec");
        exit(EXIT_FAILURE);
        break;
    default: // parent
        close(fd[0]);
        close(STDOUT_FILENO);
        dup(fd[1]); // dup2(fd[1], STDOUT_FILENO);
        close(fd[1]);
        execl(prog1, prog1, NULL);
        perror("cannot exec");
        exit(EXIT_FAILURE);
    }
}
```

Abbildung 6.2: Erzeugung eines Kindprozesses mit Verbindungs-Pipe

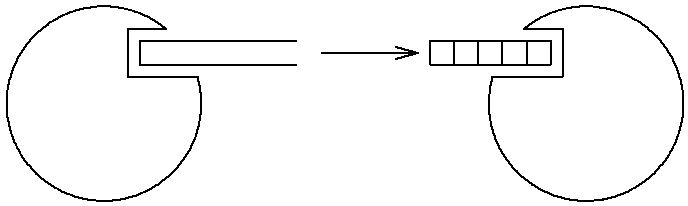
6.2.2 Sockets

Sockets (Steckdosen, Buchsen) sind das Konzept, das BSD-UNIX zur Interprozesskommunikation anbietet. Sie verallgemeinern Pipes und werden wie diese als Dateien angesprochen. Allerdings lassen sie den Datentransport in beide Richtungen zu.

socket erzeugt einen Socket. Damit dieser von einem Partner angesprochen werden kann, muss ihm ein Briefkasten (*Port*) zugeordnet werden (*bind*); das ist eine Nummer, die in Grenzen frei wählbar ist. Weiterhin muss eine Warteschlange fester Länge für die Nachrichten an den Socket eingerichtet werden (*listen*). Der Partner ruft dann *connect* mit seinem Socket und der Adresse (einschließlich Portnummer) auf, um eine Verbindung aufzubauen. Diese Anfrage landet aber zunächst in der Warteschlange, aus der sie erst geholt wird, wenn *accept* ausgeführt wird. *accept* liefert einen neuen Socket, über den die Kommunikation abgewickelt wird. Den typischen Ablauf bei der Kommunikation zwischen Client und Server zeigt Abbildung 6.3.

Der Server ruft `bind` und `listen` auf:

Der Client ruft `connect` auf:



Der Server ruft `accept` auf:

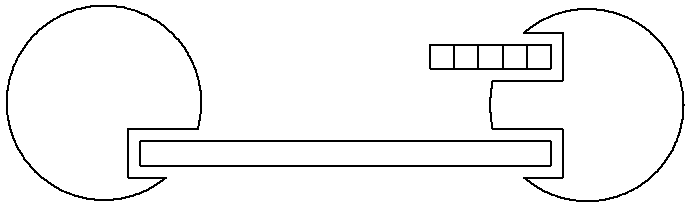


Abbildung 6.3: Herstellung der Verbindung zwischen zwei Stream-Sockets

Es gibt zwei Arten von Verbindungen zwischen Sockets: Einen Bytestrom

oder eine Folge von Paketen. Im ersten Fall können Daten ganz normal mit den Systemaufrufen *read* und *write* übertragen werden (siehe Dateisystem). Alternativ dazu sind *recv* und *send* möglich. Die Pakete im zweiten Fall werden mit den Systemaufrufen *sendto* und *recvfrom* verschickt. Jedes Paket wird individuell gesendet; der Empfänger muss damit rechnen, dass Pakete verloren gehen, in der falschen Reihenfolge ankommen oder verdoppelt werden.

Die Abbildungen 6.4, 6.5, 6.6, 6.7, 6.8 und 6.9 zeigen Stream-Sockets (TCP/IP) an einem Beispiel. Mit Datagram-Sockets (UDP/IP) wäre alles komplizierter, obwohl Client und Server nur jeweils ein Paket austauschen. Man müsste u. a. Timeouts programmieren.

```
#define PORT 10001

#define SERVICE "uname"

extern void service(int);
```

Abbildung 6.4: Gemeinsame Vereinbarungen (Server und Client)

Die Datei `uname.h` in Abbildung 6.4 wird sowohl vom Server als auch vom Client verwendet. Der Server besteht aus zwei Dateien. Die erste (Abbildungen 6.5 und 6.6) richtet den Socket ein und folgt dem in Abbildung 6.3 auf Seite 65 skizzierten Vorgehen. Zwei Punkte kommen für `bind` hinzu. Einerseits würde, falls keine Portnummer explizit definiert wurde, nachgesehen, ob ein Standardport für diesen Dienst existiert. Andererseits wirkt die Bestimmung der Socketadresse etwas abschreckend, weil Sockets mit verschiedenen Adressen arbeiten können, hier aber konkrete Internetadressen einzusetzen sind. Von `accept` erhält man, wie man sieht, neben dem zusätzlichen Socket auch die Adresse des Clients. Nebenbei erkennt man, dass die Byteordnung bei Binärzahlen ein Problem sein kann. `htons` und `htonl` wandeln **short**- bzw. **long**-Zahlen in ein Format um, auf das man sich geeinigt hat. `ntohs` und `ntohl` wandeln dies wieder zurück.

In der zweiten Datei für den Server (Abbildung 6.7) wickelt die Funktion `service` die Kommunikation mit dem Client ab, sobald der Socket dafür erzeugt wurde. Im vorliegenden Fall ermittelt sie Informationen über den Rechner (entsprechend zum Kommando `uname -a`) und verpackt sie in eine Zeichenfolge.

Der Client (Abbildungen 6.8 und 6.9) wird mit dem Namen des Rechners aufgerufen, auf dem der Dienst läuft. Die erste Aufgabe, nach der Überprüfung der Parameterzahl, besteht entsprechend darin, diesen Namen in eine numerische Internetadresse umzuwandeln. Anschließend wird, genau wie beim Server, die Portnummer hinzugefügt. Dann wird entsprechend der Abbildung 6.3 auf Seite 65 der Socket eingerichtet. Noch in der gleichen Datei

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include ünames.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BACKLOG 1024

int main(unsigned argc, char *argv[])
{
#ifndef PORT
    struct servent *sp = getservbyname(SERVICE, "tcp");
    if (sp == NULL) {
        fprintf(stderr, "%s: %s not found in /etc/services\n", argv[0], SERVICE);
        return EXIT_FAILURE;
    }
#endif

    struct sockaddr_in myaddr_in;
    memset(&myaddr_in, 0, sizeof myaddr_in);
    myaddr_in.sin_family = AF_INET;
    myaddr_in.sin_addr.s_addr = htonl(INADDR_ANY);
#ifdef PORT
    myaddr_in.sin_port = htons(PORT);
#else
    myaddr_in.sin_port = sp->s_port;
#endif

    int ls = socket(AF_INET, SOCK_STREAM, 0);
    if (ls < 0) {
        perror("socket");
        return EXIT_FAILURE;
    }

    if (bind(ls, (struct sockaddr *)&myaddr_in, sizeof myaddr_in) < 0) {
        perror("bind");
        return EXIT_FAILURE;
    }
```

Abbildung 6.5: Anmeldung des Dienstes, Teil 1 (Server)

```
    if (listen(ls, BACKLOG) < 0) {
        perror("listen");
        return EXIT_FAILURE;
    }

    for (; ; ) {
        struct sockaddr_in peeraddr_in;
        socklen_t addrlen = sizeof peeraddr_in;
        int s = accept(ls, (struct sockaddr *)&peeraddr_in, &addrlen);
        if (s < 0) {
            perror("accept");
            return EXIT_FAILURE;
        }
        service(s);
        close(s);
    }
}
```

Abbildung 6.6: Anmeldung des Dienstes, Teil 2 (Server)

werden die über den Socket empfangenen Daten in die Ausgabe kopiert.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/utsname.h>
#include ünameh"
#include <stdio.h>
#include <string.h>

void service(int s)
{
    struct utsname buffer;
    if (uname(&buffer) < 0) {
        perror("uname");
        return;
    }
    write(s, buffer.machine, strlen(buffer.machine));
    write(s, " ", 1);
    write(s, buffer.nodename, strlen(buffer.nodename));
    write(s, " ", 1);
    write(s, buffer.release, strlen(buffer.release));
    write(s, " ", 1);
    write(s, buffer.sysname, strlen(buffer.sysname));
    write(s, " ", 1);
    write(s, buffer.version, strlen(buffer.version));
    write(s, "\n", 1);
}
```

Abbildung 6.7: Dienst (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include üname.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFERSIZE 1024

int main(unsigned argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: %s host\n", argv[0]);
        return EXIT_FAILURE;
    }

    struct hostent *hp = gethostbyname(argv[1]);
    if (hp == NULL) {
        herror(argv[1]);
        fprintf(stderr, "%s: "s not found in /etc/hosts\n", argv[0], argv[1]);
        return EXIT_FAILURE;
    }

#ifndef PORT
    struct servent *sp = getservbyname(SERVICE, "tcp");
    if (sp == NULL) {
        fprintf(stderr, "%s: %s not found in /etc/services\n", argv[0], SERVICE);
        return EXIT_FAILURE;
    }
#endif
```

Abbildung 6.8: Aufruf des Dienstes, Teil 1 (Client)

```
    struct sockaddr_in peeraddr_in;
    memset(&peeraddr_in, 0, sizeof peeraddr_in);
    peeraddr_in.sin_family = AF_INET;
    peeraddr_in.sin_addr.s_addr = ((struct in_addr *)hp->h_addr)->s_addr;
#ifdef PORT
    peeraddr_in.sin_port = htons(PORT);
#else
    peeraddr_in.sin_port = sp->s_port;
#endif

    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("socket");
        return EXIT_FAILURE;
    }

    if (connect(s, (struct sockaddr *)&peeraddr_in, sizeof peeraddr_in) < 0) {
        perror("connect");
        return EXIT_FAILURE;
    }

    for (;;) {
        char buffer[BUFFERSIZE];
        ssize_t n = read(s, buffer, BUFFERSIZE);
        if (n < 0) {
            perror("read");
            return EXIT_FAILURE;
        }
        if (n == 0)
            break;
        fwrite(buffer, 1, n, stdout);
    }

    close(s);
    return EXIT_SUCCESS;
}
```

Abbildung 6.9: Aufruf des Dienstes, Teil 2 (Client)

6.2.3 Prozedur-Fernaufruf

Der *Prozedur-Fernaufruf* (*Remote Procedure Call*, RPC) erlaubt es einem Prozess, eine Prozedur, die ein anderer Prozess zur Verfügung stellt, so aufzurufen, als ob sie lokal wäre. Das Prinzip zeigt Abbildung 6.10.

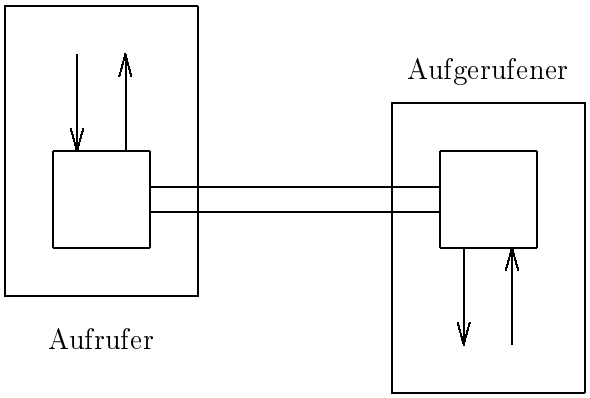


Abbildung 6.10: Funktionsweise des Prozedur-Fernaufrufs

Der aufrufende Prozess enthält einen *Prozedurstummel*, den er stellvertretend für die ferne Prozedur aufruft, und von dem er schließlich die Ergebnisse bekommt. Auch der aufgerufene Prozess enthält einen Prozedurstummel, der die gesuchte Prozedur, die vorher fern war, durch einen lokalen Prozeduraufruf erreichen kann. Nur die beiden Prozedurstummel müssen über den Transport der Daten Bescheid wissen, alle anderen Programmteile merken nichts davon. Zum Datentransport werden in der Regel Sockets verwendet.

Zur Herstellung der Verbindung muss der aufrufende Prozess natürlich den Rechner identifizieren, auf dem der aufzurufende Prozess läuft. Das geschieht durch weltweit eindeutige 4 Byte-*Internetadressen*. Die meisten Rechner besitzen aber auch symbolische Namen.

Der nächste Schritt besteht in der Identifikation des anzusprechenden Prozesses. Seine Prozessnummer ist dazu nicht geeignet, da sie zu häufig wechselt. Daher werden den Prozessen, die Dienste anbieten, wiederum Ports zugeordnet. Bei den Sun-RPCs verwaltet ein spezieller Prozess, der *Portmapper*, die Zuordnungen zwischen Ports und laufenden Prozessen. Dieser Portmapper belegt einen festen Port mit der Nummer 111. Den Verbindungsaufbau zeigt Abbildung 6.11.

Zuerst lässt sich der Server beim Portmapper registrieren (1). Irgendwann später möchte ein Client den angemeldeten Dienst nutzen. Er richtet zunächst einen Prozedur-Fernaufruf an den Portmapper (2), um sich den Port des Servers geben zu lassen. Mit dieser Portnummer kann er sich nun an den Server wenden (3). Der letzte Schritt ist schließlich der Aufruf der gewünschten Prozedur. Prozeduren sind bei den Sun-RPCs durch eine

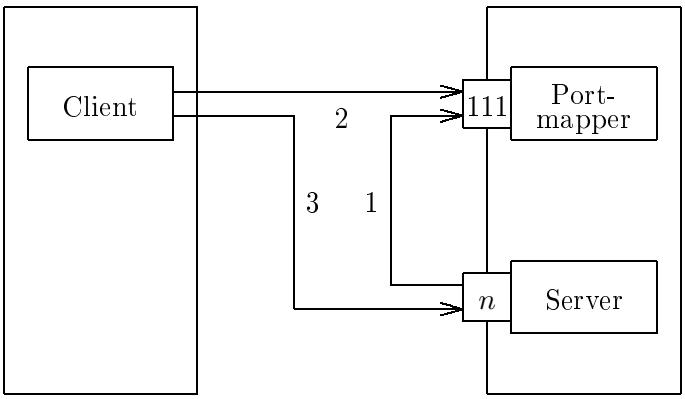


Abbildung 6.11: Typischer Ablauf beim Aufruf eines Dienstes

Programm-, Versions- und Prozedurnummer gekennzeichnet.

Ein besonderes Problem beim Austausch von Binärdaten zwischen verschiedenen Rechnern liegt darin, dass die Formate verschieden sind. Daher muss der Sender seine Daten in ein rechnerunabhängiges Format (auf das man sich geeinigt hat) bringen, das der Empfänger in sein Format zurückverwandeln muss. Ein anerkanntes Format ist Sun-XDR (eXternal Data Representation). Selbst wenn die Hardware von Sender und Empfänger gleich ist, müssen möglicherweise beide ihre Daten konvertieren.

Kapitel 7

Threads und Synchronisation

Prozesse vereinigen zwei Konzepte: Die Ablaufkontrolle (Programmzähler) und den Zugriff auf Ressourcen (Adressraum, Dateien). Falls eine hohe Leistung erforderlich ist, lohnt es sich meist, beide Konzepte zu trennen: Threads benutzen die Ressourcen gemeinsam, besitzen aber jeweils eine eigene Ablaufkontrolle. Sie sind daher schneller zu erzeugen, zu wechseln und zu zerstören; außerdem können sie miteinander einfacher kommunizieren. Ein Prozess enthält einen oder mehrere Threads; er endet, sobald alle Threads beendet sind.

Andererseits entstehen beim Zugriff auf gemeinsame Ressourcen Wettlaufbedingungen. Das macht eine Synchronisation (Zugangskontrolle) erforderlich. Die damit verbundenen Probleme sind nicht auf Threads beschränkt, sondern viel allgemeiner. Sie entstehen auch u. a. im Betriebssystemkern, der sogar auf einem einzigen Prozessor mehrere Aufgaben quasi-gleichzeitig bearbeiten kann (Unterbrechungen), oder wenn Prozesse Daten austauschen und Ressourcen gemeinsam nutzen, womit nicht nur Dateien, sondern auch gemeinsame Speicherbereiche gemeint sein können. Daher müssen sowohl Betriebssystementwickler die Probleme lösen und geeignete Mechanismen bereitstellen als auch Anwendungsprogrammierer die Probleme verstehen und die Mechanismen einsetzen. Synchronisationsprobleme werden theoretisch untersucht.

7.1 Threads

Threads sind eine „leichtgewichtige“ Alternative zu Prozessen: Sie laufen innerhalb eines Prozesses und benutzen den gleichen Adressraum und die gleichen Dateien. Bei der Umschaltung werden, wie bei Prozessen, die Registerinhalte gesichert, aber es ist keine aufwändige Kontextumschaltung nötig. Jeder Thread besitzt allerdings seinen eigenen Laufzeitkeller und (das gilt konkret für POSIX-Threads) seine eigene Variable *errno*.

Normalerweise startet jeder Prozess einen Thread – denjenigen, der die Arbeit macht. Weitere Threads können explizit erzeugt werden. Eine ty-

pische Anwendung ist ein Server-Thread, der einen von `accept` gelieferten Socket bedient. Im vorangegangenen Beispiel für Stream-Sockets (TCP/IP) wurden die Clients der Reihe nach bedient. Falls es sich um mehr als eine billige Auskunft handelt, ist das unzumutbar. Auch der Server ist nicht gut ausgelastet, wenn er zeitweise blockiert wird. Ein paar kleine Änderungen am Server genügen, um ihn nebenläufig zu machen.

Unglücklicherweise enthält die Datei `uname.h` die Deklaration der Funktion service, die sich ändert. Abbildung 7.1 zeigt die neuen Vereinbarungen (vergleiche Abbildung 6.4 auf Seite 66).

```
...

extern void *service(void *);
```

Abbildung 7.1: Gemeinsame Vereinbarungen (Server und Client)

Dem entsprechen die Änderungen an der Funktion service in Abbildung 7.2 (vergleiche Abbildung 6.7 auf Seite 69). (Außerdem heißt es einmal **return** NULL; statt **return**;).

```
...
#include <stdlib.h>

void *service(void *arg)
{
    int s = *(int *)arg;
    free(arg);
    ...
    close(s);
    return NULL;
}
```

Abbildung 7.2: Dienst (Server)

In der Endlosschleife nach der Anmeldung des Dienstes (vergleiche Abbildung 6.6 auf Seite 68) werden die Threads erzeugt, siehe Abbildung 7.3. (Der Socket s wird jetzt in der Funktion service geschlossen.) Der Server braucht nun die Bibliothek `pthread`.

Das gleichzeitige Warten auf mehrere Ereignisse ist mit Threads meistens viel einfacher zu programmieren als mit synchronem Multiplexen (`select`)-

Es folgen die wichtigsten Funktionen für Threads. *pthread_ create* startet einen neuen Thread. Übergeben werden vor allem die auszuführende Funktion und ein Parameter. Der Thread endet, sobald die Funktion ein Ergebnis

```
...
#include <pthread.h>

    ...
    pthread_t tid;
    int *arg = (int *)malloc(sizeof *arg);
    *arg = s;
    pthread_create(&tid, NULL, service, arg);
    ...
```

Abbildung 7.3: Bearbeitung der Verbindungen (Server)

liefert. Durch *pthread_exit* kann ein Thread vorzeitig beendet werden. Ein anderer Thread kann mit *pthread_join* auf das Ergebnis eines bestimmten Threads warten. *pthread_self* liefert die Thread-ID des laufenden Threads. Diese Funktionen entsprechen *fork*, *_exit*, *waitpid* bzw. *getpid* bei Prozessen. *pthread_detach* verwandelt einen Thread in einen Daemon (vergleichbar mit einem Hintergrundprozess). Auf ihn kann man nicht mehr mit *pthread_join* warten.

Bei der Verwendung von Threads sind gemeinsame Variablen problematisch, denn sie überschreiben sich gegenseitig deren Werte. Das erste Ziel ist die Beseitigung globaler (**static**) Variablen. Der typische Ablauf (ohne Fehlerbehandlung) sieht so aus:

Sobald man mit Threads programmiert, interessiert man sich dafür, ob die Funktionen der C-Bibliothek thread-sicher sind. Einige, die statische Variablen verwenden, wie z. B. der Zufallszahlengenerator rand, sind es nicht. Meist gibt es einen Ersatz mit dem Suffix _r, z. B. rand_r.

7.2 Wettlaufbedingungen und kritische Abschnitte

Wenn mehrere Threads (Prozesse etc.) auf gemeinsame Daten zugreifen, kann das Ergebnis von der Abarbeitungsreihenfolge abhängen (*Wettlaufbedingung*). Es gibt Bereiche in jedem Thread (*kritische Abschnitte*), die nicht von mehreren Threads gleichzeitig durchlaufen werden dürfen, weil sonst inkonsistente Zustände auftreten.[1] *Synchronisationsverfahren* sollen solche Fehler verhindern. Folgende Anforderungen sind zu stellen:

- *Gegenseitiger Ausschluss* (zwei Threads dürfen sich nicht gleichzeitig in ihren kritischen Abschnitten befinden) muss gesichert sein.
- Die relativen Geschwindigkeiten der Threads dürfen keine Rolle spielen.

[1] Vergleiche Transaktionen in Datenbanken.

```
static pthread_key_t key;
static pthread_once_t once_var = PTHREAD_ONCE_INIT;

void destructor(void *value)
{
    free(value);
}

void once_proc(void)
{
    pthread_key_create(&key, destructor);
}

void *thread_func(void *ptr)
{
    ...
    pthread_once(&once_var, once_proc);
    void *value = pthread_getspecific(key);
    if (value == NULL) {
        value = malloc(...);
        pthread_setspecific(key, value);
    }
    ...
}
```

Abbildung 7.4: Jeder Thread legt unter dem gleichen Schlüssel seinen speziellen Wert ab

- Ein Thread außerhalb seines kritischen Abschnitts darf andere Threads nicht am Betreten ihres kritischen Abschnitts hindern.
- Kein Thread „verhungert“ (darf nie in seinen kritischen Abschnitt).

Der Betriebssystemkern auf einer Einprozessormaschine kann in solchen Fällen einfach alle Unterbrechungen sperren. Er kann dann nicht mehr gestört werden. Für Benutzerprozesse kommt das allerdings nicht in Frage, da die Sperrung kurz sein soll und zuverlässig wieder aufgehoben werden muss.

Trotzdem brauchen Synchronisationsverfahren die Unterstützung des Betriebssystems. Beispielsweise funktioniert folgendes Verfahren *nicht*: Jeder Thread zeigt an, ob er in seinem kritischen Abschnitt ist. Wir stellen uns bildlich vor, dass er am Eingang zu seinem kritischen Abschnitt eine Fahne hisst und am Ausgang wieder einholt. Bevor er seinen kritischen Abschnitt betritt, prüft er die Fahnen aller Konkurrenten. Sieht er eine gehisste Fahne, geht er in eine Warteschleife, bis er keine Fahne mehr sieht. Dann hisst er

seine eigene Fahne und betritt seinen kritischen Abschnitt.

Erstens kostet das ständige Nachsehen, ob eine Fahne gehisst ist, Rechenzeit, die man sinnvoller verwenden könnte. Das nennt man *aktives Warten*. Ein zusätzliches Problem damit ist, dass das Betriebssystem eine Warteschleife nicht von sinnvoller Tätigkeit unterscheiden kann. So würde es ohne Skrupel einen Thread/Prozess, der sich gerade in seinem kritischen Abschnitt befindet, durch einen Thread/Prozess mit höherer Priorität verdrängen lassen. Letzterer kommt nicht aus seiner Warteschleife heraus, weil er seinen kritischen Abschnitt nicht betreten darf. Es liegt eine Verklemmung vor. Zweitens vergeht zwischen dem Nachsehen und dem Hissen der eigenen Fahne eine kleine Zeitspanne, in der ein zweiter Thread das Gleiche tun könnte. Der gegenseitige Ausschluss ist nicht gewährleistet. Übrigens geht es auch nicht, den Threads reihum den Zutritt zu ihren kritischen Abschnitten zu gewähren, denn das verstößt gegen die Anforderung, dass kein Thread außerhalb seines kritischen Abschnitts andere Threads am Betreten ihres kritischen Abschnitts hindern darf.

Synchronisationsverfahren haben daher zwei weitere Eigenschaften:

- Sie versetzen wartende Threads/Prozesse in den Zustand *blockiert*. Das Warten auf eine Eingabe beispielsweise sieht daher genauso aus wie das Warten am Eingang eines kritischen Abschnitts.

- Sie benutzen unteilbare Operationen wie z. B. Fahnen prüfen und hissen im obigen Beispiel. (Das bedeutet, dass in der Zeit, in der ein Thread prüft, ob er seine Fahne hissen darf, alle anderen Threads seine Entscheidung abwarten.) Ist kein geeigneter Maschinenbefehl vorhanden, muss das Betriebssystem entsprechende Systemaufrufe zur Verfügung stellen, während derer alle Unterbrechungen gesperrt sind.

Es folgen die bekanntesten Hilfsmittel. Es genügt, wenn eines davon verfügbar ist. (Wenn man will, kann man alle anderen damit simulieren.) Wir betrachten folgendes Beispiel: Ein Erzeuger produziert Gegenstände, die ein Verbraucher konsumiert. Zur Lagerung dient ein Puffer mit beschränkter Kapazität N. Der Erzeuger blockiert, wenn der Puffer voll ist; der Verbraucher blockiert, wenn der Puffer leer ist. Die Lösungsvorschläge werden in einem frei erfundenen Pseudokode gegeben.

Betrachten wir zunächst wieder einen Versuch, der *nicht* funktioniert: Eine Variable `count` zählt den Füllstand des Puffers. Der Erzeuger blockiert sich im Falle $\texttt{count} = N$ selbst, andernfalls produziert er einen Gegenstand, legt ihn im Puffer ab und erhöht `count` um Eins. Falls $\texttt{count} = 1$ geworden ist, schickt er dem Verbraucher ein Wecksignal – für alle Fälle, falls dieser gerade blockiert ist. Analog blockiert sich der Verbraucher im Falle $\texttt{count} = 0$ selbst, andernfalls holt er einen Gegenstand aus dem Puffer, erniedrigt `count` um Eins und konsumiert ihn. Falls $\texttt{count} = N - 1$ geworden ist, schickt er dem Erzeuger ein Wecksignal. Hier gibt es (unter anderen) eine schwerer zu

sehende Wettlaufbedingung: Wenn ein Thread beschließt sich zu blockieren, aber nicht dazu kommt, weil sich der andere Thread dazwischendrängt, dann kann dessen Wecksignal verlorengehen. Beide Threads werden sich schließlich blockieren und nicht mehr aufwachen.

7.2.1 Semaphore

Semaphore sind Variablen, die nichtnegative Werte annehmen (um Wecksignale zu zählen) und für die es zwei unteilbare Operationen `P` und `V` gibt:

$$\mathrm{P}(S)\colon \begin{cases} S := S - 1, & S > 0 \\ \text{aufrufenden Prozess in Warteliste eintragen}, & S = 0 \end{cases}$$

$$\mathrm{V}(S)\colon \begin{cases} \text{einen Prozess aus der Warteliste aufwecken}, & \text{Warteliste nicht leer} \\ S := S + 1, & \text{Warteliste leer} \end{cases}$$

Semaphore sind leicht zu implementieren. Am häufigsten benutzt werden binäre Semaphore, die nur die Werte 0 oder 1 annehmen.

Das Beispiel von Erzeuger/Verbraucher mit endlichem Puffer sieht so aus: Zur Synchronisation dienen drei Semaphore. `mutex` gewährleistet den gegenseitigen Ausschluss, denn nur ein Aufruf von `P` blockiert nicht. `full` zählt die belegten Plätze im Puffer, `empty` die freien. Statt die Semaphore zu initialisieren, könnte man sie bei 0 anfangen lassen und entsprechend oft `V` aufrufen.

Semaphore mutex $= 1$, full $= 0$, empty $= N$;

Der Erzeuger durchläuft folgende Endlosschleife:

```
while (1) {
    ProduceItem(item);
    P(empty);
    P(mutex);
    EnterItem(item);
    V(mutex);
    V(full);
}
```

Der Verbraucher durchläuft entsprechend folgende Endlosschleife:

```
while (1) {
    P(full);
    P(mutex);
    RemoveItem(item);
    V(mutex);
    V(empty);
    ConsumeItem(item);
}
```

Semaphore erlauben, Unterbrechungen zu verbergen. Wenn man jedem Gerät einen Semaphor zuordnet, dann kann man eine Ein-/Ausgabeanforderung als entsprechende `P`-Operation auffassen. Die Unterbrechung, die die Erfüllung meldet, würde die zugehörige `V`-Operation ausführen.

7.2.2 Monitore

Das Programmieren mit Semaphoren ist fehlerträchtig, da die Anweisungen über das Programm verstreut sind. Die Reihenfolge von `P`-Operationen kann entscheidend sein; außerdem ist die Zuordnung zwischen `P`- und `V`-Operationen nicht ersichtlich. Semaphore haben das Niveau von Assembler.

Monitore sind ein Mittel in (einigen wenigen) Programmiersprachen.[2] Der Compiler muss für die richtige Synchronisation sorgen. Von den Operationen, die der Monitor zur Verfügung stellt, kann zu jeder Zeit immer nur eine aktiv sein. Um innerhalb des Monitors Blockierungen zu ermöglichen, gibt es einen Datentyp `Condition` für *Bedingungen* mit den Operationen `wait`[3] und `notify`. `wait` blockiert die laufende Operation und verlässt den Monitor. `notify` weckt eine vorher blockierte Operation wieder auf. Damit dadurch nicht zwei aktive Operationen entstehen, könnte man festlegen, dass `notify` nur am Ende einer Operation aufgerufen werden darf. `wait` und `notify` entsprechen den bereits erwähnten Blockierungen und Wecksignalen. Wecksignale können hier nicht verlorengehen, da der Monitor für den gegenseitigen Ausschluss sorgt.

Es folgt wieder das Beispiel von Erzeuger/Verbraucher mit endlichem Puffer.

```
monitor Buffer {
    Condition empty, full;
    int count = 0

    operation add(item) {
        if (count == N) wait(empty);
        EnterItem(item);
        count = count + 1;
        if (count == 1) notify(full);
    }

    operation remove(item) {
        if (count == 0) wait(full);
        RemoveItem(item);
        count = count - 1;
        if (count == N - 1) notify(empty);
    }
}
```

[2] Darunter auch Java; dort kann man jede Klasse als Monitor programmieren und jedes Objekt als Bedingung verwenden.

[3] nicht mit dem gleichnamigen UNIX-Systemaufruf zu verwechseln

Damit ist die gesamte Synchronisation in übersichtlicher Form erledigt. Die Prozeduren `add` und `remove` laufen unter gegenseitigem Ausschluss. Da die Variablen `empty`, `full` und `count` nur innerhalb des Monitors verfügbar sind, sind Wettlaufbedingungen ausgeschlossen. Stellt sich innerhalb einer Operation heraus, dass sie nicht zu Ende geführt werden kann, wird sie mit Hilfe einer Bedingung auf Eis gelegt.

Der Erzeuger durchläuft folgende Endlosschleife:

```
while (1) {
    ProduceItem(item);
    Buffer.add(item);
}
```

Der Verbraucher durchläuft folgende Endlosschleife:

```
while (1) {
    Buffer.remove(item);
    ConsumeItem(item);
}
```

Im Allgemeinen ist aber folgender Ablauf typisch: Gegeben sei eine boolesche Bedingung (oder *Invariante*) B, die zum Weitermachen erfüllt sein muss. Man führt eine entsprechende Bedingungsvariable b ein. Ein Thread, der es bis in den Monitor geschafft hat, begnügt sich nicht mit einer **if**-Abfrage, sondern wartet in folgender Schleife:

while $(!B)$
    wait(b);

Jeder andere Thread, der *möglicherweise* den Wert von B verändert hat, gibt *allen* wartenden Threads eine Chance:

notifyAll(b);

POSIX-Threads stellen die Typen `pthread_mutex_t` für binäre Semaphore (*mutex* steht für *mutual exclusion* – gegenseitiger Ausschluss) und `pthread_cond_t` für Bedingungen zur Verfügung. Ein binärer Semaphor kann mit `pthread_mutex_lock` gesperrt und mit `pthread_mutex_unlock` freigegeben werden. Ein Thread, der eine gesperrte Variable zu sperren versucht, wird blockiert. `pthread_mutex_trylock` ist wie `pthread_mutex_lock` , liefert aber einen Fehlercode (`EBUSY`) anstatt zu blockieren. Die Funktionen `pthread_cond_wait` , `pthread_cond_signal` und `pthread_cond_broadcast` entsprechen den obigen Funktionen `wait`, `notify` und `notifyAll` für Bedingungen. Außerdem gibt es eine Funktion `pthread_cond_timedwait` , die nur eine gewisse Maximalzeit wartet. Bedingungen darf man nur unter gegenseitigem Ausschluss benutzen, den man durch mutex-Variablen herstellt.

7.3 Klassische Synchronisationsprobleme

Die vorangegangenen, doch recht programmiertechnischen Beschreibungen sollten vor allem das Problembewusstsein schärfen. Neben dem betrachteten Beispiel von Erzeuger und Verbraucher mit endlichem Puffer gibt es mindestens zwei bekannte Probleme, die ein Synchronisationsmechanismus lösen können muss.

7.3.1 Die speisenden Philosophen

Fünf Philosophen sitzen an einem runden Tisch. Vor jedem steht ein Teller mit besonders glitschigen Spaghetti. Zwischen je zwei Tellern liegt eine Gabel. Die Philosophen sind in Gedanken versunken, aber von Zeit zu Zeit werden sie hungrig. Um ihre Spaghetti zu essen, brauchen sie allerdings zwei (!) Gabeln.[4] Wenn links oder rechts von ihrem Teller eine Gabel liegt, dürfen sie sie nehmen. Sobald sie beide Gabeln haben, essen sie eine Weile und legen dann die Gabeln zurück. Gesucht ist ein Algorithmus für jeden Philosophen, so dass keiner am Tisch verhungert.

Der erste Versuch wäre, jeden Philosophen zuerst nach seiner linken Gabel greifen zu lassen. Sobald er sie hat, greift er nach der rechten. Leider funktioniert es so nicht. Wenn sich alle Philosophen gleichzeitig ihre linke Gabel schnappen, geht es nicht weiter (eine Verklemmung). Folglich sollte ein Philosoph vielleicht seine linke Gabel wieder zurücklegen, wenn die rechte nicht verfügbar ist, und von vorn beginnen. Aber das könnte wiederum bei allen im Gleichtakt vor sich gehen (ein *Lifelock*). Man könnte das Aufnehmen der Gabeln, Essen und Ablegen der Gabeln als kritischen Abschnitt ansehen und unter gegenseitigem Ausschluss ablaufen lassen. Dann dürfte aber immer nur ein Philosoph essen, obwohl die Bestecke für zwei reichen würden. Also wird man das Aufnehmen und das Ablegen beider Gabeln jeweils als einen kritischen Abschnitt betrachten müssen.

```
enum { THINKING, HUNGRY, EATING } state[5] = {
    THINKING, THINKING, THINKING, THINKING, THINKING
};
Semaphore S[5] = { 0, 0, 0, 0, 0 };
Semaphore mutex = 1;

void Test(int i)
{
    if (state[i] == HUNGRY && state[(i - 1) % N] != EATING
                           && state[(i + 1) % N] != EATING) {
        state[i] = EATING;
```

[4] Eine modernere Variante würde die Situation treffender mit Stäbchen und chinesischen Gerichten beschreiben.

```
            V(S[i]);
        }
}

void TakeForks(int i)
{
        P(mutex);
        state[i] = HUNGRY;
        Test(i);
        V(mutex);
        P(S[i]);
}

void ReleaseForks(int i)
{
        P(mutex);
        state[i] = THINKING;
        Test((i - 1) % N);
        Test((i + 1) % N);
        V(mutex);
}

void Philosopher(int i)
{
        while (1) {
            Think();
            TakeForks(i);
            Eat();
            PutForks(i);
        }
}
```

Diese Lösung vermeidet alle bisher angesprochenen Probleme, bannt aber nicht die Gefahr einer Verschwörung: Vier Philosophen wechseln sich beim Essen ab, ohne den fünften, der schon lange hungrig ist, an die Reihe kommen zu lassen. Abhilfe könnte ein weiterer Zustand „sehr hungrig“ (neben „denkend“, „hungrig“ und „essend“) schaffen, in den man kommt, wenn beide Nachbarn bereits gegessen haben, während man hungrig war (am grimmigen Gesichtsausdruck zu erkennen). Ein hungriger Philosoph darf nur dann essen, wenn keiner seiner Nachbarn sehr hungrig ist.

7.3.2 Leser und Schreiber

In einer Datenbank dürfen mehrere Prozesse gleichzeitig den gleichen Satz lesen. Wenn aber ein Prozess einen Datensatz verändern will, dann müssen alle anderen Schreiber und sogar die Leser so lange ausgeschlossen werden.

Noch verhältnismäßig leicht wird man eine Lösung finden, in denen die Leser Vorrang haben.

```
Semaphore mutex = 1, db = 1;
int readers = 0;

void Reader()
{
    while (1) {
        P(mutex);
        readers = readers + 1;
        if (readers == 1) P(db);
        V(mutex);
        ReadDataBase();
        P(mutex);
        readers = readers − 1;
        if (readers == 0) V(db);
        V(mutex);
        UseDataRead();
    }
}

void Writer()
{
    while (1) {
        ThinkUpData();
        P(db);
        WriteDataBase();
        V(db);
    }
}
```

Die Leser können dabei einen Schreiber völlig aussperren, wenn ständig neue Leser ankommen. Deutlich komplizierter ist es mit den vorgestellten Synchronisationsmechanismen, einem Schreiber zum frühestmöglichen Zeitpunkt Zugriff zu gewähren.

Kapitel 8

Gerätesteuerungen

Das Betriebssystem soll eine abstrakte Maschine bieten, die bequem zu bedienen ist. Das ist keine einfache Aufgabe. Ein großer Teil des gesamten Betriebssystems steckt in den Gerätesteuerungen. Andererseits gibt es relativ wenige allgemeingültige Konzepte.

Die Oberfläche soll für den Benutzer so einheitlich und abstrakt wie möglich sein. *Geräteunabhängigkeit* und *einheitliche Namensgebung* werden durch Behandlung als Datei erreicht. Gerätesteuerungen sollen in Schichten aufgebaut werden, die jeweils nur die Dienste der darunter liegenden Schicht benutzen. Die Behandlung von Fehlern soll so nahe wie möglich an der Ursache erfolgen. Das Warten auf die abschließende Unterbrechung bei DMA soll wie eine Blockierung aussehen. Außerdem soll die Synchronisation mehrerer Zugriffe nach außen verborgen werden.

8.1 Blockorientierte Geräte (Festplatten)

Festplatten (Hard Disks, Winchester Disks) bestehen aus mehreren runden Scheiben, die auf einer Spindel sitzen und sehr schnell rotieren. Knapp über den Oberflächen, auf beiden Seiten jeder Scheibe, schweben Schreib-/Leseköpfe, die an einem gemeinsamen Arm befestigt sind. Der Arm wird von einem Linearmotor bewegt. Jede Stellung des Arms erlaubt den Zugriff auf die magnetische Oberfläche in einem bestimmten Radius von der Spindel. Pro Oberfläche (Kopf) ist das eine *Spur*, für alle Oberflächen (Köpfe) zusammen ein *Zylinder*. Jede Spur enthält eine Anzahl von *Sektoren*, und die wiederum eine feste Anzahl von Bytes.

Bei älteren Platten war die Anzahl der Sektoren noch unabhängig vom Radius. Neuerdings enthalten die äußeren Spuren mehr Sektoren als die inneren. Die wahre Geometrie wird praktisch vollständig vom Controller (siehe unten) verborgen. Die Schallplatten-Abstraktion aus dem Dateisystem-Kapitel mit ihrer fortlaufenden Folge von Sektoren wird recht gut aufrecht erhalten. Bemerkenswerterweise lassen sich Platten in mehrere logisch unabhängige Teile *partitionieren*; gerade dabei muss eine Geometrie angegeben

werden, die meist frei erfunden wird.

Bei Festplatten handelt es sich um magnetisch beschichtete Aluminium- bzw. Glasscheiben. Ihre Oberfläche muss extrem eben sein. Geraten kleinste Fremdkörper zwischen Kopf und Oberfläche, wird die Beschichtung schnell abgerieben. Der Abrieb wiederum kann zur Zerstörung aller anderen Scheiben führen. Disketten (Floppy Disks) dagegen tragen eine harte Beschichtung auf einer flexiblen Scheibe; der Kopf sitzt direkt auf der Oberfläche. Wegen der geringeren Genauigkeit sind sie billiger zu produzieren, nutzen sich aber ab und erreichen vergleichsweise geringe Geschwindigkeiten und Kapazitäten (zwischen 800 KB und 100 MB). Disketten können gewechselt werden, aber es gibt auch Wechselplatten.

CDs (Compact Discs) bestehen aus einer Oberfläche mit kleinen mechanischen Vertiefungen. Sie werden durch einen Laserstrahl und einen beweglichen Spiegel optisch abgetastet. Fällt der Strahl auf eine Vertiefung, wird er schwächer reflektiert. CDs liegen in Geschwindigkeit und Kapazität (650 MB) typischerweise zwischen Disketten und Festplatten.

8.1.1 Hardware

Controller sind „intelligente“ Schaltkreise, die die Steuerung der eigentlichen Geräte vereinfachen. Sie verstehen bereits relativ komplexe Kommandos wie Lesen, Schreiben, Positionieren, Formatieren und Rekalibrieren. Sie kennen auch eine Liste der fehlerhaften Stellen (Bad Spots) einer Festplatte, die sich aus Kostengründen bei der Herstellung nicht vermeiden lassen, und verwenden selbstständig Ersatz. Sie enthalten einige Register, über die die *Gerätesteuerung* (Device Driver) mit ihnen kommunizieren kann. Die Register werden häufig auf Hauptspeicheradressen *abgebildet*. Controller puffern die Daten, die in konstanter Geschwindigkeit vom/zum Gerät übertragen werden müssen, testen die Prüfsumme, versuchen ggf. eine Wiederholung, sind häufig zu DMA fähig und lösen Unterbrechungen aus. Dass Sektoren auf der Platte übersprungen werden müssen (Interleaving), weil sonst die Datenübertragungsrate zum Hauptspeicher nicht ausreichen würde, sollte der Vergangenheit angehören. Im Gegenteil werden häufig ganze Spuren im Cache abgelegt. (Das Warten auf einen Sektor dauert im Durchschnitt sowieso eine halbe Umdrehung.) Ein Controller kann normalerweise mehrere Geräte steuern (bis zu acht *Logical Units* bei SCSI). Neuerdings sitzen die Controller aber direkt auf den Geräten. Dies erhöht die Datenübertragungsraten.

IDE (Intelligent Device Electronic), auch ATA (AT-Attachment) oder AT-Bus genannt, ist heutzutage die typische Billiglösung. Bis zu zwei IDE-Controller können über einen Host-Adapter an den Systembus angeschlossen werden. Enhanced-IDE soll mehr Leistung bringen und zusätzliche Geräte (Wechselplatten) unterstützen. SCSI II (Small Computer Systems Interface) ist deutlich leistungsfähiger. Bis zu acht SCSI-Controller können an einem Bus (der an beiden Enden einen Abschlusswiderstand tragen muss) ange-

schlossen sein. Ein beliebiges Paar davon kann Daten austauschen. Normalerweise wird der Host-Adapter beteiligt sein, der selbst einen der Controller-Plätze einnimmt. Fast-, Wide- und Ultra-SCSI sind in Gebrauch.

8.1.2 Scheduling des Plattenarms

Festplatten, Disketten und CDs sind *blockorientierte* Geräte. Da *Suchzeit* (Bewegung des Arms/Lichtstrahls zum Zylinder) und *Latenzzeit* (Rotation zum Sektor) groß gegenüber der Übertragungszeit sind, wäre es viel zu aufwändig, einzelne Bytes zu adressieren. Liegen mehrere Anforderungen (von verschiedenen Prozessen) vor, kann man eine Auswahl treffen. Die einfachste Strategie ist, die Anforderungen in der Reihenfolge des Eintreffens zu bearbeiten (*First-Come First-Served*). Einen besseren Durchsatz erhält man allerdings, wenn man von allen vorliegenden Anforderungen immer diejenige auswählt, deren Zylinder am nächsten an der momentanen Position liegt (*Shortest-Seek-First*). Der Nachteil ist nur, dass der Kopf dann die Tendenz bekommt, sich in der Mitte aufzuhalten, und außen liegende Anforderungen lange verzögert werden. Einen guten Kompromiss stellt der *Fahrstuhlalgorithmus* (SCAN) dar; dabei werden Anforderungen bevorzugt, die in der aktuellen Bewegungsrichtung des Plattenarms liegen. Noch gerechter zu den außen liegenden Anforderungen ist die *zirkuläre* Variante (C-SCAN). Sobald der Kopf an einem Ende angekommen ist, bewegt er sich direkt zum anderen Ende, ohne Anforderungen zu bearbeiten.

Selbstverständlich ist weiterhin, dass man auf Platten mit mehreren Spuren pro Zylinder solche Anforderungen bevorzugt, die in demselben Zylinder bleiben. Die Minimierung der Latenzzeit kann sich ebenfalls lohnen. Liegen Spuren im Cache, bringt das allerdings nichts.

8.1.3 RAID

Bei *RAID* (Redundant Array of Inexpensive/Independent Disks) werden mehrere Festplatten parallel betrieben. Dadurch erhöht sich die Zugriffsgeschwindigkeit (vergleiche Speicherverschränkung) und aufgrund von Redundanz die Verfügbarkeit. Folgende Klassifikation ist gebräuchlich.

Level 0: Beim *Striping* verteilt man aufeinander folgende Blöcke auf mehrere Festplatten. Dadurch erhält man eine Leistungssteigerung, aber keine Redundanz.

Level 1: Beim *Mirroring* wird außerdem jede Festplatte komplett gespiegelt, d. h. zu jeder Festplatte gibt es eine Kopie. Dadurch erhält man eine Datensicherung und beim Lesen wahlweise die doppelte Geschwindigkeit.

Level 2: Hier verteilt man die Bits eines Bytes auf mehrere Festplatten

und fügt Kontrollbits (auf zusätzlichen Festplatten) hinzu. Man setzt fehlerkorrigierende Codes ein.

Da das Vorgehen teurer ist als Level 3, wird es nicht benutzt.

Level 3: Man verteilt die Bits eines Bytes auf mehrere Festplatten und fügt ein Kontrollbit (auf einer zusätzlichen Festplatte) hinzu. Da man einen fehlerhaft gelesenen Sektor erkennen kann (dank Redundanz bei der Speicherung), reicht ein Paritätsbit bereits zur Fehlerkorrektur aus.

In den letzten beiden Fällen erfolgen die Plattenzugriffe parallel und die Steuerung muss viel rechnen.

Level 4: Kombiniert Level 0 und Level 3, d. h. die zusätzliche Festplatte enthält Paritätsblöcke. Die Plattenzugriffe erfolgen unabhängig voneinander. Das Lesen eines Blocks ist effizient, das Schreiben erfordert, dass die entsprechenden Blöcke auf den anderen Festplatten gelesen werden, die Summe gebildet wird und der Paritätsblock geschrieben wird.

Level 5: Funktioniert wie Level 4, jedoch wechseln die Rollen der Festplatten von Block zu Block, d. h. die Paritätsblöcke „laufen rundherum“. Da die Festplatte mit den Paritätsblöcken entlastet wird, ist dies etwas besser als Level 4 und sogar die beliebteste Variante.

Level 6: Kombiniert Level 2 und Level 5, d. h. fügt weitere Redundanz hinzu, so dass das System sogar gegen den Ausfall mehrerer Festplatten abgesichert ist. Das Lesen bleibt weiterhin effizient, aber das Schreiben wird ziemlich aufwändig.

8.2 Ein- und Ausgabe in UNIX

Gerätesteuerungen verbergen sich in UNIX hinter *Spezialdateien*. Eine solche enthält eine Haupt- und eine Untergerätenummer. Die Hauptgerätenummer entspricht einer Geräteklasse (in MINIX und Linux: 1 = RAM, 2 = Diskettenstation, 3 = IDE-Platte, 4 = Datensichtstation, 5 = serielle Schnittstellen, 6 = Drucker = parallele Schnittstellen), 8 = SCSI-Platte, 9 = Band usw. Die Untergerätenummern unterscheiden die Exemplare einer Geräteklasse bzw. die Betriebsarten eines Geräts. An der Hauptgerätenummer erkennt das Dateisystem die zuständige Steuerung. Diese ruft es auf und übergibt z. B. die

- angeforderte Operation,
- Untergerätenummer,
- Position (bei blockorientierten Geräten),

- Adresse des Datenbereichs im Prozess und
- Byteanzahl.

Der Rückgabewert enthält vor allem die Anzahl der übertragenen Bytes oder eine Fehlernummer.

Literaturverzeichnis

[1] Rüdiger Brause: *Betriebssysteme – Grundlagen und Konzepte* , 3. Auflage. Springer 2004

[2] Robert Love: *Linux Kernel Development*, 2nd Edition. Novell 2005

[3] S. P. Morse, E. J. Isaacson, D. J. Albert: *The 80386/387 Architecture*; Kapitel 6: *The Operating System's View*. Wiley 1987

[4] Jürgen Nehmer, Peter Sturm: *Systemsoftware – Grundlagen moderner Betriebssysteme*, 2. Auflage. dpunkt 2001

[5] Hans-Jürgen Siegert, Uwe Baumgarten: *Betriebssysteme – Eine Einführung*, 5. Auflage. Oldenbourg 2001

[6] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, 7th Edition. Wiley 2005

[7] William Stallings: *Operating Systems – Internals and Design Principles* , 5th Edition. Prentice Hall 2005

[8] Andrew S. Tanenbaum: *Modern Operating Systems*, 2nd Edition. Prentice Hall 2001

Stichwortverzeichnis