Kapitel 8

Betriebssysteme

Kapitel 8

Betriebssysteme

8.1 Die Aufgaben eines Betriebssystems

Der Betrieb eines Rechners erlaubt es heute im allgemeinen, eine gewisse Anzahl von Programmen scheinbar gleichzeitig ablaufen zu lassen. Das war nicht immer so. Die ersten Elektronenrechner konnten jeweils nur ein einzelnes Programm ausführen. Diesem Programm stand die gesamte Anlage einschließlich aller Peripherie (Drucker, Platten, Magnetbandlaufwerke, usw.) exklusiv zur Verfügung, und der Programmierer mußte alle benötigten Geräte direkt mit seinem Programm steuern.

Nun ist es einerseits recht umständlich, die Hardware-Funktionen direkt anzusteuern, andererseits werden die Geräte nur während eines kleinen Teils der Zeit wirklich genutzt. Insbesondere kam es durch die sehr unterschiedlichen Arbeitsgeschwindigkeiten der Rechnerkomponenten oft vor, daß die Zentraleinheit auf langsame Ein-/Ausgabegeräte warten mußte. Deshalb entstand schon früh der Gedanke, anstelle der *Monoprogrammierung*, bei der nur ein Programm den Rechner nutzt, mehrere Programme simultan zu betreiben. Anstatt die Zeit bis zum Abschluß einer E/A-Operation mit nutzlosem Warten zu verbringen, konnte die Zentraleinheit mit *Multiprogramming* in dieser Zeit ein anderes Programm bearbeiten. Hierdurch erreichte man also einen höheren Parallelitätsgrad und eine Verkürzung der Gesamtlaufzeit, wie sie in Abbildung 8-1 dargestellt ist.

Die Verwaltung der einzelnen Geräte muß jetzt das Betriebssystem übernehmen, und die Vergabe der Ressourcen darf nicht mehr Zeit in Anspruch nehmen, als durch die gesteigerte Parallelität frei wird.

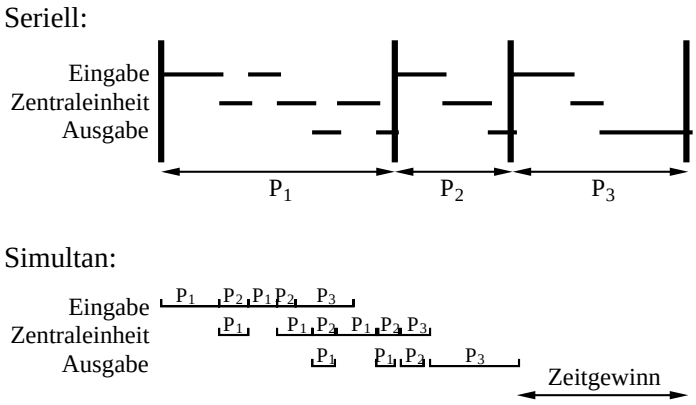

Abbildung 8-1: Vergleich Monoprogramming mit Multiprogramming

Eine spezielle Art dieses Simultanbetriebs entsteht, wenn mehrere Benutzer den Rechner gleichzeitig interaktiv verwenden wollen. Für diese Anforderung eignet sich das sogenannte *Time Sharing*: man teilt nacheinander den Benutzern kleine Zeitabschnitte zu (etwa in der Größenordnung zwischen 10 und 100 Millisekunden), in denen sie einen Abschnitt ihrer Programme abarbeiten lassen. Nach Ablauf jeder solchen *Zeitscheibe* unterbricht man das laufende Programm, sichert seinen Zustand an geeigneter Stelle, um es später fortsetzen zu können, und wählt für die nächste Zeitscheibe ein anderes Programm aus.

Von einem Betriebssystem verlangt man heute im allgemeinen, daß es einen solchen Betrieb unterstützt, indem es auch die Zentraleinheit als Betriebsmittel verwaltet und nach einem meist prioritätsgesteuerten System an simultan laufende Programme verteilt.

Wir haben also jetzt zwei Bereiche gefunden, in denen uns ein Betriebssystem unterstützen soll: es soll einerseits durch geeignete Schnittstellen die Programmierung des Rechners erleichtern, und andererseits die Verwaltung gemeinsam nutzbarer Ressourcen übernehmen.

Das Betriebssystem als erweiterte Maschine

Die erste Sicht betrachtet ein Betriebssystem als funktionale Erweiterung der Hardware. Was die reine Elektronik realisiert, ist zwar relativ einfach zu verstehen, aber recht kompliziert zu handhaben. Das gilt besonders bei der Ein-/Ausgabe. Wir wollen dies an dem Beispiel der Ausgabe eines Zeichens über eine serielle bzw. parallele Schnittstelle veranschaulichen.

Eine serielle Schnittstelle wird meistens durch einen speziell dafür entwickelten, integrierten Schaltkreis realisiert, der ein sogenanntes *Datenregister* enthält. Man kann sich dieses Register vorstellen wie eine Speicherstelle, in die der Prozessor jeweils ein Byte schreibt, das dann durch den Schnittstellenbaustein übermittelt wird. Den Abschluß der Übertragung zeigt die Schnittstelle normalerweise durch Setzen eines Bits in einem *Statusregister* an. Erst danach darf das nächste Byte in das Datenregister geschrieben werden.

Ganz anders sieht es aus, wenn man ein Zeichen in eine Datei auf einer Festplatte schreiben will: Platten und Disketten werden blockweise geschrieben und gelesen, so daß man solange Zeichen sammeln muß, bis die Größe eines Blocks erreicht ist. Dann erteilt man dem Steuerbaustein für die Festplatte den Auftrag zum Schreiben dieses Blocks, indem man Parameter wie Blockgröße, Speicheradressen, Kopf-, Spur- und Sektornummer sowie

schließlich den Kode für die Anweisung *Block schreiben* in die entsprechenden Register schreibt.

Es ist klar, daß man sich beim Programmieren eines Textverarbeitungs- oder eines CAD-Systems nicht mit solchen Details beschäftigen will. Schon die Unterscheidung zwischen den verschiedenen Arten von Peripheriebausteinen ist hier unerwünscht, notwendig ist dagegen eine einheitliche Form, in der ein Ausgabeauftrag erteilt werden kann. Es ist eine Aufgabe des Betriebssystems, diese Art der Ausgabe zu realisieren, als möglichen Systemauftrag anzubieten und dabei Besonderheiten der Hardware zu verdecken.

Das Betriebssystem als Verwaltungsprogramm

Im vorangegangenen Abschnitt haben wir ein Betriebssystem aus der Sicht eines Anwendungsprogrammierers betrachtet. Die entgegengesetzte Sichtweise untersucht die Zugriffe auf Bestandteile eines Computersystems, das von mehreren Anwendern gleichzeitig genutzt wird.

Stellen wir uns einen Drucker vor, auf den drei verschiedene Programme gleichzeitig ausgeben möchten. Es ist sofort klar, daß dabei ein heilloses Durcheinander entstehen würde: die ersten drei Zeilen könnten vom ersten Programm stammen, die nächste Zeile vielleicht vom zweiten Programm, dann zwei Zeilen aus dem dritten Programm und so weiter. Im Extremfall könnten sogar innerhalb einer Zeile Ausgabeteile verschiedener Programme gemischt sein.

Hier muß ein Betriebssystem Ordnung in das Chaos bringen. Es kann zum Beispiel die für den Drucker bestimmte Ausgabe zunächst auf einer Platte puffern, bis das Programm seine Arbeit abgeschlossen hat. Danach erst wird die erzeugte Ausgabe von der Platte zum Drucker kopiert, wenn dieser frei ist. Auf diese Weise können mehrere Programme gleichzeitig Druckaufträge erstellen, ohne sich gegenseitig zu behindern: jeder Auftrag wird in einer eigenen Datei gesammelt und nach Abschluß der Ausgabe in eine Warteschlange (engl. *queue*) eingereiht. Sobald der Drucker mit der Ausgabe einer Datei fertig ist, wird diese gelöscht und mit der Ausgabe der nächsten begonnen.

Auch wenn mehrere Benutzer nicht gleichzeitig, sondern nacheinander denselben Rechner benutzen, ist eine Zugriffskontrolle notwendig: die Daten des einen Anwenders sind nicht notwendigerweise auch für die Augen des anderen bestimmt, so daß die Zugriffe durch ein Betriebssystem überwacht werden müssen.

8.2 Begriffsdefinitionen

Die Schnittstelle zwischen Anwendungsprogrammen und dem Betriebssystem wird gebildet durch sogenannte *Systemaufrufe*. Jeder Systemaufruf (bei UNIX *System Call* genannt) fordert eine bestimmte Aktion vom Betriebssystem an.

Die Systemaufrufe erzeugen, benutzen und zerstören Software-Objekte, deren Verwaltung das Betriebssystem übernimmt. Typische Beispiele für solche Objekte sind die unten angeführten Prozesse und Dateien. Anschließend beleuchten wir kurz die Mechanismen, nach denen Systemaufrufe abgewickelt werden.

Prozesse

Ein Prozeß ist ein Vorgang, der nach bestimmten Vorschriften ablaufen soll. Im Rechner nennt man diese Vorschrift ein Programm, und außer dem Programm benötigt der Prozeß zusätzliche Informationen, Geräte und Geräteteile (Betriebsmittel), um seine Arbeit zu verrichten. Ein Prozeß ist also ein Programm in Ausführung, dem eine Umgebung aus Betriebsmitteln zugeordnet ist. Zum Prozeß gehören beispielsweise Hauptspeicher und Dateien, die der Prozeß geöffnet hat. Ein Prozeß ist eine Aktivität beliebiger Art. Man kann Prozesse unterbrechen, ihren Zustand, also den Stand der Arbeit, sichern, und den Prozeß zu einem späteren Zeitpunkt dort fortsetzen, wo man ihn unterbrochen hatte. Dies ist besonders für den Time-Sharing-Betrieb wichtig.

Ein einzelner Prozessor kann durch geeignete Auswahlverfahren nacheinander verschiedenen Prozessen zugeteilt werden, wobei jedem Prozeß eine Priorität zugeordnet ist. Die Zuteilung des Prozessors an einen anderen Prozeß bewirkt eine Unterbrechung des bisherigen Prozesses und eine Sicherung seines Zustandes für die spätere Fortsetzung. Der Wert des Befehlszählers ist ein wichtiger Bestandteil des Prozeßzustandes. Da dieses Register jeweils auf den nächsten auszuführenden Befehl zeigt, dokumentiert es den Stand der Verarbeitung und muß daher bei jedem Prozeßwechsel gerettet werden. Jeder Prozeß verfügt also konzeptionell über einen eigenen Befehlszähler. Ein wesentlicher Anteil des Umschaltens von einem Prozess zu einem anderen besteht im Sichern des Befehlszählerstandes und im Einsetzen des gesicherten Zählerstandes für den nächsten Prozeß. Bild 8-2 zeigt diesen Zusammenhang: während der einzige Befehlszähler der Hardware durch Prozeßwechsel nacheinander verschiedenen Prozessen und ihren Programmen zugeordnet wird (Bild 8-2(a)), sieht jeder Prozeß nur seinen eigenen Befehlszähler, der von allen anderen Prozessen unabhängig ist.

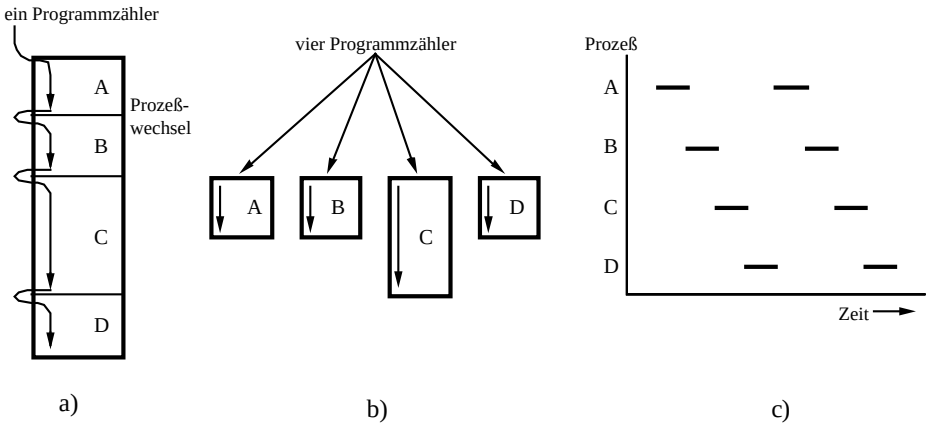

Abbildung 8-2: Vier simultan aktive Prozesse

Bestandteil des Prozeßzustandes ist auch eine Klassifizierung des Prozesses in die Kategorien *rechnend*, *rechenbereit* und *blockiert*. Diese Zustände bedeuten:

Rechnend: Der Prozessor ist dem Prozeß zugeteilt, das entsprechende Programm ist also gerade in Bearbeitung.

Rechenbereit: Der Prozeß ist ausführbar (alle Betriebsmittel sind bereit), aber der Prozessor ist einem anderen Prozeß zugeteilt.

Blockiert: Der Prozeß kann solange nicht weiterbearbeitet werden, bis ein externes Ereignis eintritt (zum Beispiel Abschluß einer Eingabe).

Die möglichen Übergänge zwischen diesen Zuständen zeigt Bild 8-3. Übergang 1 kann durch den Prozeß selbst mit einem entsprechenden Systemaufruf (zum Beispiel `BLOCK`) ausgelöst werden, häufiger ist aber eine automatische Zustandsänderung durch das Anfordern einer Ein- oder Ausgabe: der Prozeß bleibt dann solange blockiert, bis der Datentransfer abgeschlossen ist, und wird danach wieder in den Zustand *rechenbereit* überführt (Übergang 2).

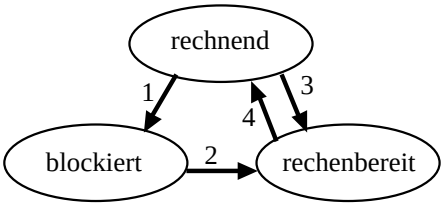

Abbildung 8-3: Prozeßzustände und Zustandsübergänge

Die Übergänge 3 und 4 werden durch das Vergabeverfahren veranlaßt, das den Prozessor nacheinander den Prozessen zuordnet. Ohne daß die Prozesse dies feststellen können, beschließt das Vergabeverfahren, daß ein *rechnender* Prozeß lange genug ausgeführt wurde und nun durch einen anderen Prozeß abzulösen ist. Der bisher bearbeitete Prozeß gelangt dann durch Übergang 3 in den Zustand *rechenbereit*, weil zwar alle Voraussetzungen für die weitere Bearbeitung noch vorliegen, der Prozessor aber andere Arbeiten übernimmt. Nachdem die anderen Prozesse ebenfalls lange genug bearbeitet wurden, wird der unterbrochene Prozeß wieder an die Reihe kommen (Übergang 4).

Für die Verwaltung der Prozesse enthält das Betriebssystem gewöhnlich eine *Prozeßtabelle*, in der es für jeden Prozeß einen Eintrag mit den notwendigen Verwaltungsinformationen gibt.

Dateien

Zur langfristigen Aufbewahrung schreibt man Informationen in Dateien. Die Verwaltung dieser Behälter erledigt eine Komponente des Betriebssystems, die *Dateisystem* (Filesystem) heißt. Das Dateisystem sorgt für die zu Beginn dieses Kapitels geforderte Geräteunabhängigkeit bei der Programmierung, indem es Systemaufrufe anbietet, mit denen Dateien erzeugt, umbenannt, gelesen, geschrieben und gelöscht werden können. Vor dem Lesen oder Schreiben muß eine Datei geöffnet werden, nach der Verarbeitung sollte sie geschlossen werden. Auch für diese Operationen gibt es entsprechende Aufrufe.

Die meisten Betriebssysteme erlauben es, Dateien in verschiedenen *Verzeichnissen* (*Directories*) zu gruppieren. So kann ein Student beispielsweise die Dateien, die er für seine Kurse benötigt, in separate Verzeichnisse aufteilen, indem er für jeden Kurs ein eigenes Verzeichnis anlegt. Auch für das Anlegen und Zerstören der Verzeichnisse muß das Betriebssystem

also Systemaufrufe anbieten. Ein Eintrag in einem Verzeichnis ist entweder eine Datei oder ein weiteres Verzeichnis, so daß eine Hierarchie wie in Bild 8-4 entstehen kann.

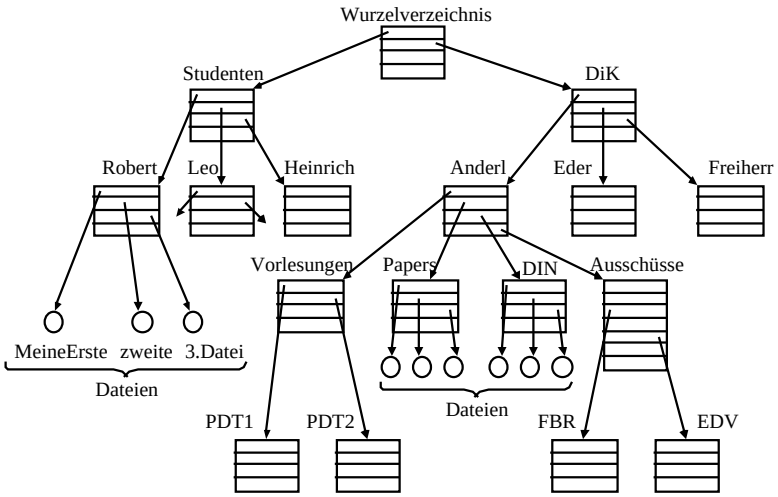

Abbildung 8-4: Ein Dateisystem

Um Dateien in einer solchen Hierarchie eindeutig bezeichnen zu können, bedient man sich der *Pfadnamen* (*Path Names*), die die Lage einer Datei ausgehend von einem bekannten Punkt beschreiben. Ein möglicher Ausgangspunkt ist dabei das *Wurzelverzeichnis* (*Root Directory*), das wie in Bild 8-4 gezeigt die Spitze der Hierarchie bildet. Der Pfadname wird gebildet, indem man dem Dateinamen die Namen der Verzeichnisse voranstellt, über die man die Datei vom Wurzelverzeichnis aus erreicht. Zur Trennung der einzelnen Namen voneinander dient bei UNIX ein Schrägstrich, MS-DOS verwendet den umgekehrten Schrägstrich, bei Apple Macintosh ist der Doppelpunkt gebräuchlich.

Beispiel 8-3 Die Datei `zweite` aus Bild 8-4 hat in einem UNIX-System den absoluten Pfadnamen `/Studenten/Robert/zweite`, das Dateisystem MS-DOS von würde dieselbe Datei mit `\STUDENTE\ROBERT\ZWEITE` bezeichnen (das Wort `STUDENTEN` ist länger als acht Buchstaben und muß daher abgeschnitten werden), ein Apple Macintosh findet dieselbe Datei mit dem Pfadnamen `Studenten:Robert:zweite`.

Oft ist es vorteilhaft, den Pfad nicht von der Wurzel des Baumes aus anzugeben, sondern mit Bezug auf das aktuelle Verzeichnis. Solche Angaben nennt man *relative Pfadnamen*. Während die absoluten (auf die Wurzel bezogenen) Pfadangaben in der Schreibweise der UNIX-Dateisysteme stets mit einem Schrägstrich beginnen, erkennt man relative Pfadnamen am Fehlen dieses Schrägstriches, sie beginnen also mit einem Buchstaben oder einer Ziffer.

Das UNIX-Dateisystem verwendet für das übergeordnete Verzeichnis die Schreibweise „`..`" Unter diesem Namen ist in jedem Verzeichnis das jeweils übergeordnete Verzeichnis eingetragen. Außerdem ist jedes Verzeichnis unter dem Namen „`.`" in sich selbst eingetragen. Dadurch wird es beispielsweise sehr einfach, eine Datei in das aktuelle Verzeichnis zu kopieren, um die Kopie anschließend zu bearbeiten.

Beispiel 8-6: Um beim Schreiben der Vorlesung Produktdatentechnologie I (aktuelles Verzeichnis: `/DiK/Anderl/Vorlesungen/PDT1`) auf die DIN 66003 zuzugreifen, kann Professor Anderl den relativen Pfad

> `../../DIN/DIN66003` verwenden: mit dem ersten „`..`" bezeichnet er dabei das Verzeichnis `Vorlesungen`, mit dem zweiten das Directory `Anderl`. Um die Datei in das Verzeichnis `PDT1` zu kopieren, könnte er den Befehl `cp ../../DIN/DIN66003.` geben. Das aktuelle Verzeichnis wird also durch den Punkt dargestellt, der die Eingabe von `/DiK/Anderl/Vorlesungen/PDT1` erspart.

Während das UNIX-Dateisystem alle Laufwerke zu einem Baum zusammenfaßt, werden bei den meisten anderen Systemen getrennte Bäume auf den einzelnen Laufwerken geführt. Deshalb benötigen zum Beispiel Windows Systeme eine besondere Schreibweise für die Laufwerksangabe, während UNIX darauf verzichten kann. Stattdessen werden die Dateisysteme aller Laufwerke zu einem gemeinsamen Baum verbunden, indem man jeweils das Wurzelverzeichnis einer Platte an geeigneter Stelle in einen vorhandenen Baum aufnimmt. Dazu dient das UNIX-Kommando `mount`.

Systemaufrufe

Die Systemaufrufe übernehmen Aufgaben, die das Anwendungsprogramm nicht selbst lösen kann. Insbesondere greifen viele Systemaufrufe auf Bestandteile der Hardware zu, die vor unkontrolliertem Zugriff geschützt werden müssen. Während der Abarbeitung eines Systemaufrufs muß der Prozessor besondere Instruktionen ausführen und auf Speicherbereiche zugreifen dürfen, deren Benutzung einem normalen Anwender verboten ist.

Moderne Rechner unterstützen durch hardwaregestützte Prüfung aller von einem Programm verwendeten Adressen und Auslösen von Unterbrechungen bei Verletzung der Zugriffsrechte. Die entsprechende Hardware heißt *Memory Management Unit* (MMU, vgl. Kap. 8.5).

Der Zugriff auf bestimmte Speicherbereiche muß also Anwendungsprogrammen verwehrt, Systemroutinen aber erlaubt werden. Zur Unterscheidung besitzen Prozessoren daher zwei (manchmal auch mehr) Betriebszustände, den Benutzer- und den Kernmodus (*User Mode* und *Kernel Mode*). Der Zugriff auf bestimmte Speicherbereiche und das Ausführen bestimmter Anweisungen (insbesondere solcher, die den Betriebszustand ändern) sind nur im Kernmodus gestattet, so daß diese Anweisungen und Speicherbereiche für normale Anwendungsprogramme nicht verwendbar sind.

Der Wechsel in den Kernmodus (in Bild 8-5 mit 1 bezeichnet) verschafft der Systemroutine Zugriff auf diejenigen Ressourcen (Speicher, Schnittstellenregister, etc.), auf die das Anwendungsprogramm nicht selbst zugreifen darf.

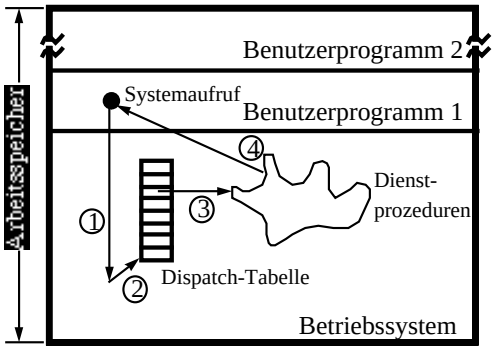

Abbildung 8-5: Die Durchführung eines Systemaufrufs

Da der TRAP-Befehl der einzige ist, mit dem eine Änderung des Betriebszustandes von Benutzermodus nach Kernmodus erfolgen kann, und da dieser TRAP-Befehl gleichzeitig auch eine Systemroutine aktiviert, die zuerst die gewünschten Prüfungen durchführt, ist ein Umgehen der Sicherungen nicht möglich.

Einer der von der Routine `read` vorbereiteten Parameter ist meist eine Kodenummer, die die gewünschte Dienstleistung bezeichnet. An dieser Nummer erkennt das Betriebssystem, daß der eben ausgeführte TRAP-Befehl den Systemdienst READ anfordert. Dazu dient meist eine Tabelle, die zu jeder Auftragskodierung (Bild 8-5, Ziffer 2) eine Routine zur Ausführung der Operation referenziert. Zur Prüfung der Parameter und Zugriffsberechtigungen und für den Transfer der Daten wird also das System die entsprechende Dienstprozedur anstoßen (Bild 8-5, Ziffer 3) und nach getaner Arbeit die Fortsetzung des Anwenderprogrammes veranlassen (Ziffer 4). Dazu gibt es eine Instruktion, die zum Beispiel RETURN FROM TRAP heißen kann. Sie bewirkt das Gegenstück zu den mit TRAP angestoßenen Aktionen.

Die Shell

Stellen die Systemaufrufe die Schnittstelle des Programmierers zum System dar, so ist es für den Anwender oft die *Shell*. Dieses Programm liest Befehle ein und führt sie aus, indem es eine geeignete Folge von Systemaufrufen ausführt. Die Schnittstelle zum Betriebssystem wird also von einem Programm gebildet, das nicht selbst Bestandteil des Systems ist, aber mit diesem intensiv kommuniziert.

Gewöhnlich wird eine Shell durch Einschalten des Rechners, Anmelden am Rechner oder Öffnen eines entsprechenden Fensters (*Shell Window*) gestartet und gibt daraufhin eine Aufforderung zur Eingabe (*Prompt*) aus. Gibt der Benutzer nun ein Kommando ein, so wird dieses von der Shell untersucht und klassifiziert. Es gibt interne Kommandos, die die Shell direkt ausführt und externe Kommandos, für die die einen neuen Prozeß startet, in dem das entsprechende Programm abläuft. Sobald die Bearbeitung des internen Kommandos abgeschlossen ist oder der angestoßene Prozeß terminiert, gibt die Shell einen neuen Prompt aus.

Aus dem Betriebssystem UNIX stammt die Idee, durch die Shell eine Umleitung der Ein- und Ausgabe für den Prozeß zu bewirken, der zur Ausführung eines Befehls von der Shell gestartet wird.

> **Beispiel 8-8:** Gibt der Benutzer einer UNIX-Shell das Kommando `date` ein, so startet die Shell das Programm `date` in einem neuen Prozeß. Gibt man das Kommando `date > file` ein wird die Ausgabe in die Datei `file` umgeleitet. Auf dem Bildschirm erscheint also lediglich ein neuer Prompt, sobald der Prozeß beendet ist.

Eine besonders interessante Form dieser Umleitung ist die *Pipeline*: dabei wird die Ausgabe eines Prozesses als Eingabe für einen anderen benutzt. Hier startet also die Shell zwei (oder mehrere) Prozesse und koppelt die Ausgabe eines Prozesses als Eingabe an den nächsten.

> **Beispiel 8-11:** Der Befehl `man` formatiert die Beschreibung zu einem UNIX-Befehl und gibt sie auf dem Bildschirm aus. Um die so erzeugte Dokumentation zu drucken kann man sie mit `man sort | lpr` direkt in die Warteschlange des Druckers einreihen lassen.

Eine weitere Variationsmöglichkeit betrifft das Warten auf die Beendigung des Kindprozesses, in dem externe Befehle ausgeführt werden (*Hintergrundverarbeitung)*. Dazu hängt man an das Ende einer Befehlszeile ein „`&`“. Dadurch veranlaßt man die Shell, auf die Beendi-

gung des Kindprozesses nicht zu warten, sondern sofort einen neuen Prompt auszugeben.

8.3 Die innere Struktur von Betriebssystemen

Für den Anwender ist zwar der innere Aufbau eines Betriebssystems im allgemeinen nicht erkennbar, die hier vorgestellten Strukturmodelle bestimmen aber bis zu einem gewissen Grade die Arbeitsweise eines Systems und sind außerdem auch auf andere Typen von Software anwendbar.

Monolithische Systeme

Die monolithische Struktur ist sehr einfach. Man gliedert die einzelnen Aufgaben in bestimmte Gruppen, etwa Prozeßverwaltung, Dateisystem usw., stellt aber zwischen diesen Gruppen gibt keine hierarchische Ordnung her. Stattdessen kann jede Routine Funktionen aus allen anderen Gruppen aufrufen.

Um etwas mehr Übersicht in ein solches System zu bringen, gliedert man manchmal die Routinen in ein Hauptprogramm sowie diverse Dienst- und Hilfsprozeduren. Eine solche Gliederung zeigt Bild 8-6. Bei der Bearbeitung eines Systemaufrufs führt der TRAP-Befehl dann zunächst in das Hauptprogramm, wo anhand der Auftragsnummer eine Dienstprozedur ausgewählt wird (vgl. Abschnitt 8.2.3). Diese Dienstprozedur verwendet zur Durchführung ihres Auftrags (zum Beispiel *schreibe Plattenblock*) Hilfsprozeduren wie zum Beispiel eine Routine, die Daten aus dem Puffer im Anwendungsprogramm holt.

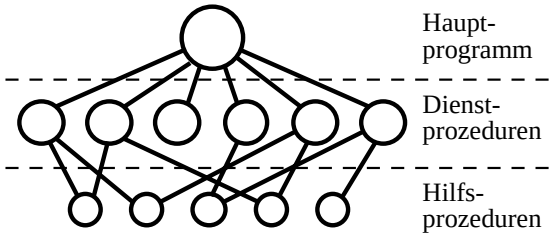

Abbildung 8-6: Ein einfaches Strukturmodell für ein monolithisches System

Die monolithische Struktur erfordert von den Programmierern ein hohes Maß an Disziplin und stellt hohe Ansprüche an die Dokumentation, wenn man das Abgleiten in ein Chaos vermeiden will. Daher eignet sich dieses Modell nur für relativ kleine Systeme. Besonders bei älteren Betriebssystemen, deren Entwurf neuere Erkenntnisse der Informatik noch nicht berücksichtigen konnte, ist dennoch die monolithische Struktur sehr verbreitet.

Geschichtete Systeme

Verfeinert man den Ansatz aus Bild 8-6, so gelangt man zu einer Struktur, die das Betriebssystem in eine feste Anzahl genau definierter Schichten unterteilt. Jede Schicht benutzt zur Erledigung ihrer Aufgaben ausschließlich Funktionen, die eine darunterliegende Schicht bereitstellt.

Ein Beispiel für ein Betriebssystem mit Schichtstruktur ist das 1968 von E. W. Dijkstra und seinen Studenten an der Technischen Hogeschool Eindhoven (Niederlande) entwickelte THE-System. Es bestand aus 6 Schichten, die in Bild 8-7 skizziert sind. Das THE-System

war ein einfaches Stapelverarbeitungssystem, bei dem ein Operateur mehrere simultan laufende Aufträge überwachte.

Schicht	Bedeutung
5	Prozeß des Systembedieners
4	Benutzerprogramme
3	Ein-/Ausgabeverwaltung
2	Operateur- zu Prozeßkommunikation
1	Speicher- und Trommelverwaltung
0	Prozessorvergabe und Multiprogrammierung

Abbildung 8-7: Die Struktur des THE-Betriebssystems

Betrachten wir beispielhaft die Schicht 2: sie behandelte die Kommunikation zwischen der Bedienerkonsole einerseits und den Prozessen andererseits. Es gab nur einen Bedienerarbeitsplatz, aber oberhalb der Schicht 2 konnte jeder Prozeß so arbeiten, als habe er seine eigene Konsole. Die notwendige Koordination fand innerhalb der Schicht 2 statt.

Solche Strukturen erfordern zwar erhebliche Vorarbeiten, um eine gute Einteilung in geeignete Schichten zu finden, sind aber dann sehr übersichtlich und dadurch gut verständlich.

Virtuelle Maschinen

Eine besonders interessante Betriebssystem-Struktur trennt zwei Aufgaben, die einem Betriebssystem zufallen: einerseits die Realisierung des Mehrprogrammbetriebs, andererseits die Erweiterung der Hardware-Funktionalität. Das wohl bekannteste System mit dieser Struktur ist das von IBM für die Großrechnerserie /370 entwickelte System VM/370. Es ist in Bild 8-8 skizziert. Das Betriebssystem VM/370 läuft direkt auf der Hardware, gleichzeitig werden aber mehrere andere Betriebssysteme eingesetzt, die auf VM/370 laufen. Die Aufgabe von VM/370 ist es, für jedes der anderen Systeme eine eigene Hardware zu simulieren, die sich genau so verhält wie der tatsächlich installierte Rechner. Dadurch war es möglich, auf demselben Rechner zum Beispiel gleichzeitig ein Stapelverarbeitungssystem und mehrere dialogorientierte Ein- und Mehrbenutzer-Systeme einzusetzen. So konnten in einem Rechenzentrum die Produktionsläufe im Stapelsystem abgearbeitet werden, während die Entwickler in den Dialogsystemen arbeiteten, ohne eigene Rechner zu benötigen.

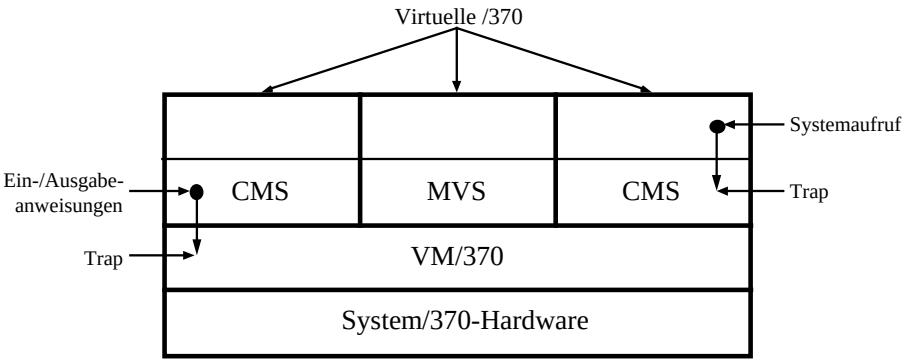

Abbildung 8-8: Die Struktur des Systems VM/370 mit CMS

Das Client-Server-Modell

Die Begriffe Client und Server sind uns im Zusammenhang mit Netzwerken schon einmal begegnet, und sie haben dort eine ähnliche Bedeutung wie hier. Die Idee jedes Client-Server-Modells ist es, Aufträge aus einem Prozeß durch einen anderen Prozeß zu bearbeiten. Den Auftraggeber bezeichnet man dabei als Client und den Auftragnehmer, der den gewünschten Dienst erbringt, als Server.

Zum Erteilen des Auftrags und für das Bereitstellen der Ergebnisse benötigt man einen Mechanismus, der das Austauschen von Nachrichten ermöglicht. Die Aufgabe eines Betriebssystems reduziert sich bei diesem Strukturmodell darauf, den Transport der Nachrichten zu gewährleisten, wie in Bild 8-9 dargestellt.

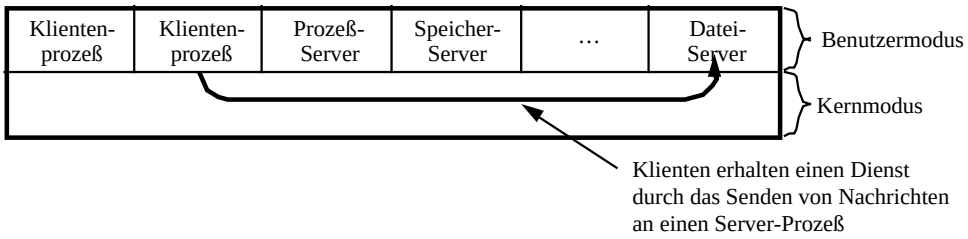

Abbildung 8-9: Das Client-Server-Modell

Das Bild ist nur als Prinzipskizze aufzufassen, denn für die technische Realisierung ist es nicht günstig, alle Server im Benutzermodus ablaufen zu lassen: manche von ihnen benötigen Zugriff auf Instruktionen und Speicherbereiche, die nur im Kernmodus verfügbar sind. Man muß also diese Serverprozesse oder wenigstens Teile davon im Kernmodus ausführen. So muß der Kern beispielsweise den Mehrprogrammbetrieb unterstützen, auch wenn der Prozeßserver die Verteilung der Rechenzeit übernimmt. Die Treiberroutinen eines I/O-Servers, in denen auf Register der Schnittstellen zugegriffen wird, gehören ebenfalls in den Kern.

Der Vorteil der Client-Server-Struktur liegt darin, daß Strategien und Mechanismen voneinander getrennt werden: der Mechanismus ist weitgehend durch die Hardware vorgeschrieben, aber für die Strategie gibt es oft mehrere Alternativen. Wenn die Strategie als separater Prozeß implementiert ist, kann man sie wesentlich leichter austauschen.

Client-Server-Strukturen können sich neben Betriebssystemen auch für andere Software eignen, wenn ein Programmpaket aus mehreren Prozessen besteht, von denen jeder für einen bestimmten Teilbereich zuständig ist. Die Rolle des Kerns kann ein eigener Kommunikationsprozeß oder das Betriebssystem übernehmen.

8.4 Prozesse, Kommunikation, Scheduling

Der Begriff des Prozesses wurde weiter oben bereits definiert. Bisher haben wir verschiedene Aspekte einzelner Prozesse betrachtet; hier soll es um das Zusammenspiel mehrerer Prozesse mit dem Betriebssystem gehen.

Im Zusammenhang mit der Shell haben wir einen Weg kennengelernt, wie mehrere Prozesse simultan für denselben Anwender aktiv sein können. Nehmen wir an, jemand hat aus seiner Shell, die in Prozeß A läuft, eine weitere Shell als Hintergrundprozeß gestartet (Prozeß B), die durch ein Skript gesteuert eine Verarbeitung kontrolliert, zu der drei Prozesse D, E und F gehören. Simultan dazu kann der Benutzer aus der ersten Shell einen Editor aufgerufen haben, der in Prozeß C läuft. Dann entsteht eine Hierarchie von Prozessen, wie sie in Bild 8-10 gezeigt ist.

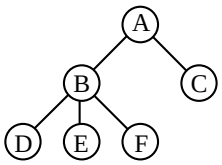

Abbildung 8-10: Ein Prozeßbaum

Offensichtlich muß das Betriebssystem Aufrufe bereitstellen, mit denen solche *Prozeßbäume* erstellt und manipuliert werden können.

Zunächst braucht man einen Mechanismus zur Kennzeichnung eines Prozesses. Dazu dient im allgemeinen eine vom Betriebssystem vergebene Nummer für jeden existierenden Prozeß, die man *Process Identification* (abgekürzt *pid*) nennt. Bei jeder Manipulation kann der gewünschte Prozeß gegenüber dem Betriebssystem eindeutig anhand seiner *pid* bezeichnet werden.

Die wichtigsten Aufrufe zur Manipulation dienen zum Erzeugen eines neuen Prozesses oder zum Abbruch. Man kann aber auch nach Eigenschaften von Prozessen fragen, etwa nach der bisher verbrauchten Rechenzeit oder nach dem Eigentümer eines Prozesses. Eigentümer ist, wer den Prozeß erzeugt hat. Dazu vergibt der Verwalter eines UNIX-Systems Gruppen- und Benutzernummern (*Group Identification (gid)* und *User Identification (uid)* genannt), mit denen jeder Benutzer einer Benutzergruppe zugeordnet und beide gegenüber dem System bezeichnet werden können. In den Tabellen des Betriebssystems sind jedem Prozeß Einträge zugeordnet, die unter anderem *gid* und *uid* des Eigentümers angeben.

Prozesse können untereinander Nachrichten austauschen, indem sie *Signale* senden und empfangen. Sendet man eine bestimmte Nachricht an einen Prozeß, der nicht auf den Empfang vorbereitet ist, so wird der Prozeß dadurch abgebrochen. Daher heißt der UNIX-Befehl zum Senden von Signalen `kill`. Häufig werden Signale aber nicht zum Abbrechen von Prozessen verwendet, sondern um den Eintritt bestimmter Ereignisse anzuzeigen. Der empfangende Prozeß kann Aktionen spezifizieren, die beim Empfang von Signalen ausgeführt werden sollen.

Die Verteilung der Rechenzeit auf die einzelnen Prozesse erledigt der *Scheduler*. Er legt die Zeitspannen fest, für die der Prozessor den Prozessen zugeordnet wird. Nach jeder Zeitscheibe wird der Scheduler erneut aufgerufen, um den nächsten rechenbereiten Prozeß auszuwählen, dem eine Zeitscheibe zugeteilt werden soll. Damit bildet der Scheduler, wie in

Bild 8-11 gezeigt, die Grundlage des Mehrprogrammbetriebs.

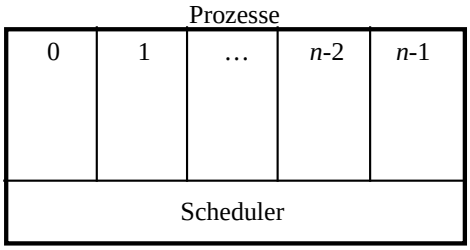

Abbildung 8-11: Scheduler in einem Betriebssystem

Bei näherer Betrachtung erweist sich diese Aufgabe als recht schwierig, weil dabei Forderungen berücksichtigt werden müssen, die sich teilweise widersprechen. Eine unvollständige Auswahl von Kriterien umfaßt:

Fairneß: Jeder Prozeß soll einen gerechten Anteil der Prozessorzeit erhalten.

Effizienz: Der Prozessor soll immer vollständig ausgelastet sein.

Antwortzeit: Die Reaktionszeiten des Systems sollen für die interaktiv arbeitenden Benutzer minimiert werden.

Verweilzeit: Die Wartezeit auf die Ausgabe von Stapelaufträgen soll möglichst kurz sein.

Durchsatz: Die Anzahl der in einem gegebenen Zeitintervall ausgeführten Aufträge soll maximiert werden.

Um die Antwortzeit der interaktiv arbeitenden Benutzer zu verbessern, könnte man zum Beispiel Stapelaufträge nur nachts ausführen, würde dann aber sicher keine günstigen Verweilzeiten erreichen.

Für eine gerechte Verteilung der Rechenzeit ist es wichtig, daß der Scheduler einem rechnenden Prozeß den Prozessor entziehen kann und nicht warten muß, bis der Prozeß von sich aus die Kontrolle abgibt. Ein Verfahren, daß gegebenenfalls einen rechnenden Prozeß suspendiert, um einen anderen weiterrechnen zu lassen, wird als *Preemptive Scheduling* bezeichnet. Dies wird im Allgemeinen mit dem Aufruf des Schedulers nach jeder Zeitscheibe erreicht. Beim *Nonpreemptive Scheduling* hingegen führen nur bestimmte Betriebssystemaufrufe des Prozesses und sein Terminieren zur Aktivierung des Schedulers. Frühe Stapelsysteme, die mit *Nonpreemptive Scheduling* arbeiteten, sind zwar leichter zu implementieren, aber für allgemeine Anwendungen nicht geeignet, weil sie sich auf die Fairneß der einzelnen Prozesse nicht verlassen können.

8.5 Speicherverwaltung

In einem Rechner, der zu jeder Zeit nur ein Programm ausführt, ist Speicherverwaltung einfach: man ordnet den gesamten verfügbaren Speicher jeweils dem laufenden Programm zu. Eine solche Methode verwendeten die frühen MS-DOS-Versionen, aber selbst in einem so einfachen Betriebssystem gab es schon bald Nachfrage nach einem Mechanismus, von einem Prozeß aus einen anderen zu starten und später die Kontrolle zurück an das erste Programm zu geben, ohne es neu zu starten. Also definierte man einen Systemaufruf, mit dem ein Prozeß einen Teil des zunächst vollständig belegten Hauptspeichers wieder freigeben kann, um darin andere Programme laufen zu lassen.

Interessanter wird die Situation, wenn man mehrere konkurrierende Prozesse zulassen will, die wechselnde Anforderungen in bezug auf den ihnen zugeteilten Hauptspeicher stellen. Wir wollen hier nicht näher auf ältere Lösungen eingehen, die zum Beispiel durch Partitionierung des Hauptspeichers den Simultanbetrieb mehrerer Prozesse erlaubten, sondern nur die heute gängigsten Methoden vorstellen. Sie setzen Unterstützung durch die Hardware voraus, die aber in allen Prozessoren vorgesehen ist, die für den Mehrprogrammbetrieb eingesetzt werden.

Eine Fragmentierung des Hauptspeichers soll vermieden werden:

Beim Laden mehrerer Programme mit unterschiedlichem Speicherbedarf entstehen nach der Beendigung eines Prozesses Lücken in der Belegung des Speichers. Bild 8-12 zeigt beispielhaft eine Folge von Speicherbelegungen, während verschieden große Programme A, B, C, D und E gestartet werden, laufen und schließlich terminieren.

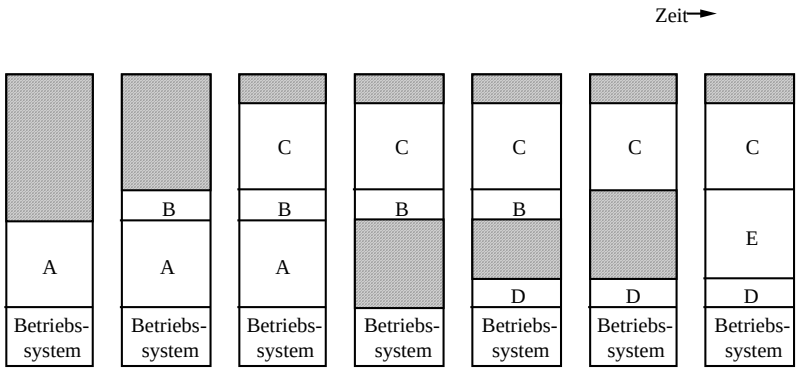

Abbildung 8-12: Fragmentierung des Hauptspeichers

Die schraffierten Bereiche markieren die entstehenden Lücken. Außerdem ist es vorteilhaft, jedem Programm vorzuspiegeln, sein Speicherbereich beginne mit der Adresse Null, weil man dann nicht schon beim Erstellen des Programmes wissen muß, an welche Stelle des Hauptspeichers es später geladen wird. Wünschenswert ist also eine Übersetzung der vom Programm benutzten Adressen: einerseits kann man dabei die vom Programm benutzten Adressen mit Null beginnen lassen und die Transformation so einrichten, daß damit die tatsächlich belegten Speicherzellen angesprochen werden, andererseits kann man bei Bedarf das Programm im Hauptspeicher verschieben (um Lücken zu schließen) und gleichzeitig die Transformation angleichen, so daß das Programm gar nicht bemerkt, wenn es verlagert wird.

Diese erste Teilaufgabe kann man elegant und einfach mit einem sogenannten *Basisregister* lösen: das ist ein Register im Prozessor, dessen Inhalt zu allen Adressen addiert wird, die ein Programm benutzt. Beim Laden des Programms setzt das Betriebssystem das Basisregister so, daß es auf den Beginn des belegten Speicherbereiches zeigt. Alle Programme beginnen ihre Adressen mit Null, aber jeder Prozeß erhält einen anderen Wert für das Basisregister. Beim Prozeßwechsel sorgt der Scheduler dafür, daß jeweils der zum aktiven Prozeß gehörende Wert im Basisregister steht. Muß man ein Programm im Hauptspeicher verschieben, so setzt man das Basisregister entsprechend der Verschiebung neu, bevor man die Kontrolle wieder an das Programm abgibt.

Laufende Prozesse sollen voreinander geschützt werden:

Die andere Teilaufgabe betrifft den Schutz der Prozesse voreinander: fehlerhafte Programme oder solche, die mit Absicht andere Prozesse stören sollen, können Adressen benutzen, die über den ihnen zugewiesenen Speicher hinausgehen. Damit wird Speicher angesprochen, der einem anderen Prozeß gehört, so daß das fehlerhafte Programm illegal auf dessen Daten zugreift, möglicherweise vertrauliche Informationen erhält oder gar verändert.

Um das zu verhindern, benutzt man oft ein zweites Register, das sogenannte *Grenzregister*. Jede Adresse, die ein Prozeß anspricht, wird nach der Addition zum Inhalt des Basisregisters mit dem Inhalt des Grenzregisters verglichen. Wenn die Summe aus Adresse und Basisregisterinhalt größer ist als der Wert im Grenzregister, bricht man den Prozeß sofort ab. Ähnlich wie das Basisregister wird auch das Grenzregister vom Betriebssystem beim Laden oder Verschieben des Programms manipuliert.

Natürlich darf dem Anwendungsprogramm kein Zugriff auf Basis- oder Grenzregister gewährt werden, weil dadurch der Schutzmechanismus umgangen werden könnte. Die zur Manipulation dieser Register notwendigen Befehle gelten also als *privilegiert* und sind somit nur im Kernmodus ausführbar.

Viele Prozesse sollen bei beschränkter Haupspeichergröße gleichzeitig aktiv sein:

Der nächste Schwierigkeitsgrad entsteht, wenn man mehr aktive Prozesse zulassen will, als gleichzeitig in den Hauptspeicher passen. Zunächst scheint das unmöglich, aber man kann ja den Scheduler so programmieren, daß er nicht nur Prozesse auswählt, denen Rechenzeit zur Verfügung gestellt wird, sondern diese Prozesse bei Bedarf auf einen Hintergrundspeicher (in der Regel ein Plattenlaufwerk) aus- und von dort wieder einlagert, so daß sich jeweils mindestens der Prozeß im Hauptspeicher befindet, der gerade rechnet. Soviele andere Prozesse wie nötig werden aus dem Hauptspeicher auf den Hintergrundspeicher transferiert und von dort wieder geladen, bevor ihnen der Prozessor zugeteilt wird. Dieses Verfahren nennt man *Swapping* (Ein-/Auslagerung), den dafür verwendeten Teil des Hintergrundspeichers *Swap Area* oder *Swap File*.

Die Methoden zur Auswahl des auszulagernden Prozesses basieren meist auf einer Vielzahl von Kriterien, zum Beispiel seiner Größe und Priorität, der Anzahl und Prioritäten anderer Prozesse im System usw.

Prozesse mit sehr viel Hauptspeicherbedarf sollen möglich sein:

Was ist jedoch, wenn man zulassen will, daß ein einzelner Prozeß mehr Hauptspeicher benötigt als der vorhandene Rechner besitzt? Zunächst half man sich damit, jeweils einen Teil des Programms zu laden und so die nacheinander zur Ausführung kommenden Programmteile nicht gleichzeitig, sondern nacheinander in demselben Hauptspeicher zu halten. Diese

Überlagerung (englisch *Overlay*) genannte Technik hielt sich recht lange, hatte aber den Nachteil, daß der Programmierer diese Vorgänge organisieren mußte: das Nachladen der benötigten Routinen geschah zwar meist automatisch, aber der Programmierer mußte festlegen, welche Programmabschnitte sich welche Speicherbereiche teilen sollten, und dabei galt es, bestimmte Regeln zu beachten.

Schon zu Beginn der 1960er Jahre entstand jedoch die Idee, die Auswahl dem Betriebssystem zu überlassen. Die von J. Fotheringham 1961 vorgeschlagene Methode wurde unter dem Namen *virtueller Speicher* bekannt und geht von folgenden Grundideen aus: Die Größe eines Programms und seiner Daten darf die Größe des verfügbaren Hauptspeicher übersteigen. Das Betriebssystem hält dann die gerade in Benutzung befindlichen Teile des Programms und der Daten im Hauptspeicher und den Rest auf einer Platte. Greift das Programm auf einen Teil (Programm oder Daten) zu, der sich gerade auf der Platte befindet, muß das Betriebssystem einen anderen Teil auslagern, dadurch Platz im Hauptspeicher freimachen und darin den benötigten Teil des Programms laden, bevor der eigentliche Zugriff erfolgt. Diese Technik heißt *Paging*.

Technische Realisierung des virtuellen Speicherkonzeptes:

Die technische Realisierung stützt sich auf besondere Hardware zur Übersetzung der virtuellen Speicheradressen in Adressen des realen Speichers: die *Speicherverwaltungseinheit* (*Memory Management Unit*, abgekürzt *MMU*). Dies ist schon deshalb notwendig, weil die virtuellen Adressen einen größeren Bereich überstreichen können als den tatsächlich vorhandenen Hauptspeicher: die ersten Exemplare der VAX-Rechner hatten zum Beispiel einen virtuellen Adreßraum von 4 GB (2^{32} Bytes), waren aber meist nur mit etwa 1 MB (2^{20} Bytes) Hauptspeicher ausgerüstet. Um den virtuellen Adreßraum in den realen abzubilden, teilt man den Speicher in *Seiten* ein, d.h. in Einheiten einer festen Größe von z.B. 512 Bytes oder 8 kB. Für jede Seite im Adreßraum eines Prozesses muß dann die Information verfügbar sein, ob sich die Seite gerade im realen Hauptspeicher befindet oder ausgelagert ist, und für die im physikalischen Speicher liegenden Seiten muß festgehalten sein, welche physikalische Seite (man sagt hier *Seitenrahmen*) die Seite aus dem virtuellen Adreßraum enthält. Bild 8-13 zeigt ein Beispiel, in dem ein virtueller Adreßraum von 64 kB in Seiten zu 4 kB eingeteilt und auf einen physikalischen Hauptspeicher von 32 kB abgebildet wird. Der virtuelle Speicher besteht dann aus 16 Seiten ($16\ \text{Seiten} \times 4\ \text{kB} = 64\ \text{kB}$), und es gibt 8 Seitenrahmen im physikalischen Hauptspeicher (denn 8 Rahmen x 4KB = 32KB).

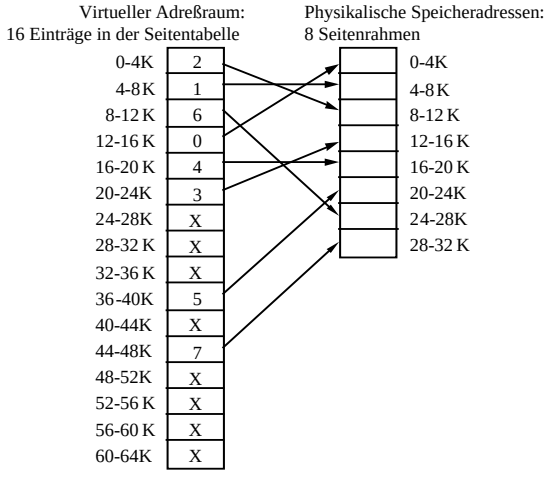

Abbildung 8-13: Funktion einer Memory Management Unit

Die Abbildung der virtuellen zu den physikalischen Seiten wird technisch durch eine *Seitentabelle* realisiert, in der es für jede Seite einen Eintrag gibt, der entweder anzeigt, daß die Seite ausgelagert ist (Bild 8-13: X), oder in welchem Seitenrahmen sie sich befindet (Bild 8-13,0...7).

Man kann nun die einzelnen Bits einer virtuellen Adresse in zwei Gruppen gliedern, von denen eine die Nummer der Seite, die andere Gruppe die Speicherstelle innerhalb der Seite bezeichnet. Die Nummer der Seite dient dann als Index in die Seitentabelle, aus der die Nummer des Seitenrahmens entnommen wird, in dem sich die Seite gerade befindet. Zusammen mit den unverändert übernommenen Bits der zweiten Gruppe ergibt sich die physikalische Adresse (siehe Bild 8-14).

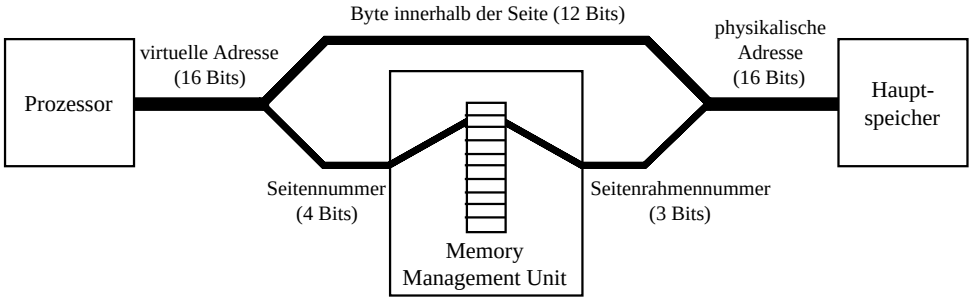

Abbildung 8-14: Adreßmodifikation in der MMU

Bei jedem Zugriff auf eine Seite, die sich nicht im Hauptspeicher befindet, muß das Programm unterbrochen und die gewünschte Seite eingelagert werden, bevor der Prozessor die entsprechende Instruktion ausgeführen kann. Diese Situation heißt *Seitenfehler* (*Page Fault*). Da im allgemeinen kein Seitenrahmen frei ist, muß vor dem Einlagern der benötigten Seite eine andere Seite ausgewählt und auf die Platte ausgelagert werden. Daher sind Seitenfehler recht aufwendige Situationen, die sehr effiziente Programme möglichst vermeiden.

8.6 Dateisysteme und Ein-/Ausgabe

Zum Abschluß unserer Betrachtungen über die Grundlagen von Betriebssystemen wollen wir uns noch mit dem inneren Aufbau von Dateien und einigen weiteren, wichtigen Details über Dateisysteme, Ein- und Ausgabe beschäftigen.

Eine Datei ist – das hatten wir schon festgehalten – ein Behälter für Daten. Jede Anwendung strukturiert ihre Daten in einer bestimmten Weise, und so liegt es nahe, nach der Struktur von Dateien zu fragen. Beginnen wir mit einer sehr einfachen Struktur, der *Stream-Datei*: solche Dateien sind aus der Sicht des Betriebssystems einfach eine Folge von Bytes ohne weitere Unterteilung. Die Anwendung ist für die Interpretation dieser Folge zuständig und muß daher selbst für die Verwaltung von Strukturmerkmalen sorgen.

Ein erster Schritt zu einer weiteren Strukturierung ist die Aufteilung von Dateien in *Sätze (Records)* mit fester Länge. Man faßt jeweils eine feste Anzahl von Bytes zu einer Gruppe zusammen. Gerade bei älteren Betriebssystemen ist dieser Ansatz weit verbreitet, oft mit Satzlängen, die die Größe der verwendeten Plattenblöcke ganzzahlig teilen.

Zur Speicherung einfacher Texte eignet sich besser eine Struktur mit variabler statt fester Satzlänge: man speichert jede Zeile als einen Satz und wählt die Satzlänge für jede Zeile so, daß der Satz gerade die ganze Zeile aufnimmt. Das Dateisystem muß nun in der Lage sein, die Länge jedes Satzes bestimmen zu können. Das kann man durch die Vereinbarung von Trennzeichen erreichen, also beispielsweise durch die Festlegung, daß zwei Sätze (bei Textspeicherung also zwei Zeilen) durch das ASCII-Steuerzeichen LF getrennt werden. Allerdings darf dann der Satz kein Byte mit dem numerischen Wert 10 enthalten, denn das wäre die Kodierung für LF und damit das Zeilenende. Daher verwendet man oft auch eine andere Methode, bei der *vor* jedem Satz, für Anwendungsprogramme unsichtbar, eine Längenangabe gespeichert wird. Das Zeilenende ist dann unabhängig vom Inhalt der folgenden Bytes erst erreicht, wenn die angegebene Satzlänge abgearbeitet ist.

Weil alle bisher besprochenen Strukturen sich vor allem für die sequentielle Verarbeitung eignen, also für Lesen oder Schreiben der aufeinanderfolgenden Bytes oder Sätze vom Anfang der Datei in der gespeicherten Reihenfolge in Richtung Dateiende, nennt man diese Strukturen *sequentielle* Organisationsformen.

Eine unter UNIX und Windows Systemen eher seltene Dateistruktur wird im Bereich der kaufmännischen Datenverarbeitung oft benutzt, um eine Funktionalität zu erreichen, die gelegentlich den Einsatz einer Datenbank ersparen kann: *indexsequentielle* Dateien sind kein linearer Strom von Daten, sondern eine baumartige Struktur, wie in Bild 8-15.

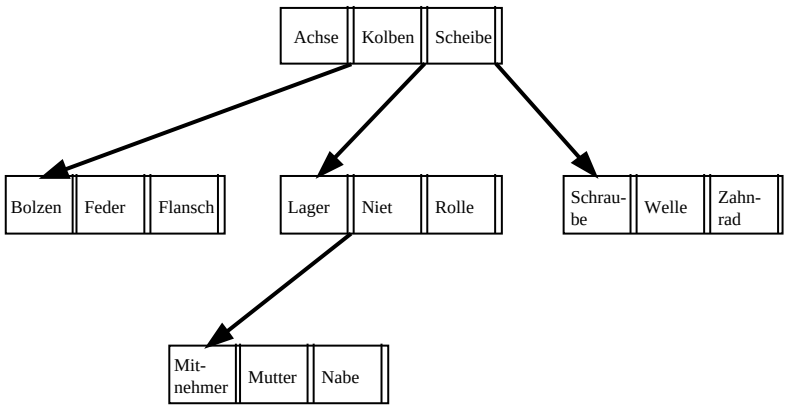

Abbildung 8-15: Struktur einer indexsequentiellen Datei

Wesentlich ist hierbei, daß jeder Satz in einer solchen Datei an einer festen Stelle des Satzes einen eindeutigen *Schlüssel* enthält: dieser Teil des Satzes bestimmt seine Position innerhalb der Datei, und wenn ein Anwendungsprogramm einen Schlüsselwert vorgibt, kann das Dateisystem den Satz mit dem gegebenen Schlüssel lesen oder einfügen.

Die technische Realisierung benutzt normalerweise bestimmte Blöcke aus der Datei, um darin eine Liste von Indexwerten zu speichern, mit denen Verweise auf die entsprechenden Datensätze verbunden sind. Reicht ein solcher Indexblock nicht aus, so speichert man Verweise auf weitere Indexblöcke anstelle der Verweise auf Daten. Dieses Vorgehen ist in Bild 8-15 durch Pfeile symbolisiert.

Nicht jedes Dateisystem unterstützt alle genannten Strukturen. Braucht eine Anwendung eine Struktur, die nicht vom Dateisystem angeboten wird, so muß die Realisierung der gewünschten Organisationsform durch die Anwendung.

Ergänzend zum Inhalt einer Datei speichert das Dateisystem einen Satz von Attributen. Welche Eigenschaften einer Datei zugeordnet werden, unterscheidet sich erheblich von einem Betriebssystem zum nächsten. Daher wollen wir hier nur einige typische Einträge nennen, die man ihn vielen Dateisystemen finden kann:

Eigentümer: Eine Angabe, wem die Datei gehört, in einer für das Betriebssystem gültigen Form.

Erstelldatum: Datum und Uhrzeit, zu der die Datei angelegt wurde.

Änderungsdatum: Datum und Uhrzeit des letzten Schreibzugriffs auf die Datei.

Zugriffsdatum: Datum und Uhrzeit des letzten Lese-Zugriffs auf die Datei.

Größe: Entweder die Anzahl der Bytes oder der Blöcke, aus denen die Datei besteht.

Schutzcode: Erlaubt oder entzieht bestimmten Benutzergruppen bestimmte Zugriffe auf die Datei (siehe folgender Text).

Das zuletzt erwähnte Attribut, der Schutzcode, findet sich in allen Mehrbenutzersystemen. Hierunter versteht man einen Mechanismus, der es dem Dateieigentümer erlaubt, bestimmte Arten von Zugriffen auf seine Datei für bestimmte Benutzer(-gruppen) zu erlauben oder zu verbieten. Wir wollen ihn am Beispiel von Standard UNIX betrachten. Hier ordnet der Systemadministrator jedem Benutzer eine oder mehrere Gruppen zu. Beim Anlegen einer Datei wird diese mit Benutzer- und Gruppenkennung des Benutzers markiert, der die Datei erstellt hat. Will nun ein Benutzer auf eine Datei zugreifen, so werden zunächst seine Benutzer- und Gruppenkennung mit den Attributen der Datei verglichen. Nach dem Ergebnis des Vergleichs unterscheidet man drei Fälle:

1. Die Benutzerkennungen stimmen überein: dann ist der Benutzer der Eigentümer der Datei.
2. Der Benutzer gehört der Gruppe an, deren *gid* als Dateiattribut eingetragen ist: dann ist der Benutzer nicht selbst Eigentümer der Datei, er befindet sich aber mit dem Eigentümer in einer gemeinsamen Gruppe.
3. Weder Benutzer- noch Gruppenkennung stimmen überein: dann hat der zugreifende Benutzer nichts mit dem Eigentümer gemeinsam und wird in eine Kategorie für *andere Benutzer* eingeordnet.

Für jeden dieser drei Fälle kann der Eigentümer einer Datei drei Arten von Zugriffen erlauben oder verbieten:

Lesen (Read): Die Datei darf zum Lesen geöffnet, aber nicht gekürzt, verlängert oder überschrieben werden.

Schreiben (Write): Die Datei darf sowohl zum Lesen als auch zum Schreiben geöffnet, also beliebig verändert werden.

Ausführen (Execute): Das in der Datei enthaltene Programm darf zur Ausführung gebracht werden.

Jeder dieser Zugriffsarten für jede Fallgruppe ist ein eigenes Bit in den Dateiattributen zugeordnet. Hat dieses Bit den Wert 1, so ist den entsprechenden Benutzern diese Art von Zugriff erlaubt, hat das Bit den Wert 0, ist der Zugriff verboten. Damit ergeben sich insgesamt neun Bits, die für Eigentümer, Gruppenmitglieder und andere Benutzer jeweils Lese-, Schreib- und Ausführrechte definieren. Anhand der englischen Abkürzungen für die Zugriffsarten nennt man die Bits auch rwx-Bits.

Beispiel 8-14: Der UNIX-Befehl `chmod` erlaubt die Manipulation der rwx-Bits. Dazu gibt man als Optionen an, welchen Benutzer welche Rechte gewährt oder entzogen werden sollen, und bei welcher Datei die Bits gesetzt werden sollen. Mit dem Befehl

```
chmod g+x tolles-skript
```

erteilt man Benutzern der zweiten Fallgruppe (`g` für Group) das Recht, die Befehle in der Datei `tolles-skript` ausführen zu lassen. Der Befehl

```
chmod o+r gags
```

erlaubt allen Benutzern (o für others) das Lesen der Datei `gags`. Mit

```
chmod u-w liste
```

entzieht sich der Eigentümer (u für user) selbst das Schreibrecht für die Datei `liste`, schützt diese also vor unbeabsichtigtem Löschen.

Die Gruppen eignen sich für die Abbildung organisatorischer Gruppen, etwa Abteilungen einer Firma, auf Objekte der Rechnerwelt: mit den rwx-Bits und einer der Abteilungsstruktur angeglichenen Gruppenstruktur kann zum Beispiel ein Projektbericht den Schutzcode `rw-r-----` erhalten, das bedeutet für den Eigentümer (Autor) Schreib- und Leserecht, für Mitglieder derselben Gruppe (Abteilung, Projekt) nur Leserecht und für alle anderen Benutzer (außerhalb der Abteilung, des Projekts) keinerlei Rechte.

8.7 Fallbeispiel: UNIX

Wohl kein anderes Betriebssystem kann auf so vielen unterschiedlichen Rechnerarchitekturen verwendet werden wie UNIX. Seine Dominanz liegt im Minicomputer- und Workstationbereich. Man findet UNIX aber außerdem auf Notebooks, PCs, Servern, Großrechnern und Supercomputern.

8.7.1 Die Geschichte von UNIX

Die Entwicklung von UNIX wird gerne auf dieselben Leute zurückgeführt, die beim Massachusetts Institute of Technology (MIT) die Idee zum Time-Sharing hatten. Dort war das MULTICS[1]-System entstanden, ein Mehrbenutzersystem für eine Hardware in der Leistungsklasse eines IBM PC/AT. Nachdem sich Bell aus dem Projekt zurückgezogen hatte, entschloß sich einer der Entwickler, Ken Thompson, eine abgespeckte Version von MULTICS zu schreiben, die er in Assembler auf einer PDP-7 kodierte. Brian Kernighan prägte etwas scherzhaft den Namen UNICS, das schließlich in UNIX umbenannt wurde.

Dennis Ritchie unterstützte Kernighan, man ging von der veralteten PDP-7 zunächst auf eine PDP-11/20, später auf PDP-11/70 über. Letztere hatte einen Hardware-Speicherschutz (für mehrere gleichzeitig arbeitende Benutzer) und war der dominierende Minicomputer in den 1970-ern.

Um das Neuschreiben des gesamten Systems bei jedem Umzug von einer Hardware auf eine andere zu vermeiden, wurde das System nicht mehr in Assembler, sondern in B geschrieben, einer eigens von Thompson entwickelten Sprache. Dieser Versuch mißlang wegen Schwächen der Sprache B, daraufhin entwarf Ritchie die Sprache C und einen Compiler dafür. Thompson und Ritchie schrieben UNIX dann in C.

Weil Rechner der PDP-11-Typenfamilie in den Informatik-Fachbereichen der Universitäten weit verbreitet waren und AT&T kaum kommerzielle Interessen an UNIX hatte, konnten viele Universitäten das System zu einem geringen Preis lizenzieren. Der erste Rechner außerhalb der PDP-11-Architektur, auf den UNIX portiert wurde, war einige Jahre später eine Interdata 8/32. Durch die räumliche Trennung zwischen der Entwicklungsmaschine, einer PDP-11 im fünften Stock der Bell Labs, und der Testmaschine 8/32 im ersten Stock begannen die Ingenieure sich für eine elektronische Verbindung der Rechner zu interessieren. Dadurch wurden die Wurzeln für die Vernetzung von UNIX-Rechnern gelegt. Später folgten Portierungen auf VAX und andere Rechner.

Nach Umstrukturierungen gründete AT&T 1984 ein Computerunternehmen. Das erste Produkt war UNIX unter der Versionsbezeichnung System III. Weil dieses Produkt im Markt nicht gut genug ankam, wurde es ein Jahr später durch eine verbesserte Version, System V, ersetzt. Eine Version mit der Nummer IV hat es anscheinend nie gegeben.

Parallel dazu wurde an der Berkeley-Universität die UNIX-Version 6 weiterentwickelt. Eine verbesserte Version für die PDP-11 kam unter dem Namen 1BSD (First Berkeley Software Distribution) heraus, bald darauf 2BSD. Größere Bedeutung erlangten 3BSD und 4BSD für die VAX, denn sie waren besser als die von AT&T vertriebene Version 32V, zum Beispiel durch Unterstützung für virtuellen Speicher mit Paging, ein leistungsfähigeres, schnelleres Dateisystem und Unterstützung für Vernetzung mit TCP/IP, das sich dadurch zu einem De-Facto-Standard entwickelte. Außerdem hatte BSD-UNIX mehr Hilfsprogramme, zum Bei-

[1] MULTiplexed Information and Computing Service

spiel den Editor `vi`, eine neue Shell (`csh`) sowie Compiler für Pascal und Lisp. Das war vielen Herstellern Anlaß, ihre eigenen Varianten von UNIX auf die Berkeley-Variante zu stützen.

Damit waren zwei weitgehend inkompatible UNIX-Versionen entstanden: 4.3BSD und System V Release 3. Nach erfolglosen Versuchen, diese Systeme zu einem gemeinsamen Standard SVID (System V Interface Definition) zusammenzuführen, wurde der POSIX-Standard als IEEE 1003 definiert: POS für Portable Operating System und die Endung IX, um den Namen UNIX-fähig zu machen. Kurz danach gründeten einige Hersteller unter der Führung von IBM, DEC und HP die OSF (Open Software Foundation), um AT&T die Vormachtstellung zu entziehen. OSF wollte ein eigenes, IEEE-kompatibles System mit zusätzlichen Erweiterungen entwickeln: Fenstersystem (X11), graphische Benutzungsoberfläche (Motif), verteilte Verarbeitung (*Distributed Computing Environment, DCE*), verteiltes Management (*Distributed Management Environment, DME*) und mehr.

Die Reaktion von AT&T bestand in der Gründung eines eigenen Konsortiums namens UI (UNIX International). Also gab es wieder zwei mächtige Industriegruppen, von denen jede ihre eigene UNIX-Version anbot. Außerdem entwickelten praktisch alle Hersteller eigene Varianten (zum Beispiel IBM: AIX) mit unterschiedlichen Zielsetzungen.

Allen diesen Systemen ist die Eigenschaft gemeinsam, daß sie groß und kompliziert sind in einer Ausprägung, die in direktem Gegensatz zu den ursprünglichen Ideen von UNIX steht. Erst wenn der Quellcode frei verfügbar ist, was nicht der Fall ist, ist es keine Frage, daß eine einzelne Person alles verstehen könnte. Diese Rolle übernahm zunächst MINIX, mit dem der Versuch unternommen wurde, neue UNIX-ähnliche Systeme zu entwickeln, die klein genug sind, um sie zu verstehen, und die mit dem gesamten Quellcode zur Verfügung stehen. MINIX wurde haupsächlich für Ausbildungszwecke genutzt.

Ein weiteres Beispiel für ein Unixderivat, das für Mikrocomputer entwickelt wurde, ist LINUX. Ursprünglich für IBM kompatible PC mit Prozessoren der Klasse 80386 entwickelt, hat LINUX inzwischen auch den Sprung auf andere Plattformen wie DEC Alpha, Sun Sun Sparc, Apple Macinthosh und Atari geschafft. Es liegt vollständig im Quellcode vor und kann über das Internet kostenlos bezogen werden. LINUX wurde ab dem März 1991 von Linus Benedict Torvalds in Helsinki entwickelt. Da er seine Arbeiten frühzeitig im Internet zur Diskussion stellte, entstand rasch ein Interesse an Mitarbeit, und so wird LINUX inzwischen von einem weltumspannenden Netz von Entwicklern betreut, die ständig Verbesserungen und Ergänzungen vornehmen [HeHM-94]. Mittlerweile ist LINUX ein vollwertiges Betriebssystem geworden, das weitgehend dem Posixstandard entspricht. Außer im privaten und universitären Umfeld steht LINUX mittlerweile auch im professionellen Einsatz. Interssierten Lesern sei die Installation von LINUX empfohlen, weil sich damit einerseits auf einfache Weise die Möglichkeit bietet, die Systemverwaltung und Benutzung eines Unixsystems kennenzulernen und einzuüben, und weil andererseits dieses System die Hardware heutiger PCs sehr effizient ausnutzt.

8.7.2 Ziele von UNIX

UNIX wurde von Programmierern für (fortgeschrittene!) Programmierer entworfen. Daher enthält es eine Vielzahl von Hilfsmitteln für Entwurf und Entwicklung von Software im Team. Die Benutzung der Werkzeuge verzichtet auf jede Redundanz, es gibt also keine umständlichen Dialoge, sondern knapp abgekürzte Kommandos wie `ls` für *list files*, `cp` für *copy* oder `rm` für *remove* (entferne, das heißt lösche eine Datei).

8.7.3 Schnittstellen zu UNIX

Man kann sich ein UNIX-System aus Schichten aufgebaut vorstellen, wie in Bild 8-16 dargestellt.

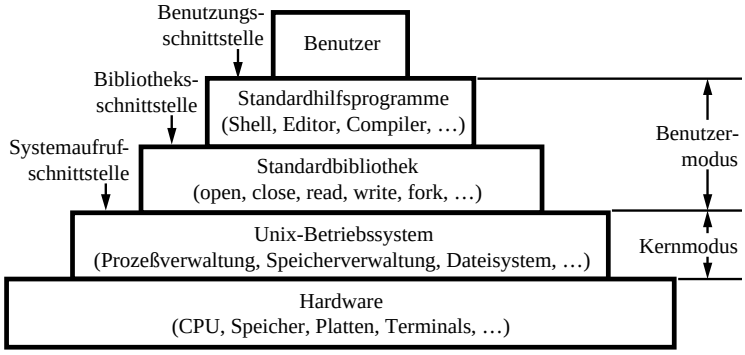

Abbildung 8-16: Die Schichten eines UNIX*-Systems*

Unten im Bild ist die Hardware zu erkennen: CPU, Speicher, Platten, Terminals und andere Geräte. Darauf läuft das Betriebssystem UNIX im Kernmodus. Es stellt Aufrufschnittstellen zur Verfügung, die entweder direkt oder in Routinen der Standardbibliothek verwendet werden können. Durch die Systemaufrufe können Anwendungsprogramme die beschriebenen Operationen zur Manipulation von Prozessen, Dateien und anderen Betriebsmitteln anfordern.

Für Programmierer, die mit höheren Programmiersprachen arbeiten, sind Aufrufmechanismen wie TRAP-Befehle nicht direkt zugänglich. Um dennoch den Zugang zu Systemaufrufen zu ermöglichen, gibt es die Standardbibliothek: diese Sammlung von Unterprogrammen enthält für jeden Systemaufruf eine Routine, die von C-Programmen aufgerufen werden kann und dabei die notwendigen Parameter übernimmt, an die geeigneten Stellen überträgt und dann einen TRAP ausführt.

Der POSIX-Standard definiert, welche Schnittstellen die Systembibliothek bereitstellen muß, und nicht, welche Systemaufrufe vorhanden sein müssen. Die Dienstprogramme stellen einen Zugang dar, der das Anstoßen von Operationen mit Befehlen von einem Terminal aus erlaubt. Damit bilden die Dienstprogramme die Oberfläche, über die der Benutzer mit dem System kommuniziert. Dies kann sowohl über die traditionelle Tastatur mit Textterminals (Shell als Kommandointerpretierer) als auch über eine Maus (graphische, meist auf X-Windows basierende Oberfläche) geschehen.

8.7.4 Zugang zu UNIX

Wie bei Timesharing-Systemen üblich, beginnt die Benutzung eines UNIX-Rechners mit einer Anmeldung, durch die der Anwender gegenüber dem Rechner seine Nutzungsberechtigung nachweist. Dazu fragt der Rechner einen Benutzernamen ab, der zuvor vom Systemadministrator vergeben wurde, und vergleicht ein ebenfalls einzugebendes Kennwort (*Paßword*) mit dem gespeicherten Schlüssel. Gehören das eingegebene Kennwort und der Benutzername zusammen, startet UNIX entweder eine Shell oder eine graphische Benutzungsoberfläche. Im Programmierkurs zu dieser Vorlesung stehen Workstations bereit, die mit verschiedenen UNIX-Systemen betrieben werden.

8.7.5 Das Dateisystem von UNIX

Aus den weiter oben gezeigten Beispielen geht bereits hervor, daß UNIX ein hierarchisch strukturiertes Dateisystem besitzt. Zur leichteren Orientierung wollen wir deshalb hier nur noch einige wichtige Verzeichnisse beschreiben, die in den meisten UNIX-Versionen zu finden sind. Solche Angaben enthält Tabelle 8-1.

Verzeichnis	Beschreibung
`/bin`	Häufig benutzte Binärprogramme
`/dev`	Spezialdateien für Ein-/Ausgabegeräte
`/etc`	Verschiedenes für die Systemadministration
`/lib`	Häufig benutzte Bibliotheken
`/tmp`	Temporäre Dateien
`/usr`	Wurzelverzeichnis für den Teilbaum mit Benutzerdateien
`/usr/adm`	Systemabrechnung
`/usr/bin`	Weitere Binärprogramme
`/usr/include`	Header-Dateien für Systemschnittstellen
`/usr/lib`	Bibliotheken, Compilerschritte, Verschiedenes
`/usr/man`	Online-Manuals
`/usr/spool`	Spooling-Verzeichnis für Drucker

Tabelle 8-1: Einige wichtige UNIX*-Verzeichnisse*

8.7.6 UNIX-Hilfsprogramme

Von den Hilfsprogrammen, die zu UNIX gehören, war schon des öfteren die Rede. Hier sollen am Beispiel von POSIX einige der *Utilities*, wie diese Programme englisch genannt werden, aufgezählt und kurz charakterisiert werden.

Tabelle 8-2 nennt einige wichtige Hilfsprogramme. Das Besondere ist, daß man durch Verbindung dieser Programme (etwa in einer Pipeline) sehr viele unterschiedliche Dinge realisieren kann, ohne allzu viele verschiedene Bausteine zu verwenden. Dieses Baukastenprinzip macht einen großen Teil der Leistungsfähigkeit von UNIX aus. Um diese Leistungsfähigkeit zu nutzen und solche Bausteine zu funktionierenden Lösungen zu kombinieren, muß man aber die notwendigen Kenntnisse besitzen. Genau diese Kenntnisse bringen Programmierer mit, und daher ist UNIX gerade bei ihnen so beliebt.

Auf den im Umgang mit Rechnern weniger geübten Anwender wirken die UNIX-Kommandos durch ihre Knappheit oft kryptisch und schwer verständlich, aber ein gutes Anwendungssystem wird dem Benutzer davon nicht viel zeigen, sondern nach dem Einloggen direkt das Anwendungsprogramm starten und den Benutzer innerhalb der Anwendung zu allen benötigten Operationen führen. Welches Betriebssystem die Grundlage einer solchen Anwendung bildet, spielt kaum noch eine Rolle, weil es fast keine Auswirkungen auf die Oberfläche der Anwendung hat.

`Ar`	Aufbau und Wartung von Archiven und Bibliotheken in mehreren Dateien
`Awk`	Sprache zur Zeichenkettenverarbeitung (pattern matching)
`Basename`	Abschneiden von Präfixen und Suffixen von Dateinamen
`Bc`	Programmierbarer Taschenrechner
`Cat`	Konkatenieren und Ausgeben von Dateien auf die Standardausgabe
`Cc`	Compilieren eines C-Programms
`Chmod`	Verändern des Schutzmodus für Dateien
`Cmp`	Vergleich von Dateien auf Gleichheit des Inhalts
`Comm`	Ausgabe gemeinsamer Zeilen zweier sortierter Dateien
`Cp`	Kopieren einer Datei
`Cut`	Erzeugt für jede Spalte eines Dokuments eine eigene Datei
`Date`	Ausgabe von Datum und Uhrzeit
`Dd`	Kopieren einer Datei unter Einbeziehung verschiedener Konventionen
`Diff`	Ausgabe aller Unterschiede zwischen zwei Dateien
`Echo`	Ausgabe seiner Argumente (meistens in Shell-Skripten)
`Ed`	Zeilenorientierter Texteditor
`Find`	Aufsuchen aller Dateien, die eine gegebene Bedingung erfüllen
`Grep`	Durchsuchen einer Datei nach Zeilen mit einem gegebenen Muster
`Head`	Ausgabe der ersten Zeilen einer oder mehrerer Dateien
`Kill`	Senden eines Signals an einen Prozeß
`Ln`	Erzeugen eines Links (einer Verbindung) auf eine Datei
`Lp`	Ausgabe einer Datei auf dem Zeilendrucker
`Ls`	Auflisten von Dateien
`Make`	Recompilieren veränderter Teile eines großen Programmes
`Mkdir`	Erzeugen eines Verzeichnisses
`Mv`	Bewegen oder Umbenennen einer Datei
`Od`	Erstellen eines oktalen Abbildes (octal dump) einer Datei
`Paste`	Kombination mehrerer Dateien als Spalten einer Datei
`Pr`	Formatieren einer Datei für die Ausgabe
`Pwd`	Ausgabe des aktuellen Arbeitsverzeichnisses
`Rm`	Löschen einer Datei
`Rmdir`	Löschen eines Verzeichnisses
`Sed`	Zeichenstromeditor (nicht interaktiv)
`Sh`	Aufruf der Shell
`Sleep`	Suspendierung der Ausführung für eine gegebene Zeit (Sekunden)
`Sort`	Sortieren der Zeilen einer Datei
`Stty`	Setzen von Terminaleigenschaften
`Tail`	Ausgabe der letzten Zeilen einer Datei
`Tee`	Kopieren der Standardeingabe zur Standardausgabe und auf eine Datei
`Tr`	Übersetzen von Zeichencodes
`Uniq`	Löschen identischer aufeinanderfolgender Zeilen einer Datei
`Wc`	Zählen von Zeichen, Wörtern und Zeilen einer Datei

Tabelle 8-2: Von POSIX geforderte UNIX*-Hilfsprogramme*

Beispiel 8-15 Der `ls`-Befehl gibt anders als das vergleichbare `DIR`-Kommando anderer Betriebssysteme keine Zusammenfassung aus, in der die Anzahl der aufgezählten Dateien steht. Sucht man diese Anzahl, so erhält man sie leicht mit `ls | wc -l`.

8.8 Fallbeispiele: MS-DOS und WINDOWS

Anders als UNIX eignet sich MS-DOS nur zum Betrieb auf Prozessoren des Typs Intel 8088 oder dessen Nachfolgern Intel 80286, 80386, 80486 oder Pentium. Die Betriebssysteme WINDOWS95 und WINDOWS98 wurden ebenfalls von der Firma Microsoft entwickelt, und dienen als Nachfolgesysteme von MS-DOS.

8.8.1 Die Geschichte von MS-DOS und WINDOWS

Der erste Personalcomputer war der Altair, der 1975 MITS in Albuquerque, New Mexico, produziert wurde. Er war mit einer 8-Bit-CPU Intel 8080 und 256 Byte Speicher (RAM) ausgestattet, hatte keine Tastatur, Bildschirm, Band oder Platte.

Im Jahre 1980 entschied sich IBM für ein Engagement im Personalcomputerbereich. Intel hatte inzwischen als Nachfolger des 8080 den 8086 als 16-Bit-Architektur entwickelt. IBM wählte den 8086, weil die von ihm unterstützten Chips (Schnittstellenbausteine) deutlich kostengünstiger waren. Auch bei der Software hatte IBM kein Interesse an einer Eigenentwicklung. Man wußte, daß im Personalcomputerbereich die Sprache BASIC sehr beliebt war, und so fragte man Bill Gates, der inzwischen eine Firma namens Microsoft gegründet hatte, um eine Lizenz für seinen BASIC-Interpreter für den IBM-PC zu bekommen. Ebenso wurde Gates nach einem Betriebssystem gefragt. Gates kannte ein Unternehmen, das ein Betriebssystem namens 86-DOS entwickelt hatte. Also kaufte Microsoft 86-DOS, benannte es um in MS-DOS und lieferte es planmäßig an IBM aus, so daß es bei der Ankündigung des IBM-PC im August 1981 mit dabei war.

Der IBM-PC war ursprünglich für den Einsatz im Wohnzimmer gedacht. Die Taktfrequenz war mit 4,77 MHz so gewählt, daß Fernsehgeräte anstelle eines Monitors verwendet werden konnten. Es war Hardware zur Unterstützung von Audio-Kassettenlaufwerken als Speichermedium vorhanden. Der Intel 8088 kann 1 MByte Arbeitsspeicher adressieren, beim IBM-PC waren davon aber nur die ersten 640 kByte für Speicher vorgesehen, der Rest wurde mit ROM, Video-Karten und anderen Dingen belegt. Folglich konnte MS-DOS auch nur Programme bis zu einer Größe von 640 kByte unterstützen. Das war zunächst kein Problem, weil der Rechner nur 64 kByte RAM hatte. Als aber später Modelle mit bis zu 64 MByte auf den Markt kamen, wurde diese Beschränkung eine wesentliche Hürde.

Da IBM die komplette Architektur des PC offenlegte, konnten andere Hersteller sich anschließen und ihrerseits sowohl Software als auch Hardware für dieses System anbieten. Das System bestand vollständig aus Standard-Komponenten, die man in jedem Elektronikgeschäft kaufen konnte und so der Nachbau des gesamten Systems kein Problem war. Diese sogenannten Clones trugen zu dem durchschlagenden Erfolg wesentlich bei, den der IBM-PC und mit ihm MS-DOS hatte.

Eine weitere wichtige Eigenschaft des IBM-PC ist, daß er keinen Hardware-Schutz kennt. Er unterscheidet also auch nicht zwischen Benutzer- und Kernmodus, so daß jedes Programm unter Umgehung des Betriebssystems direkt auf die Hardware zugreifen kann. Dies wird häufig benutzt und hat zu einer Vielzahl schlecht geschriebener, nicht portabler Programme geführt. Der IBM-PC/XT mit dem Prozessor Intel 8086 wurde 1982 eingeführt und verfügte über eine Festplatte. Mit ihm kam die Version 2.0 von MS-DOS. Diese war von Grund auf neu entwickelt und hatte viele Ideen von UNIX übernommen, zum Beispiel die Umlenkung der Ein- und Ausgabe und Pipelines. Durch die Festplatte konnten auch größere Anwendungen betrieben werden, was den PC/XT für den geschäftlichen Einsatz interessant machte. Zu dieser Zeit wurde MS-DOS im Hause Microsoft von nur vier Leuten gewartet.

Im August 1984 kam IBM mit dem PC/AT auf den Markt. Der Intel 80286-Prozessor dieses Rechners unterstützte bis zu 16 MByte Arbeitsspeicher, kennt einen Benutzer- und einen Kernmodus und unterstützt durch diese und andere Eigenschaften die simultane Abarbeitung mehrerer Programme. MS-DOS Version 3.0 unterstützte nichts von alledem und betrieb den Prozessor in einem Modus, der den 8086 mit gesteigerter Geschwindigkeit simulierte. Der Code umfaßte inzwischen 40.000 Assemblerzeilen und wurde von 30 Leuten gewartet.

Die Version 4.0 brachte Unterstützung für Festplatten mit bis zu 2 GByte Kapazität (FAT16), RAM-Platten mit bis zu 16 MByte im erweiterten Speicher und eine menüorientierte Shell. Im April 1991 wurde MS-DOS Version 5.0 angekündigt, eine wesentlich erweiterte Version, die vollständig den erweiterten Speicher nutzte, von dem die 286 und 386 oft mehrere Megabytes hatten. MS-DOS 6.0 brachte Ende 1993 neue Dienstprogramme, Befehle und Optionen.

Der Wunsch vieler Benutzer, eine einfach zu bedienende graphische Oberfläche zu verwenden, führte schließlich zur Entwicklung des Programmes Windows, das in der Version 3.1 unter MS-DOS läuft und sich sehr verbreitete. Da jedoch Windows nicht in der Lage ist, die designbedingten Probleme mit MS-DOS zu lösen und sehr instabil war, entwickelte Microsoft das Betriebssystem WINDOWS 95, das eine neue graphische Benutzeroberfläche bietet, die die vorhandene PC-Hardware gut ausnutzt und Netwerkfähigkeiten integriert hat.

WINDOWS 95 ist ein Einbenutzersystem ohne Sicherheitsmechanismen, das lediglich ein sogenanntes *non-preemptive multitasking* erlaubt, d.h. daß zwar mehrere Prozesse gleichzeitig im Speicher sein können, der Benutzer jedoch (per Mausklick) entscheidet, welcher Prozeß gerade aktiv (rechnend) ist. Für den Nachfolger, WINDOWS 98, der seine Markteinführung Mitte 1998 hatte, gelten die gleichen Eigenschaften. Lediglich ist seine Ausrichtung stärker auf den Internet Explorer von Microsoft gerichtet.

8.8.2 Die MS-DOS -Shell

Sowohl UNIX als auch MS-DOS unterscheiden zwischen *internen* und *externen* Kommandos: die internen werden von der Shell selbst ausgeführt, während die externen durch eigenständige Programme realisiert sind. Anders als bei UNIX sind aber bei MS-DOS viele Dateioperationen als interne Kommandos implementiert, um bei langsamen Zugriffe einzusparen. Einige der internen MS-DOS-Kommandos sind in Tabelle 8-4 ihren UNIX-Äquivalenten gegenübergestellt.

MS-DOS - Kommando	UNIX-Äquivalent	Beschreibung
COPY	Cp	Kopiert eine oder mehrere Dateien
DATE	Date	Anzeige oder Änderung des aktuellen Datums
DEL	rm	Löscht eine oder mehrere Dateien
DIR	Ls	Listet Dateien und Verzeichnisse auf
MKDIR	mkdir	Erzeugt ein neues Verzeichnis
REN	mv	Änderung des Namens
RMDIR	rmdir	Löscht ein leeres Verzeichnis
TYPE	cat	Ausgabe einer Datei

Tabelle 8-4: Einige der internen MS-DOS -Kommandos

Die Interpretation der Platzhalter erledigt bei MS-DOS nicht die Shell selbst, sondern die von ihr gestarteten Kommandos. Dadurch kann man zwar beispielsweise mit `DIR *.C` erfahren, daß nur die Datei `PROGRAMM.C` die gesuchte Endung besitzt. Gibt man dann aber `EDIT *.C` ein, um den Editor mit dieser Datei aufzurufen, so wird dieser erwidern, `*.C` sei kein zulässiger Dateiname, und terminieren. Der Grund für dieses zunächst überraschende Verhalten liegt darin, daß bei MS-DOS die Kommandozeile unverändert an das Programm übergeben wird, während die UNIX-Shell vorher die Platzhalter expandiert und dadurch im obigen Beispiel die Ersetzung von `*.c` zu `programm.c` bewirkt. Bei MS-DOS muß also der Benutzer wissen, welche Kommandos Platzhalter zulassen und welche nicht. Problematisch daran ist nicht die Durchführung der Expansion, sondern die Tatsache der inkonsistenten Handhabung.

Andererseits bringt die strikte Weitergabe der Befehlszeile auch Vorteile: Unter MS-DOS kann man leicht mit `REN *.C *.OLD` alle Dateien mit der Endung `.C` in `.OLD` umbenennen, unter UNIX ist das nicht mit einem einfachen `mv`-Befehl möglich.

8.8.3 Die MS-DOS - und WINDOWS 95/98-Dateisysteme

Viele Eigenschaften des MS-DOS/WINDOWS95/98-Dateisystems entsprechen denen aus UNIX, wie aus Tabelle 8-5 hervorgeht.

Eigenschaft	**UNIX**	**MS-DOS**	**WINDOWS 95/98**
Hierarchisches Verzeichnissystem	Ja	Ja	Ja
Aktuelles Verzeichnis	Ja	Ja	Ja
Absolute und relative Pfadnamen	Ja	Ja	Ja
Verzeichnisse `.` und `..`	Ja	Ja	Ja
Zeichenorientierte Spezialdateien	Ja	Ja	Ja
Blockorientierte Spezialdateien	Ja	Ja	Ja
Länge der Dateinamen	14 oder 255	8+3	8+3 bzw. 255
Trennzeichen für Pfade	/	\	\
Ist `a` dasselbe wie `A`?	Nein	Ja	Ja
Eigentümer, Gruppen, Schutz	Ja	Nein	Nein
Links (Verweise)	Ja	Nein	Nein
Verknüpfte Dateisysteme	Ja	Nein	Nein
Dateiattribute	Nein	Ja	Ja

Tabelle 8-5: Vergleich der Dateisysteme von UNIX und MS-DOS

Der Eintrag „Länge der Dateinamen" in Tabelle 8-5 weist in der Spalte für MS-DOS den Wert 8+3 aus. Gemeint ist, daß Namen im MS-DOS -Dateisystem aus zwei Komponenten bestehen, von denen die eine bis zu acht, die andere bis zu drei Zeichen lang sein kann. Die erste Komponente ist der Name an sich, der vom Benutzer frei gewählt wird. Der zweite Teil, die *Namenserweiterung*, charakterisiert den Inhalt der Datei. Typische Namenserweiterungen sind in Tabelle 8-6 aufgeführt. Diese Konventionen gelten auch für Windows95/98, jedoch ist man hier bei der Wahl der Dateinamen freier. Hier ist es möglich bis zu 255 Zeichen und Leerzeichen im Dateinamen zu verwenden.

Erweiterung	Inhalt der Datei
.ASM	Programm-Quelltext in Assembler
.BAK	Sicherungskopie eines Editors
.BAS	Programm-Quelltext in der Sprache BASIC
.BAT	(Batch) Kommandofolge
.C	Programm-Quelltext in der Sprache C
.COM	Programm zur Ausführung eines Kommandos
.DOC	Dokumentation
.EXE	Ausführbares Programm
.FOR	Programm-Quelltext in der Sprache FORTRAN
.LST	Liste (z. B. von einem Compiler)
.OBJ	Objektdatei (übersetztes Programm)
.PAS	Programm-Quelltext in der Sprache Pascal
.SYS	Allerlei für das Betriebssystem
.TXT	Text, allgemein

Tabelle 8-6: Typische Namenserweiterungen für MS-DOS -Dateien

Meist kann man bei konventionsgemäß gewählten Namenserweiterungen auch auf die Art des Inhalts schließen: Dateien mit der Endung `.TXT`, `.PAS` oder `.BAT` eignen sich zur Bearbeitung mit einem Texteditor, dagegen sollte man das bei Dateien mit der Endung `.OBJ`, `.COM` oder `.EXE` besser nicht versuchen. Durch die Verwendung des Punktes als Trennzeichen zwischen Name und Erweiterung darf weder im Namen noch in der Erweiterung ein Punkt vorkommen. Hierin besteht ebenfalls ein Unterschied zu UNIX: weil der Punkt im Dateisystem von UNIX wie ein Buchstabe behandelt wird, können Dateinamen dort durchaus mehrere Punkte enthalten.

Es gibt nur zwei Eigenschaften, die im MS-DOS und Windows95/98 -Dateisystem vorkommen, aber im UNIX-Dateisystem fehlen: die Äquivalenz zwischen Groß- und Kleinschreibung und bestimmte Dateiattribute. Für die Attribute werden vier Bits pro Datei verwendet, die für folgende Eigenschaften stehen:

Readonly: Die Datei darf nur gelesen, aber nicht beschrieben werden.

Archive: Die Datei wurde nach ihrer letzten Änderung noch nicht archiviert (es wurde noch keine Sicherungskopie angefertigt, das Attribut wird automatisch gesetzt, sobald eine Datei verändert wird).

Hidden: *Versteckte* Dateien werden bei `DIR` nicht mit angezeigt. Kennt man ihre Namen, lassen sie sich jedoch normal handhaben.

System: Es handelt sich um eine Systemdatei, die automatisch von bestimmten Manipulationen ausgeschlossen ist: solche Dateien werden weder durch den `DIR`-Befehl angezeigt noch durch `COPY` kopiert.

Im Gegensatz zum UNIX-Dateisystem betrachtet MS-DOS und WINDOWS 95/98 mehrere Laufwerke, die an einen Rechner angeschlossen sein können, als getrennte Dateisysteme, so daß für den Zugriff auf eine Datei, die sich nicht auf dem aktuellen Laufwerk befindet, eine explizite Laufwerksangabe notwendig ist. Dazu stellt man den Buchstaben, der als Laufwerksname dient, und einen Doppelpunkt vor den Pfad, über den die gewünschte Datei auf diesem Laufwerk zu erreichen ist.

8.9 Fallbeispiel: WINDOWS NT und WINDOWS 2000

8.9.1 Die Geschichte von WINDOWS NT und WINDOWS 2000

Die Ausführungen über MS-DOS und WINDOWS 95/98 haben angedeutet, daß zwar eine weite Verbreitung diesen Systemen eine erhebliche Bedeutung sichert, ihre technischen Eigenschaften jedoch vor allem für Programmierer viel zu wünschen übrig lassen. Andererseits haben die Erfahrungen gezeigt, daß den weitaus meisten Benutzern die technischen Eigenschaften ihres Betriebssystems völlig egal sind, solange nur geeignete Anwendungsprogramme zur Verfügung stehen, die mit der gewohnten Windows-Oberfläche bedient werden können.

Diese beiden Erkenntnisse führten dazu, ein System zu entwickeln, dessen Oberfläche genauso aussieht und funktioniert wie das gewohnte Windows-System, das aber in seinem Kern nichts mehr mit MS-DOS bzw. WINDOWS 95 zu tun hat und dadurch die spezifischen Schwächen vermeiden kann.

Genau dies wollte WINDOWS NT: der Name wurde mit Absicht so gewählt, sodass er Assoziationen weckt; und auch die Benutzungsoberfläche sieht genauso aus, wie es der Windows-Anwender von seinem PC gewohnt ist. Hinter dieser Oberfläche verbirgt sich aber ein völlig neu entwickelter Kern, der moderne Technologie enthält. Dieser Kern hat den zweiten Teil zum Namen WINDOWS NT beigetragen: NT steht für *New Technology*.

Die Idee, eine Windows-kompatible Oberfläche mit einem neuen Kern zu kombinieren, hatte ein DEC-Entwickler, der schon bei zwei weit verbreiteten Betriebssystemen die Entwicklungsarbeiten geleitet hatte: David Cuttler hatte bereits die Arbeiten an RSX-11M für die PDP-11-Typenfamilie geleitet und dann seine Erfahrungen in VAX/VMS für die VAX-Familie verarbeitet. Trotzdem stellt WINDOWS NT, zunächst in der Version 3.5 mit der Oberfläche von WINDOWS 3.11, eine konsequente Fortsetzung der mit RSX und VMS begonnenen Reihe dar, was neben technischen Informationen auch in einem Wortspiel mit dem Namen dokumentiert ist: ersetzt man die Buchstaben VMS durch ihre Nachfolger im Alphabet, so erhält man WNT, ausgeschrieben *Windows New Technology*.

Hinter der neuen Namensgebung WINDOWS 2000 verbirgt sich die konsequente Weiterentwicklung von WINDOWS NT Version 4. Es sind Anleihen aus WINDOWS 98 in der Benutzerführung zu erkennen. WINDOWS 2000 hat aber auch den Anspruch, durch verbesserten Speicherschutz für den Kernel, zuverlässiger und stabiler als WINDOWS NT zu sein.

8.9.2 Einsatz von WINDOWS NT/2000

Genau wie MS-DOS und WINDOWS 95/98 ist WINDOWS NT/2000 ein Betriebssystem für den Desktop-Bereich. Es ist also weder für den Einsatz auf Großrechnern noch für Echtzeitaufgaben (Prozeßsteuerung etc.) gedacht, sondern will den Einsatz von Textverarbeitungs- und Graphikprogrammen, aber auch von CAD-Systemen, ermöglichen. Zum Desktop-Bereich gehört inzwischen auch die Einbindung in ein Netzwerk für die Kommunikation in Arbeitsgruppen und darüber hinaus.

Im Gegensatz zu MS-DOS und WINDOWS 95/98 zielte WINDOWS NT von Anfang an auf professionelle Anwender. Es fordert zwar eine recht umfangreiche Hardwareausstattung, nutzt aber diese Hardware voll aus und erfüllt damit hohe Ansprüche wie z.B. ein preemti-

ves Multitasking. Im Gegensatz zu UNIX handelt es sich aber bei WINDOWS NT nicht um ein echtes Mehrbenutzersystem. Zwar gibt es in der WINDOWS NT Server Version ebenfalls Benutzer und Schutzmechanismen, zu einem Zeitpunkt aber immer nur ein Benutzer am System arbeiten und ein echter Timesharing-Betrieb ist nicht möglich. Diese Lücke versucht Microsoft mit der Version WINDOWS 2000 Datacenter Server und dem darin enthaltenen Process Control Tool zu schließen (Anmerkung: Trotz dessen sei an dieser Stelle erwähnt, dass die Unix Plattform im Server Bereich zur Prozess-, System- und Benutzerverwaltung aus Wartungs-, Sicherheits-, Performance-, Verfügbarkeits- und Skalierungsgründen im Vergleich zur WINDOWS NT/2000–Welt viele Vorteile hat). WINDOWS NT/2000 basiert auf einer 32-Bit-Architektur und unterstützt nur Intel basierende Prozessorsysteme.

Vor allem Sicherheitsaspekte wurden beim Design des Systems berücksichtigt. Es besteht aus einem *Microkernel*, um den sich verschiedene Dienstprogramme gruppieren. Die meisten dieser Dienstprogramme sind austauschbar, so daß zum Beispiel für das Dateisystem mehrere Alternativen zur Wahl stehen. In WINDOWS 2000 ist zur Verwaltung von Netzwerkresourcen das Active Directory integriert. Mit einem graphischen Verwaltungsprogramm kann dieser Service Zentral Rechner, User, Drucker usw. verwalten. Durch die serienmäßige Unterstützung mehrerer Netzwerkprotokolle ist WINDOWS NT/2000 zudem offen für die Integration in bestehende EDV-Welten.

Der Einsatz von Applikationen, die unter MS-DOS, OS/2 oder POSIX-kompatiblen Systemen entwickelt wurden, ist auch unter WINDOWS NT möglich, wenn die Applikationen sich an die Standardschnittstellen ihrer Systeme halten, also etwa direkte Zugriffe auf die Hardware unter Umgehung des Betriebssystems unterlassen.

8.9.3 Die WINDOWS NT/2000 Oberfläche

Die Oberfläche von WINDOWS NT ist nach Anmeldung am System ähnlich zu derjenigen von WINDOWS 95. Hier stehen dieselben Programme und Werkzeuge zur Verfügung, sie verwenden dieselben Sinnbilder und werden genau so bedient wie ihre WINDOWS 95-Vorbilder. Für den Netzwerkbetrieb, die Benutzerverwaltung, die Datensicherung oder für Zugriffskontrollen kommen einige neue Dienstprogramme hinzu. Bei Aufruf der integrierten „MS-DOS Shell“ kann neben den aus MS-DOS bekannten Kommandos auch Befehle aus OS/2 und POSIX. Das gleiche gilt für WINDOWS 2000. Angelehnt ist sie, wie schon erwähnt an WINDOWS 98. Zur Konfiguration des Active Directory, gibt es zusätzlich ein graphisches Verwaltungsprogramm.

8.9.4 Die WINDOWS NT/2000 Dateisysteme

Wie bereits erwähnt, unterstützt WINDOWS NT verschiedene Dateisysteme. Diese tragen die Bezeichnungen FAT (File Allocation Table), HPFS (High-Performance File System) und NTFS (WINDOWS NT File System). Zwei der Systeme stammen aus anderen Betriebssystemen: Das alte FAT16-System entspricht dem einzigen unter MS-DOS gebräuchlichen, und HPFS ist unter OS/2 üblich. Mit Windows95B wurde FAT32 eingeführt, mit dem man Platten mit mehr als 2GB Kapazität ohne Partitionierung als ein einziges Laufwerk formatieren kann. Durch die Unterstützung dieser Systeme sichert WINDOWS NT die Kompatibilität und den Zugriff auf bestehende Daten.

Das technisch überholte FAT16-System wurde schon bei OS/2 mit dem HPFS besser gegen Hardwarefehler gesichert und funktional erweitert, etwa durch Unterstützung für längere Dateinamen. Allerdings setzte sich OS/2 nicht so schnell durch wie erhofft, und damit blieb auch das Dateisystem recht selten.

Gegenüber HPFS ist NTFS erheblich erweitert: über die Plattenaktivitäten wird Protokoll geführt, so daß bei einem Stromausfall oder anderen Problemen automatische Korrekturen möglich sind. Außerdem gibt es Zugriffskontrollen, die die gemeinsame Nutzung eines Dateisystems durch mehrere Benutzer absichern. Wie unter UNIX gibt es die rwx-Bits, zusätzlich jedoch weitere Permissions für *Delete*, *Change Permissions* und *Take Ownership*, also insgesamt sechs statt der drei von UNIX bekannten Bits. Man könnte hier also von rwxdpo-Bits sprechen. Datei- und Verzeichnisnamen dürfen bis zu 256 Zeichen lang sein.
In Windows 2000 ist NTFS in seiner 5.Version implementiert. Mit dieser können sogenannte Quotas eingerichtet werden. Sie ermöglichen dem Administrator, Benutzern oder Gruppen, Festlattenplatz zuzuweisen und zu beschränken.

8.9.5 WINDOWS XP

Ende 2001 ist das Betriebssystem Windows XP, als Nachfolger aller genannten Windows Versionen von Microsoft zum Verkauf freigegeben worden. „XP“ steht dabei für Experience. Ähnlich wie bei Windows 2000 gibt es eine Server und eine Clientvariante. Letzteres wird in der XP Home Edition vor allem Windows95/98/Me ablösen und zielt damit auf den privaten Sektor.

Beide XP-Versionen bauen auf dem Betriebssystemkern von Windows 2000 auf. Für Anwendungen die nur auf „alten“ Windows Versionen laufen, ist ein Kompatibilitätsmodus enthalten, der diese emuliert. Die offensichtlich größte Änderung ist der neu gestaltete Desktop. Neue Erweiterungen sind weiterhin Möglichkeiten der Remote Steuerung und Terminal Server Funktionen zur Verwaltung von verschieden Sitzungen. Aus der Sicht von Windows 2000 ist XP, ausgenommen der genannten Detailverbesserungen, allerdings kein neues Betriebssystem, sondern der technische Zusammenschluss von Windows9x und Windows NT/2000.

8.10 Verteilte Betriebssysteme

Es gibt auch Betriebssysteme für die Anwendung in Multiprozessor- und Netzwerkumgebungen, in denen spezielle Aufgaben aus den Bereichen Kommunikation, Synchronisation und Sicherheit auftreten. Wir wollen hier lediglich einzelne Aspekte dieser verteilten Betriebssysteme herausgreifen und beispielhaft betrachten. Anhand der praktischen Bedeutung konzentrieren wir uns auf Umgebungen, die mehrere Rechner in einem Netzwerk zusammenfassen, wobei der einzelne Rechner jeweils nur einen Prozessor besitzt. Auch Konfigurationen, die mehrere Prozessoren mit einem gemeinsamen Hauptspeicher betreiben, sind vor allem bei hohem Leistungsbedarf denkbar und bei manchen Herstellern schon realisiert.

8.10.1 Struktur verteilter Systeme

Für verteilte Betriebssysteme eignet sich besonders die schon in Abschnitt 8.3.4 vorgestellte Client-Server-Struktur, denn bei ihr kann man die Kommunikation zwischen den einzelnen Prozessen über das Netzwerk abwickeln, so daß theoretisch jeder der Prozesse auf einem eigenen Rechner laufen kann. Dieses Prinzip ist in Bild 8-17 skizziert.

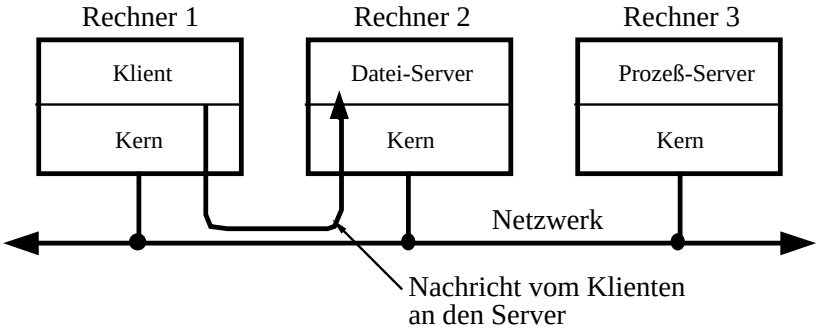

Abbildung 8-17: Das Client-Server-Modell in einem verteilten System

Tatsächlich treibt man die Verteilung der Prozesse nicht so weit, gängig ist es aber durchaus, etwa das Dateisystem auf einem *Fileserver* zu halten, die Zugangsberechtigungen von einem *Authentication Server* prüfen zu lassen, eine Datenbank auf einem separaten *(SQL-)Server* zu installieren und je nach Anwendung weitere spezielle Dienste auf mehrere Server zu verteilen.

8.10.2 Fallbeispiel: Das OSF Distributed Computing Environment

Das *Distributed Computing Environment (DCE)* der *Open Software Foundation (OSF)* stellt eine Reihe von Softwarekomponenten bereit, um die Erstellung verteilter Anwendungsprogramme auf heterogenen Rechnernetzen zu erleichtern und eine verteilte Rechnerumgebung zentral zu verwalten. Seit dem Beginn des WS 97/98 wird das DCE auch im Ausbildungszentrum Informationsverarbeitung im Maschinenbau (IiM) für die Aufgaben der Systemverwaltung eingesetzt, so daß wir es im folgenden kurz beschreiben wollen.

Die Gesamtarchitektur des DCE mit seinen Komponenten ist in der folgenden Abbildung skizziert.

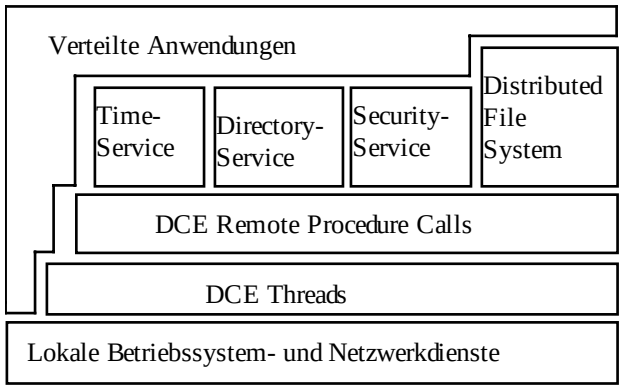

Abbildung 8-18: Architektur des Distributed Computing Environment

Alle DCE Komponenten benutzen die vom lokalen Betriebssystem (UNIX, WINDOWS NT, etc.) bereitgestellten Dienste und Netzwerkfähigkeiten (wie z.B. TCP/IP) als Basis. Das DCE selbst stellt für die Entwicklung von Anwendungen *Threads* und *Remote Procedure Calls (RPC)* zur Verfügung. Mit Hilfe von Threads können Applikationen sehr effizient parallel ablaufen (vgl. entsprechenden Abschnitt über Java-Threads in Kapitel 4.3). Mit Hilfe von RPCs können Applikationen auf einem Rechner Prozeduren auf anderen Rechnern aufrufen, so daß die Entwicklung von komplexen Client/Server Softwaresystemen ermöglicht wird. Die eigentlichen DCE-Dienste *Distributed Time Service (DTS), Cell Directory Service (CDS), Security Service (SEC)* und *Distributed File System (DFS)* verwenden die von DCE bereitgestellten RPC- und Thread-Mechanismen und sind daher als Client/Server Systeme sehr effizient implementiert.

Der Zeitdienst DTS dient zur Synchronisation der Systemuhren von Rechnern eines verteilten Systems. Grundsätzlich besteht das Problem, daß die lokalen Uhren bedingt durch unvermeidbare Toleranzen immer gewisse Differenzen aufweisen. Andererseits ist es für verteilte Algorithmen enorm wichtig, daß die Zeiten der Rechner möglichst genau übereinstimmen. Wenn z.B. mehrere Benutzer an verschiedenen Rechnern eine Datei gemeinsam bearbeiten, so müssen die Uhren der Rechner genau übereinstimmen, damit die Zeit der letzten Änderung der Datei der korrekt abgespeichert werden kann und es zu keinen Datenverlusten kommt.

Im Namensdienst CDS ist die Konfiguration der gesamten verteilten Clients/Server-Umgebung abgespeichert. Client-Applikationen stellen z.B. Anfragen an den Namensdienst, um ihre korrekten Server zu finden.

Der Sicherheitsdienst hat die Aufgabe, den Zugriff unberechtigter Benutzer oder Programme auf Daten, Dienste und Übertragungsnachrichten innerhalb einer verteilten DCE-Umgebung zu verhindern. Im wesentlichen erbringt der Sicherheitsdienst folgende Leistungen:

1. Authentisierung: Eine Instanz (Benutzer, Client, Server oder Programm) bestätigt unter Angabe einer geheimen Kennung, daß es sich bei ihr wirklich um die vorgegebene Instanz handelt.
2. Autorisierung: Eine Instanz (z.B. ein Server) vergibt Zugriffsrechte an bestimmte andere Instanzen (z.B. Clients).
3. Verschlüsselung: Übertragene Nachrichten können verschlüsselt werden, um unberechtigtes Lesen zu verhindern bzw. unberechtigte Datenmanipulationen erkennen zu können.

Das Distributed File System (DFS) wurde auf der Basis des AFS entwickelt. Es stellt allen Benutzern einen einzigen globalen Dateibaum zur Verfügung. Alle Dateien im DFS werden also auf allen Rechnern in der DCE-Umgebung über dieselben Pfadnamen angesprochen. Das DFS hat ähnlich wie das AFS spezielle Eigenschaften, die für große Rechnerumgebungen wichtig sind:

Lokationstransparenz: Unabhängig von der tatsächlichen Speicherung einer Datei auf einem DFS-Server kann immer über denselben Pfad auf die Datei zugegriffen werden. Dateien können sogar von einem Server auf einen anderen verlagert werden, ohne daß der Benutzer dies in irgendeiner Form bemerkt.

Replikation: Dateien können auf verschiedenen DFS-Servern repliziert werden, um ihre Verfügbarkeit zu erhöhen.

Caching: Ein DFS-Client hält sich oft benutzte Dateien automatisch in einem Cache auf der lokalen Platte. Dadurch ist ein sehr effizienter, lokaler Zugriff bei umfangreichen Bearbeitungsvorgängen möglich und die Netzbelastung wird stark reduziert.

Skalierbarkeit: Ein DFS-Server kann bedingt durch seine Implementierung mit Hilfe von Threads und bedingt durch das Client-Caching viele verschiedene Clients gleichzeitig bedienen.

Wenn sich ein Benutzer an einem Clienten anmeldet, wird zunächst seine Authentizität von einem der Security Server geprüft. Nach der erfolgreichen Anmeldung hat der Benutzer Zugriff auf sein Benutzerverzeichnis, das von einem DFS-Server zur Verfügung gestellt wird. Wenn nun Applikationen gestartet werden sollen und diese nicht bereits im lokalen DFS-Cache des Clients vorhanden sind, so werden sie zunächst vom entsprechenden DFS-Server in den Cache geladen und dann gestartet.

8.11 Zusammenfassung

Die Aufgaben eines Betriebssystems lassen sich einteilen in zwei Bereiche: Einerseits soll ein Betriebssystem für den Programmierer eine Abstraktionsebene über der konkreten Hardware zur Verfügung stellen. Andererseits dient eine Betriebssystem als Verwaltungsprogramm für begrenzte Ressourcen, wie z.B. für die Rechenzeit auf den Prozessoren, den Speicherplatz im Hauptspeicher und auf Magnetplatten. Für den ersten Bereich werden sogenannte Systemaufrufe zur Verfügung gestellt, mit denen man als Programmierer das Betriebsystem direkt nutzen kann. Für den zweiten Bereich war es in diesem Kapitel zunächst notwendig, Begriffe wie Prozeß und Datei zu definieren, bevor Scheduling-Mechanismen und das Konzept des virtuellen Speichers eingeführt werden konnten. Die innere Struktur von Betriebssystemen orientiert sich in älteren Systemen oft an einem monolithischen oder geschichteten Aufbau. Das Konzept der virtuellen Maschine wurde durchgängig nur im Bereich der IBM-Großrechner realisiert. Moderne Betriebssysteme besitzen bereits eine Client/Server-Architektur, die oft auf einem Mikrokernel aufbaut.

In der Praxis spielen zur Zeit die Betriebssysteme WINDOWS 95/98, WINDOWS NT/2000 und UNIX eine wichtige Rolle. Wir untersuchten diese Systeme im Hinblick auf deren Einsatzgebiet, der notwendigen Hardware und des verwendeten Dateisystems. In vernetzten Rechnerumgebungen spielen natürlich die Netzwerkfähigkeiten dieser Systeme und insbesondere die zur Verfügung stehenden Sicherheitsmechanismen eine tragende Rolle. Schließlich lassen sich diese Systeme charakterisieren hinsichtlich ihrer Multitasking- und Multiuser-Fähigkeiten. Die folgende Tabelle gibt noch einmal einen knappen Überblick über die Eigenschaften der Systeme.

	MS-DOS	**WINDOWS 95/98**	**WINDOWS NT/ WINDOWS 2000**	**UNIX**
Hardware	Intel 80x86, Intel PI – PIII, AMD K5 – K6/3	Intel 80386, Intel 80486, Intel PI – PIII, AMD K5 – K6/3	Intel 80386, Intel 80486, Intel PI – PIII, AMD K5 – K6/3, DEC Alpha	Intel 80386, Intel 80486, Intel PI – PIII, DEC Alpha, PowerPC, MIPS Rxxxx, HP-PARISC, SUN SPARC, etc.
Einsatzgebiet	PC (Desktop)	PC (Desktop)	PC (Desktop), Workstation, Server	PC (Desktop), Workstation, Server, Großrechner, Supercomputer
Dateisystem	hierarchisch mit Laufwerken, FAT16	hierarchisch mit Laufwerken, FAT16,FAT32	Hierarchisch mit Laufwerken, FAT16,FAT32, HPFS, NTFS	hierarchisch ohne Laufwerke, FAT, HPFS, NTFS, CDFS, etc..
Sicherheit	Nein	Nein	Ja	Ja
Netzwerk	Nein	Ja	Ja	Ja
Multiprogramming	Ja	Ja	Ja	Ja
Multitasking	Nein	non preemptive	preemptive	preemptive
Multiuser	Nein	Nein	Nein	Ja

8.12 Literatur

[AlBe-96] W.Alex und G.Bernö: Einführung in Unix und C. Universität Karlsruhe, 1996

[Hans-73] P. Brinch Hansen: *Operating System Principles*. Prentice-Hall, Englewood Cliffs, 1973

[Tane-94] A. S. Tanenbaum: *Moderne Betriebssysteme*. Deutsche Ausgabe von Ralph Radermacher und Uwe Baumgarten, Hanser, München; Wien; Prentice-Hall International, London, 1994

[HeHM-94] S. Hetze, D. Hohndel, M. Müller u.a: *Linux Anwenderhandbuch und Leitfaden für die Systemverwaltung*. 3. Auflage, LunetIX Softfair 1994, ISBN 3-929764-02-4