Betriebssysteme – eine Einführung

Michael Blume, Dirk Henkel, Lutz Sommerfeld

12. Mai 1998

Abstract und Links

Dies ist eine kurze, überblickartige Einführung in das Thema „Betriebssysteme“. Sie ist als Begleittext zur Vorlesung „Technische Informatik I: Betriebssysteme“ von Prof. Peter Ladkin im Sommersemester 1996 an der Universität Bielefeld entstanden.

Diese Einführung deckt nicht den gesamten Vorlesungsinhalt ab; sie ist als Ergänzung zum Foliensatz der Vorlesung gedacht. Die gesamten Vorlesungsunterlagen befinden sich im World Wide Web unter
`http://www.rvs.uni-bielefeld.de/lecture/Courses/Grundst/Tutor/` .

Für eine intensive Beschäftigung mit dem Thema „Betriebssysteme“ empfehlen wir eines der folgenden Bücher, die uns auch beim Erstellen dieser Einführung eine große Hilfe waren:

- A. Silberschatz, J.L. Peterson, P.B. Galvin: Operating Systems Concepts, Addison-Wesley.
- Andrew Tanenbaum: Moderne Betriebsysteme, Prentice-Hall/Hanser-Verlag (auch in Englisch: Modern Operating Systems).
- Andrew Tanenbaum: Betriebssysteme, Prentice-Hall/Hanser-Verlag (auch in Englisch: Operating Systems). (Nein, es ist nicht dasselbe Buch - es ist mehr ein How-To für Programmierer.)

Inhaltsverzeichnis

1 Allgemeines

1.1 Was ist ein Betriebssystem?

Definitionen des Begriffs „Betriebssystem“:

- „An operating system is the software that breathes life into a computer.“
- DIN 44300: „Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.“
- „Das Betriebssystem ist die Gesamtheit der Programme eines Rechensystems, welche die Betriebssteuerung erledigen und die den Benutzeraufträgen zugängliche Umgebung bereitstellen.“

Betriebsmittel sind Komponenten sowohl der Hardware als auch der Software (System- und Anwendersoftware), wie z.B. Prozessor, Speicher, Ein-Ausgabe-Geräte, Dateien, Programme etc.

1.2 Warum gibt es Betriebssysteme?

Das Betriebssystem ermöglicht es z.B. dem Anwender, Programme auf unterschiedlicher Hardware laufen zu lassen. D.h. das Betriebssystem bietet dem Anwender eine virtuelle Maschine an, welche die reale Hardware „unsichtbar“ macht.

1.3 Geschichte der Betriebssysteme

Die Evolution der Betriebssysteme soll hier nur in kurzen Stichpunkten wiedergegeben werden:

1. **The beginning ...**

   Reine Hardware mit sehr maschinennahen „Programmen“: Zuse Z2.

2. **Transistoren, Batchsysteme**

   Bei diesen Systemen wurden typischerweise die einzelnen Aufgaben aufgespalten und „Batches“ bearbeitet. eg. ein Programmierer bringt die Carten zum Lesegerät, der Leser legt die Liste der „Batches“ auf Tape ab, dann wird mit diesem Tape der eigentliche Rechner bestückt, das so erstellte Ausgabetape wird zum Drucker gebracht ...
   Die Programmiersprache war Fortran, das Betriebssystem FMS (Fortran Monitor System). (1956-65)

3. **IC's und Multiprogramming**

   Das Standardbetriebssystem war das IBM 360. Es verfügte schon über Spoolingtechniken und Timesharingstrategien. Dadurch konnte die Auslastung der CPU mit Prozessen gegenüber Nicht-Multiprogrammsystemen stark verbessert werden (siehe I/O von Prozessen). (1969-1980)

4. **Die PC's**

   Mit der Entwicklung der LSI-Schaltkreise („Large Scale Integration") begann das Zeitalter der PC's. Was zuvor nur großen Betrieben und Behörden vorentalten war, gab es jetzt, wenn auch etwas langsamer, auch für den Privatnutzer. Es gab hier zwei Hauptströmungen:

   (a) MS-DOS als „stand-alone" Betriebssystem auf 80X86-Prozessoren von Intel;

   (b) die UNIX-Welt für Netzwerke auf Motorola 680X0-Prozessoren.

   Seit Beginn der '80-er Jahre ist ein Trend hin zu vernetzten PC's abzusehen.

1.4 Betriebssystem-Modelle

Es gibt verschiedene theoretische Ansätze, auf denen die jeweiligen Betriebssysteme beruhen. Es seien hier nur zwei Modelle beschrieben.

1.4.1 Layered Systems

Dieses Schichtmodell hat die Eigenschaft, daß alle Ebenen bis auf die unterste auf der nächst tieferen aufbauen. Schichtmodelle finden sehr oft in der Informatik Verwendung: z.B. TCP/IP-Standard.

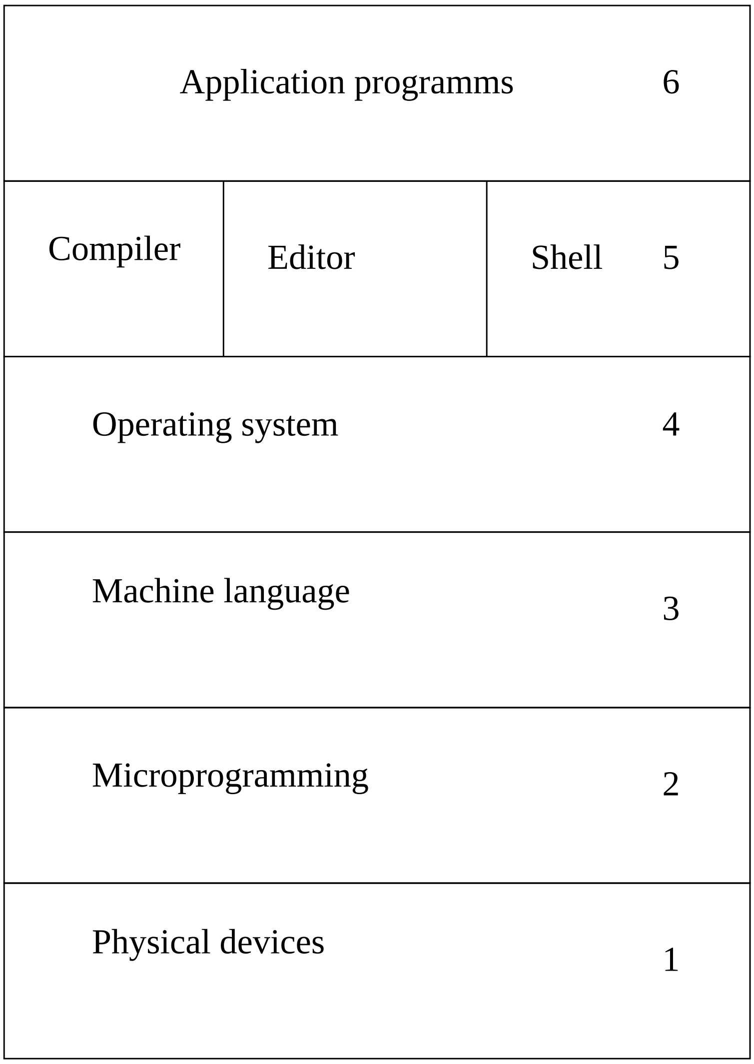

Abbildung 1: Das Betriebssystem-Schichtmodell

- Hardware

  1 Empfängt und erzeugt Interrupts und bildet die Betriebsmittel.

  2 Diese Ebene ist unter anderem für das Memorymanagement zuständig.(siehe VMS Virtual Memory Service)

  3 Diese Ebene teilt jedem Prozeß eine sog. „Operatorconsole“ zu.

- Systemprogramme

  4 Hier wird die I/O von den Ein-Ausgabegeräten gesteuert und die Datenströme ggf. gepuffert. Ab dieser Ebene kann jeder Prozeß über abstrakte I/O-Anweisungen auf die Hardware zugreifen und braucht die physikalischen Anforderungen nicht beachten!

  5 Hier laufen Programme wie Editoren, Compiler, Shells. Ab hier müssen die Prozesse sich nicht mehr selbst um Speicher, Konsole oder I/O-Management kümmern.

- Applications

  6 Anwendungsprogramme wie z.B. Office-Programme, Entwicklungsumgebungen, Web-Browser, Spiele ...

1.4.2 Client-Server-Modell

Dieses Modell stellen wir vor, da es kompatibel zu den verteilten Systemen ist. In den modernen Betriebssystemen gibt es die Bemühung, den Kernel, das eigentliche System, so klein wie möglich zu halten. Das geschieht durch die Verlagerung von Betriebssystemfunktionen in die User-Prozesse:

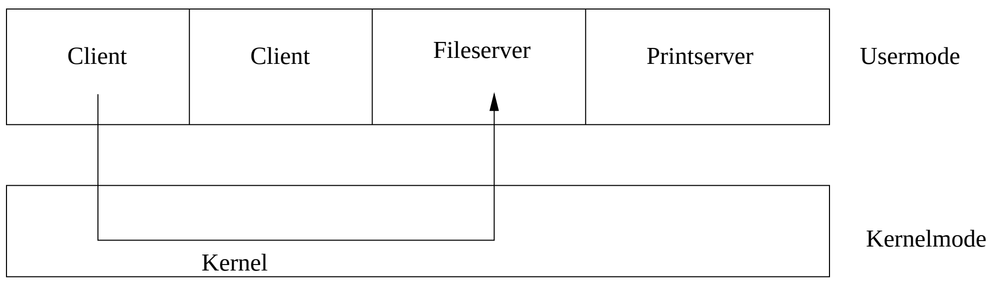

Abbildung 2: Das Client-Server-Modell

Um eine Anfrage des Clients zu erfüllen, hier das Lesen eines Files, sendet der Clientprozeß seinen Wunsch durch den Kernel an den Fileserver. Der Kernel, das eigentliche System, hat hier nur noch die Aufgabe, den I/O zwischen den Client- und Serverprozessen zu überwachen. Diese Aufspaltung hat gewisse Vorteile. Es laufen alle hardwareabhängigen Server-Prozesse im USERMODE, d.h. sie haben keinen direkten Zugang zur Hardware, und ein Bug im Serverbetrieb kann sich nicht so leicht im System fortpflanzen.

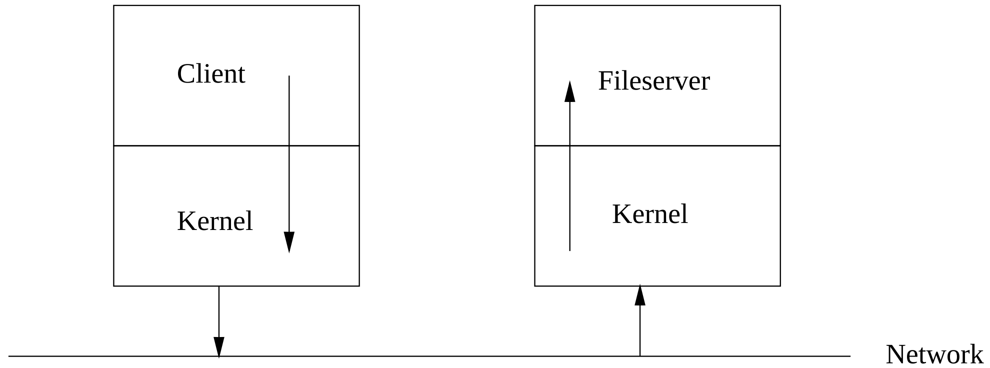

Abbildung 3: Client und Server im verteilten System

2 Der von-Neumann-Rechner

Da es Sinn und Zweck eines Betriebssystems ist, die Komponenten eines Rechners zu betreiben, kann es ein Betriebssystem-Konzept nicht unabhängig vom Konzept eines Rechners geben. Der Rechner-Grundtyp, auf den wir uns beziehen, heißt von-Neumann-Rechner. Zwar entspricht die intuitive Vorstellung von einem Computer bei den meisten Leuten diesem Konzept, aber es ist keinesfalls das einzig mögliche.

2.1 Das von-Neumann-Konzept

Im Jahre 1946 entwickelte ein Amerikaner ein Rechnerkonzept, welches universell sowohl für einfache technische, für kommerzielle, als auch für wissenschaftliche Anforderungen genutzt werden konnte. Das nach John von Neumann benannte Konzept wurde ständig weiterentwickelt, so daß die Wurzeln der meisten heutigen Rechner (z.B. PCs), mit Ausnahme von Parallelrechnern, im von-Neumann-Prinzip liegen. Eine grundlegende Neuerung im von-Neumann-Konzept war die weitestgehende Trennung von Hardware und Anwendungsgebiet des Rechners. Der universelle von-Neumann-Rechner besitzt eine **feste** Hardwarearchitektur. Der Rechner wird durch eine Bearbeitungsvorschrift, das Programm, an die jeweilige Aufgabenstellung angepaßt. Dieses Programm wird vor der eigentlichen Datenverarbeitung in den Speicher des Rechners geladen und kann für die gleiche Aufgabenstellung wiederholt verwendet werden. Diese Eigenschaft hat zu dem Namen "stored-program machine" geführt. Ohne dieses Programm ist der Rechner nicht arbeitsfähig! Weitere Bestandteile des von-Neumann-Konzepts sind:

- Alle Daten und Programmbestandteile werden in demselben Speicher abgelegt. Sie können nur durch die Reihenfolge unterschieden werden.
- Der Speicher ist in gleichgroße Zellen unterteilt, welche über ihre Adressen eindeutig referenzierbar sind (siehe als Beispiel Speicherverwaltung).
- Befehle, die im Programm nacheinander folgen, werden ihrer Reihenfolge im Programm entsprechend im Speicher abgelegt. Das Abarbeiten eines neuen Befehls wird durch die Erhöhung des Befehlszählers initiiert.

- Durch Sprungbefehle kann von der Bearbeitung der Befehle in ihrer gespeicherten Reihenfolge abgewichen werden.
- Es sind mindesten folgende Befehlstypen vorhanden:
  - Arithmetische Befehle: Addition, Multiplikation ...
  - Logische Befehle: UND, ODER, NICHT ...
  - Transportbefehle: Move
  - Ein-/Ausgabebefehle
  - Bedingte Sprungbefehle

2.2 Der Hardware-Aufbau

Die genannten Eigenschaften eines von-Neumann-Rechners wurden in verschiedenen Variationen verwirklicht. Die hier gezeigten Abbildungen sind nur Beispiele von möglichen Hardwarekompositionen, die den Eigenschaften eines von-Neumann-Rechners genügen.

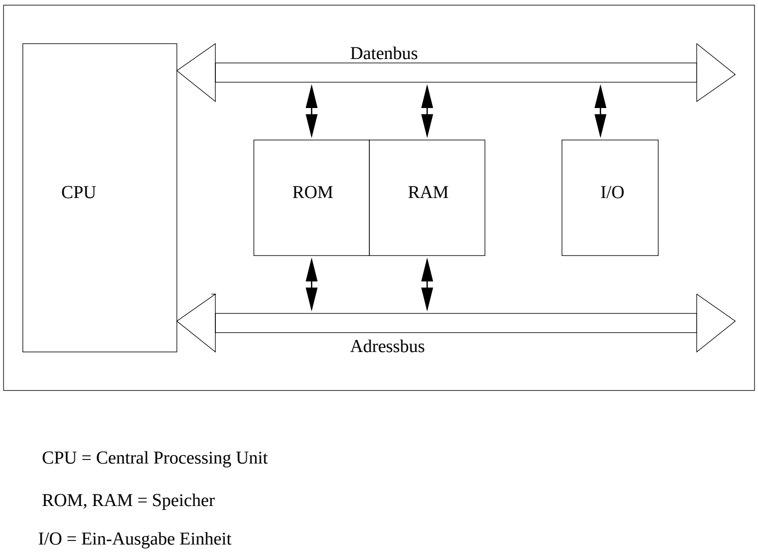

Abbildung 4: Möglicher Aufbau eines von-Neumann-Rechners

Die Abb. 4 zeigt die drei Hauptbestandteile des Rechners:

- Central Processing Unit (CPU)
- Speicher (RAM, ROM)
- Ein-/Ausgabeeinheit (I/O)

Die Verbindung zwischen diesen Obereinheiten wird durch sog. Busse hergestellt. Es gibt zwei verschiedene Bustypen, auf die im weiteren Verlauf noch eingegangen wird.

2.2.1 Central Processing Unit

Die CPU übernimmt die Ausführung von Befehlen und die Steuerung der notwendigen Vor- bzw. Nachverarbeitungsschritte. Sie ist das Herzstück des Rechners. Wie in Abb. 5 zu sehen ist, besteht die CPU aus mehreren Untereinheiten.

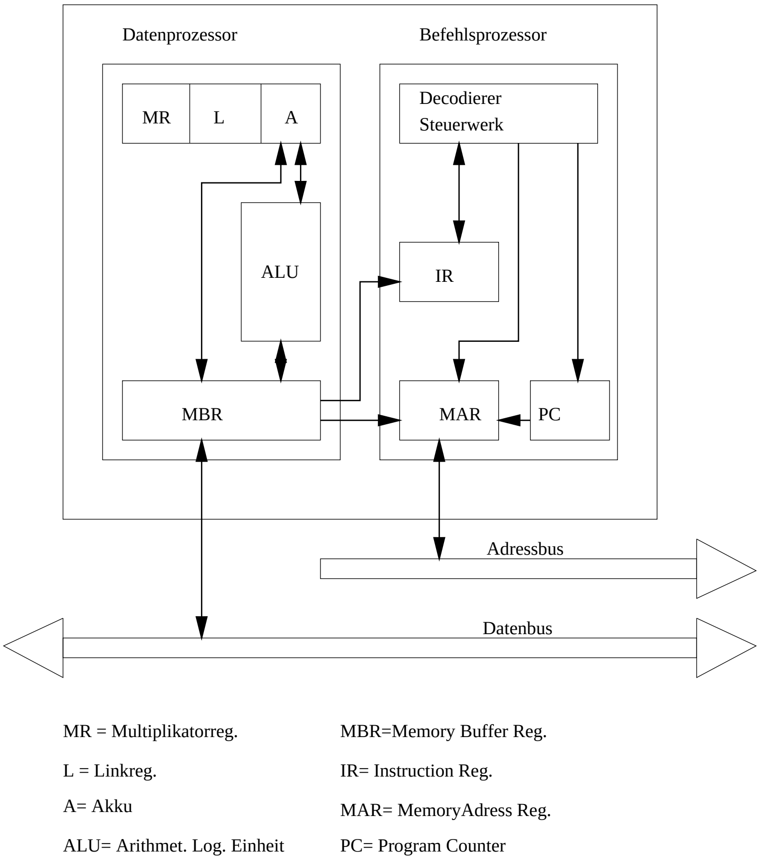

Abbildung 5: Aufbau eine CPU

Zum einen enthält die CPU einen Befehls- und einen Datenprozessor. Der **Datenprozessor** hat die Aufgabe, Daten zu verarbeiten, d.h. z.B. Multiplikationen, Additionen, etc. auszuführen. Um dieser Aufgabe gerecht zu werden, besitzt der Datenprozessor die sog. Arithmetic Logical Unit (ALU), sowie mindestens vier Register für die Operanden. Die Register haben im einzelnen folgende Aufgaben:

Akkumulator: Dieses Register, auch Akku genannt, dient zum Caching von Daten, die für eine ALU-Operation benötigt werden. Diese Registerart wird auch als "general purpose"-Register bezeichnet, da es prinzipiell für jede im Rahmen eines Programms anfallende Aufgabe verwendet werden kann. Logischerweise enthält ein "realer" Rechner eine große Anzahl von Registern dieser Art.

Multiplikator-Register: Dieses Register dient z.B. zur Aufnahme von Multiplikationsergebnissen und ist im Prinzip eine Erweiterung des Akku.

Link-Register: Dieses Register dient z.B. zur Aufnahme von Additionsüberträgen und ist im Prinzip eine Erweiterung des Akku.

Memory-Buffer-Register: Dieses Register dient der Kommunikation mit dem Speicher.

Die Multiplikator-, Link- und Memory-Buffer-Register sind alle sog. "special purpose"-Register, welche nur für eine spezielle Aufgabe konzipiert sind und auch nur für diese benutzt werden.
Der **Befehlsprozessor** hat die Aufgabe, Befehle zu entschlüsseln und deren Ausführung zu steuern und zu überwachen. Er besitzt dazu folgende Bestandteile:

Befehlsregister: Es beinhaltet den aktuell bearbeiteten Befehl.

Speicheradreßregister: In ihm wird die Adresse des Speichersegments abgelegt, welches als nächstes angesprochen wird.

Befehlszähler: Dieses Register enthält die Adresse des als nächstes abzuarbeitenden Befehls im Speicher.

Decodier- und Steuerwerk: Diese Einheit sorgt für die Entschlüsselung eines Befehls und im Anschluß für eine überwachte Ausführung.

Der Ablauf vom Laden eines Befehls aus dem Speicher, seiner Dekodierung und Bearbeitung wird später noch genauer berschrieben.

2.2.2 Speicher

Der Speicher eines von-Neumann-Rechners ist von seiner Aufgabenstellung her in zwei logische Komponenten unterteilbar: zum einen das **ROM** (Read Only Memory) und zum andern das **RAM** (Random Access Memory). Das ROM ist ein sog. Festspeicher, d.h. die Daten in diesem Speicher sind nicht mehr veränderbar. Das ROM enthält meist Befehle, welche die CPU häufig benutzt (Mikroprogrammierung).
Das RAM ist ein Speicher mit sog. wahlfreiem Zugriff und ist der eigentliche Arbeitsspeicher des Rechners. In diesen wird das Programm vor dessen Ausführung geladen, und auch Daten und Adressen werden dort zwischengespeichert.

2.2.3 Ein-Ausgabeeinheit

Die sog. I/O-Einheit steuert den Datenfluß zu den externen Hardware-Komponenten (z.B dem Drucker).

2.3 Arbeitsweise der CPU

Ein von-Neumann-Rechner arbeitet prinzipiell folgendermaßen: Der Rechner führt ein Programm aus, das als Folge von Befehlen und zusammen mit den zu bearbeitenden Daten in seinem Speicher steht. Der Inhalt jeder Speicherzelle kann als Befehl, Adresse oder

Datum interpretiert werden. Für die CPU sind diese verschiedenen Inhalte grundsätzlich nicht unterscheidbar. Die tatsächliche Interpretation hängt nur davon ab, wie das Programm die jeweilige Speicherzelle verwendet.

Zu jedem Zeitpunkt führt die CPU genau einen Befehl aus, der höchstens ein Datum im Speicher bearbeitet. Die Verarbeitung eines Befehls ist in zwei Phasen aufgeteilt:

Fetch-Phase: Der Inhalt der im Befehlszähler enthaltenen Adresse wird ins Befehlsregister geladen und als Befehl interpretiert. Falls dieser Befehl einen Operanden aus dem Speicher bearbeitet, wird der Inhalt der im Befehl angegebenen Speicheradresse in das Memory-Buffer-Register geladen.
Der Befehlszähler wird aktualisiert, d.h. entweder um 1 erhöht oder – bei Sprungbefehlen – auf die im Befehl angegebene Adresse gesetzt.

Execution-Phase: Die eigentliche Ausführung des Befehls

Zentrale Bedeutung für alle Befehle, die Daten bearbeiten, hat der Akku. Einstellige Operationen wie z.B. Negation oder Wurzelziehen arbeiten auf dem Inhalt des Akkus und müssen so keinen Operanden aus dem Speicher lesen. Zweistellige Operationen wie Addition oder Multiplikation erwarten einen der beiden Operanden im Akku und brauchen so nur auf eine Speicheradresse zuzugreifen. Auch das Ergebnis dieser Operationen wird wieder im Akku abgelegt. Auf diese Weise können Speicherzugriffe (die deutlich mehr Zeit benötigen als die Operationen innerhalb der CPU) auf das nötigste beschränkt werden.

3 Prozesse

3.1 Das Prozeß-Modell

Ein Computer tut viele Dinge scheinbar gleichzeitig, etwa Bildschirmausgaben, Tastaturabfrage, Laufwerkzugriffe, Ausführung von Anwenderprogrammen. Tatsächlich müssen all diese Aufgaben abwechselnd nacheinander von einem Prozessor geregelt werden. Eine Aufgabe des Betriebssystems ist es, die Abarbeitung vieler gleichzeitig anstehender Aufgaben zu organisieren.

Ein grundlegendes Konzept für die Beschreibung und Lösung dieser Aufgabe ist das des *Prozesses*. Alle ausführbaren Programme eines Rechners (einschließlich des Betriebssystems) werden als eine Menge von sequentiellen Prozessen betrachtet. Ein Prozess ist ein sich in Ausführung befindliches Programm einschließlich des aktuellen Wertes des Programmzählers, der Register und der Variablen.

Im Modell besitzt jeder Prozeß seinen eigenen virtuellen Prozessor, alle Prozesse laufen parallel ab, die Ausführungsgeschwindigkeiten der einzelnen Prozesse sind dabei nicht bekannt. Ausgehend von diesem Modell wird dann das Wechseln des realen Prozessors zwischen den einzelnen Prozessen beschrieben. Ein sogenannter *Scheduling*-Algorithmus entscheidet darüber, wann welcher Prozeß den Prozessor zugeteilt bekommt.

3.1.1 System-Grundstruktur

Das Prozeßmodell erlaubt eine relativ einfache Beschreibung von Betriebssystemen. Ein System gliedert sich in diesem Modell grundsätzlich in zwei Schichten: Die unterste Schicht behandelt die Erzeugung, Unterbrechung und Auflösung von Prozessen. Sie enthält den Scheduler, der die Verteilung der Rechenzeit des Prozessors zwischen den Prozessen regelt. Diese Schicht macht einen vergleichsweise kleinen Teil des Betriebssystems aus. Die darüberliegende Schicht besteht aus vielen voneinander getrennten, sequentiellen Prozessen. Hierdurch kann auch ein sehr großes Betriebssystem übersichtlich strukturiert werden.

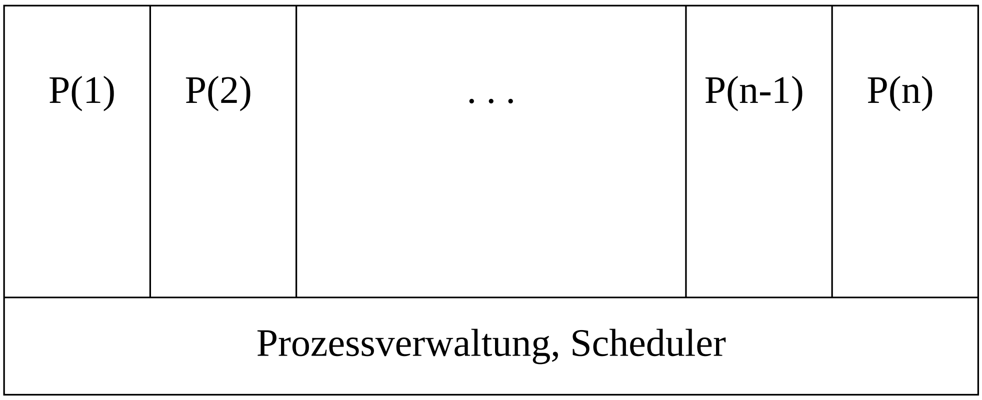

Abbildung 6: Die Grundstruktur eines Systems im Prozeßmodell

3.1.2 Prozeß und Programm

Der Unterschied zwischen *Prozeß* und *Programm* ist klein, aber trotzdem wichtig. Dazu ein Beispiel aus der Küche: Wenn man nach einem Rezept etwas kocht, dann ist dieses Rezept das Programm. Der Prozeß ist der Vorgang des Kochens, er kann zum Beispiel durch einen anderen Prozeß (etwa durch einen Anruf) unterbrochen und danach fortgesetzt werden. Diese Unterbrechung betrifft den Prozeß, nicht das Programm. Zum Koch-Prozeß gehört der aktuelle Zustand der Zutaten und Geräte, dieser bleibt gespeichert, wenn der Prozeß unterbrochen wird.

Die Grundidee ist: ein Prozeß ist eine Aktivität beliebiger Art. Er besitzt ein Programm, Eingaben, Ausgaben und einen aktuellen Zustand. Ein einzelner Prozessor kann zeitlich zwischen mehreren Prozessen aufgeteilt werden.

3.2 Prozeßverwaltung

3.2.1 Prozeßhierarchien

Ein Betriebssystem stellt Möglichkeiten zum Erzeugen und Auflösen von Prozessen zur Verfügung. In UNIX können Prozesse mit dem Systemaufruf `fork` erzeugt werden. Nach einem `fork`-Aufruf wird der Vaterprozeß parallel zum neu erzeugten Kindprozeß fortgesetzt. In Vater- und Kindprozeß können weitere `fork`-Aufrufe vorkommen. Auf diese

Weise kann ein beliebig tiefer Baum von Prozessen entstehen. In MS-DOS gibt es ebenso einen Systemaufruf, mit dem ein Kindprozeß erzeugt wird, allerdings wird hier der Vaterprozeß unterbrochen, bis der Kindprozeß beendet ist; die beiden Prozesse werden also nicht parallel ausgeführt.

3.2.2 Prozeßzustände

Ein Prozeß kann auf zwei verschiedene Arten unterbrochen werden. Eine Möglichkeit ist, daß er auf ein äußeres Ereignis wartet, etwa auf das Eintreffen von Eingabedaten. Er ist dann solange *blockiert*, bis diese Eingabedaten vorliegen. Die andere Möglichkeit ist, daß der Scheduler gerade einem anderen Prozeß den Prozessor zuteilt. Der Prozeß ist dann ebenfalls nicht *rechnend*, aber er ist *rechenbereit*. Wir unterscheiden also die drei Prozeßzustände rechnend, rechenbereit und blockiert. Ein rechenbereiter Prozeß wird rechnend, wenn er vom Scheduler den Prozessor zugeteilt bekommt. Ein blockierter Prozeß wird zunächst rechenbereit, wenn das Ereignis, auf das er gewartet hat, eintrifft. Er wird erst rechnend, wenn er erneut den Prozessor zugeteilt bekommt. Abb. 7 stellt die vier möglichen Übergänge zwischen diesen drei Zuständen dar:

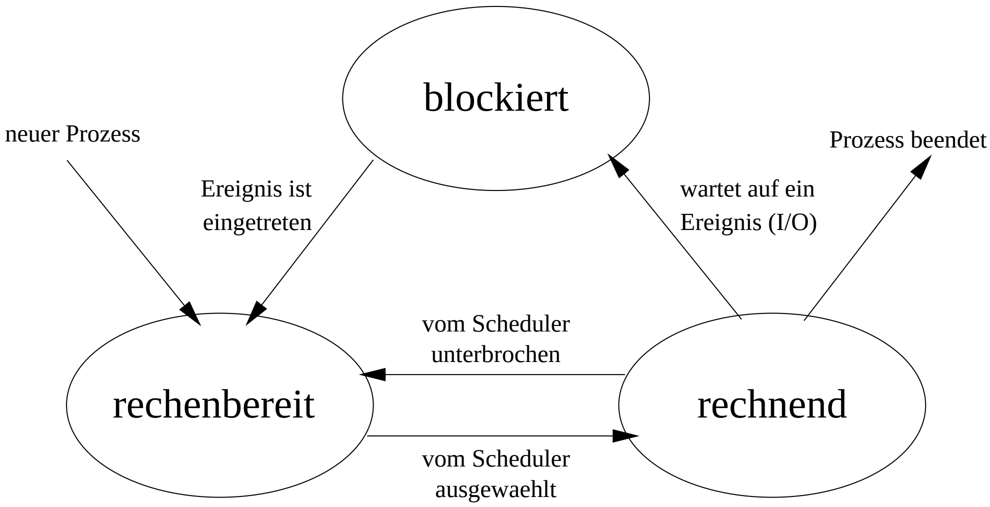

Abbildung 7: Die Übergänge zwischen den Prozeßzuständen

3.2.3 Implementierung

Zur Implementierung des Prozeßmodells verwaltet das Betriebssystem eine sogenannte *Prozeßtabelle*. Diese enthält für jeden Prozeß einen Eintrag mit allen Informationen, die gespeichert werden müssen, wenn der Prozeß unterbrochen wird. Dazu gehören zum Beispiel Informationen über den Prozeßzustand, den Programmzähler, den belegten Speicher, die geöffneten Dateien usw. Mit den Informationen aus dem entsprechenden Prozeßtabellen-Eintrag kann ein Prozeß nach einer Unterbrechung so fortgesetzt werden, als wäre er nie unterbrochen worden.

3.3 Konkurrente Prozesse

Konkurrente Prozesse entstehen immer dann, wenn mehrere Prozesse gleichzeitig bzw. parallel im Hauptspeicher bearbeitet werden. Dies führt zu Problemen wie:

- Umschaltung zwischen den verschiedenen Prozessen
- Absicherung jedes einzelnen Prozesses gegen Einflüße der anderen

Zu diesen Problemen kommt es auf Grund der gemeinsamen Benutzung von Daten durch mehrere synchrone Prozesse. Alle Prozeß-Abschnitte, die gemeinsam mit anderen Prozessen Daten verändern (sog. kritische Abschnitte), müssen in *einer* Sequenz ausgeführt werden, ohne von anderen konkurrenten Prozessen unterbrochen zu werden.

Dieses Vorgehen, das nicht nur für Daten, sondern für alle nicht gemeinsam benutzbaren Betriebsmittel gilt, nennt man wechselseitigen Ausschluß (mutual exclusion).

3.4 Synchronisation

Die Synchronisation hat die Aufgabe, den wechselseitigen Ausschluß für kritische Prozeßabschnitte sicherzustellen. Dafür werden u.a. folgende Methoden benutzt:

- Semaphore
- Monitoring

3.4.1 Semaphore

Der Semaphor (griechisch für Zeichenträger) stellt eine Synchronisationsvariable dar. Um den Wert eines Semaphors zu verändern , sind nur zwei Operationen zulässig, die in ihrer Ausführung nicht unterbrochen werden dürfen.

P-Operation: $P(S)$ vermindert den Wert des Semaphors S um 1, sofern $S > 0$ ist. Für S= 0 wird der P ausführende Prozeß gestoppt und in eine zu dem Semaphor S gehörende Warteliste eingetragen.

V-Operation: $V(S)$ erhöht den Wert von Semaphor S um 1 und belebt den nächsten wartenden Prozeß aus der Warteliste von S.

Durch den geschickten Einsatz von einem oder mehreren Semaphoren und den Operationen $P(S)$ und $V(S)$ läßt sich die Synchronisation von kritischen Abschnitten meist gut lösen. Es kann jedoch u.U. sehr schwierig sein, das Auftreten von Deadlocks zu vermeiden bzw. aufzulösen.

3.4.2 Monitoring

Eine weitere Möglichkeit, zu verhindern, daß mehrere kritische Abschnitte gleichzeitig ausgeführt werden, ist das Modell des Monitors.

In einem Monitor werden die zu den kritischen Abschnitten gehörenden Daten und Prozeduren zusammengepackt. Der Monitor achtet darauf, daß maximal ein Prozeß gleichzeitig Zugriff auf die in ihm enthaltenen Komponenten hat. Normalerweise gibt es mehrere Monitore gleichzeitig. Prozesse dürfen aus jedem Monitor Prozeduren aufrufen, soweit dieser Monitor noch nicht von einem anderen Prozeß belegt ist. Möchte ein Prozeß auf einen schon besetzten Monitor zugreifen, so muß er warten. Die Organisation von wartenden Prozessen wird über eine Bedingungsvariable und zwei Operationen, die zu der Bedingungsvariablen gehören, realisiert.

Bedingungsvariable c: gehört zu einer Synchronisationsbedingung $A(c)$ (z.B. „es darf maximal ein Prozeß in Datei x schreiben"), c wird normalerweise als Menge aller durch c blockierten Prozesse W_c beschrieben. Ist W_c leer, so blockiert c keinen Prozeß.

Operation wait: wird von einem Monitor ausgeführt, falls die zu dem anfragenden Prozeß gehörige Synchronisationsbedingung $A(c)$ nicht erfüllt ist. Der Prozeß wird in die Menge W_c eingefügt und blockiert. Der Monitor wird durch den gestoppten Prozeß nicht weiter blockiert.

Operation signal: wird vom Monitor ausgeführt, sobald eine Synchronisationsbedingung $A(c)$ erfüllt wird, die vorher nicht erfüllt war. Der Aufruf von signal hat zur Folge, daß ein bezüglich $A(c)$ in W_c eingetragener (also blockierter) Prozeß die Kontrolle über den Monitor erhält und somit nicht mehr blockiert ist. Sollte sich kein Prozeß in W_c befinden, hat signal keine Bedeutung.

Ein Vorteil der Monitore ist es, daß diese schon bei der Erstellung von ausführbarem Code (z.B. Compiler) eingerichtet werden können. Sie sind im Gegensatz zu Semaphoren nicht allein eine Aufgabe des Betriebssystems. Wie auch schon bei den Semaphoren ist auch beim Monitoring genau darauf zu achten, daß keine Deadlocks entstehen können (siehe nächster Abschnitt).

4 Scheduling und Deadlocks

4.1 Scheduling

Das Scheduling ist die Verteilung der Rechenzeit des Prozessors auf mehrere Prozesse, die gleichzeitig den Prozessor nutzen wollen. Die Aufgabe eines Schedulers ist es, nach einer vorgegebenen Strategie die Entscheidungen zu treffen, wann und wie lange welcher Prozeß den Prozessor zur Verfügung hat.

4.1.1 Ziele

Es gibt eine Reihe von möglichen Kriterien, nach denen die Scheduling-Strategie gewählt wird. Einige typische Kriterien sind:

Fairneß: Jeder Prozeß erhält einen gerechten Anteil der Prozessorzeit.

Effizienz: Der Prozessor ist immer vollständig ausgelastet.

Antwortzeit: Die Antwortzeit für die interaktiv arbeitenden Benutzer wird minimiert.

Wartezeit: Die Wartezeit auf die Ausgaben von Stapelaufträgen (d.h. Rechenaufträgen, die einmal abgegeben werden, im Hintergrund laufen und deren Ergebnisse irgendwann abgeholt werden) wird minimiert.

Durchsatz: Die Anzahl der Aufträge, die in einem bestimmten Zeitintervall ausgeführt werden, wird maximiert.

Einige dieser Ziele widersprechen sich. Um zum Beispiel die Antwortzeit zu minimieren, sollten möglichst gar keine Stapelaufträge bearbeitet werden (oder nur nachts, wenn alle interaktiven Benutzer schlafen). Das erhöht natürlich die Wartezeit auf die Ausgaben der Stapelaufträge (auf mindestens eine Nacht). Ein anderes Beispiel: wenn jeder Prozeß den gleichen Anteil an der Prozessorzeit bekommt (also auch die Prozesse, die den größten Teil ihrer Zeit blockiert sind, weil sie etwa auf Benutzereingaben warten), wird der Prozessor mit Sicherheit nicht optimal ausgelastet sein.

4.1.2 Scheduling-Strategien

Es gibt daher keine optimale Scheduling-Strategie, die alle diese Kriterien gleichzeitig erfüllt. Je nachdem, welche dieser Kriterien höhere Priorität erhalten sollen, sind bestimmte Strategien geeigneter und andere ungeeigneter. Die einfachste Strategie war in den früheren Batch-Systemen implementiert: „run to completion“ – ein Prozeß verfügt über den Prozessor, bis er beendet ist, erst dann kann ein neuer Prozeß gestartet werden. Im Gegensatz dazu werden Strategien, die die zeitweilige Unterbrechung laufender Prozesse erlauben, als „preemptive scheduling“ bezeichnet. In diesem Fall muß sichergestellt werden, daß die Daten unterbrochener Prozesse nicht vom laufenden Prozeß gestört werden können. Dazu dienen Schutzmechanismen wie zum Beispiel Semaphore und Monitore.

4.1.3 Round Robin Scheduling

Einer der einfachsten, fairsten und verbreitetsten Scheduling-Algorithmen ist *Round Robin*. Dabei werden alle rechenbereiten Prozesse in einer Warteschlange angeordnet. Jeweils der vorderste Prozeß wird aus der Schlange genommen, bekommt ein festes *Quantum* Prozessorzeit und wird dann, falls er mehr Zeit benötigt, erneut hinten an die Warteschlange angestellt. Neu hinzukommende Prozesse werden ebenfalls an das Ende der Schlange gestellt. Das Zeitquantum ist immer gleich groß, typischerweise in Millisekunden-Größenordnungen.

Die Wahl eines geeigneten Quantums ist ein Kompromiß: Je kleiner das Quantum, desto größer wird der Zeitanteil, den das System mit der Umschaltung zwischen den Prozessen verbringt. Je größer das Quantum, desto länger werden die Antwortzeiten für interaktive Benutzer.

4.1.4 Priority Scheduling

Mit Round Robin werden alle Prozesse gleich behandelt, es gibt aber auch die Möglichkeit, bestimmte Prozesse gegenüber anderen zu bevorzugen. Dazu wird jeder Prozeß einer Prioritätsklasse zugeordnet (diese kann sich etwa aus dem Rang des Auftraggebers oder aus der Unterscheidung von System- und Anwendungsprozessen ergeben).

Im Extremfall bekommt ein Prozeß nur dann den Prozessor, wenn gerade kein rechenbereiter Prozeß mit höherer Priorität existiert. Zwischen Prozessen gleicher Priorität kann dabei nach dem Round-Robin-Verfahren abgewechselt werden. Dies ergibt bei N Prioritätsklassen ein Modell mit N getrennten Warteschlangen.

Eine andere, fairere Möglichkeit ist, wie bei Round Robin alle Prozesse in *eine* Warteschlange zu stellen und die Prozesse mit höherer Priorität jeweils mehrere Quanten unmittelbar hintereinander rechnen zu lassen, ehe sie neu ans Ende der Schlange gestellt werden.

4.1.5 Dynamische Prioritäten

Die Einordnung der Prozesse in Prioritätsklassen muß nicht statisch sein. Das folgende Beispiel für dynamische Zuordnung von Prioritäten stammt von einem sehr frühen Betriebssystem, in dem die Prozeßumschaltung noch verhältnismäßig lange dauerte und daher möglichst selten durchgeführt werden sollte:

Es gibt nur eine Warteschlange. Die Prioritätsklassen unterscheiden sich durch die Anzahl der Quanten, die die entsprechenden Prozesse unmittelbar hintereinander bekommen. Ein Prozeß aus Klasse k bekommt jeweils 2^k Quanten im Stück. Jeder neue Prozeß und jeder Prozeß, der zuletzt blockiert war, kommt in Prioritätsklasse 0. Wenn ein Prozeß aus Klasse k rechnet und seine 2^k Quanten Rechenzeit voll ausnutzt (also nicht zwischendurch blockiert oder beendet wird), dann steigt er in Klasse $k + 1$ auf. Auf diese Weise brauchen rechenintensive Prozesse nicht oft unterbrochen zu werden, und trotzdem ist die Antwortzeit für interaktive Benutzer relativ klein. Ein Prozeß, der zum Beispiel 1000 Quanten Rechenzeit benötigt, erhält nacheinander 1, 2, 4, 8, 16, 32, 64, 128, 256 und 512 Quanten, muß also nur neunmal unterbrochen werden.

4.1.6 Shortest Job First

Eine andere Art, Prioritäten zu setzen, ist *Shortest Job First*: Angenommen, eine Reihe von Prozessen kommt gleichzeitig in die Warteschlange und die Rechenzeiten, die diese Prozesse benötigen, können bereits vorher abgeschätzt werden. Dann wird die Gesamt-Wartezeit aller Prozesse minimiert, wenn man die kürzesten Prozesse zuerst rechnen läßt. Dies funktioniert allerdings nicht für Prozesse, die erst später zur Warteschlange hinzukommen.

4.1.7 Scheduling auf zwei Ebenen

Bisher sind wir davon ausgegangen, daß alle Prozesse gleichzeitig im Hauptspeicher gehalten werden können. Es kann aber auch sein, daß ein Teil der Prozesse in Sekundärspeicher (etwa auf die Festplatte) ausgelagert werden muß. Dann muß auf zwei Ebenen Scheduling betrieben werden: Auf der oberen Ebene wird entschieden, welche Prozesse gerade in den Hauptspeicher geladen werden. Auf der unteren Ebene wird unter den im Hauptspeicher befindlichen Prozessen jeweils einem der Prozessor zugeteilt. Auf beiden Ebenen können unterschiedliche Strategien (Round Robin, Priority Scheduling, etc.) implementiert sein.

4.2 Deadlocks

Wie schon erwähnt, kann es trotz der Synchronisation von konkurrenten Prozessen zu Systemzuständen kommen, in denen sich mehrere Prozesse gegenseitig derart behindern, daß mindestens ein Prozeß ohne äußeres Eingreifen bis in alle Ewigkeit (meistens allerdings nur bis zum Ausschalten des Rechners) blockiert ist. Diese Zustände werden Deadlocks gennant.

Es gibt vier Bedingungen, die gleichzeitig erfüllt sein müssen, damit ein Deadlock auftritt:

- Ein Prozeß hat die exklusive Kontrolle über ein Betriebsmittel.
- Das Betriebsmittel bleibt bis zum Ende des Prozesse in dessen Gewalt.
- Während ein Prozeß auf ein Betriebsmittel wartet, belegt er schon ein anderes exklusiv.
- Es gibt mindestens zwei Prozesse, die die ersten drei Bedingungen erfüllen und auf Betriebsmittel warten, die vom jeweils anderen Prozeß belegt sind.

Es gibt nun prinzipiell zwei Möglichkeiten, Deadlocks zu behandeln:

- Erkennung und anschließende Beseitigung
- Prophylaktische Vermeidung

4.2.1 Erkennung und anschließende Beseitigung von Deadlocks

Um Deadlocks zu erkennen, ist es ausreichend, darauf zu achten, ob die vierte Bedingung erfüllt ist. Dazu wird eine Matrix bestimmt, in der zu jedem Prozeß und jedem Betriebsmitteltyp eingetragen wird, wieviele Betriebsmittel eines Typs der jeweilige Prozeß belegt. Diese Matrix heißt Zuweisungsmatrix. Analog gibt es eine weitere Matrix, in der eingetragen ist, wieviele Betriebsmittel eines Typs der jeweilige Prozeß zusätzlich angefordert hat. Diese Matrix heißt Anforderungsmatrix. Aus den beiden Matrizen läßt sich nun bstimmen, ob ein Deadlock vorliegt oder nicht. Ist für einen bestimmten Prozeß die Summe der zugewiesenen und angeforderten Betriebsmittel größer als die zur Zeit noch freien Betriebsmittel, so liegt ein Deadlock vor. Um einen erkannten Deadlock zu beenden, wird der betreffende Prozeß beendet und alle von ihm benutzten Betriebsmittel freigegeben. Dies hat den Nachteil, daß der abgebrochene Prozeß später nochmal von vorne gestartet werden muß.

4.2.2 Prophylaktische Vermeidung von Deadlocks

Um Deadlocks gar nicht erst entstehen zu lassen, ist es nötig, daß alle Prozesse zu Beginn angeben, welche Betriebsmittel sie während ihrer Abarbeitung benutzen werden. Wird von einem Prozeß ein Betriebsmittel angefordert, so wird berechnet, ob in Abhängigkeit von den anderen momentan benutzten Betriebsmitteln ein Deadlock prinzipiell möglich ist. Ist dies der Fall, so wird dem Prozeß das Betriebsmittel verwehrt, und der Prozeß muß warten. Die Nachteile dieser Methode sind, daß die Auslastung der Betriebsmittel klein ist und daß die Zeitspanne, in der das System mit der Systemverwaltung beschäftigt ist, zunimmt.

5 Verteilte Systeme

Verteilte Systeme haben wie alle Betriebssysteme die vier grundlegenden Komponenten:

1. Prozessmanagement
2. I/O Device Management
3. Memorymanagement
4. File Management

5.1 Definition des verteilten Systems

Ein verteiltes Betriebssystem besteht aus einzelnen CPU's, bzw. Rechnern, die über ein Netzwerk oder gemeinsamen Speicher kommunizieren. Der Benutzer hat im idealen Fall den Eindruck, an einer großen Maschine zu arbeiten, anstatt an einem verteilten System. Verteilte Systeme lassen sich nach dem Grad der Trennung der Komponenten einteilen:

loosely coupled distributed systems: Hier handelt es sich um Rechner, welche nur über ein gemeinsames Netz miteinander kommunizieren (z.B. Rechner im NFS-Verbund).

tightly coupled distributed systems: In diesem Fall haben die einzelnen CPU's einen gemeinsamen Speicher, den sie sich teilen und über den sie miteinander kommunizieren. Diese Verbindung heißt auch Prozessorknoten.

5.2 Vorteile verteilter Systeme

Dieses sind nur wenige Beispiele:

gesteigerte Stabilität: Gewöhnlich kann ein Knoten die Jobs eines anderen Knotens übernehmen, wenn dieser ausgefallen ist, wodurch zwar die Verarbeitungsgeschwindigkeit sinkt, aber das System stabil bleibt.

große Flexibilität: Es können fast beliebige neue Erweiterungen durch Anfügen neuer Prozessorknoten erfolgen.

6 Memory-Management

Die Speicherverwaltung ist ein grundlegender Service des Betriebssystems. So verschieden die Betriebssysteme sind, so unterschiedlich sind auch die Sepicherverwaltungsstrategien. Wir betrachten im folgenden nur Speichermodelle die auch im Mehrprogrammsystem zum Einsatz kommen (also nicht von MS-DOS!!!). Im Mehrprogrammbetrieb ist es möglich, daß mehrere Prozesse auf dem Rechner „gleichzeitig“ aktiv sind, bzw. bei einem „single processor system“durch den Scheduler verwaltet werden. Gleichzeitig bedeutet hier also, daß der Benutzer den Eindruck von Parallelität bekommt.

6.1 Swapping

Da der Arbeitsspeicher begrenzt ist, können in der Regel nicht alle Prozesse gleichzeitig im Speicher gehalten werden. Das Ein- bzw. Auslagern von nicht aktiven Prozessen nennt man SWAPPING. In einigen Systemen wird, falls ein Prozeß im Speicher gehalten wird kein Plattenplatz für ihn reserviert. Wenn der Prozeß dann ausgelagert wird, muß für ihn Platz in dem Swap-Bereich der Festplatte allokiert werden. Beide Aufgaben, sowohl die Verwaltung des Swap-Bereiches der Festplatte, als auch der Speicher werden beispielsweise mit folgenden Verfahren gelöst:

6.1.1 Speicherverwaltung mit Bitmaps

Der Speicher wird in Allokationseinheiten unterteilt und eine dazu korrespondierende Bitmap angelegt. Für jeden belegten Block in dem Speicher wird dann an einer bestimmten Stelle der Bitmap eine „1“ geschrieben. Diese Methode hat zwei wesentliche Nachteile:

Allokationseinheiten: Die Allokationseinheiten sind Blöcke fester Größe, wodurch die Methode sehr statisch auf Anfragen nach Speicher reagiert und es meist zu einer Verschwendung von Resourcen kommt.

Große Blöcke: Wenn z.B. ein Prozeß k Einheiten des Speichers benötigt, so muß bei dieser Methode die Bitmap nach einer kontinuierlichen Folge von k Nullbits gesucht werden, was sehr viel Zeit beansprucht.

6.1.2 Speicherverwaltung mit verknüpften Listen

Im Gegensatz zu dem oben beschriebenen Verfahren, das wegen der großen Nachteile selten eingesetzt wird, ist die Methode mit verknüpften Listen gebräuchlicher. Hierbei wird der Speicher als eine verknüpfte Liste von Segmenten dargestellt.

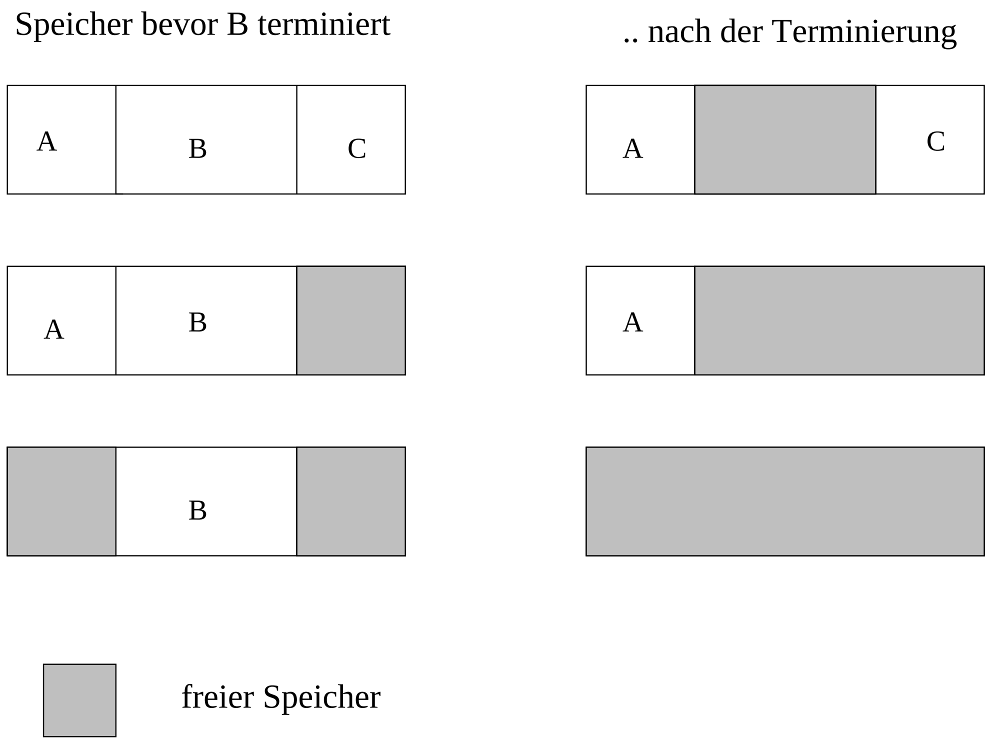

Abbildung 8: Beispiele für Speicherbelegungen

Durch die Sortierung der Segmente nach Adressen wird die Aktualisierung der Liste nach dem Terminieren eines Prozesses einfacher. Ein Prozeß hat in der Regel zwei Nachbarn (bis auf zwei Außnahmen), die entweder auch Prozesse oder Löcher, d.h. leere Speichersegmente sind. Liegen zwei Löcher nebeneinander, so verschmelzen diese, und die Liste wird um einen Eintrag kürzer. Wird der Speicher so verwaltet, so gibt es verschiedene Algorithmen, die Speicher für neu erzeugte oder eingelagerte Prozesse allokieren können. Es sei hier nur ein Beispiel genannt:

First Fit: Der Algorithmus sucht aufsteigend nach dem ersten Loch in der Liste, das mindestens die erforderliche Größe besitzt. Dann wird das Loch ggf. in zwei Segmente unterteilt: zum einen das Segment, das die Größe des vom Prozeß angeforderten Speichers besitzt, und zum anderen ein Restsegment, welches den „überschüssigen“ Speicher enthält.

6.2 Virtueller Speicher

Im Jahre 1961 wurde von Fotheringham ein Prinzip entwickelt, das bis heute eine große Bedeutung hat: Virtueller Speicher. Grundidee des virtuellen Speichers ist die, daß die zusammenhängende Größe des Programms, der Daten und des Stacks die Größe des tatsächlichen physikalischen Hauptspeichers überschreiten darf. Man spricht hier auch von dem virtuellen Adressraum, der mehr Adressen als der physikalische Adressraum umfaßt. Die Verwaltung des virtuellen Speichers übernimmt das Betriebssystem. Es hält die gerade aktiven Teile des Programms im Hauptspeicher und den Rest auf der Harddisk. Dardurch wird es möglich, daß auch ein 1MB-Programm auf einer 256kB-Hardware läuft, wobei dann zu jedem Zeitpunkt nur 256kB des Programms aktiv sein können. Das Prinzip wird häufig auch in Mehrprogramm-Betriebssystemen eingesetzt.

6.2.1 Paging

Die meisten virtuellen Speichersysteme benutzen als Technik zur Speicherverwaltung das sog. Paging. Wie zuvor beschrieben, ist bei diesem Verfahren der virtuelle Adressraum, der von den Programmen benutzt wird, in der Regel größer als der physikalischer Adressraum. Die virtuelle Adresse des Programms z.B. eines „MOVE REG,1000“-Befehls wird durch die sog. „Memory Management Unit“ (MMU) auf die physikalische Adresse umgerechnet.

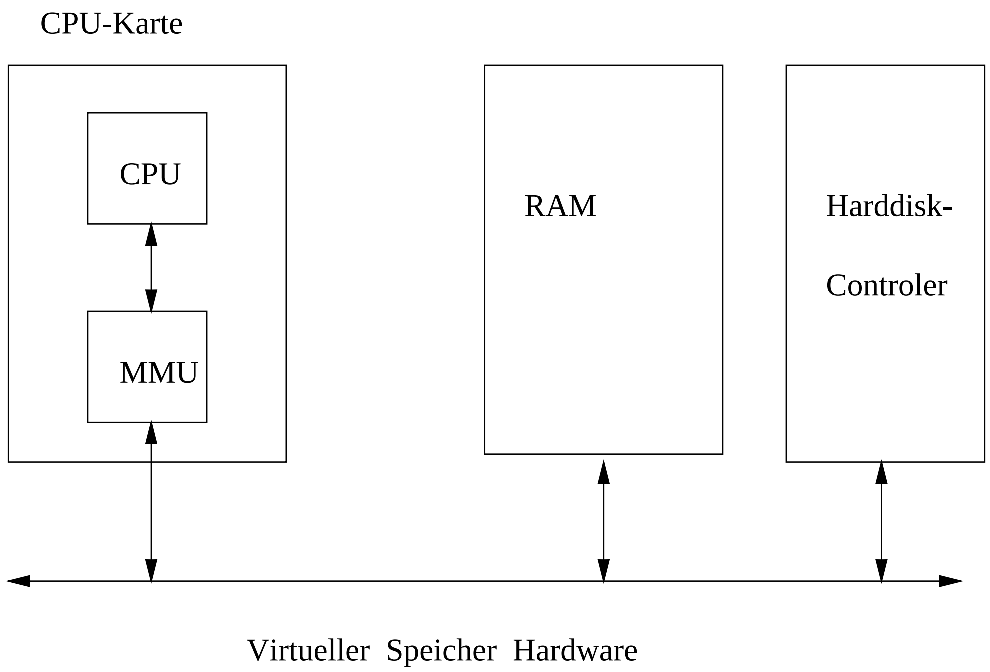

Abbildung 9: Die Memory Management Unit

Der virtuelle Adreßraum ist in Einheiten, die Seiten (Pages) unterteilt. Die dazu korrespondierenden Einheiten im physikalischen Adreßraum heißen Seitenrahmen.

Die MMU verwaltet die virtuellen und die physikalischen Seite mit Hilfe einer Seitentabelle. In der MMU wird für jede Seite, die sich im Hauptspeicher befindet, in der Seitentabelle ein Present-Bit gesetzt; ansonsten ein Absent-Bit. Wird von einem Programm eine Seite benutzt, die sich nicht im Speicher befindet, tritt ein sog. **Page Fault** auf: Die MMU erkennt, daß die Seite nicht eingeblendet ist (X siehe oben) und veranlaßt

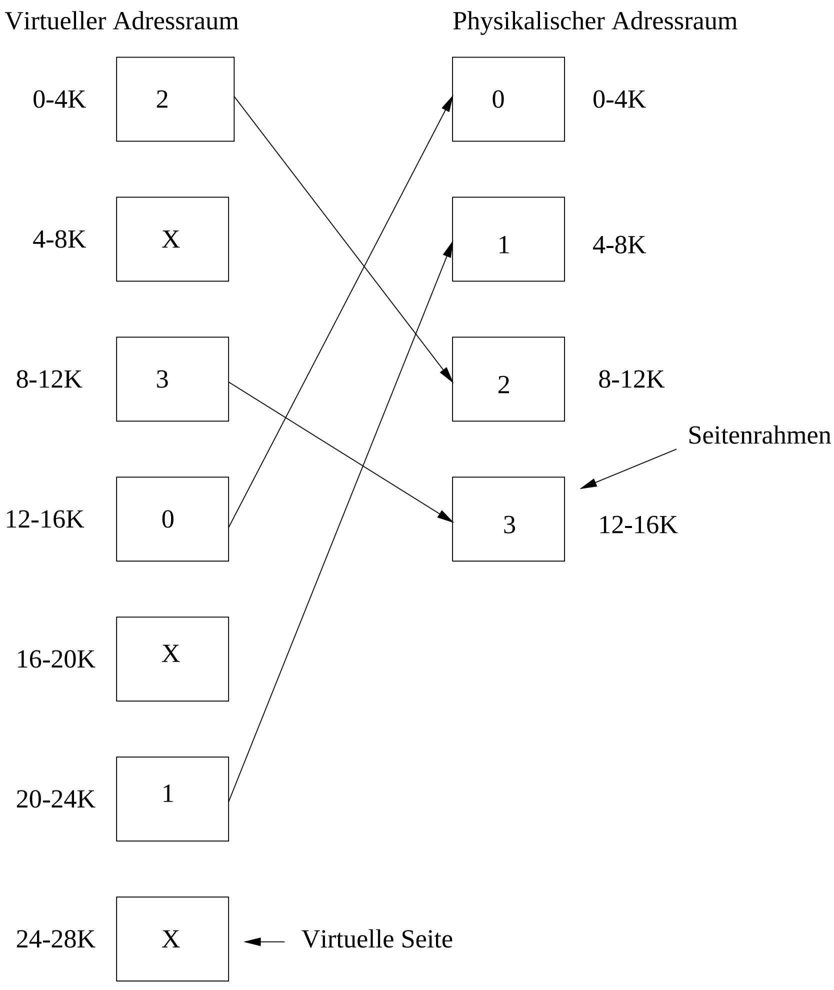

Abbildung 10: Virtueller und physikalischer Speicherraum

die CPU durch einen Interrupt, in das Betriebssystem zu springen. Das Betriebssystem ermittelt die am wenigsten frequentierte Seite im Speicher, lagert sie aus, schreibt die angeforderte Seite hinein und führt das unterbrochene Programm an der letzten Instruktion fort.

6.2.2 Die Seitentabelle

Der Zweck der Seitentabelle ist die Umrechnung der virtuellen Seite auf die Seitenrahmen. Mathematisch ist die Seitentabelle eine Funktion, die die virtuelle Seitennummer als Argument bekommt und eine physikalische Adresse, den Seitenrahmen, zurückliefert.

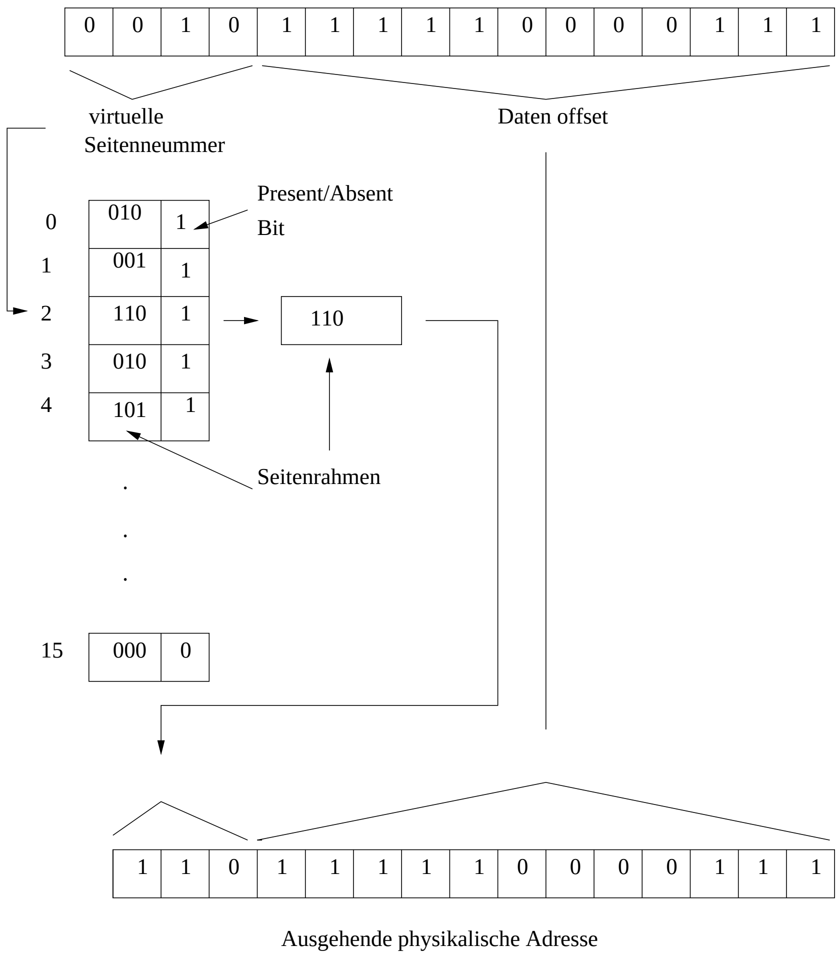

Abbildung 11: Die interne Operation der MMU mit 5 4K Seiten

Die Verwaltung der Seitentabelle bringt allerdings zwei Problemstellungen mit sich:

1. Die Seitentabelle kann sehr groß werden.

2. Die Umrechnung muß sehr schnell sein.

Um diese beiden Probleme zu bewältigen, wurden verschieden Verfahren zur Verwaltung der Tabelle entwickelt (siehe: Mehrstufige Seitentabelle, Tanenbaum Kap. 3.3).

7 Datei- und Verzeichnisverwaltung

7.1 Aufgaben der Dateiverwaltung

Zur Aufgabe von Dateien gehört zum einen die Verwaltung großer Datenmengen, insbesondere nach Beendigung der Prozesse, welche die Daten verarbeiten. Zum anderen ist dabei zu beachten, daß die Daten in den Dateien möglichst gleichzeitig mehreren Prozessen zur Verfügung stehen. Die Dateiverwaltung ist verantwortlich dafür, daß dabei keine Datenverluste entstehen können, sowie für die Verteilung des physikalischen Speicherplatzes (z.B. Festplatte) an die verschiedenen Dateien.

Die Festplatte ist üblicherweise in Blöcke unterteilt, in den Blöcken sind dann Bruchstucke der Dateien gespeichert. Um eine Datei als ganzes lesen zu können, muß sichergestellt sein, daß alle zu einer Datei gehörigen Blöcke gefunden werden.

7.2 Implementierung von Dateien

7.2.1 Kontinuierliche Allokation

Bei der kontinuierlichen Allokation wird eine der Größe der Datei entsprechende Anzahl von Blöcken auf dem Speichermedium belegt. Dazu wird eine Tabelle angelegt (File allocation table, FAT), in der angegeben ist, ob ein Block frei, belegt oder schadhaft ist. Für einen freien Block wird eine 0 eingetragen, schadhafte Blöcke werden in MS-DOS mit dem Wert FFF7 gekennzeichnet. Für einen belegten Block wird in der Tabelle angegeben, wo der nächste zu der Datei gehörende Block zu finden ist. Das Ende einer solchen Kette wird mit einem speziellen Zeichen gekennzeichnet (MS-DOS: FFFF).

0	FFF7	4	7	0	FFF7	8	A	FFFF	B	FFFF
1	2	3	4	5	6	7	8	9	A	B

Abbildung 12: Verkettung von Blöcken in einer FAT

Die in der Abb. 12 gezeigte FAT hat also zwei schadhafte Stellen (2 und 6), zwei freie Blöcke (1 und 5) und zwei Dateien. Die eine Datei belegt die Blöcke 3, 4, 7, 8, A und B, die andere nur den Block 9.
Der Nachteil dieser Methode ist, daß ein Fehler in der FAT dazu führt, daß der Rest der Datei nicht mehr aufgefunden werden kann. Außerdem muß immer die gesamte FAT im Hauptspeicher gehalten werden, diese kann mehrere Megabyte groß sein.

7.2.2 Indirekte Adressierung mittels I-Nodes

Anders als bei einer FAT werden z.B. bei UNIX sogenannte I-Nodes (Index Nodes) benutzt, um die Blöcke, die zu einer Datei gehören, zu adressieren. In dem I-Node sind außerdem Informationen über den Eigentümer, Zugriffsberechtigungen und andere Attribute enthalten.

Ein I-Node unter UNIX hat 13 Einträge, die zur Adressierung verwendet werden. Die ersten zehn Einträge zeigen direkt auf die ersten zehn Blöcke der Datei. Damit ist es möglich, Dateien bis zu 10 KByte direkt zu erreichen. Der elfte Eintrag zeigt auf einen weiteren I-Node, mit dem weitere Blöcke erreicht werden. Eintrag zwölf enthält – zur Adressierung größerer Dateien – die Adressen von weiteren I-Nodes, welche dann die Adressen der Blöcke enthalten. Der letzte Eintrag enthält wiederum Adressen von I-Nodes, die dem Typ aus Eintrag zwölf entsprechen. Das Vorgehen ist in Abb. 13 dargestellt.

Es zeigt sich, daß dieses System umso effektiver ist, je kleiner die Dateien sind. Der großte Teil der Dateien ist jedoch kleiner als 10 KByte und kann daher direkt referenziert werden.

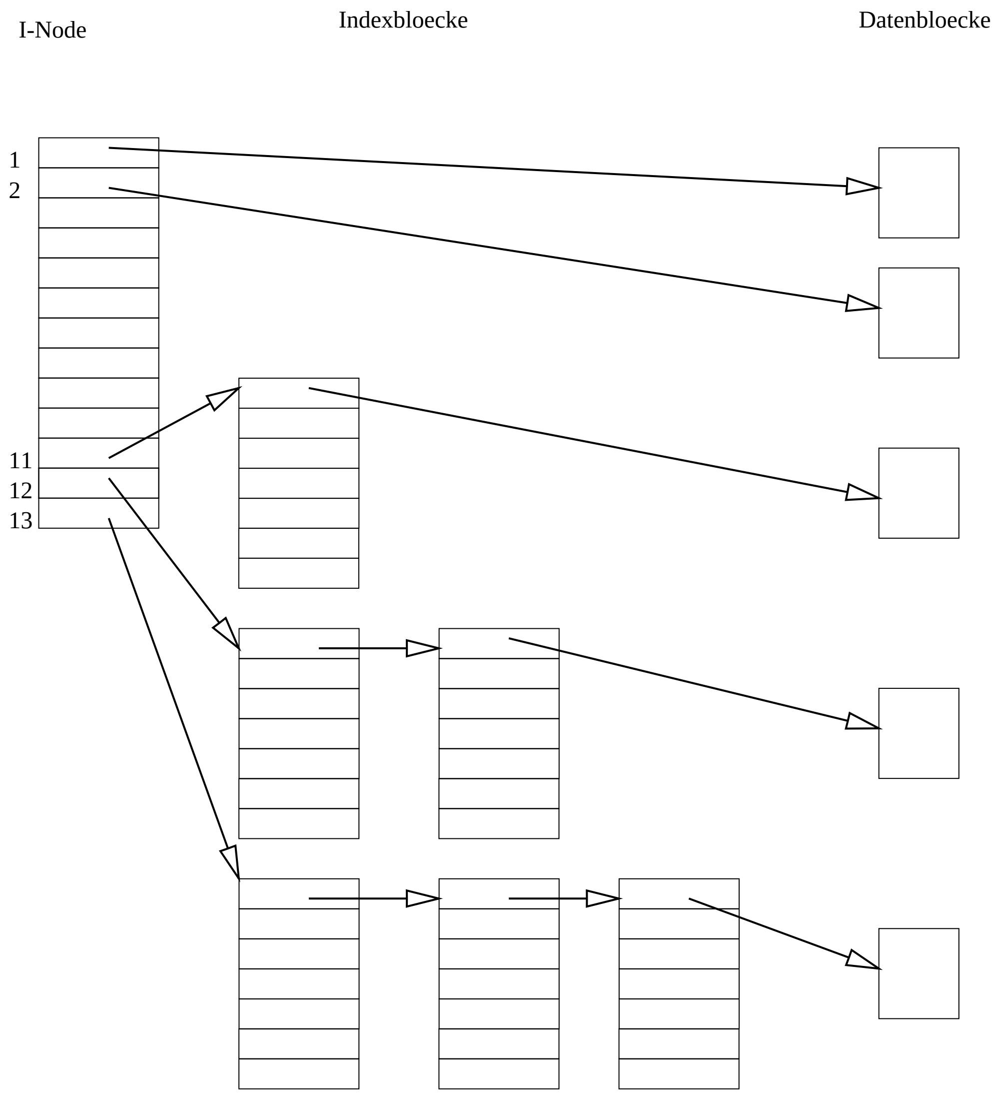

Abbildung 13: I-Nodes zur Datei-Adressierung

7.3 Implementierung von Verzeichnissen

Bei der Verwaltung von Verzeichnissen gibt es unter anderem auch wieder die zwei verschiedenen Möglichkeiten der direkten (MS-DOS) oder indirekten Adressierung (UNIX). Bei der direkten Adressierung wird prinzipiell so verfahren, wie bei der Verwaltung von Dateien. Bei der indirekten Adressierung von Verzeichnissen, wie z.B. unter UNIX, wird ein Verzeichniseintrag gemacht, der die I-Node-Nummer und den Namen des Verzeichnisses oder der Datei enthält (Abb. 14. In dem jeweiligen I-Node wird auf den Block verwiesen, in dem das Verzeichnis steht. Wie die Datei /homes/istudent/.xsession aufgefunden wird, zeigt Abb. 15.

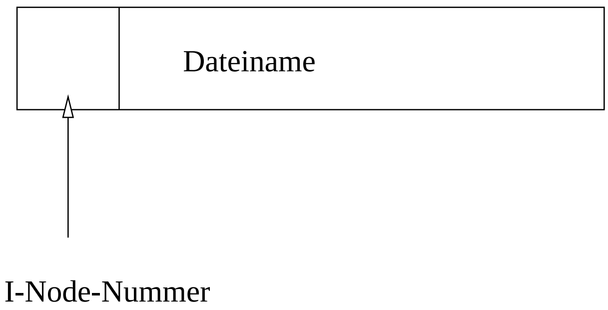

Abbildung 14: Ein UNIX-Verzeichniseintrag

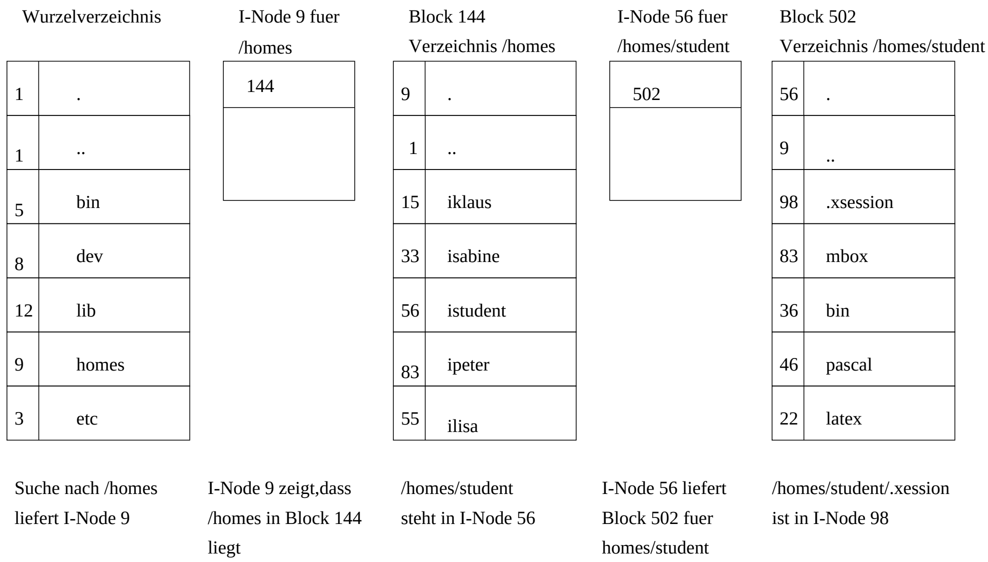

Abbildung 15: So wird /homes/student/.xsession gefunden

Mit der gefundenen I-Node-Nummer (im Beispiel: Nr. 98) kann nun, wie oben beschrieben, die Datei gelesen werden.

7.4 Gemeinsam benutzte Dateien

Soll eine Datei von meheren Benutzern gleichzeitig benutzt werden, so kann ein *Link* auf diese Datei gesetzt werden. Es gibt zwei Möglichkeiten, einen Link zu setzten.

- Beim *normalen Link* wird im I-Node der Datei festgehalten, daß ein Link gesetzt wurde. Beide Benutzer können nun die Datei bearbeiten. Versucht nun der erste Besitzer, die Datei zu löschen, so wird diese nur für ihn gelöscht. Für den Benutzer, der einen Link auf die Datei hat, bleibt die Datei erhalten. Diese Datei wird aber dennoch dem ersten Besitzer als owner zugeordnet. Die Datei wird erst endgültig gelöscht, wenn sie der *Linker* löscht.

- Beim *symbolischen Link* wird kein Eintrag in den I-Node gemacht. Löscht der Besitzer seine Datei, so ist sie auch für denjenigen gelöscht, der einen Link auf die Datei hatte. Es ist im allgemeinen besser, einen symbolischen Link zu setzen.

7.5 Plattenplatz-Management

Um den Platz einer Fetsplatte optimal ausnutzen zu können, ist es nötig, die kleinste Einheit, den Block, möglichst klein zu machen. Das hat den Nachteil, daß längere Dateien viele Blöcke brauchen und dementsprechend langsam adressiert werden können. Es muß also ein Mittelmaß zwischen Platz- und Zeitoptimierung gefunden werden. Dieses Mittelmaß liegt üblicherweise bei 512 Byte bis 2 KByte pro Block.