Betriebssysteme

Michael Jäger

15. Oktober 2006

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Was ist ein Betriebssystem?

Ein Betriebssystem ist ein komplexes Programm, das dem Rechnerbenutzer (Anwender, Programmierer, Administrator) folgendes bietet:

1. **Verwaltung der Peripheriegeräte**

   Das Betriebssystem bietet einen bequemen Zugriff auf Hardware wie Speichermedien (Platten, CD, Bänder), Drucker oder Bildschirm.

   Der Benutzer muss sich nicht um Gerätetechnik kümmern, auch nicht um die Koordination konkurrierender Zugriffe durch mehrere Prozesse.

2. **Abstrakter Rechner**

   Ein Betriebssystem kann die Eigenheiten der Rechnerplattform völlig verdecken. Es ermöglicht damit eine plattformunabhängige Rechnernutzung.

   Früher erlaubte dies zunächst nur eine einheitliche Verwendung von Rechnern derselben *Rechnerfamilie* eines Herstellers. Heute sieht man insbesondere am Beispiel UNIX, dass ein einziges Betriebssystem die Verwendung völlig unterschiedlicher Rechnertypen gestattet, vom PC über den Großrechner bis zum Supercomputer.

   Der Vorteil: Aufwendig entwickelte Software und mühsam erworbene Kenntnisse des Benutzers im Umgang mit dem Rechner sind dadurch plattformunabhängig und übertragbar.

3. **Sammlung von Dienstprogrammen**

   Ein Betriebssystem beinhaltet normalerweise eine ganze Reihe ständig benötigter Dienstprogramme, etwa Kopier- oder Sortierprogramme.

4. **Sammlung von Kommunikationsmechanismen**

   Das Betriebssystem ist die Schnittstelle zum Rechnernetz und dient somit als Ausgangspunkt für unterschiedlichste Kommunikationsaktivitäten.

In modernen Betriebssystemen sind oft verschiedene „Familien“ von Kommunikationsprotokollen (z.B. TCP/IP) integriert. Die Protokolle bilden die Basis für Kommunikationsdienste aller Art, im lokalen Netzwerk, unternehmensweit oder im Internet.

1.2 Grundbegriffe

Nachfolgend gehen wir der Einfachheit halber davon aus, dass ein Rechner nur einen (Zentral-)Prozessor (CPU = „central processing unit“) hat. Die Mechanismen zur Unterstützung von Mehrprozessormaschinen werden jeweils explizit und gesondert behandelt.

- **Prozess (Task)**

  Ein Prozess ist ein Programm, das gerade ausgeführt wird. Die Definition wird in **??**, S. **??** verfeinert.

- **Multitasking**

  Ein Multitasking-Betriebssystem führt mehrere Programme **gleichzeitig** aus. Genauer gesagt stehen mehrere Prozesse zur selben Zeit im Hauptspeicher.

  Damit ein Programm Bearbeitungsfortschritte macht, benötigt es den Prozessor des Rechners. Es entsteht eine Konkurrenzsituation: *N* im Speicher befindlichen Programme konkurrieren um einen Prozessor.

  Zwei Aktivitäten sind **nebenläufig** (engl. „concurrent“), wenn sie zeitlich „nebeneinander“, also parallel ausführbar sind. Programme werden im Multitaskingsystem nicht nacheinander ausgeführt, sondern nebenläufig, d.h. zeitlich verschränkt („quasiparallel“) oder - bei Mehrprozessorsystemen - echt parallel.

  Die **quasiparallele** Bearbeitung mehrerer Prozesse in einem Einprozessorsystem wird dadurch gewährleistet, dass das Betriebssystem allen Prozessen reihum jeweils für ein kurzes **Zeitquantum** (z.B. 100ms) die CPU zuteilt. Aus Sicht des einzelnen Prozesses gibt es also abwechselnd immer wieder Wartephasen und Bearbeitungsphasen.

  Aus Sicht des Betriebssystems ist die CPU eine Ressource, die abwechselnd den darum konkurrierenden Prozessen zugeteilt wird. Ein Wechsel der CPU-Zuteilung von einem Prozess zu einem anderen heißt **Kontextwechsel**. Die Strategie, nach der die CPU an die konkurrierenden Prozesse vergeben wird, bezeichnet man als „Scheduling“-Strategie. (Scheduling heißt in diesem Zusammenhang „Einplanung“ oder „Terminplanung“.) Die verantwortliche Betriebssystemfunktion nennen wir den **„Scheduler“**.

- **Mehrbenutzerbetrieb (Timesharing-Betrieb)**

  Mehrbenutzerbetrieb ermöglicht es verschiedenen Nutzern, gleichzeitig mit einem Rechner zu arbeiten. Die Benutzer können z.B. mit direkt an den Rechner angeschlossenen Terminals arbeiten, oder über Netzwerkverbindungen.

Beim heute üblichen Timesharing-Betrieb hat jeder Nutzer das Gefühl, den Rechner alleine zu nutzen. Dies erfordert neben Multitasking-Unterstützung vom Betriebssystem eine Zuordnung von Ressourcen - insbesondere Dateien und Hauptspeicherbereichen - zu Benutzern (bzw. Prozessen) und eine strikte Zugriffsrechtskontrolle. Kein Programm soll die Daten eines anderen Programms zerstören oder auch nur lesen können, sei es durch fehlerhafte Programmierung oder aus Absicht.

Die immer wieder auftretenden Wartephasen eines interaktiven Prozesses müssen beim Mehrbenutzerbetrieb für den Anwender unmerklich kurz sein. Dann sieht er einen kontinuierlichen Bearbeitungsfortschritt seines Programms.

- **Multiprozessorsystem**

  Ein Rechnersystem mit mehreren Prozessoren heißt Multiprozessorsystem. Das Betriebssystem unterstützt Multitasking. Es muss dafür sorgen, dass soviele Prozesse echt parallel bearbeitet werden können, wie Prozessoren vorhanden sind (vgl. auch *Multithreading* – **??**, S. **??**).

- **Stapelverarbeitung (Batchbetrieb)**

  Bei Stapelverarbeitungsbetrieb stellt ein Benutzer einen Bearbeitungsauftrag (Job) zur späteren Bearbeitung in eine Auftragswarteschlange. Ein Auftrag besteht aus einer Reihe aufzurufender Programme und zugehörigen Parametern.

  Das Betriebssystem arbeitet die Auftragsstapel nach und nach ab. So können langwierige Berechnungen in die Nacht oder aufs Wochenende verlegt werden.

- **Echtzeitsystem**

  Ein Echtzeitsystem („real time system“) dient zur Kontrolle und Steuerung rechnerexterner Abläufe (z.B. Steuerung eines Kraftwerks). Das Betriebssystem muss für dringende externe Signale bestimmte anwendungsabhängige maximale Reaktionszeiten garantieren können.

  Viele Betriebssysteme können dies deshalb nicht, weil eine sofortige Unterbrechung bestimmter Betriebssystemaktivitäten die vom Betriebssystem selbst benötigten Datenstrukturen in einem inkonsistenten Zustand hinterlassen würde.

- **Virtuelle Maschine**

  Eine virtuelle Maschine ist ein Betriebssystem, das auf einem Rechner andere Rechner und Betriebssysteme emulieren kann. Die bekannteste virtuelle Maschine ist das IBM-System *VM* („virtual machine“). Ein anderes bekanntes Beispiel ist „vmware“, das auf einem realen PC beliebig viele „virtuelle“ PCs erzeugen kann, so dass mehrere Betriebssysteme wie z.B. Windows, Linux oder BSD-UNIX gleichzeitig genutzt werden können.

- **Verteiltes Betriebssystem**

  Ein verteiltes Betriebssystem erlaubt (im Idealfall) dem Anwendungsentwickler, eine Gruppe vernetzter Rechner so zu benutzen wie einen einzigen Rechner. Es

verteilt - möglichst transparent für den Anwender - die Bearbeitung des Anwendungsprogramms auf mehrere Rechner im Netz. Es besteht aus mehreren auf die vernetzten Rechner verteilten und miteinander kooperierenden Prozessen.

Kapitel 2

Aufbau eines Betriebssystems

Wir beschreiben zwei konkurrierende Architekturmodelle:

- Monolithische Architektur – der „klassische“ Aufbau
- Mikrokern-Architektur – die „moderne Konkurrenz“

Anschliessend diskutieren wir ein Schichtenmodell für ein monolithisches Betriebssystem.

2.1 Monolithische Architektur

Der Anwender kann das Betriebssystem als eine Sammlung von Funktionen ansehen, die bei Bedarf aufgerufen werden, genauso wie Funktionen des Anwendungsprogramms oder Funktionen einer Standardbibliothek. Natürlich gehören zum Betriebssystem auch noch eine ganze Reihe von Datenstrukturen zur Verwaltung der Geräte, Benutzer, Prozesse usw.

Stellen wir uns also vor, ein Betriebssystem ist in einer höheren Programmiersprache wie C implementiert und besteht aus Funktionen und Daten. Wenn das Betriebssystem in Form eines einzigen Programms vorliegt, heißt es monolithisch. Siehe dazu Abbildung **??**.

2.1.1 Systemaufruf

Die Implementierung von Systemaufrufen ist sehr aufschlussreich für das Verständnis der Systemarchitektur.

Betrachten wir ein Anwendungsprogramm, das genau wie das Betriebssystem aus Funktionen und Daten besteht. [1] Dieses Anwendungsprogramm benutzt regelmäßig

[1] Wir vergessen dabei für einen Moment, dass Anwendungen heute oft separate, dynamisch angebundene (und gemeinsam benutzbare) Bibliotheken verwenden.

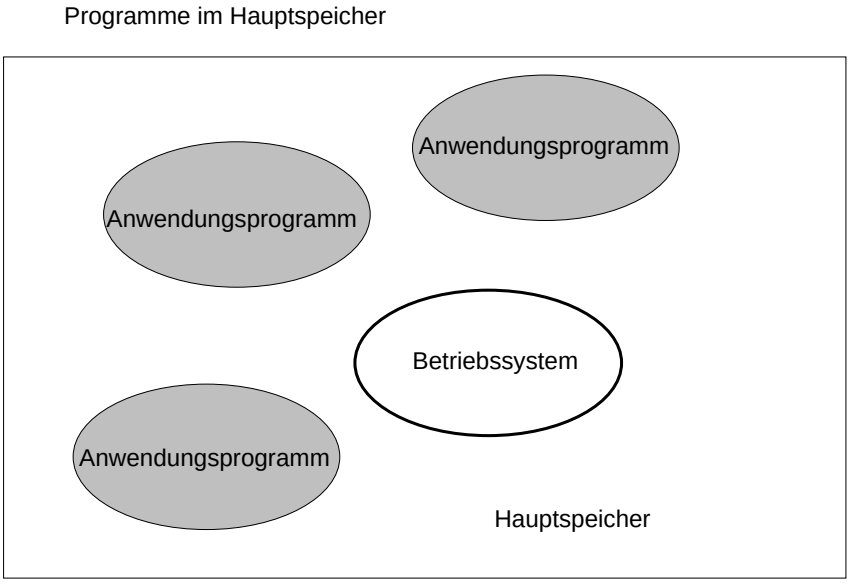


Abbildung 2.1: Programme im Hauptspeicher bei monolithischer Architektur

Betriebssystemdienste durch Aufruf von Betriebssystemfunktionen. Solche Aufrufe sind insofern etwas besonderes, als die aufgerufenen Systemfunktionen ja Bestandteil eines *anderen Programms* sind.

Betrachten wir ein Beispiel: Das Anwendungsprogramm ruft innerhalb der Funktion *main* eine Systemfunktion *write* zum Schreiben von Daten auf den Bildschirm auf.

Die *write*-Funktion ist, genauer betrachtet, nicht Bestandteil des Betriebssystems, sondern eine dem Anwendungsprogramm zugehörige Funktion. Deren Aufgabe ist die Aktivierung einer entsprechenden Betriebssystemfunktion, nehmen wir an, diese heißt *syswrite*.

Dazu wird ein spezieller Hardwaremechanismus verwendet (*SVC = „supervisor call“* oder *Trap*): Die Anwendung (*write*-Funktion) hinterlegt eine dem Systemaufruf (*syswrite*) zugeordnete Identifikation zusammen mit den Aufrufparametern an einer bestimmten dem Betriebssystem bekanntgemachten Stelle im Hauptspeicher und erzeugt einen Trap. Der Hardware-Trap unterbricht die Ausführung des Anwendungsprogramms (also der *write-Funktion*) und veranlasst den Aufruf einer Behandlungsroutine („trap handler“), die Bestandteil des Betriebssystems ist.

Die CPU-Kontrolle ist damit vom Anwendungsprogramm (Prozess im *Anwendungsmodus* / „user mode“) an das Betriebssystem übergeben worden. Der Prozess befindet sich nun im *Systemkernmodus* („*kernel mode*“). Dieser Zustandübergang spiegelt sich auf der Hardwareebene wieder: Beim Übergang wird der Prozessor in einen privilegierten Modus umgeschaltet. In diesem Modus ist der Befehlssatz ggf. erweitert und hardwareseitige Speicherzugriffsbeschränkungen sind aufgehoben.

Die von der Hardware aktivierte Trap-Behandlungsroutine kopiert die Parameter für *syswrite* in den Speicherbereich des Betriebssystems. Anhand der hinterlegten Systemfunktions-Identifikationsnummer kann die Trap-Behandlungsroutine feststellen,

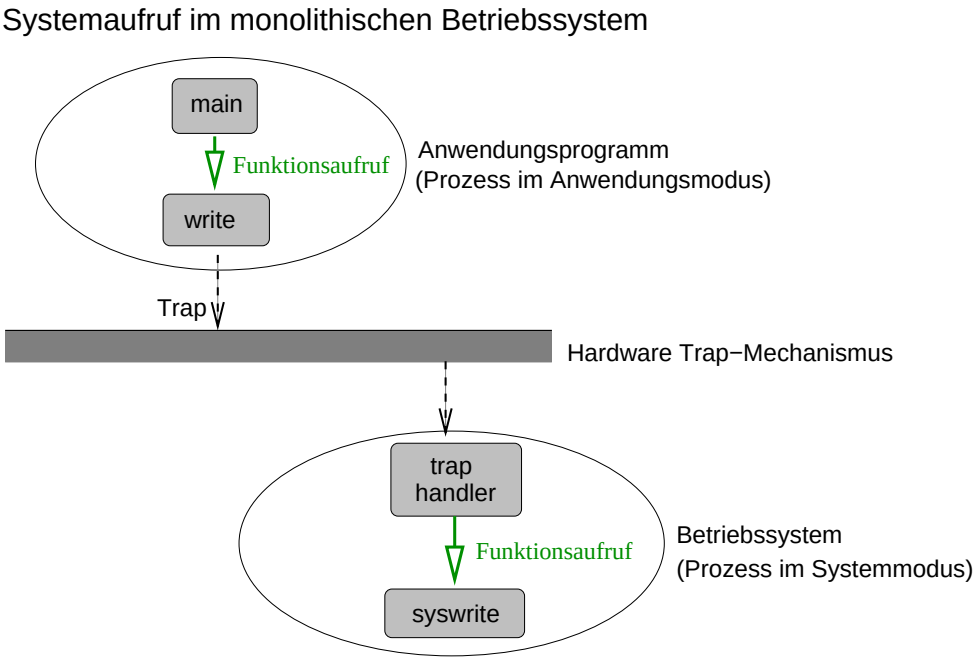


Abbildung 2.2: Sytemaufruf im monolithischen System

dass die Systemfunktion *syswrite* aufgerufen werden muss. Nach Prüfung der Argumente auf Konsistenz wird von der Trap-Behandlungsroutine der Aufruf der gewünschten Systemfunktion durchgeführt.

Nach Beendigung des *syswrite*-Aufrufs kopiert die aufrufende Funktion das Funktionsresultat bzw. Fehlercodes in den Speicherbereich der Anwendung und beendet die Trap-Behandlung. Der Prozessor wird wieder in den normalen (unprivilegierten) Zustand zurückversetzt und die Kontrolle wieder an die *write*-Funktion des Anwendungsprogramms übergeben.

2.2 Mikrokern-Architektur

Softwaretechnisch ist gegenüber einer monolithischen Struktur eine klar modularisierte Architektur vorzuziehen. Anstelle eines einzigen großen „Betriebssystem-Programms“ werden die Systemdienste auf mehrere separate Programme verteilt. Da diese Programme auftragsorientiert Dienstleistungen erbringen, bezeichnen wir ihre Aktivierungen als *Server*-Prozesse.

Ein Anwendungsprozess kommuniziert mit einem Serverprozess durch einen Interprozesskommunikations-Mechanismus (IPC=„interprocess communication“). Er sendet dem Serverprozess einen Auftrag und erhält eine Antwort.

Natürlich können nicht alle Betriebssystemfunktionen in Form separater Serverprozesse implementiert werden. Zumindest eine grundlegende Prozessverwaltung, CPU-Zuteilung und Interprozesskommunikation wird durch einen in klassischer Weise akti-

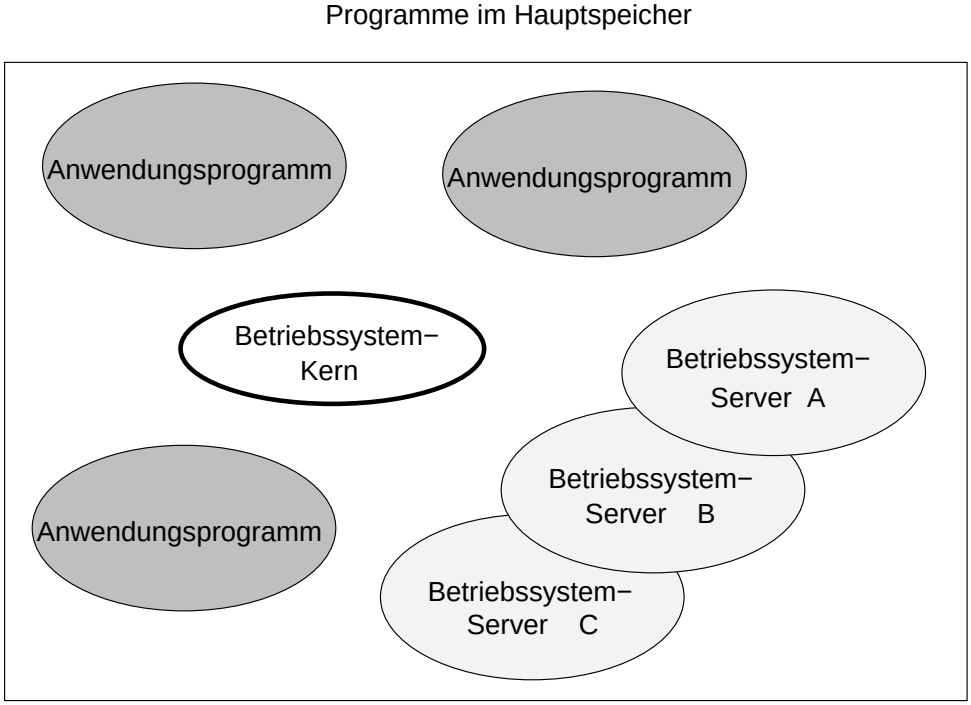


Abbildung 2.3: Programme im Hauptspeicher – Mikrokern-System

vierten Systemkern erfolgen. Wie sollte sonst ein Anwendungsprozess überhaupt entstehen, wie sollte er Nachrichten mit einem Serverprozess austauschen können bzw. den Prozessor zur Auftragsausführung an diesen übergeben?

Wenn also nur die Basisfunktionalität eines Betriebssystems durch einen Trap-aktivierten Systemkern übernommen wird, ist dieser Kern (hoffentlich) recht klein und wird deshalb *Mikrokern* („micro kernel“) genannt. Andere Betriebssystemdienste sind durch ständig aktive, auf Aufträge wartende [2] Betriebssystem-Serverprozesse realisiert, z.B. Dateisystemdienste, Hauptspeicherverwaltungsstrategien, graphische Benutzeroberfläche.

2.2.1 Systemaufruf

Die Implementierung von Systemaufrufen ist bei Mikrokernsystemen anders realisiert als bei monolithischen Systemen: durch Auftragserteilung an einen Serverprozess unter Verwendung des zugrundeliegenden IPC-Mechanismus.

Man beachte, dass der Anwendungsprozess einen Trap zur Aktivierung des Mikrokerns benutzt, um dem Systemdienstserver eine Nachricht zu schicken. Der Systemdienstserver benutzt seinerseits einen Trap zum Empfang.

Ein Systemdienstserver kann seinerseits über die Trap-Schnittstelle die Basisdiens-

[2] Ein Serverprogramm kann auch bei Bedarf erst geladen werden. Dies macht insbesondere dann Sinn, wenn seine Dienste nur selten benötigt werden.

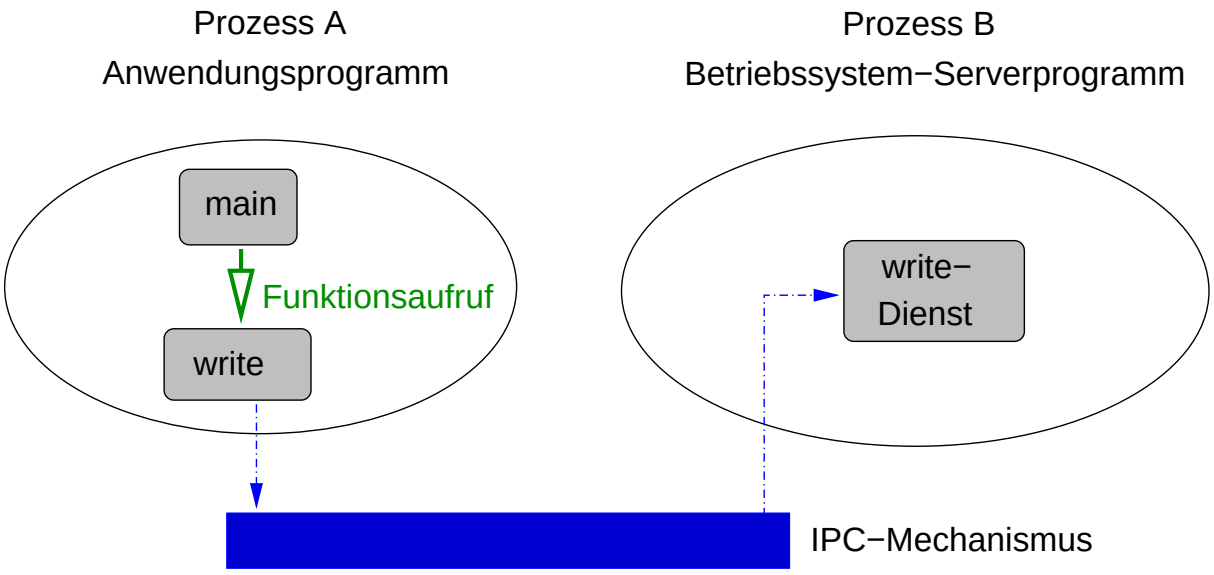


Abbildung 2.4: Systemaufruf bei Mikrokernsystemen

Die Rolle des Systemkerns bei der Nachrichtenübertragung

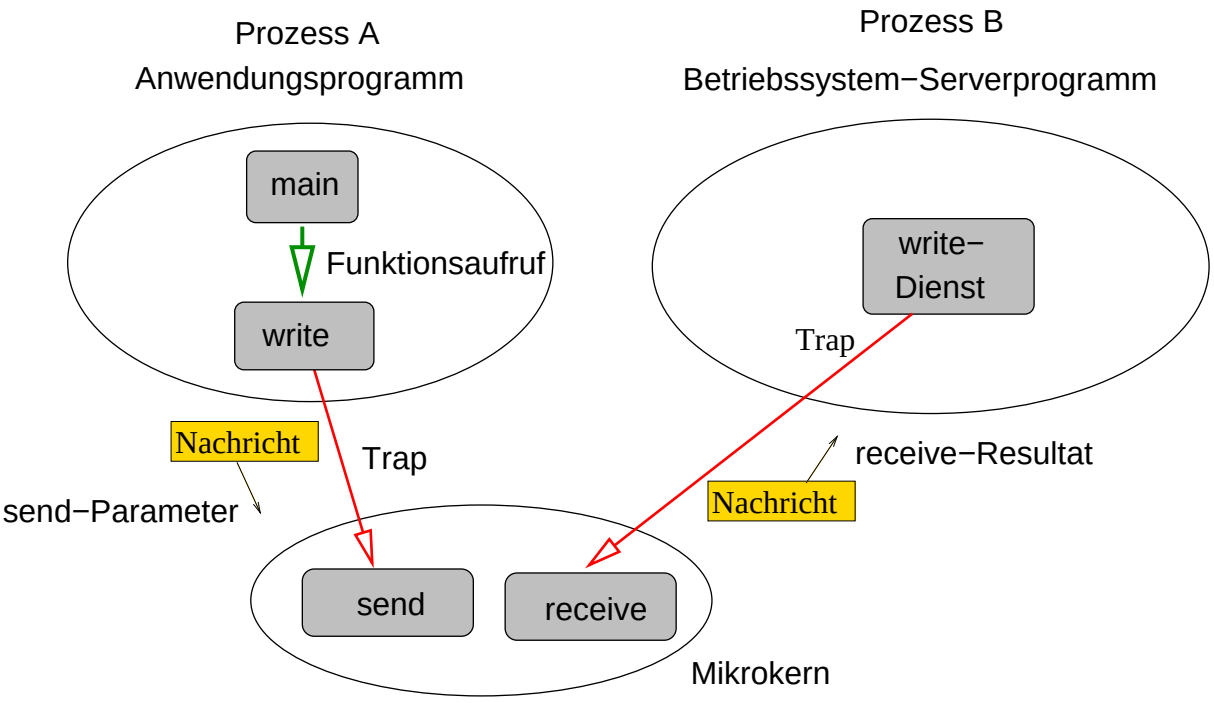


Abbildung 2.5: Systemaufruf und Nachrichten bei Mikrokernsystemen

te des Mikrokerns benutzen, weil er Basisdienste benötigt, um seinen Auftrag auszuführen, oder um Resultate an den auftraggebenden Prozess zurückzuschicken.

Eine spezielle Variante eines Systemdienstservers ist ein Server, der eine komplette Systemschnittstelle implementiert. So gibt es für das Betriebssystem *Mach* einen Serverprozess, der die UNIX-Systemaufrufschnittstelle komplett implementiert. Auch das POSIX-Subsystem von *Windows NT* dient diesem Zweck.

Weitere Formen der Arbeitsteilung sind möglich:

- Ein Systemdienstserver delegiert einen Teil der Arbeit an einen anderen Systemdienstserver
- Der Mikrokern benutzt einen Systemdienstserver für seine Zwecke

2.2.2 Vor- und Nachteile

Die Modularisierung des Betriebssystems in Form eines Mikrokerns und separater Serverprozesse macht das Betriebssystemdesign klarer, flexibler und von der Komplexität her besser beherrschbar. Probleme bereitet noch die Effizienz der Systemaufrufe, da bei der Client-Server-Struktur zusätzlicher Aufwand anfällt:

- Der Auftrag muss zum Systemdienstserver übertragen werden
- Die CPU-Kontrolle muss an den Systemdienstserver übertragen werden
- Das Resultat muss zum Anwendungsprozess zurückgeschickt werden
- Die CPU-Kontrolle muss an den Anwendungsprozess übertragen werden

Mit verschiedenen Optimierungsmethoden versucht man, den Zusatzaufwand in Grenzen zu halten.

2.3 Schichtenmodell

Innerhalb monolithischer Betriebssysteme lassen sich meist mehrere aufeinander aufbauende Schichten mit folgenden Eigenschaften identifizieren:

- Jede Schicht definiert eine Reihe von Funktionen. Diese Funktionen werden als Diensteschnittstelle für die darüberliegende Schicht angeboten.
- Die Implementierung der Funktionen einer Schicht benutzt ausschliesslich die Schnittstelle der darunter angeordneten Schicht, ein Durchgriff nach „weiter unten“ ist weder nötig noch möglich.
- Jede Schicht entspricht einer Abstraktionsebene in einem abstrakten Modell des Rechnersystems, „unten“ heißt hardwarespezifisch, „oben“ heißt anwendungsorientiert und völlig hardwareunabhängig.

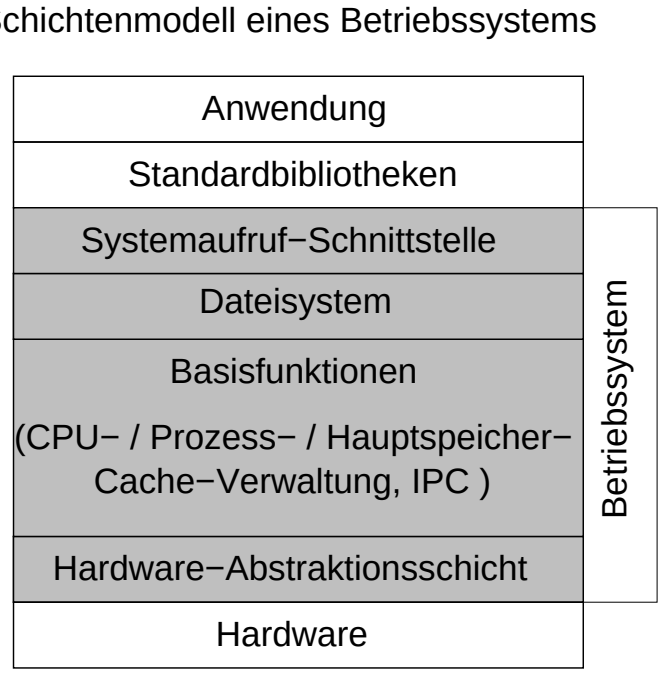


Abbildung 2.6: Schichtenmodell

In diesem Modell dient die Hardware-Abstraktionsschicht zur Kapselung hardwareabhängiger Teile des Systems. Dazu gehören

1. plattformspezifische Funktionen

   Diese Funktionen isolieren die Eigenschaften einer Rechnerplattform, d.h. eines Rechnertyps wie „Pentium-PC“ oder „Sun Ultra 1“. Plattformspezifische Hardware sind z.B. CPU mit Speicherverwaltungseinheit (MMU), Cache und Bussysteme.

   Zu den plattformspezifischen Funktionen gehören z.B. hardwareabhängige Anteile des Kontextwechsels oder der Hauptspeicherverwaltung.

   Die Kapselung der Plattformeigenschaften ist Voraussetzung für eine einfache Portierung eines Betriebssystems auf unterschiedliche Plattformen.

2. Hardwaretreiber

   Ein Treiber ist ein Modul zur Steuerung eines bestimmten, meist optional in den Rechner integrierbaren Hardware-Controllers (z.B. Festplatten, Netzwerkkarten, Scanner, Graphikkarte). Diese Module werden genau dann benötigt, wenn ein entsprechendes Gerät genutzt werden soll und können oft dynamisch in das Betriebssystem eingebunden werden.

Als Beispiel für die Nutzung aller Schichten betrachten wir ein C-Programm, das unter UNIX eine Festplattenausgabe macht.

- Das Programm benutzt eine Funktion aus der ANSI-C-Standardbibliothek, (z.B. *printf*). Diese ist **betriebssystemunabhängig**.

- Die Standardbibliothek benutzt die Systemschnittstelle des Betriebssystems, im Beispiel die UNIX-Schnittstellenfunktion *write*. Diese ist **betriebssystemspezifisch**.
- Die *write*-Funktion wird über den Systemaufrufmechanismus eine Funktion des *Dateisystems* aufrufen. Diese Funktion wird z.B. Zugriffsrechte prüfen und schließlich die Daten in den Cache-Bereich übertragen.
- Das Cache-Modul übernimmt die Daten in einen als Puffer reservierten Hauptspeicherbereich. Das Modul stellt sicher, dass nicht für jedes einzelne auszugebende Byte ein physikalischer Plattenzugriff erfolgt, indem es zunächst einmal die auszugebenden Daten sammelt.

  Zu einem späteren Zeitpunkt wird ein „Ein-/Ausgabe-Auftragspaket“ (*„I/O-Request“*) für den Plattentreiber erstellt und damit der Treiber beauftragt, die Daten tatsächlich auf die Platte zu übertragen.
- Der Treiber überträgt die Hardware-Befehle zur Ausgabe der Daten an den zuständigen Controller. Dies ist **hardwarespezifisch**.

Ein solches Modell kann nur als Annäherung an die Schichtenarchitektur realer Systeme betrachtet werden. In unserem UNIX-Beispiel gibt es sicher keine Möglichkeit, am Dateisystem vorbei irgendwelche Geräte zu manipulieren, (Ein-/Ausgabe von Daten). Ein Erfragen der Systemzeit benötigt jedoch keine Dateisystemfunktion, erfolgt als direkter Zugriff auf Information, die von der Schicht unterhalb des Dateisystems geliefert wird.

Kapitel 3

Prozesse und Threads

3.1 Prozessmodell, Grundbegriffe

Ein Prozess ist ein Programm zur Ausführungszeit, sozusagen die dynamische Sicht eines Programms. Für das Betriebssystem ist ein Prozess ein zu verwaltendes komplexes Objekt mit diversen Bestandteilen.

Ein Prozess benötigt zur Ausführung bestimmte Ressourcen, wie Hauptspeicher, Prozessor, Dateien, E/A-Geräte (E/A = Ein-/Ausgabe).

Zur Kontrolle der Prozesse verwendet das Betriebssystem eine **Prozesstabelle**, in der für jeden Prozess eine Reihe prozessspezifischer Verwaltungsinformationen gespeichert wird. Wir nennen den zu einem Prozess gehörenden Eintrag in der Tabelle **Prozesstabelleneintrag** oder **Prozessdeskriptor**.

Prozessbestandteile:

- **Programmspeicher** des Prozesses: programmspezifische Hauptspeicherbereiche
  - Code - Maschinenebefehle (durch Compiler und Linker aus dem Programmquelltext erzeugt)
  - Statische Daten - statisch (für die gesamte Ausführungszeit) reservierte Speicherbereiche für Variablen und Konstanten des Programms
  - Laufzeitstack - dynamisch verwalteter Speicher, in dem die für Unterprogrammaktivierungen notwendigen Daten (Parameter, lokale Variablen, Rücksprungadresse usw.) abgelegt werden

  Hinzu kommen bei vielen Systemen noch ein Speicherbereich, den wir „Programmkontext“ (engl: „environment“) nennen wollen. Er enthält Argumente, die dem Programm beim Aufruf übergeben wurden, und Umgebungsvariablen.

  Was ist mit dem „Heap“-Speicher, der für dynamisch erzeugte Datenstrukturen benötigt wird ? Aus Sicht des Betriebssystems wird typischerweise Stack und Heap im selben Speicherbereich untergebracht.

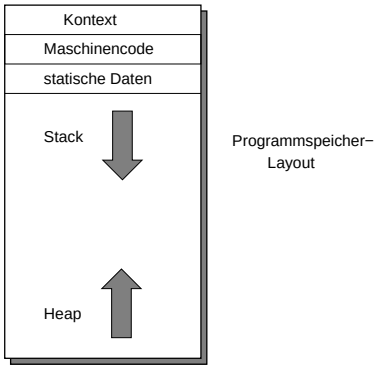


Abbildung 3.1: Programmspeicher-Layout eines Prozesses

Der Programmspeicher ist für andere Prozesse nicht zugreifbar.

- Bestandteile des Prozessdeskriptors
  - eindeutige Prozessidentifikation
  - Zustand (vgl. Zustandsmodell)
  - Zugriffsrechtsdeskriptor - definiert die Rechte des Prozesses beim Zugriff auf Ressourcen wie Dateien, Hauptspeicherbereiche usw.
  - Dateideskriptoren - definieren den Zustand der vom Prozess geöffneten Dateien (z.B. aktuelle Lese-/Schreibposition, Öffnungsmodus)
  - Hauptspeicherdeskriptor - dient zum Auffinden und zur Zugriffskontrolle der zum Prozess gehörigen Speichersegmente
  - Maschinenzustand (Register, Programmzähler, Statuswort usw.)
    (auch Prozesskontrollblock genannt)

    Er wird bei Unterbrechung der Prozessausführung gespeichert, um nach der Unterbrechungsphase die Wiederherstellung des Maschinenzustands und damit die Fortführung der Ausführung zu ermöglichen
  - Priorität(en) - bestimmen die Geschwindigkeit der Prozessbearbeitung
  - Ressourcenverbrauch - für Abrechnungszwecke (CPU-Verbrauch, Hauptspeichernutzung usw.)

  Der Prozessdeskriptor[1] wird vom Betriebssystem angelegt, aktualisiert und gelöscht. Der Anwender kann üblicherweise lesend darauf zugreifen (vgl. z.B. das UNIX-Kommando *ps* – „process status“)

  Komplexe Prozessdeskriptor-Bestandteile (z.B. Dateideskriptor, Hauptspeicherdeskriptor) werden durch separat gespeicherte Datenstrukturen realisiert, im Prozessdeskriptor verweist dann je ein Zeiger darauf.

[1] In der Literatur wird der Prozessdeskriptor teilweise Prozesskontrollblock genannt, wir haben diese Bezeichnung aber schon für den innerhalb des Prozessdeskriptors gesicherten Maschinenzustand verwendet. In der UNIX-Literatur findet man auch die Bezeichnung Prozessstruktur, da im C-Quelltext dafür der Typ `struct process` verwendet wird.

3.2 Prozesszustände

3.2.1 Einfaches Zustandsmodell

Wir unterscheiden zunächst drei Prozesszustände:

1. **ausführend** (engl. *running*)

   Der ausführende Prozess hat die Kontrolle über den Prozessor. In einem Mehrprozessorsystem mit N Prozessoren gibt es N ausführende Prozesse.

2. **bereit** (engl. *ready*)

   Ein Prozess, der bereit ist zur Ausführung (im Gegensatz zu einem blockierten Prozess). Die Prozesse in diesem Zustand konkurrieren um die Prozessorzuteilung.

3. **blockiert** (auch „schlafend“ oder „wartend“)

   Ein Prozess, der auf ein Ereignis wartet, z.B. auf

   - Dateneingabe von der Tastatur
   - die Übertragung von Daten zwischen Festplatte und Hauptspeicher (aus Sicht des Prozessors relativ langwierig)
   - eine Nachricht von einem anderen Prozess

   Ein blockierter Prozess konkurriert nicht mit anderen Prozessen um die Zuteilung des Prozessors.

Der Zustandübergangsgraph in Abbildung **??** zeigt die möglichen Zustandübergänge. Man sieht leicht ein, dass nicht beliebige Übergänge von einem Zustand in jeden anderen Zustand sinnvoll sind:

- Ein Prozess blockiert sich selbst, indem er einen blockierenden Systemaufruf ausführt, z.B. Lesen von der Tastatur. Dies kann er nur im ausführenden Zustand, nicht im Zustand *bereit*.

- Es macht keinen Sinn, einem blockierten Prozess den Prozessor zu überlassen.

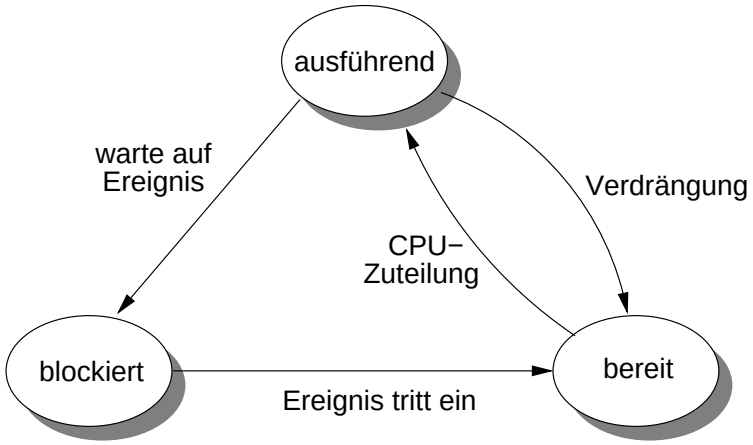


Abbildung 3.2: Einfaches Zustandsmodell

3.2.2 Erweiterungen des Zustandsmodells

Bei realen Betriebssystemen gibt es weitaus mehr Prozesszustände. Wir betrachten drei Erweiterungsmöglichkeiten:

Neue und terminierte Prozesse

- Ein Prozess ist im Zustand **neu**, wenn er schon erzeugt wurde, aber noch keine Prozessorzuteilung hatte. Der Zustand gleicht dem Zustand *bereit*, allerdings haben neue Prozesse in der Prozesstabelle noch keinen abgespeicherten Maschinenzustand (PCB).

- Ein Prozess ist im Zustand **terminiert**, wenn die Bearbeitung schon beendet wurde, aber der Prozessdeskriptor noch für irgendwelche Zwecke aufbewahrt wird.

  Beispiel: Bei UNIX-Systemen steht ein neuer Prozess in einem Kindschaftsverhältnis zu seinem Erzeugerprozess. Terminiert ein Prozess, hat sein Elternprozess die Möglichkeit, mit einem *waitpid*-Systemaufruf dessen Terminierungsstatus zu erfragen. Bis er dies tut, wird der Kind-Prozessdeskriptor (als sogenannter „zombie“) in der Prozesstabelle aufbewahrt.

Ausgelagerte Prozesse

Bei Hauptspeichermangel können die meisten Betriebssysteme Prozesse temporär vom Hauptspeicher auf eine Platte („swap“-Bereich) auslagern. Durch die Unterscheidung „eingelagert“/“ausgelagert“ bei diversen Zuständen vergrößert sich die Zustandsmenge.

Abbildung **??** zeigt ein solches Modell. Auslagerungskandidaten sind hier nur blockierte Prozesse. Man beachte, dass es nicht unbedingt sinnvoll ist, ausgelagerte Prozesse

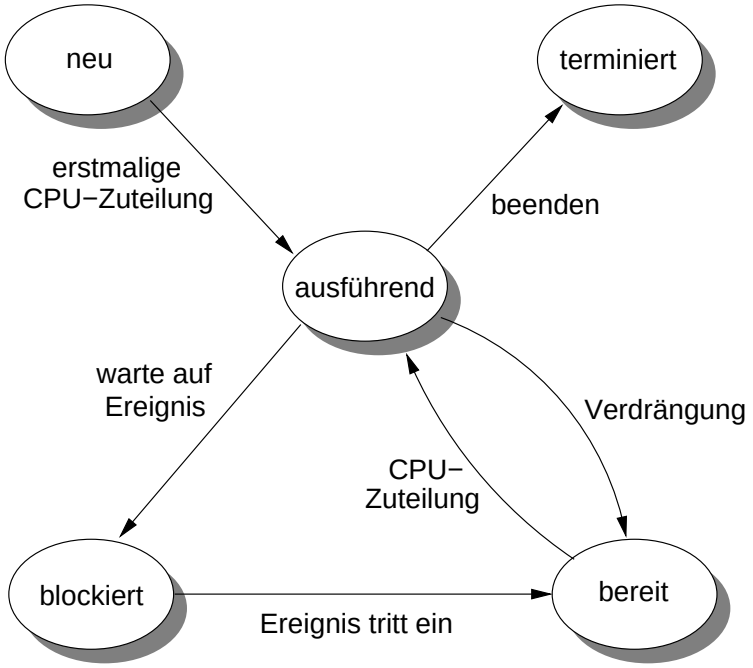


Abbildung 3.3: Zustandsmodell mit neuen und terminierten Prozessen

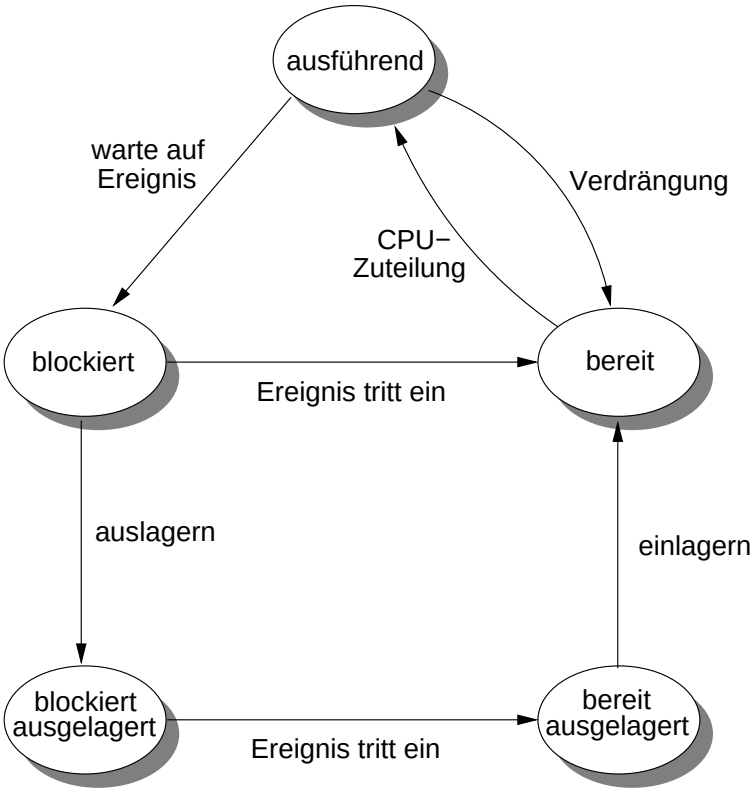


Abbildung 3.4: Zustandsmodell mit ausgelagerten Prozessen

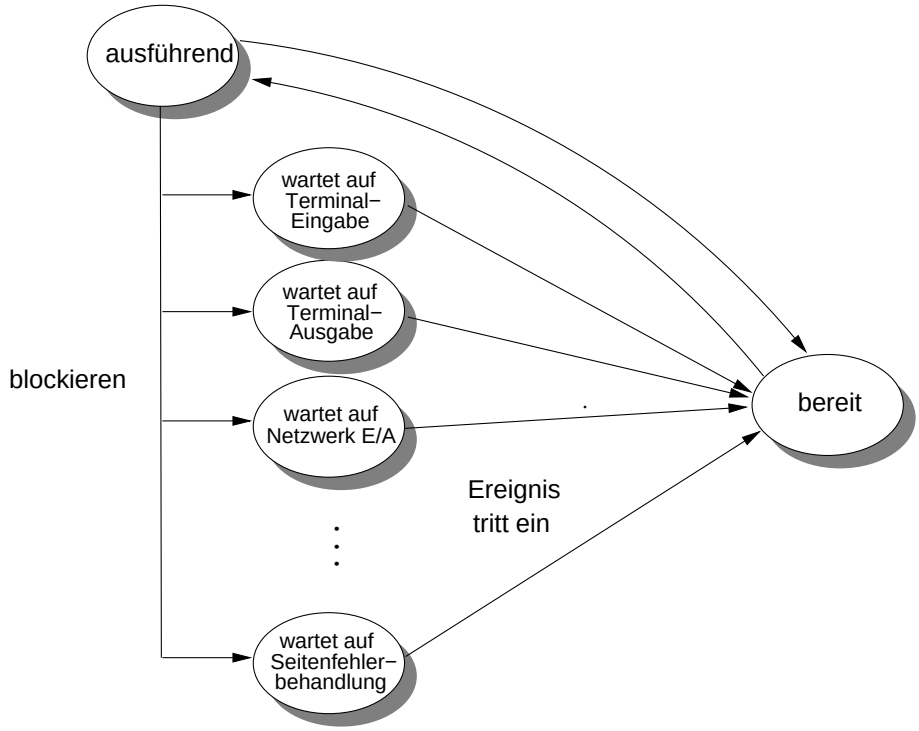


Abbildung 3.5: Zustandsmodell mit verschiedenen Wartezuständen

in den Hauptspeicher zurückzukopieren, solange sie blockiert sind.

Verschiedene Wartezustände

Betriebssysteme unterscheiden oft blockierte Prozesse nach der Art des erwarteten Ereignisses. Dies ermöglicht bei Eintritt eines Ereignisses eine effizientere Suche nach dem (den) aufzuweckenden Prozess(en).

Abbildung **??** zeigt ein Beispiel für ein solches Modell.

3.3 Synchrone und asynchrone Ereignisse und Unterbrechungen

Ein ausführender Prozess führt normalerweise die Instruktionen des zugrundeliegenden Anwendungsprogramms aus. Von Hardware oder Software ausgelöste **Ereignisse** unterschiedlicher Art führen regelmäßig zur **Unterbrechung** der normalen Ausführung. Die Prozessorkontrolle wird an das Betriebssystem übergeben. Im Betriebssystem reagiert eine für die Ereignisklasse spezifische **Ereignisbehandlungsfunktion** auf das Ereignis, bevor die reguläre Programmausführung wieder fortgesetzt werden kann.

Ein Ereignis heißt **synchron**, wenn es durch eine bestimmte Operation des ausführenden Programms unmittelbar ausgelöst wird, z.B. führt ein Division durch Null gewissermaßen zu einem „Ausnahmezustand“ des Prozessors, der im Betriebssystem behandelt wird. Synchrone Ereignisse sind meist reproduzierbar, bei wiederholter Ausführung des Programms tritt an der gleichen Stelle das Ereignis immer wieder auf.

Ein Ereignis heißt **asynchron**, wenn der Zeitpunkt des Auftretens keinen Bezug zu den vom ausführenden Prozess gerade bearbeiteten Befehlen hat. Aus Sicht des ausführenden Prozesses sind asynchrone Ereignisse unvorhersehbar.

3.3.1 Ereignisklassen

Wir unterscheiden folgende Ereignisklassen:

- **Hardware-Unterbrechungen** (**Interrupts**)

  Ein Interrupt wird asynchron von der Hardware erzeugt, oft im Zusammenhang mit Ein-/Ausgabeoperationen. (Das Konzept wird in **??**, S. **??** genauer erläutert.)

- **Hardwarefehler** (asynchron)

  Durch Spannungsschwankungen oder defekte Hardware können jederzeit Ausnahmezustände auftreten.

- **Systemaufrufe** (synchron)

  Systemaufrufe wurden schon behandelt. Die zur Implementierung verwendeten Traps gleichen in vieler Hinsicht den anderen aufgeführten Ereignissen. Aus Sicht des Programmierers sind Traps natürlich keine „unerwarteten“ Ereignisse, die Kontrolle wird gewollt an das Betriebssystem übergeben.

- **Programmfehler-Ereignisse** (synchron)

  Bestimmte Programmfehler verursachen Prozessor-Ausnahmezustände, z.B. Fließkommaüberlauf, Division durch Null oder unerlaubte Hauptspeicherzugriffe (oft durch Dereferenzierung fehlerhaft initialisierter Pointer-Variablen).

  Im Gegensatz zu Systemaufruf-Traps ist die Unterbrechung aus Sicht des Programmierers nicht beabsichtigt. Oft besteht die Ereignisbehandlung darin, den auslösenden Prozess mit einer Fehlermeldung zu terminieren.

  Viele Systeme erlauben es dem Anwendungsprogramm, die Ereignisbehandlung für solche Fehlerzustände selbst zu bestimmen. Dazu muss das Anwendungsprogramm eine Fehlerbehandlungsfunktion beim Systemkern registrieren. Die Behandlung innerhalb des Betriebssystems besteht dann in der Aktivierung der registrierten Anwenderfunktion[2].

[2] Der Mechanismus könnte folgendermaßen aussehen: Das Betriebssystem gibt zwar die Kontrolle an das Anwendungsprogramm zurück, aber nicht an die Unterbrechungsstelle. Der Programmzähler wird vom Betriebssystem so „umgebogen“, dass zunächst die Fehlerbehandlungsfunktion ausgeführt wird. Diese übergibt die Kontrolle wieder zurück an den Systemkern, der den Programmzähler dann ggf. auf den alten Wert zurücksetzt und erneut in den Anwendungsmodus wechselt. Man spricht dann von „Trampolin-Code“.

- **Seitenfehler**

  Im Kapitel **??**, S. **??** wird erläutert, wie der von einem Prozess benutzte Hauptspeicherbereich in Blöcke fester Größe, *Speicherseiten*, eingeteilt werden kann. Diese werden bei Hauptspeicherengpässen temporär auf den Swap-Bereich einer Platte ausgelagert.

  Greift nun ein Prozess auf eine seiner ausgelagerten Seiten zu, muss er unterbrochen werden, bis diese wieder in den Hauptspeicher kopiert ist. Im Unterschied zu den oben diskutierten Programmfehlern muss der fehlgeschlagene Speicherzugriff dann wiederholt werden. Für den Prozess bleibt die Ereignisbehandlung unsichtbar.

  Obwohl die Programmausführung die Ausnahmesituation unmittelbar herbeiführt, ist diese jedoch nicht programmiert und nicht reproduzierbar wie etwa Speicherzugriffsschutz-Verletzungen. Seitenfehler haben also eher asynchronen als synchronen Charakter.

- **Software-Interrupts**

  Im Gegensatz zu Hardware-Interrupts werden Software-Interrupts „programmiert“. (vgl. **??**, S. **??**)

3.3.2 Ereignisbehandlung und Prozess-Zustandsübergänge

Allen Ereignisklassen gemeinsam ist, dass die Kontrolle über den Prozessor an das Betriebssystem übergeben wird, denn schliesslich sind die von der Hardware aktivierten Ereignis-Behandlungsroutinen Bestandteile des Betriebssystems.

Dadurch erhält das Betriebssystem die Möglichkeit, im Rahmen der Ereignisbehandlung die Prozesszustände zu ändern und Kontextwechsel durchzuführen.

Betrachten wir typische Beispiele:

- Ein Prozess versucht einen unerlaubten Hauptspeicher-Zugriff, der die Hardware in einen Ausnahmezustand versetzt. Die Behandlungsroutine kann z.B. den Prozess terminieren und einem anderen Prozess den Prozessor zuteilen.

- Ein Netzwerk-Controller signalisiert durch einen Interrupt den Empfang von Daten aus dem Netzwerk. Die Behandlungsroutine wird zunächst die Daten aus dem Controller in den Hauptspeicher kopieren. Dann wird ggf. ein blockierter Prozess, der auf diese Daten gewartet hat, „aufgeweckt“, d.h. in den Zustand „bereit“ versetzt.

  Gegebenenfalls wird dem aufgeweckten Prozess sogar der Prozessor zugeteilt, weil er eine höhere Priorität hat als der gerade ausführende Prozess.

- Jeder Rechner hat einen Taktgeber („clock“), der in regelmäßigen Intervallen Interrupts erzeugt. Die Behandlungsroutine wird die Systemzeit aktualisieren und prüfen, ob der ausführende Prozess sein Quantum verbraucht hat. In diesem Fall kann eine Verdrängung veranlasst werden. (Mehr dazu in **??**, S. **??**)

3.4 Operationen für Prozesse

- **Prozess erzeugen**

  Bei der Erzeugung wird entweder ein neues Programm in den Hauptspeicher geladen oder der erzeugende Prozess kopiert (vgl. UNIX-fork). Das Betriebssystem muss eine eindeutige Prozessidentifikation vergeben und den Prozesstabelleneintrag erzeugen und initialisieren.

  Die vom Programm unabhängigen Prozessattribute werden dabei entweder explizit angegeben oder vom Erzeuger-Prozess übernommen, also „vererbt“.

  Der neue Prozess ist entweder ein Subprozess des Erzeugerprozesses oder ein unabhängiger Prozess („detached“).

- **Prozess beenden**

  Dabei wird der Hauptspeicher und ggf. andere Ressourcen, die der Prozess belegte, freigegeben, die offenen Dateien geschlossen und der Prozessdeskriptor aus der Prozesstabelle entfernt. Gegebenenfalls wird der Elternprozess von der Terminierung verständigt.

- **Priorität ändern**

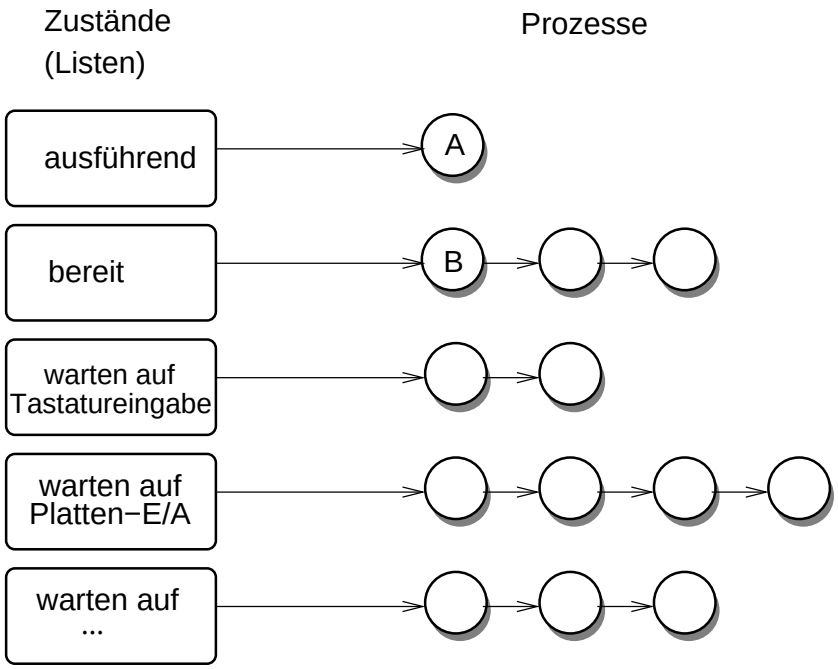
  Die Priorität kann durch explizite Systemoperationen beinflusst werden. Daneben werden Prioritäten auch dynamisch an die Interaktionshäufigkeit des Prozesses angepasst (vgl. **??**, S. **??**).

- **Zustand ändern:** verdrängen, ausführen, blockieren, aufwecken

  Die Zustandsänderungen werden, wie oben erläutert, im Rahmen von Systemaufrufen oder bei Unterbrechungen vom Betriebssystem durchgeführt.

  Eine mögliche Implementierung der Prozesszustände: Das Betriebssystem verkettet die Prozessdeskriptoren aller Prozesse, die sich in einem Zustand Z befinden zu einer Liste. Jedem möglichen Prozesszustand entspricht also eine Liste. Ein Zustandsübergang führt dazu, dass der betroffene Prozessdeskriptor in eine andere Liste eingefügt wird.

  Ein Beispiel zeigt Abbildung **??**: Der ausführende Prozess liest Daten von der Tastatur. Er wird blockiert und ein anderer Prozess bekommt den Prozessor.



ausführender Prozess liest von Tastatur
erster bereiter Prozess erhält Prozessor

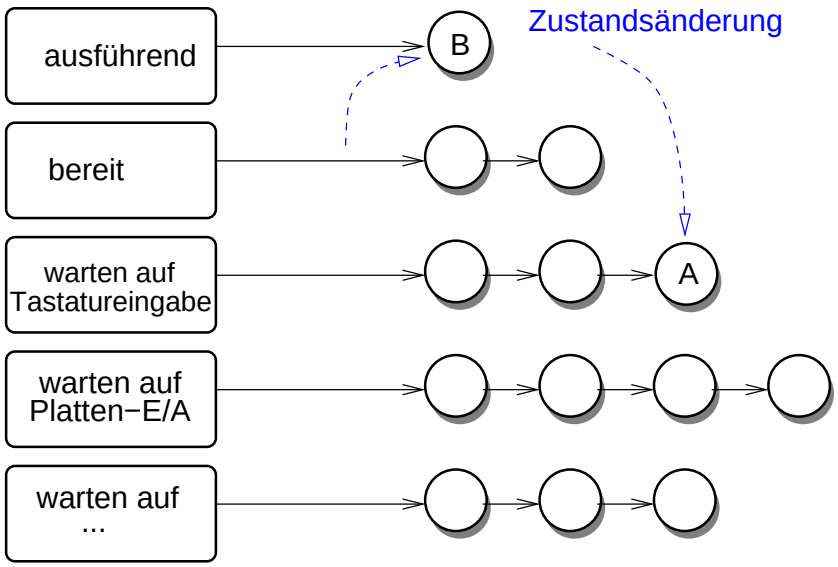


Abbildung 3.6: Zustandsverwaltung mit Listen

3.5 Kontextwechsel

Das Betriebssystem teilt den Prozessor nacheinander verschiedenen Prozessen zu. Nehmen wir z.B. an, der ausführende Prozesses P_1 blockiert, weil er auf Eingabedaten vom Netzwerk wartet. Um unnötigen Leerlauf im System zu vermeiden, wird der Prozessor während dieser Wartezeit zur Ausführung eines anderen Prozesses P_2 verwendet. Wenn P_1 später weitergeführt werden soll, muss sein Ausführungszustand zum Unterbrechungszeitpunkt bekannt sein. Diesen Zustand bezeichnet man auch als *Prozesskontext*, den Wechsel von P_1 nach P_2 als *Kontextwechsel*.

Für die betroffenen Prozesse ist der Kontextwechsel transparent. Ein normales Anwendungsprogramm enthält keinerlei Programmcode zur geordneten Übergabe des Prozessors an einen anderen Prozess, sondern definiert i.d.R. eine unterbrechungsfrei auszuführende Folge von Maschinenbefehlen.

Es ist daher Sache des Betriebssystems, den Kontext des unterbrochenen Prozesses zum Unterbrechungszeitpunkt abzuspeichern und bei der späteren Weiterführung so wiederherzustellen, dass aus Programmsicht die Unterbrechung nicht registriert wird.

Was macht nun den Zustand eines Prozesses aus? Aus Programmierersicht sind bei einer Momentaufnahme zur Ausführungszeit folgende Fragen relevant:

- An welcher Stelle im Programmcode befindet sich die Ausführung?
- Wie ist die aktuelle Unterprogrammaufruffolge?
- Welche Werte haben die lokalen und globalen Variablen?

Die aktuellen Unterprogrammaktivierungen, sowie die Werte von Parametern und lokalen Hilfsvariablen spiegeln sich im Laufzeitstack des Programms wieder, also in einem zum Prozess gehörenden Hauptspeicherbereich. Auch die anderen Daten stehen im Programmspeicher des Prozesses. Die zum Prozess gehörigen Hauptspeicherbereiche (Programmspeicher mit Code und Daten, sowie Prozessdeskriptor) werden während der Unterbrechungszeit durch andere Prozesse nicht überschrieben.

Ein kleiner, aber entscheidender Teil des Prozesszustands ist allerdings nicht im Hauptspeicher, sondern im Prozessor selbst zu finden: die prozessorinternen Speicherplätze, nämlich Mehrzweckregister, Programmzähler und Statusregister. Der Programmzähler enthält die Adresse des nächsten auszuführenden Maschinenbefehls. In den Mehrzweckregistern stehen einige vom Programm benötigte Daten und das Statusregister enthält maschinenspezifische Flags.

Das Betriebssystem muss beim Kontextwechsel all diese Register retten, d.h. in den Hauptspeicher kopieren. Platz dafür ist im Prozessdeskriptor vorgesehen. Der zur Ausführung ausgewählte Prozess bekommt den Prozessor im restaurierten Zustand, d.h. die vorher geretteten Registerwerte werden aus dem Prozessdeskriptor wieder in den Prozessor geladen.

Neben der CPU gibt es ggf. (maschinenabhängig) weitere Hardwarebausteine mit internen Speicherplätzen, die beim Kontextwechsel genauso behandelt werden müssen, z.B. ein separater Gleitkommaprozessor in einem PC oder die MMU.

Bei manchen Rechnern müssen die Register mit separaten Maschinenbefehlen einzeln gerettet bzw. restauriert werden, bei anderen reicht ein einziger Maschinenbefehl für die Übertragung aller Register.

Der Maschinencode für den Kontextwechsel ist Bestandteil des Betriebssystems und kann beispielsweise im Rahmen der Bearbeitung von Systemaufrufen aktiviert werden. Im obigen Beispiel heißt das: P_1 führt eine Leseoperation aus. Durch den Systemaufruf wird die Kontrolle an das Betriebssystem übergeben. Der aktivierte Systemcode wird nach Initiierung der Leseoperation schließlich auch den Kontextwechsel durchführen:

- Retten des Kontextes von P_1
- Restaurieren des Kontextes von P_2
- Umschalten des Prozessors in den nichtprivilegierten Modus
- Übergabe der Kontrolle an P_2

3.6 Threads

3.6.1 Nebenläufigkeit in Programmen

In vielen Situationen ist es sinnvoll, innerhalb eines Programms parallel ausführbare Bearbeitungsschritte zu spezifizieren. Einerseits ergeben sich dadurch manchmal Effizienzvorteile, andererseits kann eine derartige Zerlegung softwaretechnisch geboten sein.

Beispiel 1: Server im Netzwerk, die gleichzeitig mit mehreren Client kommunizieren (*Parallelserver*-Konzept, „*concurrent server*“).

Beispiel 2: Ausnutzung der Prozessoren eines Multiprozessorsystems durch eine rechenintensive Anwendung.

Definition

Ein **Thread** („thread of control“, „Kontrollfaden“) ist eine sequentiell abzuarbeitende Befehlsfolge innerhalb eines Programms.

Definition

Man spricht von „**Multithreading**“, wenn innerhalb eines Programms zwei oder mehr nebenläufige Befehlssequenzen spezifiziert werden. Wir nennen ein solches Programm nachfolgend „nebenläufig“.

Die Programmierung mit Threads erfolgt

- entweder durch entsprechende Konzepte der Programmiersprache (z.B. Java, ADA, Chill) oder

- durch Verwendung einer Thread-Bibliothek (z.B. POSIX-Thread-Bibliothek für C/C++)

3.6.2 Implementierung von Threads

Betrachten wir drei Programme, davon zwei nebenläufige:

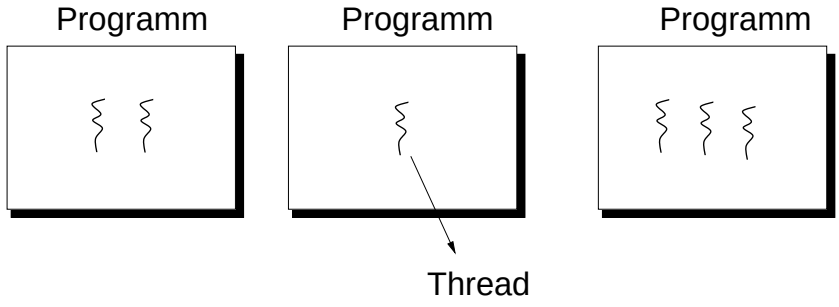


Abbildung 3.7: Threads – Nebenläufigkeit innerhalb eines Programms

Bei gleichzeitiger Ausführung der drei Programme konkurrieren 6 Threads um den Prozessor. Mehrere Konzepte der Prozessorzuteilung sind denkbar.

1. Jeder Thread eines Programms wird auf einen eigenen Prozess abgebildet. Mit dem wohlbekannten Multitasking-Konzept wird die Nebenläufigkeit der Threads auf Betriebssystemebene realisiert.

2. Eine Thread-Bibliothek auf Benutzerebene, die alle notwendigen Funktionen zur Thread-Verwaltung enthält, insbesondere einen Scheduler.

3. Ein spezifisches Betriebssystemkonzept zur Unterstützung von Threads: **leichtgewichtige Prozesse** (LGP)

3.6.3 Thread-Implementierung durch Prozesse

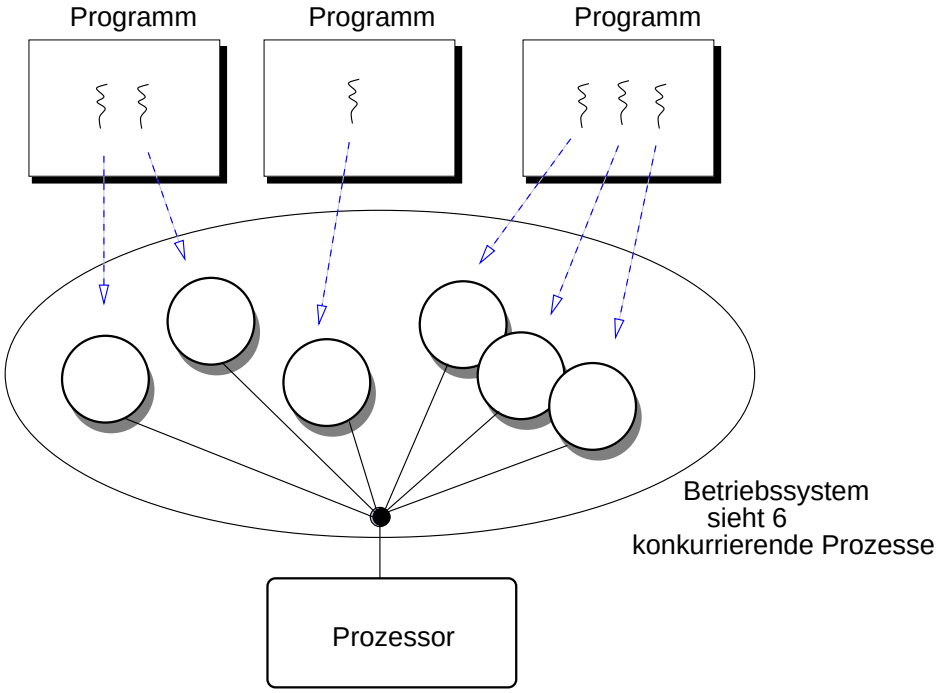


Abbildung 3.8: Thread-Implementierung durch Prozesse

Das herkömmliche Prozesskonzept ist für die nebenläufige Bearbeitung verschiedener Programme entwickelt worden. Damit verschiedene Benutzer eines Mehrbenutzersystems unabhängig voneinander arbeiten können, sind Prozesse durch separate Adressräume, separate Dateideskriptoren und strikte Zugriffschutzkontrollen gegeneinander abgeschottet. Daraus folgt:

- Das Erzeugen eines Prozesses ist aufwändig und benötigt ggf. viel Speicherplatz
- Der Kontextwechsel ist aufwändig
- Interprozesskommunikation erfolgt über das Betriebssystem und ist dadurch vergleichsweise langsam

Das Prozesskonzept ist in diesem Sinne ein **schwergewichtiges** Konzept.

Für zwei Threads desselben Programms ist eine gegenseitige Abschottung weder notwendig noch sinnvoll. Gemeinsam benutzte Datenstrukturen und Dateien dienen einer effizienten Kooperation. Ein Thread benötigt im wesentlichen einen eigenen Laufzeitstack, damit er unabhängig von den anderen Threads Funktionsaufrufe durchführen kann.

Threads durch Prozesse zu implementieren, heißt „mit Kanonen auf Spatzen zu schiessen“. Es ist ineffizient im Hinblick auf Hauptspeichernutzung (eigene Adressräume) und Geschwindigkeit (Erzeugen, Kontextwechsel, Kommunikation).

3.6.4 Thread-Bibliotheken auf Benutzerebene („Anwender-Threads“)

Das Betriebssystem muss die parallelen Aktivitäten eines Programm nicht notwendigerweise sehen. Das Programm kann die Verwaltung seiner Threads auch ohne Unterstützung des Betriebssystems durchführen. Die dazu benötigten Funktionen lassen sich in Form einer Bibliothek implementieren.

Bestandteile einer Thread-Bibliothek:

- Operationen zum Erzeugen und Zerstören von Threads
- Scheduling-Mechanismus (Blockieren, Aktivieren, Prioritäten)
- Synchronisationsmechanismen
- Kommunikationsmechanismen

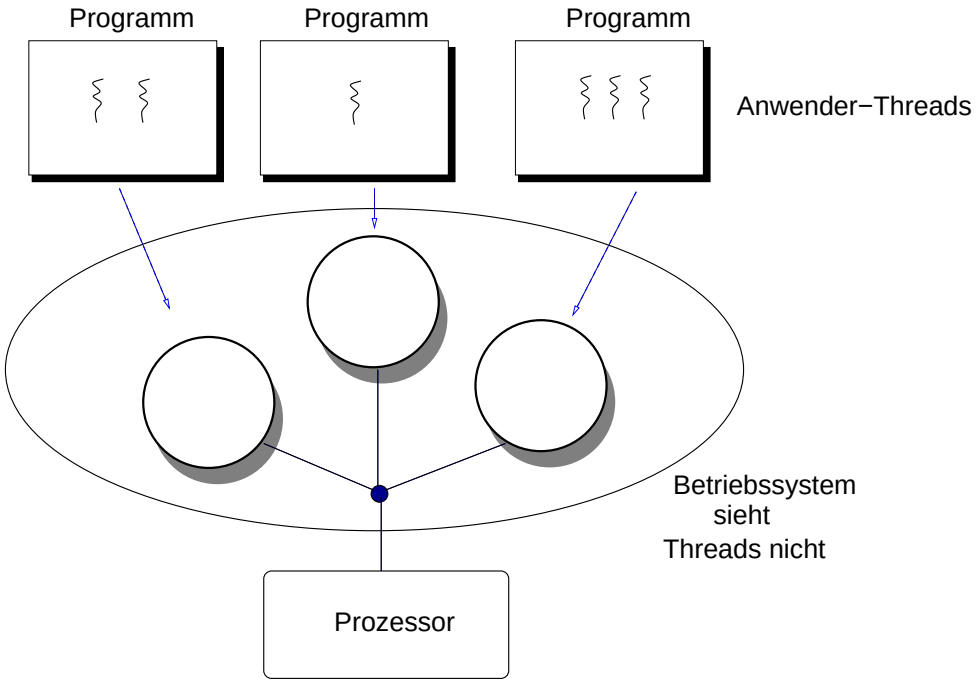


Abbildung 3.9: Anwender-Threads

Diese besonders „leichtgewichtige“ Lösung hat Vorzüge:

- Die wesentlichen Thread-Operationen (erzeugen, vernichten, Kontextwechsel, Kommunikation, Synchronisation) erfolgen ohne Zutun des Betriebssystems. Damit sind sie **effizienter** implementierbar, denn schliesslich ist die Nutzung des Systemkerns immer mit einem gewissen Aufwand verbunden.

- Anwender-Threads belegen keine Betriebssystemressourcen (wie z.B. Prozessdeskriptoren). Die Anzahl der Anwender-Threads ist damit kaum nach oben beschränkt.

Dem stehen aber auch Nachteile gegenüber:

Der Betriebssystemkern sieht nur einen einzigen Prozess, von der Existenz ausführungsbereiter Threads weiß das BS nichts:

- Wenn ein Thread im Betriebssystem blockiert (E/A, Seitenfehler), ist der Prozess als Ganzes blockiert inklusive der ausführungsbereiten Threads (von denen das Betriebssystem aber nichts weiss),

- Echte Parallelausführung ist in Multiprozessorsystemen nicht möglich, da die Prozessorzuteilung beim Betriebssystem liegt.

3.6.5 Leichtgewichtige Prozesse

Moderne Betriebssysteme tragen dem steigenden Bedarf nach Thread-Unterstützung durch die Einführung eines neuen Objekttyps Rechnung, den wir als „leichtgewichtigen Prozess“ (LGP – engl. „*lightweight process*“) bezeichnen wollen[3].

Dahinter steckt die Trennung von

- Programm-bezogener Information, z.B.
  - Zugriffsrechte
  - Adressraum
  - Dateideskriptoren

  und

- Thread-spezifischen Daten, z.B.
  - Laufzeitstack
  - Zustand
  - Priorität
  - Maschinenstatus (Programmzähler, Registerinhalte usw.)

[3] In der Literatur oft auch als „*Kern-Thread*“ (engl. „*kernel thread*“) oder einfach nur als „*Thread*“ bezeichnet

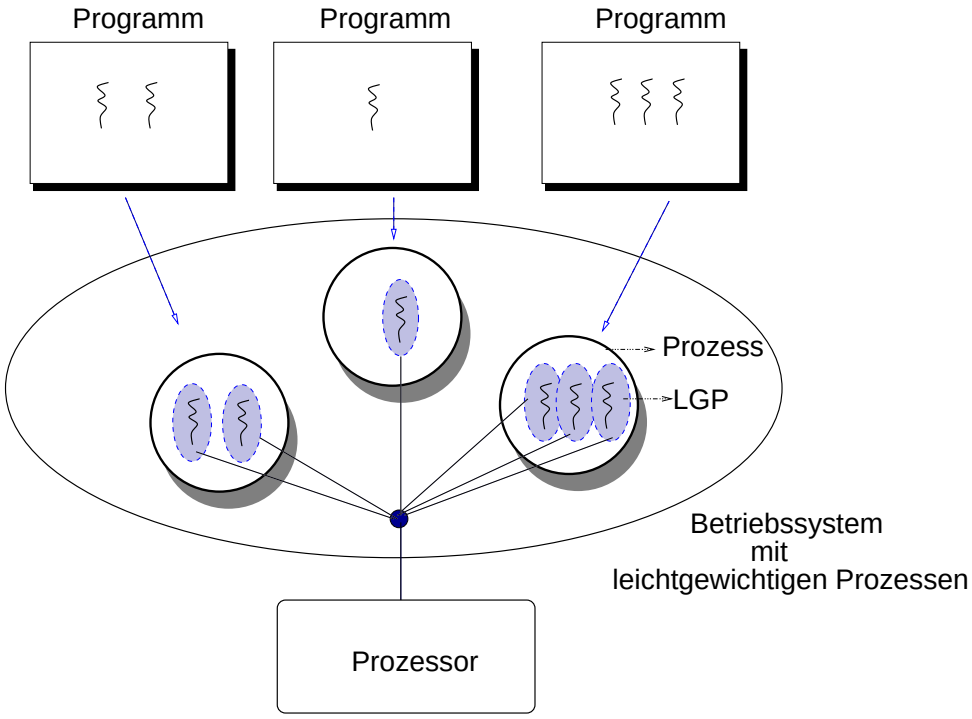


Abbildung 3.10: Leichtgewichtige Prozesse

In einem solchen Betriebssystem (z.B. Sun Solaris oder Windows NT) ist ein Prozess kein „aktives“ Objekt mehr, sondern eine eher statische Umgebung, die den Rahmen für die Ausführung der in ihr ablaufenden Threads bildet.

Nicht mehr Prozesse sind es, die um den Prozessor konkurrieren und deren Zustände verwaltet werden, sondern LGPs. Das Betriebssystem benötigt neben der Prozesstabelle pro Prozess eine LGP-Liste mit der benötigten LGP-Verwaltungsinformation.

Kapitel 4

Synchronisation nebenläufiger Prozesse

4.1 Nichtdeterminismus durch Nebenläufigkeit

Sobald nebenläufige Prozesse (oder Threads) miteinander **kooperieren** oder um exklusiven Zugriff auf Daten oder Geräte **konkurrieren**, benötigt man Synchronisationsmechanismen. Dabei spielt es zunächst keine Rolle, ob die gegenseitige Abstimmung durch Zugriff auf gemeinsam benutzte Hauptspeicherbereiche, Dateien, Geräte oder durch Nachrichtenübertragung im Netzwerk notwendig wird.

Die Problematik gemeinsamen Datenzugriffs verdeutlicht ein Beispiel:

Bei zwei nebenläufigen Prozessen lässt sich der relative Bearbeitungsfortschritt nicht vorhersagen. Betrachten wir den nebenläufigen Zugriff auf gemeinsame Variablen:

concurrency-example.cc

```
#include <ostream.h>
#include "concurrent.h"

int a=1, b=2;

f(){ a=a+b; a=2*a; }
g(){ a=5; }

main(){
  concurrent(f,g); // nebenlaeufiger Aufruf
  cout << a;
}
```

Nehmen wir an, die Aufrufe von f und g erfolgen nebenläufig. Welcher Wert von *a* wird

am Ende ausgegeben?

Mehrere zeitliche Abfolgen für die Ausführung der Wertzuweisungen sind möglich:

Abfolge	1.	2.	3.
Schritt 1	f: `a=a+b`	f: `a=a+b`	g: `a=5`
Schritt 2	f: `a=2*a`	g: `a=5`	f: `a=a+b`
Schritt 3	g: `a=5`	f: `a=2*a`	f: `a=2*a`
Resultat	5	10	14

Wenn man bedenkt, dass die Ausführung einer Wertzuweisung keine atomare Operation ist, sondern in eine überall unterbrechbare Sequenz von Maschinenbefehlen übersetzt wird, ergeben sich noch weitere Möglichkeiten:

Nehmen wir an, die in *f* ausgeführte Wertzuweisung

```
a=2*a;
```

wird in 3 Maschinenbefehle zerlegt:

```
LOAD        a, Register1    Wert von a in ein Register laden
SHIFTLEFT   Register1, 1    Registerinhalt verdoppeln
STORE       Register1, a    Registerinhalt zurückkopieren
```

Nehmen wir außerdem an, die Bearbeitung des Aufrufs von *f* wird nach der `LOAD`-Anweisung unterbrochen. Jetzt wird die in *g* enthaltene Wertzuweisung `a=5` ausgeführt. Dann wird die Bearbeitung von *f* fortgesetzt. Der im Register gespeichert Wert ist 3, nach `SHIFTLEFT` und `STORE` hat *a* den Wert 6.

Das im Beispiel sichtbare nichtdeterministische Verhalten ist sicher in den wenigsten Fällen erwünscht.

Anmerkung: Eine mögliche Implementierung der oben benutzten Funktion *concurrent* auf der Basis POSIX-konformer Threads wäre:

concurrent.cc

```
#include <pthread.h>

void concurrent(void (f()), void (g())){
  pthread_t f_thread, g_thread;
  pthread_create(&f_thread, NULL,
                 (void* (*)(void *)) f, NULL);
  pthread_create(&g_thread, NULL,
                 (void* (*)(void *)) g, NULL);
  pthread_join(f_thread, NULL);
  pthread_join(g_thread, NULL);
}
```

4.2 Wettbewerbsbedingungen und Warteoperationen

Nachfolgend sind die Begriffe *Thread* und *Prozess* weitestgehend austauschbar.

4.2.1 Wie programmiert man „Warten“?

Es gibt zwei prinzipiell unterschiedliche Möglichkeiten, das Warten auf ein bestimmtes Ereignis zu programmieren:

- **aktives Warten** („busy waiting“):

  Das Programm prüft innerhalb einer Schleife ständig, ob das erwartete Ereignis schon eingetreten ist.

  ```
  while ( ! Ereigniss eingetreten )
            ; // nichts tun !
  ```

  Das Betriebssystem wird dazu nicht benötigt. Ein Thread benötigt zum aktiven Warten den Prozessor. In einem Multitasking-System ist dies oft unerwünscht, dort könnte der Prozessor während der Wartezeit sinnvoller durch einen anderen Thread genutzt werden.[1]

[1] Manche Computerspiele werden durch aktive Warteschleifen verlangsamt und an die Reaktionszeit des Spielers angepasst. Ältere Spiele dieser Art lassen sich auf modernen PCs oft nicht mehr spielen, weil sie viel zu schnell ablaufen.

- „passives“ Warten durch **Blockieren**

  Der Thread benutzt einen Betriebssystemaufruf zum Warten. Er beauftragt das Betriebssystem, seine Ausführung solange zu suspendieren[2], bis das erwartete Ereignis eingetreten ist. Betriebssysteme beinhalten unterschiedliche Blockiermechanismen für diesen Zweck. Die wichtigsten werden unten erläutert.

  Das Betriebssystem muss beim Auftreten eines Ereignisses jeweils prüfen, ob ein oder mehrere Prozesse aufgeweckt werden müssen.

4.2.2 Wettbewerbsbedingungs-Beispiel: Mailbox

Wir betrachten ein bekanntes Synchronisationsproblem, das als „**bounded buffer**“-Problem oder „**Produzent-Konsument**“-Problem bezeichnet wird.

Der „Puffer“ ist ein beschränkt großer Bereich zum Hinterlegen von Nachrichten, mit dessen Hilfe Threads Daten austauschen können. Ein Datenproduzent schreibt Daten in diesen Puffer hinein, ein Konsument holt sie wieder heraus. Wenn der Puffer voll ist, muss der Produzent ggf. warten. Umgekehrt wartet der Konsument ggf. bei leerem Puffer auf neue Nachrichten. (Auch eine BSD-UNIX-Pipe ist ein solcher Puffer.)

Wir studieren das Problem anhand einer Beispiel-Implementierung. Ein beschränkter Nachrichtenspeicher wird als „Mailbox“ bezeichnet.

- Eine neue Nachricht wird durch die Methode *neu* in der Mailbox hinterlegt. Der Aufrufer übernimmt die Rolle des **Produzenten**.
- Die Methode *lies* liefert die nächste noch nicht gelesene Nachricht als Resultat. Die Nachricht wird aus der Mailbox entfernt und damit wieder Platz für neue Nachricht geschaffen. Der Aufrufer übernimmt die Rolle des **Konsumenten**

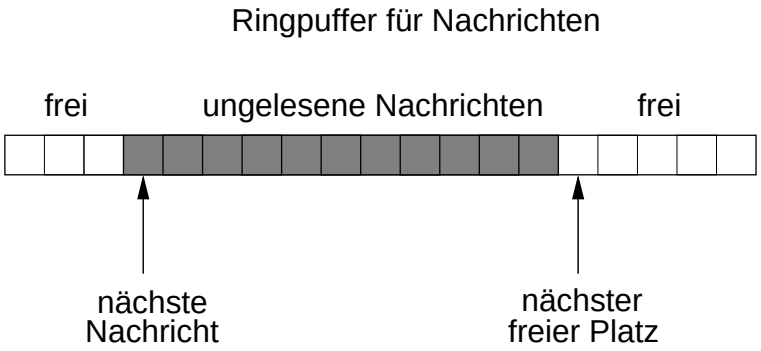


Abbildung 4.1: Ringpuffer

Der Ringpuffer wird durch ein Feld und jeweils einen Verweis auf den nächsten freien Platz und auf die nächste ungelesene Nachricht realisiert. Beide Verweise werden bei

[2] d.h. ihn bei der Prozessorvergabe nicht mehr zu berücksichtigen

Erreichen des Pufferendes wieder auf den Anfang zurückgesetzt, so dass die Feldelemente reihum wiederverwendet werden können. Ausserdem wird ein Zähler für die Anzahl der noch nicht gelesenen Nachrichten geführt.

Wir betrachten zunächst eine unzureichend synchronisierte Version:

mailbox.h

```
#define MAX 100

typedef int nachricht;

class mailbox {
  nachricht puffer[MAX];
  int ifrei, //Index freier Platz
      ilies, //Index naechste ungelesene Nachricht
      anzahl;//Anzahl ungelesener Nachrichtrn

public:
  mailbox(void);
  void neu(nachricht n);
  nachricht lies(void);
};
```

mailbox.cc

```
#include "mailbox.h"

mailbox::mailbox(){
  ifrei=ilies=anzahl=0;
}

void mailbox::neu(nachricht n){
  // falls Puffer voll, warten
  while (anzahl==MAX)
     ; // warte

  puffer[ifrei]=n;
  ifrei= (ifrei + 1) % MAX;
  anzahl++;
}

nachricht mailbox::lies() {
  nachricht n;

  // falls Puffer leer, warten
  while (anzahl==0)
     ; // warte

  n=puffer[ilies];
  ilies=(ilies+1) % MAX;
  anzahl--;
  return n;
}
```

Diskussion des Beispiels:

Im Beispiel wird sowohl beim Einfügen neuer Nachrichten, als auch beim Lesen der Nachrichten aktives Warten benutzt. Wenn ein Produzent und ein Konsument nebenläufig auf eine Mailbox zugreifen, sind damit grundlegende Anforderungen erfüllt:

- Das Senden und Empfangen von Nachrichten ist zu einem gewissen Grad zeitlich entkoppelt. Solange Nachrichtenproduktion und -konsum ungefähr mit gleicher Frequenz erfolgen, gibt es keine Probleme.

- Ist die Nachrichtenproduktion über längere Zeit zu schnell (Puffer voll), wird sie „gebremst“, also mit der Konsumentengeschwindigkeit synchronisiert.

  Das gleiche gilt auch umgekehrt.

Die Implementierung ist dennoch unzulänglich: Produzent und Konsument greifen auf gemeinsame Daten zu und modifizieren diese. Die Mailboxzugriffe *lies* und *neu* sind keine atomaren Operationen, sondern unterbrechbare Befehlsfolgen. Dabei kann es zu nichtdeterministischem Programmverhalten („race conditions“) kommen.

Wir zeigen dies zunächst in einer Umgebung mit zwei konkurrierenden Lesezugriffen, d.h. zwei nebenläufige *neu*-Aufrufe sind aktiv. Nehmen wir an, in der Mailbox ist nur eine Nachricht.

Folgende Wettbewerbsbedingung kann auftreten:

Thread: Leser 1	Thread: Leser 2	Kommentar
		Anfangswerte: anzahl=1, ifrei=11,ilies=10
`while (anzahl==0)`		
`;`		
`n=puffer[ilies];`		
`ilies=(ilies+1)%MAX;`		ilies=11
		Leser 2 erhält jetzt den Prozessor
	`while (anzahl==0)`	
	`;`	
	`n=puffer[ilies];`	**hier steht keine Nachricht !**
	`ilies=(ilies+1)%MAX;`	
	`anzahl--;`	anzahl=0
	`return n;`	
`anzahl--;`		**anzahl=-1 !**
`return n;`		

Bei dieser Abfolge entstehen zwei Fehler:

- Leser 2 greift auf einen Mailboxeintrag zu, der keine gültige Nachricht enthält
- Die globale Variable *Anzahl* wird negativ.

Andere unschöne Effekte sind möglich. (Überlegen Sie selbst!).

Auch wenn nur ein Produzent und ein Konsument im Spiel sind, gibt es Wettbewerbsbedingungen. Allerdings sind diese auf der Quelltextebene schwer zu erkennen. Der konkurrierenden Zugriff auf die Variable *anzahl* bereitet nämlich Probleme: Die C-Anweisungen `anzahl--` in *lies* und `anzahl++` in *neu* sind nämlichnicht atomar, sondern werden auf der Maschinenebene typischerweise durch unterbrechbare Befehlssequenzen folgenden Typs implementiert:

`LOAD R,ANZAHL`	Wert von *anzahl* aus Hauptspeicher in Register laden
`INCR R`	Registerinhalt inkrementieren (bei `anzahl--` dekrementieren)
`STORE R,ANZAHL`	aktualisierten Registerinhalt zurückspeichern

Betrachten wir folgendes Szenario:

- Anfangswert: anzahl=10
- Der *neu*-Aufruf lädt den Wert aus dem Hauptspeicher in ein Register und wird unterbrochen.
- Der *lies*-Aufruf dekrementiert den Wert von Anzahl (im Hauptspeicher) auf 9.
- Der *neu*-Aufruf wird fortgeführt, im Register steht noch die 10.
- Der Registerinhalt wird auf 11 inkrementiert
- Der Hauptspeicherplatz von *anzahl* wird durch den Registerinhalt überschrieben. Die Dekrementierung durch *lies* geht dabei verloren!

4.3 Kritische Abschnitte und gegenseitiger Ausschluss

Definition

Ein **kritischer Abschnitt** ist ein Bereich des Programmquelltexts, in dem nebenläufig auf gemeinsame Daten zugegriffen wird.

Falls alle beteiligten Threads nur lesend auf die gemeinsamen Daten zugreifen, ist dies unproblematisch. Sobald nur einer der Threads die Daten manipuliert, ist die Gefahr von Wettbewerbsbedingungen gegeben.

In unserem Mailbox-Beispiel sind die Mailbox-Zugriffe *neu* und *lies* kritische Abschnitte. Die Probleme treten auf, wenn zwei Threads gleichzeitig die Mailbox-Variablen modifizieren, wenn sie sich also gleichzeitig in einem kritischen Abschnitt befinden.

Definition

Gegenseitiger Ausschluss (engl.: „mutual exclusion“) heißt, dass sich von allen um den Datenzugriff konkurrierenden Threads sich immer nur höchstens einer in seinem kritischen Abschnitt befindet.

Für unser Mailbox-Beispiel gilt es, gleichzeitigen Mailbox-Zugriff zu verhindern:

- gleichzeitige Ausführung zweier *lies*-Aufrufe

  (beachte: Bei *lies* handelt es sich ungeachtet des Funktionsnamens nicht um einen rein lesenden Zugriff, *lies* **modifiziert** die Mailbox!)

- gleichzeitige Ausführung zweier *neu*-Aufrufe

- gleichzeitige Ausführung eines *lies*- und eines *neu*-Aufrufs

Anmerkung: Im allgemeinen hat der Programmierer natürlich eine gewisse Entscheidungsfreiheit bezüglich der *Feinkörnigkeit* des gegenseitigen Ausschlusses:

- grobkörnig: Man verbietet jegliche gemeinsamen Mailbox-Zugriffe
- feinkörnig: Man erlaubt die gleichzeitige Manipulationen unterschiedlicher Mailbox-Fächer und schützt nur die Manipulation der Variable *anzahl* als kritischen Bereich.

  Da ein Mailbox-Zugriff aber eine einfache und sehr schnell erledigte Operation ist, erspart man sich am besten den erhöhten Aufwand.

Implementierung gegenseitigen Ausschlusses

Wir benötigen zwei Operationen, die dafür sorgen, dass gegenseitiger Ausschluss gewährleistet wird:

- *enter_region* - Eintritt in den kritischen Abschnitt

  Die Operation wird den Thread für eine gewisse Zeit blockieren, wenn ein anderer Thread sich gerade im kritischen Abschnitt befindet
- *leave_region* - Verlassen des kritischen Abschnitts

Die Implementierung muss allerdings nicht nur für gegenseitigen Ausschluss sorgen, sondern weitere Anforderungen erfüllen:

- Insbesondere sollen sich Prozesse nicht gegenseitig so blockieren, dass überhaupt kein Bearbeitungsfortschritt mehr stattfindet. Die Implementierung muss **verklemmungsfrei** sein (Verklemmungen, engl: „deadlocks“, werden in **??**, S. **??** behandelt.)
- Darüber hinaus erwarten die beteiligten Prozesse eine gewisse **Fairness:** Kein Prozess soll ewig warten müssen, bis er seinen kritischen Abschnitt betreten darf.

4.4 Gegenseitiger Ausschluss auf Anwenderebene

Es ist möglich, auf der Anwenderebene durch aktives Warten für gegenseitigen Ausschluss zu sorgen. Dazu betrachten wir die Synchronisation zweier nebenläufiger Threads, die sich beide gemäß folgendem Schema verhalten:

```
while (TRUE) {
    .
    .  unkritische Aktivität
    .
  enter_region
    .
    .  kritischer Abschnitt
    .
  leave_region
} // end while
```

Algorithmus 1

```
/* gemeinsam benutzte Variablen */
int naechster=1;

...

thread1() {
  while (TRUE) {
          .
          .  unkritische Aktivität
          .
       while (naechster != 1)
         ;    // warte
          .
          .  kritischer Abschnitt
          .
       naechster=2;
  }
}


thread2() {
  while (TRUE) {
        .
        .  unkritische Aktivität
        .
    while (naechster != 2)
      ;    // warte
        .
        .  kritischer Abschnitt
        .
    naechster=1;
  }
}

int main() {
    concurrent(thread1,thread2);
}
```

Der Algorithmus sorgt für gegenseitigen Ausschluss, indem er den beiden Threads strikt alternierend den Eintritt in ihren kritischen Abschnitt erlaubt. Er ist für praktische Anwendungen dennoch nicht brauchbar. Abgesehen davon, dass ein streng abwechselnder Zugriff auf gemeinsame Daten wohl kaum einer realen Anwendung gerecht wird, gibt es ein gravierendes Effizienzproblem:

Durch langes Verweilen im unkritischen Bereich wird der Thread, der als nächster „dran“ ist, den anderen Thread unnötig blockieren.

Algorithmus 2

```
/* gemeinsam benutzte Variablen */
bool kritisch1=FALSE;
bool kritisch2=FALSE;

...

thread1() {
  while (TRUE) {
        .
        .  unkritische Aktivität
        .
     while (kritisch2)
       ;      // warte
     kritisch1=TRUE;
        .
        .  kritischer Abschnitt
        .
     kritisch1=FALSE;
  }
}

thread2() {
  while (TRUE) {
        .
        .  unkritische Aktivität
        .
     while (kritisch1)
       ;      // warte
     kritisch2=TRUE;
        .
        .  kritischer Abschnitt
        .
     kritisch2=FALSE;
  }
}

int main() {
    concurrent(thread1,thread2);
}
```

Der Algorithmus sorgt *nicht* für gegenseitigen Ausschluss: Wenn Thread 1 bei Eintritt in den kritischen Bereich nach der Warteschleife, aber noch vor der Wertzuweisung `kritisch1=TRUE` unterbrochen wird, kann Tread 2 in der Unterbrechungszeit unbehelligt seine Warteschleife passieren.

Algorithmus 3

```
/* gemeinsam benutzte Variablen */
bool interesse1=FALSE;
bool interesse2=FALSE;

...

thread1() {
  while (TRUE) {
       .
       .  unkritische Aktivität
       .
    interesse1=TRUE;
    while (interesse2)
       ;     // warte
        .
        .  kritischer Abschnitt
        .
    interesse1=false;
  }

thread2() {
  while (TRUE) {
       .
       .  unkritische Aktivität
       .
    interesse2=TRUE;
    while (interesse1)
       ;     // warte
        .
        .  kritischer Abschnitt
        .
    interesse2=false;
  }
}

int main() {
    concurrent(thread1,thread2);
}
```

Gegenseitiger Ausschluss ist garantiert, aber: Der Algorithmus führt ggf. zu einer Verklemmung.

Wenn Thread 1 nach der Zuweisung `interesse1=TRUE` unterbrochen wird, noch bevor er `interesse2` erstmals prüft, kann Thread 2 während der Unterbrechungszeit `interesse2` auf TRUE setzen. Beide Threads sind dann verklemmt: Sie verweilen ewig in ihrer Warteschleife.

Algorithmus 4 (Peterson-Algorithmus)

Algorithmus 4 zeigt, dass gegenseitiger Ausschluss auf Anwenderebene mit aktivem Warten programmierbar ist:

```
#define FALSE  0
#define TRUE   1
#define N      2          // Anzahl Threads

int turn;                 // Wer ist dran ?
int interested[N]         // alle Werte anfangs FALSE

void enter_region(int process)
{
 int other;

 other = 1-process;
 interested[process] = TRUE   // Eintritt anmelden
 turn = process;
 while ( turn == process && interested[other] == TRUE )
      ; /* busy waiting */
}

void leave_region(int process)
{
 interested[process] = FALSE;
}
```

Warum garantiert der Peterson-Algorithmus gegenseitigen Ausschluss ?

(Widerspruchsbeweis)

Annahme: kein gegenseitiger Ausschluss, d.h. es gibt einen Zeitpunkt t_k, zu dem die Threads P_1 und P_2 beide im kritischen Abschnitt sind.

Sei $t_{turn=1}$ der Zeitpunkt, zu dem P_1 vor Eintritt in den kritischen Abschnitt die Wertzuweisung `turn=1` durchführt. Sei $t_{i1=FALSE}$ der Zeitpunkt, zu dem P_1 nach Verlassen des kritischen Abschnitts die Wertzuweisung `interested[1]=FALSE` durchführt.

Entsprechend sei $t_{turn=2}$ der Zeitpunkt, zu dem P_2 vor Eintritt in den kritischen Abschnitt die Wertzuweisung `turn=2` durchführt und $t_{i2=FALSE}$ der Zeitpunkt, zu dem P_2 nach Verlassen des kritischen Abschnitts die Wertzuweisung `interested[2]=FALSE` durchführt.

Der kritische Abschnitt von P_1 ist ein Teilintervall des offenen Zeitintervalls $(t_{turn=1}, t_{i1=FALSE})$. Der kritische Abschnitt von P_2 ist ein Teilintervall des offenen Zeitintervalls $(t_{turn=2}, t_{i2=FALSE})$.

t_k liegt in $(t_{turn=1}, t_{i1=FALSE}) \cap (t_{turn=2}, t_{i2=FALSE})$.
Aus dem Quelltext ersieht man folgendes:

1. Zu keinem Zeitpunkt innerhalb $(t_{turn=1}, t_{i1=FALSE})$ erfolgt eine Wertzuweisung `turn=1` .

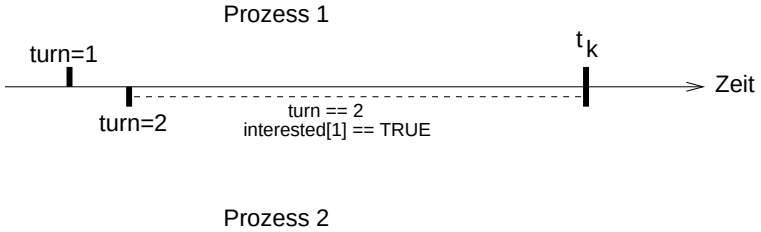
2. Zu jedem Zeitpunkt $t \in (t_{turn=1}, t_{i1=FALSE})$ hat `interested[1]` den Wert TRUE.

3. Zu keinem Zeitpunkt innerhalb $(t_{turn=2}, t_{i2=FALSE})$ erfolgt eine Wertzuweisung `turn=2` .

4. Zu jedem Zeitpunkt $t \in (t_{turn=2}, t_{i2=FALSE})$ hat `interested[2]` den Wert TRUE.

Fall 1:

$t_{turn=1} < t_{turn=2}$



Irgendwann in $(t_{turn=2}, t_k)$ muss P_2 über die busy-wait-Schleife hinausgekommen sein. Dazu muss entweder `interested[1]` den Wert FALSE oder turn den Wert 1 gehabt haben. Nun gilt

$(t_{turn=2}, t_k) \subset (t_{turn=1}, t_k) \subset (t_{turn=1}, t_{i1=FALSE})$

Aus 2 folgt, dass `interested[1]` in $(t_{turn=2}, t_k)$ immer TRUE war. Zum Zeitpunkt $t_{turn=2}$ ist turn=2. Aus 1 folgt, das turn im gesamten Intervall $(t_{turn=2}, t_k)$ den Wert 2 hat. Dies führt zum Widerspruch zur Annnahme, dass P_2 im kritischen Abschnitt ist.

Fall 2:

$t_{turn=2} > t_{turn=1}$

Symmetrisch.

Aufgabe: Zeigen Sie, dass der Algorithmus auch verklemmungsfrei ist.

4.5 Abschalten von Unterbrechungen

Gegenseitiger Ausschluss ließe sich auch dadurch implementieren, dass im kritischen Abschnitt die Hardware-Interrupts einfach „abgeschaltet“ werden. Der ausführende Prozess ist dann nicht mehr unterbrechbar und kann ohne Wettbewerb auf gemeinsame Daten zugreifen.

Innerhalb des Betriebssystems wird dieser Mechanismus auch teilweise genutzt (näheres siehe **??**, S. **??**). In einem Multitasking-System kann man eine so mächtige Operation aber nicht dem Anwender zugänglich machen. Zu groß ist die Gefahr, dass durch Programmfehler ein Prozess im kritischen Abschnitt hängenbleibt. Bei abgeschalteten Interrupts führt dies zum Stillstand des gesamten Systems.

4.6 Aktives Warten auf Maschinenebene

Für aktives Warten gibt es auf der Maschinenebene meist eine atomare Instruktion (*SWAP* oder *TEST-AND-SET*), die den Inhalt eines Registers in atomarer Weise prüft und modifiziert.

Nehmen wir an, die Instruktion heißt *SWAP* und tauscht die Inhalte zweier Register aus. Wir verwenden ein gemeinsam benutztes Register als Sperre: Ist der darin enthaltene Wert Null, ist der kritische Abschnitt frei und kann betreten werden. Dabei muss die Sperre gesetzt werden. Die Prüfung und das Setzen der Sperre kann in einer einzigen SWAP-Operation erfolgen:

```
/* gemeinsam benutzte Variablen */
int warte1=0;
int warte2=0;

...

thread1(){
  while (TRUE) {
       .
       .  unkritische Aktivität
       .
    warte1=1;
    while (warte1)
       swap(warte1, sperre);
        .
        .  kritischer Abschnitt
        .
    sperre=0;
  }
}

thread2(){
  while (TRUE) {
       .
       .  unkritische Aktivität
       .
    warte2=1;
    while (warte2)
       swap(warte2, sperre);
        .
        .  kritischer Abschnitt
        .
    sperre=0;
  }
}
```

Falls ein Rechner keine solche Maschineninstruktion besitzt, kann das Betriebssystem durch Abschalten der Interrupts zwischen Prüfen und Setzen des Sperr-Registers eine äquivalente Operation implementieren.

4.7 Spinlocks

Wir definieren ein Objekttyp *Spinlock*, der als abstrakte Schnittstelle zu einem Synchronisationsmechanismus dient, der auf aktivem Warten basiert. Spinlocks lassen sich (z.B. mit dem Peterson-Algorithmus) auch auf der Anwenderebene implementieren, normalerweise stellt aber ein Betriebssystem die Operationen zur Verfügung und

implementiert sie mittels der oben beschriebenen SWAP-Operation.

```
class spinlock {
public:
   spinlock(void);
   lock(void);
   unlock(void);
};
```

- Der Konstruktor initialisiert die Sperre mit „frei“.
- *lock* wartet aktiv auf das Freiwerden der Sperre. Ist die Sperre freigegeben, wird sie gesetzt. Man sagt auch, der aufrufende Prozess „hält die Sperre“. Prüfung und Setzen erfolgt in Form einer atomaren Operation.
- *unlock* gibt die Sperre frei. Es liegt in der Verantwortung des Programmierers, dass nur der Prozess, der die Sperre hält, *unlock* aufrufen darf.

  Wenn mehrere Prozesse an der selben Sperre warten, bestimmt der Zufall, welcher die Sperre nach der Freigabe zuerst erhält.

Gegenseitiger Ausschluss mit Spinlocks

```
spinlock l;

while (TRUE) {
    .
    .    unkritische Aktivität
    .
  l.lock(); // ggf. Warten
      .
      .  kritischer Abschnitt
      .
  l.unlock();
}
```

4.8 Blockieren von Prozessen, Warteschlangen

Blockieren ist aus Prozess-Sicht eine Meta-Operation: der Prozess manipuliert dabei nicht irgenwelche Datenstrukturen, sondern ist selbst in der Rolle eines manipulierten Objekts. Er wird schlafen gelegt, später wieder aufgeweckt. Die Implementierungen von Blockieroperationen müssen wir also nicht im Anwenderprogramm, sondern auf Betriebssystemebene suchen.

Blockierende Operation gibt es in vielen Varianten. Wir diskutieren

- Semaphore (Zählende Semaphore)
- Mutex (Binärsemaphore)
- Ereignisvariablen
- Nachrichtenübertragung

Gemeinsam ist allen Blockieroperationen, dass jeweils eine **Warteschlange** zur Implementierung benötigt wird:

- Beim Blockieren wird der Prozess, der die Blockieroperation aufruft, in die Warteschlange eingereiht.
- Beim Aufwecken wird ein Prozess aus der Warteschlange wieder herausgenommen. Aufwecken kann sich ein Prozess nicht selbst, ein anderer muss die Weckoperation aufrufen!
- Wenn ein Prozess mit einer Weckoperationen genau einen von mehreren wartenden Prozessen aufweckt, stellt sich die Frage: Welchen?

  Zwei Strategien sind üblich:

  - Der Prozess wird geweckt, der am längsten wartet (FIFO = “first in“ – „first out“)
  - Das Betriebssystem bestimmt anhand von Prioritätszahlen, dass der Prozess mit der höchsten Priorität aus der Warteschlange genommen wird.
- In manchen Anwendungen ist es dagegen sinnvoll, alle wartenden Prozesse aufzuwecken.

4.9 Semaphore (Zählende Semaphore)

Ein Semaphor wird mit einer Warteschlange und einem Zähler implementiert. Die Schnittstelle:

```
class semaphor {
public:
   semaphor(int anfangswert);
   down(void);
   up(void);
};
```

- Beim Konstruktor-Aufruf wird eine neue, leere Warteschlange eingerichtet und der Zähler mit dem angegebenen Wert initialisiert
- *down*

  - dekrementiert den Zähler, falls dieser positiv ist, oder
  - blockiert den Aufrufer, falls der Zähler gleich Null ist.

- *up*
  - weckt einen Prozess auf, falls die Warteschlange nicht leer ist, oder
  - inkrementiert den Zähler

Mit Semaphoren kann man recht einfach für die Synchronisation von Prozessen sorgen, die sich um exklusiven Zugriff auf eine von *N* verfügbaren Ressourcen bemühen.

Beispiel: In einem System gibt es 3 Drucker. Um nebenläufige Zugriffe zu synchronisieren, wird ein Semaphor mit dem Zählerwert 3 initialisiert. Jeder Druckauftrag führt zu Beginn eine *down*-Operation durch. Die ersten 3 Aufträge müssen also nicht warten, da nur der Zähler dekrementiert wird. Weitere Aufträge werden blockiert.

Nach Beendigung einer Druckernutzung wird *up* aufgerufen. Ein Drucker ist nun wieder verfügbar. Ist die Warteschlange leer, wird der Semaphorwert inkrementiert. Warten Prozesse, wird dagegen einer aufgeweckt.

Der Zähler gibt zu jedem Zeitpunkt die Anzahl freier Drucker an.

```
class druckdienst {
  semaphor drucker(DRUCKERANZAHL);
  ...
public:
  void drucke(string datei);
};

void druckdienst::drucke(string datei){

  drucker.down(); // ggf. warten, bis Drucker frei

  ...             // Drucker exklusiv nutzen

  drucker.up();   // freigeben
}
```

4.10 Mutex (Binärsemaphore)

Mit Semaphoren kann man natürlich auch für gegenseitigen Ausschluss sorgen. Gemeinsame Daten können als eine nur einmal vorhanden Ressource betrachtet werden. Somit lässt sich gegenseitiger Ausschluss als Spezialfall des oben verwendeten Musters für *N*=1 betrachten.

Der Spezialfall wird allerdings so häufig benötigt, dass wir einen neuen Konstruktor ohne Zählerinitialisierung einführen. Einen Semaphor, der nur bis 1 zählen kann, nennt man **binären** Semaphor oder **Mutex** (von „mutual exclusion“).

```
class mutex {
public:
   mutex(void);
   down(void);
   up(void);
};
```

Da *up* und *down* für Semaphor- und Mutexobjekte identisch sind, bietet sich natürlich auch die Spezialisierung durch Vererbung an:

```
class mutex : public semaphor {
public:
   mutex(void);
};

mutex::mutex()
      :semaphor(1) // Mutex = Semaphor mit Anfangswert 1
{}
```

Gegenseitiger Ausschluss mit einem Mutex

```
mutex m;

while (TRUE) {
     .
     .    unkritische Aktivität
     .
  m.down(); // ggf. Warten
       .
       .  kritischer Abschnitt
       .
  m.up();
}
```

Dies gleicht auf der Quelltextebene exakt der Spinlock-Lösung(**??**, S. **??**). Man beachte aber, dass Spinlocks durch aktives Warten realisiert sind.

Beispiel: Exklusiver Druckerzugriff

Wir kommen noch einmal auf das Druckerbeispiel zurück. Nach dem Warten auf einen Drucker

```
drucker.down();
```

muss noch festgestellt werden, welcher Drucker frei ist. Ein freier Drucker muss reserviert werden, dann wird gedruckt. Nehmen wir an zum Drucken gibt es eine Systemmethode,

```
system.print(int druckernummer, datei d);
```

die sich nicht um die Synchronisation kümmert, sondern voraussetzt, dass der Drucker mit der angegebenen Nummer exklusiv genutzt wird.

Betrachten wir folgende Implementierung:

```
class druckdienst {
  semaphor drucker(DRUCKERANZAHL);
  boolean istfrei[DRUCKERANZAHL];
public:
  druckdienst(void);
  void drucke(string datei);
};

druckdienst::druckdienst(){
  // alle Drucker als "frei" markieren
  for(int i=1; i<=DRUCKERANZAHL; i++){
          istfrei[i]=TRUE;
     }
}

void druckdienst::drucke(string datei){
  int d=-1;

  drucker.down(); // ggf. warten, bis Drucker frei

  // freien Drucker bestimmen und reservieren
  while(! istfrei[++d])
      ;
  istfrei[d]=FALSE;

  system.print(d, datei);

  // freigeben
  istfrei[d]=TRUE;
  drucker.up();
}
```

Das Problem des Wartens auf einen Drucker ist mit dem Semaphor *drucker* zwar gelöst, aber durch gleichzeitigen Datenzugriff auf das Feld *istfrei* kann es wieder zu Wettbewerbsbedingungen kommen.

Der Zugriff auf *istfrei* ist ein kritischer Abschnitt und muss unter gegenseitigem Ausschluss geschehen. Wir verwenden dazu einen Mutex *sperre*:

```
class druckdienst {
  semaphor drucker(DRUCKERANZAHL);
     // Warten auf freien Drucker
  mutex sperre; // fuer kritische Abschnitte
  boolean istfrei[DRUCKERANZAHL];
     // welcher Drucker ist frei?
public:
  druckdienst(void);
  void drucke(string datei);
};

druckdienst::druckdienst(){
  // alle Drucker als "frei" markieren
  for(int i=1; i<=DRUCKERANZAHL; i++){
          istfrei[i]=TRUE;
     }
}

void druckdienst::drucke(string datei){
  int d=-1;

  drucker.down(); // ggf. warten, bis Drucker frei

  // freien Drucker bestimmen und reservieren
  sperre.down();
  while(! istfrei[++d])
      ;
  istfrei[d]=FALSE;
  sperre.up();

  system.print(d, datei);

  // freigeben
  sperre.down();
  istfrei[d]=TRUE;
  sperre.up();
  drucker.up();
}
```

Wichtig ist, das das Drucken selbst nicht in den kritischen Abschnitt gehört. Ansonsten wird die gleichzeitige Nutzung der Drucker verhindert!

Fatal, wenn versehentlich im kritischen Abschnitt auf das Freiwerden eines Druckers gewartet wird:

```
    .
    .
    .
sperre.down();
drucker.down(); // ggf. warten, bis Drucker frei
    .
    .
    .
```

Wenn kein Drucker frei ist, kommt es zur Verklemmung, denn eine Freigabe ist nun nicht mehr möglich.

4.11 Monitore

Ein **Monitor** ist gleichzeitig ein Modularisierungs- und ein Synchronisationsmechanismus.

Bei den bisher besprochenen Beispielen ist der Programmtext, der zur Synchronistation dient, auf die verschiedenen kritischen Abschnitte der konkurrierenden oder kooperierenden Prozesse verteilt. Wenn durch eine fehlerhaft verwendete Synchronistationsoperation Verklemmmungen oder Wettbewerbsbedingungen auftreten, muss an vielen Stellen nach dem Fehler gesucht werden.

Die Monitor-Grundidee besteht darin, die kritischen Abschnitte in einem einzigen Modul unterzubringen. Damit wird die Synchronisation also gewissermassen zentralisiert.

Das Monitorkonzept findet man in manchen Programmiersprachen als Sprachkonzept wieder (z.B. in Java der *synchronized*-Mechanismus).

Definition

Ein Monitor ist ein Modul mit mehreren Monitorfunktionen zur Manipulation gemeinsamer Daten. Die Ausführung der Monitorfunktionen erfolgt unter gegenseitigem Ausschluss.

Der gegenseitige Ausschluss, der also nicht mehr explizit programmiert werden muss, reicht alleine noch nicht aus. Man braucht zusätzlich noch Blockier- und Weckoperationen. Monitore werden meist in Kombination mit Ereignisvariablen verwendet.

4.11.1 Ereignisvariablen

Eine Ereignisvariable (engl.: „event“, „condition“) ist eine Warteschlange mit einfachen Warte- und Weckfunktionen:

```
class event {
public:
   event(void);
   void wait(void);
   void signal(void);
   void broadcast(void);
};
```

- *wait* - Aufrufer blockiert
- *signal* - Falls die Warteschlange nicht leer ist, einen Prozess aufwecken. (Andernfalls keine Wirkung)
- *broadcast* - Falls die Warteschlange nicht leer ist, alle Prozesse aufwecken

Bei den „klassischen“ Ereignisvariablen fehlt die *broadcast*-Operation, sie ist jedoch manchmal nützlich. Ereignisvariablen benutzt man zusammen mit anderen Synchronisationsmechanismen: Der *wait*-Aufruf erfolgt üblicherweise, nachdem ein Prozess innerhalb eines kritischen Abschnitts festgestellt hat, dass er auf irgendwelche Ressourcen warten muss. Mit der *wait*-Operation muss der Aufrufer gleichzeitig seinen kritischen Abschnitt verlassen.

4.11.2 Funktionsweise

Ein Monitor kontrolliert den nebenläufigen Zugriff auf Ressourcen. Er beinhaltet

- die Datenstrukturen, die die Ressourcen implementieren
- die Funktionen zum exklusiven Zugriff auf die Ressourcen
- Ereignisvariablen zum passiven Warten auf Ressourcen

Ein Prozess, der auf eine vom Monitor *M* kontrollierte Ressource zugreifen will, muss dazu eine der Monitorfunktionen aufrufen. Wenn gerade eine andere Funktion des Monitor aktiv ist, muss er „am Monitor-Eingang“ warten. Wenn er innerhalb des Monitor mit einer Monitor-Ereignisvariable blockiert, macht er den Monitor wieder *frei* für einen anderen Prozess.

```
/* gemeinsame Variablen: */
static int stack[100];
static int stackpointer=0;
- - - - - - - - - - - - - - - - - - - - - - - - - -
/* event−Variablen zur Synchronis. : */
event Event−A
event Event−B
event Event−C
- - - - - - - - - - - - - - - - - - - - - - - - - -
/* Funktionen zur Verarbeitung der
   gemeinsamen Daten */
function f1 ...
function f2 ...
function f3 ...
   . . .
```

Abbildung 4.2: Monitor - statische Sicht

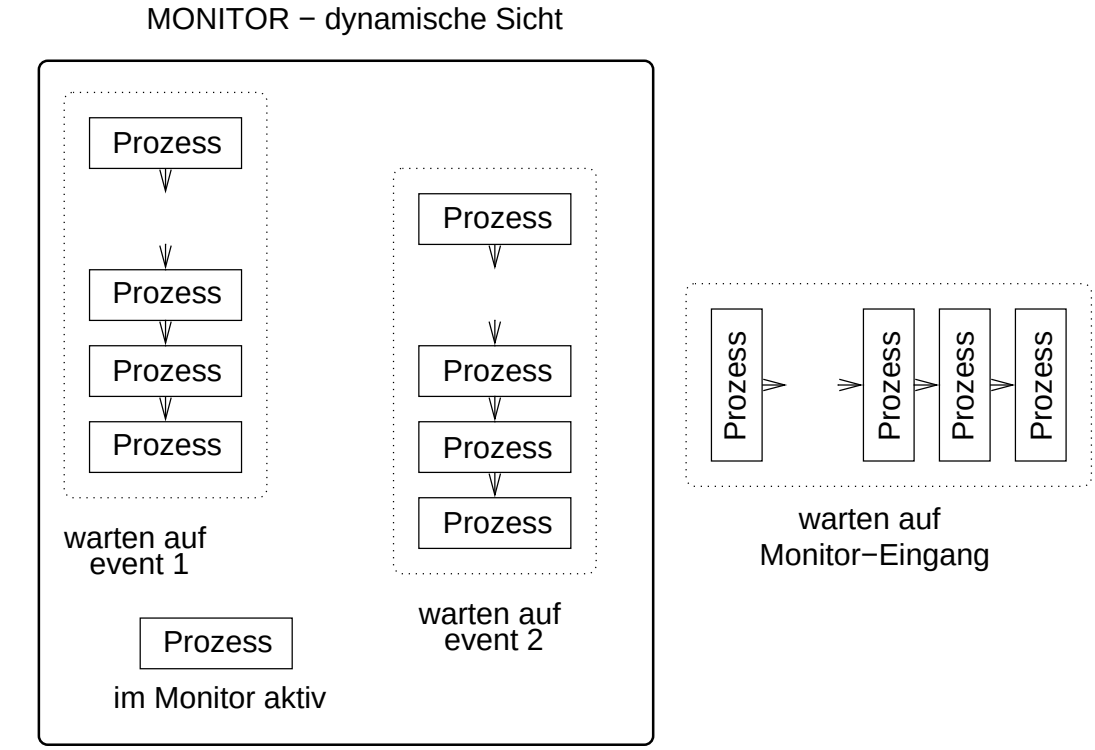


Abbildung 4.3: Monitor - dynamische Sicht

Wir versuchen unsere Mailbox in Monitor-Manier zu implementieren:

mailbox.h

```
const int max=1000;
typedef int nachricht;

class mailbox {
  mutex sperre;  // fuer gegenseitigen Ausschluss
  event neue_nachricht;
  event platz_frei;

  nachricht puffer[max];
  int ifrei,   // Index freier Platz
      ilies,   // Index naechste ungelesene Nachricht
      anzahl;  // Anzahl ungelesener Nachrichten

public:
  mailbox(void);
  void neu(nachricht n);
  nachricht lies(void);
};
```

Um den Monitor-Funktionen Exklusivzugriff auf die Ressource „Mailbox“ zu gewähren, muss der Mutex verwendet werden:

```
nachricht mailbox::lies() {
  sperre.down();
  ... Nachricht entnehmen ...
  sperre.up();
}

nachricht mailbox::neu() {
  sperre.down();
  ... Nachricht ablegen ...
  sperre.up();
}
```

Wenn aber im kritischen Bereich auf das Eintreffen einer Nachricht gewartet wird, gibt es Verklemmungen.

```
nachricht mailbox::lies() {
  sperre.down();
  if (anzahl==0)
      neue_nachricht.wait(); // fuer immer warten !
  ...
}
```

Verlassen des kritischen Abschnitts vor der *wait*-Operation ist folgender Ansatz auch keine Lösung:

```
nachricht mailbox::lies() {
  sperre.down();
  if (anzahl==0){
      sperre.up();
      neue_nachricht.wait();
      sperre.down();
  ...
}
```

Wenn nach dem *up*- und vor *wait*-Aufruf eine neue Nachricht eingefügt signalisiert wird, blockiert *lies*.

Man braucht eine atomare Operation, die den Prozess blockiert und dabei gleichzeitig den kritischen Abschnitt freigibt.

In Anlehnung an die Synchronisationsmechanismen für POSIX-Threads erweitern wir die Schnittstelle der Ereignisvariablen, um eine Warteoperation, die gleichzeitig einen als Parameter übergebenen Mutex *m* freigibt:

```
class event {
public:
   event(void);
   void wait(void);
   void wait(mutex &m);
   void signal(void);
   void broadcast(void);
};
```

Damit könnte die Lösung folgendermaßen aussehen:

mailbox.cc

```
mailbox::mailbox(){
  ifrei=ilies=anzahl=0;
}

void mailbox::neu(nachricht n){
  sperre.down();
  // falls Puffer voll, warten
  while (anzahl==MAX)
     platz_frei.wait(sperre); // Sperre freigeben und warten
  // Sperre ist bei wait-Rückkehr wieder gesetzt!
  puffer[ifrei]=n;
  ifrei= (ifrei + 1) % MAX;
  anzahl++;
  neue_nachricht.signal(); // ggf. Leser wecken
  sperre.up();
}

nachricht mailbox::lies() {
  nachricht n;

  sperre.down();
  // falls Puffer leer, warten
  while (anzahl==0)
     neue_nachricht.wait(sperre); // warte

  n=puffer[ilies];
  ilies=(ilies+1) % MAX;
  anzahl--;
  platz_frei.signal(); // ggf. Schreiber wecken
  sperre.up();
  return n;
}
```

Eine Programmiersprache, die ein Monitorkonzept enthält, macht die Sache einfacher. In Java müsste man die beiden Methoden *lies* und *neu* lediglich als **synchronized** deklarieren.

4.12 Nachrichtenübertragung

Nachrichtenübertragung („message passing“) dient zur Kommunikation zwischen kooperierenden Prozessen (oder Threads). Auch der Austausch von Informationen durch

Verwendung gemeinsamer Variablen (gemeinsam genutzter Hauptspeicher) ist eine Form der Prozesskommunikation. Gemeinsame Hauptspeicher-Nutzung benötigt immer Synchronisationsmechnismen.

Nachrichtenübertragungsmechanismen übermitteln Daten von einem Sender zu einem Empfänger. Sender und Empfänger sind Prozesse (oder Threads). Dabei beinhaltet die Übertragung auch Synchronisation:

- Bei **synchroner Nachrichtenübertragung** („Rendezvous“) sind Sender und Empfänger zum gleichen Zeitpunkt an der Nachrichtenübertragung beteiligt. Ein Beispiel ist das Telefonieren.

  Synchronisation: Die Sendeoperation blockiert den Sender ggf. solange, bis der Empfänger die Empfangsoperation ausführt. Das gleiche gilt umgekehrt.

- Bei **asynchroner Nachrichtenübertragung** wird der Sender beim Senden nicht blockiert. Stattdessen wird die Nachricht in einen Zwischenspeicher des Empfängers abgelegt, der sie später jederzeit lesen kann.

  Ein Beispiel ist die Übermittlung der Nachricht an einen Anrufbeantworter. Ein anderes Beispiel ist die Kommunikation mit einem beschränkten Puffer, z.B. die oben diskutierte Mailbox oder UNIX-*Pipes*.

  Synchronisation: Eventuell blockiert die Empfangsoperation den Empfänger bis zum Eintreffen einer Nachricht. Die Sendeoperation blockiert den Sender ggf., wenn der Zwischenspeicher des Empfängers voll ist. Oft kann der Programmierer auch zwischen blockierenden und nicht-blockierenden Operationen wählen!

Ein wichtiges Kriterium für die Beurteilung eines Nachrichtenübertragungskonzepts ist die Verwendbarkeit für rechnerübergreifende Prozesskommunikation:

- **Lokale Nachrichtenübertragung** wird zwischen Prozessen verwendet, die auf dem selben Rechner ablaufen. Die Nachrichtenübertragung kann durch Verwendung gemeinsamen Hauptspeichers bewerkstelligt werden oder dadurch, dass ein Betriebssystemdienst die Nachricht vom Sendepuffer des Senders in den Empfangspuffer des Empfängerprozesses kopiert (vgl. **??**, S. **??**).

  UNIX-Pipes sind beispielsweise nur für lokale Kommunikation verwendbar.

- **Nachrichtenübertragung im Netzwerk** wird zwischen Prozessen verwendet, die auf unterschiedlichen vernetzten Rechnern ablaufen. Die Nachrichtenübertragung wird durch Verwendung von Netzwerk-Kommunikationsmechanismen implementiert.

  Der UNIX-„Socket“-Mechanismus kann für Netzwerk-Kommunikation verwendet werden.

Kapitel 5

Verklemmungen

5.1 Das Problem der dinierenden Philosophen

Wir betrachten ein bekanntes Synchronosationsproblem:

Fünf Philosophen verbringen ihr Leben mit Denken und Essen. Sie benutzen einen gemeinsamen runden Tisch mit fünf Stühlen, jeder gehört einem der Philosophen. In der Tischmitte steht eine Reisschüssel, auf dem Tisch liegen 5 Essstäbchen:

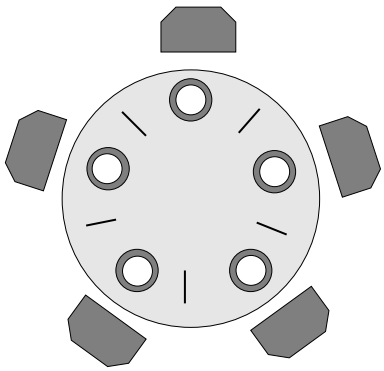


Abbildung 5.1: Esstisch der Philosophen

Das Problem: Gemeinsame Benutzung der Essstäbchen durch zwei benachbart sitzende Philosophen.

Man betrachte folgendes Modell dazu:

philosoph.h

```
#include <ostream.h>
#include <pthread.h>
#include "mutex.h"

const int N=5; // Anzahl der Philosophen

class philosoph {
  int id;    // Nummer noetig zur Staebchen-Zuordnung
  void esse(void);
  void denke(void);
public:
  philosoph(int id);
  void lebe(void);
};
```

philosoph.cc

```
mutex ess_stab[N];  // zum Warten auf ein Staebchen

philosoph::philosoph(int id){ this->id=id; } // eigene Identität merken

void philosoph::esse(){
  const int links=id, rechts=(id+1) % N;

  ess_stab[links].down();
  ess_stab[rechts].down();
  cout << "(id=" << id << ") ich esse ...\n";
  ess_stab[links].up();
  ess_stab[rechts].up();
}

void philosoph::denke(){
  cout << "(id=" << id << ") ich denke gerade \n";
}

void philosoph::lebe(){ while(1) { denke(); esse(); } }

void run (int id){ // einen Philosoph aktivieren
  (new philosoph(id))->lebe();
}
```

```
main(){
  // Philosophen als POSIX-Threads nebenläufig aktivieren
  pthread_t my_thread[N];

  for(int i=0; i<N; i++)
    pthread_create(&my_thread[i], NULL,
           (void* (*)(void *)) run, // Thread-Funktion
           (void*) i);              // Argument fuer "run"
  for(int i=0; i<N; i++)
    pthread_join(my_thread[i], NULL);
}
```

Die Lösung kann zur Verklemmung führen, wenn alle Philosophen sich an den Tisch setzen und jeder zunächst das links von ihm liegende Essstäbchen nimmt.

```
ess_stab[links].down();
```

Jeder wartet anschließend auf das Freiwerden des anderen Essstäbchens.

```
ess_stab[rechts].down();
```

Keiner kann essen, keiner wird das reservierte Stäbchen wieder freigeben, das Programm muss von außen abgebrochen werden.

Abfolge-Beispiel mit Verklemmung:

1 Philosoph 0 nimmt Essstäbchen 0
2 Philosoph 1 nimmt Essstäbchen 1
3 Philosoph 2 nimmt Essstäbchen 2
4 Philosoph 3 nimmt Essstäbchen 3
5 Philosoph 4 nimmt Essstäbchen 4
6 Philosoph 0 wartet auf Essstäbchen 1
7 Philosoph 1 wartet auf Essstäbchen 2
8 Philosoph 2 wartet auf Essstäbchen 3
9 Philosoph 3 wartet auf Essstäbchen 4
10 Philosoph 4 wartet auf Essstäbchen 0

Verklemmungen treten bei konkurrierendem Zugriff auf Ressourcen auf, die exklusiv nur von einem Prozess benutzt werden sollen. Ressourcen können z.B. E/A-Geräte, Dateien, Datenbanktabellen, oder Daten im Hauptspeicher sein.

Oder bei einer Verkehrssteuerung, das Recht, eine Kreuzung zu passieren:

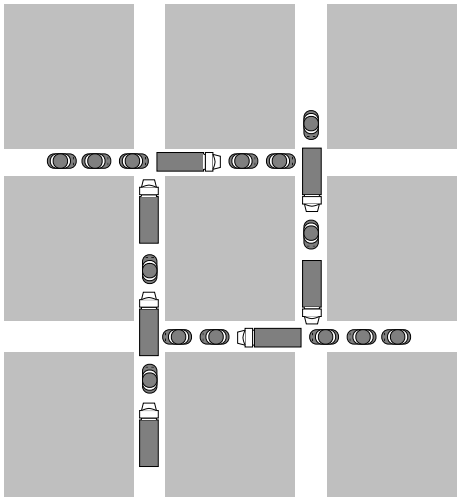


Abbildung 5.2: Verklemmungsbeispiel

5.2 Systemmodell

Betrachten wir ein System mit einer festen Menge von Ressourcen, die unter gegenseitigem Ausschluss von konkurrierenden Prozessen genutzt werden. Jeder Prozess

- reserviert eine Ressource, wartet dabei ggf., bis sie frei wird
- nutzt die Ressource
- gibt die Ressource wieder frei

Definition

Eine Menge von Prozessen heißt **verklemmt**, wenn jeder Prozess in der Menge auf eine Ressource wartet, die von einem anderen Prozess in der Menge reserviert ist.

Die Ressourcen im Philosophenbeispiel sind die Essstäbchen, die Verklemmung tritt auf, wenn jeder Philosoph ein Stäbchen reserviert hat und auf das andere benötigte Stäbchen wartet.

Verklemmungen treten auf, wenn folgende Bedingungen gemeinsam erfüllt sind:

- Exklusivnutzung: Ressourcen werden exklusiv von einem Prozess genutzt
- Reservieren und Warten: Reservierte Ressourcen werden erst nach der Nutzung freigegeben
- Keine Wegnahme: Ressourcen, die ein Prozess reserviert hat, werden ihm nicht weggenommen

- Gegenseitiges Warten: Es gibt eine Prozessmenge $\{P_0, \ldots, P_n\}$, so dass jeder P_i auf eine Ressource wartet, die $P_{i+1\ modulo\ n}$ reserviert hat

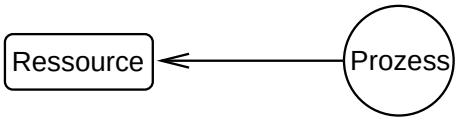
Die Ressourcenzuordnung wird in Form eines Ressourcenzuordnungs-Graphen dargestellt:

Definition

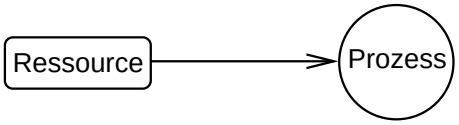
Ein **Ressourcenzuordnungsgraph** ist ein Graph mit 2 Knotentypen

- Prozessknoten
- Ressourcenknoten

Eine Kante von einem Prozessknoten *P* zu einem Ressourcenknoten *R* bedeutet: *P* wartet auf die Ressource *R*.



Eine Kante von einem Ressourcenknoten *R* zu einem Prozessknoten *P* bedeutet: *R* ist von *P* reserviert



Ist der Ressourcenzuordnungs-Graph zyklisch, liegt eine Verklemmungssituation vor.

Der Graph für das Philosophenbeispiel im Verklemmungsfall:

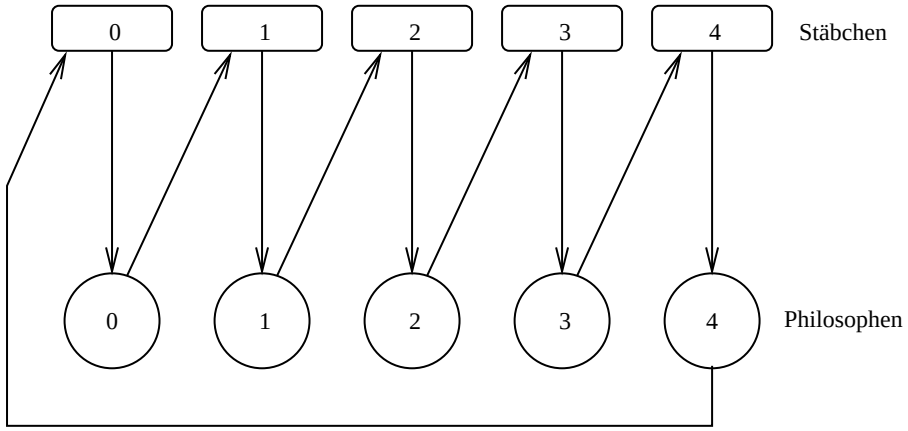


Abbildung 5.3: Ressourcenzuordnungs-Graph bei Verklemmung

Wir diskutieren Möglichkeiten, mit der Verklemmungsgefahr umzugehen.

5.3 Vogel-Strauß-"Algorithmus"

Wenn in englischsprachigen Betriebssystembüchern vom „Ostrich"-Algorithmus die Rede ist, geht es um den Vogel „Strauß". Der wird bekannterweise gerne zitiert, wenn jemand den „Kopf in den Sand steckt" und ein Problem einfach ignoriert.

Dem Verklemmungsproblem wird in der Praxis tatsächlich manchmal bewusst durch „Nichtstun" begegnet. Entdeckung, Behebung bzw. Verhindern von Verklemmungen verursachen immer Aufwand. Damit ist eine Kosten-Nutzen-Abwägung erforderlich.

Beispiel: Wenn ein System pro Monat im Durchschnitt 2 mal „abstürzt", aber nur jeder 10. Absturz durch ein Verklemmungsproblem verursacht wird, wenn außerdem die Folgen des Systemzusammenbruchs nicht gravierend sind, wird man vielleicht keinen Aufwand zur Verklemmungsbehandlung treiben wollen.

Die meisten Betriebssysteme enthalten keinerlei Verklemmungsbehandlung.

5.4 Vermeiden der Verklemmungsvoraussetzungen

Ein denkbarer Ansatz zur Behandlung des Verklemmungsproblems ist, dafür zu sorgen, dass die o.g. Verklemmungsvoraussetzungen niemals alle erfüllt sind. Für jede dieser Voraussetzungen wird überprüft, inwieweit sie in der Praxis vermieden werden kann.

5.4.1 Vermeiden unnötiger Exklusivzugriffe

Ein Prozess sollte Ressourcen, die er nicht exklusiv braucht, sicher nicht gegen anderweitigen Zugriff sperren. Ein wohlbekanntes Beispiel für die Verwendung dieses Prinzips sind Dateisperren: Hier wird zwischen Lese- und Schreibsperren unterschieden um parallele Lesezugriffe zu ermöglichen. Eine exklusive Sperre für konkurrierende Lesezugriffe wäre unnötig.

Andererseits gibt es bei vielen Ressourcen die Notwendigkeit für gegenseitigen Ausschluss, der konkurrierende Schreibzugriff auf eine Datei ist ein bekanntes Beispiel dafür.

5.4.2 Atomare Ressourcenreservierung

Verschiedene Maßgaben könnten sicherstellen, dass kein Ressourcen-Besitzer auf eine weitere Ressource wartet:

- Ein Prozess darf Ressourcen nur anfordern, wenn er bislang keine besitzt.
- Ein Prozess muss alle die von ihm benötigten Ressourcen auf einmal anfordern („alles oder nichts“).

In jedem Fall ist es dabei möglich, dass Ressourcen unnötig lange reserviert werden. Nachteile:

- schlechte Ressourcen-Nutzung
- Möglichkeit des „Verhungerns“ (ewig warten auf vielgefragte Ressourcen)

Atomare Ressourcenreservierung beim Philosophenproblem

Betrachten wir die „alles oder nichts“-Lösung für das Philosophenproblem als Beispiel: Ein Philosoph nimmt in einer atomaren Operation beide Essstäbchen. Wenn nicht beide frei sind, wartet er.

Das Warten realisieren wir zunächst aktiv. Die Prüfung, ob beide Stäbchen frei sind, wird über einen Indikator *ist_frei* durchgeführt, dabei wird mit einem Mutex *sperre* für gegenseitigen Ausschluss gesorgt.

philosoph.h

```
...
mutex sperre;

class ess_stab {
   bool frei;
public:
   ess_stab() { frei = TRUE; }
   bool ist_frei() { return frei; }
   void nimm() { frei = FALSE; }
   void gib_frei() { frei = TRUE; }
};

class philosoph ...  // wie gehabt
```

philosoph.cc

```
void philosoph::esse(){
  const int links=id, rechts=(id+1) % N;
  int habe_beide=FALSE;

  // aktives Warten auf Freiwerden beider Staebchen
  while (! habe_beide){
    sperre.down();
      if(   ess_stab[links].ist_frei()
         && ess_stab[rechts].ist_frei()) {
         ess_stab[links].nimm();
         ess_stab[rechts].nimm();
         habe_beide = TRUE;
         }
    sperre.up();
  }

  cout << "(id=" << id << ") ich esse ...\n";

  sperre.down();
    ess_stab[links].gib_frei();
    ess_stab[rechts].gib_frei();
  sperre.up();
}
```

Lösung mit passivem Warten

Da Philosophen vielleicht großen Hunger haben und relativ lange essen, wäre passives Warten sicher günstiger. Dies realisieren wir mit Ereignisvariablen. Ein Philosoph wartet auf das Freiwerden eines Stäbchens, also benötigen wir pro Essstäbchen eine Ereignisvariable. Der Einfachheit halber wird dieser Aspekt der Implementierung sichtbar gemacht: Das Ereignis *freigabe* ist eine *public*-Komponente der Klasse *ess_stab*.

Die *event*-Warteoperation gibt dabei den Mutex *sperre* frei (vgl. **??**, S. **??**).

philosoph.h

```
class ess_stab {
   bool frei;
public:
   ess_stab() { frei = TRUE; }
   bool ist_frei() { return frei; }
   void nimm() { frei = FALSE; }
   void gib_frei() { frei = TRUE; }
   event freigabe;
};
```

philosoph.cc

```
void philosoph::esse(){
  const int links=id, rechts=(id+1) % N;

  // passives Warten auf Freiwerden der Staebchen
  while (TRUE){
    sperre.down();

    // ggf. auf linkes Stäbchen warten
    if( ! ess_stab[links].ist_frei() )
         ess_stab[links].freigabe.wait(sperre);

    // linkes Stäbchen ist frei, was ist mit dem rechten?
    if( ! ess_stab[rechts].ist_frei() ) {
      // Pech gehabt, warten!
      ess_stab[rechts].freigabe.wait(sperre);
      // rechtes Stäbchen frei, linkes nicht mehr unbedingt
      // -> noch mal von vorne
      sperre.up();
      continue;
      }

    // beide frei
    ess_stab[links].nimm();
    ess_stab[rechts].nimm();
    sperre.up();
    break;
  } // end while


  cout << "(id=" << id << ") ich esse ...\n";

  sperre.down();
    // freigeben
    ess_stab[links].gib_frei();
    ess_stab[rechts].gib_frei();
    // ggf. wartenden Nachbarn wecken
    ess_stab[links].freigabe.signal();
    ess_stab[rechts].freigabe.signal();
  sperre.up();
}
```

5.4.3 Wegnahme von Ressourcen

Einem Prozess, der eine von einem anderen reservierte Ressource benötigt, könnte man seine Ressourcen wegnehmen. Der Prozess müsste dann solange blockiert werden, bis die weggenommenen und die nachträglich angeforderten Ressourcen frei werden.

Es wäre genauso denkbar, die benötigten Ressourcen dem aktuellen Besitzer abzunehmen, falls dieser blockiert ist.

Ressourcenwegnahme ist prinzipiell möglich, wenn der Zustand eines Prozesses zum Zeitpunkt der Wegnahme gespeichert und später wiederhergestellt werden kann. Ein Beispiel ist die Wegnahme des Prozessors (Verdrängung) oder die Wegnahme von Hauptspeicherbereichen (z.B. beim Paging). An diesen Beispielen sieht man aber auch, dass die Speicherung und spätere Wiederherstellung des Prozesszustands erheblichen technischen Aufwand erfordert.

Auf beliebige Ressourcen lässt sich dieser Ansatz jedenfalls nicht verallgemeinern.

5.4.4 Gegenseitiges Warten vermeiden

Eine manchmal mögliche Strategie zur Verhinderung des gegenseitigen Wartens ordnet die Ressourcen linear, z.B. durch Einführung eindeutiger Ressourcen-Nummern. Jeder Prozess darf Ressourcen nur in aufsteigender Anordnung reservieren.

Die Strategie ist z.B. im Philosophenbeispiel anwendbar: Jeder Philosoph darf die Essstäbchen nur gemäß aufsteigender Nummerierung reservieren. Dies heißt für unsere Beispiel-Implementierung: Philosoph 4 greift zuerst nach dem rechten Stäbchen (Nummer 0), dann nach dem linken (Nummer 4). Alle anderen verhalten sich wie gehabt.

Man sieht am Ressourcenzuordnungsgraphen, dass dann kein Zyklus entstehen kann: Kein Pfad darf von einer Ressource *i* zu einer anderen mit kleinerer Nummer *j* führen. Dies würde nämlich bedeuten, dass ein Prozess die Ressource *j* anfordert und schon eine mit größerer Nummer besitzt.

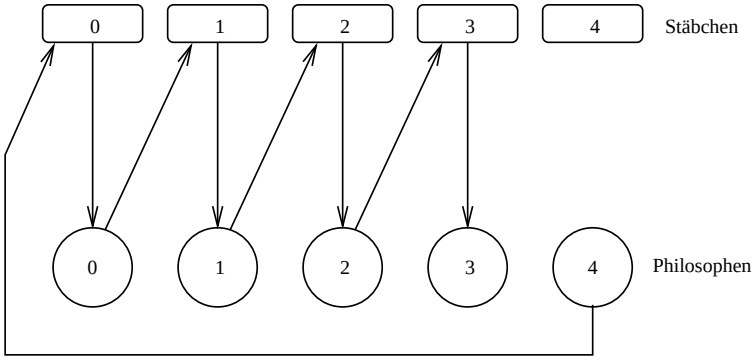


Abbildung 5.4: Anforderung nur gemäß aufsteigender Anordnung

Philosoph 4 wartet zuerst auf Stäbchen 0. Dadurch kann in dieser Situation Philosoph 3 Stäbchen 4 erfolgreich anfordern.

5.5 Sichere Ressourcenvergabestrategien

Wenn schon von vorneherein feststeht, welche Ressourcen die Prozesse im Verlauf ihrer Bearbeitung benötigen, kann man die Reihenfolge der Vergabe so zu wählen versuchen, dass keine Verklemmung auftreten kann.

Man kann dann beispielsweise den Ressourcenzuordnungsgraphen verwenden, um mögliche Verklemmungssituationen zu verhindern. Die im voraus bekanntgegebenen Ressourcenanforderungen werden in Form eines neuen Kantentyps „später benötigt“ vom Prozessknoten zum Ressourcenknoten eingetragen. Bei einer Zuteilung wird die „später benötigt“-Kante in eine Zuteilungskante (in umgekehrter Richtung) umgewandelt, bei Freigabe umgekehrt.

Zunächst wird jedoch geprüft, ob die Zuteilung zu einem Zyklus im Graphen führt. Ist dies der Fall, wird die Zuteilung versagt, weil sie in einen **unsicheren Zustand** führt.

Betrachten wir wieder das Philosophen-Beispiel. Wenn in der nachfolgend dargestellten Situation Philosoph 4 das 4. Stäbchen anfordert, würde die Zuteilung zu einem Zyklus führen.

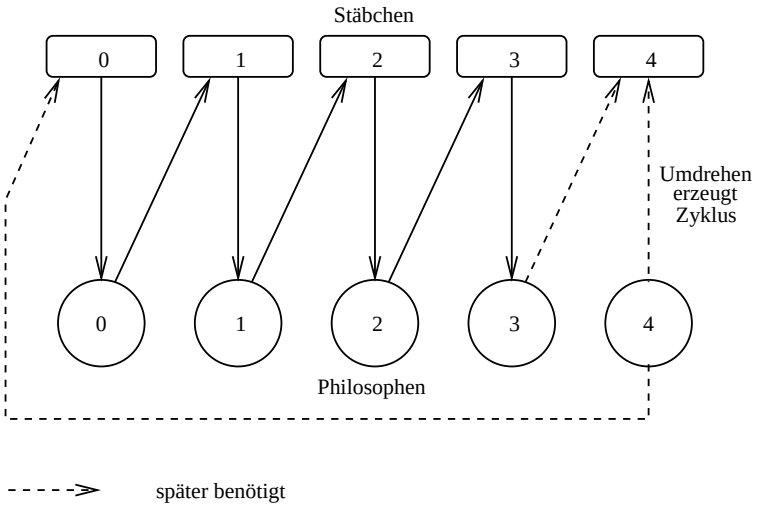


Abbildung 5.5: Sichere Vergabe im Philosophenbeispiel - Szenario 1

Philosoph 4 fordert in dieser Situation Stäbchen 4 an. Er bekommt es nicht, sondern

wird blockiert.

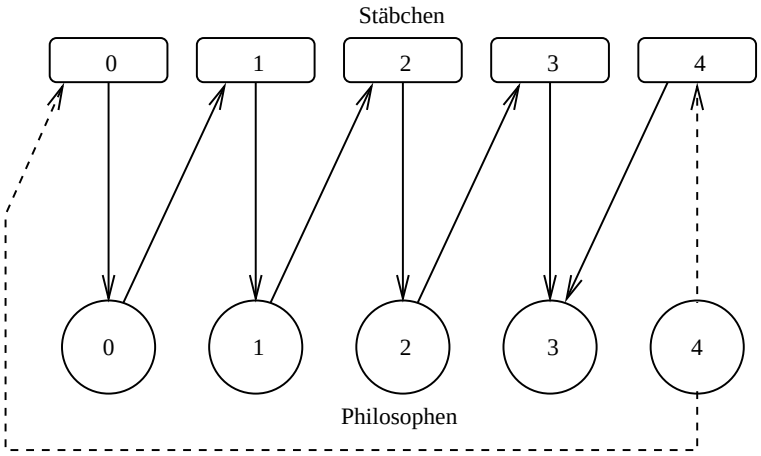


Abbildung 5.6: Sichere Vergabe im Philosophenbeispiel - Szenario 2

Philosoph 3 kommt zuerst zum Zug

Situation nach Freigabe durch Philosoph 3:

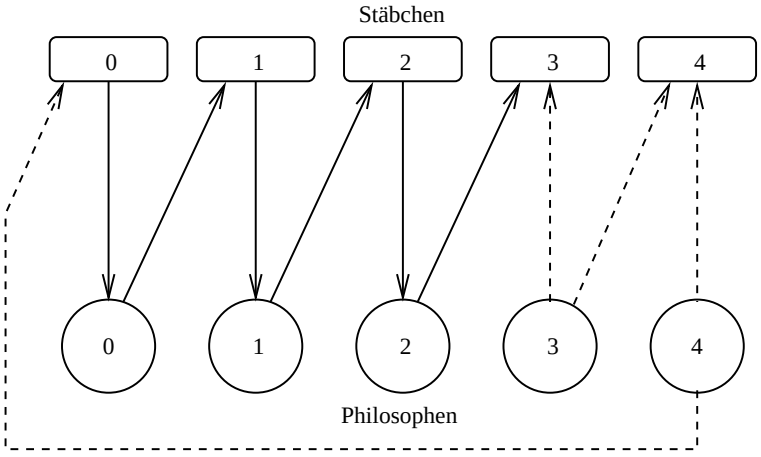


Abbildung 5.7: Sichere Vergabe im Philosophenbeispiel - Szenario 3

Nach dem Essen von Philosoph 3 ist die Vergabe von Stäbchen 4 an Philosoph 4 sicher.

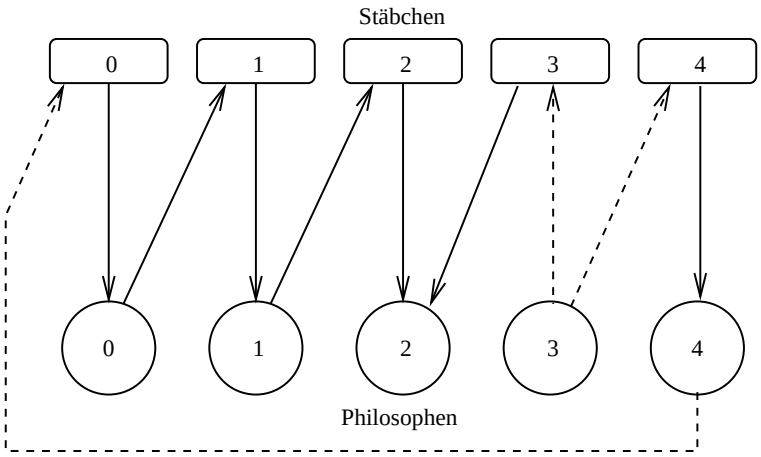


Abbildung 5.8: Sichere Vergabe im Philosophenbeispiel - Szenario 4

Vergabe von Stäbchen 4 an Philosoph 4 ohne Zyklus

Der Algorithmus ist nicht verwendbar, wenn es Ressourcentypen mit mehreren gleichwertigen Ressourcen pro Typ gibt. Ein in der Betriebssystem-Literatur gerne zitierter Algorithmus für eine sichere Ressorcenvergabe bei gleichwertigen Ressourcen ist der Bankier-Algorithmus („banker's algorithm"). Da er ineffizient ist und in der Praxis wenig Bedeutung hat, wird er hier nicht diskutiert. Der interessierte Leser konsultiere [Silberschatz/Galvin].

5.6 Entdeckung und Behebung von Verklemmungen

Um Verklemmungen zu entdecken, muss ein System darüber Buch führen, welcher Prozess auf welchen anderen wartet. Natürlich ist die Verwaltung des Ressourcenzuordnungsgraphen eine geeignete Möglichkeit, allerdings reicht auch ein einfacherer Graph, der die „wartet auf"-Beziehungen zwischen den Prozessen darstellt.

Ein Zyklus bedeutet eine Verklemmung, also muss das System regelmäßig Zyklentests durchführen. Dies bedeutet, je nach Test-Frequenz, einen nicht unerheblichen Aufwand. Ist ein Zyklus gefunden, stellt sich die in der Praxis äußerst problematische Frage, wie die Verklemmung zu bereinigen ist.

Prinzipiell muss das System „nur" den Zyklus durchbrechen, indem es einen beteiligten Prozess - oder sogar alle - beendet. Bei einem Compilerlauf mag dies unproblematisch sein, bei einer nicht ohne weiteres wiederholbaren kritischen Anwendung fatal. Die Entscheidung, welcher Prozess terminiert wird, ist also problematisch.

Statt der Terminierung ist auch eine Ressourcen-Wegnahme denkbar, die allerdings nur unter bestimmten Rahmenbedingungen funktioniert: Der betroffene Prozess muss blockiert und nach der späteren Wiederzuteilung der weggenommenen Ressourcen *im gleichen Zustand* wieder fortgesetzt werden können. In vielen Fällen wird es technisch

nicht machbar sein, den Zustand des Prozesses, inklusive des Zustands der weggenommenen Ressource, zu retten und wiederherzustellen. Auch hier stellt sich natürlich die Frage, welcher Prozess betroffen ist. Eine allgemeine Lösung existiert nicht.

Eine Alternative wäre die Verständigung eines Operators, der fallweise entscheidet, was zu tun ist.

Kapitel 6

Hauptspeicherverwaltung

6.1 Grundlagen

Ein Prozess muss während der Ausführung im Hauptspeicher stehen (zumindest teilweise). Im Multitasking-Sytem sind mehrere Prozesse gleichzeitig im Hauptspeicher, das Betriebssystem muss den Speicher entsprechend aufteilen.

6.1.1 Speicherhierarchien

Im allgemeinen Fall werden in einem Rechnersystem unterschiedliche Speichertechnologien eingesetzt, z.B.

Speicherart	**Geschwindigkeit**	**Größen-Beispiel**
CPU-integrierter Cache	sehr hoch	0,5 MB
externer Cache	hoch	4 MB
Hauptspeicher	mittel	1024 MB
Festplatte	langsam	5000 MB

Je kürzer die Zugriffszeiten bei einer bestimmten Speicherart sind, desto teurer ist dieser Speicher und desto knapper wird er bemessen sein. Es bietet sich daher prinzipiell an, möglichst die Daten, die am häufigsten benötigt werden, im schnell zugreifbaren Speicherbereich zu halten.

Dabei wird von häufig benötigten Daten entweder eine *Kopie* im schnelleren Speicher angefertigt (*Cache*-Prinzip), oder die Daten werden abhängig von der Zugriffsfrequenz in einen anderen Speicher *verlagert*.

Hauptspeicher war bislang eine knappe und teure Ressource. Deshalb werden Platten traditionell als Erweiterungsspeicher eingesetzt: Auf einen Festplattenbereich, dem *Swap*-Bereich, werden Prozesse (oder Teile davon) bei akutem Platzmangel temporär ausgelagert. In zukünftigen Rechnersystemen wird man vielleicht auf solche Auslage-

rungsmechanismen verzichten können, weil Hauptspeicher nicht mehr so teuer und damit ausreichend verfügbar sein wird.

6.1.2 Adressbindung

Die meisten Betriebssysteme erlauben es, Programme auszuführen, die an beliebigen Stellen im Hauptspeicher stehen. Die in einem Programm verwendeten Daten- und Instruktionsadressen, müssen Hauptspeicheradressen zugeordnet (**gebunden**) werden.

Phasen der Programmbearbeitung

Ein Quelltextmodul wird vom Compiler zunächst in ein Objektmodul übersetzt.

Den in einem Programmquelltext benutzten Variablen werden dabei Speicheradressen zugeordnet. Die Anweisungen werden in Maschineninstruktionsfolgen übersetzt. Den Maschineninstruktionen werden ebenfalls Speicheradressen zugeordnet.

Viele Sprachen unterstützen **getrennte Übersetzung** einzelner Quelltextmodule, die zum selben Programm gehören. Mehrere Objektmodule werden vom Binder (*Linker*) zu einem ausführbaren Programm zusammengesetzt. Das Programm wird als Programmdatei gespeichert. Aus den Modulanfang-bezogenen Adressen kann der Binder beim Zusammenfügen Programmanfang-bezogene Adressen berechnen. Auch diese Phase wird noch zur Übersetzungphase gerechnet.

Ein Beispiel zeigt Abbildung **??**.

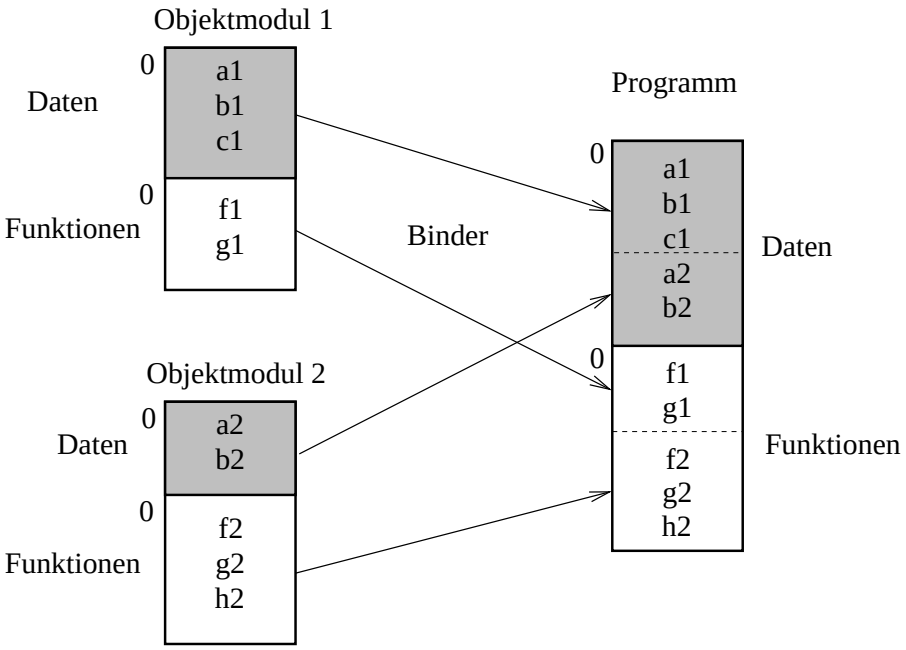


Abbildung 6.1: Module und Adressen

Im Beispiel werden zwei Quelltext-Dateien getrennt übersetzt. Der Compiler erzeugt je ein Objektmodul, in dem Daten und Maschinencode in zwei separaten Segmenten untergebracht sind. Die Bestandteile eines Segments werden jeweils mit Segment-bezogenen Adressen ab 0 adressiert, d.h. eine Adresse ist eine Byte-Distanz bezogen auf den Segment-Anfang. Bezogen auf das Datensegment des 1. Moduls hat *a1* die Adresse 0, *b1* vielleicht die Adresse 4 (abhängig von der Größe von *a1*) usw. Die erste Maschineninstruktion der Funktionen *f1* und *f2* haben in den Objektmodulen also auch die Adresse 0.

Das Zusammenfügen der Objektmodule geschieht im Beispiel segmentweise. Die im Programm verwendeten Adressen sind ebenfalls auf den Segment-Anfang bezogene Byte-Distanzen. Um diese zu berechnen, muss der Binder auf alle Adressen im Datensegment von Modul 2 die Größe des Datensegments von Modul 1 addieren. Das gleiche gilt für das Maschinencode-Segment.

Wir gehen in den nachfolgenden Abschnitten zunächst davon aus, dass ein Programm nur aus einem Segment besteht. In **??**, S. **??** wird dann die Segmentierung genauer behandelt.

Beim Programmaufruf wird aus der Programmdatei der Programmspeicher des Prozesses initialisiert. Den Vorgang bezeichnet man als „Laden“, den Zeitpunkt als **Ladezeit**.

Nach dem Laden des Programms, kann die Ausführung beginnen: Der Programmzähler des Prozessors wird mit der Adresse der ersten Instruktion des Programms initialisiert, die Kontrolle damit an das Programm übergeben. Wir sprechen von der **Ausführungszeit**.

Die Adresszuordnung oder **Adressbindung** kann zu unterschiedlichen Zeitpunkten erfolgen.

- **Übersetzungszeit:** Wenn schon zur Übersetzungszeit bekannt ist, wo im Hauptspeicher das Programm zur Ausführungszeit stehen wird, können Compiler und Binder **absolute** Adressen erzeugen (z.B. DOS `.COM`-Dateien).

- **Ladezeit:** Der Binder erzeugt **relozierbare** Adressen. Diese beziehen sich auf den Beginn des Programmspeichers (oder eines Teilbereichs). Zur Ladezeit wird festgelegt, wohin das Programm plaziert wird (*Ladeadresse*). Die relozierbaren Adressen werden dabei (durch Addition der Ladeadresse) in absolute Adressen konvertiert. Dazu muss die Programmdatei eine (von Compiler und Binder erzeugte) Relokations-Tabelle enthalten, in der die Positionen der relozierbaren Adressen vermerkt sind.

- **Ausführungszeit:** Wenn ein Prozess zur Ausführungszeit innerhalb des Hauptspeichers verlagert werden kann, muss die Adressbindung auf die Ausführungszeit verschoben werden. Dazu wird spezielle Hardware benötigt.

6.1.3 Dynamisches Laden

Eine Funktion eines Programms, die zur Ausführungszeit nicht aufgerufen wird, muss nicht im Hauptspeicher stehen. Dynamisches Laden ist eine Technik, bei der einzelne Funktionen erst bei Bedarf in den Hauptspeicher geladen werden.

Standard-Bibliotheken werden oft dynamisch geladen. Das Betriebssystem wird zur Realisierung eines dynamischen Ladekonzepts nicht benötigt.

6.1.4 Dynamisches Binden

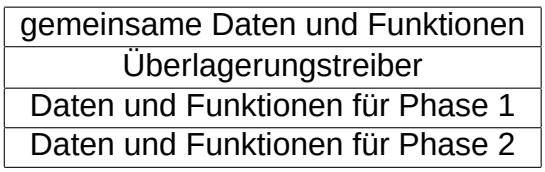
Früher wurden Standardfunktionen statisch gebunden: Aus einer Objektmodulbibliothek extrahiert der Binder die benötigte (compilierte) Funktion und fügte sie, wie andere Objektmodule auch, in das Programm ein. Häufig benutzte Standardfunktionen sind dann zur Ausführungszeit mehrfach im Hauptspeicher (als Bestandteile unterschiedlicher Programme).

Moderne Systeme unterstützen meist gemeinsam genutzte dynamisch gebundene Bibliotheken. Zur Ausführungszeit wird innerhalb des Programms beim ersten Aufruf einer Bibliotheksfunktion ein Platzhalter („Stub“) aktiviert. Dieser überprüft, ob die benötigte Bibliotheksfunktion (oder die gesamte Bibliothek) schon im Speicher steht. Wenn nicht, wird sie dynamisch geladen. Der Platzhalter wird dabei durch die Bibliotheksfunktion ersetzt, so dass beim nächsten Aufruf die Bibliotheksfunktion direkt aktiviert wird.

6.1.5 Überlagerungen („Overlays“)

Zu Zeiten, als der Hauptspeicher noch knapper bemessen war als heute, wurde in einfachen Betriebssystemen für allzu grosse Programme die Überlagerungstechnik verwendet. Die Grundidee: Der Programmierer zerlegt die Funktionalität eines Programms in zwei oder mehr nacheinander ausführbare Phasen, die weitgehend separate Datenstrukturen und Funktionen benutzen. Dann muss zur Ausführungszeit das Programm nicht komplett im Hauptspeicher stehen, sondern nur jeweils die Datenstrukturen und Funktionen, die in der aktuellen Verarbeitungsphase gebraucht werden.

Beispiel:



Das Programm überschreibt zur Ausführungszeit einen Teil von sich selbst, der nicht mehr benötigt wird, durch einen dynamisch nachgeladenen anderen Programmteil. Im

Beispiel werden zunächst die ersten 3 Speicherbereiche geladen. Nach Ende der 1. Phase tauscht das Programm den 3. gegen den 4. Teil aus.

Der Austausch wird vom Programm selbst ohne Unterstüzung durch das Betriebssystem organisiert. Der Austausch geschieht innerhalb eines speicherresidenten Teils des Programms, dem Überlagerungstreiber.

Die austauschbaren Teile nennt man Überlagerungen („Overlays“). Die Aufteilung des Programms und die Programmierung von Überlagerungen ist oft schwierig und fehlerträchtig. Die Technik hat inzwischen an Bedeutung verloren.

6.1.6 Logischer und physikalischer Adressraum

Die in einem Prozess verwendeten Adressen, d.h. die Adressen, die der Prozessor verarbeitet, bilden den **logischen Adressraum**. Die Adressen, mit denen der Hauptspeicher tatsächlich adressiert wird, nennen wir **physikalische Adressen**.

Bei absoluter Adressierung oder bei Ladezeit-Bindung sind logische und physikalische Adressen identisch.

Bei Adressbindung zur Ausführungszeit dagegen nicht: Eine vom Prozessor angesprochene logische Adresse wird von einer Addressumrechnungshardware (MMU – „memory management unit“) auf eine andere physikalische Adresse abgebildet.

Man nennt die logischen Adressen auch **virtuelle** Adressen.

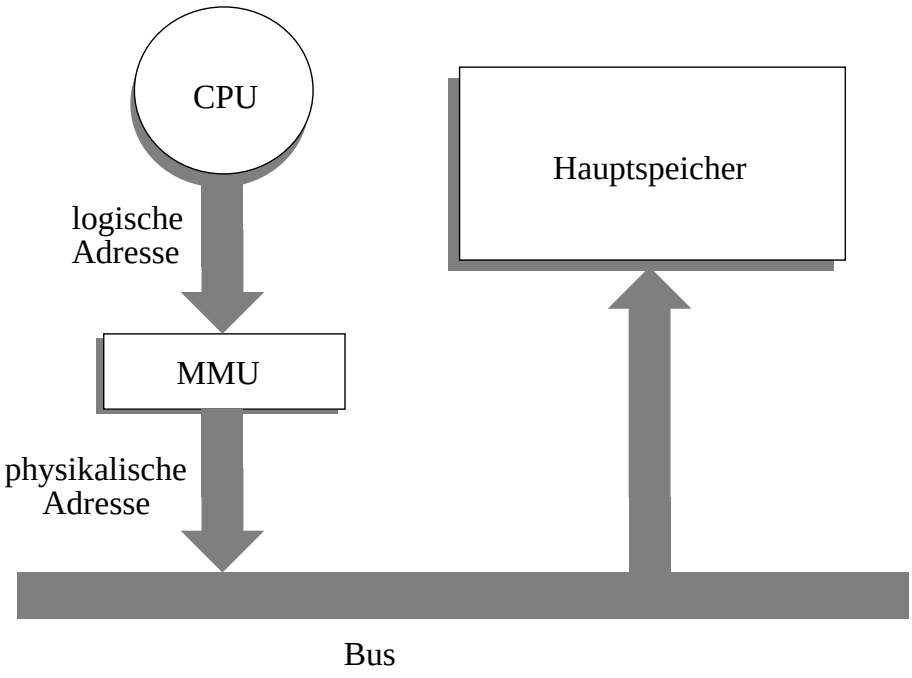


Abbildung 6.2: Virtuelle Adressen

Die Abbildung ist hardwareabhängig. Eine sehr einfache Abbildungsmethode wäre die

Addition des Inhalts eines *Basisregisters* zur logischen Adresse, die sich auf den Anfang des Programmspeichers bezieht.

Angenommen, das Programm wird bei der physikalischen Startadresse 17000 in den Hauptspeicher plaziert. Die Startadresse wird beim Laden in das Basisregister geschrieben. Wenn der Prozess auf die logische Adresse 236 zugreift, erzeugt die MMU daraus einen Zugriff auf die physikalische Adresse 17236.

Man sieht leicht, dass sich bei dynamischer Adressbindung der Prozess im Speicher während der Ausführungszeit verschieben lässt. Genau wie beim Laden muss dabei nur das Basisregister auf die neue physikalische Startadresse gesetzt werden.

6.2 Einprozess-Modelle

Am einfachsten ist eine Speicherverwaltung für Betriebssysteme ohne Multitasking zu realisieren. Im Hauptspeicher steht das Betriebssystem und ein einziges Anwenderprogramm. Meist ist das Betriebssystem im unteren Adressbereich untergebracht:

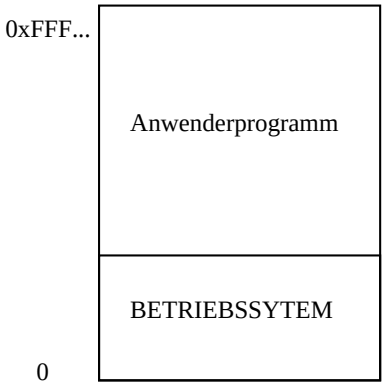


Abbildung 6.3: Einfaches Speichermodell

Bei diesem Modell können feste Adressen verwendet werden, die zur Übersetzungszeit schon bekannt sind. Speicherschutzmechanismen sind nicht nötig.

Da das Anwenderprogramm durch E/A-Aktivitäten immer wieder blockiert wird, ist der Prozessor bei diesem Modell i.d.R. nur schlecht ausgelastet. Moderne Betriebssysteme unterstützen daher durchweg Multitasking.

6.3 Multitasking-Modelle mit speicherresidenten Programmen

Bei Multitasking-Systemen stehen mehrere Anwenderprogramme gleichzeitig im Hauptspeicher. Da die Prozessorkontrolle bei Blockieren des ausführenden Prozes-

ses an einen anderen Prozess übergeben werden kann, ist die Prozessor-Auslastung und damit die Systemgeschwindigkeit durchweg besser als im Ein-Prozess-Modell.

Das Betriebssystem muss die Aufteilung des Hauptspeichers organisieren. Wir betrachten zunächst Techniken, bei denen die Programme während der gesamten Ausführungszeit komplett im Hauptspeicher stehen („speicherresidente“ Programme).

6.3.1 Feste Partitionierung

Bei fester Partitionierung wird der Hauptspeicher statisch in Partitionen verschiedener Größen aufgeteilt. Beim Programmaufruf wird dem neuen Prozess dann eine genügend große Partition für die gesamte Ausführungszeit fest zugewiesen.

Wenn das Betriebssystem mehrere Programmaufrufe zu bearbeiten hat, legt es eine Auftrags-Warteschlange an. Einen Auftrag bezeichnet man in diesem Zusammenhang als „Job“. Das Betriebssystem kann pro Partition eine Warteschlange mit den Jobs anlegen, die in die Partition passen (A). Alternativ kann eine einzige Warteschlange mit allen Jobs angelegt werden (B).

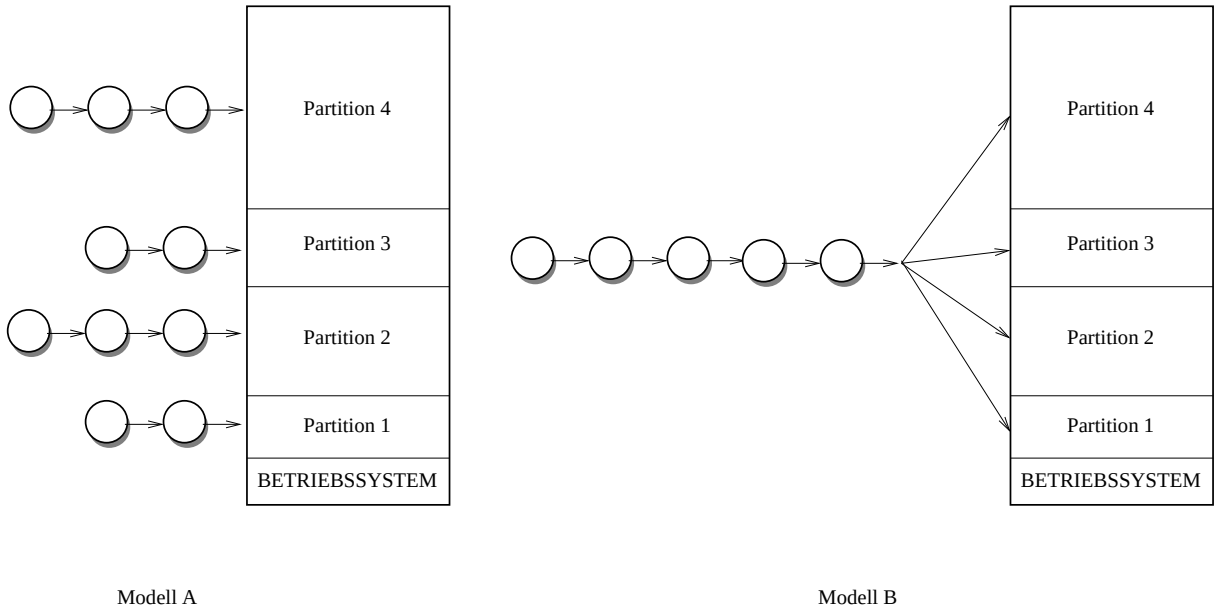


Abbildung 6.4: Feste Partitionierung

Bei Modell A kann es zu der unerwünschten Situation kommen, dass mehrere kleine Jobs auf das Freiwerden derselben kleinen Partition warten, während eine größere Partition leersteht.

Bei Modell B holt das Betriebssystem beim Freiwerden einer Partition den ersten Job aus der Warteschlange, der in die freigewordene Partition passt. Dies führt zur **Speicherverschwendung**, wenn dadurch eine sehr große Partition mit einem sehr kleinen Job für längere Zeit belegt wird. Um dies zu vermeiden, kann man immer den größten

Job aus der Warteschlange nehmen, der in die Partition passt. Dies ist wiederum **unfair** gegenüber kleinen Jobs. Sparsamer Umgang mit Ressourcen sollte schließlich belohnt, nicht bestraft werden. Um dem abzuhelfen, sollten immer genügend viele kleine Partitionen zur Verfügung stehen.

Feste Partitionierung des Hauptspeichers wurde in früheren Großrechnerbetriebssystemen (z.B. OS/360) verwendet, die Aufteilung wurde manuell durch den Operator durchgeführt.

6.3.2 Variable Partitionierung

Eine bessere Speicherausnutzung kann man mit variablen Partitionen erreichen:

- Die Anzahl und Größe der Partitionen wird den Anforderungen der Prozesse dynamisch angepasst.

Wenn der Hauptspeicher leer ist, werden zunächst zusammenhängende „maßgeschneiderte“ Partitionen für die Prozesse reserviert. Wenn ein Prozess terminiert, wird seine Partition frei und kann dem nächsten Prozess aus der Warteschlange zugeteilt werden. Dabei wird der Speicher im Laufe der Zeit mehr oder weniger „zerstückelt“, d.h. es entstehen überall freie Bereiche, die teilweise zu klein sind, um Prozesse hineinladen zu können. Dies nennt man **Fragmentierung** des Speichers.

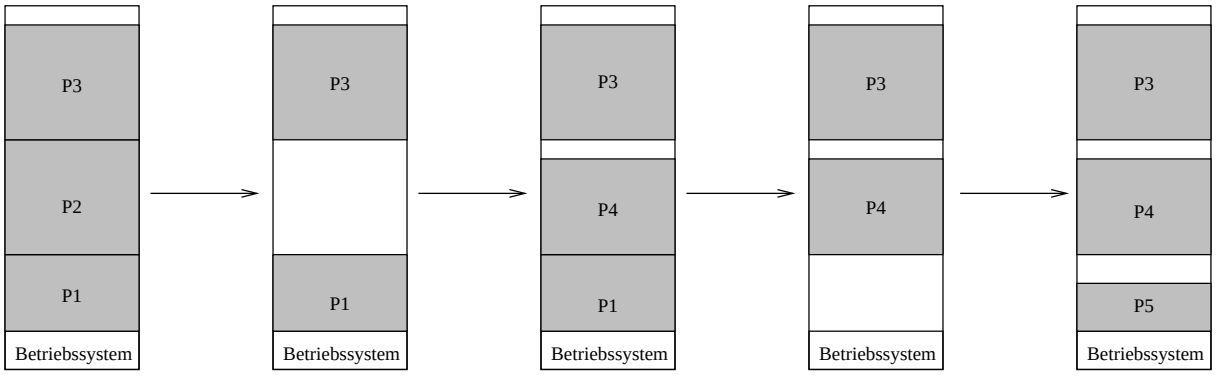


Abbildung 6.5: Fragmentierung bei variablen Partitionen

Falls für einen neuen Prozess Speicher benötigt wird, muss das Betriebssystem einen geeigneten freien Bereich suchen. Für die Buchführung über belegte und freie Bereiche bieten sich mehrere Methoden an.

Partitionsverwaltung mit verketteten Listen

Das Betriebssystem kann die belegten Partitionen und die freien Hauptspeicherbereiche in je einer doppelt verketteten Liste verwalten. Jedes Listenelement enthält Anfangsadresse und Größe des Speicherbereichs.

Bei Aufruf eines Programms wird aus der **Freiliste** (Liste der freien Bereiche) ein geeigneter Bereich gewählt:

- Die „**first fit**“-Strategie nimmt aus der Freiliste den **ersten passenden** Bereich. Dabei kann die Freiliste jedesmal von vorne durchsucht werden oder von dem zuletzt besuchten Listenelement ausgehend (dies nennt man auch „**next fit**“).
- Die „**best fit**“-Strategie wählt aus der Freiliste den **kleinsten passenden** Bereich.
- Die „**worst fit**“-Strategie selektiert den **größten freien** Bereich.

  Damit will man erreichen, dass die verbleibende Lücke noch gross genug für eine sinnvolle Nutzung ist.
- Falls von vorneherein bekannt ist, dass bestimmte Bereichsgrößen häufig benötigt werden, kann das Betriebssystem auch mehrere Freilisten halten, in denen jeweils Bereiche einer Größe verkettet sind. Bei Speicherbedarf muß dann nur die zur Größe passende Liste durchsucht werden. Dies nennt man „**quick fit**“.

Die Strategien unterscheiden sich vom Zeitaufwand und von dem durchschnittlichen Grad der Speicherausnutzung. Bei „best“- und „worst“-fit muss jedesmal die ganze Liste durchsucht werden oder die Freiliste immer nach Größe sortiert sein. Die „first fit“-Bestimmung ist schneller.

Den Speichernutzungsgrad bestimmt man durch Simulation. Dabei zeigt sich, dass „first fit“ und „best fit“ in dieser Hinsicht gleich gute Ergebnisse bringen, während „worst fit“ schlechter ist.

Bei allen Listen-basierten Verfahren muss bei der Freigabe eines Bereichs geprüft werden, ob dieser unmittelbar an benachbarte freie Bereiche angrenzt. In diesem Fall muss der neu freigegebene Bereich und der oder die freien Nachbarbereiche zu einem einzigen freien Bereich verschmolzen werden. Der Zeitaufwand für diese Verschmelzung ist ggf. deutlich höher als der Zeitaufwand für die Suche eines passenden Bereichs.

Das Buddy-System

Das Buddy-System (Knuth/Knowlton) vergibt Speicherplatz in Blockgrößen, die jeweils Zweierpotenzen sind. Damit kann die Verschmelzung bei Blockfreigabe sehr schnell erfolgen.

Für jede Blockgröße (2, 4, 8, 16, . . .) wird eine Freiliste verwaltet. Nehmen wir an, 1024 K Speicher sind vorhanden. Dann sind anfangs alle Listen leer, nur in der 1024 K-Liste ist ein einziger Block (der gesamte Hauptspeicher). Bei Speicheranforderung von N Bytes wird die nächste Zweierpotenz k bestimmt und aus der entsprechenden Liste ein passender Block zugeteilt.

Ist keiner vorhanden, werden schrittweise die höheren Blockgrößen geprüft, bis ein freier Block gefunden ist. Dieser wird durch (iterierte) Halbierung soweit zerlegt bis ein Block der benötigten Größe *k* entsteht.

Bei der Freigabe muss die Verschmelzung mit den benachbarten Blöcken geprüft werden. Zu verschmelzende Nachbarblöcke haben immer dieselbe Größe. Damit sind die Nachbarn schnell gefunden. Der Verschmelzungsprozess muss bei Erfolg für die höheren Blockgrößen iteriert werden.

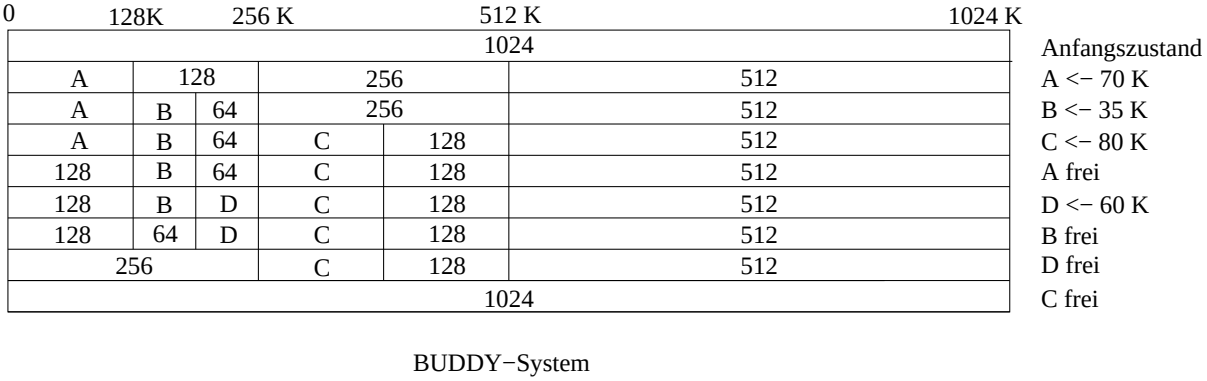


Abbildung 6.6: Buddy-System

Der Buddy-Algorithmus ist schnell, aber die interne Fragmentierung (siehe unten) der Blöcke führt zu einer schlechten Speichernutzung.

Externe und interne Fragmentierung

Die Zerstückelung des Hauptspeichers in freie Bereiche und Programme wird als externe Fragmentierung bezeichnet. Über diese Fragmente wird Buch geführt.

Interne Fragmentierung entsteht, wenn die Zuteilungseinheiten nicht Byte, sondern Blöcke einer bestimmten Größe sind. Der vom Programm benötigte Hauptspeicherbereich kann nicht exakt zugeteilt werden, stattdessen wird auf die nächste Blockgrenze aufgerundet. Der letzte einem Programm zugeteilte Block wird also i.d.R. nicht vollständig genutzt, der ungenutzte Bereich ist verschwendet. Da dieser Bereich nicht als Lücke in der Freispeicherliste auftaucht, spricht man von interner Fragmentierung.

Je größer die Blöcke sind, desto höher ist das Maß der internen Fragmentierung.

Bitmap-basierte Verwaltung

Eine andere Technik der Buchführung über freie Speicherbereiche verwendet Zuteilungseinheiten fester Größe in Verbindung mit Bitmaps. Die Blockgröße kann je nach Anwendungsbereich von wenigen Byte bis zu mehreren Kilobyte variieren.

In einer Bitmap wird für jeden dieser Blöcke ein „Frei“-Bit geführt, das für freie Blöcke den Wert 0 hat, ansonsten 1.

Falls ein Speicherbereich der Größe *N* Byte gesucht wird, wird die benötigte Anzahl von Blöcken *k* bestimmt und in der Bitmap nach einer Folge von *k* aufeinanderfolgenden Nullen gesucht.

Die Größe der Bitmap und damit auch die Geschwindigkeit der Lückensuche ist natürlich von der Blockgröße abhängig. Bei großen Blöcken wird die Bitmap kleiner und die Suche schneller. Dafür ist die Speichernutzung nicht optimal, denn der jeweils letzte einem Prozess zugeteilte Block wird im Schnitt nur zu 50% genutzt werden.

Speicherkompaktierung

Ein andere Idee zur Lösung des Fragmentierungsproblems ist das Zusammenschieben der belegten Bereiche im Hauptspeicher, so dass statt vieler verteilter Lücken immer nur ein einziger großer freier Bereich am oberen Ende des Hauptspeicher existiert. Dem steht in der Praxis entgegen, dass die dazu notwendige Reorganisation der Hauptspeicherbelegung zu lange dauert. Allerdings gab es Maschinen, die mit spezieller Kompaktierungshardware ausgestattet waren (CDC Cyber – unser ehemaliger Hochschulrechner, kompaktierte 40MB/Sekunde).

Man beachte, dass die oben vorgestellten Verfahren auch andere Anwendungen haben. Beispielsweise entstehen bei der Verwaltung des „Swapspace“ auf der Festplatte oder bei der Bearbeitung von *malloc*-oder *new*-Aufrufen im C/C++-Programm ähnliche Probleme wie bei der Hauptspeicherzuteilung für neue Prozesse.

Adressbindung und Zugriffsschutz

Bei Partitionierung des Hauptspeichers wird die Anfangsadresse eines Programms erst zur Ladezeit bekannt. Damit ist auch die Adressbindung erst zur Ladezeit möglich. Falls die Hardware eine effiziente dynamische Adressbindung unterstützt, kann das Programm auch zur Laufzeit mit logischen Adressen arbeiten. Typischerweise wird dabei bei jedem Hauptspeicherzugriff auf die logische Adresse der Inhalt eines *Basisregisters* (oder Relokationsregisters) addiert, welches die physikalische Ladeadresse des Programms enthält.

Für den Zugriffsschutz ist es notwendig, bei jedem Hauptspeicherzugriff zu überprüfen, ob die Adresse innerhalb des Bereichs liegt, der dem Prozess zugewiesen wurde. Diese Überprüfung muss aus Geschwindigkeitsgründen mit Hardwareunterstützung durchgeführt werden. Dazu dient ein *Grenzregister*, das die Grösse des Prozesses enthält. Wenn die logische Adresse größer ist als der Grenzregisterinhalt, liegt ein unzulässiger Speicherzugriff vor.

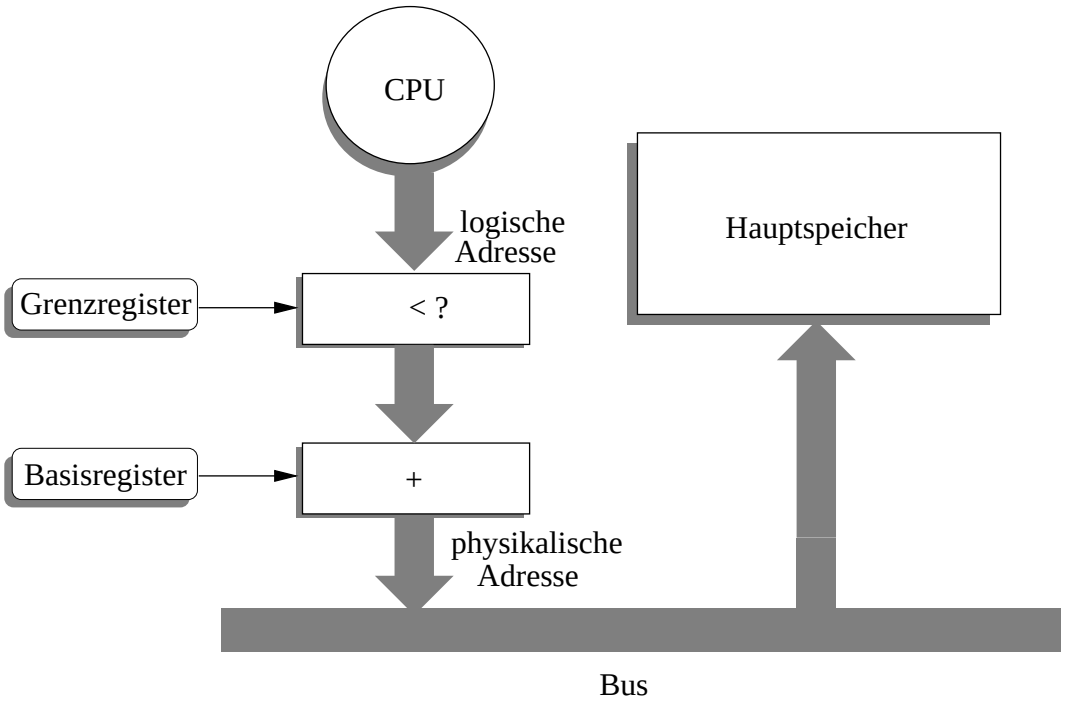


Abbildung 6.7: Zugriffsschutz bei virtuellen Adressen

6.4 Swapping

Früher wurden Programme ausschliesslich im Stapelverarbeitungsbetrieb (**??**, S. **??**) ausgeführt. Solange im Hauptspeicher genügend Prozesse Platz hatten, um den Prozessor voll auszulasten, war es kein Problem, dass einige Aufträge längere Zeit in einer Auftragswarteschlange auf Speicherzuteilung warten mussten.

Swapping ist eine Technik, die mit der Verbreitung des Timesharing notwendig wurde, denn interaktive Programme kann man schlecht vor der Ausführung minuten- oder stundenlang in einer Warteschlange stehen lassen. Der verfügbare Hauptspeicher war allerdings i.d.R. zu klein, um allen interaktiven Prozessen gleichzeitig Platz zu bieten. Also wurden bei Platzmangel Prozesse temporär aus dem Hauptspeicher auf einen *Swapbereich* einer Festplatte ausgelagert.

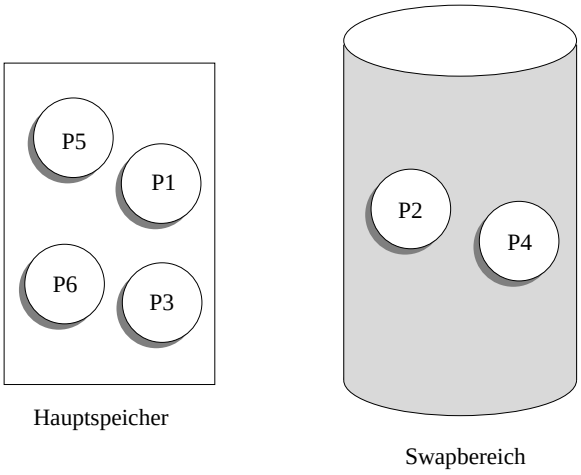
Swapping lässt sich am besten mit variabler Hauptspeicher-Partitionierung kombinieren. Bei Speichermangel werden Prozesse ausgelagert, die freigewordenen Speicherbereiche werden für andere Prozesse genutzt. Dies können neue Prozesse oder schon länger ausgelagerte Programme sein, die dann vom Swapbereich zurückkopiert werden. Ein Beispiel zeigt die nachstehende Abbildung.

Für die Verwaltung des Swapbereichs werden die gleichen Algorithmen wie für die Hauptspeichersverwaltung verwendet (Freilisten, Bitmaps). Um das Swapping zu beschleunigen, reservieren manche Systeme beim Programmstart schon einen festen Platz für den Prozess im Swapbereich. Andere bestimmen dagegen erst beim Ausla-

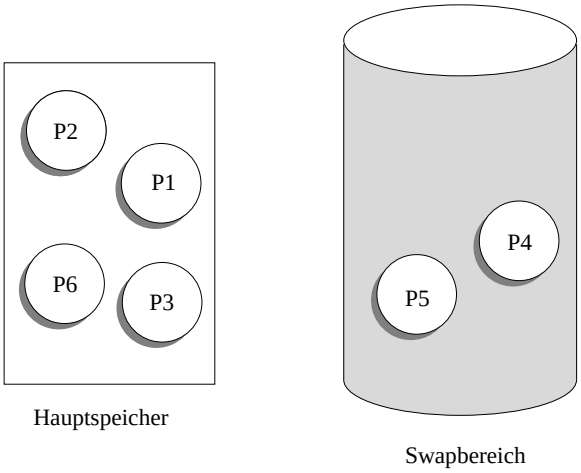
gern eines Prozesses, wohin dieser kopiert wird.

Ein Problem beim Swapping können E/A-Operationen sein. Falls ein Prozess beispielsweise auf Daten von der Festplatte wartet und diese die Daten direkt in den Adressbereich des Prozesses kopiert, ist es nicht möglich, den wartenden Prozess auszulagern. Alternativ können Eingabedaten zunächst in den Adressbereich des Betriebssystems kopiert und bei Wiedereinlagerung des Prozesses dann an diesen weitergereicht werden.

Swapping in der ursprünglichen Form ist heute nicht mehr gebräuchlich. Modifizierte Varianten der Technik, insbesondere in Kombination mit dem „Workingset“-Konzept (**??**, S. **??**), sind allerdings sehr verbreitet (z.B. bei den meisten UNIX-Systemen).



P5 wird ausgelagert, P2 eingelagert



P3 und P6 werden ausgelagert, um ein neues Programm P7 zu laden

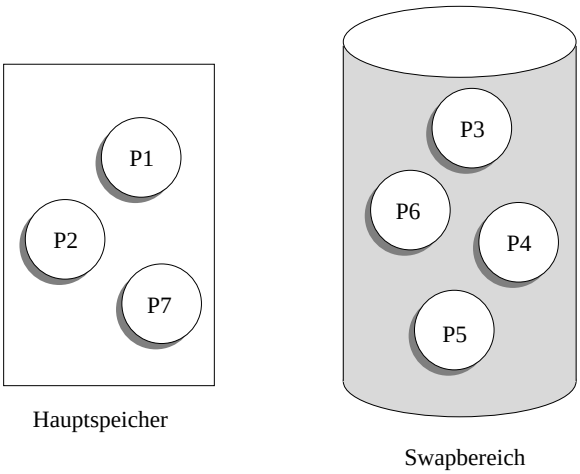


Abbildung 6.8: Swapping-Szenario

6.5 Paging

6.5.1 Grundlagen

Paging basiert auf dynamischer Adressbindung, setzt also eine MMU voraus, die logische Adressen effizient in physikalische Adressen umrechnet.

Beim Paging wird der Hauptspeicher in Blöcke fester Größe eingeteilt, die **„Seitenrahmen“** heißen. Der logische Adressraum eines Prozesses wird in **„Seiten“** aufgeteilt, die genauso groß wie die Seitenrahmen des physikalischen Speichers sind. Die Adressierung und die Hauptspeicher-Zuteilung erfolgt immer Seiten-bezogen.

Die Seitengröße ist eine Zweierpotenz, typischerweise 2 KB oder und 4 KB. Jede Adresse wird aufgeteilt in eine Seitenadresse und eine Byte-Distanz (**„Displacement“**), die sich auf den Seitenanfang bezieht.

Beispiel: 32 Bit-Adressen mit 4 KB Seitengröße

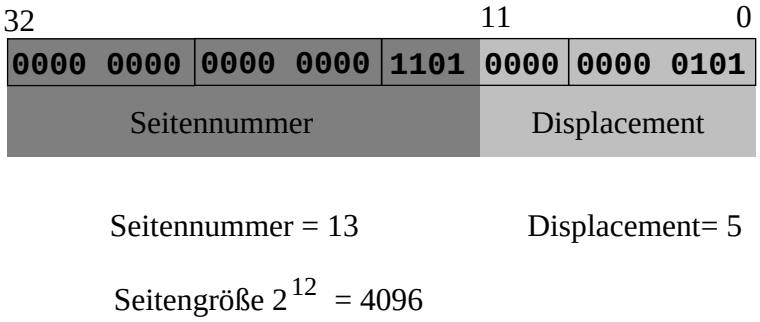


Abbildung 6.9: Seitennummer und Displacement

Jeder Seite eines Prozesses wird dynamisch ein (beliebiger!) Seitenrahmen des Hauptspeichers zugeordnet. Die Zuordnung wird mittels einer **Seitentabelle** verwaltet. Der Tabellenzugriff wird durch die MMU beschleunigt.

Das Displacement ist in der logischen und der physikalischen Adresse identisch.

Betrachten wir als Beispiel einen 20 KB grossen Prozess bei einer Seitengröße von 4 KB. Der logische Adressraum besteht aus 5 Seiten, die Seitentabelle des Prozesses hat also 5 Einträge.

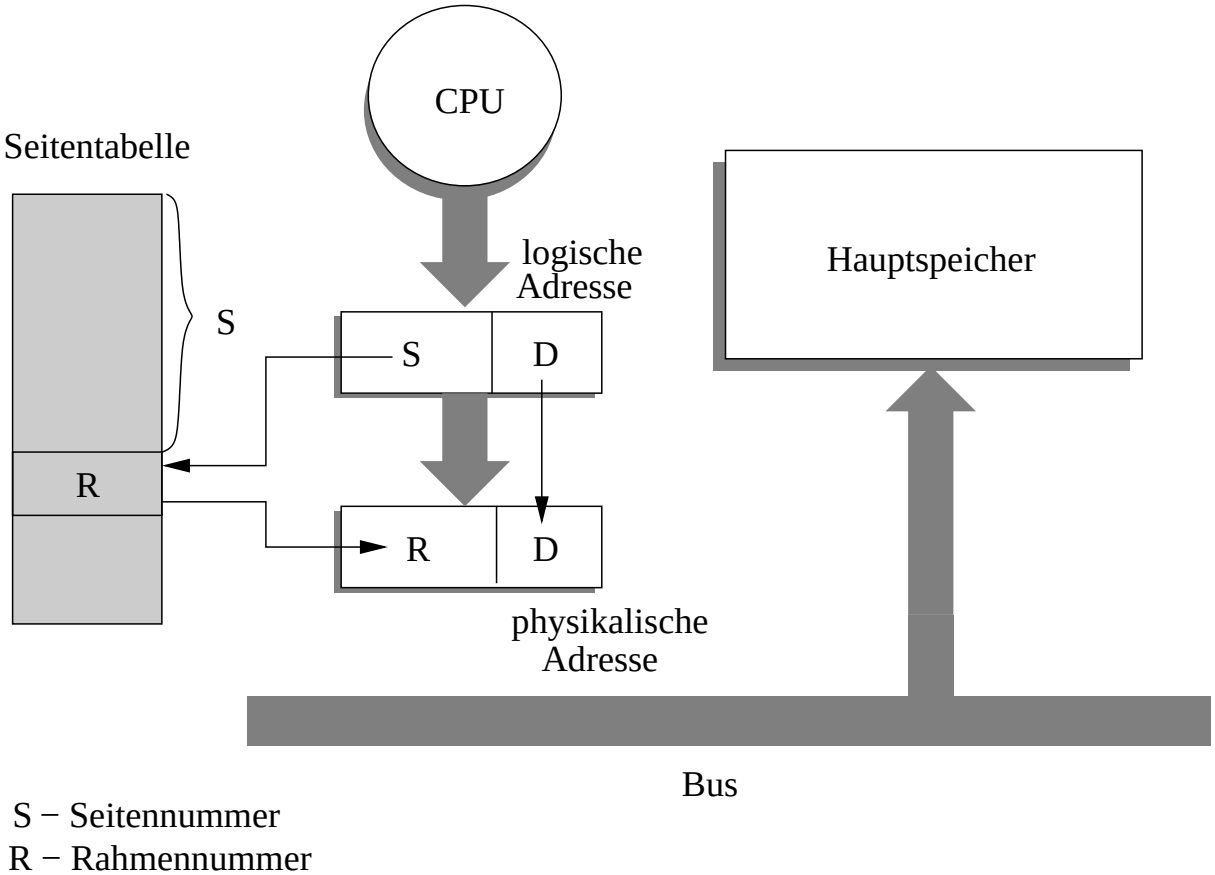


Abbildung 6.10: Adressumrechnung mit Seitentabelle

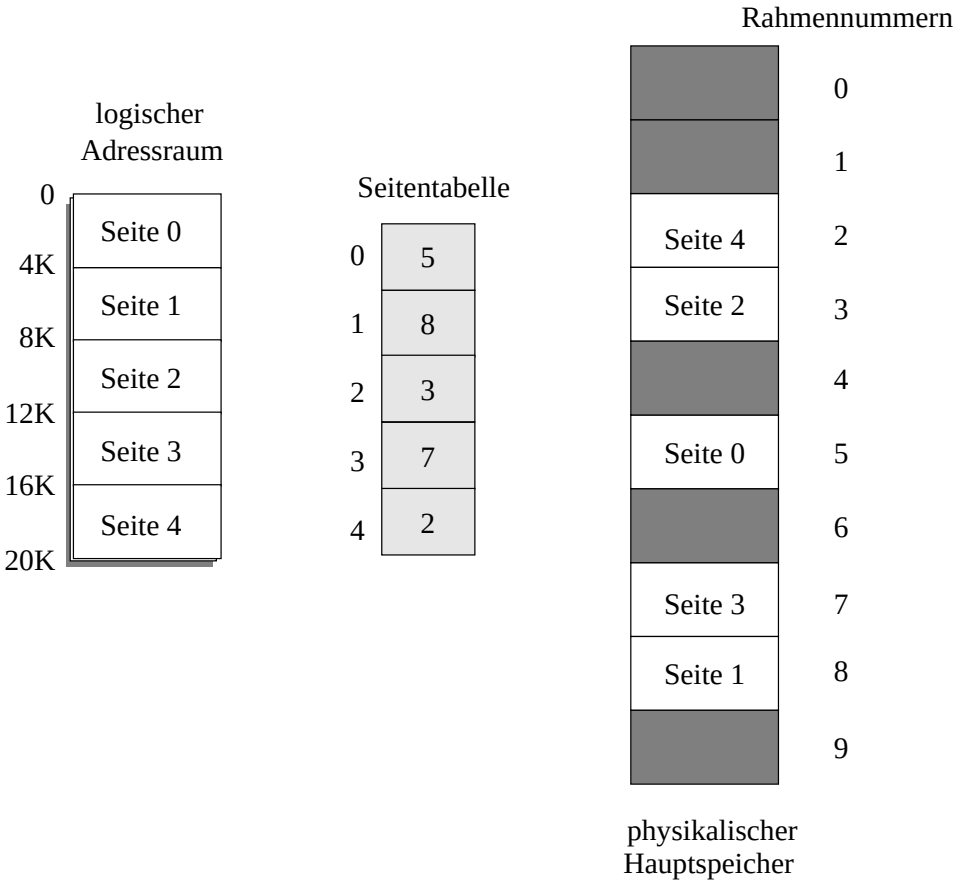


Abbildung 6.11: Seitentabellen-Beispiel

In einem einfachen Modell verwaltet das Betriebssystem also pro Prozess eine Seitentabelle, in der zu allen Seiten des Prozesses die momentan zugeordneten Rahmen verzeichnet sind.

Neben den Seitentabellen verwaltet das Betriebssystem die Seitenrahmen in einer **Rahmentabelle**. Für jeden Seitenrahmen gibt es darin einen Eintrag mit Zustandsinformation über den Rahmen, z.B.

- Rahmen frei oder belegt
- welchem Prozess zugeordnet ?
- Zugriffshäufigkeit

Die Verwendung der Rahmentabelle wird später erläutert.

Bei Paging-Sytemen differiert die Benutzer-Sicht auf den Hauptspeicher stark von der tatsächlichen Aufteilung. Der Benutzer sieht das Programm als einen einzigen zusammenhängenden, bei Adresse 0 beginnenden Speicherbereich. Diese Sicht hat nicht nur der Programmierer. Compiler, Linker und der Prozessor arbeiten ausschließlich mit logischen Adressen.

Sucht man dagegen einen Prozess im realen Hauptspeicher, so findet man ihn in Form beliebig auf den verfügbaren Realspeicher verstreuter Seitenrahmen wieder.

6.5.2 Auslagerung von Seiten, Seitenfehler

Paging erlaubt die Ausführung von Prozessen, die nicht komplett im Hauptspeicher stehen. Betrachtet man die Bearbeitung eines Maschinenbefehls, der ein Byte von einer Adresse zu einer anderen Adresse kopiert, so kann dieser Befehl ausgeführt werden, wenn 3 Speicherseiten verfügbar sind:

- die Seite mit dem Maschinenbefehl
- die Seite, die das zu kopierende Byte enthält
- die Seite, die die Zieladresse enthält

Somit kann man bei Hauptspeichermangel einen Teil der Seiten eines Prozesses auslagern, ohne damit die Ausführbarkeit des Prozesses zu beeinträchtigen. Allerdings muss Vorsorge für den Fall getroffen werden, dass der ausführende Prozess auf eine ausgelagerte Seite zugreift. Spezielle Hardware-Unterstützung ist erforderlich: Die zugreifende Instruktion muss zunächst abgebrochen und die benötigte Seite in den Hauptspeicher zurückkopiert werden. Der Prozess wird solange blockiert. Man spricht von einem **Seitenfehler** (engl.: „page fault“). Anschliessend kann die Instruktion, die vorher den Seitenfehler verursacht hat, wiederholt werden, diesmal erfolgreich.

Die Wiederholbarkeit von Instruktionen nach Seitenfehlern ist also eine notwendige, nicht selbstverständliche Hardwareeigenschaft.

Buchführung über die ausgelagerten Seiten erfolgt zweckmässigerweise auch in der Seitentabelle des Prozesses. Hier wird sowohl der Zustand der Seite vermerkt (eingelagert, ausgelagert), als auch die Auslagerungsadresse abgespeichert.

Bei einem Seitenfehler wird Platz für die einzulagernde Seite benötigt. Das Betriebssystem verwaltet einen Pool freier Rahmen in der Rahmen-Freiliste. Falls der Speicher knapp wird, d.h. die Rahmen-Freiliste eine gewisse Größe unterschreitet, wird durch Auslagerungen Platz geschaffen. Auslagerungsstrategien werden in **??**, S. **??** behandelt.

6.5.3 Seitentabellen-Hardware

Der Zugriff auf die Seitentabellen ist ohne Hardwareunterstützung zu langsam. Unterschiedliche Formen der Hardwareunterstützung sind möglich:

- Im einfachsten Fall gibt es eine Reihe von Registern, die als Seitentabelle dienen. Diese werden zusammen mit den anderen Registern beim Kontextwechsel in den Prozessdeskriptor (Prozesstabelleneintrag) übertragen bzw. aus dem Prozessdeskriptor restauriert.

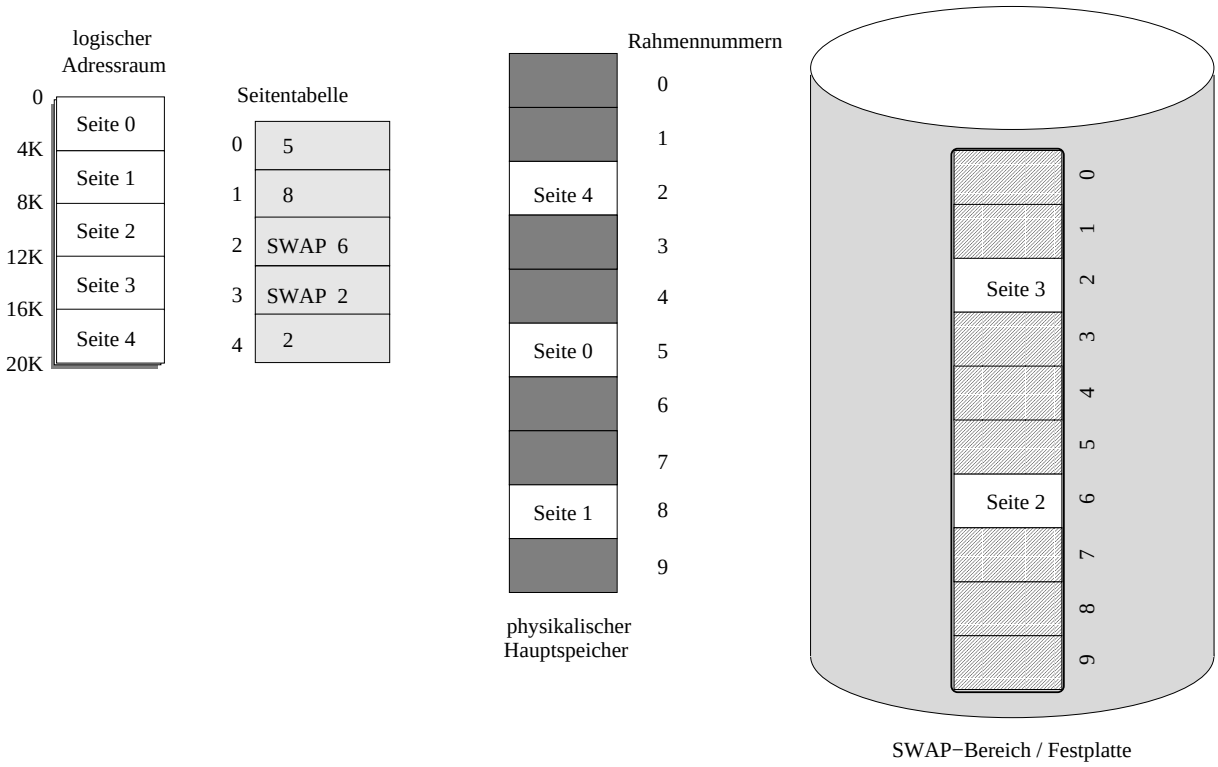


Abbildung 6.12: Buchführung über ausgelagerte Seiten

Dies funktioniert nur bei kleinen Seitentabellen (PDP-11). Bei modernen Rechnern können die Seitentabellen aber sehr groß werden.

- Die Seitentabelle kann im Hauptspeicher gehalten werden. Ein Register enthält die Adresse der Seitentabelle. Beim Kontextwechsel muss nur das Register aktualisiert werden, dies geht sehr schnell.

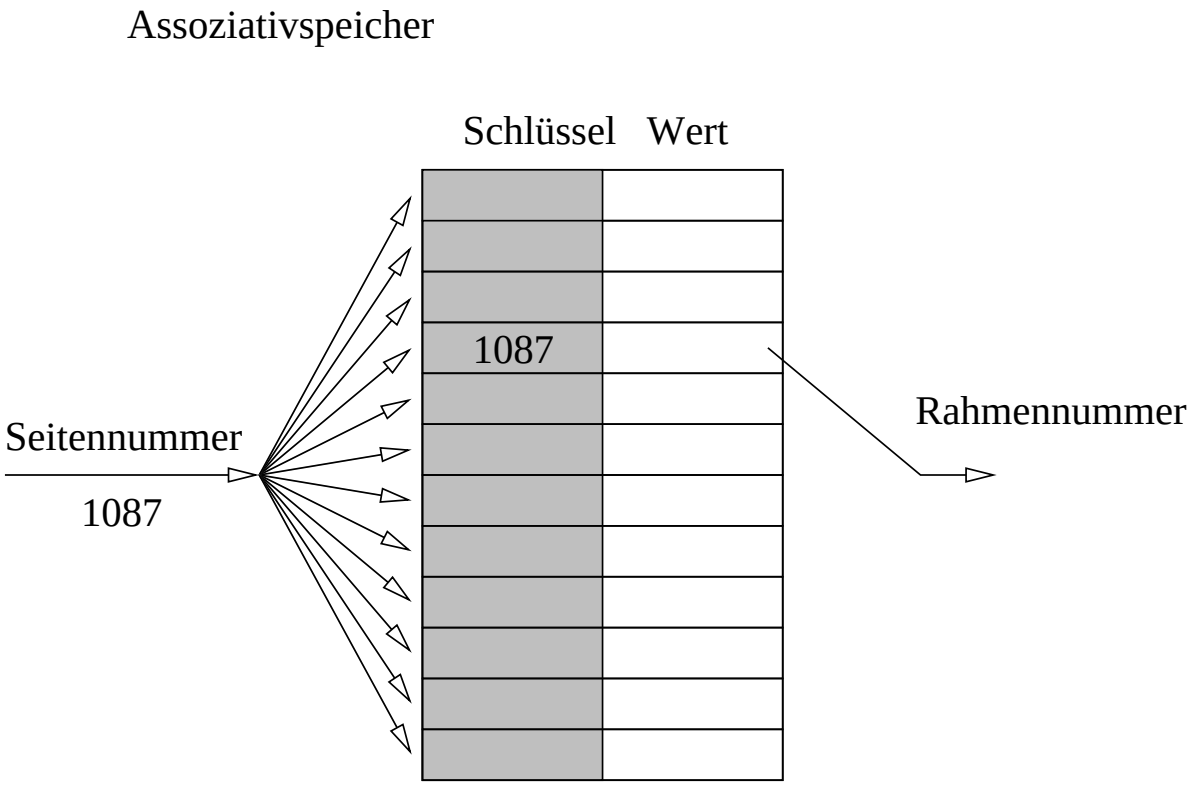
  Die Ausführung eines Prozesses ist jedoch zu langsam: Aus jedem Hauptspeicherzugriff werden zwei: Der erste auf die Seitentabelle, der zweite auf den darin verzeichneten Seitenrahmen. Wenn die Seitentabellen groß und zahlreich sind, kann man sie auch nicht speicherresidend machen. Dann unterliegen die Seitentabellen selbst dem Paging und es können bei einem Hauptspeicherzugriff gleich zwei Seitenfehler auftreten.

  In der Praxis wäre Swapping wahrscheinlich schneller.

- Die Seitentabelle wird im Hauptspeicher gehalten. Um den Zugriff zu beschleunigen, werden die am häufigsten benötigten Einträge in einem Cache gepuffert. Der Cache-Bereich ist als superschneller Speicher realisiert, **Assoziativ-Speicher** oder „**translation look-aside buffer**“ (TLB) genannt.

  Ein Register des Assoziativ-Speichers besteht aus Schlüssel und Wert. Bei Eingabe eines Schlüssels *S* in den Assoziativ-Speicher wird parallel in allen Registern eine Schlüsselvergleich durchgeführt. Das Register, dessen Schlüsselfeld

mit *S* übereinstimmt, liefert den zugehörigen Wert. Als Schlüssel wird die Seitennummer verwendet, der Wert ist die Rahmennummer.



Falls der Assoziativspeicher den Schlüssel nicht enthält, wird die Rahmennummer aus der Seitentabelle ermittelt und in den Assoziativspeicher anstelle des am längsten nicht mehr zugegriffenen Schlüssels eingetragen. In der Praxis erzielt man schon bei Assoziativspeichern mit 32 Registern eine Trefferrate von über 90 %, d.h. 9 von 10 Hauptspeicherzugriffen können ohne Zugriff auf die Seitentabelle durchgeführt werden. (Intel 80486: 32 Register, 98% Trefferrate laut Werbung)

Leider ist der Kontextwechsel aufwändig: Der TLB wird in der Regel komplett gelöscht. Die Alternative, den TLB-Zustand zu retten und später wiederherzustellen, ist auch nicht besonders schnell.

6.5.4 Mehrstufige Seitentabellen

Bei modernen Betriebssystemen ist der logische Adressraum eines Prozesses mehrere GB groß. Bei einer Adressraumgrösse von 4 GB (32 Bit-Adressen) und einer Seitengröße von 2 KB besteht der Adressraum immerhin aus über 2 Millionen Seiten. Wenn jeder Seitentabellen-Eintrag 4 Byte groß ist, wird die Seitentabelle eines Prozesses 8 MB benötigen.

Nun sind natürlich die Prozesse nicht so gross, meist wird nur ein kleiner Bruchteil des Adressraums wirklich genutzt. Wenn dies ein zusammenhängender Bereich ab Adresse 0 ist, muss nur der Teil der Seitentabelle verwaltet werden, der der Prozessgrösse entspricht.

Allerdings hat es einige Vorteile, wenn die Bereiche nicht zusammenhängen müssen. Ein Beispiel ist die Plazierung von Stack und Heap eines C-Programms: Beide Bereiche können unvorhersehbar wachsen. Man plaziert sie so, dass sie aufeinander zu wachsen (**??**, S. **??**). Wenn Sie im Adressraum zu eng plaziert werden, kann es zu einem Stack-/Heap-Überlauf kommen. Wenn man sehr viel Platz dazwischen läßt, wird die Seitentabelle zu groß.

Alles in allem tendieren moderne Systeme zu großen Adressräumen, von denen nur kleine aber unzusammenhängende Teilbereiche genutzt werden. Damit ist die einfache Lösung mit einer Seitentabelle pro Prozess nicht mehr praktikabel, man verwendet mehrstufiges Paging, invertierte Seitentabellen oder Paging in Verbindung mit Segmentierung.

Die Idee hinter mehrstufigen Tabellen ist bei Segmentierung und mehrstufigem Paging die gleiche: Ein Prozess benutzt von seinem Adressraum *N* zusammenhängende Bereiche. Zwischen diesen Bereichen können beliebige Lücken auftreten. Statt einer einzigen Seitentabelle verwaltet das System *N* Seitentabellen, pro zusammenhängendem Teilbereich eine Tabelle. Natürlich benötigt man dann noch eine übergeordnete Tabelle mit *N* Einträgen, die jeweils auf die untergeordneten Tabellen verweisen.

Im Abschnitt „Segmentierung“ (**??**, S. **??**) betrachten wir ein Beispiel für zweistufige Tabellen etwas näher.

6.5.5 Invertierte Seitentabellen

Der Begriff „**invertierte Seitentabelle**“ bedeutet nichts anderes als den völligen Verzicht auf Seitentabellen. Stattdessen wird bei Zuordnung eines Seitenrahmens *r* zu einer Seite *s* die Adresse der Seite mit der PID in die Rahmentabelle eingetragen. Die PID ist nötig, da Seitennummern prozessspezifisch sind.

Falls ein Prozess eine Rahmennummer benötigt, muss die Rahmentabelle durchsucht werden, bis der Eintrag mit seiner PID und der gewünschten Seitennummer gefunden ist. Um den Zugriff zu beschleunigen, wird eine Hash-Tabelle verwendet. Wenn durch Einsatz eines genügend grossen TLB nur selten in der Rahmentabelle gesucht werden muss, ist die durchschnittliche Speicherzugriffs-Geschwindigkeit dadurch nicht sonderlich beeinträchtigt. Verschiedene RISC-Rechner (IBM RS 6000, HP Spectrum) verwenden dieses Modell.

6.6 Segmentierung

6.6.1 Grundmodell

Segmentierung bezieht sich auf die Benutzersicht des Speichers. Der logische Adressraum wird unterteilt in Segmente unterschiedlicher Größe, die unterschiedlichen Verwendungszwecken dienen, z.B.

- der Programmcode, den der Programmierer geschrieben hat
- der Laufzeitstack
- der statisch reservierte Speicherbereich für Variablen
- der Programmcode einer dynamisch geladenen Bibliothek
- ein Speicherbereich, der von zwei Prozessen gemeinsam für Kommunikationszwecke benutzt wird
- ein Speicherbereich, in den der Inhalt einer Datei eingeblendet wird

Gemäß der Verwendung sind auch unterschiedliche Zugriffsrechte für die Segmente sinnvoll, z.B. Ausführungsrecht für das Code-Segment, Lese- und Schreibrecht für Variablen-Speicher, Nur-Lese-Zugriff für Konstanten-Speicher usw.

Ein **Segment** ist ein Teilbereich des logischen Adressraums eines Prozesses, der einem bestimmten Verwendungszweck dient und bestimmte Zugriffsrechte hat. Die Segmente eines Prozesses sind nicht geordnet und haben unterschiedliche, von der Anwendung bestimmte Größen. Üblicherweise wird die Segmentstruktur eines Programms durch den Compiler festgelegt und hängt von der Programmiersprache ab.

Wenn der Hauptspeicher segmentiert ist, werden Adressen zweigeteilt in *Segment* und *Distanz*. Zur Abbildung solcher Adressen auf die linearen Hauptspeicheradressen wird, wie beim Paging, eine Tabelle verwendet: die **Segmenttabelle** des Prozesses. Die Implementierung kann in Form von Registern oder - bei großen Segmenttabellen - im Hauptspeicher erfolgen.

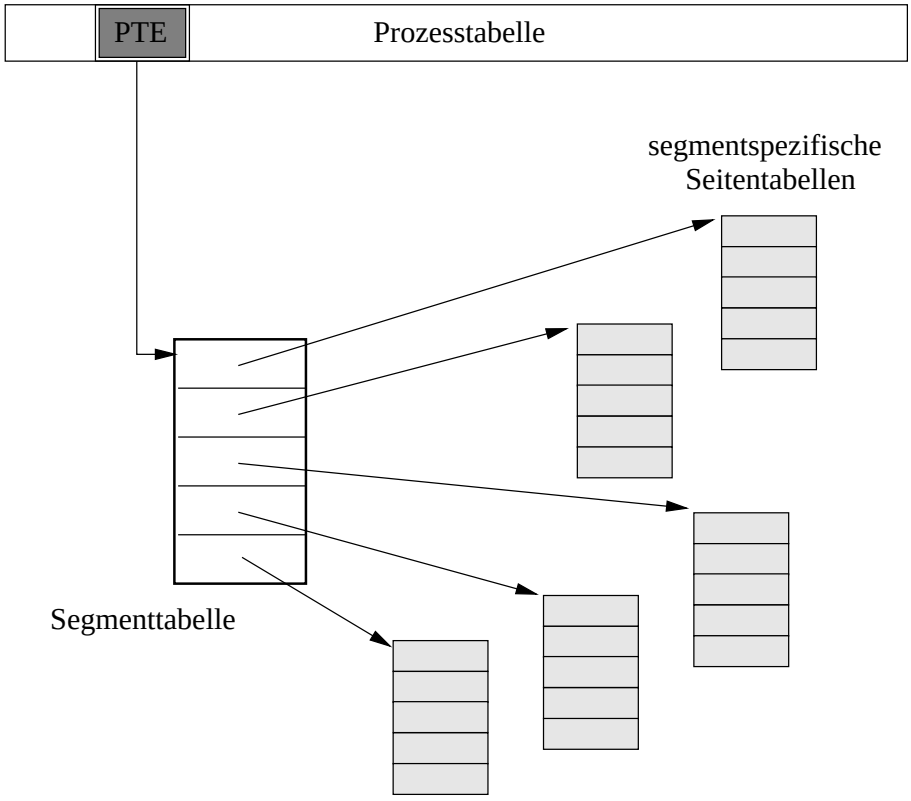
Für jedes Segment wird ein Basisregister und ein Grenzregister benötigt. Der Mechanismus gleicht dem für Prozesse mit dynamischer Adressbindung (**??**, S. **??**), an die Stelle der Prozesse treten die Segmente. Durch Hardware-unterstützte Segmenttabellen ist auch die Prüfung der Segment-bezogenen Zugriffsrechte gewährleistet, die in den Segmenttabelleneinträgen verzeichnet sind.

Wenn man so will, kann man Speicherverwaltung mit variablen Partitionen als den Grenzfall der Segmentierung betrachten, bei dem jeder Prozess aus nur einem Segment besteht. Man kann auch Paging als einen Spezialfall mit Segmenten fester Größe ansehen.

6.6.2 Segmentierung mit Paging

In diversen Systemen werden Paging und Segmentierung kombiniert. Wir betrachten ein einfaches Modell dazu:

Ein Prozess besteht aus einigen (wenigen) Segmenten. Jedes Segment wird in Seiten eingeteilt, für jedes Segment existiert eine Seitentabelle.



Eine Adresse besteht aus Segmentnummer und (Segment-)Distanz, die sich auf den Segmentanfang bezieht. Die Segmentnummer dient als Index für die Segmenttabelle. Der darin stehende Segmentdeskriptor enthält die Größe, Zugriffsrechte und einen Verweis auf die Seitentabelle des Segments. Nach der Zugriffsrechtsprüfung gegen den Segmenttabelleneintrag wird die Segmentdistanz in der gleichen Weise wie eine virtuelle Adresse bei reinem Paging in eine physikalische Adresse transformiert: Sie wird entsprechend der Seitengröße aufgeteilt in eine (Segment-bezogene) Seitennummer und eine Seitendistanz. Aus der Seitennummer wird mittels der Seitentabelle des Segments die Rahmennummer berechnet.

6.7 Virtueller Speicher

6.7.1 Auslagerungsstrategien

Wenn der Hauptspeicher zu knapp wird (Rahmen-Freiliste leer), kann das Betriebssystem Seiten auslagern, um Platz zu schaffen.

Wenn es bei der Wahl der auszulagernden Seiten keine Rolle spielt, zu welchem Prozess die Seiten gehören, spricht man von einer **globalen Auslagerungsstrategie**. Bezieht man die Prozesszugehörigkeit bei der Auswahl mit ein, sprechen wir von einer **prozessbezogenen Auslagerungsstrategie**. Ein Ziel jeder Strategie muss es sein, die Anzahl der Seitenfehler möglichst gering zu halten.

Betrachten wir zunächst globale Strategien:

- **Die optimale Strategie**

  Bei der optimalen Strategie (OPT) werden die Seiten zuerst ausgelagert, die entweder gar nicht mehr oder für eine besonders lange Zeitspanne nicht mehr benötigt werden.

  Leider weiß das Betriebssystem i.d.R. nichts über das zukünftige Verhalten der Prozesse und damit über die künftigen Hauptspeicherzugriffe. Die optimale Strategie ist nicht praktikabel. Allerdings dient sie als Messlatte für andere Strategien, die allesamt als Annäherungsversuche an die optimale Strategie betrachtet werden können.

- **FIFO – „first in – first out“**

  Bei FIFO wird die Seite zuerst ausgelagert, die schon am längsten im Hauptspeicher steht. Dazu muss das Betriebssystem eine verkettete Liste der Seiten verwalten, die bei Ein- und Auslagerungen jeweils aktualisiert wird.

  FIFO hat die unschöne Eigenschaft, dass sehr häufig benötigte Seiten (z.B der Code des Window-Managers) regelmäßig ausgelagert werden, was sofort wieder zu Seitenfehlern führt.

- **LRU – „least recently used“**

  Bei LRU wird die am längsten nicht mehr benutzte Seite zuerst ausgelagert.

  Die dahinter stehende Annahme ist, dass lange nicht mehr benutzte Seiten wohl auch in der Zukunft nur mit geringerer Wahrscheinlichkeit benötigt werden. Diese Annahme ist meist zutreffend, LRU liefert eine geringe Seitenfehlerrate. Dennoch wird LRU beim Paging selten verwendet.

  Das Problem ist der Verwaltungsaufwand: Um zu bestimmen, welche Seite die am längsten nicht mehr benutzte ist, wird eine Liste benötigt, in der die Seiten gemäß Reihenfolge des letzten Zugriffs angeordnet sind. Diese Liste muss bei jedem Seitenzugriff aktualisiert werden, nicht nur beim Ein- und Auslagern wie bei der FIFO-Strategie. Dies ist nur dann genügend schnell möglich, wenn die LRU-Liste als Hardware realisiert ist.

Alternativ könnte die Hardware bei jedem Seitenzugriff die aktuelle Zeit in der Seite vermerken. Bei Auslagerung muss dann durch Vergleich der Zeitstempel die richtige Reihenfolge ermittelt werden.

Nur wenige Computer bieten solche Unterstützung. Software-Implementierung ist dagegen außerhalb jeder Diskussion: Jeder Speicherzugriff müsste einen Interrupt für die Listen-Aktualisierung auslösen, der Prozess würde vielleicht um den Faktor 10 verlangsamt.

- **LFU – „least frequently used“**

  LFU versucht, den vernünftigen Grundgedanken der LRU-Strategie mit vertretbarem Aufwand zu verfolgen. Es wird die Seite ausgelagert, die innerhalb einer gewissen Zeitspanne am wenigsten häufig zugegriffen wurde.

  Wenn die Hardware Zugriffszähler für die Seitenrahmen enthält, ist dies sehr effizient: Bei Speicherbedarf prüft das Betriebssystem die Zähler und lagert die Seiten aus, die einen geringen Zählerwert aufweisen. Dann werden die Zähler zurückgesetzt.

  Viele Maschinen haben statt eines Zählers nur ein Referenz-Bit, das beim Seitenzugriff gesetzt wird. Das Betriebssystem wird dann eben die Seiten Auslagern, bei denen kein Zugriff während der Beobachtungsperiode vorliegt.

  Das Betriebssystem kann auch die Referenzbit-Historie einer Seite abspeichern. Beispiel: Für jede Seite wird in der Rahmentabelle ein 8-Bit-Feld angelegt, das die Zugriffe der letzten 0,8 Sekunden-Intervalls wiederspiegelt. Alle 100 ms wird das Feld um ein Bit nach links geschoben und im rechten Bit das aktuelle Referenzbit der Seite vermerkt.

- **Second Chance**

  Der Second-Chance-Algorithmus verbindet FIFO mit Referenzbits: Wenn eine Seite in der FIFO-Liste vorne steht, d.h. ausgelagert werden müßte, wird das Referenzbit betrachtet. Ist es 0, wird die Seite ausgelagert. Ist es 1, wird es auf 0 gesetzt und die Seite an das Ende der Liste verschoben. Damit verbleibt die Seite noch eine Weile im Hauptspeicher.

Ziel: Anzahl der Seitenfehler minimieren

- globale Auslagerungsstrategie: Zugehörigkeit einer Seite zu einem Prozess spielt keine Rolle
  - optimale Strategie
  - FIFO – „first in – first out“
  - LRU – „least recently used“
  - LFU – „least frequently used“
  - Second Chance
- prozessbezogene Auslagerungsstrategie

- Working-Sets

Zum besseren Verständnis vergleichen wir das Verhalten für einen 3 Rahmen großen Hauptspeicher.

Zugriffsreihenfolge:

5 8 0 2 8 6 8 4 2 6 8 6 2 0 2 8 0 5 8 0

FIFO

Rahmen 1	5	5	5	2	2	2	2	4	4	4	8	8	8	8	8	8	8	5	5	5
Rahmen 2		8	8	8	8	6	6	6	2	2	2	2	2	0	0	0	0	0	8	8
Rahmen 3			0	0	0	0	8	8	8	6	6	6	6	6	2	2	2	2	2	0

OPT

Rahmen 1	5	5	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	5	5	5
Rahmen 2		8	8	8	8	8	8	4	4	4	8	8	8	8	8	8	8	8	8	8
Rahmen 3			0	0	0	6	6	6	6	6	6	6	6	0	0	0	0	0	0	0

LRU

Rahmen 1	5	5	5	2	2	2	2	4	4	4	8	8	8	0	0	0	0	0	0	0
Rahmen 2		8	8	8	8	8	8	8	8	6	6	6	6	6	6	8	8	8	8	8
Rahmen 3			0	0	0	6	6	6	2	2	2	2	2	2	2	2	2	5	5	5

6.7.2 Demand Paging

„Demand Paging“ steht für eine Bedarfs-getriebene Strategie beim Laden von Speicherseiten. Prozesse stehen nicht komplett im Hauptspeicher, ausgelagerte Seiten werden erst bei Zugriffsversuch in den Hauptspeicher geladen.

Dies hat mehrere Vorteile:

- Nicht benötigte Programmteile müssen nicht im Hauptspeicher stehen

  Ein Beispiel sind selten benutzte Ausnahmebehandlungsfunktionen. Wenn diese so gut wie nie aufgerufen werden, ist es unökonomisch, dass sie während der ganzen Ausführungszeit knappe Speicherressourcen belegen.

  Ein anderes Beispiel sind große Arrays. Wenn der Programmierer statische Arrays verwendet, weiß er oft nichts über die zur Laufzeit tatsächlich benötigte Größe. Um einem Überlauf vorzubeugen, kalkuliert er die Feldgrößen extrem großzügig. Zur Ausführungszeit wird dann meist nur ein kleiner, zusammenhängender(!) Bruchteil des Array-Speichers genutzt. Bei Demand-Paging wird für den ungenutzten Teil kein realer Hauptspeicher reserviert.

- Kürzere Ladezeiten

  Manche Systeme erlauben das bedarfsorientierte Laden von Programmteilen aus der Programmdatei. Dadurch entfällt das Laden nicht benutzter Programmteile völlig.

- Programmgröße ist beliebig

  Es sind auch Programme ausführbar, die größer als der reale Hauptspeicher sind.

Die wesentlichen Nachteile liegen auf der Hand:

- Bei jedem Hauptspeicherzugriff muss die virtuelle Adresse über einen mehr oder weniger komplexen, teilweise durch Hardware, teilweise durch Software realisierten Umrechnungsmechanismus auf eine reale Adresse abgebildet werden. Dies kostet in jedem Fall Zeit, auch wenn kein Seitenfehler auftritt.

- Seitenfehler erfordern Kontextwechsel und Plattenzugriffe, verlangsamen damit das System. Im Extremfall (Anzahl der Prozesse gemessen am Hauptspeicher zu groß) ist das System nur noch mit Paging beschäftigt („Thrashing“).

6.7.3 Working-Sets

Programme verhalten sich oft gemäß dem **Lokalitätsprinzip:** Die Bearbeitung läßt sich in einzelne Phasen aufteilen. Innerhalb einer Phase wird ein bestimmter Teil des Programmcodes (z.B. der zu einer Wiederholungsanweisung oder einer rekursiven Funktion gehörende Code) benötigt und auch nur bestimmte Teile der Daten (z.B. eine Matrix, die innerhalb einer Schleife modifiziert wird).

Aus Sicht der Hauptspeicher-Verwaltung heißt das: Wenn jeder Prozess all die Seiten hat, die er für die aktuelle Bearbeitungsphase benötigt, wird er bis zum Ende dieser Phase keine Seitenfehler verursachen.

Die Konsequenz: Man gebe jedem Prozess einen bestimmten, genügend großen Pool von Seitenrahmen. In diese Rahmen kann er zu Beginn jeder Bearbeitungsphase die dafür benötigten Seiten laden. Innerhalb der Bearbeitungsphase werden wenige oder gar keine Seitenfehler auftreten.

Einen solchen Rahmen-Pool nennt man **Working-Set**.

Mit dem Prozess-bezogenen Working-Set-Ansatz will man Probleme früherer global orientierter Paging-Systeme bei zu hoher Belastung begegnen.

Man betrachte dazu folgendes Szenario: Im Hauptspeicher stehen so viele Prozesse, dass jeder nur eine geringe Anzahl seiner Seiten verfügbar hat. Unmittelbar nachdem ein Prozess den Prozessor bekommt, verursacht er deshalb einen Seitenfehler und wird blockiert. Fast alle Prozesse verhalten sich so, die Systemleistung sinkt stark ab. Es kann noch schlimmer kommen: Nehmen wir an, es gibt noch eine Job-Warteschlange. Das Betriebssystem beobachtet die schlechte CPU-Nutzung. Es versucht den Prozessor besser auszulasten, indem es weitere Jobs in den Speicher lädt.

Es kommt zum „Thrashing“ (engl. für „Dreschen“): die Plattenaktivität ist durch das Paging enorm, die Verarbeitung steht aber praktisch still.

Mit dem Working-Set-Modell wird garantiert, dass jeder Prozess eine „vernünftige“ Anzahl von Seiten im Speicher halten kann. Viele Betriebssysteme verknüpfen diese Modell mit den oben diskutierten globalen Seitenersetzungs-Algorithmen.

Wie betrachten eine kombinierte Strategie als Beispiel:

Jeder Prozess bekommt ein 20 Seiten großes Working-Set. Dieses füllt er mit Demand-Paging. Bei einem Seitenfehler wird nach dem FIFO-Prinzip die älteste Seite seines Working-Sets ausgetauscht gegen die benötigte Seite.

Ein Teil des Hauptspeichers wird als **Seiten-Cache** benutzt: Eine Seite, die aus dem WS eines Prozesses entfernt wird, verbleibt zunächst im Hauptspeicher. Sie wird in den Cache übernommen. Falls ein Prozess eine Seite benötigt, die nicht in seinem WS steht, wird zunächst im Cache gesucht. Falls die Seite dort vorhanden ist, liegt ein „weicher“ Seitenfehler („soft page fault“) vor. Ein Plattenzugriff ist nicht nötig.

Bei Platzmangel werden aus dem Cache (z.B. nach dem „Second Chance“-Verfahren) Seiten ausgelagert.

Kapitel 7

Prozessor-Verwaltung (CPU-Scheduling)

7.1 Rahmenbedingungen

Früher war Scheduling einfach: Interaktive Programme gab es nicht, nur Stapelaufträge. Ein Stapelauftrag (Batch-Job) ist eine nicht-interaktive Anwendung, die nicht sofort ausgeführt werden muss und zunächst in eine Warteschlange eingereiht wird.

Stapelaufträge wurden in Form von Lochkartenstapeln dem Operateur übergeben, daher die Bezeichnung. Nach dem Einlesen eines Stapels wurden gemäß den Befehlen auf den Lochkarten Prozesse erzeugt, Programme aufgerufen und abgearbeitet.

Im einfachsten Fall bestand das Scheduling darin, dass ein Auftrag nach dem anderen bearbeitet wurde.

Auf einem PC, dessen Benutzer ein Textverarbeitungsprogramm benutzt, ist oft keine weitere Anwendung aktiv. Auch hier ist Scheduling einfach: Das Textverarbeitungsprogramm bekommt die gesamte Prozessorzeit.

Schwieriger wird die Sachlage, wenn viele Benutzer gleichzeitig mit dem Rechner arbeiten, nicht-interaktive Hintergrundprozesse (z.B. Netzwerkdämonen, interaktiv gestartete Compilierungen) aktiv sind und außerdem Auftragswarteschlangen (z.B. monatliche Lohn- und Gehaltsabrechnung, tägliche Aktualisierung des Datenbestands eines Datenwarenhauses) zu bearbeiten sind. Dies ist typisch für einen Großrechner.

Der Vorteil einer Stapelauftragsverwaltung besteht heute darin, dass Aufträge zu einem genehmen Zeitpunkt, (z.B. nachts oder am Wochenende) aus der Warteschlange genommen und bearbeitet werden können, um eine gleichmäßige Lastverteilung zu erreichen. Aufträge werden heute in Form von Auftragsdateien (auf der Platte) erstellt, gespeichert und mit speziellen Kommandos in die Warteschlange(n) eingereiht.

Es gilt zu entscheiden, wieviel Anteil an der Rechenzeit die einzelnen Prozesse bekommen, wann und in welcher Reihenfolge die Stapelaufträge bedient werden und nach welchem Schema ausführende Prozesse verdrängt werden. Verdrängung (engl.: „preemption“) heißt, dem ausführenden Prozess den Prozessor zu entziehen, weil er

das ihm zugeteilte Zeitquantum aufgebraucht hat oder weil ein anderer Prozess mit höherer Priorität den Prozessor beansprucht.

7.2 Ziele

Der Teil des Betriebssystems, der die Vergabe des Prozessors an konkurrierende Prozesse kontrolliert, heißt „Scheduler“. Mehrere, teilweise divergierende Ziele müssen bei der Prozessorzuteilung berücksichtigt werden:

- **Fairness**

  Jeder Prozess soll einen gerechten Anteil der insgesamt verfügbaren Prozessorkapazität erhalten.

- **Ressourcennutzung**

  Prozessoren und andere wichtige Systemressourcen, z.B. Festplatten, sollen möglichst gleichmäßig und gut ausgelastet sein.

- **Reaktionszeit**

  Das System soll eine bestimmte Anzahl interaktiver Nutzer so unterstützen, dass es auf interaktive Eingaben genügend schnell reagiert.

- **Auftrags-Wartezeiten**

  Stapelaufträge sollen möglichst schnell bearbeitet werden.

- **Programm-Durchsatz**

  Die Anzahl der pro Zeiteinheit bearbeiteten Programme soll möglichst hoch sein.

Man sieht sofort einige Zielkonflikte. Wartezeiten sind beispielsweise dann gering, wenn man ein leistungsmäßig völlig überdimensioniertes System anschafft. Dessen Ressourcen sind dann allerdings nicht besonders gut ausgelastet, im Endeffekt gibt man unnötig viel Geld aus.

Auch die Anforderungen rechenintensiver Hintergrundprozesse und E/A-intensiver interaktiver Anwendungen sind schwer vereinbar. Für den interaktiven Nutzer wäre es sicher vorteilhaft, sämtliche Hintergrundaktivitäten nur nachts oder am Wochendende durchzuführen, um eine schnelle Reaktionszeit zu garantieren. Auftrags-Wartezeiten, Auftrags-Durchsatz und Prozessorauslastung wären dann allerdings schlecht.

7.3 Basis-Algorithmen für Batch-Betrieb

Wir betrachten zunächst nicht-verdrängende Basisverfahren für die Warteschlangenverwaltung:

7.3.1 FIFO-Scheduling

„Wer zuerst kommt, mahlt zuerst! “. Dieses Verfahren ist sehr einfach und im formalen Sinne fair. Der Prozess, der am längsten in der Warteschlange steht, wird zuerst bedient und vollständig bearbeitet.

Reines FIFO-Scheduling tritt in der Praxis selten auf. FIFO wird in Kombination mit Prioritäten aber sehr häufig genutzt.

Folgende Einwände sprechen jedoch gegen FIFO:

- Es gibt keine Unterscheidung zwischen wichtigen und unwichtigen Aufträgen. In der Praxis gibt es nun aber immer wieder sehr dringliche Aufgaben, die man vorziehen möchte.
- Kurze Aufträge müssen auf langwierige Berechnungen warten. Die mittleren Wartezeiten sind nicht optimal.

7.3.2 „Der kürzeste Job zuerst“ (KJZ)

Aus der Warteschlange wird der Auftrag mit der kürzesten Laufzeit zuerst bearbeitet. Damit erreicht man eine minimale mittlere Wartezeit. Allerdings ist die Wartezeit für längere Aufträge schwer vorhersehbar.

Ein Problem bereitet auch die Schätzung der Laufzeit, die man in der Regel dem Benutzer überlässt. Wenn der Benutzer die Laufzeit zu kurz angibt, kann das System den Auftrag nach Ablauf der angegebenen Zeit abbrechen (oder in einem Rechenzentrum für die darüber hinaus benötigte Zeit hohe „Strafgebühren“ berechnen).

7.3.3 Mischverfahren

Wenn die mittlere Wartezeit minimiert werden soll, ist die Auswahl des kürzesten Jobs optimal. Um beliebig langes Warten längerer Jobs zu vermeiden, kann man mit einer geeigneten Funktion von Wartezeit und Bearbeitungszeit einen Kompromiss zwischen FIFO und KJZ erzielen.

Ein Beispiel ist das Verfahren „höchstes Antwortzeit-Verhältnis zuerst“. Die Antwortzeit ist die Summe aus Warte- und Bearbeitungszeit.

Bei Freiwerden des Prozessors wird für jeden Job ein Priorität bestimmt:

$$Priorität = \frac{Wartezeit + Ausführungszeit}{Ausführungszeit}$$

Der Auftrag mit der höchsten Priorität wir zuerst ausgeführt. Dabei werden kürzere Jobs bevorzugt, aber auch längere Jobs, die schon lange warten.

Durch einen zusätzlichen Gewichtungsfaktor für die Wartezeit läßt sich das Verfahren variieren.

7.4 Basis-Algorithmus für Timesharing-Betrieb

Nicht-verdrängende Verfahren sind für den Timesharing-Betrieb nicht akzeptabel, da die Reaktionszeit des Systems auf interaktive Eingaben unvorhersehbar und in aller Regel zu lang ist.

Reihum-Verfahren („Round Robin“)

Bei dem Verfahren wird jedem Prozess reihum für ein festes Quantum (z.B. 100 ms) der Prozessor überlassen. Wenn der Prozess blockiert oder sein Quantum abgelaufen ist (Interrupt durch Hardware-„Clock“), führt der Scheduler einen Kontextwechsel durch.

Ein Problem ist die Festlegung des Quantums: Ist das Quantum zu groß, dauert es bei einer großen Prozessanzahl lange, bis ein Prozess nach seiner Verdrängung wieder an der Reihe ist. Dadurch ist die Reaktionszeit des Systems auf interaktive Eingaben ggf. zu lange.

Beispiel: Im System sind 25 Prozesse aktiv, das Quantum beträgt 200ms, die Kontextwechselzeit 5ms. Wenn ein Prozess gerade verdrängt wurde, dauert es, falls alle anderen Prozesse ihr Quantum voll ausnutzen, 5 Sekunden ($24 * (200ms + 5ms) = 4920ms$), bis der Prozess den Prozessor wieder erhält. Ein interaktiver Benutzer wird heute schon bei sehr viel kürzeren Reaktionszeiten ungeduldig!

Ist das Quantum dagegen zu klein, sinkt die Systemleistung: Das System benötigt für jeden Kontextwechsel den Prozessor. Je öfter Prozesse verdrängt werden, desto kleiner ist der Anteil der Prozessorzeit, die für die eigentliche Programmausführung zur Verfügung steht. Wenn beispielsweise das Quantum und die Kontextwechselzeit gleich groß sind, stehen nur 50% der Prozessorleistung für die Programmausführung zur Verfügung, der Rest wird für Kontextwechsel benötigt.

Das Quantum muss nicht notwendigerweise für alle Prozesse identisch sein. Man kann z.B. ein Prioritätsschema damit implementieren, dass man Prozessen mit hoher Priorität ein größeres Quantum gibt (z.B. POSIX-Thread-Scheduler).

7.5 Prioritäten

Prioritäten werden aus unterschiedlichen Gründen eingeführt:

- Um wichtige bzw. dringende Prozesse bei der Verarbeitung vorzuziehen
- Um besonders gut zahlenden Kunden ($\rightarrow$ Rechenzentrum) einen bevorzugten Service zu bieten
- Um interaktive Prozesse im Hinblick auf kurze Reaktionszeiten gegenüber rechenintensiven Hintergrundprozessen zu bevorzugen

- Um Echtzeitanforderungen erfüllen zu können

Statische Prioritäten werden vom Administrator bzw. vom Benutzer dem Prozess zugeordnet. Sie ändern sich nicht. Dynamische Prioritäten werden vom Betriebssystem regelmäßig neu berechnet, wobei Eigenschaften eines Prozesses (statische Priorität, Ressourcenutzung) oder Systemzustandsmerkmale (Warteschlangengrößen, Anzahl der Prozesse) mit eingerechnet werden können.

Statische und dynamische Prioritäten werden im Prozessdeskriptor eines Prozesses abgespeichert.

7.6 Verfahren für gemischten Betrieb mit dynamischen Prioritäten

Jedes Betriebssystem verwendet spezifische Scheduling-Verfahren. Die Beurteilung der Verfahren ist schwierig, weil die Divergenz der Ziele an vielen Stellen Kompromisse erfordert. Ein Scheduler, der bei stark interaktiver Nutzung des Rechners gute Ergebnisse bringt, kann sich in einer Umgebung mit vielen rechenintensiven Prozessen oder im Einsatz auf einem Netzwerk-Server als zweitklassig erweisen.

In der Praxis benutzt man immer komplexe, Prioritäten-basierte Verfahren, die in verschiedener Weise parametrisiert sind, um unterschiedlichen Einsatzbedingungen gerecht zu werden. Administratoren können in Tuning-Kursen dann lernen, an welchem „Schräubchen“ zu drehen ist, um den Scheduler an die Rahmenbedingungen anzupassen. Oft fehlt allerdings ein genau umschriebenes und zeitlich konstantes Rechnernutzungsprofil.

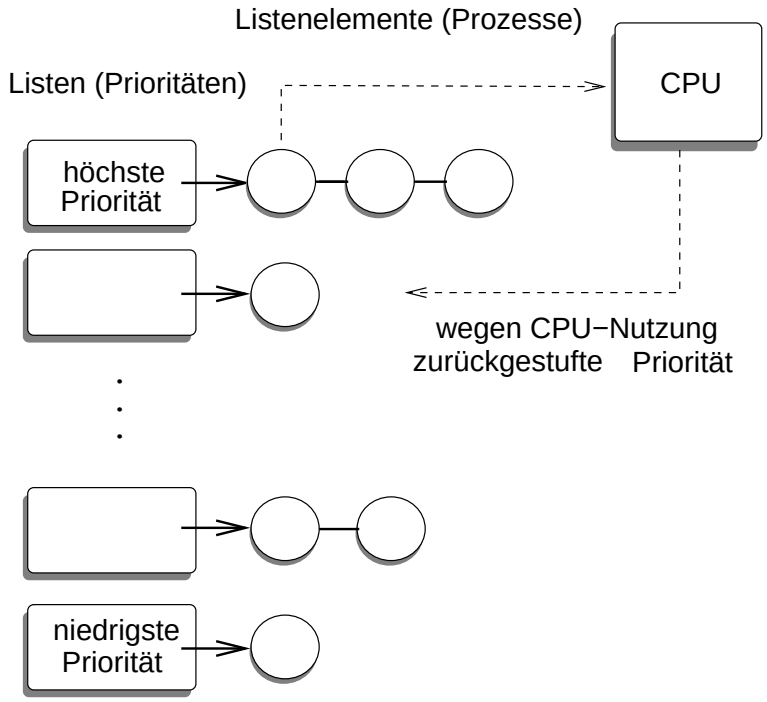
Gemeinsam ist den Verfahren, dass sie versuchen, die Prozesse als rechenintensiv oder E/A-intensiv zu klassifizieren. E/A-intensive Prozesse werden schneller bedient. Da das E/A-Verhalten eines Prozesses sich verändern kann, muss das System statistische Informationen sammeln und die Klassifikation ständig aktualisieren. Man nennt solche Verfahren deshalb adaptive Verfahren.

7.6.1 Mehrstufige Feedback-Listen

Die Grundidee: Es gibt N Prioritätsstufen, zu jeder Stufe eine Liste („run queue“) mit Prozessen im Zustand „bereit“. Alle Prozesse bekommen das gleiche Quantum.

Die Auswahl des nächsten ausführenden Prozesses richtet sich nach der Priorität: Der Prozess, der in der höchsten nicht-leeren Prioritätsliste ganz vorne steht, bekommt den Prozessor. Wenn er sein Quantum aufgebraucht hat, wird er in der nächst niedrigeren Prioritätsliste hinten angefügt.

Die Prozesse der höchsten Prioritätsstufe werden reihum ausgeführt.



Ein neuer Prozess kommt in die höchste Prioritätsstufe.

Ein Prozess der sein Quantum nicht ausnutzt, wird beim Blockieren aus dem Prioritäts-Netzwerk entfernt und in eine Warteliste eingefügt. Wenn das erwartete Ereignis eintritt, kommt er an das Ende der Liste, in der er vorher war.

Sobald ein Prozess in den Zustand „bereit“ wechselt, der eine höhere Priorität hat als der ausführende Prozess, wird der ausführende Prozess verdrängt.

Das beschriebene Grundschema hat folgende wichtige Eigenschaften:

- E/A-intensive Prozesse, die ihr Quantum nie oder selten benötigen, werden schnell bedient, weil sie in einer hohen Prioritätsklasse bleiben.
- Kurze Programme werden schnell ausgeführt, weil sie anfangs in die höchste Prioritätsstufe eingeordnet werden.
- Langlaufende rechenintensive Anwendungen bekommen im Laufe der Zeit eine geringe Priorität. Sie bekommen den Prozessor seltener, nutzen ihn dafür aber auch länger als die E/A-intensiven Prozesse.

Variationen:

- **Quantumsgröße**

  Man könnte rechenintensiven Prozesse ein größeres Quantum zubilligen, quasi als Ausgleich dafür, dass sie nur „selten“ den Prozessor bekommen.

- **Prioritätsverbesserung bei Interaktion**

  Im Grundschema bleibt ein Prozess der niedrigsten Prioritätsstufe immer in dieser Stufe stehen. Wenn ein Programm nach einer rechenintensiven Phase wieder viele Interaktionen durchführt, wird dies nicht berücksichtigt.

  Man könnte hier die Priorität wieder hochstufen, wenn der Prozess aus der Warteliste in das Prioritätsnetzwerk zurückgeführt wird. Bei einigen UNIX-Systemen wird beispielsweise die Priorität abhängig von der Art des erwarteten Ereignisses (Terminaleingabe, Terminalausgabe, Platten-E/A usw.) um mehrere Stufen erhöht.

- **Berücksichtigung von Wartezeiten**

  Prozesse der geringsten Priorität werden nur dann bearbeitet, wenn alle anderen Prioritätsstufen leer sind. Wenn sehr viele E/A-intensive Prozesse im System sind, können daher die rechenintensiven Prozesse „verhungern“.

  Man könnte hier in regelmäßigem Zyklus die Priorität abhängig von der Wartezeit stufenweise wieder erhöhen.

- **Basisprioritäten**

  Das Schema erlaubt keine Unterscheidung zwischen wichtigen und unwichtigen bzw. dringenden und weniger dringenden Prozessen.

  Man könnte hier eine vom Administrator zu vergebende, feste Basispriorität in die Berechnung mit einbeziehen. Diese Basispriorität ist ein Maß für die Wichtigkeit des Programms und wird in einer geeigneten Weise mit dem adaptiven Mechanismus kombiniert.

  Die Basispriorität könnte z.B. als unterste Prioritätsstufe dienen.

  In Verbindung mit Prioritätsverbesserungsmöglichkeiten könnte die Basispriorität auch als Einstiegsstufe für den Prozess dienen: Der Prozess kann sich auf eine höhere Stufe verbessern (z.B. durch Anrechnung von Wartezyklen). Sobald er aber den Prozessor bekommen und sein Quantum genutzt hat, fällt er wieder auf seine Basispriorität zurück.

- **Echtzeitprioritäten**

  In einigen Betriebssystemen sind die obersten Prioritätsstufen für Echtzeitprozesse reserviert, die eine garantierte maximale Antwortzeit benötigen.

  Echtzeitprioritäten sind statisch und werden ausschließlich vom Administrator vergeben. Er muss abhängig von den Anforderungen der zu überwachenden bzw. zu steuernden Anlage dafür sorgen, dass es keine Prioritätskonflikte unter den Echtzeitprozessen gibt.

  Ein Echtzeitprozess darf nicht verdrängt werden.

Kapitel 8

Dateisysteme

Dateisysteme definieren eine Abstraktion physikalischer Speichermedien. Sie ermöglichen die Nutzung der Speichermedien zur permanenten Abspeicherung von Daten unabhängig von der Gerätetechnologie. Sie definieren Zugriffschutzmechanismen und stellen die Datenintegrität im Fehlerfall sicher.

8.1 Dateien

Eine *Datei* ist eine Sammlung logisch zusammengehöriger Daten, die in der Regel auf einem permanenten Speichermedium, z.B. Platte, Diskette, Band oder CD, abgespeichert ist.

Typische Operationen für Dateien sind

- Öffnen
- Schließen
- Erzeugen
- Löschen
- Daten lesen
- Daten schreiben
- Daten modifizieren

8.1.1 Datensätze und Zugriffsarten

Die Daten sind innerhalb der Datei oft als **Datensätze** organisiert, man unterscheidet Dateien mit fester und mit variabler Satzlänge.

Bei fester Satzlänge muss die Satzlänge nur einmal als Dateiattribut gespeichert werden. Die Position des *n*-ten Datensatzes ist leicht zu berechnen: *n**Satzlänge. Bei variabler Satzlänge hat jeder einzelne Datensatz ein Längenattribut (meist am Anfang des Datensatzes untergebracht), oder es gibt ein Terminierungszeichen bzw. eine Terminierungszeichenfolge.

Man unterscheidet verschiedene Zugriffsarten:

sequentieller Zugriff

: Die Datensätze werden vom ersten Datensatz beginnend nacheinander verarbeitet. Bei Band-Dateien ist dies meist die einzige Zugriffsmethode.

relativer Zugriff

: Auf die einzelnen Datensätze kann man wahlfrei über deren Position in der Datei zugreifen.

ISAM - Index-sequentieller Zugriff

: Jeder Datensatz hat einen Satzschlüssel, der ihn eindeutig identifiziert. Der Schlüssel definiert eine logische Reihenfolge der Datensätze.

  Eine ISAM-Datei (ISAM="index sequential access method") unterstützt

  - wahlfreien Zugriff über den Satzschlüssel
  - sequentiellen Zugriff gemäß auf- oder absteigender Schlüsselanordnung

  Wollte man die Datensätze nach Schlüsseln sortiert auf dem Speichermedium anordnen, würde jedes Einfügen eines neuen Datensatzes am Anfang oder in der Mitte der Datei einen beträchtlichen Reorganisationsaufwand nach sich ziehen. Die tatsächliche Anordnung der Datensätze innerhalb einer ISAM-Datei ist deshalb unabhängig von der logischen durch die Schlüssel definierten Reihenfolge.

  Um die Zugriffsmethoden effizient zu implementieren wird eine ISAM-Datei aufgeteilt in einen Index und einen Datenteil. Der Index ist ein Suchbaum, der zu jedem Schlüssel die Position des zugrhörigen Datensatzes im Datenteil enthält. Für die logisch sequentielle Suche sind die Datensätze oft untereinander verkettet.

Im einfachsten Fall (z.B. UNIX) unterstützt das Betriebssystem nur Byteströme, das heißt Dateien ohne innere Struktur. Es bleibt dann den Anwendungsprogrammen überlassen, sich um Satzstrukturen zu kümmern.

Bei dem NTFS-Dateisystem von Windows NT gehören zu einer Datei ggf. mehrere Datenströme, es gibt also sozusagen Unter-Dateien. In einem Datenstrom befindet sich der eigentliche Datei-Inhalt, in einem anderen sind Informationen untergebracht, die für die Desktop-Darstellung der Datei benötigt werden.

8.1.2 Attribute, Deskriptor, Kontrollblock

Zur Handhabung der Dateien benutzt man Dateinamen, die auf Betriebssystemebene einer Datei zugeordnet werden. Die Zuordnung erfolgt i.d.R. beim Erzeugen der Datei und kann später verändert werden. Bei einigen Systemen hat eine Datei genau einen Namen, bei anderen dagegen ist ein Name ein Verweis auf eine Datei und es kann beliebig viele Verweise auf eine Datei geben.

Die Zuordnung von Dateinamen und Dateien wird meist in Datei-Verzeichnissen verwaltet.

Als **Dateiattribute** bezeichnen wir die zur Verwaltung der Dateien vom Betriebssystem benötigten Daten, z.B.

- Dateieigentümer
- Zugriffsrechte
- Organisationsform (Sequentiell, ISAM usw.)
- Zeitstempel (Erzeugung, letzter Zugriff, letzte Modifikation usw.)
- Dateityp (Textdatei, Verzeichnis, ausführbares Programm usw.)
- logische Größe
- Lese- / Schreibpasswort
- Position der Daten auf dem Speichermedium

Die Zusammensetzung der Verwaltungsdaten ist stark vom Betriebssystem abhängig. Die Attribute einer Datei werden meist getrennt von den Datensätzen dieser Datei in Form eines separaten „Attribut-Datensatzes“ abgespeichert, den wir **Dateideskriptor** nennen.

Die Dateideskriptoren können in Datei-Verzeichnissen untergebracht sein (z.B. DOS/Windows FAT-Dateisystem). Eine Alternative ist die Verwaltung einer separaten Dateideskriptor-Tabelle pro Speichermedium (UNIX: I-Node-Tabelle, NTFS: Master File Table).

Zur Verarbeitung einer Datei wird der Dateideskriptor natürlich benötigt. Das Betriebssystem wird beim Öffnen einer Datei im Hauptspeicher eine Datenstruktur erzeugen, die die Attribute des Deskriptors, aber auch weitere Kontrollinformationen enthält. Dazu könnte beispielsweise ein Referenzzähler oder ein Modifikationsattribut gehören. Diese während der Verarbeitung einer Datei im Hauptspeicher gehaltenen Verwaltungs- und Zugriffskontrolldaten nennen wir **Dateikontrollblock**.

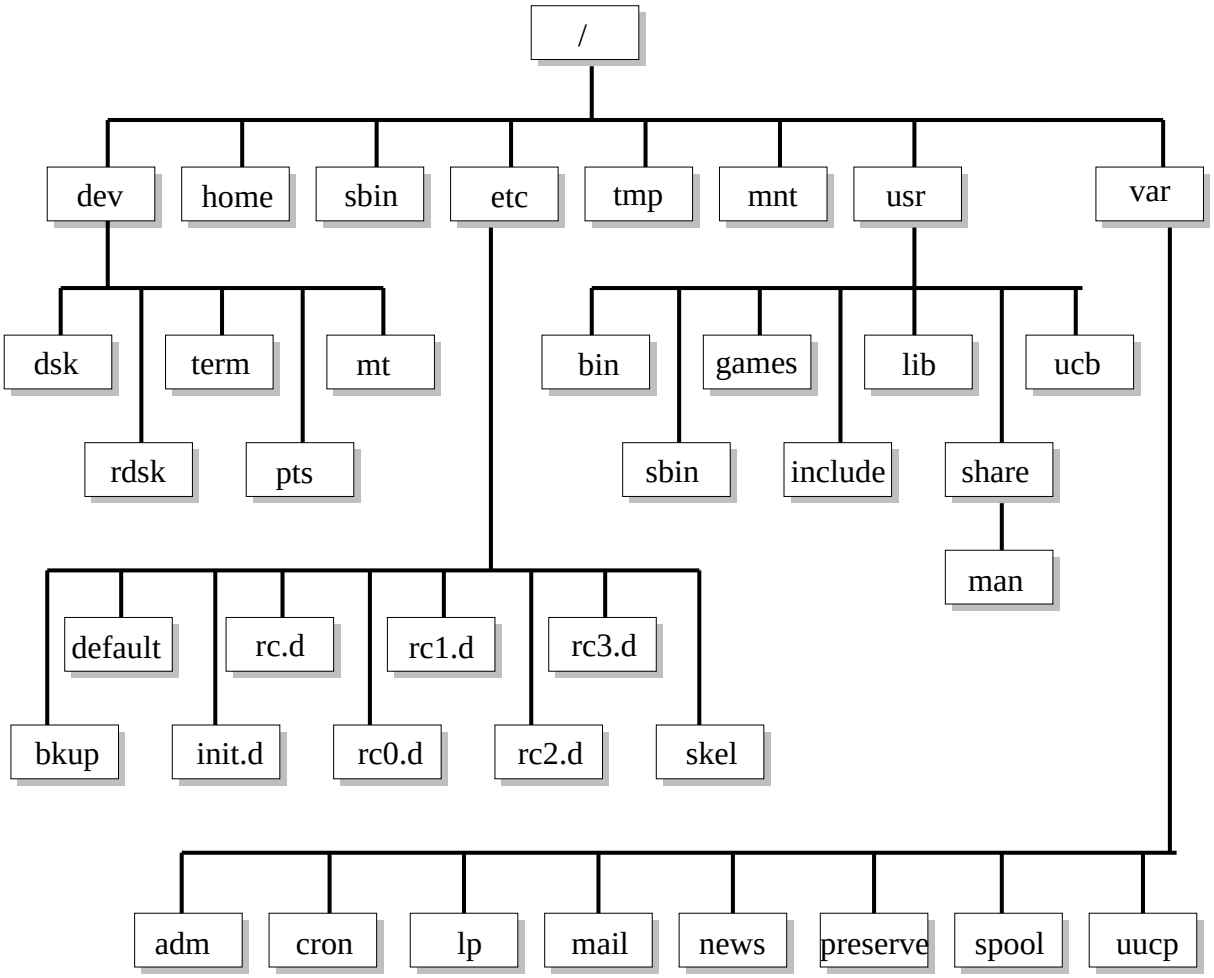
8.2 Verzeichnisse

Selbst bei einem Desktop-PC ist es heute nicht mehr aussergewöhnlich, wenn mehr als 100000 Dateien auf einer Festplatte verwaltet werden.

Verzeichnisse dienen zur hierarchichen Strukturierung der Dateien, sie helfen, Ordnung zu schaffen. Ein Verzeichnis ist ein benannter Container, der einfache Dateien und/oder Unterverzeichnisse enthalten kann.

Verzeichnisse werden üblicherweise selbst wieder als Dateien – allerdings mit speziellen Operationen – implementiert.

Beispiel für eine Dateihierarchie: Verzeichnisstruktur bei UNIX-System V.4



8.3 Implementierung von Dateisystemen

Jedes Speichermedium, das zum Abspeichern eines Dateisystems verwendet wird, ist in einer dateisystemspezifischen Weise organisiert. Typischerweise findet man Boot-Code und dateiübergreifende Verwaltungsdaten (wie etwa die Information über freien Speicherplatz) am Anfang des Mediums, gefolgt von Dateien und Verzeichnissen, die meist in Form nicht zusammenhängender Blöcke über das Medium verstreut sind.

Die Blöcke, die als Einheiten der Speicherzuteilung verwendet werden, sind typischerweise 1-8 Kb groß. Bei Platten und anderen runden Medien mit Sektorstruktur wird für die Speicherung eines Dateiblocks eine Folge zusammenhängender Sektoren verwendet. Die Blockgröße ist also dann ein Vielfaches der Sektorgröße. Die Blöcke werden bei manchen Betriebssystemen auch als „Cluster“ bezeichnet. Die Anzahl der Sektoren pro Cluster wird dann auch als Clusterfaktor bezeichnet.

In der Regel werden die Dateien, mit denen der Anwender arbeitet, auf mehrere Speichermedien verteilt sein (typischerweise Fest- und Wechselplatten, Bänder, CDs, Disketten, WORM-Medien usw.). Wir betrachten im folgenden Platten, die Ausführungen sind auf andere Medien übertragbar (andere Medien werden allerdings nicht partitioniert).

Eine Platte wird zuerst formatiert und dann in mehrere **Partitionen** eingeteilt. Sowohl zum Formatieren als auch zum Partitionieren gibt es spezielle Programme (eine SCSI-Platte wird in der Regel beim Hersteller formatiert.)

Jede Partition lässt sich unabhängig von anderen Partitionen in verschiedener Weise nutzen, z.B.:

- als **Swap-Partition** im Dienste der Speicherverwaltung
- als **Dateiarchiv**, das von einem Archivprogramm verwaltet wird,
- als **Datenbank**, die (ohne Betriebssystemunterstützung!) direkt von einem Datenbanksystem verwaltet wird,
- als **Dateisystem** im Sinne einer strukturierten, mittels Verzeichnissen und speziellen Meta-Datenstrukturen hierarchich organisierten Menge von Dateien, die vom Betriebssystem verwaltet wird.

Den überladenen Begriff „Dateisystem“ verwenden wir in diesem Abschnitt im oben angegebenen Sinne, also mit Bezug auf die logische Struktur des für die Dateien verwendeten Speichermediums.

Eine sinnvolle Planung des **Plattenlayout**, d.h. der Partitionierung, erfordert einiges Nachdenken über die zukünftige Plattennutzung, denn eine spätere Änderung ist immer aufwändig!

Wir betrachten als Beispiele drei Organisationsformen für Dateisysteme: MSDOS FAT-Dateisystem, ein einfaches UNIX-Dateisystem und das Windows NT Dateisystem.

8.3.1 DOS/Windows-FAT-Dateisystem

Bei DOS sind die Dateideskriptoren in Verzeichnissen enthalten. Ein Dateisystem heißt „Laufwerk“.

Wir betrachten als erstes Beispiel den Aufbau einer Diskette mit einem MSDOS-FAT-Dateisystem.

- Dateien sind im allgemeinen nicht zusammenhängend gespeichert.
- Eine globale FAT (file allocation table) gibt Aufschluss über die Speicherblöcke (Cluster), die zur Datei gehören.

  Die Clustergröße ist immer ein Vielfaches der hardwareabhängigen Sektorgröße.
- Der Bootsektor (Sektor 0) gibt Aufschluß über die Sektor- und Clustergröße, sowie über die Größen von FAT und Hauptverzeichnis.

Bootsektor

```
struct msdos_boot_sector {
   char ignored[3];          /* Boot strap short or near jump */
   char system_id[8];        /* Name - can be used to special case
                               partition manager volumes */
   unsigned char  BPS[2];    /* #bytes per logical sector */
   unsigned char  SPC;       /* #sectors per cluster */
   unsigned short RES;       /* #reserved sectors */
   unsigned char  NFATS;     /* #FATs */
   unsigned char  NDIRS[2];  /* #root directory entries */
   unsigned char  NSECT[2];  /* #sectors total*/
   unsigned char  MEDIA;     /* media descriptor */
   unsigned short SPF;       /* #sectors/FAT */
   unsigned short SPT;       /* #sectors per track */
   unsigned short NSIDES;    /* #heads */
   unsigned long  NHID;      /* #hidden sectors (unused) */
   unsigned long  TSECT;     /* #sectors (if sectors == 0) */
};
```

- Alle 2-Byte-Angaben beginnen mit dem llowByte, z.B. bedeutet ein 2-Byte-Bootsektor-Eintrag für das BPS-Feld:

```
Adresse 0x0B 0x0C
Wert    0x00 0x02
```

dass die Diskette 0x0200 = 512 Bytes pro Sektor hat.

- *Reserved* ist die Anzahl der für das Dateisystem am Diskettenanfang reservierten Sektoren (inklusive Bootsektor), üblich ist Reserved=1

Verzeichnisse

Jeder Verzeichniseintrag hat 32 Byte mit folgenden Feldern:

1. Dateiname	8 Byte
2. Dateityp	3 Byte
3. Attribute	1 Byte
4. Filler	10 Byte
5. Uhrzeit der letzten Modifikation	2 Byte
6. Datum der letzten Modifikation	2 Byte
7. Nummer des 1. Clusters	2 Byte
8. Größe der Datei in Bytes	4 Byte

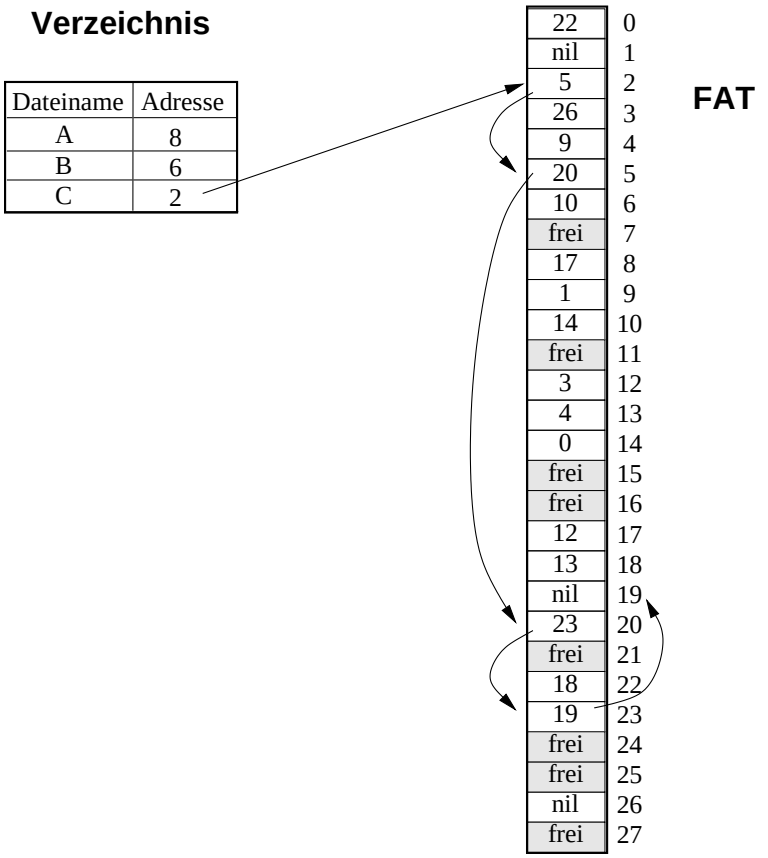
FAT-Aufbau

Die FAT enthält für alle Dateien die Information über die zugehörigen Cluster. Gleichzeitig dient die FAT zur Buchführung über freie Cluster.

Jeder Eintrag ist 12 Bit groß (für Platten werden auch 16- und 32-Bit-Einträge unterstützt).

Die Nummer des ersten Clusters einer Datei steht im Verzeichnis (siehe oben). Falls n die Nummer eines beliebigen Clusters einer Datei ist, enthält der n-te FAT-Eintrag die Nummer des nachfolgenden Clusters, bzw. 0xFFF, falls es für diese Datei kein nachfolgendes Cluster mehr gibt.

Beispiel:



Umrechnung von Clusternummern in Sektoren

Die Cluster werden im Verzeichnis und in der FAT durch logische Clusternummern identifiziert.

Clusternummern sind logische Nummern, die bei 2 beginnen (, da die ersten zwei FAT-Einträge zur Codierung des FAT-Formats verwendet werden). Cluster 2 ist also das erste Datencluster auf der Diskette, Cluster 3 das zweite usw.

Die Sektoren von Cluster 2 beginnen direkt hinter dem Hauptverzeichnis, d.h. z.B. bei

RES = 1	ein reservierter Sektor		Sektor 0
SPF = 3	3 Sektoren pro FAT		
NFATS = 2	2 FATS	FAT 1:	Sektor 1-3
		FAT 2:	Sektor 4-6
NDIRS = 112	112 Hauptverzeichniseinträge a 32 Byte:		Sektor 7-13

dass der erste Datensektor, also der erste Sektor des Clusters mit der logischen Nummer 2, der Sektor 14 ist.

Allgemein gilt (ohne Gewähr, rechnen Sie selbst nach!):

NROOTSECTORS = (NDIRS + BPS - 1) / 32

1. Sektor von Cluster n = (n-2)*SPC + RES + NFATS*SPF + NROOTSECTORS

8.3.2 Ein einfaches UNIX-Dateisystem

Bei UNIX heißt ein Dateideskriptor I-Node („index node“), ein Dateisystem „logical device“.

Wir betrachten hier ein „klassisches“ UNIX-Dateisystemlayout. Moderne UNIX-Varianten verwenden meist effizientere und robustere Organisationsformen, die allerdings auch erheblich komplizierter sind.

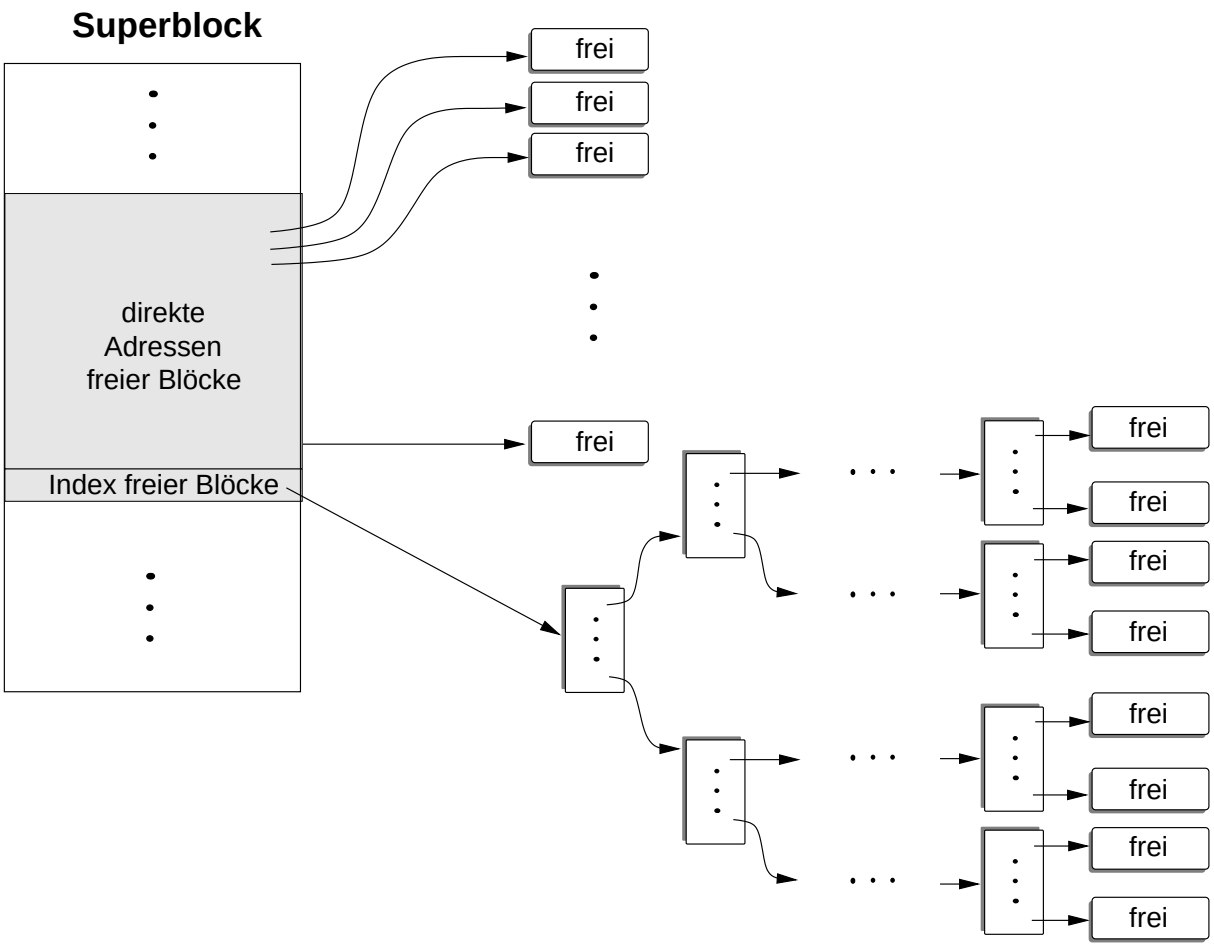
Der **Bootblock** kann Boot-Code zum Laden des Betriebssystems enthalten.

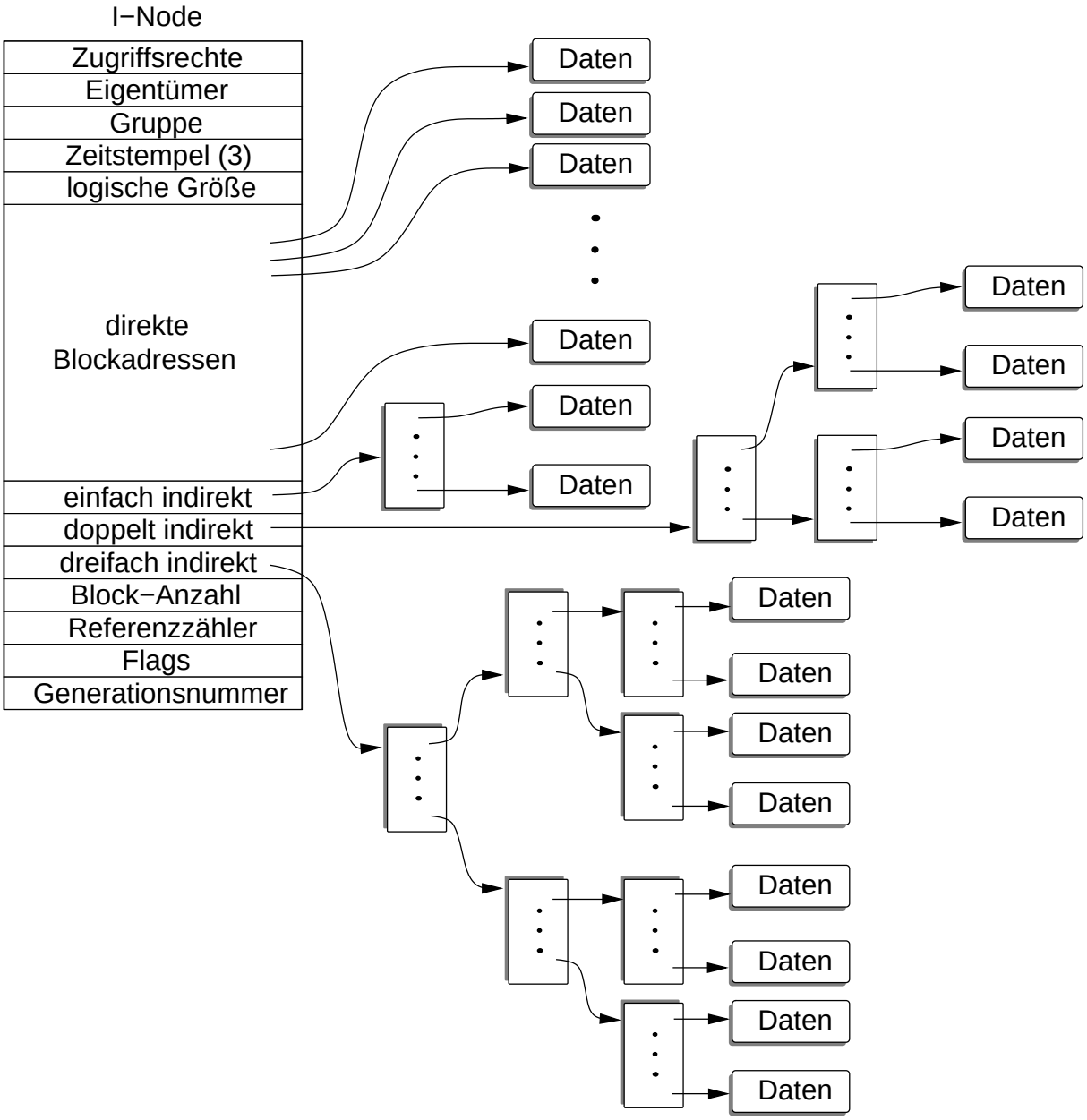
Der **Superblock** enthält die dateiübergreifenden Verwaltungsdaten, z.B.

- Größe des I-Node-Bereichs
- Größe des Datenbereichs
- Informationen über freie Cluster

Einige hundert Adressen freier Cluster stehen direkt im Superblock. Für die anderen Adressen werden Blöcke im Datenbereich als Indexblöcke verwendet.

Die **I-Node-Tabelle** enthält die dateispezifische Verwaltungsinformation. Für jede Datei wird ein I-Node angelegt, der als Dateideskriptor dient. Die Position eines I-Node in der Tabelle heißt I-Node-Nummer. (Das UNIX-Kommando `ls -i` zeigt die Nummern an.)


Superblock
direkte
Adressen
freier Blöcke
Index freier Blöcke
frei
frei
frei
frei
frei
frei
frei
frei
frei
frei
frei
frei



Man beachte, dass im I-Node kein Dateiname steht. Namen („Links“) werden Dateien (I-Nodes) auf der Verzeichnisebene zugeordnet, sie sind quasi Verweise auf Dateien. Beliebig viele Pfade innerhalb eines Dateisystems können auf den selben I-Node und damit auf die selbe Datei verweisen.

Im I-Node wird mitgezählt (Referenzzähler), wie viele Links zur Datei existieren. Dies ist beim Löschen wichtig: Wenn der letzte Link gelöscht ist, können I-Node und Datenblöcke der Datei freigegeben werden.

Verzeichnisse sind Tabellen bzw. Listen, in denen pro Eintrag der Dateiname und die zugehörige I-Node-Nummer steht.

8.3.3 NTFS

Bei Windows NT Dateisystemen (NTFS=„NT File System“) heißt die Dateideskriptortabelle MFT („master file table“), ein Dateideskriptor ist ein „MFT record“. Ein Dateisystem wird „volume“ genannt.

Die dateiübergreifenden Metadaten sind auch als Dateien organisiert. In der MFT findet man deshalb auch je einen Deskriptor für die Verwaltungsdateien.

Deskriptoren für Meta-Dateien:

Nummer	Verwaltungsdatei	
0	MFT	
1	MFT-Kopie	partielle Kopie, falls MFT kaputt
2	Log-Datei	Log-Datei für Verzeichnis- und Metadatenänderungen zur Konsistenzsicherung bei Systemabsturz
3	Volume-Datei	Volume-Header
4	Attributdefinitionen	für benutzerdefinierte Erweiterungen
5	Root-Verzeichnis	
6	Bitmap	Freispeicher-Bitmap (vgl. **??**, S. **??**)
7	Boot-Code	zum Laden des Betriebssystems
8	Defekte Blöcke	

Eine Datei wird durch eine Dateireferenz identifiziert, die aus dem Index des MFT-Eintrags (entspricht der UNIX-I-Node-Nummer) und einer Generationsnummer besteht, die bei Wiederverwendung einer Deskriptornummer inkrementiert wird.

Interessant ist die für die Zukunft vorgesehene Möglichkeit, Dateideskriptoren um benutzerdefinierte Dateiattribute zu erweitern. Diese werden in der Attributdefinitionsdatei spezifiziert.

Der „Inhalt“ einer Datei (repräsentiert durch die Datenblöcke) wird als Dateiattribut aufgefasst. Dieses Attribut wird „Stream“ (Datenstrom) genannt, da NT genau wie DOS oder UNIX nur unstrukturierte Byteströme unterstützt. Eine Datei kann mehrere Streamattribute, also mehrere Inhalte haben.

Der Standarddatenstrom hat keinen Namen, weitere Streams, falls vorhanden, haben eigene Bezeichnungen, die man beim Öffnen zusätzlich zum Dateinamen angeben muss. Unter Windows wird das Konzept bislang nicht genutzt.

Dateiattribute

Standardattribute	DOS-Attribute
Attributliste	Verkettungsinformation, falls eine Datei mehr als einen MFT-Eintrag belegt
Name	mehrere Namensattribute sind möglich, z.B. ein langer Name und ein DOS-konformer Name
Sicherheitsdeskriptor	Zugriffsrechte
Datenstrom	(siehe oben)
Indexinformation	nur für große Verzeichnisse, gibt Aufschluss über Suchindex (als B+-Baum implementiert)

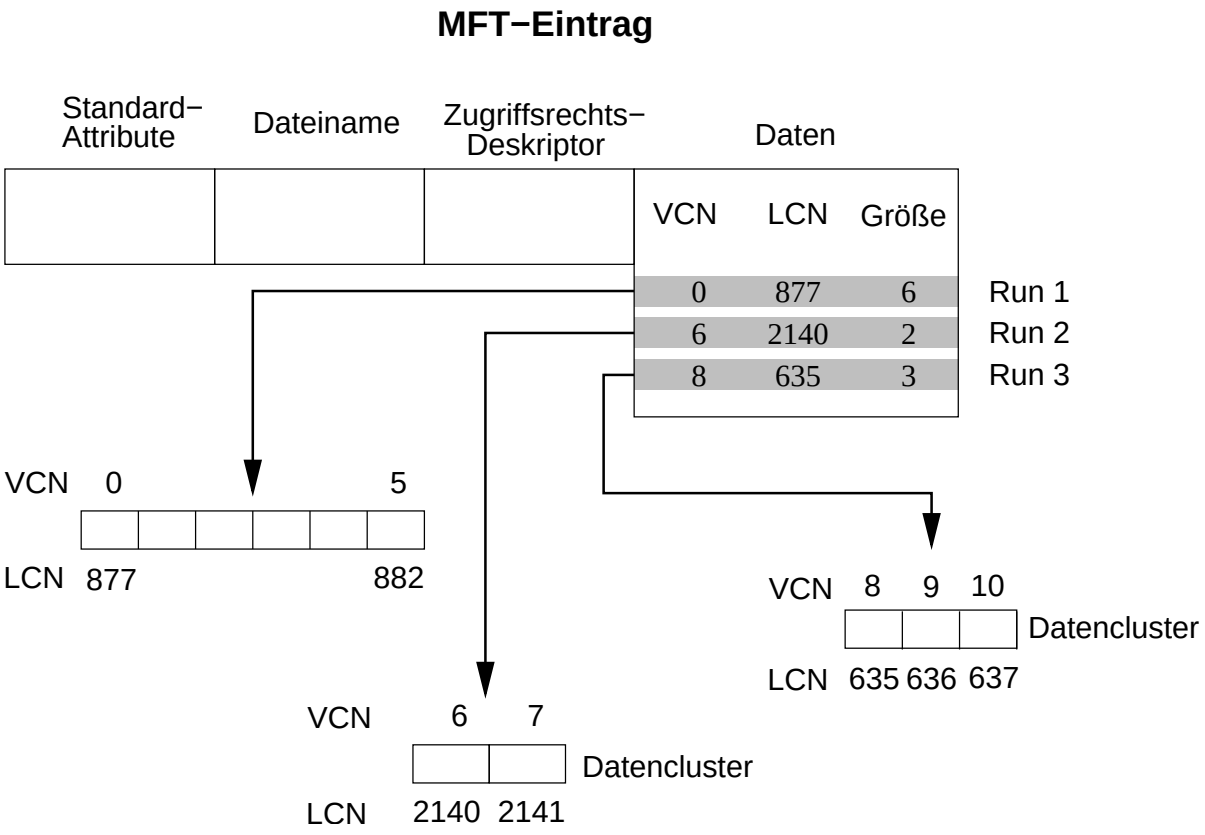
Die Anzahl der Attribute ist variabel, die MFT-Sätze haben aber eine feste Größe. Paßt ein Deskriptor nicht in einen MFT-Satz, wird er auf mehrere MFT-Einträge verteilt.

Datenstrom und Indexattribute (also die Inhalte von Dateien und Verzeichnissen) werden, falls Sie nicht in den MFT-Eintrag passen, außerhalb der MFT abgespeichert (wie bei DOS und UNIX auch).

Im einfachsten Fall passt eine sehr kleine Datei mit allen Attributen, also auch dem Datenstrom, in ihren MFT-Satz. Dann gibt es keine separaten Datenblöcke, der Zugriff ist aüßerst effizient (UNIX behandelt symbolische Links auch auf diese Weise).

In der Regel werden Dateien auch bei NTFS durch im Dateisystem verstreute Cluster repräsentiert. Anstelle einer Cluster-weisen Buchführung über die Daten nutzt NTFS die Erfahrung, das bei vielen Dateien physikalisch zusammenhängende Cluster genutzt werden können. Eine Folge zusammenhängender Cluster, die zur selben Datei gehören nennt man einen „Run“. Wenn eine Datei aus 20 Clustern besteht, benutzen DOS und UNIX jeweils 20 Indexeinträge, um die Cluster zu finden. Bei NTFS hängt dies dagegen von der Anzahl der Runs ab. Im schlimmsten Fall sind alle Cluster unzusammenhängend, dann werden auch hier zwanzig Indexeinträge benötigt, jeder verweist auf einen Run der Länge 1. Im günstigsten Fall ist die Datei zusammenhängend gespeichert, dann genügt ein Verweis auf den einzigen Run (Anfangsadresse und Clusteranzahl).

Wir betrachten ein Beispiel mit einer Datei, deren 11 Cluster auf 3 Runs aufgeteilt sind:



VCN, die virtuelle Clusternummer, ist die Clusternummer innerhalb der Datei. LCN, die logische Clusternummer, ist die Clusternummer innerhalb des Dateisystems.

8.4 Sicherheitsaspekte

Vor allem im Zusammenhang mit dem Dateisystem spielen Sicherheitsaspekte eine wichtige Rolle. Sicherheit ist ein allerdings ein facettenreicher Begriff, dazu gehören Maßnahmen zum Schutz vor Datenverlust genauso wie Abschirmung gegen unerwünschtes Ausspionieren oder gar Sabotieren der Datenbestände.

Ursachen für Datenverluste gibt es viele, von Fehlbedienungen über Hard- und Softwarefehler bis hin zur Zerstörung von Speicherungsmedien durch Feuer oder hungrige Nagetiere.

Gegen Datenverluste hilft nur eine gute Sicherungsstrategie. Gegen unerlaubten Zugriff auf Daten muss das Dateisystem entsprechende Mechanismen enthalten.

8.4.1 Identifikation und Authentifikation

Beim Login in einem Mehrbenutzersystem identifiziert sich der Benutzer mit einem eindeutigen Benutzernamen. Aus seiner Identifikation werden dann die Rechte zum Zugriff auf Daten abgeleitet. Diese Rechte können direkt an die Identifikation gebunden

sein, oder z.B. über Gruppenzugehörigkeiten des Benutzers indirekt über die Gruppenrechte.

Damit nicht jemand ohne Erlaubnis eine fremde Identifikation angeben kann, wird die Identifikation überprüft. Eine übliche Methode ist die Abfrage eines geheimen Passworts, auch die Analyse von eindeutigen biologischen Merkmalen (Mustererkennung für Fingerabdrücke, Pupillen, Stimmfrequenz usw.) wird schon stellenweise genutzt. Diese Überprüfung der Identität heißt Authentifizierung.

Ein Trend moderner Betriebssysteme (z.B. moderne UNIX-Systeme, Windows NT) sind frei austauschbare Authentifikationsmodule. Während früher die Authentifikation ein „fest verdrahteter“ Algorithmus war, kann nun der Rechnerbetreiber abhängig von seinen Sicherheitsanforderungen zwischen unterschiedlichen Authentifikationsmethoden wählen.

8.4.2 Zugriffsdomänen, Capabilities, ACLs

Ein System verwaltet Objekte, die geschützt werdem müssen (Hardware, Speichersegmente, Dateien u.a.). Jedes Objekt hat eine Reihe von Zugriffsoperationen, z.B. Lesen, Modifizieren, Löschen. Eine Zugriffsdomäne ist eine Menge von Paaren der Form (Objekt, erlaubte Operation). Zu jedem Prozess gehört eine Zugriffsdomäne, die genau definiert, welche Objekte der Prozess in welcher Weise verarbeiten darf. Ein Objekt kann gleichzeitig mehreren Zugriffsdomänen angehören.

Beispiel: Bei UNIX oder NT ist die Zugriffsdomäne eines Prozesses durch seine Benutzeridentifikation und Gruppenzugehörigkeit exakt definiert. Es wäre möglich, eine komplette Liste aller Geräte und Dateien aufzustellen, die der Prozess verarbeiten darf, zusammen mit der Information über die Art der erlaubten Operationen. (NT nennt die Datenstruktur mit Benutzeridentifikation und Gruppenzugehörigkeit „access token“). Man beachte, dass mit einem Systemaufruf ein Domänenwechsel verbunden ist: innerhalb des Systemkerns hat der Prozess erweiterte Rechte.

Andere Betriebssysteme haben komplexere Zugriffsdomänenkonzepte.

In einer Zugriffsrechtsmatrix könnte man zu jedem Objekt und jeder Zugriffsdomäne die erlaubten Operationen angeben, z.B.

	Objekt				
Domäne	Datei A	Datei B	Datei C	Drucker	ISDN
1		read, write		write	read, write
2	read, execute		write	write	
3	read, execute	read			read, write

Die Speicherung der Zugriffsrechtsmatrix ist in der Regel nicht sinnvoll bzw. nicht praktikabel, da sie sehr groß und nur dünn besetzt ist.

Wenn man nach einer Technik zur kompakten Abspeicherung sucht, kann man zeilen- oder spaltenorientert vorgehen. Zeilenorientierte Speicherung bedeutet, dass eine Zu-

griffsdomäne explizit als Datenstruktur angelegt wird, also eine jeweils Liste von Paaren (Objekt, Operationen). Man nennt diese Listen „Capability-Listen“ und einen Eintrag „Capability“.

Capability-Listen müssen gegenüber dem Benutzerprozess gegen unerlaubte Modifikation geschützt werden, damit dieser sich nicht durch Erweiterung Rechte verschafft, die ihm nicht zustehen. Dazu müssen sie im Betriebssystemadressraum gehalten (Hydra), mit kryptographischen Methoden geschützt werden (Amöba) oder mit spezieller Hardware unterstützt werden, die Capability-Objekte von modifizierbaren Objekten unterscheidet.

Spaltenorientierte Speicherung bedeutet, dass zu jedem Objekt eine Liste von Paaren (Zugriffsdomäne, Operationen) angelegt wird. Man nennt eine solche Liste Zugriffskontrollliste („access control list“) oder kurz ACL.

Windows NT verwendet beispielsweise ACLs. Jedes Objekt besitzt einen Zugriffskontrolldeskriptor mit einer ACL. Jeder ACL-Eintrag besteht aus einem Eintragsmerkmal „erlaubt“/“verboten“, einem Benutzer- oder Gruppennamen und einer Reihe von (erlaubten/verbotenen) Zugriffsoperation. Die Zugriffsverbote stehen am Anfang der Liste. „Wildcards“ für die Operationen sind möglich.

Windows NT Beispiel-ACL:

Typ	Identifkation	Rechte
verbiete	profs	Daten lesen Daten schreiben
verbiete	Meier	*
erlaube	WORLD	Datei ausführen
erlaube	Müller	Daten lesen

Die UNIX Dateizugriffsrechte kann man als eine stark vereinfachte ACL-Implementierung auffassen. Statt die individuelle Zusammenstellung von Zugriffsdomänen und Rechten zu erlauben, wird nur eine grobe Klassifikation der Zugriffsdomänen unterstützt: Objekteigentümer, Gruppenmitglieder, Sonstige. Dies ist viel einfacher zu implementieren und reicht in der Praxis meist aus. Eine Erweiterung auf richtige ACLs wäre aus heutiger Sicht sicher sinnvoll, ist aber aus Kompatibilitätsgründen problematisch. Bei viele modernen UNIX-Varianten (inkl. Linux) sind ACLs schon seit Jahren implementiert, genutzt werden sie in der Regel nicht.

8.5 Sonstige Aspekte

Für die Diskussion folgender weiterer Aspekte wird auf die weiterführende Literatur verweisen:

- Effizienz von Dateisystemen

- Konsistenz von Dateisystemen
- Wiederherstellung bei Systemabstürzen
- Datensicherung
- Verteilte Dateisysteme

Kapitel 9

Ein-/Ausgabe

Ein Betriebssystem muss sehr unterschiedliche Ein-Ausgabe-Geräte unterstützen, z.B. Drucker, Graphikkonsole, Maus, Netzwerkadapter, Platte, CD-Brenner, Scanner, Videokameras usw. Dazu kommen noch „Pseudo-Geräte“, die zwar nicht als Hardware-, sondern als Softwarekomponenten in das System integriert sind, aber in vieler Hinsicht wie echte Geräte benutzt werden. Beispiele sind RAM-Disks oder UNIX-Pseudo-Terminals.

Wir diskutieren zunächst einige Geräte-orientierte Aspekte der Ein-/Ausgabe, anschliessend die Anwenderschnittstelle zum E/A-System.

9.1 Hardware-orientierte Konzepte

Eine Hardware-Schnittstelle (auch „Port“) ist ein Anschlusspunkt, an dem ein Gerät angeschlossen werden kann. Beispiele: serielle Schnittstellen, Drucker-Schnittstelle.

Ein **Controller** ist der elektronische Bestandteil eines Geräts, der für die Kommunikation mit dem Rechner zuständig ist, genauer für

- Gerätekonfiguration, z.B.
  - Positionieren des Lese-Schreibkopfs eines Plattenlaufwerks
  - Einstellen eines Bildfensters beim Flachbettscanner
  - Setzen der Baud-Rate bei einer seriellen Schnittstelle
- Übermittlung von Statusinformation an den Rechner
- Datenübertragung zwischen Rechner und Gerät

Ein Controller kann für ein oder mehrere Geräte zuständig sein, für einen oder mehrere Ports oder für einen Bus. Typischerweise hat er ein Befehlsregister, ein Statusregister, ein Eingabedaten- und ein Ausgabedatenregister. Das Statusregister wird von der CPU gelesen, über das Befehlsregister wird das Gerät von der CPU gesteuert.

Ein **Bus** ist eine für mehrere Geräte benutzbare Schnittstelle zum Rechner. Zum Bus gehören die von den angeschlossenen Geräten gemeinsam benutzten (Daten- und Adress-) Leitungen und ein Protokoll, dass die Benutzung dieser Leitungen definiert.

Beispiele:

- Am PCI-Bus eines PC sind Prozessor/MMU/Hauptspeicher/Cache einerseits und verschiedene Controller, etwa Graphik-, Netzwerk- und SCSI-Controller (auch SCSI-Host-Adapter) angeschlossen.
- Am SCSI-Bus können verschiedene SCSI-Geräte angeschlossen werden, z.B. Platten, CD-Geräte, Scanner, Streamer. Die angeschlossenen Geräte besitzen eigene Controller.

Die Kommunikation zwischen CPU und einem Controller kann dadurch erfolgen, dass mit speziellen **E/A-Instruktionen** die Übertragung eines (oder mehrerer) Bytes zwischen CPU und einer E/A-Port-Adresse veranlasst wird.

Die andere Möglichkeit ist „**Speicher-Mapping**“. Dazu werden den Registern des Controllers Hauptspeicheradressen zugeordnet. Ein schreibender Zugriff auf diese Adressen wird für Befehls- und Datenübermittlung von der CPU an den Controller verwendet, lesender Zugriff für die Übertragung von Zustandinformation oder Daten vom Gerät an die CPU.

Manchmal werden vom selben Controller beide Techniken verwendet. Ein PC-Graphik-Controller wird über E/A-Ports kontrolliert, er benutzt aber auch Speicher-Mapping für den Bildschirminhalt. Man überlege sich, wie aufwändig es wäre, beim Bildschirm-Neuaufbau mehrere Megabyte vom Hauptspeicher über einen Port zum Graphik-Controller zu übertragen.

9.2 Polling

Polling ist eine Möglichkeit der Interaktion zwischen dem Rechner und einem Controller, bei der sich der Controller völlig passiv verhält. Wir betrachten als Beispiel schematisch eine Ausgabeoperation mit Polling:

- Der Rechner liest wiederholt das „busy“-Bit im Statusregister des Controllers, bis dessen Wert anzeigt, dass der Controller „bereit“ ist.
- Der Rechner überträgt ein auszugebendes Byte in das Ausgabedatenregister und setzt das „write“-Bit im Befehlsregister des Controllers
- Der Rechner setzt das „Ausführen“-Bit im Befehlsregister
- Der Controller erkennt, dass er einen Befehl ausführen soll und setzt das ‘busy“-Bit im Statusregister.

- Er liest den Schreibbefehl im Befehlsregister, liest das auszugebende Byte aus dem Datenausgaberegister und veranlasst die Ausgabe. Schliesslich setzt er das „Ausführen“-Bit im Befehlsregister und das „busy“-Bit im Statusregister auf Null zurück. Das „error“-Bit im Statusregister wird (bei erfolgreicher Ausführung) auch auf Null gesetzt.

Dies wird für jedes auszugebende Byte wiederholt.

Polling kann ineffizient sein. Wenn ein Controller z.B. nur einen sehr kleinen Puffer für Eingabedaten hat, muss der Rechner den Controller in einer hohen Frequenz pollen, um bei eintreffenden Daten einen Controller-internen Pufferüberlauf zu verhindern. Kommen nur selten Daten an, ist das Abfragen des Controllers meist umsonst. In vielen Fällen ist es daher besser, wenn der Controller von sich aus in der Lage ist, den Rechner zu verständigen. Dazu werden Interrupts verwendet.

9.3 Interrupts

9.3.1 Basismechanismus

Controller, die Interrupt-fähig sind, verhalten sich gegenüber dem Rechner nicht völlig passiv, sondern signalisieren dem Rechner von sich aus Zustandsänderungen. Die CPU erkennt die Interrupt-Signalisierung, unterbricht die Programmausführung und aktiviert eine Interrupt-Behandlungsroutine (ISR = „interrupt service routine“).

Die ISR prüft, welches Gerät den Interrupt erzeugt hat, bedient das unterbrechende Gerät und versetzt danach mit einer speziellen Instruktion die CPU wieder in den alten Zustand (Programmzähler und Registerinhalte). Für die Dauer der Interrupt-Behandlung werden andere Interrupts maskiert, d.h. sie führen nicht zur Unterbrechung. Nach der Behandlung werden sie wieder zugelassen.

Für moderne Rechnersysteme reicht ein solch einfacher Interrupt-Mechanismus nicht mehr aus. Man benötigt z.B. einen effizienten Mechanismus zur Aktivierung von gerätespezifischen ISRs, Interrupt-Prioritäten mit nebenläufiger Behandlung mehrerer Interrupts und andere Erweiterungen.

Der Interrupt-Mechanismus eines Systems muss neben den von Controllern bei Zustandsänderungen signalisierten Interrupts auch verschiedene maschinenspezifische Fehler- und Ausnahmesituationen behandeln. Dazu gehört auch die Behandlung von Seitenfehlern oder die Implementierung von Systemaufrufen (vgl. dazu **??**, S. **??**).

9.3.2 Interrupt-Vektoren und Interrupt-Verkettung

Bei vielen Systemen wird mit Interrupt-Vektoren die effiziente Auswahl einer gerätespezifischen ISR unterstützt. Der Interrupt-Vektor ist eine Tabelle mit ISR-Adressen. Der Vektor hat eine feste Hauptspeicheradresse und eine feste Größe.

Beim Signalisieren eines Interrupt wird von der signalisierenden Hardware eine Adresse übermittelt, die als Index für den Interrupt-Vektor dient. Im Vektor steht an der durch den Index definierten Stelle die Adresse der ISR, die den Interrupt behandelt.

Das Betriebssystem implementiert die ISRs und verwaltet die Vektor-Einträge. Die Aktivierung der ISR erfolgt auf der Hardware-Ebene. Innerhalb der ISR entfällt also die softwaremäßige und damit relativ langsame Ermittlung der Interrupt-Quelle.

Oft gibt es mehr Geräte als der Interrupt-Vektor Einträge hat. Dann kann nicht für jedes Gerät eine gerätespezifische ISR im Vektor eingetragen werden. In diesem Fall kann man Interrupt-Verkettung benutzen: Mehrere ISRs werden zu einer Liste verkettet und an den selben Vektor-Eintrag gebunden. Falls ein Interrupt mit dieser Eintragsadresse signalisiert wird, werden alle ISRs in der Liste nacheinander aufgerufen, solange bis eine ISR den Interrupt behandelt.

Dies ist ein Kompromiss zwischen einer einzigen ISR, die viel Aufwand treiben muss, um die Ursache des Interrupt zu ermitteln, und einem sehr großen Interrupt-Vektor, der Platz für alle gerätespezifischen und sonstigen ISRs bietet.

Kritische Abschnitte innerhalb des Systemkerns können durch Maskierung von Interrupts geschützt werden, oft wird zwischen maskierbaren und nicht maskierbaren Interrupts unterschieden. Ein Beispiel ist der Intel Pentium Prozessor, der 32 nicht maskierbare Interruptadressen (0-31) für verschiedene Hardware-Fehler und Ausnahmebedingungen hat, z.B.

- Divisionsfehler
- Seitenfehler
- Fließkommanfehler
- Ausrichtungsfehler
- ungültiger Opcode

Dazu kommen 224 maskierbare Interruptadressen (32-255) für gerätespezifische Interrupts.

9.3.3 Interrupt-Prioritäten

Mit Interrupt-Prioritäten werden die Interrupts nach Dringlichkeit der Behandlung sortiert. Jeder Interrupt hat eine bestimmte Priorität p Eine Service-Routine für einen Interrupt der Priorität *p*, ISR_p, kann durch einen Interrupt höherer Priorität *q* unterbrochen werden:

1. Anwenderprogramm wird ausgeführt
2. Anwenderprogramm wird unterbrochen, ISR_p wird ausgeführt
3. ISR_p wird unterbrochen, ISR_q wird ausgeführt

4. ISR$_q$ terminiert, ISR$_p$ wird weiterbearbeitet
5. ISR$_p$terminiert, Anwenderprogramm wird weiterbearbeitet

Da ISRs Datenstrukturen des Systemkerns modifizieren, ist bei der Programmierung Vorsicht geboten. ISR$_q$ darf nicht ohne weiteres dieselben Datenstrukturen manipulieren, die von ISRs geringerer Priorität benutzt werden. ISR$_p$ könnte beispielweise während der Manipulation einer verketteten Liste von ISR$_q$ unterbrochen worden sein, so dass die Liste sich während der Ausführung von ISR$_q$ in einem inkonsistenten Zustand befindet.

9.3.4 Software-Interrupts

Der Interrupt-Mechanismus eines Rechners wird für verschiedene Zwecke genutzt. Falls die Auslösung eines Interrupts programmiert wird, spricht man von einem Software-Interrupt.

Ein Beispiel dafür ist die Implementierung von Systemaufrufen mit „Traps“, die schon besprochen wurde.

Ein anderes Beispiel ist die Aufteilung einer ISR in einen dringenden und einen weniger dringenden Anteil.

Im Rahmen von ISR-Aktivierungen muss das Betriebssystem immer wieder Prozesszustände aktualisieren, aber auch Kontextwechsel durchführen. In **??**, S. **??** wurden schon Beispiele genannt:

- der Ethernet-Controller, der den Rechner vom Dateneingang verständigt, die ISR kopiert die Daten in den Hauptspeicher und veranlasst einen Zustandsübergang beim wartenden Empfängerprozess.
- Die Hardware-Clock, die den Rechner in regelmäßigen Abständen unterbricht, um dem Betriebssystem Gelegenheit zu geben, die Systemzeit zu aktualisieren und andere Takt-gesteuerte Aktionen auszuführen.

Betrachten wir als Beispiel den Interrupt eines Netzwerk-Controllers beim Empfang von Daten.

Die ISR muß im wesentlichen folgende Aktionen durchführen:

- Datenübertragung aus dem Controller in den Hauptspeicher
- Wecken des blockierten Empfängerprozesses
- Scheduler-Aufruf

Der Scheduler-Aufruf ist notwendig, weil durch den Zustandsübergang des Empfängerprozesses in den Zustand „bereit“ ein Kontextwechsel notwendig sein könnte. Um dies zu prüfen, werden die dynamischen Prioritäten neu berechnet und verglichen.

Der ISR-Aufruf darf nicht zu lange dauern, weil während des Aufrufs keine neuen Interrupts des Netzwerk-Controllers bearbeitet und damit keine weiteren Datenpakete empfangen werden können. Um einem Datenverlust vorzubeugen, wird die ISR aufgeteilt:

Dringlich ist die Übertragung der empfangenen Daten in den Hauptspeicher, weniger dringend dagegen der Aufruf des Schedulers. Die ISR kümmert sich also zunächst um den Controller. Statt aber die Scheduler-Funktion direkt aufzurufen, erzeugt die ISR einen Software-Interrupt mit geringerer Priorität, dessen ISR sich um das Scheduling kümmert. Der Software-Interrupt wird bearbeitet, sobald alle dringenderen ISRs beendet sind.

Eine zukunftsweisende Möglichkeit, ISR-Prioritäten zu implementieren, wird in dem Solaris-Betriebssystem von Sun verwendet. Solaris unterstützt die nebenläufige Ausführung von Threads im Systemkern („kernel threads“). Die Idee zur Implementierung eines prioritätengesteuerten ISR-Schedulers ist einfach und elegant: Man benutze den „normalen“ Scheduler.

Bei einem Interrupt wird für die Behandlung ein neuer Thread erzeugt. Der Thread hat eine sehr hohe Priorität, so dass die Programmbearbeitung durch den ISR-Thread unterbrochen wird. Falls jedoch während der Interrupt-Behandlung ein Interrupt mit höherer Priorität signalisiert wird, wird die aktuelle ISR zugunsten des neuen ISR-Thread verdrängt.

9.4 Programmierte Ein-/Ausgabe und DMA

Bei Geräten mit geringer Datentransferrrate wird der Datenaustausch zwischen Gerät und Hauptspeicher Byte für Byte von der CPU veranlasst und kontrolliert. Das Verfahren nennt man „Programmierte E/A“. Bei einem Interrupt-fähigen Controller wird dabei pro Byte ein Interrupt generiert.

Bei einem Gerät mit sehr schneller Datenübertragung, z.B. einer Festplatte, ist eine von der CPU ausgeführte Schleife zum byteweisen Transfer zwischen Gerät und Hauptspeicher Ressourcen-Verschwendung: Die teure CPU wird mit „primitiven“ Byte-Übertragungsoperationen ausgelastet, die ein wesentlich billigerer Spezialprozessor auch ausführen könnte.

Das Delegieren der Übertragung von der CPU an einen einfacheren Prozessor ist unter dem Begriff DMA („direct memory access“) bekannt: Der DMA-Controller sorgt anstelle der CPU für den Datentransfer vom bzw. zum Hauptspeicher. Er überträgt immer einen ganzen Block und generiert dann erst einen Interrupt.

Die CPU muss vor der Übertragung von Eingabedaten in den Hauptspeicher einen freien Speicherblock reservieren. Ziel- und Quelladresse für den Transfer, sowie die Anzahl der zu übertragenen Bytes werden in einer DMA-Request-Struktur gespeichert, deren Adresse dem DMA-Controller mitgeteilt wird. Während eines DMA-Transfers kann sich die CPU mit anderen Dingen beschäftigen. Sie kann zwar während dieser Zeit nicht auf den Hauptspeicher zugreifen, aber zumindest ihren Cache benutzen.

9.5 Applikationsschnittstelle zum E/A-System

Bei aller Gerätevielfalt versucht man, die Anwenderschnittstellen für den Gerätezugriff weitestgehend zu vereinheitlichen. Betrachten wie beispielsweise die UNIX-Funktion *write* zum Schreiben von Daten. Ein Aufruf der Funktion kann die unterschiedlichsten Ausgabeoperationen bewirken, unter anderem:

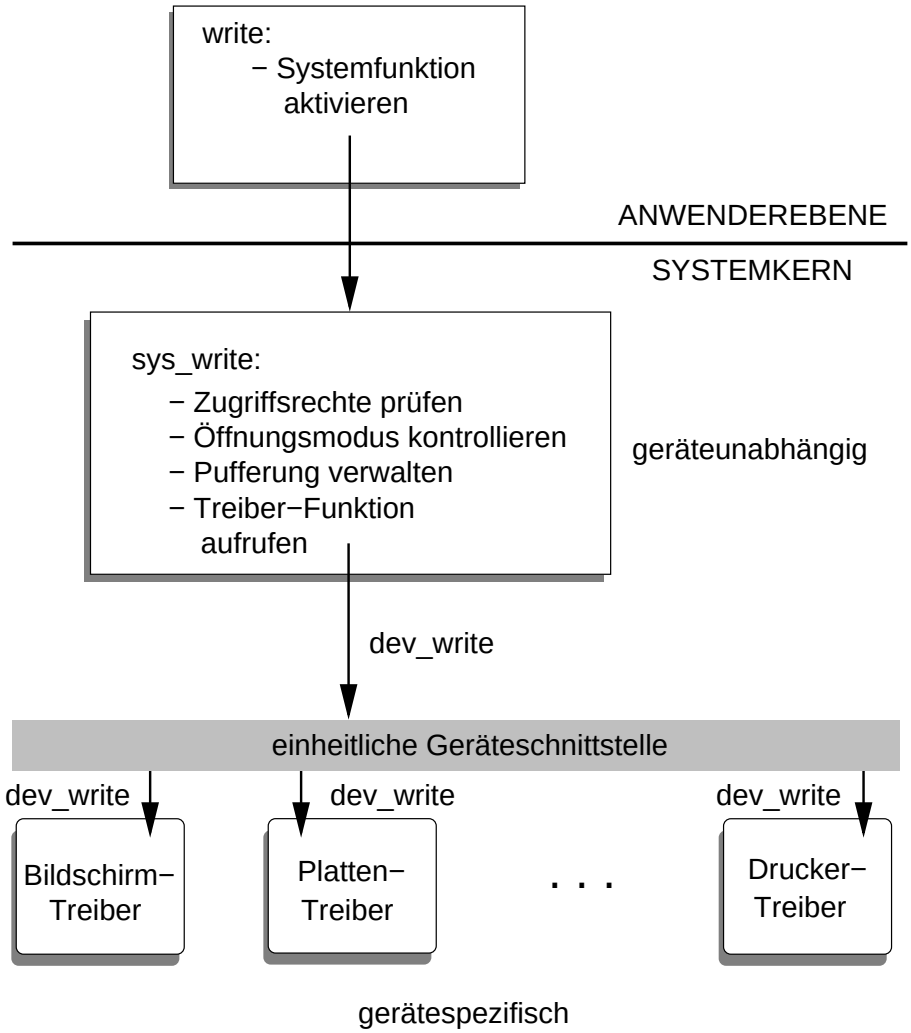
- Schreiben der Daten in eine Datei (auf einem beliebigen Speichermedium)
- Drucken der Daten
- Ausgabe der Daten auf einen Bildschirm
- Übertragung der Daten über ein Netzwerk
- Schreiben der Daten in die Partitionstabelle im Bootsektor einer Platte

Die Vereinheitlichung ist durch eine geschichtete Systemarchitektur möglich, in der geräteabhängige Teile der E/A gerätespezifisch als separate Module implementiert werden, die man als „Gerätetreiber“ (oder einfach „Treiber“) bezeichnet.

Ein Treiber ist ein Modul des Systemkerns, der ein Gerät (z.B. Graphik-Prozessor) oder mehrere Geräte desselben Typs (z.B. mehrere an einen Rechner angeschlossene Festplatten) kontrolliert.

Idealerweise implementieren alle Treiber eine einheitliche Schnittstelle für sämtliche E/A-Funktionen. In der folgenden Abbildung wird dieser Ansatz anhand eines schreibenden Gerätezugriffs illustriert:

- Das Anwenderprogramm benutzt die Bibliotheksschnittstelle *write*.
- Die *write*-Funktion aktiviert (z.B. per Trap) den Systemkern.
- Im Systemkern wird die Kern-Funktion *sys_write* aktiviert, um die Daten auszugeben. *sys_write* erledigt alle geräteunabhängigen Aufgaben direkt. Für geräteabhängige Teile der Ausgabe wird jeweils eine Treiber-Funktion aufgerufen. Alle Treiber haben die gleiche Schnittstelle, die u.a. die Funktion *dev_write* zur Datenausgabe beinhaltet. *sys_write* ruft *dev_write* auf, ein Multiplex-Mechanismus sorgt für die Zuordnung des zuständigen Treibers.



Die Kommunikation zwischen der Kern-Funktion, die den Systemaufruf bearbeitet, und dem Treiber erfolgt über ein Ein-/Ausgabe-Anforderungspaket. Das Paket ist eine Datenstruktur, die alle vom Treiber benötigten Informationen für die E/A-Operation enthält (Anzahl der zu übertragenden Bytes, Adresse, Gerät usw.). Die Anforderungspakete werden üblicherweise in gerätespezifischen Listen verwaltet. Ein Treiber muss die Pakete einer solchen Liste nicht notwendigerweise in FCFS Reihenfolge bearbeiten, sondern kann optimierende Umsortierungen vornehmen.

9.5.1 Block- und Zeichen-E/A

Beim UNIX-E/A-System wird zwischen Block- und Zeichen-orientierter E/A unterschieden („block/character devices“). Block-orientierte Geräte wie Platten-, Band-, Disketten oder CD-Laufwerke führen die Datenübertragung nicht byteweise, sondern immer blockweise durch, um die Effizienz zu steigern. Dabei werden die Blöcke im Hauptspeicher in einem *Cache*-Bereich gepuffert. Bei mehrfachen Lesezugriffen auf den selben Block innerhalb eines kurzen Zeitraums ist nur ein einziger Gerätezugriff bei der ersten

Leseoperation nötig. Anschließend stehen die Daten im Cache und Lesezugriffe sind Hauptspeicher-Zugriffe.

Beim Schreiben kann man genauso verfahren. Dateiänderungen werden nur im Cache wirksam, erst beim Schließen der Datei werden geänderte Blöcke auf den Datenträger zurückgeschrieben. Problematisch ist das Verfahren bei Stromausfall: Wenn von einer modifizierten Datei einige Blöcke auf den Datenträger zurückgeschrieben wurden, andere dagegen noch nicht, ist die Datei in einem inkonsistenten Zustand. Wenn es sich bei der Datei beispielsweise um das Root-Directory handelt, ist das gesamte Dateisystem inkonsistent. Einige UNIX-Systeme verzichten deshalb auf das Caching beim Schreiben („write through cache“). Andere sorgen durch Dämonprozesse dafür, dass in regelmässigen Abständen die modifizierten Cache-Blöcke auf den Datenträger kopiert werden. In der Praxis reicht das in Verbindung mit dem Einsatz einer USV (unterbrechungsfreien Stromversorgung) meist für einen genügend stabilen und dennoch effizienten Betrieb.

Prinzipiell vorzuziehen ist das bei Windows NT für das NTFS eingesetzte Verfahren: NT benutzt wie UNIX auch beim Schreiben den Cache. Allerdings wird vor jeder Änderung einer Metadatei ein Protokollsatz in die Log-Datei geschrieben, mit dessen Hilfe nach Ausfall des Systems beim Reboot der alten Zustand wieder hergestellt werden kann. Normale Dateien sind dadurch allerdings nicht gegen Inkonsistenz geschützt.

Bei UNIX implementieren alle Block-Gerätetreiber die gleiche Schnittstelle, die aus einer Reihe von Funktionen besteht. Dazu gehören Funktionen zur Bedienung von Anwender-Anforderungen (also Systemaufrufen), z.B.

- „Gerät initialisieren“,
- „lesen“
- „schreiben“

und Funktionen zur Bedienung von Geräte-Anforderungen (also Interrupts), z.B. „Eingabedaten entgegennehmen“.

Zeichen-orientiert sind alle nicht Block-orientierten Geräte, unter anderem Maus, Tastatur, Bildschirm, Netzwerkschnittstelle, serielle Schnittstelle, Graphik-Adapter usw. Auch für den ungepufferten Zugriff auf Plattensektoren gibt es einen Zeichen-orientierten Treiber.

Alle Zeichen-orientierten Treiber benutzen dieselbe Schnittstelle, die sich von der Block-Schnittstelle in einigen Punkten unterscheidet.

Auf die Treiber kann über die Dateisystemschnittstelle zugegriffen werden. Dazu sind spezielle Gerätedateien im Dateisystem vorhanden (`/dev/...`).

Nun kann die ganze Gerätevielfalt beim besten Willen nicht in Form zweier generischer Treiberschnittstellen komplett zugreifbar gemacht werden, man erreicht immerhin, dass diesen Operationen für „fast alle“ Gerätezugriffe ausreichen. Für die restlichen, nur für bestimmte Geräteklassen benötigten Operationen, verwendet UNIX den Systemaufruf *ioctl*, der soviele Unterkommandos hat, wie man gerätespezifische Spezialoperationen benötigt.

So wird z.B. die Schublade eines CDROM-Laufwerk wie folgt geöffnet: Der Prozess initialisiert den Treiber über die Standard-Schnittstelle (*open*-Syscall für die Gerätedatei und ruft dann *ioctl* mit dem Subkommando „Öffne Schublade“ auf. Dabei wird der von *open* zurückgelieferte Treiber-Deskriptor übergeben:

```
if((fd=open("/dev/scd0", O_RDONLY)) < 0 ) {
  perror("Fehler bei open");
  exit(1);
}
if(ioctl(fd, CDROMEJECT, NULL)) {
   perror("Fehler bei ioctl");
   exit(1);
}
exit(0);
```

Kapitel 10

Fallbeipiel: UNIX Prozess- und Signalkonzept

10.1 Grundkonzept

UNIX unterstützt Multitasking, in einigen Varianten auch Multithreading (z.B. Solaris). UNIX vergibt für jeden Prozess bei der Erzeugung eine eindeutige Prozessidentifikationsnummer (PID). Die Prozesse sind in einer strengen Eltern-Kind-Hierarchie angeordnet, alle Prozesse stammen direkt oder indirekt vom ersten Prozess mit der PID 1 (auch *init*-Prozess) ab.

Prozesserzeugung und Progammaufruf sind separate Operationen. Bei der Prozesserzeugung (*fork*) wird kein auszuführendes Programm angegeben. Stattdessen fertigt der Erzeuger einen „Klon“, also ein Duplikat von sich selbst an.

10.2 Wichtige Prozess-Operationen

10.2.1 fork – Prozess erzeugen

```
#include <unistd.h>

pid_t fork(void);
pid_t vfork(void);
```

Fork erzeugt einen neuen Prozess als *Subprozess* (auch *Kindprozess*). Der neue Prozess gleicht in fast jeder Hinsicht dem Erzeuger

- gleiches Programm
- identische Variablenwerte
- gleicher Programmzähler

- gleiche Dateien geöffnet
- gleiche sonstige Attribute (Eigentümer, Zugriffsrechte usw.)

Aus Sicht des Betriebssystems ist *fork()* eine Kopieroperation: Sowohl der Programmspeicher als auch der Prozesstabelleneintrag des Erzeugers werden kopiert.

Die wesentlichen Unterschiede zwischen Erzeuger und Subprozess:

- Der Subprozess hat eine neue PID
- der *fork()*-Resultatswert ist unterschiedlich:

$$fork - Resultat = \begin{cases} Subprozess: & 0 \\ Elternprozess: & Subprozess - PID \end{cases}$$

Bei Misserfolg (Hauptspeichermangel, Prozesstabelle voll) liefert *fork -1* als Resultat und die globale Variable *errno* enthält eine Codierung der Fehlerursache.

Es macht meist keinen Sinn, wenn Eltern- und Kindprozess sich identisch verhalten. Ein Subprozess wird i.d.R. erzeugt, um eine andere Aufgabe nebenläufig auszuführen. Dies muss durch eine explizite Fallunterscheidung anhand des unterschiedlichen *fork*-Resultats programmiert werden.

```
#include <ostream.h>
#include <unistd.h>
#include <stdio.h>

main(){
  pid_t kind_pid;  // ueblicherweise: pid_t = "unsigned long"
  pid_t meine_pid;

  printf("vor fork, PID=%d\n", meine_pid=getpid());
  switch (kind_pid=fork()) {
    case -1: perror("fork-Fehler");
             exit(1);
    case  0: printf("Kind: PID=%d\n", meine_pid=getpid());
             printf("Kind: Eltern-PID=%d\n", getppid());
             break;
    default: ("Elternprozess: nach fork, Kind-PID=%d\n", kind_pid);
  }

  printf("ich mache Feierabend: meine PID=\d\n", meine_pid);
  exit(0);
}
```

Man beachte, dass die Ausgabe hinter der *switch*-Anweisung nicht mehr vom *fork*-Resultat abhängig ist und damit von beiden Prozessen ausgeführt wird. Bei der Ausführung hat die Variable *meine_pid* in beiden Prozessen unterschiedliche Werte, jeder Prozess hat seine eigene Kopie der Variablen.

Die Funktion *getpid* liefert die PID des Aufrufers, *getppid* („get parent process identification“) die seines Elternprozesses.

Die *perror*-Funktion gibt eine zum aktuellen *errno*-Wert zugeordnete Fehlermeldung aus (zusätzlich zu der Argument-Zeichenkette).

Die Ausgabe des Programms könnte so aussehen:

```
vor fork, PID=453
Kind: PID=454
Kind: Elternprozess-PID=453
ich mache Feierabend: meine PID=454
Elternprozess: nach fork, Kind-PID=454
ich mache Feierabend: meine PID=453
```

Natürlich sind andere Abfolgen möglich, die Bearbeitung ist schliesslich nebenläufig. Auch folgende Ausgabe ist denkbar:

```
vor fork, PID=453
Elternprozess: nach fork, Kind-PID=454
ich mache Feierabend: meine PID=453
Kind: PID=454
Kind: Elternprozess-PID=1
ich mache Feierabend: meine PID=454
```

Der Elternprozess bekommt den Prozessor zuerst, macht seine Ausgaben und terminiert. Der Kindprozess ist zum Waisenkind geworden. Waisenkinder werden vom Prozess 1 adoptiert. Deswegen gibt Prozess 454 als Elternprozess den *init*-Prozess 1 an.

vfork

Eine zweite Prozesserzeugungsoperation, *vfork*, ist für den häufig vorkommenden Fall gedacht, dass der Subprozess ein anderes Programm ausführen soll. Der Aufwand, den *fork* durch Kopieren des Programmspeichers verursacht, ist in diesem Fall nicht gerechtfertigt. Der Subprozess wechselt unmittelbar nach dem Erzeugen der Programmkopie in ein anderes Programm.

Der *vfork*-Aufruf verzichtet stattdessen auf das Kopieren. Eltern- und Kindprozess benutzen den gleichen Programmspeicher. Damit keine Wettbewerbsbedingungen auftreten, wird der Elternprozess solange blockiert, bis der Subprozess entweder ein anderes Programm ausführt oder terminiert.

10.2.2 exec – Programmaufruf

Mit den *exec*...-Aufrufen wird ein neues Programm ausgeführt. Im ersten Argument wird der Programmpfad angegeben, danach die dem Programm zu übergebenden Ar-

gumente.

Neben den Argumenten erhält das Programm auch eine „Umgebung“ (Environment), die aus beliebig vielen Definitionen der Form *NAME=Wert* besteht. Die so definierten Variablen heissen **Umgebungsvariablen**. Auf diese Umgebung kann das Programm mit der *getenv*-Funktion zugreifen, *getenv*(*NAME*) liefert den Wert der Umgebungsvariablen *NAME*.

Beim Programmaufruf wird kein neuer Prozess erzeugt, der Programmspeicher des Prozesses wird vielmehr durch das neue Programm ersetzt. Es gibt also **keine Rückkehr** in das aufrufende Programm nach dem Muster eines Funktionsaufrufs.

Mehrere *exec*-Varianten sind verfügbar:

```
#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path,
            const char *arg , ...,  char* const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int system(const char *command); (ANSI)
```

Werden dem neuen Programm *n* Argumente übergeben, sind diese bei *execl* einzeln ab Argumentposition 2 aufzuzählen. Die Liste ist mit `NULL` abzuschliessen.

Man beachte beim Programmaufruf folgende Konvention: Der erste an das Programm zu übergebende Parameter ist der Programmpfad.

Beispiel: Aufruf des C++-Compilers

- aus der Kommandozeile:

  ```
  g++ -g -o uebung-1 uebung-1.cc
  ```

- aus dem C-Programm:

  ```
  execl("g++", "g++", "-g", "-o",
        "uebung-1", "uebung-1.cc", NULL)
  ```

Wahrscheinlich wird *execl* scheitern und *-1* zurückliefern, weil der Pfad der Programmdatei nicht vollständig spezifiziert wurde und somit nur im aktuellen Verzeichnis nach `g++` gesucht wird. Bei *execl* ist der komplette Pfad anzugeben!

Bei *execlp* dagegen wird nach der Programmdatei in allen Verzeichnissen gesucht, die in der Umgebungsvariablen *PATH* aufgezählt sind.

Bei *execle* wird gegenüber *execl* als weiteres Argument die Umgebung des Programms explizit übergeben. Bei Benutzung der anderen *exec*-Funktionen wird das Betriebssystem die Umgebung des aktuellen Programms ohne Änderung in das neue Programm kopieren.

Bei *execv* und *execvp* wird im zweiten Argument die Adresse eines Argumentvektors übergeben, in dem die zu übergebenden Zeichenketten (als Zeichen-Pointer) enthalten sein müssen.

Beispiel:

```
char arg0[]="g++";
char arg1[]="-g";
char arg2[]="-o";
char arg3[]="uebung-1";
char arg4[]="uebung-1.cc";

char *argv[]={arg0, arg1, arg2, arg3, arg4, NULL};

execvp("g++", argv);
```

Sind die Argumente bei Programmierung des *exec*-Aufrufs bekannt, ist diese Form natürlich zu umständlich gegenüber den *execl*-Funktionen. Wenn aber der Argumentvektor dynamisch bestimmt wird, wie etwa innerhalb einer Shell (hier werden die Argumente aus der Kommandozeile ermittelt), kann *execl* nicht verwendet werden.

10.2.3 exit – Prozessterminierung

```
#include <stdlib.h>
void exit(int status);
```

Ein Prozess beendet sich mit dem *exit*-Aufruf. Das Argument wird dem Elternprozess zugänglich gemacht, der es mit einer *wait*- oder *waitpid*-Operation erfragen kann.

Konvention: Das *exit*-Argument ist 0 bei normaler Terminierung, bei unerwarteten Fehlern dagegen 1.

Neben *exit*-Aufrufen gibt es weitere Möglichkeiten zur Prozessterminierung:

- Wenn ein Programm seine Ausführung normal beendet, wird der Prozess auch ohne expliziten Terminierungsaufruf beendet.
- Für den Terminierungszeitpunkt kann der Programmierer die Ausführung von Terminierungsfunktionen („exit handler“) vorsehen. Diese werden mit `atexit` beim Betriebssystem registriert und dann ohne expliziten Aufruf im Programm bei Prozessterminierung aktiviert.

  Der Systemaufruf `_exit` terminiert ebenfalls den Prozess, dabei werden im Unterschied zu `exit` aber keine registrierten Terminierungsfunktionen ausgeführt.

- Wenn die C-Funktion *main* durch eine *return*-Anweisung an die Aufrufstelle zurückkehrt, wird dort ein exit-Aufruf durchgeführt.

  Die erste innerhalb eines Programms aktive Funktion wird typischerweise nicht vom Anwendungsprogrammierer selbst entwickelt, sondern automatisch aus einer Bibliothek oder in Form eines Standardmodul hinzugebunden.

  Stellen Sie sich z.B. vor, diese Funktion heißt `_main` und steht als Objektmodul in einer Datei namens `crt0.o` zur Verfügung, die der Linker standardmäßig zu jedem C-Programm hinzubindet. Innerhalb von `_main` werden z.B. die Standardeingabedatei und -ausgabedatei geöffnet, die Programmparameter gelesen und dann `main` mit diesen Parametern aufgerufen.

- Ein Prozess kann natürlich auch auf Grund besonderer Ereignisse vom Betriebssystem „gewaltsam“ beendet werden. Dies wird im Zusammenhang mit Signalen erläutert.

10.2.4 waitpid, wait – Warten auf Subprozessterminierung

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

Mit beiden Operationen erhält der Elternprozess Statusinformation über einen Subprozess. Mehrere Anwendungen von *waitpid* sind möglich:

- Ein häufige Anwendung ist das Warten auf Subprozessterminierung. Der Elternprozess kann durch den Aufruf auf die Terminierung eines bestimmten (oder auch des nächsten) Subprozesses warten.

  Ist der Subprozess zur Aufrufzeit schon terminiert, kehrt der Aufruf sofort zurück.

- Der Elternprozess kann durch spezielle Optionen auch vom Betriebssystem erfragen, ob ein Subprozess schon terminiert ist, ohne darauf zu warten.

- Schließlich gibt es noch spezielle Möglichkeiten für Kommandointerpretierer. Diese möchten neben der Terminierung ihrer Subprozesse weitere Zustandsänderungen kontrollieren können, z.B. ob ein Subprozess mit sogenannten „Job-Control“-Mechanismen suspendiert (vorübergehend angehalten) wurde.

Die Statusinformation ist in Form von Statusbits codiert, deren Position und Anzahl allerdings plattformabhängig ist. Daher benutzt man zur Statusprüfung die in `wait.h` definierten Makros, mit denen man z.B. erfragen kann, ob eine freiwillige Terminierung (exit) oder eine „gewaltsame“ durch ein Signal vorliegt. Je nachdem wird ein Teilbereich der Bitfolge dann als Exit-Wert oder als Signalnummer interpretiert.

Der ältere *wait*-Aufruf liefert gegenüber dem „moderneren“ *waitpid* nur sehr eingeschränkte Möglichkeiten und ist daher heute überflüssig.

Ein Beispiel:

```
int kind_status;
pid_t kind_pid;

switch (kind_pid=fork()) {
    case -1: perror("fork-Fehler");
             exit(1);
    case  0: .
             .
             .
    default: // Warten auf Subprozess-Ende
             waitpid(kind_pid, &kind_status, 0);

             // normale Terminierung mit exit ?
             if (WIFEXITED(kind_status))
               // normale Terminierung, exit-Argument ausgeben
               cout << "Subprozess-Status:" << WEXITSTATUS(kind_status) <<
               endl ;
             else if (WIFSIGNALED(kind_status))
               // Terminierung durch Signal, welches ?
               cout << "Subprozess durch Signal abgebrochen, Signalnummer:"
                    << WTERMSIG(kind_status) <<  endl ;
  }

  exit(0);
```

10.3 Signale

Der Signalmechanismus ist ein auf Anwendungsebene verfügbarer Mechanismus zum Signalisieren und Behandeln von Ereignissen, die für das Anwendungsprogramm von Interesse sind. Beispielsweise werden Hardware-Ereignisse teilweise in Form von Signalen an die Anwendungsebene weitergeleitet, wo der Programmierer dann eine spezielle Signalbehandlung vorsehen kann.

Im einfachsten Fall dienen Signale dazu, ein ausser Kontrolle geratenes Programm mit einer bestimmten Tastatureingabe (z.B. CTRL-C) abzubrechen.

Beispiele für Signal-auslösende Ereignisse:

- Tastatur-generierte Signale
- Systemaufrufe z.B. (kill)

- Hardwarefehler (z.B. Division durch Null, Speicherzugriffsfehler)
- Subprozessterminierung
- Ablauf des „Weckers“
- Beim Eintreffen dringender Nachrichten vom Netzwerk
- Bei Beendigung asynchroner Ein-/Ausgabe-Operationen

Ein Signal wird zunächst *generiert* und dann den Empfängerprozessen *ausgeliefert*. Bei Auslieferung eines Signals wird der reguläre Kontrollfluß des Empfänger-Prozesses für die *Signalbehandlung* unterbrochen. Nach der Signalbehandlung wird die reguläre Verarbeitung ggf. fortgesetzt.

Die Signalbehandlung erfolgt nach Maßgabe des Empfängerprozesses durch

- *Ignorieren* des Signals
- Ausführung einer vorher dem Signal zugeordneten *Signalbehandlungsfunktion*
- Ausführung einer *Default-Behandlung*, die für viele Signale mit dem *Abbruch* des Empfängerprozesses verbunden ist

Signalbehandlungen können durch andere Signale unterbrochen werden.

10.3.1 Signaltypen

SIGABRT	abnormale Terminierung (*abort*-Funktion)
SIGALRM	Alarm-Timer abgelaufen (*alarm*-Funktion)
SIGBUS	System-abhängiger Hardware-Fehler
SIGCHLD	Zustandsänderung eines Subprozesses
SIGCLD	Zustandsänderung eines Subprozesses **Vorsicht:** Signal aus Kompatibilitätsgründen von alten UNIX-Systemen übernommen, aber völlig andere Semantik als SIGCHLD
SIGCONT	Job-Control-Signal, Fortsetzen der unterbrochenen Verarbeitung
SIGEMT	System-abhängiger Hardware-Fehler
SIGFPE	Arithmetik-Fehler
SIGHUP	Signal für den Sitzungsleiter bei Unterbrechung der Verbindung zum Kontrollterminal. Signal für Vordergrundprozesse bei Terminierung des Sitzungsleiters
SIGILL	illegalle Maschinenoperation
SIGINFO	Terminalstatus-Abfrage
SIGINT	Interrupt-Taste (DELETE oder CTRL-C)
SIGIO	Asynchrones E/A-Ereignis
SIGIOT	System-abhängiger Hardware-Fehler
SIGKILL	nicht verhinderbarer Prozeß-Abbruch
SIGPIPE	Schreiben in eine Pipeline, Leser terminiert
SIGPOLL	E/A-Ereignis beim Pollen
SIGPROF	Profiling Timer abgelaufen
SIGPWR	USV-Signal
SIGQUIT	Terminal-Quit-Taste
SIGSEGV	unzulässiger Hauptspeicherzugriff
SIGSTOP	Job-Control-Signal, Unterbrechen der Verarbeitung
SIGSYS	ungültiger Systemaufruf
SIGTERM	Software-Terminierung (kill-Kommando Defaultwert)
SIGTRAP	System-abhängiger Hardware-Fehler
SIGTSTP	Job-Control-Signal, Unterbrechen der Verarbeitung (CTRL-Z)
SIGTTIN	Job-Control-Signal, Hintergrundprozeß will vom Kontrollterminal lesen
SIGTTOU	Job-Control-Signal, Hintergrundprozeß will auf Kontrollterminal schreiben
SIGURG	Out-of-band Nachricht über Netz
SIGUSR1	Benutzer-definierbar
SIGUSR2	Benutzer-definierbar
SIGVTALRM	Timer abgelaufen
SIGWINCH	Änderung der Terminal-Fenstergröße
SIGXCPU	CPU-Limit überschritten
SIGXFSZ	Datei-Limit überschritten

- *kill(pid, signo)* – Sendet Signal *signo* an einen Prozeß oder eine Gruppe von Prozessen *pid* (Signal-Broadcast)

  Testet, ob ein Prozeß existiert (*signo* = 0)
- *raise(signo)* – erzeugt ein Signal für den aufrufenden Prozeß
- *alarm(n)* – erzeugt nach n Sekunden ein Alarm-Signal für den aufrufenden Prozeß
- *abort* – erzeugt SIGABRT
- *setitimer* – erzeugt Timer-Signale für einen von drei Timern

10.3.2 `signal` – ANSI-C Signalbehandlung

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

Der Prototyp der Funktion wird leichter verständlich, wenn man ihn mit einer Typdefinition für eine Signalbehandlungsfunktion strukturiert:

```
typedef void (*signal_handler)(int)

signal_handler signal(int signalnum, signal_handler handler)
```

Eine Signalbehandlungsfunktion hat ein *int*-Argument und kein Resultat. *signal(n, handler)* definiert für ein Signal *n* eine (neue) Signalbehandlungsfunktion (*handler*) und liefert die aktuelle Signalbehandlungsfunktion als Resultatswert.

Nach dem *signal*-Aufruf wird beim Ausliefern des Signals *n* die Funktion *handler* aufgerufen und die Nummer des auslösenden Signals übergeben. Damit kann die gleiche Signalbehandlungsfunktion für mehrere Signale mit unterschiedlicher verwendet werden: Über das Argument ist eine Signal-spezifische Behandlung innerhalb der Signalbehandlungsfunktion möglich.

10.3.3 `sigaction` – Spezifikation der Signalbehandlung

Syntax:

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *action,
                         const struct sigaction *oldaction);

struct sigaction {
```

```
  void (*sa_handler)(int); /* Adresse Signalbehandlungsfunktion */
  sigset_t sa_mask;        /* zusaetzlich zu block. Signale     */
  int      sa_flags;       /* verschiedene Optionen             */
};
```

Der Aufruf definiert die zukünftige Behandlung des Signals *signo* gemäß *action* und liefert die Spezifikation der bisherigen Signalbehandlung von *signo* in *oldaction*.

Ist *action* ein NULL-Pointer, bleibt die Signalbehandlung unverändert.

Die mit `sa_flags` spezifizierbaren Optionen:

SA_NOCLDSTOP	(nur SIGCHLD) kein SIGCHLD Signal bei Unterbrechung eines Subprozesses
SA_RESTART	Wiederholung von unterbrochenen System-Calls
SA_ONSTACK	Umschaltung auf benutzerdefinierten Stack während der Signalbehandlung
SA_NOCLDWAIT	(nur SIGCHLD) keine Zombies, *wait*-Semantik wird geändert
SA_NODEFER	keine automatische Blockade des gerade behandelten Signals während der Signalbehandlung
SA_RESETHAND	zu Beginn der Signalbehandlung wird `sa_handler` auf `SIG_DFL` zurückgesetzt
SA_SIGINFO	zusätzliche Parameter an den `sa_handler` übergeben

Kapitel 11

Fallbeispiel: UNIX-Dateisysteme

11.1 Begriffe

Der Begriff „Dateisystem“ wird im Zusammenhang mit UNIX in einer der folgenden Bedeutungen verwendet:

1. **Komponente des Betriebssystemkerns**, die für die Verwaltung von Dateien und den Zugriff auf Dateien zuständig ist. Besteht aus „low-level“-Operationen (Cache, I-Node- und Datenblock-Verwaltung) und Systemaufrufen wie *open*, *close*, *read*, *write*, *link* usw.

   Wir werden nachfolgend von der *Dateisystem-Komponente des Betriebssystems* sprechen.

2. **Logische Struktur eines Speichermediums** (Plattenpartition, CD, Diskette), das zur Speicherung von UNIX-Dateien dient. In der englischsprachigen Literatur auch als *device* oder *logical device* bezeichnet. (Bei MS-DOS würde man von einem logischen „Laufwerk“ sprechen.)

   Wie werden nachfolgend den Begriff *Dateisystem* in genau diesem Sinne verwenden.

3. **Logische Sicht des Benutzers** auf eine streng hierarchisch gegliederte Menge von Dateien.

   Obwohl in der Regel die von der Anwenderebene aus sichtbaren Dateien auf mehrere Dateisysteme im Sinne von *logical devices* verteilt sind, sieht der Benutzer nur eine einzige Hierarchie (im Gegensatz zu MS-DOS, wo die logischen Laufwerke und damit die physikalischen Medien, die zur Speicherung dienen, auf der Anwenderebene sichtbar sind).

   Wir werden nachfolgend von *Datei-Hierarchie* sprechen. Die Einbettung mehrerer Dateisysteme in eine Datei-Hierarchie erfolgt flexibel durch Montage (*mount*) und Demontage (*umount*) (vgl. dazu auch die Abschnitte **??**, S. **??** und **??**, S. **??**).

1. **Komponente des Betriebssystemkerns**

für den Zugriff auf Dateien

2. **Logische Struktur eines Speichermediums**

   (z.B. Plattenpartition)

3. **Logische Sicht des Benutzers**

   auf hierarchisch strukturierte Menge von Dateien
   (mehrere Speichermedien und Dateiserver)

11.2 Dateisystem-Layout

In der Regel werden die Dateien, mit denen der UNIX-Anwender arbeitet, auf mehrere Speichermedien verteilt sein (typischerweise Fest- und Wechselplatten, Bänder, CDs, Disketten, WORM-Medien usw.). Wir betrachten im folgenden Platten, die Ausführungen sind auf andere Medien übertragbar (andere Medien werden allerdings nicht partitioniert).

Eine Platte wird zuerst formatiert und dann in mehrere **Partitionen** eingeteilt. Sowohl zum Formatieren als auch zum Partitionieren gibt es spezielle Programme (eine SCSI-Platte wird in der Regel beim Hersteller formatiert.) Die Partitionierung erfolgt z.B. unter Solaris mit dem *format*-Programm, unter Linux mit *fdisk*.

Jede Partition lässt sich unabhängig von anderen Partitionen in verschiedener Weise nutzen, z.B.:

- als **swap-Partition** im Dienste der Speicherverwaltung
- als **Dateiarchiv**, das von einem Archivprogramm wie *tar* oder *cpio* verwaltet wird,
- als **Datenbank**, die (ohne Betriebssystem-Unterstützung!) direkt von einem Datenbanksystem verwaltet wird,
- als **Dateisystem**.

Bereich	**Bezeichnung**	**Verwaltung durch**
Sektor 0	Boot-Code	Boot-Lader
Sektor 0	Partitionstabelle	Partitionierungsprogramm
Partition 1	Dateisystem	Dateisystemkomponente des BS
Partition 2	Swap-Partition	Speicherverwaltungskomponente des BS
Partition 3	Dateisystem	Dateisystemkomponente des BS
Partition 4	Datenbank	Datenbanksystem
Partition 5	Dateiarchiv	Archivierungsprogramm

Eine sinnvolle Planung des **Platten-Layout**, d.h. der Partitionierung, erfordert einiges Nachdenken über die zukünftige Plattennutzung, denn eine spätere Änderung ist immer aufwändig!

Partitionen werden als **spezielle Dateien** (vgl. S. **??**) angesprochen, z.B. `/dev/sda3` (sd=SCSI-disk, a=fortlaufende Kennung der vorh. Platten, 3=3. Partition der Platte a). Disketten und CDs sind in der Regel nicht weiter partitioniert.

Meist unterstützen UNIX-Syteme mehrere **Dateisystem-Typen**, die sich in der internen Organisation der Dateien, in der Zugriffs-Effizienz und in der Funktionalität (z.B. Zugriffsrechte, Dateinamen) erheblich unterscheiden können. Viele UNIX-Systeme können ISO9660- und MS-DOS-Dateisysteme verarbeiten, bei Linux findet man darüber hinaus eine ganze Reihe verschieden mächtiger UNIX-Dateisystem-Typen (minix, ext2, ext3, reiserfs usw.).

Die organisatorische Grundstruktur einer UNIX-Partition wird mit einem speziellen Programm initialisiert *(mkfs = „make file system“)*.

Ein einfaches Beispiel für ein „klassisches“ UNIX-Dateisystem-Layout wurde in **??**, S. **??** schon vorgestellt, ebenso wurde der Aufbau von Verzeichnissen und Dateideskriptoren (I-Nodes) schon erläutert (**??**, S. **??**).

11.3 Dateitypen

UNIX unterscheidet im Vergleich zu anderen Betriebssystemen nur einige wenige Typen von Dateien

Reguläre Dateien sind ASCII-Text-Dateien, ausführbare Programme, Datenbanken, Bitmap-Dateien usw. Genauer gesagt ist jede Datei, die nicht einer der nachfolgenden anderen Typen angehört, eine reguläre Datei.

Für das Betriebssystem ist eine solche Datei nichts weiter als eine Folge von Bytes (*stream*), irgendeine innere Struktur wird vom Betriebssystem nicht beachtet.

Während etwa VMS oder Grossrechner-Betriebssysteme das Konzept der ISAM-Dateien (*index sequential access method* – effizientes Auffinden von Datensätzen über einen eindeutigen *Schlüssel*) unterstützen, bleibt dies bei UNIX den Implementieren von COBOL-Compilern oder Datenbanksystemen überlassen.

Verzeichnisse (*directories*) dienen zur hierarchischen Strukturierung einer Menge von Dateien. Das Konzept ist von MS-DOS bekannt, auf die Implementierung wird weiter unten eingegangen.

Spezielle Dateien sind die Dateisystem-Schnittstellen zur Hardware. Ausserdem gibt es eine Reihe *virtueller Geräte* mit Sonderfunktionen, z.B. Ramdisks oder virtuelle Terminals.

UNIX unterscheidet *Block-orientierte* und *Zeichen-orientierte* Geräte. Bei ersteren erfolgt die Ein- und Ausgabe wegen der besseren Effizienz immer in grösseren Blöcken (z.B. 512 oder 1024 Byte), die Cache-Komponente des Betriebssystems übernimmt die Pufferung der Daten im Hauptspeicher. Block-orientierte Ein-/Ausgabe erfolgt z.B. bei Platten, CDs oder Disketten.

Zeichenweise Übertragung von und zu den Geräten wird beispielsweise bei Terminals bzw. Tastaturen, Mäusen, und Netzwerk-Schnittstellen verwendet.

Benannte Pipes (*FIFO-Dateien*) werden zur gepufferten (asynchronen) Kommunikation zwischen Prozessen verwendet.

Die Daten, die ein Prozess in eine Pipe schreibt, können von einem anderen Prozess aus der Pipe gelesen werden. Dabei blockiert das Betriebssystem automatisch den schreibenden Prozess bei voller bzw. den lesenden Prozess bei leerer Pipe. Da über den Pipeline-Operator "|" der UNIX-Shell eine bequeme Nutzung dieses Kommunikationsmechanismus möglich ist, wird sehr häufig davon Gebrauch gemacht.

Zur Kommunikation zwischen verwandten Prozessen genügen unbenannte Pipes, der gemeinsame Zugriff erfolgt durch Vererbung der Deskriptoren. Unbenannte Pipes haben keinen Verzeichnis-Eintrag. FIFOs oder benannte Pipes werden dagegen über einen den Kommunikationspartnern bekannten Pfad angesprochen.

Symbolische Links (*symbolic links*) sind indirekte Zugriffspfade für andere Dateien. Die genaue Semantik wird weiter unten beschrieben.

(Berkeley-)Sockets bilden einen Kommunikationsmechanismus für die Kommunikation von Prozessen im gleichen Rechner oder im Netzwerk.

Mit einem *socket*- und einem *bind*-Systemaufruf kann ein Prozess einen Kommunikationsport einrichten. Danach kann ein anderer Prozess Daten an diesen Kommunikationsport senden.

Für die Kommunikation innerhalb desselben Rechners werden Socket-„Dateien“ erzeugt, sogenannte UNIX-Domain-Sockets, die nach der Kommunikation wieder explizit gelöscht werden müssen.

Man beachte, dass Symbolische Links und Sockets nicht in allen UNIX-Varianten verfügbar sind.

11.4 Dateinamen, Verzeichnisse, Links, Symbolische Links

11.4.1 Syntax von Dateinamen und Pfaden

Ein **Dateinamen** besteht aus *n* beliebigen Zeichen, dabei ist *n* abhängig von der konkreten UNIX-Variante (typische Werte sind 32 oder 255).

Obwohl die Zeichen tatsächlich beliebig sind, sollte man diejenigen Sonderzeichen, die auf der Shell-Ebene eine Sonderbedeutung haben, nicht verwenden. Sonst ist ein Einschliessen des Namens in Apostrophe erforderlich.

Vergleiche hierzu die Shell-Beschreibung.

Ein **Pfad** repräsentiert einen Knoten im Datei-Hierarchiebaum. Syntaktisch ist ein Pfad eine Folge von Verzeichnisnamen, die durch je ein "/"-Symbol voneinander abgetrennt sind.

Absolute Pfade beginnen mit "/" und beziehen sich auf das Wurzelverzeichnis "/", z.B. `/usr/jaeger/src/test.c++`, **relative Pfade** beziehen sich auf das Prozess-spezifische, jederzeit frei definierbare **aktuelle Verzeichnis**, z.B. `src/test.c++`.

Besondere Namen sind "." und ".." für das aktuelle Verzeichnis bzw. das unmittelbar übergeordnete Verzeichnis.

11.4.2 Verzeichnisse und „hard links"

- Verzeichnisse dienen zur hierarchischen Strukturierung eines Dateisystems:

  in dem die Hierarchie repräsentierenden Baum sind die inneren Knoten Verzeichnisse
- Ein Verzeichnis wird als Datei realisiert, für die besondere Operationen existieren
- Ein Verzeichnis ordnet einer Menge von Dateinamen linkseindeutig Dateideskriptoren zu (N:1-Zuordnung)

  Die Deskriptoren werden durch die I-Node-Nummer repräsentiert
- Ein Verzeichniseintrag heißt auch „hard link"

Die Ausgabe des Kommandos „ls -ai" zeigt, wie der Inhalt der Verzeichnisdatei aussieht, dazu ein Beispiel:

```
430441 #dateisystem.tex#          430204 dateisystem.dvi
430164 .                          430206 dateisystem.html
527778 ..                         430203 dateisystem.log
430166 Makefile                   429439 dateisystem.out
430167 Makefile~                  430590 dateisystem.pdf
234331 bilder                     430205 dateisystem.ps
527912 dateisystem                430202 dateisystem.tar.gz
430437 dateisystem-folien.aux     430160 dateisystem.tex
430440 dateisystem-folien.dvi     430200 dateisystem.tex-html
430163 dateisystem-folien.log     430168 dateisystem.tex~
430165 dateisystem-folien.tex     430197 dateisystem.toc
430161 dateisystem-folien.tex~    430198 frames
430207 dateisystem-pdf.zip        430199 install.sh
430196 dateisystem.aux            430201 install.sh~
```

UNIX unterscheidet zwischen Dateien und deren Namen insofern, als die Zuordnung von Namen zu Dateien als eigenständiger Mechanismus implementiert ist (siehe hierzu Abschnitt **??**, S. **??**).

Eine Datei wird innerhalb eines Dateisystems repräsentiert durch ihren I-Node und die Datenblöcke. Jede Datei ist eindeutig durch Dateisystem und I-Node-Nummer identifizierbar.

Wozu braucht man noch Dateinamen und Verzeichnisse ?

Zum einen sind Namen für den Anwender besser merkbar als Nummern. Die hierarchische Gliederung hilft, den Überblick zu behalten, auch wenn man auf Tausende von Dateien zugreifen kann.

Ein anderer wichtiger Aspekt ist die Abstraktion von konkreten Platten, Controllern, Partitionen usw. Während bei MS-DOS eine Pfadangabe der Form `A:\INSTALL.EXE` die Information über das konkrete Gerät „Diskettenlaufwerk A“ beinhaltet, gibt es bei UNIX derartige Geräteabhängigkeiten nicht.

Wie ist die Zuordnung von Dateien und Namen realisiert ?

Ein Dateiname wird auch als *Link* bezeichnet. Jede Datei kann über einen oder mehrere Links angesprochen werden. Zu jedem Link gehört ein Verzeichniseintrag. Ein Verzeichniseintrag besteht aus dem Namen der Datei und der I-Node-Nummer, also dem direkten Verweis auf die Datenstruktur mit der Datei-Verwaltungs-Information.

Alle Links sind gleichberechtigt, es gibt keinen „Haupt“-Namen und keine „Neben“-Namen. Im I-Node wird ein Zähler für die Links geführt, die zu der Datei bestehen. Mit dem Systemaufruf *unlink*, den z.B. das Kommando *rm* benutzt, wird nicht etwa eine Datei gelöscht, sondern ein Link, also ein Verzeichniseintrag.

Allerdings ist eine Datei nicht mehr vernünftig benutzbar, wenn der letzte Link gelöscht wurde, so dass in diesem Fall tatsächlich die Datei gelöscht wird, sprich, der I-Node und die Datenblöcke zur anderweitigen Benutzung freigegeben werden.

Das Kommando

```
ls -l
```

zeigt für die als Parameter angegebenen Dateien bzw. für alle Dateien im aktuellen Verzeichnis neben anderen Informationen aus dem I-Node auch die Anzahl der vorhandenen Links an.

Links werden eingerichtet beim Erzeugen von Dateien und durch explizites Einrichten mittels *ln*-Kommando oder *link*-Systemaufruf. Verzeichnisse haben immer mehrere Links, denn sie sind im jeweils übergeordneten Verzeichnis eingetragen, im Verzeichnis selbst unter dem Namen ”.” und in jedem ihrer Unterverzeichnisse unter dem Namen ”..”.

11.4.3 Symbolische Links

Die „Links“, von denen im letzten Abschnitt die Rede war, also Verweise von einem Namen in einem Verzeichnis zu einem I-Node, werden zur Abgrenzung auch als *hard links* bezeichnet.

Im Gegensatz dazu sind symbolische Links besonders markierte Dateien, deren Inhalt ein *Pfad* ist. Damit verweisen solche Dateien (über einen Verzeichnis-Eintrag) indi-

rekt auf andere Dateien. Die Links werden vom Betriebssystem automatisch verfolgt, mehrfach indirekte Verweisketten sind möglich.

Symbolic Links können (im Gegensatz zu *hard links)* Dateisystem-übergreifend eingerichtet werden. Inkonsistenzen wie zyklische Verweisketten oder Links auf nicht existente Dateien können vom Betriebssystem **nicht** entdeckt werden !

Symbolische Links werden immer explizit angelegt, z.B. mit dem Kommando `ln` und der Option `-s`.

Abbildung **??**, S. **??** soll den Unterschied zwischen den beiden Verweis-Typen noch einmal illustrieren. Zu einer bereits vorhandenen Datei `/usr/jaeger/C++.text` wird ein *hard link* und ein symbolischer Link eingerichtet:

```
ln /usr/jaeger/C++.text /usr/mueller/cpp.docu
ln -s /usr/jaeger/C++.text /home/src/docs/C.doc
```

Die daraus resultierende Link-Struktur ist in der Abbildung wiedergegeben.

11.5 Montage von Dateisystemen

Wie bereits angesprochen, lassen sich mehrere Dateisysteme zu einer übergeordneten Dateihierarchie zusammenmontieren. In dieser übergeordneten Hierarchie sind konkrete Geräte oder Partitionen für den Anwender nicht mehr sichtbar.

Die Montage erfolgt mittels *mount*-Kommando bzw. -Systemaufruf, die Demontage mittels *umount*. Betrachten wir ein Beispiel:

Nehmen wir an, die Partition `/dev/sda3` wurde dem Ladeprogramm gegenüber als Wurzel-Partition deklariert. Nach der System-Initialisierung besteht die sichtbare Datei-Hierarchie zunächst nur aus den Dateien dieses Dateisystems.

Jetzt wird das Dateisystem auf der Plattenpartition `/dev/sda4` an das leere Verzeichnis `/usr` montiert, z.B. mit folgendem Befehl:

```
mount -t 4.2 /dev/sda4 /usr
```

Nach der Montage hat `/usr` zwei Unterverzeichnisse, `/usr/bin` und `/usr/local`, die Wurzel des Dateisystems `/dev/sda4` erscheint in der Gesamt-Hierarchie als `/usr`.

Die Montagestelle (*mount point*) ist ein beliebiges leeres Verzeichnis. Bei der Montage wird mit der Option `-t` der Typ des zu montierenden Dateisystems angegeben (im Beispiel 4.2, d.h. ein Berkeley-BSD4.2-Dateisystem). Welche Typen vom Betriebssystem unterstützt werden, ist Implementierungs-spezifisch. Unsere Sun-Rechner kennen die Typen *4.2*, *pcfs* (MS-DOS), *hsfs* (*High Sierra* – für CDs) und *nfs* (*network file system*).

Vergleiche hierzu auch die Beschreibung des mount-Systemaufrufs, Abschnitt **??**, S. **??**.

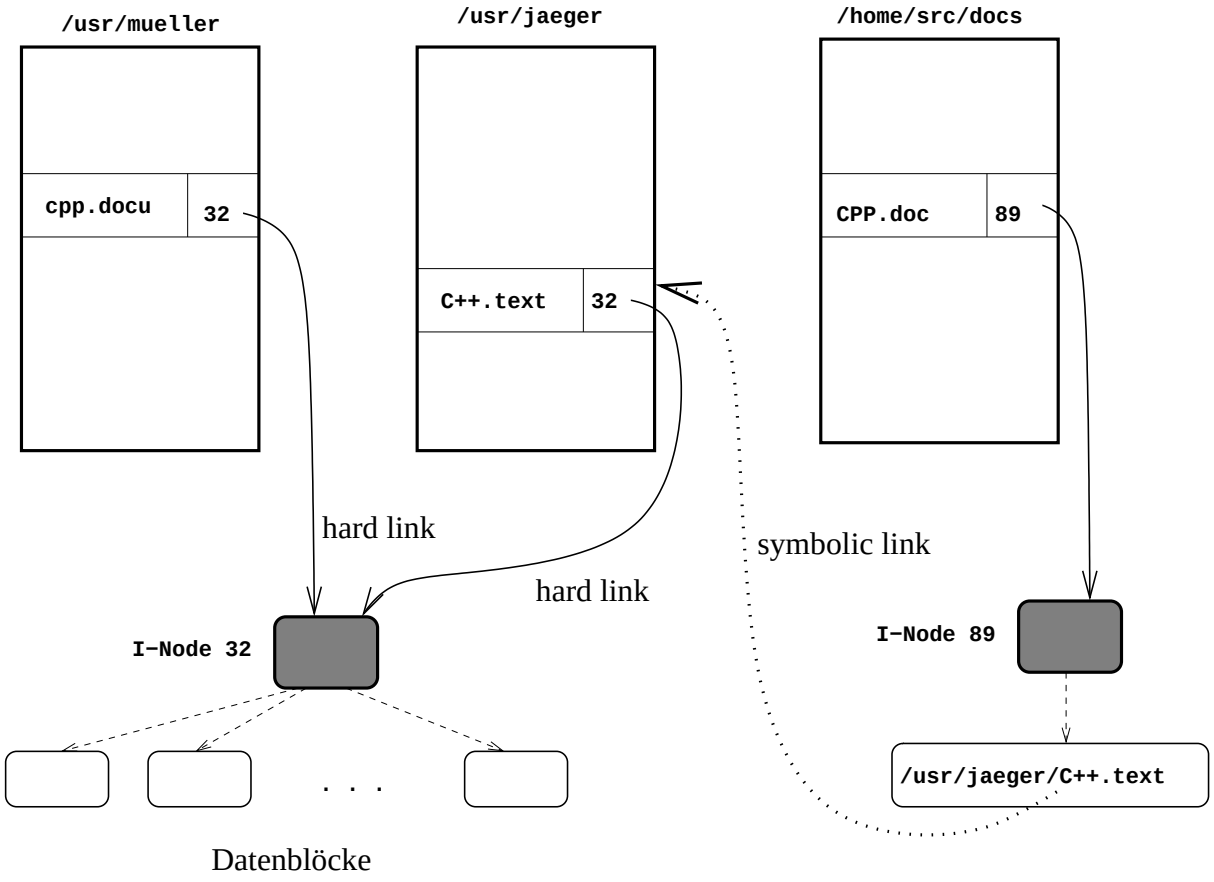


Abbildung 11.1: Verweis-Arten

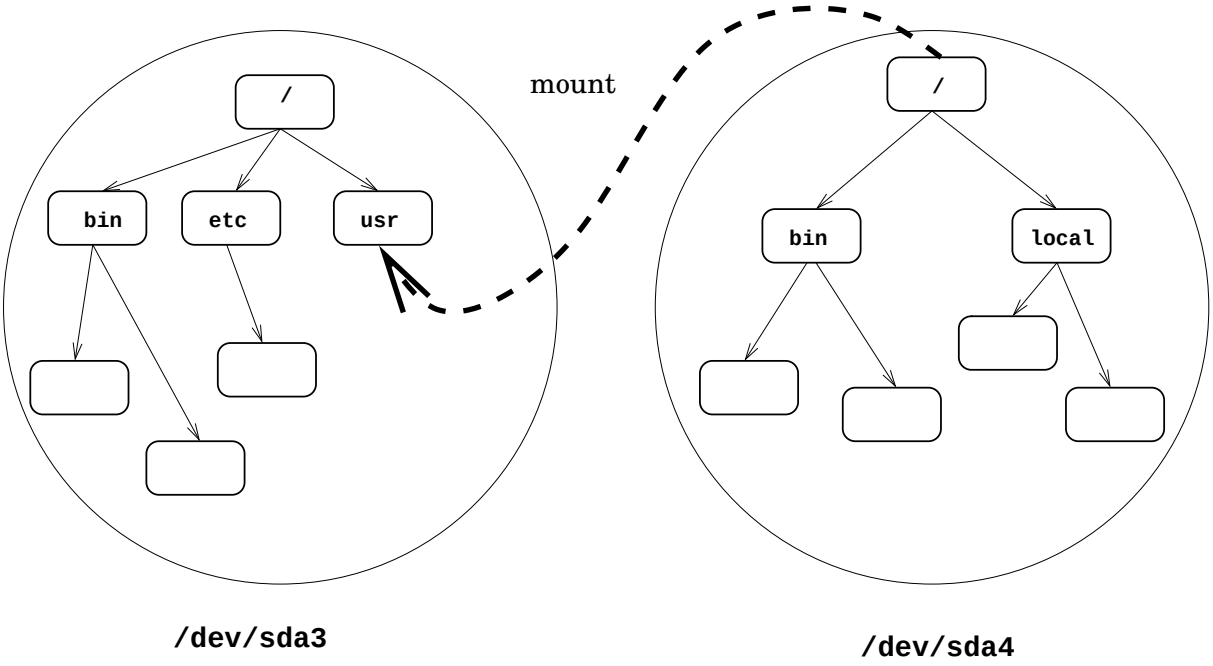


Abbildung 11.2: Dateisystem-Montage

11.6 Zugriffsrechte

Jede Datei gehört zu einem Benutzer (dem Eigentümer der Datei, auch *user*) und einer Benutzergruppe. Die Zugriffsrechte lassen sich für den Eigentümer, die Mitglieder der Gruppe und die sonstigen Benutzer separat festlegen.

Bei jedem dieser Personenkreise wird unterschieden zwischen „r“ - *read*, „w“ - *write* und „*x*“ - *execute*-Zugriff.

Dann gibt es noch den *Superuser* (user-id=0), der alles darf.

Die Rechte werden durch 9 entsprechende Bits (user-rwx, group-rwx, others-rwx) im I-Node der Datei repräsentiert.

11.6.1 Rechte für reguläre Dateien

- Mit **Lesezugriff** hat man das Recht, den Dateiinhalt zu lesen, bzw. die Datei zu kopieren.
- Mit **Schreibzugriff** kann man beliebige Modifikationen am Inhalt vorzunehmen. (Wie unten erläutert, reicht dazu aber auch Schreibzugriff auf das Verzeichnis !)
- Das **Ausführungsrecht** für eine Programmdatei oder Kommandoprozedur erlaubt deren Aufruf. Für nicht-aufrufbare reguläre Dateien ist das Recht nicht sinnvoll.

11.6.2 Rechte für Verzeichnisse

Bei Verzeichnissen ist die Semantik der Zugriffsarten anders definiert: Lesezugriff reicht nicht aus, um ein Verzeichnis zu durchsuchen bzw. anzuzeigen, dafür benötigt man das Ausführungsrecht.

Schreibzugriff gestattet Änderungen am Verzeichnis.

Vorsicht: Das Löschen einer Datei F in einem Verzeichnis D wird als Modifikation von D interpretiert. Schreibrechte für das Verzeichnis D sind erforderlich, für die zu löschende Datei F sind dagegen *keinerlei* Rechte notwendig !

Durch diese Interpretation ist natürlich auch eine Modifikation von F ohne Schreiberlaubnis möglich: Man löscht die Originalversion von F und kopiert eine modifizierte Version in das Verzeichnis D. Beide Operationen erfordern nur Schreibzugriff auf D.

11.6.3 Kommandos zur Zugriffsrechtskontrolle

Wichtige Kommandos im Zusammenhang mit den Zugriffsrechten sind:

`ls -l`	Datei-Information anzeigen
`chmod`	Zugriffsrechte ändern
`chown`	Eigentümer ändern (*change owner*)
`chgrp`	Gruppe ändern
`umask`	Maske für die Rechte neuer Dateien definieren (Shell-intern)
`find`	Dateien über Eigentümer, Gruppe, Zugriffsrechte und/oder andere Kriterien suchen

11.6.4 Besondere Zugriffsrechte

Neben den schon erläuterten 9 Zugriffsrechts-Bits gibt es drei weitere:

- Das **suid**-Bit (*set user id*) einer Programmdatei P verleiht dem Programm unabhängig vom Aufrufer die Rechte des Eigentümers von P.

  Dazu sollte man sich vor Augen halten, dass beim Aufruf zwei Benutzer-Identifikationen im Spiel sind:

  1. die Identität des Aufrufers, so wie beim *login* angegeben (**reale Benutzeridentifikation**), und
  2. die Identität des Eigentümers der Programmdatei.

  Man ruft nur im Einzelfall seine eigenen Programme auf, sodass beide Identitäten in der Regel unterschiedlich sind.

  Die Zugriffsrechte eines Programms sind durch die **effektive Benutzeridentifikation** bestimmt, in der Regel ist das die Identifikation des Aufrufers, d.h. die reale Benutzeridentifikation.

  Nur wenn in der Programmdatei des aufgerufenen Programms das *suid*-Bit gesetzt ist, ist die effektive Benutzeridentifikation diejenige des Dateieigentümers.

  Ein typisches Beispiel für die Verwendung ist das Programm *passwd*, das die Benutzer-Datenbasis `/etc/passwd` ändert.

  Es wäre fatal, wenn jedermann uneingeschränkte Schreibrechte für diese Datei hätte, man könnte sich jegliche Zugriffsrechtsprüfungen gleich ersparen. Andererseits soll es dem Benutzer erlaubt sein, jederzeit sein Passwort oder seine Login-Shell in der Datei zu ändern. Daher ist das *passwd*-Programm in einer *suid*-Datei dem Eigentümer *root*, d.h. dem *Superuser* zugeordnet. Das Programm hat also *Superuser*-Rechte und darf die Benutzerdatenbasis ändern.

- Das **guid**-Bit (*set group id*) einer Programmdatei P verleiht dem Programm unabhängig vom Aufrufer die Gruppenrechte der Gruppe von P.

- Das **Sticky-Bit** (*save text image*) wurde in älteren Systemen (mit *Swapping* als wichtigster Speicherverwaltungsmethode) benutzt, um ein Programm Speicherresident zu machen, d.h. dessen Auslagerung zu verhindern. Bei modernen *Paging*-Systemen spielt diese Verwendung keine Rolle mehr.

Für Verzeichnisse hat das Bit eine andere Semantik: Jeder Benutzer darf in dem Verzeichnis seine eigenen Dateien löschen aber nicht die Dateien anderer Benutzer.

11.7 Betriebssystem-interne Tabellen zur Dateiverwaltung

Neben der permanent gespeicherten I-Node-Tabelle (Kapitel **??**) verwendet UNIX drei Hauptspeicher-interne Tabellenformate zur Verwaltung der offenen Dateien:

Interne I-Node-Tabelle
Beim Öffnen einer Datei wird deren I-Node von der Platte in den Hauptspeicher kopiert, so dass nicht bei jedem Dateizugriff ein Plattenzugriff erforderlich wird (*I-Node-Cache*).

Die internen I-Nodes enthalten zusätzliche Felder, z.B.

- Identifikation des Dateisystems und I-Node-Nummer
- Referenz-Zähler für die Anzahl der Verweise aus der globalen Dateitabelle auf den internen I-Node

Globale Dateitabelle (*file table*)
Beim Öffnen einer Datei wird in dieser Tabelle ein Eintrag folgenden Formats angelegt:

Zugriffsmodus	Position	I-Node-Verweis	Referenz-Anzahl

Der I-Node-Verweis zeigt auf den internen I-Node. Der Referenz-Zähler gibt an, wieviele Verweise von Prozess-Dateideskriptoren auf den Eintrag existieren. Die Position wird beim Lese- oder Schreibzugriff jeweils inkrementiert, kann mit dem *lseek*-Systemaufruf jedoch jederzeit wahlfrei neu definiert werden.

Prozess-Deskriptoren-Tabelle
Jeder Prozess besitzt eine solche spezifische Tabelle mit Verweisen auf die von ihm benutzten Dateien. Ein Eintrag besteht aus einem Verweis in die globale Dateitabelle und dem „close on exec“-Bit, mit dem kontrolliert wird, ob der Deskriptor bei Aufruf eines anderen Programms geschlossen wird.

Abbildung **??** zeigt die internen Tabellen, nachdem ein Prozess *x* und ein Prozess *y* unabhängig voneinander die Datei `/etc/passwd` geöffnet haben. Zugriffsmodi und Offset sind unterschiedlich.

Anders sieht die Situation aus, wenn Prozesse mittels *fork*-Aufruf Subprozesse erzeugen. Hier wird die Deskriptoren-Tabelle des erzeugenden Prozesses für den Subprozess kopiert. Die Dateizugriffe sind danach wegen der gemeinsamen Verwendung des Offset in der globalen Dateitabelle nicht mehr unabhängig.

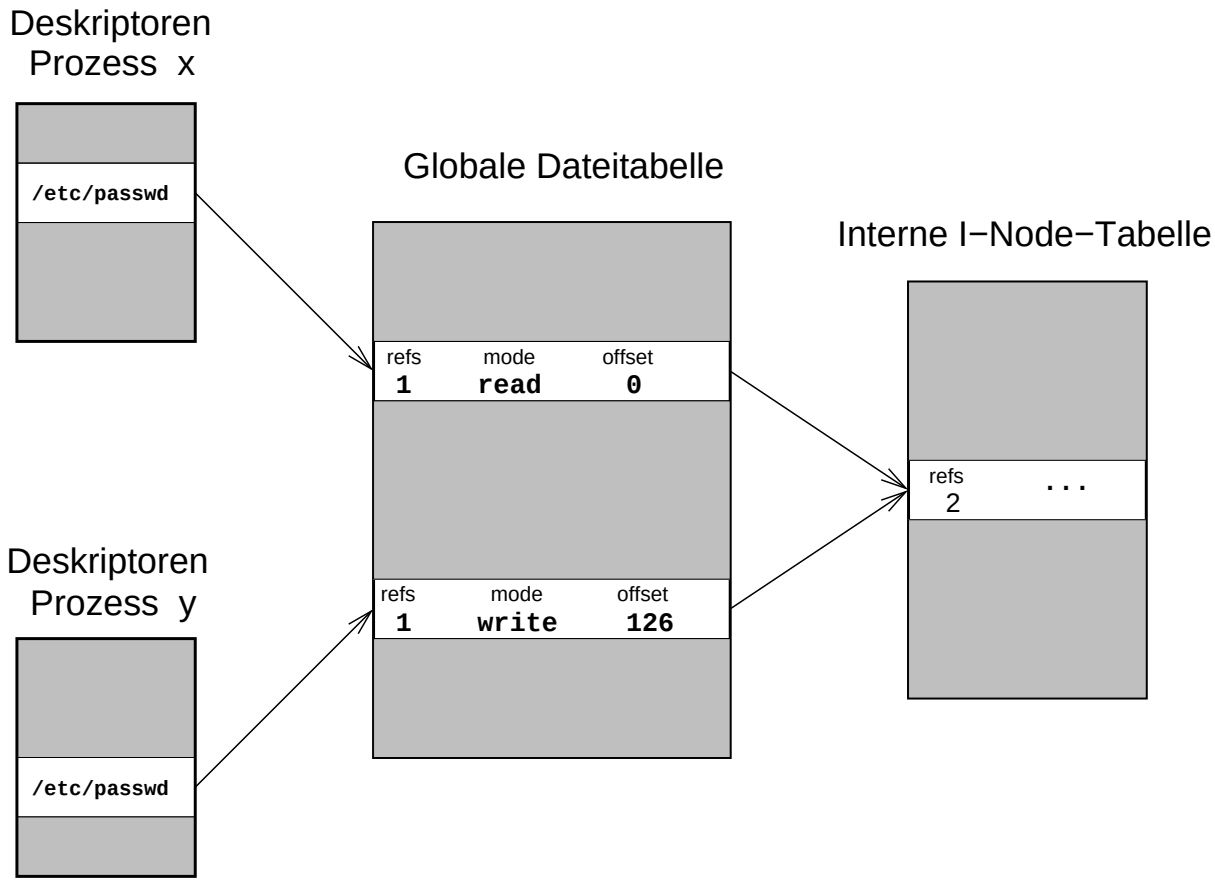


Abbildung 11.3: Interne Tabellen des Dateisystems - 1

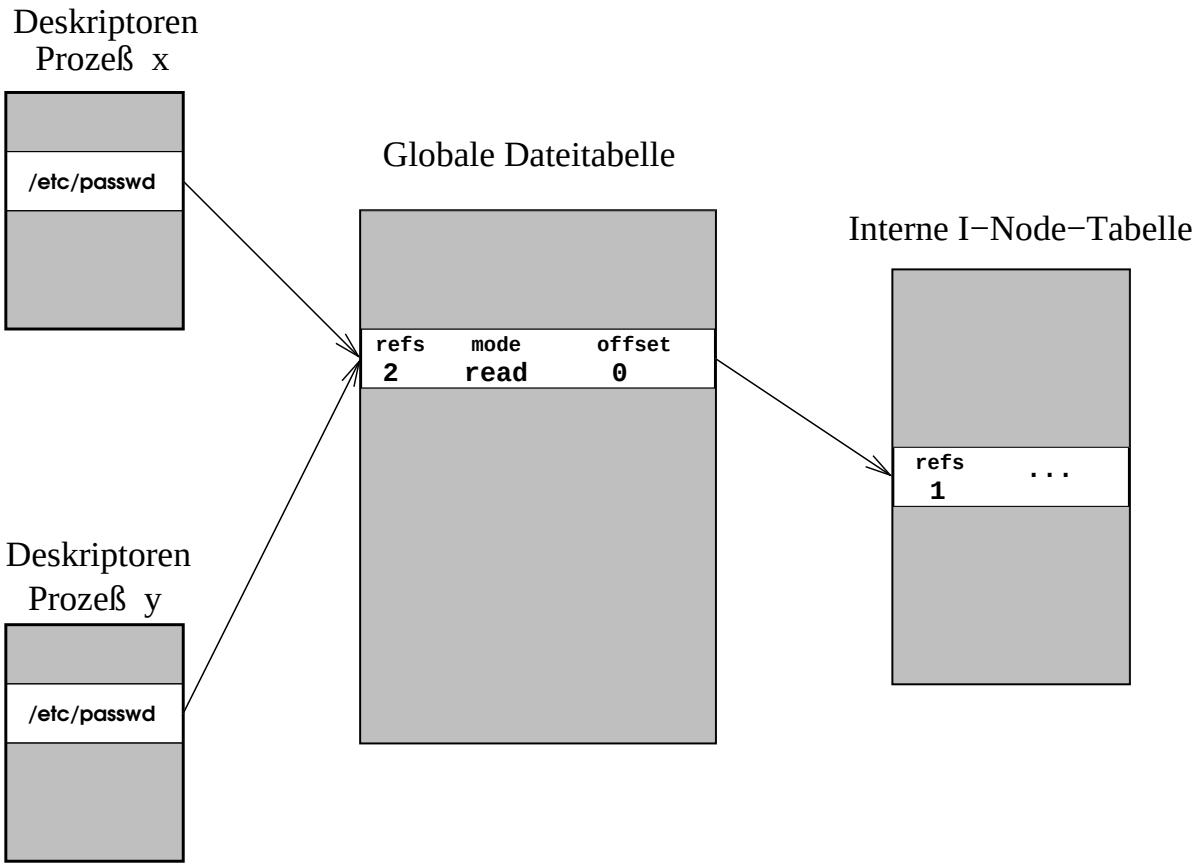


Abbildung 11.4: Interne Tabellen des Dateisystems - 2

Siehe dazu Abb. **??** (S. **??**).

Eine weitere Möglichkeit, mehrere Verweise auf den selben Eintrag in der globalen Dateitabelle zu erzeugen, ist der *dup*-Systemaufruf (Abschnitt **??**, S. **??**), der zur Ein- und Ausgabeumlenkung verwendet wird.

11.8 Wichtige Systemaufrufe des Dateisystems

11.8.1 Fehlerbehandlung

Allen Systemaufrufen ist gemeinsam, dass ein negatives Resultat einen Fehler signalisiert, dessen genaue Spezifikation durch den Wert der globalen Variablen **errno** gegeben ist. Eine vernünftige Fehlermeldung lässt sich mit **perror** erzeugen. Diese Funktion liest den zum Wert von *perror* gehörende Fehlermeldung aus der System-Meldungsliste.

11.8.2 Zugriffsrechtsprüfung

Bei jedem Systemaufruf werden die Zugriffsrechte überprüft. Das bedeutet insbesondere, dass auch nach dem Öffnen einer Datei bei allen Lese- und Schreibzugriffen immer eine separate Prüfung erfolgt, denn durch konkurrierenden Zugriff auf die gleiche Datei kann ja jederzeit eine Zugriffsrechts-Änderung, ein Eigentümer- oder ein Gruppenwechsel erfolgen.

11.8.3 Dateipfade

Einige Systemaufrufe verarbeiten Pfade (z.B. *open*). Diese Aufrufe verwenden die Funktion **namei**, die den Pfad verarbeitet und den I-Node der Datei als Resultat liefert. *namei* verarbeitet den Pfad Komponente für Komponente ausgehend vom aktuellen oder vom Wurzelverzeichnis.

Wurzel- und aktuelles Verzeichnis des ausführenden Prozesses sind in einer speziellen Datenstruktur des BS-Kerns, der *u-area*, gespeichert.

Für jede Verzeichnis-Komponente eines Pfads wird der Verzeichnis-I-Node reserviert, die Zugriffsrechte geprüft (Lesen *und* Ausführen), das Verzeichnis geöffnet und sequentiell nach dem benötigten Eintrag gesucht.

Zusätzlich muss für jede Pfad-Komponente anhand der Montagetabelle geprüft werden, ob es sich um einen Montagepunkt handelt. In diesem Fall wird die Weiterverarbeitung mit dem Wurzelverzeichnis des montierten Dateisystems fortgesetzt.

11.8.4 Übersicht

Nachfolgend werden einige ausgewählte Systemaufrufe kurz erläutert. Syntaktische Aspekte sind jeweils im Online-Manual nachzuschlagen.

open — Öffnen einer Datei

```
#include <fcntl.h>
int open (const char *file_name, int flag, ...);
```

Übergeben werden der Pfad der zu öffnenden Datei, der Öffnungsmodus (z.B. O_RDONLY, O_WRONLY, O_APPEND, O_RWWR) und als optionaler dritter Parameter noch die Zugriffsrechtsmaske, falls mit *open* eine neue Datei erzeugt werden soll.

Resultat ist eine Dateideskriptor (integer), der beim Lesen, Schreiben und anderen Operationen auf der Datei dann benutzt wird.

Open erzeugt je einen neuen korrekt initialisierten Eintrag in der globalen Dateitabelle und in der Deskriptor-Tabelle des Prozesses. Zur Zuordnung des I-Nodes zum Pfad wird *namei* verwendet.

read — Lesen aus einer Datei

```
int read(int fildes, char *buf, off_t count);
```

Argumente sind der Dateideskriptor (vgl. *open*), die Adresse eines Puffers für die zu lesenden Daten und die Anzahl der zu lesenden Bytes.

Resultat ist die Anzahl der gelesenen Bytes. Wurden weniger Bytes gelesen als angefordert, ist das Dateiende erreicht.

write — Schreiben in eine Datei

```
int write(int fildes, char *buf, off_t count);
```

Argumente sind der Dateideskriptor (vgl. *open*), die Adresse der zu schreibenden Daten und die Anzahl der zu schreibenden Bytes.

Resultat ist die Anzahl der geschriebenen Bytes. Wurden weniger Bytes geschrieben als angefordert, ist das Speichermedium voll.

lseek — aktuelle Lese-/Schreibposition ändern

```
off_t lseek(int fildes, off_t offset, int whence);
```

Der Aufruf ändert die aktuelle L/S-Position der durch *fildes* spezifizierten Datei. Die neue Position wird als relativer Byte-Offset *offset* angegeben, der sich anhängig vom 3. Parameter *whence* auf den Dateianfang (whence=SEEK_SET), das Dateiende (whence=SEEK_END) oder die derzeitige Position (whence=SEEK_CUR) bezieht.

Die neue Position relativ zum Dateianfang wird als Resultat zurückgegeben, so dass man die Funktion in der Form `lseek(fd, 0, SEEK_CUR)` auch zum Bestimmen der aktuellen Position verwenden kann.

Der Aufruf bewirkt nichts weiter als eine Änderung der im Dateitabellen-Eintrag gespeicherten aktuellen Position.

Durch Positionieren hinter das Dateiende und anschliessendes Schreiben, können in einer Datei beliebige **Lücken** entstehen. Für diese wird auf dem Speichermedium kein Platz reserviert. Beim Lesen einer Datei, die solche Lücken enthält, liefert der *read*-Aufruf Nullen für alle nicht mit *write*-geschriebenen Daten.

Der für eine Datei benötigte Speicherplatz kann also wegen solcher Lücken kleiner als die logische Dateigrösse sein.

close — Schliessen einer Datei

```
int close(int fildes)
```

Mit *close* wird die zu dem übergebenen Deskriptor gehörige Datei geschlossen.

Das Resultat ist i.d.R. Null.

Zunächst wird von *close* der Datei-Eintrag in der Deskriptortabelle gelöscht. Im zugehörigen Eintrag der globalen Dateitabelle wird der Referenzzähler dekrementiert. Hat der Zähler danach den Wert Null, wird auch der Dateitabellen-Eintrag gelöscht.

Da der Dateitabellen-Eintrag auf einen (Hauptspeicher-) I-Node verweist, wird dessen Referenzzähler nun dekrementiert und ggf. der I-Node zurückgeschrieben und freigegeben.

creat — neue Datei erzeugen

```
int creat (const char *file_name, mode_t mode);
```

creat ist eine spezielle Variante von *open* zum Öffnen einer neuen Datei.

mkdir — neues Verzeichnis erzeugen

```
int mkdir(const char *dir_name, mode_t mode)
```

mkdir erzeugt ein neues Verzeichnis. Dieses wird mit Einträgen für `"."` und `".."` initialisiert.

Als Argumente werden übergeben der Pfad und die Zugriffsrechtsmaske, Resultat ist i.d.R. Null.

mknod — spezielle Datei erzeugen

```
int mknod (const char *file_name, mode_t mode, dev_t dev);
```

mknod erzeugt eine neue Datei. Es wird i.d.R. zum Erzeugen von FIFO-Dateien oder *device*-Dateien verwendet. Im letzteren Fall wird als dritter Parameter eine Spezifikation des zugeordneten Geräts übergeben.

chmod — Zugriffsrechte setzten

```
int chmod(const char *path, mode_t mode);
```

Für die durch *path* bezeichnete Datei werden die 12 Zugriffsrechts-Bits gemäss *mode* gesetzt.

Wir verweisen auf Kapitel **??**.

chown — Eigentümer oder Gruppe ändern

```
int chown(const char *path, uid_t owner, gid_t group);
```

Der Aufruf setzt für die durch *path* bezeichnete Datei den Eigentümer und die Gruppe neu fest. Will man nur eine der beiden Identifikationsnummern ändern, übergibt man für die andere -1.

Nur der Superuser kann den Eigentümer ändern. Der Eigentümer kann die Datei-Gruppenzugehörigkeit auf jede Gruppe ändern, der er selbst angehört.

chdir, chroot — Verzeichnis wechseln

```
int chdir(const char *path);
int chroot(const char *path);
```

Der als Argument übergebene Pfad wird durch *namei* in einen I-Node umgesetzt. Der in der *u-area* verzeichnete Verweis auf das aktuelle Verzeichnis (*chdir*) bzw. auf das Wurzelverzeichnis (*chroot*) wird abgeändert.

Das aktuelle und das Wurzelverzeichnis werden beim *fork* an Subprozesse vererbt.

Eine Änderung des Wurzelverzeichnisses kommt insbesondere aus Sicherheitsgründen in Betracht, falls man einem Prozess und dessen Sub-Prozessen nicht die gesamte Datei-Hierarchie zugänglich machen will (z.B. Einschränkungen für anonyme Zugriffe über Netz).

stat, fstat — Auskunft über Dateizustand

```
int stat (const char *file_name, struct stat_buf *buf);
int fstat (int filedes, struct stat_buf *buf);
```

stat und *fstat* liefern Daten aus dem I-Node der durch den ersten Parameter spezifizierten Datei in einem Puffer *buf*.

fstat wird für bereits geöffnete Dateien verwendet, als erstes Argument wird der Deskriptor übergeben.

Bei *stat* übergibt man stattdessen den Dateipfad, der zugehörige I-Node muss mittels *namei* zunächst noch reserviert werden.

pipe — Öffnen einer unbenannten Pipe

```
int pipe(int filedes[2]);
```

Der Aufruf erzeugt eine unbenannte Pipe zur Kommunikation zwischen zwei (oder mehreren) verwandten Prozessen.

Als Argument wird ein Verweis *filedes* auf ein Feld mit zwei Integer-Komponenten übergeben. Nach dem *pipe*-Aufruf enthält $filedes[0]$ einen Lese-Deskriptor und $filedes[1]$ einen Schreib-Deskriptor für die Pipe.

Kommuniziert wird mit *read* und *write*, die Deskriptoren werden mit *close* separat wieder geschlossen.

Eine Pipe kann als Kommunikationskanal angesehen werden, in den ein Prozess auf einer Seite Daten hineinschreibt, während auf der anderen Seite ein zweiter Prozess die Daten in der gleichen Reihenfolge wieder herausliest.

Beispiel:

```
#include <unistd.h>
#include <stdio.h>

#define MAX 100

main(){
  int fd[2];
  char nachricht [MAX];

  pipe(fd);

  switch(fork()){
  case -1:
    perror("Fehler bei fork"); exit(1);
  case 0:
    /* Sohn-Prozess, liest aus der Pipe */
    close(fd[1]);           /* nicht benoetigt */
    read(fd[0], nachricht, MAX);
    printf("Nachricht vom Elternprozess empfangen: %s\n", nachricht);
    exit(0);
  default:
    /* Elternprozess, schreibt Nachricht an Kindprozess in die Pipe */
    close(fd[0]);           /* nicht benoetigt */
    write(fd[1], "Hallo Sohn !", 13);
    exit(0);
  }
}
```

Ein schreibender Prozess wird automatisch schlafen gelegt, wenn die Pipe voll ist. Umgekehrt schläft ein lesender Prozess so lange, bis in der Pipe soviele Daten vorhanden sind, wie mit der *read*-Operation angefordert wurden.

Der schreibende Prozess erhält bei einem *exit* des Kommunikationspartners ein spezielles Signal (SIGPIPE). Ein auf weitere Daten wartender Pipe-Leser wird bei *exit* des schreibenden Prozesses geweckt, der *read*-Aufruf terminiert und liefert die Anzahl der

gelesenen Bytes zurück. Dies entspricht der Dateiende-Erkennung beim Lesen einer Plattendatei unbekannter Größe. Die Terminierung des Pipe-Schreibers wird vom Betriebssystem daran erkannt, dass kein Schreibdeskriptor der Pipe mehr geöffnet ist. Wenn ein weiterer Prozess einen offenen Schreibdeskriptor der Pipe besitzt, wird der Leser nicht geweckt. Dies gilt umgekehrt auch für die Lesedeskriptoren. Deshalb ist es unbedingt erforderlich, die nicht benötigten Deskriptoren zu schließen.

Auf der Shell-Ebene steht der Pipeline-Operator `"|"` zur Verfügung, der unbenannte Pipes und Ein-/Ausgabeumlenkung (vgl. *dup*) miteinander kombiniert.

UNIX kennt auch *benannte* Pipes (FIFO-Dateien), die über einen Verzeichnis-Eintrag von beliebigen Prozessen aus angesprochen werden können. Diese werden mit *mknod* erzeugt und mit *open* geöffnet. Ansonsten wird der gleiche Mechanismus wie bei unbenannten Pipes verwendet.

Ganz wichtig ist es zwischen „klassischen“ Pipes und sogenannten „Stream-Pipes“ zu unterscheiden, falls Daten *in beide Richtungen* ausgetauscht werden: Stream-Pipes (System V.4) sind Zweiweg-Kommunikationskanäle, klassische Pipes (Linux, BSD-UNIX) lassen nur den Datentransfer in eine Richtung zu. Man benötigt also in diesem Fall zwei BSD-Pipes. BSD-Pipes sind im Prinzip Ringpuffer beschränkter Größe, während Stream-Pipes durch einen komplexen Nachrichtenübertragungsmechanismus („Streams“-Treibermodell) implementiert sind.

dup — Dateideskriptor duplizieren

```
int dup(int fildes)
```

Der Aufruf erzeugt einen neuen Deskriptor in der Deskriptor-Tabelle des aufrufenden Prozesses, der auf den selben Dateitabellen-Eintrag verweist wie der als Argument übergebene Deskriptor.

Die Abbildung **??** (S. **??**) zeigt einen Prozess vor und nach einem Aufruf von *dup(2)*. Als Resultat liefert der Aufruf 6. Nach dem Aufruf haben Lese- und Schreiboperationen für beide Deskriptoren dieselbe Wirkung.

Man verwendet *dup* zur Umlenkung der Ein- und Ausgabe. Dabei macht man sich zunutze, dass *dup* bei der Bestimmung des neuen Deskriptors den ersten freien Deskriptor verwendet. Im Beispiel ist 6 der erste freie Deskriptor.

Falls man unmittelbar vor dem *dup*-Aufruf die Standard-Eingabedatei schliesst, wird deren Deskriptor, die 0, frei. Wenn man nun den Deskriptor *d* einer beliebigen Datei *f* dupliziert, wird als Duplikat der Deskriptor 0 verwendet. Anschliessend beziehen alle *read*-Aufrufe für Deskriptor 0 (Standardeingabe) ihre Daten aus der Datei *f*.

Dies ist für die Standard-Ausgabedatei (1), die Standard-Fehlerausgabe (2) und alle anderen Dateien genauso machbar.

Beim Aufruf eines Programms mit *exec...* bleiben die Deskriptoren erhalten (sofern man nicht mit *fcntl* das *close-on-exec flag* setzt). Dadurch kann man vor einem Programmaufruf dessen Ein- / Ausgabe umlenken. Dies wird beispielsweise von der Shell genutzt um Ein-/Ausgabeumlenkung oder Pipelines zu realisieren.

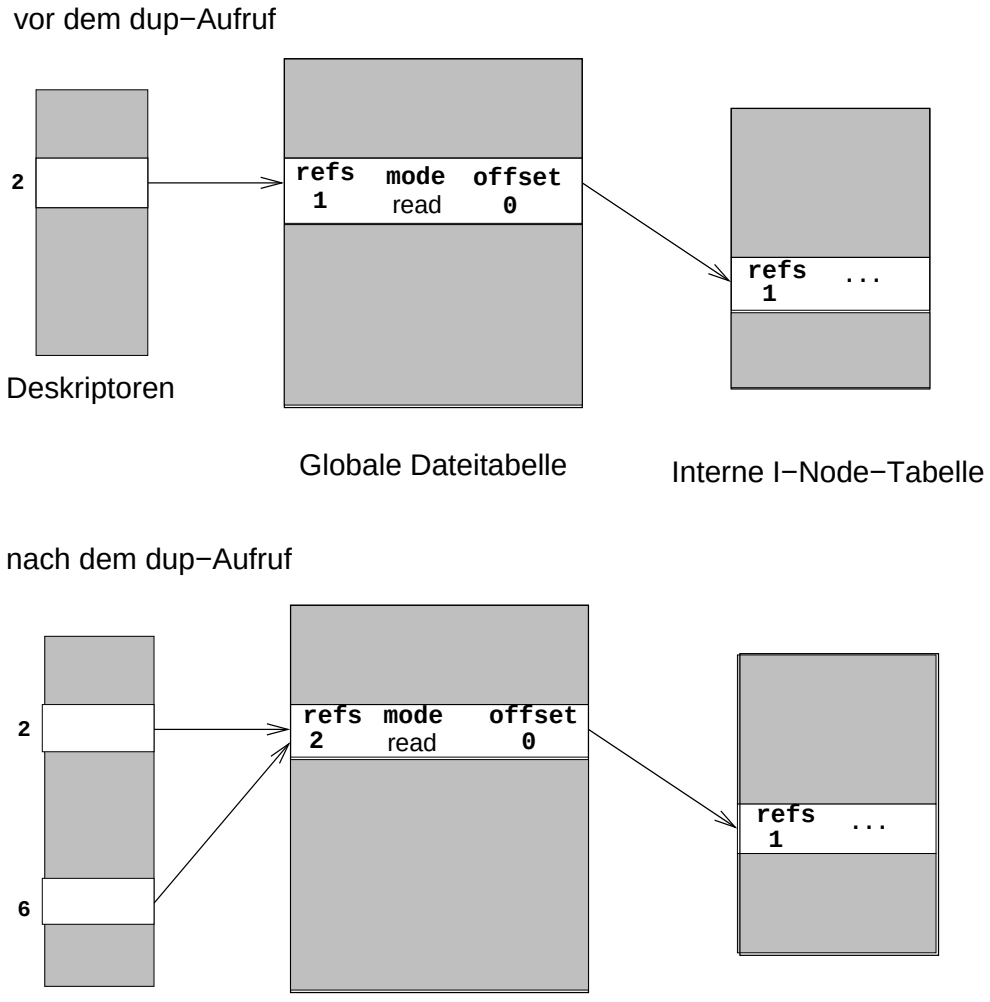


Abbildung 11.5: Der dup-Aufruf

Der Aufruf *dup2*(*alt*, *neu*) dupliziert genau wie *dup*(*alt*) den als erstes Argument angegebenen Deskriptor. Das Duplikat ist allerdings nicht der erste freie Deskriptor, sondern der im zweiten Argument angegeben Deskriptor *neu*. Verweist *neu* schon auf eine Datei, wird diese zuerst geschlossen.

fcntl — spezielle Datei-Operationen

```
int fcntl(int fildes, int command, ...)
```

fcntl ist eine Schnittstelle zu speziellen Datei-Operationen:

- Duplizieren von Dateideskriptoren (vgl. auch *dup*)
- Lesen und Setzen des *close-on-exec-flag*, das bestimmt, ob die Datei bei einem *exec...*-Aufruf geschlossen wird oder nicht
- Lesen und Setzen von *Locks* für die Datei oder auch für beliebige Teile der Datei. Dabei sind sowohl blockierende als auch nicht-blockierende Aufrufe möglich, das BS stellt also hier eine Warteschlangenverwaltung zur Verfügung.

Das erste Argument von *fcntl* ist der Deskriptor der Datei, auf die die spezielle Operation angewandt werden soll, das zweite Arument *command* gibt an, welche Operation ausgeführt wird, das dritte Argument ist in Typ und Wert abhängig von dieser Operation.

mount, umount — Dateisystem montieren und demontieren

Mit dem *mount*-Aufruf kann der *Superuser* ein Dateisystem montieren. Anzugeben ist dabei Typ und Pfad des zu montierenden Systems (Device-Datei oder NFS-Pfad mit Rechner-Adresse), der Montagepunkt (leeres Verzeichnis im vorhandenen Dateisystem), Montageoptionen wie *read-only* und ggf. eine Dateisystem-spezifische Datenstruktur, die vom Dateisystem-Treiber im BS-Kern interpretiert wird.

Jedes montierte Dateisystem hat einen Eintrag in der (globalen) Montagetabelle. Der Eintrag enthält Verweise auf den *super block* des montierten Systems, auf dessen root-I-Node und auf den I-Node des Montagepunktes.

Im I-Node des Montagepunktes wird eine spezielle Montagepunkt-Markierung gesetzt, so dass *namei* bei Verarbeitung eines Pfads leicht feststellen kann, ob mit einem montierten Dateisystem weitergemacht werden muss. Man beachte, dass *namei* Übergänge vom montierten zum übergeordneten Dateisystem auch überprüfen muss, z.B. in folgender Situation:

`mount /dev/sda1 /usr`	Montage
`cd /usr/src`	Übergang in das montierte Dateisystem
`cd ../../bin`	vom montierten in das übergeordnete Dateisystem

Mit *umount* wird ein Dateisystem wieder demontiert.

link — Link auf Datei einrichten

```
int link(const char *oldpath, const char *newpath);
```

Der *link*-Aufruf erzeugt einen *hard link* zu einer existierenden Datei *oldpath* im selben Dateisystem. Wir verweisen auf Kapitel **??**.

symlink — Symbolischen Link auf Datei einrichten

```
int symlink(const char *oldpath, const char *newpath);
```

Der *symlink*-Aufruf erzeugt einen symbolischen Link auf eine existierende Datei *oldpath*. Wir verweisen auf Kapitel **??**. Der *readlink*-Aufruf dereferenziert einen symbolischen Link.

11.8.5 Weitere Funktionen des Dateisystems

Nachfolgend werden weitere Funktionen ohne detaillierte Beschreibung aufgezählt. Die genaue Semantik kann man im Online-Manual oder in der Info-Beschreibung der C-Bibliothek nachlesen.

`rename`	Datei umbenennen
`rmdir`	Verzeichnis löschen
`remove`	Datei oder Verzeichnis löschen (*unlink*, *rmdir*)
`fchmod`	Zugriffsrechte definieren/ändern (vgl. *chmod*)
`utime`	Zugriffszeit / Modifikationszeit ändern
`opendir, rewinddir, readdir, closedir`	zum Durchsuchen von Verzeichnissen
`fchdir`	Verzeichnis wechseln (vgl. *chdir*)
`getcwd`	liefert aktuelles Verzeichnis
`sync, fsync`	Cache-Modifikationen auf Platte schreiben
`lstat`	liefert Informationen zu einer Link-Datei (*stat* dereferenziert Links)
`access`	Zugriffsrechte prüfen
`umask`	Zugriffrechtsmaske für neu erzeugte Dateien definieren
`truncate, ftruncate`	Datei kürzen

Kapitel 12

Literatur

- A. Tanenbaum: Moderne Betriebssysteme, 2. Auflage Pearson Studium, 2002
  Umfangreiche Einführung in das Gebiet, über 1000 Seiten, ca. 50

- €A. Silberschatz u.a.: Operating Systems Concepts
  Addison-Wesley (6. Auflage 2002)
  Gute und bekannte Einführung in das Gebiet, ca. 54

- €T. Letschert: Nebenläufige und Verteilte Programme
  Logos Verlag, Berlin, 1998
  Synchronisations- und Verteilungsmechanismen für den fortgeschrittenen Programmierer. Mit vielen Beispielen, sehr empfehlenswert !

- McKusick M. u.a.: The Design and Implementation of the 4.4BSD Operating System
  UNIX-Interna im Detail, als Vertiefung, sehr lehrreich
  Addison-Wesley, 1996, ca. 100 DM

- W. R. Stevens: Advanced Programming in the UNIX Environment
  Gute Beschreibung der Systemschnittstelle für Programmierer
  Addison-Wesley, 1994

- H. Custer: Inside Windows NT
  Microsoft Press 1992
  Beschreibung der NT-Architektur, leider nur noch im Antiquariat erhältlich

- D.A. Solomon: Inside Windows NT, 2. Auflage
  Microsoft Press 1998, ca. 89 DM
  im Gegensatz zu Custers o.g. 1. Version des Titels weniger an der Architektur und mehr an der Benutzerschnittstelle orientiert

- M. Jäger: UNIX-Kommandoübersicht (WWW-Homepage)

- M. Jäger: Die BASH - Bourne Again Shell (WWW-Homepage)

- Gilly, D.: UNIX in a Nutshell
  O' Reilly
  UNIX-Kompendium mit Kommando-Beschreibung, billig
- *Info*-Hypertext-Manuale
- Sun *Answerbook* (Hypertext-Manuale)
- Solaris und Linux-Online-Manuale