Skript zur Vorlesung Betriebssysteme

Martin Zeller, Hochschule Ravensburg-Weingarten

June 9, 2009

Contents

Table 1: Entwicklung von Hardware und Betriebssystemen

1945 - 1955	Relais, Röhren	Ein Benutzer, ein Programm	kein Betriebssystem
1955 - 1965	Transistoren	Batch-Betrieb	erste Betriebssysteme (FMS)
1956 - 1980	Integrierte Schaltkreise	Spooling, Multiprogramming, Time Sharing	OS360, MULTICS, UNIX
1980 - heute	LSI, VLSI, Multi-Core Prozessoren	wie 1965 - 1980 plus Netzwerkfähigkeiten, verteilte BS, Virtualisierung	DOS, Windows, Linux, MacOS, OS390

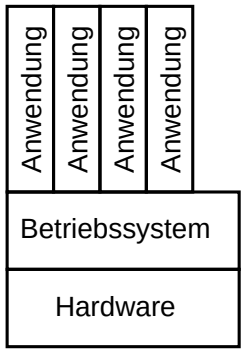
1 Einführung

1.1 Historische Entwicklung

Die Entwicklung der Betriebssysteme bzw. der Systemsoftware verlief parallel zu Entwicklung der Rechnerhardware und damit analog zur Entwicklung der Programmiersprachen.

1.2 Aufgaben eines Betriebssystems

Man kann grob zwei Gruppen von Aufgaben von Betriebssystemen unterscheiden. Aus Sicht der Rechnerhardware besteht die Aufgaben des Betriebssystems in der Verwaltung von Ressourcen. Aus Sicht der Anwendung besteht die Aufgaben des Betriebssystems darin, jedem Anwendungsprogramm eine virtuelle Maschine zur Verfügung zu stellen, die einfacher und einheitlicher zu handhaben ist, als die Rechnerhardware. Anwendungsprogramme müssen dann nicht an die Eigenheiten der jeweiligen Hardware angepasst werden, vielmehr können sie unterschiedliche Hardware über einheitliche Schnittstellen nutzen. Die für den Nutzer vielleicht wichtigste Funktion des Betriebssystems besteht darin, dass es Programme startet und beendet und die gleichzeitig laufenden Programme von einander separiert.



Moderne Prozessoren unterstützten dieses Konzept durch zwei oder mehr Betriebsarten: den User-Mode und den Kernel-Mode. Das Betriebssystem läuft zum Teil im Kernel-Mode. Es hat dadurch uneingeschränkten Zugriff auf alle Speicheradressen. Das bedeutet es kann auf den gesamten Hauptspeicher und auf die Pripheriegeräte zugreifen.

Anwendungsprogramme laufen im User-Mode. Ein Anwendungsprogramm kann daher nur auf seinen eigenen Speicher zugreifen. Es kann insbesondere nicht auf den Teil des Hauptspeichers zugreifen, der von anderen Programmen genutzt wird. Ebeso kann es nicht unmittelbar auf Pripheriegeräte zugreifen. Ein Anwendungsprogramm kann nur über das Betriebssystem Daten mit anderen Anwendungsprogrammen austauschen. Ebenso kann es nur über das Betriebssystem auf Pripheriegeräte zugreifen.

Einige Prozessoren kennen keinen User-Mode, andere unterstützen nicht nur User- und Kernel-Mode sondern auch Zwischenstufen mit unterschiedlichen Rechten. Im Weiteren wird nur

die Unterscheidung User- und Kernel-Mode verwendet. Es gibt noch andere Bezeichnungen für diese Betriebsarten, u. a. spricht man auch von User- bzw. Kernel-Land oder User- bzw. Kernel-Modus oder User- bzw. Kernel-Level.

Virtualisierungsysteme können eine weitere Schicht zwischen Hardware und Anwendungen legen. Ein Virtualisierungssystem simuliert eine Rechnerhardware z. B. einen Standard-PC oder auch einen Großrechner. Auf einem solchen simulierten Rechner lässt sich dann ein Betriebssystem als sog. Gast-Betriebssystem installieren.

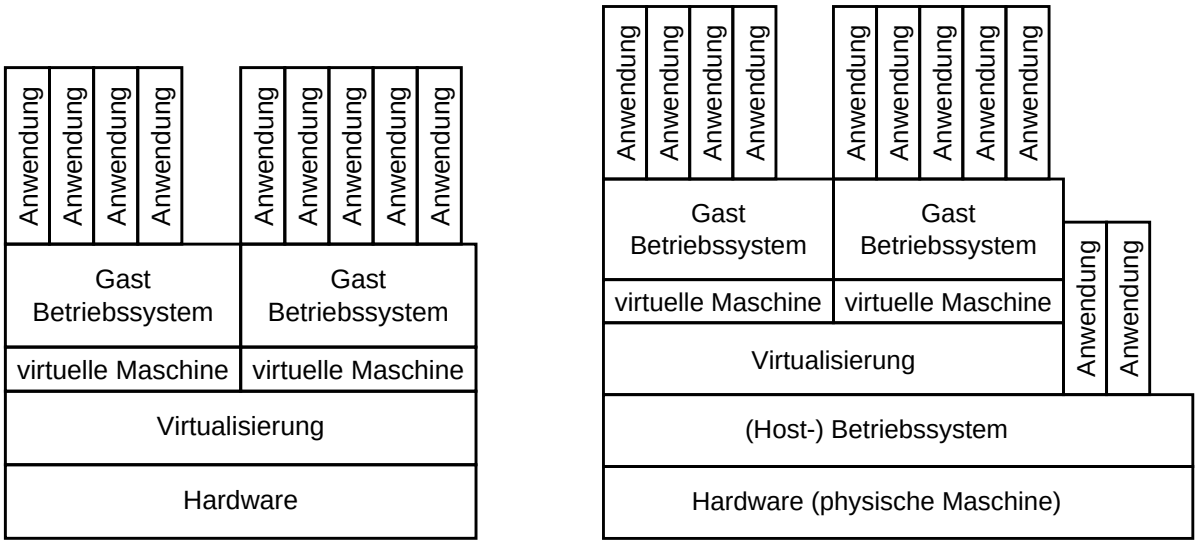


Figure 1: Modelle der Virtualisierung

Auf diese Weise können mehrere Instanzen von Betriebssystemen gleichzeitig auf einem Rechner betrieben werden. Ein Betriebssystem-Instanz kann dann heruntergefahren werden (oder abstürzen), ohne dass die anderen Instanzen beeinträchtigt werden. Eine weitere Funktion, die sich aus der Virtualisierung ergibt, ist die Möglichkeit, Prozesse zur Laufzeit von einer virtuellen Maschine in eine andere zu übertragen. Der Prozess kann dabei auch auf eine andere physische Maschine übertragen werden. Virtualisierungssysteme werden im weiteren nicht betrachtet.

1.3 Prozesse

Ein Prozess ist aus Sicht des Anwenders ein laufendes Progamm. Für das Betriebssystem ist ein Prozess eine Verwaltungseinheit. Das Betriebssystem startet und beendet Prozesse, es teilt Prozessen Ressourcen zu und nimmt sie wieder entgegen. Für diese und einige weitere Aufgaben speichert das Betriebssystem zu jedem Prozess einige Informationen ab. Die Datenstruktur, die diese Daten enthält wird i. Allg. als Process Control Block (PCB) bezeichnet. In Tabelle 2 sind die wichtigsten Parameter dargestellt.

Die Parameter lassen sich in drei Gruppen einteilen: Ausführungszustand des Programms, belegter Hauptspeicher und externer Speicher bzw. Peripheriegeräte (Devices).

Während der Prozess läuft, stehen der `PC`, der `SP` das `PSW` und ggf. einige weitere Attribute des Prozesses in Registern in der CPU.

Table 2: Prozessmanagement

PID (process identification)	Kennung des Prozesses
PC (program counter)	Befehlszähler
SP (stack pointer)	Zeiger auf die Spitze des Stacks
PSW (program status word)	Ergebnis der letzten Vergleichsoperation, Kennung User-Level/Kernel-Level, Priorität, Steuerung für Interrupts, Overflow-Anzeige, weitere Steuerungsinformationen
:	
Zeiger auf Code-Segment	Programm, globale und static Variable
Zeiger auf Daten-Segment	Heap
Zeiger auf Stack-Segment	Stack
:	
Wurzelverzeichnis	Root, in Unix: "'/"'
Arbeitsverzeichnis	
Dateideskriptor(en)	Kennung der von diesem Prozess geöffneten Dateien
Benutzer-ID	
Gruppen-ID	

Die meisten Betriebssysteme organisieren die Speicherbelegung eines Prozess wie in Abb. 2 dargestellt. Der Stack wächst von großen zu keinen Adressen, der Heap wächst von kleinen zu großen Adressen.

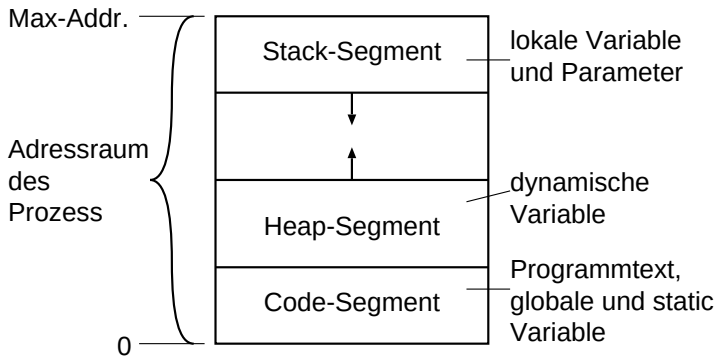


Figure 2: Speicherbelegung eines Prozess

1.4 Grundlegende Mechanismen

1.4.1 Funktionsaufrufe

Zunächst sei hier der Funktionsaufruf innnerhalb eines Programms beschrieben. Ein System-Call in das Betriebssystem wird in Abschnitt 1.4.2 beschrieben. Die Parameterübergabe an eine Funktion ist abhängig vom Compiler. Insbesondere können in manchen Fällen statt dem Stack – wie hier beschrieben – auch Register eingesetzt werden. Die meisten Compiler arbeiten aber mehr oder weniger wie hier beschrieben. Normalerweise wächst der Stack von

hohen zu niedrigeren Adressen; dies ist jedoch für das Verständnis der Mechanismen unerheblich.

Der allgemeine Ablauf ist wie folgt:

1. Die aufrufende Funktion legt die Rücksprungadresse und die Parameterwerte auf den Stack.
2. Die aufrufende Funktion springt zur aufgerufenen Funktion.
3. Die aufgerufene Funktion legt die eigenen lokalen Variablen auf dem Stack an.
4. Die aufgerufene Funktion arbeitet die Anweisungen ab.
5. Die aufgerufene Funktion legt den Rückgabewert in ein Register oder an eine bestimmte Stelle auf dem Stack, wo ihn die aufrufende Funktion findet.
6. Die aufgerufene Funktion springt zurück zu der Rücksprungadresse, die die aufrufende Funktion auf dem Stack hinterlegt hat.

Als Beispiel dient folgendes Programm:

```
1  int bar(float num, char line[]){
2      int sum = 5;
3      if( num < sum ){
4          line[2] = 'a';
5      }
6      return 5;
7  }
8
9  int foo(void){
10     char word[] = "XYZ";
11     int test = 1;
12     test = foo(0.5, word);
13     return 0;
14 }
```

Abbildung 3 zeigt den Aufbau des Call-Stacks um die Parameter aus der Funktion `foo()` in die Funktion `bar()` zu übergeben. Nachdem die Anweisung `int test = 1;` ausgeführt wurde, besitzt der Stack den Aufbau wie bei Stack (1) skizziert.

Um den Aufruf der Funktion `bar()` vorzubereiten, werden die zu übergeben Werte und die Rücksprungadresse auf dem Stack abgelegt. Bei C und C++ wird zunächst der letzte Parameter auf den Stack gelegt, dann der vorletzte bis zuletzt der erste Parameter auf den Stack gelegt wird. Andere Programmiersprachen verwenden oft die umgekehrte Reihenfolge. Unmittelbar vor dem Aufruf der Funktion `bar()`, besitzt der Stack den Aufbau wie bei Stack (2) skizziert.

Der Code für die Funktion `bar()` wird angesprungen, wenn alle Parameter auf dem Stack liegen. Die Funktion `bar()` findet die Parameter korrekt auf dem Stack, wenn der Compiler, der den Quellcode von `bar()` übersetzt hat, die gleichen Annahmen zur Größe der Datentypen und zur Reihenfolge des Stackaufbaus verwendet, wie der Compiler, der den Quellcode von `bar()` übersetzt. Die Funktion `bar()` legt dann evtl. noch die lokale Variable `sum` auf dem Stack an, s. Stack (3).[1] Die Anweisung `return 5;` schreibt den Wert 5 in ein bestimmtes

[1] Es kann auch sein, dass die Variable `sum` nur in einem Prozessor-Register angelegt wird.

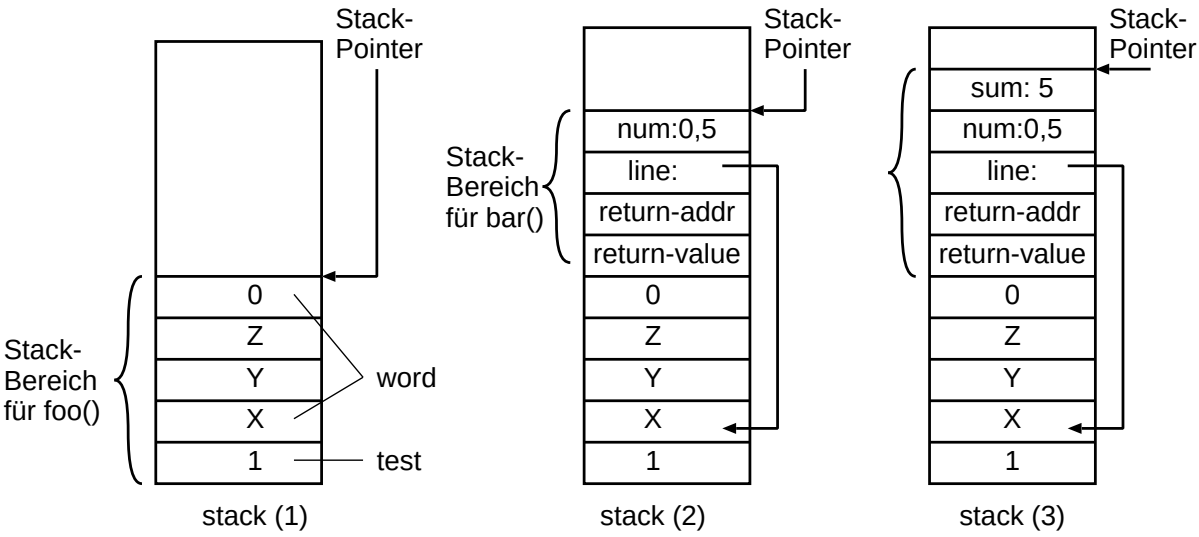


Figure 3: Aufruf der Funktion `bar()`

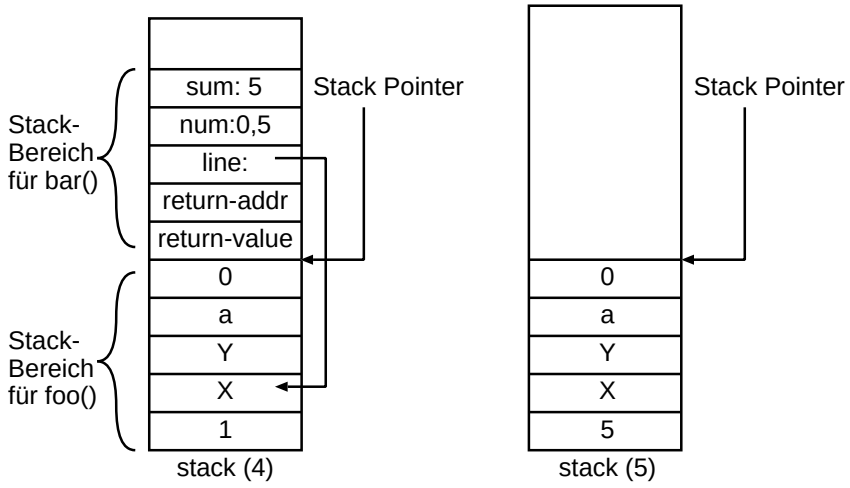


Figure 4: Rücksprung aus der Funktion `bar()`

Register[2] des Prozessors. Anschließend wird der Stackpointer wieder auf das Ende des Stackbereichs von `foo()` gesetzt. Unmittelbar vor dem Ende der Funktion `bar()`, besitzt der Stack den Aufbau wie bei Stack (4) skizziert.

Das Programm springt nun zurück in die Funktion `foo()`. Die letzte Anweisung der der Funktion `bar()` holt die Rücksprungadresse vom Stack und setzt den PC (Programm Counter) auf diesen Wert. Die Funktion `foo()` kopiert den Rückgabewert aus dem Register in die Variable `test`. Damit ist der Aufruf der Funktion `bar()` beendet.

N. B. die Reihenfolge der Werte auf dem Stack spielt keine Rolle, solange nur die aufrufende Funktion und die aufgerufene Funktion die gleichen Regeln verwenden.

1.4.2 Supervisory-Call

Funktionsaufruf von einem Programm im User Space zum Betriebssystem. Ein Anwendungsprgramm springt i. Allg. nicht direkt zum Betriebssystem. Vielmehr ruft es eine Bibliotheksfunktion auf, die dann ihrerseits zu eine Betriebssystem-Funktion verzweigt.

Die Bibliotheksfunktion speichert dazu die Nummer (Kennzahl bzw. Index) der gewünschten Betriebssystem-Funktion an einem fest vereinbarten Ort, z. B. in einem bestimmten Register. Ebenso speichert die Bibliotheksfunktion die Aufrufparamter und die Rücksprungadresse jeweils an einem fest vereinbarten Ort. Anschließend führt die Bibliotheksfunktion die `SVC`-Anweisung aus. Dies bewirkt, dass der Prozessor in den Kernel-Mode wechselt und zu einer festen Adresse im Kernel springt. Der dort vorhanden Code verzweigt anhand der hinterlegten Nummer der gewünschten Betriebssystem-Funktion zu eben dieser Betriebssystem-Funktion. Der Kernel hält sich dazu i. Allg. eine Tabelle mit Funktions-Zeigern auf Betriebssystem-Funktionen. Die Betriebssystem-Funktion holt sich bzw. verwendet die Aufrufparamter.

Die SVC-Anweisung sichert den Status d. h. die wichtigsten Daten zum Zustand des aktuellen Prozesses (PC, SP, PSW usw.) an eine fest vereinbarte Stelle. Die erste Aktivität der so aufgerufenen Betriebssystem-Funktion besteht darin, den restlichen Status des aufrufenden Programmes und die Rücksprung-Adresse an einem dauerhaft sicheren Ort zu hinterlegen (z. B. in einer Liste, die vom Betriebssystem verwaltet wird). Das aufrufende Programm kann dann nach Ende der Betriebssystem-Funktion fortgesetzt werden. Zuletzt stellt die Funktion den vorherigen Zustand des Prozessors wieder her, schaltet den Prozessor wieder in den User-Mode zurück und springt zur Rücksprungadresse, so dass das aufrufende Programm weiterläuft.

Ein Supervisory-Call funktioniert also ähnlich wie ein normaler Funktionsaufruf mit folgenden Unterschieden: Die Adresse der Funktion wird nicht direkt angesprungen, sondern über eine Tabelle, die der Betriebssystem-Kern verwaltet. Die Parameter und die Rücksprungadresse werden nicht über den Stack sondern über Register und/oder einen anderen Mechanismus übergeben. Beim Aufruf schaltet der Prozessor in den Kernel-Mode, vor dem Rücksprung schaltet er zurück in der User-Mode.

[2] Der Rückgabewert kann auch über den Stack zurückgegeben werden. Einige Compiler wie z. B. gcc verwenden aber je nach Datentyp bestimmte Register.

1.4.3 Interrupt-Behandlung

Um auf externe Ereignisse zu reagieren, muss der Prozessor das gerade laufende Programm unterbrechen können um zu einer Betriebssystem-Funktion verzweigen. Ein solches externes Ereigniss ist z. B. : Ein I/O-Controller meldet, dass er Daten an einen Prozess liefern möchte (von ein Festplatte o. ä.).

Jedem I/O-Gerät ist ein sogenannter Interrupt-Vektor zugeordnet. Der Interrupt-Vektor enthält die Adresse einer Betriebssystem-Funktion, die das Gerät steuert. Wenn ein I/O-Controller einen Interrupt erzeugt, geht der Prozessor in den Kernel-Mode und sichert den Status des aufrufenden Programmes (PC, SP, PSW usw.) des gerade laufenden Prozess in einen Speicherbereich des Betriebssystem (i. Allg. die Prozesstabelle). Anschließend verzweigt er zu der im Interrupt-Vektor enthaltenen Adresse. Dieser Mechanismus ist i. Allg. in der Hardware des Prozessors implementiert.

Die dann aufgerufene Betriebssystem-Funktion sichert zunächst den restlichen Status des unterbrochenen Prozess. Dies ist i. Allg. in Assembler implementiert. Anschließend ruft die Betriebssystem-Funktion meist eine in C implementierte Funktion auf, die die restliche Arbeit erledigt. Zuletzt wird – in Assembler und zuletzt per Hardware – der Zustand des unterbrochenen Prozess im Prozessor wieder hergestellt, so dass der Prozess weiterläuft.

Sowohl bei Supervisory-Calls als auch bei Interrupt-Behandlung kann es vorteilhaft sein, den aufrufenden bzw. den unterbrochenen Prozess nicht sofort weiterzuführen, sondern einen anderen Prozess zu starten bzw. fortzusetzen. Dies ist immer dann sinnvoll, wenn der aufrufende bzw. der unterbrochene Prozess auf eine Ein- oder Ausgabe wartet oder aus einem anderen Grund im Warte-Zustand (blockiert) ist (s. Abschnitt 2.4).

2 Multiprozess-Systeme

2.1 Prozesse

Die zentrale Aufgabe eines Multiprozess-Betriebssystems ist es, Prozesse, die gleichzeitig im System vorhanden sind, voneinander zu trennen. Jeder Prozess soll die Illusion haben, der einzige Prozess zu sein, der auf dem Rechner läuft. Ein Prozess soll nur über bestimmte Systemaufrufe mit einem anderen Prozess in Kontakt kommen können. Ein Prozess soll also CPU-Zeit, Hauptspeicher, Zugriff auf Dateien und andere Ressourcen bekommen, ohne sichtbaren Einfluss anderer Prozesse. Dies ist immer nur nährungsweise zu erreichen. Außerdem kann es für bestimmte Anwendungen sinnvoll sein, die Trennung zwischen Prozessen mehr oder weniger aufzuheben; man spricht dann von Threads (näheres s. u.).

2.1.1 Erzeugung von Prozessen

Ein Prozess wird beim Start des Betriebssystem erzeugt. Dieser ereugt denn einen oder mehrere weitere Prozesse, welche ggf. ihrerseits weitere Prozesse erzeugen. In jedem Betriebssystem gibt es eine Funktion (evtl. in mehreren Varianten) um einen Prozesse zu starten.

Unix: In Unix heißt die Funktion `fork()`, meist in Verbindung mit der Funktion `execvp()`. Die Funktion `fork()` erzeugt eine Kopie des aufrufenden Prozess. Mit der Funktion `execvp()` bzw. mit verwandten Funktionen kann der neu erzeugt Prozess ein anderes Programm laden und damit zur Ausführung bringen. Am Rückgabewert der Funktion `fork()` kann ein Prozess erkennen, ob er der aufrufende Prozess ist oder der neu erzeugte.

Linux: In Linux gibt es zusätzlich die System-Funktion `clone()`, die einen neuen Prozess oder einen neuen Thread erzeugt. Über Argumente der Funktion kann gesteuert werden, welche Ressourcen der Kind-Prozess bzw. der Kind-Thread mit dem Eltern Prozess gemeinsam nutzt. Es lassen sich auf diese Weise verschiedene Zwischenstufen zwischen Prozess- und Thread-Paar erzeugen. Üblicher Weise wird die Funktion `clone()` nicht unmittelbar benutzt sondern die Funktion `pthread_create()` einer Bibliothek (z. B. Next Generation POSIX Threads (NGPT), Linux Threads-Library, Pth – GNU Portable Threads).

Windows: In Windows heißt der Befehl `CreateProcess()`. Er spaltet einen neuen Prozess ab und läd das auszuführende Programm in diesen Prozess.

Vordergrund-/Hintergrund-Prozesse Man kann Prozesse unterscheiden in Hintergrund-Prozesse (in Unix 'daemon', in Windows 'service' genannt) und Vordergrund-Prozesse. Die Hintergrund-Prozesse nehmen keine Eingabe vom Benutzer entgegen. Sie arbeiten entweder aus eigenen Antrieb (z. B. zeitgesteuert) oder bearbeiten Eingaben von anderen Prozessen (z. B. ein Drucker-Verwalter).

Ein Vordergrund-Prozess nimmt Eingaben vom Benutzer entgegen. Als Eingabemedium dient i. Allg. die Tastatur und/oder die Maus bzw. verwandte Eingabegeräte.

Unter Unix gibt es keine besonderen Anforderungen an Hintergrund-Prozesse. Jeder Prozess kann als Hintergrund-Prozess gestartet werden. Dies ist allerdings nur sinnvoll, wenn der Prozess keine Eingabe vom Benutzer erwartet. Unter Windows muss ein Hintergrund-Prozess eine (Server-)Schnittstelle realisieren, über die das Betriebssystem den Prozess steuern kann.

2.1.2 Beenden von Prozessen

Wird ein Prozess beendet, so gibt das Betriebssystem alle Ressourcen, die der Prozess belegt hat, wieder frei.

2.2 Threads

Ein Thread ist ein Ausführungsfolge von Programmanweisungen. In einem Prozess können mehrere Threads gleichzeitig existieren. Folgenden Daten besitzt jeder Thread individuell: Programmzähler, ein Satz von Registerbelegungen, Zustand und Stack; d. h. jeder Thread verwaltet lokale Variable und Parameter selbst. Alle Threads eines Prozess verwenden die weiteren Ressourcen des Prozesses gemeinsam: Heap-Variable (dynamische Variable), globale und statische Variable, geöffnete Dateien und ggf. weitere Betriebsmittel. Man unterscheidet zwischen User Level Threads und Kernel Threads.

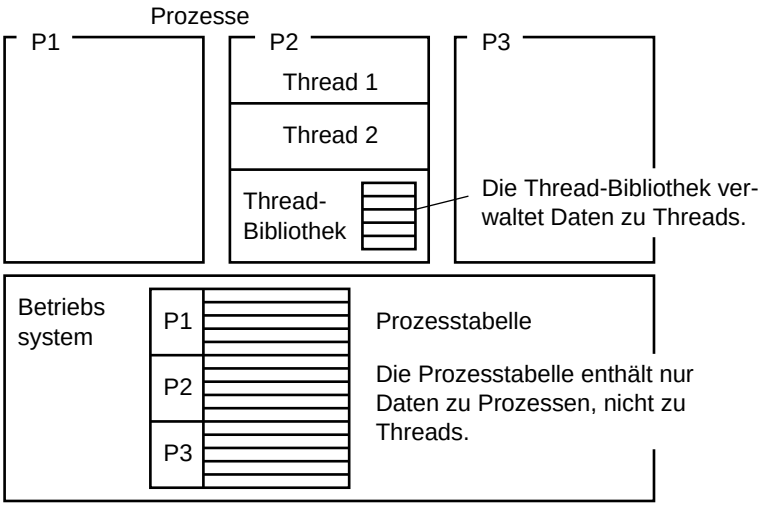


Figure 5: Eine Thread-Bibliothek verwaltet User Level Threads

User Level Threads werden von einer Bibliothek im Anwendungsprogramm verwaltet. User Level Threads sind für das Betriebssystem transparent. Der Wechsel von einem Thread zum anderen ist i. Allg. schneller als bei Kernel Threads. Wenn ein Thread blockiert, so blockiert der Prozess. User Level Threads können auch auf Mehrprozessor-Rechnern immer nur auf einem Prozessor ablaufen.

Kernel Level Threads werden vom Betriebssystem verwaltet. Der Wechsel von einem Thread zum anderen ist i. Allg. langsamer als bei User Level Threads. Wenn ein Thread blockiert, so kann das Betriebssystem einen anderen Thread desselben Prozesses aktivieren. Auf einem Mehrprozessor-Rechnern kann das Betriebssystem die Threads eines Prozesses auf mehrere Prozessoren verteilen.

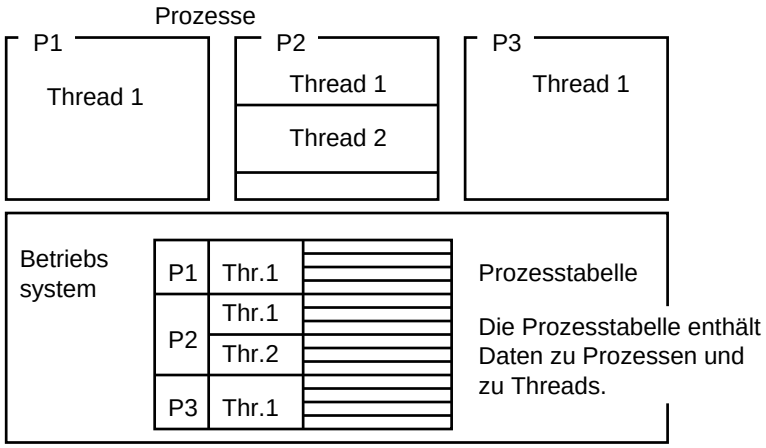


Figure 6: Das Betriebssystem verwaltet Kernel Level Threads

2.3 Prozesse und Threads unter Unix/Linux

2.3.1 Starten und beenden

Systemfunktionen

- `fork()`
- `clone()` bzw. Funktionen der PThread-Bibliothek oder andere Bibliotheken, die auf der Funktion `clone()` aufsetzen.
- `exit()`
- `exec()` bzw. verschiedene Varianten z. B. `execvp()`

2.3.2 Hintergrundprozesse

Bezeichnung Daemon (Disk And Execution MONitor). Ein Unix-System befindet sich stets in einem von mehreren "'Runlevel"'. Jedem Runlevel sind bestimmte Daemons und bestimmte Systemeigenschaften zugeordnet (z. B. Nutzer können sich einloggen, Netzwerk verfügbar etc.). Beim Betreten eines Runlevels werden jeweils bestimmte Daemons gestartet und ggf. andere beendet. Auch das Fenstersystem X-Windows ist – wenn überhaupt – nur in einigen Runleveln aktiv.

- Rl 0: angehalten
- Rl 1: Single User Modus
- Rl 2 - 5: Multi User Modus
- Rl 6: Neustart

2.4 Scheduling

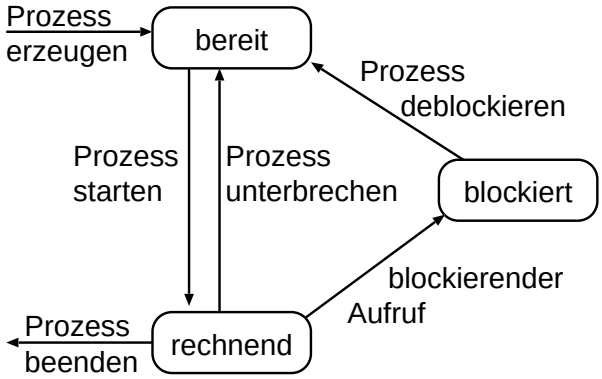


Figure 7: Zustände eines Prozesses in einem Multiprozess-System

2.4.1 Dispatching und Scheduling

Unter Dispatching versteht man den Wechsel von einem Prozess zum einem anderen. Der Ausführungszustand des laufenden Prozess wird gespeichert. Anschließend wird der Ausführungszustand eines wartenden Prozess wiederhergestellt, so dass dann der bisher wartende Prozess läuft.

Scheduling bezeichnet die Auswahl des Prozesses, der als nächstes laufen soll. Der Scheduling-Algorithmus betrachtet dabei natürlich nur Prozesse im Zustand "'bereit"'.

Der Prozesswechsel kann durch verschiedene Ereignisse ausgelöst werden; im wesentlichen sind es folgende Ereignisse:

- ▷ Ein Timer läuft ab; d. h. eine "'Zeitscheibe"' geht zu Ende.
- ▷ Ein Prozess wartet auf ein Peripherie-Gerät, z. B. Eingabe von der Festplatte.
- ▷ Ein Prozess "'legt sich selbst schlafen"' (durch einen Systemaufruf).
- ▷ Ein Peripherie-Gerät erzeugt einen Interrupt.

Wenn Prozesse durch einen Timer unterbrochen werden können, so spricht man von "'preemptive multitasking"'. Müssen Prozesse sich selbst "'schlafen legen"', um einen Prozesswechsel zu ermöglichen, spricht man von "'cooperative multitasking"'.

2.4.2 Anforderungen an Schedulingalgorithmen

Anforderungen für alle Arten von Prozessen

- ▷ fair (jeder Prozess erhält Rechenzeit)
- ▷ gleichmäßige Auslastung der Komponenten
- ▷ Strategie-Vorgaben müssen erfüllt werden (policy enforcement), z. B. Prioritäten

Anforderungen für Batchsysteme

- ▷ Durchsatz/Durchlaufzeit
- ▷ Verweilzeit
- ▷ Auslastung der Ressourcen (CPU, Speicher, I/O): Alle Ressourcen sollen möglichst voll ausgelastet sein.

Anforderungen für Dialogsysteme

- ▷ Antwortzeit

Anforderungen für Echtzeitsysteme

- ▷ Einhaltung von Fristen (meeting deadlines)
- ▷ Vorhersehbares Verhalten
- ▷ Gleichförmiges (Zeit-)Verhalten (wenig Jitter bei Multimedia-Anwendungen)

2.4.3 Schedulingalgorithmen

- ▷ First-Come First-Served
- ▷ Shortest Job First
- ▷ Shortest Remaining Time First
- ▷ Round-Robin
- ▷ Kombination von Round-Robin Prioritätsgesteuert
- ▷ Prioritätsgesteuert
- ▷ Raten-Monotonic-Scheduling
- ▷ Lotterie-Algorithmus

2.4.4 Scheduling von Echtzeitsystemen

Für Echtzeitsysteme wird häufig prioritätsgesteuertes Scheduling verwendet oder Raten-Monotonic-Scheduling.

Prioritätsgesteuert heißt, dass von allen Prozessen, die im Zustand "'bereit"' sind, stets der Prozess mit der höchsten Priorität rechnen darf. Üblicherweise bekommen alle Prozsse unterschiedliche Prioritäten.

Raten-Monotonic-Scheduling ist eine Art prioritätsgesteuertes Scheduling, das eingesetzt werden kann, wenn jeder Prozess regelmäßig in einem festen Intervall ausgeführt werden muss. Die Priorität eines Prozesses wird dann umgekehrt proportional zu seiner Intervall-Länge festgelegt. Wird ein Prozess gestartet, der eine höhere Priorität besitzt, als der gerade rechnende, so wird der bisher rechnende Prozess vom neu gestarteten Prozess verdrängt. Es lässt sich mathematisch zeigen, dass durch dieses Verfahren die alle Prozesse innerhalb ihrer Periode ablaufen dürfen, sofern die CPU-Auslastung nicht über ca. 70% ($ln(2)$) liegt. Dabei ist allerdings die Zeit für Prozesswechsel und Scheduling enthalten.

Lottery-Scheduling: Jeder Prozess bekommt eine Anzahl Lose zugeteilt. Wenn der Scheduler aktiviert wird, zieht er ein Los. Der Prozess, dess Los gezoge wurde, darf rechnen. Im Mittel wird die Rechenzeit gemäß der Anzahl der Lose verteilt. Jeder Prozess, der mindestens ein Los besitzt, darf mit eine gewissen Wahrscheinlichkeit früher oder später rechnen. Das Verfahren ist also fair.

2.4.5 Prioritäts-Umkehr - priority inversion

Bei prioritätsbasierten Verfahren kann eine sogenannte Prioritäts-Umkehr (priority inversion) auftreten: Ein System enthält z. B. drei Prozesse, P_1, P_2 und P_3. Der Prozess P_1 besitzt die höchste Priorität, P_2 besitzt eine mittlere und P_3 die niedrigste Priorität. Es kann nun folgender Fall eintreten: P_3 besitzt eine Ressource, auf die P_1 wartet während gleichzeitig P_2 bereit ist. In diesem Fall darf P_2 rechnen, da P_1 blockiert ist und P_3 eine niedrigere Priorität besitzt. Also kann P_3 nicht rechnen und blockiert weiterhin die Ressource. Dadurch erhält P_2 eine höhere effektive Priorität als P_1. Gegenmaßnahmen sind z. B. priority inheritance; ein Prozess, der eine Ressource hält, bekommt die maximale Priorität aller Prozesse, die auf die Ressource warten. Alternativ: priority ceiling; ein Prozess, der eine Ressource hält, bekommt eine Priorität, die der Ressource zugeordnet ist, Diese Priorität muss dann allerdings höher sein, als die Priorität, die ein wartender Prozess haben kann.

2.4.6 Scheduling in Linux

Linux verwendet eine Kombination aus prioritätsgesteuertem Scheduling und Round-Robin. Für die verschiedenen Prioritätswerte gibt es jeweils eine Prozess-Liste, die im Round-Robin-Verfahren bedient wird. Der Scheduler prüft also, ob in der Liste der Prozesse mit maximaler Priorität ein Prozess bereit ist. Wenn nein, geht er zur Liste der Prozesse mit der zweithöchsten Priorität und so weiter.

Die Priorität eines Prozesses wird regelmäßig neu berechnet. Dabei wird die bisherige CPU-Nutzung, die Basis-Priorität und der "'nice"'-Wert verrechnet. Für die CPU-Nutzung wird i. Allg. der zuletzt gemessene Wert höher gewichtet als der vorherige. Eine schnelle Implementierung ist z. B. $CpuWert(t_n) = (CpuWert(t_{n-1}) + CpuWert(gemessen))/2$. Der "'nice"'-Wert kann vom Benutzer vergeben werden, um die Priorität eines Prozesses zu verringern. Ein Benutzer mit "'root"'-Rechten kann auf diese Weise die Priorität eines Prozesses auch erhöhen.

Ein Prozess, der im Kernel-Mode wartet, erhält für diese Zeit eine bestimmte Priorität, je nachdem, auf welches Ereignis er wartet. Die Priorität ist i. Allg. höher als die Prioritäten der Prozesse im User-Level. Wenn ein Prozess z. B. auf Daten von der Festplatte wartet, so bekommt er eine hohe Priorität. Er hat dann gute Chancen, sofort aktiv zu werden, sobald die Daten eingetroffen sind und er dadurch in den Zustand "'bereit"' versetzt wird.

Aufgabe 2.8.3 (2.1.17) im Skript von Fr. Keller Gegeben: Anzahl von Prozessen.

Prozess Nr.	Startzeit	Dauer	Priorität
1	0	10	1
2	5	5	2
3	8	3	5
4	10	10	3
5	15	7	3

Der Scheduler wird aktiv, wenn ein Prozess neu gestartet wird bzw. wenn sich ein Prozess beendet. Wenn es zu einem Zeitpunkt zwei oder mehr Prozesse im Zustand "'bereit"' gibt, die maximale Priorität besitzen, gelten zwischen diesen Prozessen folgende Vorrangregeln:

1. Wenn ein Prozess bereits rechnend ist, darf er weiterrechnen.
2. Wenn beide/alle Prozesse "'breit"' sind, darf der Prozess, der bislang die meiste Rechenzeit verbraucht hat, rechnen.
3. Ansonsten darf der Prozess mit der kleinsten PID rechnen.

Achtung: Diese Vorrangregeln gelten zunächst nur für dieses Beispiel, real existierende Scheduler können durchaus andere Regeln einsetzen.

Gesucht: Mittlere Verweildauer (a) bei FIFO, (b) bei prioritätsbasiertem Scheduling

Prozess Nr.	Dauer (a)	Dauer (b)
1	10	35
2	10	25
3	10	3
4	18	11
5	20	13
Summe	68	87
MVD	13,6	17,4

Aufgabe (2.1.18) im Skript von Fr. Keller

Prozess Nr.	Dauer	Basis-Priorität	SVC in Zeitscheibe
1	6	5	kein SVC
2	5	3	1, 2
3	4	4	2

Alle Prozesse starten zum Zeitpunkt 0.

Regeln:

- Wenn für einen Prozess eine Zeitscheibe abläuft und dieser keinen SVC abgesetzt hat, wird sein Priorität um den Wert 1 vermindert
- Wenn für einen Prozess eine Zeitscheibe abläuft und er einen SVC abgesetzt hat, wird sein Priorität um den Wert 1 erhöht.
- Wenn ein Prozess eine höhere Priorität besitzt als alle anderen Prozesse, so darf er in der nächsten Zeitscheibe laufen.
- Wenn mehrere Prozesse die maximale Priorität besitzen und einer davon läuft, so darf dieser in der nächsten Zeitscheibe weiterlaufen.

- Wenn mehrere Prozesse die maximale Priorität besitzen und keiner davon läuft, so darf der Prozess mit der kürzesten Wartezeit in der nächsten Zeitscheibe laufen.
- Die minimale Priorität ist 0.

2.5 Speicherverwaltung für Speicherblöcke

Zunächst muss man unterscheiden, ob Speicherblöcke einheitlicher Größe oder Speicherblöcke unterschiedlicher Größe verwaltet werden sollen.

Speicherblöcke einheitlicher Größe müssen z. B. für den Dateisysteme verwaltet werden s. Abschnitt 5. Für die Verwaltung kommen u. a. folgende beiden Techniken in Frage: Speicherbelegungs-Listen bzw. Verwaltung durch Bitmaps.

Die Frage, wie Speicherblöcke unterschiedlicher Größe verwaltet werden können, stellt sich z. B. bei Prozessen, die dynamische Variable verwenden. Der Heap-Speicher enthält dann Variable unterschiedlicher Größe und unterschiedlicher Lebensdauer. Für die Verwaltung werden i. Allg. Speicherbelegungs-Listen verwendet.

Speicherbelegungs-Listen Der Prozess führt je eine Liste mit belegten bzw. mit freien Speicherblöcken.

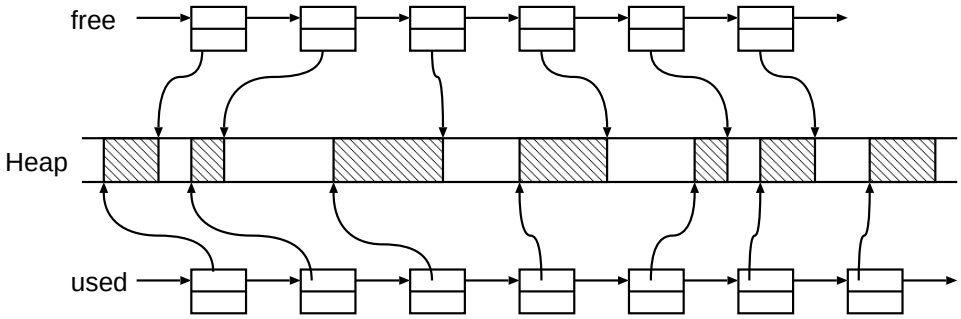


Figure 8: Speicherverwaltung anhand von Listen

Wenn ein Speicherblock vom Programm allokiert wird, sucht das Programm nach einem freien Block ausreichender Größe. Von diesem Block wird dann ein Stück der benötigten Größe allokiert. Dazu muss der allokierte Bereich der Liste der belegten Blöcke zugeschlagen werden. Der verbleibende freie Bereich steht dann in der Liste der freien Blöcke.

Die Listen zu Verwaltung der Speicherblöcke können in einem dafür reservierten Speicherbereich gehalten werden. Oft (z. B. in C/C++ unter Unix) werden die Listenelemente aber in den Anfangsstücken der Speicherblöcke selbst verwaltet (s. Abb. 9) oberes Bild. Wenn ein Programm – aufgrund eines Programmierfehlers oder eines gezielten Angriffs – auf Speicher außerhalb eines zugeteilten Speicherblockes schreibt, kann die Listenstruktur verloren gehen.

Bitmaps Der Prozess verwaltet die freien und belegten Speicherblöcke in einer Bitmap. Jedem Bit ist ein Speicherblock fester Länge zugeordnet: Das erste Bit ist dem ersten Speicherblock zugeordnet, das zweite Bit dem zweiten und so weiter. Das Bit gibt an, ob der

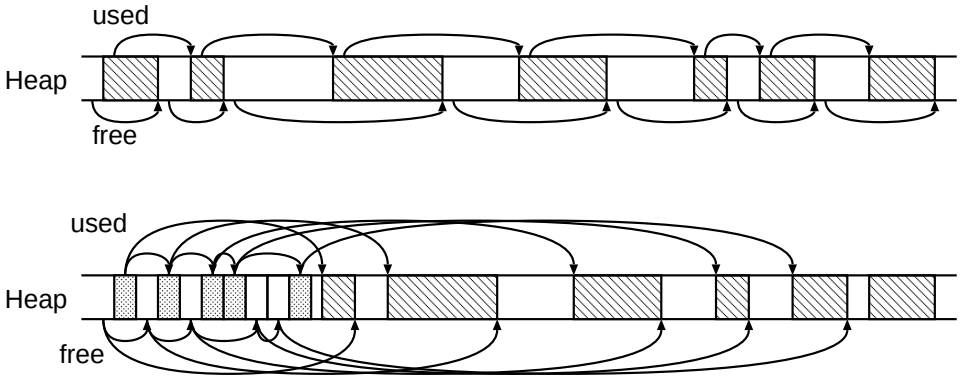


Figure 9: Platzierung der Listenelemente verteilt oder am Anfang des Heaps

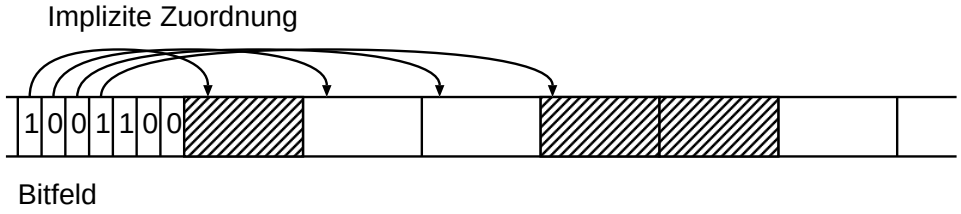


Figure 10: Speicherverwaltung mit Bitmap

zugeordnete Speicherblock frei oder belegt ist. Ein Speicherblock kann also nur als Ganzes belegt oder freigegeben werden. Die Speicherverwaltung durch Bitmaps ist einfach zu realisieren. Ein Nachteil besteht darin, dass es relativ zeitaufwändig ist, eine Bitmap nach einer Folge von Nullen zu durchsuchen, insbesondere da die Folge i. Allg. nicht an einer Wortgrenze ausgerichtet ist. Ein weiterer Nachteil besteht darin, dass die Menge des verwalteten Speichers nicht ohne weiteres erhöht werden kann. Die Länge der Bitmap gibt vor, wie viele Speicherblöcke verwaltet werden. Um die Anzahl der verwalteten Speicherblöcke zu erhöhen, muss also die Bitmap verlängert werden. Diese Art der Speicherverwaltung wird eingesetzt, um Speicherbereiche von Festplatten zu verwalten (Abschnitt 5.6.3).

Strategien Wenn ein Programm neuen Speicher anfordert, muss die Speicherverwaltung ein passendes Stück Speicher finden. Dafür kommen u. a. folgende Strategien in Frage:

- First-Fit
- Next-Fit
- Best-Fit
- Worst-Fit

Die Strategie "'First-Fit"' liefert i. Allg. die besten Ergebnisse (durch Simulationen ermittelt).

2.6 Speicherverwaltung für Multiprozess-Systeme

Ein Programm verwendet normalerweise Adressen eines linearen Adressraums für Programmanweisungen und Variable. Die verschiedenen Programme, die auf einem Prozessor laufen

können, verwenden i. Allg. den selben Adressraum. Wenn ein Prozessor zu einem Zeitpunkt mehr als ein Programm im Speicher hält, kann ein und dieselbe Adresse in mehr als einem Programm verwendet werden. Die Programme bzw. ihre Daten müssen aber in unterschiedlichen Bereichen des physischen Hauptspeichers abgelegt werden. Die Adressen, die ein Programm verwendet, werden als virtuelle Adressen bzw. als logische Adressen bezeichnet. Sie unterscheiden sich also von den physischen Adressen, auf die letztendlich zugegriffen wird. Eine zweite Frage stellt sich dadurch, dass der Adressraum eines Programms i. Allg. größer ist als der physisch vorhandene Hauptspeicher. Auch der tatsächlich verwendete Speicher eines Programms (bzw. der Speicher aller gleichzeitig geladenen Programme) kann größer sein als der physisch vorhandene Hauptspeicher.

Um diese beiden Probleme aufzulösen werden hauptsächlich zwei Techniken eingesetzt: Segmentierung und Paging. Beide Techniken können jeweils beide Probleme lösen. In modernen Systemen wird hauptsächlich Paging verwendet.

Einige Prozessoren laden bei einem Speicherzugriff mehr als ein Byte ein. Ein Prozessor kann z. B. mit einem Zugriff auf den Hauptspeicher eine 4 Byte große Zahl in ein Register laden und gleichzeitig diese Zahl sowie zusätzlich 60 Byte in den (bzw. die) Cache-Speicher übertragen. Der Abstand zwischen zwei benachbarten Adressen im Hauptspeicher beträgt aber dennoch i. Allg. ein Byte.

2.6.1 Segmentierung

Ein Programm belegt einen oder mehrere Speicherbereiche mit linearen (virtuellen/logischen) Adressen. Diese Speicherbereiche werden Segmente genannt. Eine virtuelle Adresse des Programms besteht dann aus der Segmentnummer und dem Offset innerhalb des Segments. Wenn das Programm gestartet wird, lädt das Betriebssystem die Segmente des Programms in den Hauptspeicher. Der neue Prozess enthält eine Segmenttabelle. Zu jedem Segment wird in der Segmenttabelle die Anfangsadresse, die Größe oder die Endadresse, Zugriffsrechte und ggf. einige weitere Informationen vermerkt.

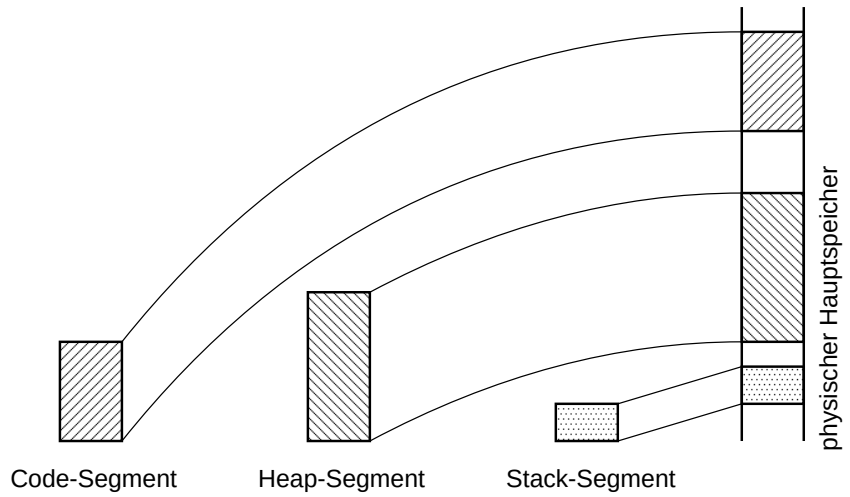


Figure 11: Abbildung von Programm-Segmenten in den physischen Hauptspeicher

Bei jedem Speicherzugriff muss die virtuelle Adresse in die entsprechende physische Adresse umgewandelt werden. Zunächst muss die physische Anfangsadresse des angesprochenen Segments ermittelt werden, dann kann auf die Anfangsadresse der Offset addiert werden. Die virtuelle Adresse bestehend aus Segmentnummer und Offset wird vom Compiler beim Übersetzen vergeben. Die physische Anfangsadresse und die Größe bzw. die Endadresse eines Segments wird vom Betriebssystem vergeben. Das Betriebssystem kann das Segment i. Allg. auch während der Laufzeit des Prozesses verschieben; es muss dann den Speicherinhalt verschieben und die physische Anfangsadresse sowie ggf. die Endadresse entsprechend anpassen.

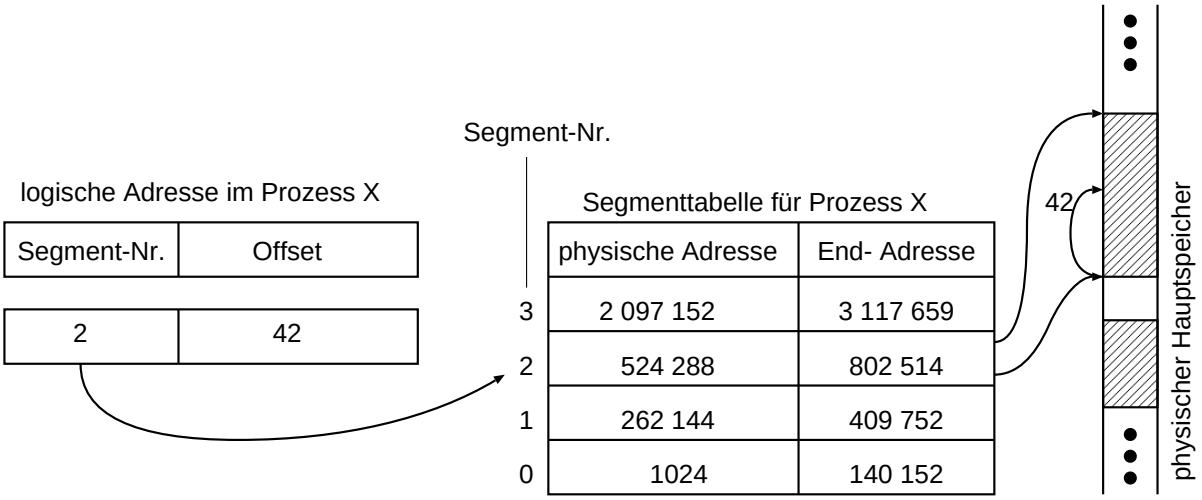


Figure 12: Umrechnung einer virtuelle (Segment-) Adresse in eine physische Hauptspeicher-Adresse

Einige Prozessoren (z. B. der Pentium) unterstützen Segmentierung, indem sie die Verwaltung und den Zugriff auf die Segment-Tabelle selbst realisieren.

Im einfachsten Fall (nicht Pentium) besteht der Speicher jedes Programms nur aus einem Segment. Die CPU enthält zwei spezielle Register: `base register` und `limit register`. Wenn das Segment des Prozesses in den Hauptspeicher geladen wird, so schreibt das Betriebssystem die Anfangsadresse, dieses Segments in das `base register` und die Endadresse in das `limit register`.

Bei jedem Speicherzugriff wird automatisch von der CPU, der Wert des `base register` auf die Adresse addiert. Außerdem überprüft die CPU, ob das Ergebnis kleiner als der Wert im `limit register` ist. Das Programm kann also stets Adressen zwischen 0 und einer bestimmten Obergrenze verwenden, unabhängig davon, wo es vom Betriebssystem im Hauptspeicher positioniert wird.

Ist nicht genügend Hauptspeicher vorhanden, um ein Segment zu laden, so kann das System ein gerade nicht genutztes Segment auf die Festplatte sichern und aus den Hauptspeicher löschen. Das Betriebssystem muss sich dann merken, wo das ausgelagerte Segment liegt.

Segment-Nr.

Segmenttabelle für Prozess X

	physische Adresse	End- Adresse
3	1 886 279	1 949 031
2	1 174 871	1 205 382
1	193 772	234 617
0	177 031	180 152

Segment-Nr.

Segmenttabelle für Prozess Y

	physische Adresse	End- Adresse
3	2 097 152	3 117 659
2	524 288	802 514
1	262 144	409 752
0	1024	140 152

logische Adr. Prozess X

Segment-Nr.	Offset
2	42
0	4 805

physische Adresse

logische Adr. Prozess Y

Segment-Nr.	Offset
3	34 028
0	4 805

physische Adresse

physische Adresse	logische Adresse	Prozess
304 028		
2 105 698		
170 358		

Figure 13: Übungsaufgabe: Berechnen Sie die fehlenden Größen

2.6.2 Paging

Jedes Programm verwendet seinen eigenen virtuellen Adressraum von 0 - X. Der Adressraum wird in Seiten eingeteilt, mit einer festen Seitengröße. Übliche Werte für die Seitengröße sind 512 Byte bis 4 MB (Stand 2008). Ein Teil des theoretisch möglichen Adressraums wird i. Allg. für Betriebssystem-Funktionen reserviert. Ein Prozess kann unter Windows auf einem 32 Bit-Rechner z. B. nur 2 GB addressieren, da die andere Hälfte des Adressraums reserviert ist. Unter Linux bzw. Windows2003-Server stehen dem Prozess in einem 32 Bit-Rechner 3 GB zur Verfügung, ein GB ist reserviert.

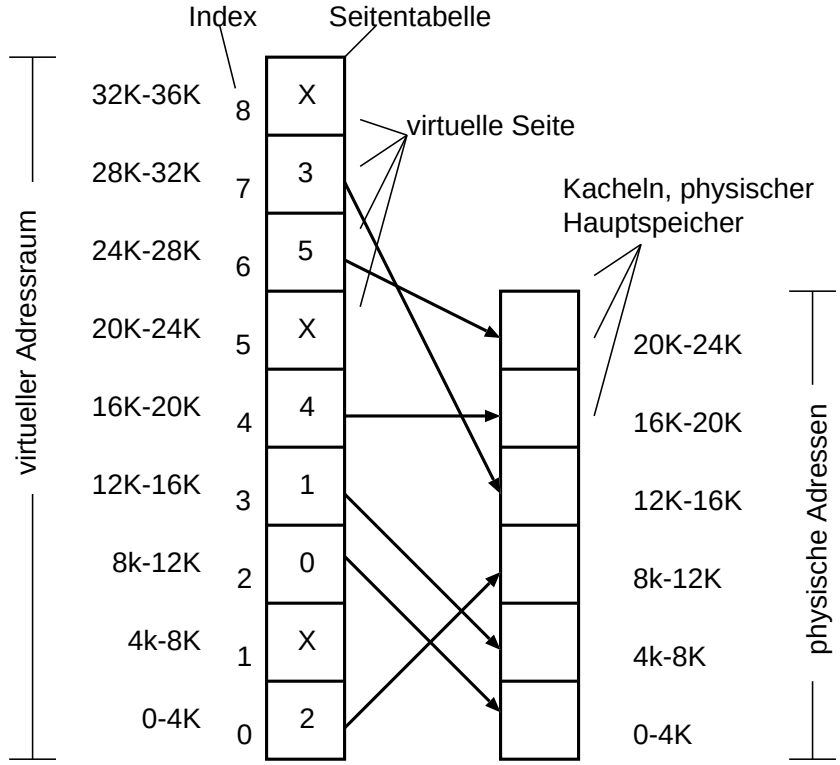


Figure 14: Seitentabelle zur Abbildung von virtuellen auf physische Adressen

Der physischen Hauptspeicher wird in "'Kacheln"' gleicher Größe aufgeteilt. Wird ein Prozess in den Speicher geladen, so werden die Speicherseiten des Prozesses in die Kacheln des Hauptspeichers kopiert. Dabei müssen nicht alle Seiten des Prozesses gleichzeitig im Hauptspeichers vorhanden sein. Seiten, auf die der Prozess momentan nicht zugreifen muss, können auf die Festplatte ausgelagert werden.

Bei jedem Speicherzugriff muss die virtuelle Adresse in eine physische Adresse umgerechnet werden. Die virtuelle Adresse besteht aus zwei Teilen: Seitennummer und Offset innerhalb der Seite. Um die passende physische Adresse zu finden, muss die Seitennummer durch die Kachelnummer ersetzt werden. Der Offset bleibt gleich, er kann also einfach angehängt werden und muss nicht addiert werden. Die Kachelnummer entspricht der physischen Anfangsadresse der Seite.

Besteht die virtuelle Adresse aus n Bit Seitennummer und k Bit Offset, so ergeben sich fol-

gende Größen: Eine Seite und damit eine Kachel ist 2^k Byte (bzw. Speicherworte) groß. Ein Prozess kann 2^n Seiten adressieren, abzüglich dem Teil des Adressraums, der für Betriebssystem-Routinen reserviert ist. Eine physische Seitenanfangsadresse liegt stets so, dass die k niederwertigsten Bits 0 sind.

Die Umrechnung von virtuellen auf physische Adresse wird i. Allg. von einer Hardware-Einheit in Zusammenarbeit mit dem Betriebssystem durchgeführt. Die Hardware-Einheit wird i. Allg. als MMU (Memory Management Unit) bezeichnet.

2.6.3 Umrechnung von virtuellen Adressen in physische Adressen

Es gibt mehrere Möglichkeiten, um virtuelle Adressen in physische Adressen umzurechnen. Zunächst genügt eine einfache Tabelle: Das Betriebssystem hält für jeden Prozess eine Seitentabelle mit einem Eintrag für jede mögliche Seitennummer. Die Seitennummer einer virtuellen Adresse wird dann als Index in die Seitentabelle verwendet. Dort steht die Kachelnummer. An die Kachelnummer wird nun der Offset angehängt, so dass sich die physische Adresse des Speicherwortes ergibt.

Beispiel:

Eine virtuelle Adresse besteht aus 12 Bit Seitennummer und 12 Bit Offset.

Das Bitmuster 000000 000111 000000 011001 ist also zu lesen als Seitennummer 7 und Offset 25. Verwendet man die Seitentabelle aus Abb. 14, so steht im Eintrag mit Index 7 die Kachelnummer 3 (binär 00000 0000011).

Hängt man den Offset an, so entsteht die physische Adresse: 000000 000011 000000 011001 (12313 oder 12 K + 25).

Zusätzlich zur Kachelnummer vermerkt das Betriebssystem in der Seitentabelle noch weitere Informationen zu jeder Seite. Diese können z. B. folgende Angaben umfassen:

- "'present flag"', gibt an, ob sich die Seite im Hauptspeicher befindet.
- "'rw flag"', gibt an, ob der Inhalt der Seite geändert werden darf.
- "'dirty flag"', gibt an, ob der Inhalt der Seite seit einer gewissen Zeit verändert wurde (näheres s. u.).
- "'access flag"', gibt an, ob der Prozess seit einer gewissen Zeit auf die Seite zugegriffen hat (näheres s. u.).

Die Umrechnung durch eine einfache Tabelle ist nur sinnvoll, wenn der Adressraum nicht oder nur wenige Größenordnungen größer ist als der physische Hauptspeicher. Ansonsten wird die Tabelle so groß, dass sie einen zu großen Teil des Hauptspeichers belegt.

Bei einem Adressraum von 4 G und einer Seitengröße von 4 K kann jeder Prozess 1 M Seiten adressieren; die Tabelle enthält also 1 M Einträge. Wenn ein Eintrag 3 Byte umfasst, benötigt die Tabelle also 3 MB Speicherplatz. Bei 64-Bit Prozessoren ist der Adressraum i. Allg. 2^{48} bis 2^{64} Byte groß. Bei einer Seitengröße von 4 K kann jeder Prozess 2^{36} bzw. 2^{52} Seiten adressieren. Eine Tabelle, dieser Größenordnung ist aber für heute existierende Rechner etwas unhandlich.

Die Größe der Seiten stellt einen Kompromiss dar, zwischen dem Speicherplatz, der für die Seitenverwaltung benötigt wird und dem Verschnitt, der dadurch entsteht, dass jeder Prozess i. Allg. einige Seiten nur teilweise füllt.

Gesucht ist also ein Verfahren, das eine schnelle Umrechnung erlaubt, aber nur wenig Speicher belegt. Dazu gibt es folgende Ansätze:

- ▷ Mehrstufige Tabellen (Digitalbaum)
- ▷ Invertierte Liste (Hashing)

Mehrstufige Tabellen: Die Seitennummer in der virtuelle Adresse wird – wie in Abb. 15 dargestellt – unterteilt in eine Seitennummer erster Stufe und eine Seitennummer zweiter Stufe. In der ersten Tabelle steht ein Verweis (Pointer) auf eine Tabelle der zweiten Stufe. Die Seitennummer zweiter Stufe wird als Index für diese Tabelle verwendet um die Kachelnummer zu ermitteln. Die Seitennummer kann auch in mehr als zwei Stufen unterteilt werden.

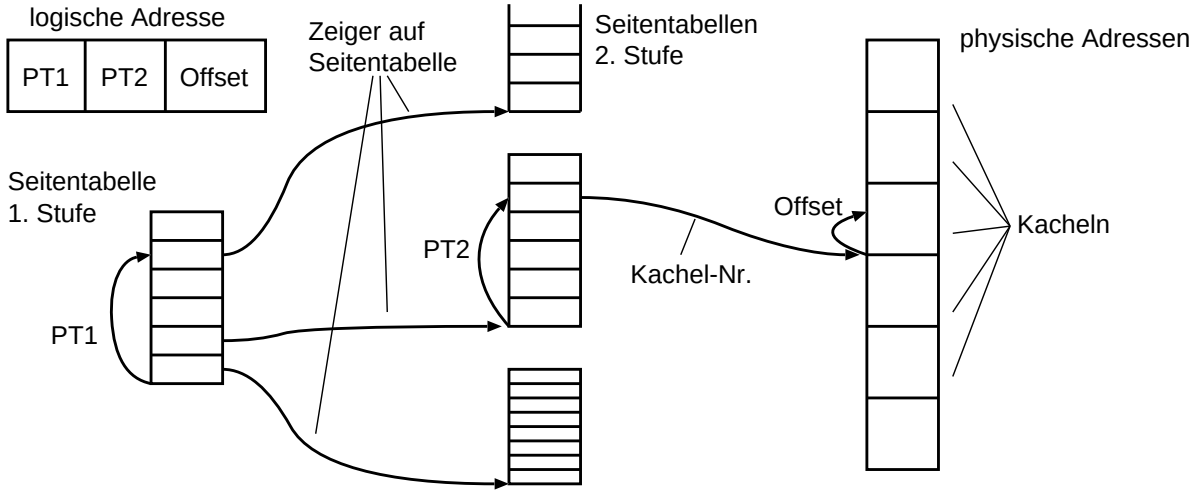


Figure 15: Zweistufige Abbildung von virtuellen auf physische Adressen

Hierbei ergibt sich statt einer großen Seitentabelle ein Baum aus kleineren Seitentabellen. Die Höhe des Baumes ist meist nur 2 oder 3 (2- oder 3-stufiges Paging). Einige bzw. viele der Seitentabellen der zweiten bzw. dritten Stufe werden i. Allg. nicht benutzt und können daher frei bleiben. Dieses Verfahren nutzt den Speicher, der für die Seitentabellen verwendet wird, effektiver als eine einzige, lineare Tabelle. Abb. 15 zeigt Zugriff über eine zweistufige Seitentabelle. Die Seitentabelle der ersten Stufe und mindestens eine Seitentabelle der zweiten Stufe sind immer notwendig, das Betriebssystem legt für jeden Prozess also mindesten zwei Seitentabellen an. Der Wert PT1 ist der Index in die erste Seitetabelle. Der Wert PT2 ist der Index in eine Seitetabelle zweiter Stufe. Nur wenn das Programm Adressen nutzt, die nicht mit der/den bisher vorhanden Seitentabelle(n) angesprochen werden können, muss das Betriebssystem eine weitere Tabelle in der zweiten Stufe anlegen.

Übungsaufgabe:

Gegeben: Gegeben ist das Format einer virtuelle Adresse:

Länge: 21 Bit, davon PT1: 4 Bit, PT2: 8 Bit, Offset: 9 Bit. Das System verwendet 2-stufiges Paging. PT1 ist der Index in die erste Seitentabelle, PT2 der Index in eine Seitentabelle der 2. Stufe.

Ein Prozess belegt Adr. 0 - 180KB für Text und Heap und die obersten 70KB des Adr.-Raums für den Stack.

Frage: Wieviel Platz belegt die Gesamtheit der benötigten Seitentabellen, wenn ein Eintrag der Seitentabelle 4 Byte belegt (dies gilt für die 1. und für die 2. Stufe)? Hinweis: Wieviel Platz belegt die Seitentabelle 1. Stufe? Wieviel Platz belegt eine Seitentabelle 2. Stufe? Wieviele Seitentabellen 2. Stufe werden benötigt?

Zum Vergleich: Nehmen Sie an, das System verwendet eine einstufigen Tabelle. Eine Adresse besteht aus 12 Bit Seitennummer und 9 Bit Offset. Wieviel Platz belegt die 1-stufige Seitentabelle in diesem Fall?

Invertierte Liste: Diese Technik kann verwendet werden, wenn der Adressraum wesentlich größer ist, als der tatsächlich genutzte Hauptspeicher. Jeder Prozess enthält eine Seitentabelle, die ungefähr so viele Einträge aufnehmen kann, wie der Prozess Seiten nutzt[3]. Als Nährung für die maximale Anzahl der genutzen Seiten wird i. Allg. die Anzahl der Kacheln genommen. Die Tabelle kann aber auch etwas größer gewählt werden, die Größe orientiert sich aber an der Anzahl der genutzen Seiten bzw. an der Anzahl der Kacheln. Die Seitennummer kann nun nicht mehr unmittelbar als Index verwendet werden, da der Wertebereich der Seitennummer i. Allg. größer ist als die Tabelle. Der Index in die Seitentabelle wird durch Hashing aus der Seitennummer berechnt. Dadurch kann die gesuchte Kachelnummer schnell gefunden werden (s. Abb. 16). Aus der Seitennummer wird also durch eine Hash-Funktion ein Hash-Wert berechnet, der als Index in die Tabelle dient. Die Hash-Funktion wird so gewählt, dass sie die Werte aus dem relativ großen Wertebereich quasi zufällig – aber natürlich immer gleich – auf den relativ kleinen Index-Bereich abbildet. Dies kann z. B. durch Bit-Operationen (XOR-Faltung) oder durch die Modulo-Funktion erreicht werden. Als Divisor der Modulo-Funktion wählt man i. Allg. eine Primzahl, die nicht in der Nähe einer Zweierpotenz liegt.

Bei jedem Zugriff auf die Tabelle wird also der Index-Wert aus der Seitennummer berechnet. Wird mehr als eine Seitennummer auf ein und denselben Hash-Wert abgebildet (Kollision), so ergibt sich eine (i. Allg. kurze) Liste von Kachelnummern[4]. Die Größe des Adressraums kann bei diesem Verfahren viele Größenordnungen größer als der physische Hauptspeicher sein; die Größe der Seitentabelle hängt nur von der Größe des physischen Hauptspeichers ab.

Normalerweise belegt alle Programm, die zu einem Zeitpunkt auf einem Rechner laufen, nicht wesentlich mehr Seiten als der Hauptspeicher Kacheln besitzt, da die Leistung des Systems ansonsten deutlich absinkt. Wenn mit diesem Verfahren wesentlich mehr Seiten als Kacheln verwaltet werden, so sollte die Seitentabelle ungefähr so groß wie die Anzahl der verwendeten Seiten sein. Zusammen mit einer guten Hash-Funktion stellt dies sicher, dass die Trefferliste an meisten Stellen der Seitentabelle nur jeweils eine Seitennummer enthält.

Eine weitere Ersparnis von Speicherplatz ergibt sich dadurch, dass zu jeder Seitennummer auch die PID des Prozesses abgespeichert wird. In diesem Fall genügt eine Seitentabelle für alle Prozesse des Systems.

[3] Die Bezeichnung ist also etwas irreführend, es handelt sich nicht um eine verkettete Liste sondern um eine Hashing-Tabelle

[4] Es gibt noch andere Verfahren um Kollisionen aufzulösen und weitere Varianten von Hashing

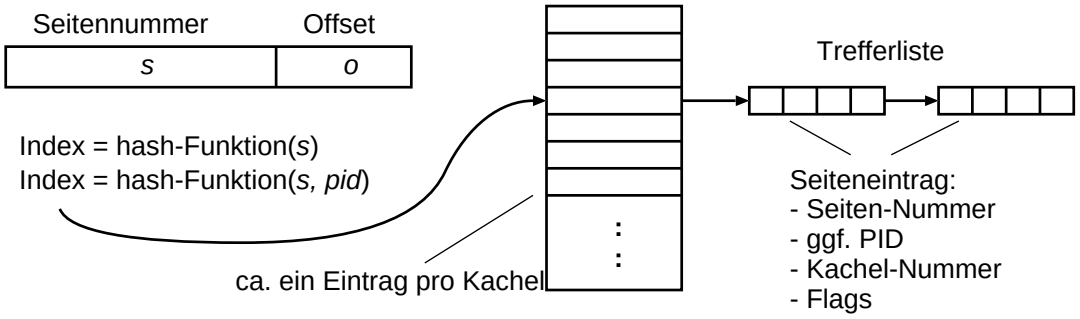


Figure 16: Abbildung von virtuellen auf physische Adressen durch eine invertierte Liste mit Hashing

Eintrag in der Seitentabelle: Ein Eintrag in der Seitentabelle enthält i. Allg. folgende Elemente

- ▷ Kachelnummer: Physische Anfangsadresse der Seite im Hauptspeicher
- ▷ gültig (valid): Die angegeben Kachelnummer stimmt bzw. die Seite ist ausgelagert
- ▷ dirty: Die Seite ist seit dem letzten Laden (nicht) verändert worden
- ▷ reference: Auf die Seite wurde lesend oder schreibend zugegriffen
- ▷ Schutz: Read- Write- und Execute-Recht (nicht) vorhanden
- ▷ auslagerbar: Seite darf (nicht) ausgelagert werden
- ▷ ggf.: Seitennummer und PID (s. u.)

Wenn das Verfahren der invertierte Liste mit Hashing eingesetzt wird, so enthält ein Eintrag auch noch die Seitennummer, da evtl. mehr als eine Seitennummer auf ein und denselben Hashwert abgebildet wird. Der richtige Eintrag muss dann in der Liste der "'Treffer-Seiten"' gesucht werden. Außerdem wird evtl. auch die PID mit abgespeichert, so dass eine globale Tabelle für alle Prozesse verwendet werden kann.

2.6.4 Caching der Seitenadressen

Die oben beschriebenen Verfahren benötigen mehrere zusätzlich Speicherzugriffe, um die physische Adresse für einen Speicherzugriff zu ermitteln. Wäre eines dieser Verfahren tatsächlich bei jedem Zugriff nötig, würde die Ausführungsgeschwindigkeit der Programme stark einbrechen. Da die meisten Prozesse häufig auf einige wenige Seiten zugreifen, lassen sich die meisten Speicherzugriffe durch Caching beschleunigen. Man unterscheidet dabei zwischen Hardware basiertem Chaching in einem (Hardware-)Baustein zur Speicherverwaltung und Software-Caching.

Software-Caching verwendet eine relativ kleine invertierte Liste, die nur einen Teil der verwendeten Seiten aufnehmen kann. Da nicht alle Seiten aufgenommen werden müssen, kann auf die Überlaufliste verzichtet werden. Bei einer Kollision verdrängt die neue Seite die bisherige aus der Tabelle.

Table 3: Aufbau eines Translation Lookaside Buffer

Gültig	evtl. PID	Virtuelle Seite	dirty	ref	Schutz	Kachel
1	1042	140	1	0	RW	31
1	1042	20	0	1	RX	38
1	557	20	0	0	RX	44
0	1042	131	0	1	RX	17
1	557	861	0	0	RX	08
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Hardware-Caching Der (Hardware-)Cache für die Seiten-Adressen wird Translation Lookaside Buffer (TLB) genannt. Der TLB ist normalerweise ein Assoziativspeicher innerhalb der Memory Management Unit (MMU). Die meisten modernen Prozessoren für Desktop und Server-Systeme besitzen eine MMU. In einer Zeile im TLB stehen die gleichen Daten wie in einer Zeile der Seitentabelle und evtl. die PID des Prozesses zu dem die Seite gehört.

Als Eingabe dient die virtuelle Seiten-Adresse ohne Offset. Ist diese Adresse irgendwo im TLB gespeichert, so wird sie durch den Assoziativspeicher schnell gefunden. Aus der Spalte "'Kachel"' kann dann unmittelbar die physische Adresse der Seite entnommen werden. Wenn die virtuelle Seiten-Adresse nicht im TLB gespeichert ist, erzeugt der TLB einen Interrupt. Das Betriebssystem ermittelt darauf hin die physische Adresse der Seite über die Seitentabelle (s. Abschnitt 2.6.2). Die Seite wird in den Hauptspeicher eingelagert; anschließend wird eine Zeile im TLB mit dem Eintrag der neu gefundenen Seite überschrieben.

Wenn die PID nicht in den TLB-Einträgen gespeichert ist, dürfen zu einem Zeitpunkt nur Einträge eines Prozesses im TLB vorhanden sein. Sonst ist die Seitennummer nicht eindeutig. Bei einem Prozesswechsel werden dann alle Einträge als "'nicht gültig"' markiert. Ist die PID Teil des TLB-Eintrags, so ist die Kombination aus PID und Seitennummer stets eindeutig. Es können dann Einträge verschiedener Prozesse gleichzeitig im TLB stehen.

2.6.5 Ein- und Auslagern von Seiten

Wenn ein Prozess auf eine Adresse der Seite p_1 zugreift, die sich im Augenblick nicht im Hauptspeicher befindet, muss die Seite eingelagert werden. Dies stellt die MMU fest, wenn sie auf einen Eintrag in der Seitentabelle zugreift, dessen Gültigkeits-Flag zurückgesetzt ist. Die MMU erzeugt dann eine "'Trap"' und verzweigt so zum Betriebssystem, das nun die Seite einlagern muss.

- ▷ Die Anweisung, die den Seitenfehler erzeugt hat, wird unterbrochen, der Prozesszustand wird gesichert.
- ▷ Das Betriebssystem prüft, ob sich die Seite p_1 auf der Festplatte befindet. Wenn nicht, liegt ein Zugriffsfehler vor, das Betriebssystem beendet i. Allg. den Prozess.
- ▷ Wenn sich die Seite auf der Festplatte befindet, sucht das Betriebssystem nach einer Seite p_2 in der Kachel f, die ausgelagert werden kann (s. Abschnitt 2.6.7).
- ▷ Wenn p_2 als modifiziert markiert ist ("'dirty"'), so wird p_2 auf die Festplatte kopiert.
- ▷ In der Seitentabelle wird der Eintrag zu p_2 als ungültig markiert.

- ▷ Die Seite p_1 wird von der Festplatte in die Kachel f kopiert.
- ▷ In der Seitentabelle wird in den Eintrag zu p_1 die Kachelnummer f geschrieben
- ▷ Im TLB der MMU wird ein Eintrag für die Seite p_1 erstellt.
- ▷ Die unterbrochene Anweisung des Prozesses wird erneut ausgeführt.

2.6.6 Speicherverwaltung in Linux

Linux verwendet Adressierung durch Paging; mehrere Seiten werden zu Bereichen zusammengefasst. Diese werden z. T. auch als Segmente bezeichnet; sie haben aber nichts mit der Adress-Rechnung zu tun. Die Bereiche verwalten folgende Informationen:

- Art der Nutzung: "'shared"' oder exklusiv von *einem* Prozess genutzt
- Zugriffsrechte (lesend, schreibend)
- auslagerbar
- Wachstumsrichtung (nach oben oder nach unten)

2.6.7 Verdrängungsstrategien / Replacement Strategies

Laufen auf einem Rechner viele und/oder große Programme, so reicht der physisch vorhandene Hauptspeicher oft nicht aus. D. h. ein Programm fordert eine Seite an, während alle Kacheln im Hauptspeicher belegt sind. Hier stellt sich die Frage, welche Seite aus dem Hauptspeicher entfernt werden soll, um für die angeforderte Seite Platz zu machen.

Die gleiche Frage stellt sich auch, wenn ein neuer Eintrag in den gefüllten TLB eingetragen werden soll: Welcher Eintrag soll überschrieben werden? Die Grundüberlegung sieht folgendermaßen aus: Die Wahrscheinlichkeit, dass in Zukunft auf eine Seite zugegriffen wird ist um so größer, je häufiger in der jüngsten Vergangenheit auf diese Seite zugegriffen wurde. Das Betriebssystem versucht also, eine Seite auszuwählen, auf die möglichst lange nicht mehr zugegriffen wurde.

Die folgenden Abschnitte skizzieren einige häufig genannten Strategien. Um die Qualität der Strategien zu vergleichen, simuliert man deren Verhalten für vorgegeben Zugriffsfolgen. Ein Zugriffsfolge ist eine Folge von Seitennummern; sie wird auch "'referenzierender String"' genannt.

Optimale Reihenfolge Sei i_p die Anzahl der Befehle, die abgearbeitet werden müssen, bis die Seite p wieder referenziert wird. Wähle die Seite, deren Wert i_p am größten ist. Die optimale Reihenfolge kann i. Allg. erst im Nachhinein festgestellt werden; sie kann also nicht tatsächlich eingesetzt werden. Die optimale Reihenfolge kann aber als Vergleichsmaßstab dienen, um zu beurteilen, wie nahe eine Strategie an das Optimum herankommt.

FIFO Die Seite, die am längsten im Speicher ist, wird ausgewählt. Diese Strategie führt – im Vergleich zu den folgenden Strategien – zu schlechten Ergebnissen.

Least Recently Used (LRU) Die Seite, die am längsten nicht referenziert wurde, wird ausgewählt. Diese Strategie führt i. Allg. zu guten Ergebnisse, auch im Vergleich zur optimalen Reihenfolge. Die Strategie ist allerdings relativ aufwendig durchzuführen und wird daher nicht eingesetzt.

Eine Möglichkeit der Implementierung besteht darin, die Seiten als Liste zu organisieren. Bei jedem Zugriff auf eine Seite wird diese Seite an den Beginn der Liste geholt. Die letzte Seite der Liste wird ausgelagert.

Die folgenden Strategien sind mehr oder weniger gute Annäherungen an die LRU-Strategie, die sich effizient implementieren lassen.

Not Recently Used (NRU) Zu jeder Seite wird ein Flag geführt (R-Bit). Jedesmal, wenn die Seite referenziert wird (lesend oder schreibend), setzt die MMU dieses Flag. In regelmäßigen Zeitabständen setzt das Betriebssystem alle Flags auf 0. Muss eine Seite ausgelagert werden, so wählt das Betriebssystem eine Seite, deren Flag nicht gesetzt ist (soweit vorhanden). Eine Verfeinerung besteht darin, dass zu jeder Seite ein zweites Bit, das M-Bit geführt wird. Es wird zusätzlich gesetzt, wenn die Seite modifiziert wird. Das M-Bit wird nur zurückgesetzt, wenn die Seite auf die Festplatte gespeichert wird. Das R-Bit und das M-Bit bilden nun eine Binärzahl mit Wertebereich 0 bis 3. Muss eine Seite ausgelagert werden, so wählt das Betriebssystem eine Seite, deren Wert in der R-M-Zahl minimal ist. Auch diese Strategie wird i. Allg. nicht eingesetzt, da die unten beschriebenen Strategien bessere Ergebnisse liefern.

Aging Zu jeder Seite wird ein Zähler geführt. In regelmäßigen Zeitabständen verschiebt das Betriebssystem den Inhalt aller Zähler um ein Bit nach rechts, wobei von links eine 0 nachgezogen wird (dies entspricht der Division durch 2). Wenn auf eine Seite zugegriffen wird, wird das höchstwertigste Bit des Zählers gesetzt. Eine Seite mit Zählerstand 0 oder – falls eine solche nicht vorhanden ist – die Seite mit dem niedrigsten Zählerstand wird ausgelagert. Eine typisch Zählerbreite ist 8 Bit.

Second Chance/Clock Zu jeder Seite wird ein Flag geführt (1 Bit). Jedesmal, wenn die Seite referenziert wird, setzt die MMU dieses Flag im TLB bzw. das Betriebssystem in der Seitentabelle. Das Betriebssystem führt einen Zeiger, der auf eine Seite zeigt. Muss eine Seite ausgelagert werden, so prüft das Betriebssystem die Seite, auf die der Zeiger zeigt. Wenn das Flag nicht gesetzt ist, so wird die Seite ausgelagert. Wenn das Flag gesetzt ist, so wird das Flag zurückgesetzt, die Seite bleibt im Speicher (second chance) und der Zeiger wird auf die nächste Seite verschoben. Anschließend wird diese Seite geprüft. Das Verfahren wird solange fortgesetzt, bis eine Seite mit nicht gesetztem Flag gefunden wird. Nach der letzten Seite wird der Zeiger auf die erste Seite gesetzt.

Das zuletzt genannte Verfahren wird häufig eingesetzt. Die Vorteile sind: Einfach zu implementieren; häufig referenzierte Seiten haben gute Chancen im Hauptspeicher zu bleiben; die Suche nach einer Seite, die ausgelagert werden kann, muss i. Allg. nicht alle sondern nur einen relativ kleinen Teil der Seiten betrachten.

In [6] wird zwischen den Verfahren Second Chance und Clock unterschieden: Second Chance beginnt die Suche nach einer auslagerbaren Seite stets an derselben Stellen, z. B. bei der ersten Seite. Clock beginnt die Suche nach einer auslagerbaren Seite unmittelbar nach der Seite,

die zuletzt ersetzt wurde. Normalerweise wird nur das letztgenannte Verfahren (hier Clock) eingesetzt. Häufig wird mit dem Begriff Second Chance auch der Clock-Algorithmus bezeichnet.

Ein weiterer Aspekt ist bei der Auswahl der auszulagernden Seite zu beachten: Wenn die Seite nicht verändert wurde, seit sie eingelagert wurde, muss die Seite nicht auf die Festplatte geschrieben werden. Veränderte Seiten müssen dagegen auf den Auslagerungsspeicher kopiert werden. Da das Schreiben auf die Festplatte relativ aufwendig ist, versuchen Betriebssysteme dies soweit wie möglich zu vermeiden. Eine Seite wird daher nicht nur markiert, wenn sie referenziert wird, sondern zusätzlich auch wenn sie verändert wird ("'dirty bit"'). Die oben angegeben Strategien werden dann so modifiziert, dass bevorzugt unveränderte Seite ausgewählt werden.

Eine zweite Überlegung ist die: Soll die Verdrängungsstrategie nur auf die Seiten des gerade rechnenden Prozesses angewendet werden oder auf alle Seite, unabhängig zu welchem Prozesse sie gehören? Lokale Strategie vs. globale Strategie.

Working Set Für die meisten Prozesse trifft folgende Beobachtung zu. Sie verbringen die meiste Zeit in einem kleinen Teil ihres Speichers. Dieser Teil wird "'Working Set"' genannt. Das Working Set ist meist nicht über die gesamte Lebensdauer eines Prozesses konstant, aber zu jedem Zeitintervall gibt es meist einen solchen relativ kleinen Teil des Speichers, der die meisten Zugriffe erzeugt. Kann das Working Set eines Prozesses vollständig in den Haupt-

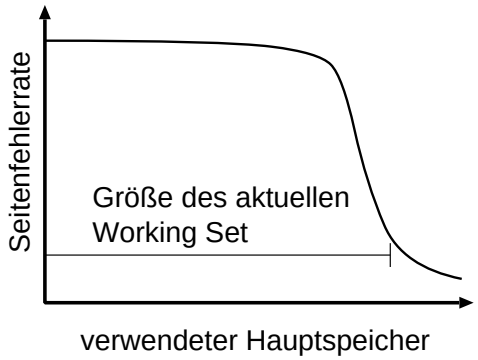


Figure 17: Anzahl der Seitenfehler eines Prozesses in Abhängigkeit seines Speicherplatzes

speicher geladen werden, erzeugt der Prozess nur noch wenige Seitenfehler. Wenn nur ein kleiner Teil des Working Sets nicht in den Hauptspeicher passt, so steigt die Rate der Seitenfehler stark an. D. h. es genügt schon, dass ein kleiner Teil des Working Sets nicht in den Hauptspeicher passt, um viele Seitenfehler zu erzeugen.

Die beste Performance erreicht ein System, wenn alle Seiten des Working Sets in den Hauptspeicher passen und durch den TLB adressiert werden können.

Der WSClock-Algorithmus Der Clock-Algorithmus kann erweitert werden, um das Working Set eines Prozesses zu berücksichtigen. Zu jeder Seite wird die Zeit des letzten Zugriffs gespeichert. Außerdem erhält jede Seite ein M-Bit in dem vermerkt wird, ob die Seite modifiziert wurde und ein V-Bit ("'Vormerk-Bit"'), das angibt, ob die Seite zum Auslagern vorgemerkt

ist. Bei Zugriff auf eine Seite wird das R-Bit gesetzt und das V-Bit zurückgesetzt. Außerdem wird der Zeitstempel auf die aktuelle Zeit gesetzt. Der Clock-Algorithmus lagert eine Seite aus, wenn das R-Bit auf 0 gesetzt ist. Der WSClock-Algorithmus prüft in diesem Fall erst, wie lange der letzte Zugriff zurückliegt (Zeit: Δt). Es werden insgesamt vier Fälle unterschieden:

- Das R-Bit ist gesetzt: Das R-Bit wird zurückgesetzt, der Zeiger rückt eine Stelle weiter.
- Das R-Bit ist nicht gesetzt.
  - Δt ist kleiner als ein Schwellwert τ: Die Seite gehört zum Working Set und wird nicht ausgelagert, der Zeiger rückt eine Stelle weiter.
  - Δt ist größerer als τ, das M-Bit ist gesetzt: Die Seite wird zum Speichern vorgemerkt (V-Bit wird gesetzt), der Zeiger rückt eine Stelle weiter.
  - Δt ist größerer als τ, das M-Bit ist nicht gesetzt: Die Seite wird ausgelagert.

Wenn der Zeiger innerhalb eines Durchlaufs wieder an seinem Startpunkt ankommt, muss man zwei Fälle unterschieden:

- Es wurde mindesten eine Seite zum Speichern vorgemerkt.
- Es wurde keine Seite zum Speichern vorgemerkt.

Im ersten Fall läuft der Algorithmus weiter, da er früher oder später auf eine "'saubere"' Seite treffen wird. Im zweiten Fall wird eine beliebige Seite ausgelagert.

Diese Form des Algorithmus setzt voraus, dass es einen separaten Prozess gibt, der vorgemerkte Seiten auf die Festplatte speichert. In der Größe τ ist eine Annahme über das Working Set eines Prozesses kodiert: Eine Seite, die länger als τ nicht mehr referenziert wurde, gehört nicht (mehr) zum Working Set des Prozesses. Der Gewinn gegenüber dem reinen Clock-Algorithmus besteht zum einen darin, dass die Auswahl der Seiten durch den Zeitvergleich verfeinert wird. Außerdem müssen geänderte Seiten nicht synchron gespeichert werden. Da ein anderer Prozess das Speichern übernimmt, kann dies i. Allg. in Zeiten passieren, in denen der Rechner nicht voll ausgelastet ist.

2.6.8 Beispiel zu Verdrängungsstrategien

Ein Rechner verwaltet einen Hauptspeicher mit drei Kacheln. Ein Prozess mit sechs Seiten läuft auf dem Rechner. Die Zahlen in Klammern sind das R- und das M-Bit. Lesender Zugriff setzt das R-Bit, schreibender Zugriff setzt das M-Bit (modified) und das R-Bit.

Referenzierender String eines Programms, Auslagerung nach NRU

Seiten-Nr.	0	1	0	2	0	0	ZS	0	4	1	5	1	3	0	4
Zugriff	r	r	r	w	r	w		r	r	w	r	r	w	r	w
K1	0 (1,0)	0 (1,0)	0 (1,0)	0 (1,0)	0 (1,0)	0 (1,1)	0 (0,1)	0 (1,1)	0 (1,1)	0 (1,1)	0 (1,1)	0 (1,1)	0 (1,1)	0 (1,1)	4 (1,1)
K2	–	1 (1,0)	1 (1,0)	1 (1,0)	1 (1,0)	1 (1,0)	1 (0,0)	1 (0,0)	4 (1,0)	4 (1,0)	5 (1,0)	5 (1,0)	3 (1,1)	3 (1,1)	3 (1,1)
K3	–	–	–	2 (1,1)	2 (1,1)	2 (1,1)	2 (0,1)	2 (0,1)	2 (0,1)	1 (1,1)	2 (1,1)	2 (1,1)	1 (1,1)	1 (1,1)	1 (1,1)

Auslagerung nach FIFO

Seiten-Nr.	0	1	0	2	0	0	ZS	0	4	1	5	1	3	0	4
Zugriff	r	r	r	w	r	w		r	r	w	r	r	w	r	w
K1	0	0	0	0	0	0	0	0	4	4	4	4	3	3	3
K2	–	1	1	1	1	1	1	1	1	1	5	5	5	5	4
K3	–	–	–	2	2	2	2	2	2	2	2	2	2	0	0

Auslagerung nach Clock (Der * bezeichnet den "'Uhrzeiger"')

Seiten-Nr.	0	1	0	2	0	0	ZS	0	4	1	5	1	3	0	4
Zugriff	r	r	r	w	r	w		r	r	w	r	r	w	r	w
K1	0 (1)	0 (1)	0 (1)	0* (1)	0* (1)	0* (1)	0* (1)	0* (1)	4 (1)	4 (1)	4* (1)	4* (1)	3 (1)	3 (1)	3* (1)
K2	*	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	1* (0)	1* (1)	1 (1)	1 (1)	1* (0)	0 (1)	0 (1)
K3	–	*	*	2 (1)	2 (1)	2 (1)	2 (1)	2 (1)	2 (0)	2 (0)	5 (1)	5 (1)	5 (0)	4* (0)	4 (1)

Auslagerung nach LRU

Seiten-Nr.	0	1	0	2	0	0	ZS	0	4	1	5	1	3	0	4
Zugriff	r	r	r	w	r	w		r	r	w	r	r	w	r	w
K1	0	0	0	0	0	0	0	0	0	0	5	6	7	7	4
K2	–	1	1	1	1	1	1	1	4	4	4	4	3	3	3
K3	–	–	–	2	2	2	2	2	2	1	2	2	2	0	0

Prüfungsaufgabe vom Sommersemester 2007:

Das Betriebssystem eines Rechners verwaltet einen Hauptspeicher mit 5 Kacheln. Das Betriebssystem verwendet den Clock-Algorithmus. Auf dem System laufen Prozesse mit insgesamt 10 Seiten. Die Seiten der Prozesse werden gemäß der ersten Zeile der folgenden Tabellen referenziert. Wird eine Seite referenziert, so soll die Wirkung in der zugehörigen Spalte dargestellt werden. Bsp.: In der Spalte, in der die Seite 0 referenziert wird, ist dargestellt, dass die Seite 0 in den Hauptspeicher eingelagert wurde. Zahl in Klammern gibt den Wert des R-Bits an, der * Bezeichnet den Zeiger des Clock-Algorithmus. Ergänzen Sie die leeren Felder in der Tabelle.

SeitenNr.	0	1	3	2	5	1	6	3	1	4	7
K1	0(1)	0(1)									
K2	– *	1(1)									
K3	–	– *									
K4	–	–									
K5	–	–									

2.6.9 Speicherverwaltung in Unix/Linux

12. S08

In Unix/Linux laufen regelmäßig einige Hintergrundprozesse, um Speicher verfügbar zu machen: Swapper, Page Daemon (in Linux auch Flush Daemon).

Der Swapper Daemon lagert ganze Prozesse aus, wenn die Page Fault Rate über einen Schwellwert steigt.

Der Page Daemon lagert Seiten aus, um immer ca. 25% der Kacheln frei zu halten. Der Page Daemon vewendet einen modifizierten Clock-Algorithmus. Er läuft ca. 1 - 4 mal pro Sekunde. Das Einlagern von Seiten wird also von den einzelnen Prozessen angestoßen, das Auslagern übernimmt der Page Daemon.

In Linux gibt es einen weiteren Daemon-Prozess – bdflush – der regelmäßig prüft, wie viele Seiten als geändert markiert sind. Wenn der Anteil der geänderten Seiten über einem Schwellwert liegt, schreibt er Seiten auf die Festplatte zurück.

Für die Adressberechnung verwendet Linux Paging und zwar ein 2- (Pentium) oder 3-stufiges (Alpha-Chip) Verfahren.

Der Adressraum wird in Bereiche (Segmente) eingeteilt. Typische Segmente sind: Text (Programm), Daten (Heap) und Keller. Segmente werden nicht zur Adressberechnung verwendet (s. Abschnitt 2.6.1) sondern zu Verwaltung von Schutzrechten und Auslagerungseigenschaften

Der Kern von Unix/Linux verwendet eine Tabelle, die einen Eintrage zu jeder Kachel des Speichers besitzt; diese Tabelle wird als "’core map"’ bezeichnet. Der Eintrag unterscheidet freie von belegten Kacheln, er enthält den Auslagerungsplatz der Seite auf der Festplatte, einen Verweis auf den Prozess, auf das Segment und den Platz der Seite im Segment sowie einige weitere Verwaltungsinformationen.

3 Interprozesskommunikation und Synchronisation

Es gibt viele Aufgaben, bei denen Prozesse Daten austauschen müssen, z. B. beim Drucken (s. u.), bei "'copy&paste"' Operationen zwischen Prozessen, bei der Überwachung und Steuerung von Prozessen durch andere Prozesse. Es stellen sich dabei mehrere Aufgaben:

- ▷ Die Daten müssen von einem Prozess zu einem anderen übertragen werden.
- ▷ Verschiedene Prozesse dürfen sich nicht wechselseitig Stören.
- ▷ Die Übertragung von Daten muss koordiniert werden, so dass es z. B. nicht zu Über- oder Unterlauf von Pufferspeichern kommt.

Beispiel Drucken: Bei einem Multiprozesssystem können verschiedene Prozesse unabhängig voneinander Daten drucken. Ein Drucker kann aber zu einem Zeitpunkt nur einen Auftrag bearbeiten. Ein Prozess darf daher den Drucker nicht direkt ansprechen, sondern nur den Druck-Daemon. Der Druck-Daemon stellt die Druckaufträge in eine Warteschlange und gibt sie dann wohlgeordnet auf dem Drucker aus. Aber auch die Kommunikation mit dem Druck-Daemon muss koordiniert werden. Zwei Prozesse dürfen nie gleichzeitig einen Auftrag an den Druck-Daemon geben.

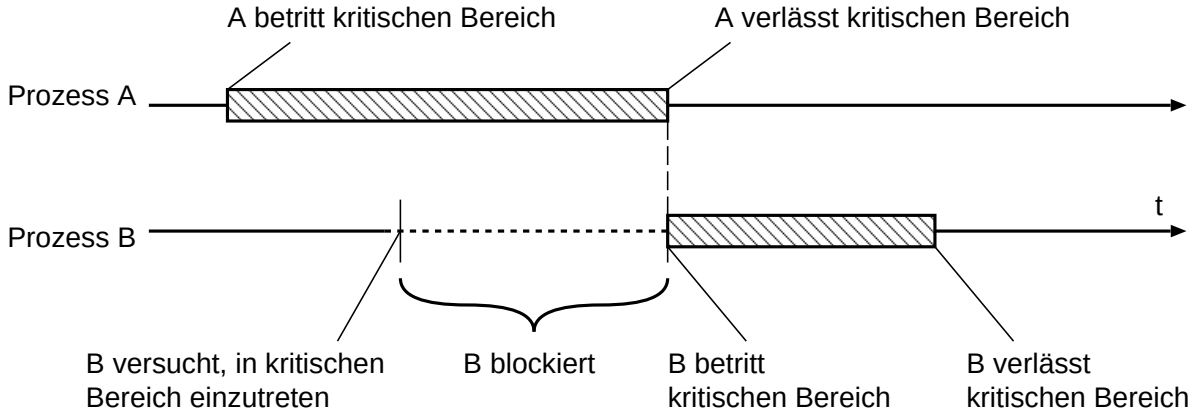


Figure 18: Wechselseitiger Ausschluss von kritischen Abschnitten

3.1 Beschreibung paralleler Prozesse durch Petri-Netze

Petri-Netze sind ein in der Informatik weit verbreiteter Formalismus zur Modellierung und Untersuchung paralleler Systeme [3, 2]. Petri-Netze sind sind verwandt mit endlichen Automaten und UML-Aktivitätsdiagrammen.

Ein Petri-Netz ist ein bipartiter, gerichteter Graph. Es gibt zwei Arten von Knoten in diesem Graphen: Stellen und Transitionen. Die Kanten des Graphs dürfen jeweils nur von Stellen zu Transitionen führen oder umgekehrt. Eine Stelle von der aus eine Kante zu einer Transitionen `t` führt, heißt Eingabestelle von `t`. Ein Stelle zu der eine Kante von Transitionen `t` führt, heißt Ausgabestelle von `t`.

Um dynamische Abläufe in einem Petri-Netz zu simulieren, werden Marken verwendet. Marken können auf Stellen liegen und durch Transitionen entfernt und erzeugt werden. Marken steuern ein Netz gemäß folgenden Regeln:

(1) Eine Transitionen t kann schalten, wenn sich auf jeder Eingabestelle von t eine Marke befindet. Außerdem muss jede Ausgangsstelle von t eine Marke aufnehmen können. Ausgangsstellen von t, die auch Eingangsstellen von t sind, können eine Marke aufnehmen.

(2) Wenn die Transitionen t schaltet, so wird von jeder Eingabestelle von t eine Marke entfernt. Außerdem wird zu jeder Ausgabestelle von t eine Marke hinzugefügt.

(3) Das Schalten einer Transition ist ein atomarer Vorgang

Wie man sieht sind Stellen passive Komponenten, während Transitionen aktive Komponenten sind. Transitionen sind i. Allg. Funktionen eines Prozess bzw. eines Threads; Abschnitt 3.2.2 erklärt, wie Stellen in einem Softwaresystem realisiert werden können.

Bedingungs/Ereignis-Netze Eine spezielle Art von Petri-Netzen heißt Bedingungs/Ereignis-Netz. Für diese Art von Netzen gilt zusätzlich folgende Einschränkung: Ein Stelle kann höchstens eine Marke enthalten. Daraus folgt eine Erweiterung der ersten Regel:

(1b) Eine Transitionen t kann schalten, wenn sich auf jeder Eingabestelle von t eine Marke befindet und wenn sich auf keiner Ausgabestelle eine Marke befindet.

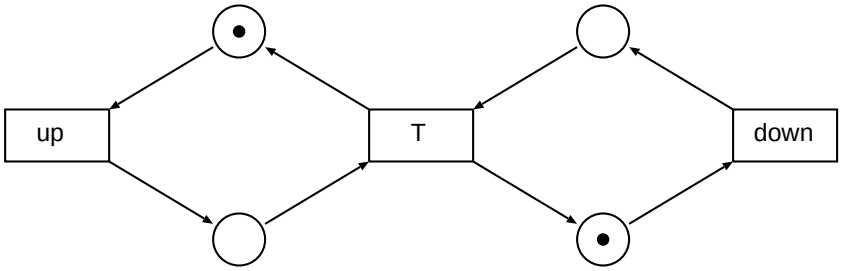


Figure 19: Ein Zweier-Zähler mit Synchronisation über Transition T

Das PetriNetz in Abb. 19 besitzt die drei Transitionen T, up und $down$ und vier Stellen. Die Transitionen T hat zwei Eingangs- und zwei Ausgangsstellen, up und $down$ besitzen jeweils eine Eingangs- und eine Ausgangsstelle. Das Netz realisiert einen Zweier-Zähler mit Synchronisation über die Transition T. Die Aktivität up kann höchstens einmal mal mehr ausgeführt werden als die Aktivität $down$ und umgekehrt.

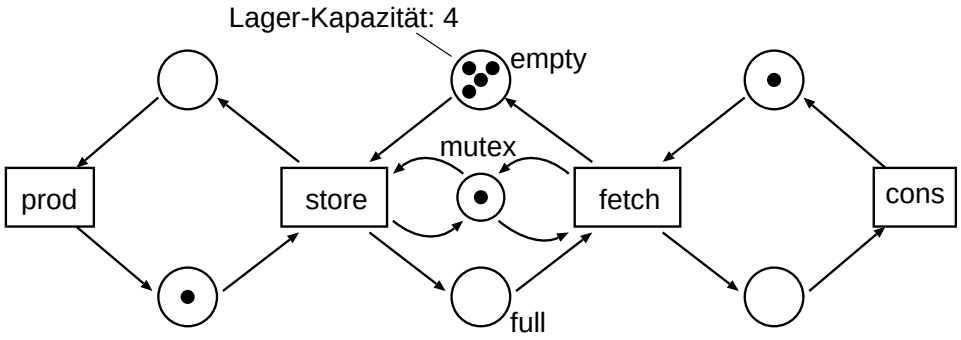


Figure 20: Synchronisation von Produzent und Konsument über einen Puffer begrenzter Kapazität

Bsp. Produzenten-Verbaucher-Problem Ein Beispiel für eine Synchronisationsaufgabe ist das Produzenten-Verbaucher-Problem. Ein Produzent stellt Teile her und legt sie in einer Lager; ein Konsument holt Teile aus dem Lager und verbaucht sie. Der Produzent kann nur arbeiten, so lange das Lager noch über freie Kapazität verfügt, der Konsument kann nur arbeiten, solange er Teile aus dem Lager holen kann.

Abb. 20 zeigt eine Modellierung des Produzenten-Verbaucher-Problems. Der Produzent kann nur so viele Teile auf Vorrat erzeugen, wie Marken vor der Transition *store* warten. Dann blockiert der Produzent, bis der Konsument mindestens ein Teil konsumiert. Der Konsument kann alle produzierten Teile holen, dann blockiert er, bis der Produzent mindestens ein Teil produziert. Im vorliegenden Beispiel gilt also:
$Anzahl(cons) \leq Anzahl(prod) \leq Anzahl(cons) + 4$.

In einem Petri-Netz ist das Schalten einer Transition ein atomarer Vorgang. In einem Programm bzw. in in einem technischen System ist i. Allg. nicht der Fall. Die Zusätzlich Stelle in der Mitte dient als sog. Mutex ("'mutual exclusion"'). Sie stellt sicher, dass die Transitionen *store* und $fetch$ nicht gleichzeitig ausgeführt werden können.

Analyse von Systemen durch Petri-Netze Verteilte Systeme können als Petrinetz modelliert werden. Das Modell kann simuliert oder analytisch untersucht werden. Die Simulation ist relativ einfach, zeigt aber i. Allg. nur einen Teil der möglichen Entwicklungen auf. Die mathematische Analyse ist schwieriger, führt aber zu allgemeineren Aussagen.

Um darzustellen, welche Zustände in einem Petri-Netz erreicht werden können, kann der Ereignisgraph (Markierungsübergangsgraph) ermittelt werden.

Jeder Knoten des Ereignisgraph repräsentiert einen Zustand des Netzes d. h. eine Belegung aller Stellen des Netzes. Bei einem Bedingungs/Ereignis-Netz mit n Stellen enthält der Ereignisgraph also 2^n Knoten. Von Knoten K_i führt eine Kante zu Knoten K_j gdw. das Netz von der Belegung K_i durch Schalten einer Transistion in die Belegung K_j überführt werden kann. Um das mögliche Verhalten eines Petri-Netzes darzustellen, genügt es, die Knoten zu betrachten, die vom Ausgangszustand aus erreicht werden können.

Ein Knoten, zu dem eine Kante hinführt und keine Kante ausgeht, entspricht einem Deadlock.

Abschnitt 3.2.2 enthält ein Beispiel zu Analyse und Implementierung von Petri-Netzen.

3.2 Synchronisation

3.2.1 Semaphor und Mutex

Ein Semaphor ist ein Abstrakter Datentyp S mit zwei Operationen:[5] $up(S)$ und $down(S)$.

S realisiert einen Zähler, der zwei oder mehr Werte annehmen kann $0 \leq S \leq n$. Für $n = 1$ spricht man von einem binären Semaphor bzw. von einem Mutex. Die Funktion $down(S)$ setzt den Wert des Semaphors dann auf 0, die Funktion $up(S)$ setzt den Wert auf 1.

Ein Prozess, der seinen kritischen Bereich betreten möchte ruft zunächst $down(S)$ auf. Danach kann der seinen kritischen Bereich betreten, anschließend ruft er $up(S)$ auf. Wenn der Wert

[5] Statt $up(S)$ und $down(S)$ werden auch die Bezeichnungen $V(S)$ und $P(S)$ verwendet.

des Semaphors 0 ist, so blockiert der Aufruf $down(S)$, bis der Wert des Semaphors durch einen Aufruf von $up(S)$ erhöht wurde.

Um einen kritischen Bereich zu schützen wird der Anfangswert des Semaphors auf 1 gesetzt. Jetzt kann ein Prozess $down(S)$ aufrufen und den kritischen Bereich betreten. Ruft ein zweiter Prozess $down(S)$ auf, so lange der erste Prozess nicht $up(S)$ aufgerufen hat, wird der zweite Prozess blockiert. Nachdem der erste Prozess den kritischen Bereich verlassen hat, ruft er $up(S)$ auf. Jetzt kann der nächste Prozess den kritischen Bereich betreten.

Die folgenden Funktionen zeigen eine unsichere Implementierung von Semaphoren.

```
1 down(S){
2   if (S == 0){
3     wait;        // warte darauf, geweckt zu werden
4   }
5   s = s - 1;
6 }
7
8 up(S){
9   s = s + 1;
10  notify;      // wecke einen Prozess, der auf S wartet
11 }             // (falls es einen solchen Prozess gibt)
```

Es kann passieren, dass zwei Prozesse bzw. Threads `down(S)` durchlaufen können, obwohl nur ein Prozess bzw. ein Thread passieren dürfte:

- ▷ Der Wert von S ist 1.
- ▷ Der erste Prozess bzw. Thread wird nach der Anweisung in Zeile 2 aber noch vor der Anweisung in Zeile 5 unterbrochen.
- ▷ Der zweite Prozess bzw. Thread durchläuft die Anweisung in Zeile 2, bevor der erste Prozess bzw. Thread die Anweisung in Zeile 5 ausführt.

Um Semaphore zuverlässig zu implementieren, muss der Prozessor eine ununterbrechbare Anweisung zur Verfügung stellen, die den Wert einer Variablen prüft *und* setzt.

Das folgende Code-Fragment zeigt eine solche Implementierung eines binären Semaphors in Pseudo-Assembler. Achtung: Hier ist die Bedeutung der Werte 0 und 1 gegenüber der Definition von Dijkstra vertauscht. Ist der Wert von Mutex 0, so ist der kritische Bereich frei, ist der Wert von Mutex 1, so ist der kritische Bereich belegt. Die Anweisung TSL (Test and Set Lock als atomare Anweisung) kopiert den Wert der Variablen Mutex in einer Register und setzt den Wert von Mutex auf 1.

```
mutex_lock:
      TSL Register, Mutex  // Wert von Mutex laden und setzten
      CMP Register, #0     // compare immediate 0: war Mutex 0?
      JZE ok               // jump on zero: zu 'ok' springen oder weiterlaufen
      CALL thread_yield    // Mutex war belegt, warten
      JMP mutex_lock       // erneut versuchen
ok:   RET                  // Mutex war 0 also frei, weiter

mutex_unlock:
      MOVE Mutex, #0       // Mutex auf 0 setzen (frei)
      RET
```

Die Anweisung TSL muss atomar sein, da sonst zwei Prozesse den Wert 0 lesen könnten, bevor einer der beiden den Wert von Mutex auf 1 setzen kann.

Um Über- oder Unterlauf von Pufferspeichern zu vermeiden, können allgemeine Semaphore eingesetzt werden. Das Semaphor wird dann mit einem Wert $n > 1$ initialisiert. Die Anzahl der nichtblockierten $down(S)$-Aufrufen ist dann maximal n plus die Anzahl der $up(S)$-Aufrufe.

Anmerkung zur Implementierung in C Semaphore können als Bibliothek zur Verfügung gestellt werden. Einzige Voraussetzung ist, dass das unterlagerte System eine atomare Funktion zum Holen *und* Setzen einer Variable besitzt wie im vorherigen Abschnitt beschrieben (alternativ: Vergleichen *und* Setzen).

Die Posix-Implementierung von Semaphoren lässt keine Benutzdefinierte Obergrenze für den Semaphor-Zähler zu. D. h. die Obergrenze ist abhängig vom Betriebssystem. In der PThread-Bibliothek gibt es für Mutex-Variable die Funktionen: `lock` und `unlock`. Die Mutex-Variable kann daher nur zwei verschiedene Werte annehmen.

3.2.2 Implementierung von Petri-Netzen mit Hilfe von Semaphoren

Ein System, das durch ein Petri-Netz beschrieben wird, kann mit Hilfe von Semaphoren implementiert werden. Transistionen werden auf Code-Blöcke abgebildet z. B. auf einen Funktionsaufruf. Stellen werden teilweise auf Semaphore abgebildet. Man unterscheidet dazu zunächst zwei Arten von Stellen: Stellen, die genau eine Transition mit genau einer anderen Transition innerhalb desselben sequentiellen Prozess verbinden und andere Stellen. Die erstere Art von Stellen werden für die Implementierung nicht berücksichtigt. Jede andere Stelle wird durch ein Semaphor realisiert. Ein Marke auf eine Stelle setzen entspricht dann der Operation `up()` bzw. `V()`. Ein Marke von einer Stellen entfernen entspricht dann der Operation `down()` bzw. `P()`. Im Beispiel unten wird das in Abb. 20 dargestellte Netz implementiert.

Semaphor mutex = 1, full = 0, empty = 4;

```
1  public class Producer {
2    Semaphor full, empty, mutex;
3    public void run(){
4      while(true){
5        produce();
6        empty.down()
7        mutex.down();
8        store();
9        mutex.up();
10       full.up
11     }
12   }
13 }
```

```
1  public class Consumer {
2    Semaphor full, empty, mutex;
3    public void run(){
4      while(true){
5        full.down();
6        mutex.down();
7        fetch();
8        mutex.up();
9        empty.up;
10       consume();
11     }
12   }
13 }
```

Achtung: Die Zeilen 6 und 7 des Producers und die Zeilen 5 und 6 des Consumers dürfen nicht vertauscht werden. Ansonsten kann es zu einem Deadlock kommen, obwohl der Ereignisgraph des Petrinetz keinen Deadlock ausweist. Angenommen die Zeilen wären vertauscht und das System ist in dem Zustand, der in Abb. 20 dargestellt ist. Der Consumer

durchläuft `mutex.down()` auf und wird dann in `full.down()` blokiert, da sich keine Marke auf `full` befindet. Der Producer wird in `mutex.down()` blockiert, beide Prozesse stehen. Der Unterschied zum Petri-Netz besteht darin, dass das Schalten einer Transition in einem Petri-Netz atomar ist, in diesem Programm aber nicht.

Beispiel Gegeben sei ein Petri-Netz gemäß Aufgabe 14 im Skript von Frau Keller. Abb. 21 zeigt den Ereignisgraph des Petri-Netz. Wie man sieht, kann das System in zwei verschiedenen Zuständen nicht mehr weiterlaufen.

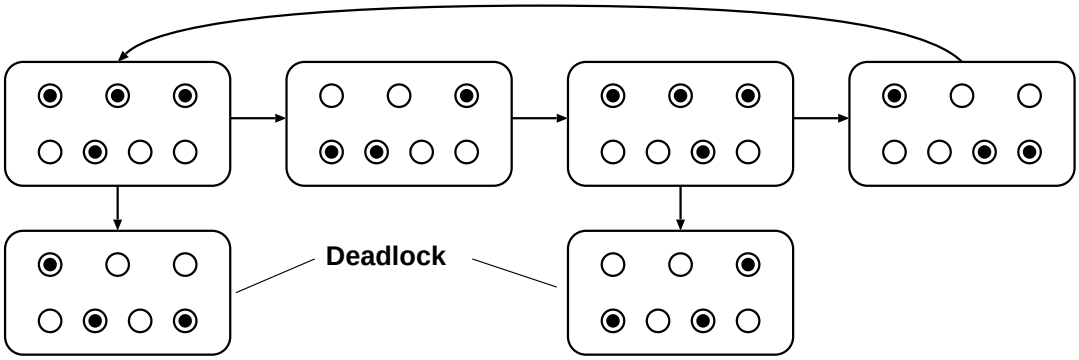


Figure 21: Ereignisgraph des Petri-Netz gemäß Aufgabe 14 im Skript SYSO von Frau Keller

Statt Pseudocode sei hier eine Implementierung als Java-Threads gegeben:

Semaphor M = 1, S1 = 1, S2 = 0;

```
1  public class Process1 {
2    Semaphor M, S1, S2;
3    public void run(){
4      while(true){
5        M.down();     // P(M)
6        a();
7        S1.down();    // P(S1)
8        b();
9        S2.up();      // V(S2)
10       M.up();       // V(M)
11     }
12   }
13 }
```

```
1  public class Process2 {
2    Semaphor M, S1, S2;
3    public void run(){
4      while(true){
5        M.down();     // P(M)
6        c();
7        S2.down();    // P(S2)
8        d();
9        S1.up();      // V(S1)
10       M.up();       // V(M)
11     }
12   }
13 }
```

Für die nicht benannten Stellen müssen keine Semaphore engesetzt werden, da die Reihenfolge der Ausführung durch die Reihenfolge der Aufrufe im Programm bereits festgelegt ist.

Wenn die Transitionen in der Reihenfolge `abcdabcd ...` verklemmungsfrei ablaufen sollen, genügen die Stellen/Semaphore $S1$ und $S2$:

Semaphor S1 = 1, S2 = 0;

```
1 public class Process1  {
2   Semaphor S1, S2;
3   public void run(){
4     while(true){
5       S1.p();     // P(S1)
6       a();
7       b();
8       S2.v();     // V(S2)
9     }
10  }
11 }
```

```
1 public class Process2 {
2   Semaphor S1, S2;
3   public void run(){
4     while(true){
5       S2.p();     // P(S2)
6       c();
7       d();
8       S1.v();     // V(S1)
9     }
10  }
11 }
```

Wenn die beiden Prozesse nicht unbedingt abwechselnd laufen sollen, sondern nur sequentiell z. B. `abababcdcdabcdcdcd ...` , so genügt ein Semaphor für den wechselseitigen Ausschluss (s. Abb. 19).

Implementierung eines allg. Semaphors Lösung zu Aufgabe 15 im Skript SYSO von Frau Keller. Stellt ein System nur binäre Semaphore zur Verfügung, so können damit allgemeine Semaphore implementiert werden.

```
1 Semaphor S, M = 1;
2 int sema;
```

```
1 public void down(){
2   M.down();      // P(M);
3   sema--;
4   if(sema < 0){
5     M.up();   // V(M);
6     S.down();
7   }else{
8     M.up();
9   }
10 }
```

```
1 public void up(){
2   M.down();      // P(M);
3   sema++;
4   if(sema == 0){
5     S.up();
6   }
7   M.up();        // V(M);
8 }
```

Die Funktion `p()` blockiert in `S.down();` den aufrufenden Prozess, falls das Semaphor `sema` belegt ist. Die Funktion `v()` deblockiert mit `S.up();` einen anderen Prozess, falls das Semaphor `sema` belegt ist.

Datenstruktur zu Semaphor Lösung zu Aufgabe 16 im Skript SYSO von Frau Keller. Das System benötigt zu jedem Semaphor eine Warteschlage für die Prozesse, die auf das Semaphor warten. Eine mögliche Implementierung ist in Abschnitt 3.2.1 in Java bzw. Pseudo-Assembler gegeben. Die Funktionen `wait()` und `notify()` verwalten die Warteschlage. In der Lösung in Pseudo-Assembler wird die Prozess-Scheduling-Warteschlange genutzt (Aufruf `thread_yield`).

Um ein Semaphor korrekt zu implementieren, müssen die Operationen testen und setzen des Semaphors ohne Unterbrechung ablaufen. Dies kann durch einen Monitor (s. Abschnitt 3.2.3) realisiert werden (in Java) oder durch eine entsprechende Assembler-Anweisung (s. Pseudo-Assembler, TSL). In C, C++ oder Pascal ist dies i. Allg. nicht möglich.

Semaphore in Unix/Linux In C unter Unix/Linux sind Semphore zu Gruppen zusammengefasst, die mindestens ein Semphor enthalten. Mit einer atomaren Aktion können dann Operationen auf allen Semphoren einer Gruppe ausgeführt werden. Dies erleichtert die Synchronisation. Wenn ein Prozess die Funktion `down()` auf mehrere Semaphore in Folge anwendet, kommt es nicht auf die die Reihenfolge an, in der die Semaphore angesprochen werden (s. Bsp. in Abb. 20). Dies entspricht dem Einziehen aller Marken von Eingangsstellen im Petri-Netz.

Timer Lösung zu Aufgabe 18 im Skript SYSO von Frau Keller.

Die Funktion `wakeMe()` könnte auch `sleep()` heißen.

```
1 IntHolder Counter;    // eine Instanz, die eine Integer enthaelt
2 Semaphor Mutex = 1, Wait = 0, Quit = 0, Tick = 0;
```

```
1  public void wakeMe(int time){
2    while (time > 0){
3      Mutex.down();
4      Counter.add(1);
5      Mutex.up();      // Mutex und
6      Wait.down();     // Wait vertauscht
7      Quit.up();
8      time--;
9    }
10 }
```

```
1  public Class Clock{
2  public void run(){
3    while(true){          // Tick.up()
4      Tick.down();        // durch
5      Mutex.down();       // Hardware
6      while (Counter.value())>0){
7        Wait.up();
8        Quit.down();
9        Counter.add(-1);
10     }
11     Mutex.up();
12   }
13 }
```

Ein Prozess ruft die Funktion `wakeMe()` auf, um sich für eine bestimmte Zeit (Anzahl Ticks) schlafen zu legen. Ruft ein Prozess `wakeMe()` auf, so zählt die Funktion den Wert des Zählers `Counter` hoch. Der Zähler enthält also die Anzahl der Prozesse, die gerade in `wakeMe()` schlafen. Anschließend blockiert der Prozess in `Wait.down()`.

Die Rechner Hardware führt bei jedem "'Tick"' die Operation `Tick.up()` aus. Der Prozess `Clock` führt dann die Schleife für jeden schlafenden Prozess einmal aus. Durch `Wait.up()` kann in jedem Schleifendurchlauf genau ein schlafender Prozess weiterlaufen und seinen Timer-Wert herunterzählen. Dieser Prozess kann bis `Mutex.down()` laufen. Wenn der Prozess `Clock` die innere Schleife beendet, steht jeder schlafende Prozess in `Mutex.down()` oder er ist noch auf dem Weg dorthin. Wenn der Prozess `Clock` dann `Mutex.up()` aufgerufen hat, kann jeder schlafende Prozess wieder bis `Wait.down()` laufen. Hat ein schlafender Prozess seinen Timer bis auf 0 heruntergezählt, so beendet er die Funktion `wakeMe()` und läuft weiter.

3.2.3 Monitore

Monitore sind ebenfalls ein Konzept, das dazu dient kritische Bereiche zu schützen. Monitore müssen in der Programmiersprache verankert sein – sie können nicht als Bibliothek zur Verfügung gestellt werden.

Ein Monitor besteht aus einer oder mehreren Funktionen, die z. B. als Klasse zusammengefasst sind. Zu einem Zeitpunkt kann nur ein Prozess bzw. ein Thread in einem Monitor aktiv sein. Versucht ein zweiter Prozess bzw. ein zweiter Thread den Monitor zu betreten, während noch ein anderer Prozess bzw. Thread im Monitor aktiv ist, so wird der zweite Prozess bzw. Thread blockiert.

In der Programmiersprache Java sind Monitore durch das Schlüsselwort `synchronized` realisiert. Diese Eigenschaft kann sich auf eine ganze Klasse beziehen, auf einzelne Methoden oder auf einen Block einer Methode. Befindet sich ein Thread aktiv in einem synchronisierten Abschnitt einer Methode, so kann kein anderer Thread einen synchronisierten Abschnitt desselben Objekts[6] betreten.

Ein Thread kann innerhalb einer Monitor-Methode die Methode `wait()` aufrufen. Der Thread wird dadurch blockiert; daraufhin kann ein anderer Thread den Monitor betreten. Ein Thread X kann innerhalb einer Monitor-Methode die Methode `notify()` aufrufen. Dies bewirkt, dass ein anderer Thread, der innerhalb des Monitors blockiert ist, weiterlaufen kann, sobald Thread X den Monitor verlassen hat.

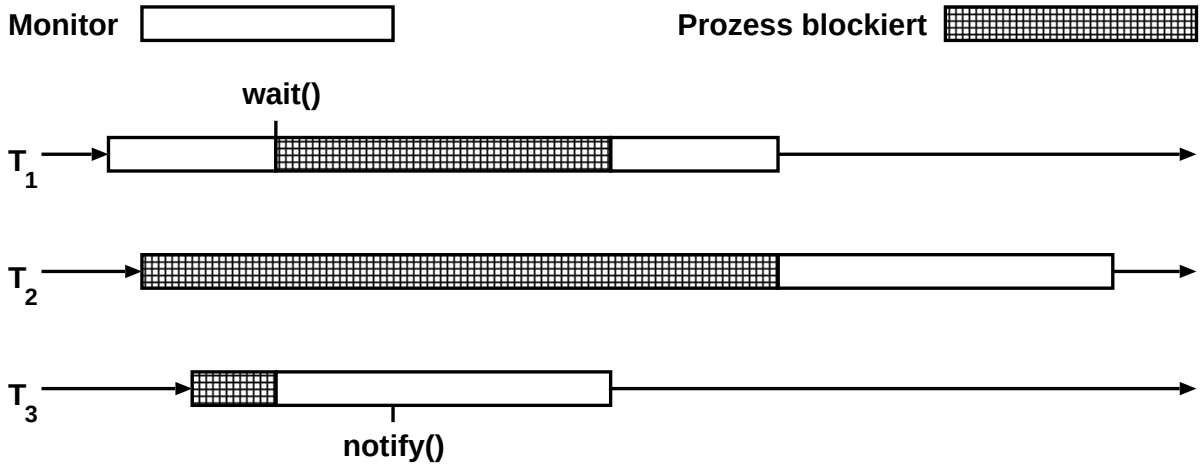


Figure 22: Drei Threads erreichen einen Monitor. Nur ein Thread kann zu einem Zeitpunkt innerhalb des Monitors aktiv sein.

In Abb. 22 erreichen drei Threads einen Monitor. Zunächst betritt T_1 den Monitor. Die beiden anderen Threads werden daher blockiert, sobald sie versuchen, den Monitor zu betreten. Der Thread T_1 ruft innerhalb des Monitors die Methode `wait()` auf und blockiert sich damit selbst. Daraufhin kann ein anderer Thread den Monitor betreten, in diesem Beispiel Thread T_3. Der Thread T_3 ruft innerhalb des Monitors die Methode `notify()` auf. Dadurch kann der Thread T_1 weiterlaufen, aber erst, wenn der Thread T_3 den Monitor verlassen hat. Wenn der Thread T_1 den Monitor verlassen hat, kann auch der Thread T_2 den monitor betreten.

Monitore sind gegenüber Semaphoren einfacher in der Handhabung und daher weniger Fehleranfällig. Sie stehen allerdings nicht in allen Programmiersprachen zur Verfügung (z. B. nicht in C/C++).

[6] Das "'Monitor-Objekt"' kann man in Java explizit angeben; i. Allg. lässt man diese Angabe weg. In diesem Fall ist das Objekt zu dem die Methode gehört das "'Monitor-Objekt"'

3.3 Signale

In POSIX-konformen Systemen können sogenannte Signale an Prozesse geschickt werden. Ein Signal bewirkt, dass der Empfänger-Prozess unterbrochen wird und eine bestimmte Funktion aufgerufen wird. Diese Funktion wird Signalhandler genannt. Wenn der Signalhandler zurückkehrt, wird der Prozess dort fortgesetzt, wo er vorher unterbrochen wurde.

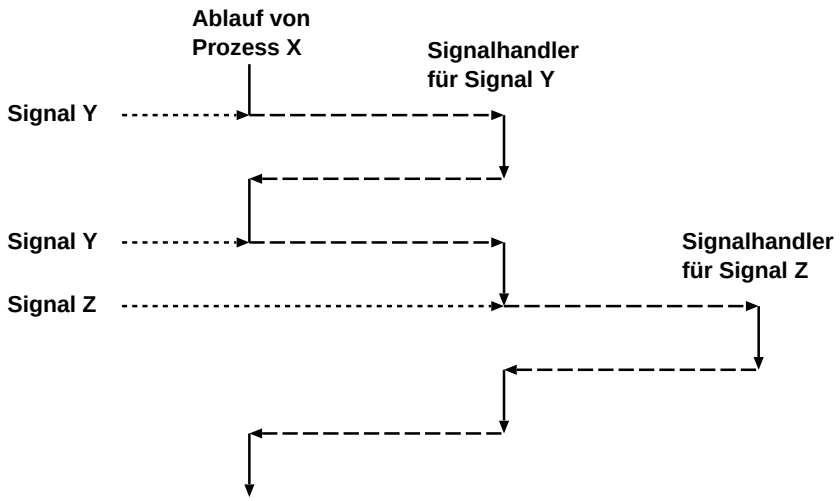


Figure 23: Der laufende Prozess wird durch ein Signal unterbrochen; der Signalhandler wird ausgeführt

Signale können z. B. von der Tastatur über ein Terminal an einen Prozess geschickt werden. Sie können aber auch durch eine Systemfunktion an einen anderen Prozess geschickt werden. Dazu muss der sendende Prozess die Prozessnummer (PID) des Empfängers kennen.

Es gibt zu jedem Signal einen vom System vorgegebenen Signalhandler. Jeder Prozess kann aber eigene Signalhandler definieren. Ein Signalhandler ist eine Funktion ohne Rückgabewert mit einem ganzzahligen Parameter `void name(int sig_nr)`. Der Parameter enthält die Signal-Nummer. Signale können also nur Benachrichtigungen sein; wenn ein Prozess Daten zu einem anderen Prozess übermitteln möchte, muss er einen anderen Mechanismus einsetzen.

Ein Prozess kann auf ein Signal warten, indem er eine entsprechende blockierende System-Funktion aufruft. Über diesen Mechanismus können sich Prozesse synchronisieren, allerdings sollten normalerweise eher Semaphore bzw. Monitore zur Synchronisation eingesetzt werden.

3.3.1 Realisierung in C unter Unix

In der C-Standard-Bibliothek heißt die Funktion, die Signale sendet `kill(pid_t pid, int sig_nr)`. Die Funktion, die einen Signalhandler setzt, heißt `sigaction(int sig_nr, const struct sigaction *act, struct sigaction *old_act)`. Ein Signalhandler sollte/muss "'reentrant"' sein, da er selbst wieder durch ein Signal unterbrochen werden kann. Ein Programmstück, das von mehreren Prozesse oder Threads gleichzeit genutzt wird, ist "'reentrant"', wenn ein Prozess bzw. Thread nie Daten eines anderen Prozess bzw. eines anderen Threads verändern kann. Das Programmstück darf also insbesondere keine System-Funktionen aufrufen,

die nicht "'reentrant"' sind. Der Signalhandler darf daher z. B. nicht die Funktion `malloc()` aufrufen, da dabei die interne Freispeicher-Verwaltung des Heaps inkonsistent werden könnte. Der POSIX-Standard definiert eine Liste "'sicherer"' Funktionen, die in einem Signalhandler aufgerufen werden können.

Wenn der Signalhandler globale, "'static"' oder Heap-Variable ändert, so muss kann es passieren, dass die Änderung durch einen anderen oder durch denselben Signalhandler überschrieben wird.

3.4 Pipes

Eine Pipe ist eine gepufferte Verbindung zwischen zwei Prozessen, über die Daten übertragen werden können. Eine Pipe besitzt zwei File-Deskriptoren (Streams) als Enden. Der eine File-Deskriptor ist zum Schreiben geöffnet, der andere zum Lesen. Ein Prozess schreibt nun über die Ausgabe-Funkion `write()` in die Pipe, während der andere über die Eingabe-Funktion `read()` aus der Pipe liest. Die Speicherkapazität einer Pipe ist begrenzt. Wenn die Pipe voll ist, blockiert der Aufruf der Ausgabefunktion. Wenn die Pipe leer ist, blockiert der Aufruf der Eingabe-Funktion.

Liest ein Prozess ein gewisse Anzahl von Bytes aus der Pipe, so werde diese Bytes aus der Pipe entfernt. Der nächste Lesevorgang liefert die nächsten in der Pipe vorhandenen Bytes. Deswegen werden Pipes typischerweise so eingerichtet, dass nur ein Prozess aus einer Pipe liest, auch wenn ggf. mehrere Prozess in die Pipe schreiben (s. Abb. 25).

Der Kernel synchronisiert konkurrierenden Schreibzugriffe auf eine Pipe bis zu einer gewissen Obergrenze. Eine Schreiboperation auf einer Pipe ist atomar, solang nicht mehr als `PIPE_BUF` Byte geschrieben werden. Der Wert der Konstanten `PIPE_BUF` ist systemabhängig, aber mindestens 512. Die Operationen müssen also nicht durch die Anwendung synchronisiert werden, solange sie "'kleine"' Nachrichten schreiben. Eine Lese-Operation liest die Daten aus der Pipe ohne Rücksicht darauf, in welchen Abschnitten diese in die Pipe geschrieben wurden. Es ist Sache der Anwendung, die Nachrichten in der Pipe voneinander zu Trennen. Eine gängige Strategie ist, alle Daten in Text-Form zu schreiben und die Nachrichten durch einen Zeilenumbruch zu trennen `\n` (ASCII-Code 10).

Die folgenden Abschnitte beschreiben verschiedene Varianten von Pipes: Anonyme Pipes, Named Pipes (fifo) und Stream Pipes.

3.4.1 Klassische (anonyme) Unix-Pipes

Einfache, anonyme Pipes können nur zwischen Eltern- und Kind-Prozess eingerichtet werden. Eine solche Pipe arbeitet im Simplex-Mode; d. h. an einem Ende der Pipe kann nur geschrieben werden, am anderen Ende kann nur gelesen werden. Der Eltern-Prozess erzeugt die Pipe (Aufruf `pipe()`) und erzeugt danach einen Kind-Prozess. Der Kindprozess hat dadurch ebenfalls Zugriff auf die Pipe. Nun schließt einer der beiden Prozesse das Schreib-Ende der Pipe und der andere das Lese-Ende der Pipe. Damit ist die Pipe als Verbindung zwischen den Prozessen eingerichtet s. Abb. 24.

Die Prozesse können nun mit den low-level Funktionen `write()` und `read()` in die Pipe schreiben bzw. aus der Pipe lesen. Aus den Datei-Deskriptoren können durch die Funk-

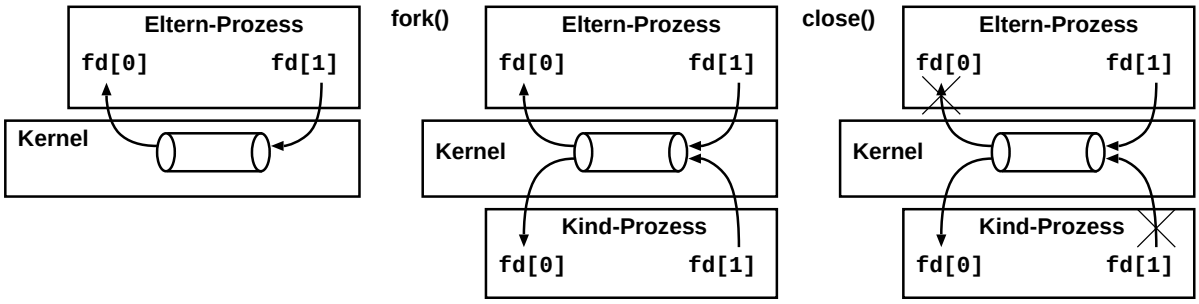


Figure 24: Aufbau einer Pipe zwischen Eltern- und Kind-Prozess

tion `fdopen()` Streams erzeugt werden. Dann lassen sich auch die Datei-Funktionen `fread()`, `fwrite()`, `fgets()`, `fputs()` usw. verwenden.

Eine anonyme Pipe verschwindet aus dem System, sobald der letzte Prozess, der die Pipe nutzt, beendet wird.

3.4.2 FIFOs / Named Pipes

Pipes können auch zwischen nicht verwandten Prozessen eingerichtet werden. Dazu wird eine Datei im Dateisystem als "'Treffpunkt"' vereinbart. Die auszutauschenden Daten werden aber nicht auf die Festplatte geschrieben. Ein Prozess legt die Pipe an. Es handelt sich um eine spezielle Art von Datei. Anschließend können Prozesse die Datei – zum Lesen oder zum Schreiben – öffnen. Da Dateinamen im Datei-System eines Rechners eindeutig sein müssen, kann es hierbei zu Namens-Kollision kommen. Es ist Sache der Anwendung, Kollisionen zu vermeiden. Named Pipes arbeiten ebenso wie die unbenannten Pipes im Simplex-Mode.

Typischerweise gibt es zu jeder Pipe nur einen lesenden Prozess (s. o.); dieser wird als Server bezeichnet. Es kann mehrere schreibende Prozesse (Clients) zu einer Pipe geben. Wenn der Server Antworten an Clients schicken soll, so benötigt jeder Client eine eigene Pipe für die Antwort. Eine Client schickt dann bei jeder Anfrage an der Server seine PID mit, so dass der Server anhand der PID die richtige Pipe für die Antwort identifizieren kann.

Um die passende Pipe zu finden wird meist eine Namenskonvention definiert z. B. :

<*path*>/<*name*>*Server.fifo* Dateiname für die Pipe zum Server
<*path*>/<*name*><*pid*>*.fifo* Dateiname für die Pipe zum Client

Der Client setzt dabei seine PID als eindeutige Kennung in den Dateinamen ein. Falls verschiedene Programme den gleichen Namen verwenden, kann es zu Kollisionen kommen.

Der Pfadname <*path*> ist i. Allg. Abhängig vom Betriebssystem. Um ein solches Programm portabel zu halten, muss er also einstellbar sein (z. B. durch eine Konfigurationsdatei).

Aufbau einer Named Pipe in Unix bzw. Linux Eine Named Pipe wird mit der Funktion `mkfifo()` im Dateisystem eingerichtet. Die Funktion `open()` ermöglicht schreibenden oder lesenden Zugriff auf die Pipe. Beide beteiligten Prozesse müssen dazu einen gemeinsamen Dateinamen (bzw. Datei-Pfad) besitzen. Dieser Pfad kann z. B. in einer gemeinsamen Header-Datei definiert sein. Ein Prozess erzeugt die Named Pipe, *danach* öffnet der eine Prozesse die

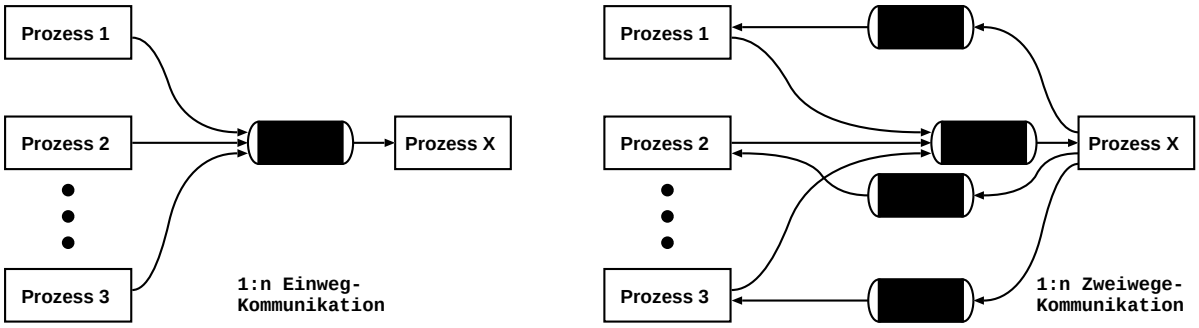


Figure 25: Aufbau einer 1:n Kommunikation mit Hilfe von Pipes

Pipe lesend, der andere öffnet sie schreibend. Weitere Prozesse können die Pipe öffnen und dadurch ebenfalls verwenden (s. Abb. 25). Eine Named Pipe bleibt im Dateisystem bestehen, bis sie explizit gelöscht wird (Funktion `unlink()`).

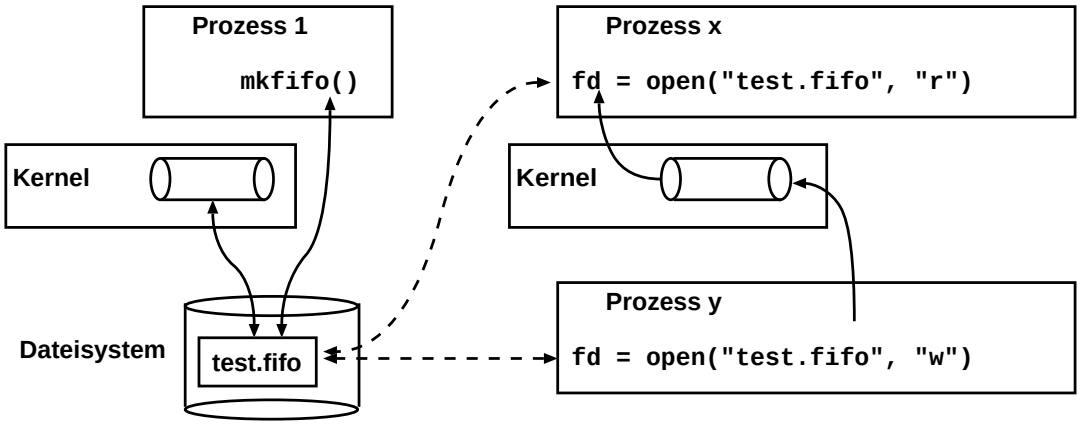


Figure 26: Aufbau einer 1:n Kommunikation mit Hilfe von Pipes

Die Funktion `open()` liefert einen Datei-Deskriptor zurück. Dieser kann durch die Funktion `fdopen()` in einen Stream umgewandelt werden (s. Abschnitt 3.4.1).

Zugriffsrechte Der Prozess, der die Named Pipe einrichtet, muss das Recht haben eine Datei mit dem verwendeten Pfad-Namen anzulegen. Der einrichtende Prozess vergibt dann seinerseits die Zugriffrechte für die Pipe-Datei. Die Rechte werden wie bei Dateien gehandhabt; d. h. es gibt Lese- und Schreib-Rechte für den Eigentümer, die Gruppe und den Rest der Welt. (Ausführbar ist eine Named Pipe nicht, insofern spielt das Recht zur Ausführung hier keine Rolle.) Andere Prozesse, die die Pipe verwenden wollen benötigen entsprechend Lese- und/oder Schreibrechte auf der Pipe-Datei.

3.4.3 STREAM-Pipes

STREAM-Pipes arbeiten im Duplex-Mode; d. h. man kann an beiden Enden der STREAM-Pipe sowohl lesen als auch schreiben. Ansonsten arbeiten sie im Wesentlichen wie die in Abschnitt 3.4.1 und Abschnitt 3.4.2 beschriebenen Pipes.

3.5 Nachrichten - Message Queues in Unix bzw. Linux

Prozesse können Message Queues einrichten, um Daten auszutauschen. Dafür sind zunächst folgende Funktionen notwendig.

- Erzeugen einer neuen Message Queue: `msgget()`
- Öffnen einer vorhandenen Message Queue: `msgget()`
- Senden einer Nachricht: `msgsnd()`
- Empfangen einer Nachricht `msgrcv()`
- Ändern der Parameter einer vorhandenen einer Message Queue: `msgctl()`
- Löschen einer Message Queue: `msgctl()`

Eine Message Queue ist eine lineare Liste von Nachrichten. An das eine Ende werden gesendete Nachrichten angehängt, vom anderen Ende werden empfangene Nachrichten entfernt (FIFO-Prinzip). In Unix können aber auch Nachrichten aus der Mitte der Liste empfangen und damit entfernt werden.

Eine Message Queue ist wie eine Named Pipe global im System sichtbar. Es können also auch nichtverwandte Prozesse auf ein und dieselbe Message Queue zugreifen. Dadurch können Namens-Kollisionen auftreten. Message Queues sind aber unabhängig vom Dateisystem, es sind keine Annahmen über existierende Pfade notwendig.

Eine Nachricht besteht aus zwei Teilen:

- Typ der Nachricht (message type)
- Nachrichten Text

Zusätzlich muss beim Versenden die Länge der Nachricht in Byte angegeben werden.

Im Unterschied zu Pipes ist für den Empfänger schon systemseitig vorgegeben, wo eine Nachricht beginnt und wo sie endet. Bei Pipes ist es Sache der Anwendung, einzelne Nachrichten zu trennen, falls dies notwendig ist. Der Inhalt einer Nachricht ist ein Byte-Array, das systemseitig nicht weiter strukturiert ist. Es ist Sache der Anwendung, den Inhalt zu interpretieren. Häufig verwenden beide Seiten eine gemeinsame Header-Datei, in der die Datenstruktur der Nachricht definiert ist. Die erste Komponenten muss immer der Nachrichtentyp sein (`long`), der Rest der Nachricht kann beliebig unterteilt sein.

Einrichten einer Message Queue Eine Message Queue wird durch die `msgget()` eingerichtet. Dabei muss ein Schlüsselwert als Bezeichner angegeben werden und ein Bit-Feld mit Optionen (Flag). Der Schlüsselwert kann auch vom System vergeben werden, dann muss jedoch der Erzeuger den Schlüsselwert an seine Kommunikationspartner weitergeben. Im Parameter Flag muss angegeben werden, dass eine neue Message Queue erzeugt werden soll.

Ändern und Löschen einer Message Queue Ein Message Queue kann mit der Funktion `msgctl()` gelöscht oder geändert werde. Die Funktion kann Parameter der Message Queue abfragen und ändern oder die Message Queue aus dem System löschen. Parameter sind u. a. die Größe der Queue und die Zugriffsrechte.

Verbinden mit einer Message Queue Eine Verbindung zu einer bestehenden Message Queue wird ebenfalls durch die `msgget()` hergestellt. Dabei muss als Schlüsselwert der Wert der gesuchten Message Queue eingesetzt werden. Im Parameter Flag muss angegeben werden, dass keine neue Message Queue erzeugt werden soll.

Senden einer Nachricht An eine Message Queue kann eine Nachricht gesendet werden `msgsnd()`. Die Nachricht wird an der Ende der Queue gehängt. Ist die Queue voll, so blockiert die Funktion oder sie liefert einen Fehlercode zurück, je nachdem, welche Flags gesetzt sind.

Lesen einer Nachricht Aus einer Message Queue kann eine Nachricht gelesen werden `msgrcv()`. Dabei wird normalerweise die erste Nachricht der Queue empfangen. Durch Angabe einer Kennung ("'typ"') kann auch auf andere Nachrichten zugegriffen werden. Wenn die Message Queue leer ist, blockiert die Funktion oder sie setzt ein Fehlerflag und kehrt zurück, je nachdem, ob das entsprechende Flag gesetzt ist. Beim Lesen muss eine maximale Länge angegeben werden, bis zu der gelesen wird. ist die Nachricht größer, kehrt die Funktion mit einer Fehlermeldung zurück oder die überzähligen Bytes werden abgeschnitten, je nachdem, ob das entsprechende Flag gesetzt ist.

Wenn mehrere Clients mit einem Server kommunizieren und der Server jedem Client einzeln antworten möchte, schreiben die Clients typischerweise ihr PID als Kennung in die Nachricht.

Implementierung eines Semaphors mittels Message-Queue Lösung zu Aufgabe 17 im Skript SYSO von Frau Keller.

```
1 public void p(SBox){
2    Buffer buffer;
3    receive(SBox, buffer);
4 }
```

```
1 public void v(SBox){
2    send(SBox, "dummy text");
3 }
```

Die Funktion `p()` blockiert in `S.pbin();` den aufrufenden Prozess, falls das Semaphor `sema` belegt ist. Die Funktion `v()` deblockiert mit `S.vbin();` einen anderen Prozess, falls das Semaphor `sema` belegt ist. Statt eines Semaphors wird eine Message-Queue erzeugt. Die Funktion `v()` stellt eine Nachricht in die Queue. Wenn die Queue voll ist, blockiert die Funktion `v()` so lange, bis die Funktion `p()` eine Nachricht aus der Queue gelesen hat. Die Funktion `p()` liest eine Nachricht aus der Queue. Wenn keine Nachricht in der Queue vorhanden ist, blockiert die Funktion `p()` solange, bis die Funktion `v()` eine Nachricht in die Queue geschrieben hat.

3.6 Shared Memory

Verschiedene Prozesse können sich einen Bereich des Hauptspeichers teilen. Jeder Prozess kann über Pointer auf den Speicherbereich zugreifen. Dies ist die schnellste Art der Interprozesskommunikation.

Die Kommunikation über Shared Memory kann als Produzenten-Verbraucher-Problem aufgefasst werden. Abb. 27 zeigt eine typische Konfiguration für eine unidirektionale Kommunikation, Abb. 28 für bidirektionale Kommunikation. Beide Shared Memory Segmente können z. B. als Ringspeicher betrieben werden.

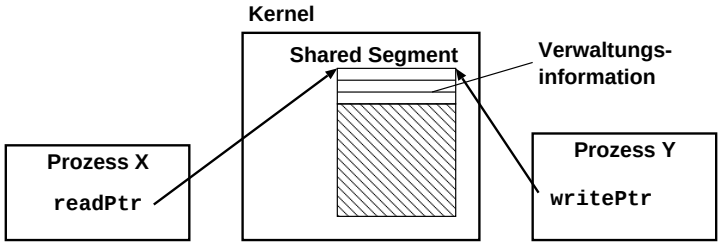


Figure 27: Unidirektionale Kommunikation über ein Shared Memory Segmente

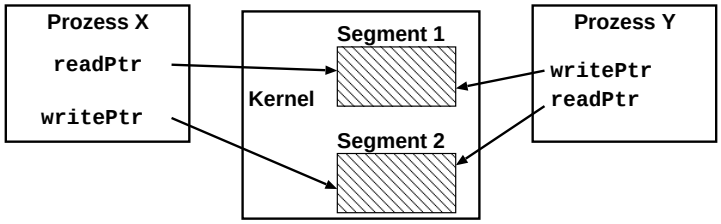


Figure 28: Bidirektionale Kommunikation über zwei Shared Memory Segmente

Für diese Art den Kommunikation ist es i. Allg. sinnvoll, die Struktur des gemeinsam genutzten Bereichs fest zu vereinbaren, z. B. über eine Header-Datei, die von alle beteiligten Prozessen importiert wird. Die Größe des Segments kann fest vereinbart werden oder an einer fest vereinbarten Stelle des Segments abgelegt werden.

Der Zugriff auf ein Shared Memory Segment muss in jedem Fall über ein Mutex synchronisiert werden. Die Synchronisierung kann aus dem Produzenten-Verbraucher-Problem übernommen werden.

Ein mögliche Lösung sieht folgendermaßen aus:

- ▷ Die aktuelle Schreib-Position und die aktuelle Lese-Position wird jeweils an einer fest vereinbarten Stelle des Segments abgelegt.
- ▷ Der Füllstand wird in zwei Semaphoren verwaltet (Anzahl freie Plätze und Anzahl belegte Plätze).

3.6.1 Shared Memory unter Linux

Ein Shared Memory Segment wird unter Linux ähnlich verwaltet wie eine Message Queue:

- Erzeugen eines neuen Shared Memory-Segment: `shmget()`
- Öffnen eines vorhandenen Shared Memory-Segment: `shmget()`
- Anbinden eines Shared Memory-Segment: `shmat()`
- Ablösen eines Shared Memory-Segment: `shmdt()`
- Ändern der Parameter eines vorhandenen Shared Memory-Segment: `shmctl()`
- Löschen eines Shared Memory-Segment: `shmctl()`

4 Ein- Ausgabe

Eine zentrale Aufgabe des Betriebssystem ist es, Daten zwischen den verschiedenen Teilen eines Rechners zu bewegen. Dies umfasst zum einen die Steuerung von Geräten, wie z. B. Festplatte, Tastatur, Netzwerkkarte. Auf der anderen Seite stellt das Betriebssystem den Anwendungsprogrammen Schnittstellen zur Verfügung, um auf die verschiedenen Geräte zuzugreifen.

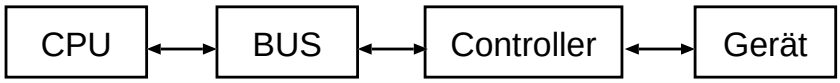


Figure 29: Schematische Darstellung der Koppelung von CPU und einem Ein-/Ausgabe-Gerät

Der Controller enthält i. Allg. einen Pufferspeicher. Dieser hat zwei Aufgaben:

- Der Pufferspeicher gleicht die Geschwindigkeitsdifferenz zwischen Gerät und Bus bzw. Hauptspeicher aus.
- Der Controller kann im Pufferspeicher eine Prüfsumme berechnen bzw. testen.

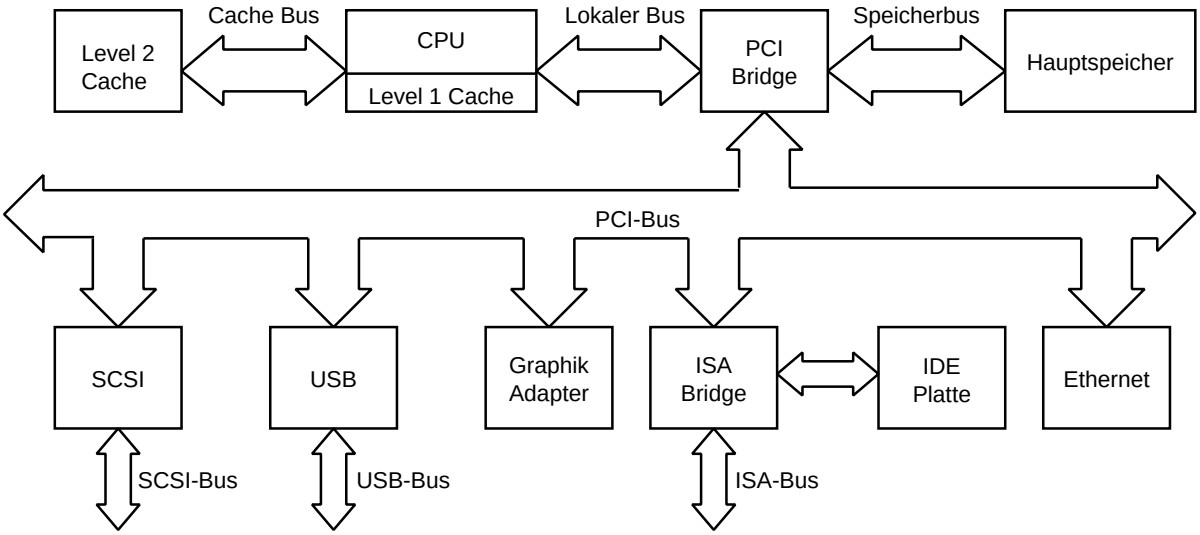


Figure 30: Schematische Darstellung der Koppelung von CPU und Ein-/Ausgabe-Geräten über unterschiedlich schnelle Bus-Systeme

Es werden i. Allg. zwei oder drei Kategorien von Geräten unterschieden: Blockorientierte Geräte und zeichenorientierte Geräte, unter Linux bilden Netzwerkkarten eine dritte Kategorie. Blockorienierte Geräte speichern Daten in Blöcken, die meist eine Größe von 512 Byte bis 32768 Byte besitzen. Jeder Block kann unabhängig von anderen Blöcken gelesen oder geschrieben werden. Typische blockorientierte Geräte sind Festplatten oder Memory-Sticks. Zeichenorientierte Geräte erzeugen dagegen eine Folge von Zeichen oder erwarten als Eingabe eine solche Folge. Eine Folge von Zeichen – ein Stream – hat keine Blockstruktur, die Daten können nicht direkt adressiert werden. Typische zeichenorientierte Geräte sind Maus

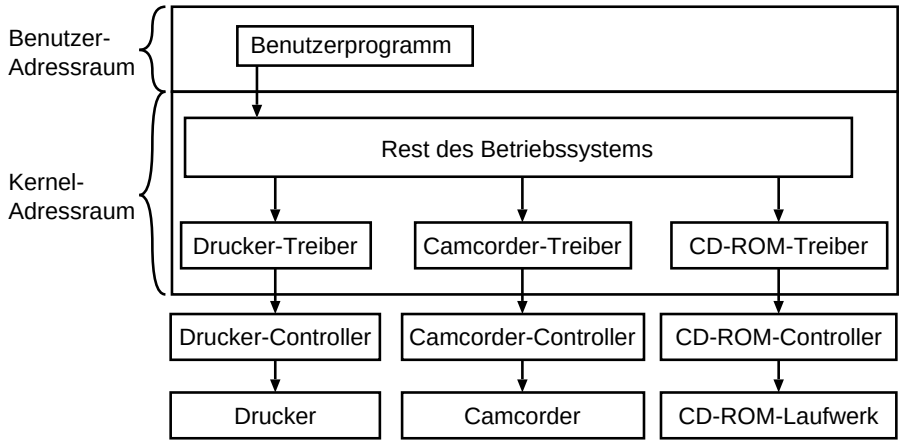


Figure 31: Schichtenmodell der Ein-/Ausgabe

und Tastatur. Netzwerkkarten können als zeichenorientierte Geräte betrieben werden, unter Linux werden sie aber mit einem eigenen Satz von Befehlen angesprochen.

Einige Geräte können nicht in dieses Schema eingeordnet werden, z. B. Uhr oder Graphik-Karte; andere Geräte können – zumindest theoretisch – sowohl zeichen- als auch blockorientiert betrieben werden, z. B. Magnetbandlaufwerke.

4.1 Adressierung von Geräten

Ein Gerät wird i. Allg. über einige wenige Register des Controllers angesprochen. Das Steuer-Register dient wie der Name sagt zur Steuerung; die Daten werden über ein anderes Register des Controllers übertragen. Es gibt im wesentlichen zwei Möglichkeiten, auf Geräte zuzugreifen: Mit speziellen Befehlen oder mit speziellen Adressen. Abb. 32 veranschaulicht die beiden Möglichkeiten sowie eine Mischform, wie sie bei Pentium-Systemen eingesetzt wird.

Verwendet der Prozessor spezielle Befehlen (z. B. IN, OUT) so gibt es z. B. folgende unterschiedliche Anweisungen:

- `IN R0, 4`
- `MOV R0, 4`

Der erste Befehl holt den Wert des I/O-Registers 4 in das Register 0 des Prozessors. Der zweite Befehl holt den Wert der Speicherstelle 4 in das Register 0 des Prozessors. Der Befehl `IN` erzeugt ein Signal auf einer zusätzlichen Adressleitung. Dies bewirkt, dass nicht der Hauptspeicher antwortet sondern das angesprochene Gerät. Es gibt in diesem Fall also einen zweiten Adressraum, der nur Geräte anspricht (s. Fall a) in Abb. 32). Dieser zweite Adressraum ist i. Allg. relativ klein, meist umfasst er 256 - 65536 Adressen; die einzelnen Adressen werden auch als Port-Nummern bezeichnet.

Verwendet der Prozessor spezielle Adressen, um Geräte zu adressieren, so spricht man von *memory mapped I/O*. In diesem Fall gibt es keine speziellen Befehle zu Ein-/Ausgabe, vielmehr ist ein Bereich des Adressraums für die Ein-/Ausgabe-Register reserviert (s. Fall b) in Abb. 32). Gibt der Prozessor eine Adresse auf den Adress-Bus, so prüft jedes Gerät, ob es angesprochen

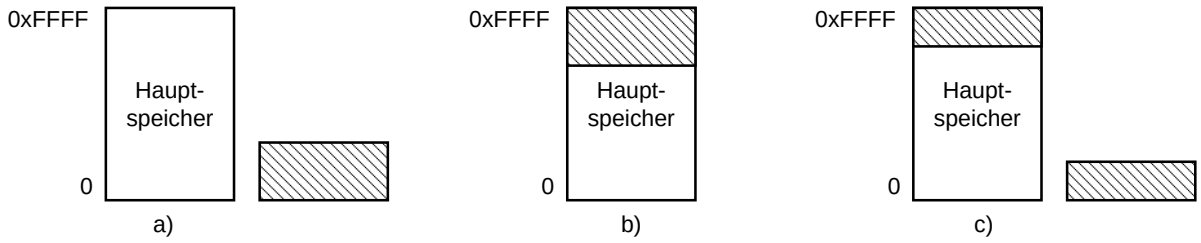


Figure 32: Adressräume für die Ein-/Ausgabe

ist. Bei Pentium-Systemen unterscheidet schon die PCI-Bridge, ob der Hauptspeicher angesprochen wird oder ob die Adresse auf den PCI-Bus gelegt werden muss. Dazu lädt die PCI-Bridge beim booten die Adressen der angeschlossenen Geräte. Der Hauptvorteil von memory mapped I/O besteht darin, dass die Geräte-Treiber in C realisiert werden können. Es sind keine speziellen Assembler-Befehle notwendig. Außerdem kann das Betriebssystem leicht verhindern, dass Anwendungsprogramme auf Geräte zugreifen, indem der Speicherbereich für die Ein-/Ausgabe-Register nicht in den Adressraum von Anwendungsprogrammen eingeblendet wird. Werden spezielle Befehle verwendet, muss es einen Schutzmechanismus geben, der verhindert, dass Anwendungsprogramme diese Befehle verwenden. Ein Nachteil von memory mapped I/O ist, dass das Bussystem komplexer wird, da die Auflösung der Adressen nun Aufgabe der Geräte bzw. Aufgabe eines Koppelbausteins (z. B. PCI-Bridge) ist. Ein weiteres Problem besteht darin, dass die Werte von I/O-Registern nicht im Cache gespeichert werden dürfen. Dies erfordert die Möglichkeit das Caching auszuschalten z. B. auf Seitenbasis. Auf der Ebene der Programmiersprache wird dies mit einem zusätzlichen Schlüsselwort (meist `volatile`) erreicht.

4.2 Gerätetreiber

Gerätetreiber sind Softwaremodule, die die Schnittstelle zwischen dem Betriebssystem-Kern und dem Geräte-Controller bilden. Gerätetreiber werden i. Allg. vom Betriebssystem bei Bedarf geladen. Es gibt sehr viele verschiedene Geräte mit jeweils eigenen Treibern. In einem Rechner sind aber meist nur wenige Geräte installiert. Es wäre daher unwirtschaftlich, alle vorhanden Treiber zu laden, obwohl nur einige wenige benötigt werden.

Die Gerätetreiber arbeiten bei den meisten Betriebssystem im Kernel Space (s. Abb. 31). Es gibt aber auch Betriebssystem (z. B. Minix, Hurd), die Gerätetreiber im User Space oder einem eigenen Adressraum betreiben (s. Abb. 33). Der zusätzliche Übergang vom User Space in den Kernel Space verlangsamt das System ein wenig. Gerätetreiber im Kernel Space können allerdings bei einem Fehler das gesamte System zum Stillstand bringen bis hin zu einer evtl. nötigen Neuinstallation. Außerdem kann ein Angreifer, der eine Sicherheitslücke in einem Gerätetreiber im Kernel Space ausnutzt, das System vollständig unter seine Kontrolle bringen. Ein Gerätetreiber im User Space kann nicht auf das gesamte System zugreifen. Ein solcher Gerätetreiber kann das System daher nicht bzw. nicht so leicht zum Absturz bringen oder kompromittieren. Gerätetreiber im User Space führen also tendenziell zu stabileren und sichereren Systemen; sie kosten aber etwas Performance.

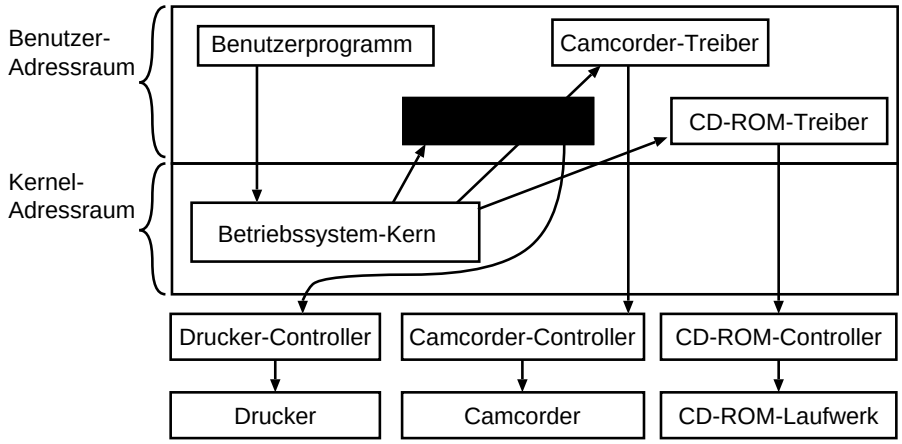


Figure 33: Gerätetreiber im User Space

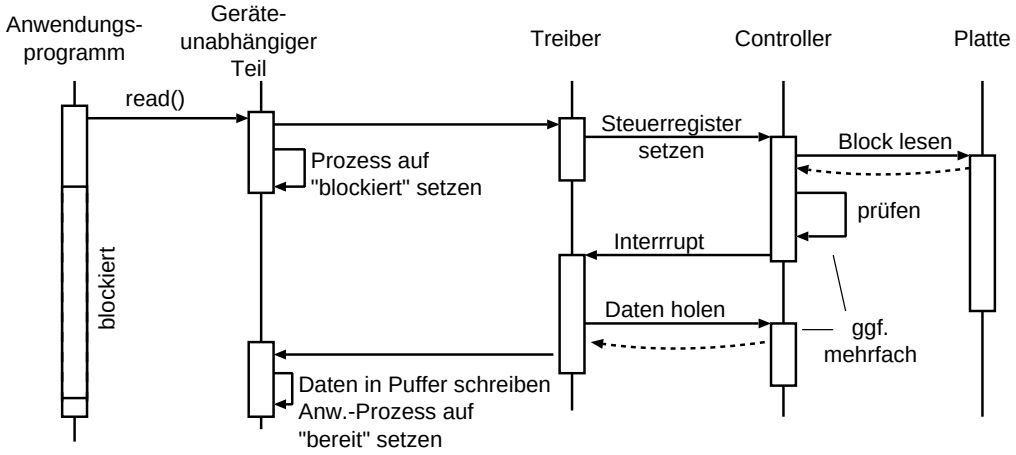


Figure 34: Lesen aus einer Datei

4.3 Direct Memory Access (DMA)

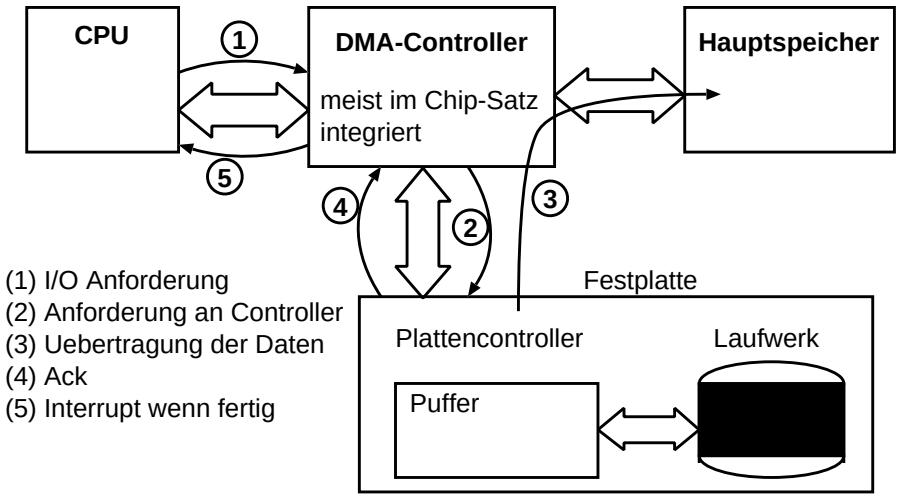


Figure 35: Direct Memory Access

Die CPU bzw. der Geräte-Treiber "'programmiert"' den DMA-Baustein. Daraufhin steuert der DMA-Baustein den Datentransfer zwischen Controller und Hauptspeicher. Während der Controller Daten in den Hauptspeicher überträgt, kann die CPU nicht auf den Hauptspeicher zugreifen. Da die CPU aber nach wie vor Zugriff auf den bzw. die Cache-Speicher hat, kann sie meistens dennoch weiterarbeiten.

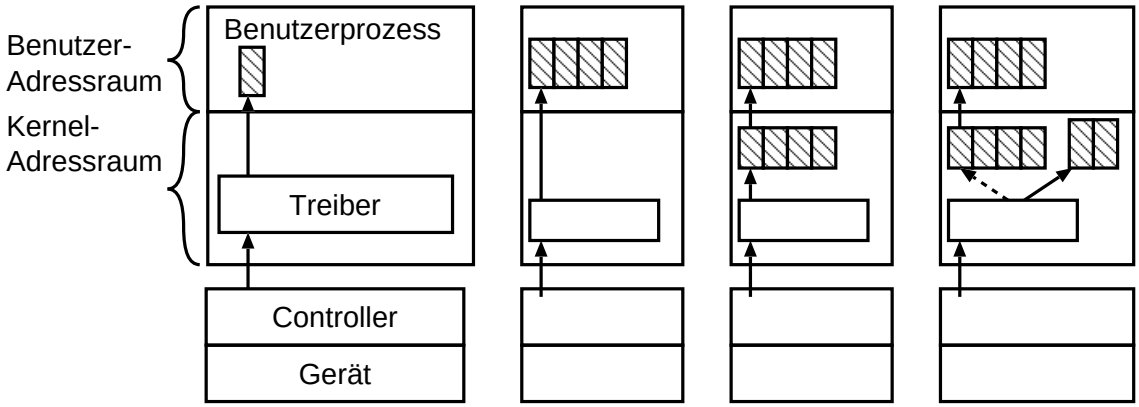


Figure 36: Pufferung der Eingabe

5 Dateisysteme

Desktop-Rechner und Server verfügen meist über eine oder mehrere Festplatten die Dateien enthalten; eingebettete Systeme verwenden oft Flash-ROM Speicher um Programme und Daten dauerhaft zu speichern. Die Dateien sind dazu in einem Dateisystem organisiert. Aufgabe des Dateisystems ist es, Daten persistent zu speichern, den Zugriff auf die Daten zu erleichtern und zu regulieren.

5.1 Datenträger

Eine Festplatte besteht aus einem "'Stapel"' von Platten mit magnetisierbarer Oberfläche, einem Satz von Schreib-Lese-Köpfen und einem Controller, der die Schreib-Lese-Köpfe steuert und die Schnittstelle zum Rechner bedient. Daten werden auf Festplatten stets in Blöcken geschrieben bzw. gelesen. Die Größe eines Datenblock beträgt meist 512 Byte. Dateisysteme arbeiten oft mit größeren Blöcken, in diesem Fall wird ein Block des Dateisystems auf mehrere Festplatten-Blöcke abgebildet.

Die meisten Festplatten-Controller enthalten einen Pufferspeicher (Cache) von ca. 2 bis 16 MB Größe (Stand 2007). Der Cache kann die Performanz eine Festplatte erhöhen; je nach Anwendung werden in der Literatur ca. 10 - 15 % genannt. Wird ein Block von der Platte gelesen, so wird er eine gewisse Zeit im Cache zwischengespeichert. Soll derselbe Block nochmals gelesen werden, so kann der Block aus dem Cache gelesen werden, was erheblich schneller ist, als auf die Platte zuzugreifen. Wird ein Block geschrieben, so speichert ihn der Controller zunächst im Cache und meldet dem Betriebssystem zurück, dass der Block geschrieben wurde. Anschließend schreibt der Controller den Block auf die Festplatte. Sind mehrere Blöcke im Cache für das Schreiben vorgemerkt, so kann der Controller die Schreibreihenfolge optimieren und auf diese Weise den Durchsatz erhöhen. Das Puffern von Schreiboperationen hat aber auch einen Nachteil: Bei einem Stromausfall gehen die Daten im Cache i. Allg. verloren (außer es handelt sich um nichtflüchtigen Speicher). D. h. es kann der Fall eintreten, dass das Betriebssystem bzw. eine Anwendung der Meinung ist, dass ein Datensatz sicher gespeichert ist, während er in Wirklichkeit nicht gespeichert wurde. Dieses Risiko ist nicht für alle Anwendungen tragbar. Daher bieten die meisten Festplatten-Controller die Möglichkeit, das Caching für Schreiboperationen auszuschalten. Der Controller meldet dem Betriebssystem das Ende der Schreiboperation erst dann, wenn die Daten wirklich auf der Platte gespeichert sind.

Ein Zugriff auf einen Block der Festplatte dauert im Mittel ca. 5–10 ms. Eine CPU kann in dieser Zeit ca. 10–20 Millionen Instruktionen ausführen. Werden mehrere Blöcke sequentiell gelesen oder geschrieben ist der Zugriff auf den zweiten und jeden weiteren Block wesentlich schneller. Maßgeblich ist dann die Transferrate, d. h. die Datenrate, mit der die Daten von der Platte gelesen bzw. auf die Platte geschrieben werden können. Die Transferrate von Festplatten liegt z. Zt. bei ca. 50–100 MB/s.

5.2 Partitionen

Eine Festplatte (oder ein anderes block-orientiertes Gerät) kann i. Allg. in mehrere Partitionen aufgeteilt werden. Jede Partition kann ein eigenes Dateisystem enthalten.

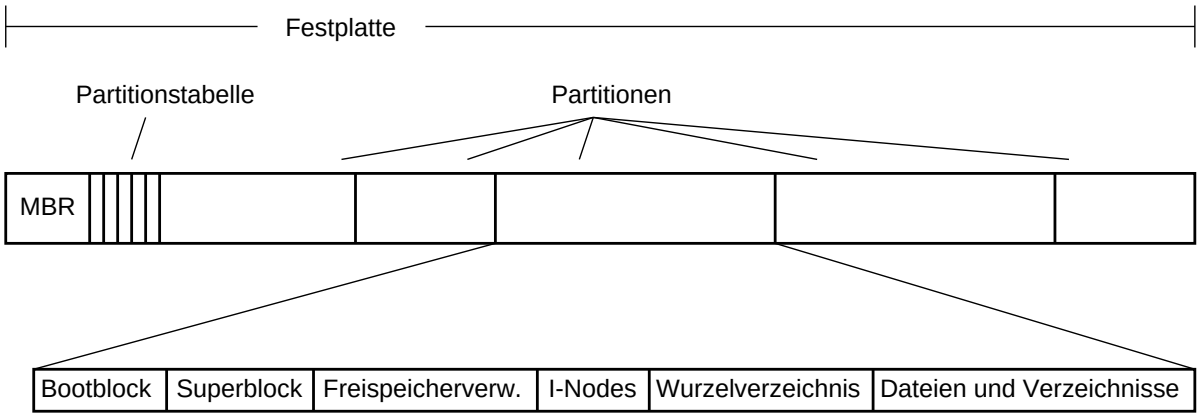


Figure 37: Beispiel für die Aufteilung einer Festplatte in Partitionen (teilweise Unix- bzw. Linux- spezifisch)

Für die X86-kompatiblen Rechner unterscheidet man dabei primäre und logische Partitionen. Eine Platte kann bis zu vier primäre Partitionen enthalten. Alternativ kann sie auch bis zu drei primäre Partitionen und eine sog. erweiterte Partition enthalten. Die erweitertete Partition kann beliebig viele logische Partitionen enthalten. Allerdings unterstützen Partitionierungsprogramme i. Allg. nur eine gewisse Anzahl von Partitionen z. B. 12 oder 24.

Der erste Sektor der Festplatte wird MBR (Master Boot Record) genannt (s. Abb. 37). Er enthält ausführbaren Code zum Starten des Rechners und die sogenannte Partitionstabelle. In dieser Tabelle sind die Anfangs- und Endadresse jeder primären Partition und ggf. der erweiterten Partition gespeichert. Außerdem ist in der Tabelle eine Partition als "'aktiv"' markiert.

Beim Starten des Rechners liest das BIOS zunächst den MBR und führt ihn aus. Das MBR-Programm lokalisiert die aktive Partition, liest den ersten Block – den Bootblock – dieser Partition und führt ihn aus. Das Programm im Bootblock übernimmt damit den weiteren Boot-Vorgang. Es kann nun unmittelbar ein Betriebssystem von dieser Partition starten oder einen Boot-Loader. Auch Partitionen, die kein Betriebssystem beherbergen, enthalten einen Bootblock; dieser enthält dann aber kein ausführbares Programm. Das weitere Layout einer Partition hängt stark vom Dateisystem der Partition ab. Das Layout aus Abb. 37 ist also nur ein Beispiel aus der Unix-Welt.

Ist die Partitionstabelle einer Platte ungültig oder zerstört, so ist es i. Allg. nur mit erheblichem Aufwand möglich, die Daten der Platte zu retten.

5.3 Dateien

Das Dateisystem ist eine Zwischenschicht, zwischen den Anwendungsprogrammen und einem Programm, das block-orientierte Geräte verwaltet. Dies kann ein Treiber-Programmen für eine Festplatte oder eine Speicherkarte sein, es kann aber auch ein Programm sein, das mehrere Gräte zusammen verwaltet (Logical Volume Manager (LVM) bzw. Storage Pool). Das Dateisystem bietet Anwendungsprogrammen Dateien und Operationen auf Dateien an. Es benötigt seinerseits die Möglichkeit, auf Blöcke eines Geräts bzw. eines Geräte-Pools zuzugreifen. Die Treiber- bzw. LVM-Programme bieten dem Dateisystem diese Funktionalität an. Dateisysteme speichern zu diesem Zweck Informationen über belegte und freie Blöcke sowie über Dateien

Table 4: Datei-Typen unter Linux

ID	Bedeutung	Berechtigungen	Beispiel
-	normale Datei	-rw-r–r–	/etc/fstab
d	Verzeichnis	drwxr-xr-x	/usr/bin
b	blockorientiertes Gerät	brw-rw—-	/dev/hda
c	zeichenorientiertes Gerät	crw-rw—-	/dev/audio
l	symbolischer Link	lrwxrwxrwx	/usr/lib/librpm.so -> librpm-4.4.so
p	namend Pipe	prw-rw—-	/home/martin/testPipe.fifo

in unterschiedlicher Form. Diese Informationen werden häufig an verschiedenen Stellen auf dem Speichermedium redundant gehalten, so dass die Informationen auch dann noch verfügbar sind, wenn ein Teil des Speichermediums nicht mehr lesbar ist. (Die Informationen über belegte und freie Blöcke ist an sich schon redundant. Ohne diese Information wäre die Suche nach einem freien Block aber sehr viel langsamer.)

Betriebssysteme unterscheiden i. Allg. verschiedene Datei-Typen:

- Reguläre Datei
- Verzeichnis/Directory (auch Katalog, Ordner oder Folder genannt)
- Verweis-Datei (Link)
- Spezial-Dateien (z. B. spezielle Zeichendatei, spezielle Blockdatei, Pipe-Datei)

Ausführbare Dateien müssen vom Lader des Betriebssystems als solche erkannt und von anderen Dateien unterschieden werden. Ausführbare Dateien müssen ein bestimmtes vom Betriebssystem vorgegebenes Format besitzen; wobei die meisten Betriebssysteme mehrere Formate unterstützen.

Verzeichnisse werden i. Allg. wir reguläre Dateien verwaltet. Ein Verzeichnis enthält zu jeder Datei, die durch diese Verzeichnis verwaltet wird, einen Eintrag. Der Eintrag besteht z. B. aus Name und Anfangsadresse der Datei. Unter Linux/Unix ist die Anfangsadresse ein Verweis auf einen I-Node, Näheres s. Abschnitt 5.6.1.

Verweis-Dateien enthalten einen Verweis auf eine andere Datei. Man kann dadurch mit verschiedenen Namen auf eine Datei zugreifen.

Spezial-Dateien dienen zum Zugriff auf Ein-/Ausgabegeräte oder andere vom Betriebssystem verwaltete Ressourcen. Sie werden hier nicht weiter betrachtet.

5.3.1 Strukturierung von Dateien

Für die meisten gebräuchlichen Betriebssysteme (Windows, Linux und andere) sind Dateien nur eine Folge von Bytes ohne weitere Struktur. Auf älteren Großrechnern besaßen Dateien oft eine vom Betriebssystem vorgegebene Record-Struktur, die sich an Lochkarten (80 Zeichen pro Zeile) oder an Zeilendruckern (132 Zeichen pro Zeile) orientierten. Einige Dateisysteme unterstützten auch die Suche nach einem Record, indem die Records in einer baumartigen Struktur abgelegt wurden.

Besitzen Dateien keine innere Struktur, so müssen Such-Strukturen (Indizes) von der Anwendung verwaltet werden. Dies ist eine Aufgabe die z. B. von Datenbank-Management-Systemen übernommen wird. Das Dateisystem unterstützt die DBMS-Systeme (oder eine andere Anwendung) nicht, es behindert sie aber auch nicht.

5.4 Zugriffs-Strukturen für Dateisysteme

5.4.1 Ein- und zweistufige Dateisysteme

Die ersten Dateisysteme besaßen nur ein Verzeichnis, in dem alle Dateien gespeichert waren. Der nächste Schritt der Entwicklung war eine feste Anzahl von Hierarchiestufen, z. B. zwei Stufen, so dass jeder Benutzer ein eigenes Verzeichnis erhält.

5.4.2 Hierarchische Dateisysteme

Die meisten heute gebräuchlichen Betriebssysteme (Windows, Linux und andere) verwenden ein hierarchisches Dateisystem, das beliebig viele Hierarchiestufen unterstützt.

5.4.3 Relationale Dateisysteme

Auch relationale Dateisysteme sind denkbar und teilweise schon realisiert Seit Ende des letzten Jahrtausends sind relationale Dateisysteme angekündigt und teilweise auch implementiert worden (s. [1], bzw. WinFS); sie werden aber bislang nur zu Forschungszwecken eingesetzt.

5.4.4 Namenserweiterungen

Die meisten Dateisysteme unterstützen Dateinamen, die aus zwei Komponenten bestehen, die durch einen Punkt getrennt sind. Der Namensteil nach dem Punkt – die Namenserweiterung – gibt einen Hinweis auf das Format der Datei bzw. auf die Art des Inhalts. Manche Betriebssysteme interpretieren diesen Hinweis streng und versuchen auf diese Weise zu erzwingen, dass eine Datei nur mit einem dafür geeigneten Programm geöffnet wird. Bei anderen Betriebssystemen dient die Namenserweiterung nur als Hinweis für den Benutzer. Außerdem ist es natürlich jedem Anwendungsprogramm überlassen, eine Datei mit einer bestimmte Namenserweiterung zu öffnen oder das Öffnen zu verweigern.

5.5 Implementierung von Zugriffsstrukturen

5.5.1 Zusammenhängende Belegung

Jede Datei ist in zusammenhängenden Blöcken abgelegt. Dieses Verfahren ist (nur) sinnvoll für Dateisysteme, die als Ganzes angelegt und dann nicht mehr verändert werden z. B. CDs/DVDs oder für Magnetbänder.

5.5.2 Verkettete Listen

Jeder Block einer Datei enthält die Adresse des nächsten Block oder eine Ende-Markierung. Befindet sich diese Datenstruktur nur auf der Festplatte, so ist der wahlfreie Zugriff auf Blöcke der Datei sehr langsam. Um den Zugriff zu beschleunigen kann ein File Allocation Table (FAT) eingesetzt werden.

Die FAT-Tabelle enthält einen Eintrag zu jedem Platten-Block. In Jedem Eintrag steht die Adresse des nächsten Blocks oder eine Ende-Markierung. Die Blöcke auf der Festplatte müssen dann keine Verkettungs-Pointer mehr enthalten. Die Tabelle selbst wird an einer fest vereinbarten Stelle der Partition gespeichert. Die Tabelle wird beim Booten in den Hauptspeicher geladen und kann dann wesentlich schneller durchsucht werden, als die entsprechende Datenstruktur auf der Festplatte. Die Tabelle wird i. Allg. mehr als einmal auf der Partition gespeichert, um das System robuster zu machen. In der Tabelle werden auch freie und defekten Blöcke protokolliert.

Der größte Nachteil dieser Lösung besteht darin, dass die FAT bei großen Platten/Partitionen viel Speicherplatz (Hauptspeicher) benötigt und/oder dass die Blockgröße erhöht werden muss, was wiederum zu mehr Verschnitt führt. Die FAT enthält einen Eintrag pro Block, sie wächst daher proportional zur Größe der verwalteten Partition. Ein Eintrag kann z. B. 2, 3 oder 4 Byte groß sein. Besitzt ein Eintrag 3 Byte, so kann die FAT 2^{24} d. h. 16 777 216 Blöcke verwalten. Bei einer Blockgröße von 512 Byte kann die FAT also 8 GB verwalten und belegt selbst $2^{24} * 3$ Byte, also 48 MB Speicherplatz. Die Fat32 Implementierung von Windows erhöht die Blockgröße auf bis zu 32KB (bei Partitionen ab 32 GB). Dadurch verringert sich die Größe der FAT (4 MB für eine 32 GB Partition), es erhöht sich aber der Verschnitt pro Datei auf durchschnittlich 16 KB.

Diese Art der Dateiverwaltung wird daher i. Allg. nur noch für kleinere Partitionen eingesetzt.

5.6 Baumbasierte Systeme

5.6.1 I-Nodes

Eine verbreitete Methode, um Dateien zu verwalten, sind sog. I-Nodes (Index-Knoten bzw. Informations-Knoden). Diese Methode realisiert einen nicht balancierten Baum. Jede Datei wird durch genau einen I-Node verwaltet. Ein I-Node hat eine feste Größe (z. B. 64 Byte oder 128 Byte unter Linux). Der I-Node enthält die Attribute der Datei außer den Namen und den Typ der Datei. Außerdem enthält er Verweise auf die Blöcke in denen die Daten der Datei
24. B. W0708 stehen. Über den I-Node können also alle Blöcke d. h. alle Daten einer Datei erreicht werden.

Das Betriebssystem muss nur die I-Nodes der geöffneten Dateien im Hauptspeicher halten. Ist die Anzahl der geöffneten Dateien auf einen festen Wert k beschränkt und ein I-Node n Byte groß, so belegt das Dateisystem nur $n * k$ Byte Hauptspeicher.

Die I-Nodes von Linux Ext2 enthalten folgende Datei-Attribute: Besitzer, Gruppe, Flags (Typ der Datei), Größe der Datei, die Anzahl der benutzten Blöcke, Zugriffszeitpunkt, Änderungszeitpunkt, Löschzeitpunkt, Anzahl der Links und einige mehr.

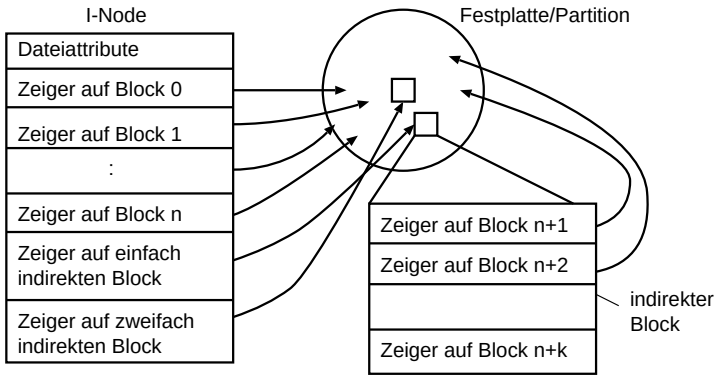


Figure 38: Aufbau eines I-Node mit nur einem indirekten Block

5.6.2 Verwaltung großer Dateien

Da die I-Nodes eine feste relativ kleine Größe besitzen, könnten zunächst nur relativ kleine Dateien damit verwaltet werden. Darum können I-Nodes nicht nur Zeiger auf Datenblöcke besitzen, sondern auch Zeiger auf einfach, zweifach und dreifach indirekte Blöcke. Damit entsteht der oben erwähnte nicht balancierten Baum.

Die I-Nodes von Linux Ext2 enthalten 12 Zeiger auf Datenblöcke und jeweils einen Zeiger auf einen einfach, zweifach bzw. dreifach indirekten Block. Wenn also die direkt adressierbaren Blöcke nicht ausreichen, so können weitere Blöcke über die indirekten Blöcke erreicht werden. Ein einfach indirekter Block zeigt auf einen Block, der Zeiger auf Datenblöcke enthält. Ein zweifach indirekter Block zeigt auf einen Block, der Zeiger auf einfach indirekte Blöcke enthält. Ein dreifach indirekter Block zeigt auf einen Block, der Zeiger auf zweifach indirekte Blöcke enthält. Abb. 39 zeigt den so entstehenden Baum.

Typische Größen sind: 4 Byte für einen Zeiger auf einen Block, 1 KB für einen Block. Mit 12 Zeiger auf Datenblöcke können also 12 KB adressiert werden. Mit einem indirekten Block können 1024/4 also 256 Blöcke adressiert werden; dies entspricht 256 KB. Mit einem Zeiger auf einen zweifach indirekten Block können $256 * 256$ Blöcke adressiert werden; dies entspricht 64 MB. Mit einem Zeiger auf einen dreifach indirekten Block können also 256^3 Blöcke adressiert werden; dies entspricht 16 GB. Insgesamt können mit einem solchen I-Node also $12 + 256 + 256^2 + 256^3$ Blöcke zu je 1 KB verwaltet werden. Verwendet man Blöcke zu 2 KB, kann eine Datei etwas mehr als 256 GB umfassen, bei einer Blockgröße von 4 KB liegt die Grenze bei knapp über 4 TB.

5.6.3 Freispeicherverwaltung

In einem Dateisystem befinden sich i. Allg. freie und genutzte Blöcke. Ebenso werden i. Allg. bei der Formatierung einer Partition eine gewisse Anzahl I-Nodes reserviert. Es gibt also eine feste Menge von Blöcken und eine feste Menge von I-Nodes, von denen jeweils ein Teil belegt und ein Teil frei ist. Die freien Elemente werden entweder durch eine Freispeicherliste oder durch eine Bitmap verwaltet (Abschnitt 2.5). Die Größe bzw. die Lage einer Partition kann i. Allg. nicht ohne weiteres verändert werden, wenn Programme auf Daten dieser Partition zugreifen – die Größe und die Lage einer Partition sind also im laufenden Betrieb fest. Daher

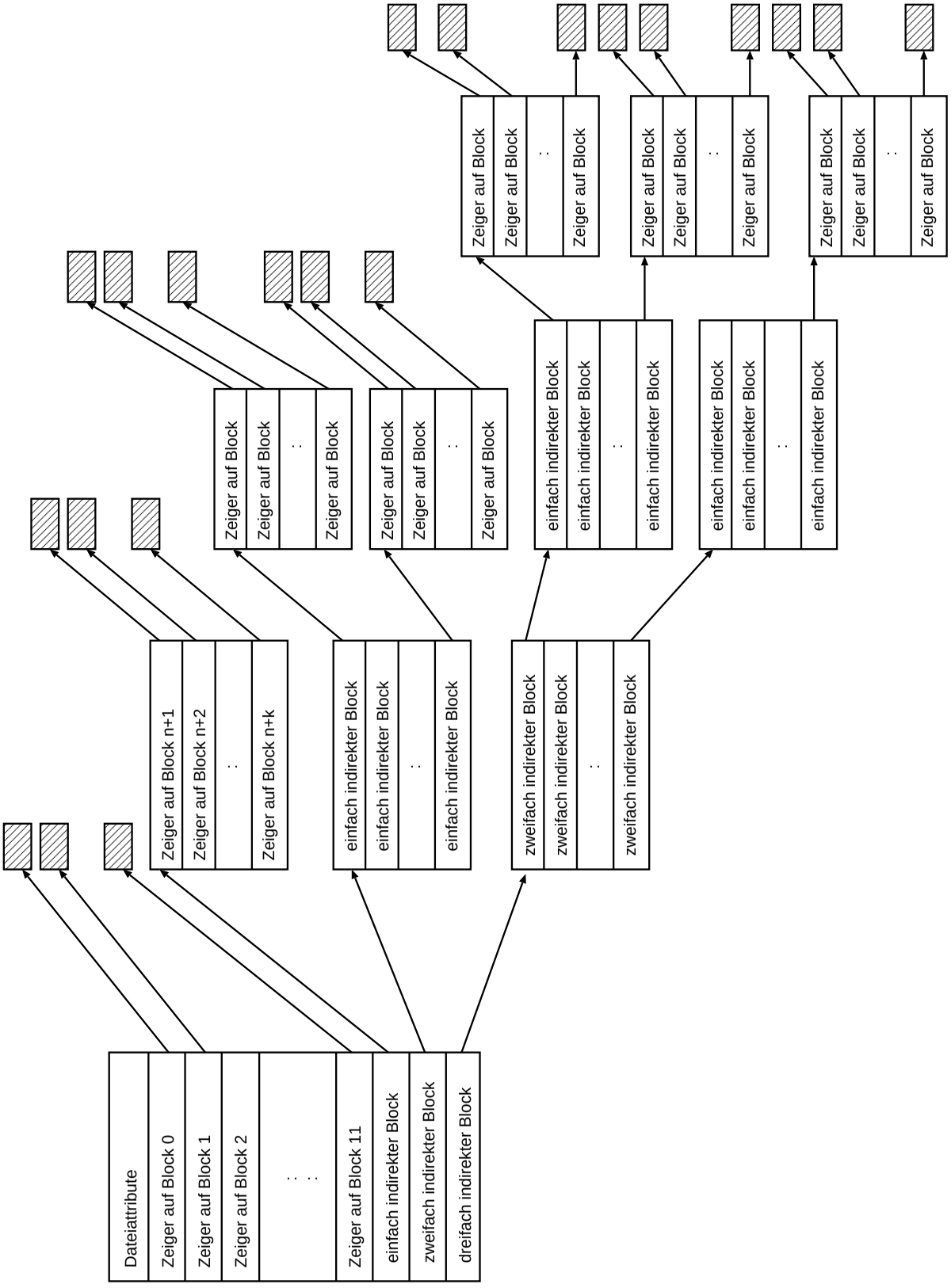


Figure 39: Aufbau einer Datei in Linux: I-Node, Blöcke mit Zeigern und Datenblöcke

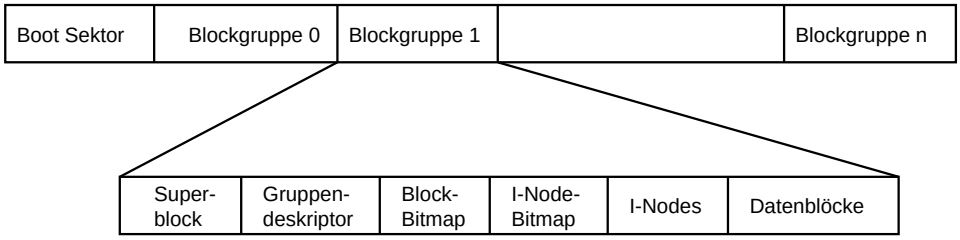


Figure 40: Aufbau einer Partition mit Ext2 Dateisystem von Linux

eignet sich für die Verwaltung auch die Bitmap-Technik. Ein zusätzlicher Vorteil der Bitmap-Technik besteht darin, dass nur die Bitmap (mit i. Allg. wenigen Zugriffen) gelesen werden muss, um einen leeren Block zu finden. Es muss keine Liste durchlaufen werden – eine Operation, die im Hauptspeicher relativ schnell geht, auf der Festplatte aber sehr zeitaufwändig sein kann.

Ein unter Linux weit verbreitetes Dateisystem ist das System Ext2 bzw. erweiterte Versionen Ext3 und Ext4. Abb. 40 zeigt den Aufbau einer solchen Partition. Die Partition wird in Blockgruppen zerlegt, jede Blockgruppe erhält eine Kopie des Superblocks und einen Deskriptor. Der Superblock enthält u. a. die Blockgröße sowie die Anzahl der Blöcke und der I-Nodes für das gesamte Dateisystem. Der Deskriptor enthält Informationen zur Blockgruppe; u. a. die Positionen der Bitmaps für die Freispeicherverwaltung, die Anzahl der freien Blöcke und der freien I-Nodes sowie die Anzahl der Verzeichnisse. Letztere Information verwendet Ext2 dafür, die Verzeichnisse möglichst gleichmäßig über die Blockgruppen zu verteilen. Eine Datei kann sich über mehr als eine Blockgruppe erstrecken. Das Dateisystem versucht jedoch zunächst jede Datei innerhalb einer Blockgruppe zu speichern.

5.6.4 Verzeichnisse

Ein Verzeichnis in Ext2ist eine Datei, die eine Liste mit Dateinamen und der Nummer des I-Nodes der jeweiligen Datei enthält. Abb. 41 zeigt einen Ausschnitt aus einer Verzeichnisstruktur aus Sicht des Nutzers. Abb. 42 zeigt diesen Ausschnitt etwas vereinfacht aus Sicht des Dateisystems d. h. mit I-Nodes und Blöcken.Die I-Nodes stehen in einer Tabelle s. Abb. 40, so dass jeder I-Node mit einem Index – der I-Node-Nummer – adressiert werden kann. Die Abbildung zeigt eine reguläre Datei: `/home/todo.txt`. Bei den anderen Dateien handelt es sich um Verzeichnisse.

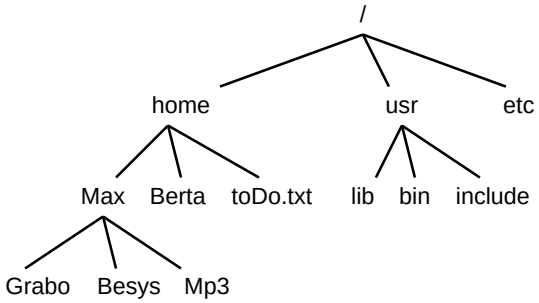


Figure 41: Sicht des Nutzers auf das Root-Directory (Ausschnitt)

Neuere Dateisysteme erlauben i. Allg. relativ lange Dateinamen. Sie speichern die Dateinamen daher nicht in einer einfachen Tabelle sondern in einer etwas aufwendigeren Struktur. Das System Ext2 erlaubt z. B. – wie andere moderne Dateisysteme auch – Dateinamen von z. Zt. bis 255 Byte Länge. Da die meisten Dateinamen deutlich kürzer sind, würde eine einfache Tabelle, wie in Abb. 42 dargestellt, viel Speicherplatz verschwenden.

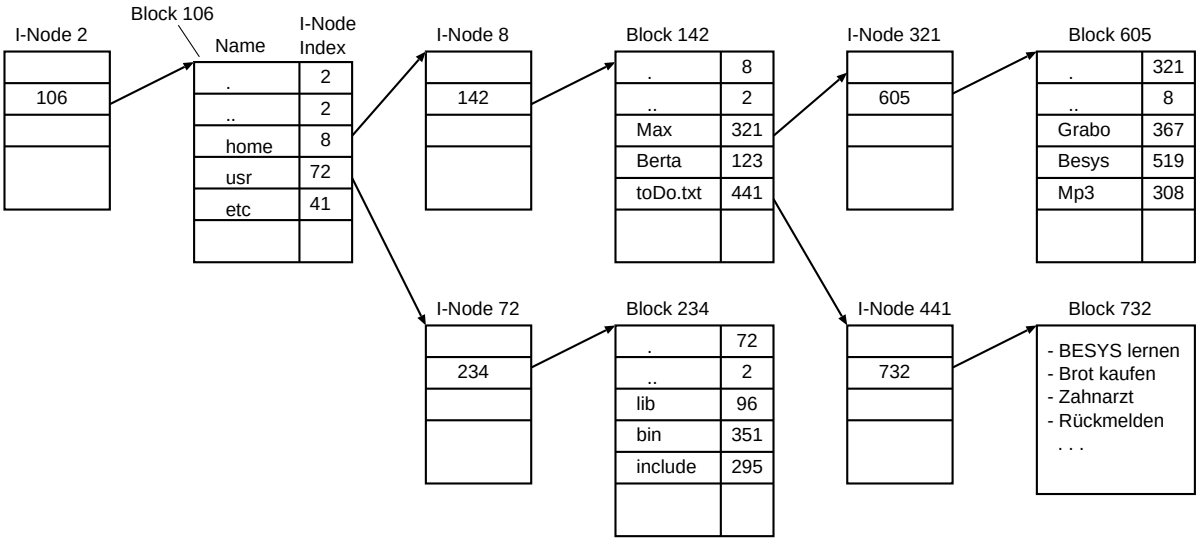


Figure 42: Verzeichnisse mit I-Nodes (vereinfacht)

Daher wird ein Verzeichnis in Ext2 nicht als Tabelle mit Einträgen fester Länge realisiert sondern als eine verkettete Liste. Ein Eintrag der Liste enthält die I-Node-Nummer, die Länge des Eintrags, die Länge des Namens und den Dateinamen s. Abb. 43. Die Länge des Eintrags erlaubt es, den nächsten Eintrag im Verzeichnis zu lokalisieren. Ein so organisiertes Directory muss also stets von Anfang an durchlaufen werden.

Einige Dateisysteme verwenden aufwendigere Suchstrukturen, um Verzeichnisse mit vielen Einträgen zu organisieren z. B. erweiterbares Hashing.

Die I-Node-Nummer des Wurzelverzeichnis steht an einer fest vereinbarten Stelle im Superblock (in Ext2 ist die I-Node-Nummer des Wurzelverzeichnis z. B. stets 2).

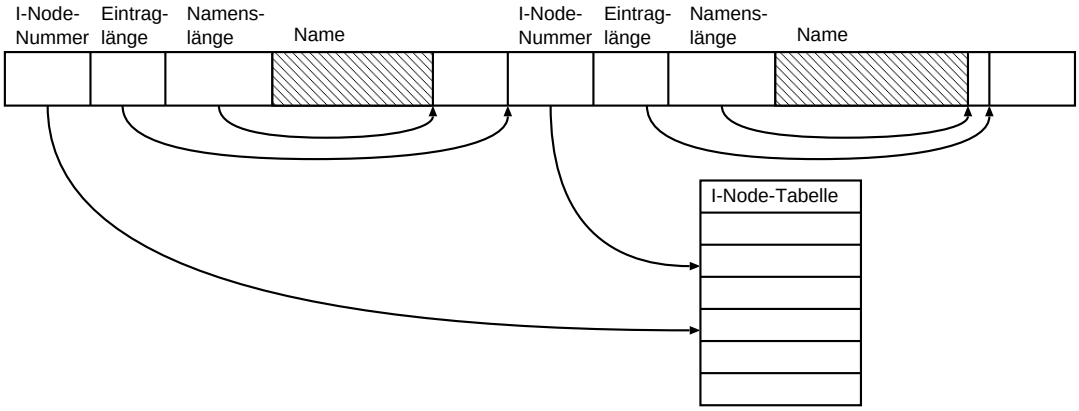


Figure 43: Liste von Einträgen in einem Verzeichnis für ein Dateisystem mit I-Nodes und Dateinamen variabler Länge.

Anmerkung: Bei älteren Dateisystemen war die Länge eines Namens oft auf einen relativ kleinen Wert begrenzt. Ein Eintrag hatte dann eine feste Länge, so dass keine Verkettung notwendig war sondern eine einfache Tabelle, wie in Abb. 42 dargestellt, ausreichend war.

5.6.5 Verweise / Links

Oft werden Dateien an verschiedenen Stellen im Dateisystem benötigt, d. h. eine Datei soll über verschiedene Pfade bzw. Namen zugänglich sein. Dateisysteme bieten für diese Anforderung Verweise bzw. Links an. Man unterscheidet dabei i. Allg. zwischen Soft-Link und Hard-Link.

Ein Hard-Link ist ein zusätzlicher Eintrag in einem Verzeichnis, der auf einen I-Node verweist. Dieser Eintrag unterscheidet sich nicht von anderen Einträgen.

Ein Soft-Link ist eine zusätzlich Datei, die als "'Soft-Link"' gekennzeichnet ist, und den Pfad zur Zieldatei als Datensatz enthält.

5.7 Konsistenz in Dateisystemen

Bei den meisten Dateisystemen können die Daten des Dateisystems in einen inkonsistenten Zustand kommen. Mögliche Ursachen sind u. a. Fehler der Dateisystemsoftware, ein Absturz des Rechners bzw. des Betriebssystem oder ein Fehler im Speichermedium. Die meisten Dateisysteme versuchen, "'wichtige Daten"' – d. h. Daten, die für die Konsistenz des Systems wichtig sind – möglichst schnell auf das Speichermedium zu schreiben. Dies vermindert die Gefahr von Inkonsistenzen durch Absturz des Rechners, es schließt diese Gefahr aber nicht aus. Das Dateisystemen ZFS versucht im Gegensatz dazu durch copy on write (s. u.) sicherzustellen, dass die Daten auf der Festplatte stets in einem konsistenten Zustand sind.

Konsistenzprüfung: Die meisten Dateisysteme enthalten ein oder mehrere Prüfprogramm(e), um Inkonsistenzen zu erkennen und ggf. teilweise automatisch zu reparieren. Unter Linux gibt es z. B. fsck, unter Windows scandisk sowie jeweils noch weitere Prüfprogramme. Geprüft wird die Konsistenz der Blockverwaltung und die Konsistenz der Dateiverwaltung.

Journaling: Neuere Dateisysteme führen Buch über geplante und abgeschlossene Schreibvorgänge (Journal). Beim Hochfahren des Dateisystems müssen dann nur die Teile des Dateisystems geprüft werden, bei denen Schreibvorgänge nicht abgeschlossen werden konnten.

Prüfsummen: Die Konsistenz von Daten kann über Prüfsummen getestet werden. Dabei wird aus einem Datensatz ein Hash-Wert errechnet und abgespeichert. Wird der Datensatz gelesen, so wird der Hash-Wert wieder errechnet und mit dem abgespeicherten Wert verglichen.

Das Dateisystem ZFS (SUN micro systems) verwendet für alle Daten Prüfsummen. Die Prüfsumme eines Blocks wird jeweils im übergeordneten Block gespeichert. Die Prüfsummen werden im laufenden Betrieb kontinuierlich geprüft, so dass Fehler erkannt und unmittelbar repariert oder berichtet werden können.

Copy On Write: Alle Schreibvorgänge, die nicht atomar sind, werden erst als Kopie ausgeführt. Das System definiert dann in einer atomaren Aktion die neu geschriebenen Werte als gültig. Dadurch ist das Schreiben transaktional, d. h. die Daten auf dem Speichermedium sind stets in einem konsistenten Zustand (implementiert in ZFS).

5.7.1 Prüfung der Blockverwaltung

26. B. W0809

Ein Block ist entweder frei oder genau einer Datei zugeordnet. Das Prüfprogramm legt zwei Tabellen an, jede Tabelle enthält einen Zähler mit Anfangswert 0 für jeden Block. Der Zähler in der ersten Tabelle verfolgt, wie oft ein Block in einer Datei vorkommt. Der Zähler in der zweiten Tabelle verfolgt, wie oft ein Block in der Freispeicherverwaltung vorkommt. Das Programm liest nun alle I-Nodes ein und ermittelt zu jedem I-Node die von dort erreichbaren Blöcke. Jedesmal, wenn das Programm einen Block erreicht, erhöht es den entsprechenden Zähler in der ersten Tabelle. Das Programm durchläuft nun die Freiliste bzw. die Freispeicher-Bitmap. Jedesmal, wenn das Programm einen Block als frei erkennt, erhöht es den entsprechenden Zähler in der zweiten Tabelle.

Danach durchläuft das Programm die beiden Tabellen. Ist das Dateisystem konsistent, so ist jeder Block mit einer 1 in der einen und mit einer 0 in der anderen Tabelle vertreten. Ein Paar (1, 0) heißt dann, der Block ist Teil einer Datei; Ein Paar (0, 1) heißt, der Block ist frei.

Man kann nun folgende Fehlerfälle unterscheiden:

a)	(0, 0)	Der Block ist "'vermisst"'
b)	(1, 1)	Der Block ist benutzt und als frei markiert
c)	$(0, n > 1)$	Der Block ist mehr als einmal als frei markiert
d)	$(n > 1, X)$	Der Block wird von mehr als einer Dateien benutzt und ist evtl. als frei markiert

In den Fällen c) und d) können ggf. auch größere Zahlen auftauchen.

Fall a) ist leicht zu reparieren, indem der Block in die Freispeicherverwaltung aufgenommen wird. Im Fall b) wird der Block aus der Freispeicherverwaltung herausgenommen. Fall c) kann bei Verwendung einer Bitmap als Freispeicherverwaltung nicht auftreten, wohl aber bei einer verketteten Liste. Der Block wird genau einmal in die Freispeicherverwaltung eingetragen. Im Fall d) kann das Programm folgendes tun: Der Inhalt des Blocks wird in einen freien Block kopiert und dieser dann in eine der beteiligten Dateien eingegliedert. Ggf. muss der Block noch als in der Freispeicherverwaltung als belegt gekennzeichnet werden. Das System ist damit wieder konsistent, allerdings ist es sehr wahrscheinlich, dass zumindest eine der beteiligten Dateien durcheinander geraten ist.

5.7.2 Prüfung der Dateiverwaltung

Das Programm prüft u. a., ob der Link-Count eines I-Nodes übereinstimmt mit der Anzahl der Verweise aus Verzeichnissen auf diesen I-Node. Außerdem darf ein I-Node, auf den ein Verzeichnis zeigt, nicht als frei markiert sein.

Das Programm durchläuft rekursiv alle Verzeichnisse das Dateisystems. Es legt für jede gefundene Datei einen Eintrag mit einem Zähler an. Dabei bestimmt nicht der Name der Datei sondern die I-Node-Nummer die Identität. Ist für eine gefundene Datei bereits ein Eintrag vorhanden, so wird der der Zähler um eins erhöht. Dadurch entsteht eine Liste, in der zu jeder Datei – d. h. zu jedem I-Node – vermerkt ist, wie viele Verzeichniseinträge auf diesen I-Node verweisen. Das Programm vergleicht nun den Link-Count der I-Nodes mit dem jeweiligen Eintrag in der Liste. Stimmen die beiden Werte nicht überein, so wird der Link-Count im I-Node angepasst.

Unter Unix bzw. Linux werden Dateien, auf die kein Verzeichniseintrag verweist, in ein fest vereinbarten Verzeichnis eingetragen (`/lost+found`). Da kein Name für diese Dateien bekannt ist, werden sie nummeriert.

Zusätzlich kann das Programm Einstellungen auf Plausibilität prüfen und z. B. ungewöhnliche Berechtigungen melden; etwa wenn der Besitzer einer Datei weniger Rechte hat als andere Benutzer.

5.8 Öffnen und Schließen einer Datei in Linux

Wird eine Datei geöffnet, so legt der Kernel für die Datei eine Datenstruktur an. Ebenso erhält der öffnende Prozess eine Datenstruktur für die Datei s. Abb. 44.

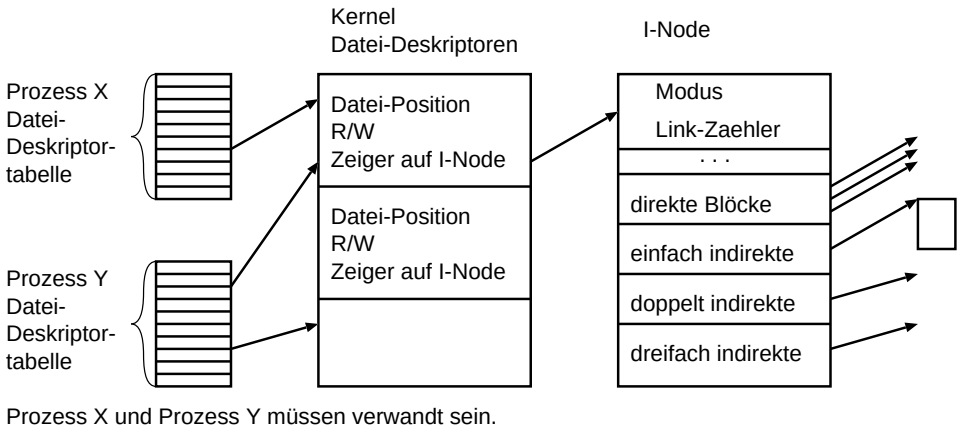


Figure 44: Aufbau der Datenstrukturen zum Dateizugriff in Linux/Unix

5.9 Das Netzwerk-Dateisystem NFS

Die meisten modernen Betriebssysteme erlauben es, von einem Rechner aus auf Dateien auf der Festplatte eines anderen Rechners zuzugreifen – sofern der andere Rechner dem zustimmt.

Linux verwendet eine zusätzliche Abstraktionsschicht, um auf verschiedene Dateisysteme zuzugreifen, das "'Virtual File System"' (VFS). Diese Architektur erlaubt es, auch entfernte Dateisysteme einzubinden. Abb. 45 zeigt die Einbindung verschiedener Dateisysteme. Die Schnittstelle, mit der auf das VFS zugegriffen wird orientiert sich stark an der Schnittstelle von Ext2. Andere Dateisysteme müssen ggf. das I-Node-Konzept simulieren. Wird eine Datei geöffnet, so legt die VFS-Schicht einen V-Node für diese Datei (im Hauptspeicher) an. Der V-Node verweist entweder auf einen I-Node (lokale Datei) oder auf einen R-Node im NFS-Client, falls es sich um eine entfernte Datei handelt.

Bemerkenswert ist dabei insbesondere, dass – bis zur Version 3 – eine entfernte Datei nur auf der Client-Seite geöffnet wird. Der NFS-Server bearbeitet nur einzelne Anfragen, er ist also zustandslos. Der Client erhält vom Server die Geräte-Nummer und die I-Node-Nummer. Der Client hält sich einen Dateideskriptor mit Positionszeiger.

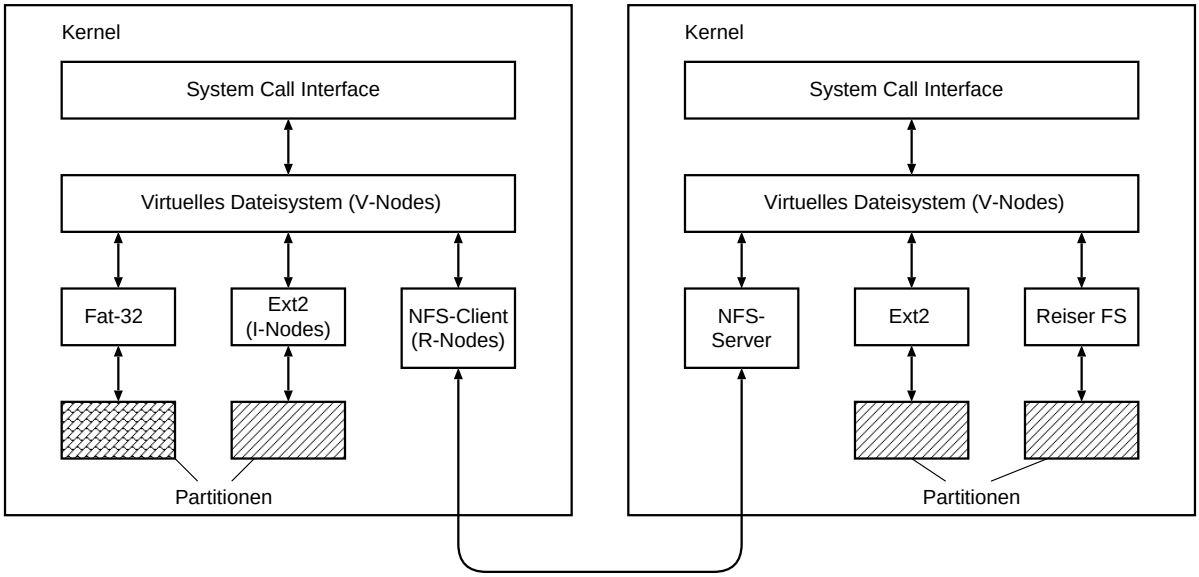


Figure 45: Integration verschiedener Dateisysteme unter Linux

Seit Version 4 wird eine Datei bei einem Zugriff über NFS auf dem Server geöffnet und wieder geschlossen.

References

[1] GIAMPAOLO, DOMINIC: *Practical File System Design with the Be File System*. Morgan Kaufmann, 1998. ISBN-13: 978-1558604971.

[2] HORNSTEIN, PETRA: *PetriEdiSim – Tutorial und Editor für Petri-Netze*. http://olli.informatik.uni-oldenburg.de/PetriEdiSim/index.html, 2001, gespeichert 2005.

[3] REISIG, W.: *Petri Nets, An Introduction*. EATCS Monographs on Theoret. Comput. Sci. Springer-Verlag, 1985.

[4] SEDGEWICK, ROBERT: *Algorithms in Java. Parts 1-4 / Part 5*. Addison-Wesley, Amsterdam, 3rd edition, 2003.

[5] STEVENS, W. R.: *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2005. ISBN: 0201433079 (70 Euro).

[6] TANENBAUM, ANDREW S.: *Moderne Betriebssysteme*. Pearson Studium, München, 2003. ISBN: 3-8273-7019-1 (50 Euro).

[7] WOLF, JÜRGEN: *Linux-UNIX-Programmierung*. Galileo Press, 2005. ISBN 3-89842-570-3, 50 Euro, auch online: http://www.pronix.de/pronix-6.html.

A Algorithmen

Die folgenden Beschreibungen der Algorithmen sind nur als Überblick gedacht. In der Literatur z. B. [4] finden sich wesentlich genauere Beschreibungen zu Hashing. Unter dem Namen "'Extendible Hashing"' werden in der Literatur auch abgewandelte Algorithmen beschrieben.

A.1 Extendible Hashing

Dieser Algorithmus kombiniert Hashing mit einem Digital-Baum (auch "'trie"' genannt). Objekte werden in einem Digital-Baum gespeichert. Der Schlüssel errechnet sich aus einer Hash-Funktion, die auf einen Wert bzw. Schlüssel des jeweiligen Objekts angewendet wird.

Bsp.: Ein Objekt besteht aus drei Komponenten: String Name, String Datum und Integer Größe. Objekte sollen schnell anhand des Namens gefunden werden.

Ein Knoten in einem Digitalbaum besitzt n Kinder. Diese Anzahl ist i. Allg. für alle Knoten gleich und während der Lebensdauer des Baums fest. Die Zeiger auf die Kinder sind in einem Array der Länge n gespeichert.

Der Schlüssel wird beim Speichern und beim Suchen eines Elements in Komponenten zerlegt, so dass jede Komponente den Wertebereich $1 - n$ bzw. $0 - (n - 1)$ hat.

In der Wurzel des Digitalbaums wählt man den Kindknoten, indem an die erste Komponente des Schlüssels als Index für das Array der Kindknoten verwendet. In diesem Kindknoten verwendet man die zweite Komponente des Schlüssels. In einem Knoten der Höhe k verwendet man also die k-te Komponente des Schlüssls um den Kindknoten auszuwählen.

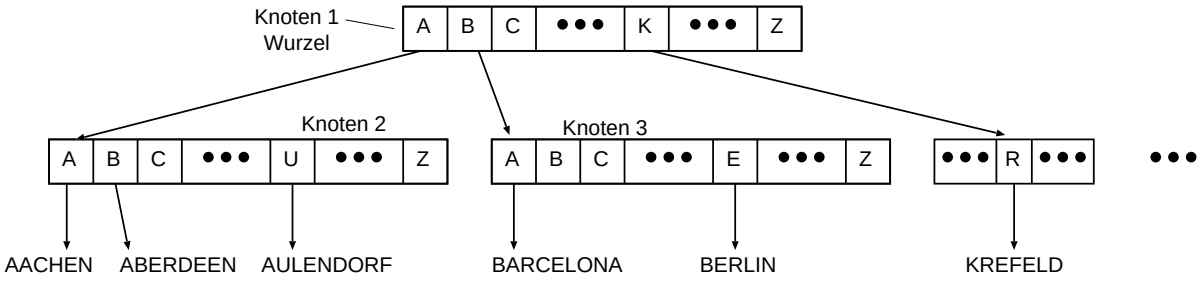


Figure 46: Beispiel für einen Digitalbaum

Abb. 46 zeigt ein Beispiel für einen Digitalbaum. Die Schlüssel sind Strings, eine Schlüsselkomponenten ist ein (Groß-)Buchstabe. Das Array der Kindknoten muss nun 26 Zeiger aufnehmen können (für jeden Großbuchstaben einen). Um auf das Array der Kindknoten zuzugreifen, nimmt man den n-ten Buchstaben des Schlussels und zieht vom ASCII-Code des Buchstabens noch den ASCII-Code von 'A' ab. Dieser Wert dient als Index für den Zugriff in das Array der Kindknoten.

Eine Kollision kann man durch eine verkettete Liste auflösen oder zunächst dadurch, dass der Baum nach unten wächst. Wird in den Baum in Abb. 46 ein zusätzliches Objekt eingefügt, dessen Schlüssel mit einem bereits vorhanden Schlüssel kollidiert, so wächst der Baum nach unten (s. Abb. 47). Da i. Allg. auf diese Weise nicht alle Kollision aufgelöst werden können, muss in Blattknoten stets ein Mechanismus zur Auflösung von Kollision realisiert werden.

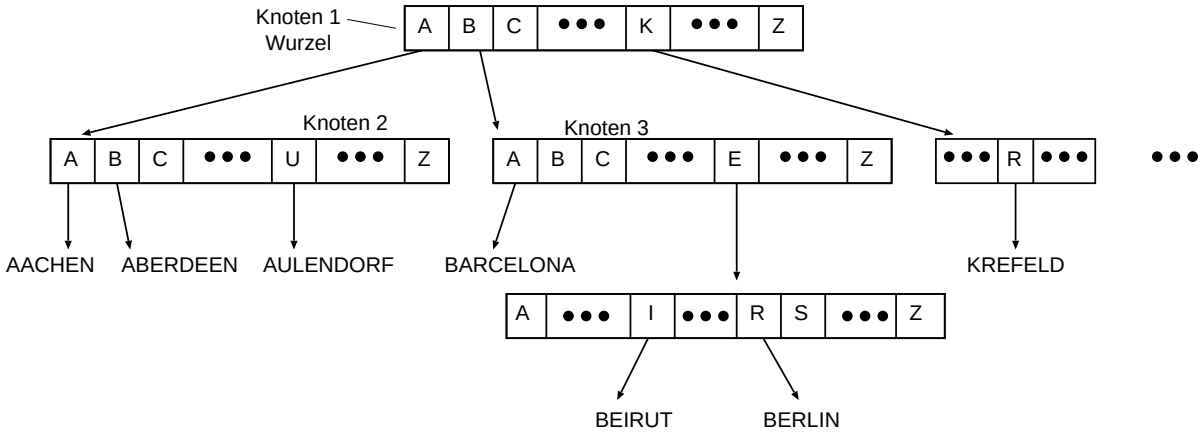


Figure 47: Ein zusätzlicher Knoten löst die Kollision auf

Extendible Hashing verwendet also einen Digitalbaum, um Objekte zu speichern. Der Suchschlüssel wird aber aus dem Objekt-Schlüssel durch eine Hash-Funktion gewonnen. Dadurch ist der Baum i. Allg. gut balanciert. Im obigen Beispiel wird also beim Speichern und bei der Suche zunächst der Hashwert des Strings gebildet. Dieser Hashwert kann nun Stückweise als Index verwedet werden. Üblich sind z. B. je 4 Bit oder je 8 Bit. Die Arrays in den Knoten enthalten dann 16 bzw. 256 Einträge.

B Java-Implementierung des Produzenten-Verbraucher-Systems

B.1 Die Main-Methode

```
1 package besysMonitor;
2 public class ProducerConsumer {
3     public static void main(String[] args) {
4         int initialFill = 5;
5         int prodDelay = 400;
6         int consDelay = 400;
7         if (args.length > 0){
8             initialFill = Integer.parseInt(args[0]);
9         }
10        if (args.length > 1){
11            prodDelay = Integer.parseInt(args[1]);
12        }
13        if (args.length > 2){
14            consDelay = Integer.parseInt(args[2]);
15        }
16        ProdConsBuffer buffer = new ProdConsBuffer(initialFill);
17        Producer prod1 = new Producer(buffer, prodDelay);
18        Consumer cons1 = new Consumer(buffer, consDelay);
19        prod1.start();
20        cons1.start();
21        return;
22    }
23 }
```

B.2 Der Puffer-Speicher

```
1 package besysMonitor;
2 public class ProdConsBuffer {
3     private final int capacity = 10;
4     private final int[] buffer = new int[capacity];
5     private int count = 0;
6     private int low = 0;
7     private int high = 0;
8
9     public ProdConsBuffer(int initialFill){
10        if ((initialFill > 0) && (initialFill < capacity)){
11            for (int i = 0; i < initialFill; i++){
12                buffer[i] = i;
13            }
14            high = initialFill;
15            count = initialFill;
16        }
17    }
```

```
18
19      public synchronized void insert(int value){
20          while (count == capacity){
21              gotoSleep();
22          }
23          buffer[high] = value;
24          ++count;
25          high = ++high % capacity;
26          if (count == 1){
27              notify();
28          }
29          printMsg( "added", value, count);
30          return;
31      }
32
33      public synchronized int getItem(){
34          int value;
35          while (count == 0){
36              gotoSleep();
37          }
38          value = buffer[low];
39          --count;
40          low = ++low % capacity;
41          if (count == capacity - 1){
42              notify();
43          }
44          printMsg( "   removed", value, count);
45          return value;
46      }
47
48      private void gotoSleep(){
49          try{
50              wait();
51          }
52          catch(InterruptedException ex){
53              ex.printStackTrace();
54          }
55      }
56
57      private void printMsg(String start, int value, int count){
58          System.out.print(start + " value: " + Integer.toString(value));
59          System.out.println(" count: " + Integer.toString(count));
60      }
61  }
```

B.3 Der Produzent

```
1 package besysMonitor;
2 public class Producer extends Thread {
3     private ProdConsBuffer buffer = null;
4     private int delay;
5     private int count = 0;
6
7     public Producer(ProdConsBuffer buffer, int delay){
8         this.buffer = buffer;
9         this.delay = delay;
10    }
11
12    public void run(){
13        int item;
14        while(true){
15            item = produceItem();
16            buffer.insert(item);
17            sleepAmoment(delay);
18        }
19    }
20
21    int produceItem(){
22       count = ++count % 99;
23       return count;
24    }
25
26    void sleepAmoment(int time){
27        int rand = (int) Math.round(400 * Math.random());
28        try {
29            this.sleep(time + rand);
30        } catch (InterruptedException e) {
31            e.printStackTrace();
32        }
33    }
34 }
```

B.4 Der Konsument

```
1 package besysMonitor;
2 public class Consumer extends Thread {
3     private ProdConsBuffer buffer = null;
4     private int delay;
5
6     public Consumer(ProdConsBuffer buffer, int delay){
7         this.buffer = buffer;
8         this.delay = delay;
9     }
10
```

```
11      public void run(){
12          int item;
13          while(true){
14              item = buffer.getItem();
15              consumeItem(item);
16              sleepAmoment(delay);
17          }
18      }
19
20      private void consumeItem(int item){
21          // System.out.println("consumed: "+ Integer.toString(item));
22          return;
23      }
24
25      void sleepAmoment(int time){
26          int rand = (int) Math.round(400 * Math.random());
27          try {
28              this.sleep(time + rand);
29          } catch (InterruptedException e) {
30              e.printStackTrace();
31          }
32      }
33  }
```

C Beispielprogramm für Signale in C

C.1 Die Signal-Handler

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <string.h>
5 #include <sys/signal.h>
6
7 void sigusr1_handler(int sig_nr){
8     int sleepVal;
9     printf("\n got signal SIGUSR1 %d ...", sig_nr);
10    fflush(stdout);
11    sleepVal = sleep(1);
12    printf(" ... SIGUSR1 done");
13    fflush(stdout);
14    return;
15 }
```

```
1 void sigusr2_handler(int sig_nr){
2     int sleepVal;
3     printf("\n got SIGUSR2 %d ...", sig_nr);
4     fflush(stdout);
5     sleepVal = sleep(1);
6     printf(" ... SIGUSR2 done");
7     fflush(stdout);
8     return;
9 }
```

C.2 Installation der Signal-Handler

```
1 void installSignalHandler(){
2     struct sigaction sa_usr1;
3     struct sigaction sa_usr2;
4     sigset_t child_sigset;
5
6     sigemptyset(&child_sigset);
7
8     sa_usr1.sa_handler = sigusr1_handler;
9     sa_usr1.sa_mask  = child_sigset;
10    sa_usr1.sa_flags = 0;
11
12    if (sigaction(SIGUSR1, &sa_usr1, NULL) != 0){
13        printf("\n could not install handler for SIGUSR1 ");
14    }else{
15        printf("\n handler for SIGUSR1 installed");
```

```
16      }
17
18      sa_usr2.sa_handler = sigusr2_handler;
19      sa_usr2.sa_mask  = child_sigset;
20      sa_usr2.sa_flags = 0;
21
22      if (sigaction(SIGUSR2, &sa_usr2, NULL) != 0){
23          printf("\n could not install handler for SIGUSR2 ");
24      }else{
25          printf("\n handler for SIGUSR2 installed");
26      }
27
28      return;
29  }
```

C.3 Der Kind-Prozess

```
1  void childFunction(){
2    int i = 0;
3    int sleepVal;
4
5    installSignalHandler();
6
7    while (i < 10){
8      sleepVal = sleep(1);
9      i++;
10   }
11   printf("\n exit child process \n");
12   exit(0);
13 }
```

C.4 Der Eltern-Prozess

```
1  void parentFunction(pid_t pid){
2    int i = 0;
3    int sleepVal;
4    pid_t waitVal = 0;
5
6    kill(pid, SIGUSR1);
7    kill(pid, SIGUSR2);
8    kill(pid, SIGUSR2);
9    kill(pid, SIGUSR1);
10
11   while (waitVal == 0){
12      kill(pid, SIGUSR1);
13      sleepVal = sleep(2);
14      kill(pid, SIGUSR2);
```

```
15       waitVal = waitpid(pid, NULL, WNOHANG);
16       sleepVal = sleep(2);
17   }
18   return;
19 }
```

C.5 Die Main-Funktion

```
1  int main(void){
2    pid_t pid;
3
4    printf("\n SIGUSR1: %d, SIGUSR2: %d ", SIGUSR1, SIGUSR2);
5    pid = fork();
6    if (pid == 0){
7      childFunction();
8    }
9    if (pid > 0){
10     parentFunction(pid);
11   }
12   if (pid < 0){
13     printf("\n P: could not create process ");
14   }
15
16   return;
17 }
```

D Ein Beispiel für Shared Memory in C

D.1 Globale Datentypen und Funktionen

```
1  #ifndef SHAREDMEMCOMMON
2  #define SHAREDMEMCOMMON
3
4  #include <sys/types.h>
5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <sys/sem.h>
8  #include <unistd.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 extern const key_t shmKey;
13 extern const int semKey;
14
15 struct ipc_msg{
16     pid_t fromPid;
17     float value;
18     char text[20];
19 };
20 typedef struct ipc_msg IPC_MSG;
21
22 #define SHM_BUF_CAPACITY 10
23 extern const int shmBufCapacity;
24
25 struct shm_buffer{
26     int high;
27     int low;
28     IPC_MSG messages[SHM_BUF_CAPACITY];
29 };
30 typedef struct shm_buffer SHM_BUFFER;
31 typedef SHM_BUFFER *SHM_BUF_PTR;
32
33 union semun {
34     int val;
35     struct semid_ds *buf;
36     unsigned short  *array;
37 };
38 typedef union semun SEM_ARG;
39
40 int  getSemaphores();
41 void getReadAccess(int);
42 void getWriteAccess(int);
43 void releaseMutex(int);
```

```
44 void printMsg(IPC_MSG);
45
46 #endif
```

D.2 Implementierung der globalen Funktionen

```
 1 #include <errno.h>
 2 #include "sharedMemCommon.h"
 3
 4 const key_t shmKey = 1928;
 5 const int   semKey = 1928;
 6 const int   shmBufCapacity = SHM_BUF_CAPACITY;
 7
 8 void printSemaphoreState(int semID);
```

```
 1 void printMsg(IPC_MSG aMsg){
 2     printf("\n msg from process: %d;", aMsg.fromPid);
 3     printf(" value: %5.2f;", aMsg.value);
 4     printf(" text: %s;",  aMsg.text);
 5     fflush(stdout);
 6     return;
 7 }
```

```
 1 int getSemaphores(){
 2     int semId;
 3     int flags = 0600 | IPC_CREAT;
 4
 5     semId = semget(semKey, 3, flags);
 6     if (semId == -1){
 7         printf ("\n could not get semaphores ");
 8         exit(-4);
 9     }
10     return  semId;
11 }
```

```
 1 void getReadAccess(int semId){
 2     int retVal;
 3     int semFlag = 0;  /* not SEM_UNDO */
 4     struct sembuf semCmds[3];
 5     printSemaphoreState(semId);
 6     /* down on mutex */
 7     semCmds[0].sem_num =  0;
 8     semCmds[0].sem_op  = -1;
 9     semCmds[0].sem_flg = semFlag;
10
11     /* increment free slots */
12     semCmds[1].sem_num =  1;
13     semCmds[1].sem_op  = +1;
```

```
14      semCmds[1].sem_flg = semFlag;
15
16      /* decrement filled slots */
17      semCmds[2].sem_num =  2;
18      semCmds[2].sem_op  = -1;
19      semCmds[2].sem_flg = semFlag;
20
21      retVal = semop(semId, semCmds, 3);
22      printSemaphoreState(semId);
23      if (retVal == -1){
24          printf ("\n could not get read access on semaphores ");
25          printf ("error no: %d ", errno);
26          exit(-6);
27      }
28      return;
29  }
```

```
 1  void getWriteAccess(int semId){
 2      int retVal;
 3      int semFlag = 0;  /* not SEM_UNDO */
 4      struct sembuf semCmds[3];
 5      printSemaphoreState(semId);
 6      /* down on mutex */
 7      semCmds[0].sem_num =  0;
 8      semCmds[0].sem_op  = -1;
 9      semCmds[0].sem_flg = semFlag;
10
11      /* decrement free slots */
12      semCmds[1].sem_num =  1;
13      semCmds[1].sem_op  = -1;
14      semCmds[1].sem_flg = semFlag;
15
16      /* increment filled slots */
17      semCmds[2].sem_num =  2;
18      semCmds[2].sem_op  = +1;
19      semCmds[2].sem_flg = semFlag;
20
21      retVal = semop(semId, semCmds, 3);
22      printSemaphoreState(semId);
23      if (retVal == -1){
24          printf ("\n could not get write access to semaphores ");
25          printf ("error no: %d ", errno);
26          exit(-7);
27      }
28      return;
29  }
```

```
1 void releaseMutex(int semId){
2     int retVal;
3     struct sembuf semCmd = {0, +1, 0};
4     retVal = semop(semId, &semCmd, 1);
5     if (retVal == -1){
6         printf ("\n could not releas mutex ");
7         exit(-8);
8     }
9     printf ("  mutex released ");
10    return;
11 }
```

```
1 void printSemaphoreState(int semId){
2     int retVal;
3     SEM_ARG semArg;
4     unsigned short  semValues[3];
5     semArg.array = semValues;
6     retVal = semctl(semId, 0, GETALL, semArg);
7     if (retVal == -1){
8         printf ("\n could not query semaphores ");
9         exit(-5);
10    }
11    printf(" <%d, %2d, %2d> ", semValues[0], semValues[1], semValues[2]);
12    fflush(stdout);
13    return;
14 }
```

D.3 Der lesende Prozess

```
1 #include "sharedMemCommon.h"
2
3 void readFromSharedMem(int, SHM_BUF_PTR);
4 void initSharedMem(SHM_BUF_PTR);
5 SHM_BUF_PTR attachSharedMem(int *, int, int);
6 void initSemaphores(int);
7
8 int main(void){
9     int result = 0;
10    int shMemId = 0;
11    SHM_BUF_PTR shmBuffer = NULL;
12    int semId = getSemaphores();
13    initSemaphores(semId);
14
15    shmBuffer  = attachSharedMem(&shMemId, shmKey,sizeof(SHM_BUFFER));
16    initSharedMem(shmBuffer);
17    readFromSharedMem(semId, shmBuffer);
18
```

```
19      result = shmctl(shMemId, IPC_RMID, NULL);
20      if (result == -1){
21          printf("\n reader could not mark shared memory for deallocating");
22      }
23      result = shmdt(shmBuffer);
24      if (result == -1){
25         printf("\n reader could not dettach shared memory");
26
27      }
28
29      return 0;
30  }
```

```
1  void readFromSharedMem(int semId, SHM_BUF_PTR bufferPtr){
2      IPC_MSG aMsg;
3      int i;
4      for(i = 0; i < 24 ; i++){
5         printf("\n reader . .");
6         fflush(stdout);
7         getReadAccess(semId);
8         aMsg = bufferPtr->messages[bufferPtr->low];
9         bufferPtr->low = (bufferPtr->low + 1) % shmBufCapacity;
10        releaseMutex(semId);
11
12        printf(" . sleeping ");
13        fflush(stdout);
14        sleep(3);
15        printMsg(aMsg);
16     }
17     return;
18 }
```

```
1  SHM_BUF_PTR attachSharedMem(int *shMemId, int key, int size){
2      SHM_BUF_PTR shmBuffer = NULL;
3
4      /* int flag = 432 | IPC_CREAT | IPC_EXCL | IPC_RMID; */
5      int flag = 432 | IPC_CREAT;
6
7      *shMemId = shmget(key, size, flag);
8      if (*shMemId == -1){
9          printf("\n could not create shared memory");
10         exit(-1);
11     }
12     shmBuffer = shmat(*shMemId, NULL, 0);
13     if ((int) shmBuffer == -1){
14        printf("\n could not attach shared memory");
15        exit(-2);
```

```
16      }
17      return shmBuffer;
18  }
```

```
1  void initSharedMem(SHM_BUF_PTR bufferPtr){
2      if (bufferPtr == NULL){
3          exit(-3);
4      }
5      bufferPtr->high = 0;
6      bufferPtr->low = 0;
7      return;
8  }
```

```
1  void initSemaphores(int semId){
2      int retVal;
3      SEM_ARG semArg;
4      unsigned short  semValues[] = {1, SHM_BUF_CAPACITY, 0};
5      semArg.array = semValues;
6      retVal = semctl(semId, 0, SETALL, semArg);
7      if (retVal == -1){
8          printf ("\n could not initialize semaphores ");
9          exit(-5);
10     }
11     return;
12 }
```

D.4 Der schreibende Prozess

```
1  #include "sharedMemCommon.h"
2  #include <string.h>
3
4  void writeToSharedMem(int, SHM_BUF_PTR);
5  void writeToSharedMem(in
```

```
1  void writeToSharedMem(in
2  int main(void){
3      int flag = 432 | IPC_CREAT;
4      int result = 0;
5      int shMemId = 0;
6      SHM_BUF_PTR shmBuffer = NULL;
7      int semId = getSemaphores();
8
9      shMemId = shmget(shmKey, sizeof(SHM_BUFFER), flag);
10     if (shMemId == -1){
11         printf("\n could not create shared memory");
12         exit(-1);
13     }
14     shmBuffer = shmat(shMemId, NULL, 0);
```

```
15      if ((int)shmBuffer == -1){
16          printf("\n could not attach shared memory");
17          exit(-2);
18      }
19
20      writeToSharedMem(semId, shmBuffer);
21
22      result = shmdt(shmBuffer);
23      if (result == -1){
24          printf("\n writer could not dettach shared memory");
25
26      }
27      return 0;
28  }
```

```
1  void writeToSharedMem(int semId, SHM_BUF_PTR bufferPtr){
2      int i;
3      pid_t myPid = getpid();
4      int msgId;
5
6      for(i = 0; i < 12 ; i++){
7          printf("\n writer . .");
8          fflush(stdout);
9          getWriteAccess(semId);
10         msgId = bufferPtr->high;
11
12         bufferPtr->messages[msgId].fromPid = myPid;
13         bufferPtr->messages[msgId].value   = (float) (myPid%(19+i)) / 3;
14         strncpy(bufferPtr->messages[msgId].text, "shared mem msg:  ", 19);
15         bufferPtr->messages[msgId].text[17] = i%10 + '0';
16         bufferPtr->messages[msgId].text[19] = 0;
17
18         bufferPtr->high = (bufferPtr->high + 1) % shmBufCapacity;
19         releaseMutex(semId);
20
21         printf(". sleeping");
22         fflush(stdout);
23         sleep(2);
24         printf(" . done %d ", i);
25         fflush(stdout);
26     }
27     return;
28 }
```

E Ein Beispiel für Sockets in Java

E.1 Der Server

```
1 package besysSockets;
2
3 import java.net.ServerSocket;
4 import java.net.Socket;
5 import java.io.InputStream;
6 import java.io.OutputStream;
7
8 public class Server {
9    public static void main(String[] args) {
10      try{
11         ServerSocket  aServerSocket = new ServerSocket(4242);
12         int result = 0;
13         while (result != 10){
14            Socket aSocket = aServerSocket.accept();
15
16            InputStream  inStr  = aSocket.getInputStream();
17            OutputStream outStr = aSocket.getOutputStream();
18
19            int length = inStr.read();
20            int width  = inStr.read();
21            result = length * width;
22            System.out.println("read length: " + Integer.toString(length));
23            System.out.println("read width: " + Integer.toString(width));
24
25            outStr.write(result);
26            aSocket.close();
27         }
28         aServerSocket.close();
29      }catch(Exception ex){
30         ex.printStackTrace();
31      }
32      return;
33   }
34 }
```

E.2 Der Client

```
1 package besysSockets;
2
3 import java.net.ServerSocket;
4 import java.net.Socket;
5 import java.io.InputStream;
6 import java.io.OutputStream;
```

```
 7
 8 public class Client {
 9    public static void main(String[] args) {
10       int length = 6;
11       int width  = 8;
12       if (args.length > 0){
13          length = Integer.parseInt(args[0]);
14       }
15       if (args.length > 1){
16           width = Integer.parseInt(args[1]);
17       }
18       try{
19          Socket aSocket = new Socket("localhost", 4242);
20
21          InputStream  inStr  = aSocket.getInputStream();
22          OutputStream outStr = aSocket.getOutputStream();
23
24          outStr.write(length);
25          outStr.write(width);
26          int result = inStr.read();
27          System.out.println("read result: " + Integer.toString(result));
28
29          aSocket.close();
30       }catch(Exception ex){
31          ex.printStackTrace();
32       }
33       return;
34    }
35 }
```