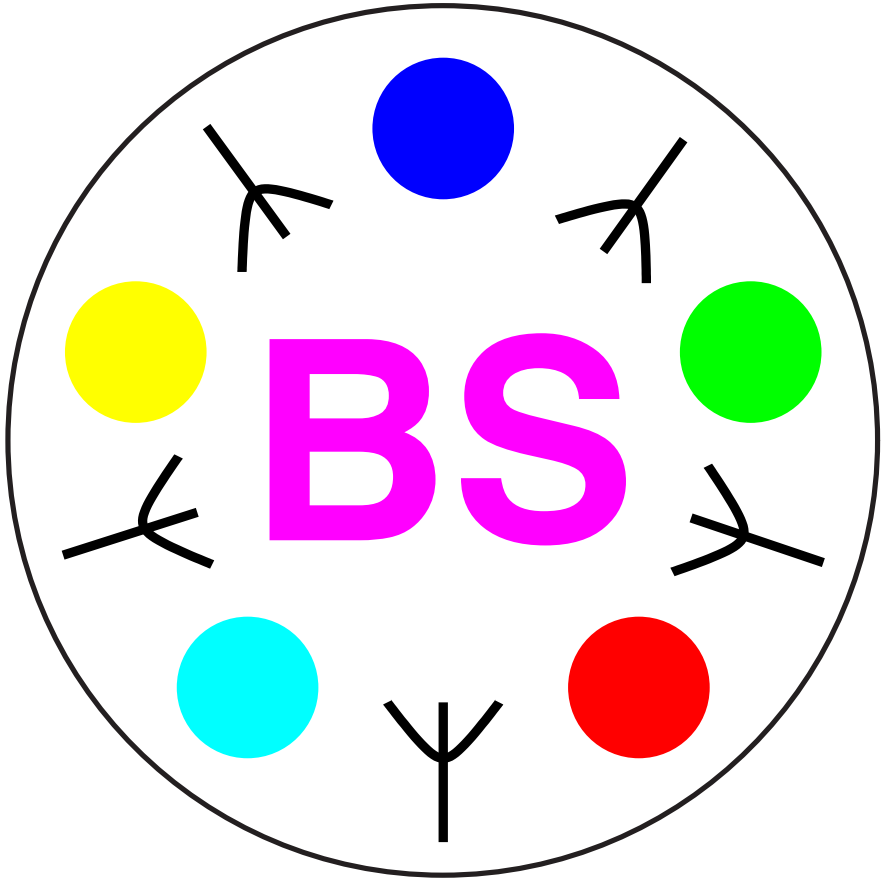
Prof. Dr. H. Pagnia

Betriebssysteme



DHBW Mannheim

Wichtiges zur Vorlesung

Prof. Dr. Henning Pagnia

- Wirtschaftsinformatik
- Email: **pagnia@dhbw-mannheim.de**
- Telefon: **0621 / 4105-1131**
- Raum: **149 B**

EINLEITUNG

Zum Verlauf der Vorlesung

- **Überblick über das Themenfeld**
- **Hardware-Grundlagen**
- **Prozessverwaltung**
  - ◇ Prozesse und Threads
  - ◇ Gegenseitiger Ausschluss
  - ◇ Dispatcher und Scheduler
  - ◇ Ein- und Ausgabe
  - ◇ Semaphore
- **Speicherverwaltung**
  - ◇ Freispeicherverwaltung
  - ◇ Virtueller Speicher
- **Dateisysteme**
  - ◇ Dateideskriptoren
  - ◇ Verzeichnisbäume
  - ◇ Dateischutz

Literatur

[Bau06] Uwe Baumgarten: *Betriebssysteme: Eine Einführung*. Oldenbourg, 6. Auflage, 2006, ISBN 978-3486582116.

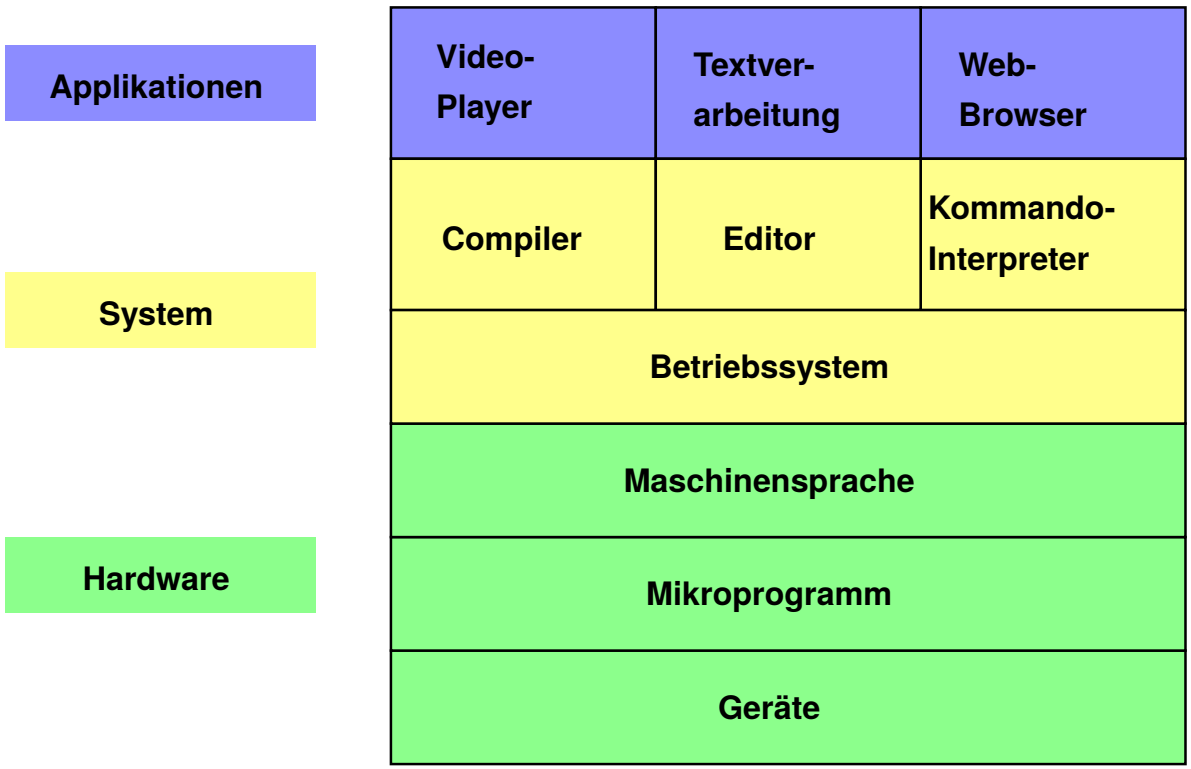
[Bra17] Rüdiger Brause: *Betriebssysteme: Grundlagen und Konzepte*. Springer Vieweg, Berlin, 4. Auflage, 2017, ISBN 978-3662540992.

[Man14] Peter Mandl: *Grundkurs Betriebssysteme:*. Springer Vieweg, Wiesbaden, 4. Auflage, 2014, ISBN 978-3-658-06217-0.

[Sta17] William Stallings: *Operating Systems: Internals and Design Principles*. Pearson, 9. Auflage, 2017. ISBN 978-1292214290.

[TB16] Andrew S. Tanenbaum und Herbert Bos: *Moderne Betriebssysteme*. Pearson Studium, 4. Auflage, 2016. ISBN 978-3868942705.

Was ist ein Betriebssystem ?



Welche Aufgaben sollten von einem Betriebssystem erledigt werden?

Anforderungen an ein Betriebssystem

- Verwaltung von nebenläufigen Abläufen
- Bereitstellung von Synchronisationsmechanismen
- Hauptspeicherverwaltung
- Dateiverwaltung

- Hardware-unabhängige Programmierung und Bedienung
- Unterstützung für möglichst viele verschiedene E/A-Geräte
- Ressourcenverwaltung, Energiemanagement

- Benutzerverwaltung
- Benutzerschnittstelle, grafische Benutzeroberfläche

- Netzwerkkommunikation

Definitionen des Begriffs “Betriebssystem”

- Definition des Betriebsystems nach ANSI: Operating System

  Software which controls the execution of computer programs and which may provide scheduling, debugging, input/output control, accounting, compilation, storage assignment, data management, and related services

- Definition nach DIN 44300

  Die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

- “An elephant is a mouse
  running an Operating System.”
  (D. Knuth ?)

Wichtige Dienste von Betriebssystemen

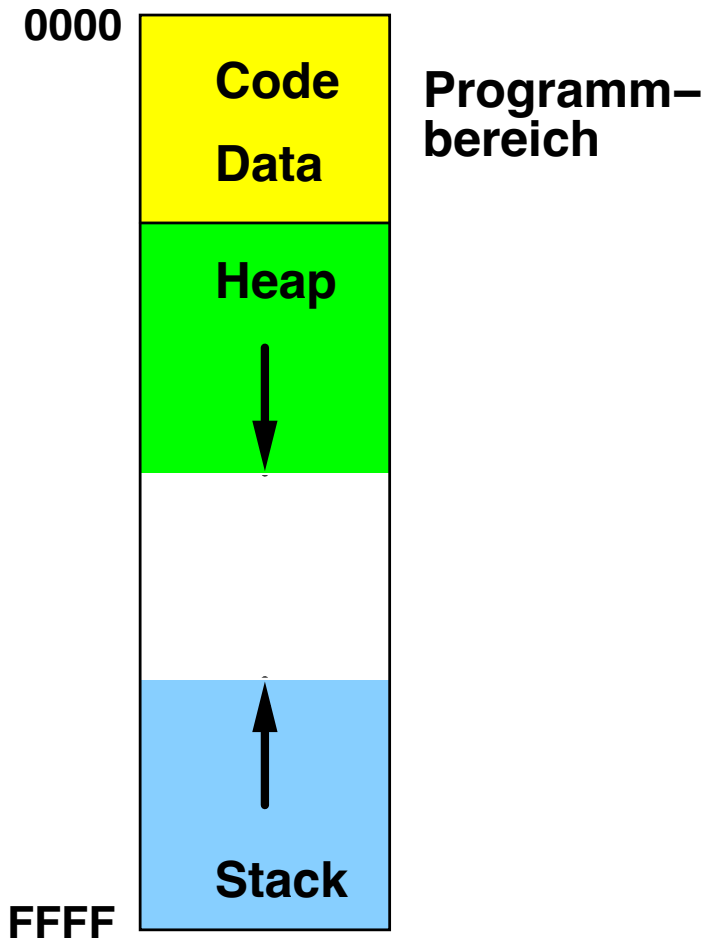
Prozessverwaltung

- Dienst zur Verwaltung von **Benutzeraufträgen**
  $\Rightarrow$ auch *Rechenauftrag* oder *Auftrag* (*Task*, *Job*) genannt
- Benutzeraufträge werden vom Augenblick ihrer **Erzeugung** bis zur **Beendigung** als **Prozesse** organisiert.
- Die Prozessverwaltung soll es ermöglichen, zu einem Zeitpunkt mehrere Prozesse in Arbeit zu haben.
  ($\Rightarrow$ *Multi-Processing*)
  $\Rightarrow$ Bessere Auslastung des Prozessors und der E/A-Geräte!
  $\Rightarrow$ Abstraktion von der tatsächlich vorhandenen Anzahl an Prozessoren.

Ein Prozess definiert eine **Arbeitseinheit** für den Prozessor.

Speicherverwaltung

- Verwaltung des Hauptspeichers
  - Welche Bereiche sind belegt, welche frei ?
- Verwaltung und Organisation von Adressräumen
  - Erzeugen eines **Adressraums**
    (Def.: virtueller Speicher eines Prozesses)
  - Zuteilung von Hauptspeicher an Adressräume
  - Verdrängung von ganzen Adressräumen oder Adressraumteilen in den Hintergrundspeicher
  - Löschen von Adressräumen
- Realisierung und Kontrolle des Zugriffsschutzes für Adressräume



Prozessinteraktion

- Über diesen Dienst können Prozesse gezielt **Informationen** austauschen.
  ($\Rightarrow$ IPC: Inter Process Communication)
  - ◇ über gemeinsame Speicherbereiche
  - ◇ über Botschaftenaustausch

- Weitere Aufgabe: **Prozesssynchronisation**
  - Bei auftretenden Zugriffskonflikten werden zugreifende Prozesse ggf. entsprechend verzögert bis der Zugriff sicher möglich ist.
    - ◇ z. B. beim Zugriff auf gemeinsam genutzte Daten oder Geräte

Datenhaltung

- Aufgabe dieses Dienstes ist die **langfristige** Aufbewahrung von Daten und Programmen
- wichtigste realisierte Abstraktion: **Datei**
- Operationen auf Dateien
  - erzeugen $\rightarrow$ `create`
  - löschen $\rightarrow$ `delete`
  - öffnen (für eine vorgesehene Bearbeitungsform, z.B. nur lesender Zugriff) $\rightarrow$ `open`
  - schließen $\rightarrow$ `close`
  - lesen $\rightarrow$ `read`
  - erweitern (d.h. am Ende Daten hinzufügen) $\rightarrow$ `append`
  - ändern $\rightarrow$ `write`
- Datenhaltungsdienst ist Grundlage zum Aufbau weiterer Dienste
  $\rightarrow$ z.B. Verzeichnisdienst, Zugriffskontrolle mittels Dateischutz

Einige Betriebssysteme

- **Microsoft Windows**
  - BS-Familie: u. a. Win95, WinNT, WinXP, Win7, Win10
  - BS für Intel-Architekturen
- **Unix und UNIX-ähnliche**
  - BS für Workstations, Mainframes, PCs, Handhelds, ...
  - implementiert in C $\Rightarrow$ portierbar
  - netzwerkfähig (TCP/IP)
  - kommerziell: AIX, HP-UX, Solaris, ...
  - Open-Source-Varianten: FreeBSD, Linux, ...
- **MacOS**
  - BS früher für Power-PC Architekturen, heute Intel
  - Mac-OS-X basiert auf dem Mach-Kernel und FreeBSD-Unix
- **Mobile Operating Systems: Android, iOS**
  - BS i. W. für Smartphones und Tablets
    - ◇ App-Konzept, App Stores, Energieverwaltung, Offline-Modus

Betriebsarten eines Rechensystems

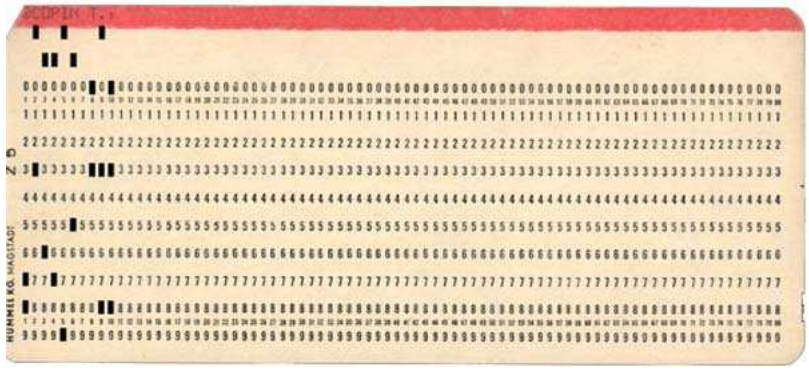
Stapelbetrieb (*Batch Processing*)

- Anforderungen an das System werden zusammenhängend als **Auftrag** (*Task*, *Job*) gestellt
  $\Rightarrow$ Abarbeitung (am Stück) ohne weitere Einflussnahme von außen

- (mögliche) Bestandteile eines Auftrags:
  - Programme, Daten und Steueranweisungen

- Verwendung des Stapelbetriebs für verarbeitungs- und datenintensive Aufgaben

- Eingabemedium beim Stapelbetrieb
  - früher: Lochkartenstapel



  - heute: Kommandodatei (auch: *Shell-Skript*)

Interaktiver Betrieb

- (auch: Dialogbetrieb, Interactive Processing)
- Die Ausführung einer Aufgabe besteht aus einer Folge von **Interaktionszyklen**.
  - Definition von Teilaufgaben durch den Benutzer
  - Antwort des Systems
- Worin besteht die Interaktion?
  - Beschreibung einer Teilaufgabe (z.B. Kommandos und Daten) und Ausführung derselben

  $\Rightarrow$ Benutzer kann auf den Ablauf der Auftragsbearbeitung Einfluss nehmen.

Echtzeitbetrieb (*Real Time Processing*)

- Anwendung: z.B. Steuerung technischer Prozesse
- Wichtigste Anforderung:
  - auf **asynchron** anfallende Daten muß innerhalb eines **vorgegebenen Zeitintervalls** reagiert werden (harte / weiche Echtzeitanforderung)
- Verspätung könnte katastrophale Folgen haben $\rightarrow$ Notbremsung in einem Auto

Einbenutzerbetrieb (*Single-User*)

- Nur ein Benutzer arbeitet mit dem Betriebssystem.

Mehrbenutzerbetrieb (*Multi-User*)

- mehrere Benutzer teilen sich die Betriebsmittel (z.B. Prozessor, Speicher, Festplatte)

Freie Programmierbarkeit (*User-Programmable*)

- Benutzer darf eigene Programme schreiben und anwenden.

Keine Programmierbarkeit (*Non-Programmable*)

- Benutzer darf lediglich bestehende Programme anwenden.
- Beispiele:
  - Endanwendersysteme in Banken und Versicherungen
  - Embedded-Systems: Navigationssysteme, Betriebssysteme einfacher Mobiltelefone, Sat-Receiver

Einprogrammbetrieb (*Uniprogramming*)

- Es befindet sich maximal ein Programm zur Ausführung im Hauptspeicher.

Mehrprogrammbetrieb (*Multiprogramming*)

- Es können gleichzeitig mehrere ausführbare Programme im Hauptspeicher sein.

Time-Sharing-Betrieb

Auch: „Die Fähigkeit, mehrere Programme gleichzeitig abstürzen zu lassen."

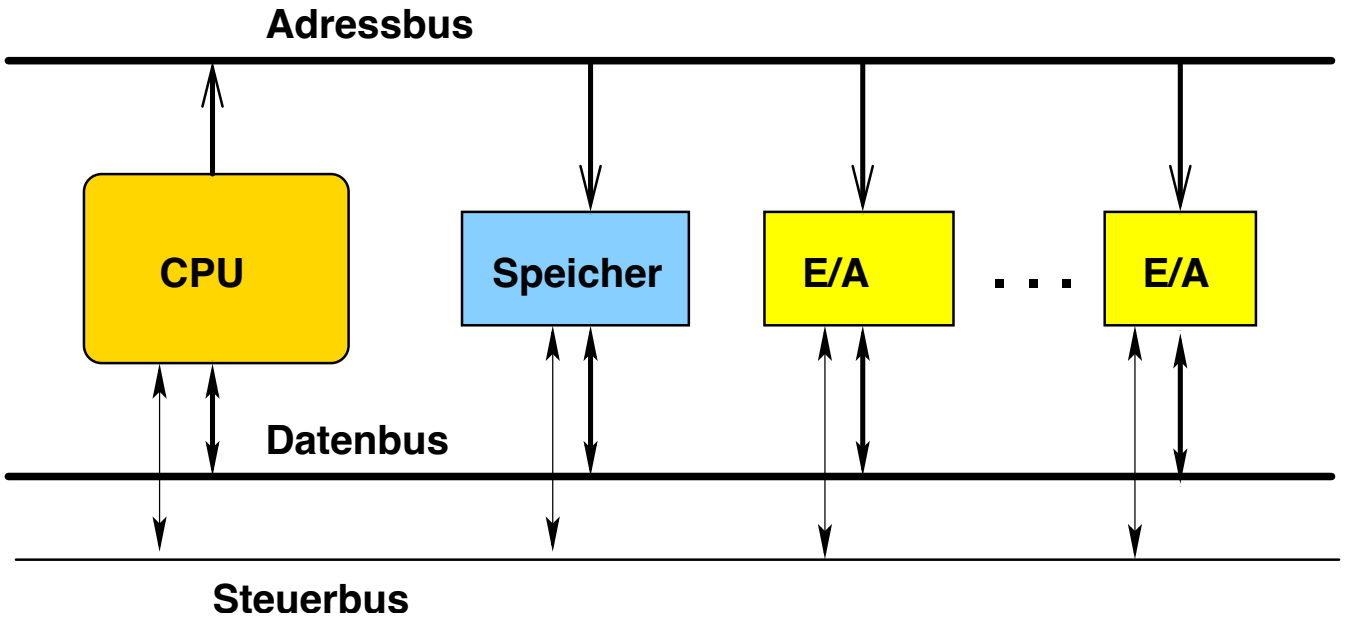
- Prozessor arbeitet im **Multiplex-Betrieb**
  $\Rightarrow$ mehrere Programme werden stückweise abwechselnd ausgeführt.
- Beispiele: Großrechnerbetriebssysteme, Windows (ab NT), Unix/Linux, MacOS, etc.

Hardware-Grundlagen

Von-Neumann-Rechner

- John (Janos) von Neumann (geb. 1903 in Ungarn, gest. 1957)
- problemunabhängige Maschinenarchitektur
- **damals revolutionär:**
  - programmgesteuerter Universalrechner !
    $\Rightarrow$ Daten **UND** Programm gleichzeitig im Speicher

Architektur nach von Neumann



Komponenten

- **Prozessor (Central Processing Unit, CPU)**
  - führt Befehle aus und verändert Daten
    $\Rightarrow$ Daten- und Befehlsprozessor
  - 1 Befehl/Zeit
    $\Rightarrow$ arbeitet sequenziell

- **Speicher**
  - Feld von gleichartigen Speicherzellen, die Werte aufnehmen können
  - Speicherzelle = kleinste adressierbare Einheit
  - Byte-orientiert (1 Byte = 8 Bit) oder
    Wort-orientiert (1 Wort = 16, 24, ..., 64 Bit)
  - Speicherkapazität:
    - 1 KiB = 2^{10} Byte (wird ugs. meist als 1 KB bezeichnet)
    - 1 MiB = 2^{20} Byte (ebenso: 1 MB)
    - 1 GiB = 2^{30} Byte (ebenso: 1 GB)
    - 1 TiB = 2^{40} Byte (ebenso: 1 TB)

- **E/A-Geräte**
  - Bildschirm
  - Tastatur
  - Festplatte
  - Drucker
  - CD, DVD
  - ...

  $\Rightarrow$ Ein-/Ausgabegeräte bilden die Schnittstelle zur Außenwelt

  - E/A-Geräte arbeiten oft unabhängig vom Prozessor

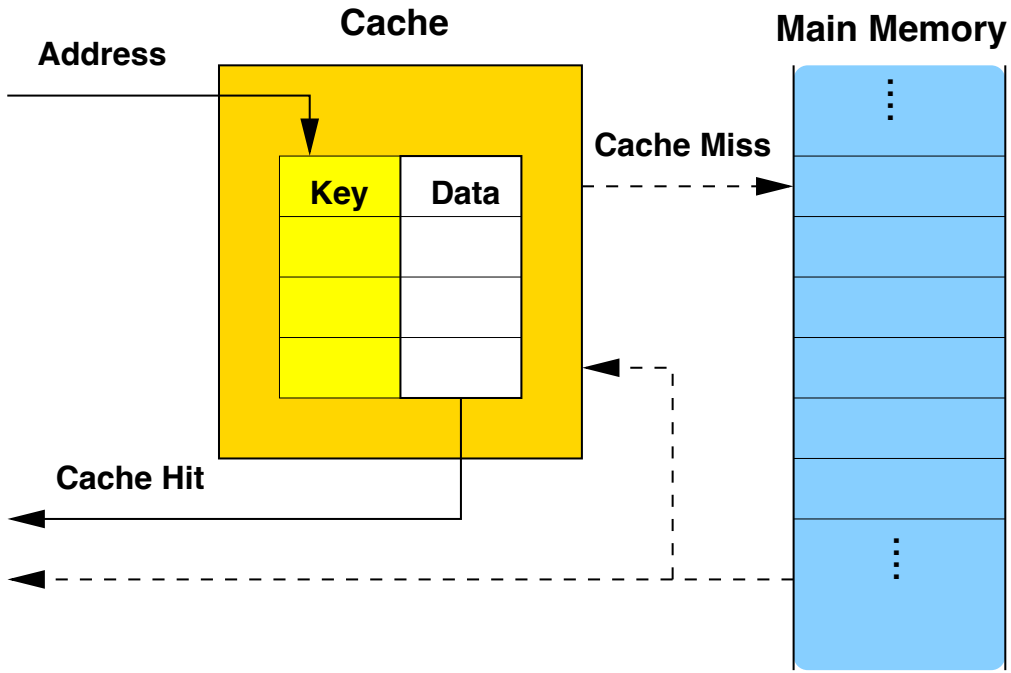
    $\Rightarrow$ parallele Abläufe im System $\Rightarrow$ Synchronisation notwendig

  - DMA (Direct Memory Access):
    - ◇ Kopieren von Daten zwischen Speicher und E/A-Gerät wird vom E/A-Controller ohne Hilfe der CPU durchgeführt

Speicher

- **Hauptspeicher (Main Memory, MM [ ])**
  - **ROM** (Read Only Memory)
    - ◇ behält Inhalte auch ohne Stromversorgung
    - ◇ enthält z. B. grundlegendes Bootprogramm (BIOS)
  - **RAM** (Random Access Memory)
    - ◇ Speichern und Auslesen von Daten und Befehlen
    - ◇ enthält pro Problem eine Befehlsfolge ($\Rightarrow$ Programm)

- **Cache-Speicher**
  - schneller Assoziativspeicher
    - ◇ Nach Anlegen eines Schlüsselwertes werden alle Cache-Lines **gleichzeitig** nach dessen Auftreten durchsucht.



Vorteil: Geschwindigkeitsteigerung bei vielen Cache-Hits

- Kaskadierte Caches
  - ◇ Level-1-Cache, auf Prozessorchip
  - ◇ Level-2-Cache, im Prozessor oder zwischen Prozessor und Speicher
  - ◇ Level-3-Cache / Last-Level-Cache, zwischen Prozessor und Speicher
- Typische Speichergrößen und -zugriffszeiten
  - ◇ Register: 1 – 2 KB $<$ 1 ns
  - ◇ L1-Cache: 64 – 512 KiB, 1–2 ns
  - ◇ L2 / L3-Cache: 1 – 16 MiB
  - ◇ Hauptspeicher: 1 – 64 GiB, 10 ns
  - ◇ (Festplatte: mehrere TB, 10 ms)
- Aber: Cache-Kohärenz wahren!
  - ◇ Verschiedene Cache-Strategien: Write-Back, Write-Through, Snoopy,...

Bestandteile des Prozessors

- **Rechenwerk** (Arithmetic and Logic Unit, ALU)
  - erlaubt Datenmanipulation, z.B. Rechenoperationen
- **Steuerwerk** (Steuerung der Befehlsabarbeitung)
  - koordiniert Ausführung der Instruktionen
- **Register** (schnelle Mehrzweckspeicher)
  - Mehrzweckregister (Rechnen und Adressieren)
  - Gleitpunktregister (Gleitkommarechnungen)
  - Steuerregister (Steuer- und Zustandsinformationen)
    - ◇ Befehlszähler IC (Instruction Counter)
    - ◇ Befehlsregister IR (Instruction Register)
    - ◇ Kellerregister SP (Stack Pointer)
    - ◇ ...

  $\Rightarrow$ i.d.R. mehrere Registersätze (Registerbänke)

Betriebszustände des Prozessors

- **Maschinenzustand:**
  - AN / AUS / BOOTEND
- **Funktionszustand:**
  - definiert durch gültigen Registersatz
- **Privilegierungszustand:**
  - zur Umsetzung eines Schutzkonzeptes sind mindestens zwei Modi notwendig
    - ◇ USER
      - ⇒ nur eingeschränkter Befehlssatz
    - ◇ SYSTEM (auch PRIVILEGED)
      - ⇒ alle Maschinenbefehle zulässig
  - Zwischenabstufungen sind gängig (prozessorabhängig)

Kennen Sie ein Beispiel für einen privilegierten Befehl?

Befehlszyklus

- **Schematisch:**

```
while (true){
  IR := MM[IC]
  IC := IC + 1
  IR ausfuehren
}
```

$\Rightarrow$ **Es resultiert eine sequenzielle Befehlsabarbeitung.**

Sequenziell reicht aber nicht !

- **Sprungbefehl (Jump):**

  `IC := <wert>`

- **Prozedur-/ Methodenaufruf (Call):**

```
push IC  // auf den Stack
  IC := <Adresse des ersten Befehls>
  <Prozedur-Befehle abarbeiten>
pop IC   // vom Stack
```

- **Unterbrechung (Interrupt):**
  - spezielles Signal , meist von E/A-Geräten an den Prozessor
    - ◇ von Tastatur, Netzwerkkarte, Festplatte, ...

Unterbrechungskonzept

- **Ursachen einer Unterbrechung**
  - programmbezogen (synchron) $\Rightarrow$ **Software-Interrupt, Trap**
    - ◇ Adressfehler, Division durch Null, System-Call, ...
  - systembezogen (asynchron) $\Rightarrow$ **Hardware-Interrupt**
    - ◇ Disk-Interrupt, Timer-Interrupt, ...

- **Wirkung einer Unterbrechung**
  - Sie unterbricht den sequenziellen Befehlszyklus des laufenden Programms
  - Welcher Code wird ausgeführt ?

  $\Rightarrow$ Der **Interrupt-Vektor** enthält Einsprungadressen der Behandlungsroutinen (in Form einer Sprungtabelle).
  - Nach Abarbeiten einer **asynchronen** Unterbrechung bzw. von System-Calls wird das Programm normal fortgesetzt

- **Zulassen und Verbieten von Unterbrechungen** (prozessorabhängig)
  - Setzen eines Bits im Unterbrechungs**anforderungs**register (IRR)
  - Maskierung über Unterbrechungs**erlaubnis**register (IER)
  - Annahmereihenfolge ist festgelegt, z.B. über Prioritäten

Unterbrechungen - wie funktioniert's ?

- **Interrupt-Handler-Routine**
  $\Rightarrow$ Unterbrechungsbehandlungsroutinen liegen im BS-Kern

- **Bei Annahme einer Unterbrechung...**
  - Umschalten in den SYSTEM-Modus
  - sofort weitere Unterbrechungen sperren ($\Rightarrow$ WIESO ?)
  - Maschinenzustand retten:
    - ◇ Statusregister
    - ◇ allgemeine Register
- **Beim Beenden ...**
  - durch speziellen Maschinenbefehl, bspw. RTI (Return from Interrupt)
  - Register zurückladen
    - ◇ Unterbrechungssperre aufheben
    - ◇ Rücksprung ins Benutzerprogramm, dabei Rückkehr in den USER-Modus
  - nächsten Benutzerbefehl ausführen

- **Wie kommt man in den SYSTEM-Modus?**
  - Es gibt hierfür keinen expliziten Maschinenbefehl! (**Wieso nicht?**)
    - ◇ Stattdessen: $\Rightarrow$ **System-Calls** (von Benutzerprogrammen aus aufrufbare Routinen im BS)
- **Einige POSIX System-Calls zur Prozessverwaltung:**
  (POSIX = Portable Operating System Interface, Unix-ähnliche Systeme)
  - **pid = fork ()**
    - ◇ erzeugen eines identischen Kind-Prozesses
    - ◇ Rückgabewert pid == 0 im Kind-Prozess, neue PID sonst
  - **pid = waitpid (pid, &statloc, options)**
    - ◇ warten bis Kind-Prozess zu Ende ist
    - ◇ irgendein Kind: pid = –1, **statloc** enthält exit-Status
  - **s = execve (name, argv, environp)**
    - ◇ ausführen eines Programms aus einer Datei
  - **exit (status)**
    - ◇ Prozess beenden, exit-Status setzen
  - **s = kill (pid, signal)**
    - ◇ Signal an einen Prozess senden

- Beispielprogrammstück einer *Shell* in **C**, das immer wieder neue Prozesse erzeugt:

```
while (1) {
  read_command (&command, &parameters);
  if (fork() != 0)    /* parent code */
     waitpid (-1, &status, 0);
  else                /* child code */
     execve (command, parameters, 0);
}
```

Wie funktioniert das?

- **Einige System-Calls des Dateisystems (nicht vollständig):**
  - open, read, write, stat, lseek, close, chmod
  - mkdir, chdir, rmdir
  - link, unlink
  - mount, umount

PROZESSVERWALTUNG

Begriffe

Sequenzielles Programm

- Ein sequenzielles Programm besitzt genau einen **Kontrollfluss**.
  - Für zwei (beliebige) Anweisungen `bx` und `by` des Quellprogramms gilt:
    Entweder `bx` wird vor `by` ausgeführt oder `by` vor `bx`.
  - Es ist nicht vorgesehen, dass `bx` und `by` parallel ausgeführt werden, z.B. auf einer Mehrprozessormaschine.

- Viele Programmiersprachen erlauben ausschließlich die Formulierung sequenzieller Programme:
  - C, Pascal, ...

Nebenläufiges Programm

- Auf der Ebene der Programmiersprache werden mehrere Kontrollflüsse definiert.
  - Die Kontrollflüsse dürfen parallel ausgeführt werden.
- Auf einer Mehrprozessorarchitektur können die Kontrollflüsse unterschiedlichen Prozessoren zugeordnet und damit echt **parallel** ausgeführt werden.
- **Nebenläufigkeit** eines Programms:
  - die **maximale Parallelität**, die mit unbegrenzt vielen Prozessoren erreichbar ist
- **Parallelität** eines Programms:
  - der tatsächlich erreichte Grad an paralleler Ausführung
  - beschränkt durch die Anzahl der vorhandenen Prozessoren
- **Spezifikation** mittels
  - Programmiersprache mit entsprechenden Sprachkonstrukten
    - ◇ z.B. Java $\rightarrow$ Klasse `Thread`, Schnittstelle `Runnable`
  - sequenzieller Programmiersprache zusammen mit einer speziellen Bibliothek
    - ◇ z.B. C mit *Threads Library* von Sun für Solaris

Sequenzieller Prozess

- Ein sequenzieller Prozess stellt die **Ausführungsumgebung** für ein sequenzielles Programm zur Verfügung.
- Wichtige Bestandteile der Ausführungsumgebung:
  - ein **Adressraum**
  - das (sequenzielle) **Maschinenprogramm**, das im Adressraum abgelegt ist
  - ein **Aktivitätsträger**, der das Programm ausführt $\rightarrow$ **Prozessorkern**
  - eine Liste der **belegten Betriebsmittel** (z.B. geöffnete Dateien)

Nebenläufiger Prozess

- Ein nebenläufiger Prozess stellt die **Ausführungsumgebung** für ein nebenläufiges Programm zur Verfügung.
- Im Gegensatz zum sequenziellen Prozess können mehrere Aktivitätsträger zur Verfügung gestellt werden.

Implementierung von Prozessen

Das Zustandsmodell

- Abb. PRV–1 zeigt das **Zustandsübergangsdiagramm** für Prozesse.

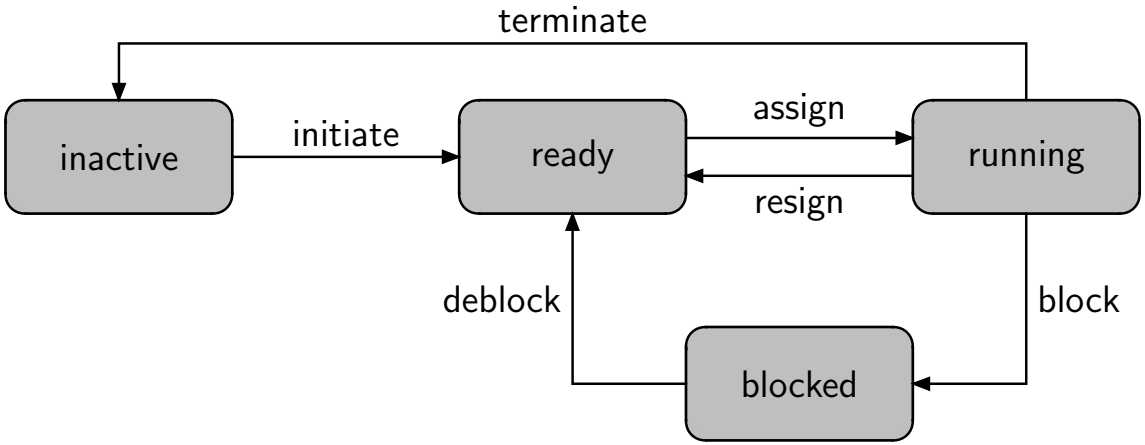


Abb. PRV–1 Zustandsübergänge von Prozessen

Zustände

- **laufend** (*running*):
  - Dem Prozess ist ein (realer) Prozessor zugeordnet.
- **bereit** (*ready*):
  - Der Prozess beteiligt sich am Wettbewerb um einen Prozessor.
    - ◇ Der Prozess besitzt alle Betriebsmittel, die er zum Ablaufen benötigt (außer Prozessor).
- **blockiert** (*blocked*):
  - Der Prozess ist nicht ablaufbereit.
    - ◇ Der Prozess wartet auf den Eintritt eines Ereignisses.
  - Es gibt viele unterschiedliche Blockiertzustände (einen pro Ereignis, auf das gewartet wird).
- **inaktiv** (*inactive*):
  - Dem Prozess ist kein Programm zugeordnet.

- Es liegt ein **geschlossenes** Modell vor, in dem Prozesse als Objekte auftreten.
- Zu jedem Zeitpunkt befindet sich ein Prozess in **genau einem** Zustand.

Übergangsfunktionen

- **Blockieren**: `block ( q )`
  - Der laufende Prozess wird blockiert.
  - Er wird in die Warteschlange `q` eingefügt.
- **Deblockieren**: `deblock ( q )`
  - *Einen* der in `q` blockierten Prozesse auswählen.
  - Überführung in den **bereit**-Zustand.
  - *Auswahlstrategie ?*
- **Zuordnen**: `assign ()`
  - *Einen* zum Ablauf bereiten Prozess auswählen.
  - Zuordnung des Prozessors.
  - *Auswahlstrategie ?*
- **Abgeben**: `resign ()`
  - Den laufenden Prozess in den **bereit**-Zustand überführen.

- **Initiieren**: `initiate ( context )`
  - Einen inaktiven Prozess auswählen.
  - Initialisierung der Prozessumgebung mit den Werten aus `context`.
  - Überführung in den **bereit**-Zustand.

- **Terminieren**: `terminate ()`
  - Der laufende Prozess gibt den Prozessor ab, da die Programmausführung beendet ist.
  - Überführung in den **inaktiv**-Zustand.

- `block`, `resign` und `terminate` sind **selbstbezogen**.

  $\Rightarrow$ nach ihrer Ausführung ist dem Prozessor logisch kein Prozess mehr zugeordnet
- Auf eine derartige Operation **muss immer sofort** ein `assign` folgen.

Die Betriebssystemkomponente Prozessverwaltung

Dispatcher und Scheduler

- Einordnung des Prozesszustandsmodells
  - Es gehört zur untersten Betriebssystemschicht $\rightarrow$ elementarer Dienst des **Kerns**
- Wichtige Bestandteile der Prozessverwaltung:
  - **Dispatcher**
    - ◇ Durchführung der Zustandsübergänge
    - ◇ Implementierung der Übergangsfunktionen
  - **Scheduler**
    - ◇ Aufruf beim Übergang von **ready** nach **running**
    - ◇ Auswahl des nächsten Prozesses, der den Prozessor erhält

Prozesskontrollblock

- Innerhalb des Dispatchers wird ein Prozess durch den **Prozesskontrollblock** (*Prozessleitblock*, *Process Control Block*, PCB) repräsentiert.

- Was steht im Prozesskontrollblock ?
  - Prozesszustand
  - Sicherungsbereich für Registerinhalte
  - Adressrauminformationen
  - Scheduling-Informationen
  - belegte Betriebsmittel (z.B. geöffnete Dateien)
  - weitere Verwaltungsdaten

- Wichtige Operationen auf einem Prozesskontrollblockobjekt ($\rightarrow$ siehe Prg. PRV–1 (S. 10)).

```
public abstract class Pcb {
  // Konstantendefinitionen für Prozesszustände
  public static final int INACTIVE = 1;
  public static final int READY    = 2;
  public static final int RUNNING  = 3;
  public static final int BLOCKED  = 4;

  int state = INACTIVE;
  // Sicherungsbereich fuer Register
  // weitere Attribute eines Prozesses

  public abstract void storeRegister ();       // Sichern der Register

  public abstract void loadRegister ();        // Zurückladen der Register

  public abstract void initPcb ( Context c ); // Initialisierung des PCBs
} // Pcb
```

Prg. PRV–1 Operationen auf einem Prozesskontrollblock

- `storeRegister()`
  - Abspeichern der aktuellen Registerwerte innerhalb des Sicherungsbereichs des PCBs.
  - Hierzu gehören insbesondere auch alle Register, auf die nur im **privilegierten Modus** des Prozessors zugegriffen werden kann.
  - Diese Operation wird oft durch einen speziellen Prozessorbefehl unterstützt.

- `loadRegister()`
  - Überschreiben der Registerinhalte des Prozessors mit den im PCB gesicherten Werten.
  - Diese Operation wird oft durch einen speziellen Prozessorbefehl unterstützt.

- `initPcb ( Context c )`
  - Der Prozesskontrollblock wird mit allen notwendigen Informationen initialisiert.
  - Danach kann ein Zustandsübergang von **initiate** nach **ready** erfolgen.
  - Ein Objekt der Klasse `Context` enthält alle dafür benötigten Informationen.

- Üblicherweise wird bei der Prozesserzeugung ein Dateiname spezifiziert.
- Diese Datei enthält dann das Maschinenprogramm und alle Informationen, die für den Aufbau des `Context`-Objekts notwendig sind (z.B. die Startadresse des Maschinenprogramms).

Prozesswarteschlangen

- **Problem:** Wie wird auf einen Prozess Bezug genommen ?
  - Durch eine eindeutige PID (*Process ID*)
  - Innerhalb der Prozessverwaltung wird die **Referenz** auf den PCB verwendet, der dem Prozess zugeordnet ist.

- Prg. PRV–2 zeigt die Operationen auf einer Prozesswarteschlange.

```
public abstract class ProcessQueue {
  // Definition einer geeigneten Datenstruktur

  public abstract void put ( Pcb process );
  public abstract Pcb  get ();
} // ProcessQueue
```

Prg. PRV–2 | Operationen auf einer Prozesswarteschlange

- `put ( Pcb process )`
  - Der Prozess wird an das Ende der Warteschlange angehängt.

- `Pcb get ()`
  - Der Prozess, der vorne in der Warteschlange steht, wird zurückgeliefert ($\rightarrow$ FIFO).

- Implementierung der Prozesswarteschlangen ($\rightarrow$ siehe Abb. PRV–2 (S. 14)):
  - Jeder Prozess befindet sich (zu jedem Zeitpunkt) in genau einer Warteschlange (und damit in genau einem Zustand).
  - Die verschiedenen Warteschlangen können durch eine **Verkettung** der PCBs realisiert werden.
  - Außerhalb der PCBs ist dann pro Warteschlange nur noch ein **Warteschlangenkopf** notwendig, der auf das erste Element der Warteschlange zeigt (oder bei leerer Warteschlange $null$ enthält).

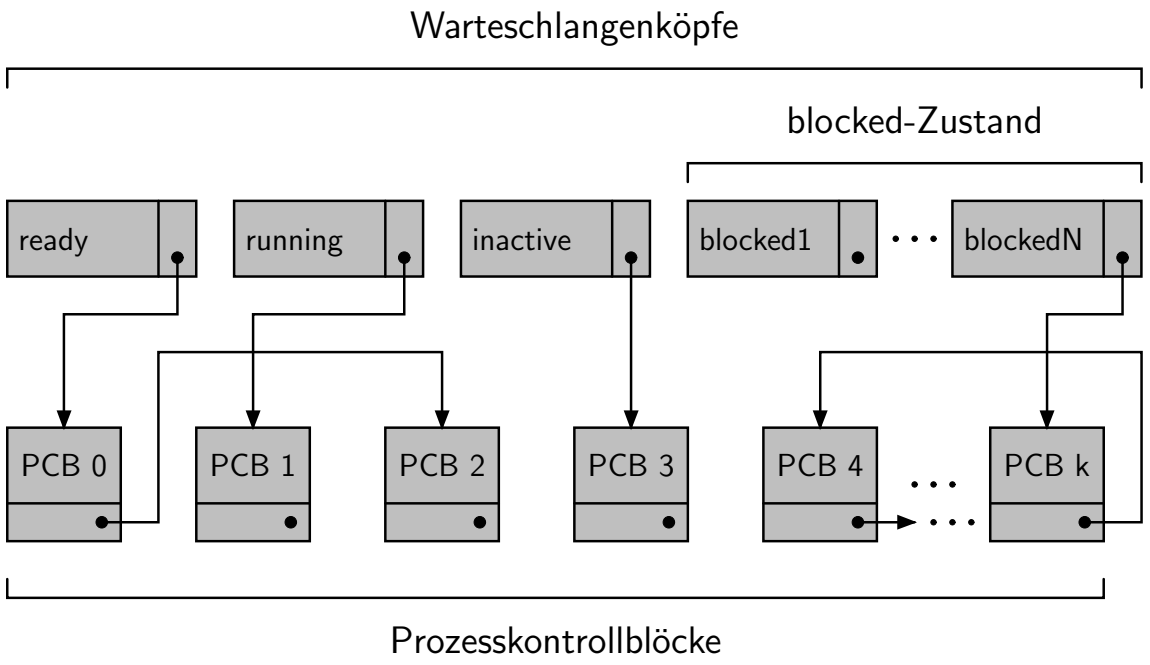


Abb. PRV–2 Implementierung der Prozesswarteschlangen

- Es gibt Scheduling-Strategien, die mehrere `ready`-Warteschlangen benötigen.

  ($\Rightarrow$ z. B. für Multiprozessor-Systeme)

  – Analog zum **blocked**-Zustand kann der **ready**-Zustand dann in Unterzustände zerlegt werden, denen jeweils eigene Warteschlangen zugeordnet sind.

Scheduler

- Die Operation `schedule` (vgl. Prg. PRV–3) wählt aus der `ready`-Warteschlange einen Prozess aus.

```
public abstract class Scheduler {
        // Realisierung der Scheduling-Strategie
  public abstract Pcb schedule ();
} // Scheduler
```

Prg. PRV–3 Die abstrakte Klasse `Scheduler`

Basismechanismen zur Realisierung des gegenseitigen Ausschlusses

Probleme mit nebenläufigem Code

- Nebenläufiger Zugriff z. B. auf unsere Prozesswarteschlangen ist gefährlich!

```
class ProcessQ {
        public Process head;

        public ProcessQ () {
                head = null;
        }

        public void put (Process newP) {
                if ( head == null)
                        head = newP;
                else {
                        Process p = head;
                        while (p.next != null)
                                p = p.next;
                        p.next = newP;
                }
        }
        public Process get () {
                Process p = head;
                if (p != null) {
                        head = p.next;
                        p.next = null;
                }
                return p;
        }
} // ProcessQ

class Process {
        public int pid;
        public  Process next = null;
        // ...

        public Process (int id) {
                pid = id;
        }
} // Process
```

Wo kann es hier zu Problemen kommen?

- Es kommt zu zahlreichen Problemen, wenn mehrere Prozesse gleichzeitig auf eine Prozesswarteschlange zugreifen.

  - Beispiel 1: P1 und P2 rufen gleichzeitig $q.get()$ auf.
    - ◇ Beide Prozesse ermitteln gleichzeitig das erste Element.
    - ◇ Beide Prozesse entnehmen dasselbe Element. $\rightarrow$ Fehler!

  - Beispiel 2: P1 und P2 rufen gleichzeitig $q.put(...)$ auf.
    - ◇ Beide Prozesse ermitteln das letzte Element mit dem null-Zeiger.
    - ◇ Beide Prozesse hängen an dieses Element ihr neues Element an.
    - ◇ Nur das zuletzt angehängte Element befindet sich anschließend in der Warteschlange. $\rightarrow$ Fehler!

  $\Rightarrow$ Die gefährlichen (kritischen) Codeabschnitte müssen unter **gegenseitigem Ausschluss** ausgeführt werden.

Nebenläufige Zugriffe auf gemeinsame Datenstrukturen müssen immer durch eine geeignete Synchronisation geschützt werden.

Unterbrechungssperren

- Unterbrechungen werden während der Ausführung des kritischen Abschnittes verboten.
  - bei Einprozessorsystemen oft benutzt und korrekt
    - ◇ Prozess ist dann nicht unterbrechbar (und der Prozessor wird auch nicht freiwillig abgegeben)
      ⇒ kritischer Abschnitt wird am Stück ausgeführt
  - bei Mehrprozessorsystemen nicht ausreichend !
    ⇒ Auf anderen Prozessoren könnten weitere ausgeschlossene kritische Abschnitte ausgeführt werden.

- Nachteile
  - Herabgesetzte Reaktionsfähigkeit auf externe Unterbrechungen
    - ◇ Mögliche Folge: Verlust von Daten bei E/A-Operationen
  - Unterbrechungen können nur im **privilegierten Zustand** gesperrt werden. ⇒ **System Call** ?!
  - Fehleranfälligkeit:
    - ◇ **Vergisst** ein Prozess, die Unterbrechungen wieder zuzulassen, führt dies zu einer Monopolisierung des Prozessors.

- Einsatz dieser Technik:
  - Bei Einprozessorsystemen für die Realisierung von (zeitlich) kurzen kritischen Abschnitten innerhalb des Betriebssystemkerns.

Atomare Speicheroperationen

- Beobachtung:
  Das Abspeichern eines Wertes in den Hauptspeicher ist atomar.
  - Falls mehrere Prozessoren gleichzeitig versuchen, je einen Wert in dasselbe Speicherwort zu speichern, trifft die Hardware die Entscheidung, welche Speicheroperation zuerst ausgeführt wird.
    ($\rightarrow$ Busarbitrierung)

- Dekker hat als erster eine korrekte Lösung für das Problem des gegenseitigen Ausschlusses gefunden, die alleine auf der Atomarität der Speicheroperationen beruht.

- Prg. PRV–4 (S. 20) enthält eine erheblich vereinfachte Version, die von Peterson (vgl. [SG98]) vorgeschlagen wurde.

```
public class Dekker { // Mutual Exclusion für zwei Prozesse mit IDs 0 und 1
  private static boolean[] need = { false , false };
  private static volatile int turn;  // globale Var.

  public static void enterMutex ( int myProcessID ) {
    int otherPID = 1 - myProcessID;
    need[myProcessID] = true;              // Prozess will in den kA
    turn = otherPID;                    // nächstes Mal darf der andere
    while ( need[otherPID] && turn == otherPID ); // ggf. warten
  } // enterMutex

  public static void exitMutex ( int myProcessID ) {
    need[myProcessID] = false;
  } // exitMutex
} // Dekker
```

Prg. PRV–4 | Dekkers Algorithmus in der Version von Peterson – Teil 1

- Ein möglicher Ablauf:

```
enterMutex( 0 );
    need[ 0 ] = true;


    turn = 1;


    while ( need[ 1 ] &&
                turn == 1);
```

```
enterMutex( 1 );
    need[ 1 ] = true;


    turn = 0;


    while ( need[ 0 ] &&
                turn == 0);
```

Frage: Welcher Prozess darf den k.A. betreten?

- Bewertung
  - \+ Es funktioniert!
  - – Die Lösung ist zunächst auf zwei Prozesse und bestimmte Prozessidentifikationen beschränkt.
    - ◇ Eine Lösung für $n > 2$ Prozesse existiert, ist aber deutlich komplizierter.
  - – Aktives Warten (*Busy Waiting*) wird verwendet.
    - ◇ Dies ist nur für sehr kurze kritische Abschnitte akzeptabel.

- Dekkers Algorithmus ist (nur) historisch interessant, weil er die Existenz eines Verfahrens zur Realisierung des gegenseitigen Ausschlusses ohne die Verwendung spezieller Hardware-Befehle gezeigt hat.
- Eine praktische Anwendung ist aus den genannten Gründen ausgeschlossen!

Spezielle Hardware-Befehle

- Prg. PRV–5 (S. 24) zeigt die Struktur eines `Test&Set`-Befehls.
  - Das Lesen und nachfolgende Schreiben des Speicherwortes (`busy`) wird **atomar** durchgeführt.
  - Jeder moderne Mikroprozessor besitzt mindestens einen Befehl, der nach diesem Prinzip arbeitet.

```
public class GlobalVariable {          // realisiert Referenzübergabe in Java
  public boolean value = false;
} // GlobalVariable

public class SyncStatement {
  // busy bezeichnet die zur Synchronisation benutzte globale Variable
  public static boolean testAndSet ( GlobalVariable busy ) {
    // Die folgenden beiden Anweisungen werden atomar durchgeführt:
      boolean local = busy.value;
      busy.value = true;
    return local;
  } // testAndSet
} // SyncStatement
```

Prg. PRV–5 Der Test&Set-Befehl

- Prg. PRV–6 zeigt die Realisierung des gegenseitigen Ausschlusses mittels des Test&Set-Befehls

```
public class MutualExclusion {
// Eintrittsprotokoll für einen kritischen Abschnitt
  public static void enterMutex ( GlobalVariable busy ) {
    boolean local;
    do {
       local = SyncStatement.testAndSet ( busy );
    } while ( local == true );
  }

// Austrittsprotokoll aus dem kritischen Abschnitt
  public static void exitMutex ( GlobalVariable busy ) {
    busy.value = false;
  }
} // MutualExclusion
```

Prg. PRV–6 | Gegenseitiger Ausschluss mit dem Test&Set-Befehl

- Ein möglicher Ablauf:

Prozess 0:

```
enterMutex(busy);
spin:
    [[local=busy, busy=true;]]

    if ( local==true )
        GOTO spin;
```

```
exitMutex(busy);
    busy=false;
```

Prozess 1:

```
enterMutex(busy);
spin:
    [[local=busy, busy=true;]]

    if ( local==true )
        GOTO spin;
```

```
exitMutex(busy);
    busy=false;
```

Frage: Welcher Prozess darf hier den k.A. betreten?

- Vorteile:
  - Eintritts- und Austrittsprotokoll sind unabhängig von der Prozessanzahl und Prozessidentifikationen.
  - Durch die Parametrisierung mit einer globalen Variablen kann der gegenseitige Ausschluss auf bestimmte kritische Abschnitte eingeschränkt werden.
    - ◇ Beachte: Bei der Verwendung von Unterbrechungssperren ist eine derartige Differenzierung nicht möglich.

- Nachteile:
  - **Aktives Warten** (*Busy Waiting*)
  - Ein **Aushungern** (*Starvation*) eines Prozesses kann auftreten, wenn er ständig von anderen Prozessen beim Betreten des kritischen Abschnittes überholt wird.

- Verwendung dieser Technik:
  - Grundlage für eine Realisierung des gegenseitigen Ausschlusses kurzer kritischer Abschnitte bei **Mehrprozessorsystemen**. ($\rightarrow$ **Spin Locks**)
    - ◇ Diese werden zur Realisierung mächtigerer Synchronisationsmechanismen verwendet, die bei langen kritischen Abschnitten eingesetzt werden.

Realisierung des gegenseitigen Ausschlusses für Einprozessorsysteme

- Prg. PRV–7 realisiert `enterMutex` und `exitMutex` für ein Einprozessorsystem.
  - `interruptsOff` und `interruptsOn` sind (**privilegierte**) Befehle des Prozessors zum An- und Abschalten der Unterbrechungen.

```
private void enterMutex () {
  Processor.interruptsOff();
} // enterMutex

private void exitMutex () {
  Processor.interruptsOn();
} // exitMutex
```

Prg. PRV–7 | Gegenseitiger Ausschluss

Gegenseitiger Ausschluss auf einer Mehrprozessormaschine

- Verwendung von **Spin Locks** → vergl. Prg. PRV-9 (S. SPI-18)
- Zusätzlich müssen jedoch die Unterbrechungen gesperrt werden, um die Gefahr einer **Endlosschleife** zu vermeiden.
- Hier tritt die Endlosschleifenproblematik auf:
  - Ein Prozess hat eine Schnittstellenoperation der Prozessverwaltung aufgerufen.
    ⇒ Gegenseitiger Ausschluss liegt vor.
  - Der Interrupt-Handler der Prozessverwaltung zum Prozesswechsel wird aufgerufen.
  - Da dieser ebenfalls mit `enterMutex` und `exitMutex` geklammert ist, bleibt er am `Test&Set`-Befehl hängen.
  - Die Operation, die den gegenseitigen Ausschluss hergestellt hat, kann nicht beendet werden!

`enterMutex` und `exitMutex` sind lokale Methoden der Prozessverwaltung
⇒ **in Anwendungsprogrammen nicht verwendbar**!

Implementierung der Prozessverwaltung

Für eine übersichtliche Beschreibung, gelten die folgenden vereinfachenden Annahmen:

- Es existiert nur eine einzige `ready`-Warteschlange.
- Alle Operationen der Prozessverwaltung werden innerhalb der Klasse `ProcessManagement` beschrieben.
  $\Rightarrow$ Alle Operationen können direkt auf die Datenstrukturen der Prozessverwaltung zugreifen.

Struktur der Prozessverwaltung

- Operationen des Dispatchers:
  - Die Zustandsübergangsfunktionen sind **interne** Operationen der Prozessverwaltung.
    ⇒ Prozesse können nicht direkt auf diese Operationen zugreifen (→ Geheimnisprinzip).

- **Schnittstellenoperationen** der Prozessverwaltung (d.h. nach außen sichtbare *System Calls*) sind z. B.:
  - Erzeugung/Vernichtung von Prozessen
  - Semaphoroperationen
  - Ein-/Ausgabeoperationen

- Synchronisationskonzept:
  - Alle Schnittstellenoperationen der Prozessverwaltung werden unter **gegenseitigem Ausschluss** ausgeführt.
    ⇒ Konsistenz der internen Datenstrukturen (z. B. Warteschlangen) ist sichergestellt.

- **Unterbrechungsbehandlungsroutinen** (*Interrupt-Handler*):
  - Integration als Schnittstellenoperationen der Prozessverwaltung
    ◇ Sie dürfen jedoch nicht direkt von Anwendungsprogrammen aufgerufen werden!
  - Damit werden diese Operationen unter gegenseitigem Ausschluss ausgeführt.

Implementierung der Dispatcher-Operationen

- Prg. PRV–8 bis Prg. PRV–10 (S. 34) enthalten eine Implementierung der Zustandsübergangsfunktionen aus Abb. PRV–1 (S. 4).

```
public class ProcessManagement {
  private Pcb[] processes;
  private Pcb   runningProcess;
  private ProcessQueue readyQueue;
  private ProcessQueue inactiveQueue;
  private Scheduler s;

  public ProcessManagement ( int processCount ) {
    processes = new Pcb[processCount];
    // Erzeugung eines geeigneten Scheduler-Objekts
    // Erzeugung der benoetigten Prozesswarteschlangen
    // Erzeugung der Prozesskontrollbloecke
  } // Konstruktor
```

Prg. PRV–8 Implementierung der Dispatcher-Operationen – Teil 1

```
// Dispatcher-Operationen
  private void initiate ( Context c ) {
    Pcb process = inactiveQueue.get();
    process.initPcb ( c );
    process.state = Pcb.READY;
    readyQueue.put( process );
  } // initiate

  private void terminate () {
    runningProcess.state = Pcb.INACTIVE;
    inactiveQueue.put ( runningProcess );
  } // terminate

  private void block ( ProcessQueue q ) {
    runningProcess.state = Pcb.BLOCKED;
    q.put ( runningProcess );
  } // block
```

Prg. PRV–9 Implementierung der Dispatcher-Operationen – Teil 2

```
  private void deblock ( ProcessQueue q ) {
    Pcb process = q.get();
    process.state = Pcb.READY;
    readyQueue.put ( process );
  } // deblock

  private void resign () {
    runningProcess.state = Pcb.READY;
    readyQueue.put ( runningProcess );
  } // resign

  private void assign () {
    Pcb process = s.schedule ();
    process.state = Pcb.RUNNING;
    runningProcess.storeRegister();
    runningProcess = process;
    runningProcess.loadRegister();
  } // assign
} // ProcessManagement
```

Prg. PRV–10 Implementierung der Dispatcher-Operationen – Teil 3

Analyse der `assign`-Operation

- Wirkungsweise von `assign`:
  - Der Aufruf führt zum **Verlust** des Prozessors für den **aufrufenden** Prozess (`P1`).
  - `P1` läuft weiter, wenn ein anderer Prozess (`P2`) seinerseits `assign` aufgerufen hat und damit `P1` die **Beendigung** seines `assign`-Aufrufes ermöglicht.
  - Die Wartezeit, bis `P1` den Prozessor wieder erhält, ist für `P1` *transparent*.

- Durchführung des Prozesswechsels:
  - Die Ausführung von `loadRegister` stellt den Prozesswechsel dar:
    - ◇ Jeder Prozess besitzt seinen eigenen Stack.
    - ◇ Beim Laden der Register wird der **Stackpointer** überschrieben.
    - ⇒ Der Stack des unterbrochenen Prozesses wird zum **aktuellen** Stack.
    - ⇒ Auf diesem Stack liegt die Rücksprungadresse des `assign`-Aufrufs.
  - **Achtung:** `storeRegister` bzw. `loadRegister` dürfen **nicht** das Befehlszählerregister sichern bzw. zurückladen (→ sonst **Endlosschleife**) !

- Aus der Sicht eines Prozesses stellt `assign` einen **Operationsaufruf** dar.
- Aus der Sicht des Systems führt `assign` einen **Prozesswechsel** aus.

Scheduling

Überblick

Zielsetzung

- Verwaltung des Betriebsmittels **Prozessor**.
- Vermeidung der **Prozessor-Monopolisierung** durch eine Anwendung.
- Unterstützung der verschiedenen **Betriebsarten** eines Betriebssystems:
  - **Dialogbetrieb** ($\rightarrow$ interaktive Anwendungen)
    - ◇ Ziel: schnelle Reaktion auf Benutzereingaben

      $\Rightarrow$ Bevorzugte Auswahl von Prozessen, die nach dem Aufruf einer blockierenden Ein-/Ausgabeoperation durch eine Benutzereingabe wieder ausführbar sind.
  - **Stapelbetrieb** ($\rightarrow$ *Batch-Betrieb*)
    - ◇ Ziel: Hoher Durchsatz sowie hohe Auslastung der Hardware-Ressourcen.

      $\Rightarrow$ Bevorzugte Auswahl von Prozessen, die gerade freie Betriebsmittel benötigen.
  - **Echtzeitbetrieb**
    - ◇ Ziel: Einhaltung von Zeitgarantien

      $\Rightarrow$ Bevorzugte Auswahl der Prozesse, deren Ausführungsfristen ablaufen.

Einsatzgebiete

- **Kurzzeitablaufsteuerung** (→ *Short-Term-Scheduling*)
  - innerhalb der Prozessverwaltung:
    - ◇ Welcher Prozess soll den Prozessor erhalten ?
    - ◇ Aufruf des Schedulers (→ `assign`) in kurzen Zeitabständen (ca. 10 bis 100 ms)
- **Langzeitablaufsteuerung** (→ *Long-Term-Scheduling*)
  - **Swapping** (z.B. Unix).
    - ◇ Neue Aufträge werden sofort zu Prozessen.
    - ◇ Bei Überlastzuständen wird ein Prozess ausgewählt, dessen komplette Prozessumgebung auf die Festplatte ausgelagert wird.
  - **Auftragsverwaltung** (→ bei Großrechnerbetriebssystemen)
    - ◇ Neue Benutzeraufträge werden zunächst in eine **Auftragswarteschlange** eingefügt.
    - ◇ Aus dieser werden je nach Last Aufträge ausgewählt und Prozessen zugeordnet.
    - ◇ Bei **Überlastzuständen** werden Aufträge zur **Verdrängung** ausgewählt.
      - ▷ Der gesamte Zustand (insb. der Adressraum) wird auf Platte ausgelagert.
      - ▷ Der Auftrag kommt zurück in die Auftragswarteschlange.

Mögliche Scheduling-Kriterien

- **Prozessorauslastung**: Maß für die Auslastung eines Prozessors durch die Ausführung von Befehlen der Anwendungsprogramme.
- **Durchsatz**: Anzahl der pro Zeiteinheit fertiggestellten Aufträge.
- **Wartezeit**: Gesamtverweildauer eines Prozesses in der `ready`-Warteschlange.
- **Umlaufzeit** (*Turnaround Time*): Zeit zwischen zwei Aktivierungen desselben Prozesses.
- **Antwortzeit**: Zeit zwischen der Ankunft einer Benutzereingabe und einer Prozessorzuteilung an den zugehörigen Prozess.
- **Echtzeitfähigkeit**: Einhaltung der von Anwendungen definierten Echtzeitvorgaben.
- **Leistungsfähigkeit** der Implementierung des Scheduling-Verfahrens:
  - Die Zeitanteile zur Ausführung der `schedule`-Operation gehen den Anwendungsprogrammen verloren.

Kein Scheduling-Verfahren kann alle Anforderungen erfüllen!

Klassifikation von Scheduling-Verfahren

- **Kooperatives Scheduling** (*Cooperative Scheduling*)
  - Die `schedule`-Operation kann nur ausgeführt werden, wenn ein Prozess von sich aus den Prozessor abgibt.
  - Die System-Software kann nicht gezielt die Kontrolle übernehmen, um einem Prozess den Prozessor zu entziehen.

- Läuft ein Prozess in einer Endlosschleife, liegt eine Prozessormonopolisierung vor.
- Dennoch kann die System-Software die Kontrolle zurückerhalten, z.B. bei Platten-Interrupt
  $\rightarrow$ Behandlung innerhalb des Betriebssystemkerns.

- **Verdrängendes Scheduling** (*Preemptive Scheduling*)
  - Die System-Software kann einem Prozess den Prozessor zu entziehen (ohne dass ihn dieser von sich aus abgibt).

Scheduling-Verfahren

First-Come-First-Served (FCFS)

- Auswahl des Prozesses, der bereits am längsten im `ready`-Zustand ist.
- Keine Verdrängung $\rightarrow$ der Prozess läuft, bis er sich blockiert oder terminiert.
- Einfache Implementierung $\rightarrow$ Verwendung einer FIFO-Warteschlange.
- Bei der Auswahl des nächsten Prozesses wird insb. die **aktuelle Systemauslastung** nicht berücksichtigt.
  - Ein rechenzeitintensiver Prozess kann E/A-intensive Prozesse daran hindern, ihre E/A-Aufträge anzustoßen.
    $\Rightarrow$ Schlechte Auslastung der E/A-Geräte (z.B. Festplatte).
  - Später liegen dann vorwiegend E/A-intensive Prozesse vor.
    $\Rightarrow$ Nun sammeln sich die E/A-Aufträge vor den E/A-Geräten ($\rightarrow$ *Konvoi-Effekt*).
    $\Rightarrow$ Jetzt ist die Prozessorauslastung sehr schlecht.

Shortest-Processing-Time-First (SPTF)

- Auswahl des Prozesses mit der kürzesten Rechenzeit bis zum nächsten `blocked`/`inactive`-Zustand ($\rightarrow$ **kürzester CPU-Burst**).
- Strategie minimiert die mittlere Gesamtwartezeit der Prozesse.
- Problem: Wie wird der CPU-Burst abgeschätzt ?
- Die Strategie wird auch als **Shortest-Job-First** (SJF) bezeichnet.

Shortest-Remaining-Time-First (SRTF)

$\Rightarrow$ Verdrängende Variante der SPTF-Strategie.

- Wird ein anderer Prozess mit einem kürzeren CPU-Burst ablaufbereit, so erhält dieser den Prozessor.
  - Kürzerer CPU-Burst heißt hier, dass dessen CPU-Burst kürzer als die **Restbearbeitungszeit** des CPU-Burst des gerade laufenden Prozesses ist.

Round Robin (RR)

- Ziel: Gleichmäßige Aufteilung der verfügbaren Rechenzeit auf alle ablaufbereiten Prozesse.
- Verwendung des **Zeitscheibenverfahrens**:
  - Die Hardware muss ein **Zeitgeberregister** besitzen.
    - ◇ Laden des Registers durch die Operation `loadTimer (timeSlice)`.
    - ◇ Der Registerinhalt wird automatisch in bestimmten Zeitabständen dekrementiert.
    - ◇ Beim Wert 0 wird eine **Zeitscheibenunterbrechung** (*Timer Interrupt*) erzeugt.
  - Die `assign`-Operation muss um das Laden des Zeitgeberregisters ergänzt werden.
  - Prg. PRV–11 zeigt den Timer-Interrupt-Handler als **Schnittstellenoperation** der PV.

```
public void timerInterruptHandler () {
  enterMutex();
    resign();
    assign();
  exitMutex();
}
```

Prg. PRV–11 Implementierung des Timer-Interrupt-Handler

- Kontrollfluss bei einer Zeitgeberunterbrechung: siehe Abb. PRV–3

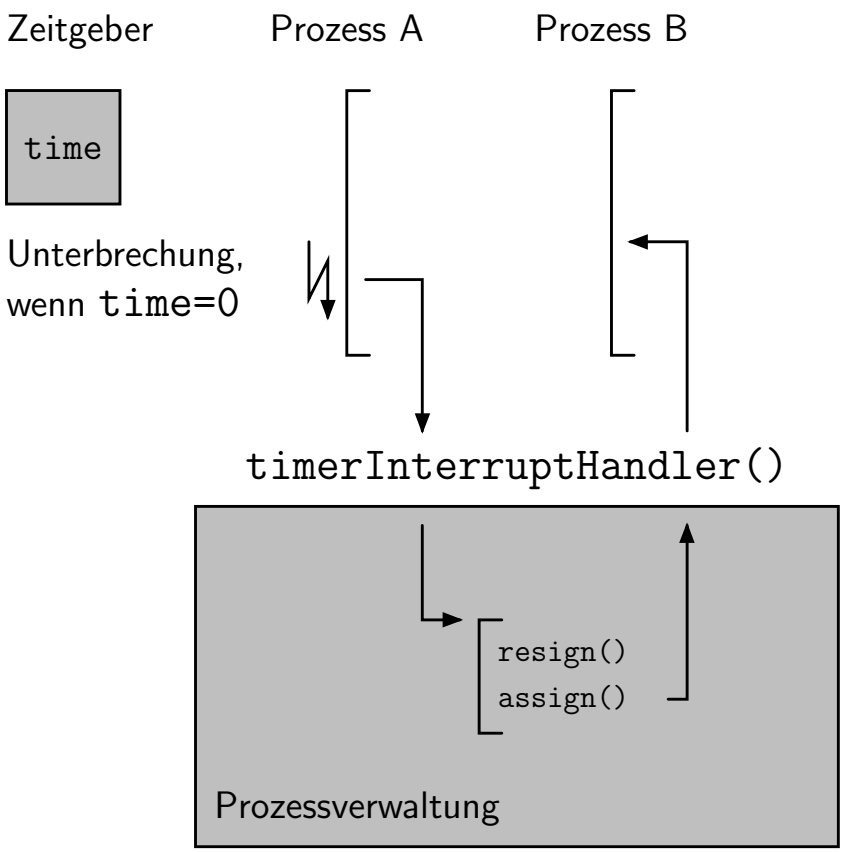


Abb. PRV–3 Kontrollfluss bei einer Zeitgeberunterbrechung

- **Realisierung** des RR-Verfahrens:
  - Gleiches **Zeitscheibenquantum** Q für alle Prozesse
  - Verwendung einer FIFO-Organisation für die `ready`-Warteschlange

- **Welche Wahl für Q ist günstig ?**
  - Q zu groß

    $\Rightarrow$ schlechtes Antwortzeitverhalten
  - Q zu klein

    $\Rightarrow$ hoher Verwaltungsaufwand (viele Prozesswechsel)

    $\Rightarrow$ Wegen der kurzen Prozesslaufzeiten können die Cache-Speicher keine Lokalität aufbauen

    $\rightarrow$ Cache bleibt **kalt**.
  - **Typische Werte für Q:** 10 bis 100 ms

Highest-Priority-First (HPF)

- **Konzept:**
  - Jedem Prozess wird eine Priorität zugeordnet.
  - Kleinere Zahlenwerte repräsentieren dabei üblicherweise eine höhere Priorität.
  - Der Prozess mit der höchsten Priorität erhält den Prozessor.
- **Nicht verdrängende** HPF-**Strategie** ($\rightarrow$ *non preemptive*):
  - Der Prozess läuft, bis er den Prozessor von selbst abgibt.
- **Verdrängende** HPF-Strategie (PHPF) ($\rightarrow$ *preemptive*):
  - Falls ein Prozess mit höherer Priorität in den `ready`-Zustand wechselt, erhält dieser den Prozessor.
- **Statische** Priorität:
  - Die Priorität von Prozessen wird bei Erzeugung festgelegt (und ändert sich nicht mehr).
- **Dynamische** Priorität:
  - Die Priorität kann durch das Betriebssystem verändert werden (und in kontrollierter Form auch durch den Anwender).

- **Kriterien für die Prioritätsfestlegung**
  - Statische Prioritäten:
    - ◇ extern vorgegeben
    - ◇ abhängig von geschätzter Prozessorzeit
    - ◇ abhängig von E/A-Nutzung
    - ◇ abhängig von angefordertem Speicher
  - Dynamische Prioritäten:
    - ◇ abhängig von momentaner Nutzung der Betriebsmittel
    - ◇ abhängig von bisheriger Wartezeit
    - ◇ abhängig von verbrauchter Prozessorzeit
    - ◇ abhängig von bisheriger E/A-Aktivität
    - ◇ abhängig von Gesamtverweilzeit im System

- Bei Verwendung einer *HPF*-Strategie ist ein **Aushungern**(*Starvation*) von Prozessen prinzipiell nicht auszuschließen !
- **Gegenmaßnahme:** Verwenden einer **Aging**-Technik.

  → Die Priorität von Prozessen steigt, je länger sie im System sind.

Mehrstufenverfahren

- Definition **kombinierter** Scheduling-Verfahren:
  - *Statische* Mehrstufenverfahren: **Multilevel-Scheduling**
  - *Dynamische* Mehrstufenverfahren: **Multilevel-Feedback-Scheduling**

- Struktur eines Mehrstufenverfahrens:
  - Verwendung **mehrerer** `ready`-Warteschlangen.
  - Bestimmung der **aktuellen** Warteschlange:
    Jede `ready`-Warteschlange erhält eine **Priorität**.
    $\rightarrow$ Auswahl eines Prozesses aus der höchstpriorisierten, nichtleeren Warteschlange.
    ODER: Zwischen den Warteschlangen wird eine **RR-Strategie** benutzt, wobei jede Warteschlange ein ggf. unterschiedliches Zeitquantum zugewiesen bekommt.
  - Jede Warteschlange kann eine **eigene Auswahlstrategie** besitzen.

- **Multilevel-Scheduling**:
  - Prozesse werden beim Erzeugen einer `ready`-Warteschlange zugeordnet.
  - Dort verbleiben sie bis zu ihrer Terminierung.

- Tab. PRV–1 zeigt eine typische Anwendungsstruktur für ein Multilevel-Scheduling.

Priorität	Prozessklasse	Strategie
1	Echtzeitsteuerungsprozesse	(PHPF)
2	E/A-intensive Prozesse	(RR)
3	Interaktionsprozesse	(RR)
4	Rechenintensive Stapelprozesse	(FCFS)

Tab. PRV–1 Multilevel-Scheduling (statisches Mehrstufenverfahren)

- **Multilevel-Feedback-Scheduling**:
  - Prozesse können in Abhängigkeit ihres Verhaltens zwischen den `ready`-Warteschlangen wechseln.

- Ein Beispiel
  - **Struktur:**
    - ◇ Es werden mehrere RR-Warteschlangen benutzt.
    - ◇ Je höher die Priorität einer RR-Warteschlange ist, desto kleiner ist das Zeitscheibenquantum Q, das ein Prozess aus dieser Warteschlange erhält.
  - **Feedback-Verfahren:**
    - ◇ Prozesse, die Q mehrmals komplett ausgenutzt haben, kommen in eine `ready`-Warteschlange mit niedrigerer Priorität aber größerem Wert für Q.
    - ◇ Prozesse, die sich in einer E/A-intensiven Phase befinden ($\rightarrow$ **I/O-Burst**), kommen in eine Warteschlange mit höherer Priorität aber kleinerem Wert für Q.

Fallbeispiel: Scheduling in Unix

- BSD-Unix setzt **Multilevel-Feedback-Scheduling** ein
- **Struktur:** (siehe Abb. PRV–4 (S. 51)):
  - Die `ready`-Warteschlangen werden nach dem RR-Verfahren verwaltet.
  - E/A-intensive Prozesse, die häufig ihr Zeitscheibenqunatum nicht ausnutzen, erhalten eine höhere Priorität.
  - Warteschlangen mit **negativen** Prioritätswerten verwalten Prozesse, die durch den Aufruf einer Systemfunktion im Kern blockiert sind.
    - ◇ Wenn ein derartiger Prozess wieder ablaufbereit wird, erhält er mit sehr hoher Priorität den Prozessor und seine **User-Mode-Priorität** zurück.
  - Anpassung der **User-Mode-Priorität** mittels **Aging**

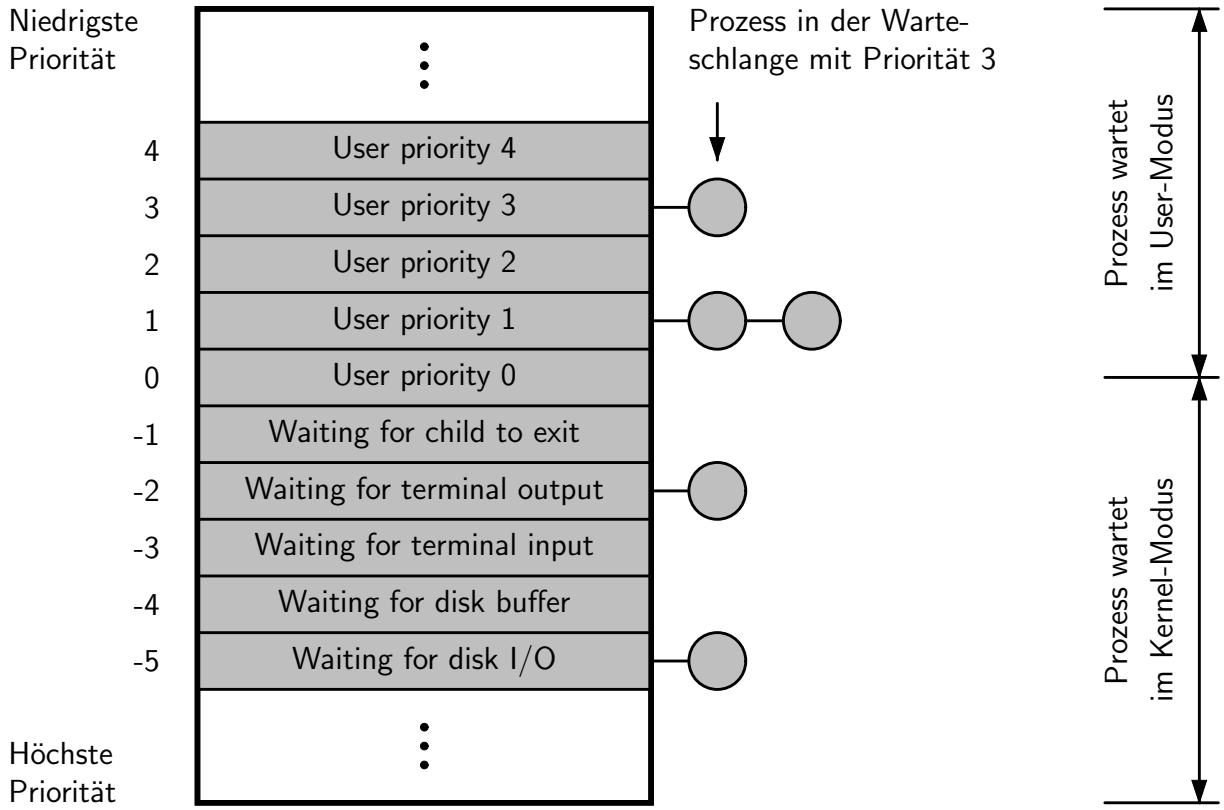


Abb. PRV–4 Scheduling in BSD-Unix

Ein-/Ausgabeoperationen an der Schnittstelle zur Prozessverwaltung

Integration der E/A-Verarbeitung in das Prozesskonzept

- **Aufgabenstellung:**
  - Integration der Steuerung von E/A-Geräten in das Betriebssystem.
  - Bereitstellung eines stark vereinfachten E/A-Modells.
- **Randbedingungen:**
  - Geschwindigkeitsunterschiede zwischen CPU und E/A-Geräten $\Rightarrow$ Faktor 10^3 bis 10^5.
  - Parallelarbeit von CPU und E/A-Geräten notwendig.
- **Lösungskonzept:**
  - Ein-/Ausgabe wird auf Hardware-Ebene fast immer **asynchron** ausgeführt.
  - E/A-Geräte werden als unabhängige Prozessoren aufgefasst.
  - Annahme: Jedes E/A-Gerät kann zu einem Zeitpunkt nur eine E/A-Operation bearbeiten.
  - Prinzipieller Ablauf eines E/A-Auftrags:
    - ◇ E/A-Gerät durch speziellen E/A-Befehl anstoßen.
    - ◇ Eigenständige Operationsausführung des E/A-Gerätes
    - ◇ Meldung des E/A-Endes per Unterbrechung (Interrupt) an die CPU.

- **Integration der E/A-Verarbeitung in das Prozesskonzept:**
  - Prozesse rufen E/A-Operationen auf.
  - Durch die Nebenläufigkeit der Prozesse können E/A-Operationen mehrfach aufgerufen werden – auch wenn das Gerät nur exklusiv nutzbar ist.

- **Realisierungsstruktur:**
  - Verwendung einer Warteschlange für E/A-Aufträge zur Pufferung der angeforderten E/A-Operationen.
  - Sequenzialisierte Abarbeitung der E/A-Aufträge.

Aus Optimierungsgründen muss die Bearbeitungsreihenfolge nicht der Anforderungsreihenfolge entsprechen ($\rightarrow$ z. B. Elevator-Strategie zur Plattensteuerung).

- Schnittstellenvarianten des E/A-Systems aus Anwendungssicht:
  - **synchron**
  - **asynchron**

Synchrone E/A-Schnittstelle

- Ablauf der E/A-Operation aus Prozessverwaltungssicht → Abb. PRV–5.

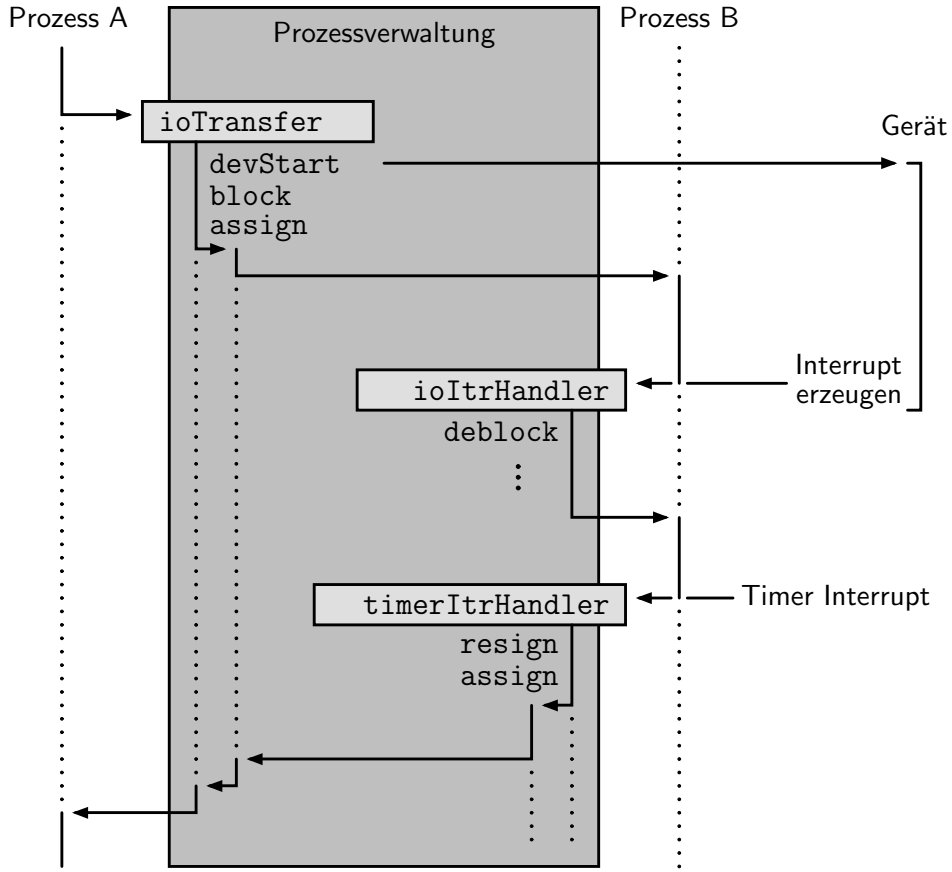


Abb. PRV–5 Ablauf einer synchronen E/A-Operation

- Bemerkungen zu Abb. PRV–5 (S. 54):
  - Der Kontrollfluss ist nur bei Aufrufen der `assign`-Operation detailliert eingezeichnet.
  - Nach Ausführung von `ioItrHandler` könnte auch ein Aufruf des Schedulers erfolgen.
  - Der Aufruf von `ioItrHandler` kann aus Sicht des Prozesses B als **erzwungener Prozeduraufruf** aufgefasst werden.

- Zugeordnete Schnittstellenoperation der Prozessverwaltung: `ioTransfer`
  - Aus Prozess-Sicht ist dies einen normale Prozedur.
  - Prozedurausführung endet, wenn die E/A-Operation beendet ist.

- Der Wartezustand ist für den `ioTransfer` aufrufenden Prozess – und damit auch für das zugeordnete Anwendungsprogramm – **transparent**.
- Durch die **synchrone E/A-Schnittstelle** wird die **asynchrone Geräteschnittstelle** verdeckt !

- Benötigte Warteschlangen in der Prozessverwaltung:
  - **Jedem E/A-Gerät** wird eine eigene `blocked`-Warteschlange zugeordnet.

- Verwaltung von E/A-Aufträgen:
  - Da `ioTransfer` von mehreren Prozessen nebenläufig aufgerufen werden kann, muss für jedes Gerät zusätzlich eine **Auftragswarteschlange** angelegt werden.
  - In `ioTransfer` muss geprüft werden, ob das Gerät frei ist.
    - ◇ Falls ja $\Rightarrow$ Gerät mit `devStart` starten.
    - ◇ Falls nein $\Rightarrow$ Auftrag in der Auftragswarteschlange zwischenspeichern.
  - Der dem Gerät zugeordnete Interrupt-Handler muss dann den nächsten Auftrag auswählen und das Gerät neu starten oder es als **frei** markieren.

Asynchrone E/A-Schnittstelle

- Zugeordnete Schnittstellenoperationen der Prozessverwaltung:
  - `ioStart`
    - ◇ Anstoßen bzw. Anmelden der E/A-Operation.
    - ◇ Der Prozess läuft dann **parallel** zum Gerät weiter.
  - `ioWait`
    - ◇ Der Prozess will feststellen, ob die E/A-Operation beendet ist.
    - ◇ Nur falls die E/A-Operation noch nicht beendet ist, wird der Prozess **blockiert**.

Nach der Rückkehr aus der `ioWait`-Operation ist sichergestellt, dass die E/A-Operation ebenfalls fertig ist!

- Abb. PRV–6 (S. 58) zeigt den Ablauf einer asynchronen E/A-Operation.
  - Wie bei der synchronen E/A-Ausführung könnte ein Scheduler-Aufruf nach der Ausführung von `ioItrHandler` erfolgen.
  - Der Ablauf ist um eine **Auftragsverwaltung** zu erweitern.

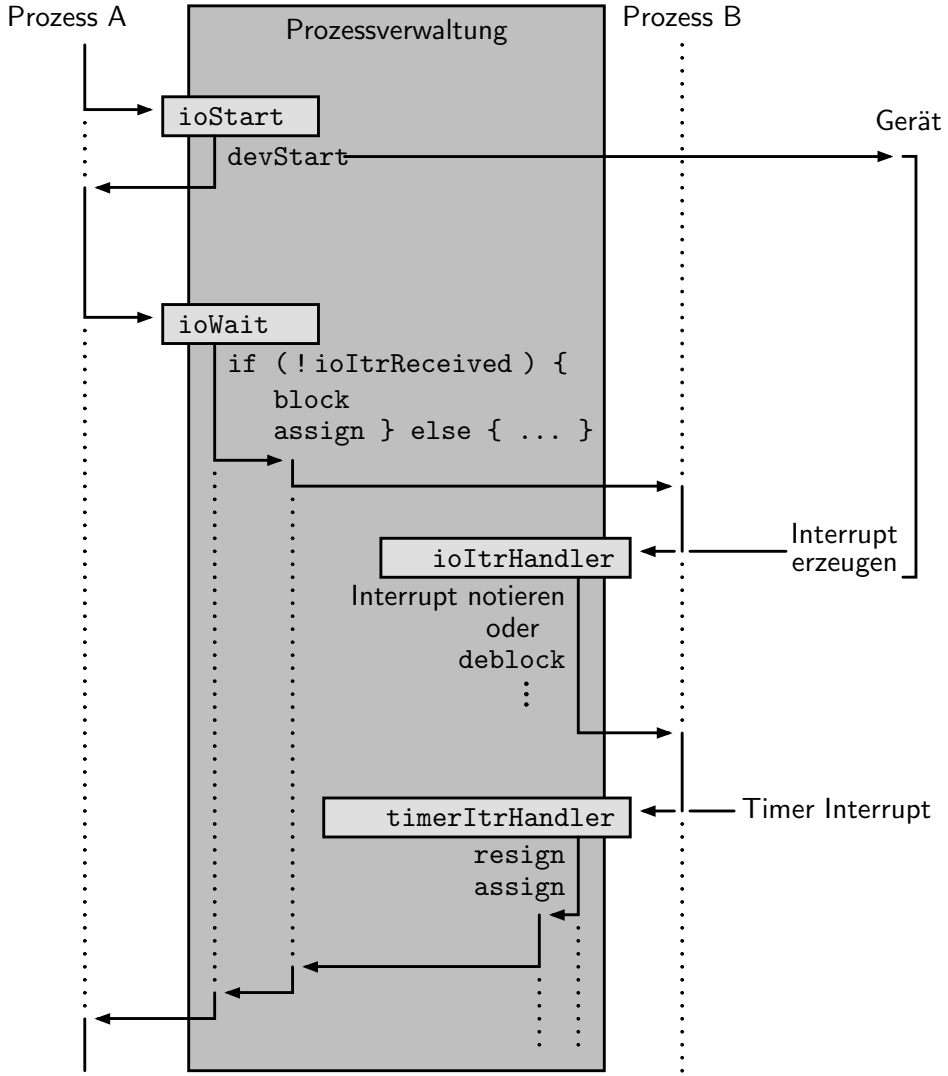


Abb. PRV–6 Ablauf einer asynchronen E/A-Operation

Das Semaphor-Konzept

Semaphor-Definition

- Zur Synchronisation innerhalb und außerhalb des Betriebssystems; von Dijkstra eingeführt.
- **Semaphor**-Definition
  - Objekt, auf dem die beiden Operationen `p` (auch: `up`) und `v` (auch: `down`) existieren.
  - Interne Komponenten eines Semaphors `sem`:
    - ◇ **Wartezustand** für Prozesse (blocked-Queue)
    - ◇ **Wert** des Semaphors: `sem.ctr`
    - ◇ Interpretation von `sem.ctr`
      - ▷ positiv: `sem.ctr` = Anzahl der noch verfügbaren Betriebsmittel
      - ▷ negativ: $|$`sem.ctr`$|$ = Anzahl der wartenden Prozesse
  - Eigenschaften:
    - ◇ `p` und `v` sind **atomar** (Einsatz von Unterbrechungssperren und atomaren Speicheroperationen).
  - Implementierung als Schnittstellenoperation der Prozessverwaltung: siehe Prg. PRV–12 (S. 60)

```
public class Semaphore {   // als Schnittstellenoperation der PV
  private int ctr;
  private ProcessQueue q;  // FIFO
  public Semaphore ( int initValue ) {
    ctr = initValue;
    q = new (ProcessQueue);
  }

  public void p () {                    public void v () {
    enterMutex();                         enterMutex();
    ctr = ctr - 1;                        ctr = ctr + 1;
    if ( ctr < 0 ) {                      if ( ctr <= 0 )
       block (q);    // warten               deblock (q);   // Wartenden wecken
       assign ();                         exitMutex();
    }                                     }
    exitMutex();
  }
} // Semaphore
```

Gegenseitiger Ausschluss

- Gegeben sei eine Menge von kritischen Abschnitten, die unter gegenseitigem Ausschluss bearbeitet werden sollen.

- Verwendung eines Semaphors, das mit 1 initialisiert wird (auch: *binäres Semaphor*)
  ($\rightarrow$ siehe Prg. PRV–13 (S. 62))

  - Jeder der kritischen Abschnitte wird mit einem **Eintritts**- und einem **Austrittsprotokoll** geklammert:
    - $\diamond$ p-Operation: Eintrittsprotokoll
    - $\diamond$ v-Operation: Austrittsprotokoll

```
public class MutualExclusionSemaphore {
  // mutex ist ein Semaphor, das von mehreren Prozessen verwendet werden kann
  // für gegenseitigen Ausschluss wird es mit dem Wert 1 initialisiert
  static Semaphore mutex = new Semaphore (1);
  public static void main ( String[] args ) {
    // ...
    // Prozess führt irgendwelche Aktionen aus;
    // ...
    // Prozess will kritischen Abschnitt betreten;
    mutex.p();                    // Eintrittsprotokoll
      // Prozess befindet sich im
      // kritischen Abschnitt;
    mutex.v();                   // Austrittsprotokoll
    // ...
    // Prozess führt irgendwelche Aktionen aus;
  }
} // MutualExclusionSemaphore
```

Prg. PRV–13 Gegenseitiger Ausschluss mit einem Semaphor

- Ein möglicher Ablauf:

Prozess 0:

```
mutex.p();
    [[ mutex.ctr--;
       if (mutex.ctr < 0)
          warten;    ]]
```

Prozess 1:

```
mutex.p();

                    [[ mutex.ctr--;
                       if (mutex.ctr < 0)
                          warten;    ]]
```

```
mutex.v();
    [[ mutex.ctr++;
       if (mutex.ctr ≤ 0)
          aufwecken;    ]]
```

Beispiel: Betriebsmittelverwaltung

- Problemstellung:
  - Mehrere Prozesse bewerben sich gleichzeitig um einen Pool von exklusiv benutzbaren, gleichwertigen Betriebssmitteln.
- Beispiel:
  - Zu einer Konfiguration gehören `n` Drucker, die hier die Rolle der Betriebsmittel spielen.
  - Die Drucker sind gleichwertig, d.h. welcher Drucker verwendet wird, ist einem Prozess egal.
- Prg. PRV–14 (S. 65) zeigt die abstrakte Klasse `File`, die als Parameter der `print`-Operation verwendet wird. (Die **read**– und die **write**–Operationen müssen in der Unterklasse implementiert werden.)

```
public abstract class File {
  protected byte[] content;

  public File ( int n ) { // Erzeugung einer Datei mit der Anfangslänge n
    content = new byte[n];
  } // Konstruktor

// Schreiboperation; eventuell wird die Datei dabei verlängert
  public abstract void write (int index, byte[] data);

// Leseoperation; Rückgabe: Anzahl gelesener Bytes
  public abstract int read (int index, byte[] data);

  public int length() { // Länge der Datei
    return content.length;
  } // length
} // File
```

Prg. PRV–14 Die abstrakte Klasse `File`

- Prg. PRV–15 enthält die Lösung für das Druckerverwaltungsproblem.

```
public class IdenticalPrinter1 {
  private Semaphore printer;
// Verwaltungsobjekt erzeugen, das n identische Drucker verwaltet
  public IdenticalPrinter1 ( int n ) {
    printer = new Semaphore (n);
  } // Konstruktor
  public void print ( File f ) {// Ausgabe einer Datei auf einem der Drucker
    printer.p();
      // Datei f drucken;
    printer.v();
  } // print
} // IdenticalPrinter1
```

Prg. PRV–15 Zugriff auf identische Drucker

Bei dieser Lösung ist nicht nachvollziehbar, welcher Drucker jeweils verwendet wurde!

- Verbesserung:
  - Verwendung einer **gemeinsam** genutzten Tabelle, um die Zuordnung

    Prozess $\leftrightarrow$ *Drucker*

    vorzunehmen. ($\rightarrow$ siehe Prg. PRV–16 und Prg. PRV–17 (S. 68))

```
public class IdenticalPrinter {
  private Semaphore printer;
  private Semaphore mutex = new Semaphore (1);
  // Definition einer Belegungstabelle

        // Verwaltungsobjekt erzeugen, das n
        // identische Drucker verwaltet
  public IdenticalPrinter ( int n ) {
    printer = new Semaphore (n);
  }
```

Prg. PRV–16 Druckerverwaltung – Teil 1

```
// Ausgabe einer Datei auf einem der Drucker
  public void print ( File f ) {
    printer.p(); // einen Drucker belegen
    mutex.p();
      // Feststellen, welcher Drucker frei ist und belegen
      // durch Eintragen in die Belegungstabelle;
      // Ausgabe des Druckernamens;
    mutex.v();

       // Drucker benutzen;

    mutex.p();
      // Freigeben des Druckers in der Belegungstabelle;
    mutex.v();
    printer.v(); // einen Drucker freigeben
  }
} // IdenticalPrinter
```

Prg. PRV–17 Druckerverwaltung – Teil 2

SPEICHERVERWALTUNG

Organisation des Hauptspeichers

Bereiche fester Länge

- **Konzept:**
  - Speicher wird in Bereiche **fester Länge** (d.h. nicht änderbarer Länge) aufgeteilt (*Partition*, *Region*)
    $\rightarrow$ siehe Abb. SPV–1.
  - Diese Bereiche können unterschiedlich lang sein.
  - Üblicherweise ist die Länge eines Bereiches ein Vielfaches einer Grundeinheit (z.B. 2 KB).

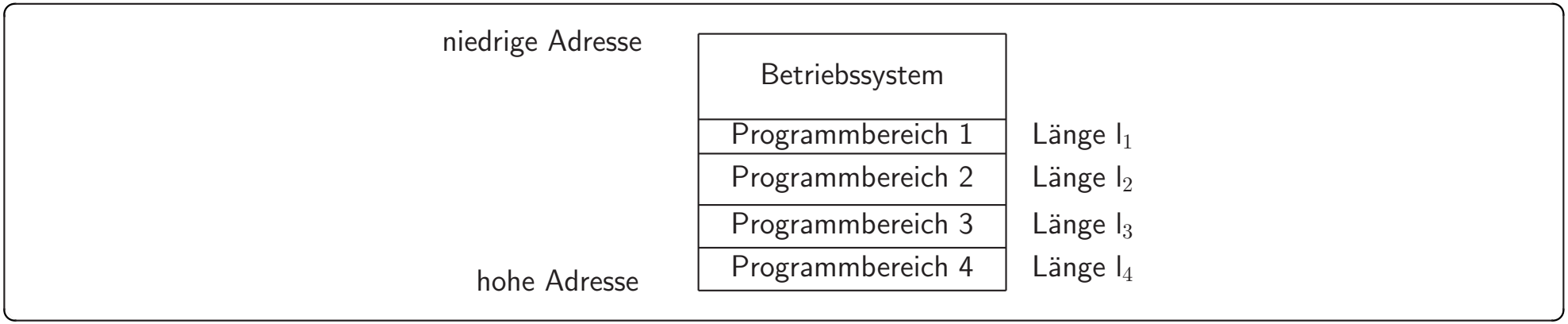


Abb. SPV–1 Hauptspeicherorganisation mit statischen Bereichen unterschiedlicher Länge

- **Nachteile:**
  - Jedes Programm muss so geschrieben sein, dass es in einen – zumindest den größten – Bereich passt.
    - ◇ Aufgrund der adresssequenziellen Abarbeitung der Befehle durch den Prozessor muss der Bereich zwingend zusammenhängend sein!
  - Auftreten **interner Fragmentierung**: (siehe Abb. SPV–2)
    Der nicht genutzte Speicher kann aufgrund der eingesetzten Verwaltungsstrukturen nicht für andere Speicheranforderungen genutzt werden.

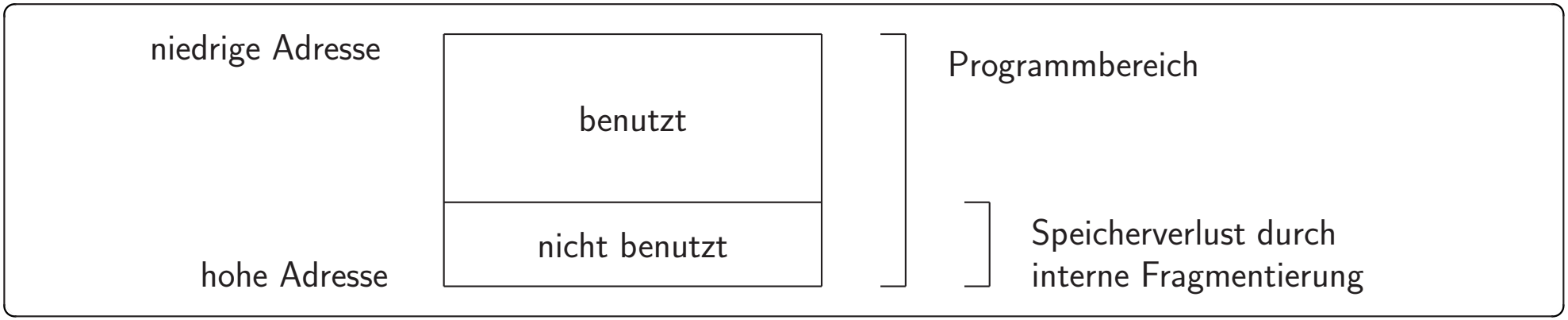


Abb. SPV–2 Interne Fragmentierung bei Verwendung von Speicherbereichen fester Länge

- **Flexiblere Alternative:**
  - Speicher wird in (relativ) kleine Bereiche fester Länge aufgeteilt $\rightarrow$ Abb. SPV–3.
  - Ein Programm belegt mehrere (aufeinanderfolgende) Bereiche.
  - Verlust durch interne Fragmentierung: durchschnittlich ein halber Bereich pro Programm.

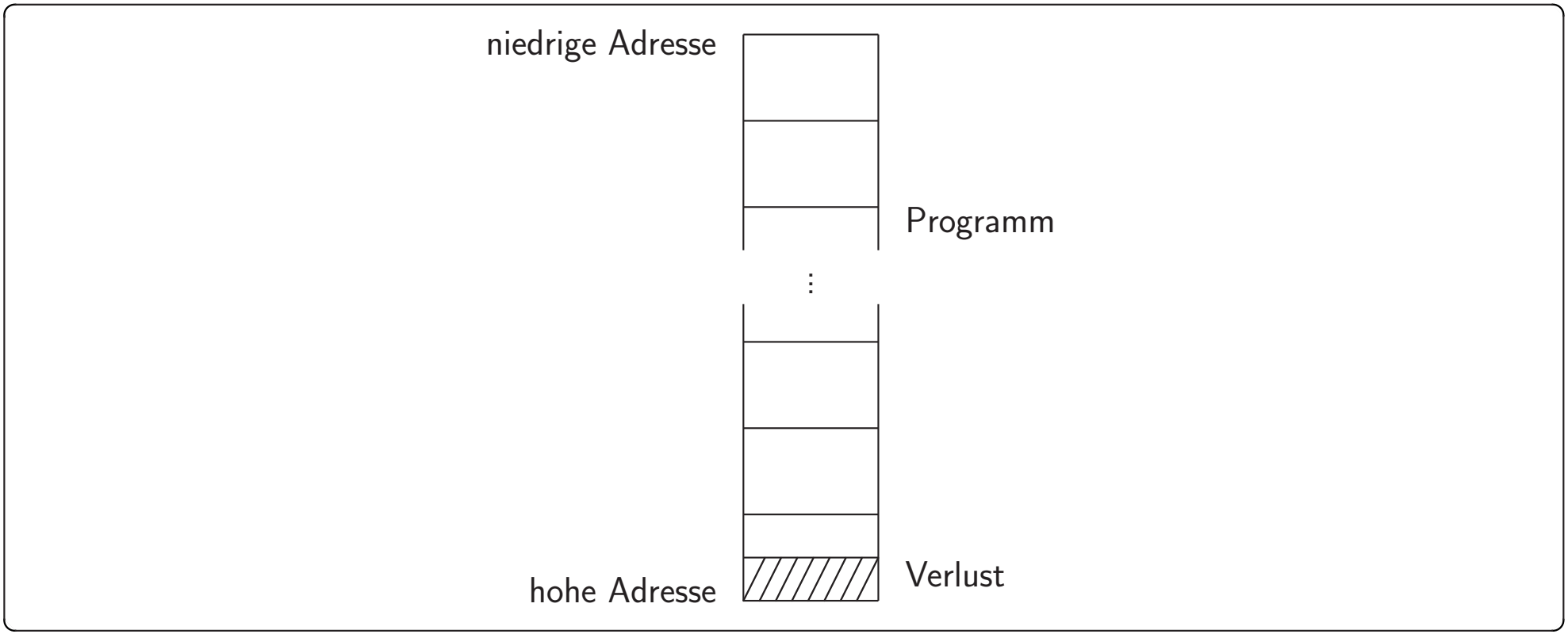


Abb. SPV–3 Hauptspeicherorganisation mit statischen Bereichen gleicher Länge

Bereiche variabler Länge

- **Konzept:**
  - Betriebssystem stellt Speicherbereich in der gewünschten Länge bereit.
- Problem: **externe Fragmentierung**

  $\Rightarrow$ Insgesamt ausreichend freier Speicher vorhanden, jedoch nicht zusammenhängend.

  ($\rightarrow$ siehe Abb. SPV–4).

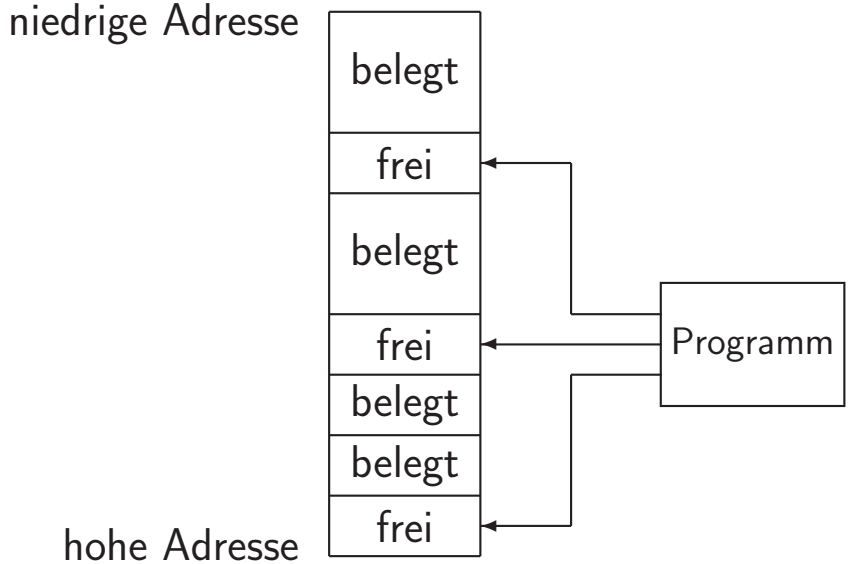


Abb. SPV–4 Externe Fragmentierung bei Speicherbereichen variabler Länge

- **Reaktionsmöglichkeiten:**

  (Angeforderter Bereich ist größer als der größte noch vorhandene Bereich.)

  (a) Anforderung ablehnen.

  (b) Andere Einheit (z.B. ein Programm) auslagern ($\rightarrow$ Dateispeicher).

  (c) Speicherkompaktierung (*Compaction*) durchführen.

  $\Rightarrow$ Aufwändig, da i. A. Adressen umgeschrieben werden müssen !

  $\Rightarrow$ Falls gerade ein E/A-Vorgang auf diesem Bereich läuft, ist Datenverlust möglich !

Überlagerung von Programmen

- Programm ist größer als Hauptspeicher $\rightarrow$ **Überlagerungsstruktur** (*Overlay*) verwenden !

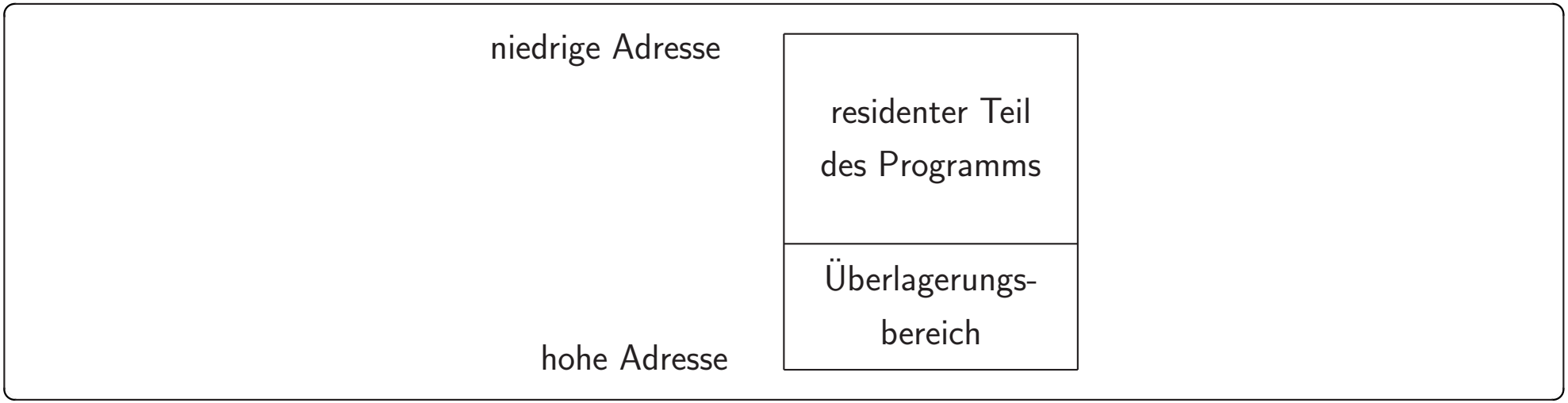


Abb. SPV–5 Verwendung eines Überlagerungsbereiches

- **Probleme:**
  - Wie stellt man sicher, dass immer die richtigen Programmteile im Überlagerungsbereich sind ?
    $\Rightarrow$ Aufwändige Programmierung.

Hauptspeicherverwaltung und -zuteilung

Bereiche fester Länge

- **Verwaltung der statischen Bereiche:**
  - Für jeden Bereich wird ein Bit benötigt, das angibt, ob er frei oder belegt ist ($\Rightarrow$ **Core Map**).
  - Falls sich ein Programm über mehrere Bereiche erstreckt, muss man die Zuordnung

    Programm $\leftrightarrow$ *belegte Bereiche*

    verwalten (z.B. zum dynamischen Umladen).

- **Einfache Speicherplatzzuteilung:**
  - Programm wird **einem** statischen Bereich zugeordnet.
    - ◇ Voraussetzung: Programmlänge muss bekannt sein!
    - ◇ Alternative Zuteilungsstrategien:
      - ▷ Anwender gibt (in der Auftragsbeschreibung) an, in welchem Bereich das Programm laufen soll.

        $\Rightarrow$ Programm wird in diesen Bereich geladen, sobald er frei wird.
      - ▷ Die Auftragssteuerung entscheidet, in welchen Bereich das Programm geladen wird.

- **Variabler aber komplexer:**
  - Programm erstreckt sich über **mehrere** Bereiche fester Länge.
    - ◇ Voraussetzung: Programmlänge muss bekannt sein.
    - ◇ Zuteilungsstrategie:
      - ▷ Auftragsverwaltung muss eine entsprechende Menge zusammenhängender Bereiche finden, ggf. durch Verschieben von Programmen ($\rightarrow$ äußerst aufwändig).
      - ▷ Problem: Falls Programme **nicht verschiebbar** sind, tritt neben **interner** auch **externe** Fragmentierung auf!
    - ◇ **Externe Fragmentierung:** Der nicht genutzte Speicher könnte zwar prinzipiell für andere Speicheranforderungen genutzt werden, ist jedoch zu klein als dies praktisch möglich wäre.

Bereiche variabler Länge

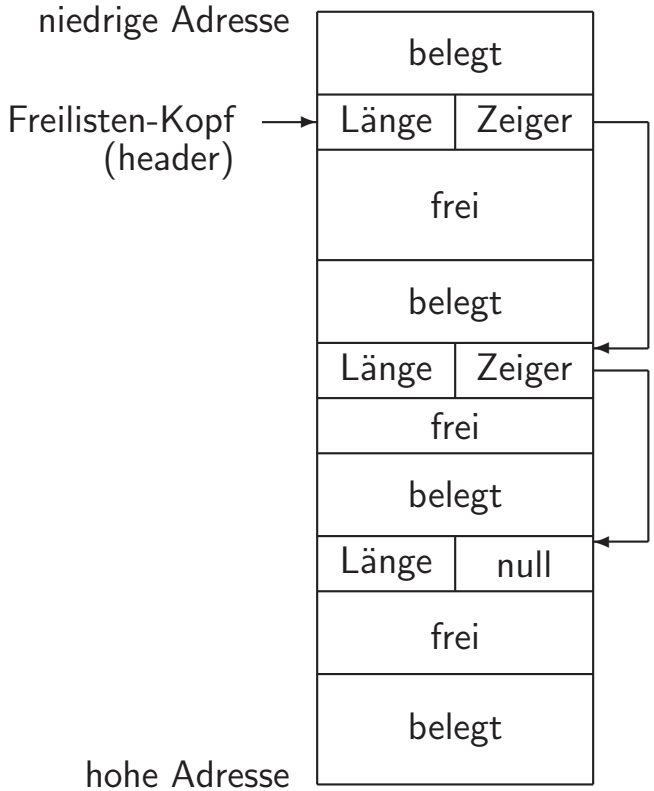


Abb. SPV–6 Freiliste für unbelegte Speicherbereiche

- **Verwaltung der dynamischen Bereiche:**
  - Verkettung der unbelegten Bereiche in einer **Freiliste** (→ Abb. SPV–6).
  - Bei der Rückgabe von Bereichen werden aneinandergrenzende Bereiche verschmolzen.

- Einsatz insb. für die **Freispeicherverwaltung** auf dem Heap-Speicher

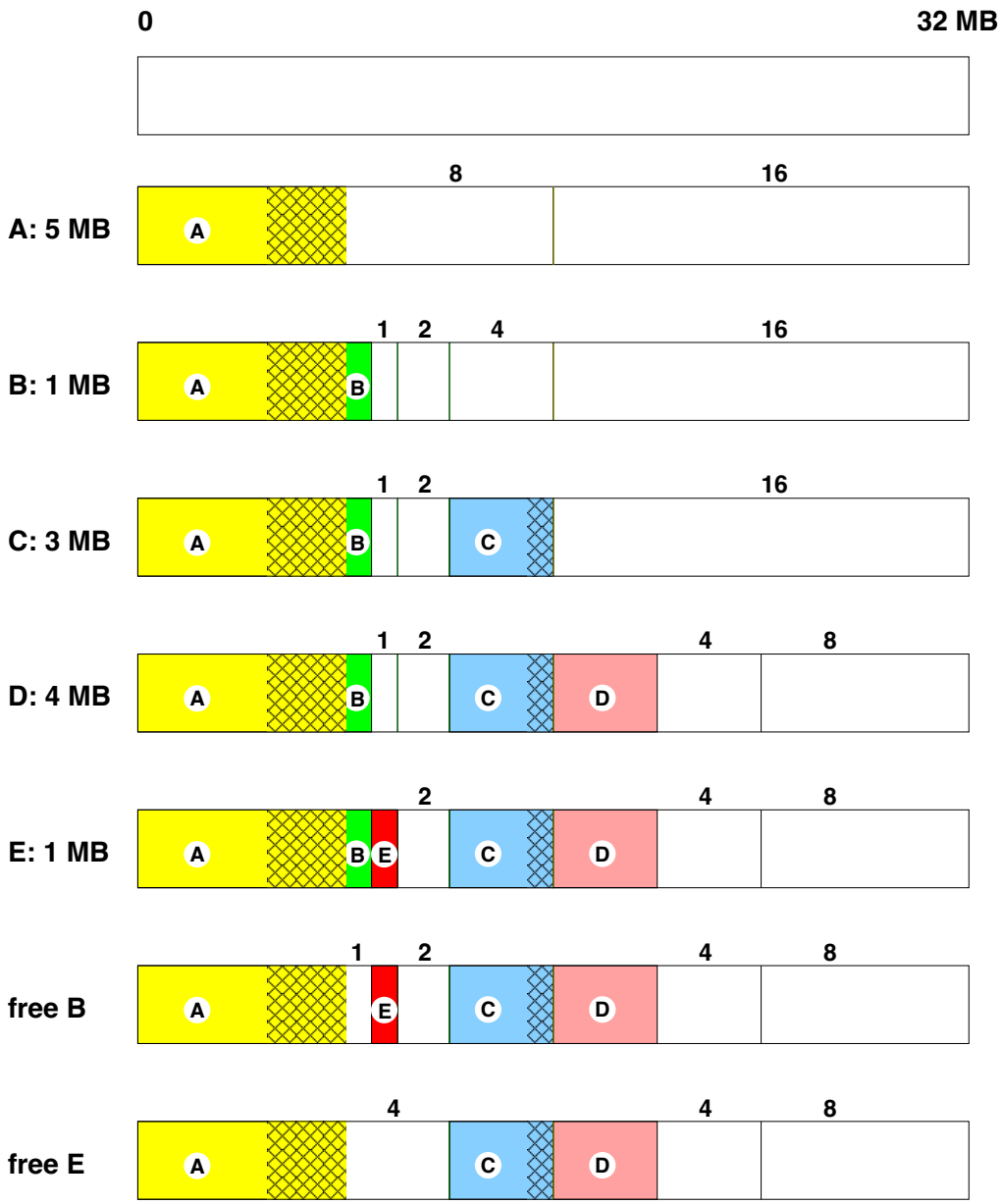
- **Zuteilungsstrategien:**
  - First-Fit
  - Best-Fit
  - Buddy-System

- First-Fit :
  - Verwende den ersten freien Bereich, der groß genug ist.
  - Der nicht benötigte Rest des Bereichs wird wieder in die Freiliste eingekettet.

- Best-Fit:
  - Finde kleinsten Bereich, der gerade noch passt.
  - Der nicht benötigte Rest des Bereichs wird in die Freiliste eingekettet.

- Buddy-System:
  - Der Speicher wird in Bereiche konstanter Länge l zerlegt.
  - Jede Anforderung wird auf $l \cdot 2^k$ ($k \in I\!N_0$, k minimal) aufgerundet.
  - Annahme: Gesamtgröße des Speichers ist $l \cdot 2^S$.
    - ◇ Es werden $S + 1$ Listen verwendet.
    - ◇ Jede Liste i enthält die freien Bereiche der Länge $l \cdot 2^i$.
  - Verarbeiten einer Anforderung der Länge $l \cdot 2^i$ (aufgerundet):
    (1) Falls mind. ein Element in Liste i vorhanden:
    $\Rightarrow$ Wähle eines aus; Ende.
    (2) Betrachte die nächstgrößere Liste: Falls diese mind. ein Element enthält, dann wähle eines aus;
    $\Rightarrow$ (a) Halbiere das Element $\rightarrow$ zwei *Buddies* entstehen;
    (b) Trage eine Hälfte in die nächstkleinere Liste ein;
    (c) Hat die andere Hälfte die Länge $l \cdot 2^i$? $\Rightarrow$ Verwende diese; Ende.
    (d) Wiederhole Schritt (a) mit dieser Hälfte.
    (3) Wiederhole Schritt (2) bis so ein freier Bereich gefunden oder die Anforderung komplett abgelehnt wurde.
  - Beim Freigeben von Speicher: Verschmelzen mit jeweiligem *Buddy*; ggf. rekursiv wiederholen.

- **Beispiel:**


0
32 MB
A: 5 MB
8
16
A
B: 1 MB
1
2
4
16
A
B
C: 3 MB
1
2
16
A
B
C
D: 4 MB
1
2
4
8
A
B
C
D
E: 1 MB
2
4
8
A
B
E
C
D
free B
1
2
4
8
A
E
C
D
free E
4
4
8
A
C
D

- **Bewertung:**
  - First-Fit / Best-Fit :
    - ◇ Best-Fit ist teurer als First-Fit, da (fast immer) die gesamte Liste zu durchsuchen ist.
    - ◇ Simulationen haben gezeigt, dass First-Fit fast immer erfolgreicher als Best-Fit ist.
    - ◇ *Variante 1:* Die Freiliste wird nach Größe sortiert.

      $\Rightarrow$ First-Fit und Best-Fit sind identisch.
    - ◇ *Variante 2:* Anstatt einer Liste können mehrere benutzt werden, die Bereiche bestimmter Größenklassen enthalten.
    - ◇ Bei beiden Varianten ist die Suche eines geeigneten Bereichs schneller, die Rückgabe ist jedoch aufwändiger.
  - Buddy-System :
    - ◇ Das Zusammenfügen von Programmbereichen geht sehr schnell.
    - ◇ Das Verfahren ist insgesamt schneller als First-Fit.
    - ◇ Bei **zufälliger** Verteilung der angeforderten Speichergrößen wird ca. 1/3 mehr Speicherplatz als bei First-Fit benötigt ($\rightarrow$ stärkere interne Fragmentierung).

Virtueller Speicher

Grundlagen

Modell des virtuellen Speichers

- **Ideen:** (→ siehe Abb. SPV–7 (S. 15))
  - Der Hauptspeicher wird durch einen Teil des **Hintergrundspeichers** (→ Swap-Bereich auf Festplatte) ergänzt.
    - ◇ Folge: Das Programm muss **nicht vollständig** im Hauptspeicher liegen.
      - ⇒ Das Betriebssystem lädt fehlende Programmteile nach.
  - Programmadressen sind prozessindividuelle **virtuelle Adressen**, die in **reale** Hauptspeicheradressen oder ggf. solche des Hintergrundspeichers umgerechnet werden müssen.

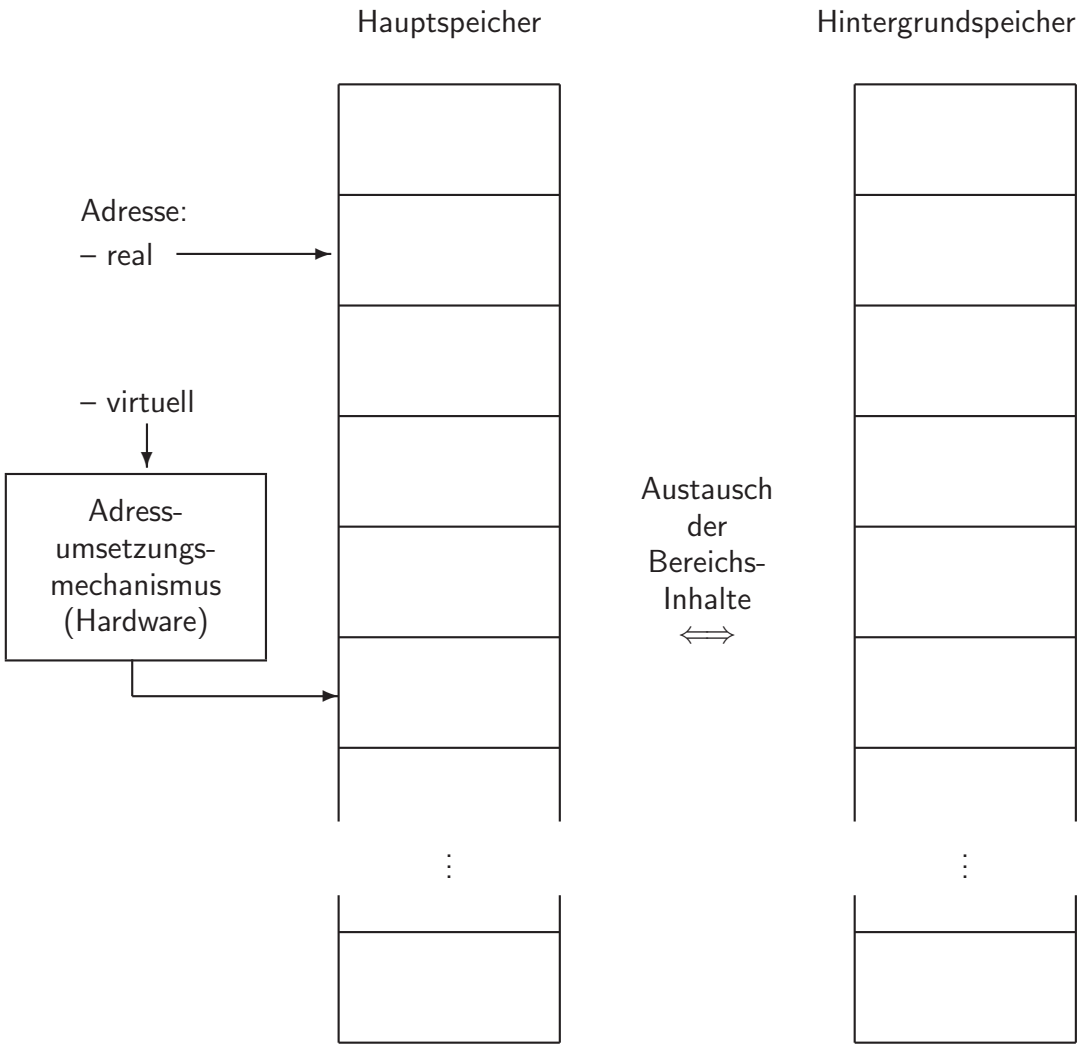


Abb. SPV–7 Konzeptuelle Struktur des virtuellen Speichers

Adressumsetzung

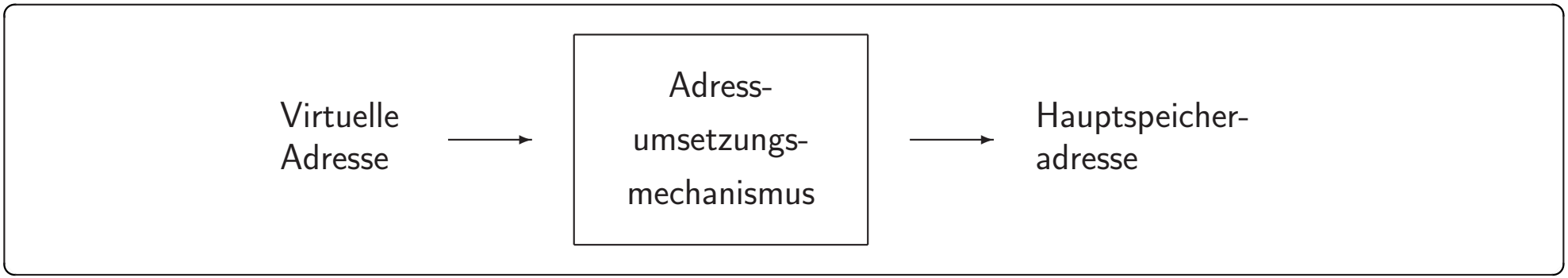


Abb. SPV–8 Umsetzung einer virtuellen Adresse

- Adressumsetzung muss Hardware-unterstützt durchgeführt werden ($\rightarrow$ vgl. Abb. SPV–8); andernfalls wird die Gesamtsystemeffizienz stark beeinträchtigt.
  - Typische Adresslängen: 32 oder 64 Bit
    - ◇ 32 Bit $\rightarrow$ 2^{32} Worte = 4 Gigaworte Adressraum
    - ◇ 64 Bit $\rightarrow$ 2^{64} Worte = 16 384 Petaworte Adressraum

- Nachladbare Programmteile:
  - **Seiten** (Pages; Bereiche fester Länge)

Seitentausch (*Paging*)

Hauptspeicherorganisation

- Einteilung des Hauptspeichers in **Kacheln** (*Page Frames*) fester Größe
  $\Rightarrow$ einfache Adressierung.

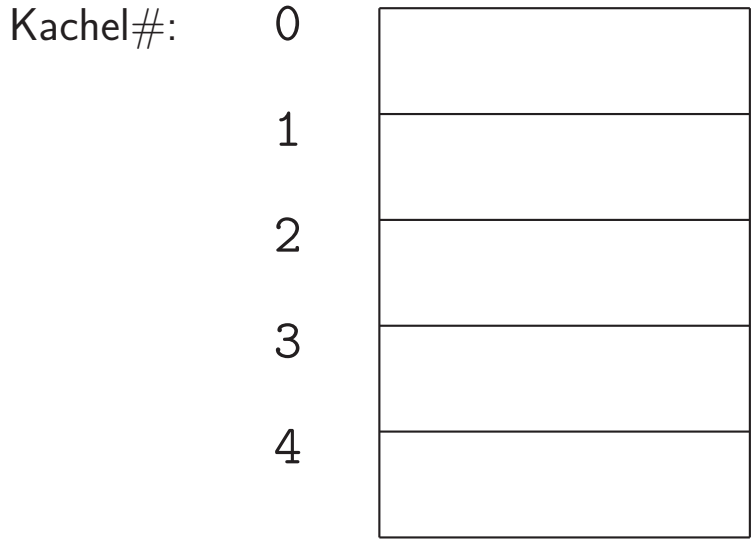
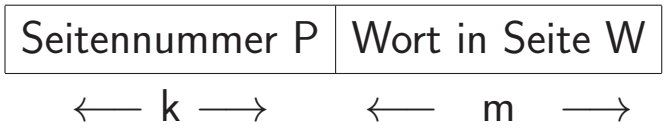


Abb. SPV–9 Einteilung des Hauptspeichers in Kacheln fester Länge

Wie finde ich ein Speicherwort ?

- Interpretation einer vom Übersetzer erzeugten virtuellen Adresse (wird von der Hardware erledigt):



  - Anzahl der Seiten: 2^k
  - Seitengröße: 2^m Worte

- Ziel bei der Adressumsetzung: Aus der Seitennummer P die Kachelnummer K bestimmen.
- **Vorgehensweise:**
  - Verwenden einer **Seitentabelle** ($\rightarrow$ siehe Abb. SPV–10 (S. 19)).
  - **Konkatenieren** der Kachelnummer K mit der Wortadresse W.
  - Hauptspeicheradresse $= \langle$ K,W $\rangle$

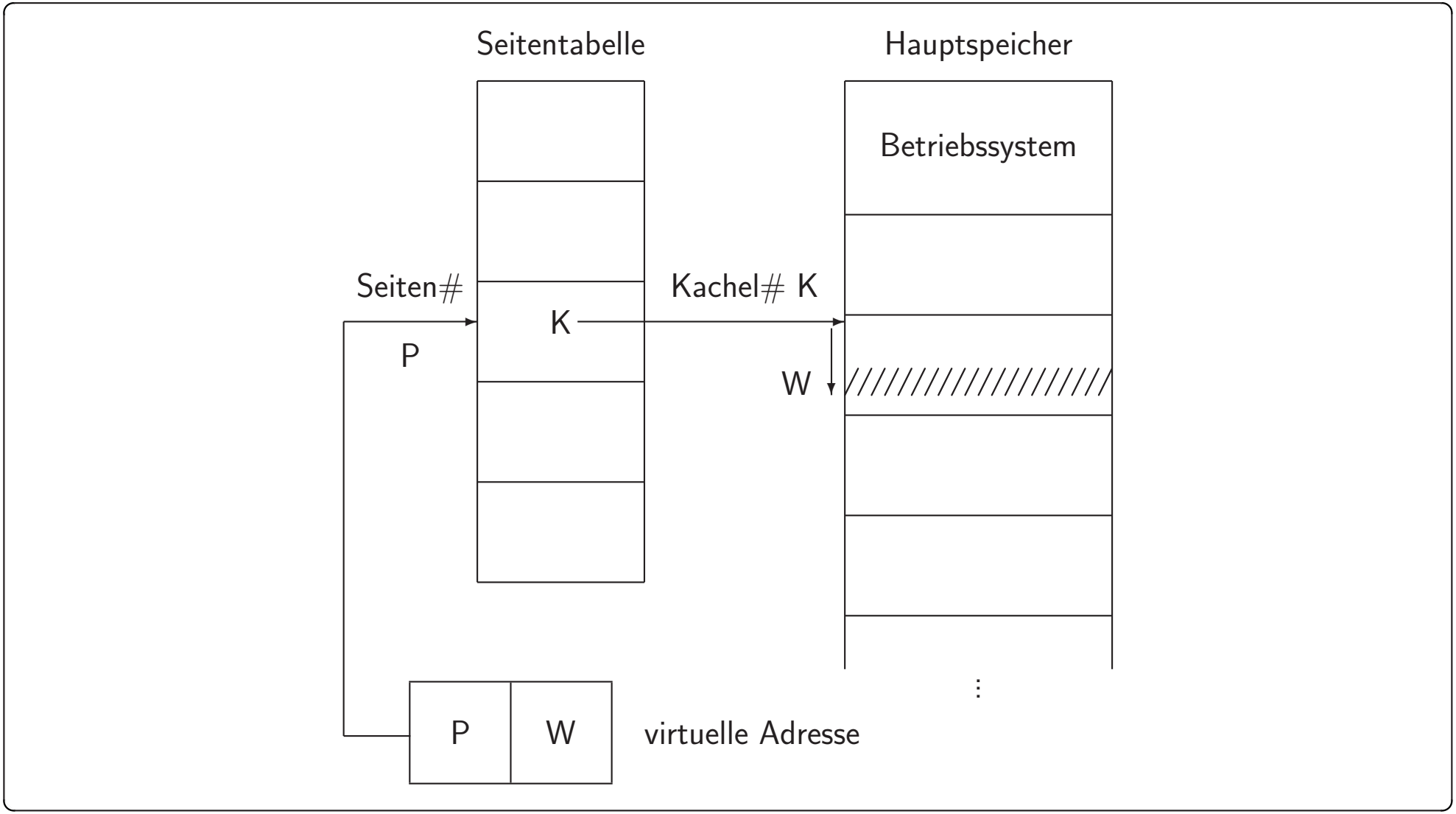


Abb. SPV–10 Adressumsetzung bei Seitentausch

- Folge: Die Seiten eines Programms können im Hauptspeicher **verstreut** liegen.

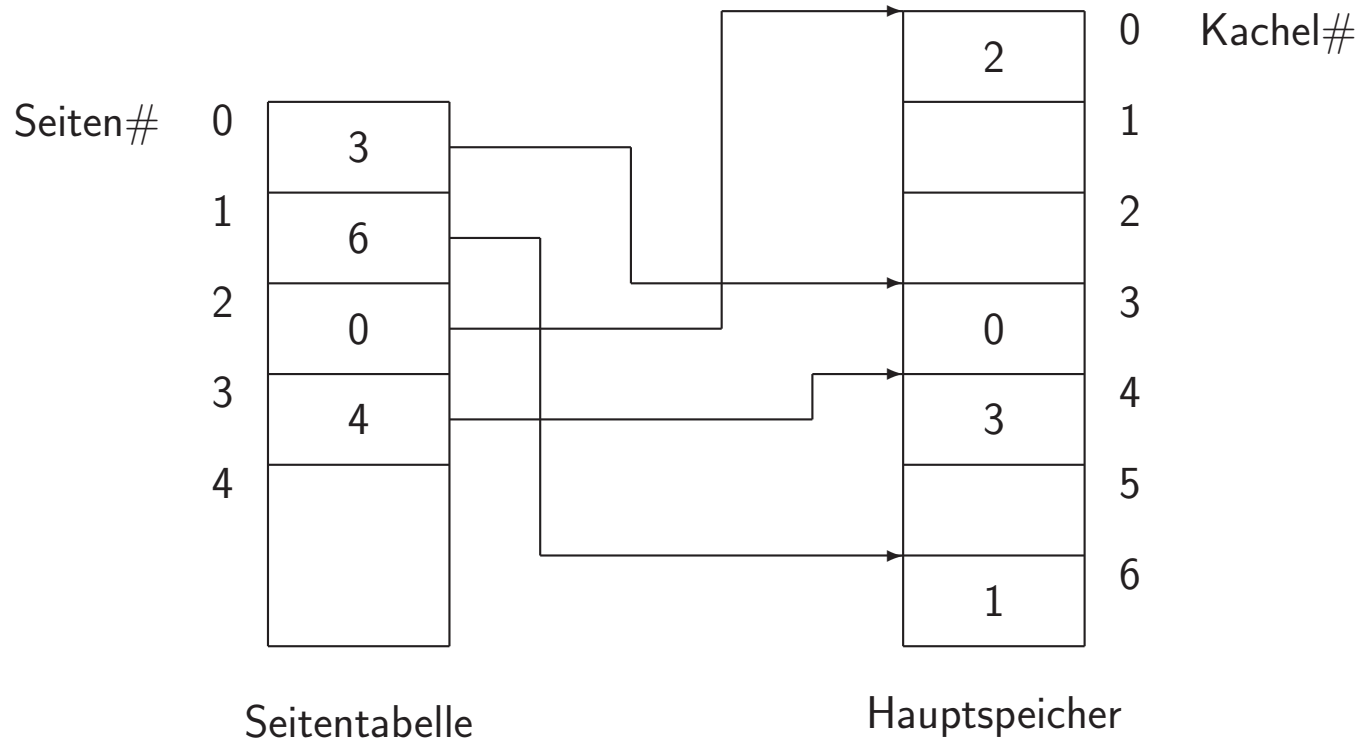


Abb. SPV–11 Abbildung virtueller Seiten auf Hauptspeicherkacheln

- Aufbau eines **Seitentabelleneintrages** (*Page Table Entry*) (→ Abb. SPV–12).

K	P-Bit	Ref-Bit	C-Bit	R-Bit	W-Bit	E-Bit	Hintergrundadresse

K: Kachelnummer
P-Bit: *Presence*-Bit, Präsenz-Bit
Ref-Bit: *Referenced*-Bit, *Used*-Bit, *Accessed*-Bit
C-Bit: *Change*-Bit, *Dirty*-Bit, *Modified*-Bit
R-Bit: *Read*-Bit, Lese-Bit
W-Bit: *Write*-Bit, Schreib-Bit
E-Bit: *Execute*-Bit, Ausführ-Bit

Abb. SPV–12 Struktur eines Seitentabelleneintrages

- C-Bit = 0 ⇒ Daten auf dieser Seite wurden nicht verändert.
- Verwendung von Schutzbits auf Seitenebene ⇒ R-Bit, W-Bit, E-Bit
- P-Bit = 0 ⇒ Adressumsetzungs-Hardware erzeugt **Seitenfehler-Interrupt** bei Zugriff.
  - ◇ Struktur der Interrupt-Behandlungsoperation: siehe Prg. SPV–1 (S. 22)

Ein-/Auslagerungen stellen E/A-Operationen dar ⇒ Prozesswechsel erfolgt.

```
if (freie Kachel) {
   Seite einlagern;
   Kachelnummer in Seitentabelle eintragen;
}
else {
   andere Seite des Hauptspeichers auslagern, ggf. zurückschreiben;
       //  (welche Seite? Strategie?)
   im Seitentabelleneintrag dieser Seite P-Bit := 0;
   im Seitentabelleneintrag der gewünschten Seite
       Kachelnummer eintragen;
   gewünschte Seite einlagern;
       //  (woher? wohin?)
}
P-Bit := 1;
C-Bit := 0;
```

Prg. SPV–1 Struktur der Seitenfehlerunterbrechungsbehandlungsoperation

Schutzaspekt: Adressraumtrennung

- Jeder Prozess besitzt seine eigene Seitentabelle.
  - Folge: Prozesse adressieren mit identischen virtuellen Adressen unterschiedliche physikalische Speicherbereiche (vgl. Beispiel in Abb. SPV–13).
- Deren Anfangsadresse liegt im **Seitentabellenadressregister** (gehört zum Prozesskontext).

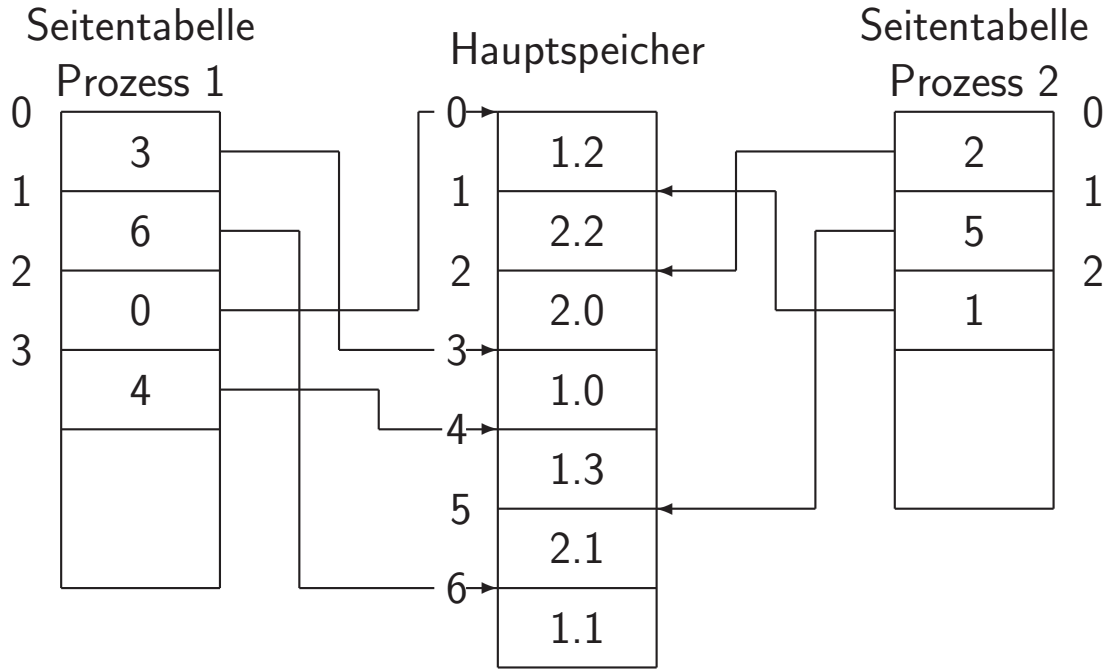


Abb. SPV–13 Zugriffe auf Hauptspeicherkacheln durch zwei Prozesse

Gemeinsame Nutzung von Seiten

- Prozesse greifen manchmal auf dieselben Daten zu (Beispiel: Abb. SPV–14).
  - Schutzproblem: Diese Daten sind oft kürzer (oder länger) als eine Seite.

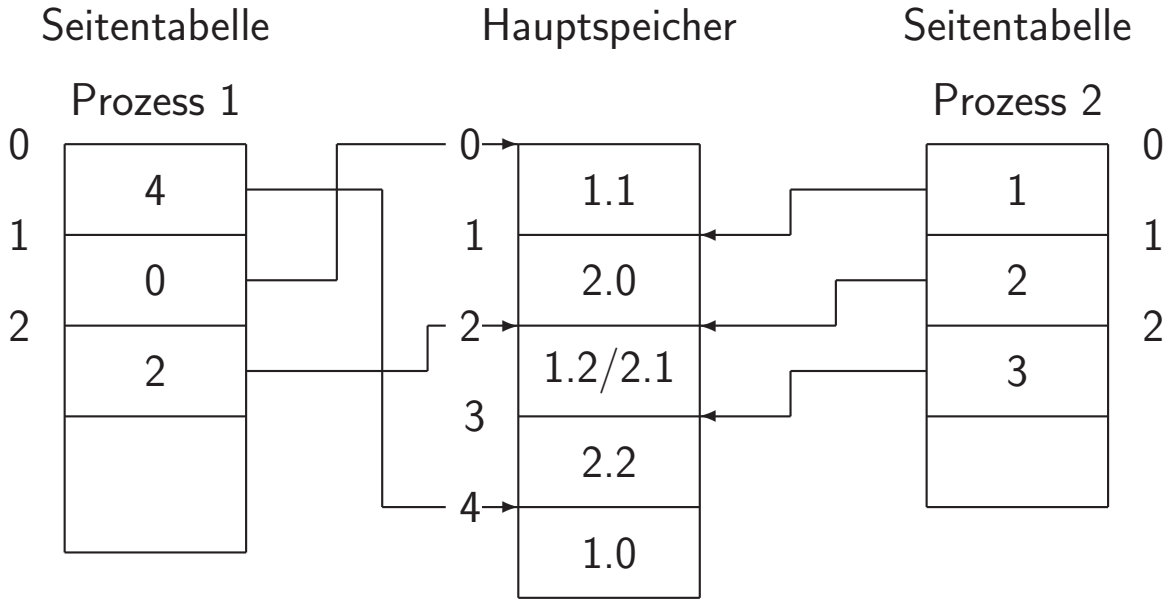


Abb. SPV–14 Gemeinsame Benutzung von Seiten

Bewertung des Seitentauschs

- **Vorteile:**
  - Durch Verwendung einer festen Kachellänge einfache Hauptspeicherverwaltung.
  - Einfache Struktur des Hintergrundspeichers (*Swap-Space*) als eigene Partition oder als Datei.
- **Nachteile:**
  - Keine Berücksichtigung logischer Einheiten beim Schutz und gemeinsamer Benutzung von Speicherbereichen.
  - Interne Fragmentierung des Hauptspeichers:
    - ◇ Pro Adressraum geht im Schnitt eine halbe Seite verloren.
  - Effizienz?!

Effizienz

- **Problem:** Aus jedem Hauptspeicherzugriff werden zwei!
  - Pro Adressumsetzung muss zusätzlich einmal auf Seitentabelle zugegriffen werden.

- **Lösung:** Hardware-Unterstützung
  - Verwendung eines Adress-Cache als Teil der **Memory Management Unit** (MMU):
    TLB (*Translation Look Aside Buffer*)
    - ◇ Realisiert als schneller **Cache-Speicher** (i. A. prozessorintern vorhanden, 40 – 512 Einträge).
    - ◇ Leistet Abbildung: *virtuelle Seitennummer* $\rightarrow$ *Kachelnummer*.
    - ◇ Einträge im TLB werden von der Hardware verwaltet (meist nach LRU-Ersetzungsstrategie).
    - ◇ Bei einem Prozesswechsel muss der TLB invalidiert werden. ($\rightarrow$ **Wieso ?**)
    - ◇ Erreichbare Trefferquoten: 80 bis 95 %
  - Verwendung von Daten-Caches: log. bzw. phys. Cache
    - ◇ Speichern Inhalte von Wortadressen unter deren virt. bzw. realer Adressen.
    - ◇ Log. Cache muss bei Prozesswechsel invalidiert werden.

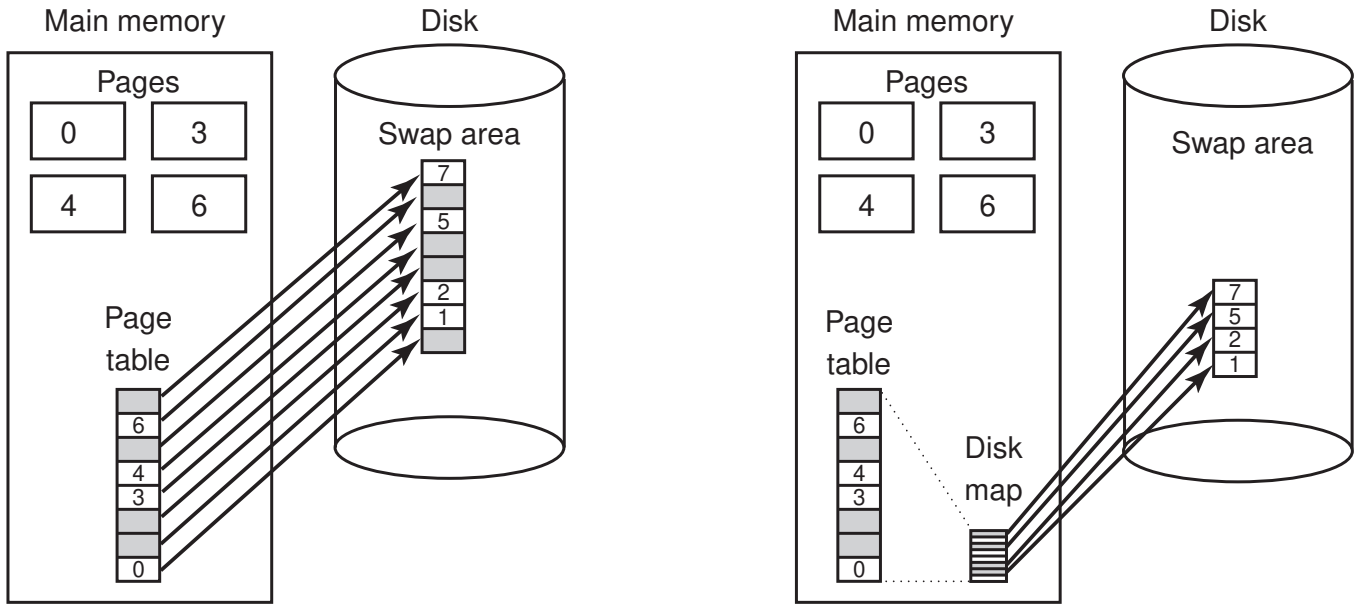
Speicherung der Seitentabellen

- **Ungefährer Platzbedarf pro Adressraum**
  - bei 32-Bit Adressen:
    - ◇ 1 Mio. Einträge: bei Seitengröße von 4 KB $\rightarrow$ ca. 10 MB
  - bei 64-Bit Adressen:
    - ◇ 2^{52} Einträge $\rightarrow$ ca. 20.000 TB (!!!)

- **Lösungen**
  - Hierarchische Seitentabellen
    - ◇ Idee: Adressumsetzung erfolgt in mehreren Stufen
    - ◇ Vorteil: die nachgeordneten Teilbäume müssen nicht alle vollständig angelegt werden
  - Invertierte Seitentabellen (*inverted page tables*)
    - ◇ Idee: ein Eintrag **pro Hauptspeicherkachel** (enthält PID, Seitennr.)
    - ◇ Vorteil: nur eine einzige Tabelle ausreichender Größe für den tatsächlich vorhandenen Speicher nötig.
    - ◇ Nachteil: Seitennr. kann nicht mehr als Index in die Seitentabelle dienen $\Rightarrow$ teure Suche!
      - ▷ Abhilfe:
        (1) großen TLB verwenden
        (2) Hash-Tabelle zur schnellen Suche einsetzen (Größe: Anzahl der Kacheln)

Verwaltung des Hintergrundspeichers

- **Swap Space**
  - **(a)** statisch: Anfangsadresse pro Prozess bekannt
  - **(b)** dynamisch: Adresse pro Seite bekannt



Speicherverwaltung virtueller Systeme

Speicherplatzzuteilung

- **Seitentausch:**
  - Verwendung von Einheiten fester Länge.
    - ◇ Seitengröße klein $\Rightarrow$ geringe interne Fragmentierung.
    - ◇ Seitengröße groß $\Rightarrow$ effiziente Plattenzugriffe; kleinere Seitentabellen.
    - ◇ Seitengröße identisch zur Größe eines physikalischen Plattenblocks $\Rightarrow$ einfach beim Ein-/Auslagern.
    - ◇ In der Praxis: 4 KiB bis 16 KiB

- **Verwaltungsstrategien für den Heap:**
  - First-Fit-, Best-Fit-Verfahren oder Buddy-System

Einlagerungsstrategien

- **Vorgeplanter Seitentausch** (*Pre-Paging*)
  - **Seiten im Voraus** in den Hauptspeicher bringen, um Seitenfehler zu vermeiden
  - **Viele Nachteile:**
    - ◇ Prozessorzeit notwendig, um die Auswahl zu treffen.
    - ◇ Zu teuer, wenn Seite nicht benötigt wird:
      - ▷ Unnötige Plattenübertragung.
      - ▷ Prozessorzeit verschwendet.
      - ▷ Platz im Hauptspeicher verschwendet.

    $\Rightarrow$ Trefferrate muss extrem hoch sein (lohnt in der Praxis selten).

- **Seitentausch auf Verlangen** (*Demand Paging*): wird fast immer verwendet.

Auslagerungsstrategien

Überblick

- Die Gesamtleistung eines Computersystems hängt wesentlich von der Wahl der Auslagerungsstrategie ab!
- Auslagerungsstrategien:
  - **Random**
  - **First In First Out** (FIFO)
  - **Not Frequently Used** (NFU)
  - **Least Recently Used** (LRU)
  - **Optimal Replacement**
  - **Second Chance**

Random

- *Lagere eine zufällig ausgewählte Seite aus.*
  - Einfache Implementierung ohne zusätzliche Datenstrukturen.

First In First Out (FIFO)

- *Lagere die Seite aus, die sich schon am längsten im Hauptspeicher befindet (also die älteste Seite).*
  - Implementierung über zyklischen Kachelzeiger.
  - In der Praxis schlechte Ergebnisse, da das Zugriffsverhalten der Programme nicht berücksichtigt wird.

Not Frequently Used (NFU)

- *Lagere die Seite aus, die am seltensten benutzt wurde.*
- Implementierung: Zähler für Zugriffe verwenden; Seite mit niedrigstem Zählerwert auslagern.
- Problem: Seiten, die früher häufig genutzt wurden, werden im Speicher gehalten, neue eher ausgelagert.
  - ◇ Abhilfe **Aging**: Zeitstempel der letzten Nutzung speichern und berücksichtigen.

Least Recently Used (LRU)

- *Lagere die Seite aus, auf die am längsten nicht mehr zugegriffen wurde.*
  - Implementierung mittels Hardware-mäßig zu setzenden **Referenz-Bits** (*Reference Bit*):

```
if (Kachelzugriff) referenzbit := 1;
```

    - ◇ Verwaltung einer Tabelle mit jeweils einem **Zähler** pro Hauptspeicherkachel.
    - ◇ Verwendung von Messintervallen:
      - ▷ Am Intervallanfang: alle Referenzbits werden auf 0 gesetzt.
      - ▷ Am Intervallende: Referenzbit $= 0 \Rightarrow$ Zähler wird inkrementiert.

```
if (referenzbit == 1) {      // (in Software)
        referenzbit := 0;
        zaehler := 0;
        }
    else
zaehler := zaehler + 1;
```

    - ◇ Die Seite wird ausgelagert, deren Zähler den höchsten Wert hat; Zähler auf 0 setzen.
    - ◇ Um Zählerüberläufe zu vermeiden, werden nach einer gewissen Zeit (lang im Vergleich zum Messintervall) alle Zähler auf 0 zurückgesetzt.

Optimal Replacement

- *Lagere die Seite aus, für die der nächste Zugriff am Weitesten in der Zukunft liegt.*
  - Nicht implementierbar ($\rightarrow$ **Wieso nicht ?)**
  - Wird als Referenz für andere Strategien verwendet.

Second Chance

- **Idee:** Jede Seite erhält eine zweite Chance, veränderte Seiten sogar eine dritte !
- Implementierung als **Clock-Algorithmus**:
  - Neben **Referenzbit** und **Change-Bit** (beide CPU) wird ein **Auslagerbit** (in Software) verwendet. Über einen **zyklischen Kachelzeiger** werden nacheinander alle Kacheln betrachtet.
- Zustandsübergänge einer Seite beim Second Chance Algorithmus $\rightarrow$ siehe Abb. SPV–15 (S. 35).

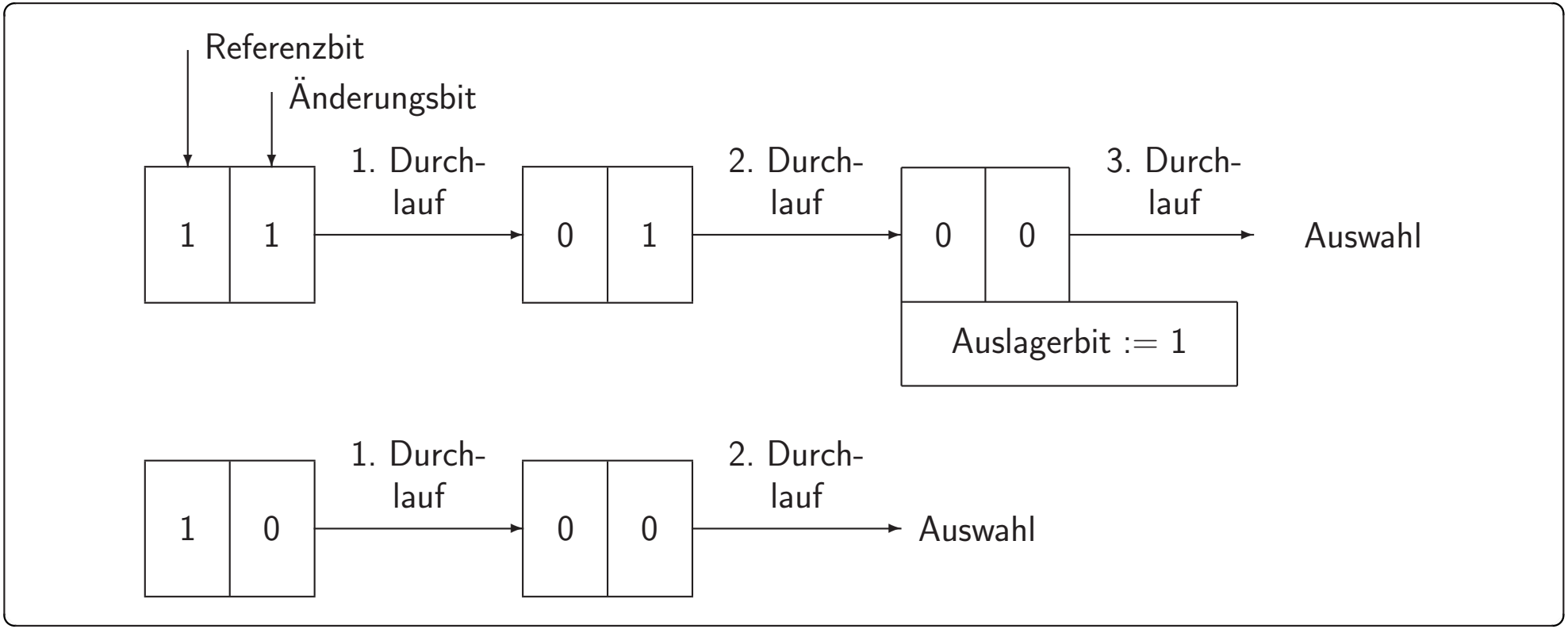


Abb. SPV–15 Zustandsübergänge einer Seite beim Second Chance Algorithmus

- Struktur des Algorithmus → siehe Prg. SPV–2 (S. 36).
- **Mögliche Endekriterien:**
  - Es ist genau eine Kachel frei.
  - Es sind einige Kacheln frei (z.B. mindestens 25 %).

```
do {
   if (referenzbit == 1)
      referenzbit := 0;
   else                                  // R == 0
      if (aenderungsbit == 1) {
         aenderungsbit := 0;
         auslagerungsbit := 1;}
      else {                            // R == 0 & C == 0
         Auswahl dieser Seite;
         if (auslagerungsbit == 1)
            Seite auslagern, d.h. Kachelinhalt auf Platte zurückschreiben;
         Kachel freigeben;
      }
   erhöhe zyklischen Kachelzeiger;
} while (Endekriterium nicht erfüllt)
```

Prg. SPV–2 Struktur des Second Chance Algorithmus

Je mehr Kacheln desto schneller? Belady's Anomalie

- Prozess mit 5 Seiten greift in dieser Folge darauf zu:

  $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 1 \Rightarrow 2 \Rightarrow 5 \Rightarrow 1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5$

- Strategie: FIFO (a) mit 4 Kacheln

Referenz	0	1	2	3	(Fehler)
1,2,3,4	1	2	3	4	(4)
1	1	2	3	4	(–)
2	1	2	3	4	(–)
5	5	2	3	4	(1)
1	5	1	3	4	(1)
2	5	1	2	4	(1)
3	5	1	2	3	(1)
4	4	1	2	3	(1)
5	4	5	2	3	(1)

$\Sigma = 10$

(b) mit 3 Kacheln

Referenz	0	1	2	(Fehler)
1,2,3	1	2	3	(3)
4	4	2	3	(1)
1	4	1	3	(1)
2	4	1	2	(1)
5	5	1	2	(1)
1	5	1	2	(–)
2	5	1	2	(–)
3	5	3	2	(1)
4	5	3	4	(1)
5	5	3	4	(–)

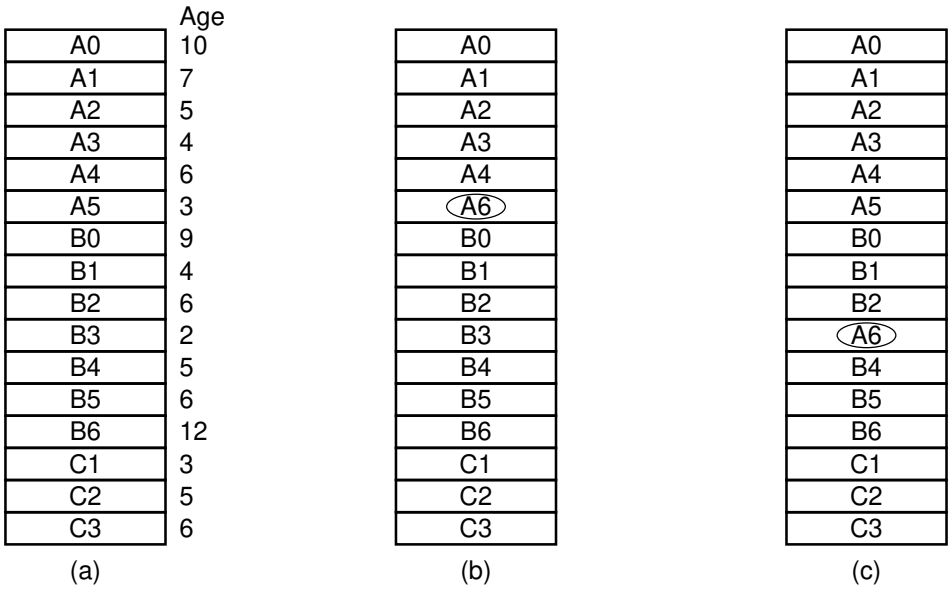
$\Sigma = 9$

- Optimal: 6 Fehler bei 4 Kacheln, 7 Fehler bei 3 Kacheln

Prozesse und virtueller Speicher

- **Seitenflattern** (*Thrashing*)
  - Der Prozessor verbringt einen Großteil der Zeit mit dem Ein- und Auslagern von Seiten.
    $\Rightarrow$ Antwortzeiten steigen dramatisch an!
    - ◇ Dieser Effekt tritt insbesondere dann auf, wenn zuviele Prozesse im System sind und den Prozessen nicht ausreichend viele Kacheln zur Verfügung stehen.
  - **Lösungsansatz:** Neuen Prozesszustand einführen:
    - ◇ **Langes Warten** (*Long Suspend*)
      - ▷ Seiten von Prozessen, die (z.B. im Vergleich zur Zeitscheibe bzw. zu einer Plattenübertragung) lange warten müssen, sind sehr gute Kandidaten zum Auslagern.
      - ▷ Alternativ kann auch ein ganzer derartiger Prozess ausgelagert werden.
      - ▷ Durch Einführen des zusätzlichen Prozesszustandes **Long Suspend** können solche Prozesse identifiziert werden.
      - ▷ Anwendungsbeispiel: Warten auf Terminaleingabe im Teilnehmerbetrieb.

- Verwenden einer **lokalen** oder einer **globalen** Ersetzungsstrategie ?
  - Lokal: Nur die Seiten des den Seitenfehler verursachenden Prozesses werden betrachet.
  - Global: Seiten können unabhängig von ihrer Prozesszugehörigkeit ausgelagert werden.



(a) Ausgangskonfiguration (b) lokale Strategie (c) globale Strategie

- **Global** ist im Allgemeinen besser (aus Sicht des Gesamtsystems)

- **Residente Seiten**
  - Bestimmte Seiten dürfen nicht ausgelagert werden:
    - ◇ Seiten, die in eine E/A-Operation einbezogen sind.
      - ▷ Grund: E/A-Controller adressieren **real** und nicht virtuell.
    - ◇ Seiten mit wichtigen Betriebssystemdaten oder -code
      - ▷ z.B. Unterbrechungsroutinen, Dispatcher, Seitentauschsystem
    - ◇ Bestimmte Seiten von Anwendungsprogrammen $\rightarrow$ Echtzeit-Software

  $\Rightarrow$ Verwendung eines **Locked Down Bits**:
  - ◇ Ist für eine Seite das Bit gesetzt, wird diese vom Auslagerungsalgorithmus nicht betrachtet.

- **Paging Daemon**
  - Prozess, der regelmäßig (aber nicht zu oft) einen gewissen Anteil an Kacheln frei räumt.
  - Vorteil: Bei Seitenfehlern ist Auslagern i. Allg. unnötig.

- **Das Arbeitsmengenmodell** (*Working Set Model*)
  - Die Seitenbenutzung eines Prozesses erfolgt normalerweise nicht zufällig:
    - ◇ Zu jedem Zeitpunkt werden einige Seiten sehr oft, andere jedoch selten oder gar nicht angesprochen.
      ($\Rightarrow$ **Wieso ?**)
  - Definition: **Working Set** *eines Prozesses*
    - ◇ Menge der Seiten, auf die ein Prozess gerade zugreift (während einer kurzen Zeitdauer).
    - ◇ $\boxed{1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1 \Rightarrow 2 \Rightarrow 1}\ \boxed{\Rightarrow 4 \Rightarrow 5 \Rightarrow 4 \Rightarrow 5 \Rightarrow 4 \Rightarrow 5}\ \Rightarrow 6 \Rightarrow 7 \Rightarrow 9 \Rightarrow \ldots$
      - ▷ $WS_1 = \{\ 1, 2, 3\ \}$, $WS_2 = \{\ 4, 5\ \}$
  - Problematisch: Die Größe des Working-Sets ändert sich mit der Zeit
    $\Rightarrow$ dynamisch weitere Kacheln anfordern bzw. freigeben ist sinnvoll
  - Definition: **Lokalität** $\rightarrow$ vgl. Abb. SPV–16 (S. 42).
    - ◇ Eigenschaft eines Prozesses, sich in seinem Working Set aufzuhalten.
  - **Beobachtung:**
    - ◇ Befindet sich das Working Set im Hauptspeicher, treten nur wenige Seitenfehler auf – eine Erhöhung der Kachelanzahl für diesen Prozess bringt dann nicht mehr viel.

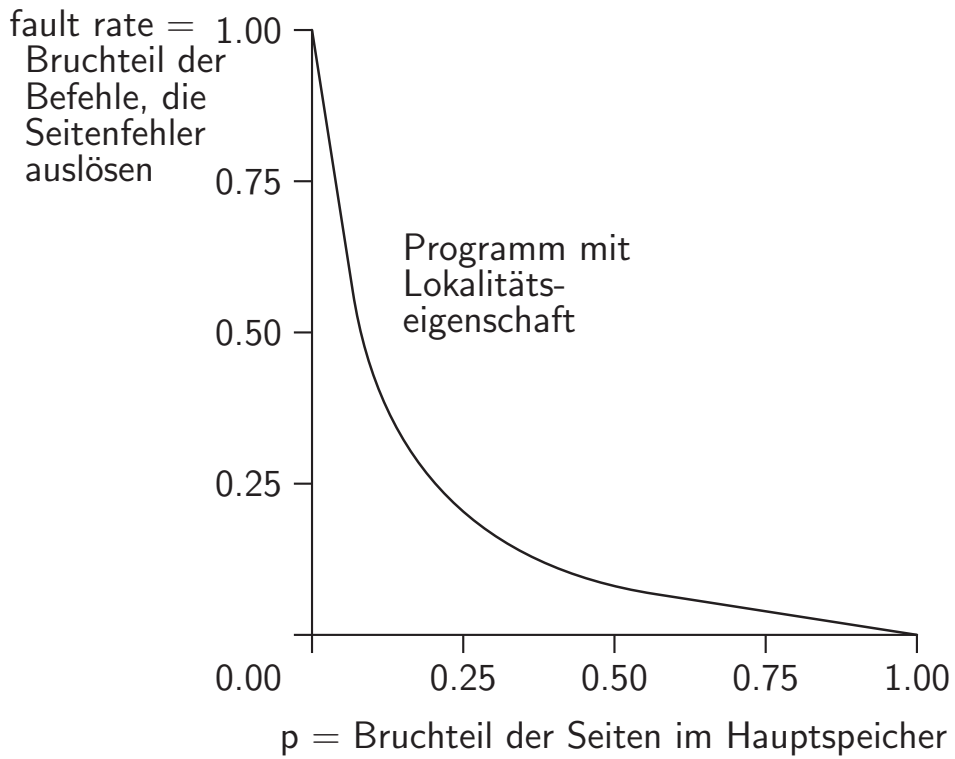


Abb. SPV–16 Das Working Set Modell

- ◇ Wünschenswert: (eingeschränkte) **Aufhebung der Benutzertransparenz!**

  ⇒ Wie programmiert man so, dass das Working Set möglichst klein bleibt ?

- **Gemeinsam verwendete Seiten**
  - Zwei Prozesse, die dasselbe Programm ausführen, können die Code-Seiten teilen.

    Vorteil: Speicherplatzersparnis, Zeitersparnis
    - ◇ problematisch: Prozessterminierung und Swap-out

      $\Rightarrow$ spezielle Verwaltungsstrukturen nötig
  - Daten können nur geteilt werden, falls sie nicht verändert werden.

  - **Copy-on-Write** (CoW):
    - ◇ Bei **fork** muss eigentlich eine vollständige Kopie des Elternprozesses erstellt werden
    - ◇ Effizienter:
      - ▷ Nur die Seitentabelle wird kopiert.
      - ▷ Eltern- und Kindprozess verwenden dieselben Seiten.
      - ▷ **Alle** Seiten werden als *Read-Only* markiert.
      - ▷ Bei Schreibzugriff $\rightarrow$ Schutzverletzung $\rightarrow$ Interrupt !
      - ▷ Entsprechende Seite wird erst jetzt kopiert und Schreiben auf beiden Kopien erlaubt.

DATEISYSTEME

Überblick

- **Anforderungen kommen aus den Anwendungen:**
  - Daten müssen über eine längere (manchmal sehr lange) Zeit aufgehoben werden.
  - Dies muss sowohl für sehr kleine als auch riesige Datenmengen effizient möglich sein.
  - Die Daten müssen von mehreren Prozessen aus zu bearbeiten sein, oft sogar gleichzeitig!
  - Daten dürfen nicht immer von 'fremden' Prozessen aus sichtbar oder veränderbar sein.
- **Beobachtung:** Die Semantik der Daten ist i.Allg. für deren Verwaltung unerheblich.

Definition: Datei

Generischer Container für beliebige digitale Informationen in Form einer Bit-Folge.

- **Aufgaben des Dateisystems:**
  - Organisation des Speicherplatzes auf externen Datenträgern.
  - Bereitstellen von Funktionen zum *Speichern*, *Ändern* und *Wiedergewinnen* von Dateien.
  - Verwirklichung eines **abstrakten Modells**:
    - ◇ **Logische Zugriffsmechanismen**, z.B. datensatzorientierter Zugriff.
    - ◇ Abbildung des abstrakten Dateimodells auf Datenträger, z.B. Band, Platte, CD-ROM.
      - ▷ *Ziel:* Verdeckung der **physikalischen Eigenschaften** der Datenträger.
  - Realisierung wirksamer Schutzmechnismen:
    - ◇ Unterscheidung verschiedener Benutzer und Vergabe von Zugriffsrechten.
    - ◇ Erhaltung der Datenintegrität, z.B. einfache Leser-/Schreibersynchronisation.

Unterschiedliche Dateiarten

- Auftragsbeschreibungen und Makros
- Quellprogramme
- Objektprogramme
- Binärprogramme
- Konfigurationsdateien
- Bibliotheksroutinen
- Auftragsprotokolldateien
- Systemprotokolldateien
- Spool-Dateien
- Gerätedateien
- Archivdateien
- Multimediadateien
- ...

$\Rightarrow$ Der Typ einer Datei wird oft über ihre Endung kodiert.

Schichtenmodell eines Dateisystems

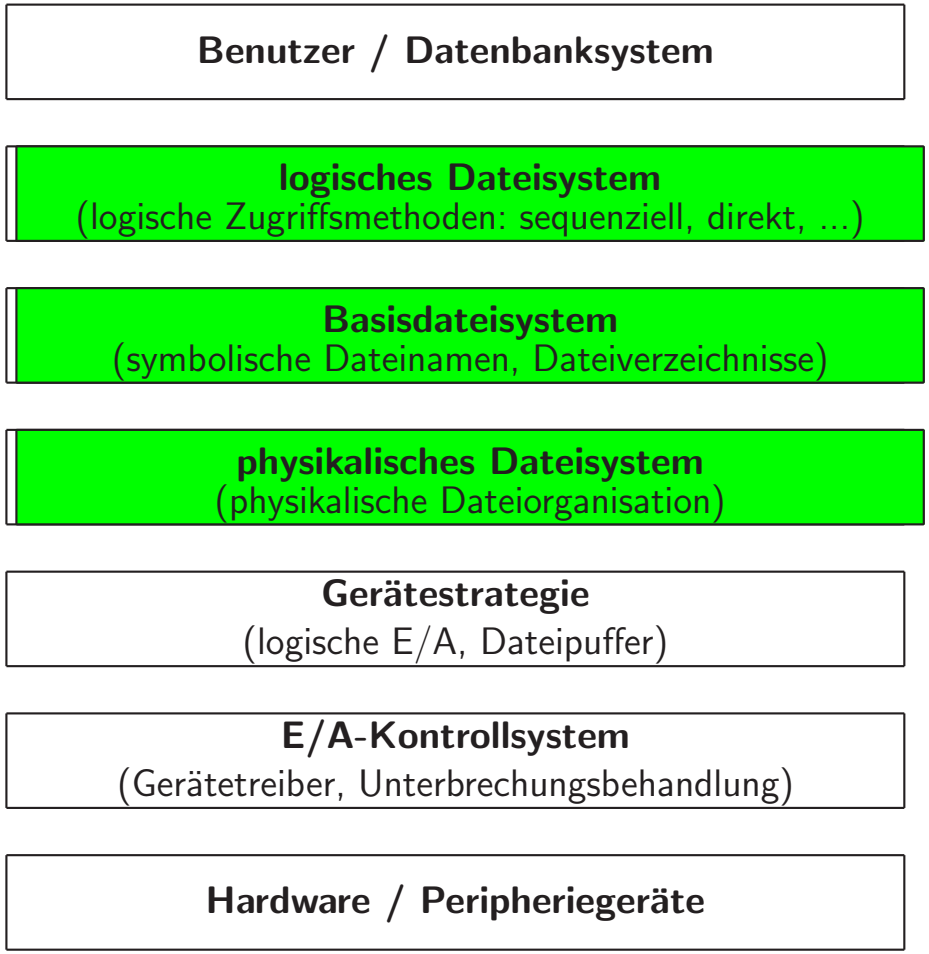
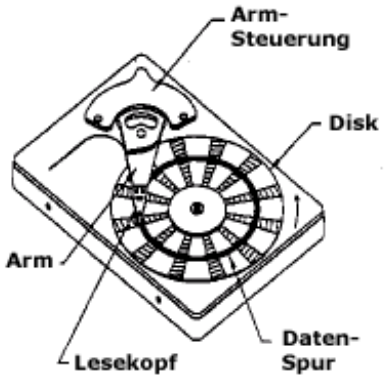


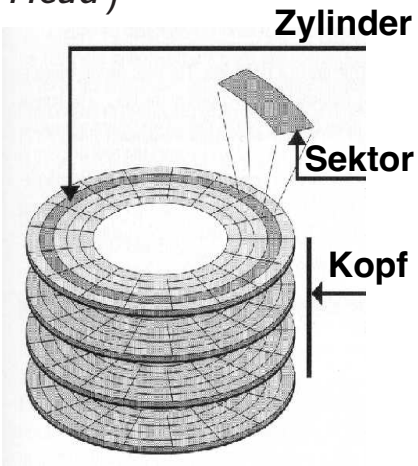
Abb. DAT–1 Schichtenmodell eines Dateisystems

Hardware

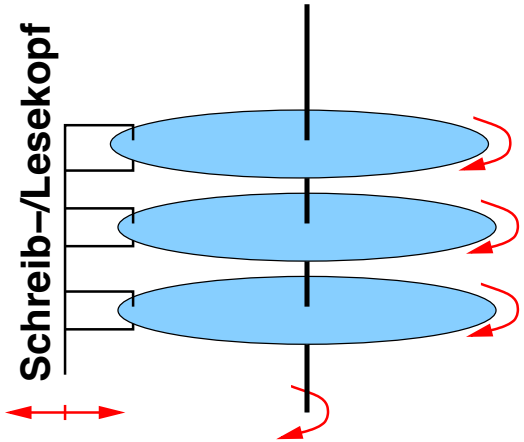
Magnetische Festplatte (HDD)

- Physikalischer Aufbau:
  - **Plattenstapel** (*Disk Pack*) aus mehreren Scheiben, die um eine gemeinsame Achse rotieren.
  - Auf den Oberflächen der einzelnen Scheiben liegen **konzentrische Spuren** (*Track*).
  - Übereinanderliegende Spuren aller Scheibenflächen bilden einen **Zylinder** (*Cylinder*).
  - Spuren sind in **Sektoren** (*Sector*) unterteilt.
  - Sektoren nehmen **Blöcke** (*Block*) auf.
  - Pro Plattenebene: eigener **Lese-/Schreibkopf** (*Read/Write Head*)





- Alle Köpfe werden zusammen mit dem sie tragenden Arm immer nur **als Einheit** nach innen oder außen bewegt.



- Die **Armbewegung** stellt eine vergleichsweise langsame und deshalb **teure Operation** dar.
  $\rightarrow$ ca. 10 ms mittlere Zugriffszeit auf eine beliebige Spur

- Aufbau einer Plattenblockadresse:
  - `cylno, headno, blockno`
    `cylno`: horizontale Einstellung
    `headno`: vertikale Einstellung (entscheidet zwischen Plattenebenen)
    `blockno`: Positionierung innerhalb einer Spur

Solid State Drive (SSD)

- Physikalischer Aufbau
  - Flash-Speichereinheit
  - Controller
    - ◇ Verteilt Write-Zugriffe gleichmäßig über alle Speicherzellen, um Lebensdauer zu erhöhen.

- Vergleich mit magnetischen Festplatten
  - schneller, da direkter Zugriff ohne mechanische Bewegungen
    (Faktor 100 bei der Zugriffszeit, im laufenden Betrieb Faktor 2 – 4)
  - leiser
  - geringerer Energieverbrauch
  - robuster gegen Stöße

  - teurer (ca. Faktor 10 pro GB)
  - i. A. geringere Speichergrößen
  - schnellerer Verschleiß (ca. 3.000 – 100.000 Schreibvorgänge pro Flash-Speicherzelle)
  - gezieltes Löschen einer Speicherzelle i. A. nicht möglich ($\rightarrow$ *Secure Erase*)

Physikalisches Dateisystem

- **Begriffe**
  - **Physikalische Datei:**
    - ◇ Menge von zusammengehörigen Blöcken auf externem Datenträger.
    - ◇ Zuordnung eines **systemweit eindeutigen Bezeichners**, z.B. Durchnummerierung der physikalischen Dateien mit 32-Bit-Werten.
  - **Physikalischer Block:**
    - ◇ Physikalisch zusammenhängender Speicherbereich.
    - ◇ Zuordnung einer physikalischen Adresse (auf einem Datenträger).

- **Aufgaben des physikalischen Dateisystems:**
  - Speicherorganisation auf externen Datenträgern.
  - Verwaltung der Dateien durch Datei-Deskriptoren.
  - Verwaltung der systemweit eindeutigen Dateibezeichner.
  - Abbildung der physikalischen Blocknummern auf physikalische Blockadressen.
    - ◇ Eine Datei besitzt eine Menge physikalischer Blöcke
      → physikalische Plattenblockadressen.
    - ◇ Diese werden (dateibezogen) mit 0 beginnend durchnummeriert
      → physikalische Blocknummern.
  - Bereitstellung wichtiger Dateioperationen:
    - ◇ Erzeugung
    - ◇ Verlängerung, Verkürzung
    - ◇ Löschen

Physikalische Dateiverwaltung auf Plattenspeichern

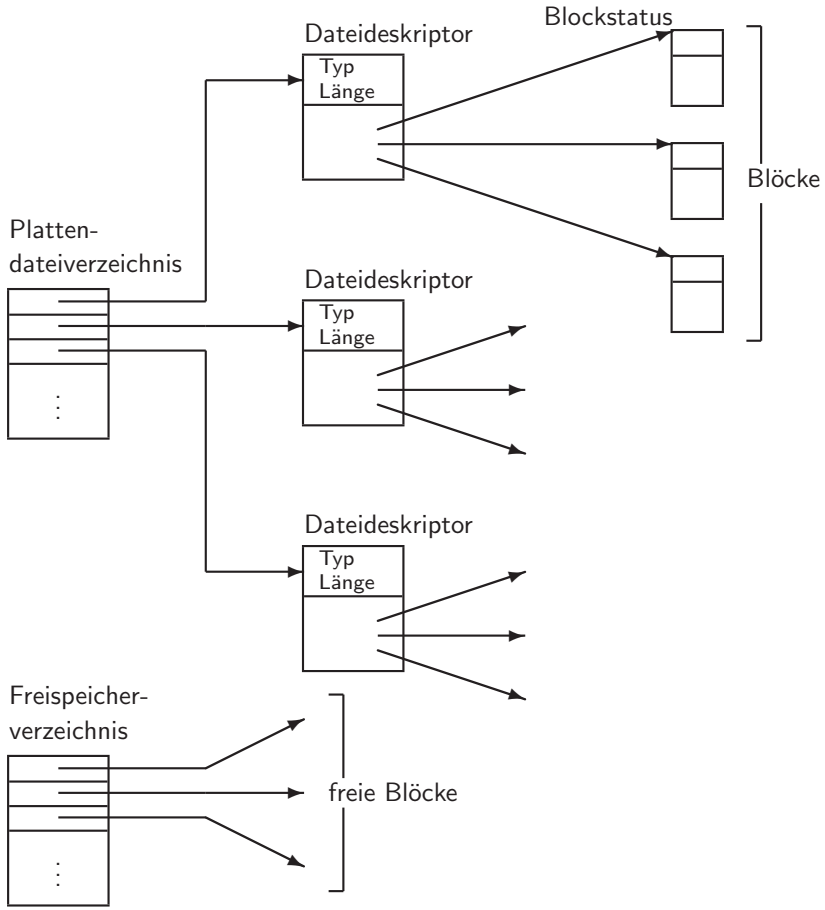


Abb. DAT–2 Physikalische Dateiverwaltung auf Plattenspeichern

- **Plattendateiverzeichnis** (*Volume Directory*):
  - Inhaltsverzeichnis des Datenträgers (hier: der Platte)
  - Liegt auf einer fest definierten Position auf der Platte.
  - Struktur eines Eintrags:
    - ◇ Systemweit eindeutiger Bezeichner
    - ◇ Verweise zu den Dateideskriptoren

- **Dateideskriptor:**
  - Dateityp:
    - ◇ Textdatei/Binärdatei
    - ◇ Verzeichnis
    - ◇ Gerätedatei
    - ◇ Blockdatei (mit eigenem Dateisystem)
  - Physikalische Position der Datei auf der Platte
  - Länge der Datei

- **Freispeicherverzeichnis:**
  - Liste aller nicht von Dateien belegten Speicherbereiche der Platte.
    - **Listenmethode:**
      - Freie Blöcke werden als verkettete Liste organisiert.
      - Ein Zeiger zeigt auf den ersten Block der Liste (Listenkopf).
      - Jeder freier Block enthält Zeiger auf einen weiteren freien Block (oder **null**).
    - **Indexmethode:**
      - Freie Blöcke werden indexiert verwaltet.
      - Der erste freie Block enthält die Zeiger auf freie Blöcke bzw. auf Indexblöcke der nächsten Stufe.
      - Die Indexblöcke können linear oder baumartig verkettet sein.
    - **Vektormethode:**
      - Ein Bit-Vektor gibt für jeden Block an, ob er frei oder belegt ist.

Physikalische Dateiorganisationsformen

- **Zusammenhängende Speicherung**

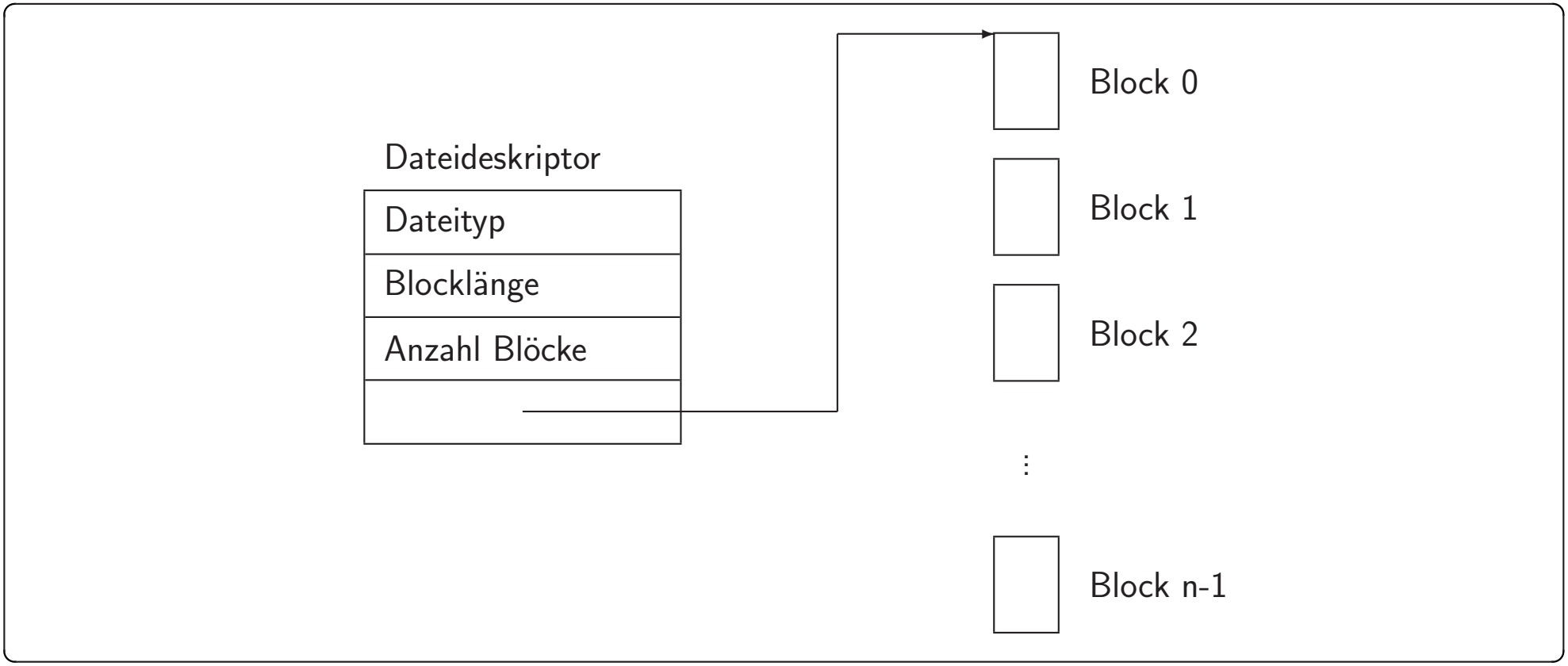


Abb. DAT–3 Zusammenhängende Speicherung

- **Gestreute Speicherung**

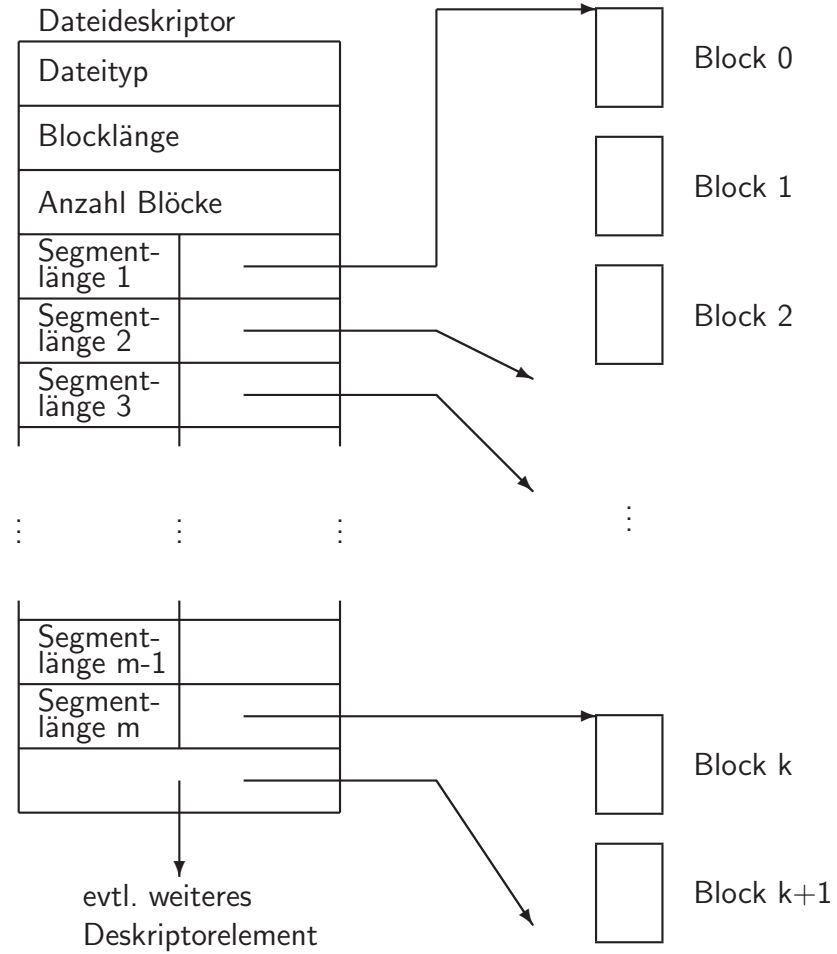


Abb. DAT–4 Gestreute Speicherung

Basisdateisystem

- **Aufgaben des Basisdateisystems:**
  - Abbildung:

    symbolischer Dateiname (benutzerdefiniert) $\rightarrow$ *systemweit eindeutiger Bezeichner*

    $\Rightarrow$ Verwaltung von **Benutzerdateiverzeichnissen**

  - Bereitstellung von Zugriffsmechanismen über eine **logische Blocknummer**.

    $\Rightarrow$ Verdecken der physikalischen Struktur externer Datenträger.

  - Bereitstellung der Dateioperationen:

    Einrichten / *Öffnen* / *Lesen* / *Schreiben* / *Schließen* / *Löschen* / *Umbenennen*

- **(Basis-) Datei**:
  - Menge von **logischen Datenblöcken** mit logischen Blocknummern.
    - Ein logischer Block besteht aus n physikalischen Blöcken.
    - **Vorteil:** Unabhängigkeit von der physikalischen Struktur des Datenträgers

Logisches Dateisystem

- **(Logische) Datei:**
  - Menge von **Datensätzen** (**records**) mit logischer Organisationsform
- **Aufgaben des logischen Dateisystems:**
  - Abbildung von Datensätzen auf Blöcke
  - Bereitstellung logischer Dateistrukturmodelle durch Zugriffsmethoden
    - ◇ sequenzielle Organisation, sequenzieller Zugriff:
      - ▷ totale Ordnung der Datensätze
      - ▷ Zugriff nur in vorgegebener Reihenfolge
    - ◇ wahlfreier Zugriff:
      - ▷ Zugriff auf Datensätze in beliebiger Reihenfolge
      - ▷ indexsequenzielle Organisation
      - ▷ baumsequenzielle Organisation

Schutzaspekte in Dateisystemen

Discretionary Access Control (DAC)

Benutzerkategorien und Zugriffsrechte

- Beispiele für **Benutzerkategorien**
  - S: Systemverwalter (*System*)
  - O: Besitzer der Datei (*Owner*)
  - G: Gruppe des Besitzers (*Group*)
  - W: Allgemeinheit aller Benutzer (*World*)
- Beispiele für **Zugriffsrechte**
  - R: Lesen (*Read*)
  - W: Schreiben (*Write*)
  - E: Ausführen (*Execute*)
  - D: Löschen (*Delete*)

- **Anwendungsbeispiele:**
  - Vax/Vms, vgl. Abb. DAT–5
  - in abgewandelter Form: Unix

- Es wird eine Benutzergruppenverwaltung benötigt.

	R	W	E	D
S	1	0	0	1
O	1	1	1	1
G	1	0	1	0
W	0	0	1	0

Abb. DAT–5 Benutzerkategorien und Zugriffsrechte

Zugriffsmatrix

	Objekt 1	Objekt 2	···	Objekt n
Subjekt 1	R,W		···	
Subjekt 2			···	
⋮	⋮	⋮	⋱	⋮
Subjekt m			···	R,E,D

Abb. DAT–6 Das Zugriffsmatrix-Modell

Problem: Hoher Speicherbedarf, Zugriffsmatrix ist i.d.R. nur **dünn besetzt**.

Zugriffskontrollliste (*Access Control List*)

- Für jedes Objekt speichert man eine Liste derjenigen Subjekte, die Zugriff auf das Objekt haben, sowie die zugehörigen Zugriffsrechte.
- Diese Liste entspricht einer Spalte der Zugriffsmatrix.
- Subjekte ohne Zugriffsrechte werden weggelassen.

Objekt A

Subjekt	Zugriffsrechte
B	ZGR
C	ZGR
D	ZGR
⋮	⋮

Abb. DAT–7 Zugriffskontrollliste

Befähigungsliste (*Capability List*)

- Für jedes Subjekt speichert man eine Liste der Objekte, auf die es zugreifen darf, sowie die zugehörigen Zugriffsrechte (oft in Form von *Tickets* in verteilten Systemen $\Rightarrow$ **Probleme ?**).
- Jede Liste entspricht einer Zeile der Zugriffsmatrix.
- Objekte, auf die ein Subjekt keinen Zugriff hat, werden weggelassen.

Subjekt S

Objekt	Zugriffsrechte
T	ZGR
U	ZGR
V	ZGR
⋮	⋮

Abb. DAT–8 Befähigungslisten

Problematisch: Löschen eines Objekts u.U. aufwändig.

Mandatory Access Control (MAC)

Bell-La Padula, 1973

- Subjekte und Objekte werden klassifiziert
- Zugriffsrechte werden anhand der Klassifikationen automatisch vom System errechnet
- Beispiel einer Klassifikation:
  - *streng geheim (3) / geheim (2) / vertraulich (1) / öffentlich (0)*
  - z.B. lesen eines Objekts: Klassifikation des Subjekts ist mindestens gleich hoch.

Rollenbasiert

- Benutzern werden unterschiedliche Rollen zugeordnet
- Zugriffsrechte ergeben sich dem jeweiligen Kontext und der momentanen Rolle

Namensverwaltung

- Verwendung **hierarchisch** strukturierter Dateinamen (Verkettung von Teilnamen).
- **Vorteile:**
  - Derselbe Name ist in verschiedenen Verzeichnissen verwendbar.
  - Operationen können auf alle in einem Verzeichnis eingetragenen Dateien bezogen werden.
  - Die Verzeichnisstruktur kann eine (unternehmensseitig) vorgegebene Organisationsstruktur widerspiegeln.
    $\Rightarrow$ Dateien eines bestimmten Projekts können in **einem** Verzeichnis gespeichert werden.
- Abb. DAT–9 (S. 24) zeigt die Struktur hierarchischer Dateiverzeichnisse.

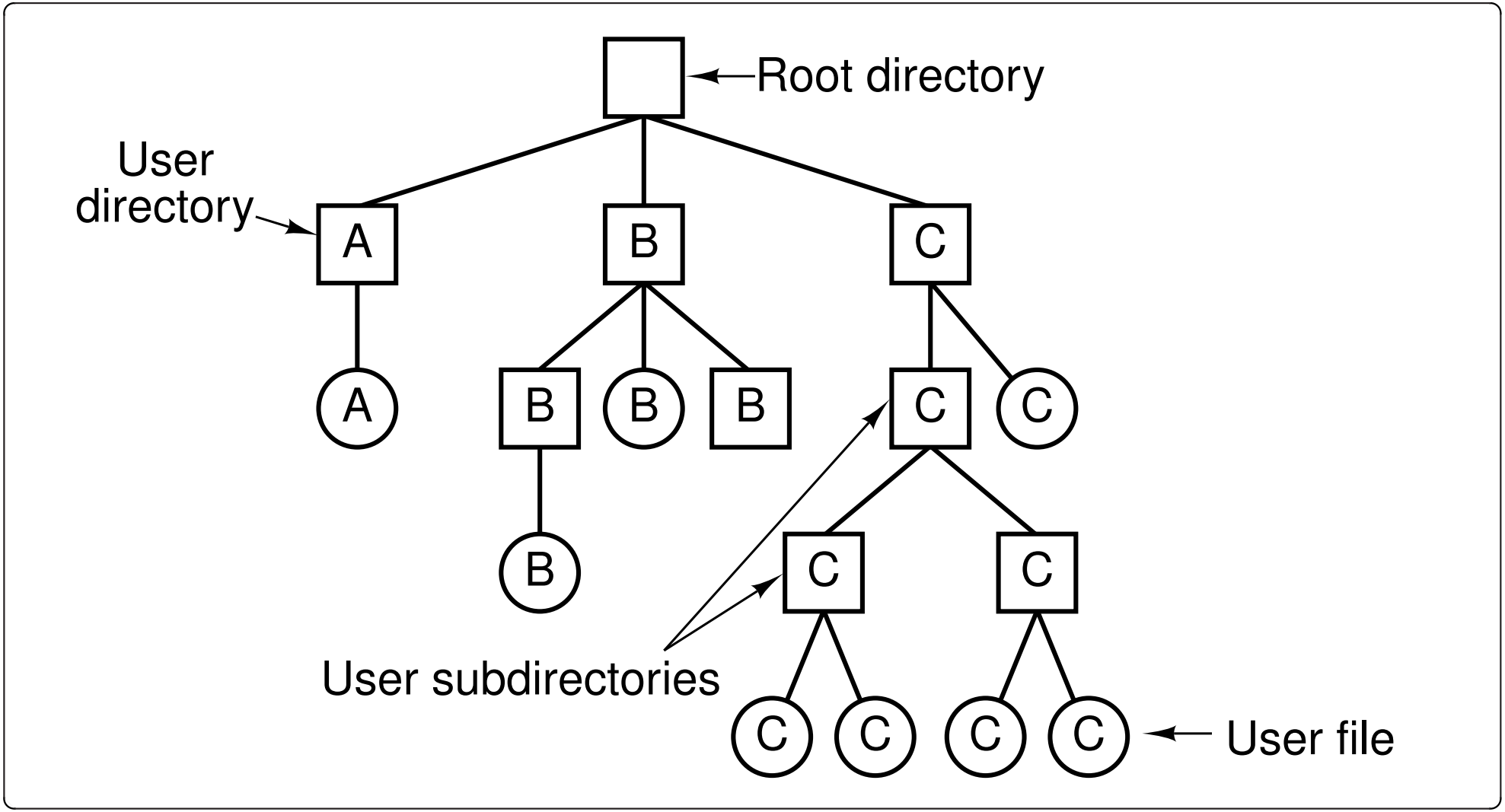


Abb. DAT–9 Hierarchisch strukturierte Dateiverzeichnisse

- **Alternative Datenstrukturen zur Verwaltung der Dateiattribute**
  - (a) im Dateiverzeichniseintrag (z.B. MS-DOS/Windows-Ansatz)
    - ◇ benötigt viel Platz innerhalb des Verzeichniseintrags
  - (b) unabhängig vom Dateiverzeichniseintrag (Unix-Ansatz)
    - ◇ es wird nur ein Verweis auf die entsprechende Verwaltungsstruktur (*i-node*) gespeichert

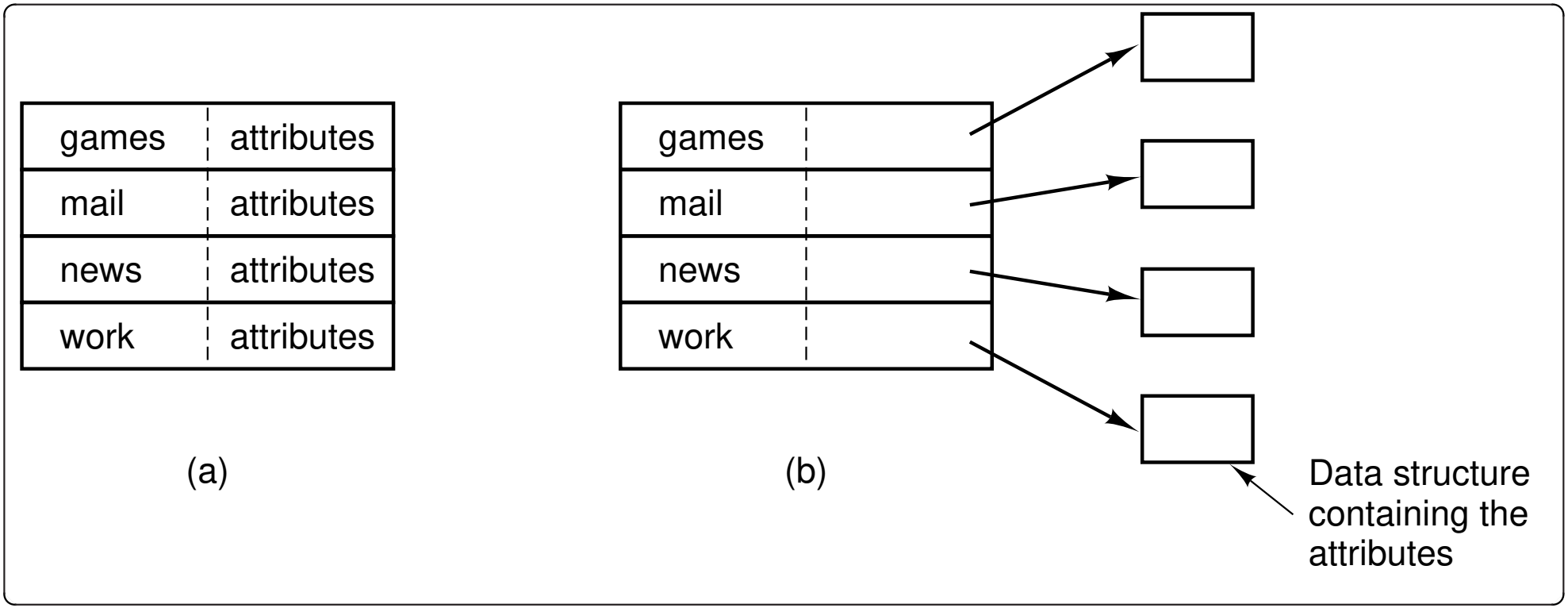


Abb. DAT–10 Verwaltung der Dateiattribute

- **Link: Verschiedene Namen für dieselbe Datei**
  - unter Angabe des Pfadnamens (*Softlink, Symbolic Link*)
  - Dateiverzeicheintrag verweist direkt auf i-node (*Hardlink*)
    - ◇ Resultat: Dateisystem ist nun kein Baum mehr, vgl. Abb. DAT–11

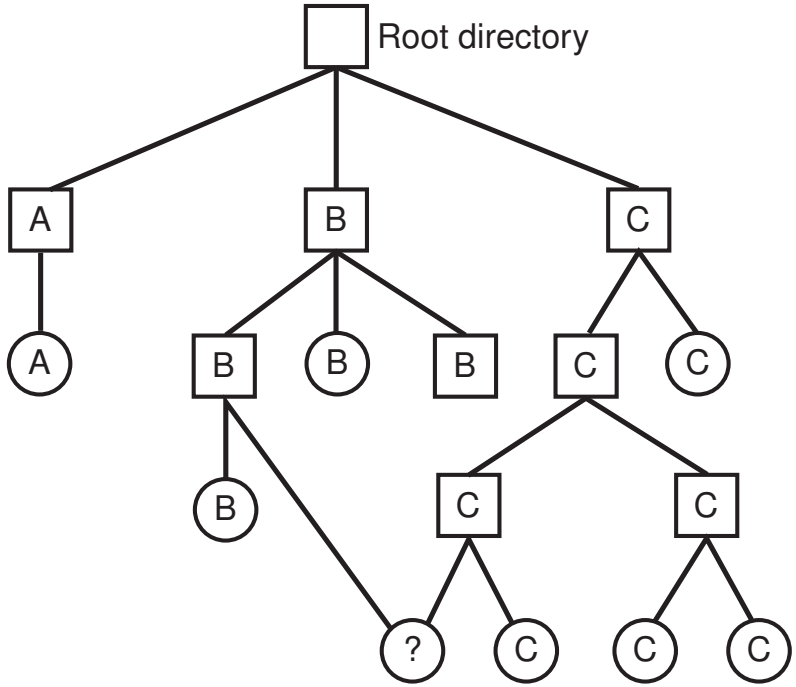


Abb. DAT–11 Dateisystem als (azyklischer) Graph

◇ Wann darf eine Datei tatsächlich gelöscht werden? ($\Rightarrow$ Link Counter)

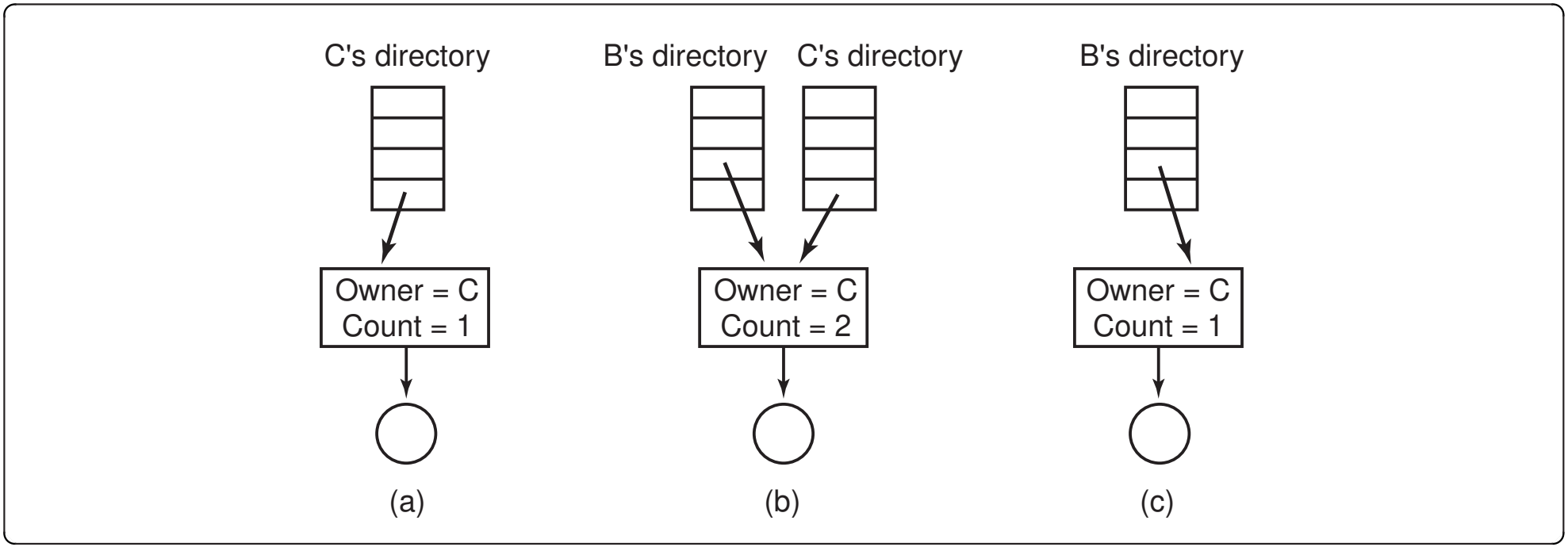


Abb. DAT–12 Anlegen und löschen eines Hardlink

◇ **Frage:** Was passiert, wenn Zyklen entstehen?

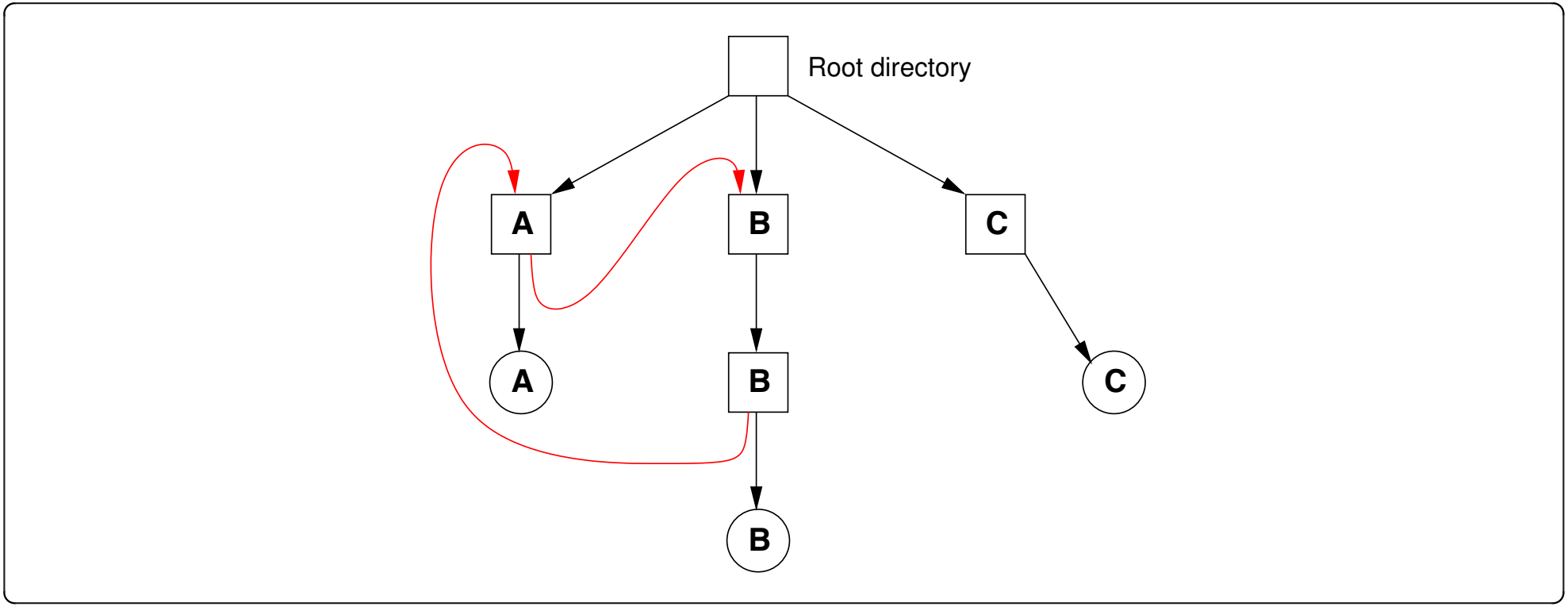


Abb. DAT–13 Möglicher Zyklus mit Hardlinks

- Lösungsvarianten:
  - ▷ aufwändige Zyklenerkennung durchführen
  - ▷ keine Hardlinks auf Verzeichnisse zulassen

Verwaltung von Dateisystemen

- Platzierung mehrerer Partitionen auf einer Platte (vgl. Abb. DAT–14)

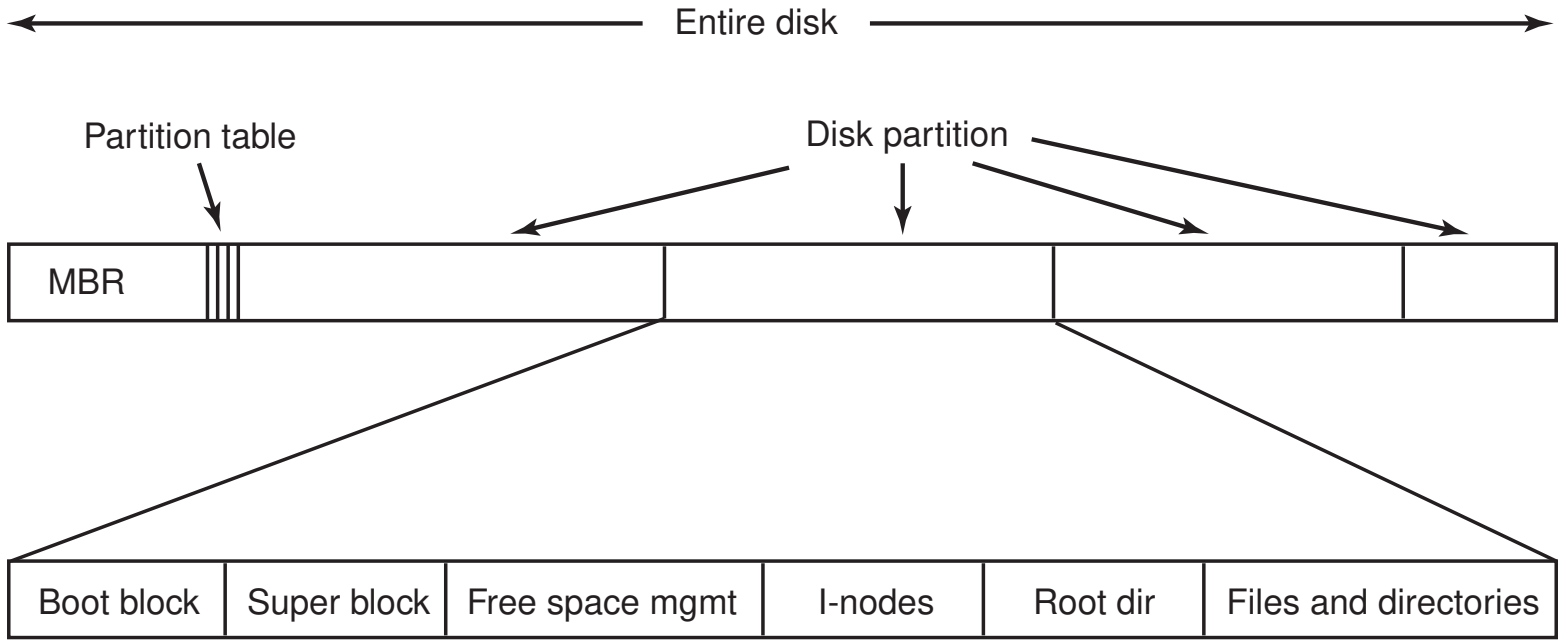


Abb. DAT–14 Beispiel-Layout eines Dateisystems

- **Master Boot Record** (MBR)
  - ◇ wird vom BIOS beim Booten geladen
  - ◇ platziert in Sektor 0
  - ◇ enthält u.a. Partitionstabelle
  - ◇ bezeichnet aktive Partition
- **Boot Block**
  - ◇ pro Partition
  - ◇ lädt das Betriebssystem dieser Partitition (falls vorhanden)
- **Super Block**
  - ◇ pro Partition
  - ◇ enthält:
    - ▷ Typ des Dateisystems
    - ▷ Anzahl der Blöcke
    - ▷ weitere vom Dateisystem abhängige Verwaltungsinformationen

Journal Dateisysteme

- **Caching kann Probleme verursachen:**
  - Um eine akzeptable Zugriffsgeschwindigkeit zu erreichen, werden sog. *File Caches*, d.h. Pufferbereiche im Hauptspeicher verwendet
  - schreiben in den Puffer $\Rightarrow$ zeitweise Inkonsistenzen zur Platte
  - periodisches Zurückschreiben notwendig (*sync*)
  - stürzt der Rechner aber vorher ab, gibt es Probleme!
  - besonders problematisch: Inkonsistenzen innerhalb der i-nodes

- Anforderungen an ein modernes Dateisystem:
  - Handhabung sehr großer Partitionen
  - gute Skalierbarkeit
  - hohe Zugriffsgeschwindigkeit
  - **schnelle Wiederherstellung nach einem System-Crash (!!)**

- **Lösungskonzepte:**
  - Einsatz von *Logs* (*Journals*)
    - ◇ alle Änderungsoperationen (sowie deren Argumente) auf Meta-Daten (i-node, Freispeicherliste, ...) werden im Log persistent festgehalten
      - ⇒ nach einem Crash kann das Log einfach wieder eingespielt werden
    - ◇ einige Journal Dateisysteme: JFS, XFS, ZFS, ext3FS, ext4FS, NTFS
  - Organisation der Verzeichniseinträge mittels eines oder oder mehrerer B+Bäume
  - Organisation der Dateiblöcke mittels B+Bäumen oder B*-Bäumen
  - bei kleineren Dateien werden Daten direkt innerhalb der i-nodes gespeichert (sehr vorteilhaft bei der Speicherung von Soft-Links)

- Maximale Größen moderner Dateisysteme:
  - 4 TiB (ext3FS) bis 16 EiB (XFS)

Fallbeispiele

UNIX - Dateisubsystem

Logischer Dateibaum

- Hierarchisches Dateisystem $\rightarrow$ siehe Abb. DAT–15 (S. 34).
- Erweiterung durch **explizit** definierte **Querverbindungen** (*Symbolic Links*).
- Partitionierung der Festplatte und Einrichtung von Dateisystemen auf den Partitionen.
- Das **Root**-Filesystem enthält die **Urlade**-Informationen (wichtig für den **Boot**-Vorgang).
- Zugriff auf ein bestehendes Dateisystem durch den `mount`-Befehl.

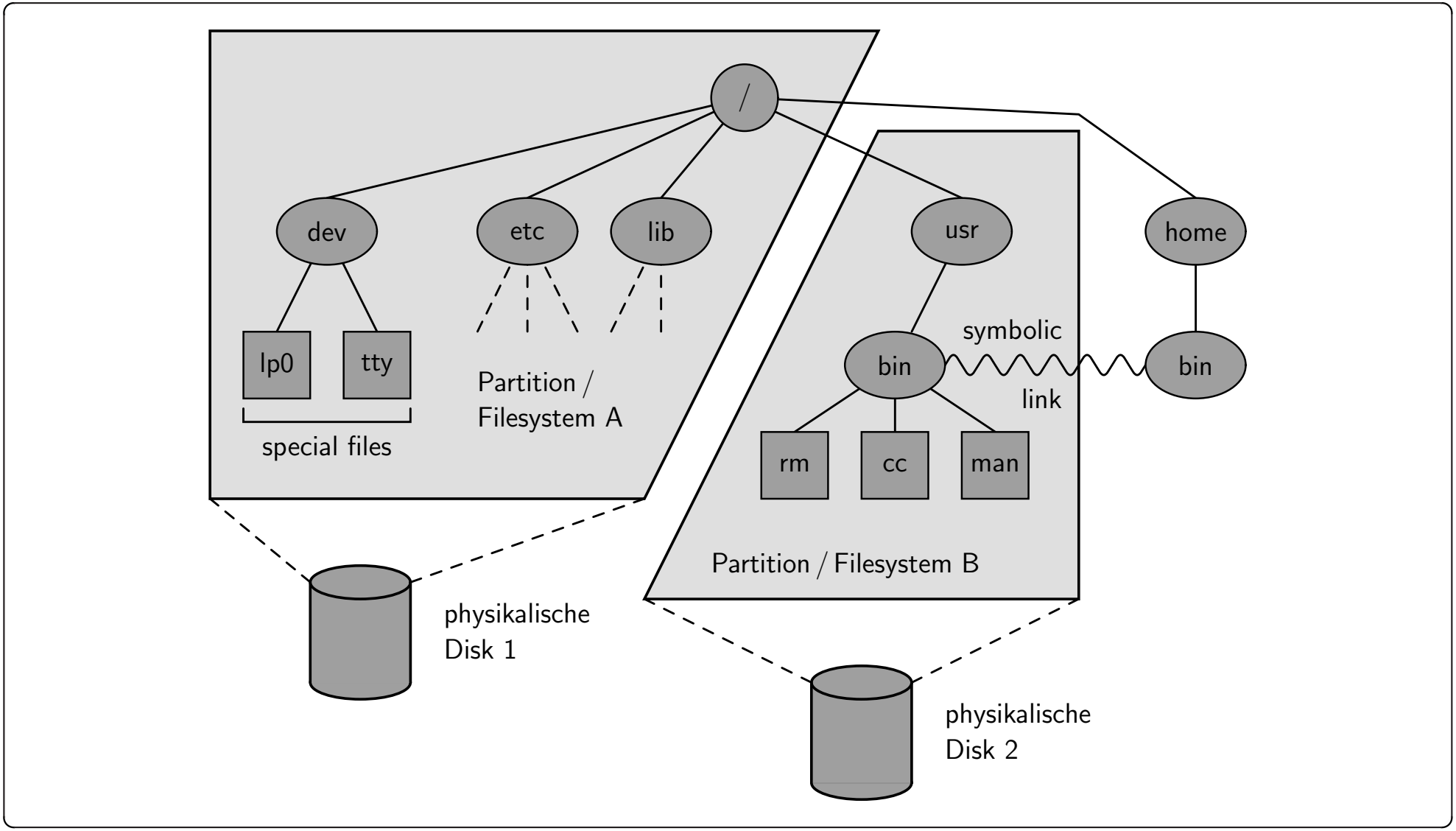


Abb. DAT–15 Beispiel für einen logischen Dateibaum

Interne Darstellung von Dateien

- **Inode** (*Index Node*) $\rightarrow$ siehe Abb. DAT–16 (S. 36).
  - File Deskriptor: Datenstruktur zur Verwaltung einer Datei durch den Systemkern.

- **Inhalt eines Inode:**
  - Beschreibung der Dateiattribute.
  - Liste von Verweisen auf die physikalischen Dateiblöcke.

- **Zugriff auf eine Datei:**
  - Systemkern lädt den zugehörigen Inode in den Systemadressraum.
  - Überprüfung der Zugriffsrechte.
  - Zugriff auf die Datenblöcke:
    - ▷ Jeder Inode hat zehn Einträge, die (jeweils) direkt die Blockadresse des zugehörigen Datenblocks enthalten.
    - ▷ Verwendung von **Ein**-, **Zwei**- und **Dreifach**-Indirektionen für große Dateien.

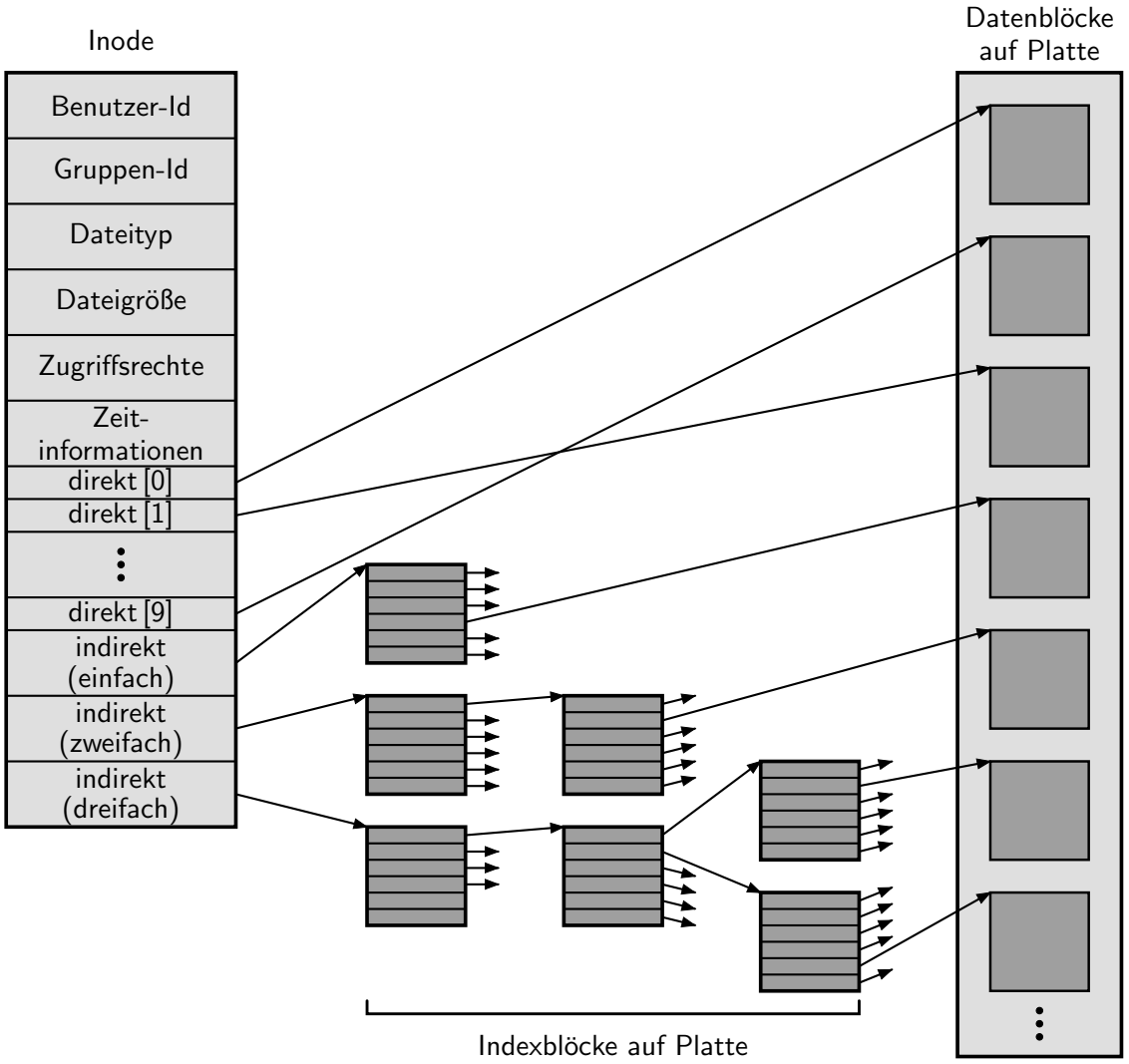


Abb. DAT–16 Struktur eines Inodes

Zugriffskontrolle

- **Zugriffsrechte:**
  - Lesen (r)
  - Schreiben (w)
  - Ausführen (x)
- **Definition jeweils für:**
  - Dateibesitzer (u)
  - Gruppenmitglieder (g)
  - alle anderen Systembenutzer (o)
- Bedeutung bei Verzeichnissen:
  - **r**: Recht zum Auflisten des Verzeichnisinhalts
  - **w**: Recht zum Umbennen, Löschen und Anlegen von Dateien im Verzeichnis
  - **x**: Recht, das Verzeichnis zum Arbeitsverzeichnis machen
- Die Zugriffsrechte können vom Dateibesitzer eingestellt werden (`chmod`).

- **Beispiel:** `ls -l`

```
-rw-r-----    1 pagnia   kurs      416815 Aug 30 11:03 skript.pdf
```

  - Zugriffrechte für die Datei **skript.pdf**:

Besitzer: **pagnia**	Rechte: **r w –**	→	lesen und schreiben
Gruppe: **kurs**	Rechte: **r – –**	→	lesen
Rest:	Rechte: – – –	→	keine Rechte

- **Eine Besonderheit:** SUID-Bit
  - Vornehmen einer Rechteübertragung
    - ◇ Prozess erhält Rechte des Besitzers der aufgerufenen Programmdatei für die Dauer der Ausführung
    - ◇ Beispiel: *shutdown* mit SUID *root*
  - SGID-Bit analog

ext3-Dateisystem

- kompatibel zu ext2
- Unterstützung großer Partitionen (max. 16 TiB)

- Blockgröße: i. Allg. 4 KiB (min. 512 Byte)
- max. Dateigröße: 2 TiB (wegen System Call Schnittstellen oft jedoch nur 2 GiB)

- inode-Größe: 128 Byte (oder Vielfaches)
  $\Rightarrow$ optional erweiterte Attribute möglich, z. B. ACLs
- pro inode 12 direkte Zeiger auf Blöcke sowie 1- , 2- und 3-fache Indirektionen

- Verzeichniseinträge als Bäume, Superblock und mehrere Sicherheitskopien
- Journal für Metadaten (optional auch für Daten selbst $\Rightarrow$ langsam!)
- Verwendung von Blockgruppen zur Reduzierung der Kopfpositionierungszeiten
- kein Defragmentierungs-Tool (Workaround: tar −c | tar −x)

- seit 2008: ext4FS (bis zu 1 EiB Datei- und Dateisystemgröße, verbesserte Performance, ...)
- Alternativen: XFS; Btrfs (B-tree FS) von Oracle unter GPL
  - Beide 8 EiB max. Dateigröße sowie 16 EiB Dateisystemgröße

MS-DOS Dateisystem

- **Überblick:**
  - hierarchisches Dateisystem
    - ◇ Gerätename ist Teil des Pfadnamens einer Datei
  - keine Mehrprogramm-Unterstützung

- **File Allocation Table (FAT)**
  - dient der Verwaltung der Dateien und der freien Blöcke
  - 1 Eintrag pro Plattenblock
    - ⇒ FAT-16: mit 16 Bits sind 65 536 Blöcke adressierbar
  - FAT muss vollständig im Hauptspeicher liegen
    - ◇ Änderungen müssen sofort in der FAT nachgefahren werden
  - freie Plattenblöcke werden (nur) als "frei" markiert
  - maximale Größe einer Partition:
    - ◇ 2 GiB (bei FAT-16, Blockgröße 32 KiB)
    - ◇ 2 TiB (bei FAT-32, Blockgrößen ab 8 KiB)

- **Verzeichniseintrag:**
  - Dateinamen:
    - ◇ max. 8 Zeichen + max. 3 Zeichen Erweiterung (optional)
    - ◇ keine Unterscheidung von Groß- und Kleinschreibung
  - Dateiattribute:
    - ◇ *read only*: Datei kann nicht geändert werden
    - ◇ *archive*: Datei wurde nach dem letzen Archivieren geändert
    - ◇ *hidden*: Datei kann nicht aufgelistet (*dir*) werden
    - ◇ *system*: Datei kann nicht gelöscht (*del*) und auch nicht aufgelistet werden
  - Zeit und der Datum der letzten Änderung
  - Dateigröße
  - Zeiger auf den ersten Plattenblock bzw. in die FAT

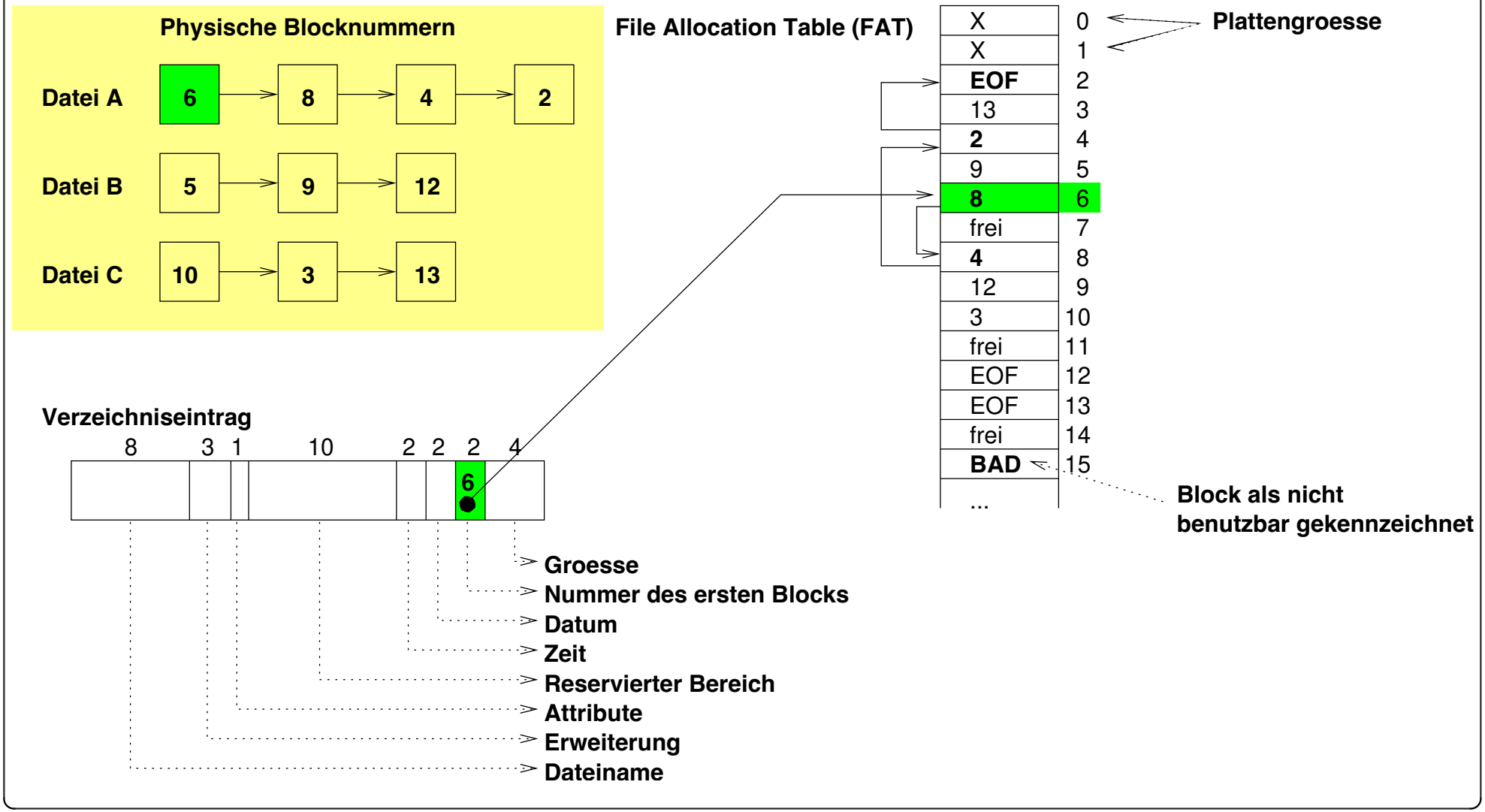


Abb. DAT–17 Aufbau des MS-DOS Dateisystems

NTFS

- **Überblick:**
  - hierarchisches Dateisystem für Windows ab Win NT
  - Blockgrößen: 512 Byte - 64 KiB; i.Allg. 4 KiB
  - Dateinamen: Groß- / Kleinschreibung (Unicode), max. 255 Zeichen
  - Unterstützung von Multiprocessing
  - Unterstützung großer Platten
  - symbolische Links und Hardlinks
  - optionale Komprimierung
  - optionale Verschlüsselung

- **Master File Table (MFT)**
  - Datenstruktur (Datei) zur Verwaltung der Metainformationen pro Partition
  - Verweis auf Wurzelverzeichnis
    - ◇ Verweis auf Freiliste
    - ◇ Verweis auf Sicherungskopie der MFT
    - ◇ ...
    - ◇ 1 Eintrag (Größe: 1 KiB) pro Datei bzw. Verzeichnis:
      - ▷ Liste von Verweisen auf Dateiattribute (Name, eindeutige OID, Zeitstempel, ...)
      - ▷ Liste der Plattenblockadressen
        - * ein **Run** (aufeinanderfolgende Blöcke) bzw. mehrere Runs
        - * mehrere MFT-Einträge für große Dateien notwendig
      - ▷ ...

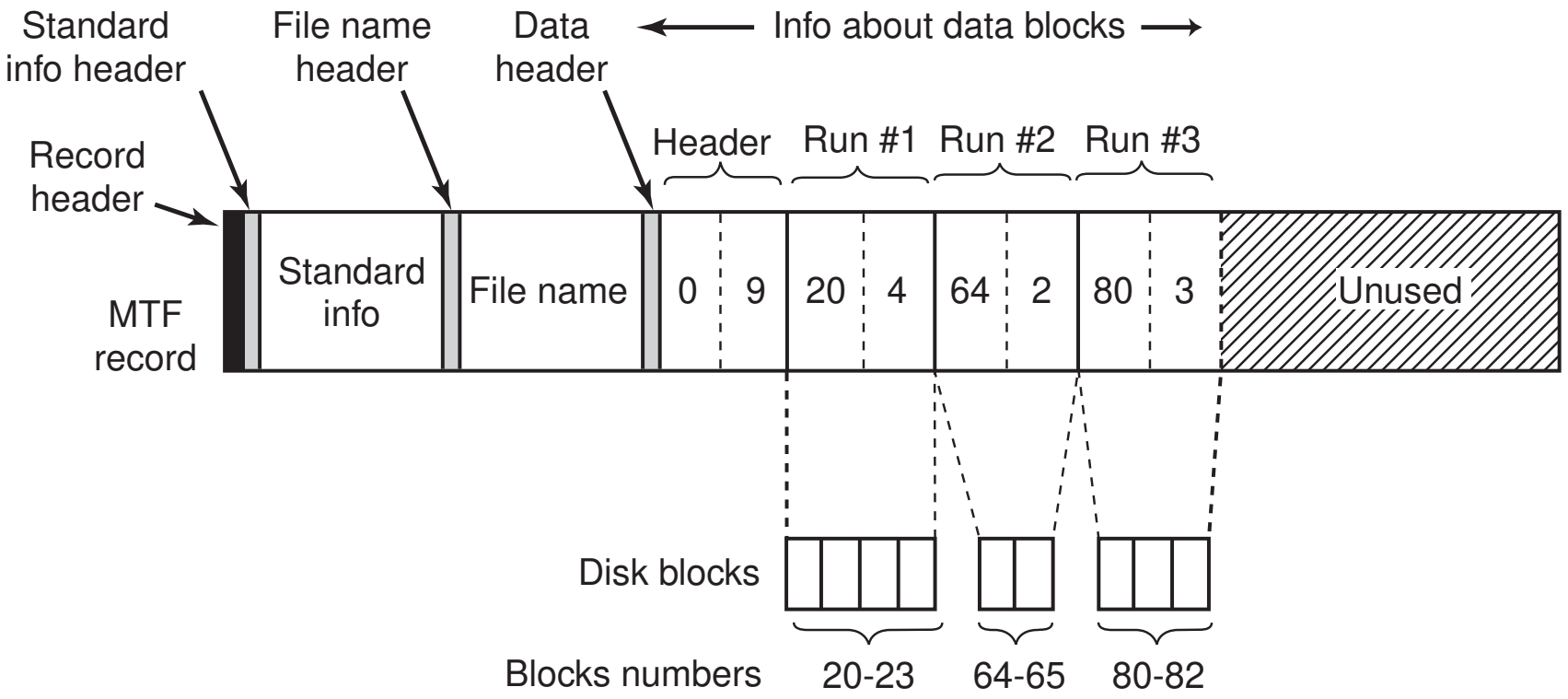


Abb. DAT–18 MFT-Eintrag für eine Datei mit 3 Runs und 9 Blöcken

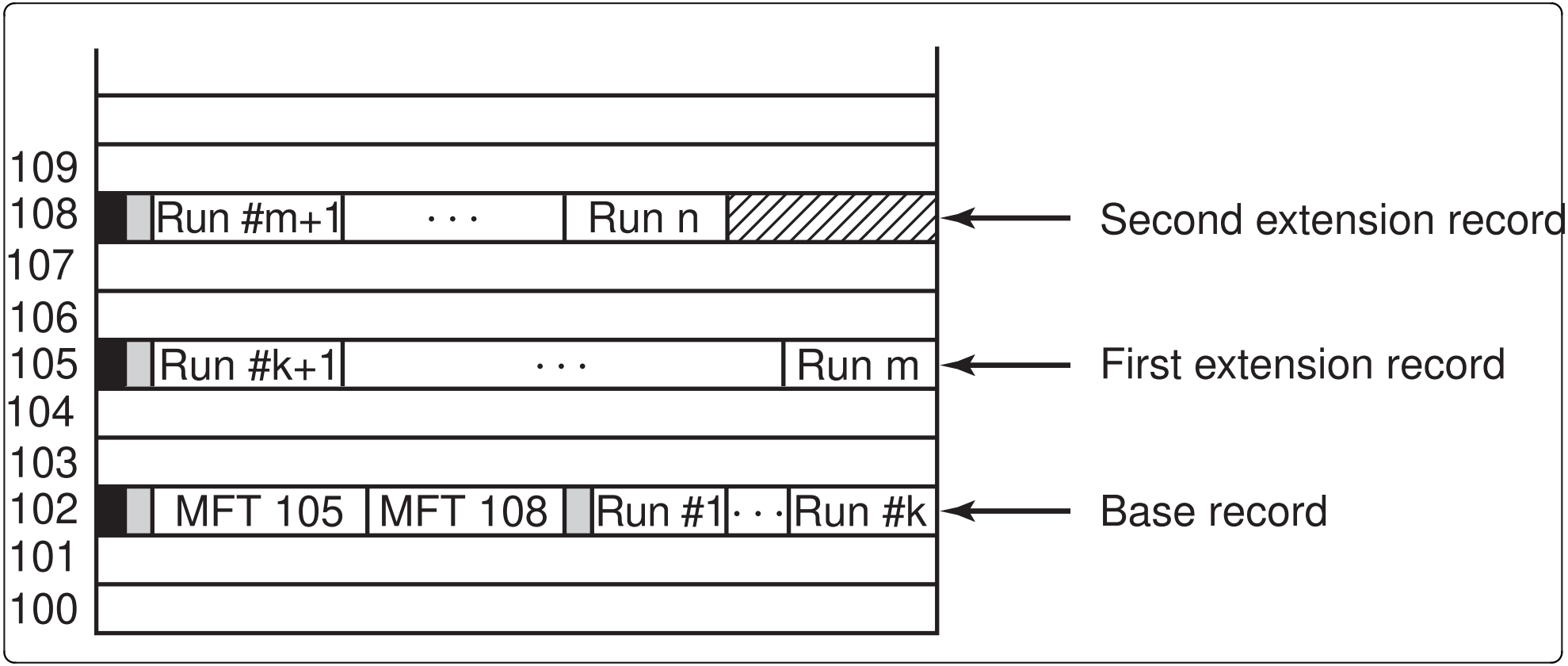


Abb. DAT–19 MFT-Einträge einer größeren Datei mit 3 MFT-Einträgen