Grundlagenkurs Betriebssysteme

Grundliteratur:

S. Tanenbaum: *Moderne Betriebssysteme* (Pearson Studium – IT)

R. Brause: Betriebssysteme (Springer)

U. Baumgarten, H. J. Siegert: Betriebssysteme: Eine Einführung (Oldenbourg)

Web-Links:

Prof. Jürgen Plate: *Einführung in Betriebssysteme*
https://www.tu-chemnitz.de/informatik/friz/Grundl-Inf/Betriebssysteme/Script

Gunnar Teege: *Betriebssysteme*
https://www.unibw.de/interdependenz/inf3/lehre/archiv/ehem.-inf3/vorl08/bs/skript/at_download/down1

Johann Schlichter: *Grundlagen: Betriebssysteme und Systemsoftware*
http://docplayer.org/3556898-Grundlagen-betriebssysteme-und-systemsoftware-gbs.html (online lesen oder Download)

Viele der behandelten Begriffe auch in Wikipedia ausführlich diskutiert.

Allgemeine Hinweise

Hinweis zu den Folien:

- Die Folien sind kein vollständiges Skript und genügen normalerweise nicht zur Prüfungsvorbereitung oder als Nachschlagewerk!
- Sie sollten sich deshalb auf jeden Fall zumindest mit der aufgeführten Basis-Literatur beschäftigen
- Bemerkung am Rande: Diese Folien sind zum Teil aus Folien anderer Kollegen (auch anderer Hochschulen) zusammengestellt.

  Credits to: Alfred Strey, Johann Schlichter, Peter Puschner, Margarita Esponda u. a.

Betriebssysteme: Einführung

Inhalt:

- Einführung
  - Historischer Überblick
  - Parallele Systeme, Modellierung, Strukturen
- Prozesse
- Speicherverwaltung

Definition eines Betriebssystems

Was ist ein Betriebssystem ?

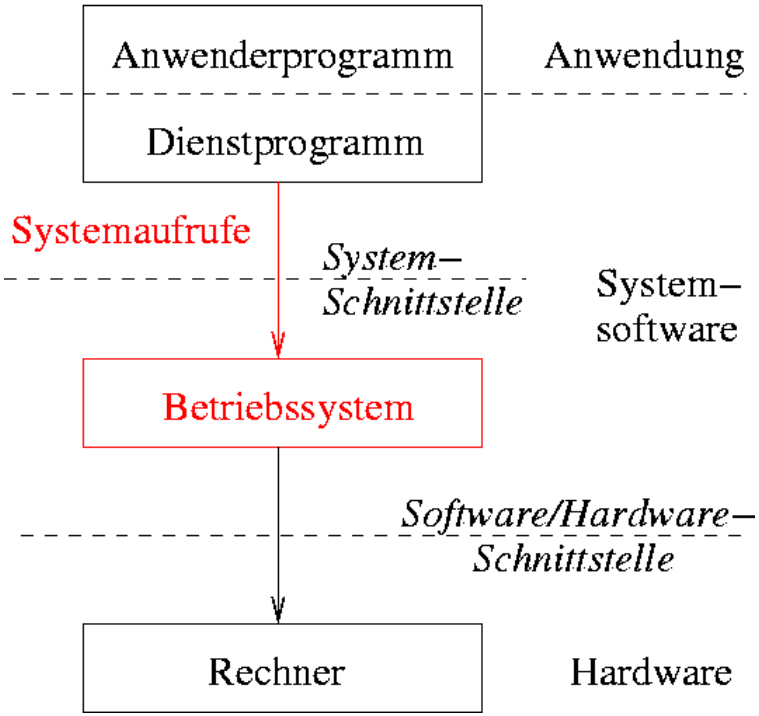
- einfache Definition:

  „*Als Betriebssystem bezeichnet man die Software, die den Ablauf von Programmen auf der Hardware steuert und die vorhandenen Betriebsmittel verwaltet.*“

- zu einem Betriebssystem gehört ggf. auch die Bereitstellung von Dienstprogrammen (z.B. einfacher Editor, Übersetzer, Datenfernverarbeitung, Netzverwaltung)

- Systemschnittstelle stellt für den Anwender abstrakte Maschine dar

einfaches Schichtenmodell:



Anforderungen an ein Betriebssystem

- **hohe Zuverlässigkeit**
  - Korrektheit
  - Sicherheit
  - Verfügbarkeit
  - Robustheit
  - Schutz von Benutzerdaten
- **hohe Benutzerfreundlichkeit**
  - angepaßte Funktionalität
  - einfache Benutzerschnittstelle
  - Hilfestellungen
- **geringe Kosten**
- **hohe Leistung**
  - gute Auslastung der Ressourcen des Systems
  - kleiner Verwaltungsoverhead
  - hoher Durchsatz
  - kurze Reaktionszeit
- **einfache Wartbarkeit**
  - einfache Upgrades
  - einfache Erweiterbarkeit
  - Portierbarkeit

Betriebsarten von Betriebssystemen

Klassifikation nach Art der Auftragsbearbeitung:

- Stapelverarbeitung („*Batch Processing*“)
- Interaktiver Betrieb („*Interactive Processing*“)
- Echtzeitbetrieb („*Real Time Processing*“)

weitere Möglichkeit der Klassifikation:

- Einbenutzer- / Mehrbenutzerbetrieb („*single-user / multi-user*“)
- Einprogramm- / Mehrprogrammbetrieb („*uniprogramming / multiprogramming*“)
- Einprozessor- / Mehrprozessorbetrieb („*uniprocessing / multiprocessing*“)

Aufgaben eines Betriebssystems

- Steuerung der E/A-Geräte
  - Bereitstellung von universellen Gerätetreibern
  - Überwachung der E/A-Geräte
- Bereitstellung eines Dateisystems
- Benutzerschnittstelle
  - Kommandosprache
  - ggf. graphische Oberfläche
- Verwaltung der Betriebsmittel bei Mehrprogrammbetrieb
  - Speicherverwaltung (Swapping, Paging, virtueller Speicher)
  - Prozessverwaltung (Scheduling)
  - Geräteverwaltung (Zuteilung, Freigabe)
  - ggf. auch Prozessorverwaltung (bei Mehrprozessorbetrieb)
- Schutz der Anwenderprogramme bei Mehrprogrammbetrieb

Betriebssystemdienste

Minimale Dienste aus der Sicht eines Benutzers:

- Benutzerschnittstelle
  - CLI Kommandozeilen-Interpreter
  - Batch-Schnittstelle
  - Graphische Benutzerschnittstelle
- Programmausführung
- Ein- Ausgabeoperationen
- Dateiverwaltung
- Kommunikation
  - Gemeinsame Speicher
  - Nachrichtenverkehr
- Fehlererkennungs-System

Shell Scriptsprache

CDE X-Windows

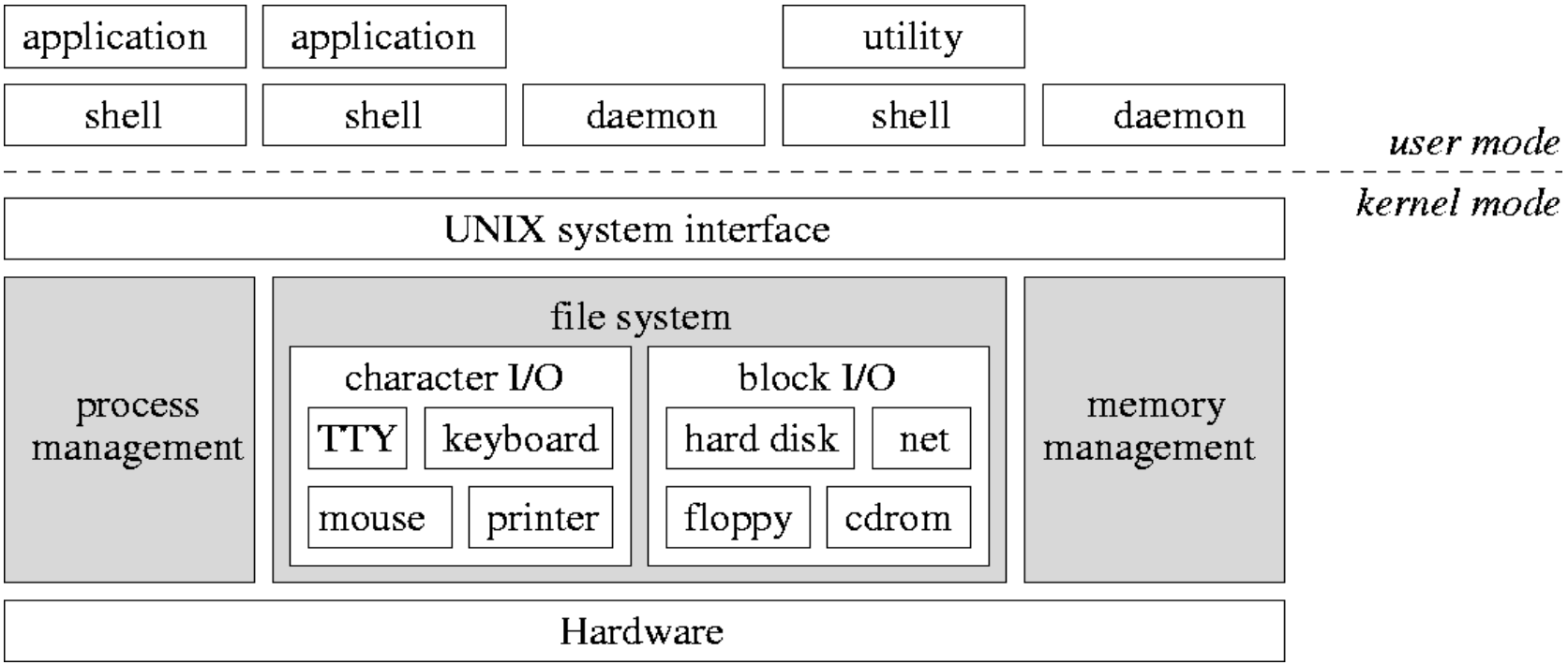
KDE Gnome

Beispiel: UNIX

- Früher: Hardwarespezifische Betriebssysteme (z.B. VAX VMS, IBM OS/360, MS-DOS)
- Heute: überwiegend hardwareunabhängige Betriebssysteme (z.B. MVS, VM, UNIX-Derivate wie AIX, Solaris, Linux)
- historische Entwicklung von UNIX:
  - 1969 von K. Thompson in den AT&T Bell Laboratories für PDP-7 entwickelt und in Maschinensprache implementiert
  - 1973 von D. Ritchie für PDP-11 größtenteils in C umgeschrieben ($\Rightarrow$ leichte Portierbarkeit, nur kleiner maschinenabh, Betriebssystemkern)
  - 1982 – 1995 zwei Linien der Weiterentwicklung: UNIX System V (AT&T) und BSD4.x UNIX (University of California in Berkeley)
  - 1987: erster UNIX-Standardisierungsvorschlag POSIX
  - 1991: Linus Torvalds schreibt einen neuen Betriebssystemkern Linux 0.01
  - seit 1995: OSF legt UNIX Standards fest (UNIX95, UNIX98)

Beispiel: UNIX (Forts.)

- Komponenten von UNIX:



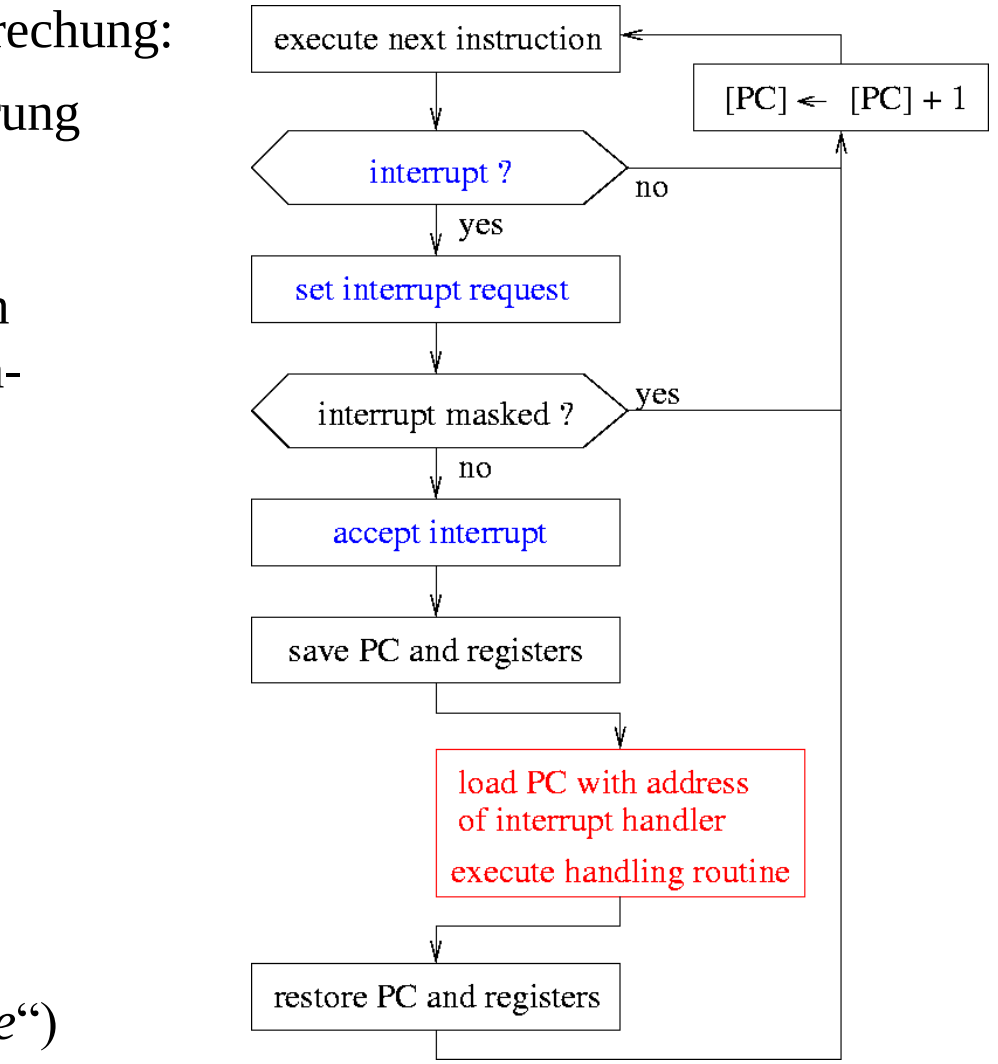
- System- und Benutzerprogramme werden einheitlich als Prozesse ausgeführt (jedoch mit anderen Prioritäten und Zugriffsrechten)
- Dateien und E/A-Geräte werden logisch einheitlich behandelt

Unterbrechungen („*Interrupts*“)

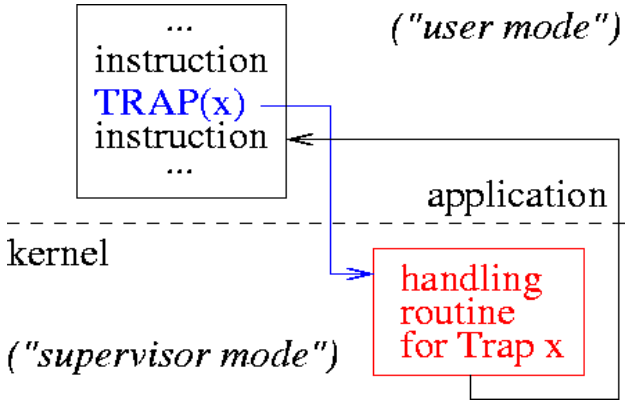
- Ziel: Parallelität von Ein-/Ausgabe und Programmausführung
- Idee: Prozessor initialisiert Ein-/Ausgabe (z.B. Transfer eines Datenblocks), die von E/A-Werk selbständig und überlappend mit Programm ausgeführt wird
- zwei Arten der Unterbrechungen:
  - Möglichkeit für E/A-Werk, Prozessor über Zustand eines E/A-Gerätes zu informieren, z.B. über den Abschluß eines Transfers oder über ein neu empfangenes Datenpaket (externe Unterbrechung)
  - Möglichkeit für Prozessor, Ausnahmebehandlungen durchzuführen, z.B. bei Division durch 0 (interne Unterbrechung)
- die meisten moderne Prozessoren verfügen über Instruktionen, um Unterbrechungen zu verbieten (maskieren) bzw. zu erlauben

Unterbrechungen (Forts.)

- allgemeiner Ablauf einer Unterbrechung:
- Unterbrechung nur nach Ausführung einer jeden Instruktion



- interne Unterbrechung kann auch der Implementierung von System-aufrufen („*Traps*“) dienen:



- häufig erweiterer Befehlssatz für Systemaufrufe („*supervisor mode*“)

Betriebssysteme: Prozesse

Inhalt:

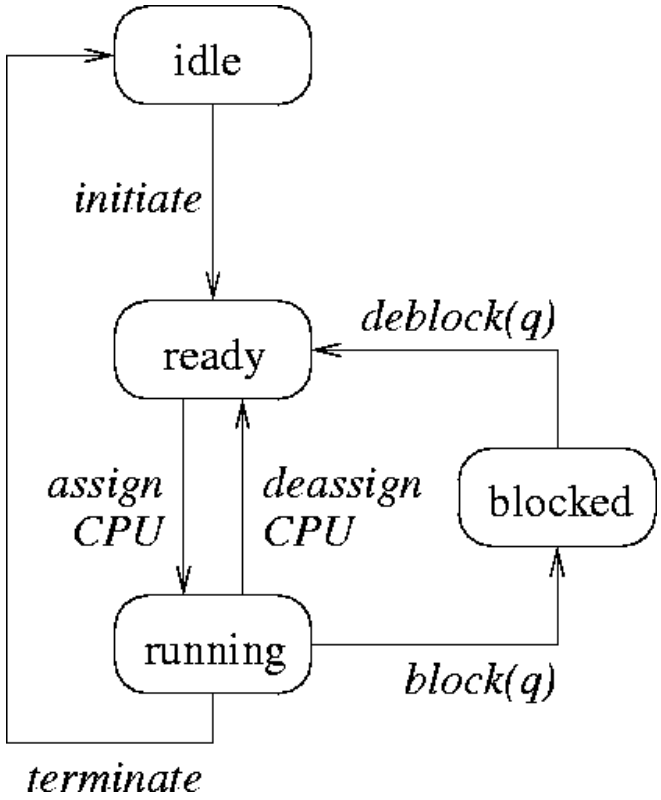
- Einführung
- Prozesse
  - Prozess- und Prozessorverwaltung (*Scheduling*)
  - Prozesskommunikation, Konflikte
- Speicherverwaltung

Prozessbegriff

- einfache Definition:
  „*Ein Prozess ist ein Programm während der Ausführung im Arbeitsspeicher einschließlich seiner Umgebung*.“
- Umgebung (oder Kontext) eines Prozesses:
  - Inhalt vom Programmzähler (PC)
  - Inhalt von Daten-, Adress- und Status-Registern
  - Daten im Speicher (Inhalt aller Programmvariablen)
  - Programmcode
- ein Prozess kann einen anderen Prozess erzeugen; diese werden als ElternProzess bzw. KindProzess bezeichnet
- Ein Prozess kann unterbrochen werden
- zu einem Zeitpunkt kann in einem Einprozessorsystem nur ein Prozess aktiv sein

Zustände eines Prozesses

- vier grundlegende Prozesszustände:
  - bereit („*ready*“): Prozess ist zwar ausführbar, aber Prozessor belegt
  - aktiv („*running*“): Prozess wird auf Prozessor ausgeführt
  - blockiert („*blocked*“): Prozess wartet auf ein externes Ereignis q
  - inaktiv („*idle*“): Prozess wurde gerade erzeugt oder ist terminiert
- Scheduler steuert die Übergänge „*assign/deassign*“
- für die Prozesszustände „*ready*“ und „*blocked*“ werden i.a. eigene Warteschlangen bereitgestellt



Prozesswarteschlangen

Bereit-Warteschlange:

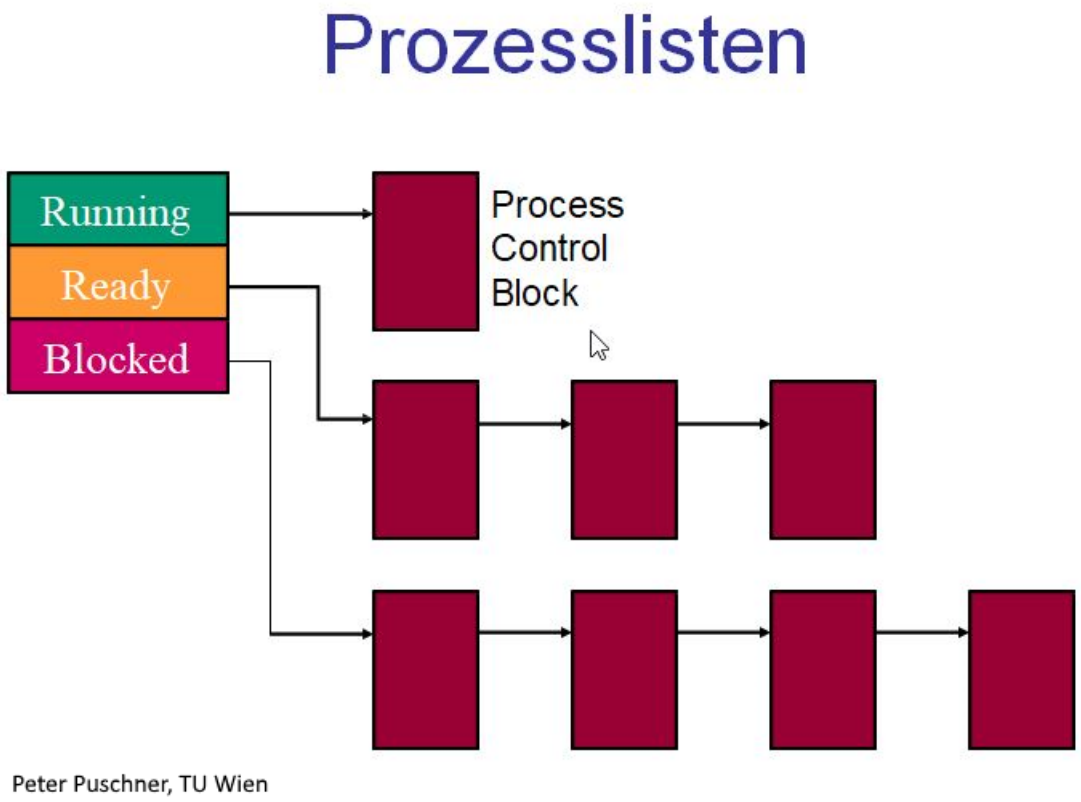
Enthält alle „*ready*“ Prozesse in der Reihenfolge ihrer Ankunft.

Aktivierung des vordersten Prozesses bei Freiwerden der CPU

BlockiertListe (unsortiert):

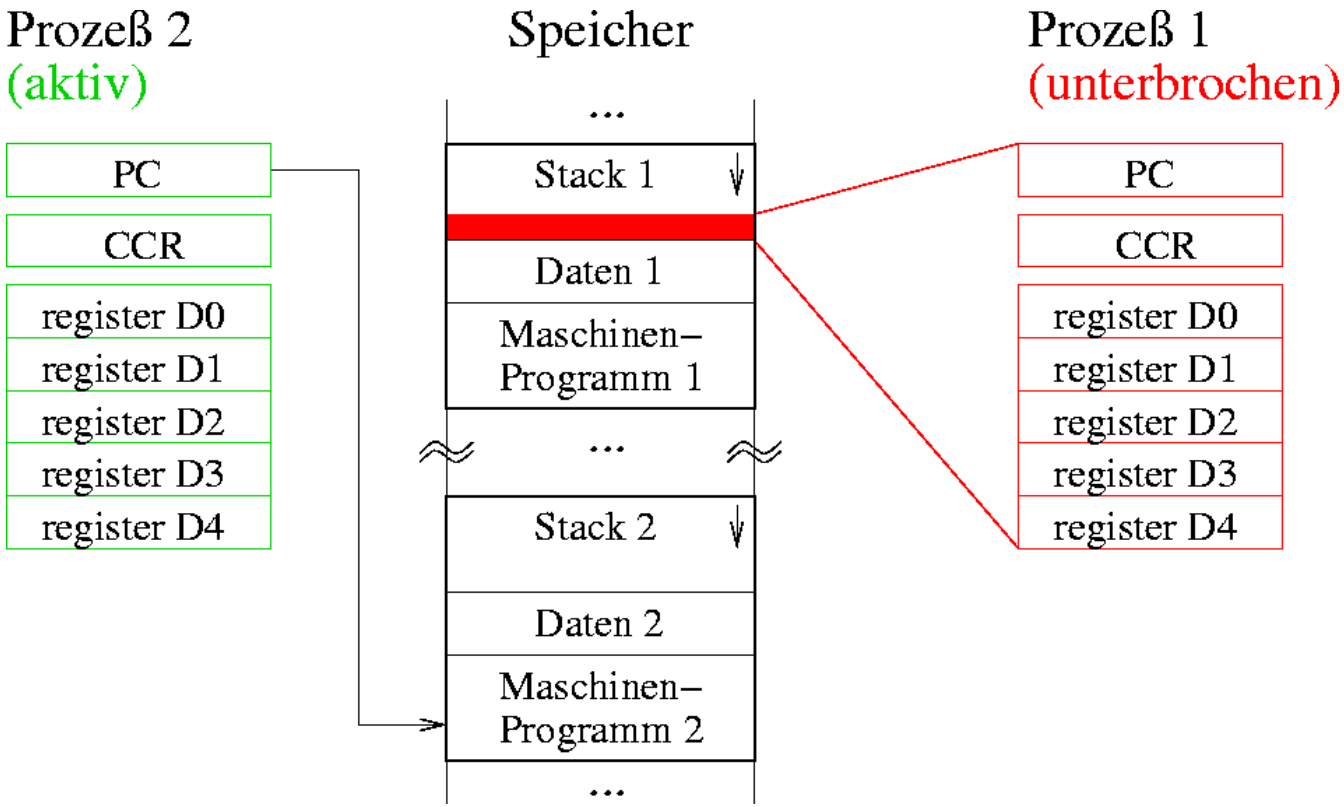
Wartende Prozesse (z.B. auf das Ende einer Ein/Ausgabe)

Wechsel in die Bereit-Warteschlange, wenn das Ereignis eingetreten ist.



Prozesswechsel

- wird ein Prozess unterbrochen, d.h. nimmt er den Zustand blockiert oder bereit ein, so müssen die Registerinhalte gerettet werden, bevor ein anderer Prozess aktiv werden kann
- Beispiel:



Prozessverwaltung

- Komponente eines Betriebssystems, die für die Zuteilung von Betriebsmitteln an wartende Prozesse zuständig ist
- alle für die Prozessverwaltung („*process management*“) wichtigen Informationen eines Prozesses sind im Prozessleitblock enthalten, z.B.:
  - Zustand
  - erwartetes Ereignis
  - Scheduling-Parameter
  - Bebtriebsmittelverbrauch
  - Speicherreferenzen
- die Prozesstabelle enthält die Prozess-leitblöcke sämtlicher Prozesse

Beispiel:
UNIX Prozeßtabelle (Auszug)

process state
process flags
process priority
user id
user time
system time
controlling tty
process id
parent process id
pointer to code
pointer to data
pointer to stack
...

Prozessscheduling

- Aufgabe eines Schedulers besteht darin, nach einer bestimmten Strategie zu entscheiden, welchen Prozess die CPU als nächstes wie lange ausführen darf

- Ziele aller Scheduling-Strategien:
  - hohe Auslastung der CPU
  - hoher Durchsatz (Zahl der Prozesse je Zeiteinheit)
  - möglichst kleine Ausführungszeit („*elapsed time*“) für jeden Prozess, d.h. im wesentlichen möglichst kleine Gesamtwartezeit
  - kurze Antwortzeit
  - faire Behandlung aller Prozesse

- da Ziele sich z.T. gegenseitig widersprechen, gibt es keine für jede Situation optimale Scheduling-Strategie

Scheduling-Strategien

- wir unterscheiden zwei Arten von Scheduling-Strategien:
  - nonpreemptive (nicht verdrängende, *kooperative*) Strategien:
    ein aktiver Prozess läuft, bis er terminiert oder in den Zustand *blockiert* übergeht (z.B. weil er auf ein E/A-Gerät wartet)
  - preemptive (verdrängende) Strategien:
    einem aktiven Prozess kann die CPU vom Scheduler entzogen werden

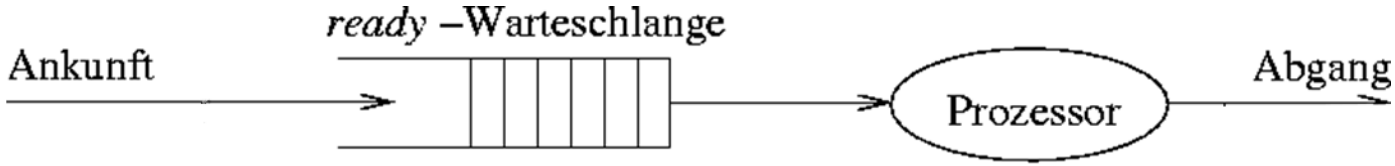
- in modernen Betriebssytemen nur preemptive Strategien, non-preemptive Strategien nur für spezielle Systeme und Peripherie

- Grundlage für die Implementierung der meisten preemptiven Scheduling-Strategien ist die Existenz einer Hardware-Uhr:

  nach jeweils t ms (z.B. 50 ms) erhält der Scheduler durch eine Unterbrechung die Kontrolle über die CPU – Zeitscheibe (*time slice*)

Nonpreemptive Scheduling-Strategien

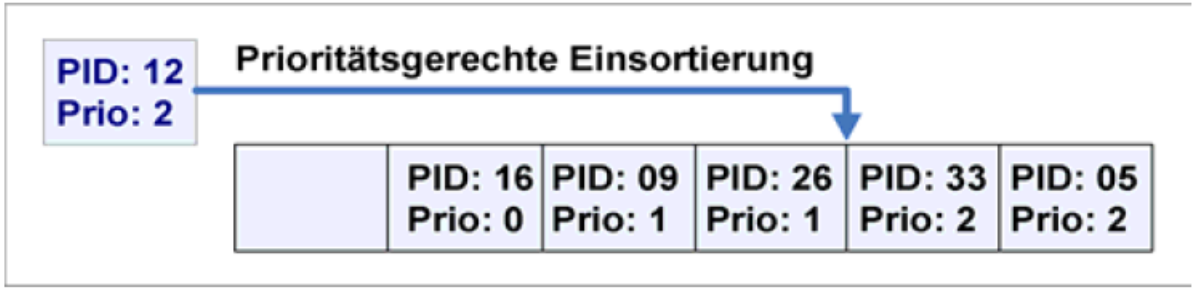
- „*First Come First Serve*“ (FCFS):
  - Die Prozesse werden in Reihenfolge ihres Eintreffens in die Wareschlange einsortiert. Alle Prozesse kommen an die Reihe:



  - Abarbeitung der Warteschlange nach dem FIFO-Prinzip
- „*Shortest Job First*“ (SJF):
  - Der Prozess mit der (geschätzten) kürzesten Bedienzeit wird allen anderen vorgezogen
  - Bei gleichen Bedingungen minimiert die Wartezeit eines Jobs
  - Bedienzeit vs. Antwortzeit
  - Ziel: Durchschnittliche Antwortzeit minimieren
  - Beim hohen Prozess-Aufkommen können Prozesse „**Verhungern**“

Nonpreemptive Scheduling (Forts.)

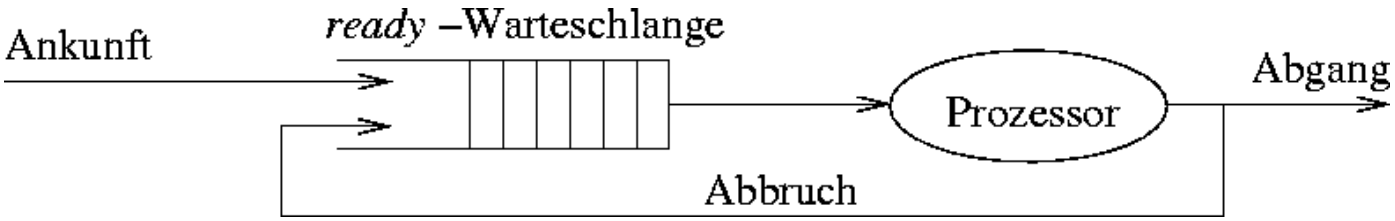
- „*Highest Response Ratio Next*“ (HRRN):
  - Bevorzugt Prozesse mit hüheren (geschätzten) (Antwortzeit / Bedienzeit) Verhältnis
  - Vornehmlich bei hohem Interaktiv-Anteil eingesetzt
- „*Priority Scheduling*“ (PS):
  - jeder Prozess i erhält eine Priorität p_i
  - er behält die CPU, bis er terminiert



  - Abarbeitung der Warteschlange nach FIFO-Prinzip
  - Gleiche Prioritäten werden z.B. nach *FCFS* einsortiert

Preemptive Scheduling-Strategien

- „*Round Robin*“ (RR):
  - Zeitscheibenverfahren, kombiniert mit *FCFS* .
  - Jeder Prozess erhält für die Zeitdauer t_{slice} die CPU und wird wieder am Ende der Warteschlange eingereiht:



  - Problem: gute Wahl von t_{slice}

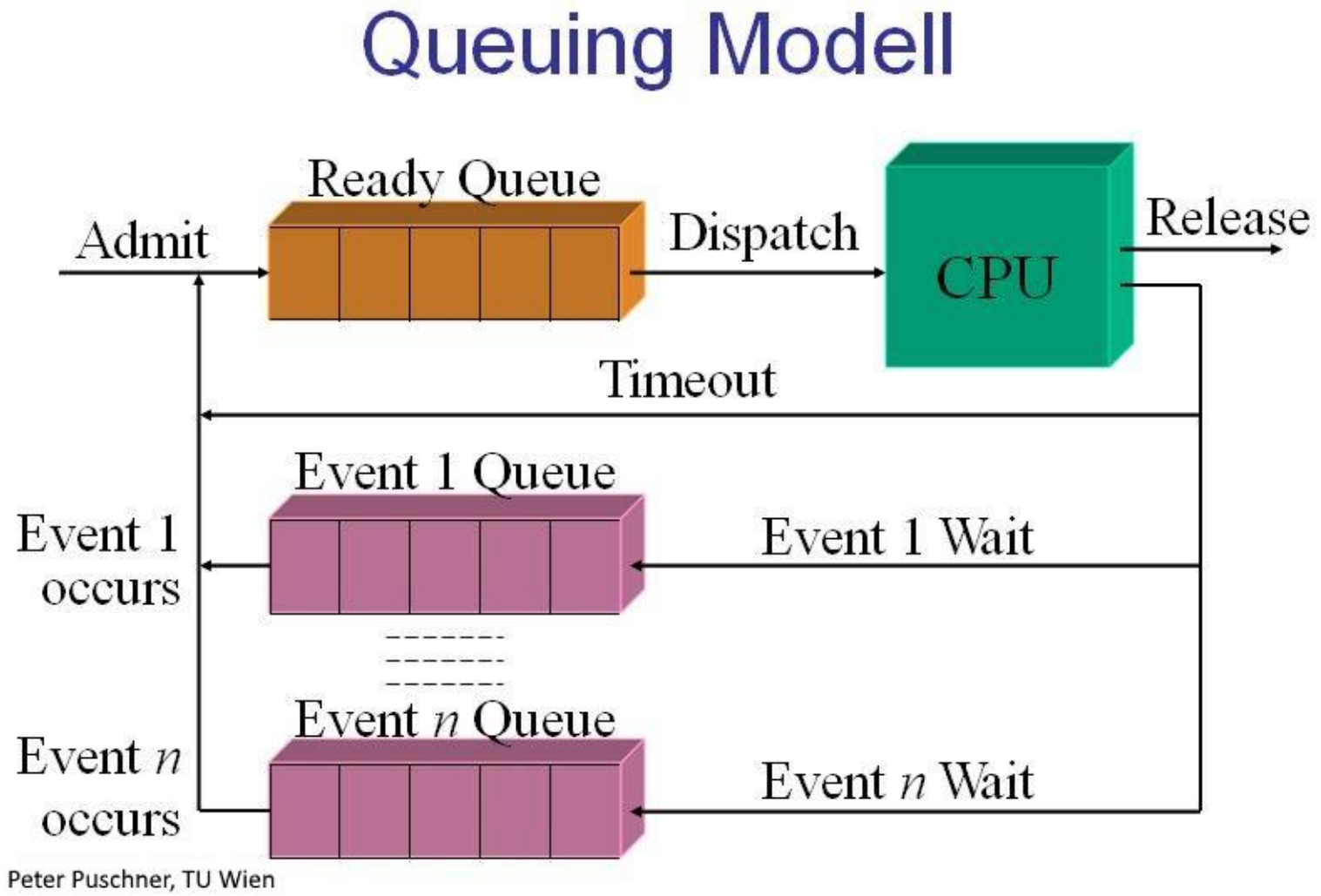
- „*Dynamic Priority Round Robin*“ (DPRR):
  - Prioritätsgeführte Warteschlangen als Vorstufe des *RR*
  - Prioritäten werden mit jeder Zeitscheibe erhöht
  - wer die Schwellenpriorität erreicht, kommt in die *RR* Warteschlange

Preemptive Scheduling-Strategien (Forts.)

- „*Shortest Remaining Time First*“ (SRTF):
  - nach jeder Unterbrechung erhält derjenige Prozess die CPU, dessen restliche Rechenzeit t_{rest} minimal ist (SJF)
  - Problem: t_{rest} ist oft schwer zu schätzen

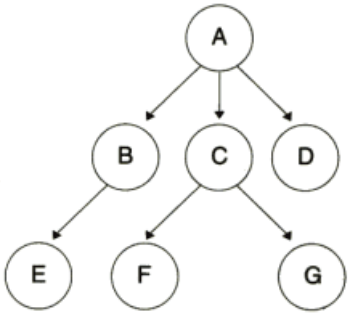
- Beispiel: UNIX Prozess-Scheduling
  - „*Round Robin*“ kombiniert mit dynamischen Prioritäten
  - jeder Prozess i erhält hat eine initiale Priorität $p_i(0)$, die sich beim Warten mit der Zeit erhöht
  - für die Prozesse jeder Priorität p gibt es eine eigene Warteschlange Q_p
  - sei *pmax* die maximal vorhandene Priorität, dann werden zunächst alle Prozesse in Q_{pmax}, dann in Q_{pmax-1} usw. abgearbeitet
  - Prozesse im „*kernel mode*“ sind ununterbrechbar
  - Benutzer kann initiale Priorität mit dem Kommando `nice` herabsetzen

Multiple Warteschlangen



Prozesshierarchie

- Ein Prozess kann einen neuen Prozess starten (fork, spawn)
  - "Kind-“ oder "SohnProzess" (child process)
  - Erzeuger: "Eltern-“ oder "VaterProzess" (parent process)
- Prozesshierarchie
  - Jeder KindProzess hat genau einen ElternProzess
  - Ein ElternProzess kann mehrere Kindprozesse besitzen
  - Eltern- und Kindprozesse können miteinander kommunizieren
  - Wird ein Elternprozess beendet, beenden sich normalerweise auch alle seine Kindprozesse
- Präzedenzgraph

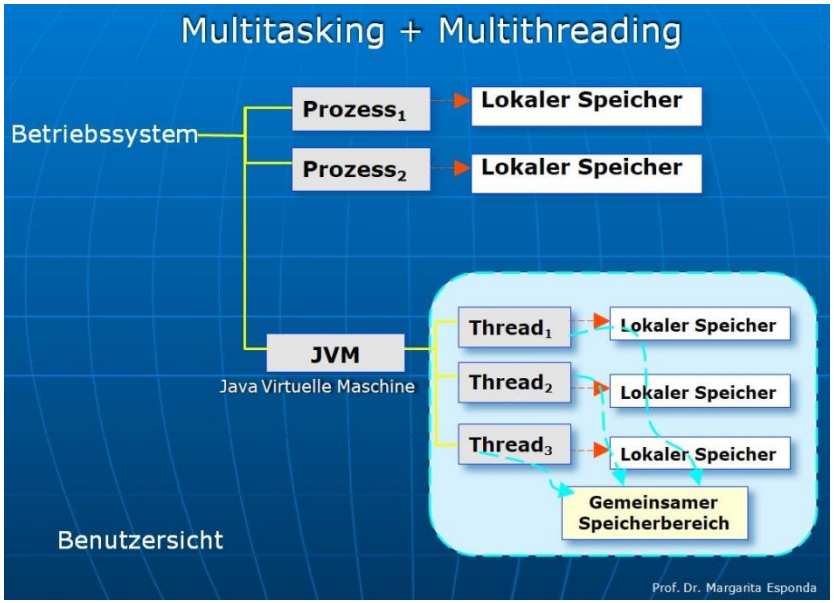


Threads

- Bisher betrachtete Prozesse bilden Einheit für
  - Ressourcenverwaltung
  - Dispatching (kurzfristiges Scheduling)
- Entkopplung von (1) und (2):
  - *Process (Task)*: Einheit der Ressourcenverwaltung
  - *Thread (Lightweight Process):* Einheit für das Dispatching
- *Multithreading*: $n > 1$ Threads pro Prozess

Threads (Forts.)

- Process
  - virtueller Adressraum mit „*process image*“
  - Speicherschutz, Files, I/O Ressourcen
- Thread
  - Ausführungszustand (*running*, *ready*, …)
  - Kontext (wenn nicht gerade laufend)
  - Stack
  - thread-lokale statische und lokale Variable
  - Zugriff auf Prozessspeicher und Ressourcen



Threads (Forts.)

- häufiger Prozesswechsel stellt in einem Betriebssystem eine hohe Belastung dar; auch erfordert die Generierung eines neuen Prozesses viele System-Resourcen
- in vielen Anwendungen werden nur quasi-parallel agierende Codefragmente benötigt, die im gleichen Prozesskontext arbeiten (d.h. insbesondere auf die gleichen Daten zugreifen !)
- Leichtgewichtsprozesse („*threads*“) stellen ein weiteres Prozess- system innerhalb eines Prozesses dar
- Betriebssystem ist hier beim Prozesswechsel i.a. nicht involviert; ein einfacher Scheduler für Leichtgewichtsprozesse wird vom Laufzeitsystem bereitgestellt
- vorgestellte Methoden der Synchronisation von Prozessen auch für *Leichtgewichtsprozesse* anwendbar

Threads (Forts.)

- Multithreading:

  Fähigkeit eines Prozesses, mehrere Bearbeitungsstränge gleichzeitig abzuarbeiten.

  Multithreading setzt neben Multitasking entsprechende Fähigkeiten des Betriebssystems voraus.

- Abgrenzung zu Multitasking:

  Multitasking: parallele Ausführung mehrerer Prozesse

  Multithreading: parallele Ausführung von Bearbeitungssträngen innerhalb eines Prozesses.

  Teilung der zugeteilten Rechenzeit und des Speichers (**gemeinsamer Adressraum!**).

- Hyperthreading:

  Spezielle Funktionen von Intel Prozessoren (z.B. Pentium 4, Xeon) zur Unterstützung von Multithreading-Anwendungen.

Kommunikation zwischen Prozessen

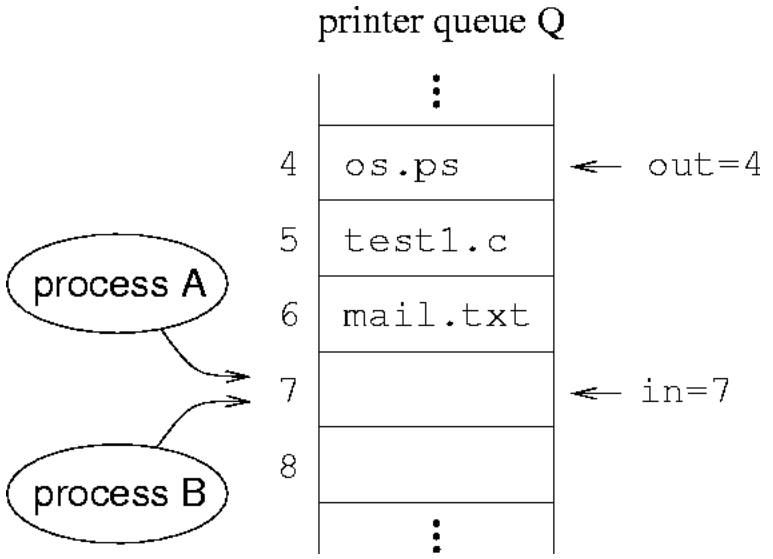
- Prozesse müssen häufig miteinander kommunizieren, z.B. um die Ausgabe einem anderen Prozess zu übergeben
- Möglichkeiten der Prozesskommunikation:
  1) Nutzung eines gemeinsamen Speicherbereiches (d. h. Nutzung gemeinsamer globaler Variablen)
  2) Kommunikation über Dateien des Dateisystems
  3) expliziter Austausch von Nachrichten (Botschaften)
  4) Kommunikation über „*Pipes*“ (UNIX)
- eine sichere Prozesskommunikation mit 1) und 2) setzt eine Prozesssynchronisation voraus, um
  1) einen gegenseitigen Ausschluß zu realisieren
  2) auf die Erfüllung einer Bedingung durch einen anderen Prozess zu warten (Bedingungssynchronisation)

Kritischer Abschnitt

- Problem: mehrere Prozesse wollen gleichzeitig eine gemeinsame Speichervariable ändern
- Beispiel:

  Zwei Prozesse A und B wollen einen Druckauftrag in die Warteschlange Q des Drucker-Spoolers schreiben:

  – Prozess A hat CPU, liest `in`, und wird vom Betriebssystem unterbrochen
  – Prozess B erhält CPU, liest `in`, und schreibt Auftrag nach `Q[in]`
  – Prozess A bekommt CPU zurück und überschreibt Auftrag `Q[in]`



- Code-Abschnitte, die nicht unterbrochen werden dürfen, werden als kritische Abschnitte bezeichnet

Gegenseitiger Ausschluß

- es muß verhindert werden, dass zwei Prozesse gleichzeitig in ihren kritischen Abschnitten sind !
- Anforderungen an ein Verfahren zur sicheren Realisierung eines gegenseitigen Ausschlusses („*mutual exclusion*“):
  1) Sicherstellung, dass sich stets nur maximal ein Prozess im kritischen Abschnitt befindet
  2) Bereitstellung einer Möglichkeit, mit der sich Prozesse beim Betreten und Verlassen des kritischen Abschnitts abstimmen können
  3) korrektes Verhalten auch bei mehr als zwei Prozessen
  4) faire Vergabe des Zutritts zu einem kritischen Abschnitt bei zwei oder mehr wartenden Prozessen
- viele Implementierungsmöglichkeiten ...

  (Lösungsbeispiele)

Bedingungen der Auflösung

- Vier Bedingungen für eine gute Lösung (nach Tanenbaum):
  1. Höchstens ein Prozess darf sich in einem kritischen Abschnitt aufhalten. (Korrektheit)
  2. Es dürfen keine Annahmen über Ausführungsgeschwindigkeit und Anzahl der Prozessoren gemacht werden.
  3. Kein Prozess, der sich in einem kritischen Abschnitt befindet, darf andere blockieren.
  4. Kein Prozess soll unendlich lange warten müssen, bis er in einen kritischen Bereich eintreten darf.

- Die letzten beiden Punkte dienen der Stabilität, sie sollen Prozessverklemmungen verhindern.

Unterbrechungssperre

- Idee: während Ausführung eines jeden kritischen Abschnittes werden sämtliche Unterbrechungen maskiert

  $\Rightarrow$ auch Betriebssystem kann den laufenden Prozess nicht unterbrechen, da es keine Kontrolle über die CPU erhält

- Vorteil:
  - \+ einfache Realisierung
- Nachteile:
  - Anwender kann vergessen, die Maskierung aufzuheben
  - falsche „*user time*“ und „*system time*“ Zeiten in Prozesstabelle, da keine Unterbrechungen durch Hardware-Uhr möglich
  - hohe Reaktionszeit bei eintreffenden E/A-Unterbrechungsanforderungen können zu Datenverlust führen
  - funktioniert nicht bei Mehrprozessorbetrieb

„Test & Set“ – Instruktion

- Lesen und Schreiben eines Speicherwertes wird zur unteilbaren Operation (z.B. durch Hardwareunterstützung):

```
test&set(busy,local) : [local = busy; busy = TRUE]
```

- Gegenseitiger Ausschluß mit `test&set` :

  (`busy` ist mit `FALSE` initialisiert)

```
do {
   test&set(busy,local);
   if (local == FALSE) {
      kritischer Abschnitt;
      busy = FALSE;
   }
}
while (local != FALSE);
```

- Nachteile: je nach Scheduling-Strategie unfaire Vergabe oder Verklemmungen möglich

Semaphore

- Ein Semaphor (Dijkstra 1965) ist ein Variable `S`, auf der die zwei folgenden ununterbrechbaren Operationen `P` (Passieren) und `V` (Verlassen) definiert sind :

  zunächst eine einfache Implementierung ...

```
P(S) : [ while (S≤0) { /* do nothing */ } ; S=S-1 ]
V(S) : [ S=S+1 ]
```

nur hier unterbrechbar !

- Realisierung eines gegenseitigen Ausschlusses mit Semaphor: (Initialisierung `S=1` erforderlich)

```
P(S);
kritischer Abschnitt;
V(S);
```

- `S>0` bezeichnet hier die Zahl der verfügbaren Betriebsmittel
- Nachteil obiger Implementierung: Prozess, der die Operation `P` ausführt, benötigt viel CPU-Zeit („*busy waiting*“)

Semaphore (Forts.)

- effizientere Implementierung eines Semaphors durch Definition von `S` als Objekt mit Komponenten `S.ctr` (Wert des Semaphors) und `S.list` (Liste wartender Prozesse):

```
P(S) : [ S.ctr=S.ctr-1 ;
         if (S.ctr < 0) {
            put pid in S.list; sleep() } ]

V(S) : [ S.ctr=S.ctr+1 ;
         if (S.ctr ≤ 0) {
            get pid from S.list; wakeup(pid) } ]
```

- Systemaufruf **`sleep()`** bewirkt, dass Prozess `pid` blockiert wird und somit keine CPU-Zeit verbraucht, bis er durch einen **`wakeup(pid)`** Systemaufruf wieder bereit wird
- `S` bezeichnet hier allgemein bei `S>0` die Zahl der verfügbaren Betriebsmittel und bei `S<0` die Zahl der wartenden Prozesse

Beispiel: Erzeuger/Verbraucher-Problem

- Erzeuger-Prozess („*producer*“) erzeugt Daten und schreibt sie in einen Puffer mit `N` Speicherplätzen (davon `used` belegt)
- Verbraucher-Prozess („*consumer*“) liest Daten aus Puffer mit unterschiedlicher Geschwindigkeit

Erzeuger-Prozess:

```
while (TRUE) {
  item = produceItem();
  if (used == N)
    sleep();
  putInBuffer(item);
  used = used+1;
  if (used == 1)
    wakeup (consumer);
}
```

Verbraucher-Prozess:

```
while (TRUE) {
  if (used == 0)
    sleep();
  item = getFromBuffer();
  used = used-1;
  if (used == N-1)
    wakeup (producer);
  consume(item);
}
```

- inkorrekt bei gegenseitigen Unterbrechungen !

Beispiel: Erzeuger/Verbraucher-Problem (Forts.)

- erste Idee: Realisierung eines gegenseitigen Ausschlusses mit Semaphor **`mutex`** (initialisiert mit `mutex=1`)

Erzeuger-Prozess:

```
while (TRUE) {
  item = produceItem();
  P(mutex);
  if (used == N)
    sleep();
  putInBuffer(item);
  used = used+1;
  if (used == 1)
    wakeup (consumer);
  V(mutex);
}
```

Verbraucher-Prozess:

```
while (TRUE) {
  P(mutex);
  if (used == 0)
    sleep();
  item = getFromBuffer();
  used = used-1;
  if (used == N-1)
    wakeup (producer);
  V(mutex);
  consume(item);
}
```

- unsicher, da für **`used=0`** und **`used=N`** Verklemmungen („*Deadlocks*“) auftreten können

Beispiel: Erzeuger/Verbraucher-Problem (Forts.)

- zweite Idee: Verwendung weiterer Semaphore für
  1) die Anzahl **`used`** belegter Speicherplätze (Initialisierung **`used=0`**)
  2) die Anzahl **`free`** freier Speicherplätze (Initialisierung **`free=N`**)

  ersetzt Inkrement/Dekrement von `used` sowie Aufrufe von `sleep()` und `wakeup()`

Erzeuger-Prozess:

```
while (TRUE) {
   item = produceItem();
   P(free);
   P(mutex);
   putInBuffer(item);
   V(mutex);
   V(used);
}
```

Verbraucher-Prozess:

```
while (TRUE) {
   P(used);
   P(mutex);
   item = getFromBuffer();
   V(mutex);
   V(free);
   consume(item);
}
```

- korrekte Implementierung, auch bei mehr als zwei Prozessen !

Bewertung des Semaphor-Konzeptes

+ mächtiges Konzept, das flexibel verwendet werden kann
+ fast alle Synchronisationsprobleme sind mit Semaphoren lösbar
+ einfache Realisierung eines gegenseitigen Ausschlusses
– die Suche nach einer korrekten Lösung ist bei komplexen Problemstellungen, insbesondere zur Implementierung einer Bedingungssynchronisation schwierig
– Verklemmungen sind möglich
– unübersichtlich, da `P` und `V` Operationen im Code verstreut
– führt leicht zu Fehlern bei der Implementierung, z.B. durch
  1) Vertauschen der Reihenfolge von `P(S)` und `V(S)`
  2) Vertauschen der Reihenfolge bei mehreren Semaphoren

Monitore

Ein Monitor (B. Hansen, 1975) stellt einen abstrakten Datentyp (Klasse) dar, mit dem Synchronisation und Kommunikation zwischen Prozessen auf höherer Ebene beschrieben werden können

Komponenten eines Monitors:

- private Variable(n)
- **`entry`** Methoden, die nur im gegenseitigen Ausschluß von verschiedenen Prozessen aufrufbar sind und den Zugriff auf private Variablen ermöglichen
- Bedingungsvariable **`condition c`** (z.B. empty oder full)
- **`wait(c)`** : ausführender Prozess gibt Monitor frei und wartet in einer Warteschlange zu `c`, bis Signal gesendet wird
- **`signal(c)`** : ausführender Prozess weckt einen auf c wartenden und verläßt Monitor

Programmiersprache muß entsprechende Sprachkonstrukte anbieten

Monitore (Forts.)

```
monitor buffer {
  int in, out, used, buf[N];
  condition notFull, notEmpty;

  entry void put (int d) {
    if (used == N)
      wait(notFull);
    buffer[in] = d;
    ++used; in = (in+1)%N;
    signal(notEmpty);
  }
  entry int get (void) {
    if (used == 0)
      wait(notEmpty);
    int d = buf[out];
    --used; out = (out+1)%N;
    signal(notFull);
    return d;
  }
}
```

Erzeuger/Verbraucher-Problem in Pseudo-C mit Monitor:

```
void producer (void) {
  int d;
  while (TRUE) {
    d = produceItem();
    buffer.put(d);
  }
}

void consumer (void) {
  int d;
  while (TRUE) {
    d = buffer.get();
    consumeItem(d);
  }
}
```

Monitore (Forts.)

Erzeuger/Verbraucher-Problem mit Monitor in Java (Auszug):

```
class buffer {
  int in=0, out=0, used=0, buf[]=new int[N];

  synchronized public void put (int d) {
    if (used == N)
      try{wait();} catch(InterruptedException e) {}
    buffer[in] = d;
    ++used; in = (in+1)%N;
    notify();
  }
  synchronized public int get (void) {
    if (used == 0)
      try{wait();} catch(InterruptedException e) {}
    int d = buf[out];
    --used; out = (out+1)%N;
    notify();
    return d;
  }
}
```

Kommunikation mit Nachrichten

- zwei Kommunikationsprimitive

  ```
  send(destination, message)
  message = receive(source)
  ```

  ermöglichen Senden und Empfangen einer Nachricht an bzw. von einer gegebenen Adresse

- EmpfängerProzess blockiert, wenn bei Aufruf von `receive` keine Nachricht vorliegt

- Sender und Empfänger können (vereinfacht betrachtet)
  - Prozesse auf dem gleichen Rechner sein (`destination` und `source` entsprechen den jeweiligen `pid´s`)
  - Prozesse auf verschiedenen Rechnern sein (`destination` und `source` sind als `pid@machine.domain` darstellbar)

Kommunikation mit Nachrichten (Forts.)

- Nachrichtenkanal wird als unsicher betrachtet:
  1) Empfänger bestätigt Empfang einer Nachricht
  2) Sender wiederholt Nachricht, wenn nach einer vorgegebenen Zeitspanne keine Bestätigung eintrifft
  3) Empfänger muß zwischen neuer und wiederholter unterscheiden können, da auch Bestätigung verlorengehen kann

- jede Nachricht kann als Objekt wie folgt implementiert werden:

```
class message {
  string   destination;     // Zieladresse
  string   source;          // Sendeadresse (z.B.für Bestätigung)
  int      no;              // fortlaufende Nummer
  int      type;            // Kennzeichnung für Typ
  unsigned size;            // Größe der Nachricht (z.B. Anzahl Bytes)
  void*    data;            // Zeiger auf Inhalt
}
```

Kommunikation mit Nachrichten (Forts.)

- beim synchronen Senden einer Nachricht blockiert der Sende-Prozess, bis eine Bestätigung eintrifft
- asynchrones Senden ist nichtblockierend, d.h. SendeProzess arbeitet weiter, Betriebssystem kümmert sich um korrekte Übertragung und puffert automatisch gesendete, aber noch nicht empfangene Nachrichten
- Implementierung des Erzeuger/Verbraucher-Problems durch Austausch von Nachrichten:

Erzeuger-Prozess:

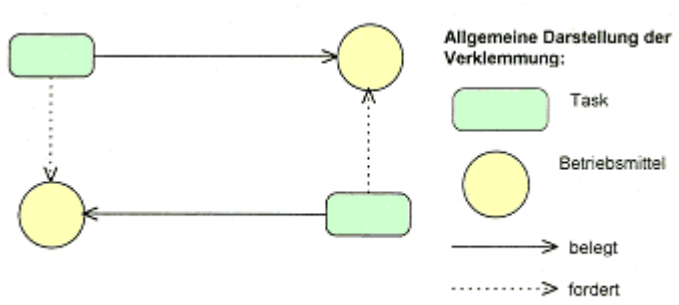
```
while (TRUE) {
  item = produceItem();
  m = buildMessage(item);
  send(consumer,m);
}
```

Verbraucher-Prozess:

```
while (TRUE) {
  m = receive(producer);
  item = extractItem(m);
  consume(item);
}
```

Verklemmungen (Deadlocks)

- Nahezu jede Situation, in der Prozesse Ressourcen **exklusiv** anfordern, kann zur Verklemmung führen.
- Für das Auftreten müssen folgende 4 Bedingungen erfüllt sein:
  1. Exklusive Nutzung (Mutual Exclusion) - BM kann immer nur ein Prozess nutzen
  2. Wartebedingung (Hold & Wait) - Prozess belegt verfügbare BM, während er auf zusätzliche BM wartet
  3. Nichtentziehbarkeit (Non-Preemption) - Prozess muss BM explizit freigeben
  4. Geschlossene Kette (Circular Wait) - Geschlossene Kette von 2 bis n Prozessen, jeder wartet auf ein BM, gehalten vom nächsten Prozess



Strategien zur Deadlock-Behandlung

- Ignorieren des Problems
  - seltenes Auftreten, interaktives System
- Entdecken und Beheben von Deadlocks
  - Zyklisch prüfen, ob ein Prozess längere Zeit blockiert
  - dann abbrechen oder zurücksetzen
  - Daten-Inkonsistenz ?!
- Verhindern: Negation (Umkehr) einer der 4 Bedingungen
  - z. B. Nur ein Drucker-Dämon darf den Drucker halten
- Vermeiden: sorgfältige BM-Vergabe
  - Grafische Analyse
  - Bekannte Algorithmen, wie. z. B. *Banker-Algorithmus*

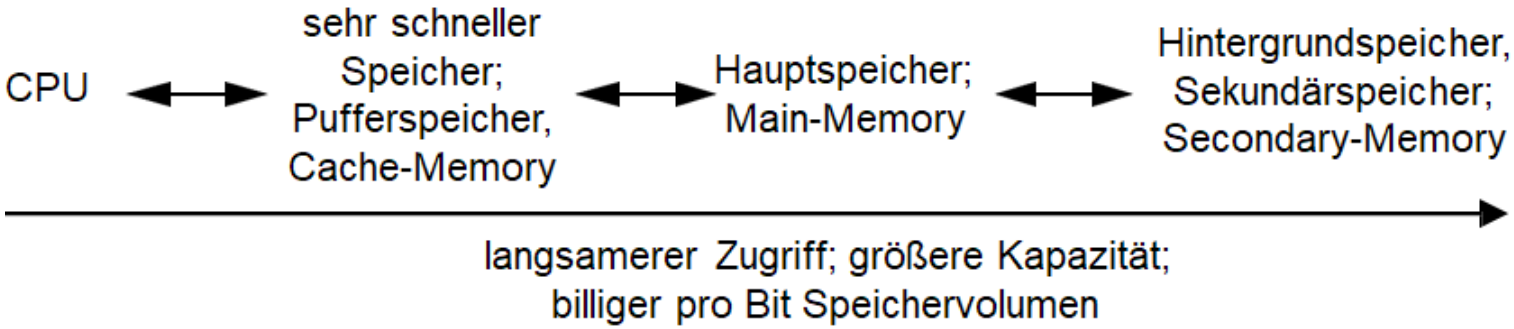
Betriebssysteme: Speicherverwaltung

Inhalt:

- Einführung
- Prozesse
- Speicherverwaltung
  - Speicherverwaltung, Partitionierung, Swapping
  - Virtueller Speicher, Adressumsetzung

Aufgaben Speicherverwaltung

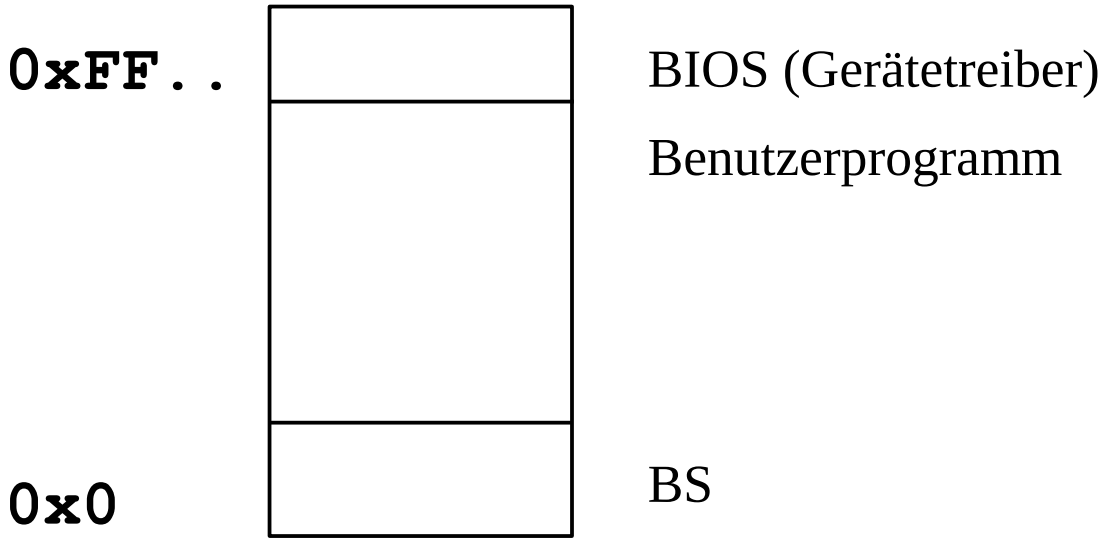
- Problemlage
  - Speicherhierarchie
  - Programmgröße
  - Parallelität von Prozessen



- Aufgaben
  - Bereitstellung von Adressräumen
  - Aufbau von Adressräumen durch Zuordnung logischer Objekte
  - Verwaltung des Betriebsmittels Hauptspeicher
  - Schutz vor unerlaubten Zugriffen
  - Organisation gemeinsamer Nutzung („Sharing“) physischen Speichers und logischer Objekte

Statische Speicher

- Statische Speicherverwaltung
  - ➔ d. h. keine Ein-/Auslagerung von Programmen/Daten
    - Einfachrechner (MS-DOS Version ??)
    - Gerätesteuerungen (embedded systems)
- Monoprogramming (ein Programm gleichzeitig)
- Programme werden nacheinander geladen und ausgeführt



Multiprogramming

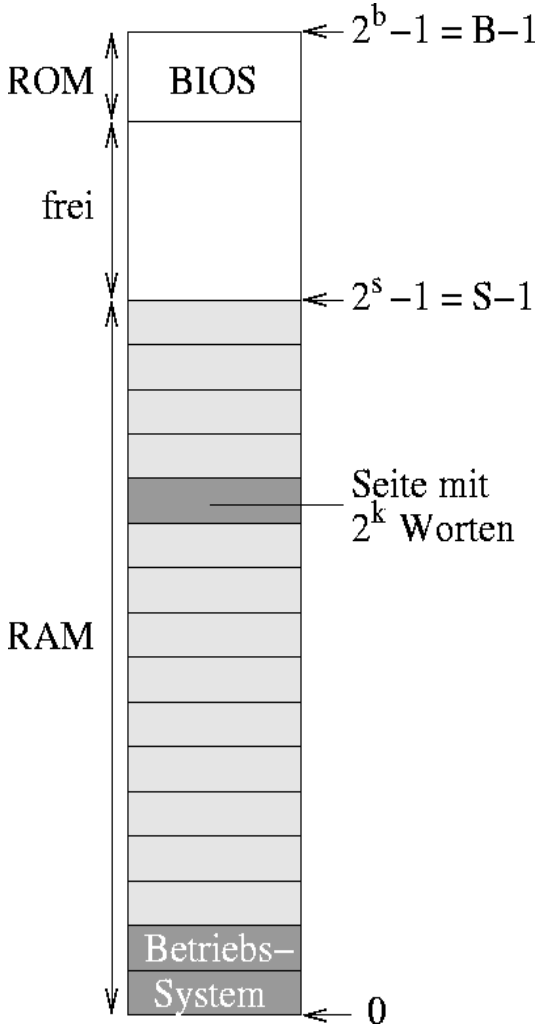
- Mehrere Benutzer-Programme gleichzeitig im Rechner; für jedes Benutzerprogramm gibt es einen oder mehrere Prozesse

- Motivation (vgl. Prozesse):
  - mehrere Benutzer eines Rechners (multiuser)
  - mehrere Benutzerprozesse eines Benutzers
  - Benutzerprozesse und Systemprozesse

- Multiprogramming vs. parallele Threads/Prozesse:
  - parallele Prozesse Voraussetzung für Multiprogramming (mit oder ohne erzwungenen Prozesswechel)
  - denkbar ist Monoprogramming mit vielen parallelen Prozessen
    - z. B.: die ersten BS für Parallelrechner erlaubten nur ein Benutzerprogramm zur gleichen Zeit, das aber aus vielen Prozessen/Threads bestehen konnte

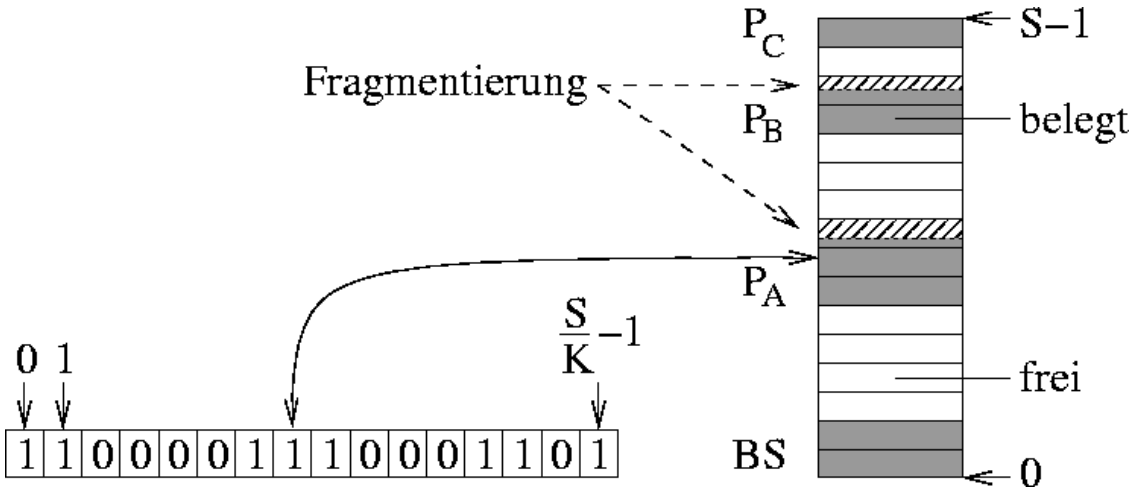
Speicherverwaltung

- Komponente eines Betriebssystems, die den Speicher den Prozessen zuteilt („*memory management*“)
- Adressraum eines Systems aus 2^b Worten wird genutzt für
  - RAM-Bereich (Arbeitsspeicher)
  - ROM-Bereich (enthält Urlader bzw. BIOS = „*Basic Input Output System*“)
- Arbeitsspeicher eines Systems aus S Worten wird eingeteilt in S/K Seiten aus jeweils $K=2^k$ Worten (typisch 4 Kbyte)
- Betriebssystemkern wird in die ersten Seiten geladen, Anwendungsprogramme in die übrigen Seiten



Speicherpartitionierung

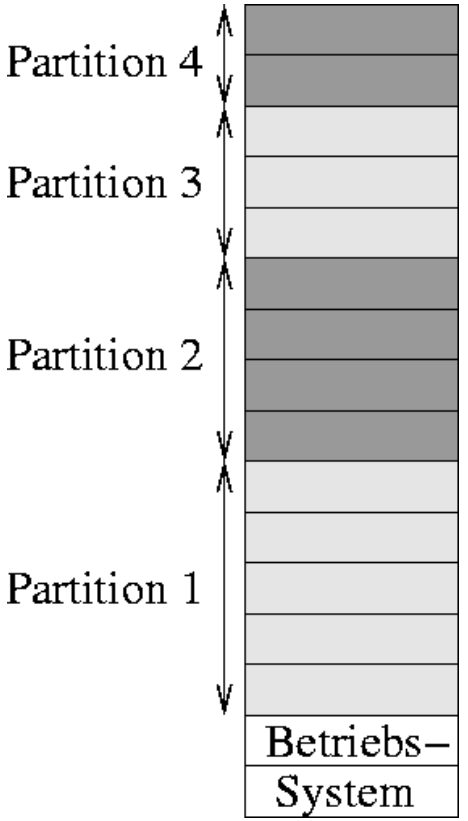
- Anwendungsprogramm benötigt einen zusammenhängenden Adressbereich (d.h. eine Partition aus benachbarten Seiten)
- nur ganze Seiten werden vergeben ($\Rightarrow$ interne Fragmentierung des Speichers, mittlerer Verlust: 1/2 Seite je Programm)
- Belegungstabelle zeigt freie Seiten:



- zunächst Betrachtung einer einfachen Speicherverwaltung, bei der Programme vollständig in Arbeitsspeicher geladen werden
- Speicheraufteilung in feste oder variable Partitionen möglich ...

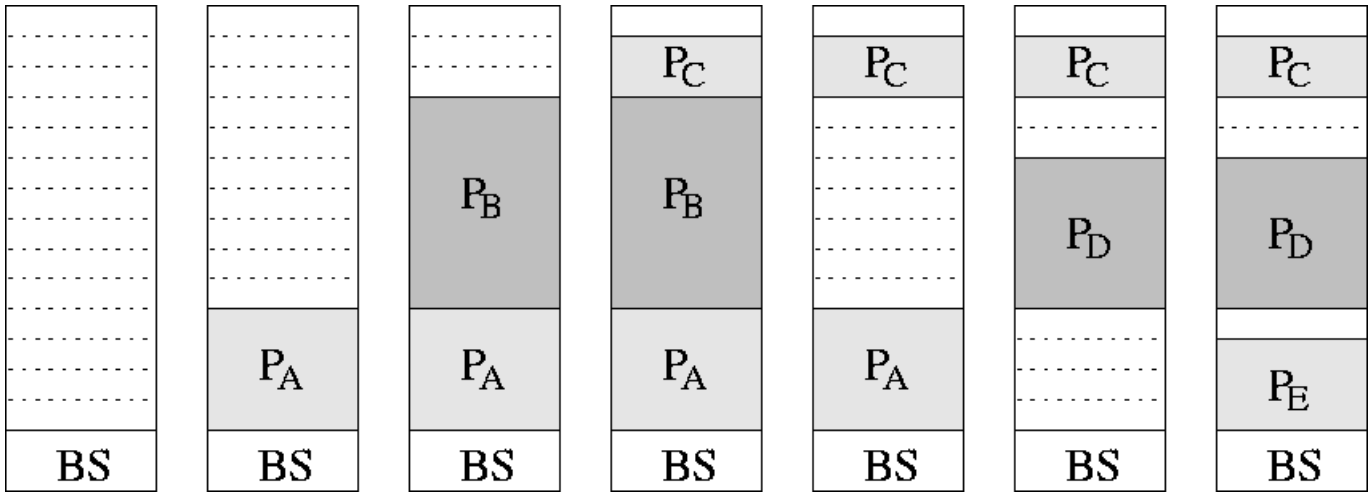
Speicher mit festen Partitionen

- Speicher ist aufgeteilt in mehrere feste Partitionen unterschiedlicher Größe
- für jeden neuen Prozess wird die kleinste ausreichend große Partition ermittelt
- Speicherverwaltung entweder durch
  - separate Warteschlange je Partition (Nachteil: große Partitionen bleiben ungenutzt, wenn alle kleinen Partitionen vergeben)
  - zentrale Warteschlange (Nachteil: große Partitionen durch kleine Prozesse belegt)
- sinnvoll vor allem für Batchverarbeitung, realisiert z.B. in IBM OS/360



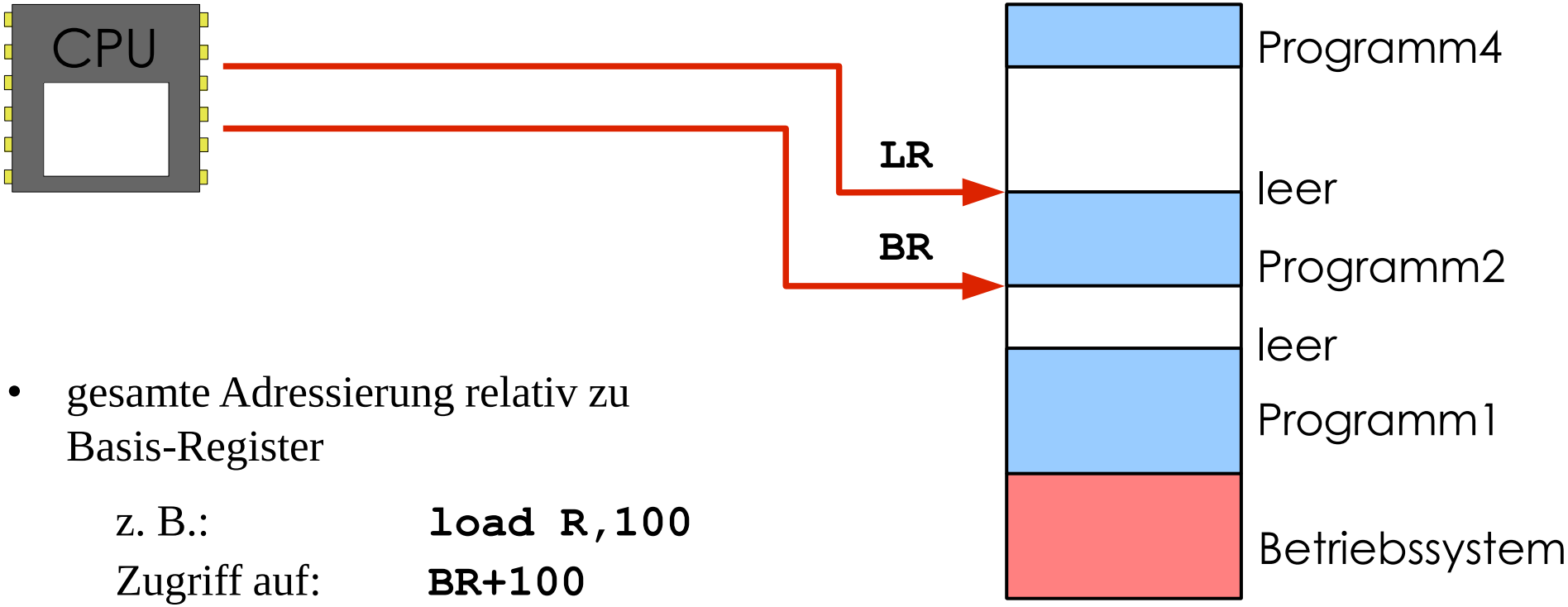
Speicher mit variablen Partitionen

- dynamisch variierende Anzahl und Größe der Partitionen
- Beispiel:



- beim Ersetzen von Partitionen können viele kleine freie Bereiche entstehen (externe Fragmentierung) $\Rightarrow$ neue Prozesse finden keinen ausreichend großen zusammenhängenden Bereich mehr
- weiteres Problem: Bestimmung einer Partitionsgröße bei einer dynamischen Speicherallokation

Basis- und Limit-Register



- gesamte Adressierung relativ zu Basis-Register

  z. B.: **`load R,100`**

  Zugriff auf: **`BR+100`**

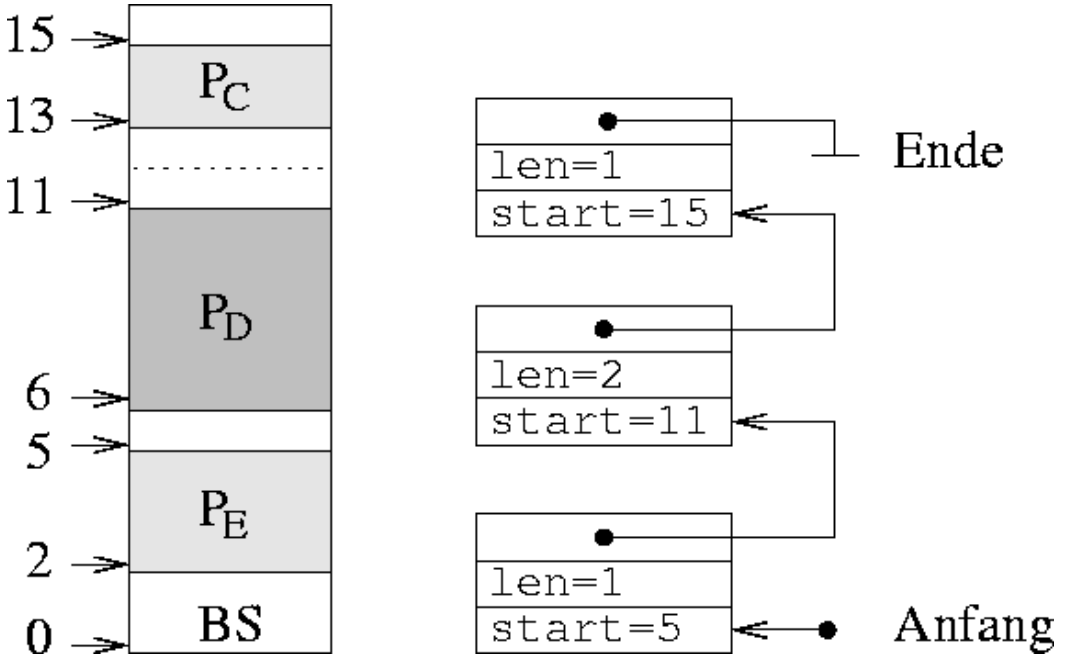
- Unterbindung aller Zugriffe auf Bereiche außerhalb [**`BR,LR`**]
- Konsequenz für Implementierung eines Prozess-Systems:
  bei Umschaltung müssen auch BR und LR umgeschaltet werden

Verwaltung freier Partitionen

- Einlagerungsalgorithmen zur „Minimierung“ des Verschnitts
- Suche nach einer ausreichend großen freien Partition in einem großen Speicher mittels Belegungstabelle ist sehr aufwendig
- Bitmaps, Listen

  Alternative:
  verkettete Freiliste

  je Eintrag in Liste Startadresse, Länge und Zeiger auf nächsten Freiblock



- Operationen auf Freiliste: Entfernen eines Freiblocks, Einfügen eines Freiblocks und Zusammenfassen benachbarter Freiblöcke

Einlagerungsalgorithmen

Algorithmen zur Suche nach einem freien Bereich mit minimaler Verschnitt:

- „*first fit*“ : Durchsuchen der Freiliste von Anfang an, bis ausreichend große Lücke gefunden
- „*best fit*“ : Durchsuchen der Freiliste von Anfang bis Ende und Wahl der kleinsten ausreichenden Lücke
- „*buddy system*“ : für Speichergröße $S=2^s$ und Seiten- größe 2^k gibt es $s–k$ Freilisten für Blöcke der Größe 2^s, 2^{s-1} ... 2^k
  1) für eine Anforderung der Größe 2^i wird zunächst in der 2^i-Freiliste, dann in 2^{i+1}-Freiliste, usw. geschaut, bis freier 2^q-Block gefunden
  2) bei $q>i$ wird der 2^q-Block in zwei 2^{q-1}-Blöcke („*buddies*“) geteilt, einer hiervon in zwei 2^{q-2} Blöcke usw., bis 2^i-Block entstanden ist
  3) alle ggf. bei der Teilung entstanden Freiblöcke der Größe 2^{q-1}, ...,2^i werden in die entsprechenden Freilisten eingetragen
  4) Gewichtete Buddy-Systeme: Ein Block der Größe 2^{r+2} wird z.B. im Verhältnis 1:3 in Blöcke der Größe 2^r und $3 \cdot 2^r$ zerlegt

  $\Rightarrow$ schnelles Verfahren, führt jedoch zu größerer Fragmentierung

Swapping

- bei vielen im System vorhandenen Prozessen ist Speichergröße S nicht ausreichend, um alle Prozesse im Arbeitsspeicher zu halten

  $\Rightarrow$ temporäre Ein-/Auslagerung von ganzen Prozessen auf Festplatte

- langsam (typische Transferrate: $\leq$ 10 Mbyte/s)
- häufig eingesetzter Swapping-Algorithmus:
  - zuerst Auslagerung von Prozessen, die bereits längere Zeit blockiert sind
  - danach ggf. Auslagerung von Prozessen, die sich bereits am längsten im Arbeitsspeicher befinden (sofern ihnen bereits CPU erteilt wurde)
  - Betriebssystemkern wird nicht ausgelagert !
- Swapping in UNIX:
  - `swapper` (pid 0) lagert Prozesse aus, wenn freier Speicherplatz < `min` und holt Prozesse wieder herein, sobald freier Speicherplatz $\geq$ `max`
  - bei Erzeugung eines neuen Prozesses wird automatisch Platz im Swap-Bereich auf Festplatte reserviert

Nachteile einfacher Speicherverwaltung

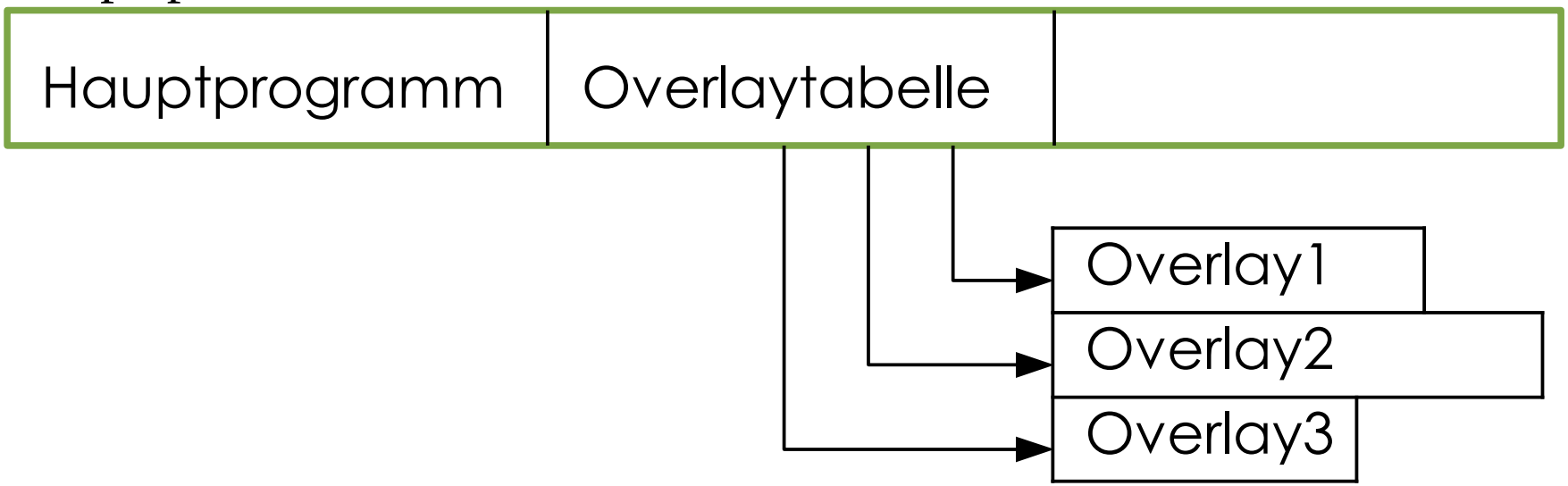
- Programme, die mehr Speicherplatz benötigen als vorhanden ist, können nicht (bzw. nur mit Überlagerung) ausgeführt werden
- hoher ungenutzter Speicheranteil (Fragmentierung):
  - 50%-Regel: bei n Prozessen im System gibt es im Mittel n/2 Lücken
  - wenn k das Verhältnis aus mittlerer Größe einer Lücke zur mittleren Prozessgröße ist, so gilt: f = k/(k+2)
- Relokation eines Programms aufwendig: beim Laden an eine Adresse a müssen alle in den Befehlen enthaltenen Daten- und (absoluten) Sprungadressen angepasst werden
- kein Speicherschutz: ein Prozess kann durch inkorrekte Adressierung Daten anderer Prozesse manipulieren
- Swapping ist ineffizient, da Prozesse immer vollständig aus- und eingelagert werden müssen

Overlays – Überlagerungstechnik

Beliebig große Programme → „Overlays“

- Programmierer organisiert seine Programme und Daten in Stücken, von denen nicht zwei gleichzeitig im Hauptspeicher sein müssen

Hauptspeicher



Adressraum

- Begriff allgemein
  - Menge direkt zugreifbarer Adressen und deren Inhalte
  - Größe bestimmt durch Rechner-Architektur
- Physischer Adressraum
  - durch Adressleitungen gebildeter AR,
    z. B. am Speicher- oder Peripheriebus eines Rechners
  - Abbildung der Prozessor-Adressen auf die vorhandenen Speicherbausteine und E/A-Controller
  - Adressumsetzung statisch durch HW-Adressdecoder
- Virtueller (logischer) Adressraum eines Prozesses
  - dem Prozess zugeordneter Adressraum
  - Adressumsetzung durch MMU,
    veränderliche Abbildungsvorschrift

Virtueller Speicher

- Forderungen an Adressräume und ihre Implementierung
  - Beginnt bei Adresse 0
  - Groß, soweit es die Hardware zulässt (z. B. jeder bis zu 4 GB auf Pentium)
  - frei teilbar und nutzbar
  - Fehlermeldung bei Zugriff auf nicht belegte Bereiche
  - Schutz vor Zugriffen auf andere Adressräume
  - Einschränken der Zugriffsrechte auf bestimmte Bereiche (z. B. Code nur lesen)

  - Sinnvoller Einsatz des (Haupt-)Speichers
    - damit Speicher anderweitig nutzbar
    - wegen kurzer Ladezeiten
    - ohne großen Aufwand für Programmierer

Virtueller Speicher

Grundidee:

Zuordnen von Speicherobjekten (z. B. Segmente, Dateien, Datenbanken, Bildwiederholspeicher) bzw. Ausschnitten davon zu Regionen von Adressräumen

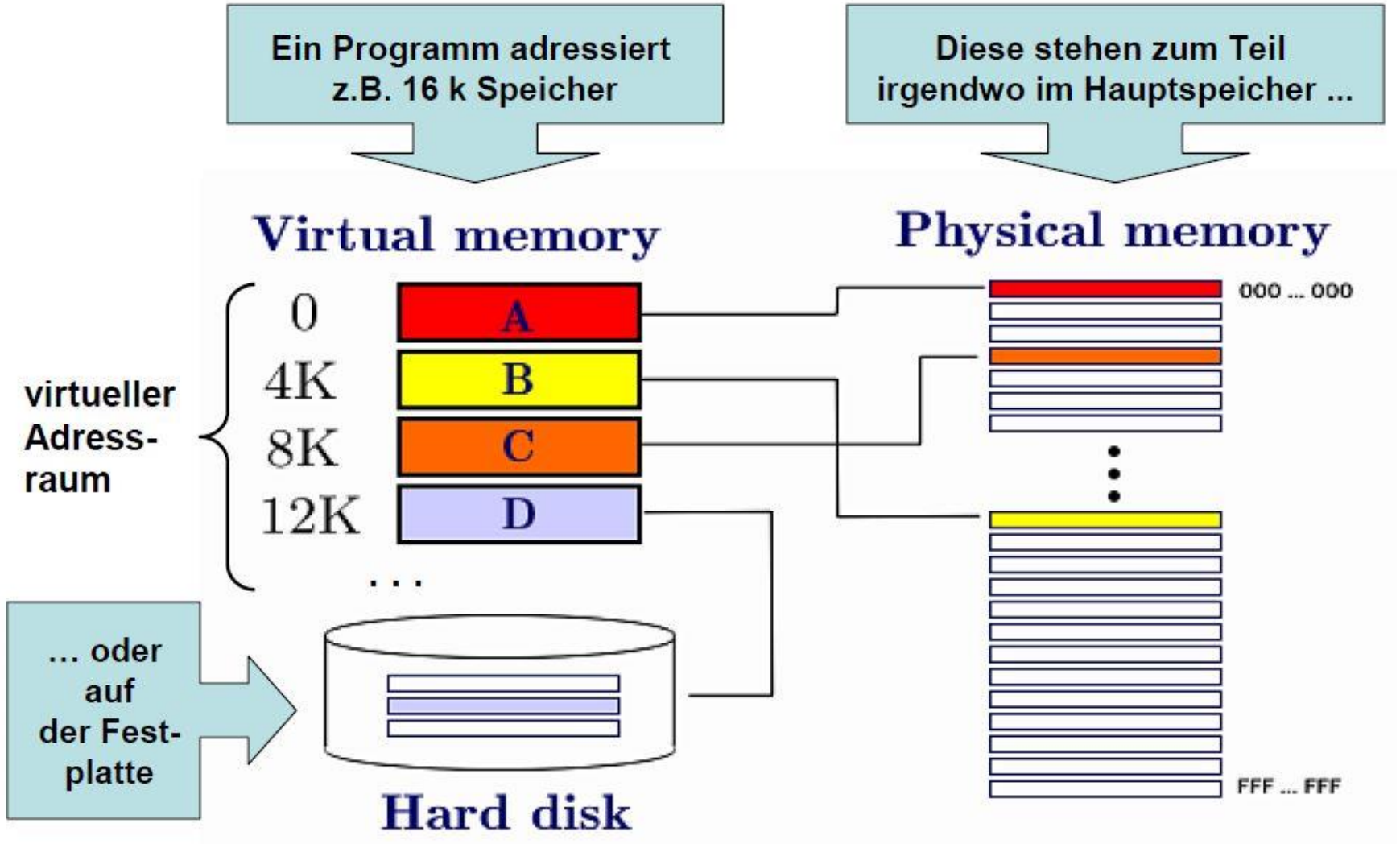
- strikte Trennung von virtuellem und physikalischem Speicher
- hardwareunterstützte, schnelle Abbildung von virtuellen Adressen auf physikalische Adressen
- virtueller Adressraum ist in Seiten fester Größe („*pages*“), physikalischer Speicher in Kacheln gleicher Größe („*page frames*“) eingeteilt
- Seiten des zusammenhängenden virtuellen Adressraums werden auf nicht zusammenhängende Kacheln des physikalischen Speichers abgebildet
- Programme haben i.a. eine hohe Lokalität: nur Seiten mit den aktuell benötigten Codefragmenten müssen im Arbeitsspeicher sein, übrige Seiten werden erst bei einem Seitenfehler („*page fault*“) eingelesen
- eine Seitentabelle („*page table*“) enthält für jede Seite die Nummer der zugeordneten Kachel; jeder Prozess besitzt eine eigene Seitentabelle

VS: Begriff, Aufgaben

- Der virtuelle Speicher ist eine Technik, die jedem Prozess einen eigenen, vom physischen Hauptspeicher unabhängigen logischen Adressraum bereitstellt, basierend auf
  - der Nutzung eines externen Speichermediums
  - einer Partitionierung von Adressräumen in Einheiten einheitlicher Größe einer Adressumsetzung durch Hardware (und Betriebssystem)
  - einer Ein- und Auslagerung von Teilen des logischen Adressraumes eines Prozesses durch Betriebssystem (und Hardware).

- Teilaufgaben im Betriebssystem
  - Seitenfehler-Behandlung
  - Verwaltung des Betriebsmittels Hauptspeicher
  - Aufbau der Adressraumstruktur (Speicherobjekte und Regionen)
  - Bereitstellung spezifischer Speicherobjekte
  - Interaktion Prozess- und Speicher-Verwaltung

Seiten, Kacheln, Blöcke

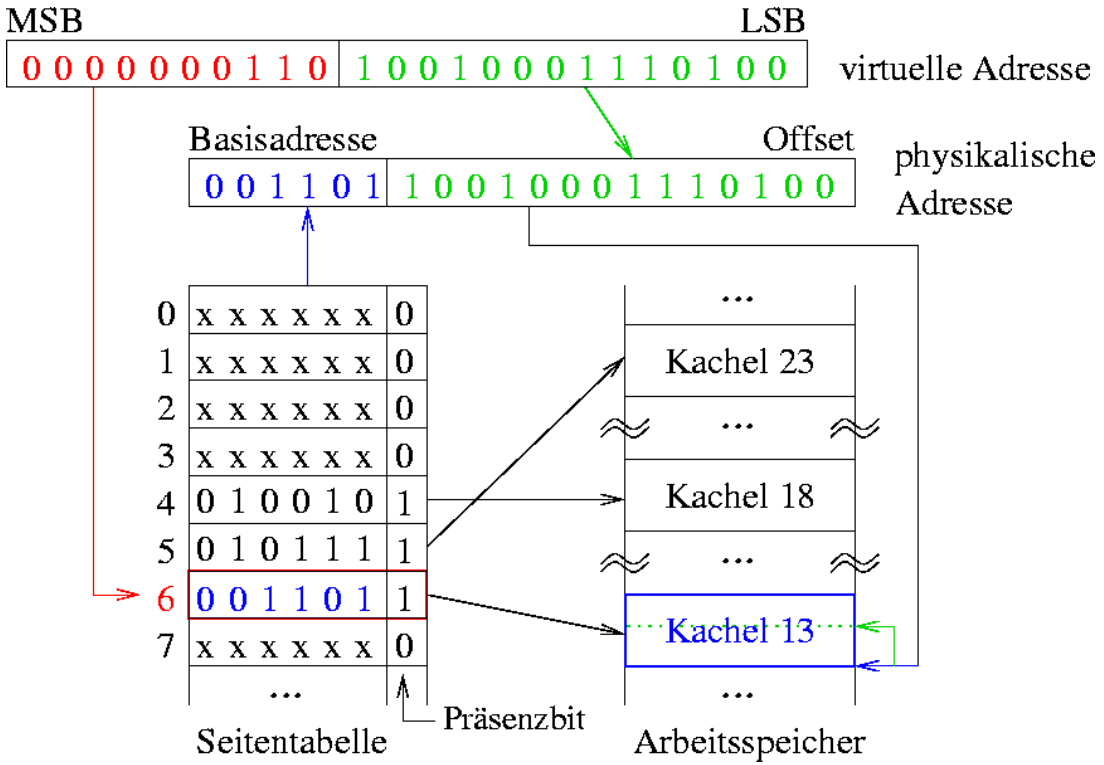


Adresstransformation

- Realisierung:

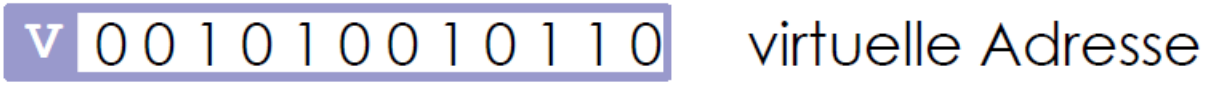
  höherwertige Bits der virtuellen Adr. bilden Seiten-Nr.

  niedrigwertige Bits der virtuellen Adr. stellen Wortadresse in Kachel dar

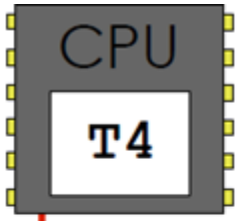


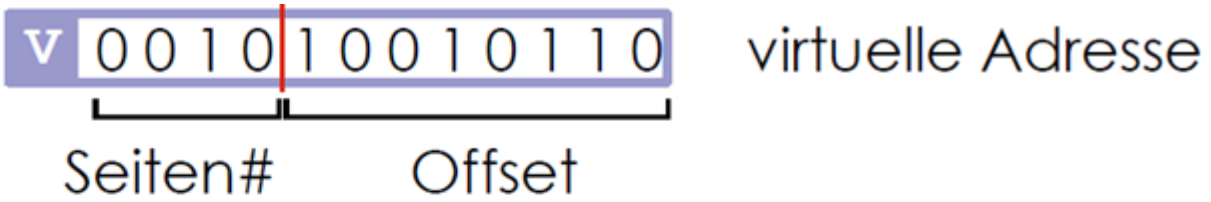
- Präsenzbit gibt an, ob Seite präsent (1) oder ausgelagert (0) ist
- neben Kachel-Nr. und Präsenzbit kann Seitentabelle z.B. noch Referenzbit, Modifikationsbit und Zugriffbits enthalten

Prinzipielle Arbeitsweise der MMU



Prinzipielle Arbeitsweise der MMU (2)



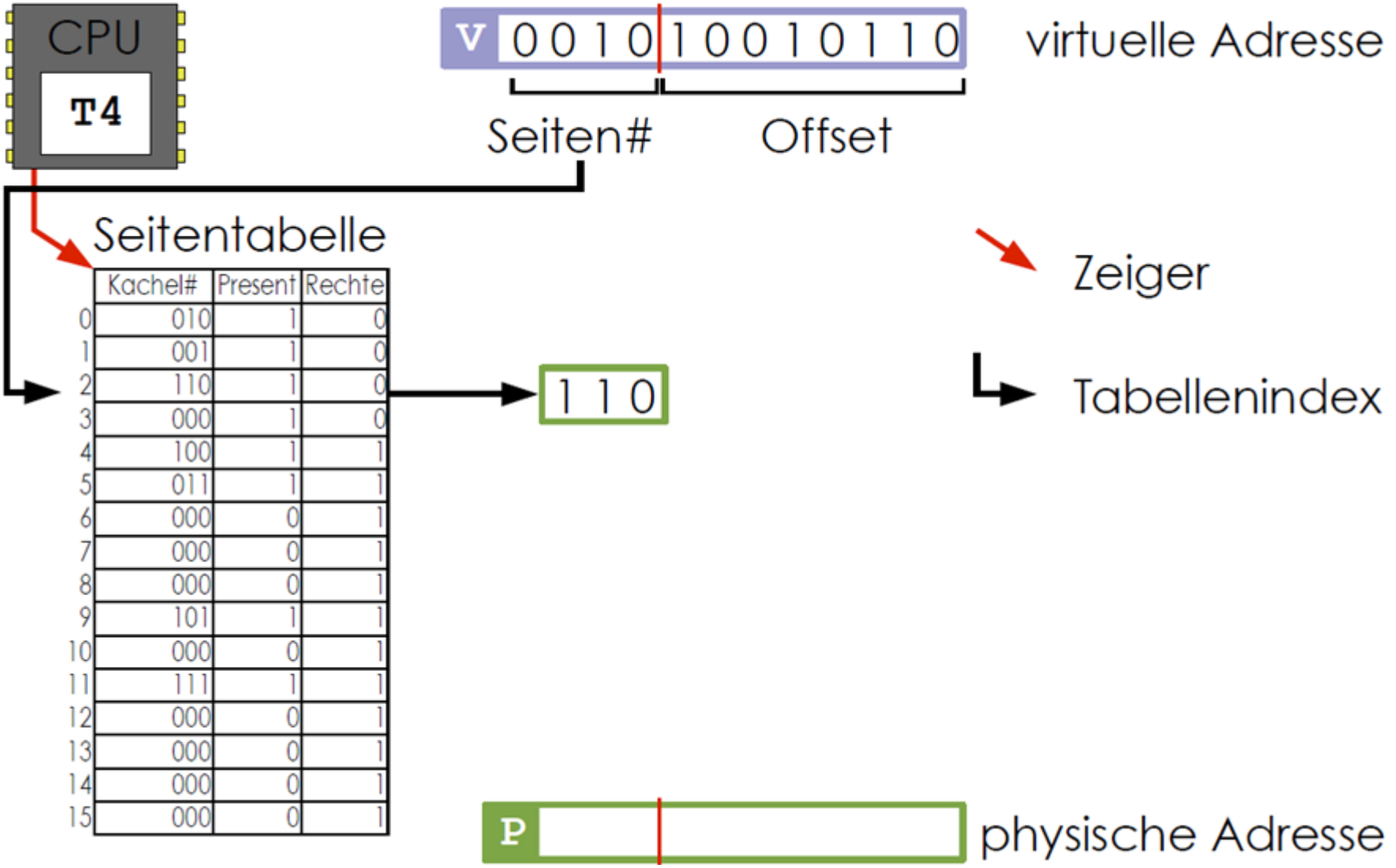


Seitentabelle

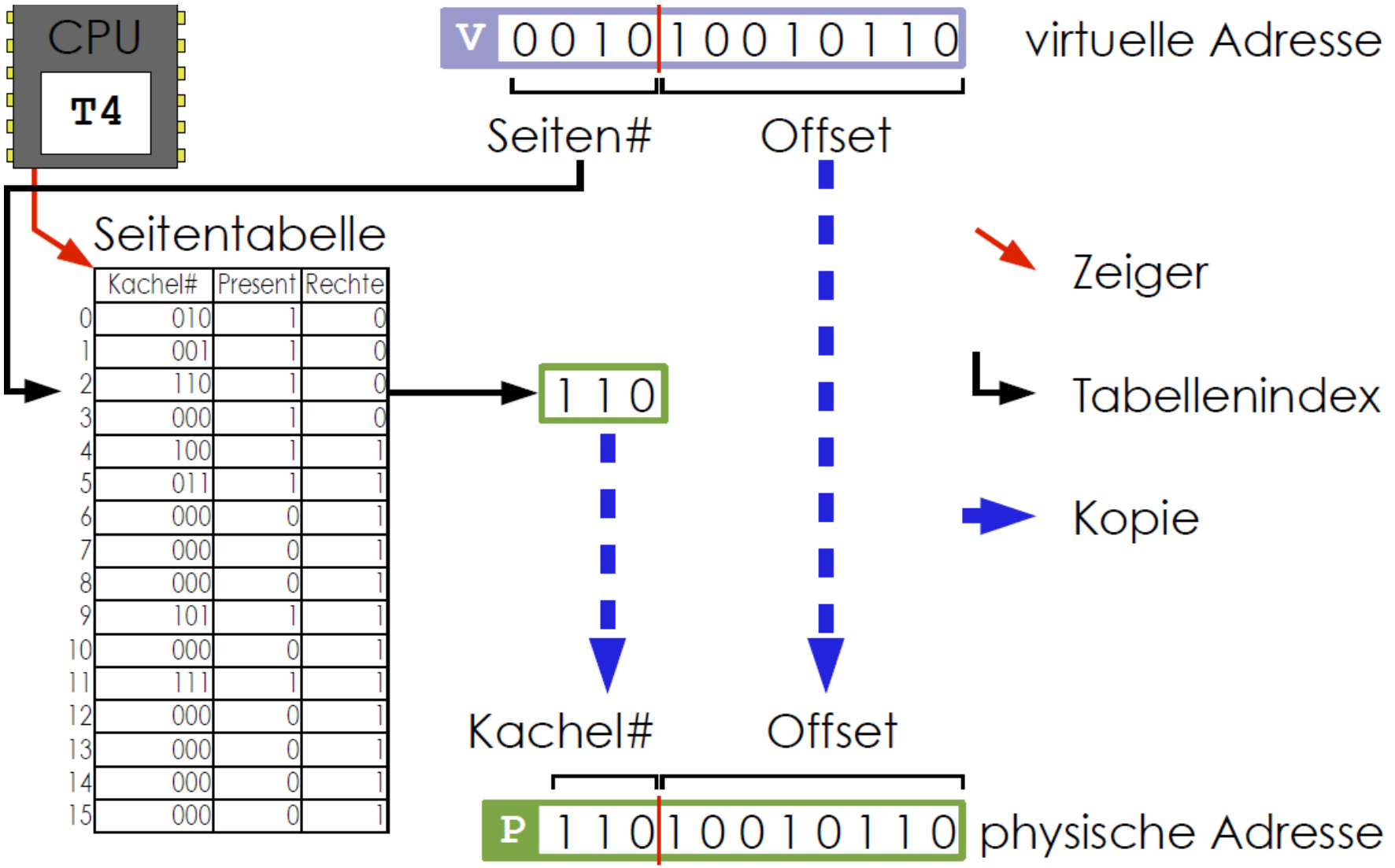
	Kachel#	Present	Rechte
0	010	1	0
1	001	1	0
2	110	1	0
3	000	1	0
4	100	1	1
5	011	1	1
6	000	0	1
7	000	0	1
8	000	0	1
9	101	1	1
10	000	0	1
11	111	1	1
12	000	0	1
13	000	0	1
14	000	0	1
15	000	0	1

Zeiger

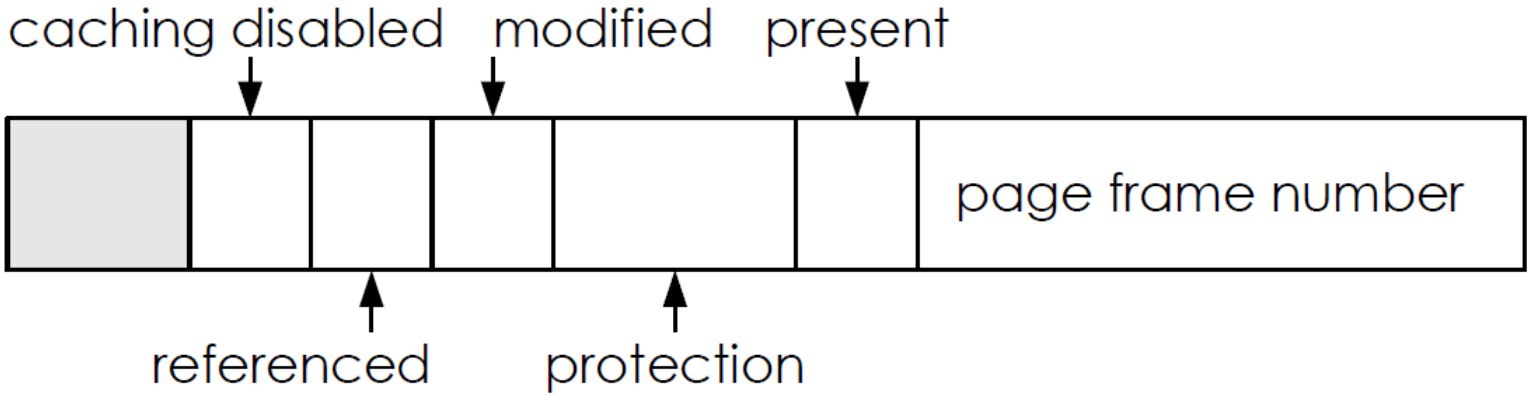
Prinzipielle Arbeitsweise der MMU (3)



Prinzipielle Arbeitsweise der MMU (4)



Prinzipielle Aufbau Seitentabelleneintrag



Seitenattribute:

- present Seite befindet sich im Hauptspeicher
- modified schreibender Zugriff ist erfolgt („dirty“)
- used irgendein Zugriff ist erfolgt
- caching ein/aus (z. B. wegen E/A)
- protection erlaubte Art von Zugriffen in Abhängigkeit von CPU-Modus

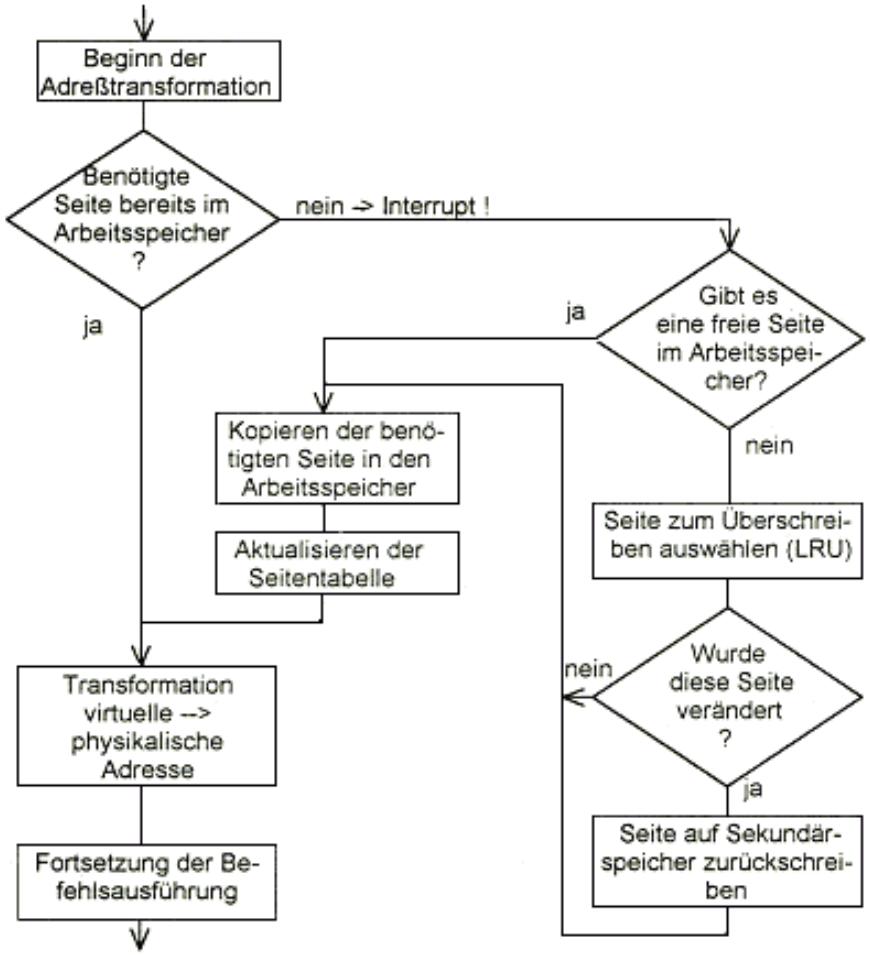
Protection:

mode \ operation	read	write	execute
kernel			
user			

Seiten-/Segmentwechsel-Algorithmen

- Einlagerungsstrategien:
  - „*pre-paging*“ : vorgeplante Einlagerung der in naher Zukunft benötigten Seiten, bevor sie vom Programm adressiert werden
  - „*demand paging*“ : Seiten werden nur bei Seitenfehler durch Auslösen einer Unterbrechung eingelagert
- Auslagerungsstrategien:
  - „*random*“ : es wird eine zufällig gewählte Seite ausgelagert
  - „*first in first out*“ (FIFO) : es wird stets die älteste Seite ausgelagert
  - „*least recently used*“ (LRU) : es wird die Seite ausgelagert, die am längsten nicht mehr benutzt wurde (in UNIX eingesetzt)
  - „*least frequently used*“ (LFU) : es wird die Seite ausgelagert, die am seltensten benutzt wurde
  - „*optimal replacement*“ : es wird die Seite ausgelagert, die am spätesten in der Zukunft wiederverwendet wird

Ablauf Seiten-/Segmentwechsel



- Auslagerungsstrategie
- "dirty bit" in der Seitentabelle

Bewertung

- Vorteile bei Einsatz von virtuellem Speicher mit Paging:
  - + geringere E/A-Belastung als bei vollständigem Swapping
  - + automatischer Speicherschutz: jeder Prozess kann nur auf seine eigene Seiten zugreifen !
  - + prinzipiell beliebig große Prozesse ausführbar
  - + keine externe Fragmentierung des Speichers
  - + Speicherplatz für jeden Prozess kann dynamisch vergrößert werden

- Nachteile:
  - – hoher Speicherbedarf für Seitentabellen
  - – hoher Implementierungsaufwand
  - – hoher CPU-Bedarf für Seitenverwaltung, falls Hardware-Unterstützung nicht vorhanden bzw. unzureichend

Tabellengröße und Geschwindigkeit

- **Problem 1: Größe – ein Beispiel**

Realspeicher :	256 MB
virtuelle Adressen:	32 Bit
Seitengröße:	4 KB
Aufteilung virt. Adr.:	20 Bit Index in der Seitentabelle
	12 Bit innerhalb einer Seite (Offset)

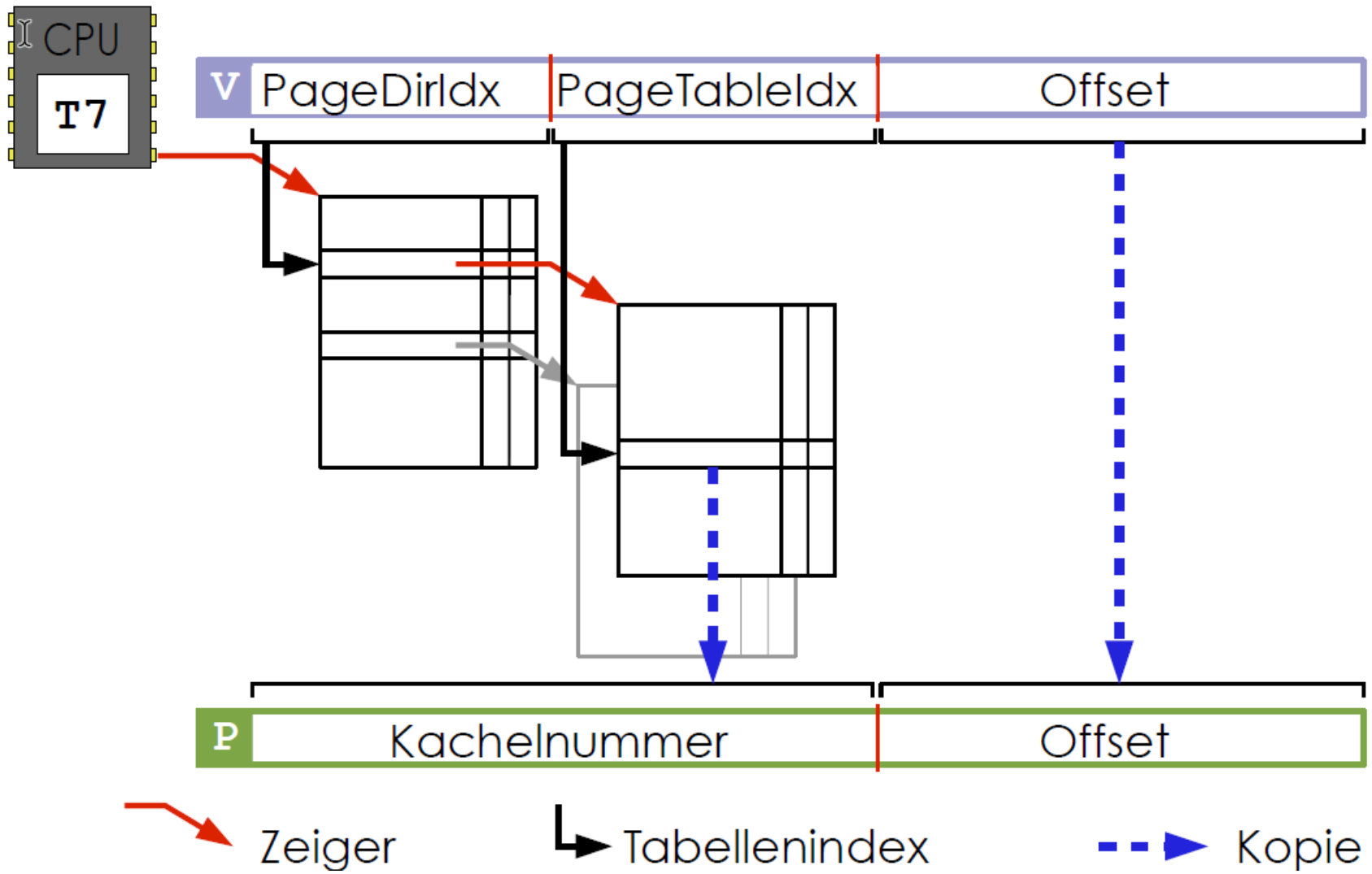
  → Größe der Seitentabelle für einen Adressraum: 2^{20} *4B → 4 MB

  → Bei 32 Prozessen 4*32 MB für die 32 Seitentabellen !

- **Problem 2: Geschwindigkeit der Abbildung – ein Beispiel**

CPU-Takt :	1 GHz → 2 Instruktionen in 1ns (4 Speicherzugriffe)
Speichertakt:	256 MHz → 4ns (... 70ns)

Problem 1: Baumstrukturierte Seitentabellen



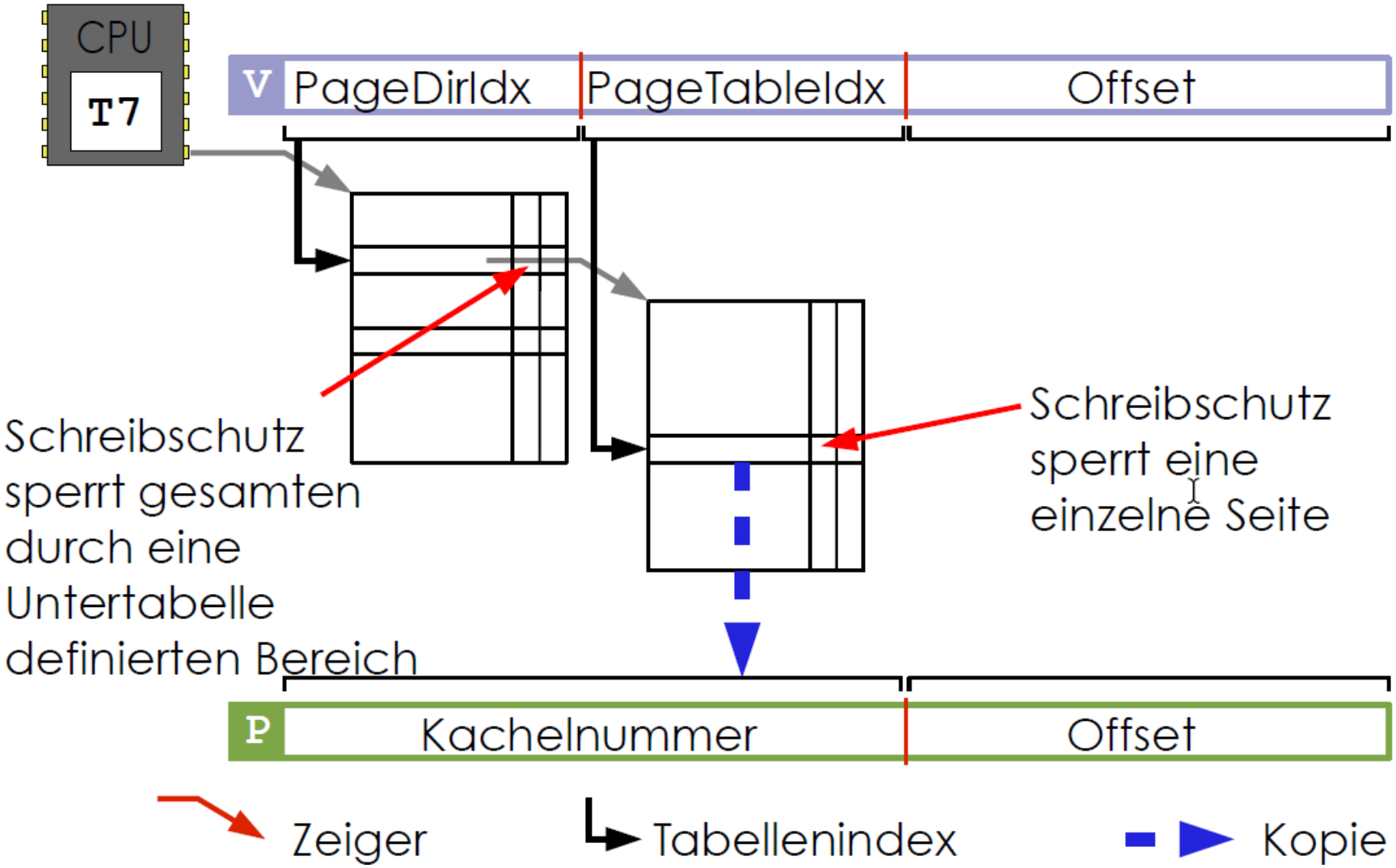
Eigenschaften baumstrukturierter Seitentab.

- Beliebig schachtelbar ⇨ 64-Bit-Adressräume!
- Seitentabellen nur bei Bedarf im Hauptspeicher bei Zugriff auf Seitentabelle kann Seitenfehler auftreten
- Zugriff auf Hauptspeicher wird noch langsamer, 2 oder mehr Umsetzungsstufen
- Hierarchiebildung möglich (nächste Folie)

  z. B. durch Schreibsperre in höherstufiger Tabelle ist ganzer Adressbereich gegen Schreiben schützbar
- Gemeinsame Nutzung („Sharing“)

  Seiten und größere Bereiche in mehreren Adressräumen gleichzeitig

Hierarchiebildung



Invertierte Seitentabellen

Seiten-Kachel-Tabelle

Seiten#

	Rahmen#	Attribs
0	0001	
1	0000	
2	1001	
3	0110	
4	0111	
5	1100	
6	1101	
7	0110	
8	0010	
	. . .	

Kachel-Seiten-Tabelle

Seiten#

	PID	Seiten#	Attribs
0	1	0001	
1	1	0000	
2	1	1000	
3	2	1000	
4	7	0001	
5	-	0000	
6	1	!	
7	1	0100	
8	2	0010	
	. . .		