

Betriebssysteme

Zusammenfassung

Stallings Kapitel 1-14

# Organisatorisches

## Klausur:

### ◆ nicht erlaubt

- eigene Unterlagen, Materialien, Lehrbücher, Folienkopien

### ◆ bitte mitbringen

- Ausweis
- Kugelschreiber
- Papier für Notizen (eigentlich nicht erforderlich)

### ◆ Zeit

- Montag 10.7. 16.00-17.30 Uhr

Wahl eines späten Nachmittagstermins auf Wunsch der Teilnehmer wegen des WM Endspiels am 9.7.

### ◆ Bewertungsschema:

- Vorleistung mit max 10 P
- 5 Aufgaben mit zusammen max 90 Punkten

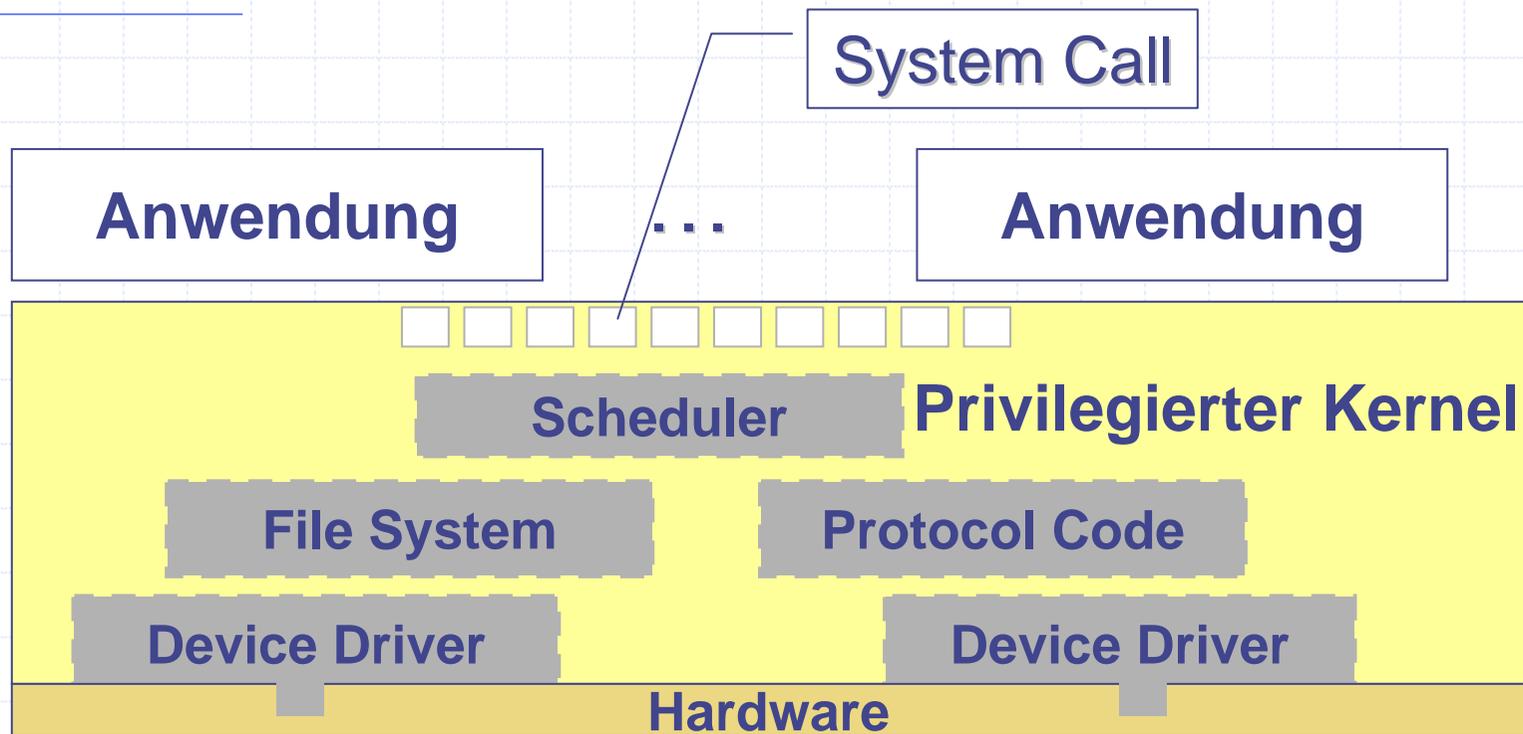
# Übersicht

- ◆ Einführung: Ein Betriebssystem - Was ist das ?
- ◆ Prozeßmanagement
  - Prozesse, Threads
  - Wechselseitiger Ausschluß, Deadlocks & Starvation
- ◆ Speicher Management
  - einfache Speicherverwaltung, virtueller Speicher, Paging
- ◆ Prozessor Scheduling
  - Time-sharing, Realtime, Scheduling bei 1 CPU, mehreren CPUs
- ◆ E/A Management, Festplattenscheduling
  - Schichtenmodell, SCAN Verfahren für Platten, RAID
- ◆ Datei Management
  - Dateiarten, Records, Blöcke, Unix: Inode
- ◆ Netzwerke

# Was ist ein Betriebssystem ?

- ◆ Ein Betriebssystem (OS: Operating System) ist Software, ein Programm. Es erlaubt andere Programme auf einer Hardwareplattform durchführen zu können.
- ◆ Ein Betriebssystem verwandelt Hardware in eine virtuelle Maschine. Diese virtuelle Maschine bietet alle Funktionen zum Betrieb von Anwendungsprogrammen. Die Besonderheiten zur Bedienung der jeweiligen realen Hardwareplattform werden durch das Betriebssystem verdeckt.
- ◆ Architekturmodelle für Betriebssysteme
  - monolithisches System
  - monolithischer Kernel, OS Prozesse haben privilegierten Kernelmodus
  - Mikrokernarchitektur, Kernel bietet lediglich minimalen Funktionsumfang, um Nachrichten zwischen Prozessen zu vermitteln, z.B. in variiertem Umfang in Windows 2000

## OS Variante: Monolithischer Kernel

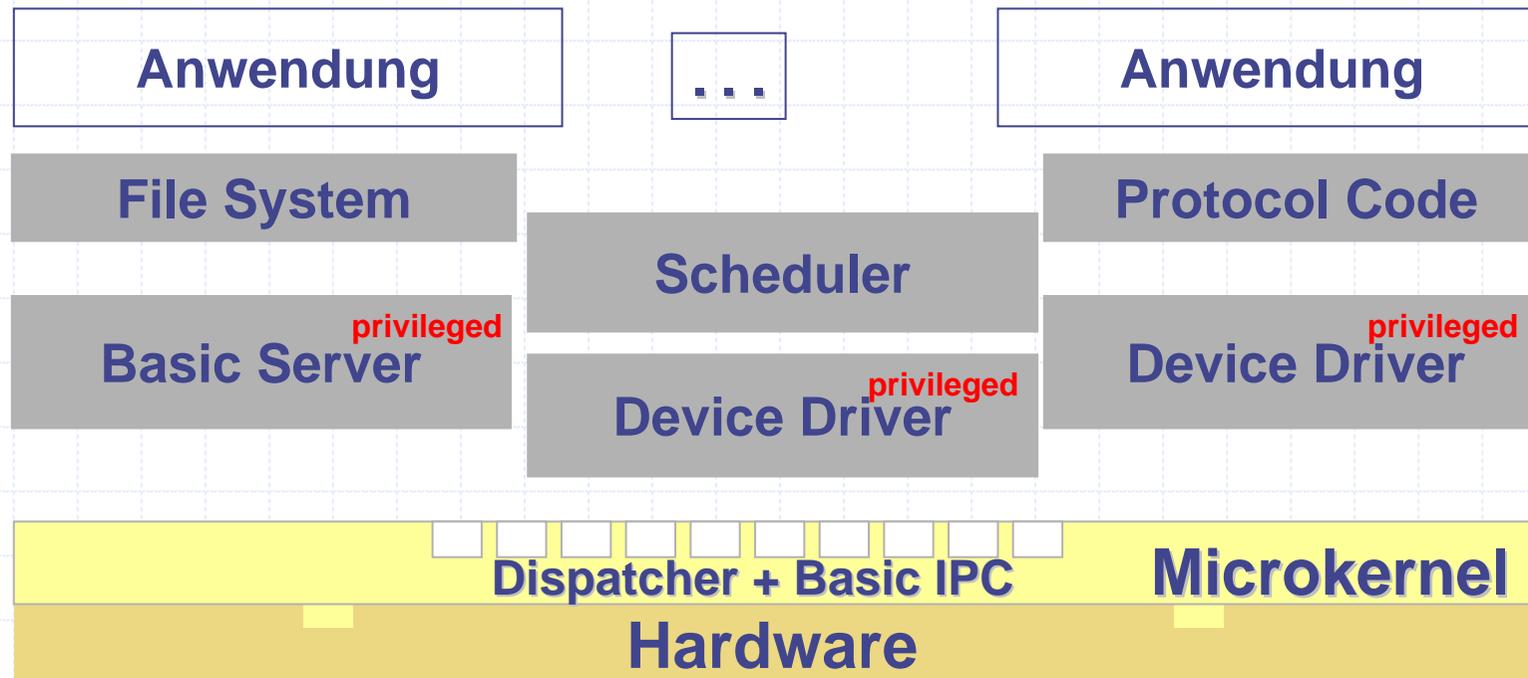


### 2. Ansatz, Unterscheidung von System- und Anwendungssoftware

#### Bewertung:

- Mehrzahl der OS Dienste im "privilegierten Kernel": =>
- Anwendungen: geschützt gegenüber anderen Anwendungen (virtueller Adreßraum), Anwendungen müssen sich auf Kernel verlassen
- Kernel Komponenten: geschützt gegen Anwendungen, aber nicht geschützt gegenüber anderen Kernel Komponenten.

## OS Variante: Mikrokernelarchitektur



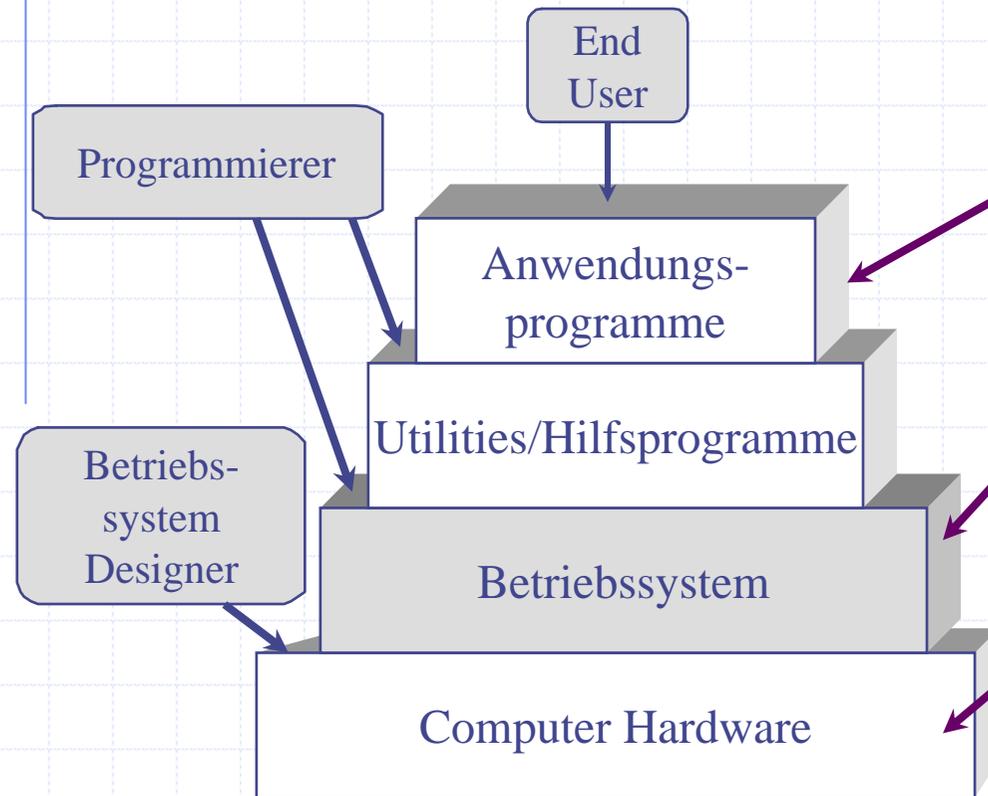
3. Ansatz: objektorientiertes Design, modulare Struktur, minimaler Kernel mit wenigen Operationen, Kommunikation über Nachrichtenaustausch

### Bewertung:

- klare Abgrenzung von Teilaufgaben, dynamischer Austausch von Teilen
- natürliche Erweiterung/Integration in verteiltes System
- ACHTUNG: PERFORMANCE IST KRITISCH

Beispiel: Windows 2000

# Schichten (auch Sichten) eines Computer Systems



Beispiel:

Powerpointpräsentation  
Powerpoint nutzt

- Dateisystem,
- logische E/A Geräte

Powerpointprozeß wird vom OS verwaltet, Dateisystem, E/A Geräte werden unter Nutzung realer Hardware verfügbar gemacht.

Das Betriebssystem vermittelt also zwischen Anwendungsprogrammen und realer Hardware. Z.B. Powerpoint hat eine „Save“ Funktion zum Speichern einer Präsentation in einer Datei. Das Betriebssystem stellt hierzu elementare Operationen zum Lesen / Schreiben von Dateien zur Verfügung und verwaltet Dateien.

## Prozeßmanagement: Prozeß ( auch Task )

- ◆ Ein Prozeß ist ein in Ausführung befindliches Programm.
- ◆ Unterschied zu "Programm": dasselbe Programm kann mehrfach aufgerufen werden. Dies führt zu mehreren individuellen Prozessen.

Für ein Betriebssystem bedeutet dies,

- dass mehrere Prozesse versetzt ausgeführt werden müssen, um den Prozessor auszulasten wobei gleichzeitig akzeptable Antwortzeiten für einzelne Prozesse eingehalten werden müssen.
  - dass Ressourcen (CPU, Speicher, E/A Geräte, Dateien) für diese Prozesse verfügbar gemacht werden müssen, wobei Konflikte erkannt und behandelt werden müssen.
  - dass Prozesse ggfs selbst Prozesse erzeugen und zwischen Prozesse eine Kommunikationsmöglichkeit bestehen muss.
- ◆ Multiprogramming, Multithreaded Applications

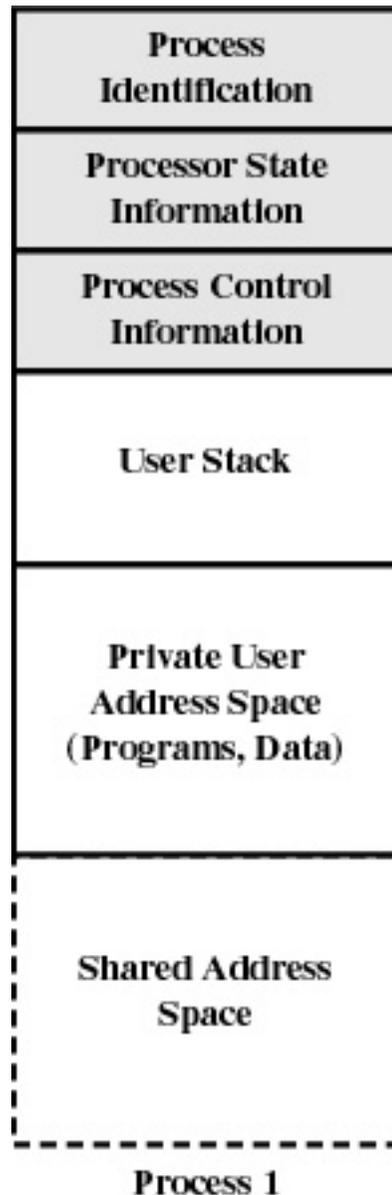
# Simultane Ausführung mehrerer Prozesse

## ◆ Prozeßmanagement

- Prozesse müssen unterbrechbar und fortsetzbar sein
- PSW, Prozeß/Moduswechsel, Zustandsdiagramm
- Kommunikation zwischen Prozessen: messages, pipes, geteilte Dateien

## ◆ Ressourcenverwaltung

- Generell: Gefahr von Verklemmungen/Deadlocks und Starvation/Verhungern
- Speicher Management (Hauptspeicher)
  - ◆ virtueller Speicher: Paging, logische vs physikalische Adressen Paging, Resident Set & Working Set
- CPU Scheduling
  - ◆ Time-sharing, Realtime, Scheduling bei 1 CPU, mehreren CPUs
  - ◆ Kurze Reaktionszeiten & Umlaufzeiten, Fair, kein Starvation
  - ◆ Schedulingverfahren
- E/A Management, Festplattenscheduling
  - ◆ Zuordnung von Devices: Rechte, Exklusiver Zugriff
  - ◆ Kurze Zugriffszeiten: SCAN Verfahren für Platten, Verfügbarkeit: RAID
- Datei Management
  - ◆ Directories, mit Zugriffsrechten
  - ◆ Dateiarten, Records, Blöcke, Unix: Inode
- Netzwerke
  - ◆ Kommunikation zwischen Prozessen
  - ◆ Verteilte Algorithmen: Schnappschuss, Zeitstempel, verteilte Warteschlange



Der PCB = Process Control Block beinhaltet Verwaltungsinformation für das OS:

- 1) Identifizierung, (wer ist das?)
- 2) CPU Status (wo weitermachen?)
- 3) Kontrollinformation (wann auswählen?)

Aufrufstack der Funktionen/Methoden im aktuellen Zustand der Bearbeitung

Speicherbereich für Daten/Objekte/Heap, und Programm-Code

Prozeßimage bildet logischen Block, der in der Realität aus einzelnen Speicherseiten (Pages) zusammengesetzt sein kann.

Je nach OS kann noch zusätzlich ein Kernelstack im Prozeßimage auftreten.

Prozeßmanagement arbeitet mit Prozeßtabelle und den PCBs der einzelnen Prozesse.

# Process Control Block

## Identifizierung eines Prozesses

- Numerische Bezeichner:
  - i.d.R. für den Prozeß selbst, für den erzeugenden Prozeß(parent), für den Benutzer (effektive Benutzer ID  $\neq$  User ID wg Privilegien)

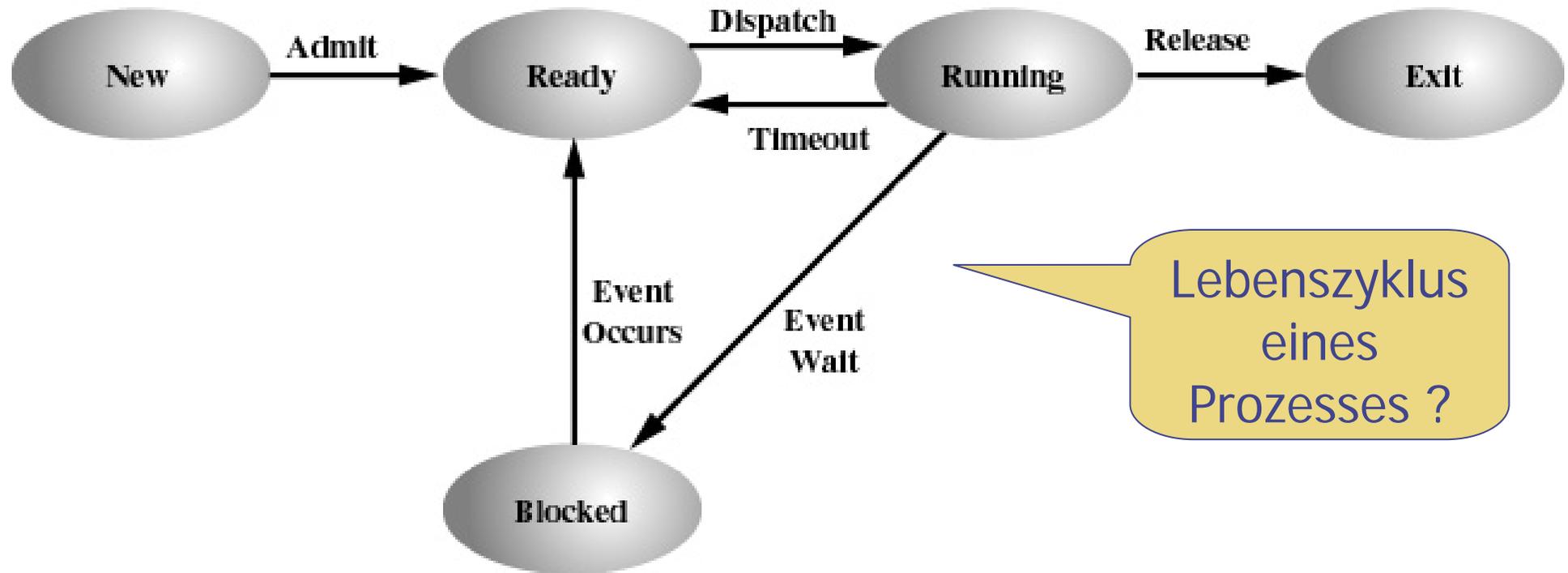
## Prozessor Status Information

- User-Visible Register
  - ◆ Instruktionen aus dem Code des Prozesses (in Maschinensprache) können diese Register referenzieren. Anzahl: typisch 8-32, einige RISC Architekturen haben über 100 Register
- Control und Status Register, PSW=Prozessor Status Word

PSW ist Grundlage für Prozeßwechsel !!!

Eine Reihe von Registern werden zur Steuerung der CPU genutzt:

- ◆ Program counter (PC): Adresse der nächsten zu ladenden Instruktion
  - ◆ Conditions Codes: Ergebnisse arithmetischer oder logischer Operationen (z.B. Vorzeichen, NULL, Übertrag, Gleichheit, Over/Underflow)
  - ◆ Status Information: z.B. Interrupt enabling/disabling flags, Ausführungsmodus (Kernel, User Mode, ...)
- + weitere Informationen ...



Lebenszyklus eines Prozesses ?

Abb. 3.5 aus Stallings: Prozeßmodell  
 mögliche Zustände und Zustandsübergänge eines Prozesses  
 Eine Unterscheidung von genau 5 Zuständen ist nicht zwingend!

## Feinere Unterscheidung für reales OS, z.B. UNIX : Swapping und User/Kernel Modus

### Swapping:

- ◆ Motivation: die CPU ist wesentlich schneller als E/A Geräte, so daß viele/alle Prozesse auf E/A warten können
- ◆ Idee: Auslagern wartender Prozesse auf Plattenspeicher schafft freien Hauptspeicher für einen größeren/weiteren Prozeß
- ◆ Swapping impliziert 2 neue Zustände im Zustandsdiagramm
  - Blocked/sleeping, swapped
  - Ready, swapped

### User/Kernel Modus:

- ◆ Unterscheidung nach Art des Prozesses, Anwendungsprozeß oder OS Prozeß, bzw nach den aktuellen Rechten des Prozesses

# UNIX Prozeßmodell

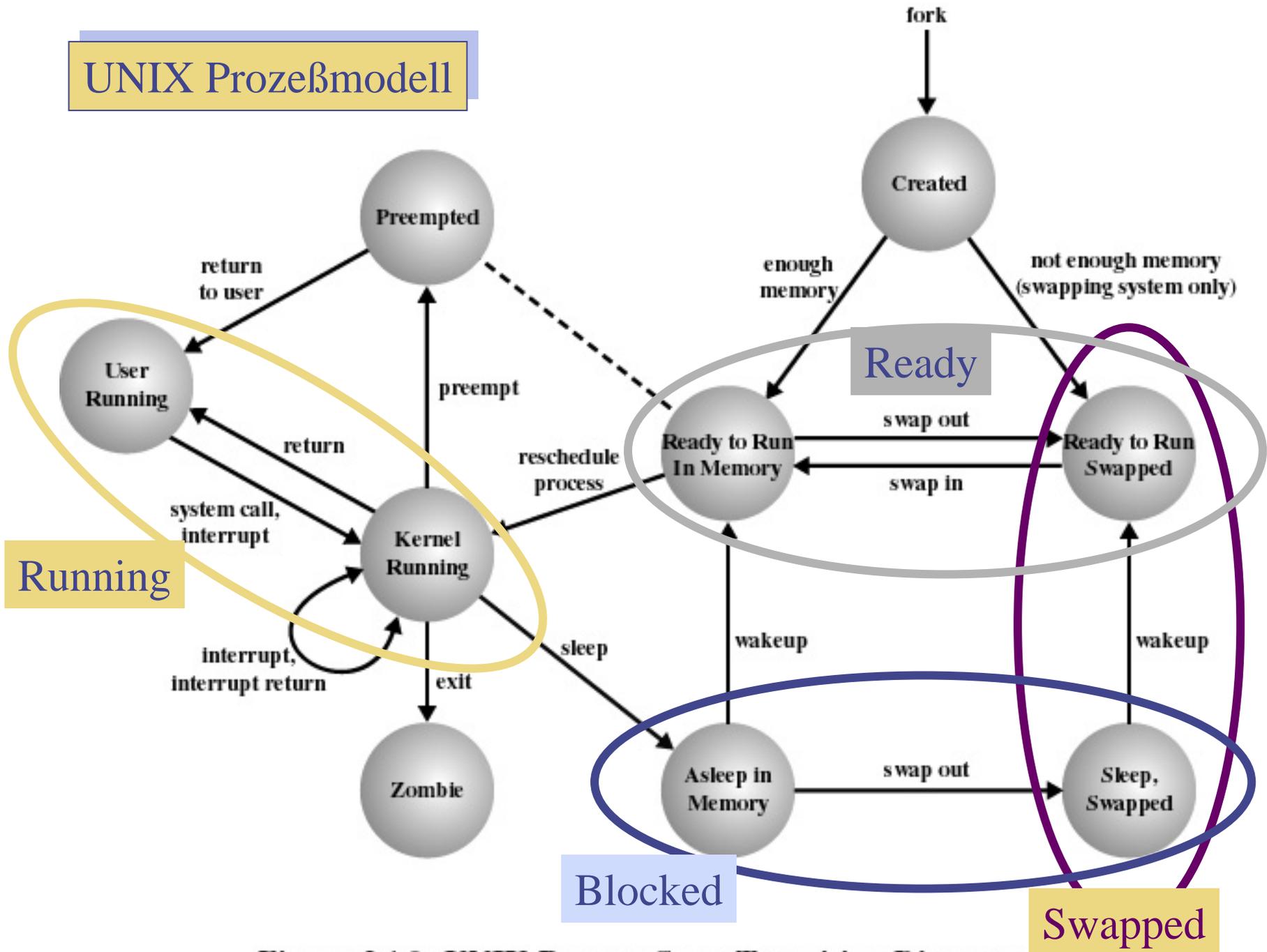


Figure 3.16 UNIX Process State Transition Diagram

## Wann erfolgt ein Prozesswechsel ?

### ◆ Clock Interrupt

- Prozeß wurde maximal erlaubte Zeit von CPU bearbeitet (Zeitscheibe erschöpft)

### ◆ E/A Unterbrechungssignal (I/O Interrupt)

### ◆ Speicherfehler (Memory Fault, Page Miss)

- virtuelle Adresse hat zur Zeit keine physikalischen Gegenpart im Hauptspeicher (Datentransfer Festplatte -> Hauptspeicher)

### ◆ Trap

- ein Fehler ist aufgetreten,
- kann zu Prozeßterminierung führen (EXIT)
- kann zu Recoverymaßnahmen, Benachrichtigung des Anwenders führen

### ◆ Supervisor Call, System Call

- z.B. Öffnen einer Datei im Kernel Mode

## Wie erfolgt ein Prozesswechsel ?

1. CPU Kontext, PSW speichern
2. PCB aktualisieren
3. PCB in Warteschlange verschieben, je nach Status:  
ready, blocked
4. anderen Prozeß auswählen
5. PCB des ausgewählten Prozesses aktualisieren
6. Memory Management Datenstrukturen aktualisieren
7. Kontext rekonstruieren (laden) für den ausgewählten Prozess



Aufwand!!!

### Einfachere Variante: Moduswechsel User-Kernel Modus

erfordert nur Schritte 1 & 6, um die Bearbeitung des Anwendungsprozesses zu unterbrechen, eine OS Routine (z.B. Interrupthandler und Dispatcher) durchzuführen und anschliessend mit demselben Anwendungsprozeß weiterzumachen.

# Übersicht

## ◆ Konzepte zur Beschreibung von parallelen Anwendungen

- Prozesse
- Threads

## ◆ Threads:

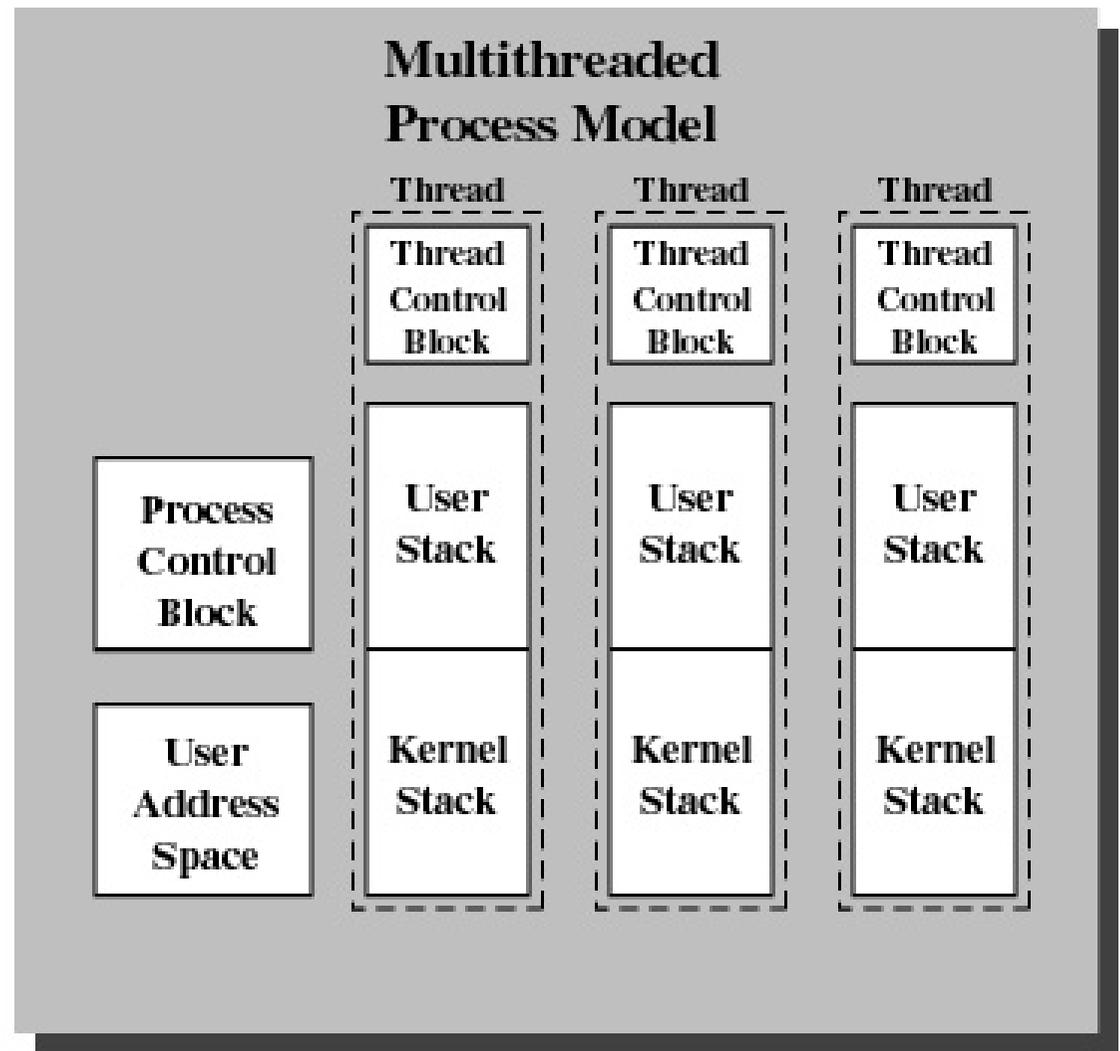
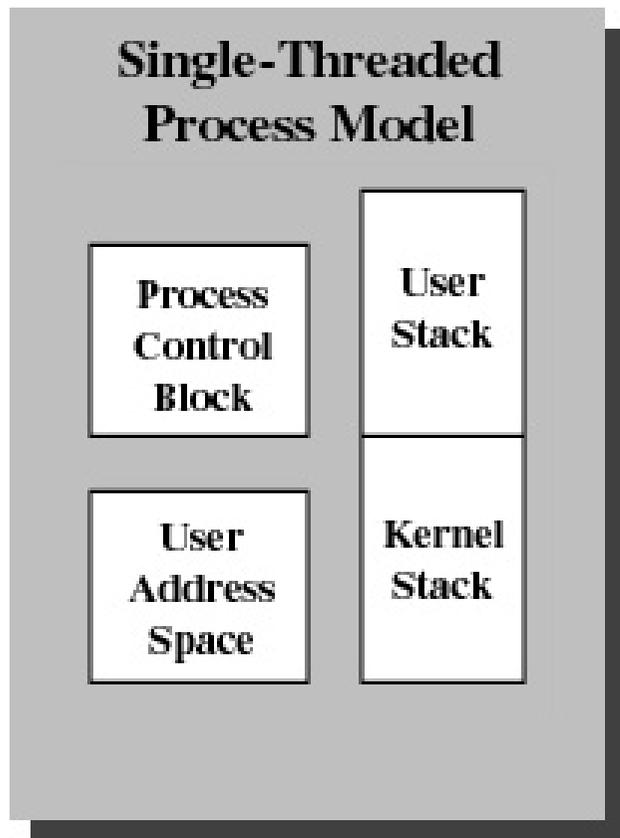
- Was ist das ?
- Wofür setzt man Threads ein ?
- Wie programmiert man mit Threads in Java ?

## ◆ Betriebssysteme und Threads

- Wie werden Threads durch Betriebssysteme unterstützt ?
- Wie wird eine Mehrprozessorarchitektur für Threads genutzt ?
  - ◆ Symmetrischer Mehrprozessorbetrieb
- Wie werden Prozesse/Threads zur Gestaltung von Betriebssystemen genutzt ?
  - ◆ Microkernel Architektur

## ◆ Threadmanagement in realen Betriebssystemen

- Windows 2000



Stallings Fig. 4.2 Prozeßmodell ohne Threads und mit Threads

je Prozeß: Verwaltung (PCB) + Ressourcen/Adreßraum

je Thread: Verwaltung (TCB) + Thread-interner Aufrufstack

# Bewertung des Threadkonzeptes

Pro:

- ◆ Performancegewinne bei der
  - der Erzeugung eines Threads,
    - ◆ es wird kein neues Prozeßimage angelegt
  - der Terminierung eines Threads,
    - ◆ es wird nur beim letzten Thread ein Prozeßimage abgebaut
  - dem Wechsel von Threads innerhalb eines Prozesses,
    - ◆ es wird im wesentlichen der CPU Kontext gesichert, weniger Datenstrukturen aktualisiert
  - der Kommunikation zwischen Threads innerhalb eines Prozesses
    - ◆ es wird nicht jeweils der Kernel bemüht, um Sicherheitskonzepte zu prüfen und die Kommunikation herzustellen, sondern lediglich auf geteilten Speicher zugegriffen

im Vergleich zu gleichwertigen Operationen für Prozesse

- ◆ ermöglicht einfachere Programmstrukturen, Softwarearchitekturen bei nebenläufigen Aktivitäten in Programmen

Contra:

- ◆ erweiterte Verwaltungsdatenstrukturen für Prozesse
- ◆ Threadorientiertes Schedulingverfahren

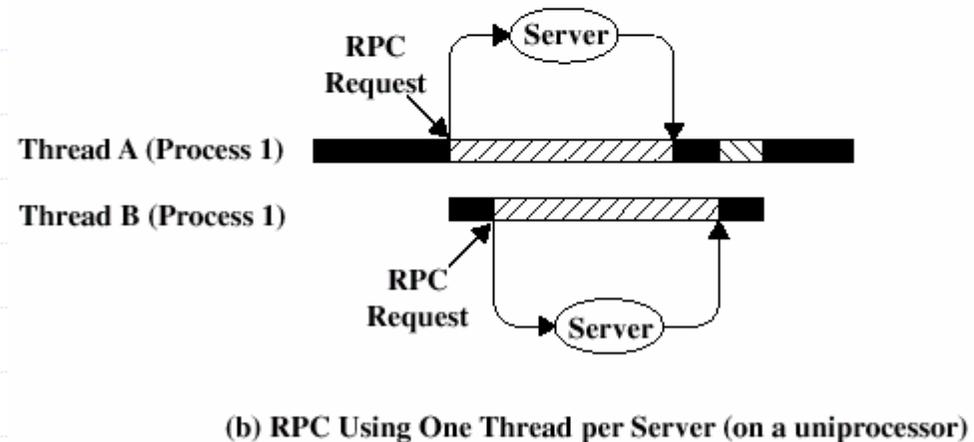
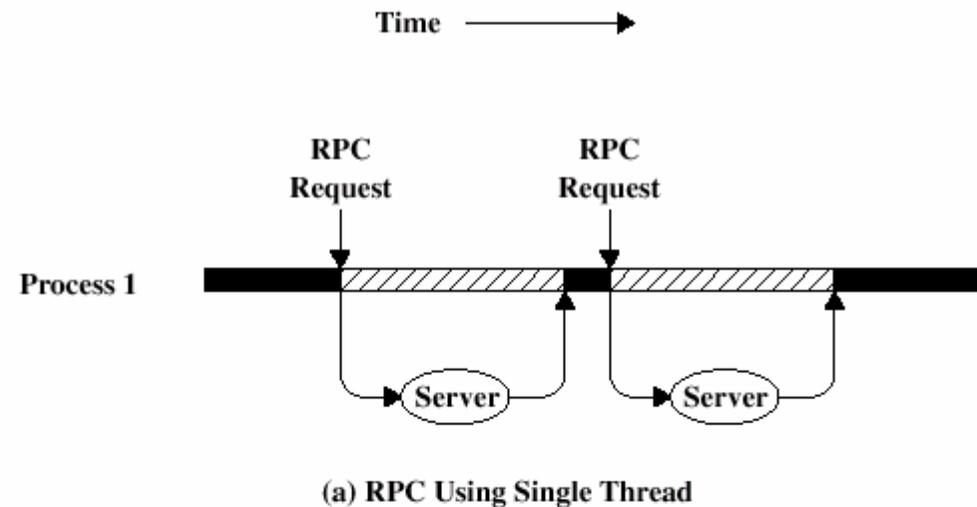
## Threads bei nur 1 CPU ?

Threads können wg Blockierungen bei E/A etc und zur Verbesserung der Softwarearchitektur auch bei Einprozessormaschinen Sinn machen.

Beispiel

Remote Procedure Calls für unterschiedliche Server falls Ergebnisse von RPC 1 für Parameter von RPC 2 nicht erforderlich sind

- a) mit 1 Thread
- b) mit 2 Threads



-  Blocked, waiting for response to RPC
-  Blocked, waiting for processor, which is in use by Thread B
-  Running

(Stallings, Fig.4.4)

# Ein kurzer Ausflug in Threadprogrammierung in Java

## Wie sind Threads in gängige Programmiersprachen integriert ?

- C/C++: OS spezifische Bibliotheken, zusätzlich POSIX Standard konforme Bibliotheken mit API zur Threaderzeugung, Synchronization, mit Semaphoren etc. ...
- Java: unmittelbar im Sprachumfang enthalten,
  - ◆ je nach Virtual Machine: Green Threads, Native Threads

## Threads in Java

- durch Vererbung aus der Thread Class
- durch ein Runnable Interface

## ◆ Lebenszyklus

- Deklaration: `Thread t = new Thread(this);`
- Initiieren: `t.start()`
- Ablaufen lassen: `run()`
- Terminieren -> durch Terminierung von `run()`

# Threads in Java

## ◆ Thread Scheduling:

- `yield()`: freiwillige Aufgabe des Prozessors
- `getPriority()`, `setPriority()`: Prioritäten zur programmgesteuerten Auswahl eines Threads

## ◆ Wechselseitiger Ausschluß:

Ziel des wechselseitigen Ausschlusses ist es, dass jeweils nur 1 Thread zu einem Zeitpunkt den Zugriff auf eine Ressource/ein Datum erhält

## Maßnahmen

- `Synchronized`: je Objekt existiert in Java ein Lock (=Sperre), der in einer atomaren Operation gesetzt oder freigegeben werden kann. Java VM sichert anschliessend, dass alle Methoden mit Attribut `Synchronized` jeweils nur von demjenigen Thread ausgeführt werden können, der den Lock gesetzt hat.
- `Wait()`: Warten auf Benachrichtigung.
- `Notify()`, `NotifyAll()`: Versenden einer Benachrichtigung

`Wait-Notify` wird eingesetzt, um eine Reihenfolge in der Bearbeitung von Teilschritten zu erreichen, z.B. falls A vor B, B vor C, C vor D bearbeitet werden muss und 1 Thread A, C ein 2. Thread B und D bearbeitet:

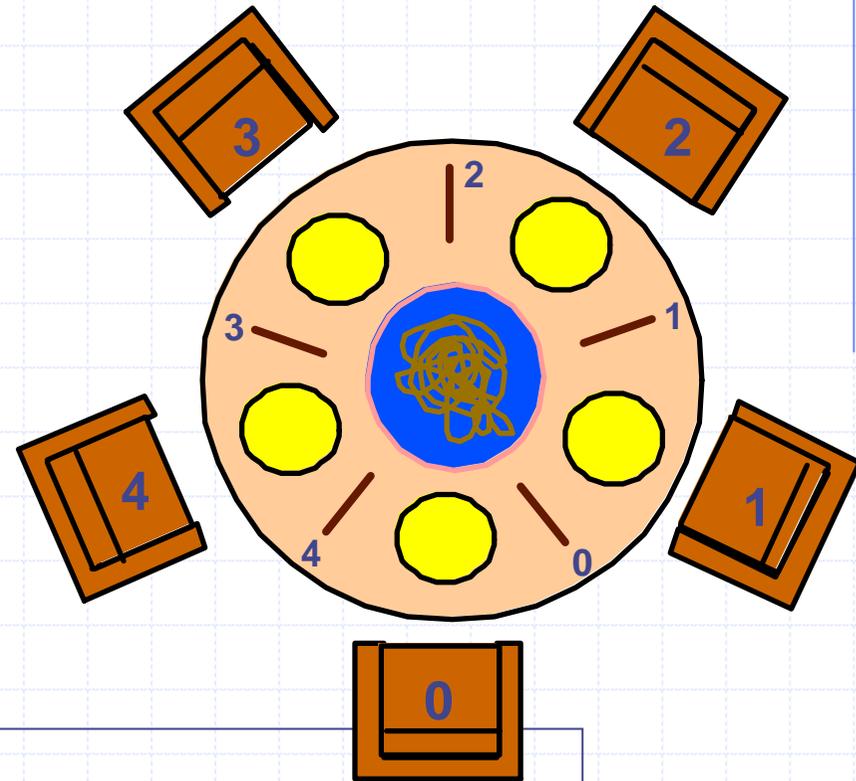
1. Thread: A, **notify**, **wait**, C, **notify**      2. Thread: **wait**, B, **notify**, **wait**, D

## Übersicht: Probleme durch nebenläufige Prozesse

- ◆ Die simultane Nutzung von Ressourcen durch nebenläufige Prozesse kann eine destruktive Wirkung haben.
  - Abhilfe: Ressourcennutzung im wechselseitigen Ausschluß  
Verfahren/Konzepte wechselseitigen Ausschluß zu erreichen
    - ◆ Software Lösungen: Dekker/Dijkstra, Peterson
    - ◆ Semaphore
    - ◆ Monitore
    - ◆ Nachrichtenübermittlung (Message Passing)
  - Wechselseitiger Ausschluß zeigt Nebenwirkungen
    - ◆ Verklemmung (Deadlock)
      - Bedingungen
      - Strategien: 1) ignorieren, 2) verhindern 3) vermeiden  
4) erkennen & beseitigen
    - ◆ Verhungern (Starvation)
- ◆ Bei Prozessen, die durch geteilten Speicher oder Nachrichten kommunizieren, kann es ebenfalls zu Verklemmungen und Verhungern kommen.

## Ein Klassiker: Dining Philosophers Problem

5 Philosophen verbringen ihr Leben mit Essen und Denken in stetigem Wechsel. Am gemeinsamen Tisch finden sich jedoch nur 5 Gabeln, von denen jeder Philosoph jeweils die Gabel zu seiner rechten und zu seiner linken Seite zum Essen verwendet.



Prozesse: Philosophen

Ressourcen: Gabeln

Besonderheit: 2 Ressourcen je Aktion erforderlich

Gesucht: Algorithmus mit 5 Threads (Philosophen) und 5 geteilten Objekten (Gabeln) der unbegrenzten Ablauf ohne Deadlocks und Starvation erlaubt.

## Dining Philosophers

Deadlock tritt auf, wenn

- jeder Philosoph Gabeln nacheinander aufnimmt
- alle Philosophen bei der Gabelaufnahme die gleiche Reihenfolge wählen
- und alle Philosophen 1 Gabel aufgenommen haben.

Starvation kann auftreten, wenn

- jeder Philosoph beide Gabeln gleichzeitig aufnimmt oder wartet bis beide frei sind.

Lösung ohne Deadlock & Starvation:

1. Anzahl Philosophen am Tisch wird auf 4 begrenzt
2. ... und natürliche existieren weitere Lösungen

Dining Philosophers sind eine interessante Programmieraufgabe zur Anwendung von Threads und Monitoren!

# Wechselseitiger Ausschluß: Semaphore

Grundidee für Mutex:

Semaphore sichert Zutritt zu kritischem Abschnitt

Dijkstra '65: binäre Semaphore mit Werten  $\{0,1\}$  und Operationen P, V

Schwache Semaphore

- Auswahl des nächsten wartenden Prozesses aus der Warteschlange ist beliebig, Verhungern ist daher möglich

Starke Semaphore

- FIFO-Auswahl aus der Warteschlange, d.h. der Prozeß, der am längsten gewartet hat, wird als erster fortgeführt
- Verhungern nicht möglich

Beurteilung:

- ◆ Pro: kein aktives Warten, signal() Operation weckt ggfs wartenden Prozeß
- ◆ Contra: bei mehreren Semaphoren ist die Verteilung der wait-signal Operationen im Code sehr fehlerträchtig
- ◆ Implementierung: führt zu Mutex Problem auf nächster Ebene für wait, signal
  - Hardware, Firmware Lösung, Dekker, Peterson, test&set Operation

## Wechselseitiger Ausschluß: Monitor

Ein Monitor ist ein Software Modul

- ◆ Lokale Variable nur für Monitor sichtbar (Kapselung)
  - ◆ Prozeß nutzt (betritt) Monitor über Monitorfunktionen
  - ◆ Nur 1 Prozeß kann zu 1 Zeitpunkt Monitor aktiv nutzen, andere Prozesse werden durch den Monitor suspendiert
  - ◆ Innerhalb des Monitors können Prozesse mit **wait** zurücktreten und später von anderen Prozessen im Monitor mit **notify** geweckt werden
- } Typisch für Objekte

Wir betrachten Ansatz von Lampson/Redell:

- ◆ **cnotify** legt die Verantwortung für die Bedingung auf wartende Prozesse
- ◆ **cwait(c)**: Prozeß blockiert ggfs wg Bedingung **c** und gibt dabei den Monitor für andere Prozesse frei
- ◆ **cnotify(c)**: Prozeß gibt Signal zum Wecken für 1 suspendierten Prozeß, kann dann aber selbst fortfahren, ganz analog **cnotifyAll(c)**
- ◆ Suspendierte Prozesse, die durch ein **notify** gelegentlich geweckt werden, müssen Bedingung erneut prüfen daher stets:  
**while (not c) cwait( c ) ;**
- ◆ Vorteile: einfachere Programmierung, modular, robuster

## Monitore in Java

In Java gehört zu jedem Objekt ein eigener Monitor.

Überwachte Methoden sind als **synchronized** gekennzeichnet.

**Monitor und Objekt** kapseln somit Daten + Zugriffsmethoden, so daß bzgl der Methoden wechselseitiger Ausschluß gilt. Zusätzlich existiert die Möglichkeit zur freiwilligen, vorübergehenden Suspendierung des aktiven Threads, wobei der Monitor für andere Prozesse/Threads frei wird (Mutex !!!)

```
while (not c) wait() ;
```

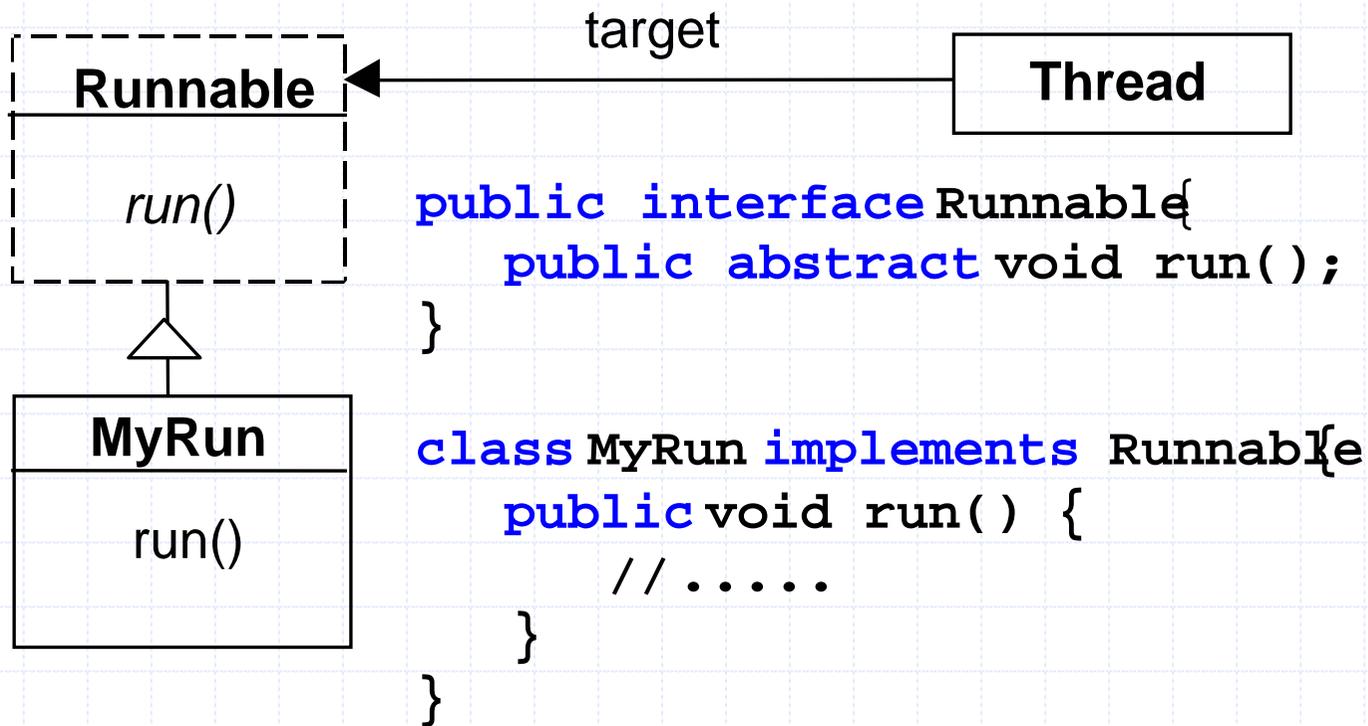
Die Änderung einer Bedingung kann die jeweils aktive Funktion (der aktive Prozeß/Thread) den suspendierten Threads mittels **notify()** bzw **notifyAll()** mitteilen. In **Java**: der aktive Thread bleibt aktiv. Geweckte Threads prüfen zu späterem Zeitpunkt die Gültigkeit der Bedingung.

Das Monitorkonzept hat entscheidende Vorteile:

- implizite Handhabung zur Belegung / Freigabe von Locks
- Kapselung von Belegung und Freigabe
- kein aktives Warten

# Threads in Java

Implementierung eines Interfaces "Runnable".

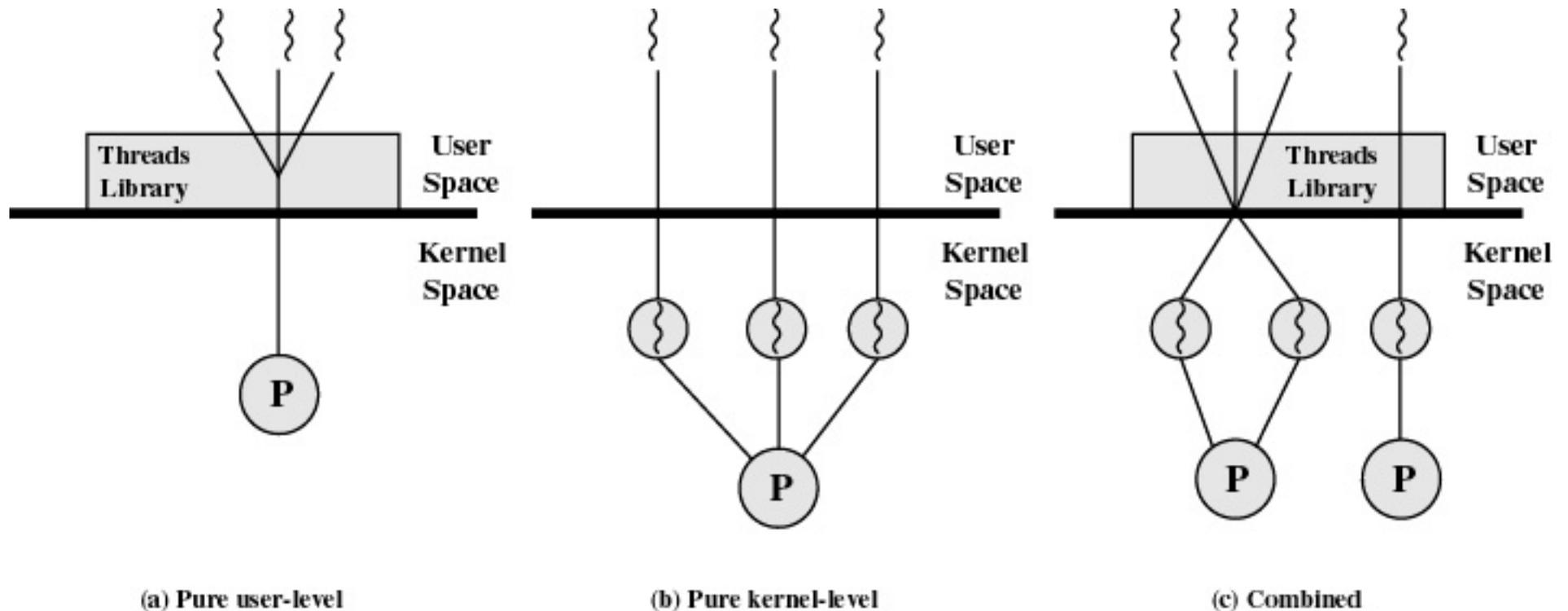


Wie erzeugt MyRun Methode Thread mit eigener run() Methode?  
Thread lokalMyRun = new Thread(this);

## Threads in Java

- ◆ Threads durch Vererbung oder Implementierung des Runnable Interfaces
- ◆ Methoden zum Initialisieren (start) und Betreiben (run)
- ◆ weitere Thread Unterstützung in JAVA:
- ◆ Kommunikation/Synchronisation:
  - wait-notify: Warten und Benachrichtigen (Aufwecken)
  - interrupt- wait,sleep,join (ähnliches Prinzip)
- ◆ Scheduling
  - yield : freiwillige Aufgabe des Prozessors
  - getPriority, setPriority: Prioritätenvergabe durch Programmierer Java  
Thread Scheduling der VM arbeitet nach Prioritäten und unterbrechend

Java zeigt, dass Threads auf Anwendungsebene sichtbar sind.  
Heißt das, dass Threads ebenso vom OS gesehen werden?  
User-level Threads vs Kernel-level Threads



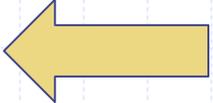
Stallings Fig 4.6: Threads aus OS Sicht, 3 Umsetzungsvarianten

a) OS sieht nur 1 Prozeß, Threads existieren nur innerhalb der Applikation

b) OS sieht n Threads, führt Thread Scheduling durch (Windows 2000, Linux)

c) OS sieht m Kernel Threads = leichtgewichtige Prozesse bei n Threads innerhalb der Applikation (UNIX-Solaris)

# Übersicht

- ◆ Einführung: Ein Betriebssystem - Was ist das ?
- ◆ Prozeßmanagement
  - Prozesse
  - Threads
  - Wechselseitiger Ausschluß, Deadlocks & Starvation
- ◆ Speicher Management
  - einfache Speicherverwaltung  HEUTE !
  - virtueller Speicher
- ◆ Prozessor Scheduling
- ◆ E/A Management, Festplattenscheduling
- ◆ Datei Management
- ◆ Netzwerke
- ◆ Sicherheit

# Übersicht

- ◆ Grundlagentechniken der Speicherverwaltung
  - Feste Partitionierung (fixed partitioning)
  - Dynamische Partitionierung (dynamic partitioning)
  - einfaches Seitenverfahren (simple paging)
  - einfaches Segmentierungsverfahren (simple segmentation)
- ◆ Vorgang beim Binden und Laden eines Programms
- ◆ Paging und Segmentierung
  - grobe Vorgehensweise
  - Adreßtransformation, i.w. hardwareunterstützt
- ◆ Strategien zur Gestaltung eines virtuellen Speichers, die in OS Software umgesetzt werden.  
Wir fokussieren auf Paging und betrachten
  - das Laden von Seiten
  - das Plazieren von Seiten
  - das Ersetzen von Seiten
  - das Verwalten des Resident Sets
  - das Sichern von Seiten

## Wiederholung: Speicherverwaltung bei Multiprogramming

- ◆ Anforderungen an Speicherverwaltung bei n Prozessen:
  - jeder Prozeß muß ausreichend Speicher erhalten
  - Versorgung der CPU (schnellste Komponente im System!) mit Instruktionen und Daten möglichst verzögerungsfrei
  - die Größe aller Prozeßimages darf den Hauptspeicherplatz überschreiten

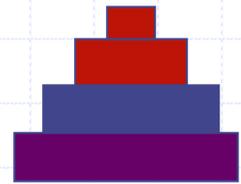
- ◆ dafür verfügbare Ressourcen: Speicherhierarchie

Level 1 & Level 2 Caches, Hauptspeicher, Plattenspeicher, ...

zunehmendes Fassungsvermögen, zunehmende Zugriffszeiten

- ◆ Lösungen

- Paging: Aufteilen des Hauptspeichers in **gleichgroße** Partitionen (Seitenrahmen),  
ein Prozeß belegt eine variable Anzahl von Seitenrahmen
- Segmentierung: Aufteilen des Hauptspeichers in **variabel große** Partitionen je nach Bedarf eines Prozesses



## Grundlagen und Grundidee des virtuellen Speichers

- ◆ Trennung von logischem und physikalischem Adreßraum
- ◆ Prozeßimage enthält logische Adressen
  - erfordert Umrechnung in physikalische Adressen zur Laufzeit
    - ◆ mit Seiten / Segmenttabellen
    - ◆ mittels Aneinanderhängen / Addition
    - ◆ mit Bereichsbegrenzung (Schutzfunktion)
  - erlaubt beliebige Position des Images im Hauptspeicher
- ◆ Paging/Segmentierung stellt Prozeß durch Menge von Speicherbereichen dar.

### Virtueller Speicher

Nur ein Teil der Seiten/Segmente eines Prozesses belegen Platz im Hauptspeicher, fehlende Bereiche liegen auf der Festplatte (im Swap Space). Bereiche werden ggfs geladen, bei Platzmangel im Hauptspeicher auch im Wechsel.

## Vorteile des virtuellen Speichers

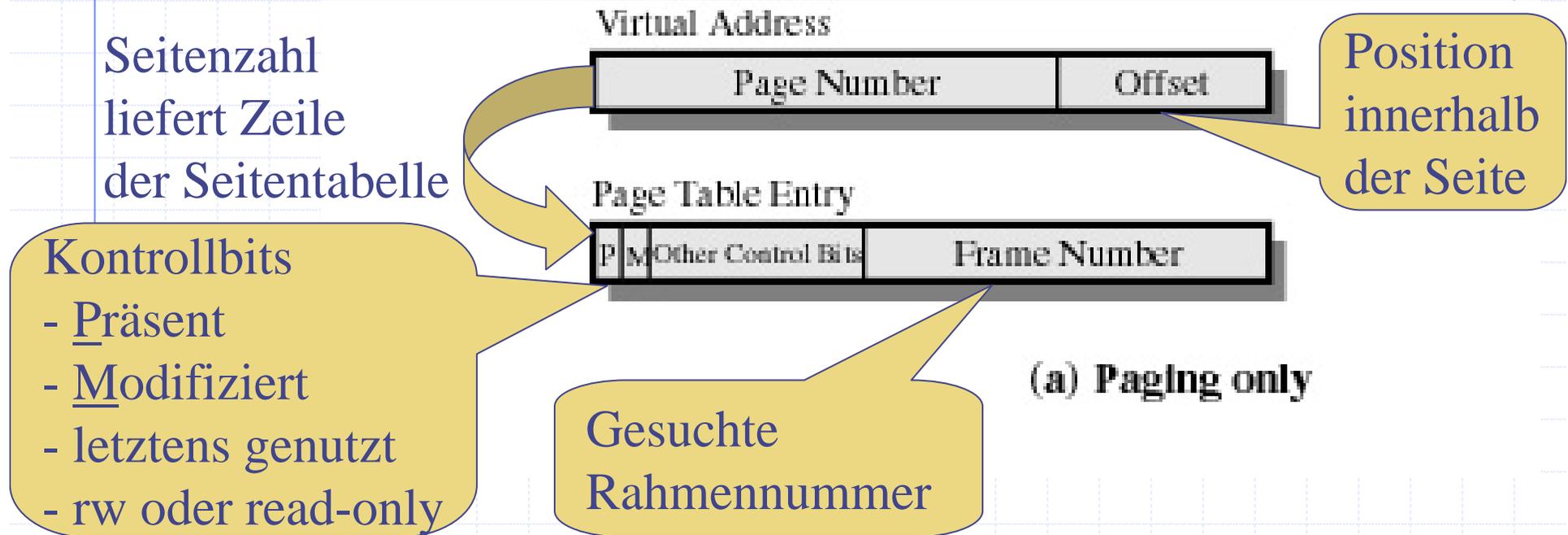
- ◆ Prozessimages dürfen größer als der Hauptspeicher sein
  - aber nicht größer als der Festplattenplatz
- ◆ größere Anzahl von Prozessen „simultan“ bearbeitbar
  - es werden von vielen Prozessen nur relativ kleine Teile geladen
  - viele bearbeitbare Prozesse erhöhen die CPU Auslastung
- ◆ Nachteile ?
  - Verwaltungsaufwand, zusätzl. Prozeßwechsel, AdressierungWoher resultieren zusätzliche Prozeßwechsel ???

im folgenden betrachten wir im wesentlichen **Paging**

zusätzliche Begriffe

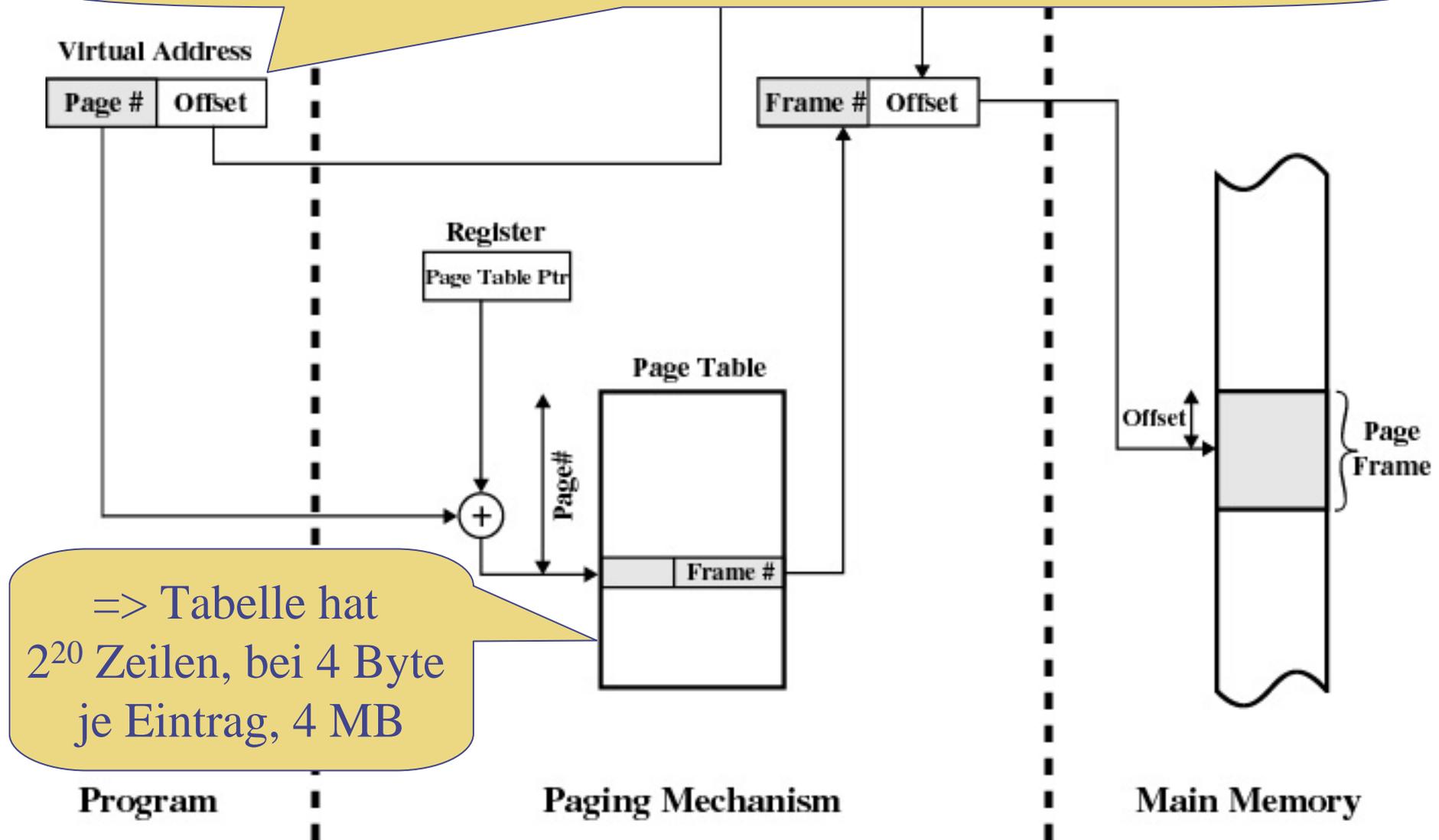
- **resident set**: Menge der Seiten eines Prozesses, die im Hauptspeicher residieren/verfügbar sind
- **working set**: Menge der Seiten eines Prozesses, die über einen Zeitraum zur Bearbeitung benötigt werden

## Verwaltung von Seiten in Seitentabellen, je Eintrag:



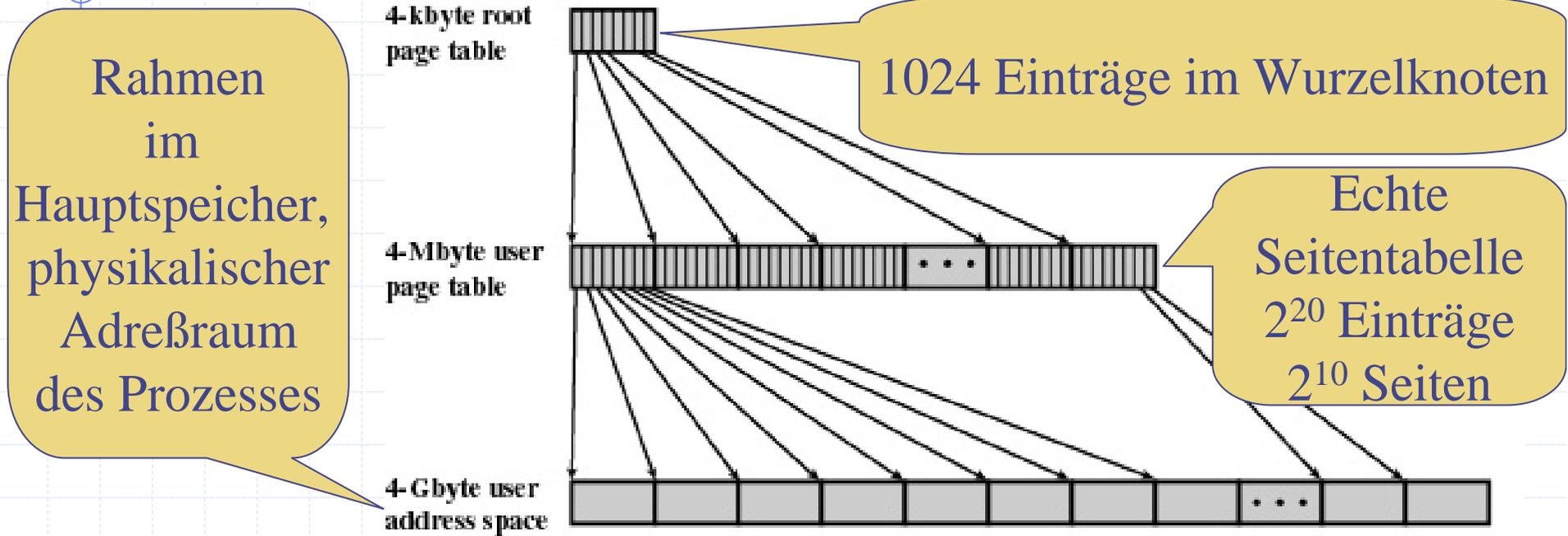
- Jeder Prozeß hat eine eigene Seitentabelle mit Rahmennummern je Seite.
- Kontrollbits signalisieren, ob eine Seite verfügbar ist oder modifiziert wurde.
- Modifizierte Seiten müssen vor dem Überschreiben auf den Plattenspeicher (in den Swap Space) übertragen werden.

Z.B. 32 Bit Adresse, je Byte 1 Adresse, 12 Bit = 4 K Seitengröße



Adreßübersetzung in einem Paging System, 2 Seitenzugriffe je Adresse  
wg Zeitaufwand: Hardwareunterstützung innerhalb der CPU  
wg Platzbedarf, hier z.B. 4 MB je Prozeß: Seitentabelle hierarchisch

## Variante 1: Schema mit 2 Ebenen für 32 Bit Adressen



Idee: Auch die Seitentabelle wird mit Paging verwaltet.

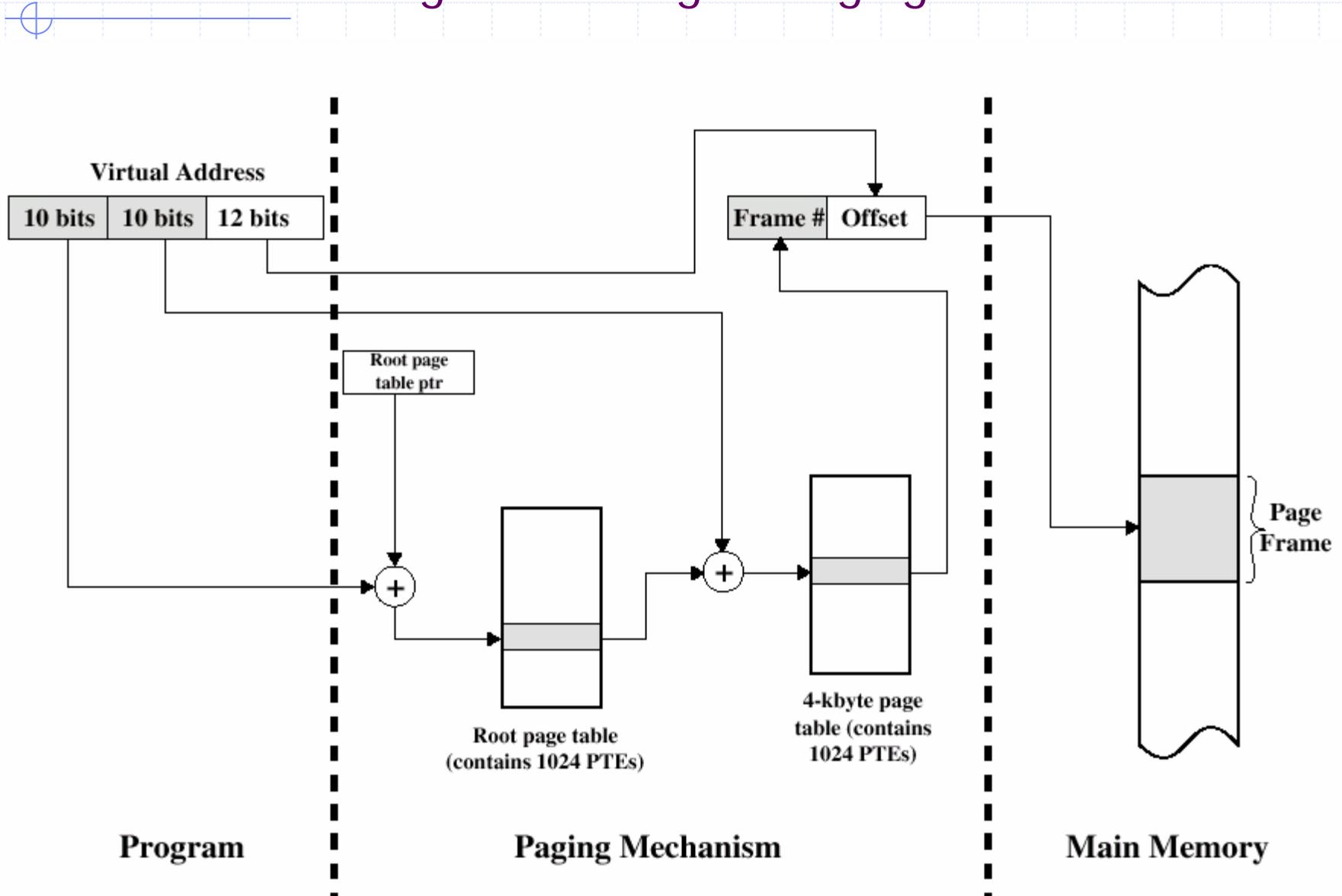
Der Wurzelknoten bleibt stets im Hauptspeicher.

Dies bewirkt 3 Speicherzugriffe je Adreßrechnung,  
bei 64 Bit Adressen sogar 3-4 Ebenen,

Läßt sich damit eine akzeptable Rechengeschwindigkeit erzielen ?

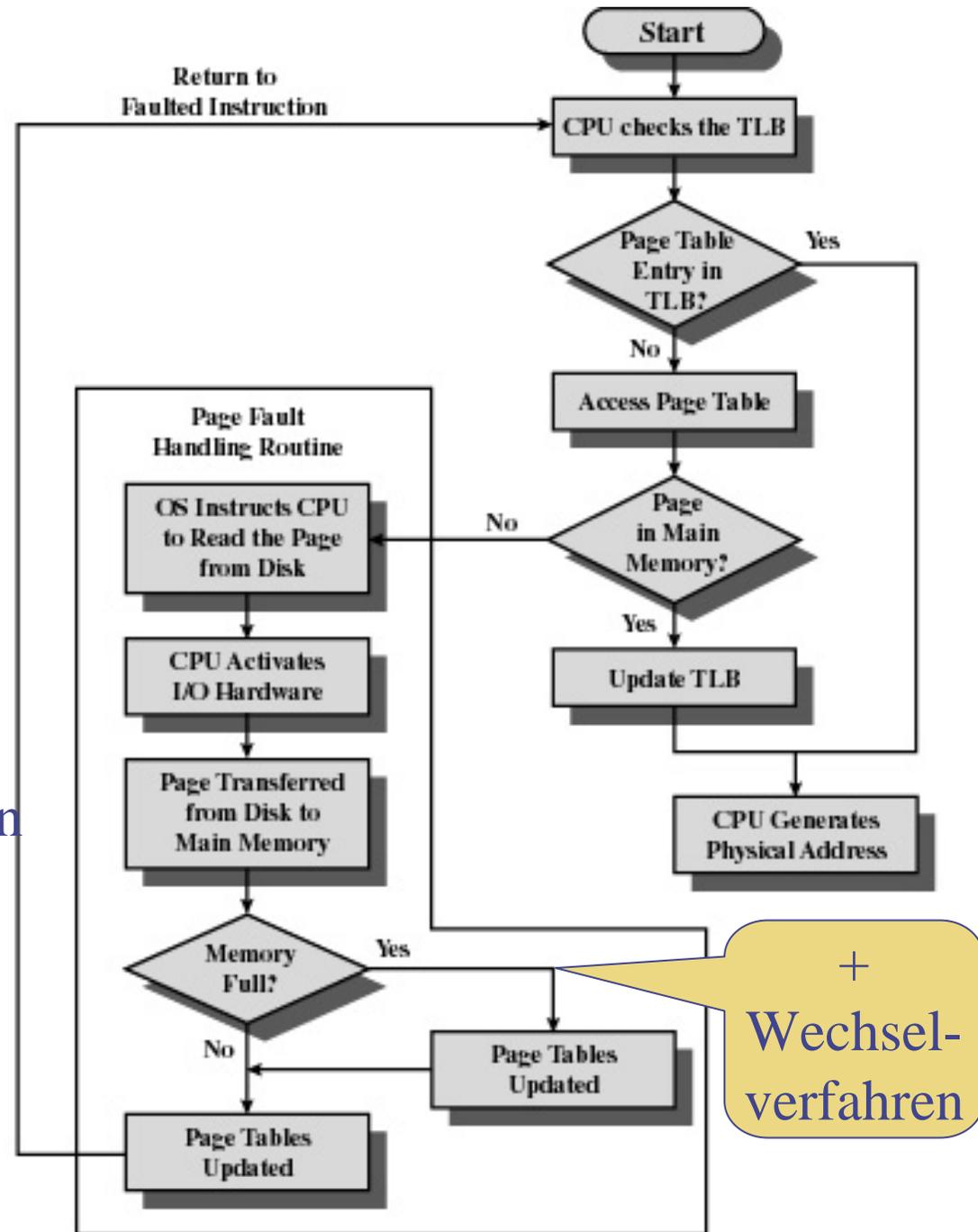
=> Hardwareunterstützung => Cacheing !

# Adreßübersetzung bei 2-stufigem Paging



Cacheing Idee: merke  
Antworten auf häufige  
Anfragen  
Hardware:  
Translation Lookaside Buffer  
(TLB) = Assoziativspeicher  
speichert Zeilen der  
Seitentabelle als Cache.  
Der TLB wird zusätzlich  
abgefragt, ggfs aktualisiert.

Der Ablauf muss noch um den  
Seitenwechselmechanismus  
bei vollem Speicher erweitert  
werden.



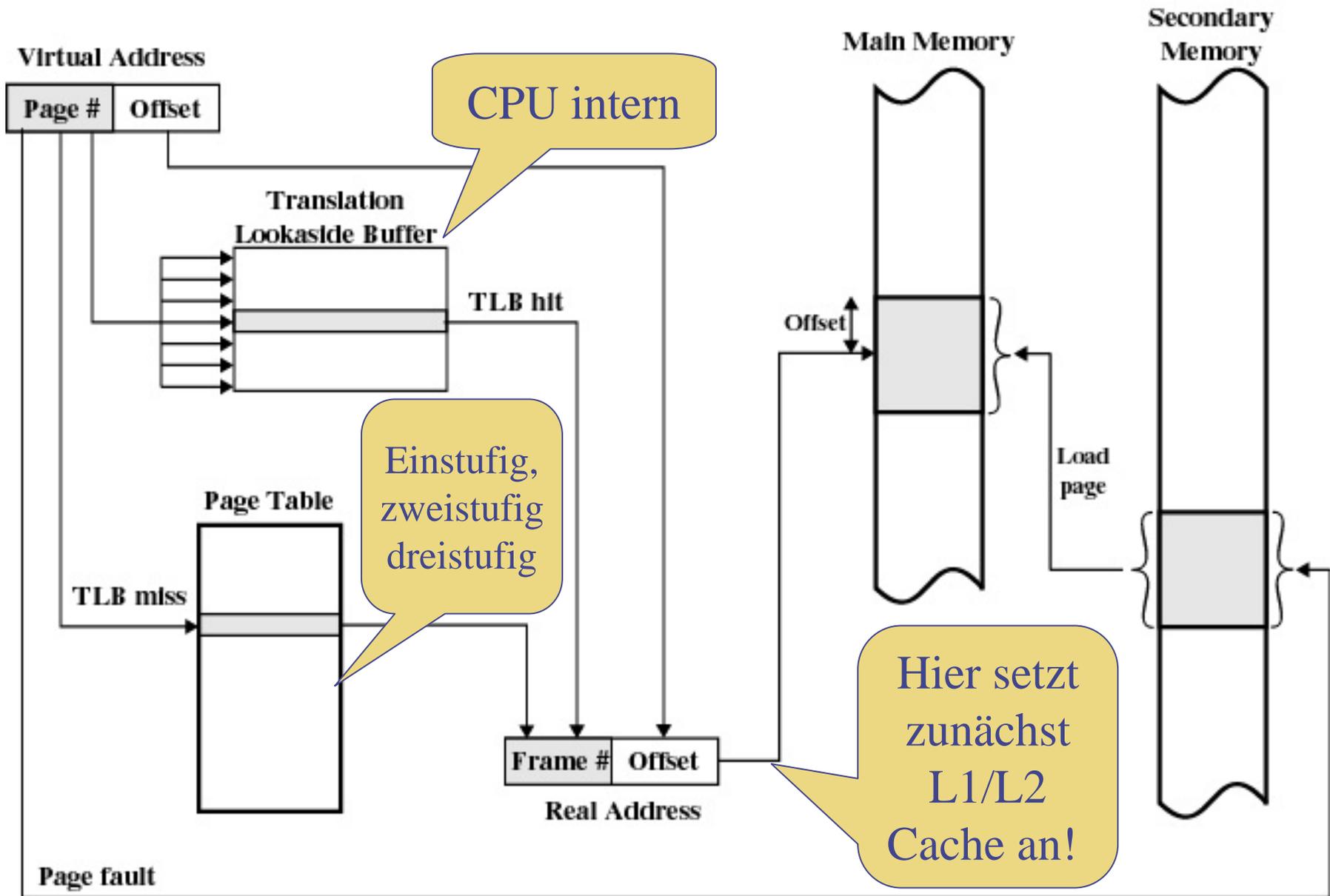
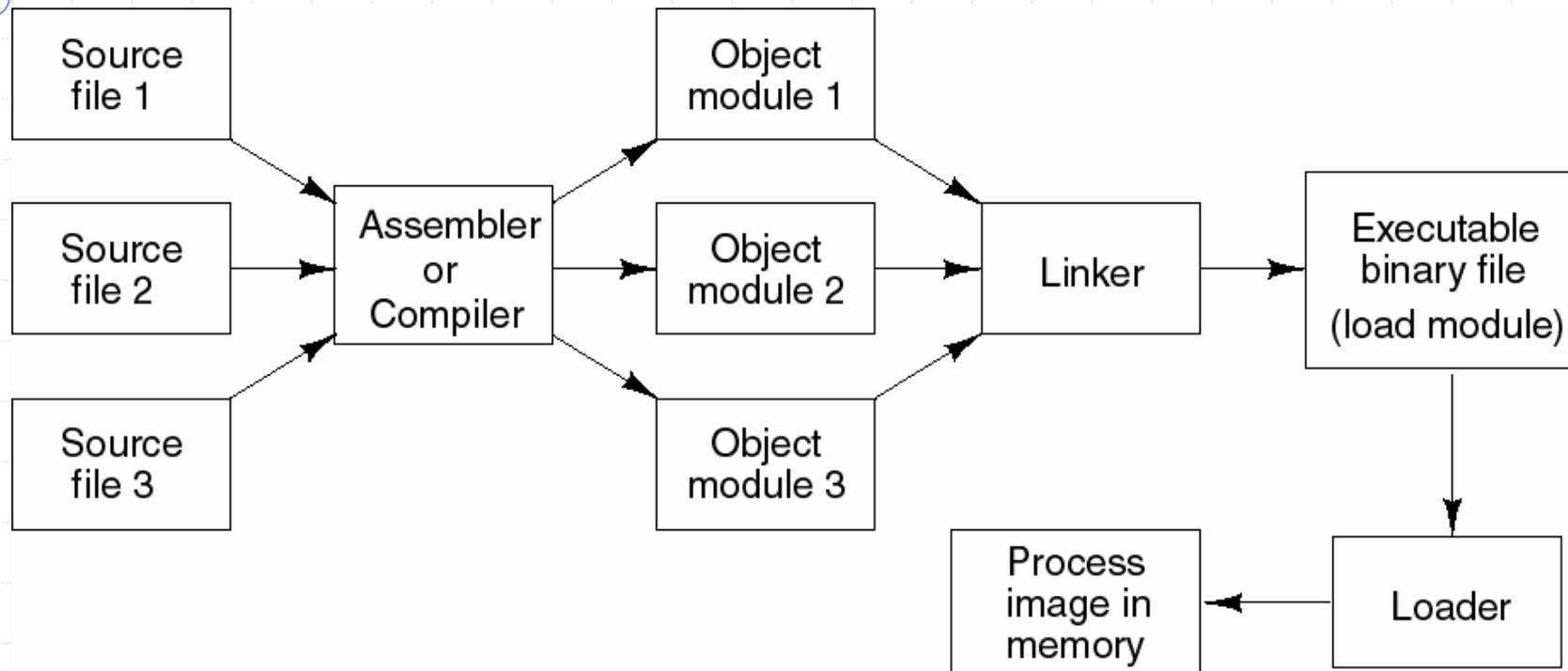


Abb. 8.7: Verwendung des TLBs  
 Achtung: bei Prozeßwechsel muß TLB gelöscht werden!

## Schritte zum Laden eines Prozesses in den Speicher



- ◆ Compiler: übersetzt einzelnes Modul, erzeugt Adressraum für Modul
- ◆ Linker: bindet Module zu Executable, bildet einheitlichen Adressraum  $0, \dots, n$  über alle Module hinweg.
- ◆ Loader: plaziert ein ausführbares Programm im physikalischen Speicher, bindet Libraries.

## Prozeßmanagement -> E/A Management

Das Betriebssystem verdeckt weitgehend die Besonderheiten der jeweiligen Hardwareeinheiten eines Rechners, insbesondere die Vielzahl Ein-/Ausgabegeräte (E/A Geräte, I/O Devices) werden hinter weitgehend homogenen Programmierschnittstellen verborgen.

Für die Realisierung benötigt ein OS für jeden Gerätetyp eine passende Steuerungssoftware (Gerätetreiber, Device driver).

Aufgrund der großen Zeitunterschiede zwischen der Bearbeitung in der CPU und in E/A Geräten erfolgt E/A typischerweise asynchron (z.B. Direct memory access DMA), wodurch zusätzliche Kommunikationsmöglichkeiten erforderlich werden (Interrupts).

Wesentliche Ziele im OS Entwurf sind:

Effizienz & Allgemeingültigkeit, d.h. E/A muss möglichst schnell sein und eine einheitliche Sichtweise muß die Programmierung von E/A Funktionen und die Implementierung E/A Software kennzeichnen.

Typische Konzepte sind: Puffern, Caching

# Übersicht

## ◆ Scheduling für Einprozessorarchitekturen

- Unterscheidung nach Zeitskalen
- Algorithmen für kurzfristiges Scheduling (Dispatching)
  - ◆ FCFS, RR, SPN, SRT, HRRN, FB
- Bsp: traditionelles Unix Scheduling

## ◆ Scheduling für Mehrprozessorarchitekturen

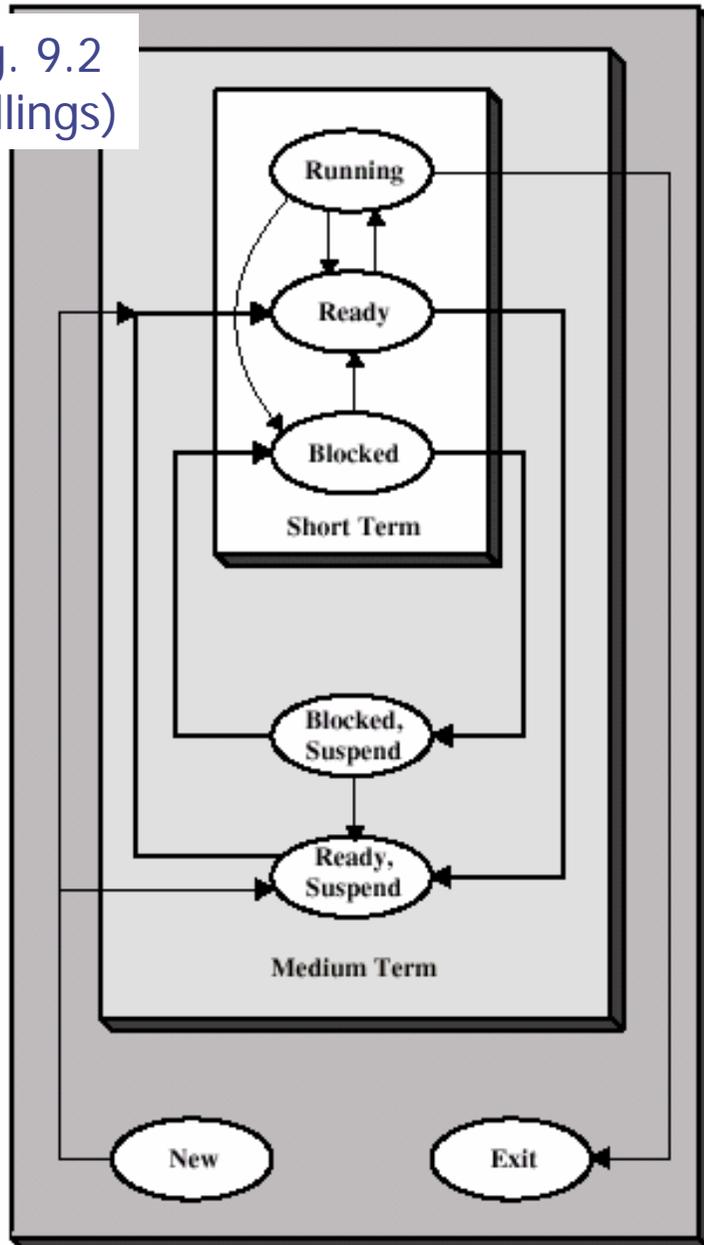
## ◆ Realtime Scheduling

## ◆ Bsp

- Linux, Unix SVR4, Windows 2000

## CPU Scheduling: Aufteilung der Zustände nach Scheduling

(Fig. 9.2  
Stallings)



Der langfristige Scheduler steuert den Grad des Multiprogramming. Bei Batch Processing ist die Menge anstehender Aufgaben vorab bekannt, bzw. neue Jobs werden zunächst auf der Festplatte gelagert ohne ein Prozeßimage zu erzeugen. Er wird aktiv, wenn neue Jobs eintreffen, ein bestehender terminiert oder die CPU einige Zeit idle ist.

Für eine gleichmäßige Ressourcennutzung kann eine Mischung aus E/A-lastigen und CPU-lastigen Prozessen erwünscht sein.

Der mittelfristige Scheduler ist Teil der Swap Funktion und reguliert ebenfalls den Grad des Multiprogramming.

Der kurzfristige Scheduler (Dispatcher) wird häufig aktiviert: bei Clock, E/A Interrupts, Systemaufrufen und Signalen.

# Scheduling Algorithmen für kurzfristiges Scheduling

## Kriterien

### anwenderorientiert

- Turnaround Time, Umlaufzeit (für Batch Jobs)
- Antwortzeiten bei interaktiven Prozessen, z.B.  $< 2$  sec
- Deadlines/Endtermine
- Vorhersagbarkeit

### systemorientiert (bei Mehrbenutzerbetrieb interessant)

- Durchsatz
- CPU Auslastung
- Fairness
- Berücksichtigung von Prioritäten
- Ausgeglichene Nutzung von Ressourcen

## Möglichkeiten

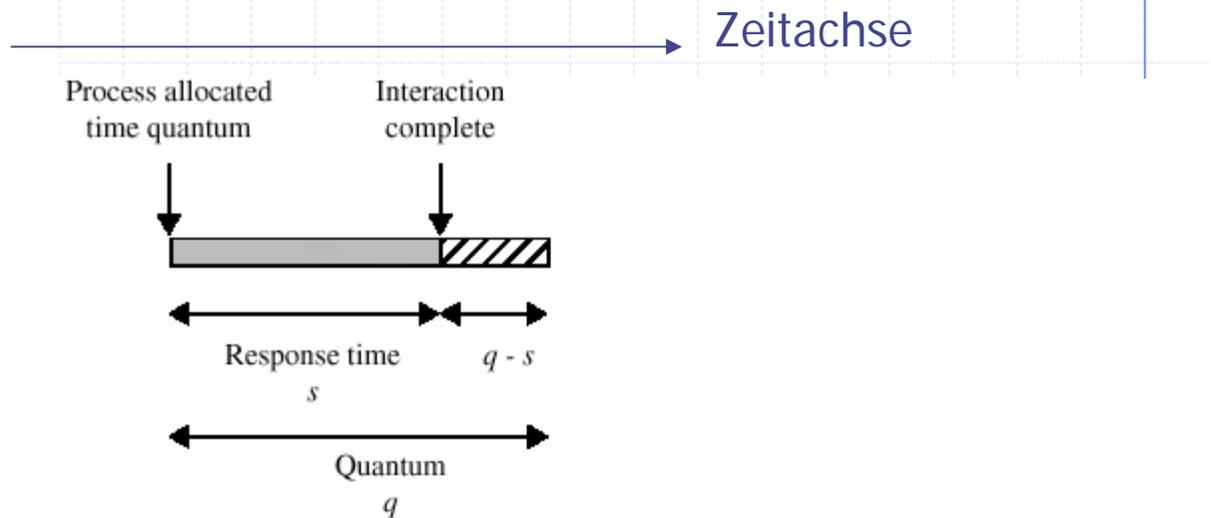
- Prioritäten
- Unterbrechungen

# Scheduling Algorithmen für kurzfristiges Scheduling

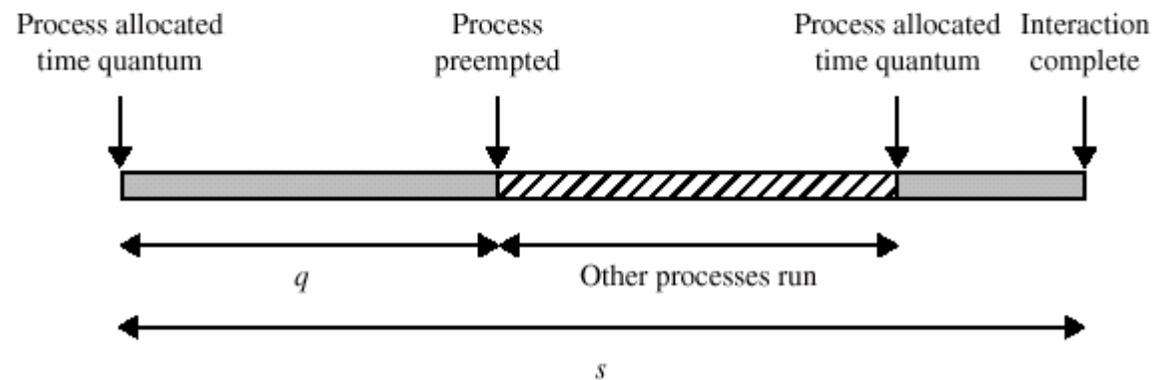
- ◆ FCFS: First-come-first-served
  - der Prozeß mit längster Wartezeit in der Ready Queue wird solange ausgeführt, bis er blockiert
- ◆ RR: Round robin
  - Zeitscheibenverfahren, Prozesse werden reihum jeweils für eine festgelegte Zeitspanne (Zeitquantum) bearbeitet
- ◆ SPN: Shortest process next
  - der Prozeß mit der kürzesten zu erwartenden Bearbeitungszeit wird ohne Unterbrechung abgearbeitet
- ◆ SRN: Shortest remaining time
  - unterbrechende Version von SPN, wählt Prozeß mit der kürzesten zu erwartenden Restbearbeitungszeit
- ◆ HRRN: Highest response ratio next
  - wählt den Prozeß mit größtem Wert für  $(w+s)/s$ , wobei  $w$ =Wartezeit,  $s$ =erwartete Bedienzeit ist
- ◆ FB: Feedback
  - je länger Prozesse laufen, desto geringer wird ihre Priorität. FB wählt innerhalb von Prioritätsklassen nach FCFS aus, auf unterster Stufe dann RR, um Verhungern auszuschliessen.

# Round Robin Besonderheiten

- ◆ Prozesse werden spätestens nach  $K$  Zeiteinheiten unterbrochen, aus der Ready Queue wird ein Nachfolger gemäß FCFS ausgewählt: Zeitscheibenverfahren
- ◆ bewirkt kurze Antwortzeiten für Kurzläufer
- ◆ erfordert passende Wahl des Zeitquantums, Wert sollte nah an typischer Interaktionszeit liegen
- ◆ CPU-lastige Prozesse werden bevorzugt
- ◆ Fairness wird ggfs durch dynamische Prioritäten ausgeglichen



- a) Zeitquantum  $>$  Interaktionszeit
- b) Zeitquantum  $<$  Interaktionszeit



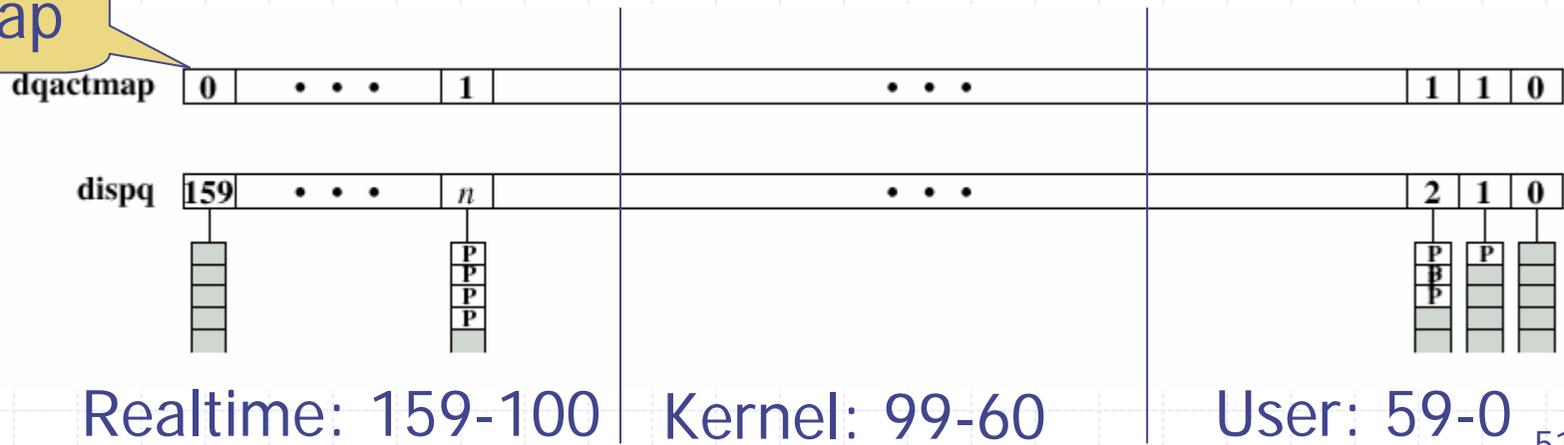
## Beispiele für Scheduling: LINUX

- ◆ basiert auf traditionellem UNIX Scheduling mit 3 Klassen wg Echtzeitanforderungen
  - SCHED\_FIFO: Realtime Threads mit FIFO Scheduling und Prioritäten, Unterbrechungen nur bei höherpriorigen Threads, Blockierung oder freiwilliger Aufgabe.
  - SCHED\_RR: Realtime Threads mit RR Scheduling und Prioritäten wie FIFO nur mit Zeitscheiben und entsprechenden Interrupts.
  - SCHED\_OTHER: alle anderen Prozesse/Threads werden mit Multilevel Feedback und RR behandelt. Prozesse haben eine Basispriorität und können vom Anwender zusätzlich noch mit nice innerhalb des zulässigen Prioritätsbereichs herabgestuft werden. Natürlich werden Betriebssystemtasks bevorzugt behandelt.

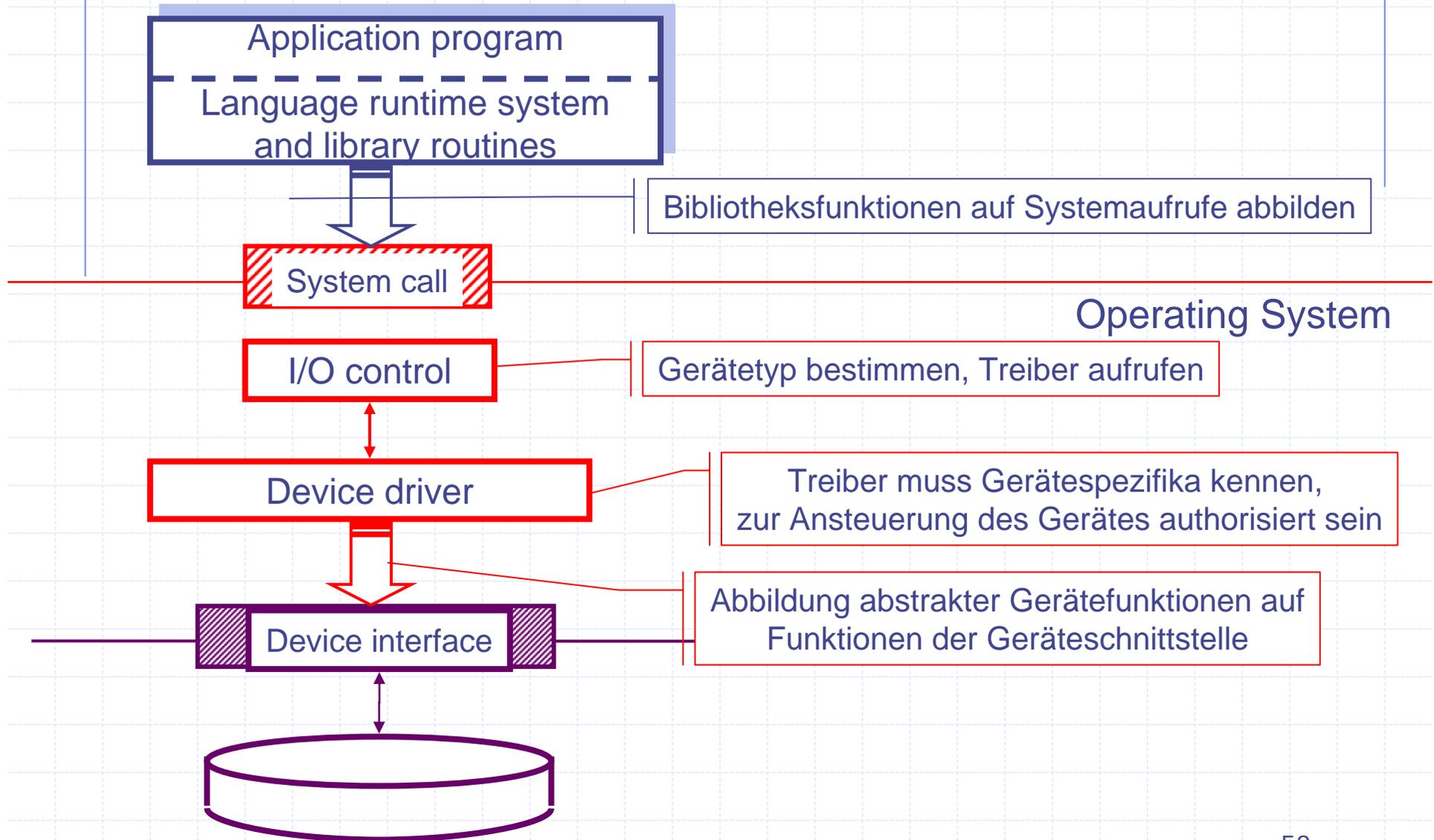
# Beispiele für Scheduling: UNIX SV R4

- ◆ Rangordnung:
  - Realtime Prozesse vor Kernel- vor Usermode Prozessen behandelt
- ◆ Statischer Prioritätsscheduler mit Unterbrechungen und 160 Prioritätsstufen, in 3 Gruppen unterteilt
- ◆ Preemption Points, um Kernelunterbrechungen zu ermöglichen, (safe places in Kernelroutinen)
- ◆ RR für jede Prioritätsstufe
  - im Bereich der Userprozesse sind Prioritäten dynamisch um CPU lastige Prozesse nicht zu bevorzugen, Zeitscheiben für niederpriorige Prozesse werden darüberhinaus größer 10ms -> 100ms
  - im Realtime Bereich sind Prioritäten und Zeitscheiben fest

Bitmap



# Typische Schichten eines I/O-Subsystems



## Festplatten Scheduling

- ◆ Festplattenzugriffe sind etwa 4 Größenordnungen langsamer als Hauptspeicherzugriffe und der Abstand wird mit der technischen Entwicklung größer!
- ◆ Wegen des virtuellen Speichers und der dauerhaften Speicherung von Datenmengen auf der Festplatte ist die Performance von Festplatten als E/A Gerät besonders wichtig.
- ◆ Parameter für die Performance eines Plattenzugriffs
  - Suchzeit, 5-10ms: Zeit zur Positionierung des Schreib/Lesekopfes auf dem richtigen Track, Zeiten für Beschleunigung+ Bewegung+Justieren,
  - Rotationsbedingte Latenzzeit, 3ms bei 10000 rpm: Zeit bis passender Sektor den Schreib/Lesekopf passiert; etwa 1/2 Umdrehung
  - Zugriffszeit T: abhängig von der Rotationsgeschwindigkeit

$$T = b / rN \text{ für } b \text{ Bytes}$$

bei N Bytes je Track, r Umdrehungen/s

$$\text{Insgesamt: } T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

## Scheduling Algorithmen für Festplatten

- ◆ Random Scheduling als Benchmark liefert schlechteste Performance wegen häufiger Suchzeiten
- ◆ FCFS ist fair, nutzt jedoch nur sequentielles Auftreten, falls dies zufällig auftritt, bei vielen Prozessen wie Random
- ◆ Prioritätsscheduling, liegt außerhalb der Gerätesteuerung, könnte z.B. kurze interaktive Jobs bevorzugen
- ◆ Last-in-first-out, gut bzgl. Lokalität, jedoch Risiko für Starvation bei hoher Last

Weitere Ansätze wählen gezielt aus der Menge von wartenden Anfragen anhand der Position und Datenmenge aus

- ◆ Shortest-service-time-first wählt Anfrage mit geringster Suchzeit (Entfernung) aus, typische Greedy-Strategie, die häufig gut funktioniert, jedoch globales Optimum nicht garantieren kann. Achtung: Starvation möglich!

## Disk Scheduling Algorithmen II

- ◆ SCAN, maximiert Weg in einer Richtung
    - verfolgt die aktuelle Bewegungsrichtung bis zum Ende, nimmt alle Anfragen auf dem Weg soweit möglich mit, dann rückwärts.
    - verhält sich ähnlich zu SSTF jedoch kein Verhungern,
    - nutzt jedoch Lokalität weniger als SSTF und LIFO,
    - bevorzugt Tracks am Rand und Jobs, die zuletzt eintreffen, falls sie auf den aktuellen Track zugreifen.
  - ◆ C-SCAN (circular SCAN)
    - verfolgt 1 Richtung bis zum Rand, dann „Rücksprung“ zum Anfang
- Problem bei STTF, SCAN und C-SCAN: bei vielen Zugriffen auf 1 Track müssen entfernte Zugriffe lange warten („arm stickiness“)
- ◆ N-step-SCAN
    - Warteschlange in Segmente der Länge N unterteilt
    - je Segment von N Jobs in der Schlange wird SCAN angewendet
    - bei  $N=1$  wie FIFO, bei großen Werten für N wie SCAN
  - ◆ FSCAN
    - 2 Warteschlangen, von denen eine abgearbeitet wird und in dieser Zeit werden neue Anfragen in die andere Schlange verwiesen

## Dateimanagement

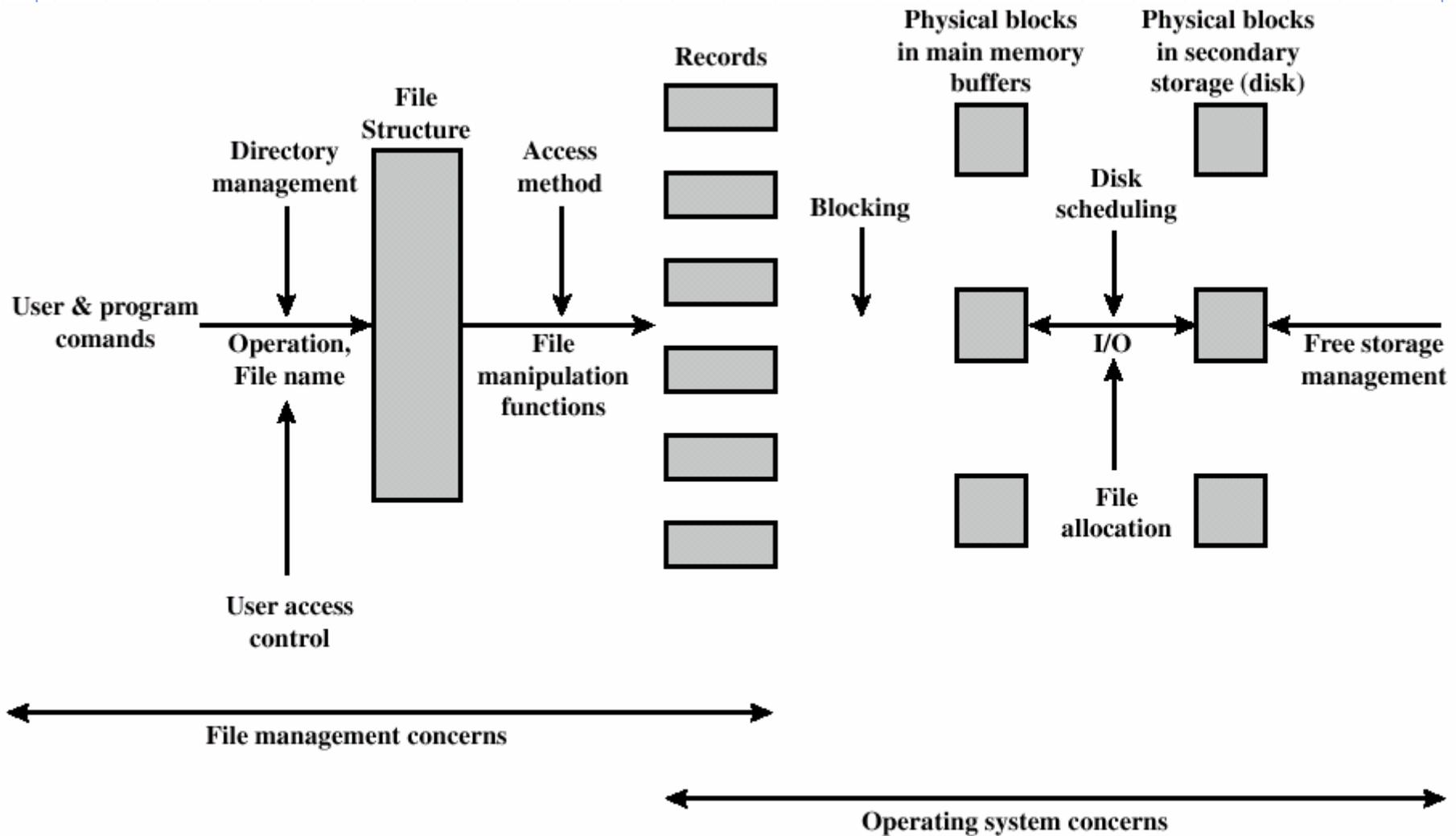
Dateien bieten die Möglichkeit, die Resultate eines Programms über das Ende des Prozesses hinaus dauerhaft zu speichern.

Das Dateimanagement hat die Aufgabe, für das logische Konzept der Datei, das durch ein OS angeboten wird, zu realisieren.

Typische Verwaltungsstrukturen sind baumartige Directories, deren Wege von der Wurzel durch verschiedene Unterdirectories bis zur konkreten Datei einen eindeutigen Dateinamen ergeben, zu dem das OS dann die zugehörigen Inhalte speichern muß.

Anforderungen sind: Effizienz (bzgl Zeit und Platz), Zuverlässigkeit.

# Dateimangement, einige Aspekte



## Verteilte Systeme, Netzwerke

Unterstützung von Client/Server Architekturen durch ein Betriebssystem:

- Message passing, zwischen Prozessen werden Nachrichten übermittelt
- Remote Procedure Calls, Prozesse greifen auf Funktionen zu, die nicht vom lokalen Rechner durchgeführt werden, sondern an einen externen Server weitergeleitet werden und dort ausgeführt werden. Die Ergebnisse dieser Berechnungen müssen natürlich übermittelt werden.

Verteilte Betriebssysteme gehen noch einen Schritt weiter, sie werden auf einem Netzwerk realisiert und wirken wie eine lokale „virtuelle“ Maschine. Sie erlauben die Migration von Prozessen.

# Verteiltes Prozessmanagement

## ◆ Prozeßmigration

- Mechanismen, Verhandlung, Abschiebung

## ◆ Verteilter globaler Zustand

- verteilter Schnappschuß und Konsistenz
- Algorithmus als Grundlage für viele Verteilte Algorithmen

## ◆ Verteilter wechselseitiger Ausschluß

- Wichtiger Basisalgorithmus: Zeitstempelfverfahren (mit Zählern)  
=> konsistente Reihenfolge von Ereignissen über alle Knoten
- verteilte Warteschlange

## ◆ Verteilter Deadlock

- bei der Ressourcenbelegung
- bei der Nachrichtenübermittlung

# Zusammenfassung

- ◆ Einführung: Ein Betriebssystem - Was ist das ?
  - Software, Ablauf von Programmen, virtuelle Maschine, kapselt Hardware
  - Architektur: monolithisch, monolithischer Kernel, Schichten, Mikrokern
- ◆ Prozeßmanagement
  - Prozesse, Threads, User-Level Threads, Kernel-Level Threads, Java
  - Wechselseitiger Ausschluß, Semaphore, Monitor
  - Deadlocks (4 Bedingungen), Starvation
- ◆ Speicher Management
  - einfache Speicherverwaltung, logische vs physikalische Adressen
  - virtueller Speicher, Paging, Adressrechnung, Seitentabellen, TLB
- ◆ Prozessor Scheduling
  - Time-sharing vs Realtime, Scheduling bei 1 CPU, mehreren CPUs
  - Scheduling Algorithmen: FCFS, RR, SPN, SRN, HRRN, FB
- ◆ E/A Management, Festplattenscheduling
  - Schichtenmodell, SCAN Verfahren für Platten, RAID Systeme
- ◆ Datei Management
  - Dateiarten, Records, Blöcke, Unix: Inode
- ◆ Netzwerke und Algorithmen im verteilten Prozessmanagement