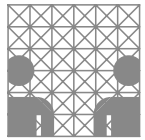


64-040 Modul InfB-RSB
Rechnerstrukturen und Betriebssysteme

https://tams.informatik.uni-hamburg.de/lectures/2019ws/vorlesung/rsb

– Kapitel 15 –

Andreas Mäder

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2019/2020

Kapitel 15

Betriebssysteme

Betriebssysteme in RSB

- genug Stoff für eigene Vorlesungen
- Themen
  - Prozesse und Threads
  - Synchronisation und Kommunikation; Deadlocks
  - Scheduling
  - Speicherverwaltung; *Virtual Memory*
  - Dateiverwaltung und I/O
- nicht behandelt
  - Praxisbeispiele: Windows, Unix, Linux, Android ...
  - Dateisysteme
  - Virtualisierung; Container

  → *VSS (Verteilte Systeme und Systemsicherheit)*
  Sicherheit, RAID

  → *ES (Eingebettete Systeme)*
  Eingebettete Betriebssysteme, Echtzeitverhalten

→ Grafiken, wenn nicht anders angegeben, aus: W. Stallings: *Operating Systems – Internals and Design Principles* [Sta17]

Betriebssysteme

Was sind Betriebssysteme?

Im Prinzip Software, wie jedes andere Programm auch!

Was machen Betriebssysteme?

- Verwalten der „teuren“ Hardware für optimale Nutzung
  - Prozessor(en)
  - Systembus(se)
  - Hauptspeicher
  - Festplatten / SSDs
  - Ein-/Ausgabeeinheiten (I/O)

  $\Rightarrow$ Anpassen der Geschwindigkeiten
- Koordination aller Programme, Dienste und Benutzer
  Wer darf wann worauf zugreifen?
- Bereitstellen von Systemdiensten („*Service*“) und Schnittstellen („*System-Call*“) für (andere) Programme, bzw. die Benutzer
  Wie ist der Zugriff geregelt?

$\Rightarrow$ BS sind meist die komplexeste Software auf dem Computer!

1. erste Computer / serielle Verarbeitung

- kein Betriebssystem
- Programmierer arbeitet direkt mit Computer an Konsole
- Benutzer können nur nacheinander am Computer arbeiten

– Reservierung des Systems

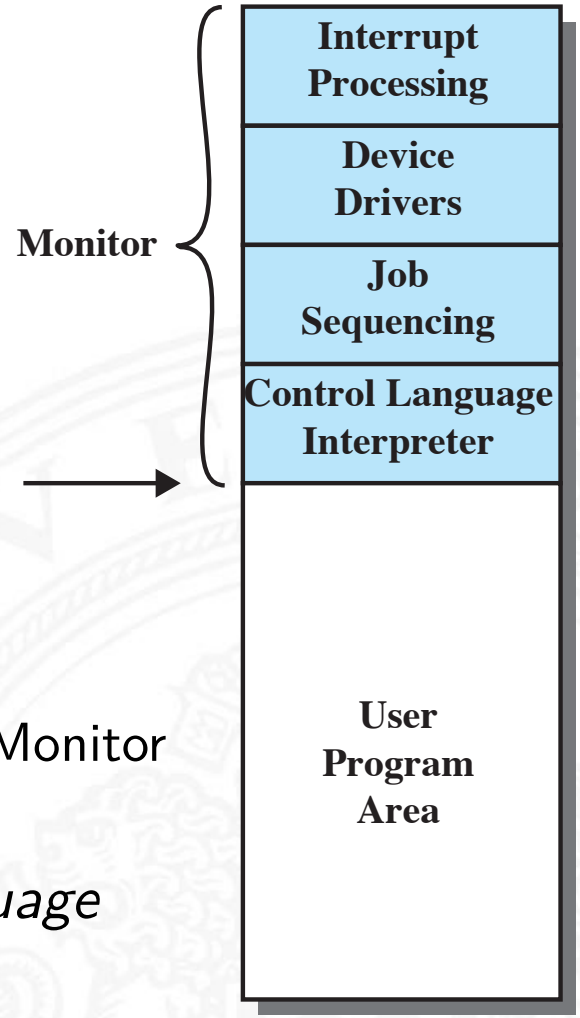
  längerer Job: wird nicht fertig oder
  Reservierungen verschieben sich

  kürzerer Job: System bleibt ungenutzt

– „Rüstzeit“: Vorbereitung auf Programmlauf

2. einfache Batch-Systeme

- Benutzer hat keinen direkten Zugriff
- Operator bündelt Jobs als „*Batch*“
- *Monitor* als zentrales Programm arbeitet Job-Queue ab
- immer im Speicher (*Resident Monitor*)
- Funktionsweise
  - Monitor liest Job ein
  - übergibt Kontrolle an Benutzerprogramm ($\widehat{=}$ Prozeduraufruf)
  - Programm übergibt nach Ende Kontrolle an Monitor ($\widehat{=}$ Rücksprung)
- Instruktionen für Monitor: *Job Control Language*



2. einfache Batch-Systeme (cont.)

- wichtige Eigenschaften
  - **Memory protection**: Jobs haben keinen Zugriff auf Monitor-Speicherbereich
  - **Timer** begrenzt Laufzeit von Jobs
  - **privilegierte Instruktionen** nur durch Monitor ausführbar
  - **Interrupts** bessere, flexiblere Kontrolle der Jobs

$\Rightarrow$ zwei Modi

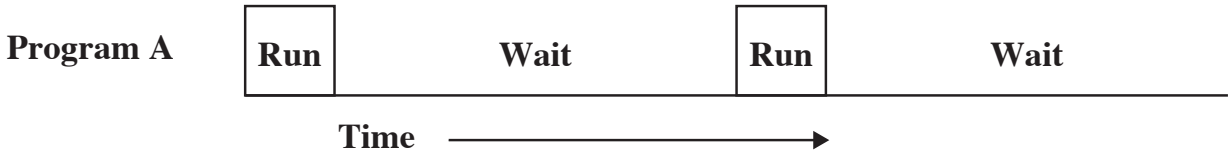
1. **User Mode** für Batch-Job
   - einige Speicherbereiche sind gesperrt
   - einige Befehle sind nicht ausführbar
2. **Kernel Mode** für Monitor
   - Zugriff auf geschützte Speicheradressen
   - privilegierte Befehle sind ausführbar

- Overhead, verglichen mit serieller Abarbeitung
  - $-$ Prozessor muss zusätzlich Monitor bearbeiten
  - $-$ zusätzlicher Speicherbedarf für Monitor
  - $+$ insgesamt aber bessere Auslastung des Computers

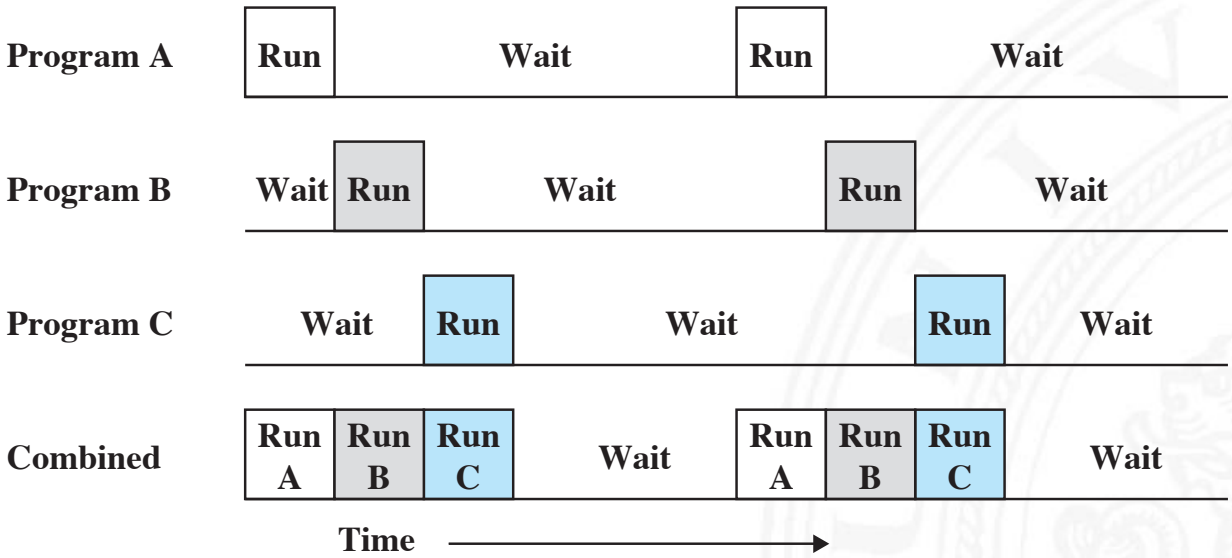
3. Multiprogramm Batch-Systeme

unterschiedliche Geschwindigkeiten ⇒ Prozessor wartet meist

- Uniprogramming



- Multiprogramming, Multitasking



\+ Job wartet auf I/O ⇒ Monitor wechselt zu anderem Job

– Speicherbedarf für Monitor und alle Jobs

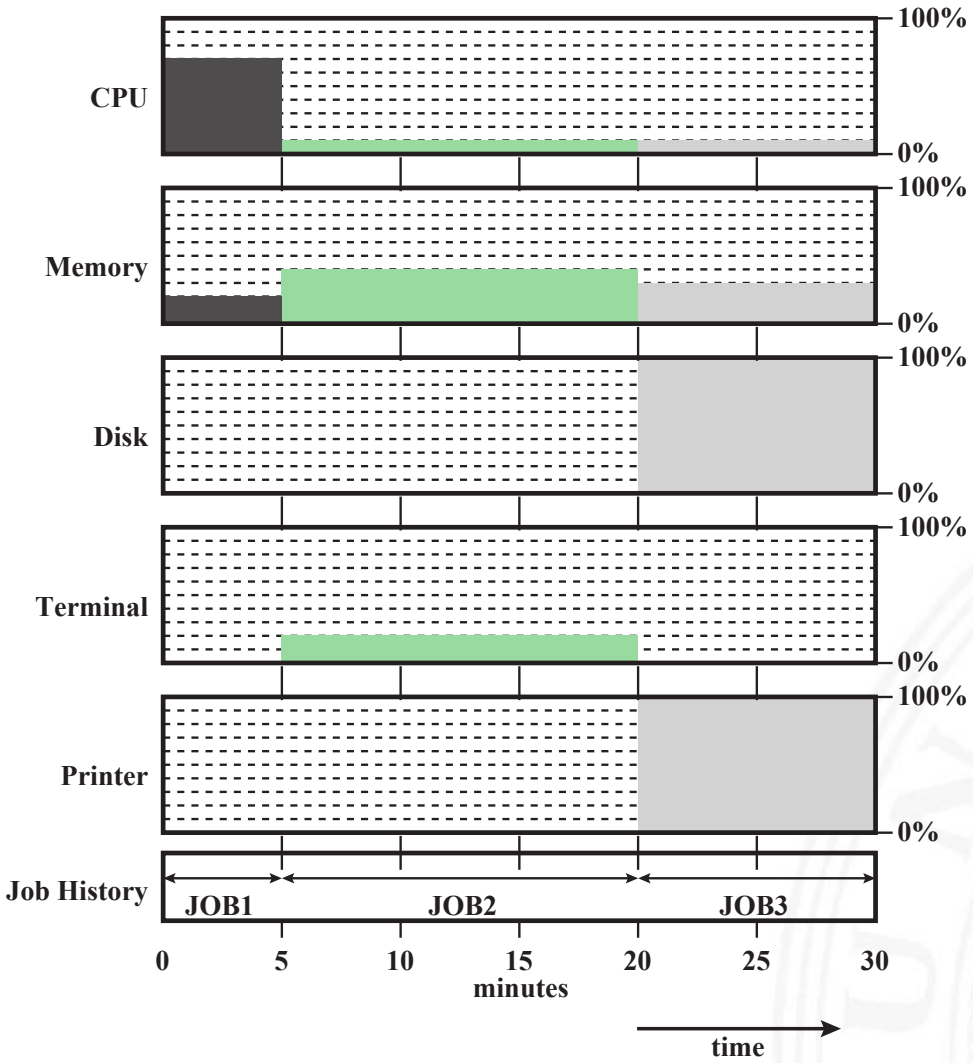
3. Multiprogramm Batch-Systeme (cont.)

- Beispiel

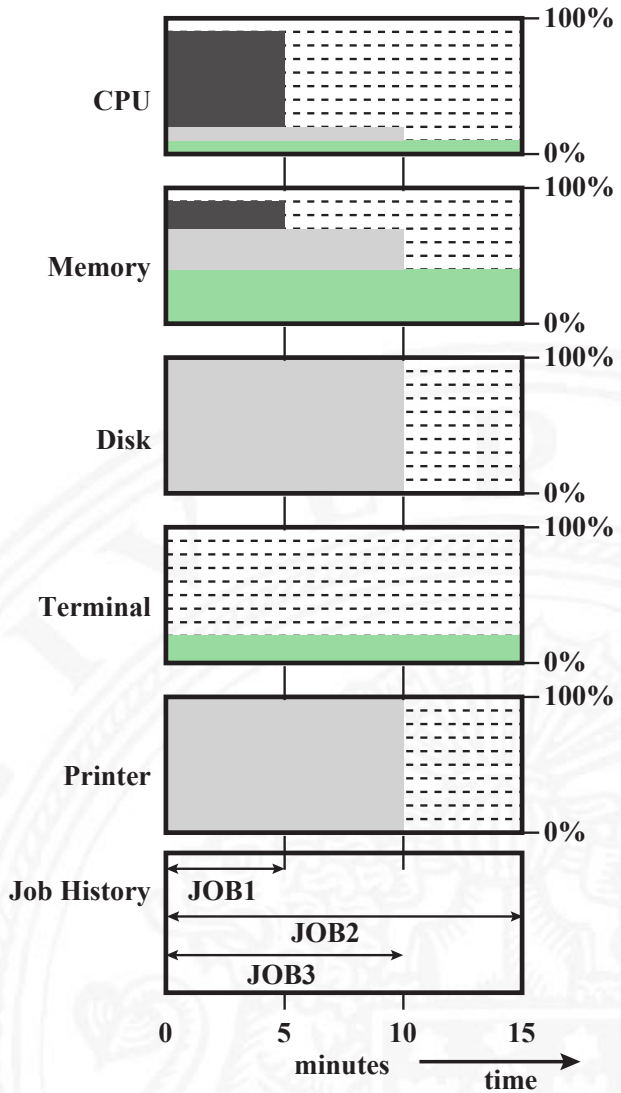
	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

3. Multiprogramm Batch-Systeme (cont.)



(a) Uniprogramming



(b) Multiprogramming

4. *Time-Sharing* Betrieb

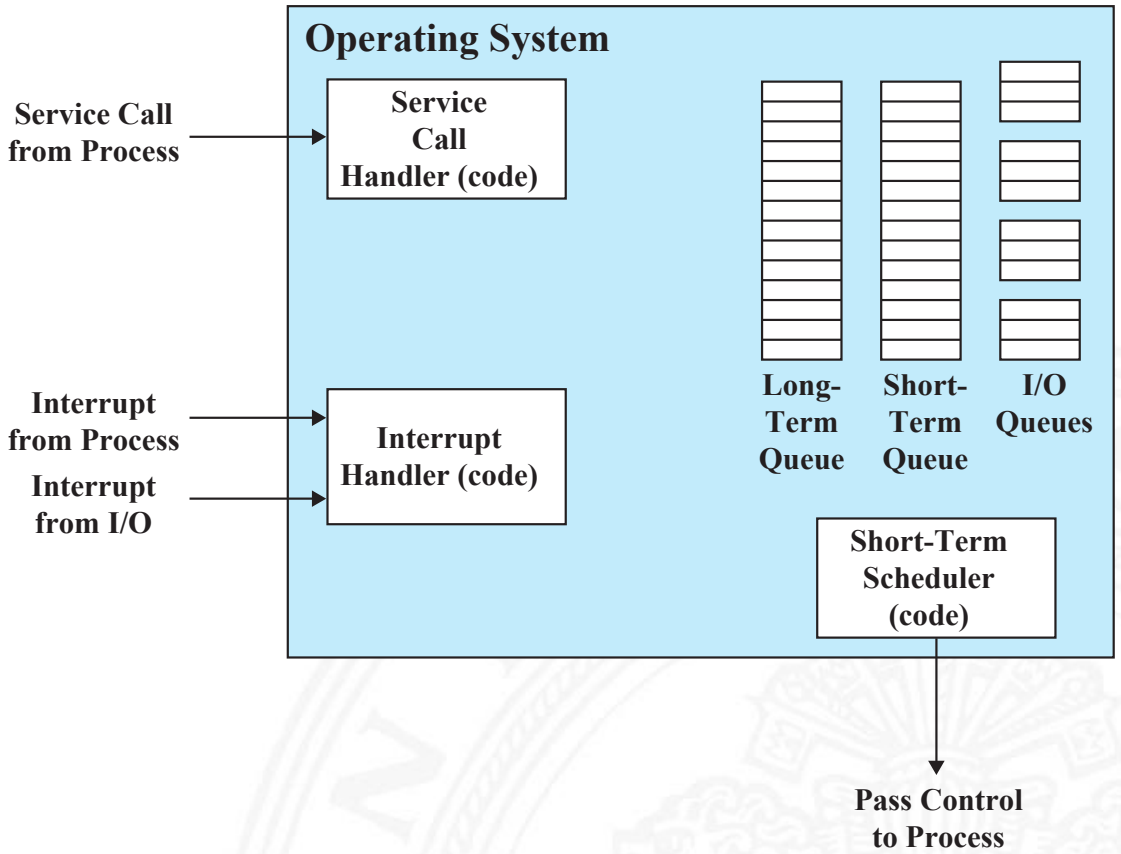
- Erweiterung von Multitasking für interaktive Jobs
- Prozessor/Ressourcen werden zwischen Benutzern geteilt
- Zugriff über Terminals (Kommandozeile)
  später grafische Oberflächen
- Optimierungsziel

	Batch Multiprogramm	Time-Sharing
Optimierung	maximale Prozessornutzung	minimale Antwortzeit
BS Kontrolle	Job Control Language	Benutzereingabe

- typisch: Zeitscheiben Verfahren (*Time Slicing*)
  - periodische Interrupts durch Systemclock
  - Betriebssystem übernimmt Kontrolle
  - prüft ob anderer Prozess laufen soll
  - Benutzerprozess ist „unterbrochen“ („*Preemption*“)
  - sein Status wird gesichert
  - Daten für neuen / fortzusetzenden Prozess werden geladen

aktuelle Betriebssysteme

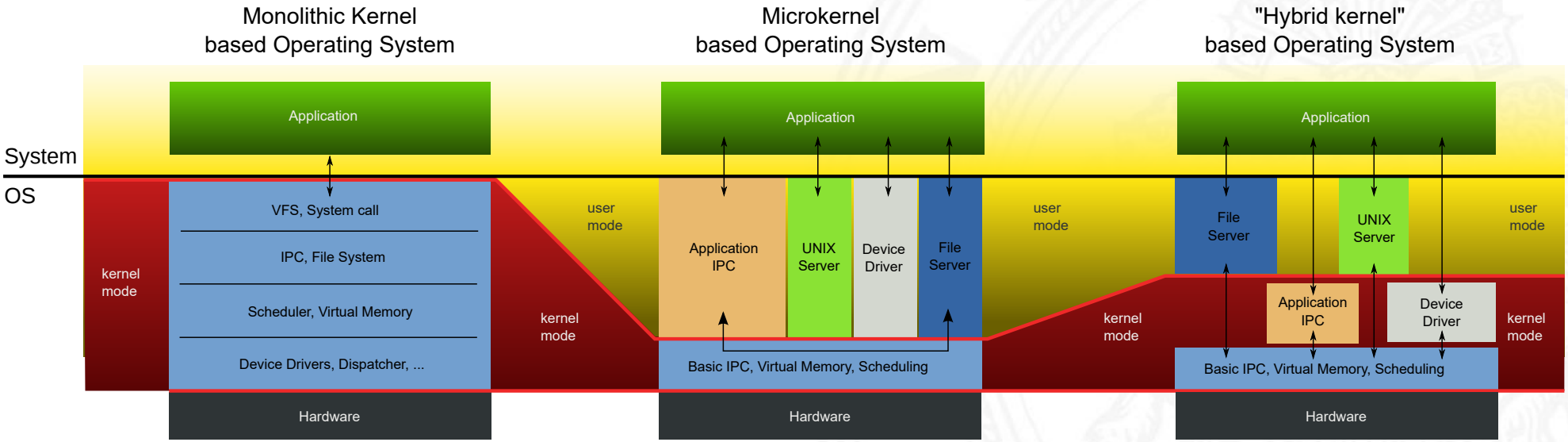
- zentrale Elemente



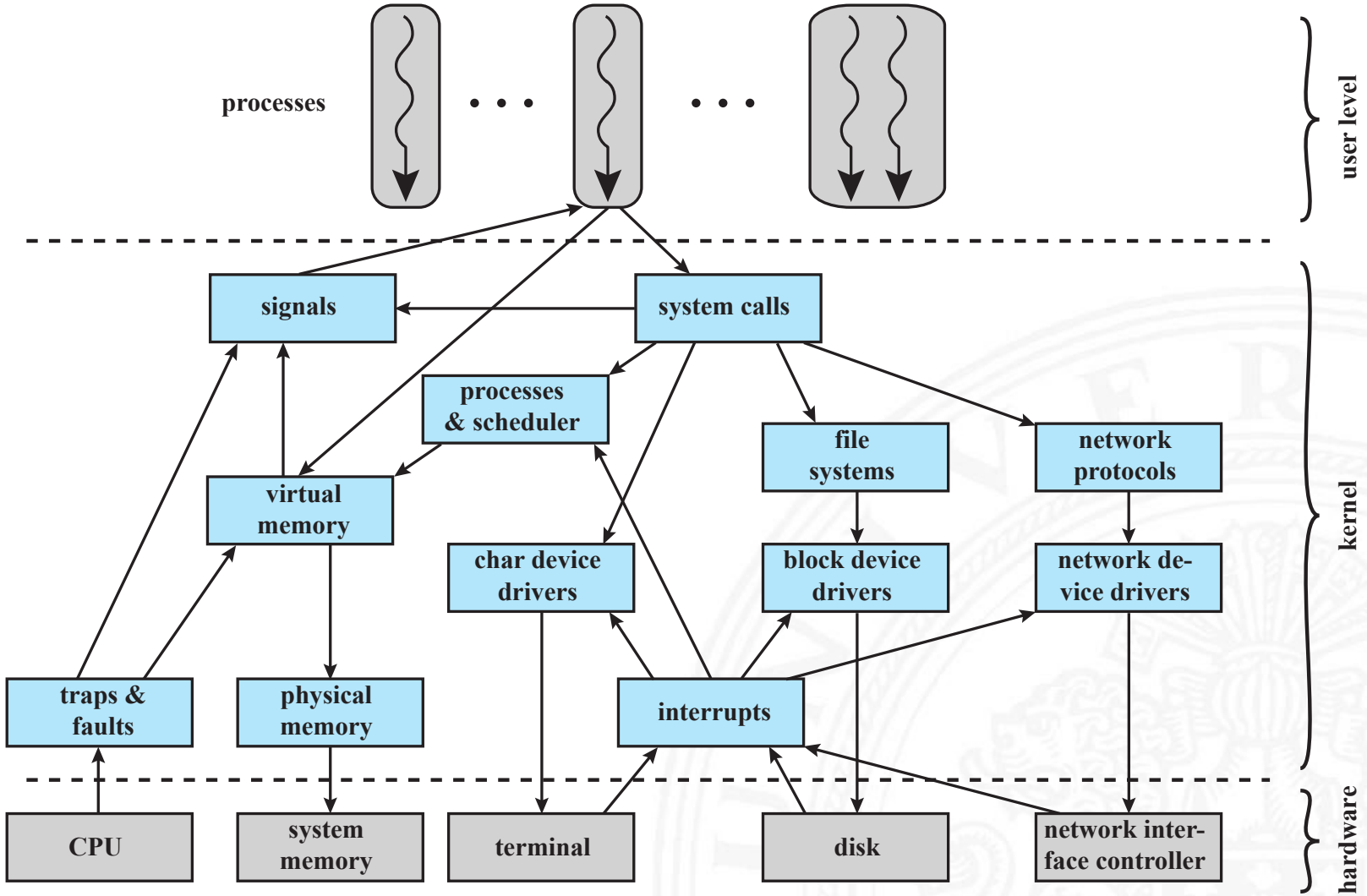
- **Service Call / System Call**:
  Programm ruft BS-Funktion auf $\Rightarrow$ Ressourcenzugriff
- **Interrupt**: besonderes Ereignis
- **FIFO Queues**: Warteschlangen
- **Scheduler**: CPU / Kontrolle wird an Prozess übergeben

aktuelle Betriebssysteme (cont.)

- Architekturansätze
  - **Monolithischer Kernel** alle Funktionalitäten, Treiber etc. in gemeinsamen Kernel; das Programm „Betriebssystem“
  - **Microkernel** enthält nur
    - Scheduling
    - Interprozess-Kommunikation
    - Adressverwaltung
    - restliche Funktionalität als getrennte Prozesse
  - **hybride Kernel** Mischformen



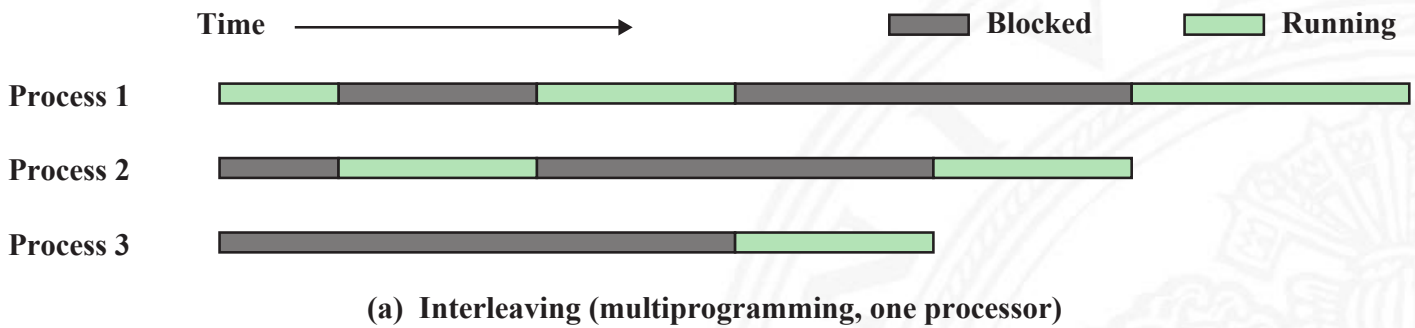
aktuelle Betriebssysteme (cont.)



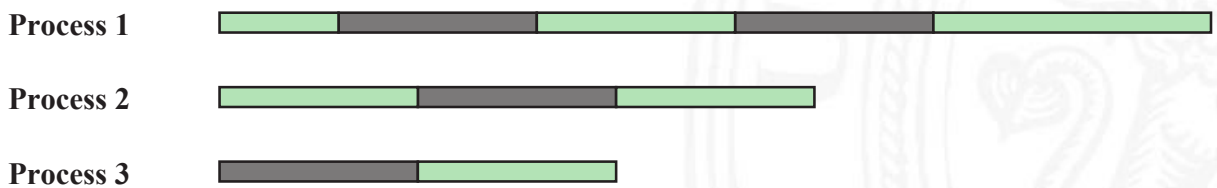
Monolithischer Kernel (Linux): Teilkomponenten

aktuelle Betriebssysteme (cont.)

- weitere Konzepte
  - Multithreading $\Rightarrow$ bessere Granularität
  - Multiprocessing (SMP)
    - Verwaltung mehrerer Prozessoren
    - für Benutzer transparent
    + Verfügbarkeit, Performanz, Skalierbarkeit etc.
    − schwierig zu implementieren ...



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

aktuelle Betriebssysteme (cont.)

- verteilte Betriebssysteme, einheitliche Sicht auf Cluster
- spezielle Anforderungen
  - Echtzeit Betriebssysteme
  - Fehlertoleranz

Aufgaben eines Betriebssystems

- Prozessverwaltung
  - Prozesse starten und beenden
  - Scheduling: Prozesse CPUs zuordnen
  - Prozesswechsel
  - Prozesssynchronisation und Interprozesskommunikation
  - Verwaltung der dazu notwendigen Datenstrukturen (Prozesskontrollblock)
- Speicherverwaltung
  - Zuordnung des (virtuellen) Adressraums zu Prozessen
  - *Swapping*: Hauptspeicher $\Leftrightarrow$ sekundärer Speicher
  - Seitenadressierung (*Paging*) und Segmentierung
- Ein-/Ausgabeverwaltung
  - Verwaltung von FIFOs
  - Zuordnung von I/O-Geräten und -Kanälen zu Prozessen
- weitere Funktionen
  - Interruptverarbeitung
  - Abrechnung der Ressourcen (*Accounting*)
  - Protokollierung (*Monitoring*)

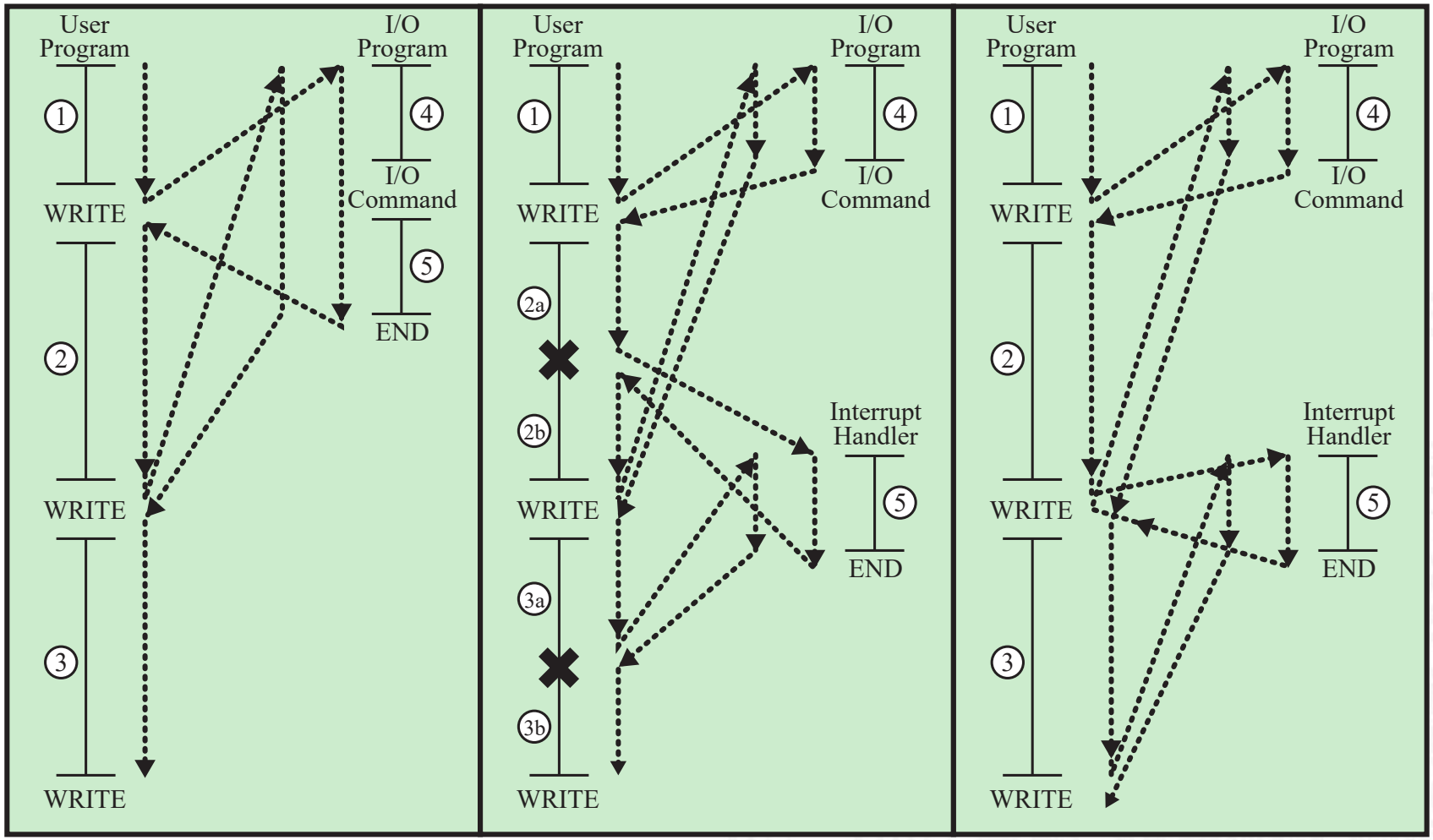
Interrupts

- sequenzieller Ablauf der Programmabarbeitung wird unterbrochen
- Bessere Ausnutzung des Prozessors
  - I/O, Platten, Hauptspeicher langsamer als CPU
  - CPU muss „warten“ $\Rightarrow$ schlechte Nutzung
- Interrupts durch
  - **Programm**: Ausnahmebehandlung („*Exception*“) z.B. Überlauf, Division durch 0, illegale Anweisung, ungültige Speicheradresse
  - **Timer**: regelmäßige Ausführung von Aufgaben
  - **I/O**: externe Hardware meldet: Ende einer Operation, Fehler
  - **Hardwarefehler**: Speicherparität, Spannungsversorgung, Temperatur ...

Interrupt: Beispiel

- Benutzerprogramm schreibt auf Festplatte, rechnet (1,2,3)
- I/O-Programm für Plattenzugriff (4,5)
  - Teil des Betriebssystems
  - Schnittstelle durch *System-Call*
- Zeit zwischen *I/O Command* und *Interrupt* wird durch langsames Gerät bestimmt

Interrupt: Beispiel (cont.)

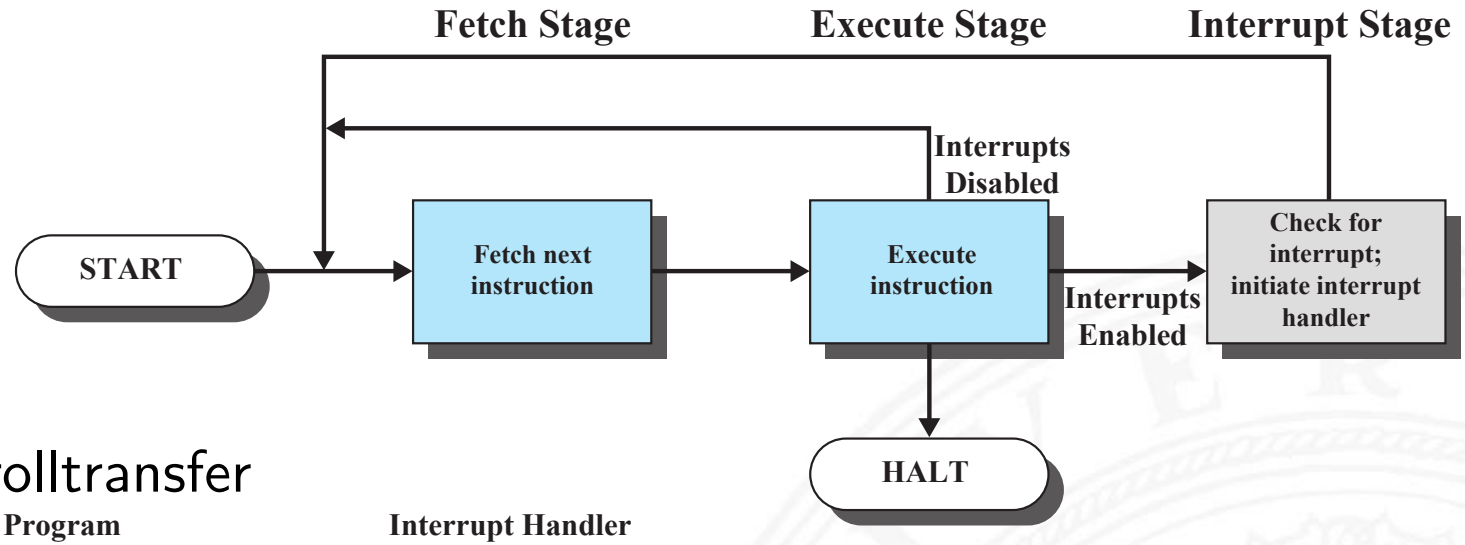


(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

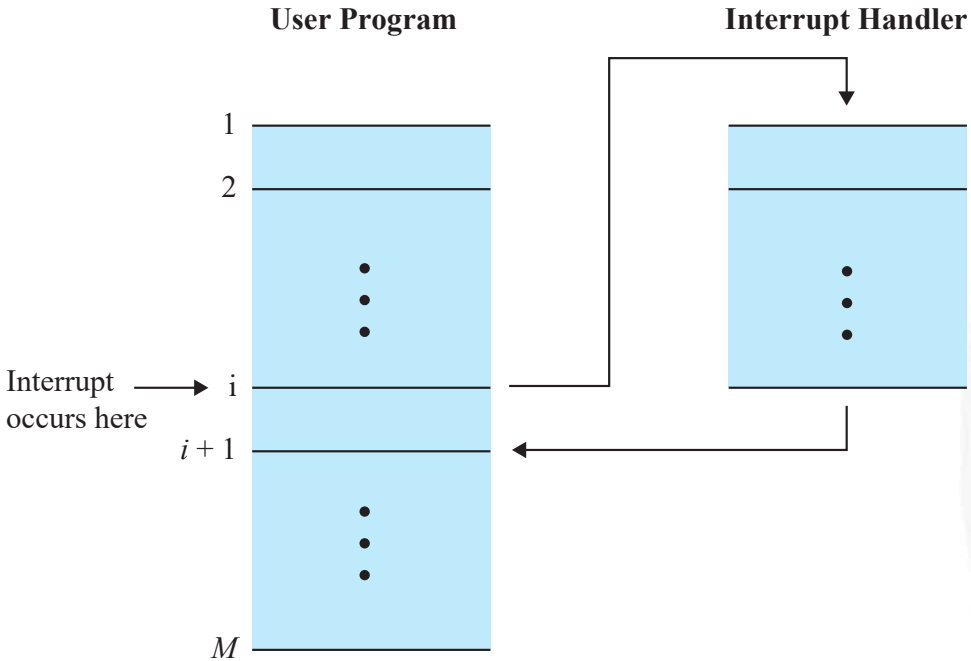
✖ = interrupt occurs during course of execution of user program

Interrupt: Programmablauf

- Ausführungszyklus der Befehle ergänzt

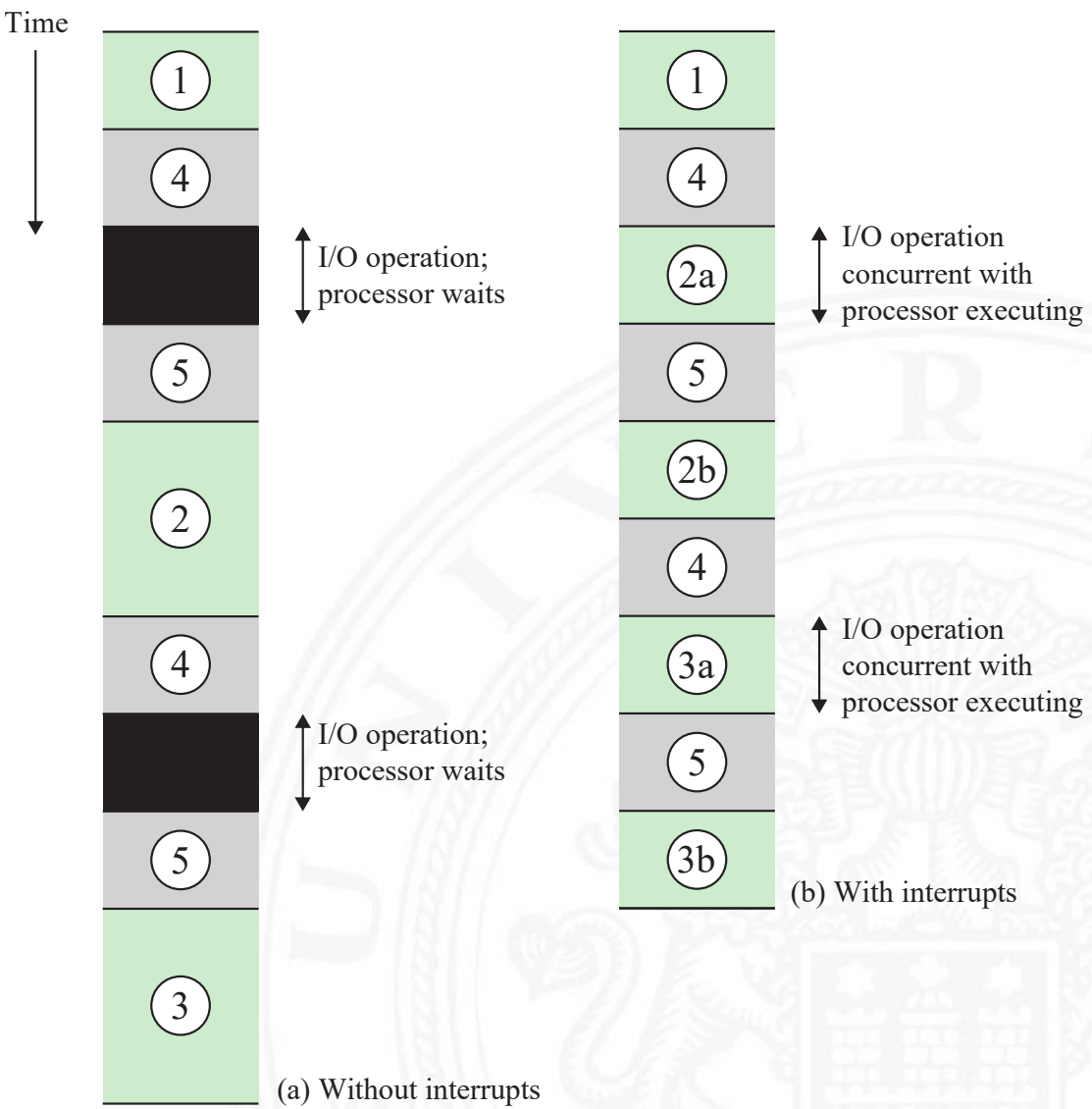


- Kontrolltransfer



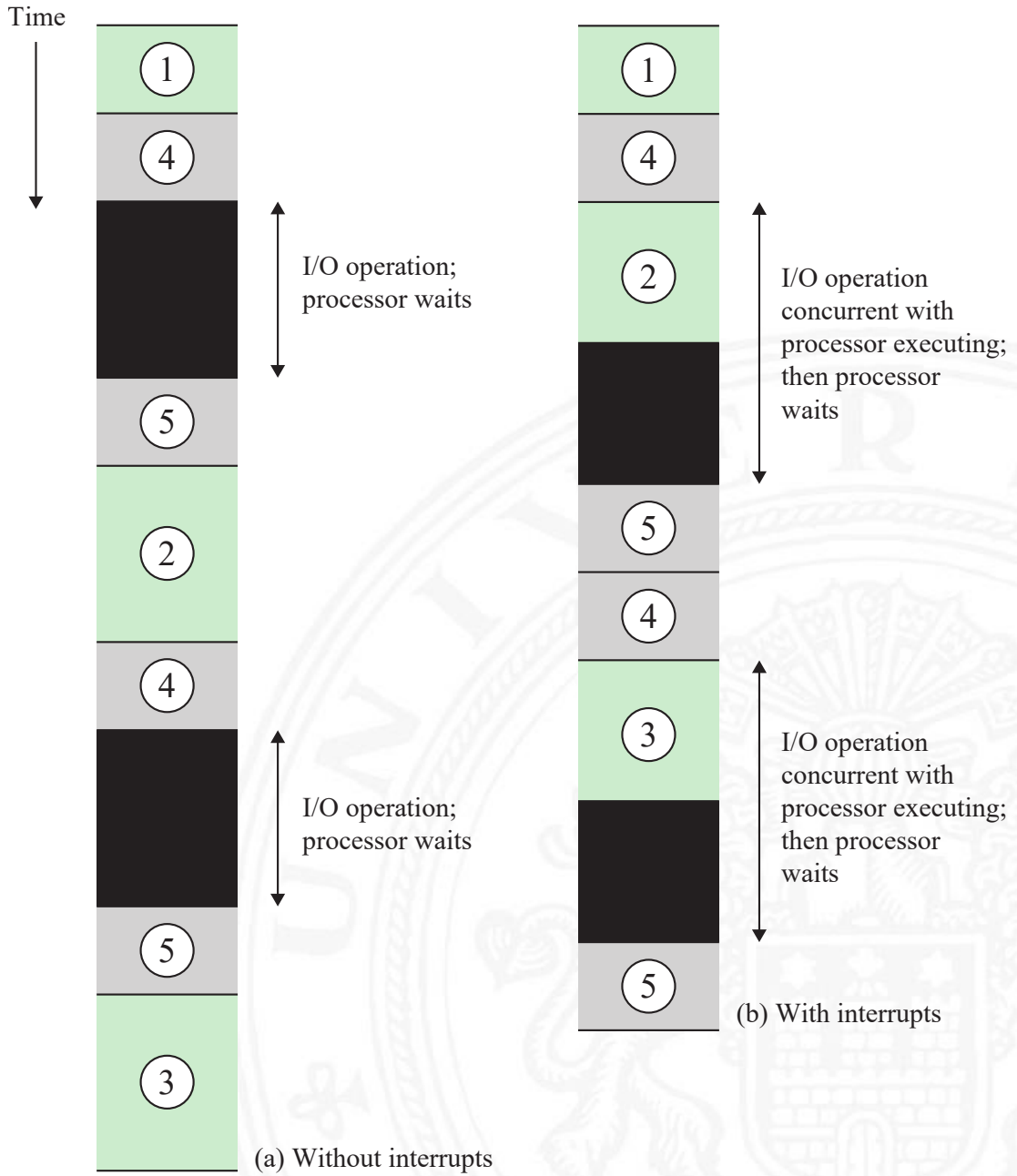
Interrupt: Programmablauf (cont.)

- kurze I/O Wartezeit



Interrupt: Programmablauf (cont.)

- lange I/O Wartezeit

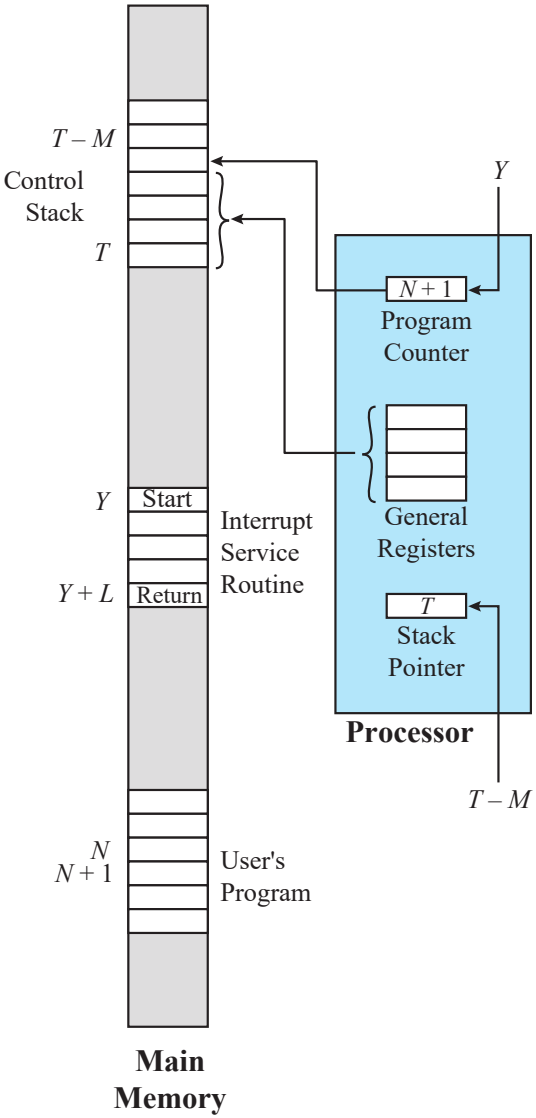


Interruptverarbeitung

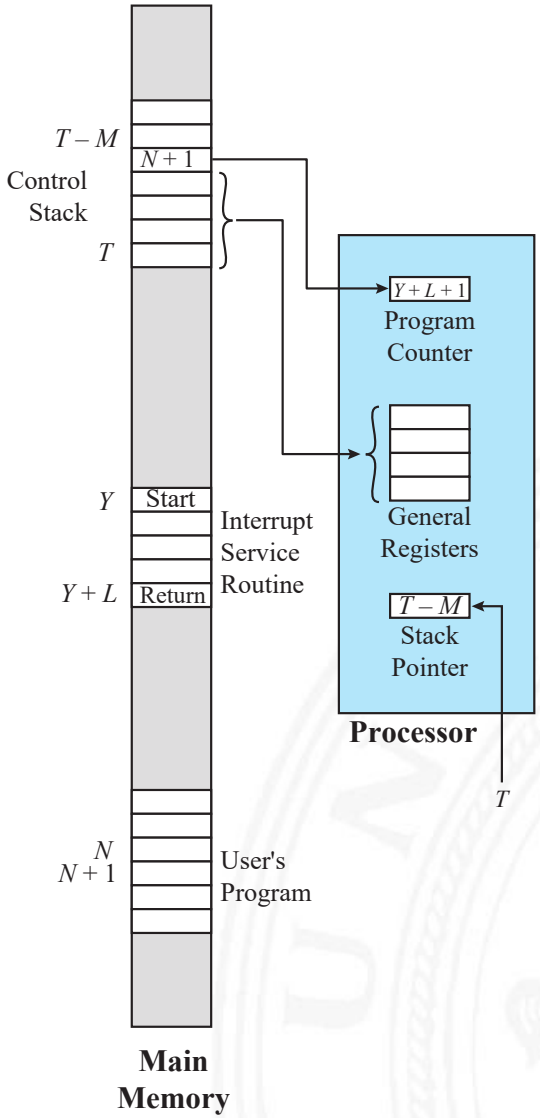
1. HW Gerät, Systemhardware liefert Interrupt
2. HW Prozessor beendet aktiven Befehl Pipelining!
3. HW Prozessor bestätigt Interrupt
4. HW Programmstatus (PC, Register, Speicherinformation etc.) auf *Control Stack* sichern
5. HW PC mit Interrupt (-startadresse) initialisieren

   Wechsel in privilegierten Modus (*Kernel Mode*)
6. SW ggf. weitere Informationen auf *Control Stack* sichern
7. SW Interruptverarbeitung / *Interrupt Handler* (Programm)
8. SW Programmstatus aus 6. wiederherstellen
9. SW Programmstatus aus 4. wiederherstellen;
   PC mit Programmfortsetzung initialisieren

Interruptverarbeitung (cont.)



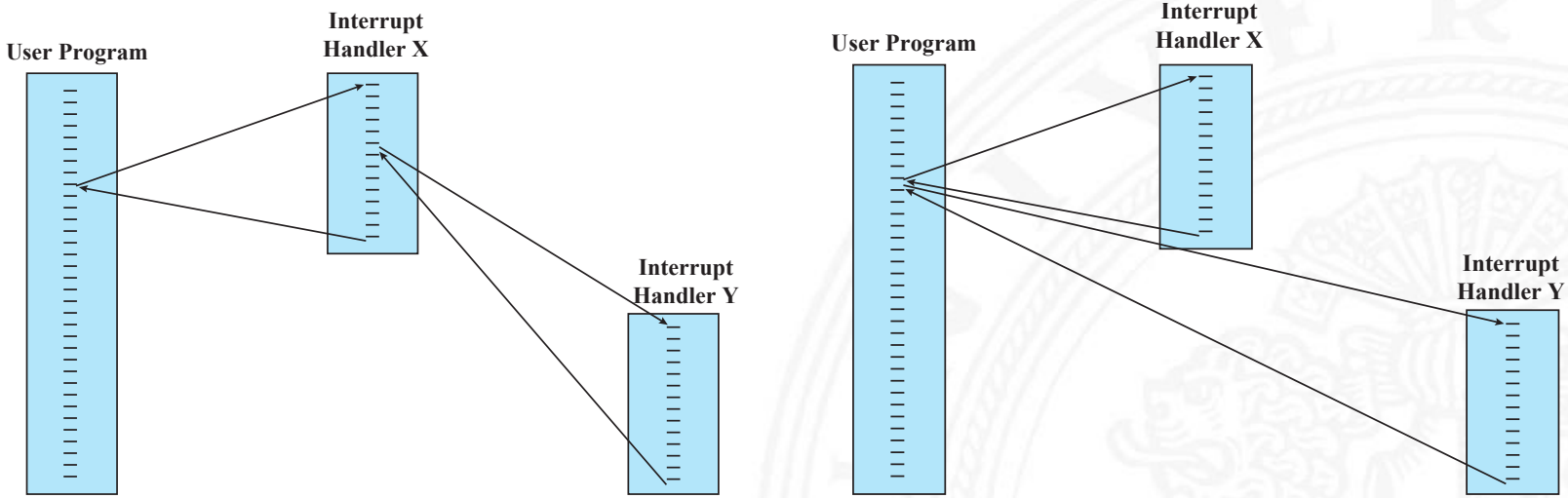
(a) Interrupt occurs after instruction at location N



(b) Return from interrupt

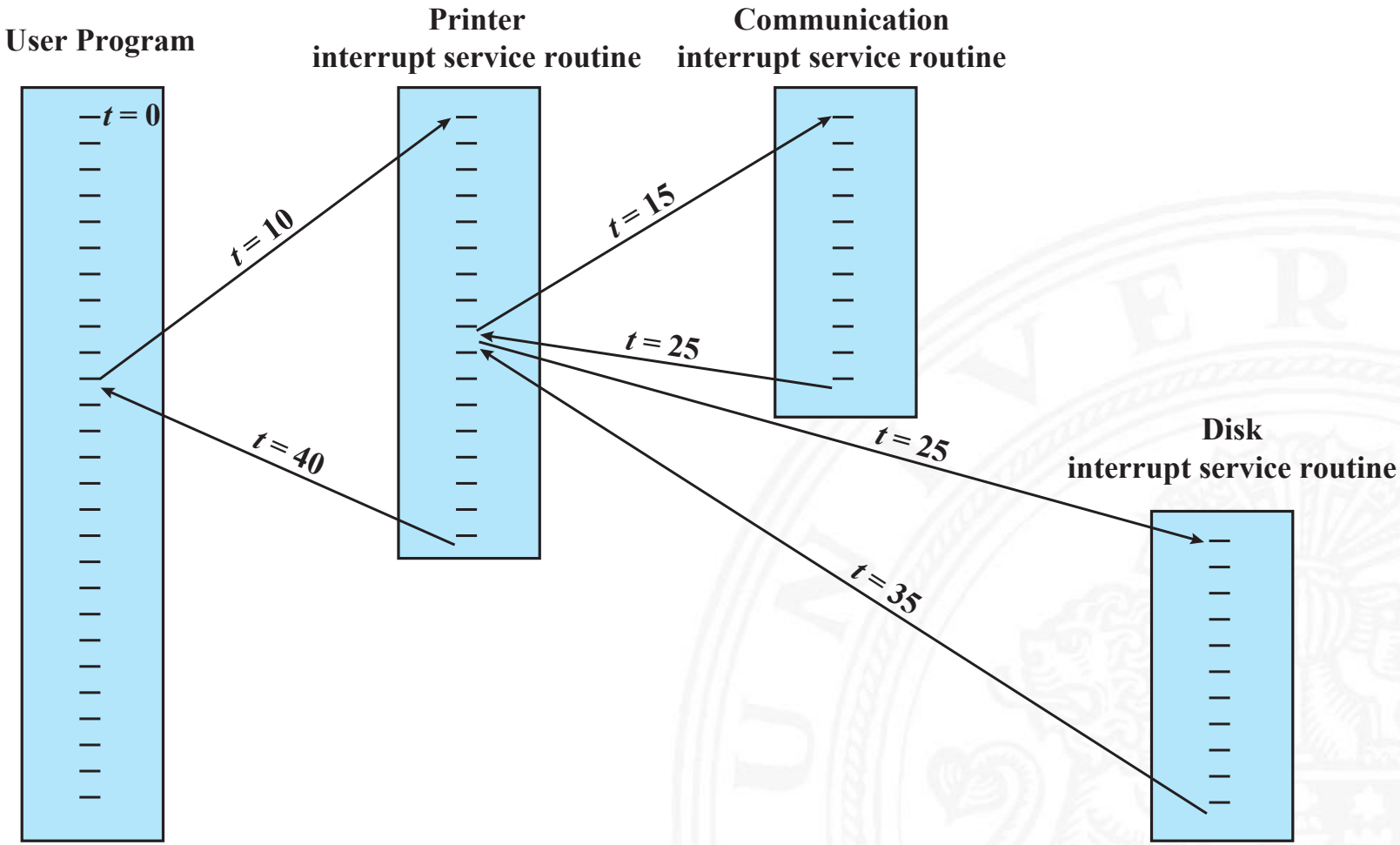
Mehrfache Interrupts

- während Interruptverarbeitung kommt Interrupt
  1. weitere Interrupts deaktivieren
  2. verschiedene Interruptprioritäten
- Interrupts können „verloren gehen“, ggf. Zwischenspeichern
- Schachtelung und/oder sequenzielle Abarbeitung



Mehrfache Interrupts (cont.)

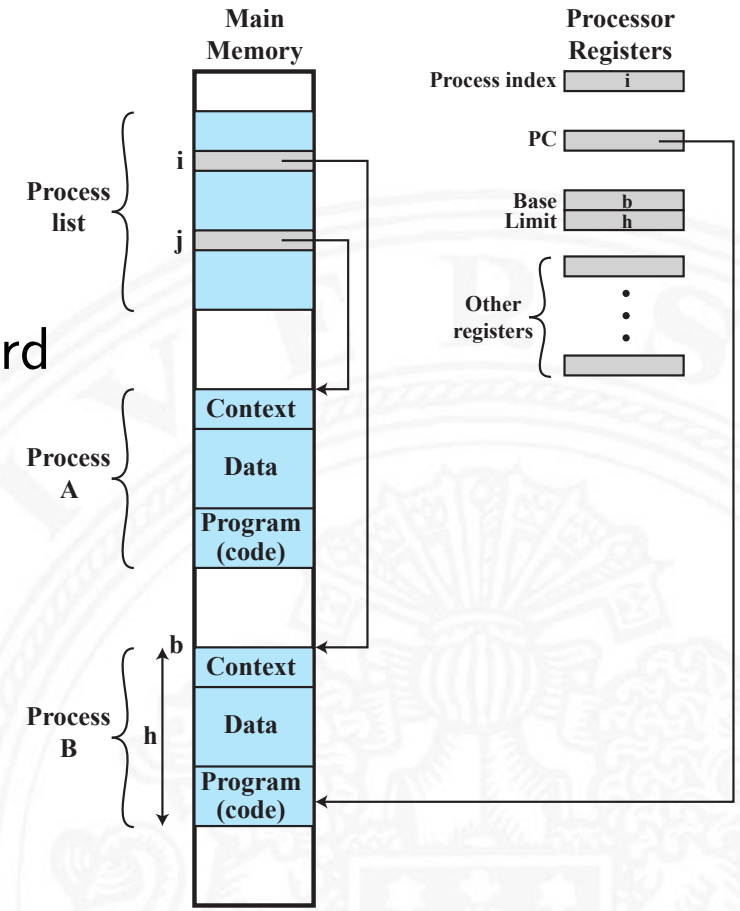
- Beispiel: zeitlicher Verlauf



Prozesse

Prozess: zentral verwaltete Einheit in Betriebssystemen

- Programm während der Ausführung
- Instanz eines laufenden Programms
- Einheit, die Prozessor zugewiesen wird
  –"– die von Prozessor ausgeführt wird
- Aktivität bestehend aus
  - einem einzelnem sequenziellen Ablauf
  - einem Zustand
  - zugehörigen Systemressourcen



Fehler bei Prozessabarbeitung

- Synchronisationsfehler
  - Prozess muss warten
  - Reaktivierung durch externes Ereignis

  $\Rightarrow$ Ereignis wird nicht, wird mehrfach ausgelöst
- gegenseitiger Ausschluss (*Mutual Exclusion*)
  - mehrere Prozesse mit gleichzeitigen Zugriff auf Ressource, z.B. Speicher, Datei

  $\Rightarrow$ Sperrmechanismen: Semaphore, Mutex, Monitor
- nichtdeterministisches Verhalten
  - mehrere Prozesse/Threads kommunizieren über *Shared Memory*

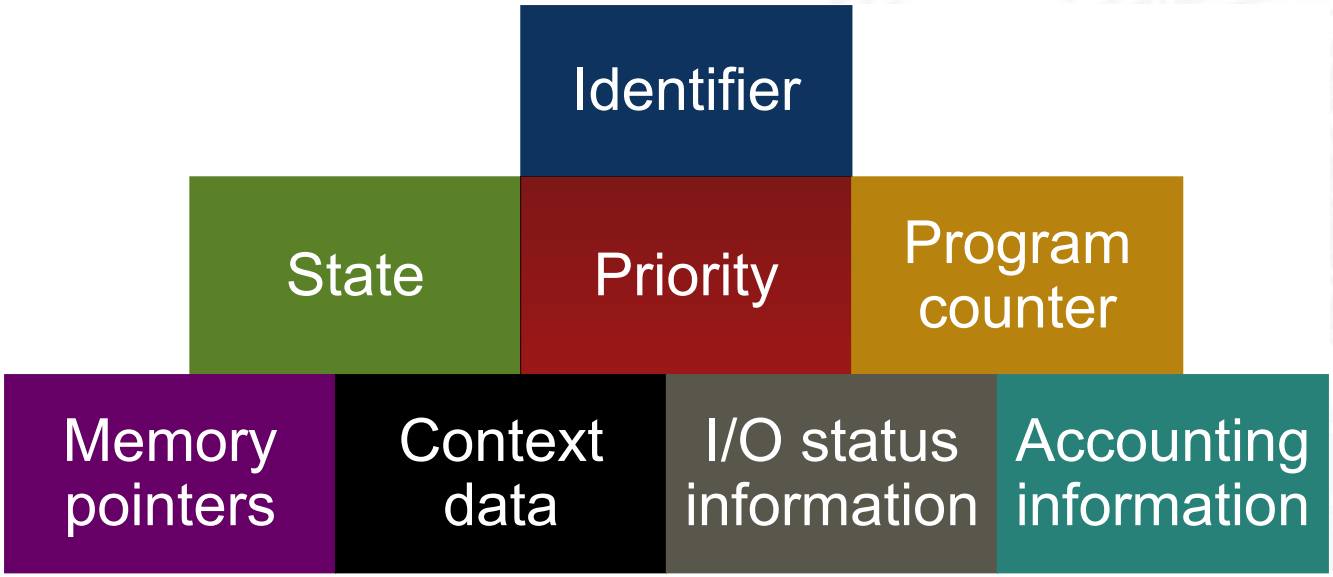
  $\Rightarrow$ transiente Effekte: Programme überschreiben sich Werte abhängig vom Scheduling durch Betriebssystem
- Deadlocks

  $\Rightarrow$ Prozesse warten (zyklisch) aufeinander

$\rightarrow$ siehe Abschnitt *15.4 Synchronisation und Kommunikation*

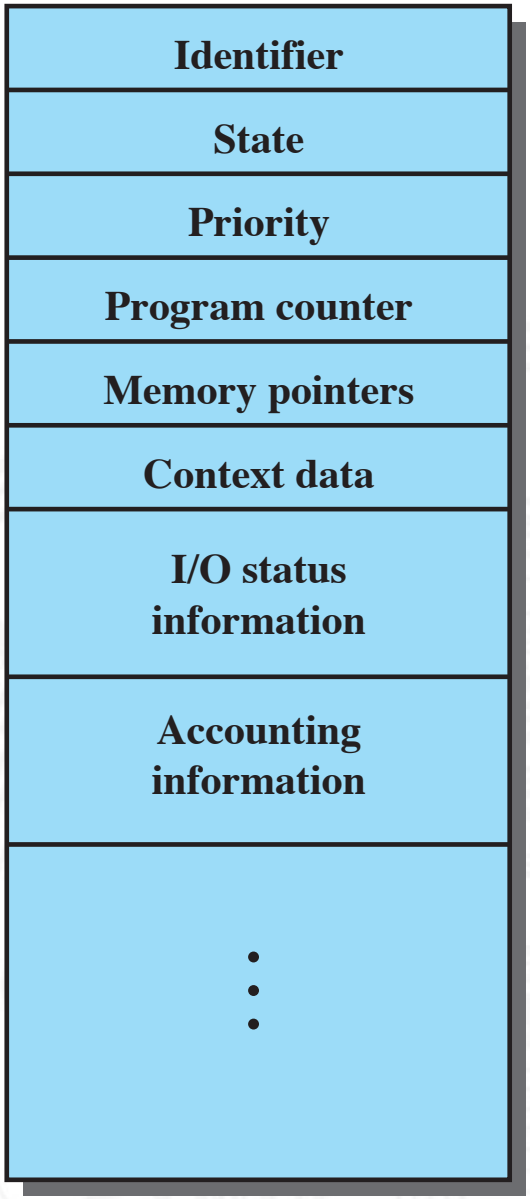
Komponenten eines Prozesses

1. das ausführbare Programm (*Text*-Segment)
2. die zugehörigen Daten (*Data*-Segment)
3. der Programmkontext
   - prozessspezifische Daten des Betriebssystems
   - Inhalt der Prozessorregister
   - Warten auf Ereignisse?
   - Prioritäten, Rechte, Abrechnungsinformationen etc.



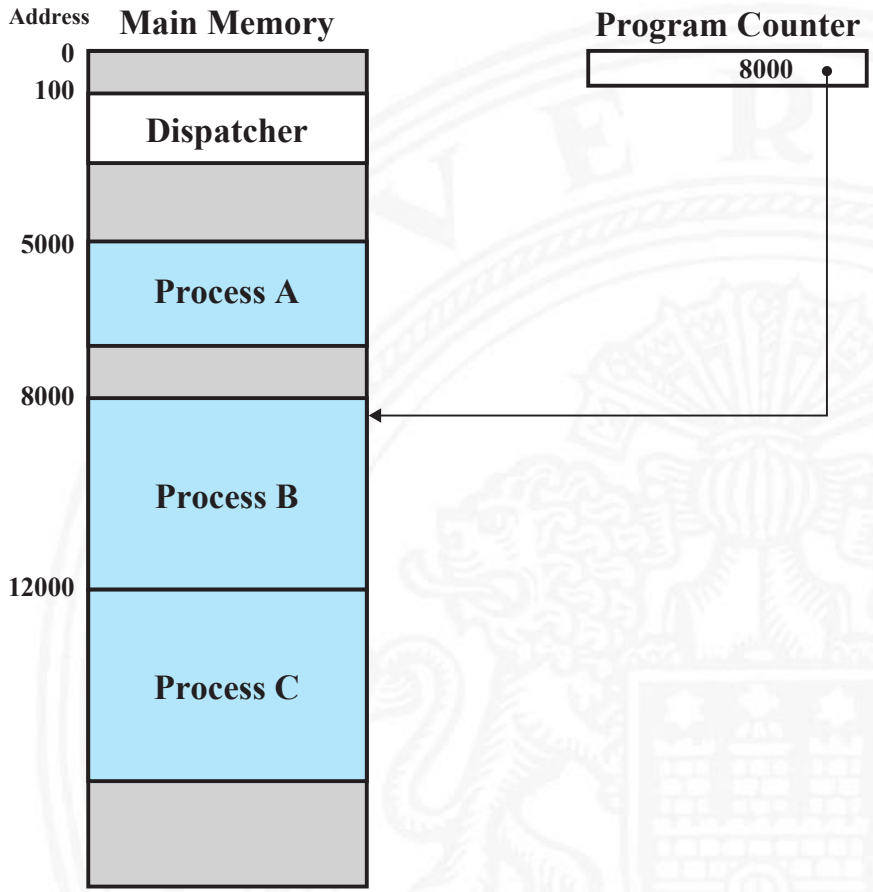
Komponenten eines Prozesses (cont.)

- Prozesskontrollblock
  speichert Verwaltungsinformation



Prozessablauf

- *Trace*: Folge von Instruktionen
  - für einzelnen Prozess ⇒ Laufzeitverhalten
  - für Prozessor ⇒ zeigt Prozesswechsel
- *Dispatcher*: BS-Komponente, die Prozessor Prozessen zuordnet
- Beispiel



Prozessablauf (cont.)

- 3 Prozesse: A, B, C
- Dispatcher (blau)

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A **(b) Trace of Process B** **(c) Trace of Process C**

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
---------- I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------Timeout	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------Timeout	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------Timeout	

laufende Prozesse

- Prozesse starten
  - neuer Batch-Job
  - interaktiver Login (Kommandozeile / *Shell*)
  - durch Betriebssystem: neuer Dienst, z.B.: nach Booten
  - durch laufenden Prozess: explizite Parallelität

- *Parent*: laufender Prozess, der Neuen startet

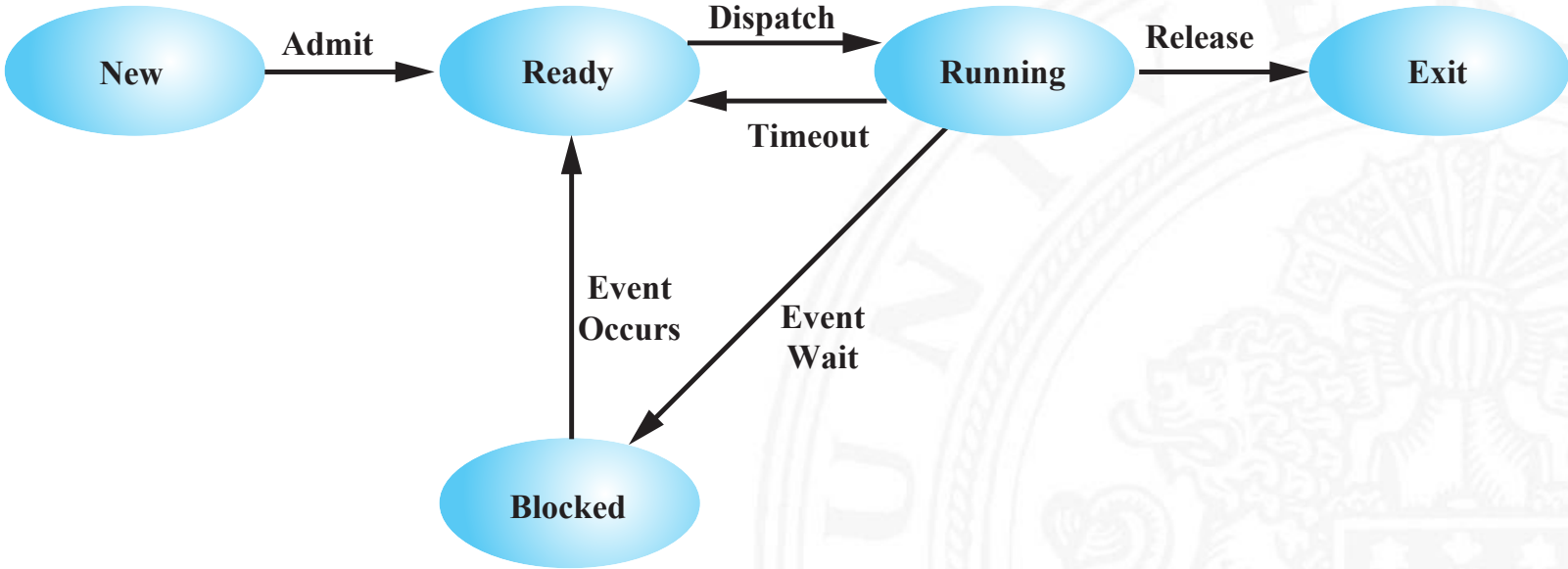
  Child: neu gestarteter Prozess
- Prozesshierarchie (Baumstruktur) (`htop`)

laufende Prozesse (cont.)

- Prozessende
  - normales Programmende, Berechnung/Aufgabe fertig
  - Zeitlimit überschritten:
    maximale Laufzeit, Warten auf Benutzereingabe ...
  - Timeout: Warten auf Event/Systemsignal
  - Speicherlimit: kein (virtueller) Speicher mehr verfügbar
  - Adressverletzung: versuchter Zugriff auf ungültige Speicheradresse
  - Zugriffsfehler: ungültiger Zugriff auf Ressource,
    z.B.: Schreiben in read-only Datei
  - Ein-/Ausgabefehler:
    Lesefehler in Datei, Datei nicht vorhanden ...
  - Arithmetischer Fehler: Teilen durch 0 ...
  - Datenfehler: „falscher“ Typ in Datenstrukturen ...
  - ungültiger Befehl: kein Assemblerbefehl (in Datensegment?)
  - unerlaubter Befehl: privilegierter Befehl im User Mode
  - Parent Anfrage an Child
  - Parent terminiert $\Rightarrow$ Child-Prozess beenden
  - Abbruch durch: Betriebssystem, Operator, Benutzer

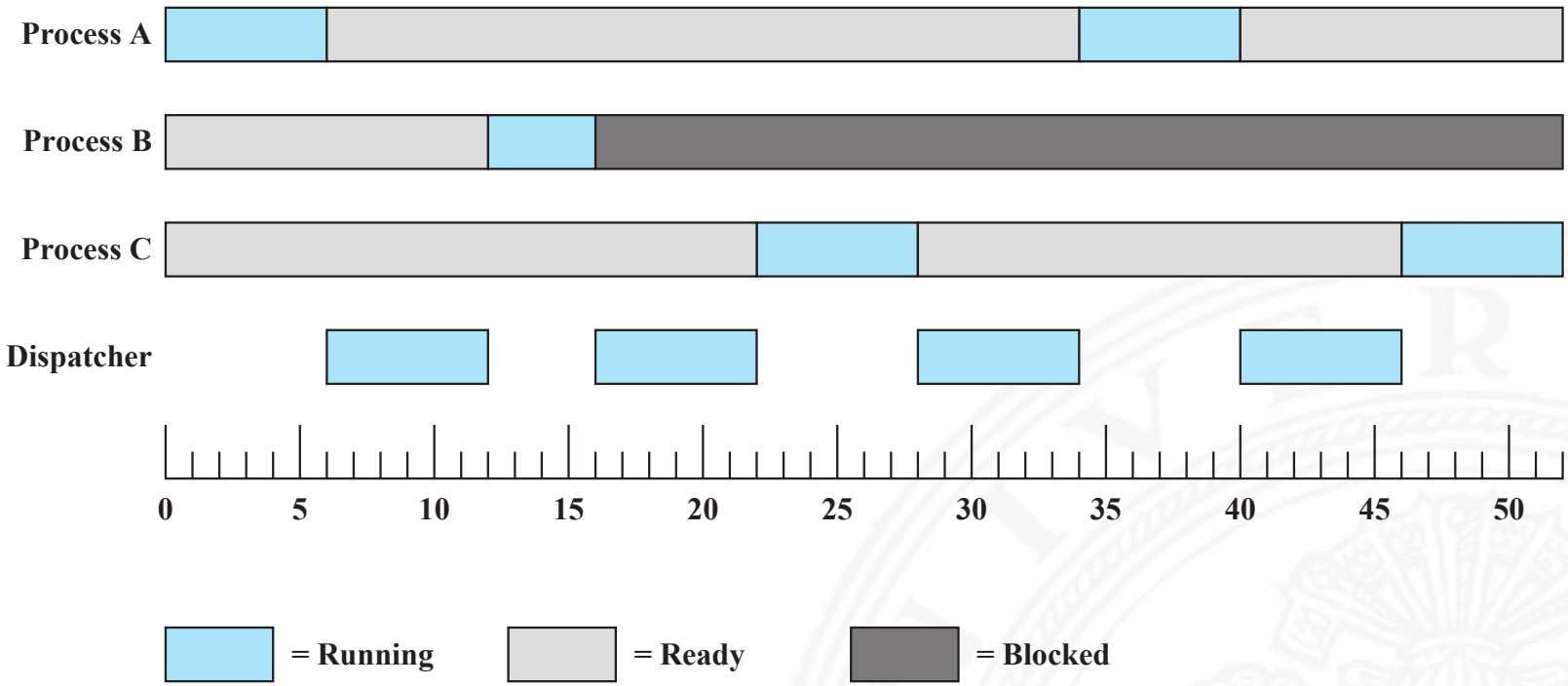
Prozessmodell

- Prozesse werden dynamisch gestartet und beendet
- –"– warten auf I/O / Systemereignisse
- –"– werden unterbrochen: Time-Sharing Betrieb
- Verwaltung durch Dispatcher
- Zustandsautomat



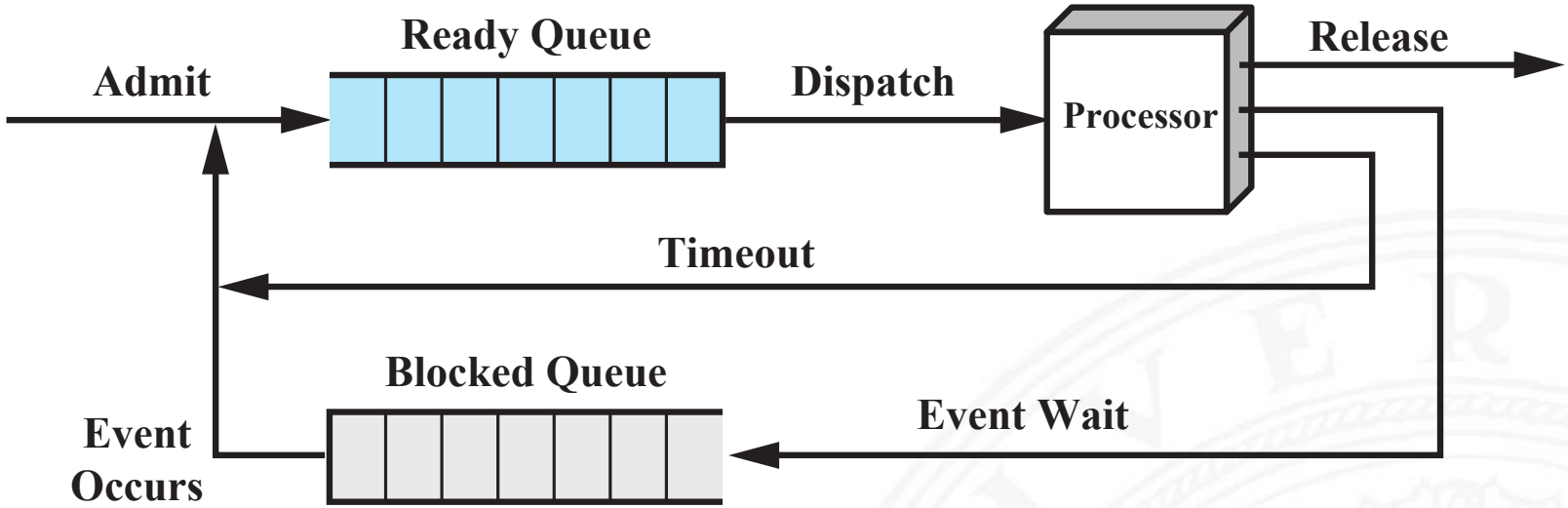
Prozessmodell (cont.)

- Trace

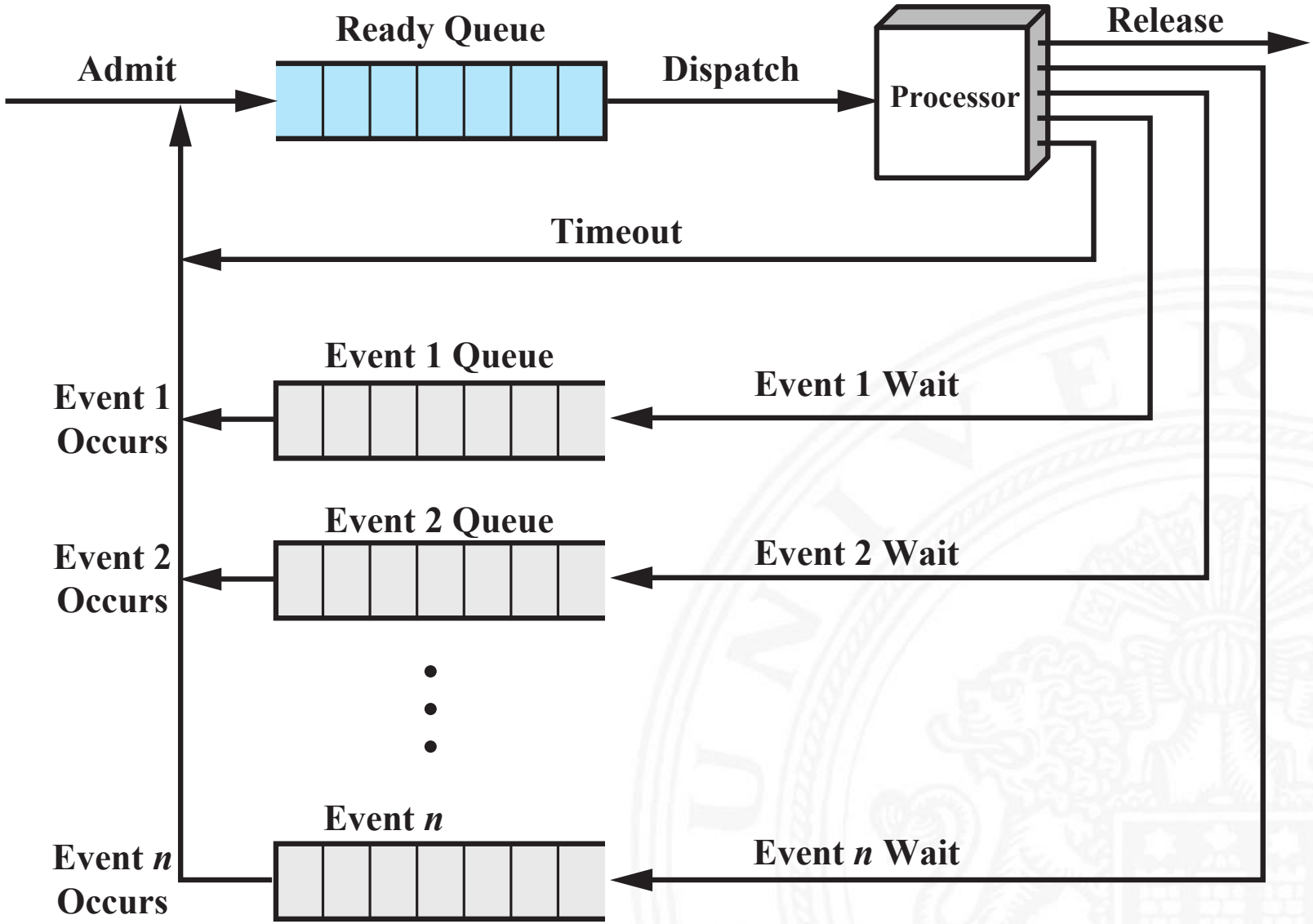


Prozessmodell (cont.)

- Warteschlangen, ggf. mit Trennung nach Signal

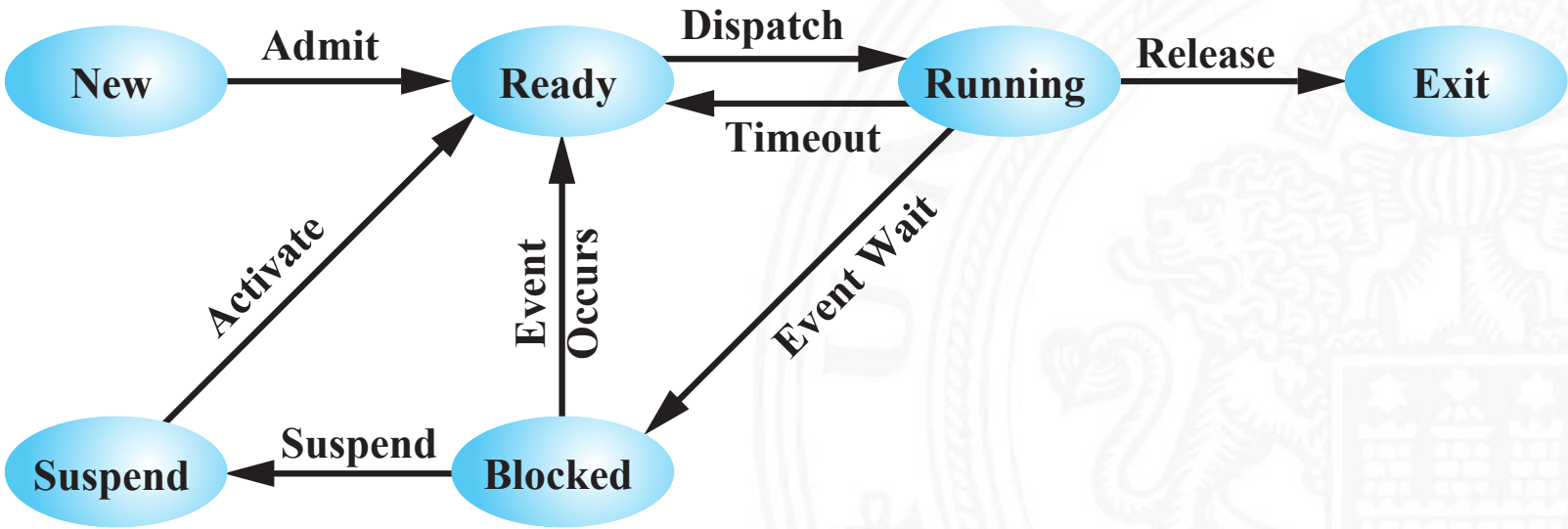


Prozessmodell (cont.)

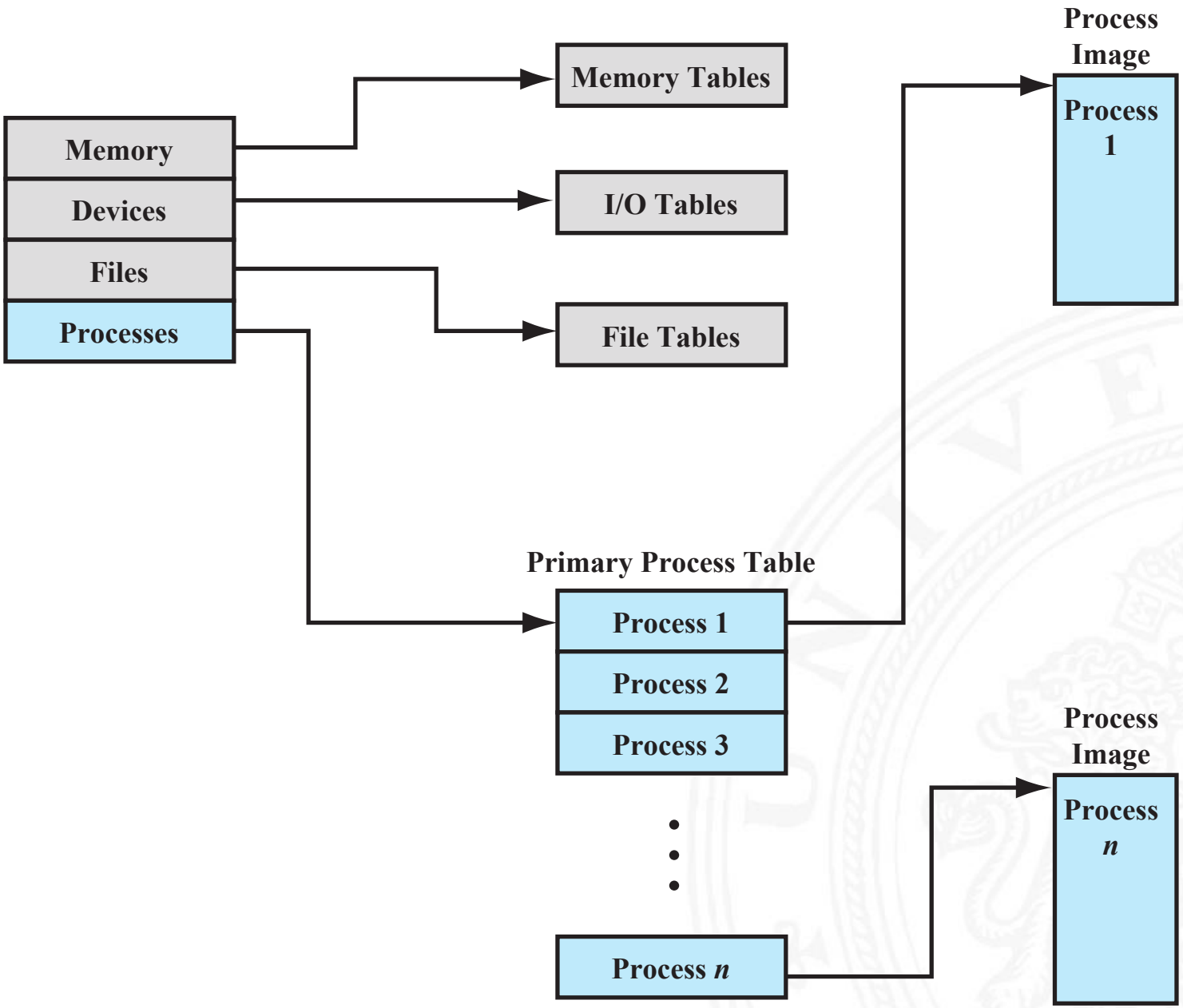


Prozessunterbrechung

- *Swapping*: Auslagerung von Prozessen, bzw. Teilen von Prozessen auf sekundären Speicher (HDD, SSD)
- Prozessunterbrechung (*suspend*) durch
  - Swapping: Betriebssystem benötigt Hauptspeicher
  - Timing: periodische Ausführung ...
  - Parent Anfrage an Child, z.B.: zur Synchronisation
  - Unterbrechung durch: Betriebssystem, Operator, Benutzer
- Erweiterung des Prozessmodells



Kontrollstrukturen



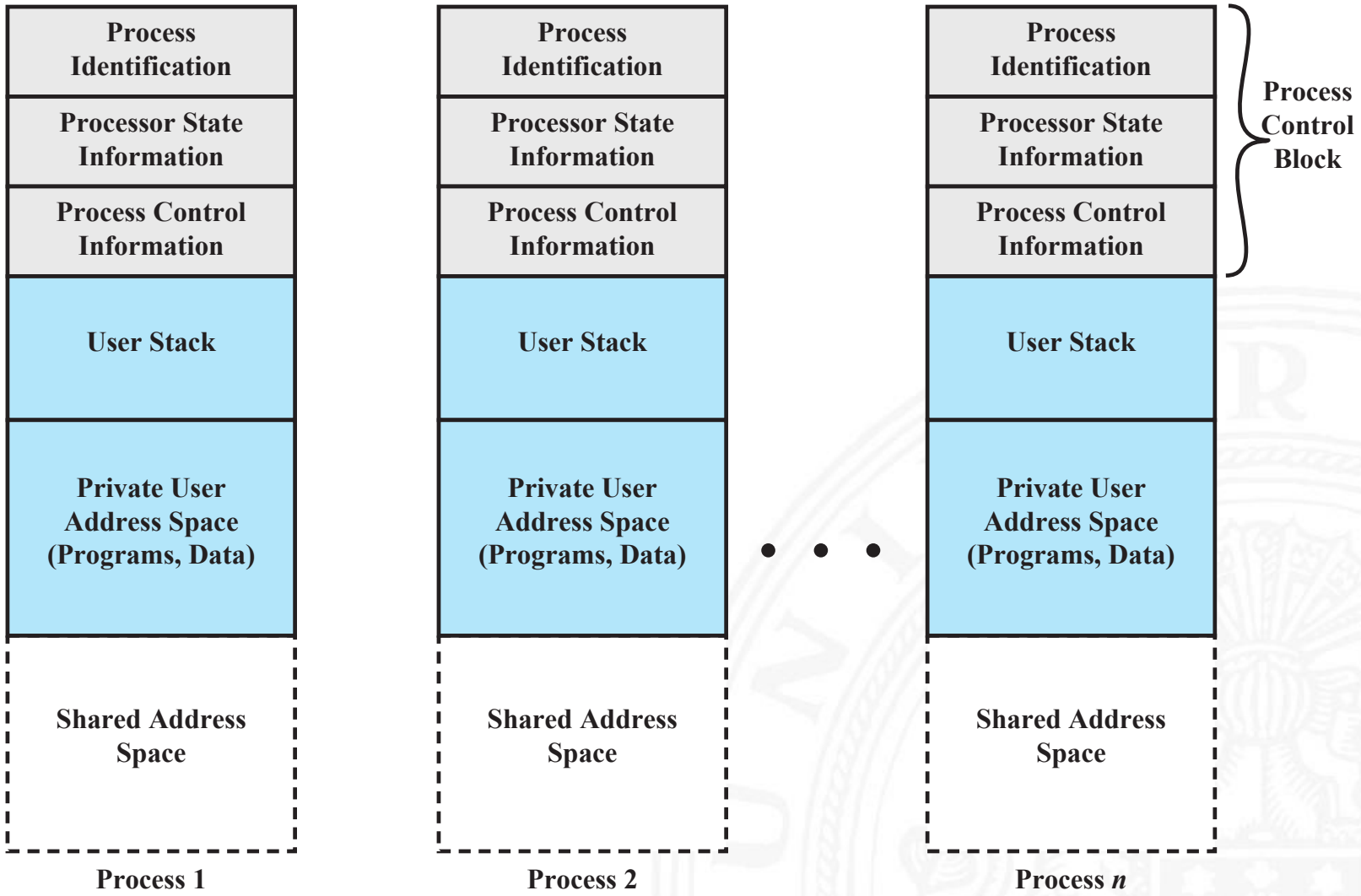
Kontrollstrukturen (cont.)

- Prozesstabellen
- Speicherverwaltung: *Memory Tables*
  - Verwaltung von virtuellem Speicher
  - Zuordnung für Hauptspeicher (RAM)
  - Zuordnung für sekundären Speicher (HDD, SSD)
  - Attribute für Speicherblöcke, z.B.: Speicherschutz
- Ein-/Ausgabeverwaltung: *I/O Tables*
  - Zuordnung zu Prozessen
  - Status von I/O-Befehlen
  - Informationen zu Befehlen: Startadresse in Hauptspeicher, Datengröße
- Datei-Verwaltung: *File Tables*
  - Existenz von Dateien / Dateinamen
  - Ort auf Sekundärspeicher
  - Status, z.B.: geöffnet (rw, ro)
  - weitere Attribute: Zugriffsrechte, Zeitstempel etc.

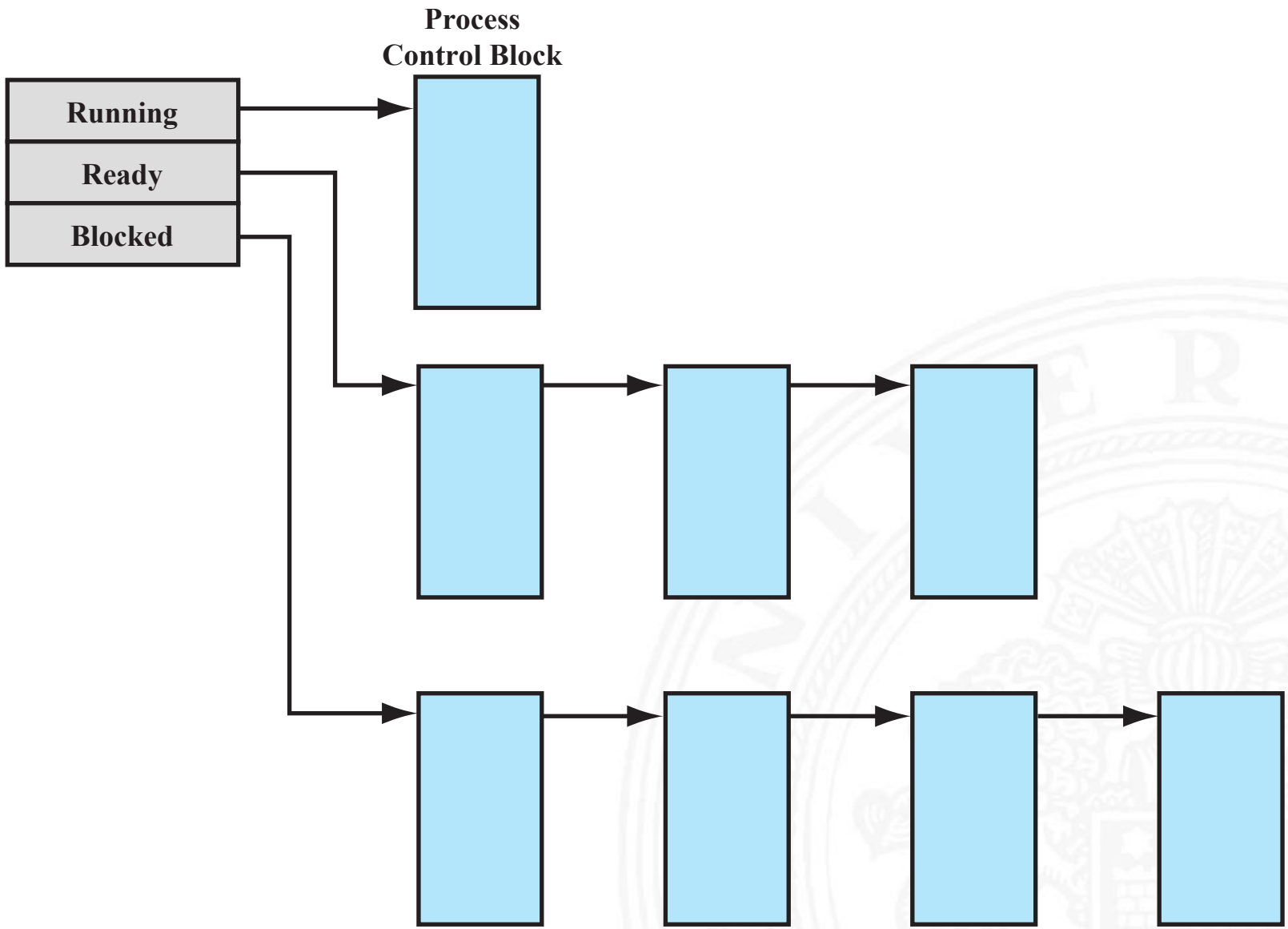
Prozessabbild / *Process image*

- Programmcode
- Datenstrukturen des Programms: statische Datenstrukturen und dynamischer Speicher, z.B.: *Heap*
- *Stack*: Unterprogrammaufrufe und -Datenstrukturen
  → siehe Abschnitt *13.3 Funktionsaufrufe und Stack*
- Prozesskontrollblock, siehe Folie 1171
  - Identifier, Parent, Child-Liste
  - Register: für Benutzer sichtbar + „*Rename-Register*“
  - Status-Register: Programmzähler, Flags, Modus Interrupts Enabled ...
  - Stack-Pointer
  - Scheduling Information: Zustand des Prozessmodells, Priorität ...
  - Informationen für Interprozesskommunikation
  - Privilegien: Zugriffsrechte auf Speicherbereiche, I/O ...
  - Speicherverwaltung: Tabellen für *Virtual Memory*
  - aktuelle Ressourcen, z.B.: geöffnete Dateien
  - ...

Prozessabbild / *Process image* (cont.)



Prozessabbild / *Process image* (cont.)

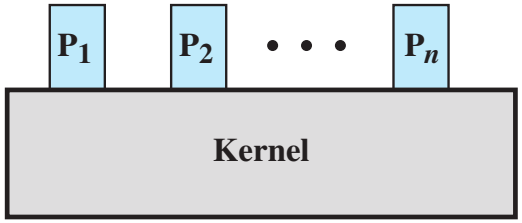


Kontextwechsel

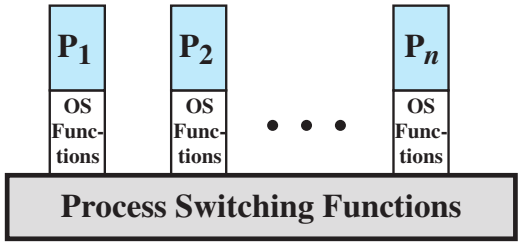
- Interrupt-Mechanismen
  - Interrupt: asynchrones, externes Ereignis
  - Trap: Fehler während der Programmabarbeitung
  - System-Call: Aufruf einer Betriebssystemfunktion
- kein „wartender“ Interrupt $\Rightarrow$ nächsten Befehl holen
- Interrupt löst Kontextwechsel aus
  - Programmzähler mit Interrupt Handler initialisieren
  - Wechsel *User Mode* $\Rightarrow$ *Kernel Mode* für privilegierte Instruktionen
- Kontextwechsel
  - Kontext des Prozesses sichern
  - Prozesskontrollblock aktualisieren
  - –"– in „passende“ Warteschlange einfügen
  - anderen Prozess zur Ausführung wählen
  - dessen Prozesskontrollblock aktualisieren
  - Speicherstrukturen für neuen Prozess aktualisieren (Seitentabelle)
  - Kontext des neuen Prozesses einrichten

Kontextwechsel (cont.)

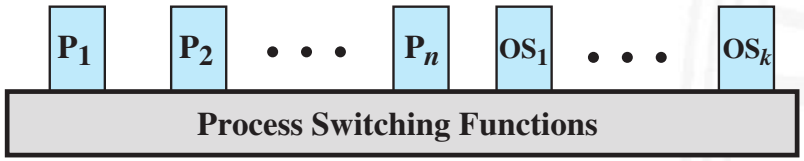
- Realisierungen
  - getrennter Betriebssystemkernel

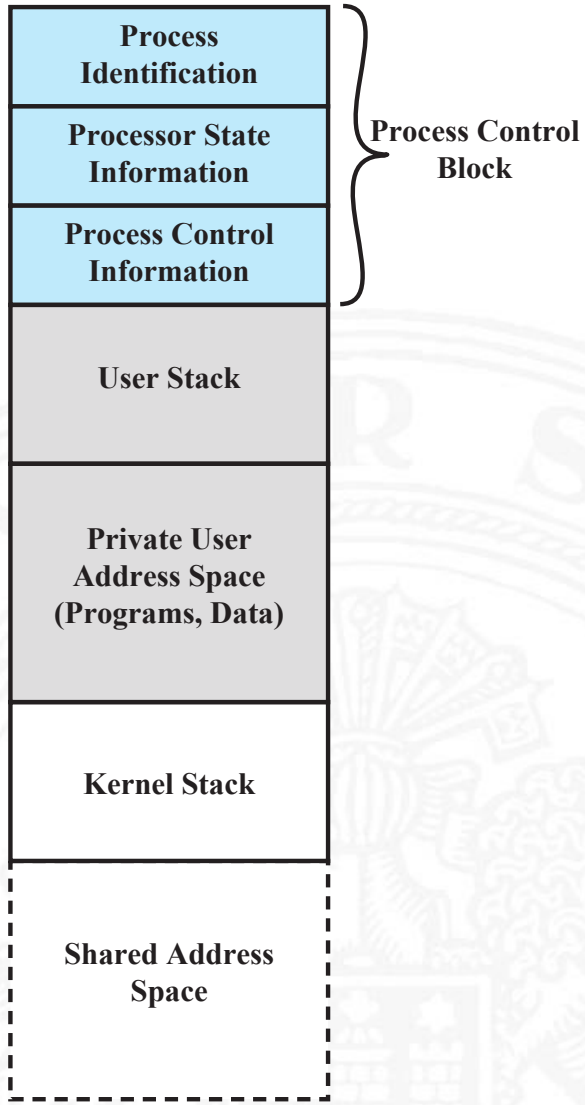


  - innerhalb des Benutzerprogramms



  - eigene Services/Prozesse (*Microkernel*)





Prozesse und Threads

- *Thread / Lightweight Process*
  - Betriebssystem: Zuordnung zu Prozessor (CPU)

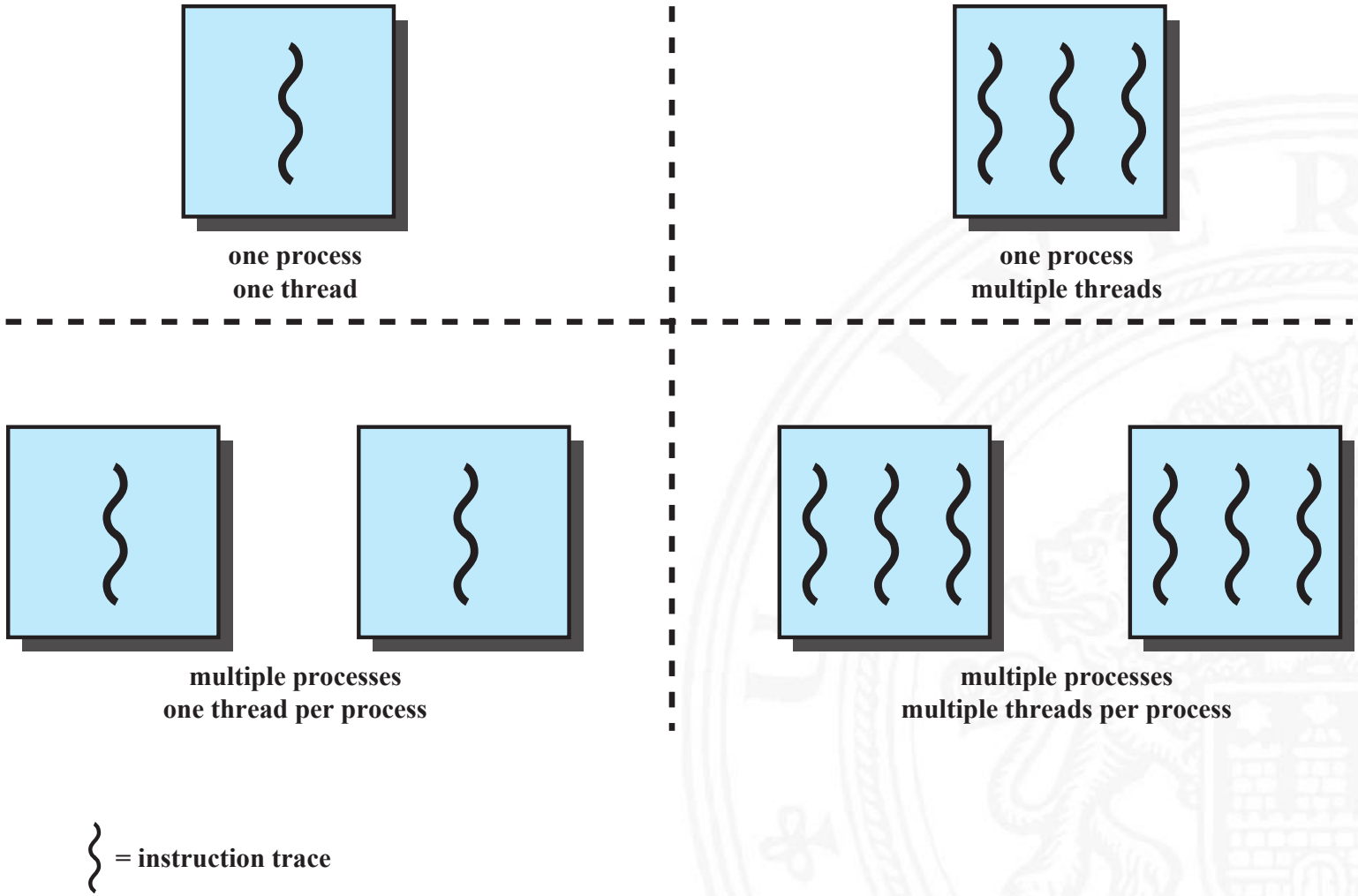
  $\Rightarrow$ Programmablauf (*Scheduling*, *Dispatching*)

- Prozess
  - Betriebssystem: Zuordnung zu Ressourcen (Speicher, Dateien, I/O ...)

  $\Rightarrow$ gesamter Kontrollblock

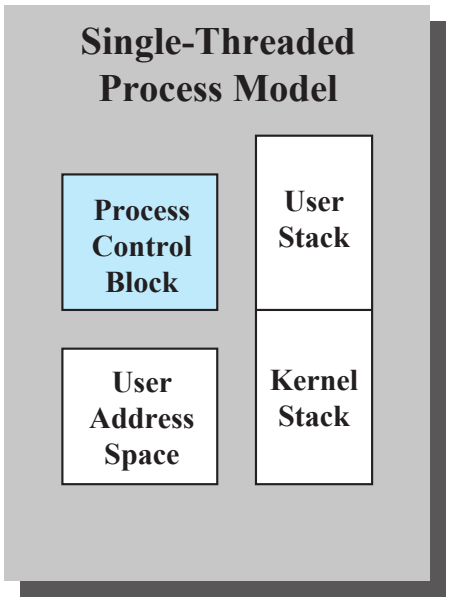
Prozesse und Threads (cont.)

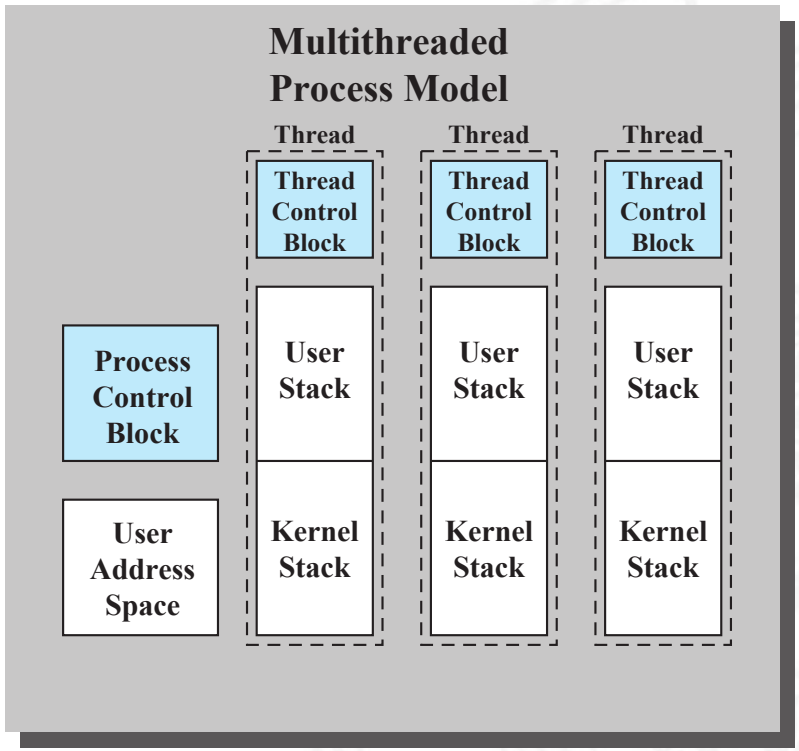
- *Multithreading*
  - mehrere parallele Ausführungen innerhalb eines Prozesses
  - von Programmiersprache und Betriebssystem abhängig



Threads

- eigener Zustand ($\widehat{=}$ Prozesszustand)
- eigener Kontext
- eigener Stack
- ggf. eigenen, statischen (Variablen-) Speicher
- Zugriff aller Datenstrukturen und Ressourcen des Prozesses
- geteilt mit allen anderen Threads



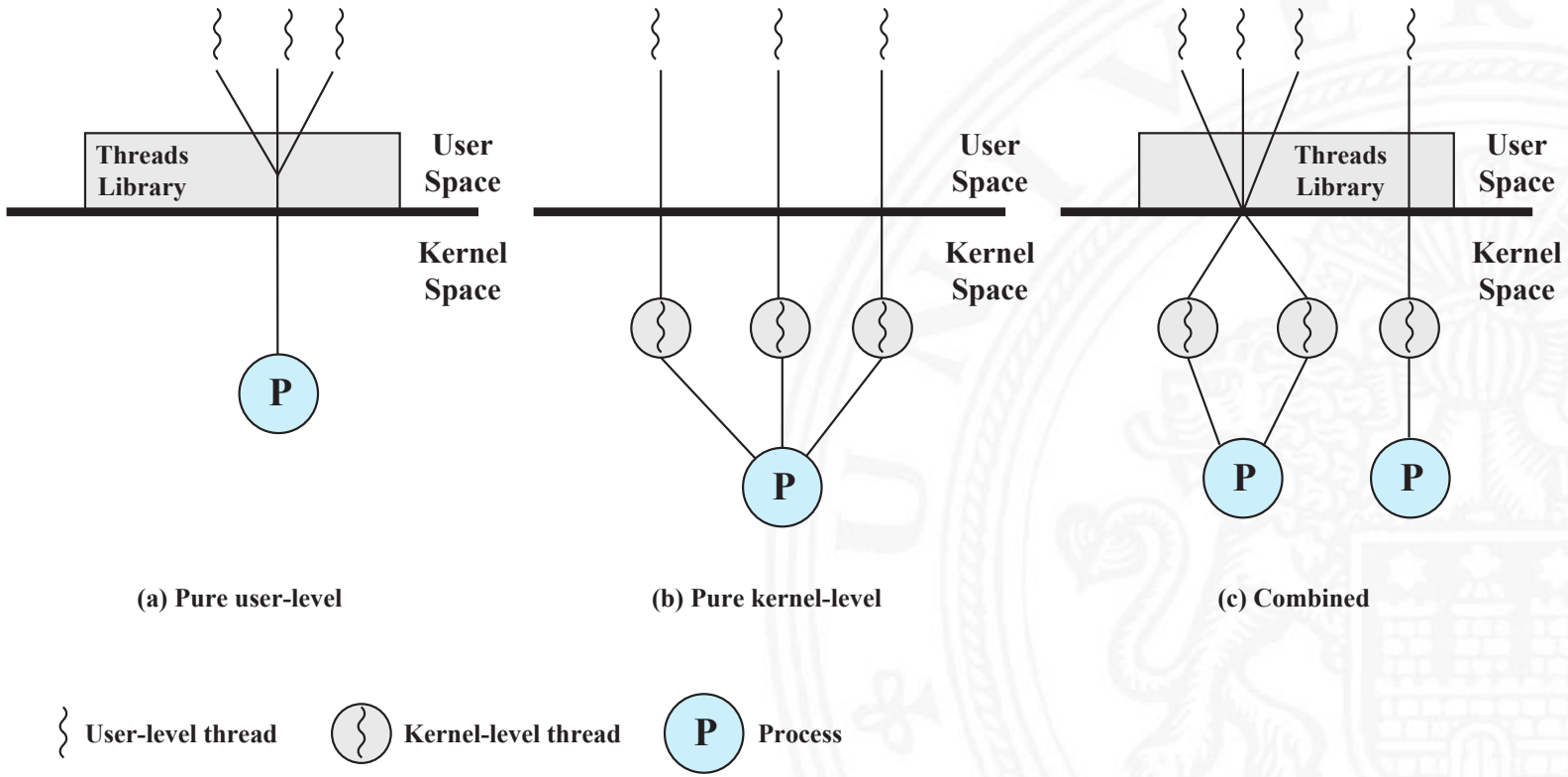


Threads (cont.)

- Vor- und Nachteile
  - \+ einfacher zu verwalten / erzeugen
  - \+ schneller zu beenden
  - \+ Wechsel zwischen Threads schneller als Prozess-Kontextwechsel
  - \+ effizientere Kommunikation
  - \+ effektiv, wenn auf I/O gewartet wird, z.B.: Serverdienste, RPC (Remote procedure calls), Browser-Tabs ...
  - \+ Parallelität ausnutzen
  - – Synchronisation wichtig! alle Threads arbeiten im gleichen Adressraum
- Arten von Threads
  - User Level Thread (ULT), eigene Bibliotheken
    - \+ Thread-Wechsel ohne Kernel Privilegien
    - \+ spezifisches (eigenes!) Scheduling
    - \+ läuft auf allen Betriebssystemen
    - – für Kernel nur ein Ablauf → keine Parallelität, System-Call blockiert alles

Threads (cont.)

- Kernel Level Thread (KLT)
  - \+ mehrere Threads in Multiprozessorumgebung
  - \+ Prozess (andere Threads) kann trotz blockiertem Thread weiterlaufen
  - \+ Betriebssystem selbst kann Multithreaded sein
  - – Wechsel zwischen Threads eines Prozesses bedingt Moduswechsel
- Mischformen



Nebenläufigkeit

- nebenläufige Prozesse und Threads
  - Multiprogramming: viele Prozesse, ein Prozessor
  - Multiprocessing: viele Prozesse, mehrere Prozessoren
  - verteiltes Rechnen

$\Rightarrow$ abwechselndes und überlapptes Rechnen

$\Rightarrow$ Timing / Abarbeitungsgeschwindigkeit nicht vorhersehbar

  - Aktivitäten anderer Prozesse oder der Benutzer
  - Interrupts
  - Scheduling durch Betriebssystem

- Begriffe
  - **atomare Operation**: Funktion oder Aktion; kann nicht unterteilt/unterbrochen werden, auch wenn sie aus mehreren Schritten besteht. Wird komplett oder gar nicht wirksam. Zentraler Mechanismus, zur Trennung nebenläufiger Prozesse.
  - **Critical Section / kritische Sektion**: Codebereiche mehrerer Prozesse, in denen auf gemeinsame Ressourcen (z.B. Speicher) zugegriffen wird.

Nebenläufigkeit (cont.)

- **Deadlock**: zwei oder mehr Prozesse können nicht weiterarbeiten, da sie gegenseitig aufeinander warten.
- **Livelock**: zwei oder mehr Prozesse wechseln ständig ihre Zustände durch Aktivitäten jeweils anderer Prozesse, ohne Fortschritte in der Bearbeitung.
- **Mutual Exclusion / gegenseitiger Ausschluss**: wenn ein Prozess in seiner Critical Section ist, kann kein zweiter Prozess in einer Critical Section sein, die die gleichen Ressourcen nutzt.
- **Race Condition**: mehrere Threads/Prozesse lesen und schreiben Daten, wobei das Ergebnis von deren zeitlicher Reihenfolge abhängig ist.
- **Starvation / „verhungern“**: ein lauffähiger Prozess könnte (weiter-) arbeiten, wird aber nie bedient.

- Kommunikationsmechanismen zwischen Prozessen/Threads
  - gemeinsamer Speicher (*Shared Memory*) $\Rightarrow$ Mutual Exclusion
  - Nachrichtenaustausch

Mutual Exclusion / gegenseitiger Ausschluss

- notwendig, um Race Conditions zu vermeiden
- durch Sicherung von Critical Sections
- mögliche Probleme: Deadlock, Starvation

- Uniprozessor: keine Interrupts in Critical Section
- atomare Hardwareoperationen („*compare & swap*“)
  - \+ gilt für: Uni-/Multiprozessor, beliebige Anz. Prozesse
  - \+ einfach zu verifizieren
  - \+ für beliebige Anzahl kritischer Sektionen
  - – *Busy-waiting*: Prozessor arbeitet immer
  - – Starvation möglich (wenn mehrere Prozesse warten)
  - – Deadlock möglich

Mutual Exclusion / gegenseitiger Ausschluss (cont.)

- Software Mechanismen
  - Implementierung in Software nicht trivial!!!
    (Dekker-Algorithmus; E. W. Dijkstra; Peterson-Algorithmus)
  - **Semaphor**: Integer Variable, für die drei atomare Operationen möglich sind: initialisieren, increment, decrement
  - **Mutex / binärer Semaphor**: Werte 0 und 1
  - **Monitor**: Programmiersprachen Konzept, das Variablen und Zugriffsprozeduren als Datentyp kapselt. Immer nur ein Prozess hat Zugriff darauf.

Semaphor

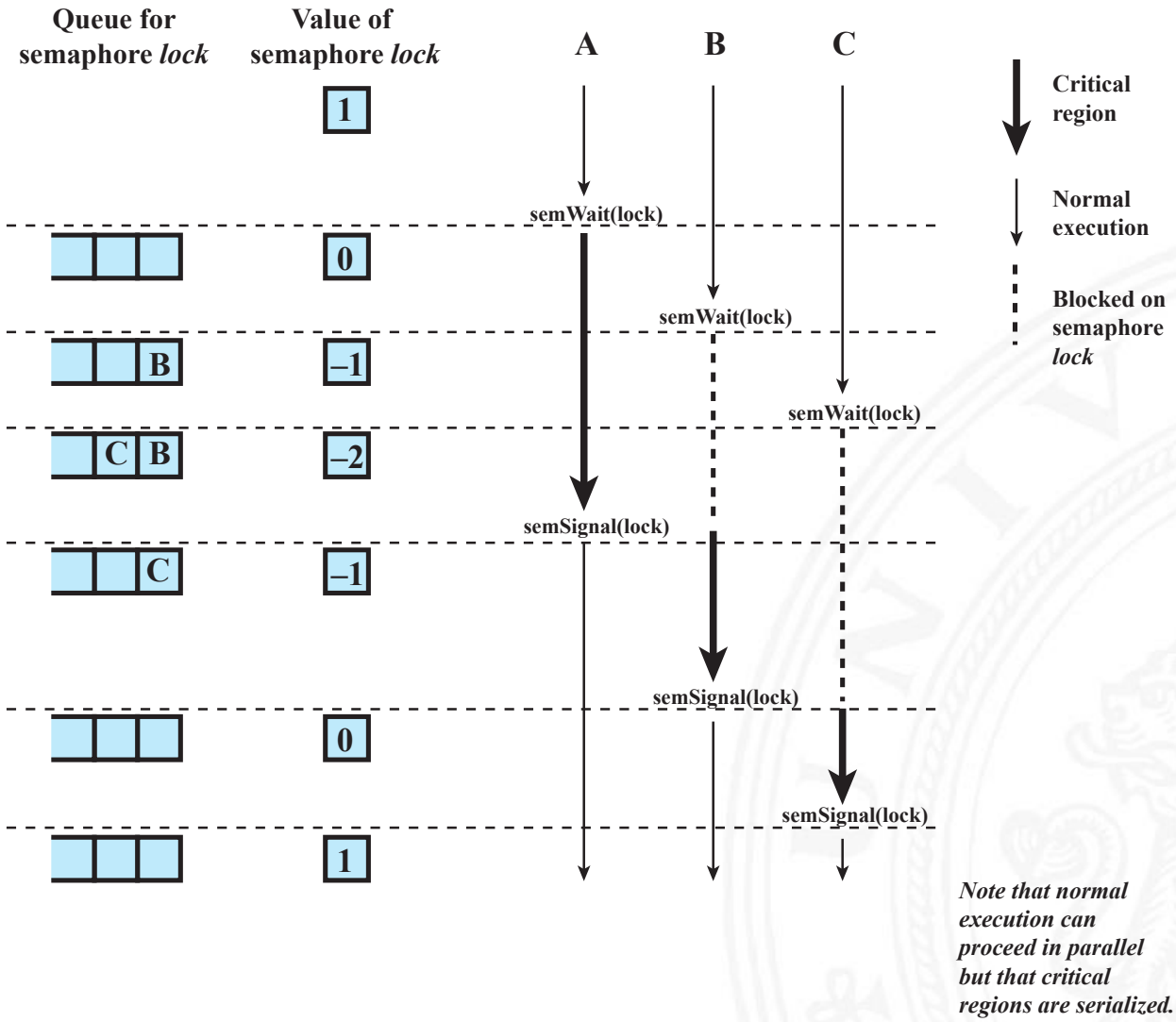
- Integer Variable, initialisiert mit Anz. gleichzeitiger Zugriffe
- atomare Operationen

  semWait decrement + aufrufender Prozess muss ggf. warten
  $\Rightarrow$ Beginn der Critical Section

  semSignal increment + ein wartender Prozess kann starten
  $\Rightarrow$ Ende der Critical Section
- starke Semaphor: am längsten wartender Prozess wird gestartet
  $\Rightarrow$ Queue für wartende Prozesse
- schwache Semaphor: beliebiger, wartender Prozess wird gewählt

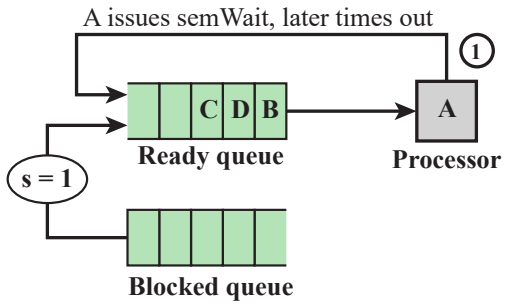
Semaphor (cont.)

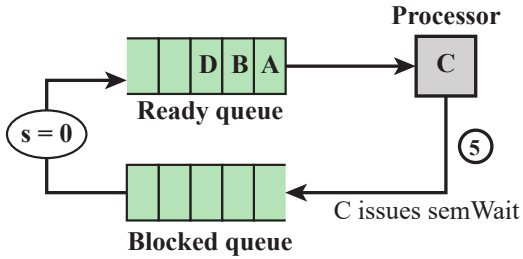
- Beispiele

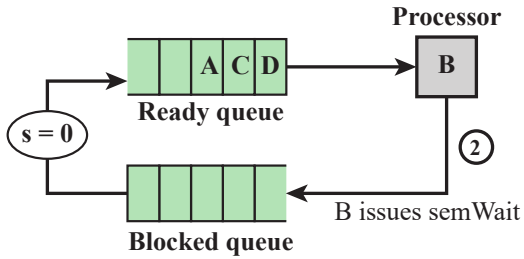


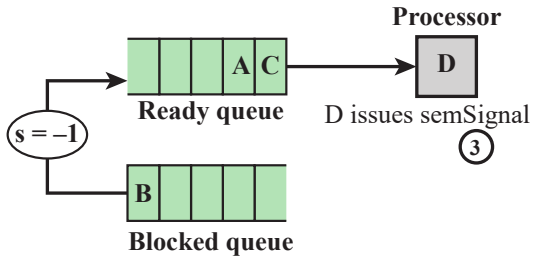
Note that normal execution can proceed in parallel but that critical regions are serialized.

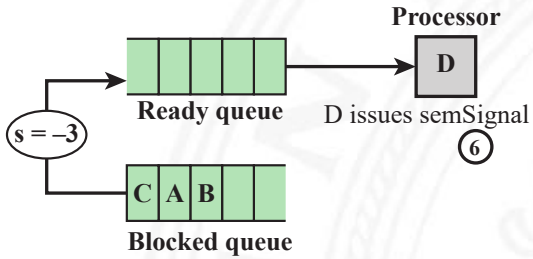
Semaphor (cont.)

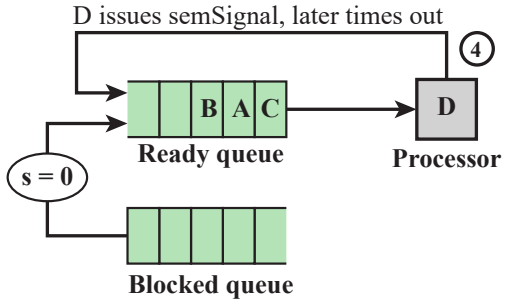


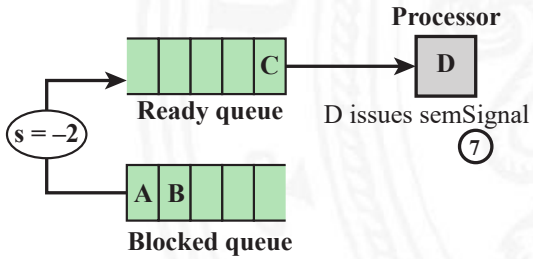








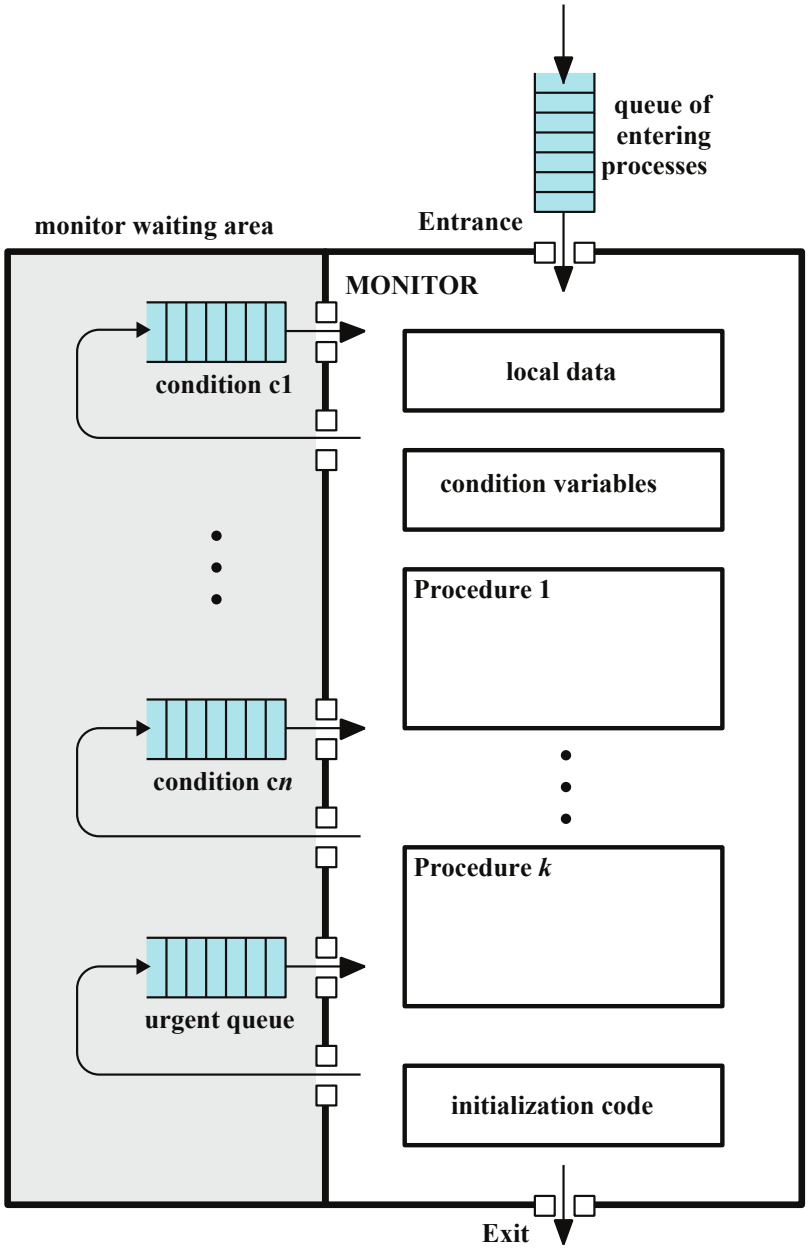




Monitor

- in Programmiersprachen: Concurrent Pascal, Ada, Modula ...
- in Java (`synchronized`), aber keine Bedingungsvariablen
- einfacher zu handhaben als Semaphor, gleiche Funktionalität
- nur ein Prozess darf im Monitor sein
- Synchronisation: Bedingungsvariablen (*condition variables*) und Funktionen `wait`, `signal`
- mehrere Warteschlangen: extern (Eintritt in Monitor), für jede Bedingungsvariable

Monitor (cont.)



Nachrichtenaustausch (*Message Passing*)

- geht auch für (räumlich) verteilte Systeme
- Kommunikationsfunktionen

  send (⟨*dst*⟩ , ⟨*data*⟩) sendet Daten
  blockierend / nicht blockierend

  receive (⟨*src*⟩ , ⟨*data*⟩) empfängt Daten
  blockierend / nicht blockierend / testend
- Varianten
  - block. `send` + block. `receive` ⇒ *Rendezvous*
    nicht block. `send` + block. `receive`
    nicht block. `send` + nicht block. `receive`
  - direkte Adressierung (s.o.) / indirekte Adressierung
    ⇒ 1:1, 1:n, m:1, m:n Beziehungen

```
process P is
  ...
  send(Q, A)
  ...
```

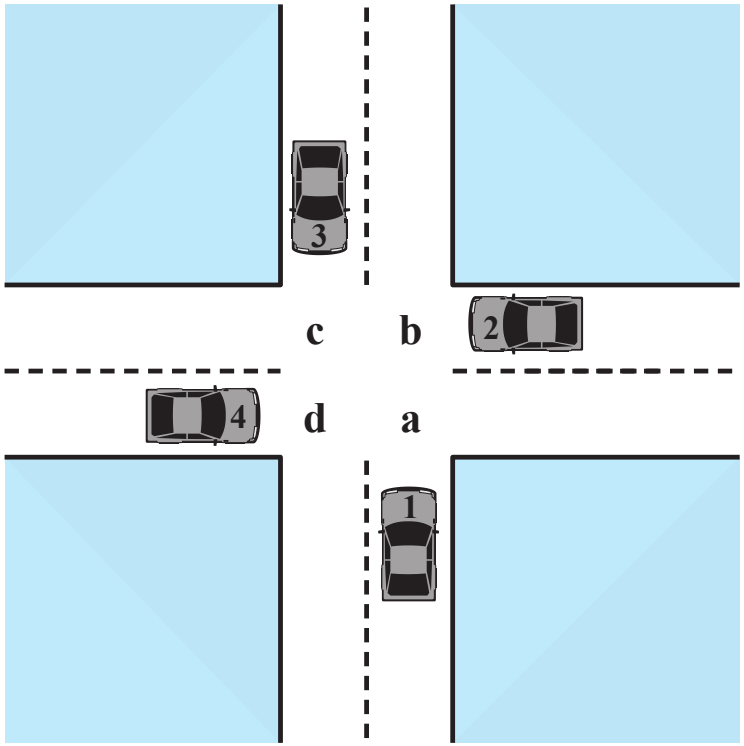
```
process Q is
  ...
  receive(P, A)
  ...
```

Deadlock

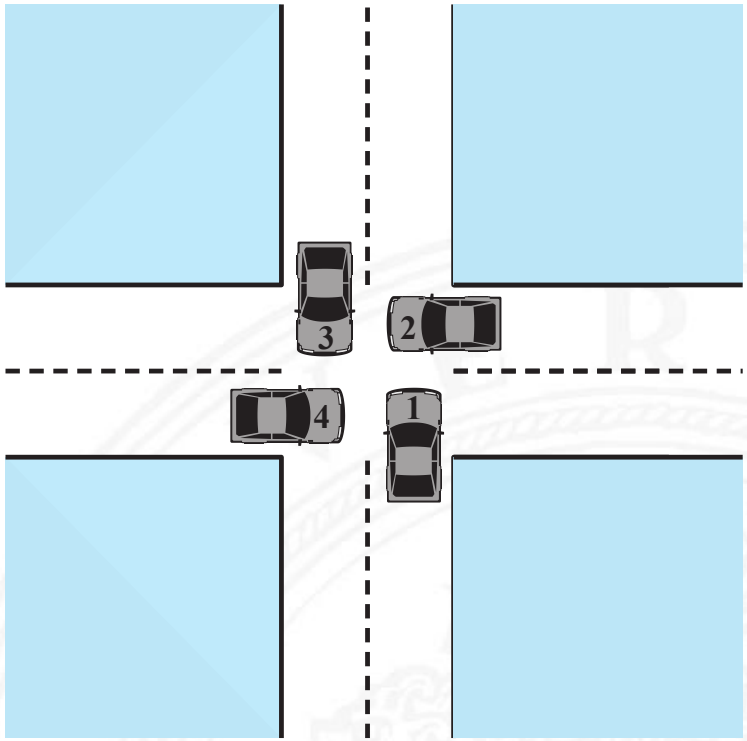
- Dauerhaftes Blockieren mehrerer Prozesse, die um Ressourcen konkurrieren, bzw. miteinander kommunizieren
- Deadlock, wenn jeder Prozess blockiert auf etwas wartet, was nur ein anderer blockierter Prozess anstoßen kann
- im Allgemeinen: keine effiziente Lösung

Deadlock (cont.)

- Beispiel: „rechts vor links“



(a) Deadlock possible



(b) Deadlock

- jedes Fahrzeug braucht 2 Ressourcen
  1: a,b 2: b,c 3: c,d 4: d,a

Deadlock (cont.)

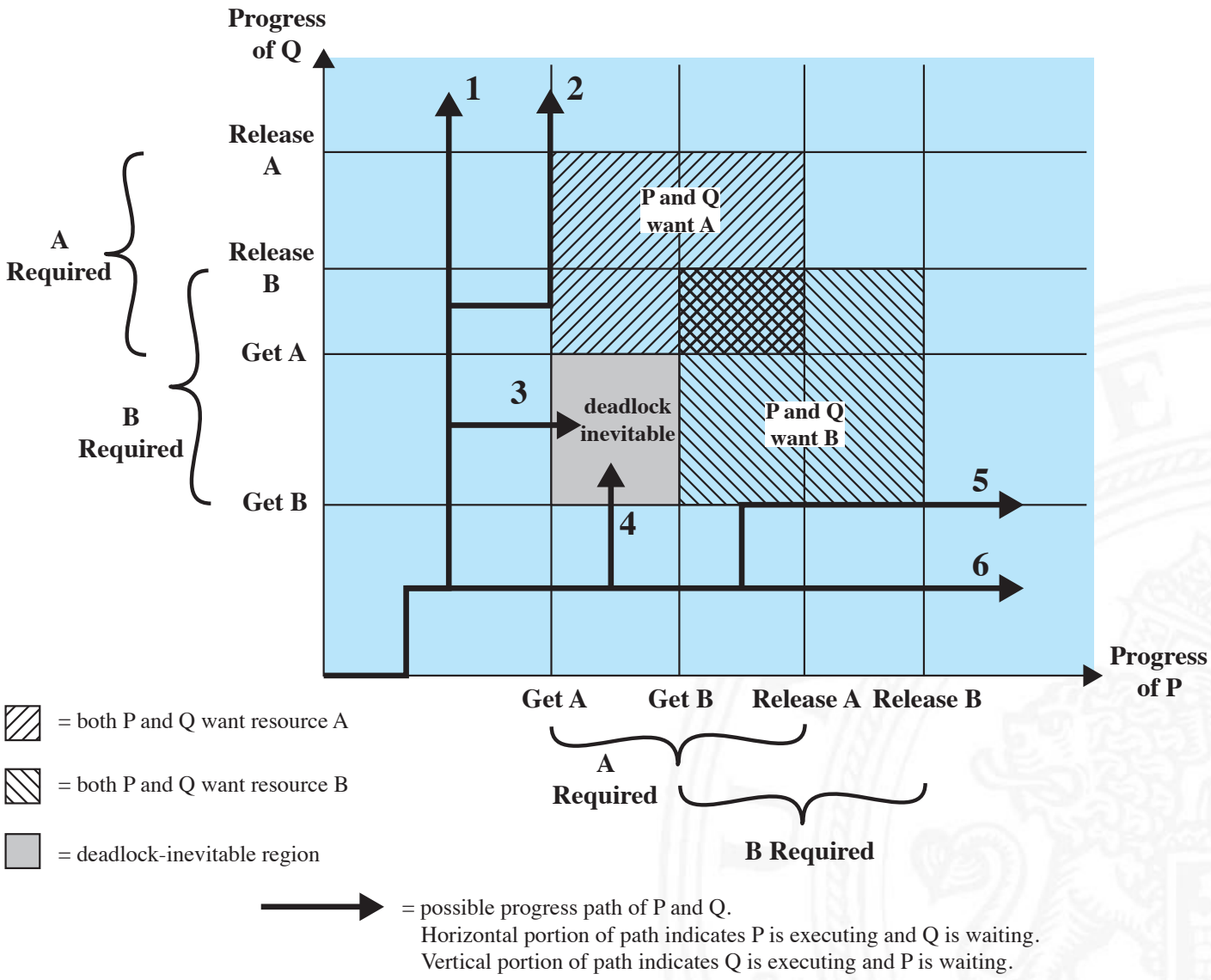
- Beispiel: zwei Programme, zwei Mutexe

```
process P is
  ...
  get(A)
  ...
  get(B)
  ...
  release(A)
  ...
  release(B)
  ...
```

```
process Q is
  ...
  get(B)
  ...
  get(A)
  ...
  release(B)
  ...
  release(A)
  ...
```

  - alternierender Ablauf der Prozesse
  - zweidimensional dargestellt

Deadlock (cont.)



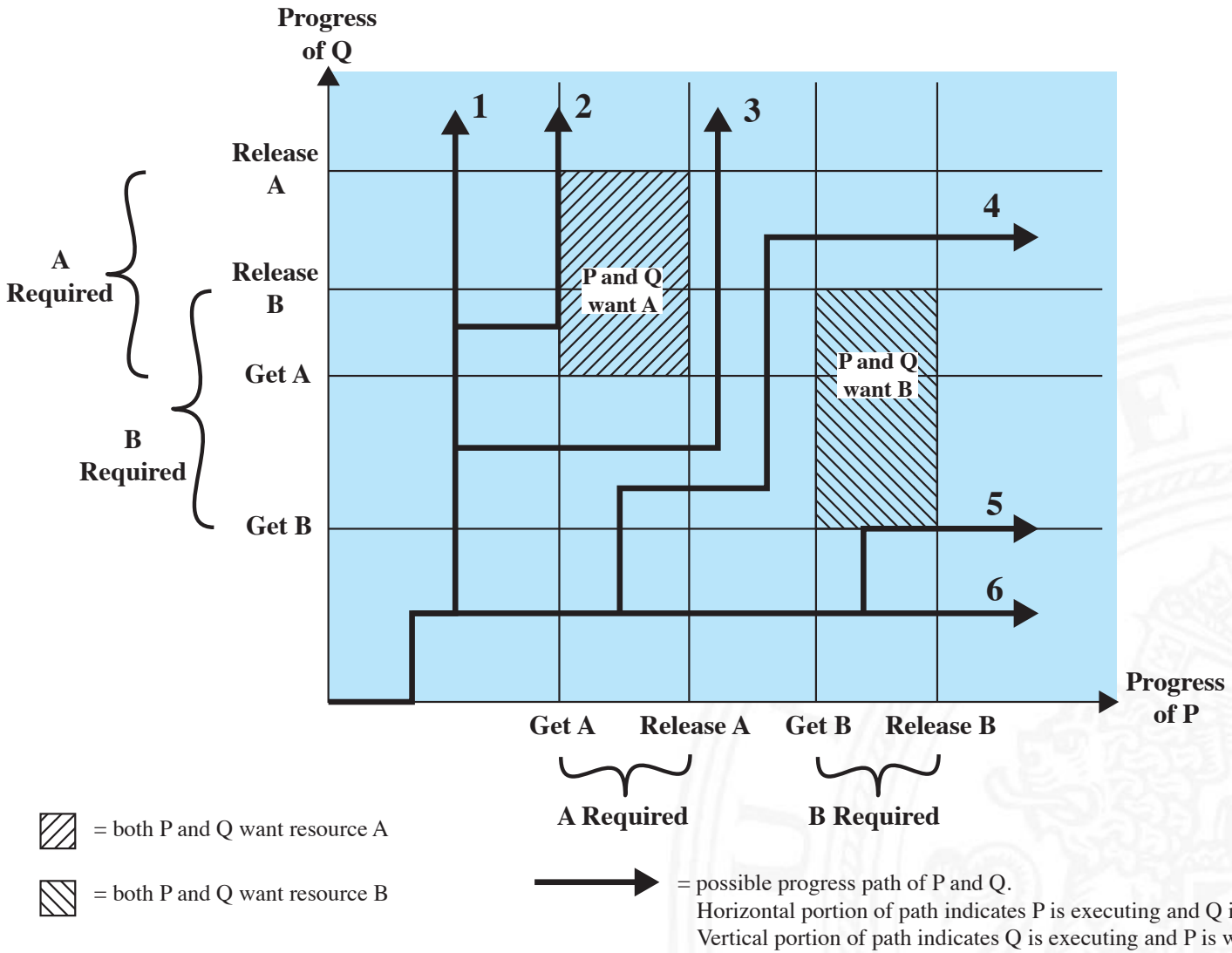
Deadlock (cont.)

- vorheriges Beispiel ohne Deadlock

```
process P is
  ...
  get(A)
  ...
  release(A)
  ...
  get(B)
  ...
  release(B)
  ...
```

```
process Q is
  ...
  get(B)
  ...
  get(A)
  ...
  release(B)
  ...
  release(A)
  ...
```

Deadlock (cont.)



Ressourcen

- wiederverwendbare (*reusable*) Ressource
  - wird bei Benutzung durch Prozess/Task nicht verbraucht
  - Prozessor, I/O-Kanal, Hauptspeicher, sekundärer Speicher, Datenstrukturen
- verbrauchbare (*consumable*) Ressource
  - wird erzeugt und bei Nutzung verbraucht
  - Interrupts, Signale, Nachrichten etc. (in FIFOs)
- Beispiel: wiederverwendbare Ressource = 200 KiB Speicher

```
process P is
  ...
  malloc(70 KiB)
  ...
  malloc(80 KiB)
  ...
```

```
process Q is
  ...
  malloc(80 KiB)
  ...
  malloc(60 KiB)
  ...
```

Ressourcen (cont.)

- Beispiel: verbrauchbare Ressource = Nachrichten,
  `receive` blockierend

```
process Q is
  ...
  receive(P, M1)
  ...
  send(M2, P)
  ...
```

```
process P is
  ...
  receive(Q, M3)
  ...
  send(M4, Q)
  ...
```

Bedingungen für Deadlock

1. Mutual Exclusion
   - ohne Mutual Exclusion kein Deadlock
     $\Rightarrow$ aber u.U. inkonsistente Daten
2. Hold-and-Wait
   - Prozess hat exklusiven Zugriff auf Ressource und fragt weitere an
3. No Preemption: Ressourcen können nicht entzogen werden
   - *Preemption* hier als zwangsweiser Entzug der Ressource
- Circular Wait: mehrere Prozesse/Tasks warten zyklisch aufeinander

$\Rightarrow$ 1. bis 3. notwendige Bedingungen
$+$ Circular Wait (zur Laufzeit) $=$ Deadlock

Maßnahmen gegen Deadlock

1. Deadlock verhindern
   - indirekt: drei notwendige Bedingungen für Deadlock
     - zu Mutual Exclusion: meist unverzichtbar
     - zu Hold-and-Wait: Prozess fordert gleichzeitig (atomar) alle Ressourcen/Locks an
     - zu No-Preemption: Test auf Ressource, wenn nicht verfügbar: kein Warten, sondern Rückgabe; Betriebssystem „entzieht“ Ressource
   - direkt: *Circular Wait* nicht zulassen
     - Einführen einer Ordnung/Reihenfolge für alle Ressourcen
     - muss in allen Prozessen eingehalten werden
2. Deadlock vermeiden
   - Ressource nicht zuteilen, wenn Deadlock möglich
     $\Rightarrow$ algorithmisch lösbar (*Banker's algorithm*)
   - Prozess nicht starten, der zu Deadlock führen kann

   \+ weniger Restriktiv als „Deadlock verhindern“

   \+ kein Rollback nötig, wie in „Deadlock Erkennung“

Maßnahmen gegen Deadlock (cont.)

3. Deadlock Erkennung
   - ▸ Periodischer Test auf Deadlock und ggf. (partielles) Rücksetzen
   - \+ 1. und 2. schränken Prozesse ein;
     gegenteiliger Ansatz: Zugriff erlauben
   - \+ einfacher Algorithmus
   - – Overhead durch periodische Checks
   - – „Zurücksetzen“ der Prozesse nicht trivial; Checkpoints

⇒ Einteilung der Ressourcen in „Klassen“ mit verschiedenen „Arten/Typen“ von Deadlocks und Einsatz unterschiedlicher Deadlock Strategien

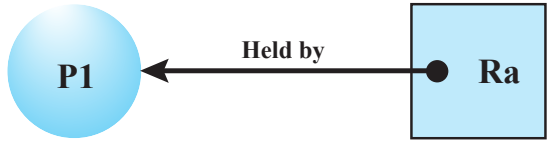
Maßnahmen gegen Deadlock (cont.)

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

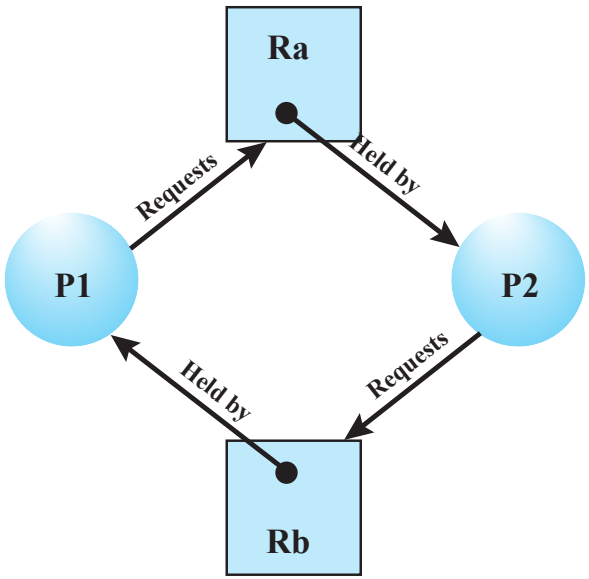
Maßnahmen gegen Deadlock (cont.)

- Graph zum Ressourcenbesitz

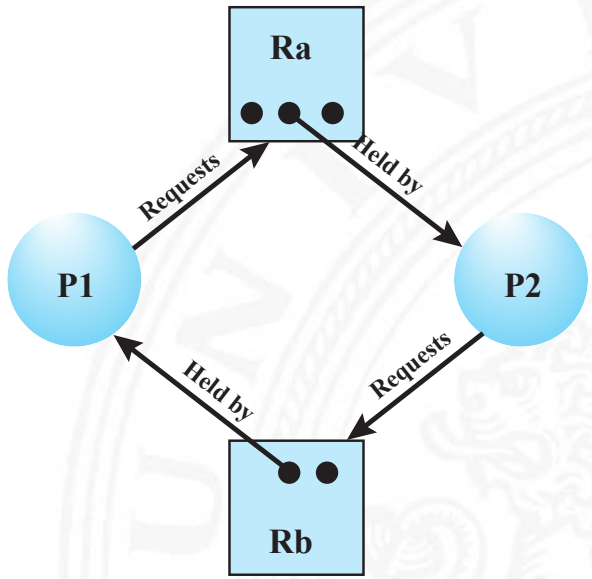
(a) Resouce is requested



(b) Resource is held

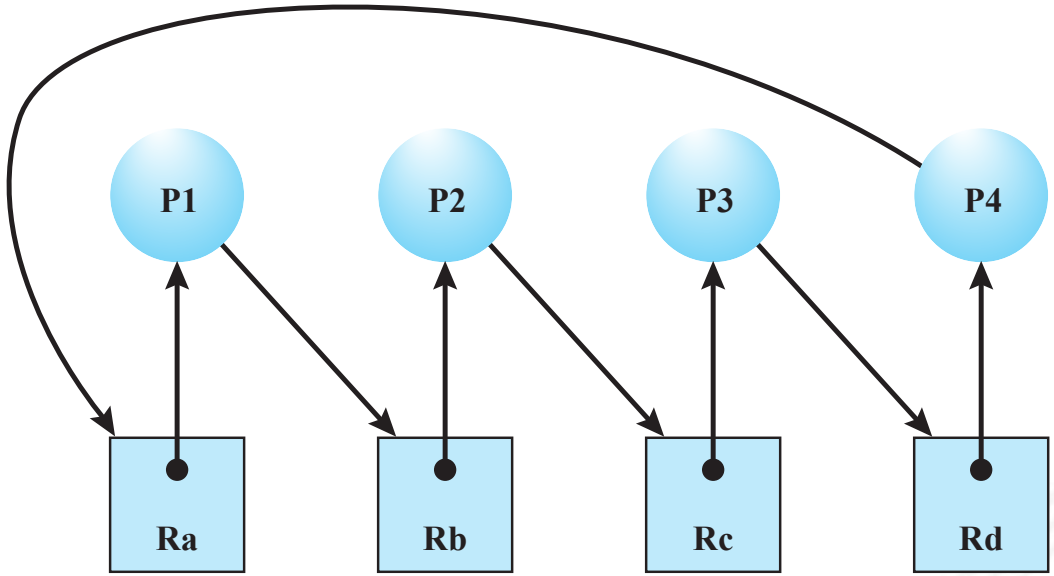


(c) Circular wait



(d) No deadlock

Maßnahmen gegen Deadlock (cont.)



Kreuzung: „rechts vor links“

- Algorithmen zu Deadlock Vermeidung oder Erkennung nutzen daraus abgeleitete Matrizen zu: Ressourcenanfragen und -besitz

Scheduling und Ressourcenmanagement

- Hauptfunktionalität von Betriebssystemen: Ressourcenmanagement
- wichtig dabei Effizienz
  Antwortverhalten (*Responsiveness*)
  Fairness

$\Rightarrow$ Scheduling / Ablaufplanung

- betrifft mehrere Ressourcen: Prozessor, Speicher, I/O Geräte

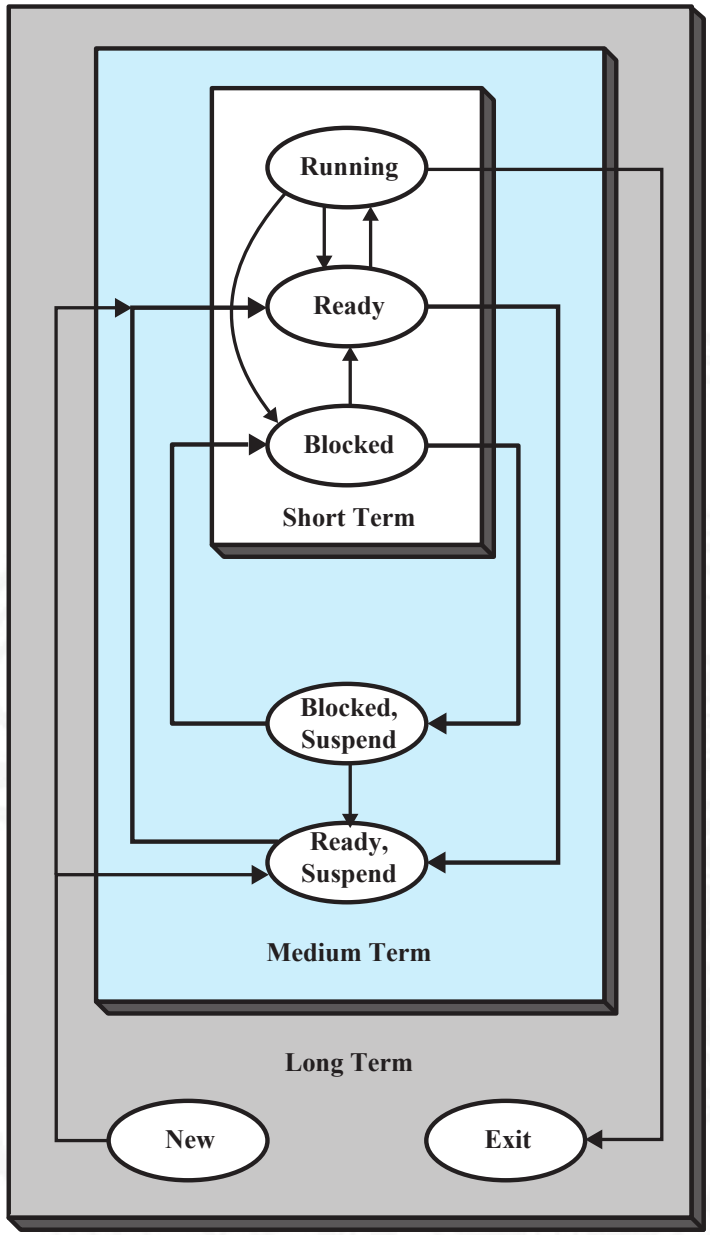
- **Long-term**: Welche Prozesse sollen in Menge der Jobs?
  - beeinflusst Multiprogramming: Anzahl der Jobs auf Computer
  - Strategien: First-come, First-served; nach Prioritäten; Ressourcen
- **Medium-term**: Welche Prozesse sollen in Hauptspeicher?
  - Teil der Speicherverwaltung $\rightarrow$ Abschnitt *15.6 Speicherverwaltung*
  - Auswirkungen auf Multiprogramming: Prozesse nicht lauffähig, wenn nicht im Speicher

Scheduling und Ressourcenmanagement (cont.)

- **Short-term**: Welcher Prozess wird durch CPU ausgeführt?
  - Dispatcher: wird häufig aufgerufen
  - Start durch: Interrupts, System-Calls, Signale (Semaphor, Mutex)
  - quantitative Kriterien
    Benutzer: Antwortverhalten (*Responsiveness*)
    System: Prozessornutzung, Job-Durchsatz, Ressourcenauslastung
  - qualitative Kriterien Fairness, Deadlockfrei, keine Starvation, Vorhersagbarkeit, Echtzeitfähigkeit etc.
- **I/O Scheduler**: Welche I/O-Anfrage geht an Gerät?
  - mehrere
  - gerätespezifisch

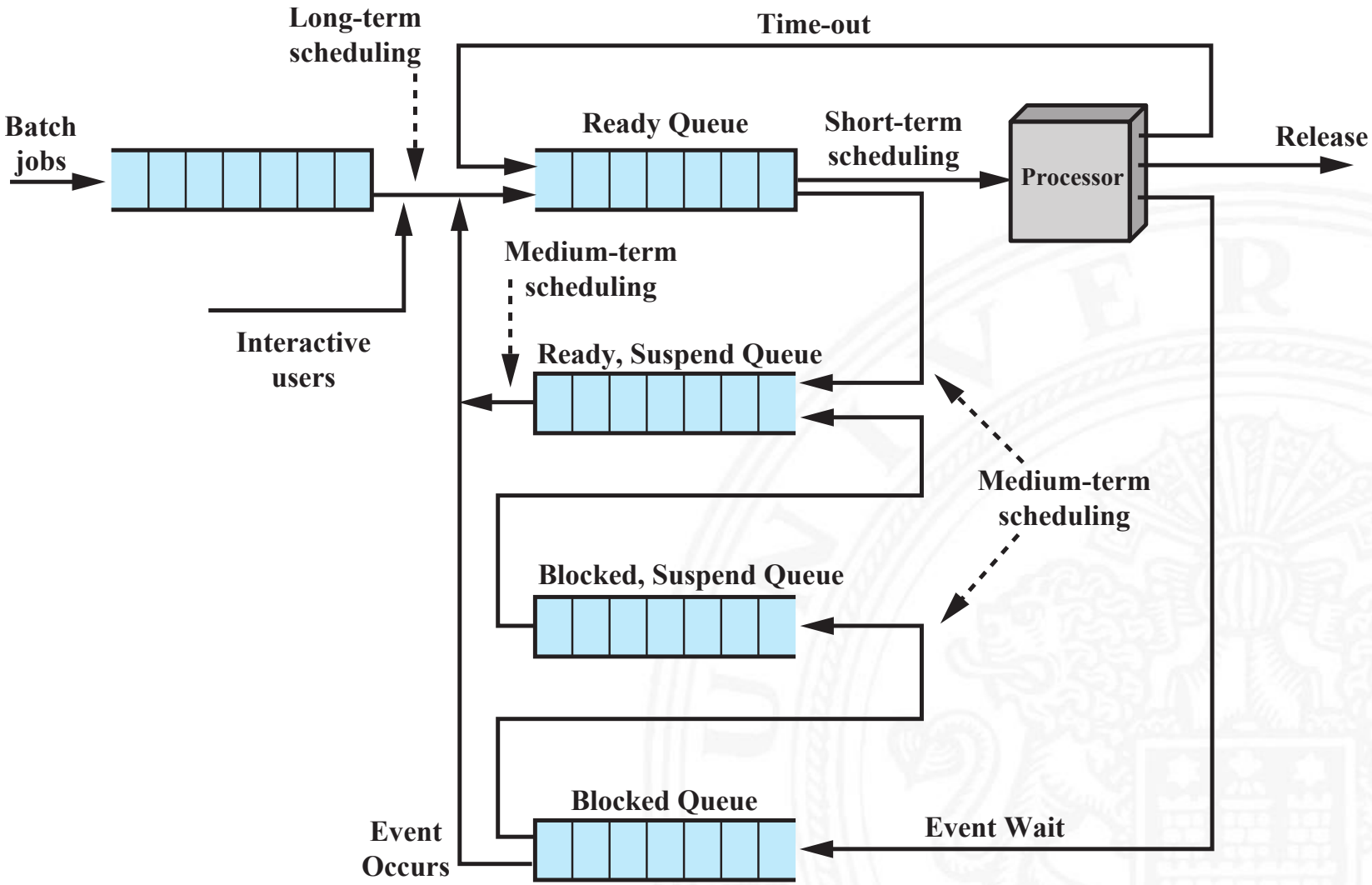
Scheduling und Ressourcenmanagement (cont.)

- verschiedene Zustände im Prozessmodell (vergl. Folie 1177)



Scheduling und Ressourcenmanagement (cont.)

- Scheduling Queues



Short-term Scheduling

- Aktivierung
  - neuer Prozess in *Ready* Queue
  - Interrupt (bringt Prozess in *Ready*)
  - periodisch
- Funktion zur Auswahl der Prozesse abhängig von
  - Prioritäten
  - Ressourcenbedarf
  - Prozessabarbeitung

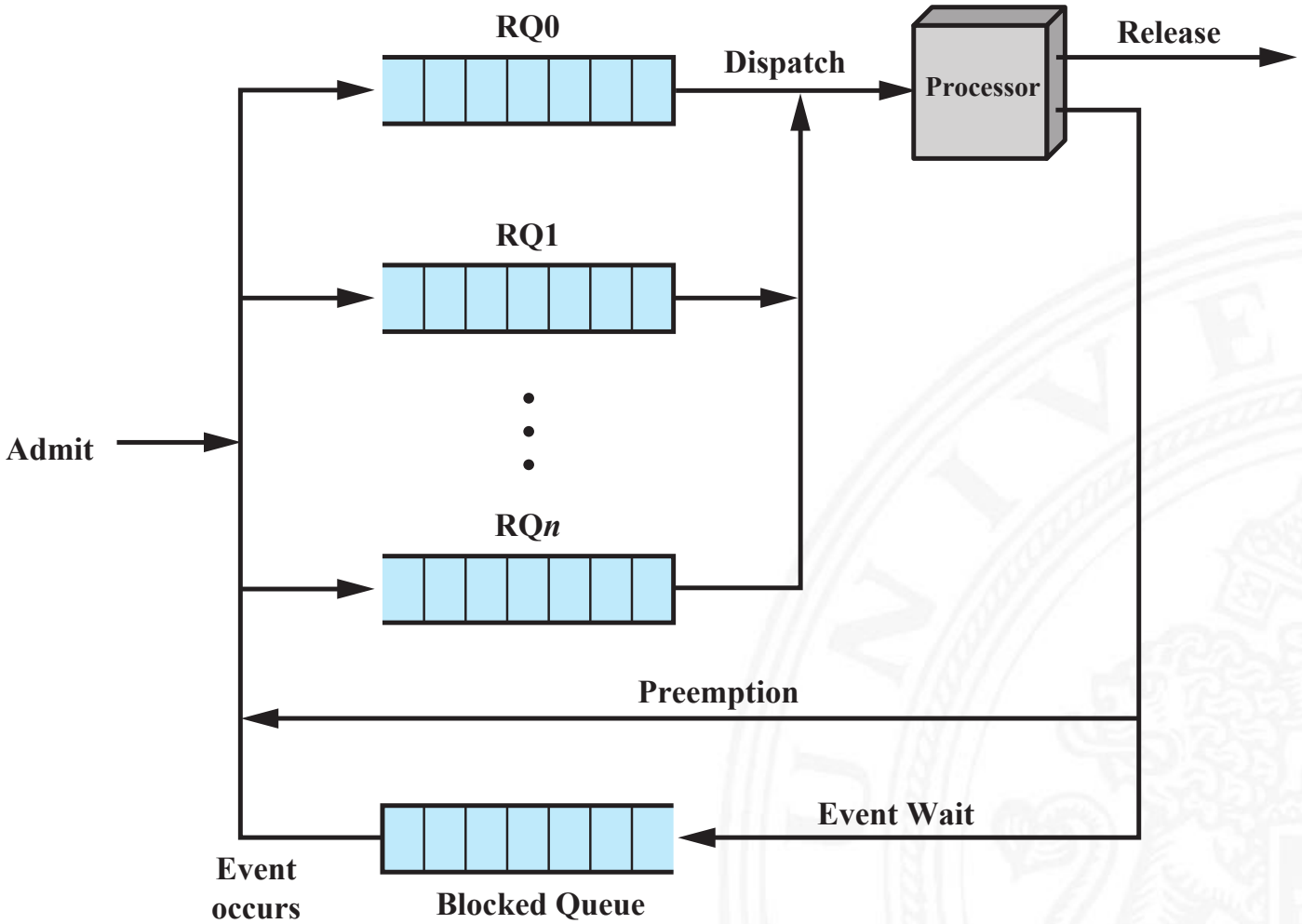
    w : bisherige Wartezeit

    e : bisherige Ausführungszeit (*Execution time*)

    s : gesamte Ausführungszeit (*Service time*)
- Preemption: Unterbrechung von Jobs?
  - **ohne**: gestarteter Prozess läuft bis Ende oder I/O waiting
  - **mit**: Prozess wird unterbrochen und in *Ready*-Queue eingereiht

Short-term Scheduling (cont.)

- Short-term Queues mit Prioritäten



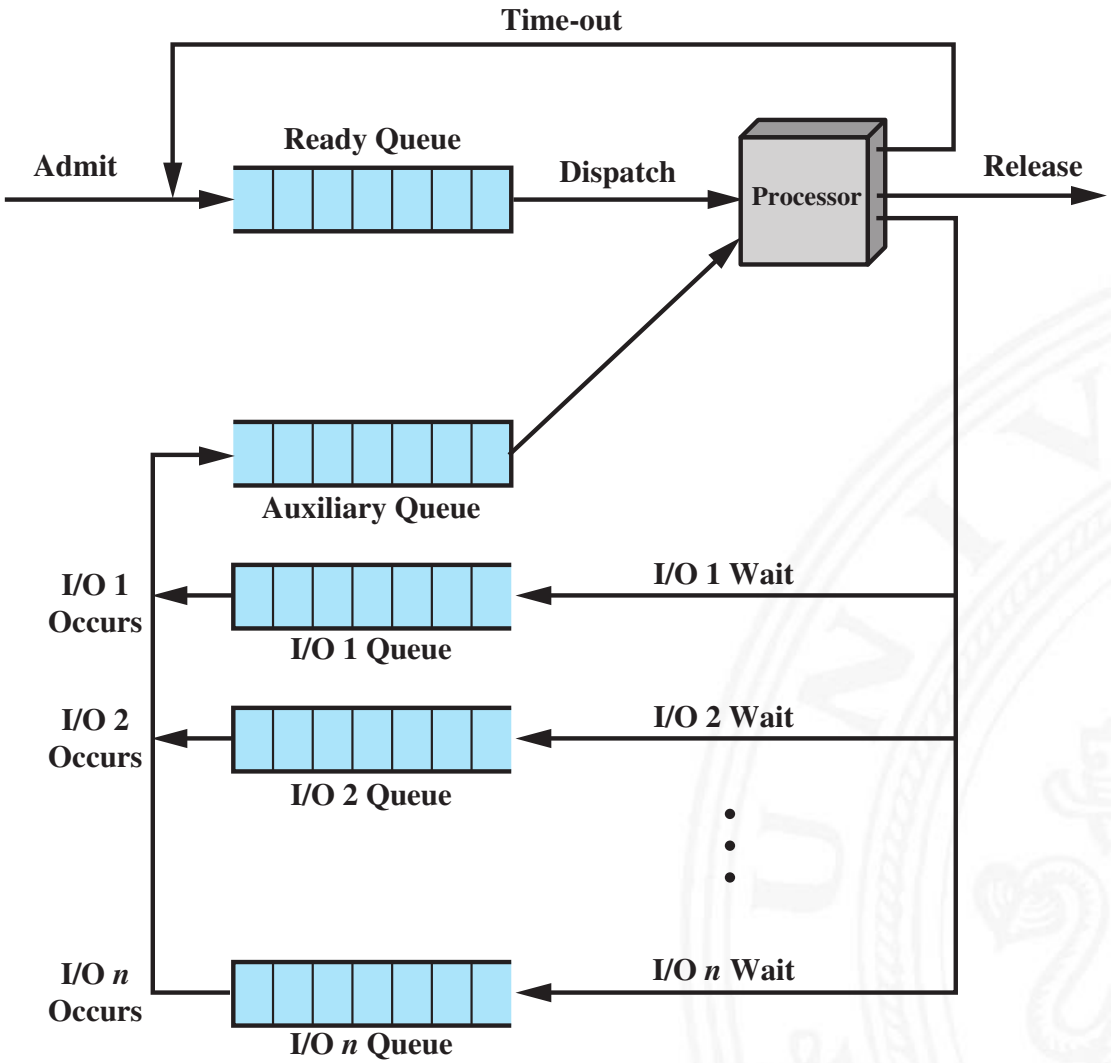
Scheduling Algorithmen

- **First-come, First-served** (FCFS)
  - Non-Preemptive
  - einfache Implementation: FIFO
  - bevorzugt länger laufende Prozesse
  - bevorzugt rechenlastige Prozesse gegenüber I/O-lastigen

- **Round-Robin** (RR) – Zeitscheiben Verfahren
  - Preemptive
  - Länge des Zeitslots?
  - Gut für Transaction Processing (Durchsatz, Fairness)

Scheduling Algorithmen (cont.)

- Round Robin Queues

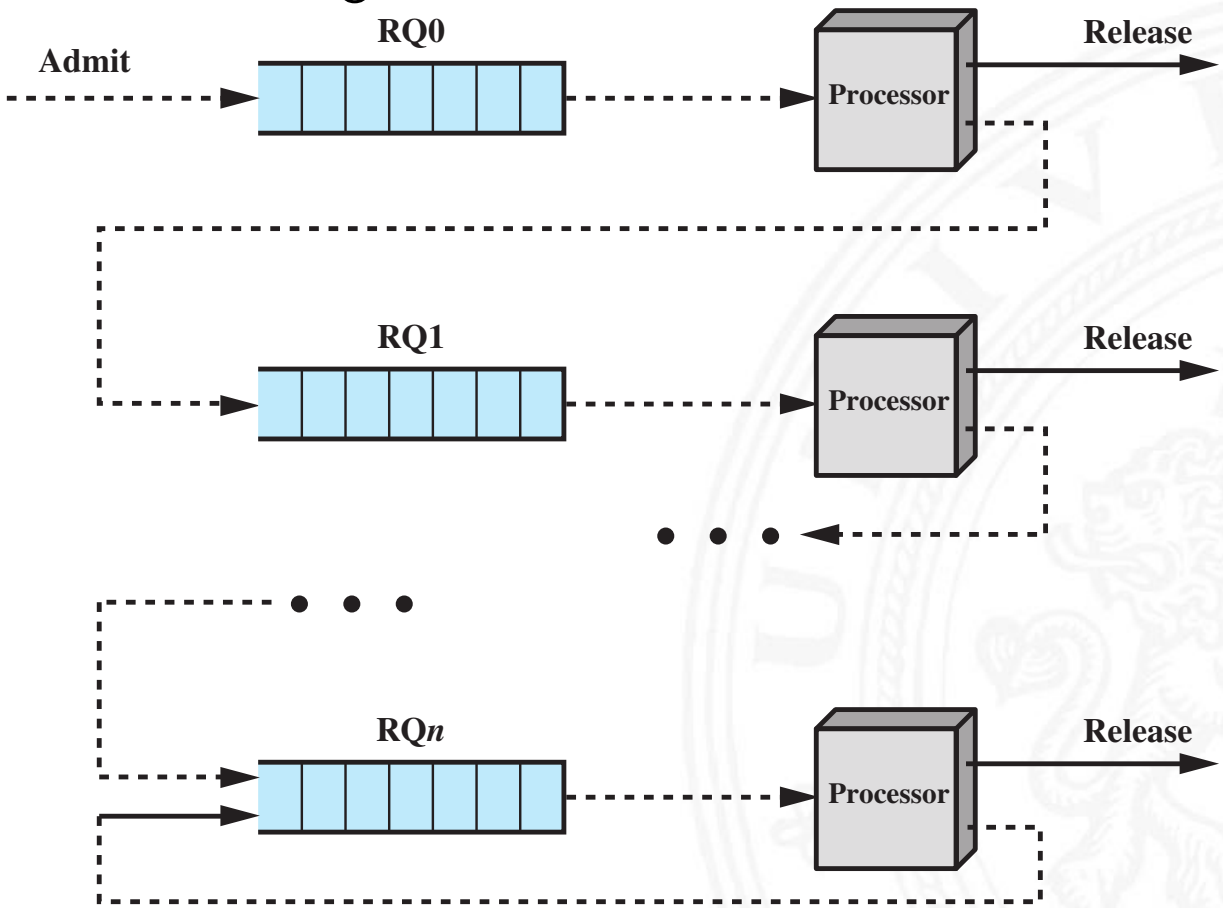


Scheduling Algorithmen (cont.)

- **Shortest Process Next** (SPN)
  - Non-Preemptive
  - Prozess mit kürzester Ausführungszeit startet
  - Ausführungszeit s schätzen?
  - Starvation für lang laufende Prozesse möglich
  - Interaktive Prozesse: Durchschnittsbildung der letzten Aktivitäten, ggf. „exponentielles Altern“ (= Wichtung älterer Werte nimmt ab)
- **Shortest Remaining Time** (SRT)
  - Preemptive Version von SPN
  - Prozess mit kürzester Restzeit startet
  - Ausführungszeit s schätzen?
  - Starvation für lang laufende Prozesse möglich
- **Highest Response Ratio Next** (HRRN)
  - Non-Preemptive
  - Response Ratio: $r = \frac{(w+s)}{s}$
  - Prozess mit größtem r startet
  - Fair, auch für lang laufende Prozesse wegen w

Scheduling Algorithmen (cont.)

- **Feedback Scheduling**
  - Preemptive
  - Round-Robin mit mehreren Queues: Start in Hochpriorisierter, sukzessiver Abstieg in weniger priorisierte Queues
  - relativer Vorzug kurz laufender Prozesse



Scheduling Algorithmen (cont.)

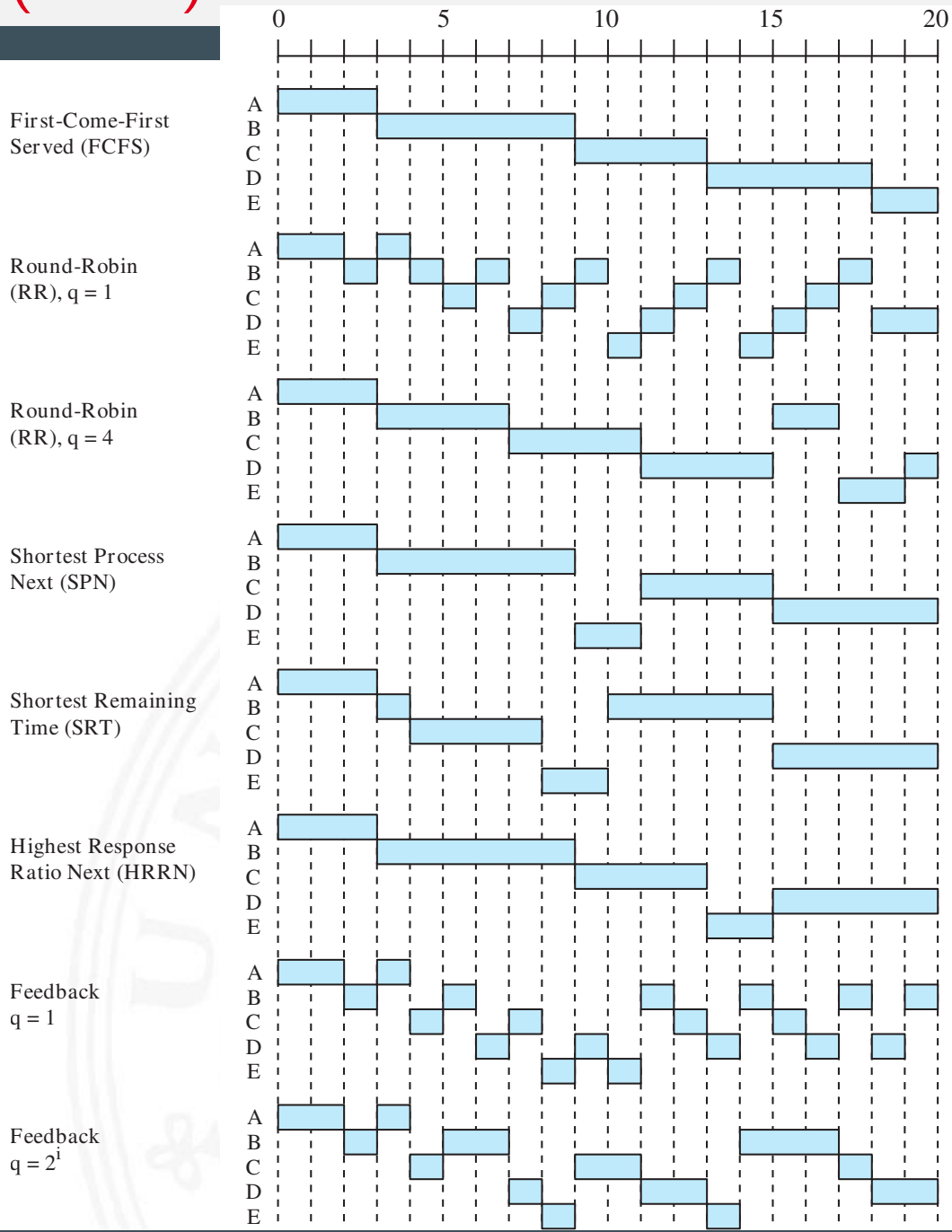
- ... viele weitere Algorithmen, es fehlen
  - Gruppierung von Prozessen (*Process Groups*)
  - periodische Tasks
  - Echtzeitsysteme: Prozesse haben eine Deadline!

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]		(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Scheduling Algorithmen (cont.)

- Beispiel

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Scheduling Algorithmen (cont.)

- Performanz des Scheduling? – Bewertung?
  - Rechnersystem: Desktop, Kontroll-/Steuerungsrechner, Server (DB, Web-Dienste . . .), HPC (Großrechner, Supercomputer)
  - Anwendungsszenarien: welche, wie viele Prozesse?
  - I/O: welche Geräte, wie schnell?
  - Aufwand und Effizienz des Scheduling
  - Aufwand für Kontextwechsel

$\Rightarrow$ Modellierung über Warteschlangentheorie, stochastische Prozesse

Multiprozessor Scheduling

- unterschiedliche Kopplungen:
  Cluster (schwach) ... Symmetrical Multiprocessing (stark)
- Scheduling auch für Spezial- (Co-) Prozessoren
- Granularität, wichtig für Synchronisation
  - Parallelität inhärent in Thread
  - Parallelität innerhalb eines Prozesses / Multithreading
  - kommunizierende Prozesse (Shared-Memory)
  - verteilte Prozesse in Netzwerk (Message-Passing)
  - Menge unabhängiger Prozesse

Multiprozessor Scheduling (cont.)

- Zuordnung von Prozessen zu Prozessoren
  - dynamisch: Menge von Prozessen $\rightarrow$ Pool von Prozessoren
  - statisch: Prozess wird Prozessor zugeordnet
    - \+ Scheduling einfacher
    - \+ Group-Scheduling
    - – ggf. Prozessorleerlauf (dann Load-Balancing)
- Architekturen (Wo läuft der Scheduler?)
  - Peer Systeme / verteiltes Scheduling
  - Master-Slave
    - \+ einfach zu implementieren
    - \+ weniger Overhead
    - – Point of Failure
    - – Bottleneck

Aufgaben des Betriebssystems

- Trennung der Prozesse voneinander
- Verwaltung von dynamischem Speicher
- Unterstützung modularer Programme
- Schutz: Integrität der Daten
- Schutz: Durchsetzung von Zugriffsrechten
- Realisierung von Langzeitspeicher (Dateisysteme)

Virtueller Speicher

- sekundärer Speicher (HDD, SSD) ist Teil des Speichers
- logische Adressen in Programmen sind unabhängig von
  - dem physikalisch vorhandenem Speicher
  - physikalischen Adressen (Adressen zur Laufzeit)
- mehrere Prozesse, Benutzerjobs. . . sind gleichzeitig im Speicher

$\Rightarrow$ Adressen im Code werden zu **virtuellen Adressen**:
logische Adresse + Adressübersetzung

- Adressübersetzung entspricht Funktion
- meist als Tabelle realisiert

Virtueller Speicher (cont.)

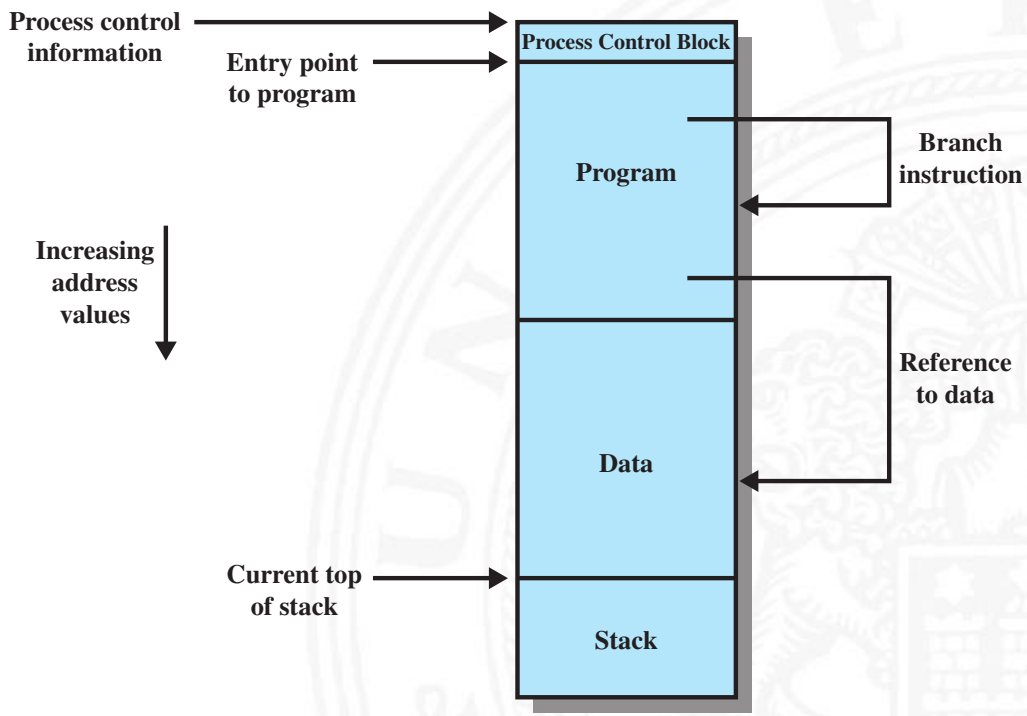
- **Frame / Kachel**
  - Block fester Größe im Hauptspeicher
- **Page / Seite**
  - Block fester Größe im sekundären Speicher (HDD, SSD),
  - kann temporär in Frame (im Hauptspeicher) kopiert werden

  $\Rightarrow$ Paging
- **Segment**
  - Block variabler Größe im sekundären Speicher, kann temporär
  - in passenden Bereich im Hauptspeicher kopiert werden

  $\Rightarrow$ Segmentierung
  - in Seiten unterteilt werden, die jeweils kopiert werden

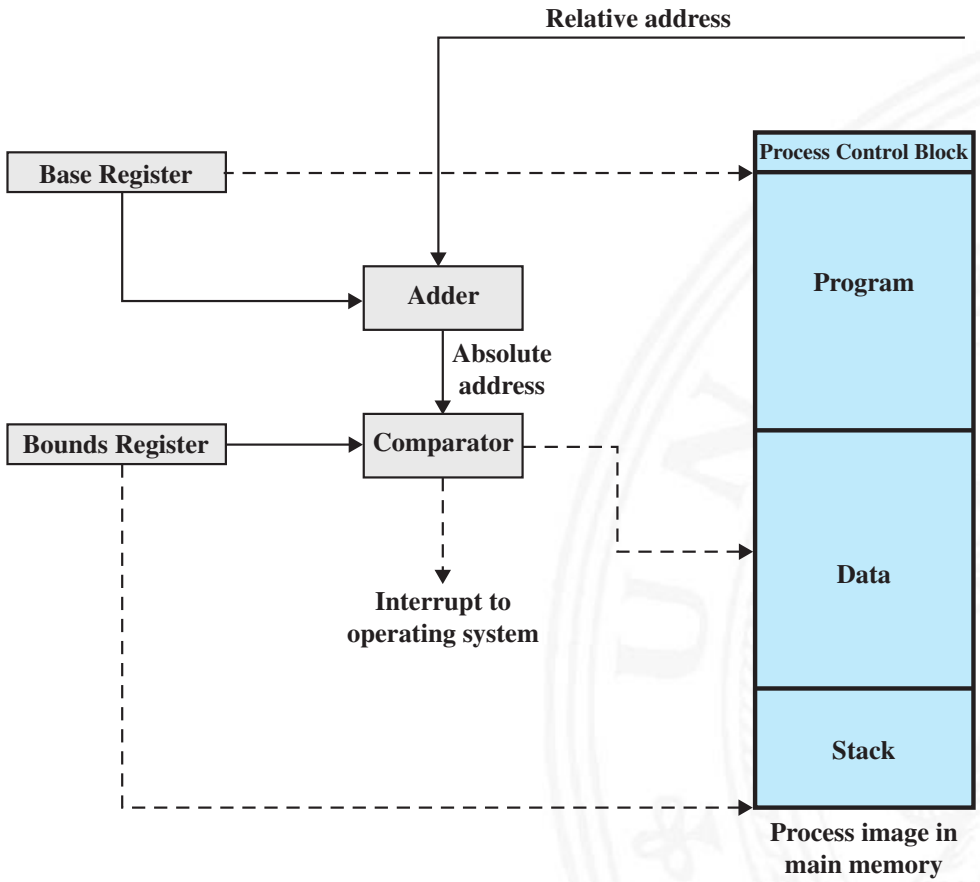
  $\Rightarrow$ Segmentierung + Paging

- Adressumsetzung / *Relocation*
  - **logische Adressen**: Adressen in (Assembler-) Programm
  - **relative Adressen**:
    relativ zu Bezug (Basisadresse), i.d.R. logische Adressen
  - **physikalische / absolute Adressen**:
    Adressen des Hauptspeichers



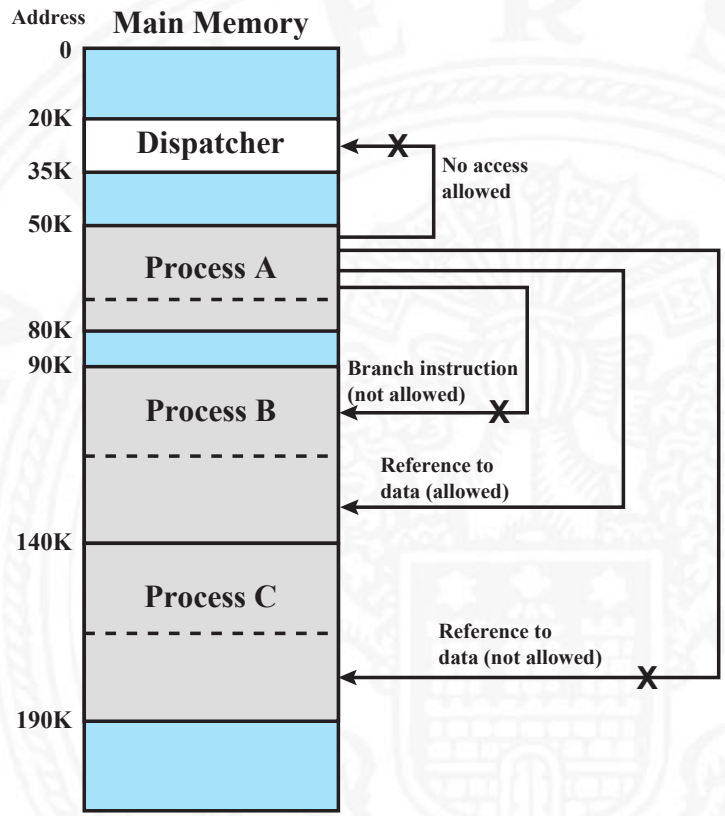
Memory Management (cont.)

- Swapping: Prozess auslagern, u.U. an anderer Stelle in Hauptspeicher fortsetzen
- Abbildung: logische $\rightarrow$ physikalische Adressen



Memory Management (cont.)

- Schutz, hier Zugriff
  - Relocation verwaltet auch Zugriffsrechte auf Speicherbereiche
  - bei Adressumrechnung in Segment- und Seiten-Tabellen
- gemeinsam genutzte Code- und Datenbereiche
  - Relocation ermöglicht auch Einbindung von Speicherbereichen in Adressraum mehrerer Prozesse
  - gemeinsam genutzte Segmente



Memory Management (cont.)

- Trennung von logischer und physischer Organisation
  1. Segmentierung
  2. Paging / Seitenadressierung
  3. Kombination von: Segmentierung und Paging
- bei Programmlauf: nicht alle Segmente/Seiten des Prozesses müssen gleichzeitig in Hauptspeicher sein
  - **Resident Set**: Adressbereiche (Text, Data) in Hauptspeicher
  - **Working Set**: im Programm gerade genutzt (Lokalität)
- während Programmlauf
  1. Interrupt, wenn Adresse des Prozesses nicht in Hauptspeicher
  2. Prozess wechselt in „blocked“
  3. Datentransfer: sekundärer Speicher $\rightarrow$ Hauptspeicher durch DMA (im Hintergrund)
  4. Dispatcher lässt anderer Prozess rechnen
  5. Interrupt, wenn Datentransfer fertig
  6. Prozess wechselt in „ready“

Memory Management (cont.)

+ kleinerer Adressraum je Prozess $\Rightarrow$ mehr Prozesse im System
  $\Rightarrow$ bessere CPU-Auslastung
+ Prozesse können größer sein als gesamter Hauptspeicher
+ für Programmierer transparent,
  von Betriebssystem und HW verwaltet

Memory Management (cont.)

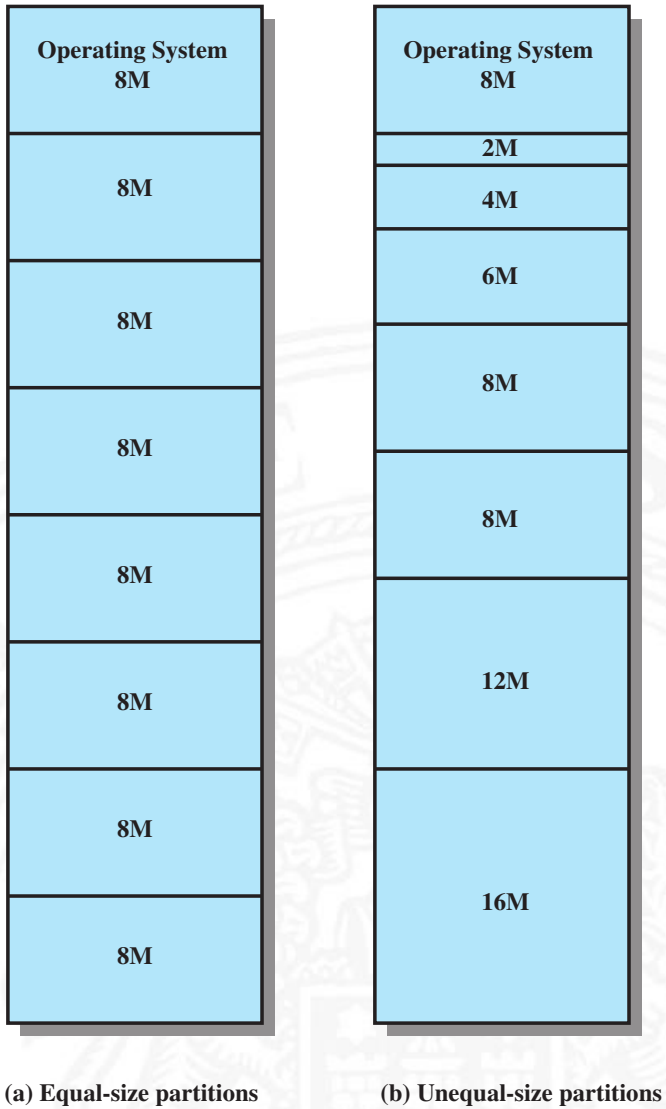
- Memory Management

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Partitionierung

- Speicher in feste Bereiche unterteilt
  - Anzahl, Größe der Speicherbereiche?
  - Programme zu groß für Partition
    ⇒ Overlay-Techniken
  - interne Fragmentierung:
    ungenutzter Speicher in Partition

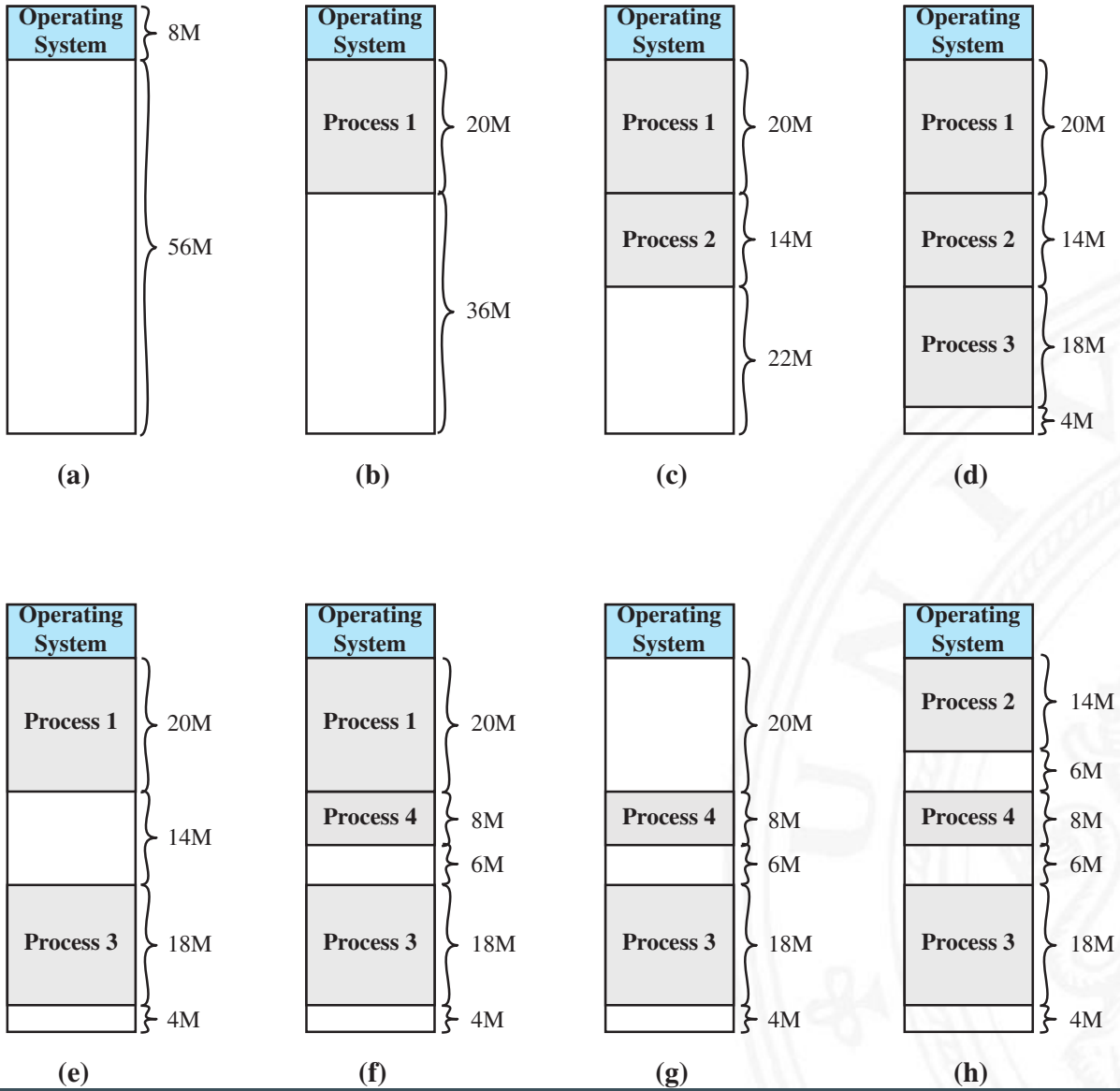
⇒ schlechte Speicherausnutzung
⇒ obsolet



(a) Equal-size partitions (b) Unequal-size partitions

Partitionierung (cont.)

- dynamische Partitionierung

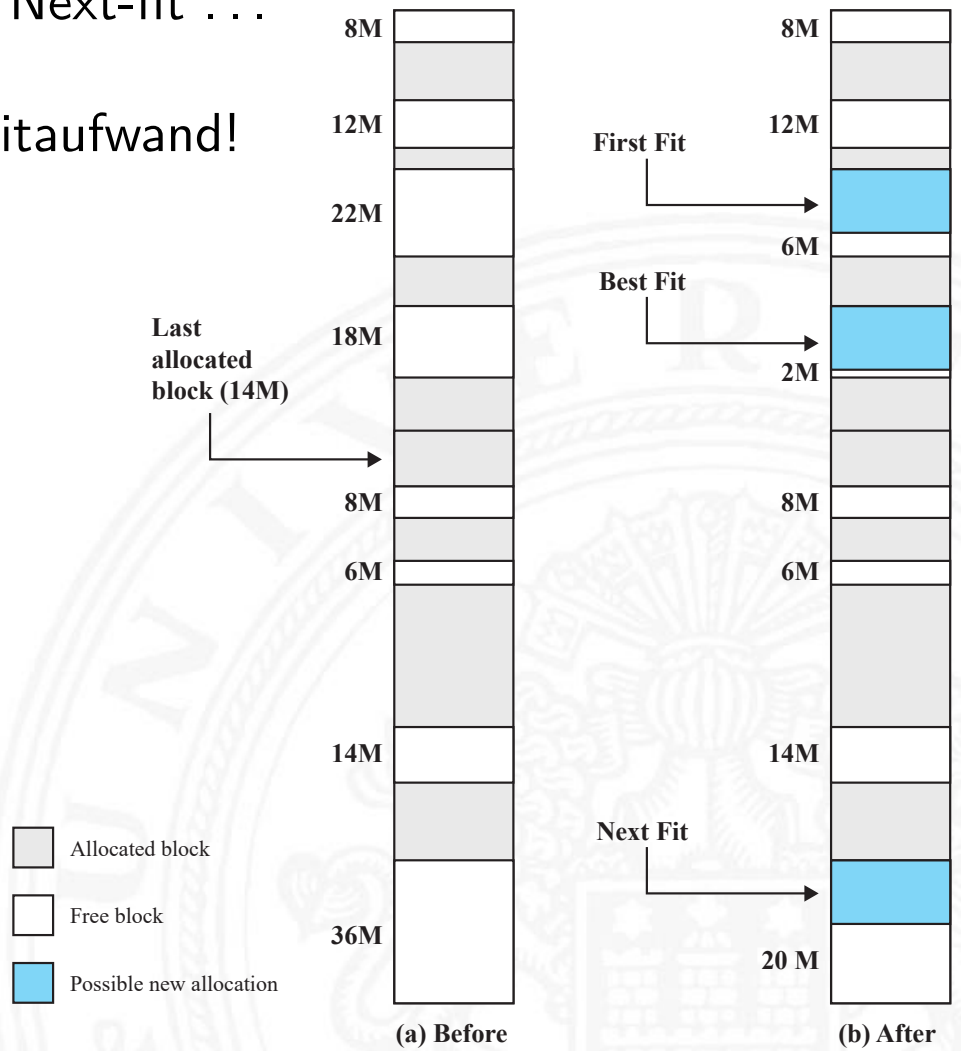


Partitionierung (cont.)

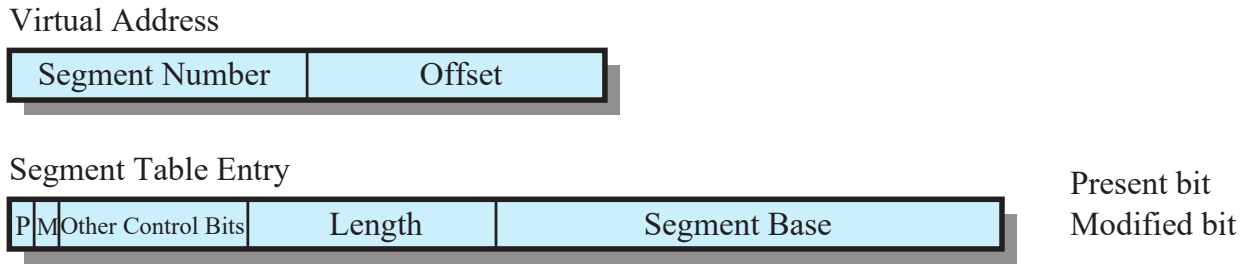
- externe Fragmentierung durch Ein- und Auslagern von Prozessen
- Platzierung: Best-fit, First-fit, Next-fit ...

– Kompaktierung notwendig, Zeitaufwand!

⇒ obsolet



Segmentierung



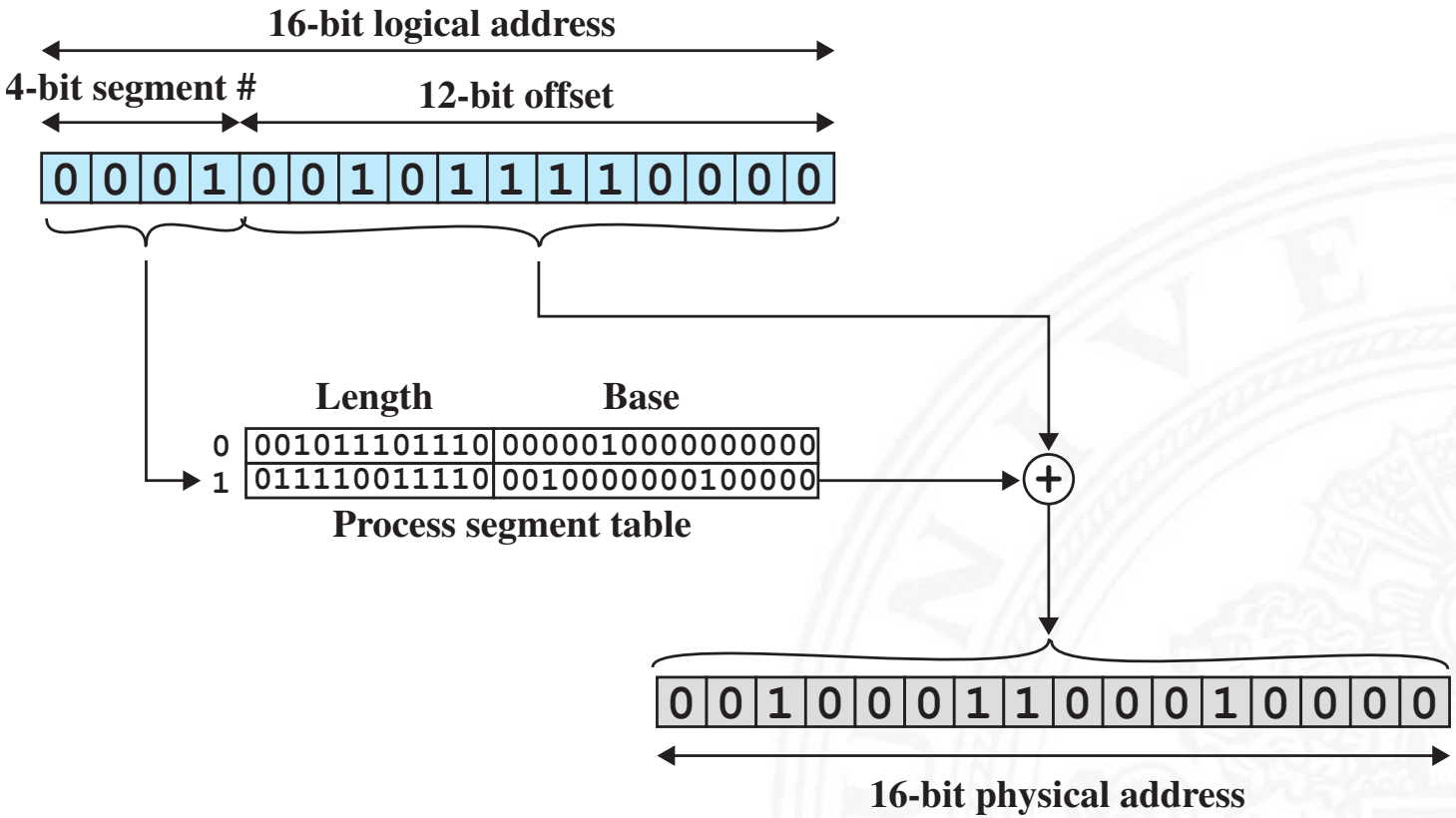
- „logische“ Unterteilung des Programms (Programmiersicht)
  - *Text*: Binärcode, read only
  - *Data*: statische Daten + dynamischer Speicher (*Heap*), read write
- variable Größe der Segmente
- ähnlich dynamischer Partitionierung
- für Programmierer sichtbar

\+ keine interne Fragmentierung

– externe Fragmentierung

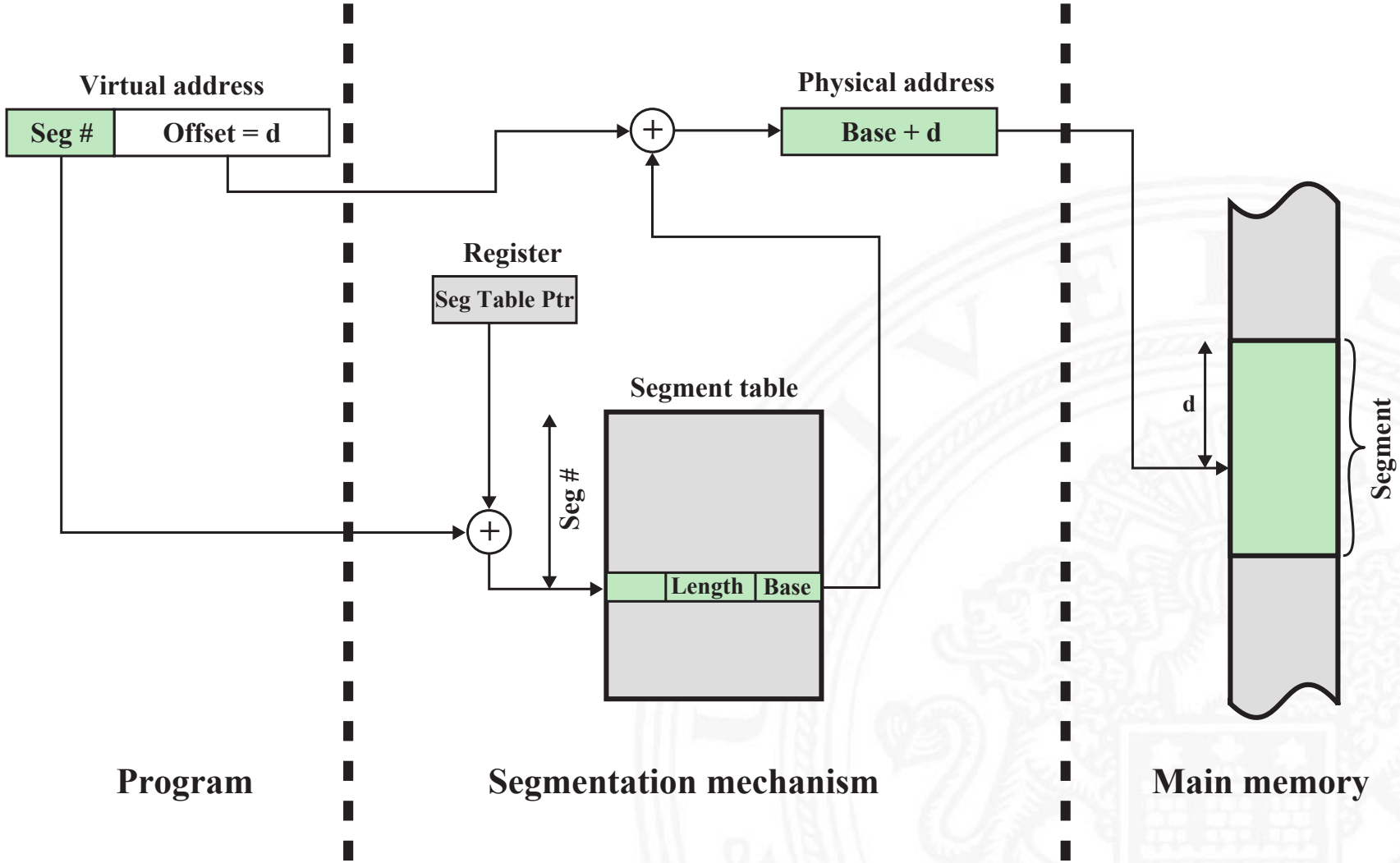
Segmentierung (cont.)

- Segmenttabelle $\langle segNr \rangle \rightarrow \langle Basisadresse \rangle + \langle Länge \rangle$
  - wird für jeden Prozess angelegt



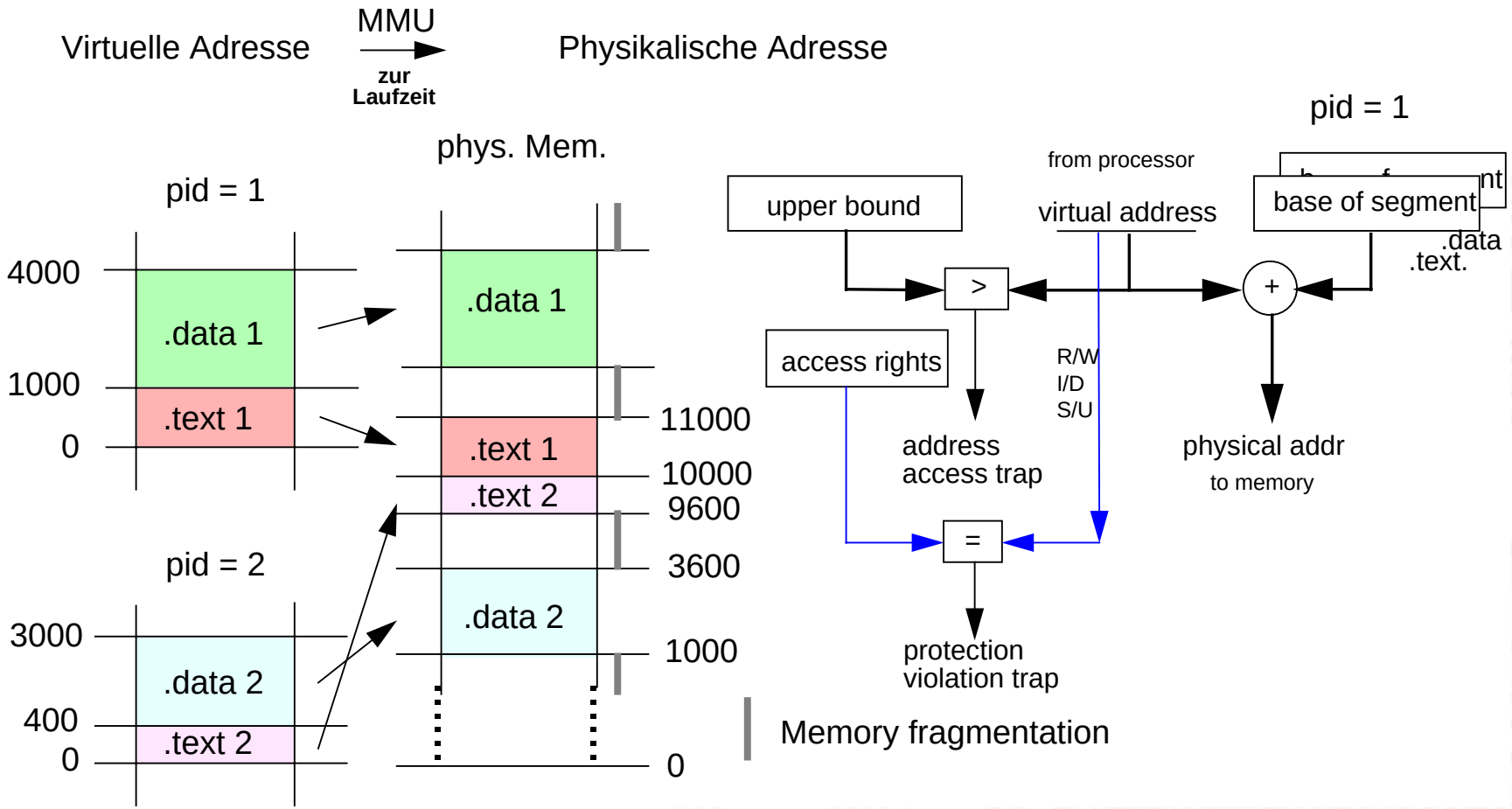
Segmentierung (cont.)

- Adressierung

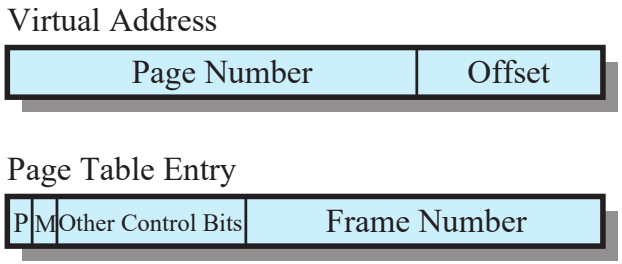


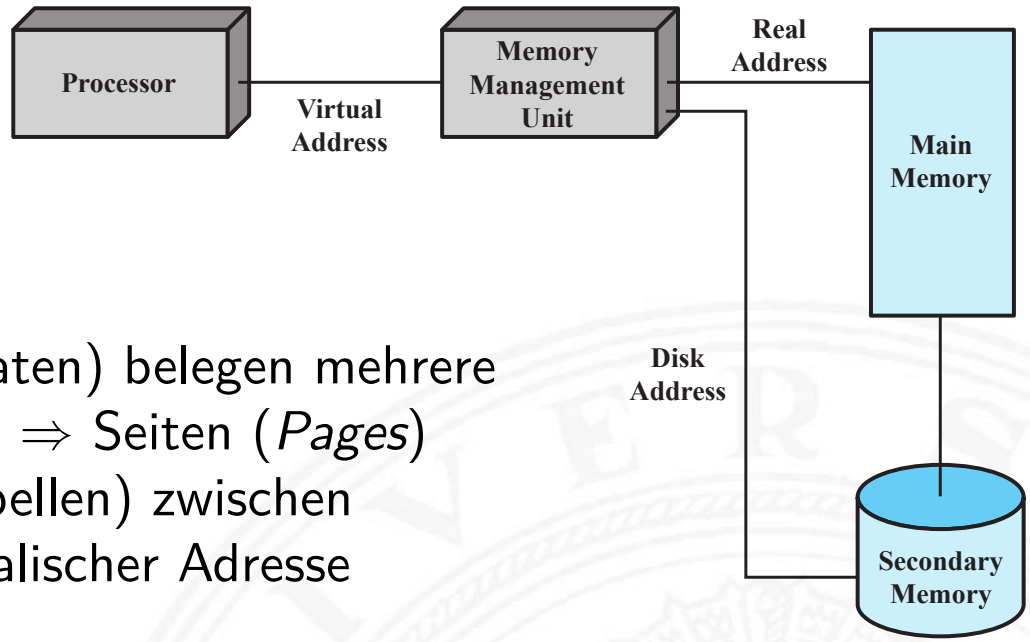
Segmentierung (cont.)

- Beispiel



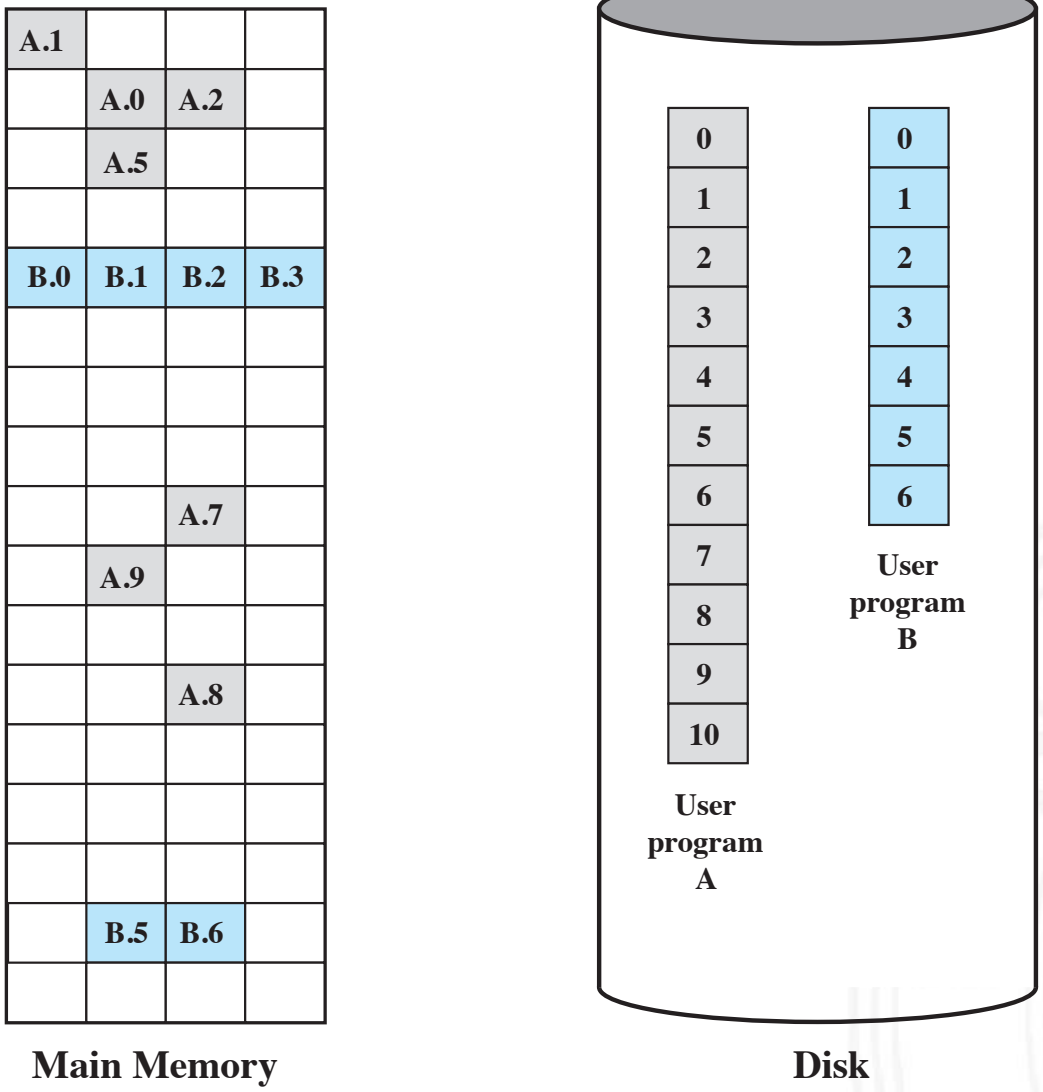
Paging





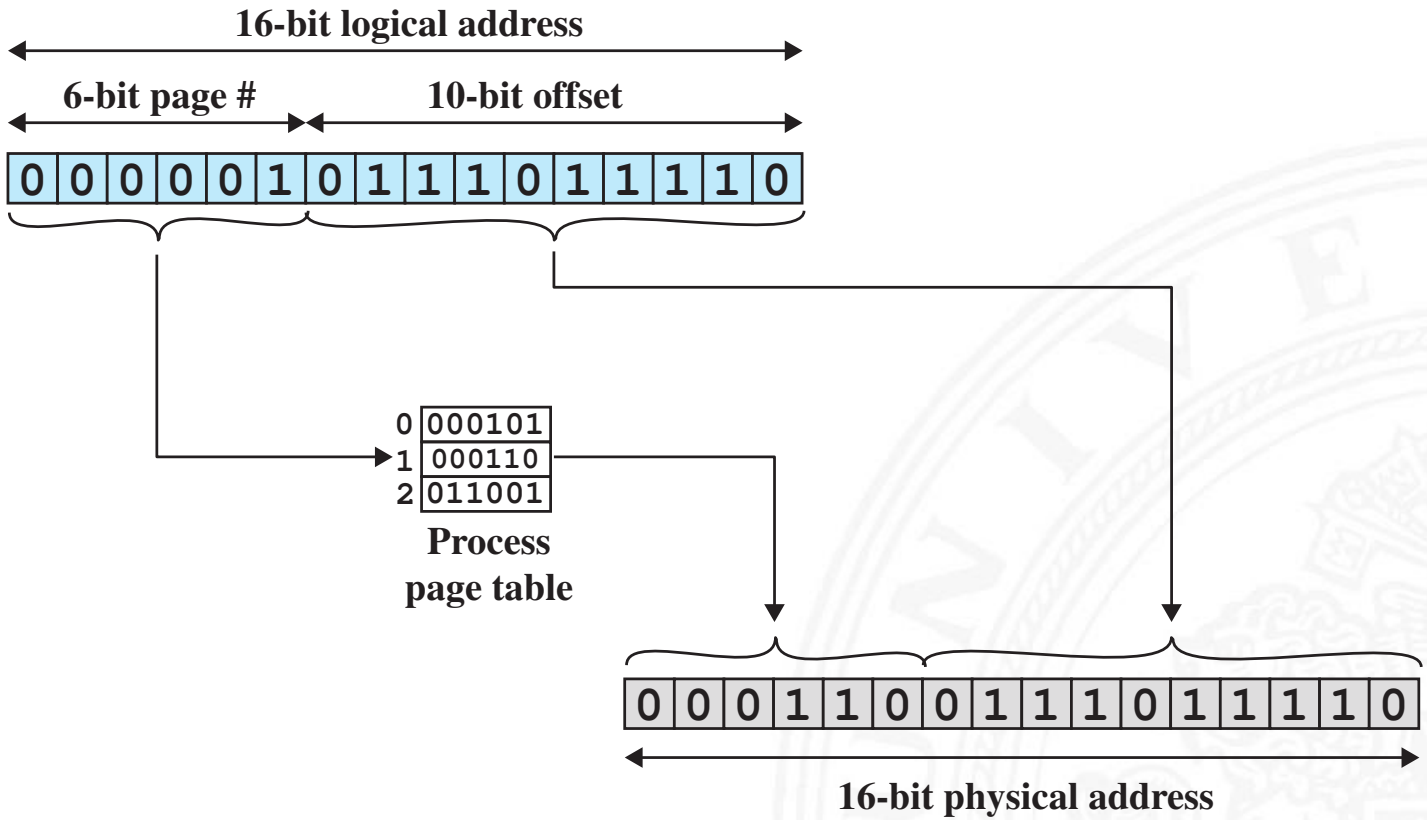
- Unterteilung des Programms
  - Prozesse (Instruktionen+Daten) belegen mehrere Speicherblöcke fester Größe ⇒ Seiten (*Pages*)
  - dynamische Abbildung (Tabellen) zwischen virtueller und realer, physikalischer Adresse
  - Speicherzugriff:
    $\langle \mathit{virt.\ Adresse} \rangle = \langle \mathit{Seitennr.} \rangle + \langle \mathit{offset} \rangle$
- Seite kann
  1. an beliebiger Stelle Hauptspeicher stehen
  2. auf sekundären Speicher (HDD, SSD) ausgelagert sein
     - Verwaltung durch *Memory Management Unit* (MMU)
- für Programmierer transparent

Paging (cont.)


A.1
A.0
A.2
A.5
B.0
B.1
B.2
B.3
A.7
A.9
A.8
B.5
B.6
Main Memory
0
1
2
3
4
5
6
7
8
9
10
User program A
0
1
2
3
4
5
6
User program B
Disk

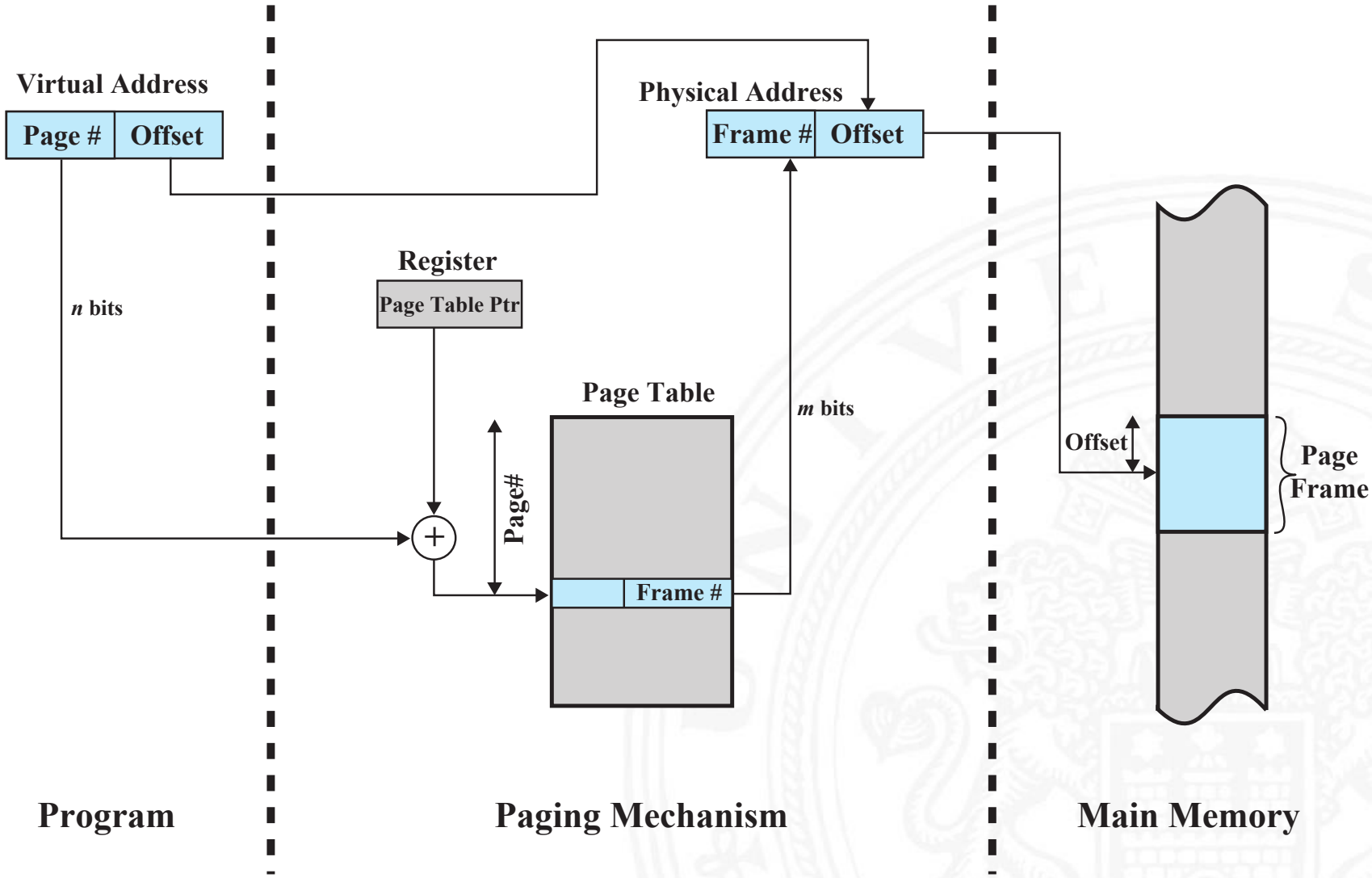
Paging (cont.)

- Seitentabelle / Page Table $\langle pageNr \rangle \rightarrow \langle frameAddr \rangle$
  - wird für jeden Prozess angelegt



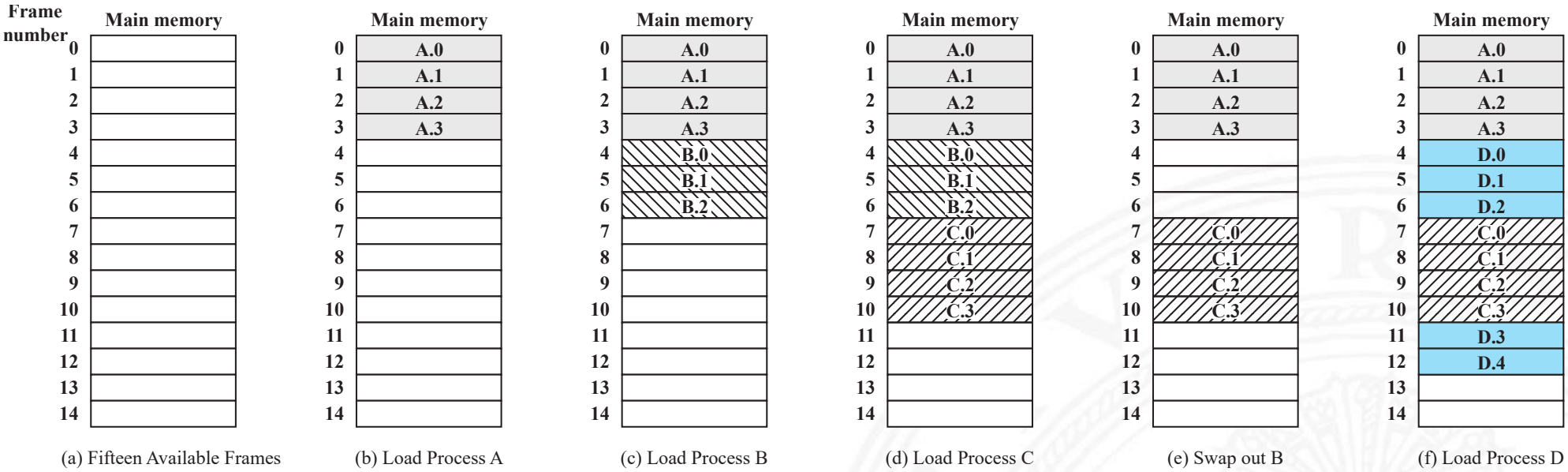
Paging (cont.)

- Adressierung



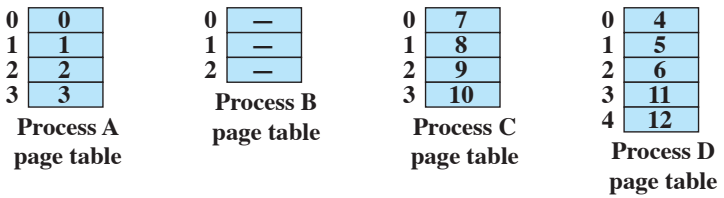
Paging (cont.)

- Beispiel



(a) Fifteen Available Frames (b) Load Process A (c) Load Process B (d) Load Process C (e) Swap out B (f) Load Process D

nach (f)

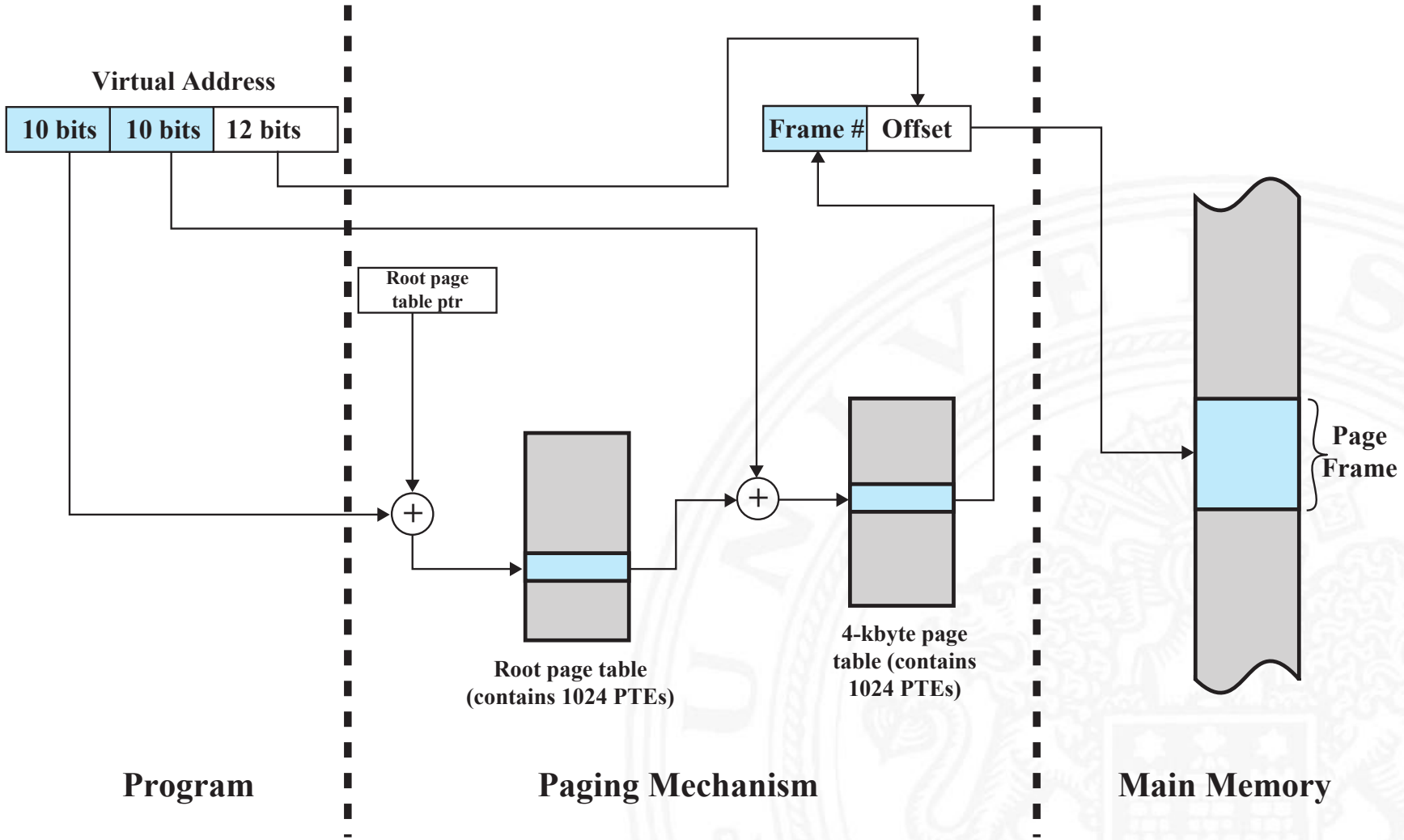


Paging (cont.)

- Seitengröße
  - 1 KiB, 4 KiB, 8 KiB ... 16 GiB (Hardware-, BS-abhängig!)
  - klein vs. groß
    - \+ weniger interne Fragmentierung
    - – aber sehr viele Seiten
    - – Prozesstabelle größer
    - – Datentransfer zu sekundärem Speicher für große Seiten effizienter
- mehrstufige Übersetzung
  - Tabelle wird zu groß $\Rightarrow$ mehrstufige Tabellen
  - – mehrfacher Zugriff langsam

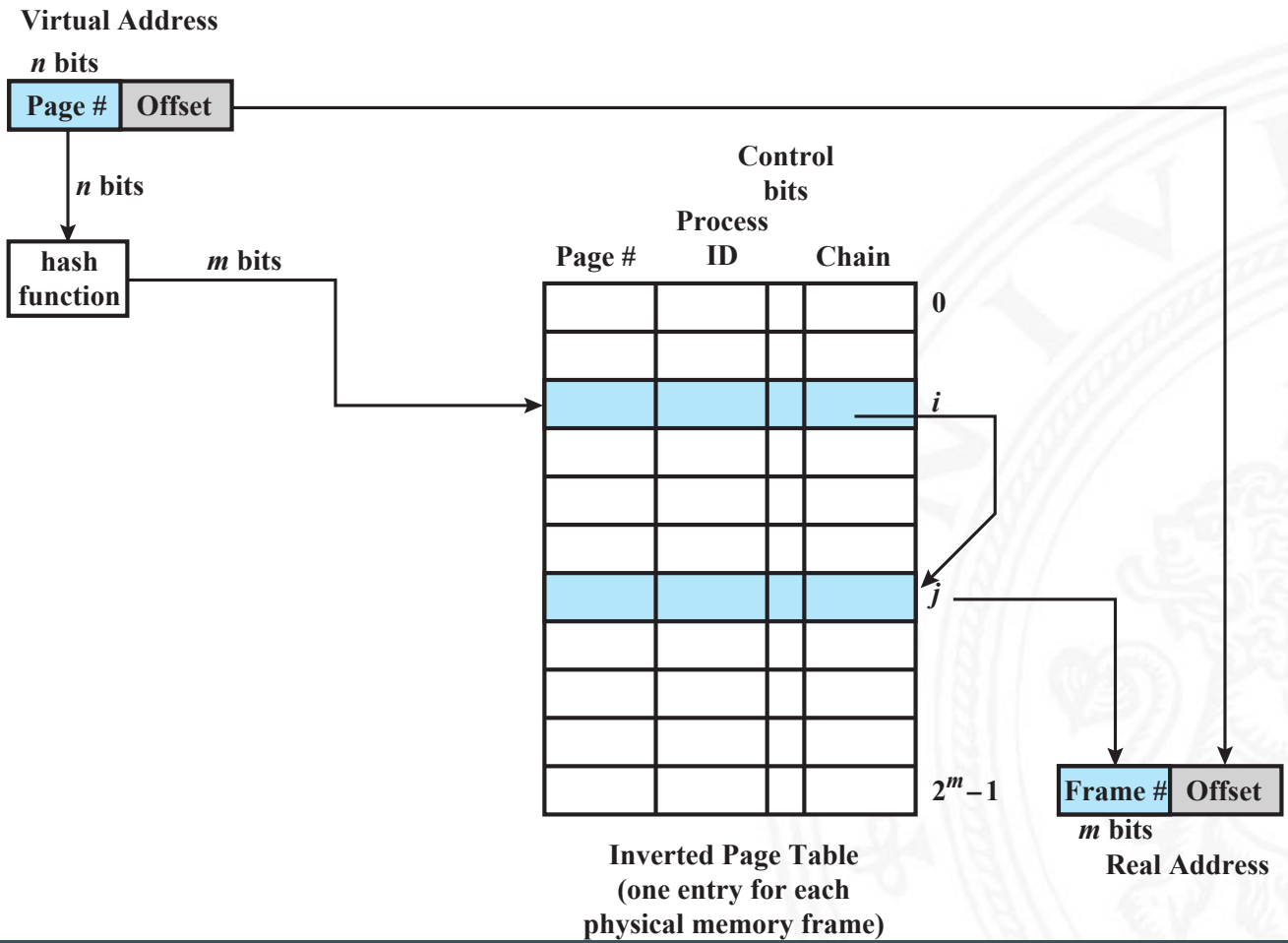
Paging (cont.)

mehrstufige Übersetzung



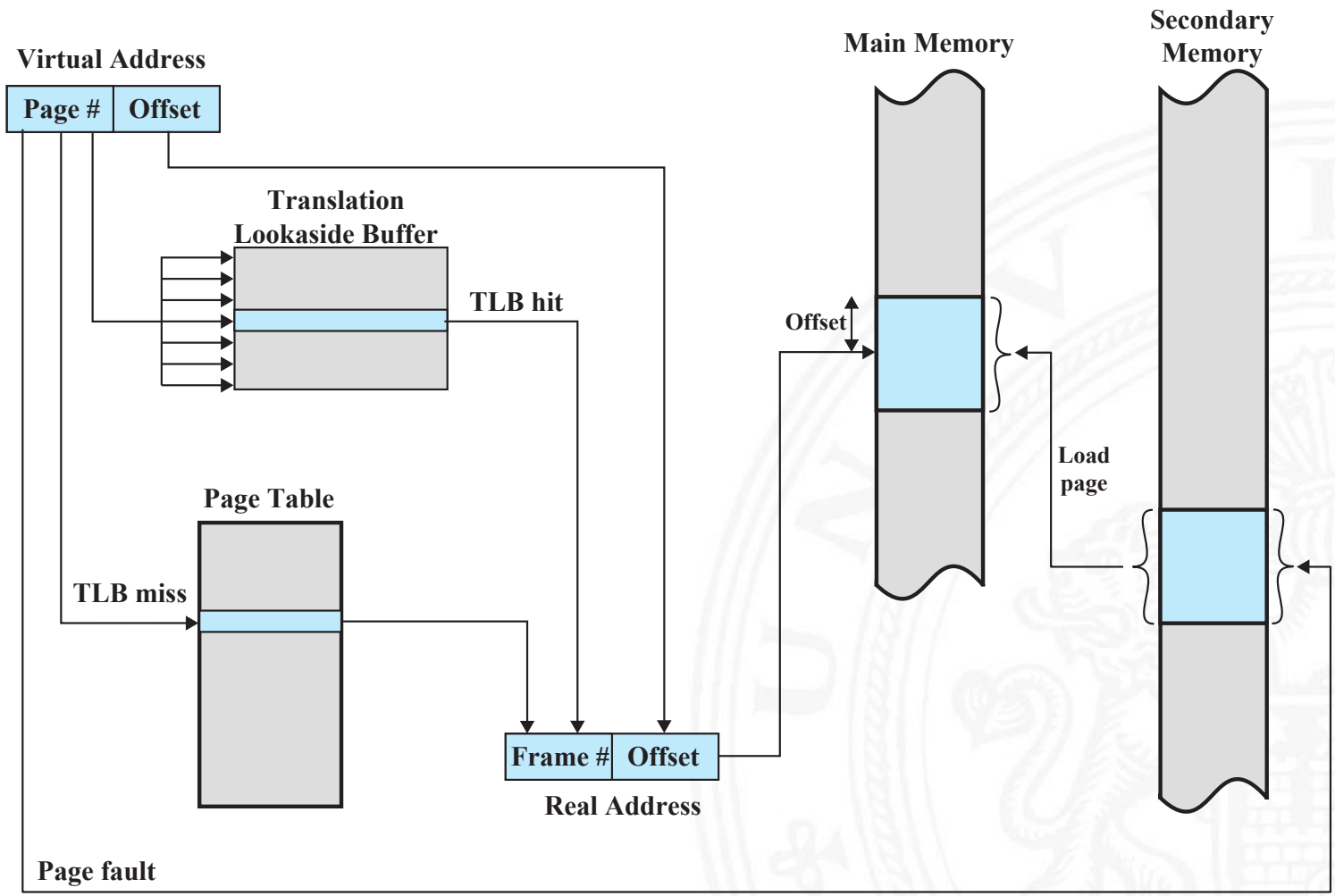
Paging (cont.)

- *Inverted Page Table*
  - statt (mehrerer) großer Seitentabellen (für alle virtuellen Adressen)
  - eine Tabelle (für alle physikalischen Adressen)
  - Hash-Funktion + verkettete Liste (Suche!)



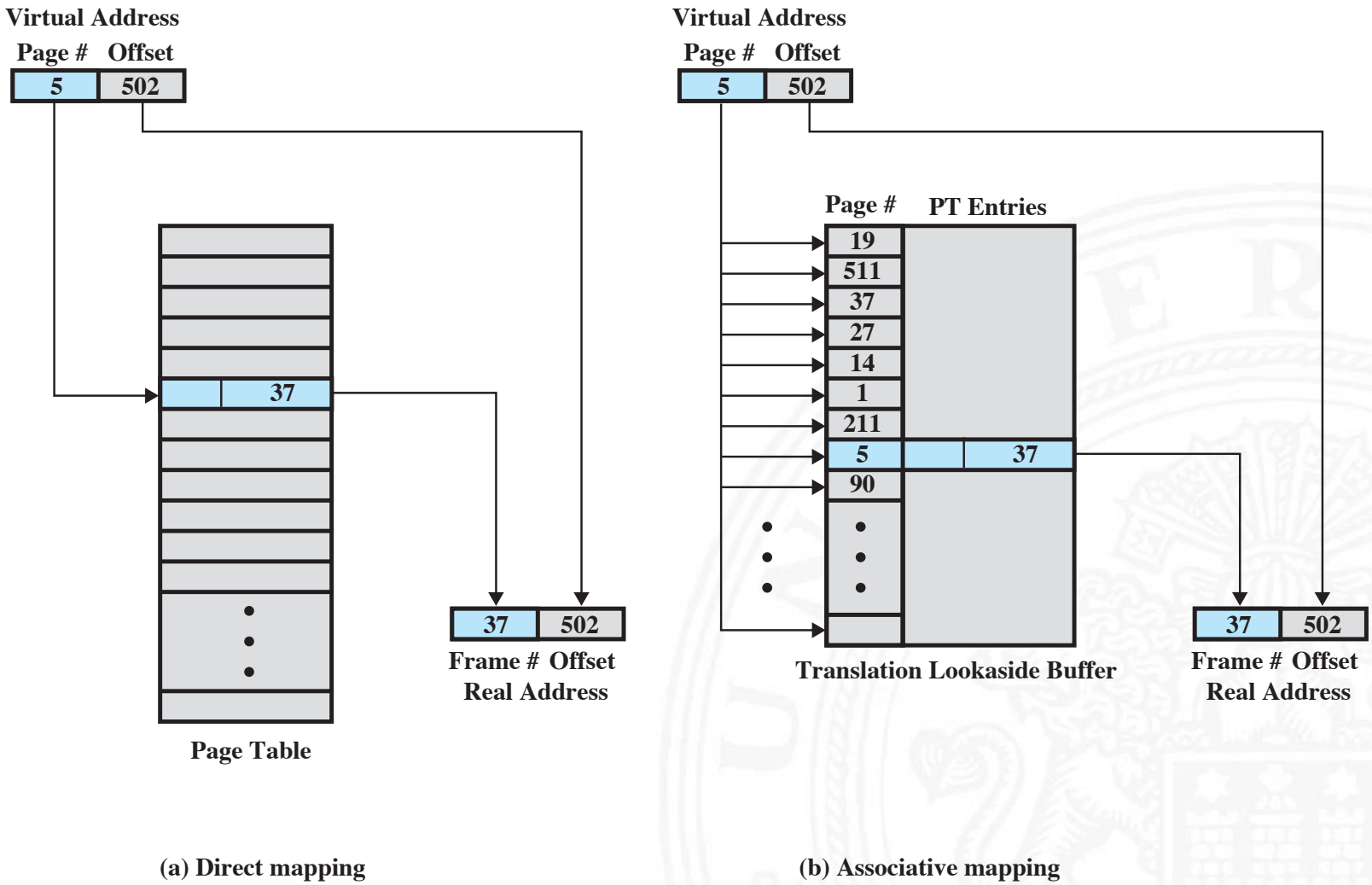
Paging (cont.)

- *Translation Lookaside Buffer (TLB)*
  - schneller Zugriff
  - voll-assoziativer Cache

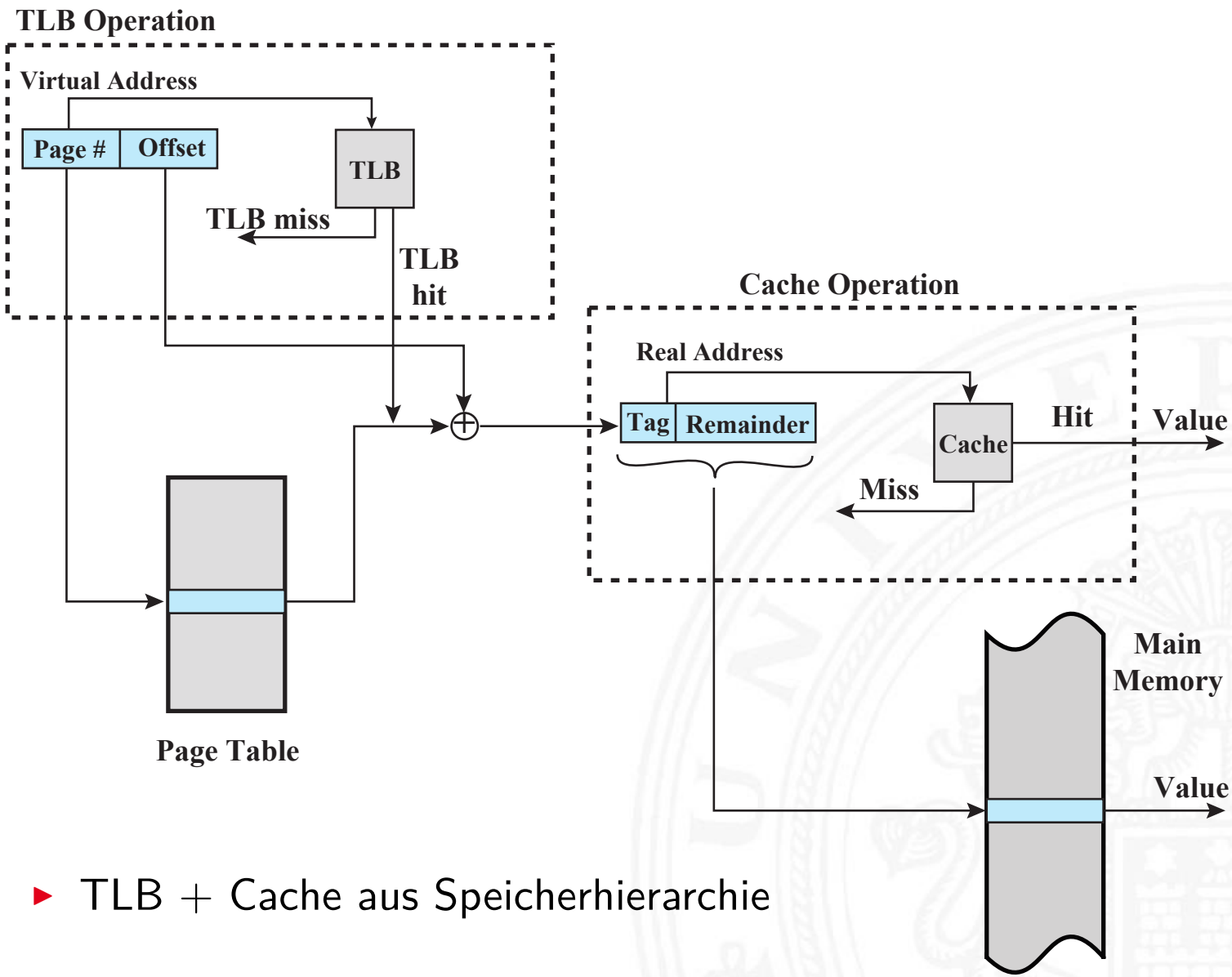


Paging (cont.)

- voll-assoziativer Cache

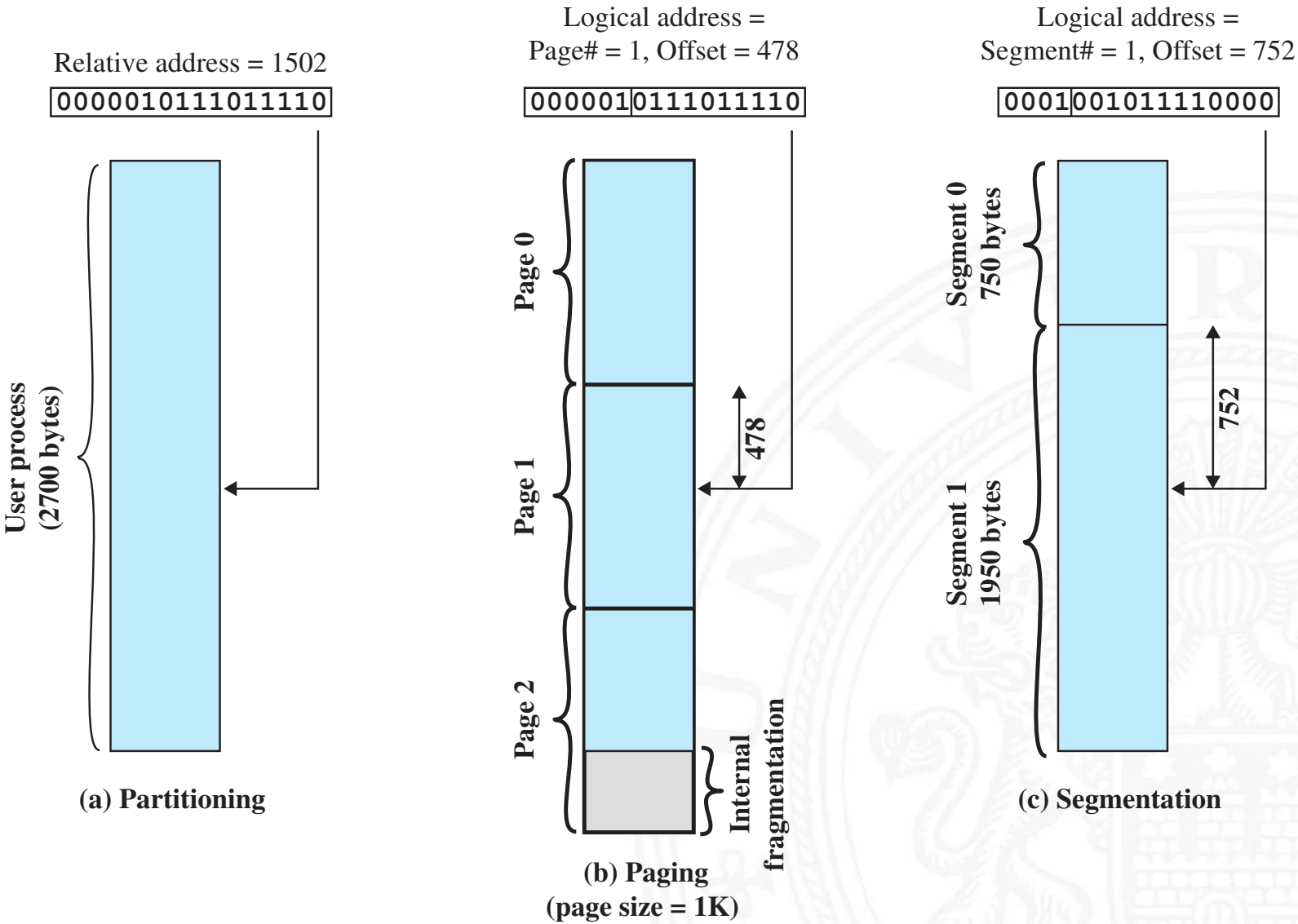


Paging (cont.)

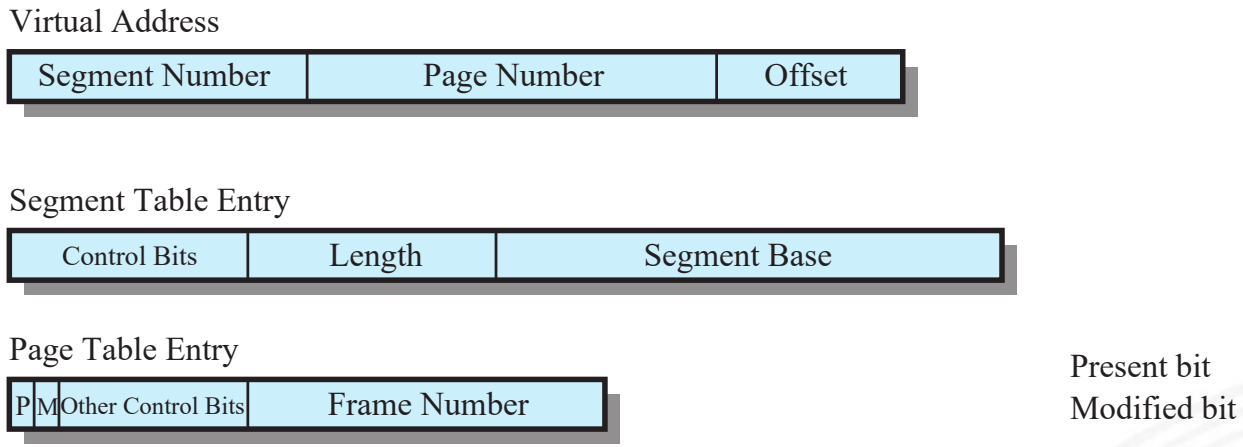


- TLB + Cache aus Speicherhierarchie

- Übersicht Adressierung



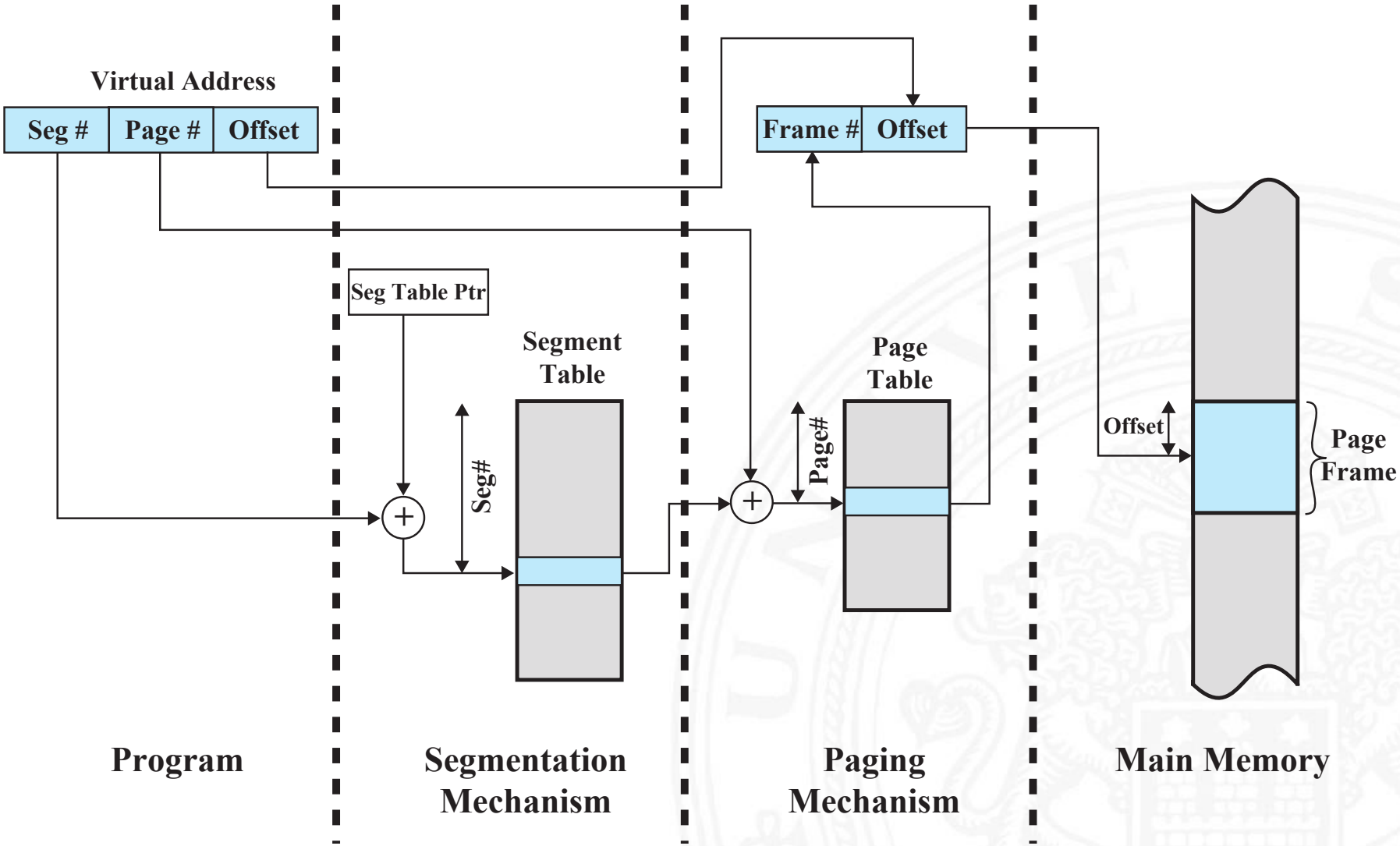
aktuelle Betriebssysteme (cont.)



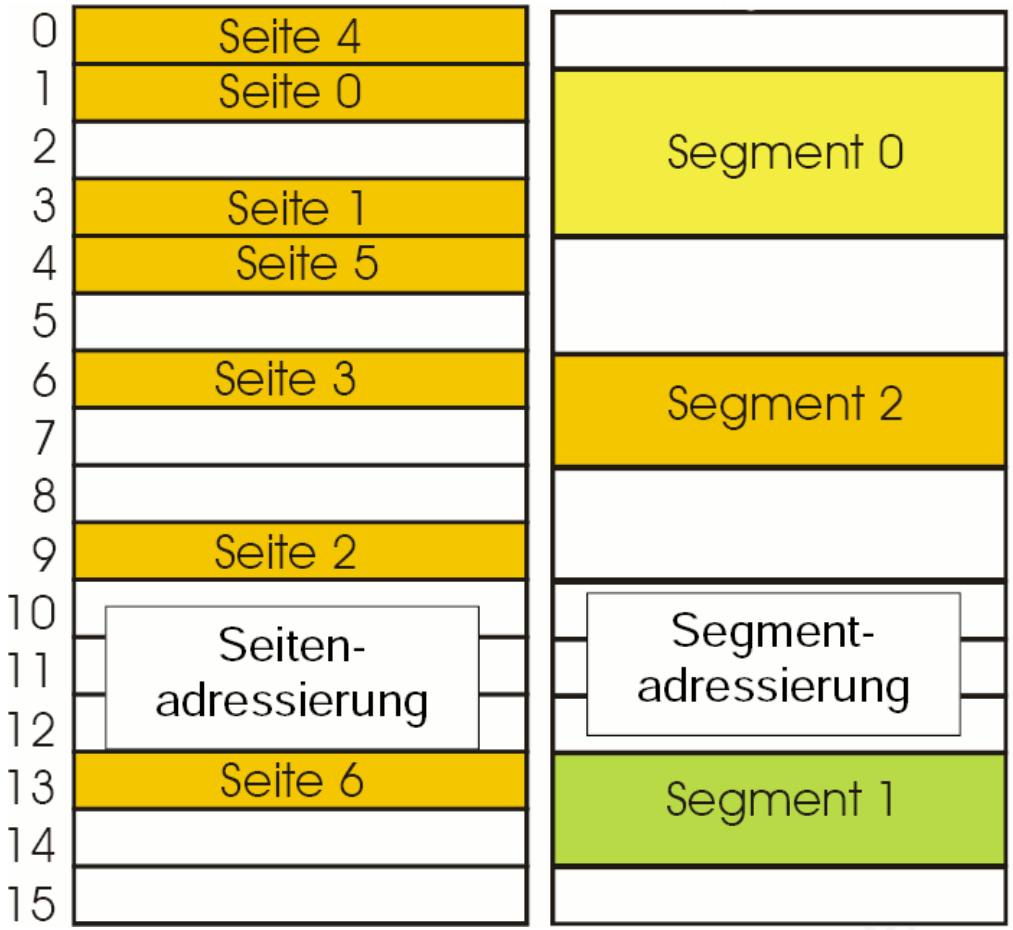
- Segmentierung + Paging
  - Segmentierung ⇒ logische Trennung (Text, Data)
  - + Paging ⇒ effiziente Verwaltung
- im Betriebssystem festlegen
  - wann Seiten Laden? (auf Anfrage, Prepaging)
  - wo in Hauptspeicher?, welche Seiten werden ausgelagert?
    = Platzierungs- und Ersetzungsstrategie
  - wieviel Multiprogramming? (Anzahl der Prozesse)
  - wieviel Hauptspeicher pro Prozess (fest, dynamisch . . .)
  - vielfältige Wechselwirkungen: Anwendungsszenario, Caching etc.

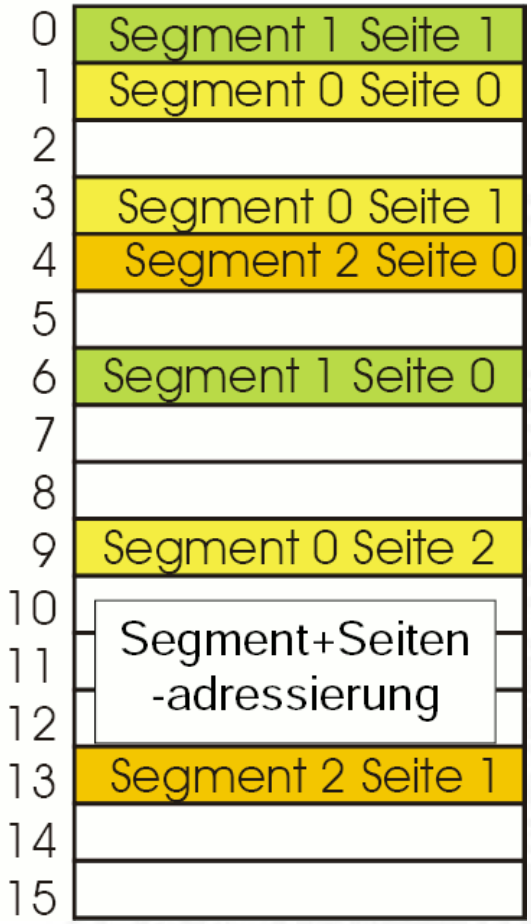
aktuelle Betriebssysteme (cont.)

- Adressierung



aktuelle Betriebssysteme (cont.)



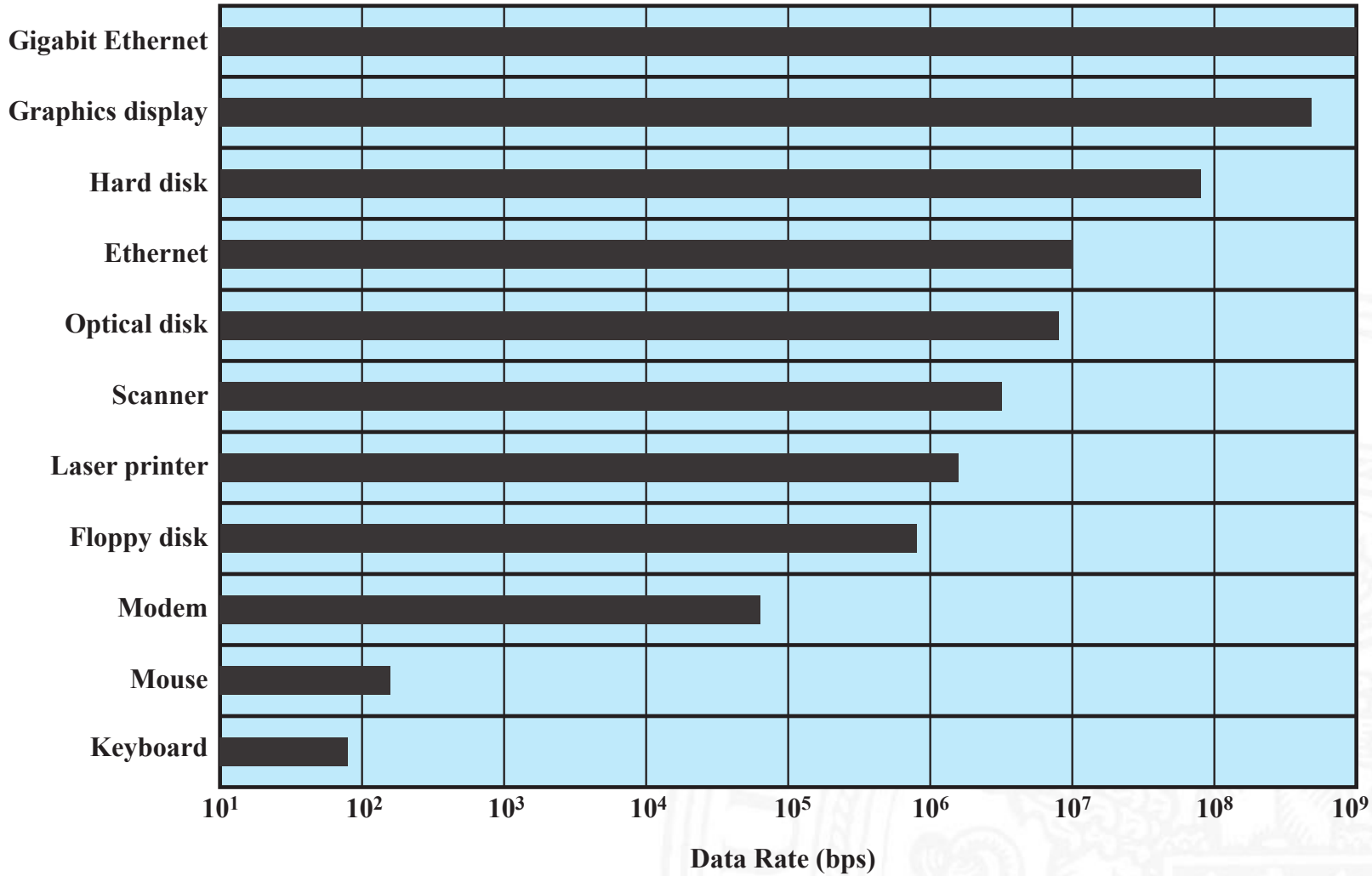


Ein-/Ausgabegeräte

- Mensch-Maschine Schnittstellen
  - Displays, Tastatur, Maus, Terminals, Drucker ...
- Schnittstellen zu Computerperipherie
  - Festplatten, Speichermedien, Sensoren, Controller ...
- Kommunikation zwischen Computersystemen
  - Netzwerk, Modems ...

- Charakteristika
  - Datenrate
  - Anwendung
  - Schnittstellen
  - Protokoll / Fehlerbedingungen
  - Daten (Zeichen, Datenstrukturen, Blöcke von Bytes ...)
  - Repräsentation der Daten
  - ...

Ein-/Ausgabegeräte (cont.)



I/O-Optimierung

- Effizienz
  - I/O oft als „*Bottleneck*“ im System
  - extrem langsam, verglichen mit Hauptspeicher oder CPU
- Generalität
  - einheitliche Schnittstelle für (möglichst viele) Geräte (-klassen)
  - sowohl als Programmierschnittstelle
  - als auch für das Betriebssystem selber
  - hierarchische, modulare Konzepte

  $\Rightarrow$ wegen der völlig unterschiedlichen Geräteeigenschaften und der Dynamik der technischen Entwicklung schlecht realisierbar
- stufenweise Entwicklung
  1. Prozessor kontrolliert Gerät direkt
  2. Prozessor kontrolliert Gerät über I/O-Modul (Controller)
  3. + Interruptsteuerung
  4. I/O-Controller hat Hauptspeicherzugriff (DMA)
  5. I/O-Prozessor: eigener Befehlssatz (programmierbar), statt „Automat“
  6. I/O-System mit eigenem Speicher = Computer, realisiert I/O-Dienst

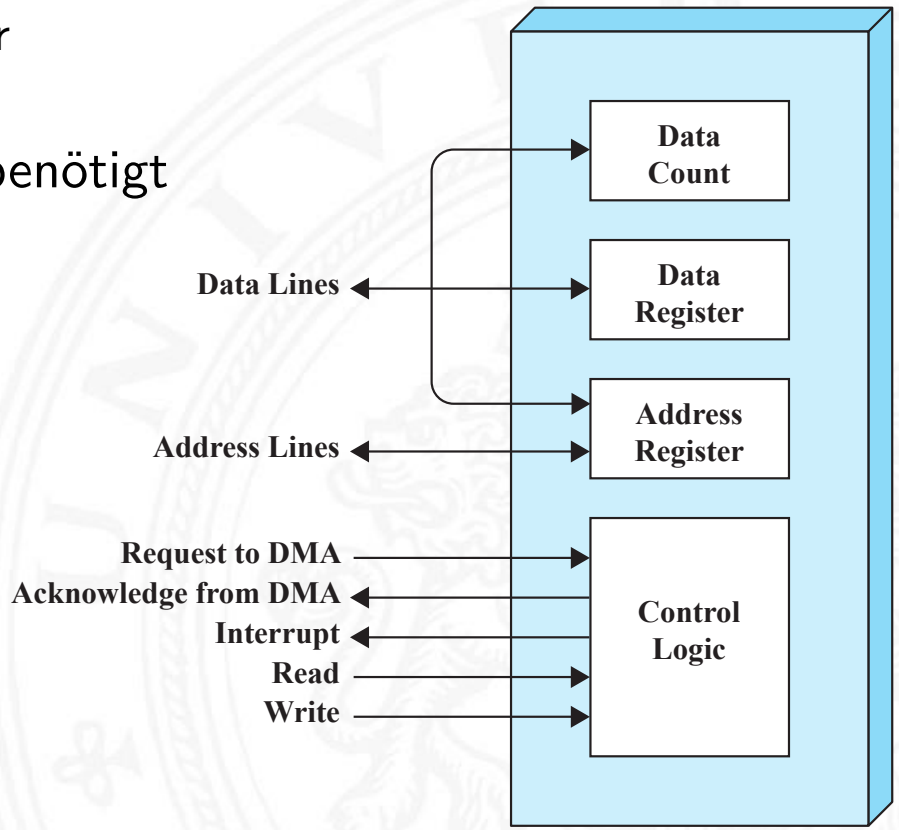
Ein-/Ausgabe Behandlung

1. Programmed I/O
   - I/O-Modul führt Operation aus
   - setzt Bits in Status-Register (Kommunikation)
   - Prozessor fragt periodisch Status ab
   - – „*Busy-waiting*“, schlechte Performanz

2. Interrupt gesteuert
   - Befehl von Prozessor an I/O-Modul
   - Interrupt, wenn I/O bereit
   - Datentransfer durch Prozessor (in ISR)
   - – Beteiligung des Prozessors
     Performanz (mehrere Instruktionen pro Datentransfer)
   - → siehe *15.2 Betriebssysteme – Interrupts*

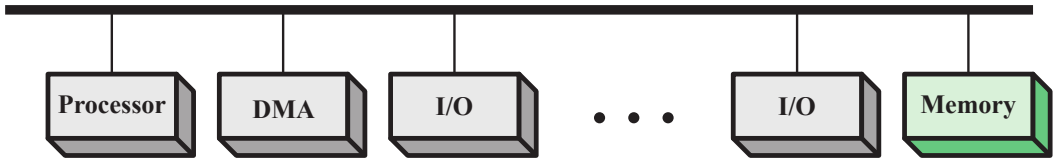
Ein-/Ausgabe Behandlung (cont.)

3. DMA (*Direct Memory Access*)
   - separate Hardwareeinheit mit Zugriff auf Systembus
   - DMA-Controller übertragt selbständig Daten
   - DMA-Kommando mit
     - Lesen / Schreiben
     - Adresse des I/O Geräts
     - Startadresse in Hauptspeicher
     - Anzahl Datenwörter
   - Konflikt, wenn Prozessor Bus benötigt

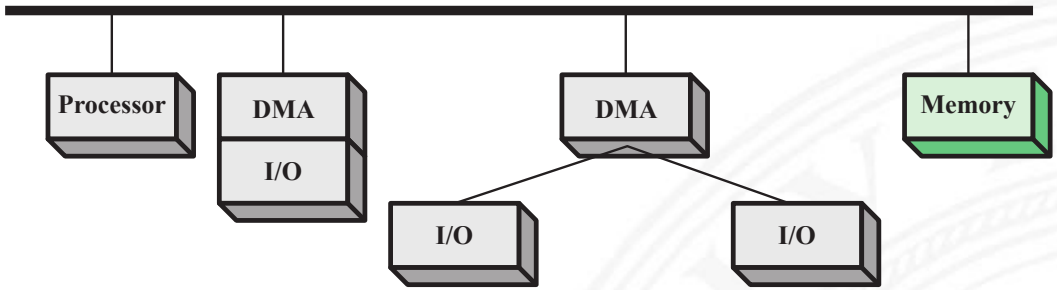


Ein-/Ausgabe Behandlung (cont.)

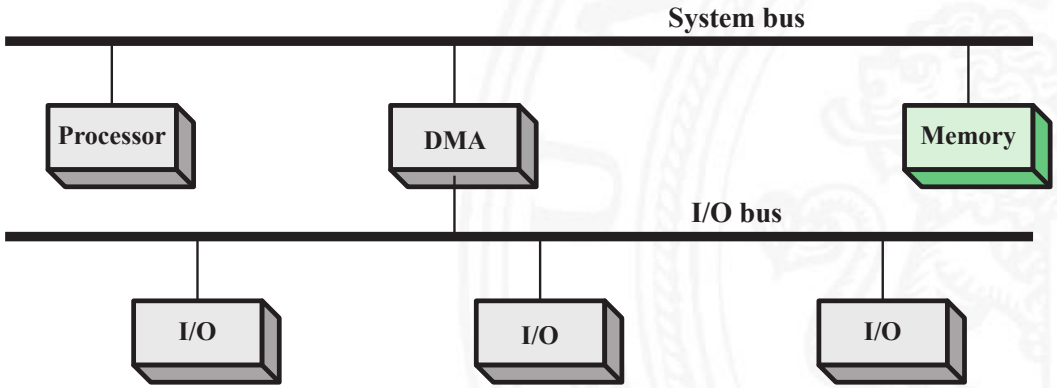
DMA



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus

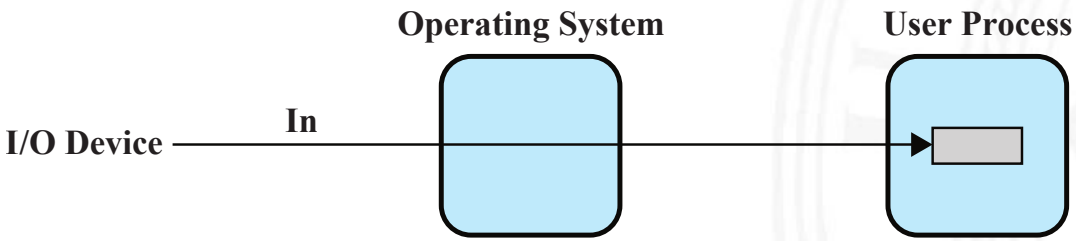
Ein-/Ausgabe Behandlung (cont.)

- I/O Betriebsarten

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

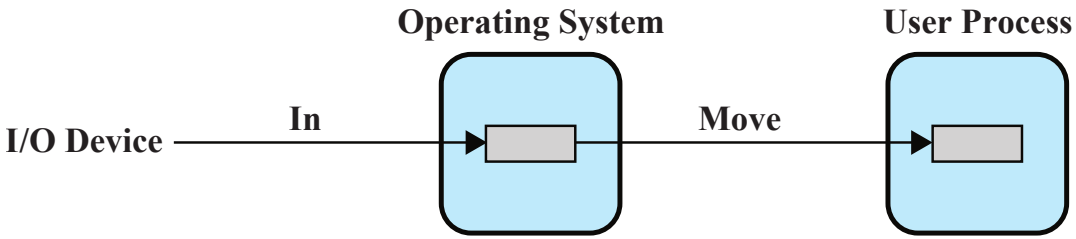
- Schnittstelle zum Benutzerprogramm / *Buffering*
  - lesend: „read-ahead“, Daten schon bereitstellen
  - schreibend: verzögerte (autonome) Ausführung

  $\Rightarrow$ Zwischenspeicher, in der Regel FIFO

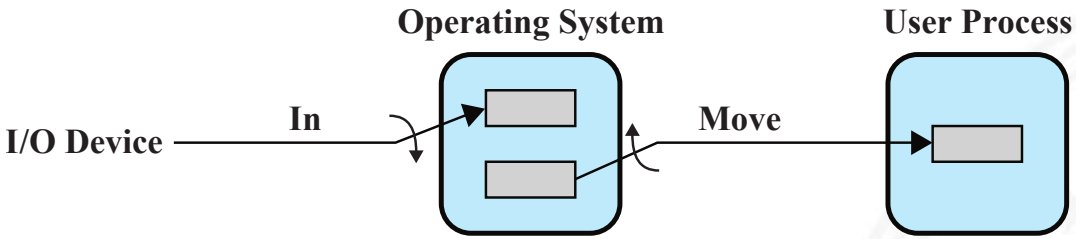


(a) No buffering

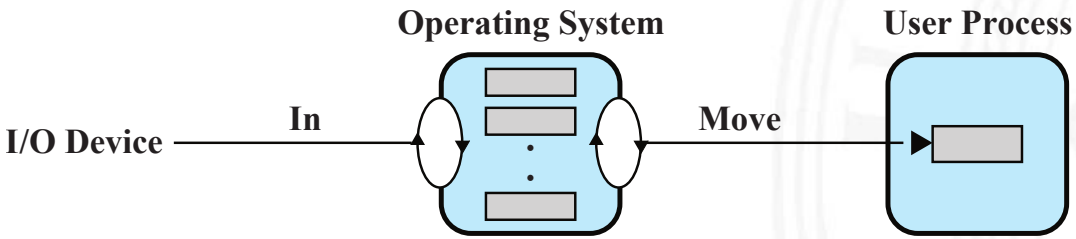
Ein-/Ausgabe Behandlung (cont.)



(b) Single buffering



(c) Double buffering

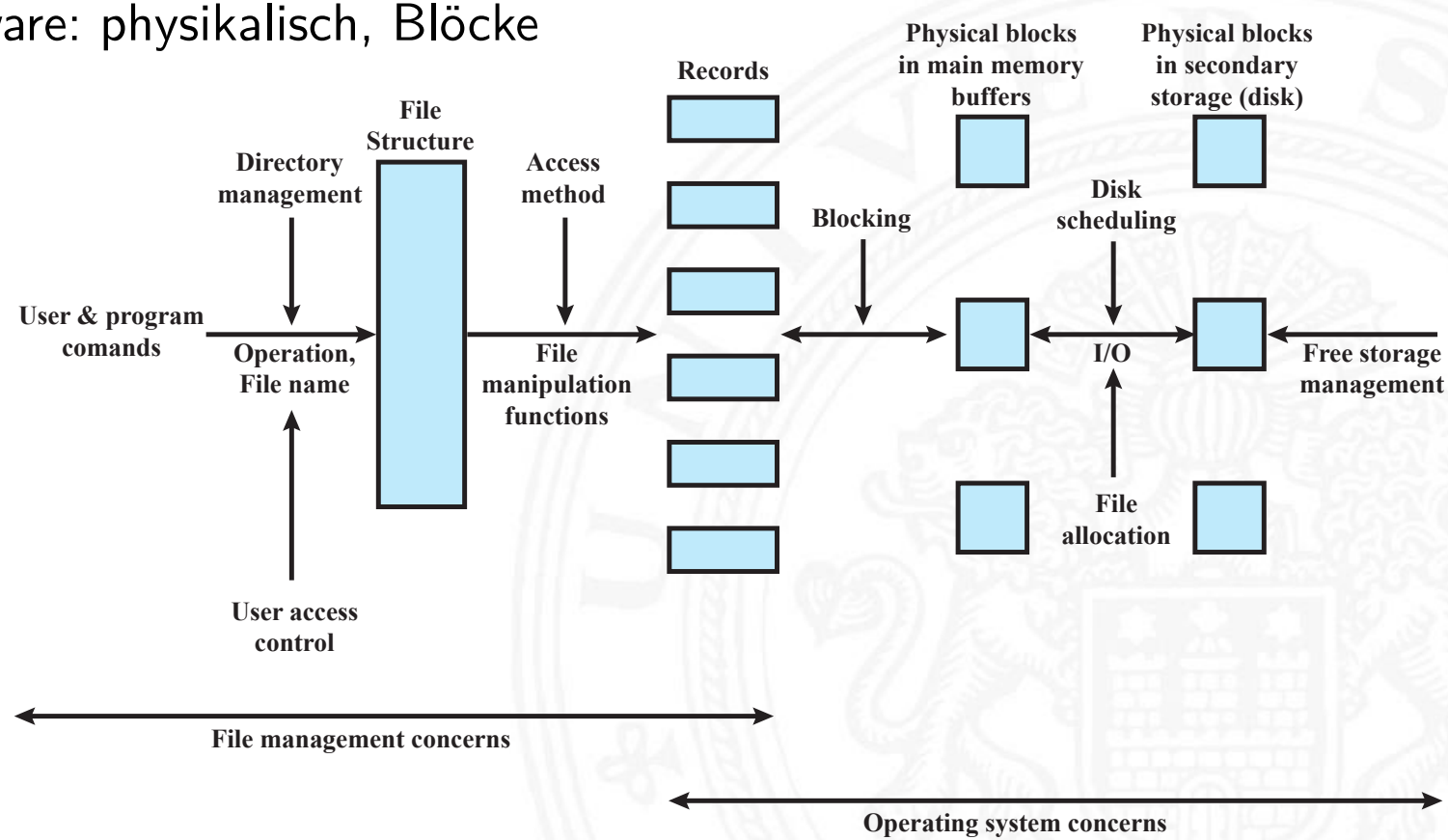


(d) Circular buffering

Weitere Themen

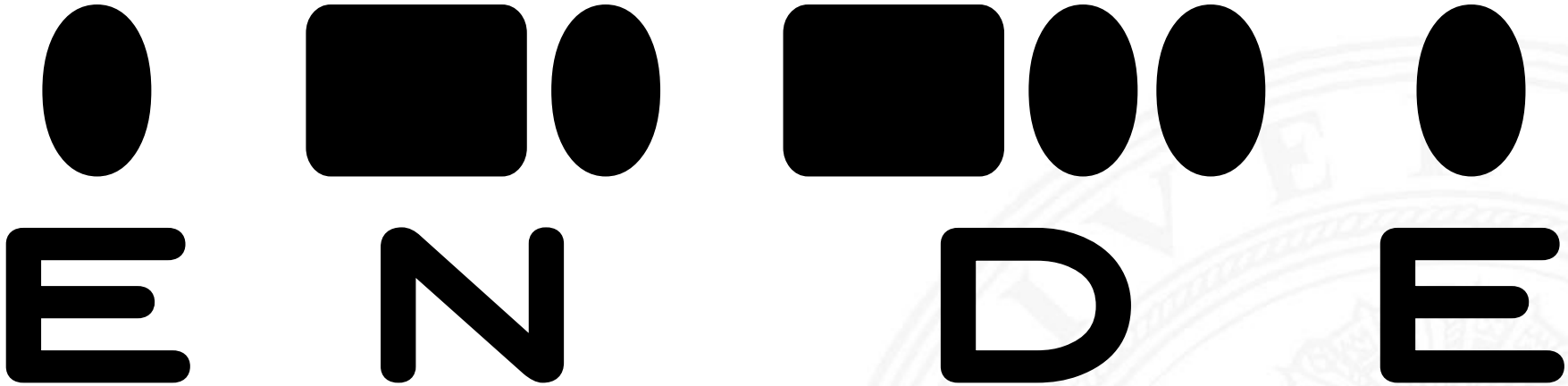
- Festplattenzugriffe: das zentrale Thema bei I/O-Optimierung
  - viele Algorithmen zum „Umsortieren“ von Schreib-/Lesebefehlen

  $\Rightarrow$ Optimierung des Durchsatzes
- Daten-Organisation
  - Benutzer: logisch, Dateisysteme und Verzeichnisse
  - Hardware: physikalisch, Blöcke



Weitere Themen

Weitere Themen


E N D E

Literatur

[Sta17] W. Stallings: *Operating Systems – Internals and Design Principles*.
9th, global ed., Pearson Education, 2017.
ISBN 978–1–292–21429–0

[Bau17] C. Baun: *Betriebssysteme kompakt*.
Springer-Verlag GmbH, 2017.
ISBN 978–3–662–53142–6

[Bra17] R. Brause: *Betriebssysteme – Grundlagen und Konzepte*.
Springer-Verlag GmbH, 2017.
ISBN 978–3–662–54099–2

[TB16] A.S. Tanenbaum, H. Bos: *Moderne Betriebssysteme*.
4. Auflage, Pearson Deutschland GmbH, 2016.
ISBN 978–3–8689–4270–5

Literatur (cont.)

[BO15] R.E. Bryant, D.R. O'Hallaron:
Computer systems – A programmers perspective.
3rd global ed., Pearson Education Ltd., 2015.
ISBN 978–1–292–10176–7. `csapp.cs.cmu.edu`