

Inhaltsverzeichnis

1.1 Der Begriff des Betriebssystems

1.2 Zur Geschichte der Betriebssysteme

1.3 Aufbau eines Rechners

- E/A-Operationen, Speicherstrukturen

1.4 Aufbau eines Betriebssystems

- Komponenten, Systemaufrufe, Systemprogramme

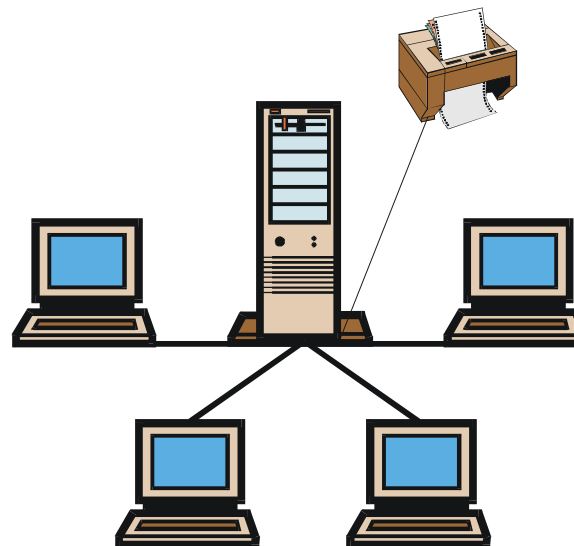
1.5 Beispielarchitekturen

- MS-DOS, UNIX, OS/2, Schichtenkonzepte

1.1 Der Begriff des Betriebssystems

Betriebssystem:

Ein Betriebssystem ist die Gesamtheit der Hardwaremechanismen und der Softwareroutinen zur effizienten Aufteilung der Betriebsmittel wie Speicher, CPU, Ein-/Ausgabe-Geräte (E/A), Kommunikationskanäle und Zeit an die Benutzer des Rechensystems.



Organisation des Zusammenspiels der einzelnen Komponenten

1.1 Der Begriff des Betriebssystems / Aufgaben

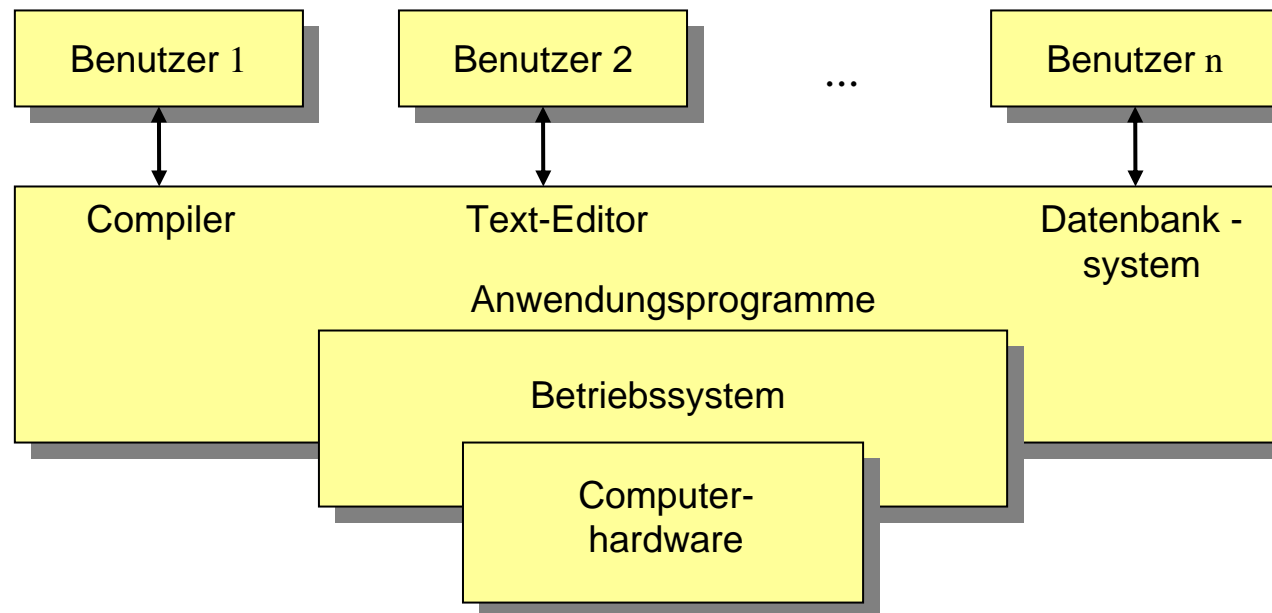
Das Betriebssystem ist u.a. verantwortlich für:

- Laden/Verdrängen von Jobs, Prozessen oder Programmen
- Betriebsmittelzuteilung
- Dateiverwaltung
- Steuerung der Reihenfolge der Jobs → Scheduling
- Interrupt-Bearbeitung → Problem: Beim Zusammenspiel zwischen schneller CPU und erheblich langsamer E/A ist ein Zeitverhältnis 1:1000 nicht unüblich
- Fehlerbehandlung
- Gewährleistung von Schutzmechanismen
- Gebührenberechnung
- Leistungsmessung und Systemtuning

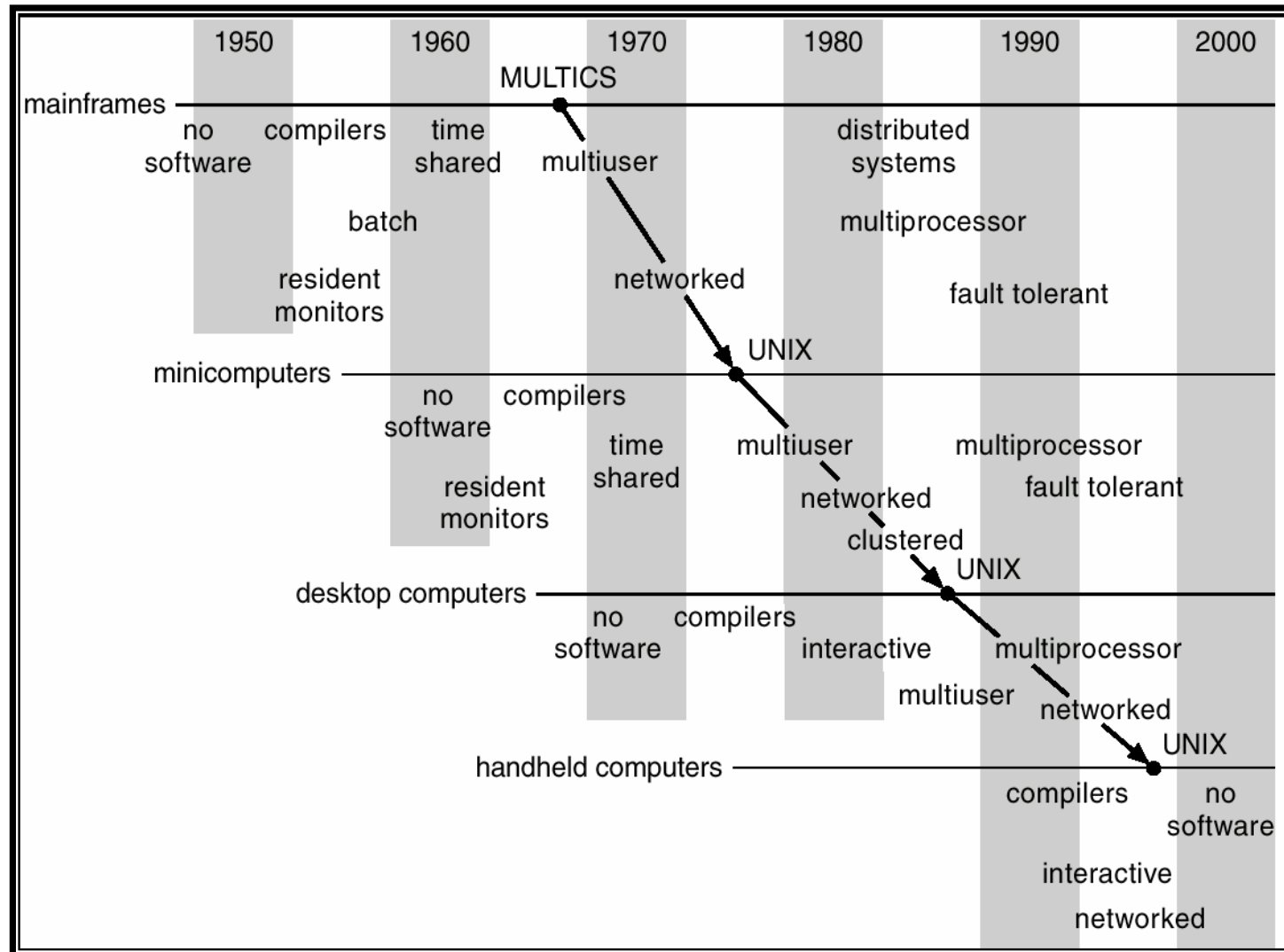
1.1 Der Begriff des Betriebssystems

Managementaspekte

- Konfigurationsmanagement
- Fehlerbehandlungsmanagement
- Performance-Management
- Accounting
- Security-Management



1.2 Zur Geschichte der Betriebssysteme



Quelle:
A.Silberschatz,
**Operating
System
Concepts,**
6. Auflage.

1.2 Zur Geschichte der Betriebssysteme

Single-User-Betriebssystem im Open-Shop-Betrieb

- Ist z.T. heute durch PC-Entwicklung wieder modern.
- Exklusive Nutzung durch einen einzigen Benutzer, der durch ein vergleichsweise einfaches Betriebssystem unterstützt wird.

SPOOLing Batch (Simultaneous Peripheral Operation On-Line)

- Reduktion der Diskrepanz zwischen schneller CPU und langsamer E/A und erster Schritt zur Parallelarbeit durch
 - Eingabe über Kartenleser auf Band
 - Ausführung von Jobs auf der CPU
 - Ausgabe der Ergebnisse vom Band auf den Drucker

1.2 Zur Geschichte der Betriebssysteme

Multiprogrammierung

- Verwaltung mehrerer Jobs gleichzeitig durch das Betriebssystem, um die Geschwindigkeitsdiskrepanz $E/A \ll CPU$ besser auszugleichen.
- Nur Teile von Jobs laden und bei Bedarf durch aktive Teile verdrängen
 - ➔ Speicherverwaltungsproblem
- Betriebssystem bedient abwechselnd aufgrund von Interrupt-Meldungen die weiteren Jobs.

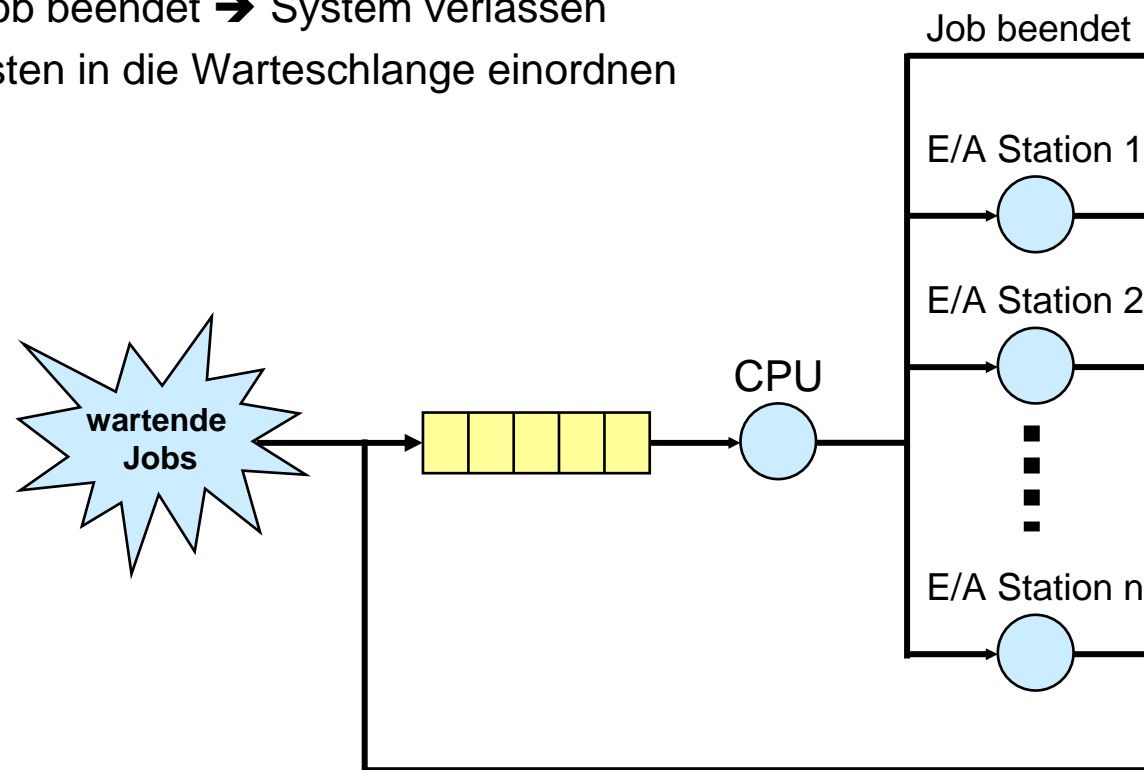
Timesharing-Systeme

- Programme nicht nur bei Interrupts unterbrechbar sondern jetzt auch nach Zeitscheibenablauf.
- Dialogmöglichkeit zwischen Benutzer und Programm vom Terminal aus
 - ➔ interaktiver Betrieb
- **Illusion des Benutzers:** Anlage scheint nur für ihn zu arbeiten

1.2 Zur Geschichte der Betriebssysteme

Central Server Model

- 1 Server = CPU im Zentrum
- Jeder Job wird eine bestimmte Zeit (Zeitscheibe) bearbeitet
 - Falls Job beendet → System verlassen
 - Ansonsten in die Warteschlange einordnen



1.2 Zur Geschichte der Betriebssysteme / Mehrrechnersysteme

Bisher Einzelrechner:

- eine CPU und viele langsame E/A Einheiten

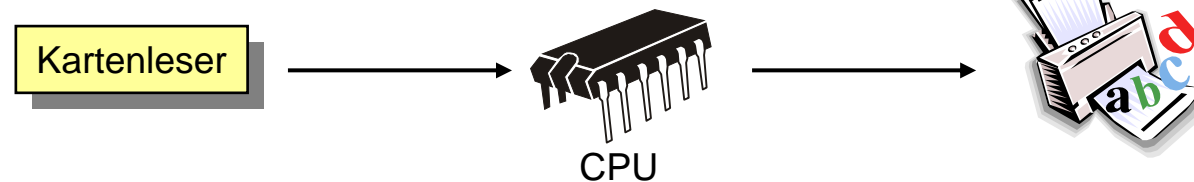
Jetzt Mehrrechner:

- mehrere (evtl. spezialisierte) CPU. Vor allem für sehr rechenintensive Aufgaben
➔ „grand challenges“
- über ein Netz gekoppelt ➔ verteiltes System
- viele Endgeräte, die z.T. eigene Speichermedien und Rechenkapazitäten haben
➔ Workstations. Die Kommunikation mit der CPU wird dadurch reduziert.
- wachsender Bedarf zur Übertragung großer Datenbestände ➔ File Transfer
z.B. bei Höchstleistungsrechnern für Meteorologie

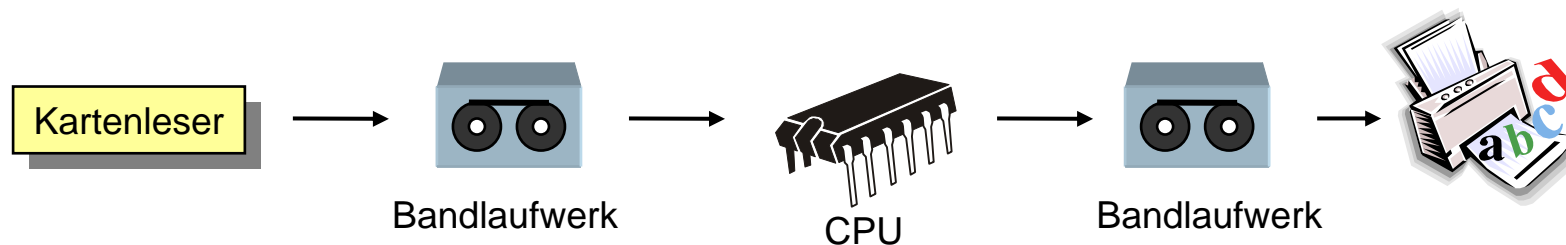
1.2 Zur Geschichte der Betriebssysteme

Beispiel: Beschleunigung durch Entkopplung von CPU und E/A-Geräten

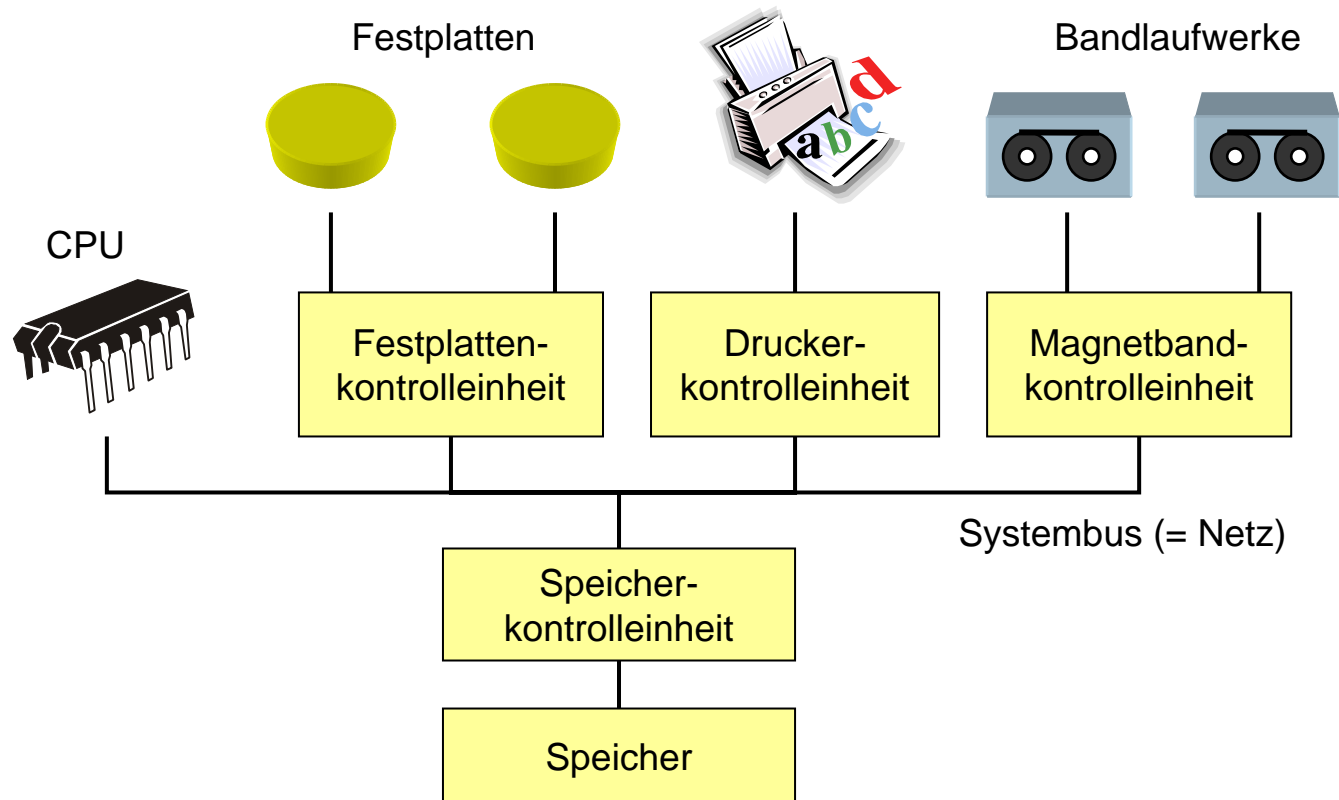
1. Online-Operationen mit I/O-Geräten



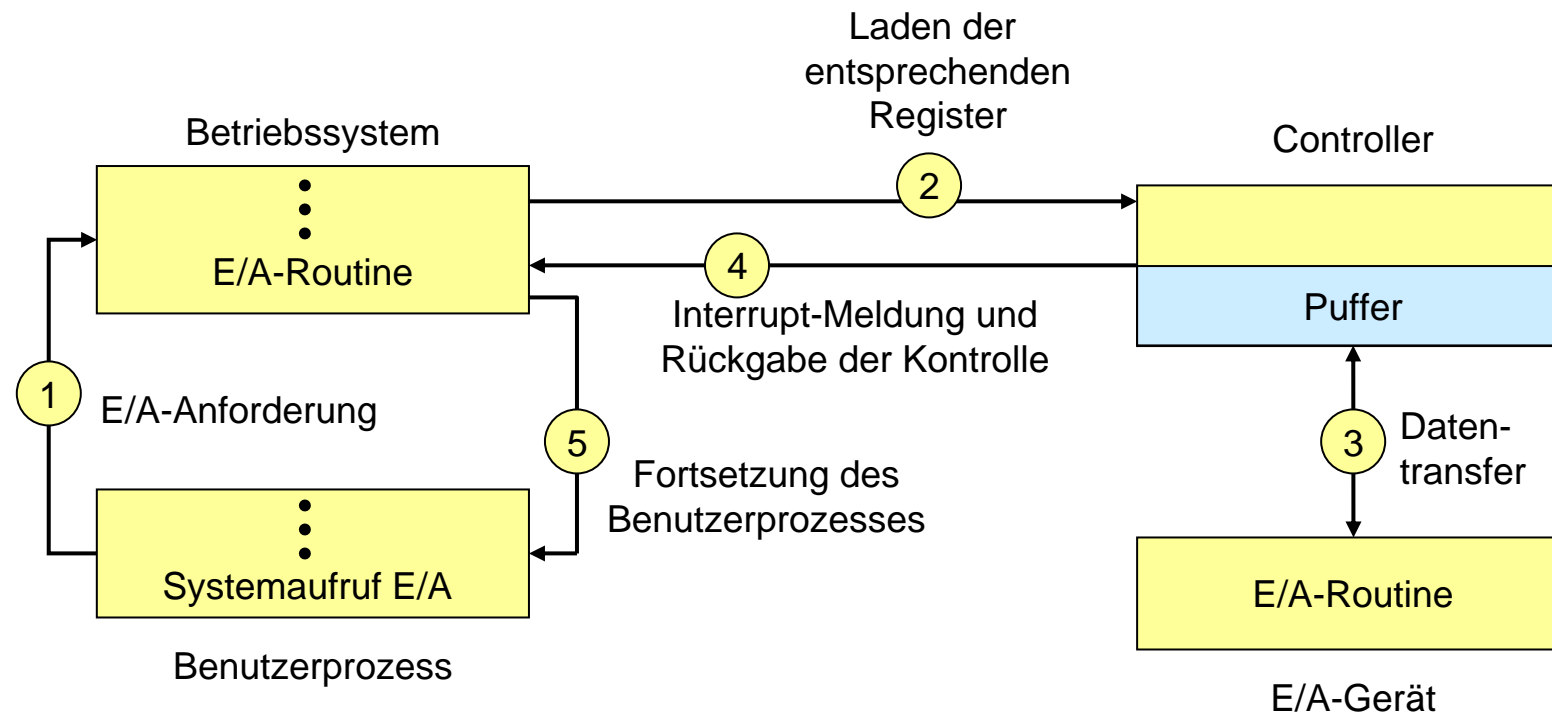
2. Offline-Operationen mit I/O-Geräten



1.3 Aufbau eines Rechners / Grundstruktur

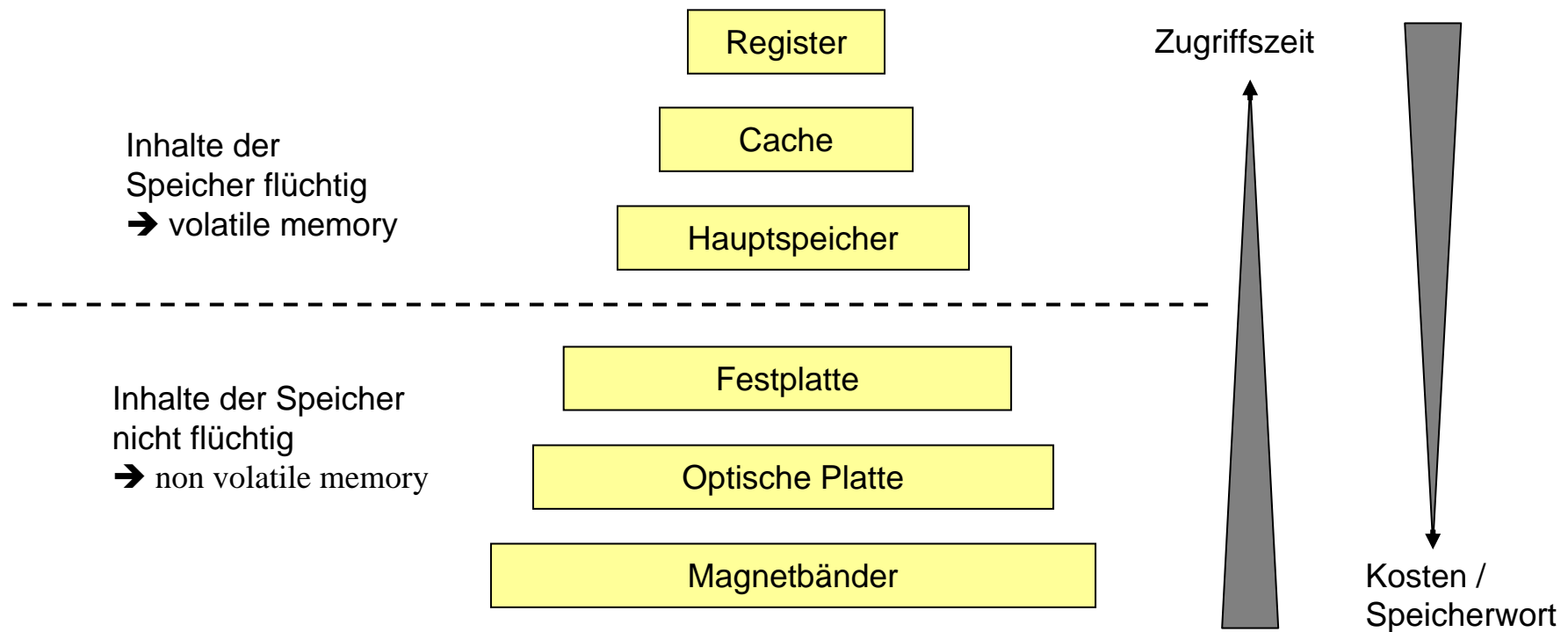


1.3 Aufbau eines Rechners / Ablauf einer E/A-Operation



- Während ② - ④ schaltet das Betriebssystem auf andere Prozesse um
- Synchrone vs. asynchrone Ein-/Ausgabe
- Direct Memory Access (DMA)

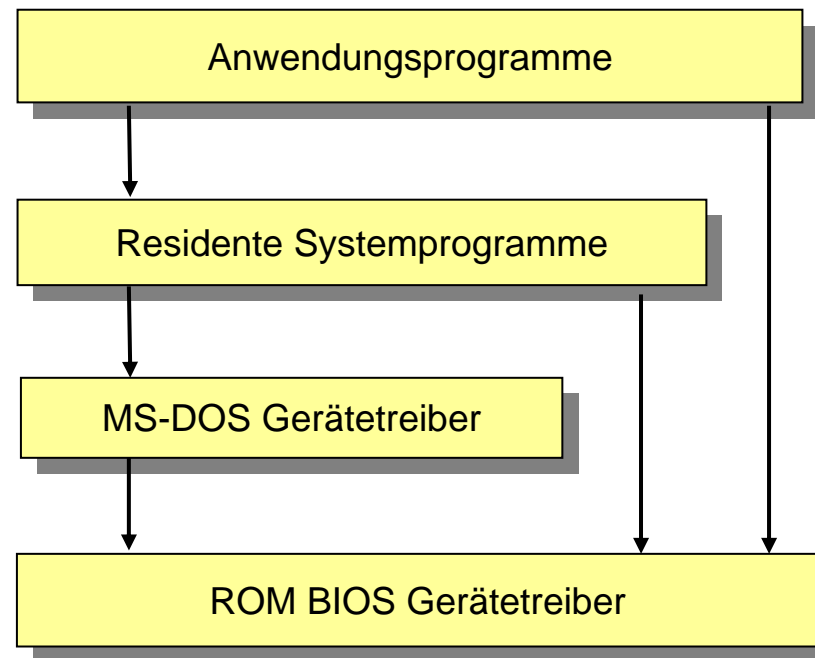
1.3 Aufbau eines Rechners / Speicherstrukturen



1.4 Aufbau eines Betriebssystems

Systemkomponenten	<ul style="list-style-type: none"> - Prozessverwaltung - Hauptspeicherverwaltung - Sekundärspeicherverwaltung - Dateiverwaltung - Kommandointerpreter
Systemaufrufe	<ul style="list-style-type: none"> - Prozesskontrolle - Dateimanipulation - Gerätemanipulation - Kommunikation - Informationsabruf
Systemprogramme	<ul style="list-style-type: none"> - Compiler - Texteditoren - ...

1.5 Beispielarchitekturen / MS-DOS



Nachteile:

- keine Unterscheidung zwischen User- und Supervisor-Mode
- kein Multitasking

1.5 Beispielarchitekturen / UNIX

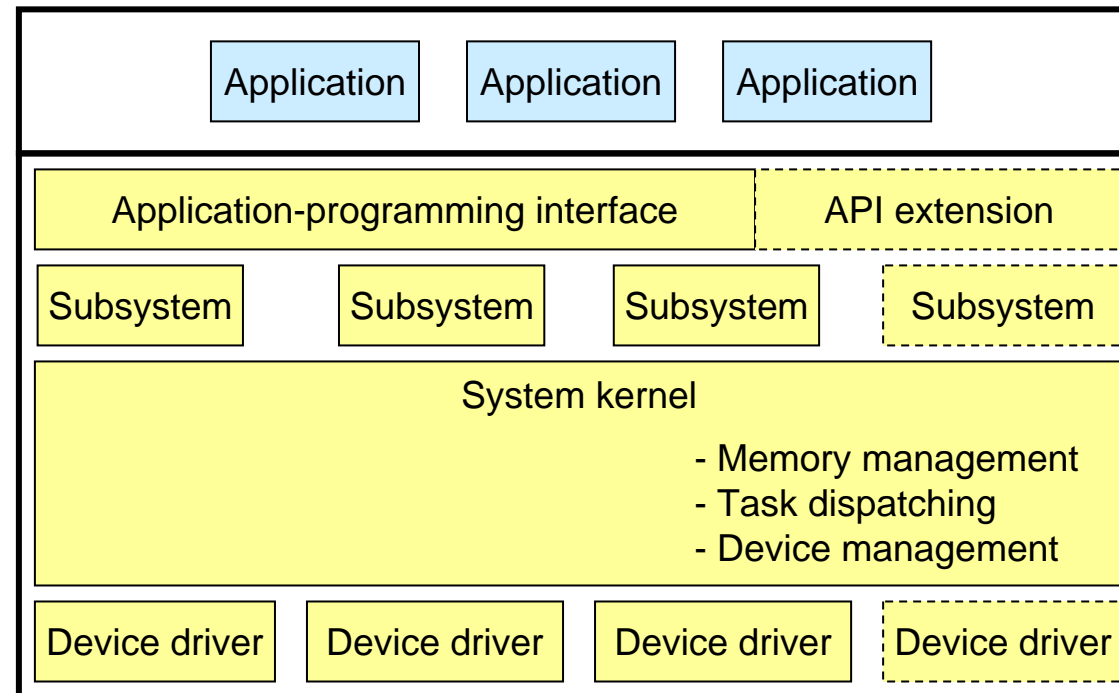
user		
shells and commands compilers and interpreters system libraries		
system-call interface to kernel		
signals terminal handling character I/O system terminal drivers	file system swapping block I/O services disk and tape drivers terminal drivers	CPU schedulers page replacement demand paging virtual memory
kernel interface to the hardware		
terminal controllers terminals	device controllers disk and tapes	memory controllers physical memory

Vorteile

- Unterscheidung zwischen User- und Supervisor-Mode
- Multitasking



1.5 Beispielarchitekturen / OS2



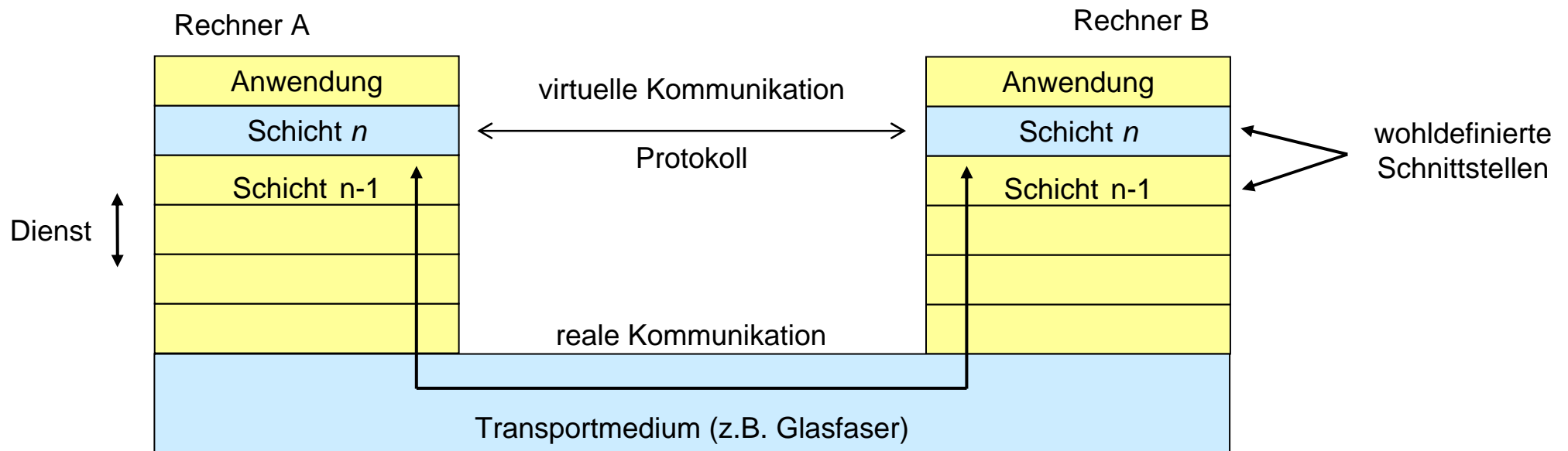
Vorteile

- Unterscheidung zwischen User- und Supervisor-Mode
- Multitasking

1.5 Beispielarchitekturen / Schichtenkonzept

Wenn eine Aufgabe allzu komplex wird, dann zerlegt man sie zweckmäßigerweise in Schichten. Die Kommunikation von A mit B wird durch Dienste bewerkstelligt. Durch mehrere Schichten wird die Aufgabenmenge einer Schicht viel übersichtlicher.

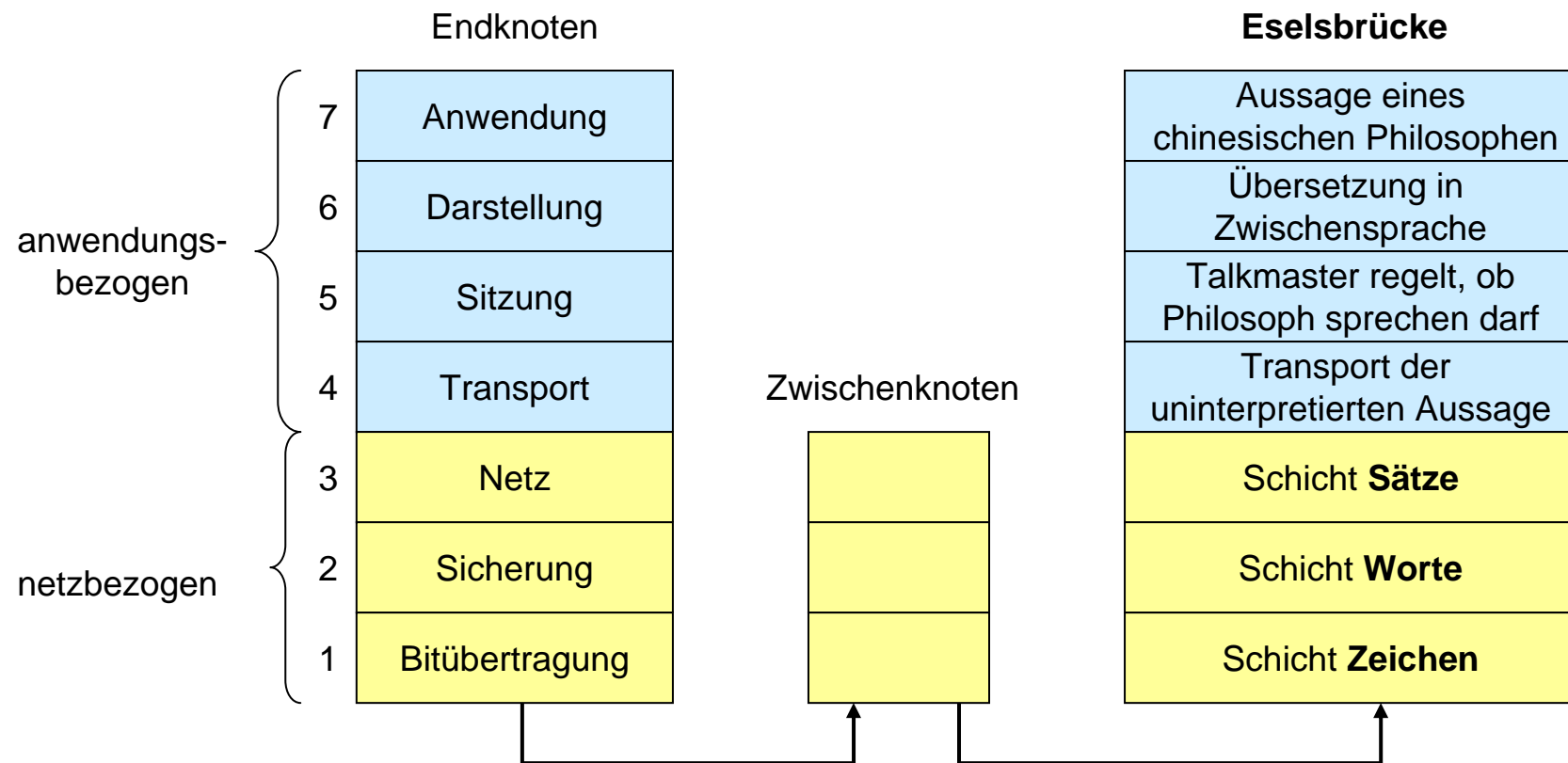
- **Vorteil:** übersichtlich
- **Nachteil:** mehr Aufwand (Overhead), jede Schicht verursacht Zusatzaufwand



1.5 Beispielarchitekturen / Schichtenkonzept

ISO (International Standard Organisation) /

OSI (Open Systems Interconnection) Referenzmodell



Inhaltsverzeichnis

2.1 Was ist ein Prozess?

- Definition
- Prozesszustände
- Prozesskontrollblöcke

2.2 Scheduling

- Singletasking
- Multiprogramming
- Multitasking

2.3 Interprozesskommunikation

- Message-Passing
- Shared-Memory

2.4 Thread-Systeme

- Sinn und Zweck
- Thread-Arten
- Thread-Management

2.1 Was ist ein Prozess?

Prozess:

- Ein Programm im Stadium der Ausführung; die Reihenfolge der Ausführung ist sequenziell.
- Prozess, Task und Job sind für uns synonym.

Oft größere Anzahl von nebeneinander existierenden Benutzer- und Systemprozessen

- Benutzerprozesse haben normalerweise eine niedrigere Priorität als Systemprozesse

Prozess-Ressourcen:

- Speicherplatz
- CPU-Zeit
- Zugriff auf E/A-Geräte

Effiziente Koordination notwendig bei gleichzeitigem Zugriff mehrerer Prozesse auf die gleiche Ressource

2.1 Was ist ein Prozess?

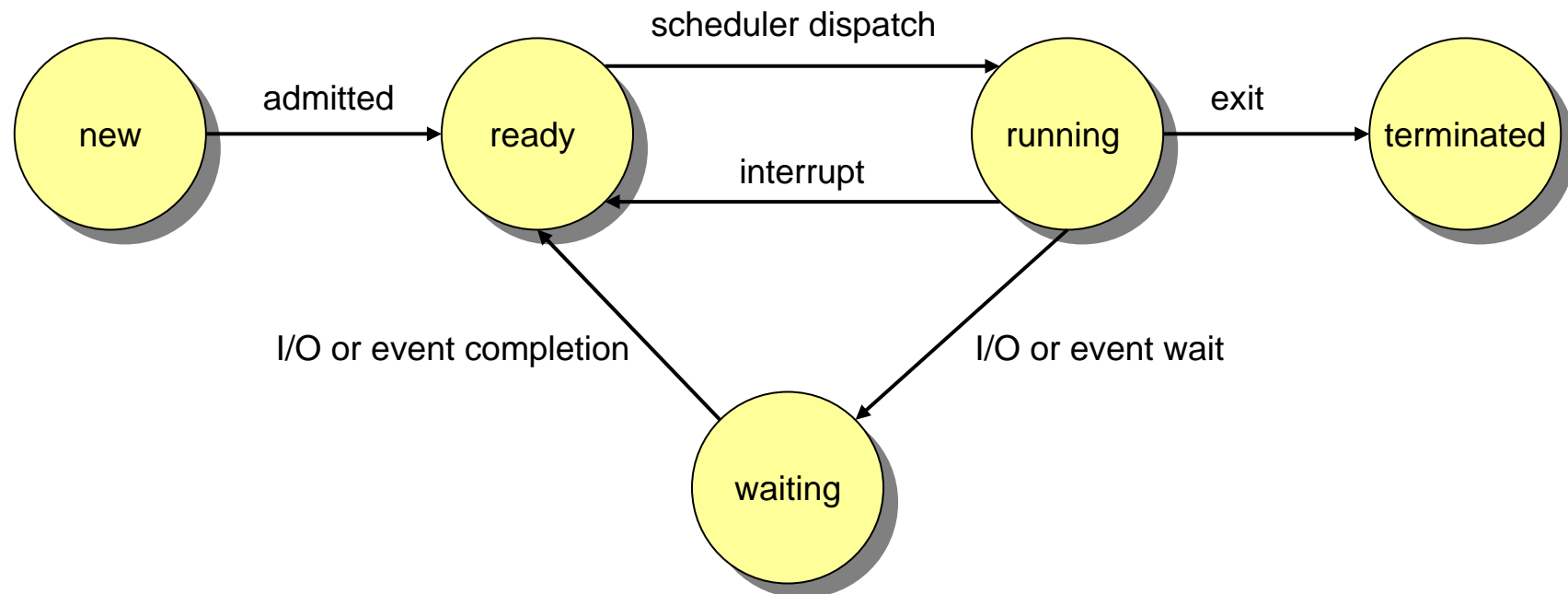
Prozesse befinden sich in unterschiedlichen Zuständen:

Zustand	Bedeutung
running	Anweisungen des Prozesses werden gerade ausgeführt
ready	Prozess ist bereit zur Ausführung, wartet aber noch auf freien Prozessor
waiting	Prozess wartet auf das Eintreten eines Ereignisses (z.B. darauf, dass ein von ihm belegtes Betriebsmittel fertig wird)
blocked	Prozess wartet auf ein fremdbelegtes Betriebsmittel
new	Prozess wird erzeugt (kreiert)
killed	Prozess wird (vorzeitig) abgebrochen
terminated	Prozess ist beendet (alle Instruktionen sind abgearbeitet)

Zu jedem Zeitpunkt kann auf einem beliebigen Prozessor nur ein Prozess laufen!

2.1 Was ist ein Prozess?

Übergänge zwischen Zuständen werden mit Hilfe von Prozesszustands-Diagrammen dargestellt.



2.1 Was ist ein Prozess?

Betriebssystem ist u.a. zuständig für

- Erzeugung und Beendigung von Benutzer- und Systemprozessen
- Aufteilung von Speicherplatz und CPU-Zeit
- Bereitstellung von Mechanismen zur Synchronisation und Kommunikation zwischen Prozessen
- Vorgehen in Deadlock-Situationen (Vorbeugen, Vermeiden, Entdecken)

Repräsentation von Prozessen innerhalb des Betriebssystems erfolgt über Prozesskontrollblöcke (**Process Control Block, PCB**)

PCBs enthalten alle für das Betriebssystem relevanten Informationen.

2.1 Was ist ein Prozess?

Beispiel für Informationen in einem Prozesskontrollblock (PCB)

PCB-Feld	Bedeutung
Status	Prozesszustand
Programmzähler	enthält die Adresse der nächsten auszuführenden Instruktion
CPU-Scheduling	hier finden sich z.B. Prioritäten, Zeiger auf Warteschlangen und ähnliche Scheduling-Parameter
Speichermanagement	enthält die letzten Werte der Speicherregister, Seitentabellen und ähnliche Informationen für die Speicherverwaltung
E/A-Status	Liste von zugeordneten E/A-Geräten, offenen Files usw.
Accounting	benötigte CPU- und Realzeit, Zeitbeschränkungen, Prozessnummern u.ä.

2.2 Scheduling

Hauptspeicher enthält Betriebssystemkern und darauf aufsetzend den Kommandointerpreter (Command Interpreter - CI).

Grobe Einteilung von Betriebssystemen mittels ihrer Art, Prozesse zu behandeln.

Singletasking	Multiprogramming / Multitasking
Prozess wird bei Aufruf exklusiv in den Speicher eingelesen	Mehrere Prozesse gleichzeitig im Speicher
<p>Speicher</p> <p>Speicher</p> <p>frei</p> <p>Prozess (z.T. kann CI Überschrieben werden)</p> <p>CI</p> <p>BS-Kern</p>	<p>Speicher</p> <p>P3</p> <p>frei</p> <p>P2</p> <p>P1</p> <p>CI</p> <p>BS-Kern</p>

2.2 Scheduling

Das Scheduling regelt den Zugriff der Prozesse auf die Ressourcen, insbesondere auf die CPU.

- Dazu gibt es eine Vielzahl möglicher Strategien, z.B. FIFO (First In First Out)

Scheduling auf verschiedenen zeitlichen Ebenen:

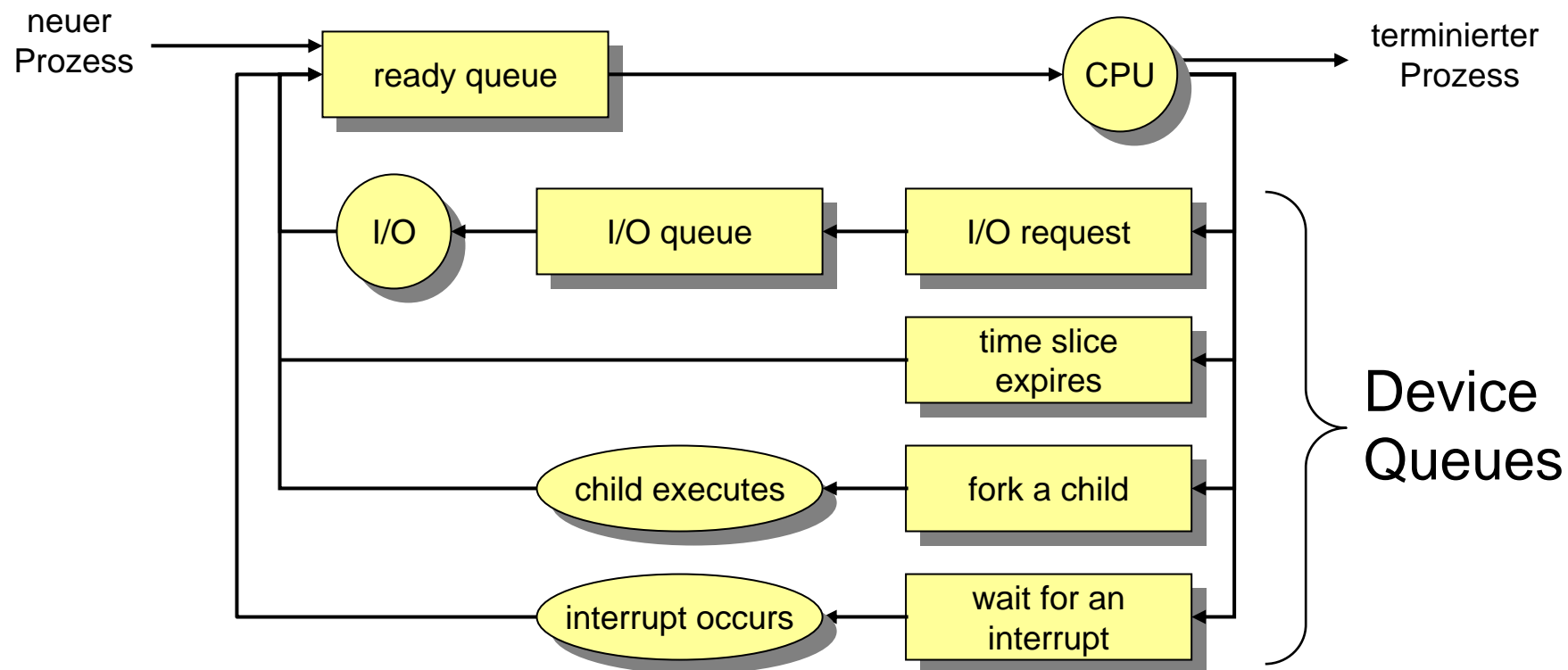
- Langzeit-Scheduler (**Job-Scheduler**) wählt Prozesse aus, die als nächstes in den Speicher geladen werden.
- Kurzzeit-Scheduler (**CPU-Scheduler**) entscheidet, wann bereits geladene und sich im Zustand **ready** befindliche Prozesse auf die CPU zugreifen dürfen.

Verschiedene Warteschlangen werden durch das Scheduling verwaltet:

- **Job-Queue** enthält alle Prozesse des Systems.
- **Ready-Queue** enthält alle im Hauptspeicher befindlichen Prozesse, die sich im Zustand **ready** befinden.
- **Device-Queues** enthalten alle Prozesse, die auf ein E/A-Betriebsmittel warten.

2.2 Scheduling

Prozesse migrieren zwischen den Warteschlangen (Central Server Model)



2.2 Scheduling

Unterschied zwischen **Multiprogramming** und **Multitasking**

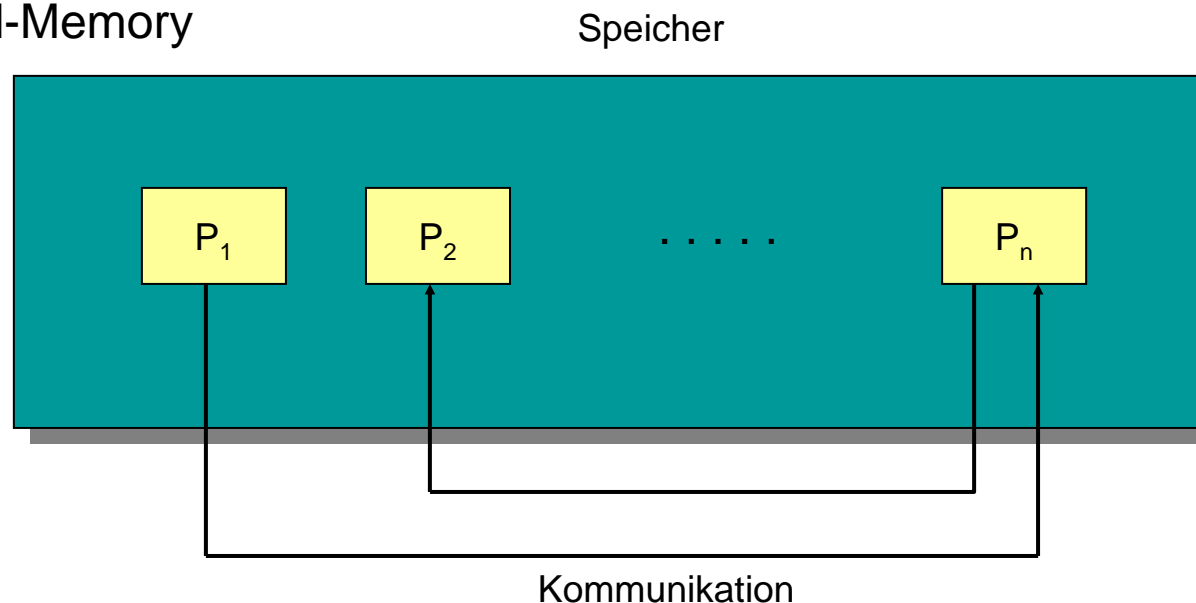
- Beim **Multiprogramming** kann der Benutzer während seiner CPU-Nutzung nicht unterbrochen werden; das kann bei rechenintensiven Prozessen zur Blockade des Systems führen.
- Beim **Multitasking (time sharing)** kann ein Prozess nicht nur bei einem Interrupt unterbrochen werden, sondern z.B. auch nach Ablauf einer bestimmten Zeitspanne. Dadurch wird interaktiver Betrieb durch mehrere Benutzer möglich, wobei jeder die Illusion hat, die CPU arbeite exklusiv für ihn.

2.3 Interprozesskommunikation

Koexistenz bzw. Zusammenarbeit verschiedener Prozesse erfordert Möglichkeit der Kommunikation zwischen den Prozessen.

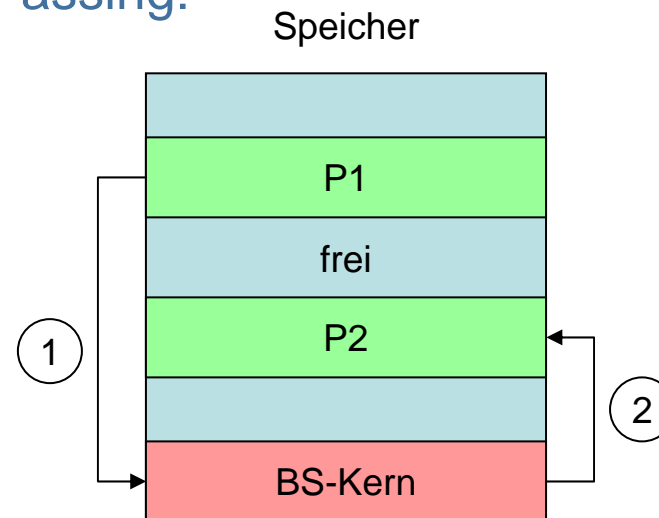
Zwei verschiedene Ansätze:

- Message-Passing
- Shared-Memory



2.3 Interprozesskommunikation / Message-Passing

- Betriebssystem baut Verbindung zwischen Prozessen auf
- Prinzip des Message-Passing:

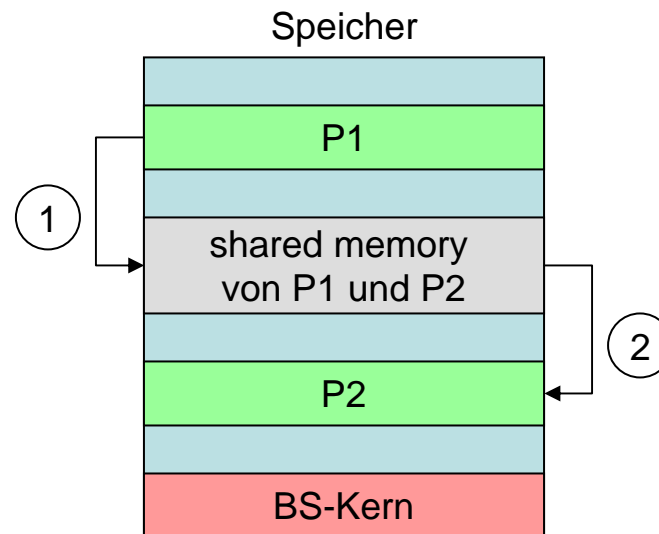


- Unterschiedliche Eigenschaften dieser Verbindung
 - gehört einem oder mehreren Prozessen
 - unterschiedliche Kapazität
 - unterschiedliche Nachrichtenlängen
 - uni- oder bidirektional

2.3 Interprozesskommunikation / Shared-Memory

Temporäre Aufhebung der exklusiven Benutzung von Speicherbereichen

- Prozess P1 speichert Nachricht direkt in Speicher, auf den auch P2 Zugriff hat:

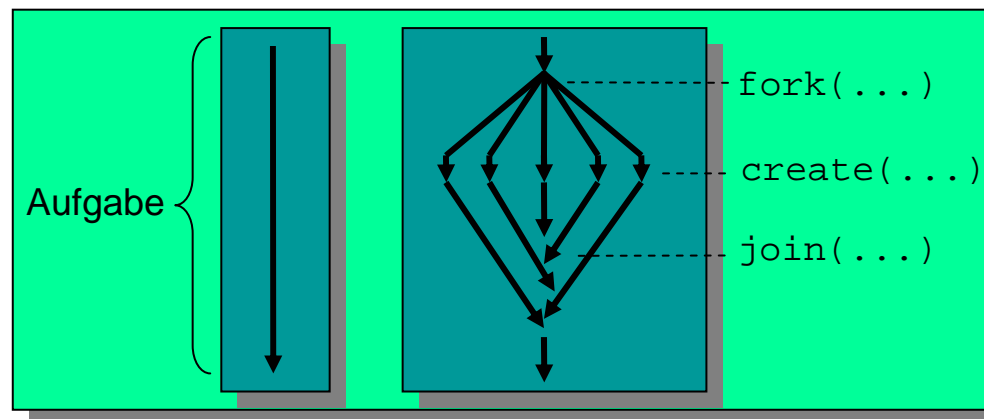


- Vorteil: bei großen Datenmengen schneller als Message-Passing
- Nachteil: Synchronisations- bzw. Konsistenzprobleme
(z.B. liest P2 bereits, während P1 noch schreibt)
- Lösung: wechselseitiger Ausschluss, d.h. nur ein einziger Prozess darf "Shared Memory" zu einer gegebenen Zeit bearbeiten. Dies erfordert Sperr- und Freigabemechanismen.

2.4 Thread-Systeme

Thread (eigentlich Faden) auch Leight Weight Process

- Nützlich vor allem bei paralleler Programmierung



Ziel: Aufgabe soll schneller bearbeitet werden!

Probleme:

- Zerlegung in unabhängige Teil-Threads oft schwierig oder unmöglich
- Management eines solchen Geflechts sowie der zusätzliche Kommunikationsaufwand
- Effizienzprobleme, wenn z.B. ein Thread deutlich früher fertig ist als ein anderer, mit dem er ein `join(...)` machen muss

2.4 Thread-Systeme

Es gibt vier Arten von Threads

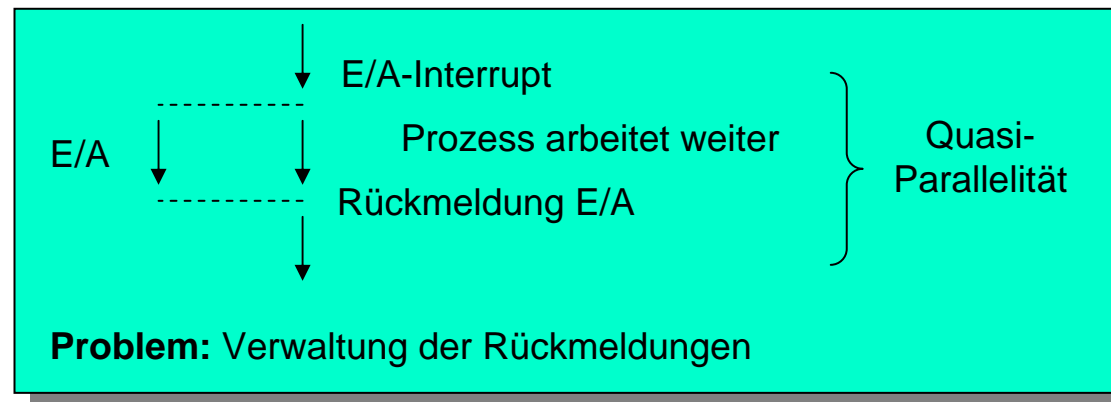
T1 Sequenziell, d.h. ohne Parallelität

T2 Prozesse parallel ausführen (**Shared-Memory-Prinzip**)

Problem: Datenzugriffskonflikte - jeder Prozess hat seinen eigenen Adressraum

Fehlende **Kapselung**, d.h. Parallelität wird nicht nach außen verborgen

T3 Ein Prozess mit asynchronen Systemaufrufen



T4 Ein Prozess wird in mehrere Teile zerlegt und parallel ausgeführt (auf Multiprozessor-System), wodurch echte Parallelität möglich wird.

2.4 Thread-Systeme

Übersicht der Programmiermodelle

Thread Art	T1	T2	T3	T4
Anzahl Prozessoren	1	n	1	n
Anzahl Adressräume	1	n	1	1
Systemaufrufe	synchron	synchron	asynchron	synchron
Programmieraufwand	++	0	-	+
Parallelität	-	++	+	++
Kapselung	+	-	+	+
Ressourcenverbrauch	+	-	+	+

2.4 Thread-Systeme / Thread-Management

Standardaufrufe:

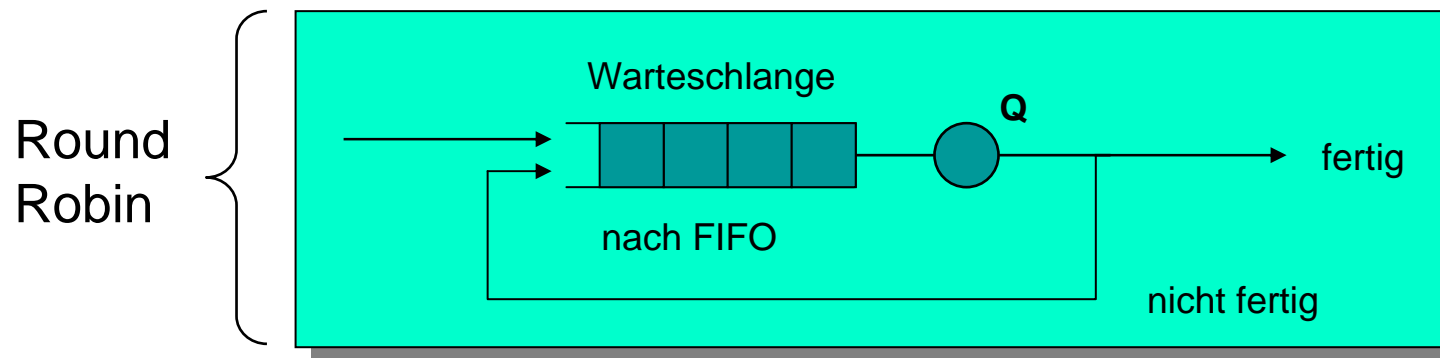
Name	Beschreibung
self ()	Liefert die ID des aktuellen Threads
create (name)	Erzeugt einen neuen Thread
exit ()	Beendet den aktuellen Thread
join ()	Wartet auf das Ende eines anderen Threads, liefert evtl. dessen Rückgabewert
detach ()	Löst einen anderen Thread von den übrigen, d.h. kein anderer Thread wird auf ihn warten
cancel ()	Beendet einen anderen Thread (asynchron)
Abbruchmaskierung	Erlaubt einem Thread, Bereiche anzugeben, in denen er ein ihn betreffendes cancel() aufschiebt (maskiert)
Abbruchvorsorge	Erlaubt einem Thread, Aufräumfunktionen (sog. cleanup-handler) anzugeben, die im Falle eines unmaskierten cancel() noch ausgeführt werden.

2.4 Thread-Systeme

Scheduling von Prozess-/Thread-Systemen: Zwei **diametrale** Strategien

- FIFO: First In First Out
- RR: Round Robin (Zeitscheibe, Q)
Timeslice Quantum

- $Q \rightarrow 0$: Round Robin \rightarrow Processor Sharing (PS), falls Verwaltungsaufwand vernachlässigbar ist
- $Q \rightarrow \infty$: Round Robin \rightarrow FIFO



Priority Scheduling

- wähle den Prozess/Thread mit aktuell höchster Priorität. Priorität ist z.B. gegeben durch Wartezeit, Dringlichkeit, Tarif, ...

2.4 Thread-Systeme

Spezielle Probleme mit Prioritäten beim Priority Scheduling

- a) Monopolisierung durch wenige Endlosprozesse höchster Priorität
- b) Priority Inversion: Prozesse niedriger Priorität laufen normalerweise langsamer als solche mit höherer Priorität. Wenn ein niederpriorer Prozess ein Eintrittssignal bekommt, auf das ein hochpriorer Prozess wartet, wird der hochpriorer Prozess verlangsamt.

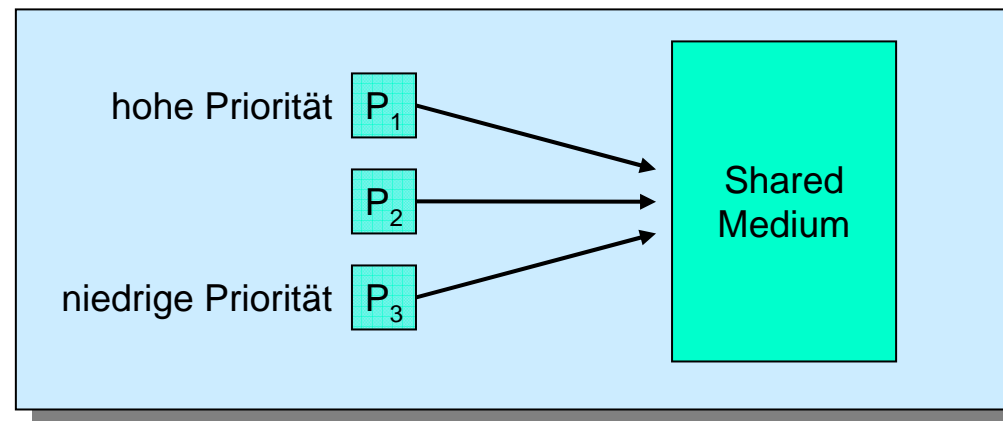
Lösung für b) könnte priority inheritance sein, d.h. die Priorität wird vererbt

- Prozess, der das Eintrittssignal besitzt, wird zeitweise auf die Priorität hochgestuft, welche der höchstpriorer auf das Signal wartende Prozess hat.

2.4 Thread-Systeme

Prioritätsumkehrung auch sonst nicht selten

➤ Beispiel: Kommunikationsprotokolle mit Zugriff auf gemeinsame Medien



Es ist möglich, dass ein hochpriorer Prozess häufiger zugreift, aber trotzdem langsamer wird. Betrachte folgende Situation:

- P_1 macht 16 schnelle Zugriffe, die alle vergeblich sind, und führt dann ein Reset durch.
- P_3 macht dann 1 langsamen aber erfolgreichen Zugriff.

2.4 Thread-Systeme

Typen von Threads:

- **User Threads** sind vom Benutzer frei definierbar und haben Vorteile bzgl. Flexibilität und Parallelität ➡ niedrigere Priorität
- **Kernel Threads** gibt es auf der Ebene des Betriebssystems ➡ höhere Priorität

Nachteil von User Threads:

- Thread-Scheduler auf der User-Ebene kommuniziert meist nicht mit Prozess-Scheduler auf Betriebssystem-Ebene

Prozess, welcher verschiedene Threads initialisiert hat, wird beendet,

- wenn Initial-Thread beendet wird
- wenn letzter Thread (nicht notwendigerweise Initial-Thread) fertig ist

Inhaltsverzeichnis

3.1 Erzeuger-Verbraucher-Problem

- Allgemeine Lösung
- Implementierung

3.2 Wechselseitiger Ausschluss

- Bakery-Algorithmus
- Enqueue/Dequeue
- Synchronisation

3.3 Semaphore

- Konzept
- Erzeuger-Verbraucher-Problem
- Reader-Writer-Problem
- Fünf-Philosophen-Problem

3.4 Petrinetze

3.5 Bedingte kritische Regionen

- Lösung des Erzeuger-Verbraucher-Problems
- Implementierung

3.6 Monitore

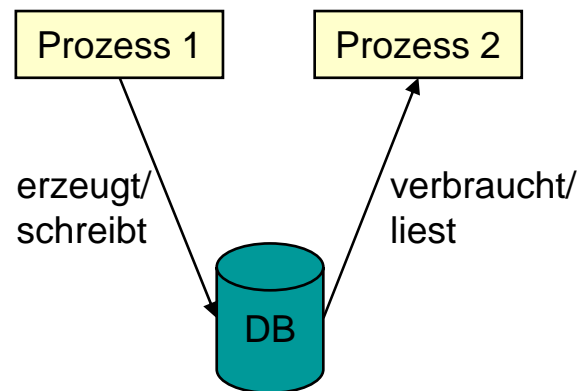
- Konzept
- Beispiellösungen für Synchronisationsprobleme

3.7 Datenbankorganisation

Motivation

Zur Lösung eines Problems werden oft mehrere Prozesse benötigt.

→ Prozesse können voneinander abhängig sein!



Damit die Daten immer konsistent bleiben, müssen die Prozesse synchronisiert werden, d.h. der Zugriff auf die gemeinsamen Daten muss gesteuert werden!

3.1 Erzeuger-Verbraucher-Problem / Einleitung

Prozess E erzeugt etwas und Prozess V verbraucht es; notwendig:
(Zwischen-)Lager

- Wenn das Lager voll ist, kann E nichts ablegen
- Wenn das Lager leer ist, kann V nichts entnehmen
- Der Lagerbestand (S) kann nicht gleichzeitig von E und V verändert werden

Abstrakte Beschreibung der Kommunikation zwischen Prozessen durch Ein- und Ausgabevorgänge unter Verwendung eines Lagers mit einer Kapazität von MAX Einheiten.

Erzeuger

repeat
erzeuge Element;
while (S = MAX) do noop;
lege Element in Puffer ab;
S := S + 1;
until FALSE;

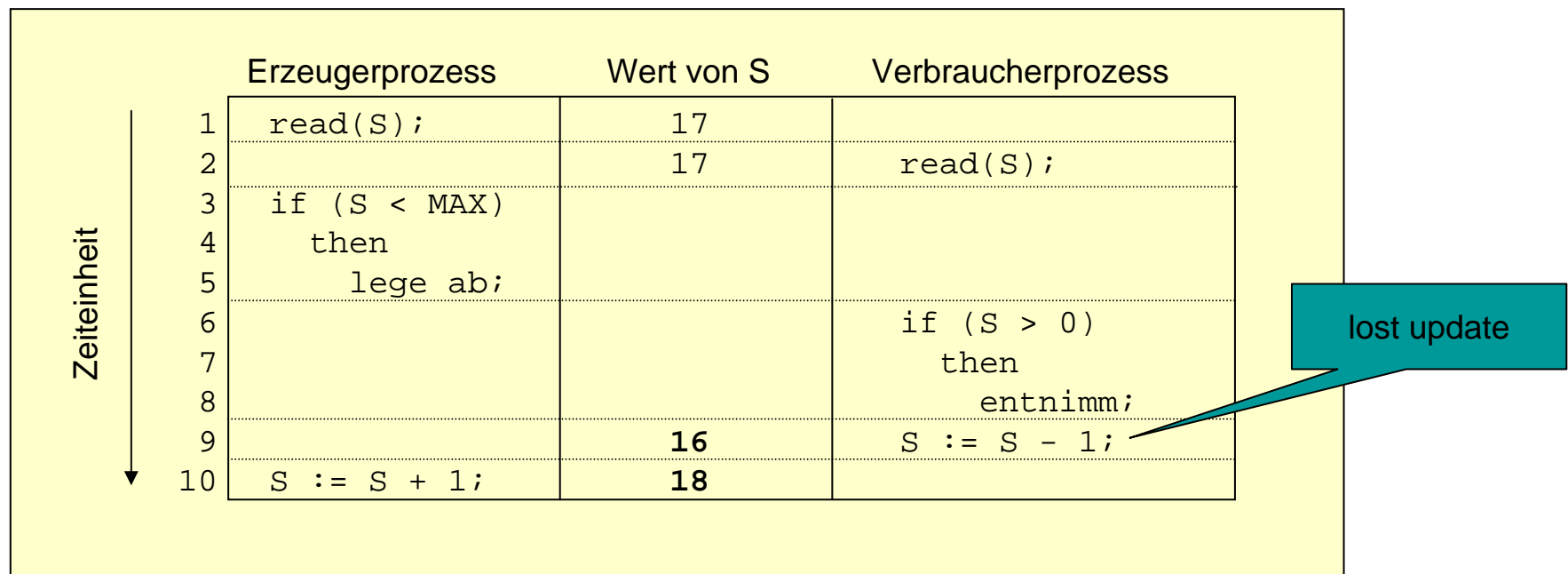
Verbraucher

repeat
while (S = 0) do noop;
entnimm Element aus Puffer;
S := S - 1;
verbrauche Element;
until FALSE;

kritisch unkritisch

3.1 Erzeuger-Verbraucher-Problem / Allgemein

Problem: Erzeuger legt Produkt ab, kommt aber nicht mehr dazu, den Bestand (S) zu korrigieren

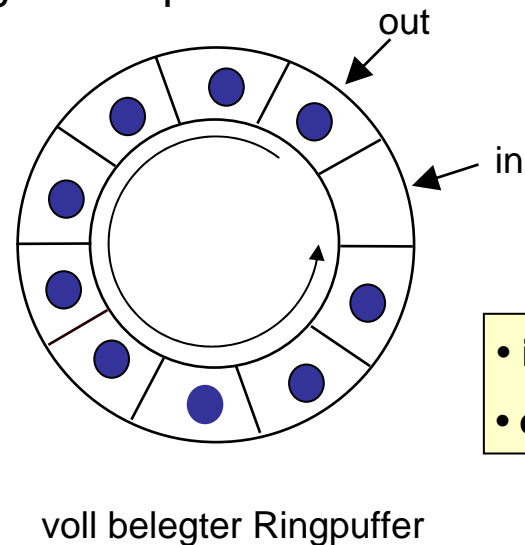
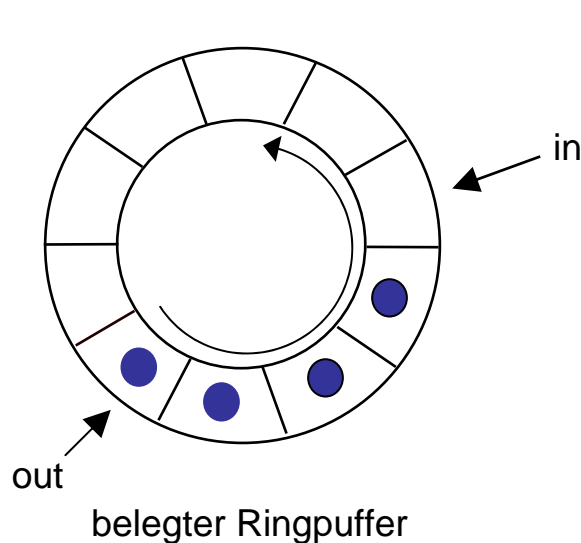


3.1 Erzeuger-Verbraucher-Problem / Allgemein

Realisierung mittels Ringpuffer (ringförmiges Diamagazin)

Idee:

- E legt sein Produkt im ersten freien Pufferplatz ab
- V entnimmt Produkt aus dem ältesten belegten Platz
- Puffer hat maximal MAX-1 belegte Komponenten



- **in** wird nur von **E** geändert
- **out** wird nur von **V** geändert

3.1 Erzeuger-Verbraucher-Problem / Implementierung

Erzeuger

```
repeat  
  produziere ein Element und lege es in nextp ab;      (* Zwischenlager *)  
  while (in+1 mod MAX = out) do noop;                (* Puffer voll *)  
  buffer[in] := nextp;                                  (* ablegen *)  
  in := (in+1) mod MAX;                                 (* Zeigerkorrektur *)  
until FALSE
```

busy waiting

Verbraucher

```
repeat  
  while (in = out) do noop;                            (* Speicher leer *)  
  nextc := buffer[out];                                  (* entnehmen *)  
  out := (out+1) mod MAX;                                (* Zeigerkorrektur *)  
  verbrauche das in nextc enthaltene Element;  
until FALSE
```

Lösung funktioniert nur bei einem einzigen Erzeuger und Verbraucher!

3.1 Erzeuger-Verbraucher-Problem / Implementierung

Bei mehreren Erzeugern und Verbrauchern werden gemeinsame Datenbestände von mehreren Prozessen gleichzeitig benutzt.

Erzeuger

```
repeat
  produziere ein Element und lege es in nextp ab;
  while (counter = MAX) do noop;
  buffer[in] := nextp;
  in := (in+1) mod MAX;
  counter := counter + 1;
until FALSE;
```

Verbraucher

```
repeat
  while (counter = 0) do noop;
  nextc := buffer[out];
  out := out + 1 mod MAX;
  counter := counter - 1;
  verbrauche das in nextc enthaltene Element;
until FALSE;
```

**Unkorrekte Lösung, da
counter von beiden
geändert werden kann.**

3.1 Erzeuger-Verbraucher-Problem / Implementierung

Problem: Abarbeitung des Füllgrades

Operation `counter := counter ± 1` ist teilbar:

```
register1 := counter;
register1 := register1 ± 1;
counter   := register1;
```

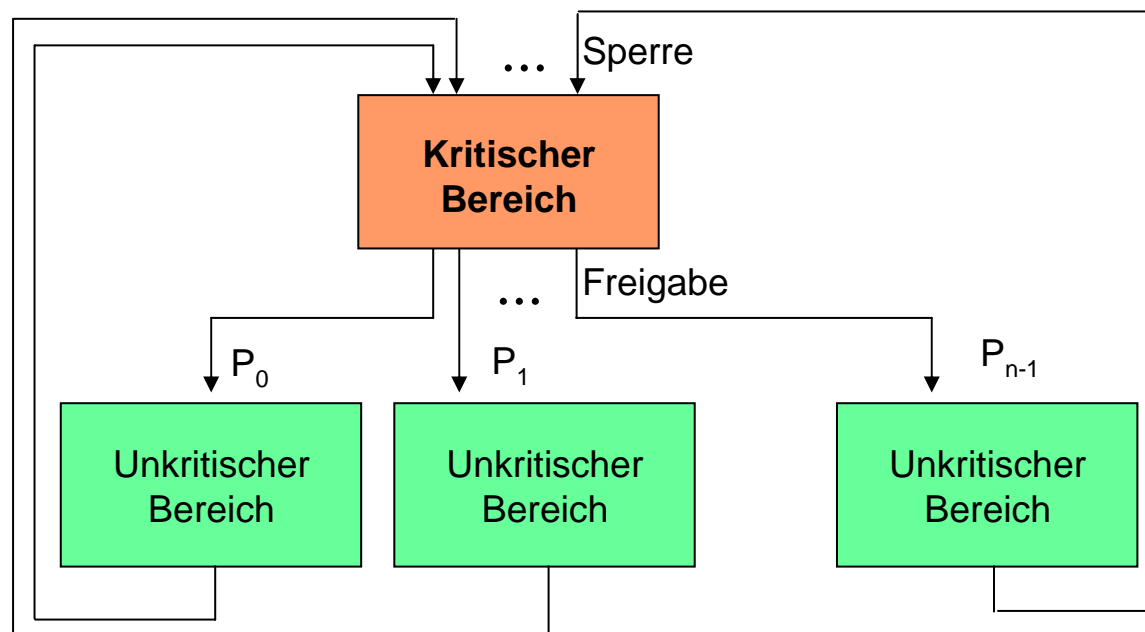
Wirkung: (`counter := counter ± 1`)
aufgeteilt in 3 Operationen, d.h. **nicht atomar**.

Taktzyklus	Prozess	Anweisung	Resultat
1	E	<code>register1 := counter</code>	<code>register1 = 5</code>
2	E	<code>register1 := register1+1</code>	<code>register1 = 6</code>
3	V	<code>register2 := counter</code>	<code>register2 = 5</code>
4	V	<code>register2 := register2-1</code>	<code>register2 = 4</code>
5	E	<code>counter := register1</code>	<code>counter = 6</code>
6	V	<code>counter := register2</code>	<code>counter = 4</code>

3.2 Wechselseitiger Ausschluss

Kritischer Bereich: Phase, in der ein Prozess gemeinsame (**globale**) Daten benutzt.

- Verschiedene Prozesse P_0, \dots, P_{n-1} greifen im **kritischen Bereich** auf globale Daten zu (analog unkritischer Bereich).
- Befinden sich zwei Prozesse gleichzeitig in ihrem kritischen Bereich, so ist das Resultat von zufälligen Umständen abhängig.



→ zu jedem Zeitpunkt darf sich maximal ein Prozess im kritischen Bereich befinden !

3.2 Wechselseitiger Ausschluss / Allgemein

Zur Lösung des wechselseitigen Ausschlusses sind notwendig:

- **Sperrfunktion:** Erlaubt nur einem einzigen Prozess das Betreten des kritischen Bereichs
- **Freigabefunktion:** Signalisiert Verlassen des kritischen Bereichs
- **Atomare Funktionen:** Funktionen können während ihrer Ausführung nicht unterbrochen werden

Korrekte Lösung muss folgenden 3 Kriterien genügen:

- **Mutual exclusion:** zu jedem Zeitpunkt darf **höchstens ein Prozess** in seinen kritischen Bereich.
 - **Progress:** befindet sich kein Prozess im kritischen Bereich, und gibt es andererseits Prozesse, die ihren kritischen Bereich betreten möchten, so hängt die Wahl des nächsten Kandidaten nur von den Kandidaten ab und fällt in **endlicher** Zeit.
 - **Bounded waiting:** für einen Prozess, der den kritischen Bereich betreten möchte, muss eine obere Grenze für die Zeit, die er warten muss, angegeben werden können.
 - ➔ jeder Prozess muss seinen kritischen Bereich in **endlicher** Zeit wieder verlassen.
- (➤Außerdem oft gefordert: Fairness ohne Prioritäten)

3.2 Wechselseitiger Ausschluss / Allgemein

Zunächst einige **falsche** Ansätze:

Atomare Operationen: Einfache Lese-/Schreibzugriffe, z.B. $x := \text{FALSE}$

Variante 1 für Prozess P_i (analog P_j)

```
repeat
  unkritischer Bereich;
  while (turn  $\neq$  i) do noop;
  kritischer Bereich;
  turn := j;
until FALSE;
```

Problem:

- Falls (**turn = j**) und P_j im unkritischen Bereich stirbt, kann P_i nie mehr in seinen kritischen Bereich.
- Ein **langsamer** Prozess P_i könnte einen **schnellen** Prozess P_j erheblich behindern.

Variante 2 für Prozess P_i (analog P_j)

```
flag[i] := FALSE;
repeat
  unkritischer Bereich;
  flag[i] := TRUE;
  while flag[j] do noop;
  kritischer Bereich;
  flag[i] := FALSE;
until FALSE;
```

Problem:

Zeit	P_i	P_j
1	flag[i]:=TRUE	flag[j]:=TRUE while flag[i] do noop
2		
3		
4	while flag[j] do noop	

} Deadlock

3.2 Wechselseitiger Ausschluss / Allgemein

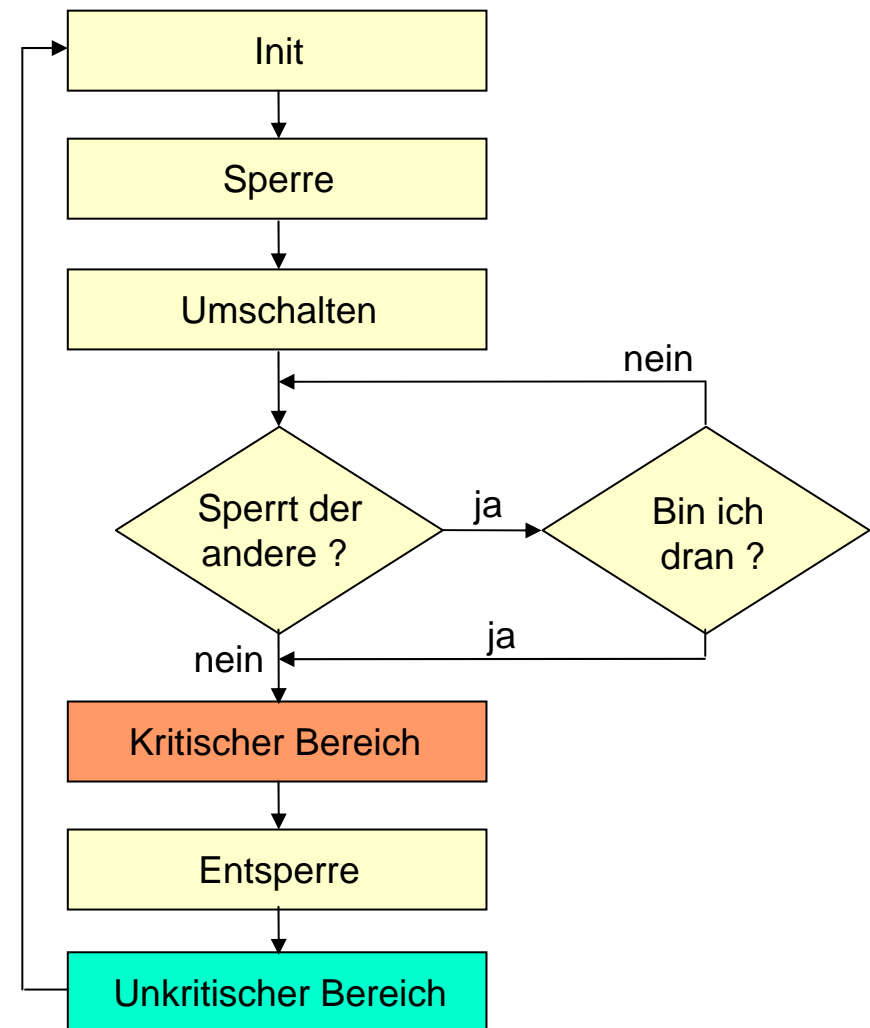
Kombination aus beiden
Varianten ergibt eine korrekte Version

Für P_i (für P_j symmetrisch)

```
repeat
→ flag[i] := TRUE;
→ turn := j;
  while (flag[j] and turn=j) do noop;
  kritischer Bereich;
→ flag[i] := FALSE;
  unkritischer Bereich;
until FALSE;
```

Atomare Operationen hier:
Einfache Lese-/Schreibzugriffe

Verallgemeinerung auf n Prozesse möglich,
aber sehr komplex und unübersichtlich.



3.2 Wechselseitiger Ausschluss / Bakery-Algorithmus

Bakery-Algorithmus arbeitet mit Nummern für wartende Kunden wie z.B. beim Einwohnermeldeamt bzw. einer (großen) Bäckerei

- n Kunden befinden sich in einer Warteschlange
- Jeder Kunde erhält eine steigende Nummer
- Kunde mit der kleinsten Nummer wird bedient

Implementierung

- Kritischer Bereich ist die Ziehung der Nummern.
- Prozess P_i setzt `choosing[i] := TRUE`, wenn er eine Nummer ziehen möchte.
- Nach erfolgter Nummernziehung setzt P_i seinen Feldeintrag zurück: (`choosing[i] := FALSE`).
- Um gleichzeitiges Ziehen von Nummern zu verhindern, fließt ein weiterer Faktor in die Entscheidung ein. Beim Einwohnermeldeamt ist das z.B. der Familienname, hier die Prozessnummer.

3.2 Wechselseitiger Ausschluss / Bakery-Algorithmus

Algorithmus für i-ten Prozess von n Prozessen

Vereinbarung: $(a,b) < (c,d) \leftrightarrow$ entweder $a < c$ oder $(a=c \text{ und } b < d)$

```
repeat
  choosing[i] := TRUE;
  number[i] := max(number[0],...,number[n-1]) + 1;
  choosing[i] := FALSE;
  for j := 0 to n-1 do
    begin
      while choosing[j]
        do noop; (* warte, bis j seine Nummer gezogen hat *)
      while (number[j] <> 0) and ((number[j],j) < (number[i],i))
        do noop; (* j ist „älter“, d.h. früher dran; warten *)
      end;
    Kritischer Bereich;
    number[i] := 0;
    Unkritischer Bereich;
  until FALSE;
```

Problem bei dieser Lösung:
In ungünstigen Fällen können die gezogenen
Zahlen sich ständig vergrößern und
schließlich zu einen Zahlenüberlauf führen.

3.2 Wechselseitiger Ausschluss / Enqueue und Dequeue

Enqueue und Dequeue

- Arbeitet mit atomaren Operationen zur Verwaltung einer FIFO-Warteschlange.
- Warteschlange ist eine verkettete Liste von Prozessnummern, die in einem Array $F[0...n]$ verwaltet werden. $F[0]$ ist der Prozess, der sich aktuell im kritischen Bereich befindet. $F[F[...F[0]]]$ ist die Nummer des ältesten Prozesses.

```

enqueue(i)
{
    F[p] := i;
    p := i;
}

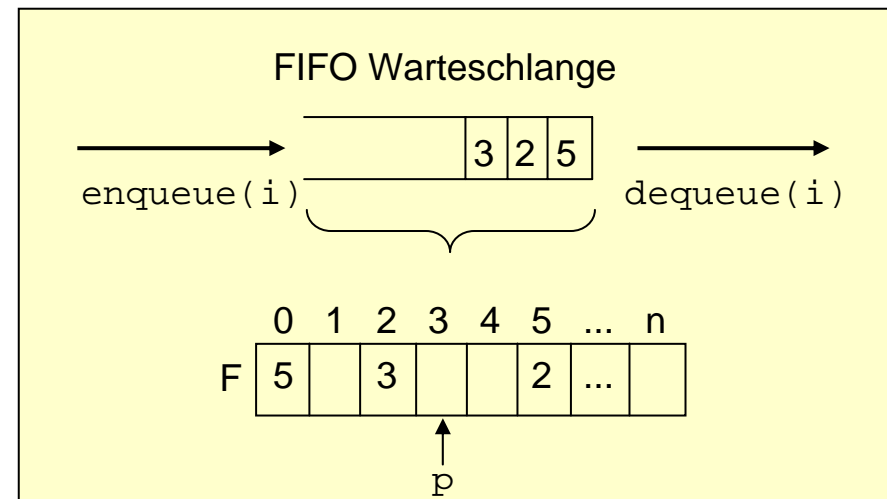
dequeue(i)
{
    if (p ≠ i) then
        F[0] := F[i];
    else p := 0; F[0] := 0;
}
    
```

} atomar !

} atomar !

```

enqueue (i) = Sperre
dequeue (i) = Freigabe
    
```



3.2 Wechselseitiger Ausschluss / Enqueue und Dequeue

Algorithmus für P_i :

```
repeat
  Unkritischer Bereich;
  enqueue(i);
  while (F[0] ≠ i) do noop;
  Kritischer Bereich;
  dequeue(i);
until FALSE;
```

Problem:

Wenn enqueue und
dequeue nicht atomar sind,
dann ist die Lösung nicht
mehr korrekt !

ewiges warten

Prozess P_j	Prozess P_k	Wirkung
$F(p) := j;$		$F(0) := j;$
	$F(p) := k;$	$F(0) := k;$
$p := j;$	$p := k;$	$p := k;$
	$p := j;$	$p := j;$
	while ($F[0] \neq k$) do noop; Kritischer Bereich; if ($p \neq k$) then $F[0] := F[k];$ else { $p := 0; F[0] := 0;$ }	P_k betritt den KB $F[0] := F[k];$
while ($F[0] \neq j$) do noop;		

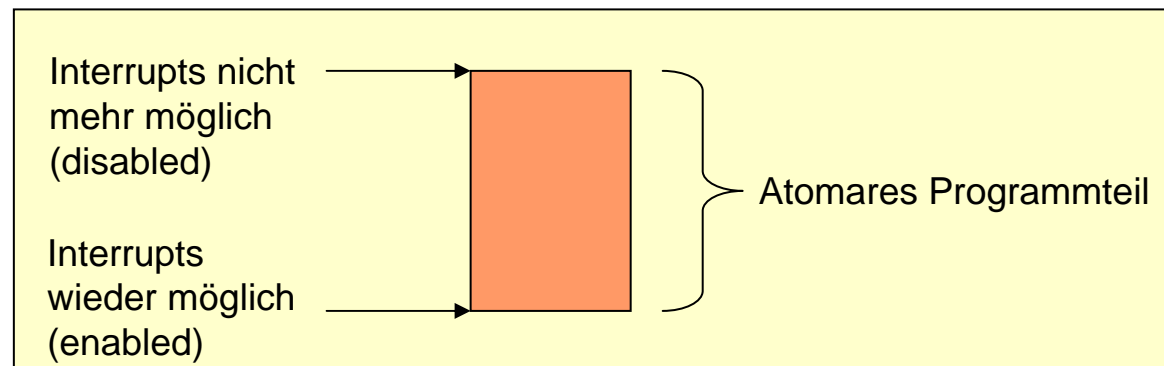
falsch !

3.2 Wechselseitiger Ausschluss / Synchronisation

Ziel: Synchronisation mit atomaren Operationen

Interrupt

- Am einfachsten lassen sich atomare Operationen über die Interrupt-Steuerung realisieren.
- Atomares Programmstück fängt mit einem Befehl an, der Interrupts unmöglich macht, und endet mit einem Befehl, welcher Interrupts wieder ermöglicht.



3.2 Wechselseitiger Ausschluss / Synchronisation

Test-and Set:

- Einige Computer bieten die Möglichkeit, das Testen und Modifizieren von Variablen in einer unteilbaren (atomaren) Operation durchzuführen.

```
function Test-and-Set(var target: boolean): boolean;  
begin  
    Test-and-Set := target;  
    target := TRUE;  
end;
```

Atomare
Operation

Wechselseitiger Ausschluss mit Test-and-Set (Zu Beginn: lock:=FALSE)

```
repeat  
    while Test-and-Set(lock) do noop;  
    Kritischer Bereich;  
    lock := FALSE;  
    Unkritischer Bereich;  
until FALSE;
```

busy waiting

- Es ist gewährleistet, dass immer **höchstens ein Prozess** in seinem kritischen Bereich ist.
- Die **bounded-waiting-Bedingung ist nicht erfüllt**, da ein Prozess beliebig oft Pech haben kann, wenn er die Variable lock jedesmal erst abfragt, nachdem ihm schon wieder ein anderer Prozess zuvorgekommen ist.

3.2 Wechselseitiger Ausschluss / Synchronisation

Swap: Atomares Vertauschen des Inhalts zweier boolescher Variablen

Realisierung der swap-Operation:

```
procedure swap(var a,b: boolean): boolean;  
var temp: boolean;  
begin  
    temp := a;  
    a := b;  
    b := temp;  
end;
```

Atomare
Operation

```
repeat  
    key:= TRUE;  
    repeat  
        swap(lock, key);  
    until key = FALSE;  
    Kritischer Bereich;  
    lock:= FALSE;  
    Unkritischer Bereich;  
until FALSE;
```

- Nutzung der **swap-Operation** zum atomaren Aufschließen eines globalen Schlosses **lock** mit einem lokalen Schlüssel **key**.
- **lock** und **key** sind boolesche Variablen.
- Wie bei Test-and-Set ist auch hier die **bounded-waiting-Bedingung nicht garantiert**.

3.3 Semaphore / Allgemein

Bisher

- Software-Lösungen basieren auf mehr oder weniger elementaren atomaren Operationen.
- Bei der Übertragung auf komplexe Aufgabenstellungen werden diese schnell unübersichtlich.

Ziel

- Höhersprachliche Konstrukte, die ein größeres Maß an Übersichtlichkeit ermöglichen.
- Realisierung dieser Konstrukte durch einfache Operationen. Atomarität muss immer gewährleistet sein.

Erstes wichtiges Hilfsmittel: **Semaphor**

Beispiel: Parkhaus mit einem Zähler am Eingang

- Zähler wird am Morgen initialisiert und zeigt die Anzahl der verfügbaren Parkplätze an.
- Jedes einfahrende Auto verringert den Zähler um eins (atomar!).
- Jedes wegfahrende Auto erhöht den Zähler um eins (atomar!).
- Wenn Zähler = 0, muss ankommendes Auto warten, bis der Zähler erhöht wird.

3.3 Semaphore / Allgemein

Semaphor: Integer-Variable s zusammen mit 3 atomaren Operationen:

- `init (s, Anfangswert)`
- `wait (s)`
- `signal (s)`

`init(s, Anfangswert)`

- Wirkung: $s := \text{Anfangswert}$
- Normalerweise entspricht Anfangswert der Anzahl von Prozessen, die sich gleichzeitig im kritischen Bereich aufhalten dürfen.
- Bei wechselseitigem Ausschluss ist der Anfangswert 1 (binäre Semaphore).
- Bei Zählsemaphoren ist der Wertebereich nicht nach oben beschränkt.

3.3 Semaphore / Allgemein

von niederländisch **proberen** = probieren

```
L:Wait(S) (oder auch P(S))  
{  
    if (S = 0) goto L;  
    S := S - 1;  
}
```

von niederländisch **verhogen** = erhöhen

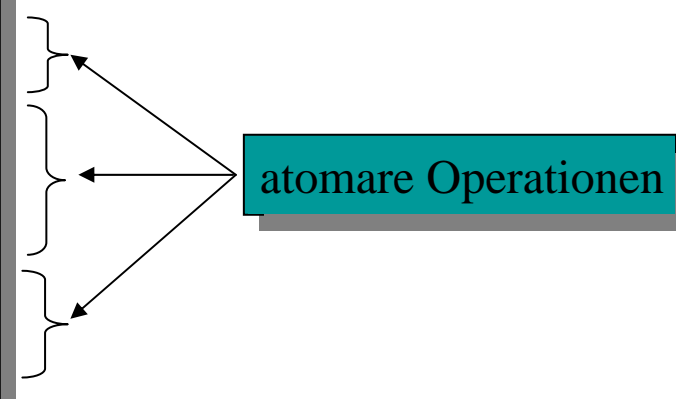
```
Signal(S) (oder auch V(S))  
{  
    S := S + 1;  
}
```

3.3 Semaphore / Semaphore ohne assoziierte Warteschlange

Atomare Operationen

➤ hier: Wechselseitiger Ausschluss, d.h. nur ein Prozess in kritischer Region erlaubt.

Operation	Wirkung
init(s,1)	s := 1;
wait(s)	L: if (s=1) then s:=0; else goto L;
signal(s)	s := 1;



atomare Operationen

3.3 Semaphore / Semaphore ohne assoziierte Warteschlange

Schwerwiegender Nachteil der bisherigen Lösungen

- Prozesse verbrauchen unnötige Ressourcen, wenn sie immer wieder probieren, in den kritischen Bereich zu kommen.
- Busy-Waiting, d.h. beim warten wird CPU-Zeit verbraucht
➔ Verschwendung von CPU-Zeit.

Verbesserung

- Prozess, der den kritischen Bereich belegt vorfindet, wird schlafen gelegt.
- FIFO-Warteschlange verwaltet schlafende Prozesse, um endliche Wartezeit zu garantieren.
- Durch eine **signal**-Operation wird der Prozess wieder geweckt.

Methode

- Zu einer Semaphor wird eine Warteschlange assoziiert.

3.3 Semaphore / Semaphore mit assoziierter Warteschlange

Atomare Operationen:

Operation	Wirkung
<code>init(s,x)</code>	<code>s := x;</code>
<code>wait(s)</code>	<code>s := s-1;</code> if (<code>s < 0</code>) then „Ordne Prozess an Position -s der assozierten Warteschlange ein“;
<code>signal(s)</code>	<code>s := s+1;</code> if (<code>s <= 0</code>) then „Wecke den ältesten wartenden Prozess und sende ihn in den kritischen Bereich“;

3.3 Semaphore / Ausschlussproblem

Algorithmus für das wechselseitige Ausschlussproblem mit Semaphoren

Globale Initialisierung

```
init(s,1)
```

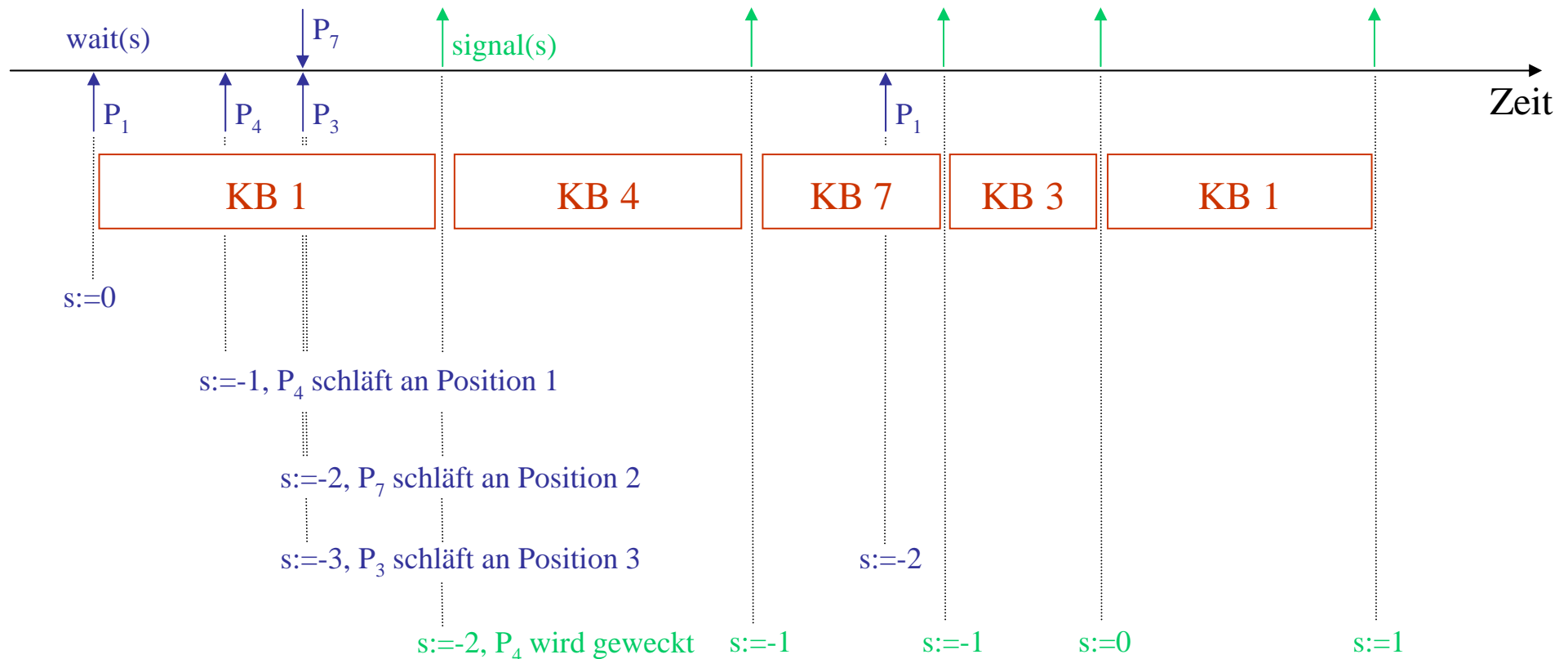
Algorithmus für Prozess P_i

```
repeat  
  wait(s);  
  Kritischer Bereich;  
  signal(s);  
  Unkritischer Bereich;  
until FALSE;
```

3.3 Semaphore / Semaphore mit assoziierter Warteschlange

Beispiel:

init(s, 1)



3.3 Semaphore / Erzeuger-Verbraucher-Problem

Erzeuger-Verbraucher Problem

- n Erzeuger
- m Verbraucher
- Zwischenlager der Größe MAX

Nebenbedingungen werden mit zusätzlichen Semaphoren f (full) und c (clean) gesteuert

- In ein volles Lager kann nichts deponiert werden: $c=0 \leftrightarrow$ Lager ist voll
- Aus einem leeren Lager kann nichts entnommen werden: $f=0 \leftrightarrow$ Lager ist leer
- Zugriff auf Lager wird durch Semaphore s gesteuert.

Kritische Bereiche

- Ablegen/Entnehmen von Produkten in/aus Lager, da Erzeuger wie Verbraucher dabei auf gemeinsame Daten zugreifen müssen.

Voreinstellungen

- `init(s,1);`
- `init(f,0);`
- `init(c,MAX);`

Während der gesamten Laufzeit soll $f+c$ invariant bleiben!

3.3 Semaphore / Erzeuger-Verbraucher-Problem

Erzeuger

```
repeat  
  produziere ein Element und lege es in nextp ab;  
  wait(c);                // prüfe, ob Lager voll  
  wait(s);                // betrete kritischen Bereich  
  füge nextp in das Lager ein;  
  signal(s);              // verlasse kritischen Bereich  
  signal(f);              // erhöhe Lagerfüllstand  
until FALSE;
```

unkritisch

kritisch

Verbraucher

```
repeat  
  wait(f);                // prüfe, ob Lager leer  
  wait(s);                // betrete kritischen Bereich  
  entferne Element aus dem Lager und lege es in nextc ab;  
  signal(s);              // verlasse kritischen Bereich  
  signal(c);              // erhöhe Zähler f. leere Plätze  
  verbrauche das Element in nextc;  
until FALSE;
```

kritisch

unkritisch

3.3 Semaphore / Reader-Writer-Problem

Klassisches Problem, das im Zusammenhang mit der Verwaltung von Datenbanken auftaucht. Zwei Arten von Prozessen haben Zugriff auf ein gemeinsames Objekt:

- **Writer:** Prozesse, die schreiben dürfen. Es muss sichergestellt werden, dass der Schreibzugriff exklusiv erfolgt, d.h. kein anderer Prozess darf während des Updates aktiv sein.
- **Reader:** Prozesse, die Anfragen stellen, die simultan stattfinden können.

Beispiel Bahnhofsfahrplan

- Fahrplanwechsel (Update) wird exklusiv durchgeführt. Beamter wechselt ausgehängten Fahrplan in einem Augenblick, in dem ihn niemand konsultiert.
- Prinzipiell können beliebig viele Interessenten gleichzeitig auf den Fahrplan schauen.

Im Allgemeinen gilt:

- **Unkritisch:** gleichzeitiges Lesen
- **Kritisch:** gleichzeitiges Schreiben oder gleichzeitiges Lesen und Schreiben.

3.3 Semaphore / Reader-Writer-Problem

1. Reader-Writer-Problem:

- Kein Reader muss auf eine Eintrittserlaubnis warten; es sei denn, ein Writer befindet sich im kritischen Bereich.
- Problem: **Reader** können das System **monopolisieren**, und Writer kommt nicht dazu, sein Update durchzuführen → **starvation problem** bezüglich Writer.

2. Reader-Writer-Problem:

- Sobald ein Writer wartet, wird neuen Readern der Zugang verwehrt.
- Die **Writer** können das System **monopolisieren**.

3. Reader-Writer-Problem:

- Fairness durch alternierende Lese- und Schreibphasen
- Ist gerade Lesephase und meldet sich ein Writer an, werden keine neuen Reader mehr zugelassen.
- Sobald alle Reader fertig sind, darf der Writer zugreifen.
- Ist gerade eine Schreibphase und meldet sich ein Reader an, wird das Ende dieser Schreibphase abgewartet und dann eine Lesephase gestartet.

3.3 Semaphore / Reader-Writer-Problem

Lösung für das erste Reader-Writer-Problem, die Semaphore und Zählvariable verwendet:

- Semaphore s für die Schreibphase
- Semaphore w für die Lesephase
- Variable n zählt Anzahl der gleichzeitig aktiven Leser

Funktionen:

- Semaphore s wird durch `init(s, 1)` initialisiert und wird sowohl von Reader- als auch von Writerprozessen verwendet, um Writern einen exklusiven Zugriff zu ermöglichen.
- Semaphore w stellt sicher, dass ein Update von n ungestört erfolgen kann.

Writer

```
repeat  
    wait(s);  
    schreibe;  
    signal(s);  
until FALSE;
```

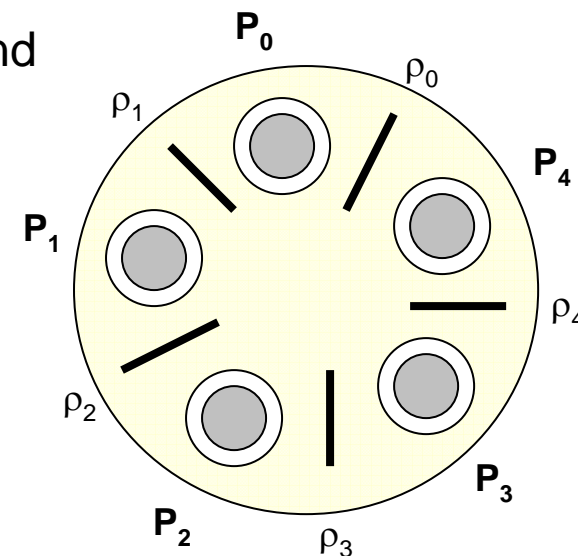
Reader

```
repeat  
    wait(w);  
    n := n + 1;  
    if n = 1 then wait(s);  
    signal(w);  
    lies;  
    wait(w);  
    n := n - 1;  
    if n = 0 then signal(s);  
    signal(w);  
until FALSE;
```


3.3 Semaphore / Fünf-Philosophen-Problem

Problem:

- Tisch mit 5 Philosophen, Tellern und Stäbchen
- Abwechselnde Ess- und Denkphasen
- Essen nur möglich, wenn 2 Stäbchen frei sind



Abstrakt:

- 5 Prozesse P_0, \dots, P_4
- 5 Betriebsmittel Stäbchen ρ_0, \dots, ρ_4
- Jeder Prozess P_i benötigt zeitweise 2 Betriebsmittel ρ_i (linkes Stäbchen) und $\rho_{(i+1) \bmod 5}$ (rechtes Stäbchen)

3.3 Semaphore / Fünf-Philosophen-Problem

Schematische Struktur:

```
repeat  
  Unkritischer Bereich → Denken;  
  Kritischer Bereich → Essen:  $P_i$  benötigt  $\rho[i]$  und  $\rho[(i+1) \bmod 5]$ ;  
until FALSE;
```

Naiver, aber falscher Algorithmus für P_i ($i=0,\dots,4$):

```
repeat  
  wait( $\rho[i]$ );  
  wait( $\rho[(i+1) \bmod 5]$ );  
  Essen;  
  signal( $\rho[i]$ );  
  signal( $\rho[(i+1) \bmod 5]$ );  
  Denken;  
until FALSE;
```

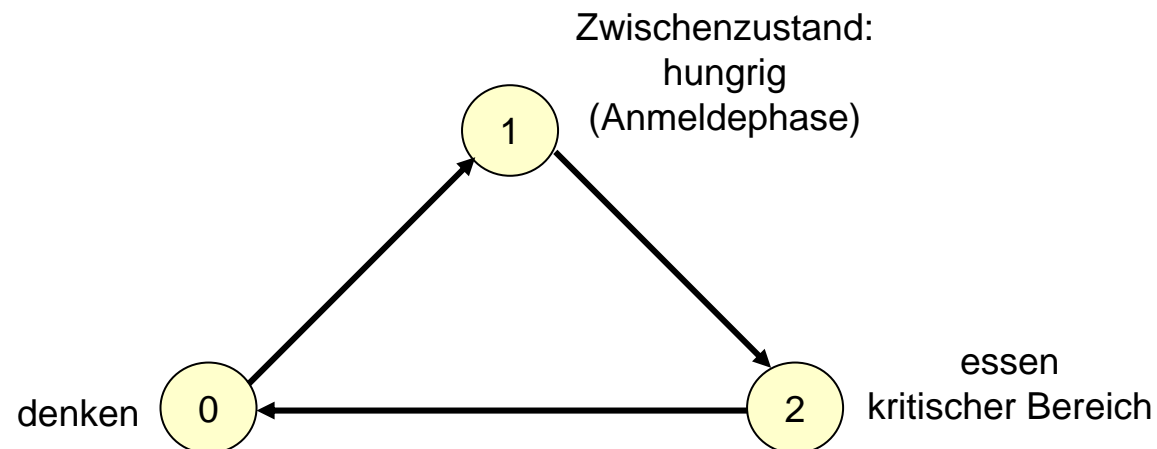
3.3 Semaphore / Fünf-Philosophen-Problem

Problem:

➤ Alle 5 Philosophen wollen gleichzeitig in ihren kritischen Bereich. Jeder greift zunächst nach dem linken Stäbchen und wartet mit diesem in der Hand auf das rechte Stäbchen. Dies führt zu einer Blockade ➔ Deadlock.

Lösung:

➤ Einführung eines Zwischenzustandes **hungrig**



3.3 Semaphore / Fünf-Philosophen-Problem

Benötigt wird:

- Semaphore s ($\text{init}(s, 1)$)
- Semaphore h_0, \dots, h_4 ($\text{init}(h_i, 0)$), wobei $h_i=1$ bedeutet, dass beide Stäbchen für P_i frei sind und P_i hungrig ist.
- Kontrollvariable $c_i \in \{0, 1, 2\}$ gibt die Zustände der Philosophen an.
- Testprozedur

Testprozedur:

```
procedure test(k)
  if (c[(k-1) mod 5] ≠ 2)      (* linker Nachbar isst nicht *)
    and c[k]=1                (* ich bin hungrig *)
    and c[(k+1) mod 5] ≠ 2)    (* rechter Nachbar isst nicht *)
  then
  begin
    c[k] := 2;
    signal(h[k]);
  end
```

3.3 Semaphore / Fünf-Philosophen-Problem

Algorithmus für den Philosophen P_i

```
repeat
  Denken;

  wait(s);
  c[i]:= 1;          (* hungrig werden          *)
  test(i);          (* Test, ob Stäbchen frei *)
  signal(s);

  wait(h[i]);

  Essen;

  wait(s);
  c[i]:=0;          (* zurück zum Denken          *)
  test(i+1 mod 5);  (* Test, ob Nachbarn essen können *)
  test(i-1 mod 5);
  signal(s);

until FALSE;
```

3.3 Semaphore / Fünf-Philosophen-Problem

- Diese Lösung ist deadlockfrei.
- Allerdings können zwei Philosophen einen zwischen ihnen liegenden verhungern lassen, d.h. es ist möglich, einen Prozess für längere Zeit zu blockieren.

Mögliches Szenario:

→ P_1 isst.
 P_2 ist hungrig und wartet darauf, dass P_1 fertig wird.
Kurz bevor P_1 fertig ist, beginnt P_3 zu essen.
 P_1 ist fertig.
 P_2 ist immer noch hungrig und wartet darauf, dass P_3 fertig wird.
Kurz bevor P_3 fertig ist, beginnt P_1 zu essen.
 P_3 ist fertig.
 P_2 ist immer noch hungrig und wartet darauf, dass P_1 fertig wird.

3.3 Semaphore / Fünf-Philosophen-Problem

Eine weitere Möglichkeit, einen Deadlock zu vermeiden, ist die Einführung einer Asymmetrie unter den Prozessen, indem ein Philosoph zum Rechtshänder deklariert wird.

Lösung des Starvation- und Deadlock-Problems:

$P_0 \dots P_3$ (Linkshänder)	P_4 (Rechtshänder)
$\text{wait}(\rho_i)$	$\text{wait}(\rho_0)$
$\text{wait}(\rho_{i+1})$	$\text{wait}(\rho_4)$
Essen	Essen
$\text{signal}(\rho_i)$	$\text{signal}(\rho_0)$
$\text{signal}(\rho_{i+1})$	$\text{signal}(\rho_4)$

Einzige Schwäche dieser Lösung ist die Asymmetrie.

3.3 Semaphore / Drei-Raucher-Problem

3 Raucher R1, R2, R3 wollen rauchen und brauchen dazu Betriebsmittel B1, B2 und B3 [B1 = Feuer, B2 = Pfeife, B3 = Tabak].

Werbeslogan vor geraumer Zeit (für Stanwell-Tabak):

„3 Dinge braucht der Mann: Feuer, Pfeife, Stanwell!“

Jeder Raucher besitze eines dieser Betriebsmittel.

R1 habe B1

R2 habe B2

R3 habe B3

R1 braucht also zusätzlich B2 und B3 usw.

Agent A hat beliebig viele von allen Betriebsmitteln.

Synchronisationsaufgabe:

Agent A legt in einem dunklen Raum zwei unterschiedliche Betriebsmittel hin. Raucher greifen auf diese Betriebsmittel zu. Wenn sich einer die beiden „richtigen“ - d.h. ihm fehlenden - Betriebsmittel genommen hat, raucht er und informiert den Agenten, wenn er fertig ist. Dieser legt zwei neue zufällig gewählte Betriebsmittel hin und das Spiel beginnt von Neuem.

3.3 Semaphore / Drei-Raucher-Problem

Schwierigkeit bei der Synchronisation: Deadlockrisiko!

Beispiel: Auf dem Tisch liegen B1 und B3. Diese „passen“ nur für Raucher R2.

Aber: Es ist nicht gewährleistet, dass R2 beide hingelegten Betriebsmittel erhält.

Es könnte z.B. R3 beide BM erhalten (und nicht weiterkommen, weil ihm B2 fehlt).

Oder R2 könnte B3 erhalten und R1 könnte B1 kriegen (und dann geht nichts mehr).

Zurücklegen „falscher“, d.h. unbrauchbarer, Betriebsmittel bringt auch nichts, weil sich eine ähnliche Situation beim erneuten Zugriff wiederholen kann.

Warten auf spätere Freigabe eines fehlenden Betriebsmittels funktioniert auch nicht, wenn alle stur an ihrer Zuteilung festhalten.

3.3 Semaphore / Drei-Raucher-Problem

Lösungsidee:

Jeder Raucher fragt zunächst nach seinem (!) Betriebsmittel, obwohl er das ja schon hat (er stellt sich so als hätte er es noch nicht!).

Genau derjenige, dessen BM nicht auf dem Tisch liegt, der also der „Rauchfähige“ ist, wird an dieser Abfrage [realisiert mit wait(eigenes BM)] „hängenbleiben“.

Die beiden anderen können weitermachen (sie erkennen ihre Nicht-Rauchfähigkeit genau daran!). Sie müssen dann zusammenarbeiten, um die Identität des Dritten herauszufinden und ihm zum Rauchen zu verhelfen, d.h. ihm die BM abzutreten, ihn zu entsperren und in den Raucherzustand zu überführen.

Abschließend müssen alle sich in den Ausgangszustand zurückbegeben.

3.3 Semaphore / Drei-Raucher-Problem

Bestandteile der Lösung:

Semaphore B1, B2, B3 (jeweils mit 0 vorbesetzt - auch für den, der das betreffende Betriebsmittel besitzt).

Variablen:

Rückmelden1, Rückmelden2, Rückmelden 3 (zur Wiederherstellung des Ausgangszustands für die Raucher R1, R2, R3 nach einem Rauchzyklus).

Testnötig1, Testnötig2, Testnötig3 (damit der Rauchfähige die Auswahlprozedur - d.h. das Herausfinden des Rauchfähigen - nicht ebenfalls durchführen muss).

Zählvariablen w1, w2, w3 (zum Herausfinden des Berechtigten; wenn $w_j = 2$ ist, dann ist Rj rauchfähig).

3.3 Semaphore / Drei-Raucher-Problem

Lösung

Vorbesetzungen:

```
init(B1,0);  
init(B2,0);  
init(B3,0);  
init(rückmelden1,1);  
init(rückmelden2,1);  
init(rückmelden3,1);  
init(s,1);  
w1:=0; w2:=0; w3:=0;  
testnötig1:=true; testnötig2:=true; testnötig3:=true;
```

Agent A:

```
repeat  
  wait(rückmelden1); wait(rückmelden2); wait(rückmelden3);  
  w1:=0; w2:=0; w3:=0;  
  
  p := rand(0;1);  
  if (p < 1/3) then (signal(B1); signal(B2))  
  else if (p < 2/3) then (signal(B2); signal(B3))  
  else (signal(B3); signal(B1))  
until false
```

3.3 Semaphore / Drei-Raucher-Problem

repeat

Start1:
wait(B1);
/*fragt nach eigenem BM!*/

Raucher R1

```
if testnötig1 then
begin
  wait(s);
  w2 := w2+1;
  w3 := w3+1;
  signal(s);
  while w2<2 and w3<2
  do noop;
  signal(rückmelden1);
  if w2=2 then
  begin
    testnötig2 := false
    signal(B2);
  end;
  goto Start1
end;
Rauchen
testnötig1 := true;
signal(rückmelden1);
until false
```

repeat

Start2:
wait(B2);
/*fragt nach eigenem BM!*/

Raucher R2

```
if testnötig2 then
begin
  wait(s);
  w3 := w3+1;
  w1 := w1+1;
  signal(s);
  while w3<2 and w1<2
  do noop;
  signal(rückmelden2);
  if w3=2 then
  begin
    testnötig3 := false
    signal(B3);
  end;
  goto Start2
end;
Rauchen
testnötig2 := true;
signal(rückmelden2);
until false
```

repeat

Start3:
wait(B3);
/*fragt nach eigenem BM!*/

Raucher R3

```
if testnötig3 then
begin
  wait(s);
  w1 := w1+1;
  w2 := w2+1;
  signal(s);
  while w1<2 and w2<2
  do noop;
  signal(rückmelden3);
  if w1=2 then
  begin
    testnötig1 := false
    signal(B1);
  end;
  goto Start3
end;
Rauchen
testnötig3 := true;
signal(rückmelden3);
until false
```

3.3 Semaphore / Drei-Raucher-Problem

Erläuterungen zum Ablauf:

1. Die Prozeduren für R1, R2 und R3 sind symmetrisch zueinander.
2. Beispielsweise liegen B1 und B3 auf dem Tisch. R2 bleibt mit wait (B2) hängen.

R1 und R3 setzen (zusammen betrachtet): $w1 = 1$, $w2 = 2$, $w3 = 1$.

R1 erkennt daran, dass R2 rauchfähig ist. R3 erkennt, dass R1 nicht rauchen kann.

R1 und R3 melden sich zurück.

R1 entsperrt R2 mit signal (B2) und vermeidet durch `testnötig2 := false`, dass B2 einen Test durchführt.

3. R2 raucht, setzt `testnötig2 := true`, abschließend meldet er sich zurück.
4. Das Spiel beginnt von Neuem.

3.4 Petrinetze

Höbersprachliche Synchronisationsmechanismen

- Auch Semaphore sind in vielen Fällen zu unhandlich.
- Daher sind abstrakte Notationen notwendig.
- Graphische Notationen, z.B. Petri-Netze: Graphisches und mathematisches Hilfsmittel zur Modellierung von nebenläufigen, asynchronen, verteilten, parallelen, deterministischen oder stochastischen Systemen.
- Erlauben eine formale (mathematische) Analyse qualitativer und quantitativer Systemeigenschaften. Beispiele hierfür sind:
 - Qualitative Systemeigenschaften: Lebendigkeit, Erreichbarkeit, Verklemmungen
 - Quantitative Systemeigenschaften: Verzögerung, Durchsatz

3.4 Petrinetze

Vorteil:

- Mathematische Verfahren zur Analyse des modellierten Systems

Nachteil:

- In der Praxis oft zu umfangreich für eine sinnvolle Analyse
- Keine hierarchische Struktur

PrivateMeinung:

- Mit Petri-Netzen lässt sich wenig Neues zeigen, aber zeitliche Abläufe lassen sich sehr schön darstellen.

Fragen sind z.B.

- Gibt es Situationen, in denen nicht mehr gefeuert werden kann (Deadlock) oder wo man aus einem Muster nicht mehr herauskommt?

3.4 Petrinetze

Definition Petrinetz:

Ein **Netz** sei definiert als ein endlicher, gerichteter Graph $X = (\{S \cup T\}, F)$, bestehend aus einer endlichen Menge $S = \{s_1, s_2, \dots, s_m\}$ von **Stellen** / **Places** gezeichnet als Kreise (sie stehen für Bedingungen), einer endlichen Menge $T = \{t_1, t_2, \dots, t_n\}$ von **Übergängen** / **Transitionen** gezeichnet als Rechtecke (sie stehen für Ereignisse) und einer Menge $F \subseteq (S \times T) \cup (T \times S)$ von gerichteten **Kanten** gezeichnet als Pfeile.

Durch Angabe der Elemente von S , T und F ist ein Petrinetz vollständig beschrieben.

3.4 Petrinetze

Definition Vor- und Nachbereich:

Für eine beliebige Stelle bzw. Transition $x \in S \cup T$ sei

$$\bullet x := \{y \mid (y, x) \in F\}$$

$$x \bullet := \{y \mid (x, y) \in F\}$$

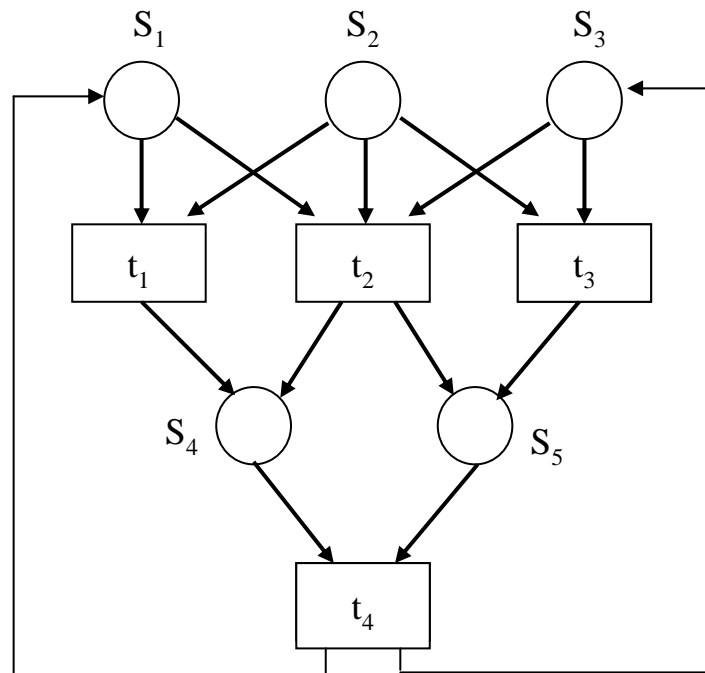
$$-x := \{(y, x) \mid (y, x) \in F\}$$

$$x^- := \{(x, y) \mid (x, y) \in F\}$$

Bezeichnungen:

- Die Menge $\bullet x$ wird Vorbereich einer Stelle $x \in S$ bzw. einer Transition $x \in T$ genannt. Analog heißt $x \bullet$ Nachbereich einer Stelle bzw. einer Transition.
- Für eine Transition $t \in T$ heißen die Elemente der Menge $\bullet t$ **Eingangsstellen** und die Elemente der Menge $t \bullet$ **Ausgangsstellen** der Transition.
- Die Kanten $(s, t) \in -t$ werden als **Eingangskanten** und die Kanten $(s, t) \in t^-$ als **Ausgangskanten** der Transition t bezeichnet.

3.4 Petrinetze / Beispielnetz



Stellen: Zustände, bereitgestellte Daten, Signale, Bedingungen oder Speicher für Objekte.

Transitionen: Ereignisse, Vorgänge, Berechnungsschritte, Jobs oder auch Prozessoren.

3.4 Petrinetze

Definition Stellen-/Transitions-System:

Ein **Stellen-/Transitions-System (S/T-System)** ist ein 6-Tupel $Y = (S, T, F, K, W, M_0)$, bestehend aus einem Netz (S, T, F) sowie den Abbildungen:

Kapazität	$K : S \rightarrow \mathbb{N} \cup \{\infty\}$, Wert ∞ entspricht keiner Einschränkung,
Kantengewicht	$W : F \rightarrow \mathbb{N}$, Wert 1 entspricht keiner Gewichtung und
Markierung	$M : S \rightarrow \mathbb{N}$, ordnet jeder Stelle S eine Anzahl Marken (Token) zu.

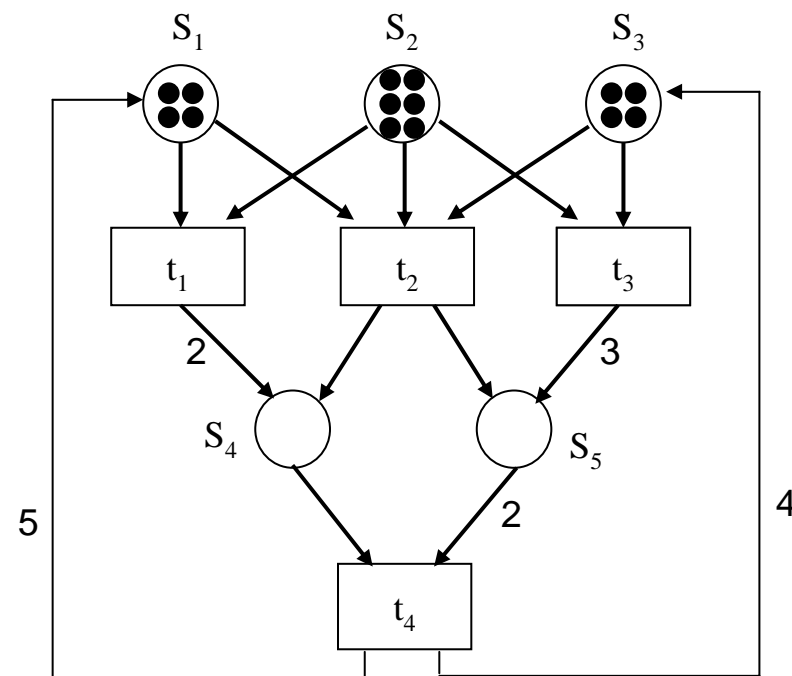
Eigenschaften:

- M wird Markierung von Y genannt, wobei eine Stelle nicht mehr Marken aufnehmen kann als ihre Kapazität erlaubt, d.h. es gilt: $M(s) \leq K(s) \forall s \in S$.
- M_0 bestimmt die Anfangsmarkierung zu Beginn des Prozesses.
- Die Anzahl der Token pro Stelle ist zeitabhängig.
- Marken werden graphisch als schwarze Punkte innerhalb von kreisförmigen Stellen dargestellt.

3.4 Petrinetze

Beispielnetz: alle Stellen haben die Kapazität ∞ , aber unterschiedliche Kantengewichte

$$M_0 = \{ (s_1, 4), (s_2, 6), (s_3, 4), (s_4, 0), (s_5, 0) \}$$



3.4 Petrinetze

Jeder Markierung M wird eine Menge $T_{akt}(M)$ zugeordnet mit

$$T_{akt}(M) = \left\{ t \in T \mid \left(M(s) \geq W(s, t) \forall s \in \bullet t \right) \wedge \right. \\ \left. \left(M(s) \leq K(s) + W(t, s) \forall s \in t \bullet \right) \right\}$$

Die in $T_{akt}(M)$ enthaltenen Transitionen heißen aktiviert unter der Markierung M .

Aktivierte Transitionen t können schalten (feuern). Aus der Markierung M ergibt sich durch Feuern von t die unmittelbare Folgemarkierung M' als

- $M'(s) = M(s) - W(s, t)$ falls $s \in \bullet t \setminus t \bullet$
- $M'(s) = M(s) + W(t, s)$ falls $s \in t \bullet \setminus \bullet t$
- $M'(s) = M(s) - W(s, t) + W(t, s)$ falls $s \in t \bullet \cap \bullet t$

3.4 Petrinetze

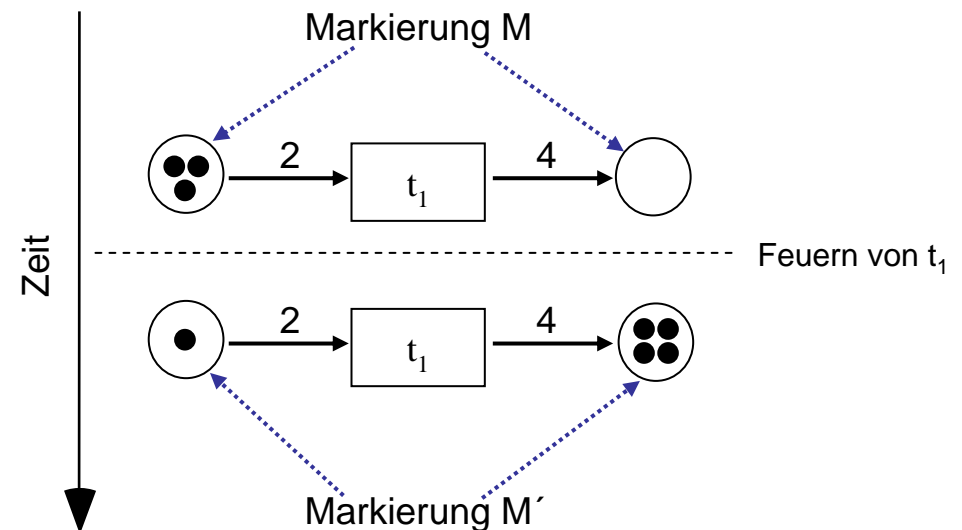
Transition ist aktiviert, wenn

- in all ihren Eingangsstellen die erforderliche Anzahl von Marken vorhanden ist (mindestens ein Token) und
- die Kapazität keiner der Ausgangsstellen durch das Feuern überschritten wird.

Die Anzahl der für die Aktivierung erforderlichen Marken wird durch die jeweiligen Gewichte der Eingangskanten bestimmt.

Feuern:

$M[t > M']$ (t unter der Markierung M schaltet zur Markierung M') bedeutet, dass die festgelegte Anzahl von Marken, bestimmt durch die Gewichte der Eingangskanten, aus den Eingangsstellen der Transition entfernt wird und dafür die durch die Ausgangskantengewichte beschriebene Anzahl an Marken den Ausgangsstellen hinzugefügt wird.



3.4 Petrinetze / Analyse

Analyse eines Petrinetzes Y mit Hilfe

- der Erreichbarkeitsmenge E_Y und
- des Erreichbarkeitsgraphen G_Y

Es sei $T = \{t_1, t_2, \dots\}$ die Menge der einfachen Transitionen und $T^* = \bigcup_{n=0}^{\infty} T^n$.

Eine Markierung M' ist von M aus **erreichbar**, wenn gilt:

$\exists t_1 t_2 t_3 \dots t_n \in T^*, n \in \mathbf{N}_0 : M [t_1 > M_1 [t_2 > M_2 \dots [t_n > M'$

Man schreibt dann auch $M [\omega > M'$ mit $\omega \in T^*$.

M' wird auch als **mittelbare Folgemarkierung** von M bezeichnet.

Definition Erreichbarkeitsmenge:

Die **Erreichbarkeitsmenge** E_Y ist definiert als:

$$E_Y = E_Y(M_0) := \{M' \mid \exists \omega \in T^* : M_0 [\omega > M' \}$$

3.4 Petrinetze / Analyse

Definition Erreichbarkeitsgraph:

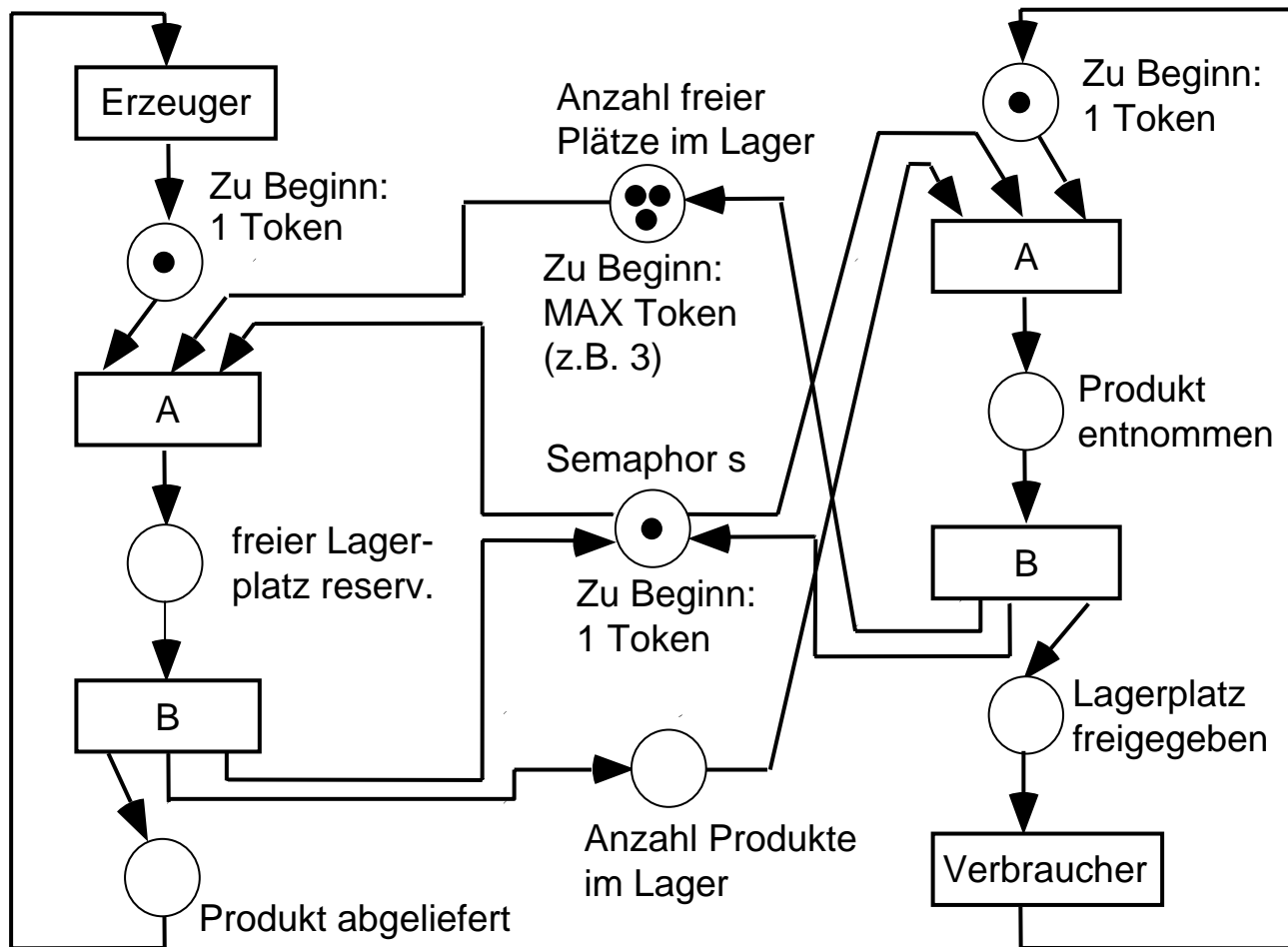
Der **Erreichbarkeitsgraph** G_Y verbindet über eine Kante alle Markierungen $M, M' \in E_Y$, für die gilt: $\exists t_i \in T : M [t_i > M'$ (d. h. M' ist unmittelbare Folgemarkierung von M). Als Kantenbeschriftung wird die Transition angegeben, deren Schalten die entsprechende Änderung der Markierung des Netzes verursacht.

Eigenschaften von Petrinetzen:

- **Teilweise Verklemmung:** $\exists M \in E_Y \mid M$ ist nicht von jeder anderen Markierung aus erreichbar
- **Verklemmung/Deadlock :** $\exists M \in E_Y \mid$ es existiert keine Nachfolgemarkierung M' mit $M [t > M'$
- **Tote Transition:** t ist unter keiner Markierung $M \in E_Y$ aktiviert

3.4 Petrinetze / Erzeuger-Verbraucher-Problem

Startphase: 3 Lagerplätze = 3 Token



A: Warte auf Eintritt in kritische Sektion

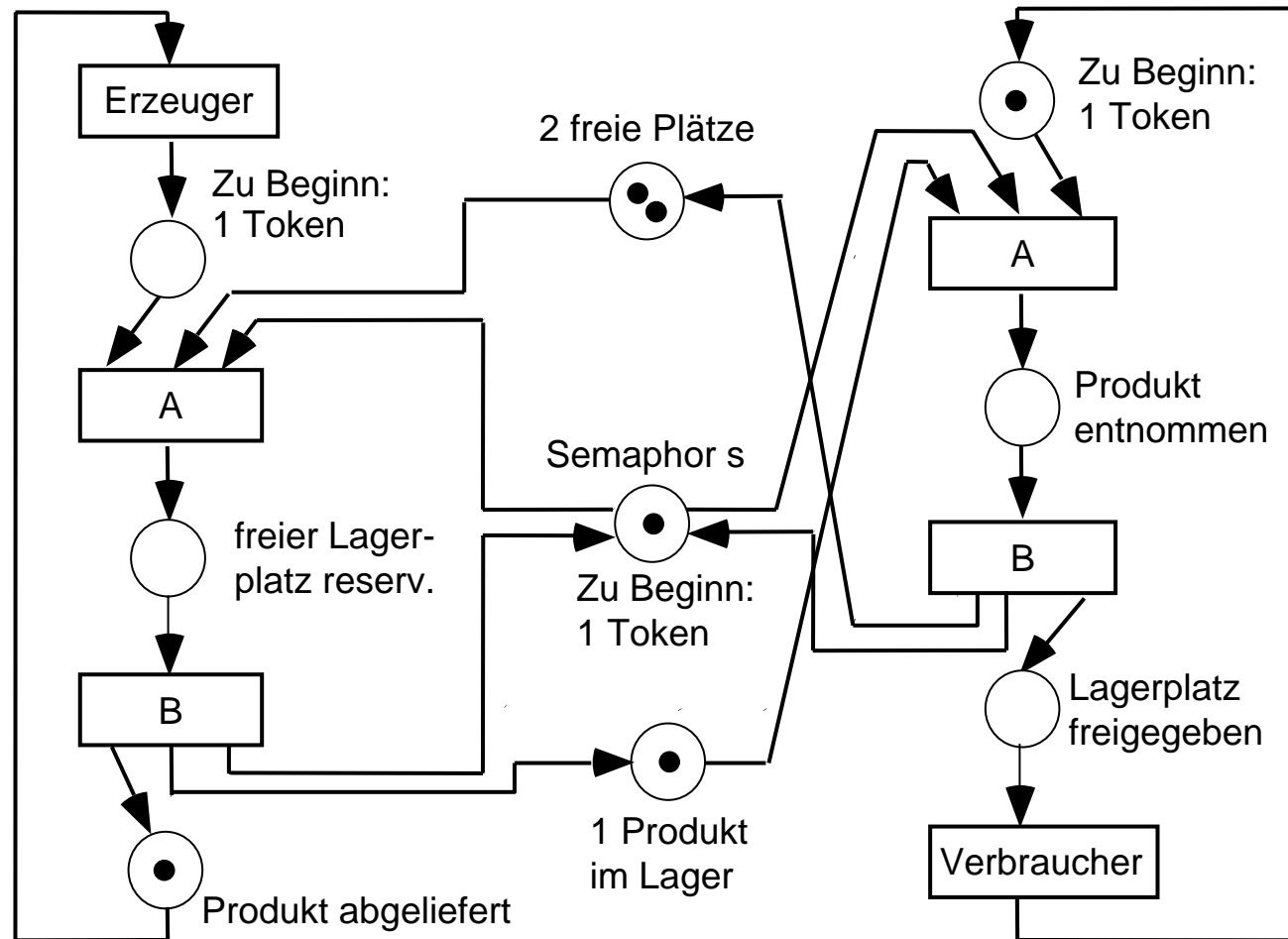
3 Bedingungen:

- Sperre
- Lagerbestand
- Erzeuger-/Verbraucher-Prozess beendet

B: Freigabe nach kritischer Sektion

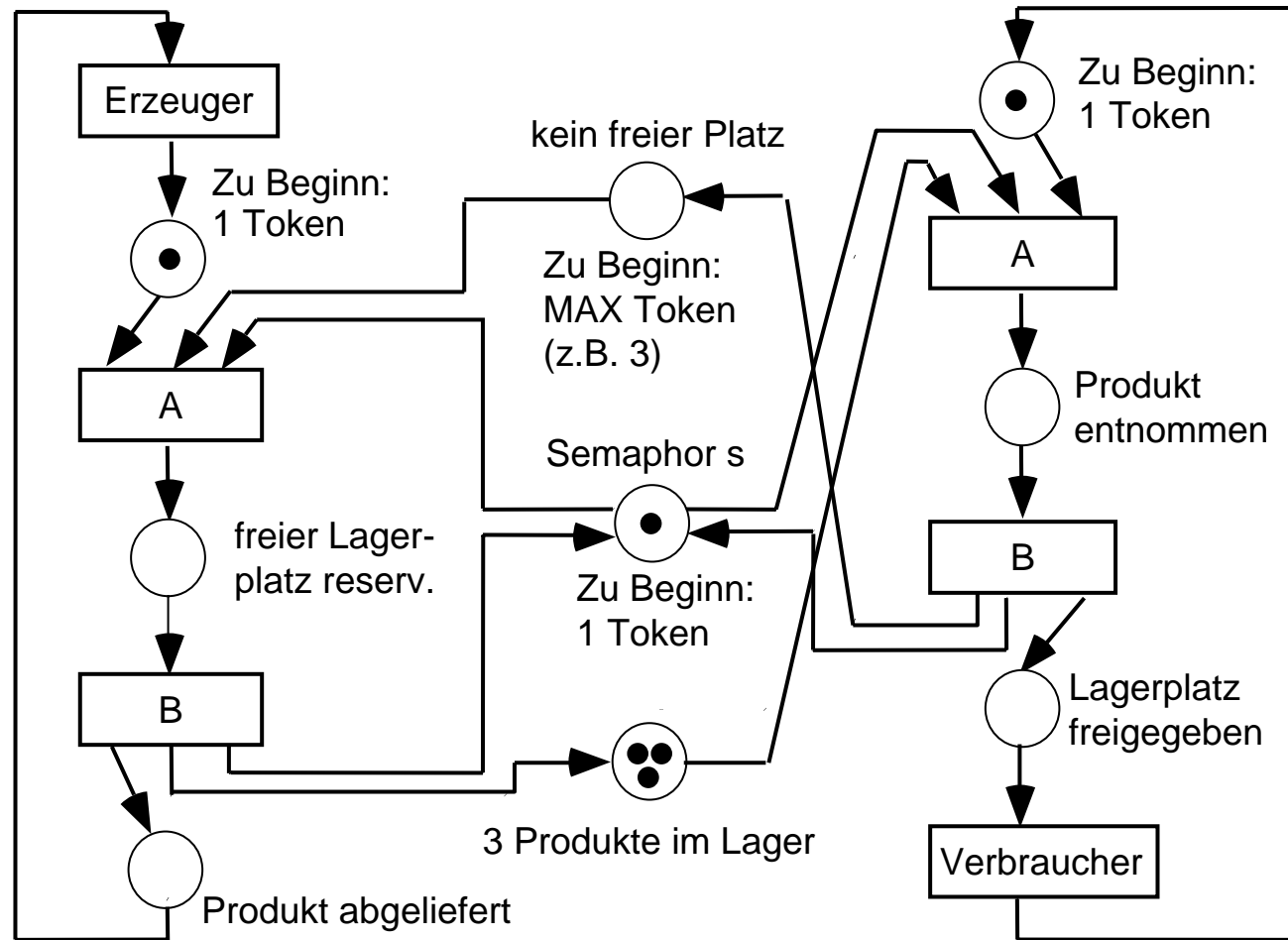
3.4 Petrinetze / Erzeuger-Verbraucher-Problem

Phase 1: 1 Erzeuger hat Produkt abgeliefert



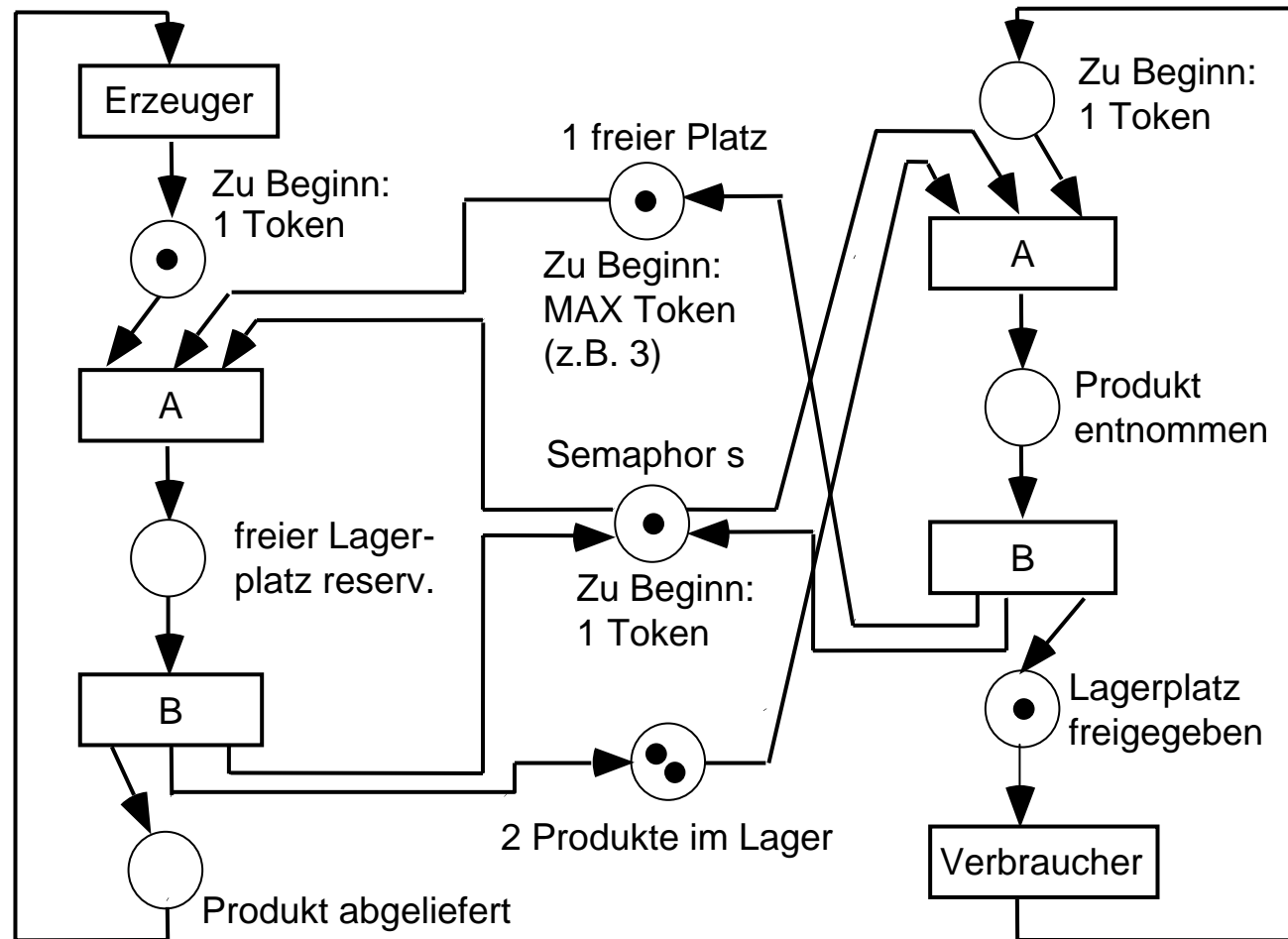
3.4 Petrinetze / Erzeuger-Verbraucher-Problem

Phase 2: 3 Erzeuger haben abgeliefert, Lager ist voll

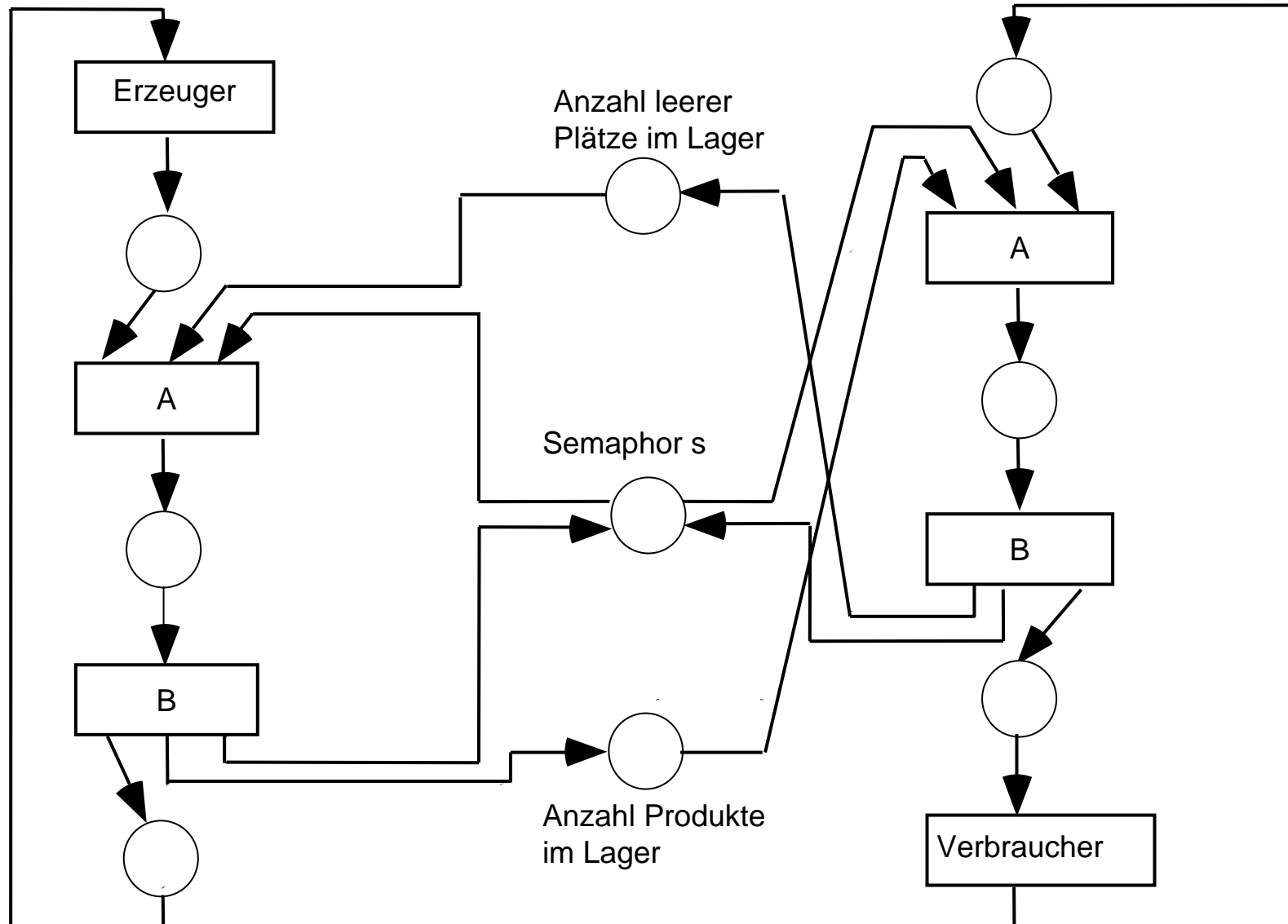


3.4 Petrinetze / Erzeuger-Verbraucher-Problem

Phase 3: 1 Verbraucher hat ein Produkt entnommen



3.4 Petrinetze / Erzeuger-Verbraucher-Problem



3.5 Bedingte kritische Regionen

Bisherige Hilfsmittel zur Prozesssynchronisation

- Einfache Lese-/Schreiboperationen
- Test-and-Set / Swap
- Semaphore
- Petri-Netze

Diese Konzepte werden für komplexe Aufgaben unübersichtlich bzw. fehleranfällig. Um Fehler auszuschließen wurden höhersprachliche Konstrukte entwickelt.

- In diesem Abschnitt werden **kritische Regionen** vorgestellt.

Ein anderes höhersprachliches Konstrukt – Monitore – wird in Abschnitt 3.6 vorgestellt.

3.5 Bedingte kritische Regionen

Kritische Region:

- Prozess besteht aus sequenziellem Programm und lokalen Daten, auf die nur dieser Prozess zugreifen kann
- Darüber hinaus: globale Daten, auf die von mehreren Prozessen zugegriffen wird
- Deklaration einer globalen Variable v vom Typ T für eine kritische Region:
`var v: shared T;`
- Variable v kann nur in einer kritischen Region manipuliert werden:
`region v do S;`

Bedeutung:

- Während der Ausführung von S kann kein anderer Prozess auf die Variable v zugreifen, d.h. S ist kritischer Bereich bezüglich v .

3.5 Bedingte kritische Regionen

Kritische Regionen wenig hilfreich zur Synchronisation.

Deshalb: Bedingte kritische Region

➤ `region v when B do S;`

Auswertung von B:

- gilt `B=TRUE` und befindet sich kein anderer Prozess in einem mit `v` assoziierten kritischen Bereich, dann wird die kritische Region `S` betreten.
- gilt `B=FALSE` oder bearbeitet ein anderer Prozess `v`, dann legt sich der Prozess schlafen; wird später `B` wahr, wird einer der wartenden Prozesse geweckt und `S` ausgeführt.

3.5 Bedingte kritische Regionen / Beispiel

Lösung des Erzeuger-Verbraucher-Problems mit Hilfe von bedingten kritischen Regionen:

- Ringförmiges Zwischenlager `pool[0..MAX-1]`
- Zählvariable `counter` gibt den Lagerstand an ($0 \leq \text{counter} \leq \text{MAX}$)
- Kapselung der o.g. globalen Variablen in `buffer`:

```
var buffer: shared record
  pool: array[0..MAX-1] of item;
  counter,in,out: integer;
end;
```

Auf `buffer` kann nur in einer bedingten kritischen Region zugegriffen werden !

3.5 Bedingte kritische Regionen

Algorithmen für das Erzeuger-Verbraucher-Problem

Erzeuger

```
region buffer when counter < MAX do  
  begin  
    pool[in] := nextp;  
    in := (in+1) mod MAX;  
    counter := counter + 1;  
  end;
```

Verbraucher

```
region buffer when counter > 0 do  
  begin  
    nextc := pool[out];  
    out := (out+1) mod MAX;  
    counter := counter - 1;  
  end;
```

3.5 Bedingte kritische Regionen

Wie implementiert man bedingte kritische Regionen

Eine Lösung (siehe Skript und Silberschatz):

- verwendet 2 Warteschlangen und
- außerdem 3 Semaphore

mutex → mutual exclusion, d.h. gegenseitiger Ausschluss.

Sperrvariable für neue Prozesse

first_delay → Sperre vor Warteschlange 1

(first_count enthält Zahl der in Warteschlange 1 wartenden Prozesse)

second_delay → Sperre vor Warteschlange 2

(second_count enthält Zahl der in Warteschlange 2 wartenden Prozesse)

3.5 Bedingte kritische Regionen

Ein Prozess, der `B=FALSE` vorfindet, wartet zunächst auf die `first_delay`-Warteschlange, dann auf die `second_delay`-Warteschlange.

Sobald der kritische Bereich von einem Prozess erfolgreich durchlaufen wurde, wird zunächst die `first_delay`-Warteschlange und anschließend die `second_delay`-Warteschlange überprüft, ob einer der dortigen Kandidaten aufgeweckt werden kann.

3.5 Bedingte kritische Regionen / Algorithmus

```
wait(mutex);  
while not B do  
begin  
    first_count := first_count+1;  
    if (second_count > 0) then  
        signal(second_delay)  
    else signal(mutex);  
    wait(first_delay);  
    first_count := first_count-1;  
    second_count := second_count+1;  
    if (first_count > 0) then  
        signal(first_delay)  
    else signal(second_delay);  
    wait(second_delay);  
    second_count := second_count-1;  
end;
```

Kritischer Bereich;

```
if (first_count > 0) then  
    signal(first_delay);  
else  
    if (second_count > 0) then  
        signal(second_delay)  
    else signal(mutex);
```

3.5 Bedingte kritische Regionen / Beispiel

Beispiel mit 5 Prozessen, die zunächst jeweils an ihrem B scheitern:

first_delay	1	2	3	4	5
second_delay					

Ein sechster Prozess verlässt seinen kritischen Bereich und setzt das **first_delay** Signal; alle Prozesse wandern in die zweite Schlange:

first_delay					
second_delay	1	2	3	4	5

Die Prozesse prüfen ihre Bedingungen; Prozess 3 findet als erster B=TRUE und betritt den kritischen Bereich:

first_delay	1	2			
second_delay	4	5			

Prozess 3 signalisiert nach Verlassen des kritischen Bereiches und es ergibt sich:

first_delay					
second_delay	4	5	1	2	

USW ...

3.5 Bedingte kritische Regionen

Warum braucht man eigentlich zwei (!) Warteschlangen ?!

Versuch mit einer einzigen WS:

```
init(mutex,1);  
init(delay,0);  
count:= 0;  
wait(mutex);  
while not B do begin  
    count := count + 1;  
    signal(mutex);  
    wait(delay);  
    count := count - 1;  
    if (count > 0)  
    then signal(delay);  
end;  
S;  
if (count > 0) then signal(delay)  
else signal(mutex);
```

Fehler wäre korrigierbar
durch **Erstversuch**

Das Problem ist:

Alle wartenden Prozesse wecken sich nach `signal(delay)` gegenseitig nacheinander auf, genauso wie bei der korrekten Lösung. Gilt zu Anfang etwa `count = 3`, werden sie sich permanent erfolglos aufrufen und zudem laufend `mutex` erhöhen.

→ Zweite Warteschlange dient als Schleuse.

3.5 Bedingte kritische Regionen

Noch einfacherer Versuch:

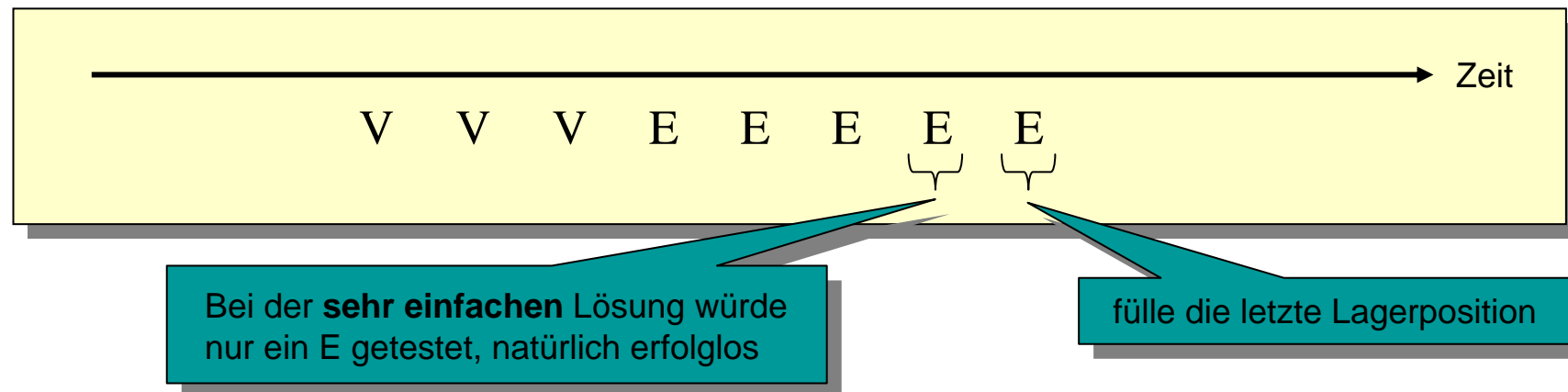
1. Wartende Prozesse sollen sich nicht nacheinander wecken
2. Zu häufiges `signal(mutex)` bei wiederholtem vergeblichen Test von B vermeiden durch Variable `erstversuchi` (für Prozess i)

Einziger Nachteil dieser Lösung:

- Performanceprobleme !
- Bei jedem Verlassen des kritischen Bereiches werden nicht alle Prozesse getestet, sondern lediglich einer.

```
wait(mutex);  
count := count + 1;  
erstversuchi := 1;  
while not B do begin  
    if (erstversuchi=1)  
    then signal(mutex);  
    erstversuchi := 0;  
    wait(delay);  
end;  
S;  
count := count - 1;  
if (count > 0)  
then signal(delay)  
else signal(mutex);
```

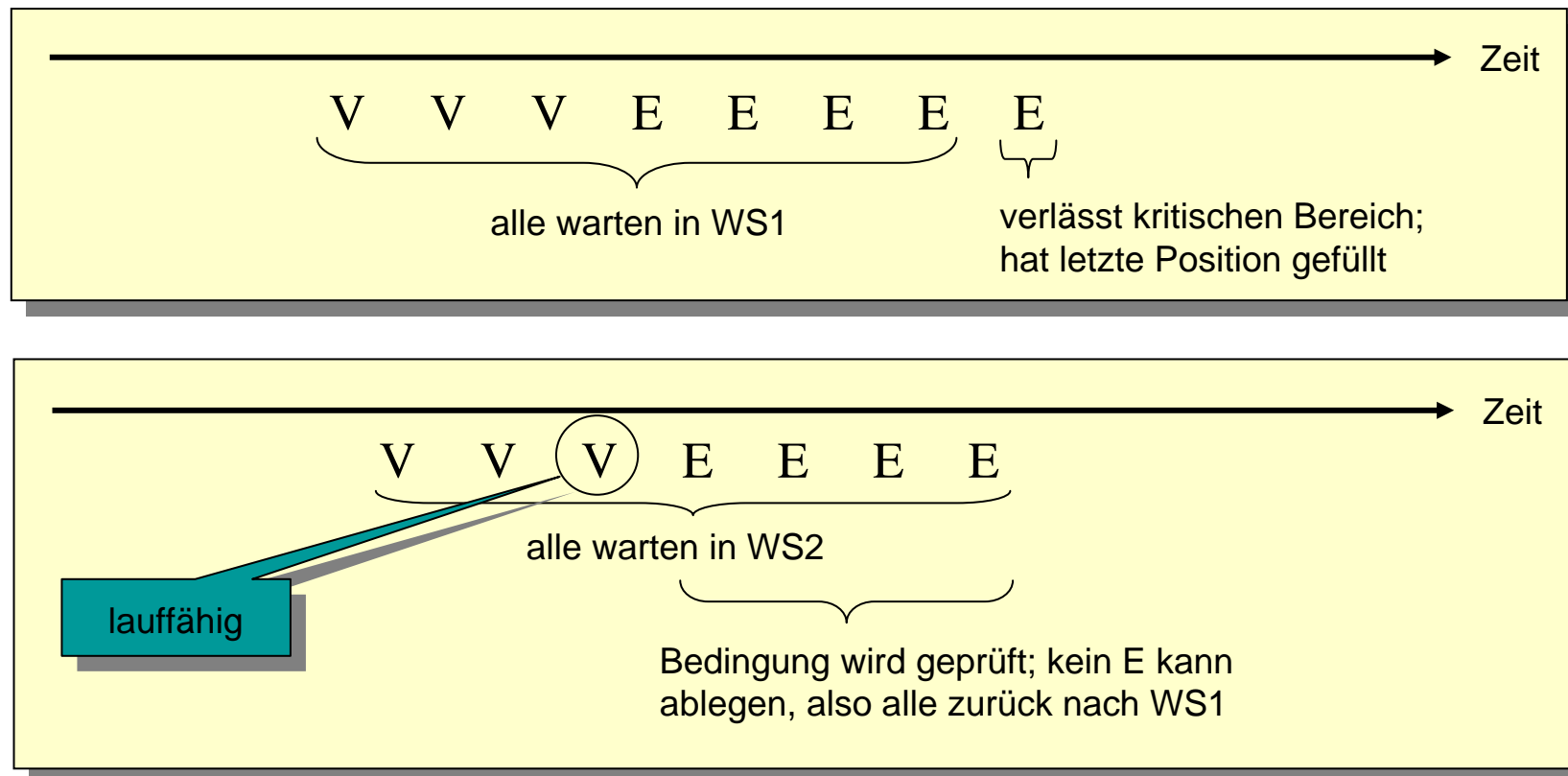
3.5 Bedingte kritische Regionen / Beispiel



Damit es weitergeht, müsste ein neuer Verbraucher ankommen und einen Erzeuger wecken.

3.5 Bedingte kritische Regionen

Bei der korrekten Version geschieht folgendes:



Danach wird der älteste Erzeuger geweckt.

3.5 Bedingte kritische Regionen

Weiteres Beispiel zu bedingten kritischen Regionen:

Lösung des **2. Reader-Writer-Problems**, d.h. die Writer haben Priorität.

Variablen:	Invarianten:
ar := Zahl aktiver Reader lr := Zahl laufender Reader aw := Zahl aktiver Writer lw := Zahl laufender Writer (aktiv = laufend oder laufwillig)	1) $0 \leq lr \leq ar$ $0 \leq lw \leq aw$ $0 \leq lw \leq 1$ 2) $\neg(lr > 0 \wedge lw > 0)$ 3) $(lr = 0 \wedge lw = 0) \wedge (ar > 0 \vee aw > 0)$ $\Rightarrow (lr > 0) \text{ oder } (lw > 0) \text{ nach endlicher Zeit}$ 4) $ar \rightarrow lr \text{ nur möglich, wenn } (aw = 0)$

3.5 Bedingte kritische Regionen

Lösung des 2. Reader-Writer-Problems (Fortsetzung)

Reader:

```
region (aw,lr) do
begin
  when (aw > 0) do
  begin
    await aw=0;
    lr := lr + 1;
  end;
end;
lies;
region lr do lr := lr - 1
unkrit. Bereich;
goto Reader;
```

Writer:

```
region (lr,aw) do
begin
  aw := aw + 1;
  await lr=0;
end;
region lw do schreibe;
region aw do aw := aw - 1;
unkrit. Bereich;
goto Writer;
```

3.6 Monitore

Eigenschaften von Monitoren

- Monitor entspricht High-Level-Konstrukt zur Synchronisation von Prozessen.
- Monitor-Objekt besteht aus einer Menge von Prozeduren und Daten.
- Monitor-Prozeduren dürfen zu einem Zeitpunkt nur von einem Prozess genutzt werden.
- Syntax eines Monitors sieht in der Regel wie folgt aus:

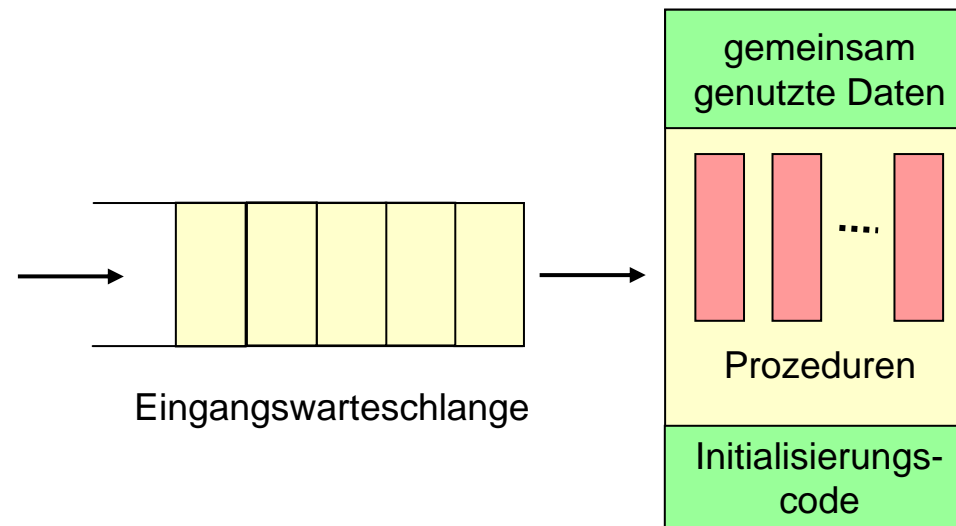
```
monitor name
begin
  Variablendeklarationen;
  procedure P1;
  procedure P2;
  ...
  Vorbesetzung der Variablen;
end;
```

Monitorkaufrufe:

```
name.P1;
name.P2;
```

3.6 Monitore

Schematische Sicht eines Monitors:



Abarbeitung von Monitorprozeduren wird von Bedingungen abhängig gemacht. Bedingungsvariablen sind vom Typ `condition`.

3.6 Monitore

Sei `cond` eine Bedingungsvariable:

- `cond.wait()` (bzw. `wait(cond)`) veranlasst einen Prozess, sich schlafen zu legen, d.h. sich an das Ende der assoziierten Warteschlange einzureihen.
- `cond.signal()` (bzw. `signal(cond)`) weckt den ältesten Prozess in der Warteschlange.
- Unterschied zu Semaphore: `signal` hat keine Wirkung, wenn zum Zeitpunkt des Absendens kein Prozess wartet !
- Mit anderen Worten: `cond.signal()` signalisiert das Eintreffen der Bedingung `cond` und `cond.wait()` realisiert das Warten von Prozessen aufgrund des Nichterfülltseins von `cond`.

3.6 Monitore

Zu jeder Zeit darf nur ein Prozess den Monitor benutzen.

Komplikation:

Was passiert, wenn Prozess **A** während Monitorbenutzung ein Signal setzt,
auf das ein anderer Prozess **B** wartet ?

Es ist verboten, dass beide gleichzeitig weiterlaufen!

Möglichkeiten

- **A** legt sich sofort schlafen und **B** darf weitermachen.
- **B** darf nicht geweckt werden, bis **A** Monitor verlassen hat.

Erste Möglichkeit scheint vernünftiger, da **A** sich bereits im Monitor befindet.

Wichtig: Wenn **A** den Monitor verlässt, kann Bedingung für **B** bereits wieder veraltet sein.

Kein Königsweg: Verwendete Semantik muss explizit festgelegt werden.

Beste Lösung: "signal" nur am Ende einer Monitorprozedur geben.

3.6 Monitore / Beispiel

Exklusiver Zugriff auf ein gemeinsam genutztes Betriebsmittel

```
monitor EXKL;  
begin  
  var busy: boolean;  
  var frei: condition;  
  
  procedure belegen();  
  begin  
    if busy then wait(frei);  
    busy := TRUE;  
  end;  
  
  procedure freigeben();  
  begin  
    busy := FALSE;  
    signal(frei);  
  end;  
  
  busy := FALSE    (* Vorbesetzung! *)  
end;
```

Nutzung:

```
EXKL.belegen();  
Kritischer Bereich;  
EXKL.freigeben();
```

3.6 Monitore / Erzeuger-Verbraucher-Problem

Hauptprogramm

```
monitor LAGER;  
  
begin  
  
var buffer: array[0..MAX-1] of items;  
var lastpointer, count: integer;  
var nonempty, nonfull: condition;  
  
procedure Einfügen(x);  
procedure Entnehmen(x);  
  
count := 0;  
lastpointer := 0; (* Vorbesetzung *)  
  
end;
```

Prozeduren

```
procedure Einfügen(x)  
begin  
    if (count = MAX) then  
        wait(nonfull);  
    lastpointer := (lastpointer+1) mod MAX;  
    count := count + 1;  
    buffer[lastpointer] := x;  
    signal(nonempty);  
end;  
  
procedure Entnehmen(x)  
begin  
    if (count = 0) then  
        wait(nonempty);  
    x := buffer[(lastpointer-count+1) mod MAX];  
    count := count - 1;  
    signal(nonfull);  
end;
```

3.6 Monitore / Erzeuger-Verbraucher-Problem

Erzeuger- und Verbraucher-Routinen sind nun sehr übersichtlich:

Erzeuger:

```
repeat
    erzeuge x;
    LAGER.Einfügen(x);
until FALSE;
```

Verbraucher:

```
repeat
    LAGER.Entnehmen(x);
    verbrauche x;
until FALSE;
```

3.6 Monitore / Drittes Reader-Writer-Problem

Zur Erinnerung:

➤ ankommender Reader erhält Priorität vor später ankommenden Writern und umgekehrt.

Hauptprogramm

```
monitor RW;  
begin  
  
    var readercount: integer;  
    var busywrite: boolean;  
    var okread, okwrite: condition;  
  
    procedure startread;  
    procedure endread;  
    procedure startwrite;  
    procedure endwrite;  
  
    readercount := 0;  
    busywrite := FALSE;  
  
end;
```

3.6 Monitore / Drittes Reader-Writer-Problem

Prozeduren

```
procedure startread;
begin
    if (busywrite or okwrite.nonempty)
    then wait(okread);
    readercount := readercount + 1;
    signal(okread);
end;

procedure endread;
begin
    readercount := readercount - 1;
    if (readercount = 0)
    then signal(okwrite);
end;
```

Prozeduren

```
procedure startwrite;
begin
    if (busywrite or (readercount > 0))
    then wait(okwrite);
    busywrite := TRUE;
end;

procedure endwrite;
begin
    busywrite := FALSE;
    if okread.nonempty
    then signal(okread)
    else signal(okwrite);
end;
```

3.6 Monitore / Drittes Reader-Writer-Problem

Zugehörige Reader- und Writer-Routinen:

Reader:

```
repeat
  RW.startread;
  Lesen;
  RW.endread;

  andere Operationen;

until FALSE;
```

Writer:

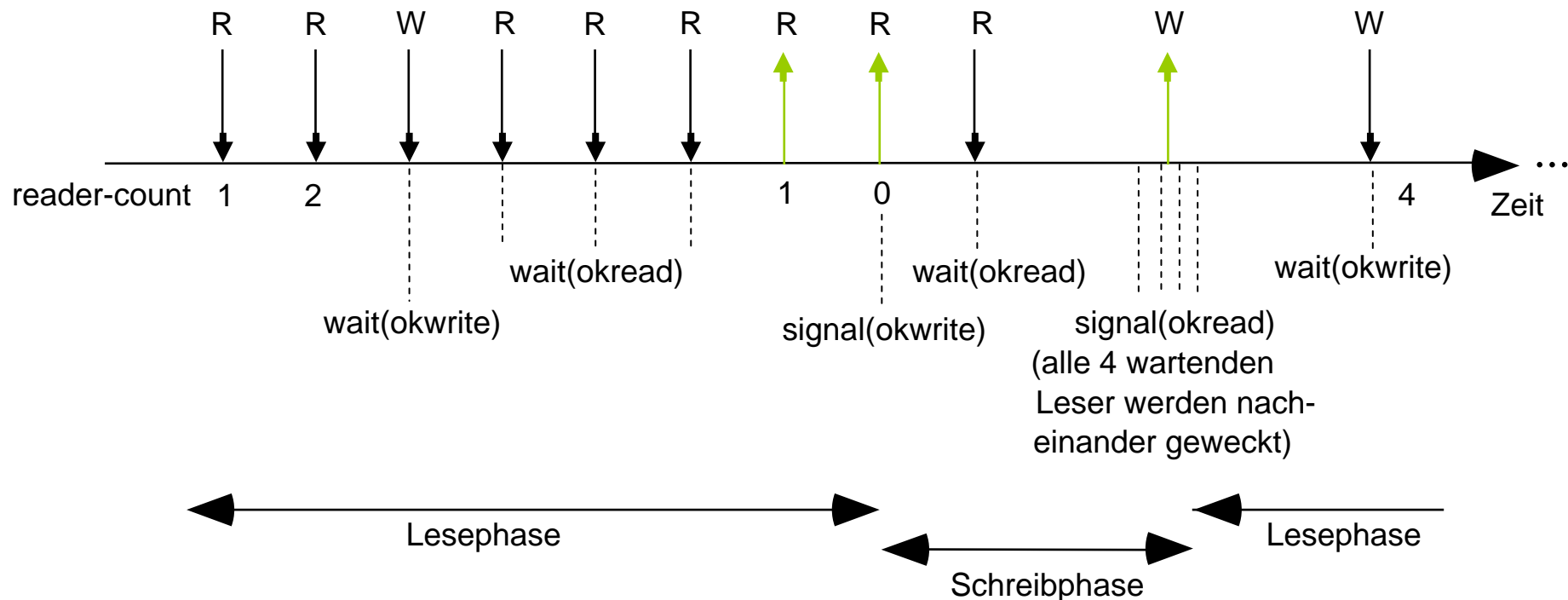
```
repeat
  RW.startwrite;
  Schreiben;
  RW.endwrite;

  andere Operationen;

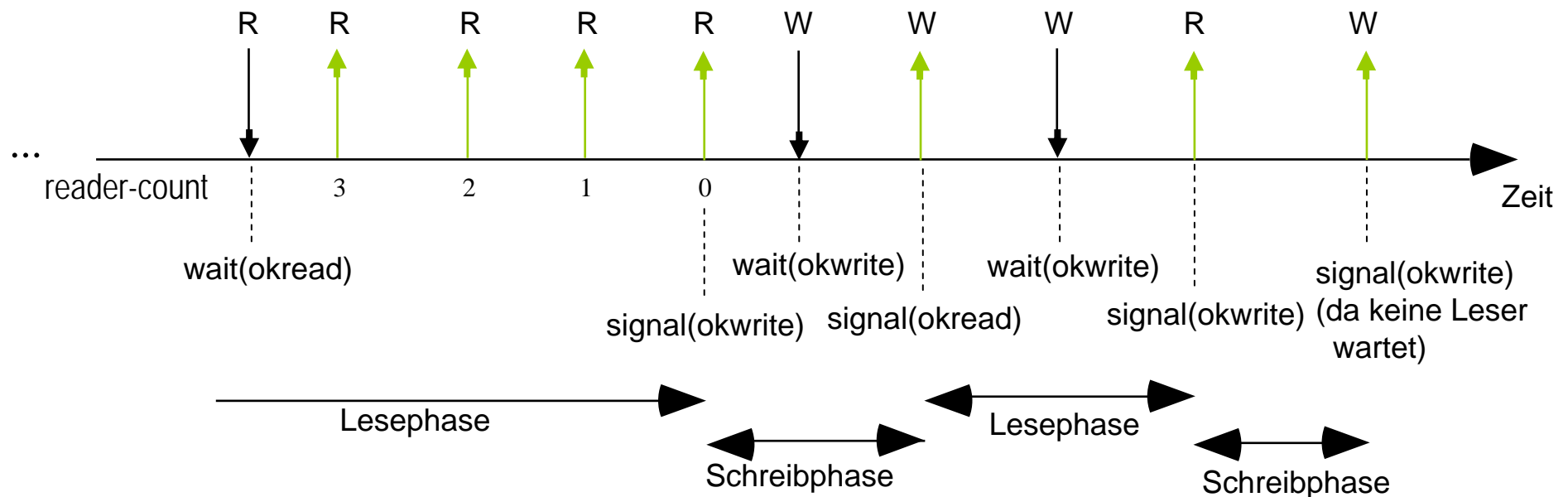
until FALSE;
```

3.6 Monitore / Drittes Reader-Writer-Problem

Ablaufbeispiel für das 3. Reader-/ Writer-Problem:



3.6 Monitore / Drittes Reader-Writer-Problem



3.7 Datenbankorganisation

Praktische Anwendung für atomare Zugriffe auf kritische Bereiche:

→ Datenbankorganisation

Hauptaufgabe ist die Sicherung, Wiedergewinnung und Konsistenzerhaltung großer Datenbestände durch **Transaktionen**. Diese sind eine Menge von Operationen, die Lese- und Schreibzugriffe auf Datenbestände durchführen.

Problem: **Atomarität** der Transaktionen zur Garantie der **Datenkonsistenz**

Eine **atomare** Operation ist nur gültig, wenn sie komplett beendet wurde, andernfalls ist alles ungültig, was sie ausgeführt hat.

Zunächst

- Betrachtung einer zentralen Datenbank (am Ende der Vorlesung auch für verteilte DB)
- Bearbeitung einzelner Transaktionen, d.h. nicht paralleles Ausführen
- Beispiel für Transaktionen sind Reise-, Flug- oder Hotelbuchungen usw.

3.7 Datenbankorganisation

Transaktionsablauf:

- TA-Start
- Operationen
- Abschluss einer Transaktion:
 - bei Erfolg ➔ TA-COMMIT Operation
 - bei Misserfolg, d.h. ein Fehler ist aufgetreten ➔ TA-ABORT Operation

Im Falle einer TA-ABORT Operation muss eine **Recovery-Operation**, z.B. ein so genanntes **Rollback** ausgeführt werden, um einen konsistenten Datenzustand wieder herzustellen.

Rollback wird ausgeführt durch

- undo Funktion
- redo Funktion

3.7 Datenbankorganisation

Probleme bei Einzelausführung von Transaktionen:

- Auftreten von Systemfehlern

Probleme bei paralleler Ausführung von Transaktionen:

- Auftreten von Systemfehler
- Transaktionen stören sich gegenseitig, z.B. durch Deadlocks

Speichertypen bei Rechnersystemen:

- flüchtiger Speicher → **volatile storage**, z.B. Hauptspeicher, RAM
- nichtflüchtiger Speicher → **nonvolatile storage**, z.B. Platte
- stabiler Speicher → **stable storage**, z.B. CD-ROM

Annahme:

- Die Daten in einem flüchtigen Speicher sind bei einem Systemfehler verloren.
- Die Daten in nichtflüchtigem oder stabilem Speicher überstehen einen Systemfehler.

Speicherart	Sicherheit	Zugriff
flüchtig	niedrig	schnell
nicht-flüchtig stabil	hoch	langsam

3.7 Datenbankorganisation

Recovery-Operationen benutzen flüchtigen und nichtflüchtigen Speicher

Einfachste Methode zur Recovery bei Systemfehler → **Logbuch**

Logbuch auf nichtflüchtigem Speicher enthält folgende Einträge jeder Operation innerhalb einer Transaktion:

- Transaktionsname
- Namen der Daten, die neu geschrieben wurden
- Alter Wert des Datums
- Neuer Wert des Datums

Typische Logbucheinträge

- Bei Start einer Transaktion T_i Logbucheintrag **T_i starts**
- Jede **Schreiboperation** im Logbuch eintragen
- Bei Erfolg Logbucheintrag **T_i commits**

3.7 Datenbankorganisation

Fehlerhafte Transaktionen lassen sich bereinigen. Dazu zwei idempotente Funktionen, d.h. Mehrfachausführung der selben Operation ändert nichts:

undo(T_i): durch T_i überschriebene Daten werden auf ihre **alten** Werte zurückgesetzt

redo(T_i): alle durch T_i aktualisierte Daten werden auf ihre **neuen** Werte gesetzt

Transaktion T_i schlägt fehl: mit **undo(T_i)** den Zustand vor Beginn von T_i wiederherstellen.

Bei einem Systemfehler:

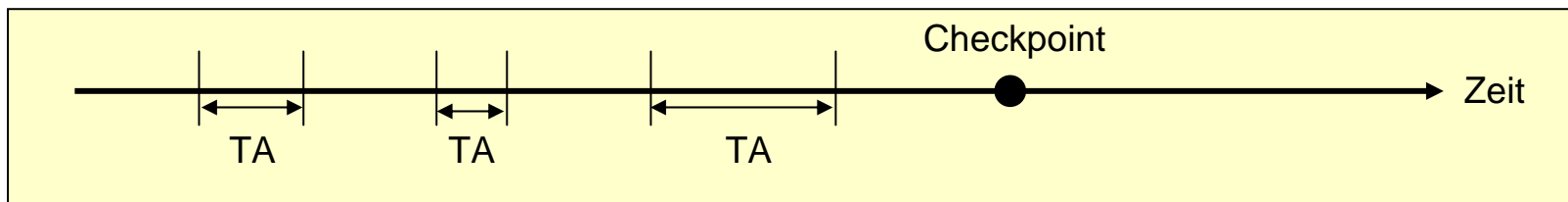
- Logbuch enthält den Eintrag T_i **starts** aber nicht T_i **commits** → **undo(T_i)**
- Logbuch enthält den Eintrag T_i **starts** und T_i **commits** → **redo(T_i)**

3.7 Datenbankorganisation

Um im Fehlerfall die Anzahl der zu wiederholenden Operationen zu begrenzen, werden

Checkpoints (CP) eingeführt.

- CP können gesetzt werden, wenn alle laufenden Transaktionen korrekt terminiert sind.
- Alle im flüchtigen Speicher gehaltenen Daten und Logbuch-Einträge werden auf nichtflüchtigen Speicher übertragen.
- Die gesamte **Vergangenheit** vor dem **Checkpoint** wird als korrekt gesetzt.
- Jeder **Checkpoint** wird in das Logbuch eingetragen.



Dadurch **undo(T_i)** und **redo(T_i)** nur noch für Transaktionen T_i anwenden, die nach dem Checkpoint starten.

3.7 Datenbankorganisation

Ablauf der Recovery-Operation:

- Im Logbuch ist die erste Transaktion nach dem letztem Checkpoint zu finden.
- Für diese und alle nachfolgenden Transaktionen T_i prüfen:
 - Logbuch enthält nicht den Eintrag T_i **commits** → **undo**(T_i)
 - Logbuch enthält den Eintrag T_i **commits** → **redo**(T_i)

Analogie für Checkpoints:

- Datenkommunikation über nicht-perfekte Leitungen
- Übertragungsfehler \Leftrightarrow gekippte, verlorene oder zusätzliche Bits

Empfänger kann sehr oft Fehler entdecken, manchmal sogar korrigieren, wenn der Sender Redundanz in die Daten einfügt.

3.7 Datenbankorganisation

Beispiel:

Sender könnte Code verwenden, bei dem die Codeworte eine feste Länge von n Bits und eine **Hammingdistanz** $\geq X$ haben.

Codewort $u = u_1, u_2, \dots, u_n$ ($u_i \in \{0,1\}$)

Codewort $v = v_1, v_2, \dots, v_n$ ($v_i \in \{0,1\}$)

$$h(u,v) = \sum_{i=1}^n |u_i - v_i|$$

Die Hammingdistanz $h(u,v)$ beschreibt die Zahl der Stellen, an denen sich die Codeworte unterscheiden.

Je größer die Hammingdistanz, desto kleiner ist die Zahl zulässiger Codeworte.

3.7 Datenbankorganisation

Im Allgemeinen gilt:

- Übertragungsfehler treten eher selten auf. Beispiel: $P(\text{Bit } i \text{ falsch}) = 10^{-12}$ bei Glasfaser.
- Wenn die Fehler unabhängig wären $\rightarrow P(5 \text{ Bitfehler bei Glasfaser}) \approx (10^{-12})^5 = 10^{-60}$.

Definition **Einzugsbereich** bei Hammingdistanz-Codes:

Wenn Hammingdistanz $= 2t + 1$ ($t \in \mathbb{N}$), dann gilt für den **Einzugsbereich** von Codewort x :

$$\text{Einzugsbereich}(x) = \{x\} \cup \{v \mid 1 \leq d(x, v) \leq t\}$$

Wenn eine solche Bitfolge v ankommt:

- Decodiere sie zu x , d.h. nehme an, sie sei durch Störung aus x entstanden.
- Maximum-Likelihood-Prinzip \rightarrow Forward-Error-Correction (FEC) Code.

3.7 Datenbankorganisation

Viele andere Codes können Fehler **nicht korrigieren**, aber viele Fehler **entdecken**.

Einfachstes Beispiel: **Parity-Bit** ergänzt Anzahl der Einsen auf geraden Wert.

```
1 1 0 0 1 0 0 1
                ↑
1 1 0 0 1 0 1 0
                ↑
              parity
```

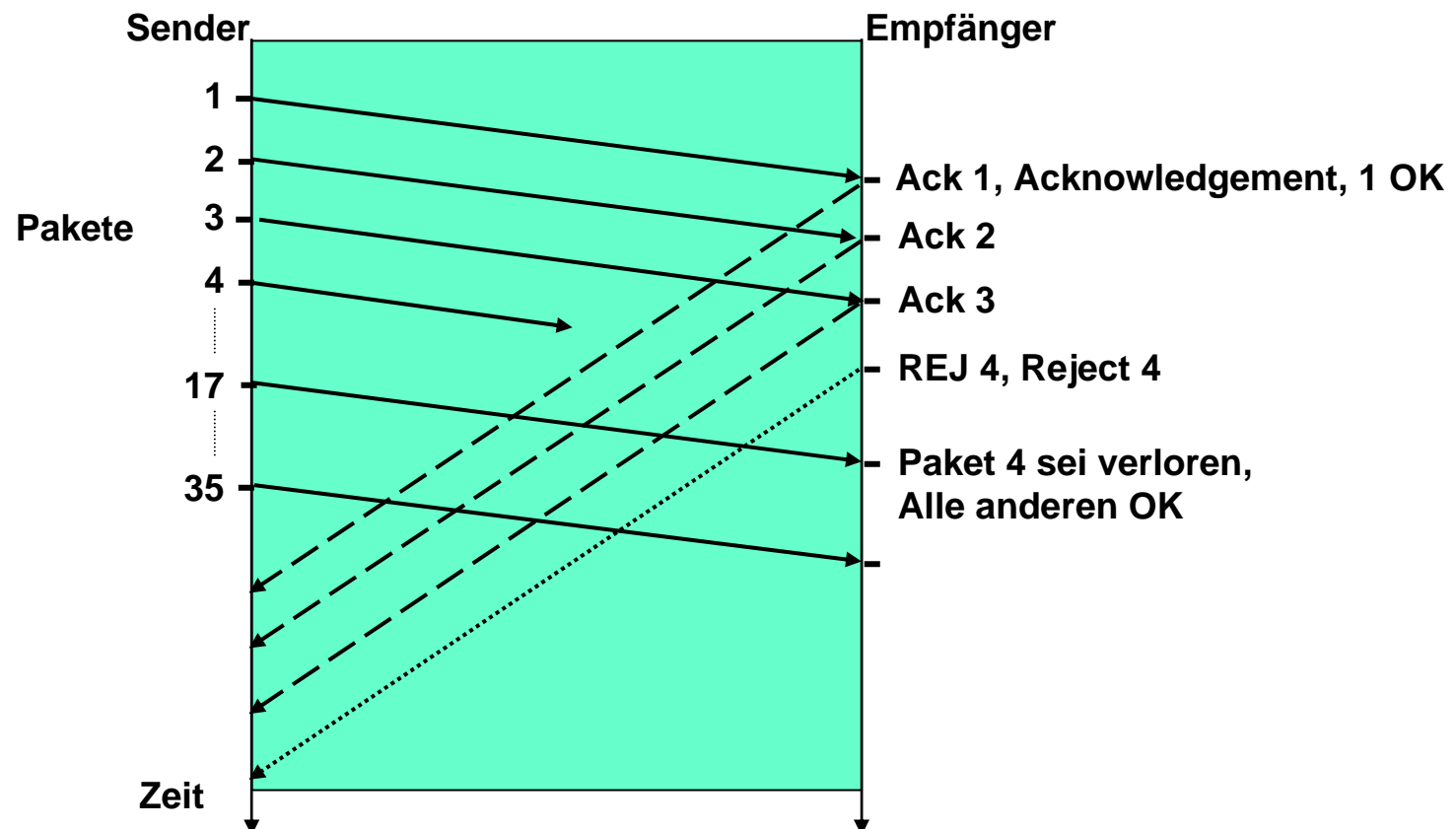
Beispiel:

- Sender verwendet Codeworte v_1, \dots, v_n mit gerader Parität, d.h. $\sum v_i$ ist gerade.
- Wenn Empfänger v_1', \dots, v_n' mit ungerader Parität empfängt → Empfänger hat 1-Bit-, 3-Bit- oder $(2m+1)$ -Bit-Fehler **entdeckt**.
- $(2m)$ -Bit-Fehler bleiben unentdeckt !

Wenn Empfänger Fehler entdeckt, wird er den Sender **informieren**. Der Sender wird die Übertragung **wiederholen**.

3.7 Datenbankorganisation

Nach entdecktem Übertragungsfehler: Entweder eigener Korrekturversuch (FEC, Forward-Error-Correction) oder Automatic-Repeat-Request, ARQ.
Mehrere Möglichkeiten:



3.7 Datenbankorganisation

Möglichkeit 1: Empfänger sendet **REJ 4**, d.h. alle Pakete bis 3 OK (man hätte auf Ack 1 – Ack 3 sogar verzichten können). Bitte sende mir Nr. 4 **und** alle folgenden, die schon gesendet wurden (5, 6,), nochmals! Auch: **Go-Back-n Verfahren**.

- **Vorteil für Empfänger:** Braucht nur einen Pufferplatz.
- **Nachteil für Sender:** Muss viel überflüssig (zweimal) senden.

Möglichkeit 2: Empfänger sendet **SREJ 4** (Selective Reject 4), d.h. alle Pakete bis 3 OK. Bitte sende mir **nur** Nr. 4 nochmals!

- **Vorteil für Sender:** Weniger Wiederholungen.
- **Nachteil für Empfänger:** Viel Pufferplatz (5,6,7,...,35 schon da - puffern, dann erst 4).

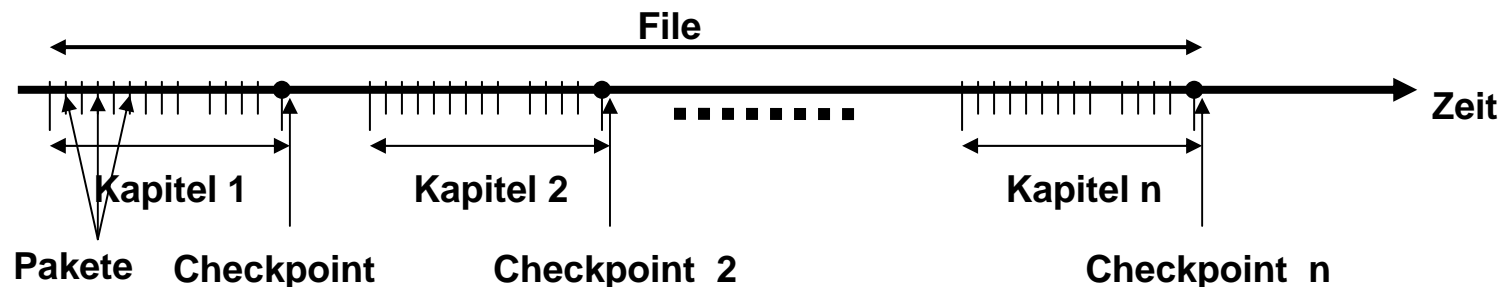
3.7 Datenbankorganisation

SREJ i und **SREJ k** ($i \neq k$) können nicht koexistieren, d.h. nur ein **SREJ** zu gegebener Zeit. Wenn $i < k \rightarrow$ **SREJ k** würde **i** als OK quittieren, ist aber **nicht OK**.

Möglichkeit 3: Empfänger sendet **SREPEAT 4** (Selective Repeat 4). Wie SREJ 4, **ohne** alle Pakete bis 3 OK.

Beispiel für Checkpoints: Dateiübertragung

Zweckmäßig: Einteilung in **Kapitel**, Kapitelende = Checkpoint



3.7 Datenbankorganisation

Naive Methode (ohne Kapitel):

- Prüfe Pakete mit ARQ oder FEC
- Prüfe Gesamtfile am Ende nochmals. Vorsichtsmaßnahme, falls Paketfehler unentdeckt bleibt. Im Fehlerfall ist das gesamte File neu zu übertragen!

Besser mit Checkpoints:

- Prüfe kapitelweise
- Prüfe Gesamtfile am Ende nochmals. Vorsichtsmaßnahme, falls Paketfehler im Kapitel unentdeckt bleibt. Im Fehlerfall ist hier jedoch nur betreffendes Kapitel zu wiederholen!

3.7 Datenbankorganisation

Zurück zur **Transaktionsverarbeitung**

- **Logbuch:** alles eintragen: Start, Änderungen, Checkpoints, Commits, ...
- Zunächst Beschränkung auf den Fall, dass nur **eine** Transaktion gleichzeitig aktiv ist.

Wann wird die Konsistenz eines Datenbank-Systems nicht gestört?

– **Antwort:** Wenn Transaktionen T_1, T_2, \dots, T_n in irgendeiner Reihenfolge hintereinander ausgeführt wurden (es gibt $n!$ Möglichkeiten) dann OK, d.h. Zustand bleibt konsistent.

- **Seriellles Schedule** T_{i_1} vor T_{i_2} vor ... vor T_{i_n}

Konsistenz bleibt ebenfalls erhalten, wenn aktuelle Ausführung der TAs zwar nicht seriell, aber in ihrer Wirkung gleich mit einem seriellen Schedule ist.

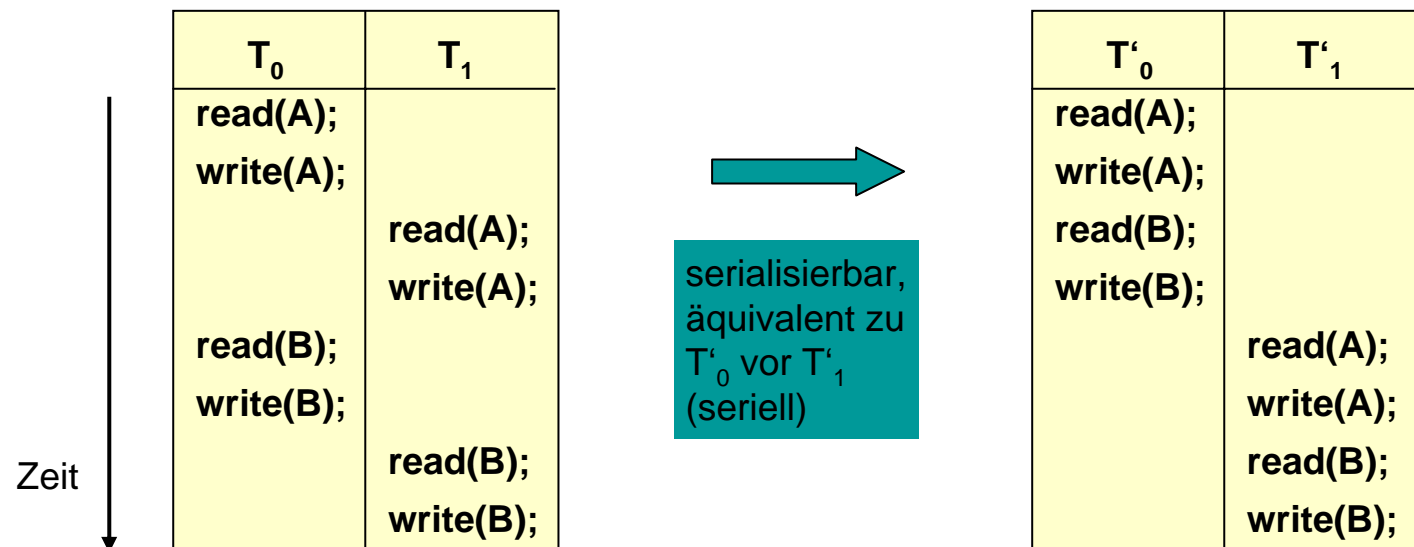
- Man nennt dies ein **serialisierbares** Schedule.
- Hinreichend für Konsistenzerhaltung: **Serialisierbarkeit**
- Hinreichend für Serialisierbarkeit: **Atomarität** einer Transaktion

3.7 Datenbankorganisation

Serialisierbarkeit

- Gegeben seien Transaktionen T_0, T_1, \dots, T_n
- **Schedule**: verzahnte Ausführung der Transaktionen, z.B. einige Operationen von T_2 dann einige von T_1 usw.
- Ein Schedule heißt **seriell**, wenn er der Ausführung einer beliebigen Permutation der Transaktionen T_0, T_1, \dots, T_n entspricht. D.h. die Transaktionen werden nicht ineinander verzahnt, sondern jede einzelne auf einmal ausgeführt, dafür aber in einer beliebigen Reihenfolge

Beispiel eines nicht-seriellen, aber serialisierbaren Schedules:



3.7 Datenbankorganisation

Nicht-serielle Schedules sind nicht notwendigerweise inkorrekt

Schedule aus vorherigem Beispiel ist nicht-seriell, aber korrekt:

Wie prüft man die Serialisierbarkeit?

➤ Probleme können auftreten bei lesenden bzw. schreibenden Zugriffen auf gemeinsame Daten.

Definition:

- **WM(i)** heißt **Write-Menge** der Transaktion T_i , d.h. die Menge der Daten, auf die eine write-Operation von T_i zugreift.
- **RM(i)** heißt **Read-Menge** der Transaktion T_i , d.h. die Menge der Daten, auf die eine read-Operation von T_i zugreift.

T_0	T_1
read(A); write(A);	read(A); write(A);
read(B); write(B);	read(B); write(B);

Serialisierbares
Schedule

3.7 Datenbankorganisation

Definition:

Für zwei Transaktionen T_i und T_j liegt ein potenzieller Konfliktfall vor, wenn gilt:

$$(WM(i) \cap WM(j)) \cup (WM(i) \cap RM(j)) \cup (RM(i) \cap WM(j)) \neq \emptyset$$

- Ist $WM(i) \cap WM(j) \neq \emptyset$, so kommt es darauf an, welche der beiden Transaktionen zuletzt schrieb; die write-Operation der anderen Transaktion geht verloren.

T_i	T_j
write(C);	write(C);

geht verloren

- Ist $WM(i) \cap RM(j) \neq \emptyset$, bzw. $RM(i) \cap WM(j) \neq \emptyset$, so können in einer Transaktion unmittelbar hintereinander ausgeführte Leseoperationen unterschiedliche Ergebnisse liefern.

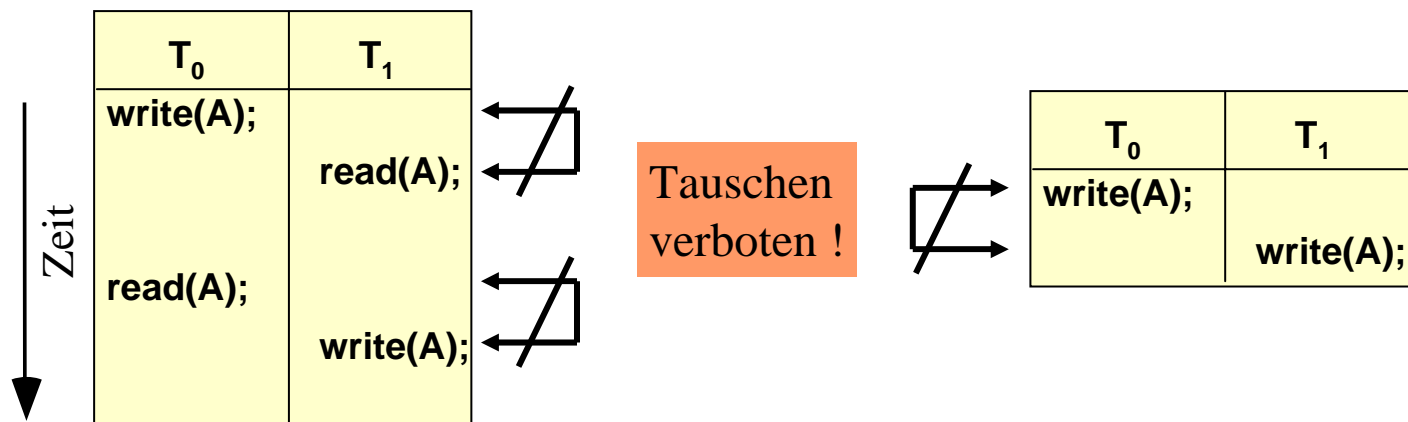
T_i	T_j
write(D);	read(D); (alter Wert)
	read(D); (neuer Wert)

3.7 Datenbankorganisation

- Betrachtet man das vorherige Beispiel , so stellt man fest, dass hier $RM(T_0) \cap WM(T_1) \neq \emptyset$ gilt.

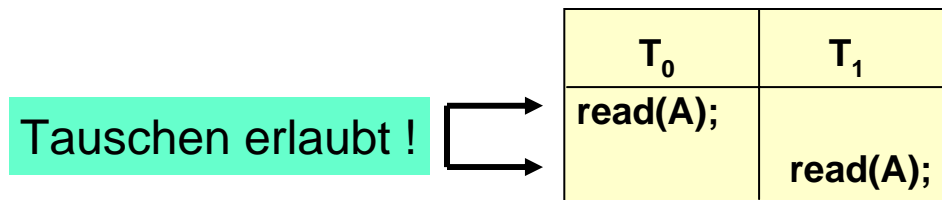
Dies zeigt, dass in **Einzelfällen** die Konsistenz erhalten bleiben kann, auch wenn ein Konfliktrisiko besteht.

- Swapping:** Sukzessives Vertauschen von zeitlich benachbarten und nicht in Konflikt stehenden Operationen.



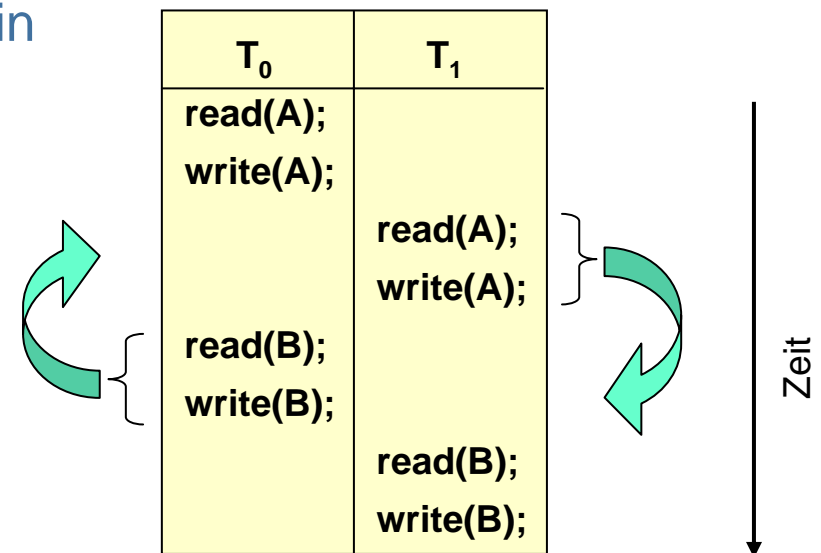
- Zwei Operationen dürfen **nicht** miteinander zeitlich getauscht werden, wenn sie sich auf **dasselbe Datum** beziehen **und** mindestens eine davon eine **write-Operation** ist.

3.7 Datenbankorganisation



Folgerung: Schedules sind **konfliktserialisierbar**, wenn sich das in Konflikt stehende Schedule durch **Swapping** in ein serielles umwandeln lässt.

Beispiel eines konfliktserialisierbaren Schedules: die angedeuteten Pfeile erfordern mehrere einzelne **Swap-Operationen**.



3.7 Datenbankorganisation

Hinreichend für Serialisierbarkeit: **Atomarität einzelner Transaktionen**

Das kann z.B. mit **Wait/Signal**-Konstruktion erreicht werden.

Problem: Für allgemeine Zwecke oft zu umständlich und ineffizient. Effizientere Protokolle zur Gewährleistung der Serialisierbarkeit (Konsistenzerhaltung):

- Methodenklasse A: **Sperrprotokolle (Locking protocols)**
- Methodenklasse B: **Timestamp-Mechanismen**

Sperrmechanismen:

Atomare Sperren (**lock**) vor dem Zugriff auf Daten, die auch von anderen genutzt werden können, setzen und nach dem Zugriff wieder freigeben (**unlock**).

Beim Design der Protokolle ist die **Granularität** der Sperren entscheidend:

- Grobgranular, d.h. geringe Anzahl von Sperren, die größere Gebiete innerhalb der Datenbank abschließen
- Feingranular, d.h. hohe Anzahl von Sperren mit zugehörigem Verwaltungsaufwand

3.7 Datenbankorganisation

Gröbstgranulare Sperre: Verwehren des Zugangs zu Datenbeständen für andere Prozesse, z.B.

```
repeat
    wait(s);
    Transaktion;
    signal(s);
until FALSE;
```

Ineffizient, da während der Transaktionszeit kein anderer etwas tun kann.

Feingranulare Sperre bezieht sich nur auf **eine** Variable.

Nachteile:

- Eine Transaktion kann viele davon brauchen → **Verwaltungsaufwand**
- Hohes **Verklemmungsrisiko** → Deadlock

Sperrtypen:

- Exclusive Locks: Wer diese hat, darf lesen und schreiben (auf gesperrten Bereichen).
- Shared Locks: Es darf nur gelesen werden.

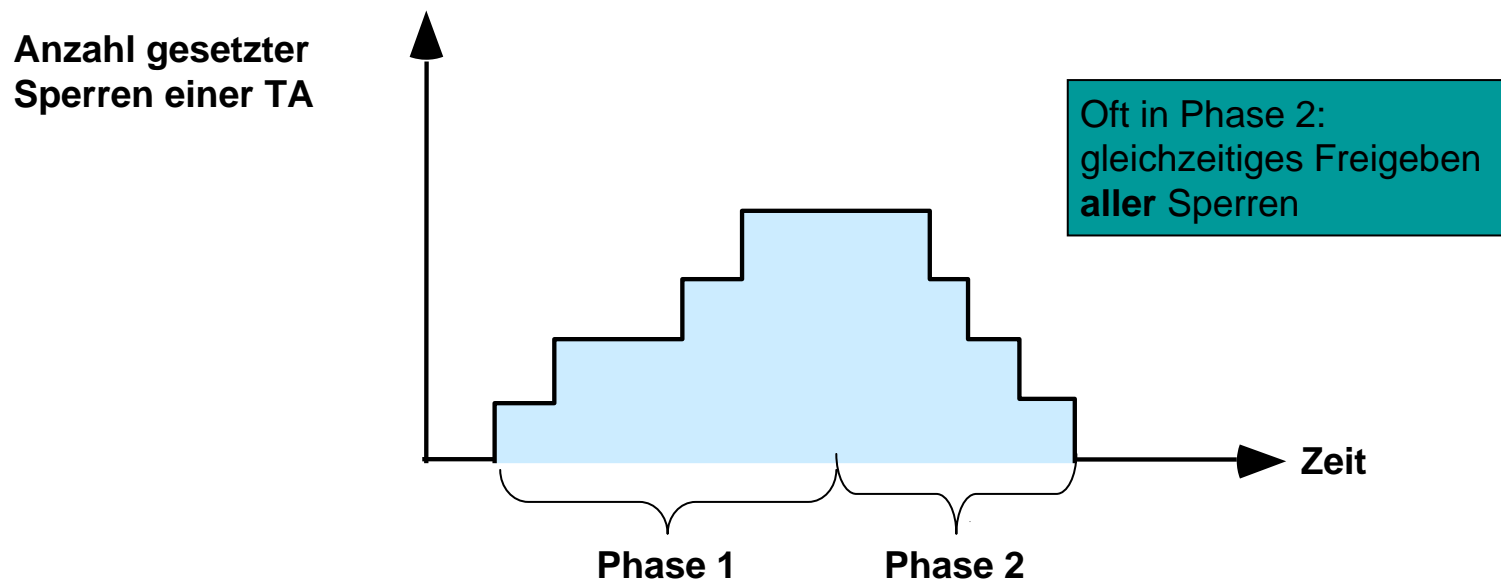
3.7 Datenbankorganisation

Zwei-Phasen-Sperrprotokoll (Two-Phase-Locking, 2 PL)

Transaktion fordert erst dann Sperren an, wenn sie benötigt werden.

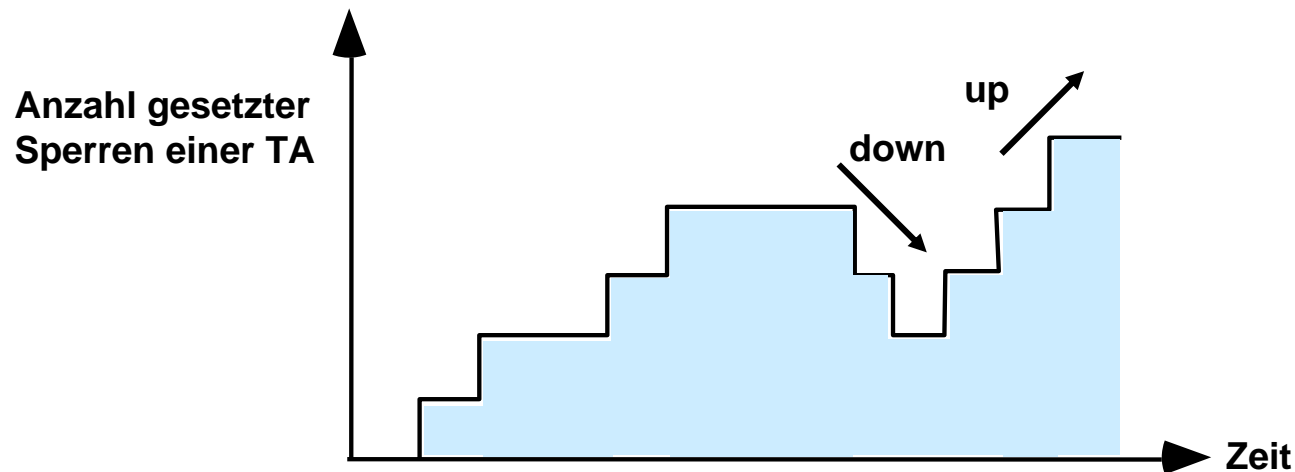
Protokoll besteht aus einer **up**- und einer **down**-Phase

- Erste Phase (**up**): Zunehmende Anzahl von Sperren, neue Sperren nur in dieser Phase.
- Zweite Phase (**down**): Alle aufgebauten Sperren wieder freigeben.



3.7 Datenbankorganisation

Unzulässiges Verhalten: Nach Freigabe einer Sperre eine neue anfordern.



Probleme:

- down-Phase: Datenbankeinträge wurden verändert und danach freigegeben.
- up-Phase: Zusätzlich benötigte Sperren sind unter Umständen nicht erhältlich (Deadlock)
- ➔ Einige Objekte werden geändert, andere aber nicht, d.h. keine Konsistenz der Daten.

3.7 Datenbankorganisation

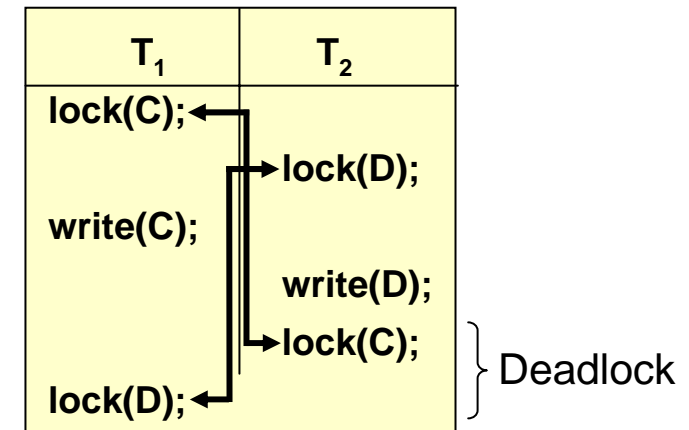
Bei einem Deadlock muss mindestens eine Transaktion ein Rollback durchführen.
Hätte diese Transaktion schon einzelne Sperren freigegeben (d.h. dort Werte geändert und „committed“), dann würden andere Transaktionen auf die neuen, aber nach dem Rollback ungültig gewordenen Werte, zugreifen.

➤ Inkonsistenz!

- Einfachste Möglichkeit, um Deadlocks aufzulösen, ist **Simple 2 PL**: Sobald eine Transaktion eine Sperre anfordert, die bereits belegt ist, gibt sie auf und macht ein Rollback.

- Später, evtl. nach zufälliger Wartezeit, einen Restart durchführen, um eine Wiederholung dieser Situation unwahrscheinlicher zu machen.
- Methode der zufälligen Wartezeit nach Konflikt wird auch bei lokalen Netzen verwendet, etwa bei Ethernet (**exponential backoff**) nach Kollisionserkennung auf dem gemeinsamen Medium.

Beispiel:



3.7 Datenbankorganisation

Generell ist Two-Phase-Locking brauchbar bei:

- Kurzen Transaktionen
- Transaktionen mit wenigen Sperren
- Transaktionen, die sehr feingranular arbeiten, d.h. bei denen Sperrwünsche sich selten ins Gehege kommen

Nachteil von Two-Phase-Locking:

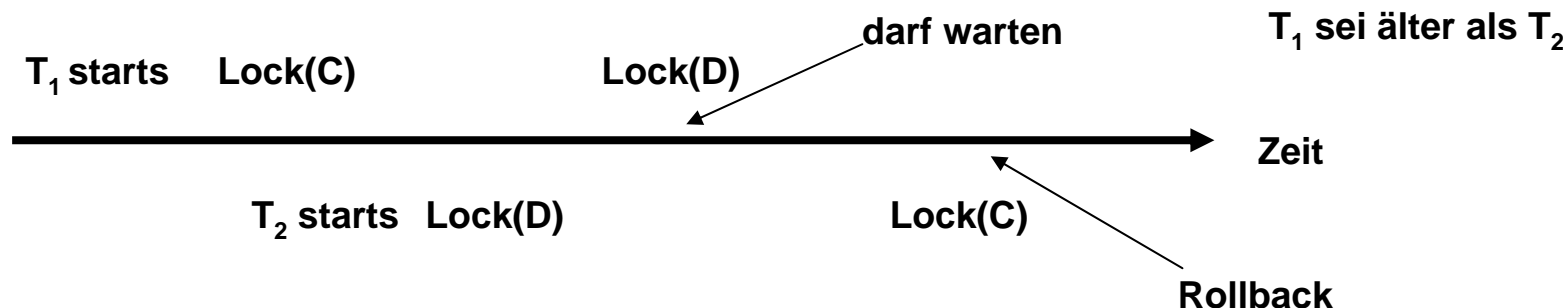
- Verschwendung von Arbeitszeit, wenn **alte** Transaktion bereits viele Sperren hat und kurz vor Schluss aufgibt

3.7 Datenbankorganisation

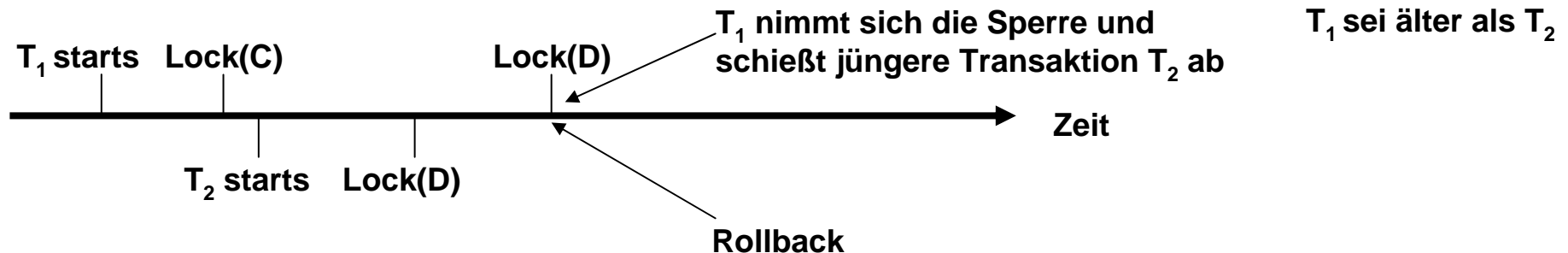
Andere Möglichkeiten beziehen sich auf das **Alter** der Transaktionen:

1. **Wait-Die**: Alter von Transaktionen wird durch Zeitstempel bestimmt. Ältere Transaktion wartet (**wait**) auf Sperre, die zur Zeit belegt ist. Jüngere Transaktion muss aufgeben (**die**) → Rollback

2. **Wound-Wait**: Ältere Transaktion nimmt sich die Sperre (**wound**), die zur Zeit belegt ist. Jüngere Transaktion wird rausgeworfen (Rollback). Wenn jüngere Transaktion Sperre will, darf sie warten (**wait**).



3.7 Datenbankorganisation



Konsequenzen für **ältere** Transaktionen:

- **Wait-Die:** Je älter die Transaktion, desto länger wartet sie.
- **Wound-Wait:** Ältere Transaktion wartet nie.

Konsequenzen für **jüngere** Transaktionen:

- **Wait-Die:** Jüngere Transaktion muss eventuell oft neu starten, d.h. ggf. viele Rollbacks.
- **Wound-Wait:** Jüngere Transaktion wartet, weniger Rollbacks.

3.7 Datenbankorganisation

Methodenklasse B zur Konsistenzerhaltung: Timestamp-Mechanismen

- Zeitliche Anordnung von Transaktionen, die einen Konflikt verursachen.
- Jede Transaktion T_i erhält zur Startzeit einen **Timestamp** $TS(T_i)$ von zentraler oder lokaler Uhr. $TS(T_i)$ muss eindeutig sein.
- Bei Konflikt: Transaktion T_i erhält Vorrang vor T_j wenn $TS(T_i) < TS(T_j)$.
- Verfahren kann auf Read- bzw. Write-Operationen beschränkt werden.

Transaktionen sind z.B. dann **serialisierbar**, wenn ihre Wirkung so ist, als ob sie in der Reihenfolge der Timestamps atomar ausgeführt würden, d.h.

T_i vor T_j , falls $TS(T_i) < TS(T_j)$.

Wie kann überprüft werden, ob Transaktionen in diesem Sinne **serialisierbar** sind ?

- Lösung: **Timestamp Ordering Protocol**

3.7 Datenbankorganisation

Gegeben: T ältere Transaktion und T* jüngere Transaktion.

Fall 1

T	T*
⋮	⋮
	write(A);
read(A);	⋮
⋮	

T: read(A) verboten !

Fall 2

T	T*
⋮	⋮
	read(A);
write(A);	⋮
⋮	

T: write(A) verboten !

Fall 3

T	T*
⋮	⋮
	write(A);
write(A);	⋮
⋮	

T: write(A) verboten !

Fall 4

T	T*
⋮	⋮
write(A);	
⋮	read(A);
	write(A);
	⋮

T*: read(A) oder
T*: write(A) möglich !

Fall 5

T	T*
⋮	⋮
read(A);	
⋮	write(A);
	⋮

T*: write(A) erlaubt !

3.7 Datenbankorganisation

Implementierung: Pro Datum Q zwei Zeitwerte:

- W-timestamp(Q) = größter Zeitwert einer Transaktion (d.h. Timestamp der jüngsten TA), die eine erfolgreiche write-Operation auf Q ausgeführt hat.
- R-timestamp(Q) = größter Zeitwert einer Transaktion (jüngste TA), die Q erfolgreich gelesen hat.

Timestamp Ordering Protocol: Sei T_i eine Transaktion, die **read(Q)** ausführen will:

- Falls $TS(T_i) < W\text{-timestamp}(Q)$ (**Fall 1**) → **spätere** Transaktion hat bereits erfolgreich geschrieben, d.h. Q bereits geändert → read(Q) verwerfen und Transaktion T_i , mit entsprechend neuem Timestamp, neu starten.
- Falls $TS(T_i) \geq W\text{-timestamp}(Q)$ (**Fall 4**) → read(Q)-Operation ausführen und $R\text{-timestamp}(Q) = \max \{ R\text{-timestamp}(Q), TS(T_i) \}$

3.7 Datenbankorganisation

Timestamp-Protokoll (Fortsetzung):

Es sei T_i eine Transaktion, die **write(Q)** ausführen will:

- c) Falls $TS(T_i) < R\text{-timestamp}(Q)$ (**Fall 2**) \rightarrow Q wurde bereits früher gelesen.
(Würde T_i ihre write-Operation jetzt durchführen, so hätte eine andere Transaktion einen **alten** Wert gelesen.) \rightarrow write(Q) verwerfen und Transaktion T_i zurücksetzen und neu starten.
- d) Falls $TS(T_i) < W\text{-timestamp}(Q)$ (**Fall 3**) \rightarrow Schreibvorgang wäre bereits zum Zeitpunkt seiner Ausführung veraltet \rightarrow write(Q) verwerfen und Transaktion T_i zurücksetzen.
- e) Falls $TS(T_i) \geq \max \{ R\text{-timestamp}(Q), W\text{-timestamp}(Q) \}$ (**Fall 4 oder 5**)
 \rightarrow kein Konflikt, write-Operation ausführen und $W\text{-timestamp}(Q) = TS(T_i)$

3.7 Datenbankorganisation

Mit dem Timestamp Ordering Protocol wird Serialisierbarkeit sichergestellt. Deadlocks sind unmöglich, weil eine Transaktion entweder weiterkommt oder aufgibt, aber niemals wartet.

Nachteil: Alte Transaktionen machen häufig Rollback und zwar ggf. mehrfach hintereinander (**Cascading Rollback**).

Bessere Strategie: Warten bis Probleme durch andere Transaktionen sich gelöst haben. Warten durch Puffern der entsprechenden Requests.

Wenn Transaktion T_i

- **write(x)** ausführen will, dann dies solange aufschieben wie ältere Transaktion **read(x)** bzw. **write(x)** ausführen will und dies nicht getan hat.
- **read(x)** ausführen will, dann dies solange aufschieben wie ältere Transaktion **write(x)** ausführen will und dies noch nicht getan hat.

Inhaltsverzeichnis

4.1 Systemmodell und notwendige Bedingungen

- Was sind Deadlocks?
- Darstellungsarten von Prozessabhängigkeiten
- Notwendige Bedingungen für Deadlocks

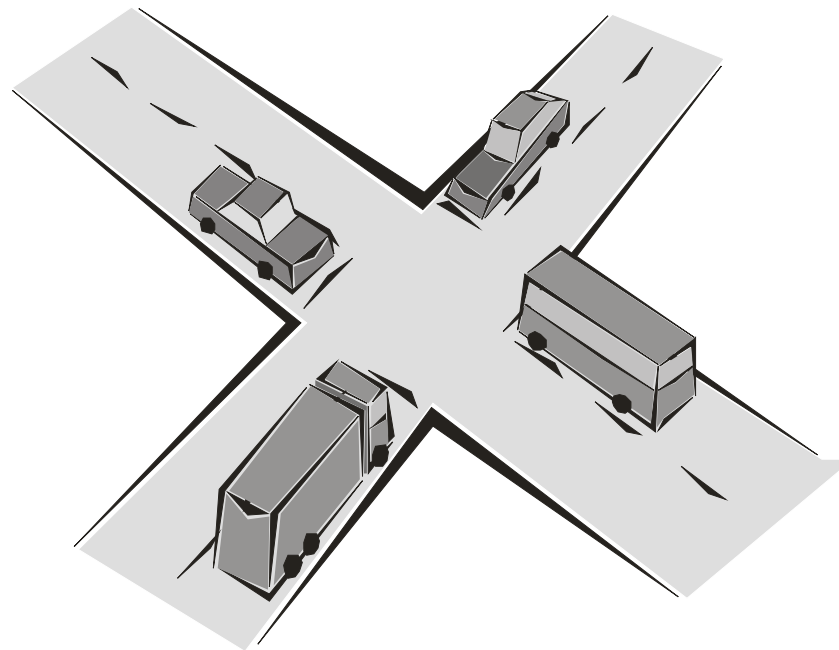
4.2 Gegenmaßnahmen

- Deadlock-Prevention
- Deadlock-Avoidance
- Deadlock-Detection
 - Banker's Algorithmus
 - Verfahren von Holt

4.1 Deadlocks

Deadlock bzw. Verklemmung:

- Systemzustand, in dem Prozesse auf Ereignisse warten, die niemals eintreten können.

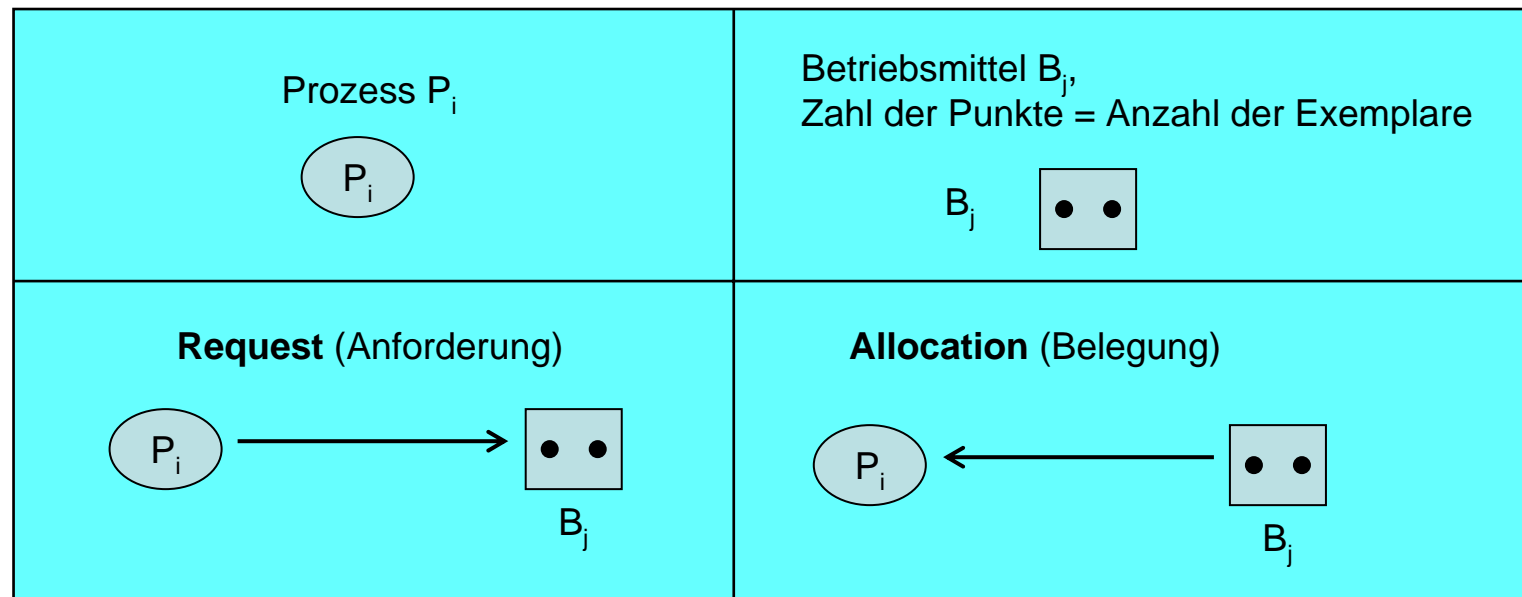


4.1 Request-Allocation-Graph

Annahme: $n \geq 2$ Prozesse sind für Deadlock notwendig, d.h. einzelner Prozess ist lauffähig.

- Prozesse P_1, \dots, P_n
- Betriebsmittel B_1, \dots, B_m , wobei eventuell mehrere Betriebsmittel vom Typ B_i möglich. Im Allgemeinen können die Betriebsmittel nur exklusiv genutzt werden.

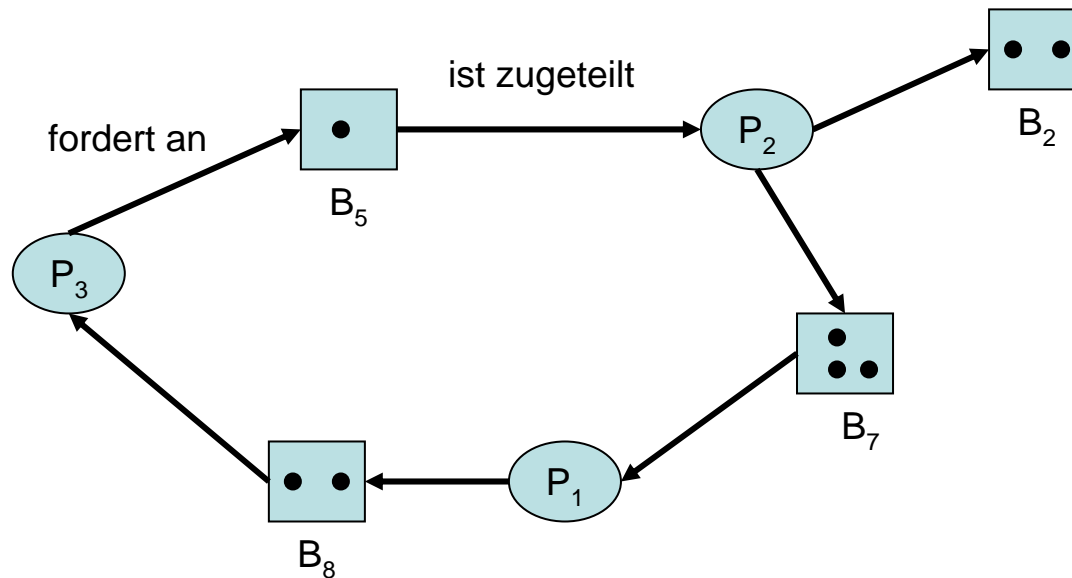
Graphische Zustandsbeschreibung:



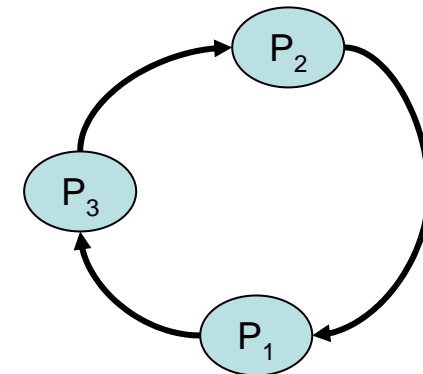
Bei Zuteilung wird ein Request-Pfeil sofort umgedreht.

4.1 Wait-for-Graph

Request-Allocation-Graph



Wait-for-Graph



Weglassen der Betriebsmittel \rightarrow Abhängigkeiten unter Prozessen

Systemzustand = δ = {Prozesse, zugeteilte Betriebsmittel}

4.1 Einfache Methode zur Vermeidung von Deadlocks

Sei δ_i der Zustand nach der i-ten Betriebsmittelzuteilung.

$$\delta_0 \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \delta_3 \rightarrow \delta_4 \rightarrow \delta_5 \rightarrow \dots$$

Startzustand, ist per
Definition sicher

Sei δ_i sicher \rightarrow prüfe vor Zuteilung des (i+1)-ten Betriebsmittels, ob δ_{i+1} sicher ist.

Falls

Ja \rightarrow zuteilen!

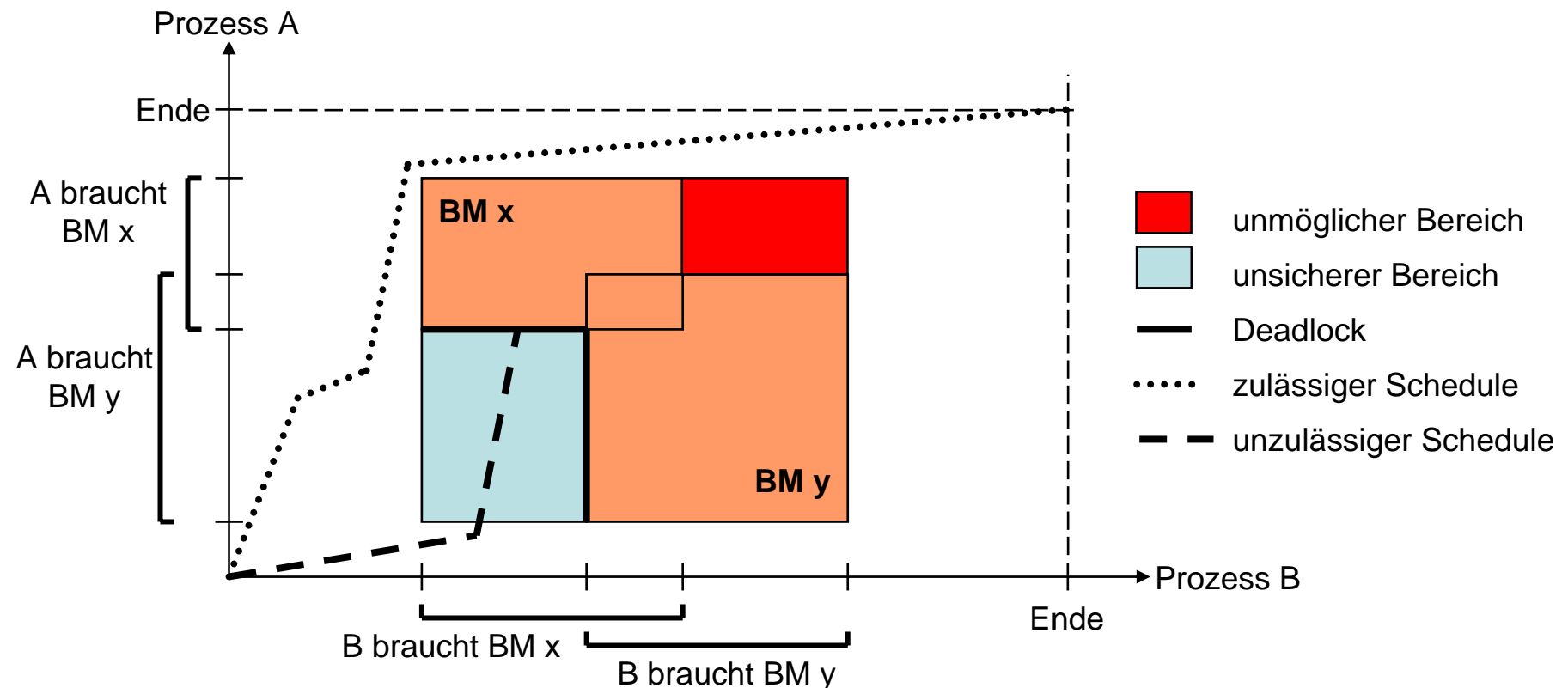
Nein \rightarrow verwerfen!

Definition: Systemzustand heißt **sicher** \Leftrightarrow Es gibt eine Reihenfolge $(P_{i_1}, \dots, P_{i_n})$, mit (i_1, \dots, i_n) einer Permutation von $(1, \dots, n)$, sodass für alle j der Prozess P_{ij} weitergeführt werden kann, wenn P_{ij} seine maximalen Anforderungen stellt und alle vorherigen Prozesse $P_{i_1}, \dots, P_{i_{j-1}}$ ihre Betriebsmittel zurückgegeben haben.

4.1 Prozessfortschrittsdiagramm

Systemzustand wird über Prozessfortschritt (Fortschrittsdiagramm) dargestellt

- Betriebsmittelbelegungen werden pro Prozess eingetragen
- Das Betreten eines unsicheren Bereiches führt zwangsläufig zum Deadlock



4.1 Notwendige Bedingungen für Deadlock

4 Bedingungen müssen gleichzeitig erfüllt sein:

a) Circular Wait:

Geschlossene Kette im Request-Allocation- bzw. Wait-for-Graphen.

b) Exclusive Use:

Kein Sharing von Betriebsmitteln.

c) Hold and Wait:

Weitere Betriebsmittelanforderungen durch Prozess sind erlaubt, ohne dass bisherige Betriebsmittel zurückgegeben werden müssen.

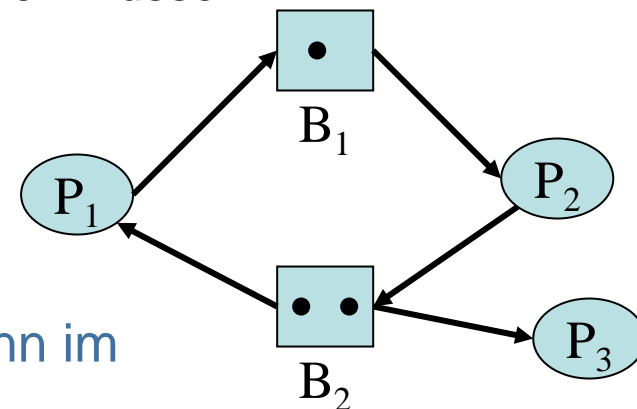
d) No Preemption:

Entzug von Betriebsmitteln ist nicht möglich.

Bemerkung:

(a) allein ist nicht ausreichend für Deadlock, denn im

Beispiel kann P3 sein Exemplar von B2 zurückgeben.



4.2 Gegenmaßnahmen zu Deadlocks

Prevention

- Deadlock unmöglich machen, indem eine der vier notwendigen Bedingungen verboten wird.

Avoidance

- Verhindern von Deadlocks durch Ausnutzen von Zusatzinformationen und Tests.

Detection

- Entdecken und beseitigen von vorhandenen Deadlocks.

Ignoring

- Es wird angenommen, dass Deadlocks so gut wie nie auftreten.
Überraschenderweise verfahren die meisten Betriebssysteme wie z.B. UNIX oder Windows NT nach dieser Methode. Vogel-Strauß-Ansatz.

4.2 Deadlock-Prevention

Vermeidung von Deadlocks durch den Ausschluss einer der 4 notwendigen Bedingungen

- No Exclusive Use, bzw. Sharing von Betriebsmitteln
 - ➔ Geht oft nicht! Beispiel: gleichzeitiges Drucken.

- Preemption
 - ➔ Gewaltsamer Betriebsmittelentzug.

- No Hold and Wait
 - ➔ Entweder: Rückgabe gehaltener Betriebsmittel vor Neuansforderung
 - ➔ Oder: Alles oder nichts, d.h. alle Betriebsmittel gleichzeitig anfordern.
 - Nachteil: Im Allgemeinen sehr ineffizient.

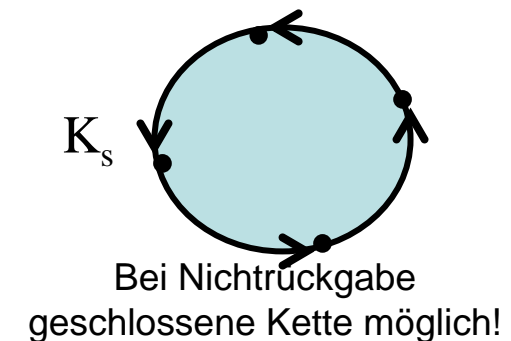
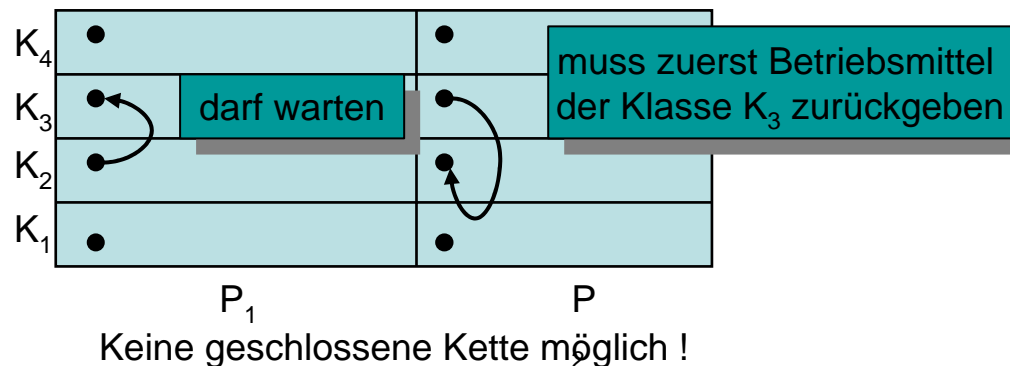
- No Circular Wait
 - ➔ z.B. durch Betriebsmittel-Hierarchie.

4.2 No Circular Wait

Deadlock-Prevention durch Betriebsmittel-Hierarchie:

Teile Betriebsmittel in Klassen K_1, K_2, \dots, K_h ein.

- Wenn Prozess P_j ein Betriebsmittel der Klasse K_r und ggf. niedrigerer Klassen hat, darf er Betriebsmittel einer Klasse K_u mit $u > r$ anfordern.
- Fordert er ein Betriebsmittel einer Klasse K_m an mit $m \leq r$, dann muss er zunächst alle Betriebsmittel der Klassen K_m, K_{m+1}, \dots, K_r zurückgeben und ggf. neu anfordern.



Wichtig:

- Auch bei Anforderung der gleichen Klasse zurückgeben!
- Extremfall: Nur eine Betriebsmittelklasse
➔ Rückgabe vor Neuansforderungen, also sequenzielle Betriebsmittelnutzung.

4.2 Deadlock Avoidance

Deadlock Avoidance

- Vermeidung von Deadlocks durch Ausnutzung von Zusatzinformationen!

Gängige Methode: Prüfe vor jeder Betriebsmittelzuteilung

- ob Gefahr für Deadlock besteht oder
- ob der Nachfolgezustand, d.h. Vektor der Zuteilungen an Prozesse, sicher ist.

Prüfung beruht auf pessimistischen Annahmen, d.h. worst case:

- Alle Prozesse fordern das Maximum ihrer zulässigen Betriebsmittelwünsche an und benutzen sie bis zum Prozessende.
- Diese Maxima seien bekannt. (MRU = Maximum Resource Usage)

4.2 Deadlock-Avoidance / Beispiel

Beispiel: 1 Betriebsmittel mit 12 Exemplaren

➤ $P_i: a (+b)$ bedeutet, dass P_i bereits a Exemplare hat und noch maximal b weitere anfordern darf, d.h. $a + b = \text{MRU}$ von Prozess P_i .

P_1	P_1	P_1	P_1
P_1	P_3	P_3	
P_2	P_2		

12 Exemplare (3×4), wobei 3 Exemplare frei

Zustand:

$P_1: 5 (+5) \rightarrow \text{MRU}_1 = 10$

$P_2: 2 (+2) \rightarrow \text{MRU}_2 = 4$

$P_3: 2 (+7) \rightarrow \text{MRU}_3 = 9$

Behauptung: (P_2, P_1, P_3) ist eine zulässige Folge!

- P_2 kann 2 zusätzliche aus 3 freien erhalten. Er gibt $2 + 2 = 4$ zurück ➔ 5 frei.
- P_1 kann nun 5 erhalten, gibt später 5 mehr frei. ➔ 10 frei.
- P_3 kann 7 erhalten.

➔ Zustand ist sicher.

4.2 Beispiel für Deadlock-Avoidance

Zu Beachten: Aus einem sicheren Zustand kann ein unsicherer Zustand werden.

➤ Teile P_3 ein weiteres Betriebsmittel zu.

P_1	P_1	P_1	P_1
P_1	P_3	P_3	P_3
P_2	P_2		

Zustand:

$P_1: 5 (+5) \rightarrow MRU_1 = 10$

$P_2: 2 (+2) \rightarrow MRU_2 = 4$

$P_3: 3 (+6) \rightarrow MRU_3 = 9$

12 Exemplare (3×4), wobei 2 Exemplare frei

Annahme: Alle Prozesse stellen ihre maximalen Zusatzanforderungen.

- P_1 ist nicht bedienbar,
- P_3 ist nicht bedienbar,
- P_2 ist bedienbar. P_2 beendet seine Arbeit und gibt $2 + 2 = 4$ Exemplare frei.
- P_1 und P_3 sind weiterhin nicht bedienbar !

Es gibt einen Deadlock (P_1, P_3). Der Zustand ($P_1 = 5, P_2 = 2, P_3 = 3$) ist unsicher.

4.2 Banker's Algorithmus

Banker's Algorithmus: Der Algorithmus überprüft, ob ein Zustand sicher ist.

Anwendung:

- Deadlock-Avoidance: Es wird geprüft, ob der Folgezustand sicher ist.
- Deadlock-Detection: Es wird geprüft, ob ein Deadlock vorliegt und welche Prozesse daran beteiligt sind.

$Q_{j,s}^{\max}(k) :=$ zur Zeit k von P_j maximal zusätzlich anforderbare Betriebsmittel vom Typ B_s

$Q_{j,s}^{\text{akt}}(k) :=$ zur Zeit k von P_j aktuell angeforderte Betriebsmittel vom Typ B_s

$H_{j,s}(k) :=$ zur Zeit k von P_j gehaltene Betriebsmittel vom Typ B_s

$V_s(k) :=$ zur Zeit k verfügbare Betriebsmittel vom Typ B_s

Im Fall a) benutzt er die maximalen Wünsche, im Fall b) die aktuellen Zuweisungen.

Q = Query, H = Hold, V = Vrij (holländisch)

4.2 Banker's Algorithmus

Zustand ist unsicher \Leftrightarrow Es ex. eine Teilmenge D der Prozesse, sodass für keinen Prozess aus D seine maximale Anforderung erfüllbar ist, selbst wenn alle nicht in D befindlichen Prozesse alle Betriebsmittel zurückgeben.

Formal:

$$\exists D (D \neq \emptyset) \text{ mit } D \subset \{P_1, \dots, P_n\} \text{ mit } Q_{j,s}^{\max}(k) > V_s(k) + \sum_{P_r \notin D} H_{r,s}(k) \quad \forall j \in D$$

Zustand ist Deadlock \Leftrightarrow Es ex. eine Teilmenge D der Prozesse, sodass für keinen Prozess aus D seine aktuelle Anforderung erfüllbar ist, selbst wenn alle nicht in D befindlichen Prozesse alle Betriebsmittel zurückgeben.

Formal:

$$\text{wie oben, jedoch } Q_{j,s}^{\text{akt}}(k) \text{ anstelle } Q_{j,s}^{\max}(k): Q_{j,s}^{\text{akt}}(k) > V_s(k) + \sum_{P_r \notin D} H_{r,s}(k)$$

4.2 Banker's Algorithmus / Sicherheitsüberprüfung

Sicherheitsüberprüfung: Zunächst seien alle Prozesse P_i unmarkiert.

- (1) Durchsuche P_1, \dots, P_n bis erster nicht markierter Prozess gefunden mit $Q_{j,s}^{\max}(k) \leq V_s(k)$. Wenn es keinen solchen Prozess gibt, weiter bei (3).
- (2) $V_s(k) := V_s(k) + H_{i,s}(k)$, d.h. P_i wird beendet und gibt die bisher gehaltenen Betriebsmittel zurück. Markiere P_i und gehe zurück zu (1).
- (3) Wurden alle P_i markiert, dann ist das System bezüglich des Betriebsmittelvektors B_s sicher, andernfalls ist es unsicher.

Wiederholen mit jedem einzelnen Betriebsmittel B_1, \dots, B_m .

Anwendung:

- Zur Deadlock-Avoidance vor jeder Betriebsmittelzuteilung prüfen, ob der neue Zustand (kleineres $V_s(k)$) noch sicher ist.
- Zur Deadlock-Detection wird bei (1) die Bedingung in $Q_{i,s}^{\text{akt}}(k) \leq V_s(k)$ geändert.

4.2 Banker's Algorithmus / Beispiel

	P1	P2	P3	P4	P5	V (frei)
H (hat) Q (will)	1 5	1 4	1 3	1 2	1 1	1
H Q	1 5	1 4	1 3	1 2	- -	2
H Q	1 5	1 4	1 3	- -	- -	3
	⋮	⋮	⋮	⋮	⋮	

Die Reihenfolge der Ausführung, d.h. die Folge, in der die Prozesse getestet werden, hat keinen Einfluss auf übrigbleibende Prozesse, die nicht markierbar sind.

4.2 Banker's Algorithmus / Laufzeit

Laufzeit des Banker's Algorithmus:

- m sei die Anzahl der Betriebsmittelarten
- n sei die Anzahl der Prozesse
- ➔ Laufzeit: $O(m \cdot n^2)$

Im **best-case** beträgt die Laufzeit: $O(n \cdot m)$

Der **worst-case** lässt sich folgendermaßen plausibel machen:

- Im ersten Schritt werden alle n unmarkierten Prozesse P_1, \dots, P_{n-1} getestet, da erst der n -te passt. ➔ n
- Dann P_1, \dots, P_{n-2} prüfen, P_{n-1} passt. ➔ $n-1$
- usw.
- Letzter Schritt ➔ 1

In der Summe ergibt sich die Anzahl der Tests als:
$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

4.2 Verfahren von Holt / Laufzeit

Verfahren von Holt

- liefert eine Verbesserung der Laufzeit im Falle $m=1$.

Algorithmus:

- Sortiere die Betriebsmittelanforderungen der Prozesse nach aufsteigender Größe.
- Teste Prozesse der Reihe nach von der kleinsten bis zur größten Anforderung.

Dabei gilt: Falls kleine Anforderung nicht passt, dann größere auch nicht!

Eine Folge von n Elementen zu sortieren kann mittels Heapsort durchgeführt werden.

Aufwand ist $O(n \cdot \log n)$. Gesamtaufwand für Verfahren von Holt ist $O(n \cdot \log n)$.

Für m Betriebsmittel ergibt sich schließlich:

$$m \cdot O(n \cdot \log n) = O(m \cdot n \cdot \log n) \leq O(m \cdot n^2)$$

Bemerkung: Verbesserung ist nur für sehr große Werte von n merklich.

Inhaltsverzeichnis

5.1 Motivation der Scheduling-Strategien

5.2 Bedienzeitunabhängige Strategien

- FIFO, LIFO, LIFO-PR

5.3 Bedienzeitabhängige Strategien

- SPT, SRPT
- Exponential-Averaging
- Exkurs: Digitale Sprachübertragung

5.4 Andere Verfahren

- Round-Robin
- Priority-Scheduling
- Multilevel-Feedback-Queueing

5.5 Mehrprozessorsysteme

- Besonderheiten und Anomalien

5.1 Motivation

Gegeben:

- Ein Einprozessorsystem, das Multiprogrammierung erlaubt.
- Prozess, der von der CPU bedient wird, ist im Status **running**.
- Prozesse, die auf die CPU warten, sind im Status **ready**.

Frage: Welcher der lauffähigen Prozesse soll als nächster in den **running**-Status?

- Kriterien aus Sicht der Prozesse:
 - Fairness: kein Prozess soll zu lange auf CPU-Zuteilung warten.
 - Wichtigkeit: Prozesse mit hohen Prioritäten werden bevorzugt abgearbeitet.
- Kriterien aus Sicht der CPU:
 - Maximaler Durchsatz
 - Maximale Auslastung der CPU
 - Minimale mittlere Wartezeit (Zeit, bis ein Prozess gestartet wird)
 - Minimale mittlere Systemzeit (Wartezeit + Bedienzeit)
- Kriterien sind z.T. widersprüchlich.
- Im Allgemeinen wird eine Mischung dieser Kriterien gewählt.

5.1 Scheduling-Strategien

Entsprechend der gegebenen Kriterien wird eine Scheduling-Strategie ausgewählt.

Unterschiedliche Arten von Scheduling-Strategien:

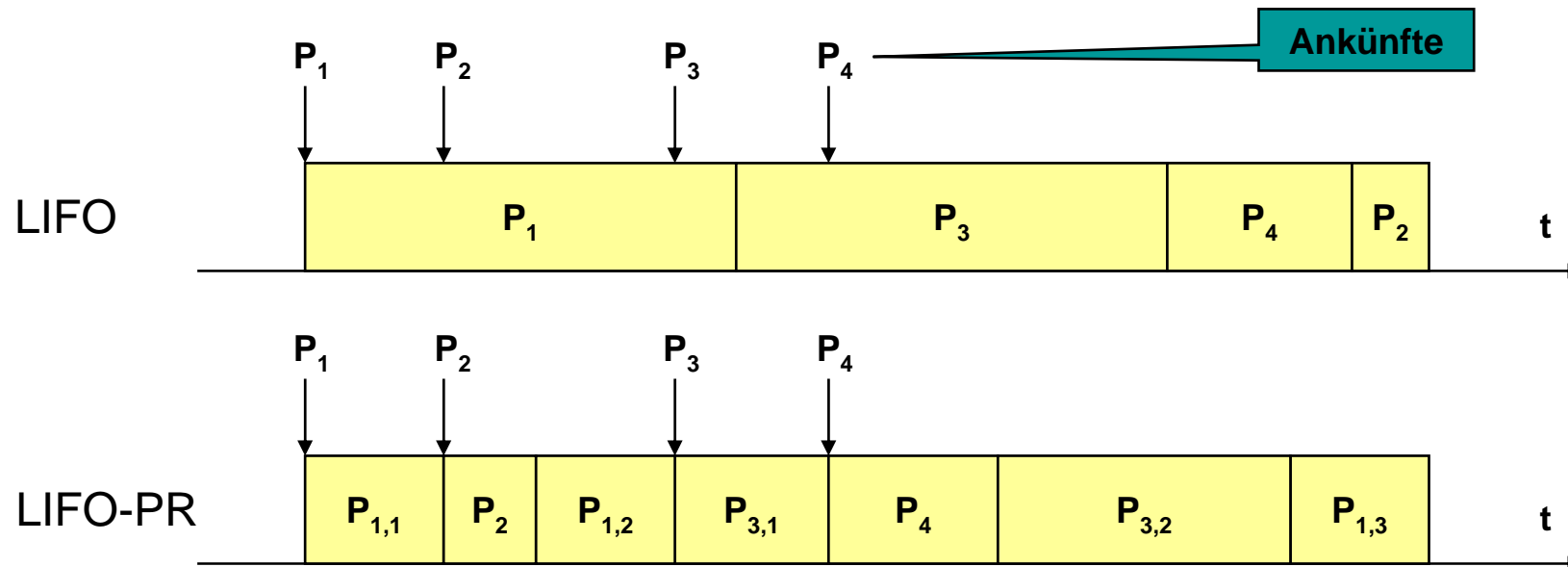
- **Work-conserving** und **nicht work-conserving**: Eine Strategie heißt work-conserving, falls die CPU immer bedient, wenn sie kann, und das Umschalten zwischen Prozessen nur eine vernachlässigbar geringe Zeit bedarf. Ansonsten ist die Strategie nicht work-conserving. Die work-conserving-Annahme ist oft problematisch!
- Auswahl abhängig oder unabhängig von den Laufzeiten der Prozesse
- **Preemptive** und **non-preemptive**: Eine Strategie ist non-preemptive, wenn die einmal begonnene Abarbeitung eines Prozesses bis zu seiner Terminierung nicht mehr unterbrochen wird. Strategien, die eine Unterbrechung eines Jobs zugunsten anderer Jobs erlauben, nennt man preemptive.

5.2 FIFO / LIFO

Scheduling-Strategie	First-In-First-Out (FIFO) auch First-Come-First-Served (FCFS)	Last-In-First-Out (LIFO)
Scheduling	- Prozess, der als erster CPU-Zeit angefordert hat, d.h. der am längsten wartet, erhält CPU-Zugriff	- Prozess, der als letzter CPU-Zeit angefordert hat, erhält CPU-Zugriff
Vorteile	- fair - einfachste Strategie, Realisierung mittels FIFO-Warteschlange - non-preemptive	- ähnlich einfache Strategie - non-preemptive
Nachteile	- Langzeitjobs werden bevorzugt (blockieren andere) - oft schlechte mittlere Wartezeit, Antwortzeit, Anzahl wartender Kunden etc.	- Verhalten der non-preemptive Variante wie bei FIFO - mittlere Wartezeiten, Antwortzeiten etc. gleich - manche Prozesse warten sehr lange, manche nur kurz

5.2 LIFO / LIFO-PR

Es gibt auch eine preemptive Variante LIFO-PR (Preemptive-Resume):



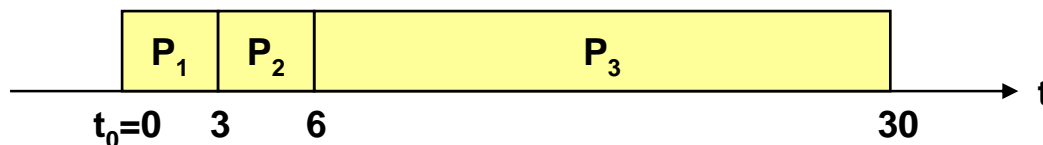
LIFO-Preemptive-Repeat: Unterbrochener Prozess wird nochmals völlig neu gestartet. → Strategie ist nicht mehr work-conserving.

5.2 FIFO / LIFO / Beispiel

Gegeben seien drei Prozesse mit den Rechenzeiten $P_1=3$, $P_2=3$, $P_3=24$.

Graphische Veranschaulichung von Schedules mittels Gantt-Charts.

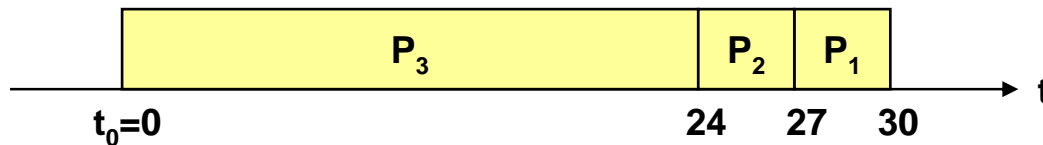
FIFO:



Mittlere Wartezeiten

$$\bar{t}_{FIFO} = \frac{0+3+6}{3} = \frac{9}{3} = 3$$

LIFO (non-preemptive):



$$\bar{t}_{LIFO} = \frac{0+24+27}{3} = \frac{51}{3} = 17$$

- Beispiel lässt vermuten, dass mittlere Wartezeiten unterschiedlich sind.
- Aber: Die mittlere Wartezeit ist für alle non-preemptive Strategien, bei denen die Auswahl des nächsten Jobs nicht von seiner Bediendauer abhängt, gleich groß (über alle Schedules). Mittlere Kundenzahl im System ist ebenfalls gleich.

5.2 Mittlere Wartezeit und Varianz

Unterschiede bei den Scheduling-Strategien bestehen hingegen bei den höheren Momenten. Die Varianz (Streuung) ist ein solches Kriterium. Beispiel:

$$V_{FIFO} = \frac{1}{3} \left((0-3)^2 + (3-3)^2 + (6-3)^2 \right) = 6 \quad \sigma_{FIFO} = \sqrt{6}$$

$$V_{LIFO} = \frac{1}{3} \left((0-17)^2 + (24-17)^2 + (27-17)^2 \right) = 146 \quad \sigma_{LIFO} = \sqrt{146}$$

Allgemein:

- die Varianz der Wartezeiten ist bei LIFO größer als bei FIFO
- Grund: bei LIFO werden einige Jobs sehr schnell abgearbeitet; andere hingegen werden sehr lange verzögert (Stack-Prinzip).

Beispiel zur Bestätigung:

- Ethernet (Lokales Netz): Gemeinsames Medium, das mit „**senden auf Verdacht**“ arbeitet und bei einem Konflikt den Vorgang abbricht und wiederholt.
- Wiederholung nach gesteuerter Wartezeit: je mehr Konflikte, desto längeres Warten bis Neuversuch ➔ LIFO-Effekt.

5.2 Bedienzeitabhängige Strategien

Hauptproblem FIFO/LIFO ist die Benachteiligung kurzlaufender Jobs durch Langläufer

- Beispiel aus dem täglichen Leben: analoges Problem an Supermarktkassen
- Lösung hier: Schnellkassen für Kunden mit maximal $10 \pm \varepsilon$ Artikeln

Entsprechend beim CPU-Scheduling: **bedienzeitabhängige Strategien**

- **Shortest-Processing-Time-First (SPT)**, auch **Shortest-Job-First (SJF)**
- Voraussetzung: Dauer der Jobs ist vorhersehbar.

Man kann zeigen, dass die mittlere Kundenzahl bzw. mittlere Wartezeit sowie die mittlere Systemzeit von SPT für die Klasse der non-preemptive Strategien minimal ist, d.h. SPT ist die optimale Strategie bzgl. der Wartezeit. Die Kundenzahl und die Systemzeit hängen voneinander ab:

$$\bar{N} = \lambda \cdot \bar{S}$$

mittlere Zahl von (wartenden) Prozessen im System = Ankunftsrate · mittlere Systemzeit (Wartezeit)

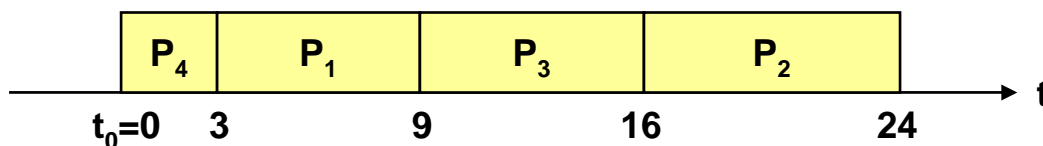
Little's Result

5.3 Vergleich bedienzeitabhängige/-unabhängige Strategien

Beispiel für SPT-Minimalität:

- vier Prozesse mit den Dauern $P_1 = 6$, $P_2 = 8$, $P_3 = 7$, $P_4 = 3$
- SPT weist geringste mittlere Wartezeit auf
- Ankünfte unmittelbar hintereinander in der Reihenfolge P_1, P_2, P_3, P_4 mit Ankunftszeitpunkt $t = 0$.

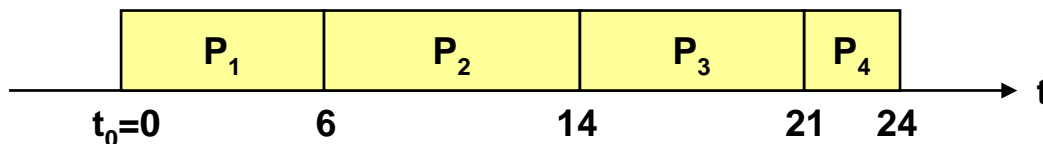
SPT



Mittlere Wartezeiten

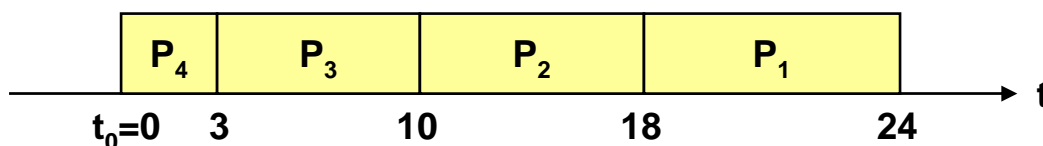
$$\bar{t}_{spt} = \frac{3+9+16}{4} = \frac{28}{4} = 7$$

FIFO



$$\bar{t}_{FIFO} = \frac{6+14+21}{4} = \frac{41}{4} = 10.25$$

LIFO

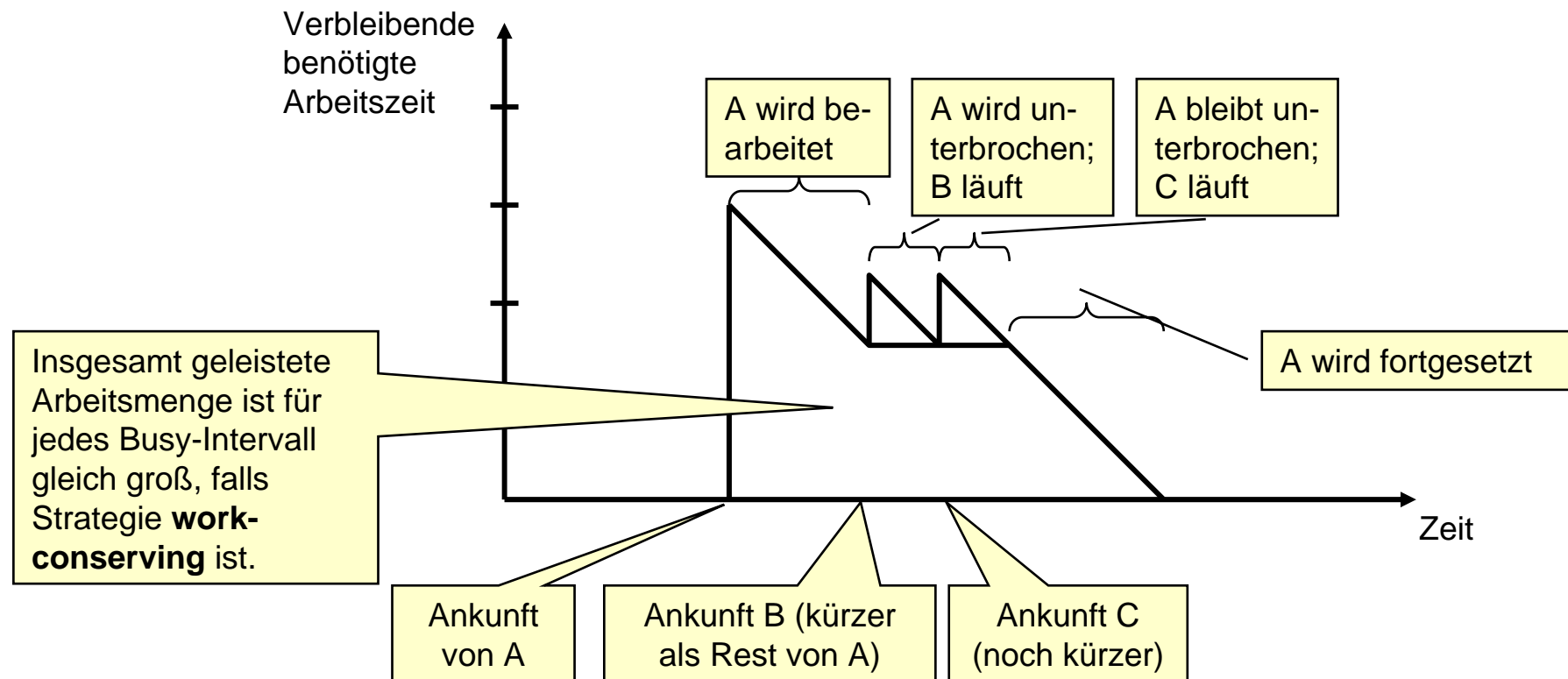


$$\bar{t}_{LIFO} = \frac{3+10+18}{4} = \frac{31}{4} = 7.75$$

5.3 SRPT

Verbesserung von SPT ist **Shortest-Remaining-Time-First (SRPT)**

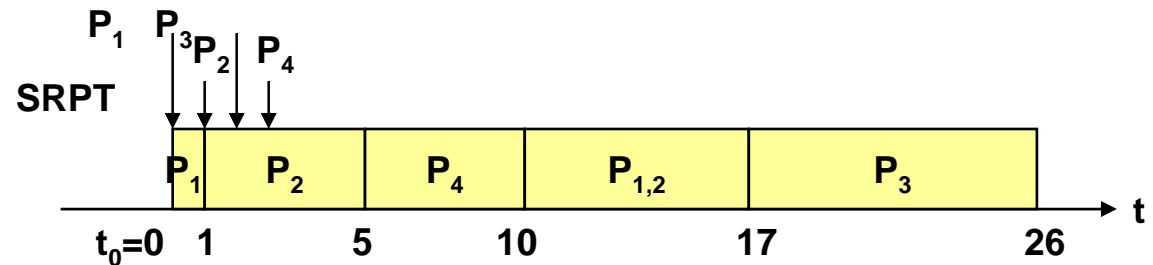
➤ SRPT ist wie SPT, aber preemptive, d.h. ein Job wird unterbrochen, sobald ein weiterer ankommt, dessen Bearbeitungszeit kürzer als die Restzeit des momentan aktiven Jobs ist.



5.3 SRPT / Beispiel

Vier Prozesse mit folgenden Ankunfts- und Bedienzeiten:

	Ankunftszeit	Bedienzeit
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Mittlere Wartezeit:

$$\bar{t}_{SRPT} = \frac{9+0+15+2}{4} = \frac{26}{4} = 6.5$$

Es lässt sich zeigen:

- SRPT ist optimal bzgl. der mittleren Wartezeit unter allen Strategien, die work-conserving sind, d.h. vernachlässigbare Umschaltzeiten aufweisen.

Grundlegende Probleme mit SRPT:

- Kosten für Unterbrechungen werden nicht eingerechnet.
- Woher weiß man a-priori, wie lange ein Job dauern wird?

5.3 SEPT / SERPT

Mögliche Ansätze, um a-priori Wissen über Jobdauer zu erhalten:

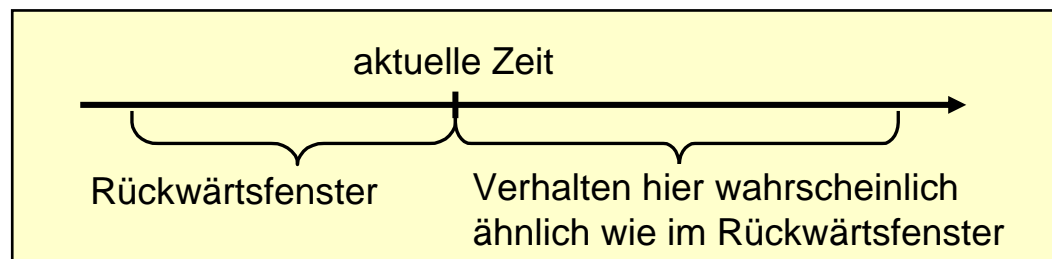
Prozessdauer durch den Nutzer angeben

- Problematisch: Ist die Zeit zu kurz, wird ein anderer Job abgewürgt; ist die Zeit zu lang, beginnt die Abarbeitung des Jobs unnötig spät.

Erwartete Prozessdauer aus Erfahrung der Vergangenheit schätzen

- Jobs werden von verschiedenen Kundenquellen kreiert
- Aus der Dauer der Jobs in der Vergangenheit auf das Verhalten in der Zukunft schließen.
- SEPT: Shortest-Expected-Processing-Time-First
- SERPT: Shortest-Expected-Remaining-Processing-Time-First
- Einfache Schätzverfahren verwenden Zeitfenster, z.B. den Durchschnitt der letzten n Jobs oder 80% des Maximalwerts der letzten n Jobs

Zeitliches Lokalitätsverhalten:



5.3 Exponential-Averaging

Exponential-Averaging versucht, adaptiv aus der Vergangenheit zu **lernen**:

Sei τ_n die Schätzung für den n -ten Job und t_n seine tatsächliche Dauer.

Dann erhält man die Schätzung für den $(n+1)$ -ten Job durch

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$$

α liegt in $[0,1]$ und beeinflusst die Art des Lernens.

Extremfälle:

➤ $\alpha = 0 \rightarrow \tau_{n+1} = \tau_n = \dots = \tau_0$, d.h. die Schätzung ist immer gleich. Es findet kein Lernen statt.

➤ $\alpha = 1 \rightarrow \tau_{n+1} = t_n$, d.h. nur der letzte Wert wird für die Schätzung herangezogen, was einem hektischen Verhalten entspricht.

Kompromiss: Verwendung von $0 < \alpha < 1$, wodurch ein **exponentielles Abklingen** der Vergangenheit erreicht wird.

5.3 Exponential-Averaging

Besonderheit: **Exponentielles Abklingen** der Vergangenheit ($0 < \alpha < 1$). Dies wird durch explizites Ausschreiben der Rekursion deutlich:

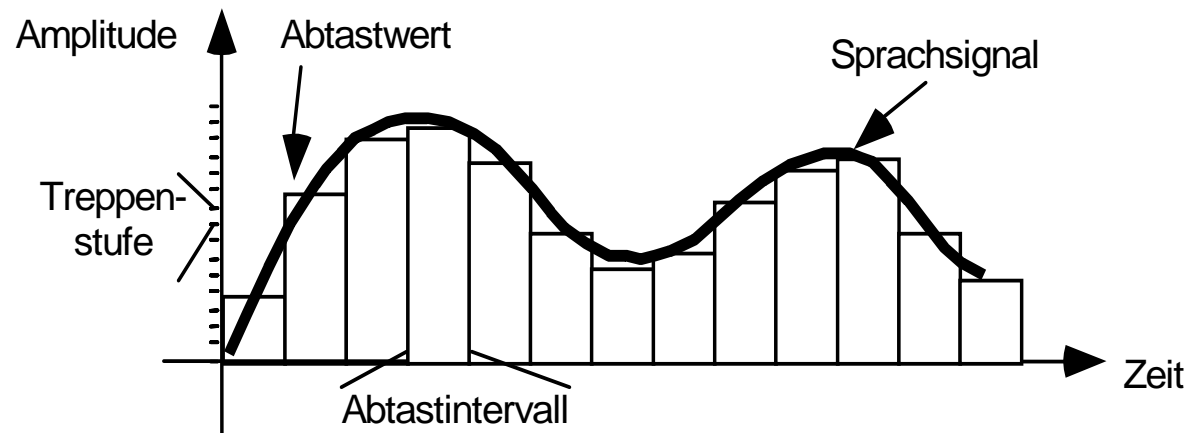
$$\begin{aligned}\tau_{n+1} &= \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n \\ &= \alpha \cdot t_n + (1 - \alpha) \cdot (\alpha \cdot t_{n-1} + (1 - \alpha) \cdot \tau_{n-1}) \\ &= \alpha \cdot t_n + (1 - \alpha) \cdot \alpha \cdot t_{n-1} + (1 - \alpha)^2 \cdot \tau_{n-1} \\ &= \dots \\ &= \alpha \cdot t_n + (1 - \alpha) \cdot \alpha \cdot t_{n-1} + \dots + (1 - \alpha)^n \cdot t_0 + (1 - \alpha)^{n+1} \cdot \tau_0\end{aligned}$$

Je größer α , desto schneller wird die Vergangenheit **vergessen**:

- α zu groß ➔ Starke Auswirkung kurzfristiger Schwankungen, kaum Glättung.
- α zu klein ➔ Korrektur bei Trendveränderungen möglicherweise zu langsam.

5.3 Exkurs: Digitale Sprachübertragung

Exponential-Averaging: Kurve folgt der tatsächlich von Prozessen benötigten CPU-Zeit. Vergleich: Folgen der Frequenzkurve mit Abtastwerten beim Digitalisieren von Sprachsignalen



Wie oft muss ein Signal abgetastet werden?

Nyquist-Theorem:

Wenn Signalwert exakt digitalisiert werden kann, lässt sich die Originalkurve aus den digitalen Signalen genau dann wiederherstellen, wenn die **Abtastrate** (Abtastfrequenz) mindestens doppelt so hoch ist wie die maximale im Signal vorkommende Frequenz.

5.3 Exkurs: Digitale Sprachübertragung

Beispiel für Nyquist-Theorem: **Telefon**

- (Audio-)Signalfrequenz üblicherweise zwischen 300 und 3400 Hertz
 - ➔ Abtastfrequenz von ca. 6,8 kHz ausreichend (bei exakter Abtastung)

Pulse-Code-Modulation (PCM):

- bekanntes Verfahren zur Sprachkodierung
- Abtastfrequenz 8 kHz
- kodiert jeden Abtastwert mit 8 Bit. Dies ist nicht exakt, aber für das menschliche Ohr ausreichend.
 - ➔ alle 125µs 8 Bit übertragen, d.h. 64 kBit/s

ISDN-B-Kanal hat genau 64 kBit/s Kapazität

Ziel für Verbindungen mit geringerem Durchsatz: Datenrate senken

- Abtastintervalle vergrößern ➔ Qualitätsverlust
- Delta-Modulation

5.3 Exkurs: Digitale Sprachübertragung

Delta-Modulation: Anstelle der einzelnen Abtastwerte werden ein Startwert und danach jeweils die Änderungen (Delta) übertragen (vgl. Videokompressions-Verfahren, z.B. MPEG).

Signalverlauf wird somit beim Sender und Empfänger nachgeführt. Dabei gibt es notwendige Festlegungen:

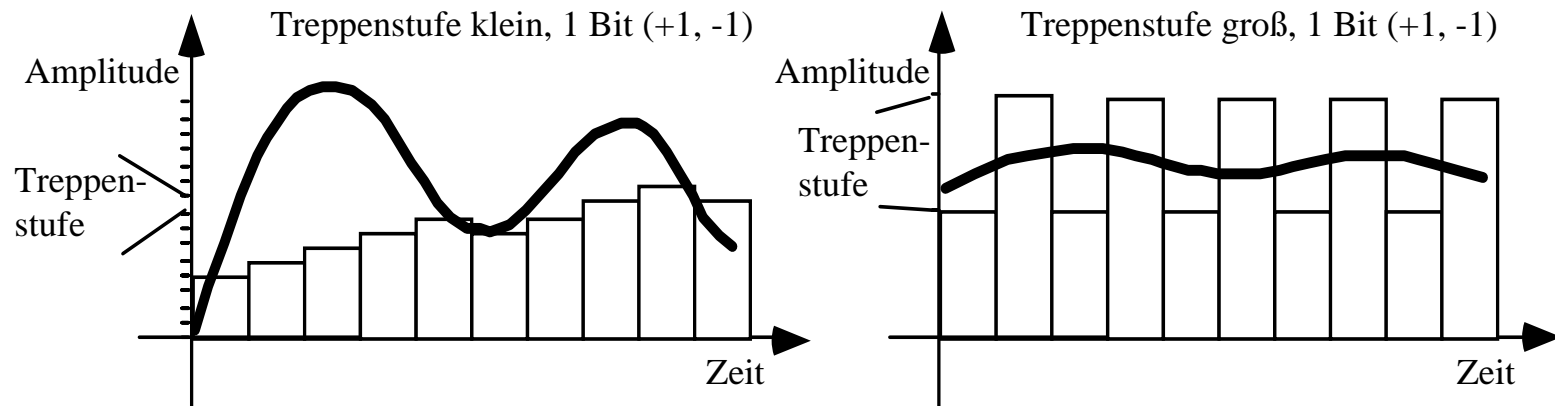
- Größe der Treppenstufe
- Anzahl der Bits, mit denen Delta kodiert wird, d.h. wie viele Treppenstufen auf einmal genommen werden können.

Beispiel: 1-Bit-Delta

- Übertrage 0 (alter Wert + Delta) oder 1 (alter Wert - Delta).
- Kleines Delta: kleine Amplitudenunterschiede werden erkannt und nachgebildet. Aber: schlechte Reaktion bei großen Schwankungen (vgl. kleines α beim Exponential-Averaging).
- Großes Delta führt zu einem umgekehrten Verhalten.

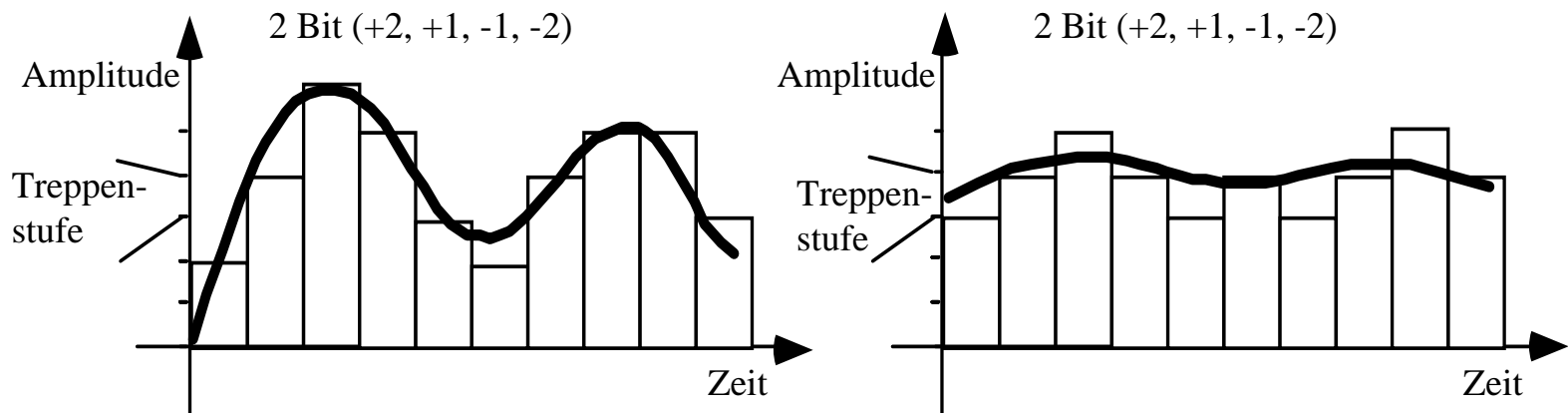
5.3 Exkurs: Digitale Sprachübertragung

1-Bit-Delta:



Offensichtlich ist 1-Bit-Delta oft nicht in der Lage, dem Kurvenverlauf zu folgen. Besser 2-, 3- oder 4-Bit-Delta auswählen, wodurch aber die Datenrate erhöht wird.

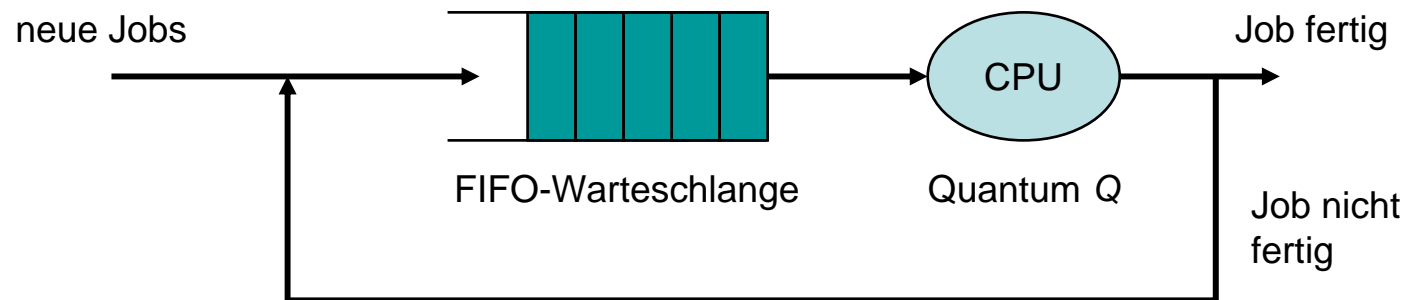
2-Bit-Delta:



5.4 Round-Robin-Verfahren

Weitere Scheduling-Variante versucht, Fairness gegenüber Kurzläufnern und soweit möglich auch gegenüber Langläufern zu gewähren: **Round-Robin (RR)**.

Ziel: Gesamtzeit des Jobs soll proportional zu seiner Bediendauer sein. Round-Robin verwendet Approximation:

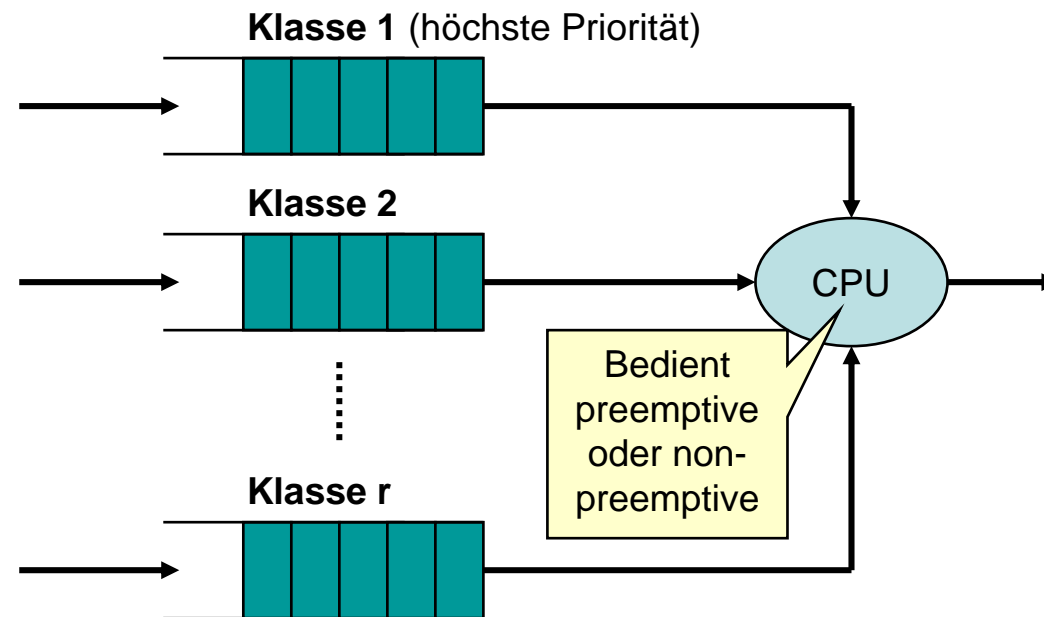


- $Q \rightarrow \infty$: FIFO
- $Q \rightarrow 0$: Processor-Sharing (PS), d.h. CPU wird gleichmäßig auf die n z. Zt. aktiven Prozesse aufgeteilt, wodurch jeder Prozess genau $1/n$ der CPU-Leistung erhält.
- sehr problematische Annahme, dass **Zerhacken** in kleinste Scheiben nichts kostet.

5.4 Priority-Scheduling

Jedem Job wird eine Priorität 1, ..., r zugeordnet. Abarbeitung nach Schema **Highest-Priority-First (HPF)**. Ein Job aus Klasse k wird bedient, falls:

- die CPU frei ist bzw. bei preemptive: CPU bedient Job aus Klasse $> k$,
- der Job der **älteste** Job der Klasse k ist und
- alle Klassen 1 bis $k-1$ momentan unbesetzt sind.



5.4 Multilevel-Feedback-Queueing

Verfahren mit Kombination aus Prioritätsklassen und Zeitscheiben: **Multilevel-Feedback-Queueing**

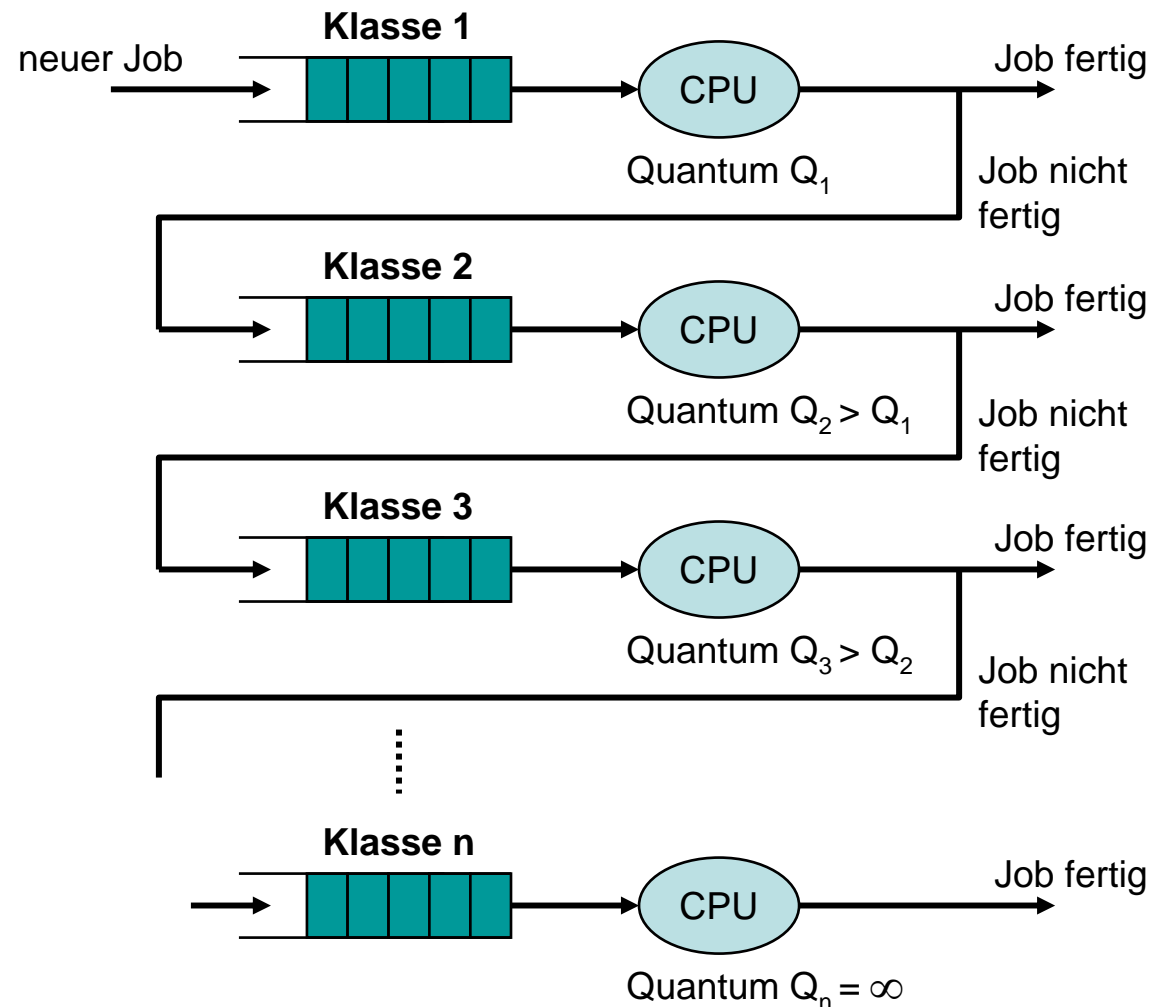
Idee:

- Neu ankommende Jobs in höchster Prioritätsklasse.
- Abarbeitung nach Round-Robin: unterschiedliche Quantengröße, Klassen hoher Priorität haben kleine Quanten, bei kleiner Priorität große Quanten, d.h. $Q_1 < Q_2 < \dots < Q_n$.
- Wenn Job in seiner Zeitscheibe nicht fertig wird, wandert er in die nächst niedrigere Prioritätsklasse.

Bedient wird non-preemptive der älteste Job der höchsten besetzten Klasse.

➔ Langläufer können hier sehr benachteiligt werden.

5.4 Multilevel-Feedback-Queueing



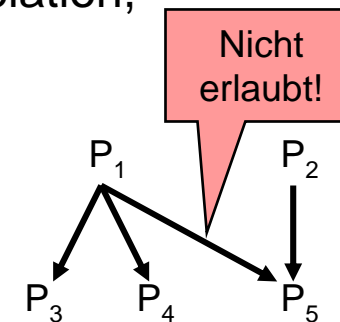
5.5 Mehrprozessorsysteme

Bisher: Scheduling mehrerer Jobs auf einen Prozessor

Jetzt: Spezielle Schedulingprobleme bei Mehrprozessorsystemen

Vereinfachende Annahmen:

- m identische Prozessoren
- im Voraus bekannte Laufzeiten (deterministic scheduling problem)
- Prozesse P_1, \dots, P_n evtl. mit Vorgänger-Nachfolger-Relation, d.h. $P_i \rightarrow P_j$ bedeutet P_i vor P_j
- die Relationen sollten nicht komplizierter sein als ein **Wald** (d.h. Menge von Bäumen).



Bewertungskriterium:

Ein Schedule, d.h. eine spezielle Aufteilung von P_1, \dots, P_n auf m Prozessoren, sei besser als ein anderer, wenn die Dauer bis zur Terminierung des letzten Prozesses geringer ist.

5.5 Mehrprozessorsysteme

Nahe liegende Idee:

- Jeden Prozessor besetzen, sobald er frei wird und ein Prozess bedienbar ist.
- **Longest-Processing-Time-First (LPT)**, d.h. zunächst Langläufer abarbeiten.

Leider gibt es viele Gegenbeispiele.

Außerdem gibt es überraschende Anomalien. So kann z.B. die Dauer steigen, wenn

- mehr Prozessoren eingesetzt werden
- die Ausführungszeit pro Prozess sinkt
- weniger Freizeit pro Prozessor vorhanden ist
- weniger Vorgänger-Nachfolgerrelationen gegeben sind

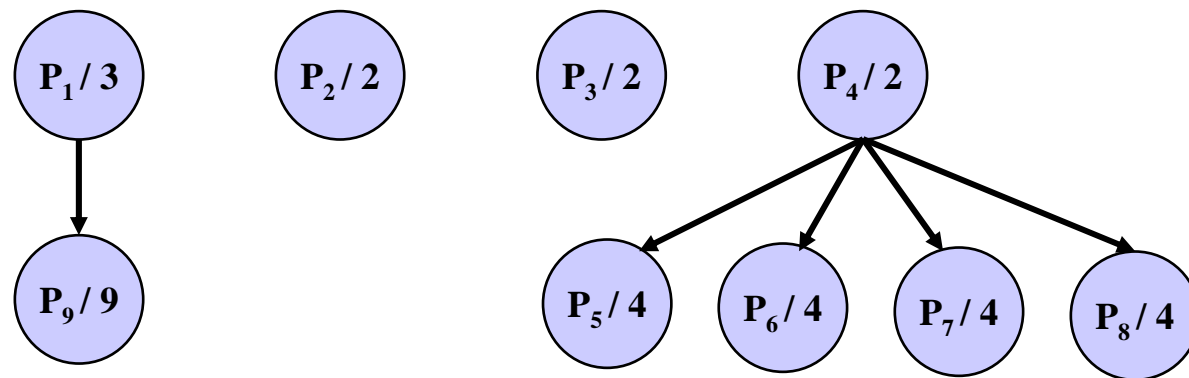
All dies erscheint widersinnig!

Nachfolgend werden einige dieser Anomalien diskutiert.

5.5 Mehrprozessorsysteme / Beispiel

Betrachtet wird folgender Fall:

- Gegeben seien neun Prozesse P_1, \dots, P_9 mit folgenden Abhängigkeiten.
- P_i / k bedeutet: Prozess i benötigt k Zeiteinheiten.

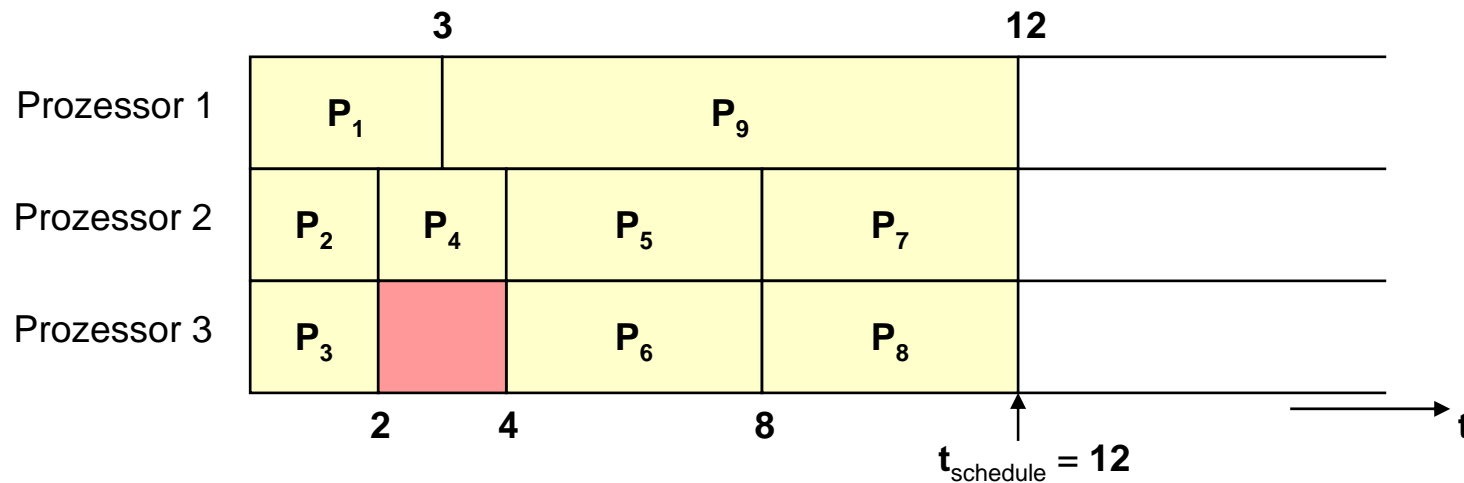


Strategie:

- Bediene nach aufsteigender Listennummer, d.h. Liste $[1,2,\dots,9]$, und zwar jeweils den frühesten bedienbaren Prozess.
- Lege diesen Prozess auf Prozessor 1, 2, 3, ..., m , 1, 2, ... (d.h. zyklisch).

5.5 Mehrprozessorsysteme / Beispiel

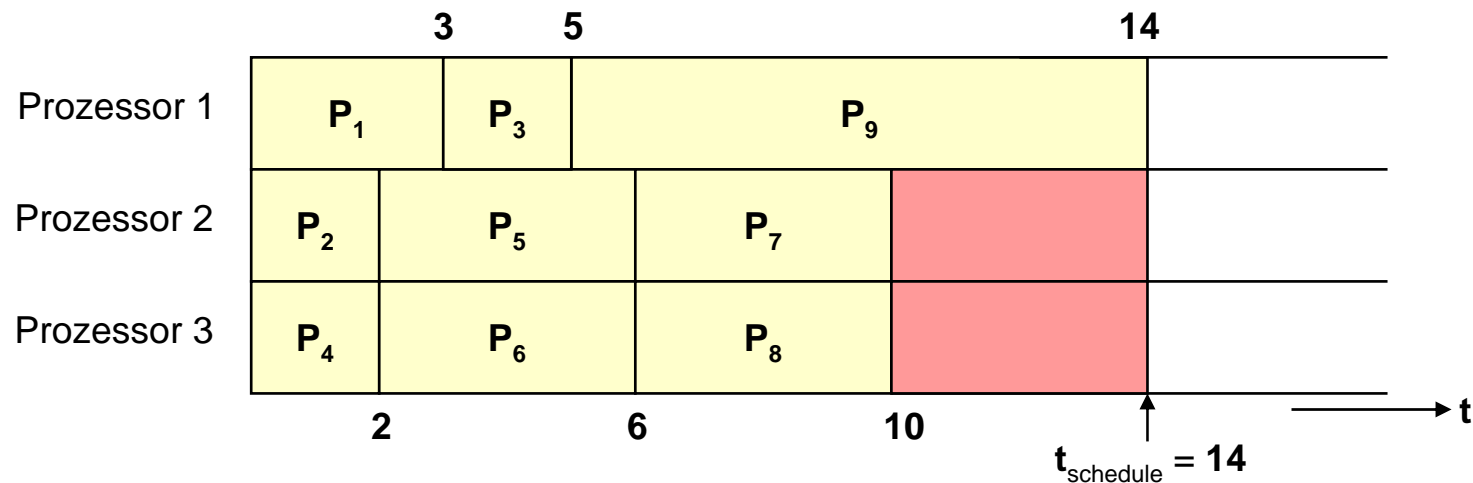
Somit ergibt sich für $m = 3$ Prozessoren:



5.5 Mehrprozessorsysteme / Anomalien

Vermutung: Bedienung so schnell wie möglich ist bestmögliche Strategie.

Anomalie 1: Gegeben sei nun eine neue Liste [1,2,4,5,6,3,9,7,8]. Somit ergibt sich:

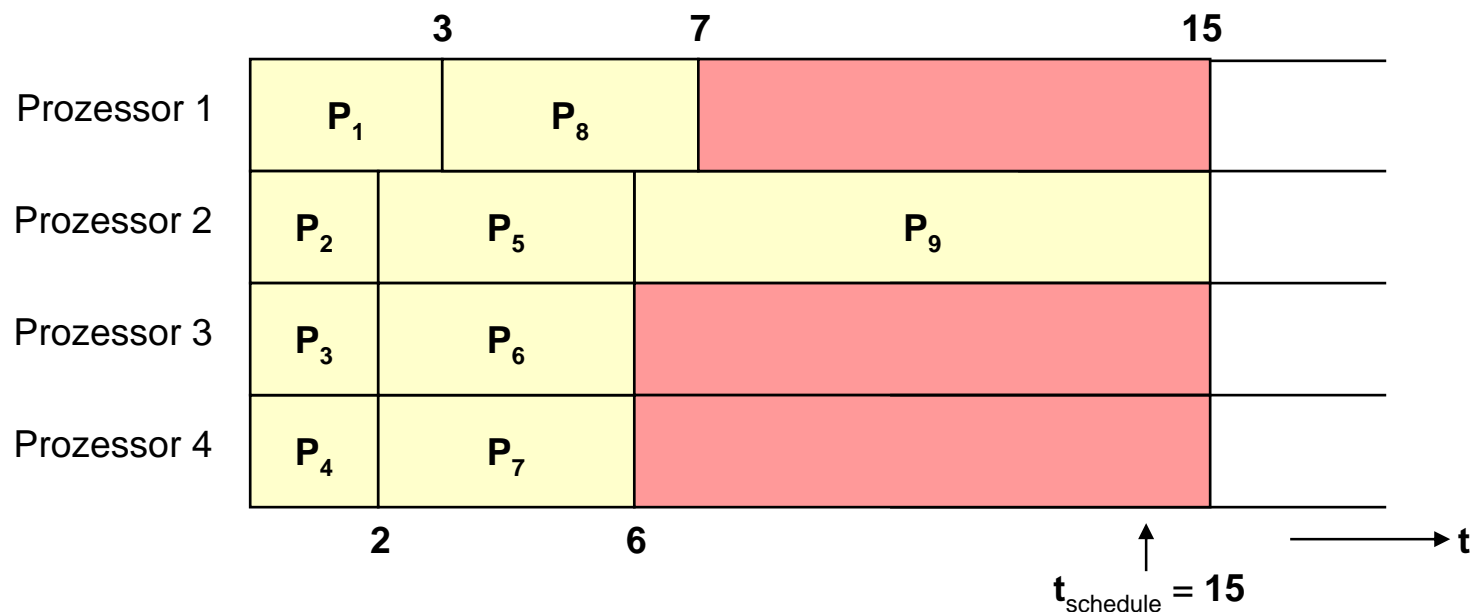


Somit schlechter, obwohl die Füllung zu Beginn besser ist!

5.5 Mehrprozessorsysteme / Anomalien

Vermutung: mehr Prozessoren → kürzerer Schedule. Falsch!

Anomalie 2: Sei $m' = 4$, d.h. ein Prozessor mehr.

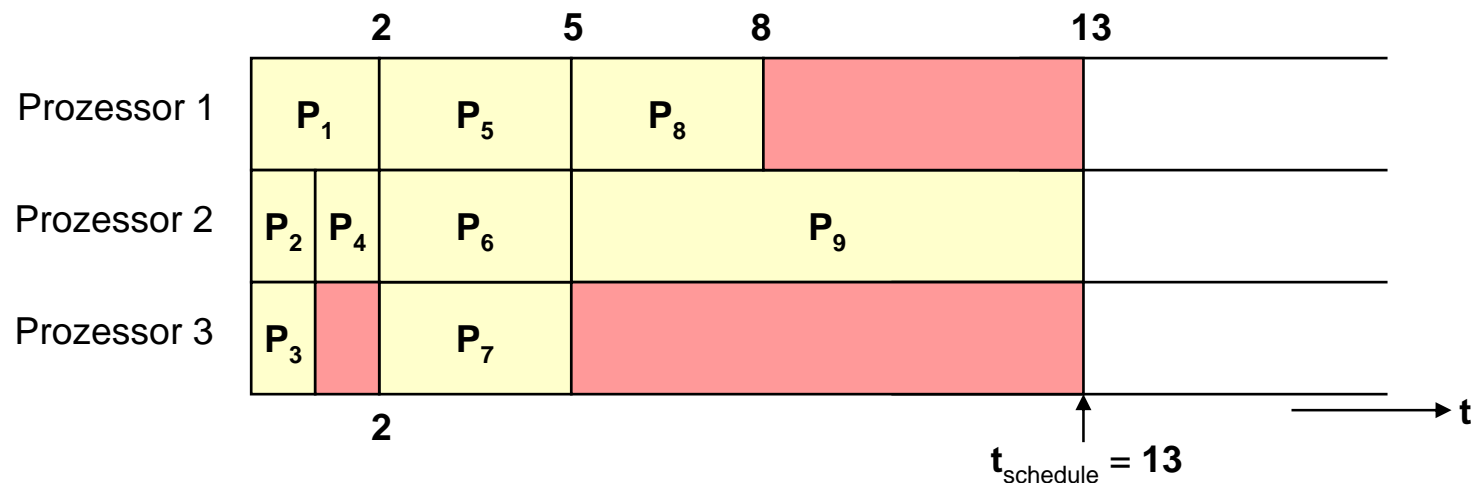


Somit trotz höherer Prozessorzahl längere Ausführungszeit.

5.5 Mehrprozessorsysteme / Anomalien

Vermutung: Kürzere Ausführungszeiten → kürzerer Schedule. Falsch!

Anomalie 3: Im Folgenden seien die Ausführungszeiten aller Prozesse jeweils um 1 ZE kürzer. Die Liste sei wie anfangs [1,2,...,9].

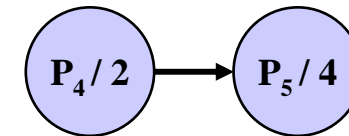


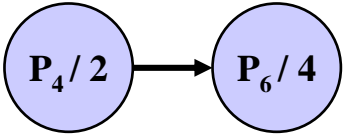
5.5 Mehrprozessorsysteme / Anomalien

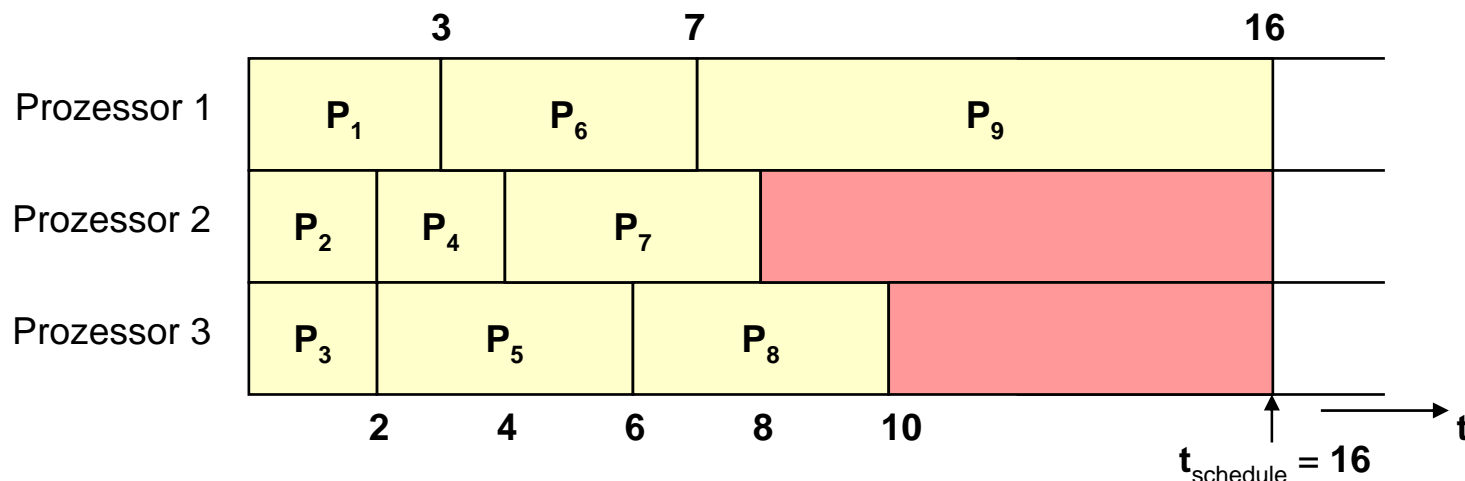
Vermutung: Weniger Abhängigkeiten → einfachere Platzierung → kürzerer Schedule ist Falsch!

Anomalie 4: Wie anfangs, allerdings seien weniger Vorgänger-Nachfolgerrelationen gegeben.

Nehme hierfür aus bisherigem Wald die Relationen



und  heraus.



5.5 Klarstellung

Zur Klarstellung:

- Solche Anomalien sind nicht die Regel, sondern Ausnahmen!
- Man kann sie z.B. dadurch verhindern, dass man einen Prozessor leer lässt oder künstliche Leerphasen zulässt.
- Wichtige Frage bei Echtzeitsystemen, wenn man ein optimales Schedule nicht sinnvoll ermitteln kann. Der Grund ist, dass diese Berechnung sehr schnell **NP-hart** werden kann.
- Kann man eine sichere untere Schranke angeben? Wie lange braucht man mindestens?
- Kann man Schranken dafür angeben, um wie viel schneller ein aktuelles Schedule im **worst-case** sein kann?

Dies ist oft möglich! Merkwürdigerweise sind die meisten bekannten Schranken von folgender Art:

$$\frac{\text{Laufzeit aktuell}}{\text{Laufzeit optimal}} \leq 2 - \frac{1}{m}$$

wobei m = Zahl der Prozessoren

Inhaltsverzeichnis

6.1 Hauptspeicherverwaltung

- Speicherhierarchie
- Virtueller Speicher

6.2 Segmentierung

- Segmentierungsstrategien
- Vergleich der Strategien

6.3 Buddy-Systeme

- Adressierung
- Gewichtete Buddy-Systeme

6.4 Demand-Paging

- Paging
- Paging-Strategien

6.5 Nicht-Demand-Paging

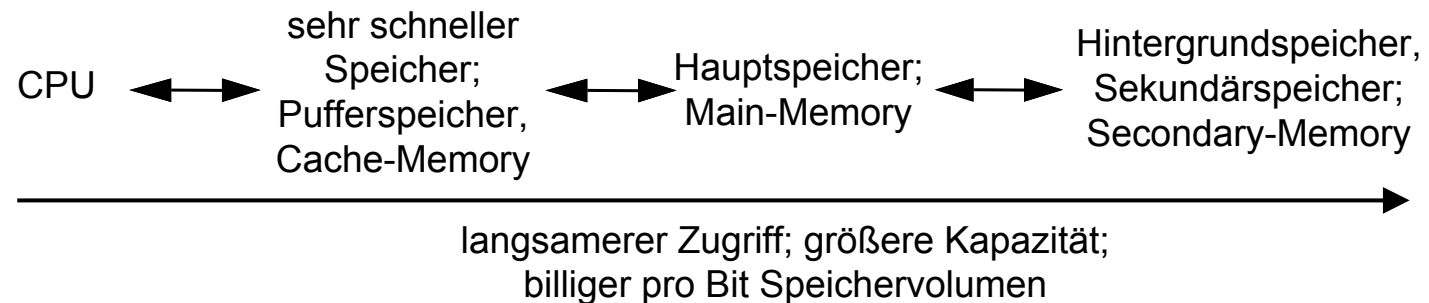
- One-Block-Look-Ahead

6.6 Diskussion der Paging-Algorithmen

- Kosten von Paging-Algorithmen
- FIFO-Anomalie
- Stack-Algorithmen
- Prioritätsalgorithmen

6.1 Hauptspeicherverwaltung / Speicherhierarchie

Der Speicher ist hierarchisch organisiert:



Probleme:

- Wie wird der jeweilige Speicher organisiert?
 - ➔Paging, Segmentierung, ...
- Was soll im jeweils kleineren Speicher stehen?
 - ➔aktuelle Daten, Lokalitätsmodelle, was wird bei Bedarf verdrängt?

Ziel:

- Verdränge diejenigen Daten, die **wahrscheinlich** in näherer Zukunft nicht gebraucht werden.

6.1 Hauptspeicherverwaltung / Virtueller Speicher

Die Systembenutzer sollen nicht merken, dass es mehrere Speicherstufen gibt.

→ virtueller Speicher

Es ist virtuell mehr Speicher vorhanden als reell.

Bestandteile des virtuellen Speichers:

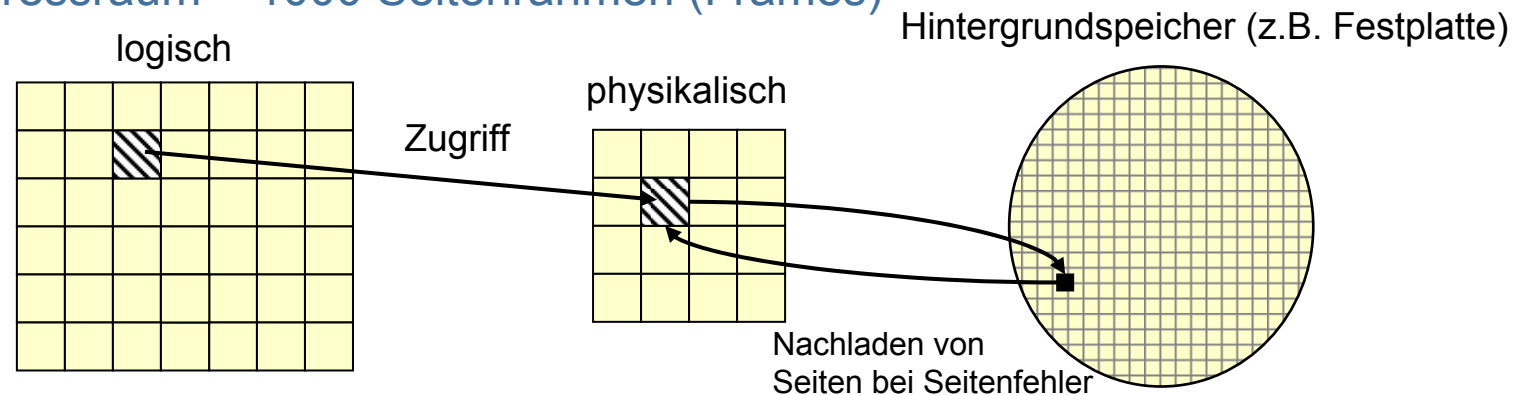
- logischer Adressraum:
Adressraum eines Programms → beliebige Größe!
- physikalischer Adressraum:
Tatsächlich (physikalisch) zur Verfügung stehende Speicherzellen im Hauptspeicher.
- Tauschmechanismus:
Strategie zum Ein- bzw. Ausladen von Speicherteilen.

6.1 Hauptspeicherverwaltung / Virtueller Speicher

Beispiel: Speicherorganisation in **Seiten**, d.h. in Blöcken fester Größe.

Annahme: logischer Adressraum = 1 Mio Seiten mit je 1024 Worten

physikalischer Adressraum = 1000 Seitenrahmen (Frames)



Zugriff auf logischen Speicher:

Hit \Rightarrow Seite ist im physikalischen Speicher; Zugriff sehr schnell

Seitenfehler (**Nicht-Hit**) \Rightarrow eine oder mehrere Seiten müssen geladen oder ausgetauscht werden.

6.1 Hauptspeicherverwaltung / Virtueller Speicher

Wie realisiert man den Tauschmechanismus?

- **von Hand**

z. B. Overlay: Der Benutzer erkennt Speicherbedarf und lagert einzelne Teile auf eine Festplatte aus.

- **automatisch**

z. B. durch das Betriebssystem.

Eine Strategie: Ersetze bei Bedarf diejenige Seite, auf die am längsten nicht mehr zurückgegriffen wurde. → Least-Recently-Used (LRU)

Beobachtung: **Von Hand** ist oft schlechter als **automatisch**.

Größe der Tauscheinheiten

- Segmentierung
- Seitenkonzept → Paging
- Kombinationen aus Segmentierung und Paging
- Buddy-Systeme

6.2 Segmentierung

Idee: Logischer Adressraum ist eine Sammlung von **Segmenten**

Ein **Segment** ist eine logische Einheit zusammenhängender Informationen (z.B. Unterprogramm, Programmdatei) und besitzt einen Namen (Nummer) und eine (variable) Länge.

Eine logische Adresse wird durch das Paar (Segment-Nummer, Offset) beschrieben.

Segmente, die für den Ablauf eines Prozesses benötigt werden, müssen in den Hauptspeicher geladen werden. Hierzu wird zunächst eine genügend große Lücke gesucht und das Segment linksbündig platziert.

⇒ Erforderlich: Verwaltung von Lückenlisten mit Anfangsadresse und Größe.

6.2 Segmentierung

Durch Platzierungen und Freigaben entsteht ein Teppich von belegten Bereichen und Lücken.

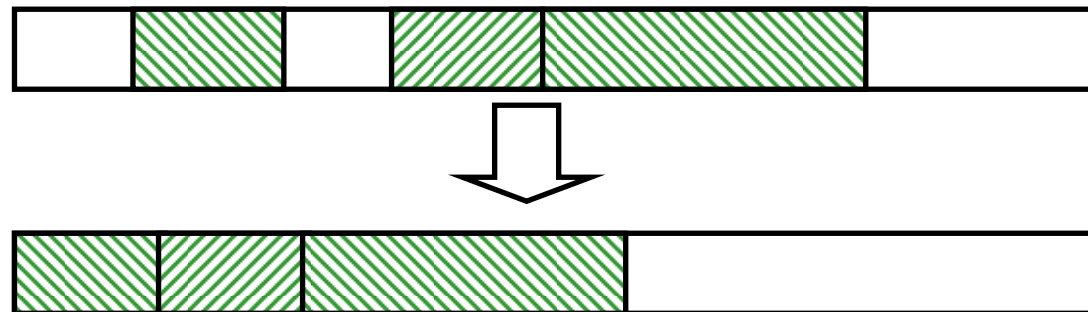


Zweckmäßig: Lücken und belegte Bereiche als verkettete Liste verwalten.

Typisch für Segmentierung: Entstehen von vielen kleinen, praktisch unverwertbaren Lücken.

Weitere Möglichkeiten:

Umordnung, Garbage-Collection



Problem: wann, bzw. wie oft ? Heuristik: ab 80% Belegung.

6.2 Segmentierung / Segmentierungsstrategien

Strategien zur Lückenauswahl

- First-Fit (FF)
Platziere das Segment in die erste passende Lücke
- Best-Fit (BF)
Platziere das Segment in die kleinste passende Lücke
- Worst-Fit (WF)
Platziere das Segment in die größte passende Lücke
- Rotating-First-Fit (RFF)
wie FF, jedoch Suche der neuen Lücke ab der zuletzt gefundenen Lücke

6.2 Segmentierung / Vergleich der Strategien

- Hoher Verschnitt durch die Bildung kleiner Lücken am Anfang des Speichers bei FF.
- Bei BF können die entstehenden kleinen Lücken später nur schlecht genutzt werden.
- Simulationen haben gezeigt: RFF>FF>BF>WF
- Für jede Strategie gibt es besonders schlechte und besonders gute Szenarien.
- Beispiel für die Nichtoptimalität von Best-Fit:

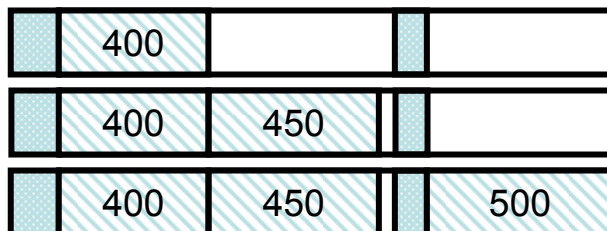
Ausgangssituation



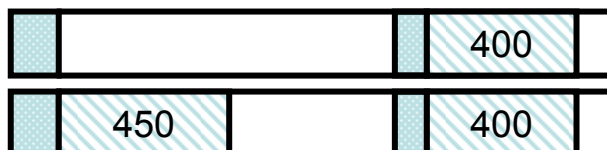
Reihenfolge der Anforderungen



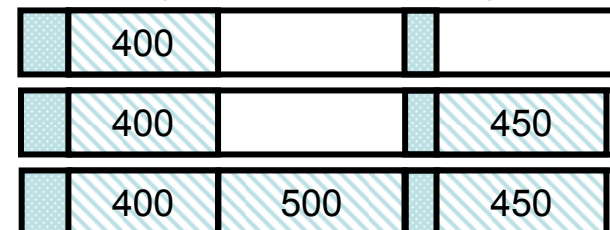
First-Fit



Best-Fit



Rotating-First-Fit, vorne begonnen



Rotating First Fit, hinten begonnen

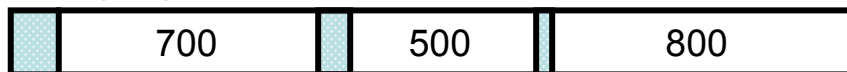


Nur FF und RFF (vorne begonnen) erfüllen die Speicheranforderung

6.2 Segmentierung / Vergleich der Strategien

Beispiel für Nichtoptimalität von First-Fit und Rotating-First-Fit:

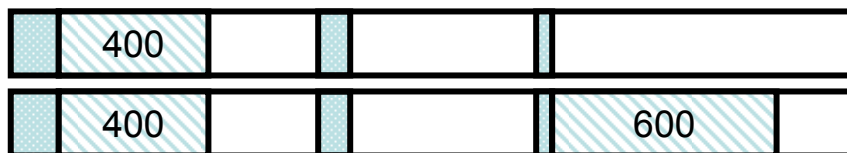
Ausgangssituation



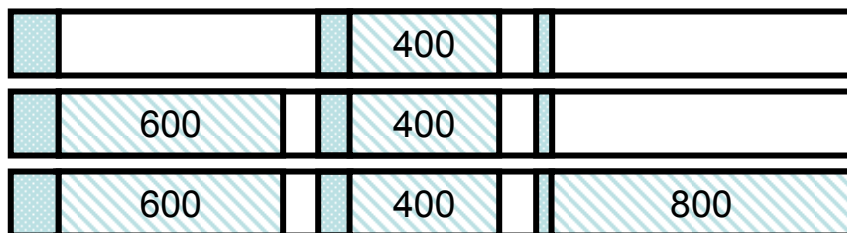
Reihenfolge der Anforderungen



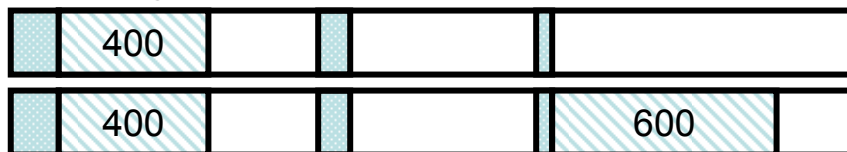
First-Fit



Best-Fit



Rotating-First-Fit



Worst-Fit



Nur BF erfüllt die Speicheranforderung. Nachteil ist der hohe Suchaufwand beim Auffinden der Lücken.
Alternative:

Garbage-Collection, d.h. nachträgliches Umordnen des Speichers.

Allgemein:

Speicherverwaltung bei Segmentierung aufwändig:
Liste von Lücken incl. Größe, ...

6.3 Buddy-Systeme

Buddy-Systeme stellen einen Kompromiss zwischen dem starren Seitenkonzept und dem verschnittreichen Segmentkonzept dar:

- Anforderungen nur in Größe einer 2-er Potenz erlaubt
- Wurzel des Buddy-Baums ist 2^{\max} → Gesamtgröße des physikalischen Speichers
- Führe jeweils eine Liste $L_{\max}, L_{\max-1}, \dots, L_{\min}$, mit
 $L_{\text{pot}} :=$ Liste freier Blöcke der Größe 2^{pot}
→ Zu Beginn ist $L_{\max} \neq \emptyset$, die restlichen Listen sind leer

Einfügen eines Segments der Größe 2^{pot} :

- $L_{\text{pot}} \neq \emptyset$: Platziere Segment und lösche den verwendeten Block aus der Liste
- $L_{\text{pot}} = \emptyset$: Suche freien Block in $L_{\text{pot}+1}, L_{\text{pot}+2}, \dots$ und spalte diesen solange in zwei Hälften (Buddies), bis eine passende Buddy-Größe entsteht

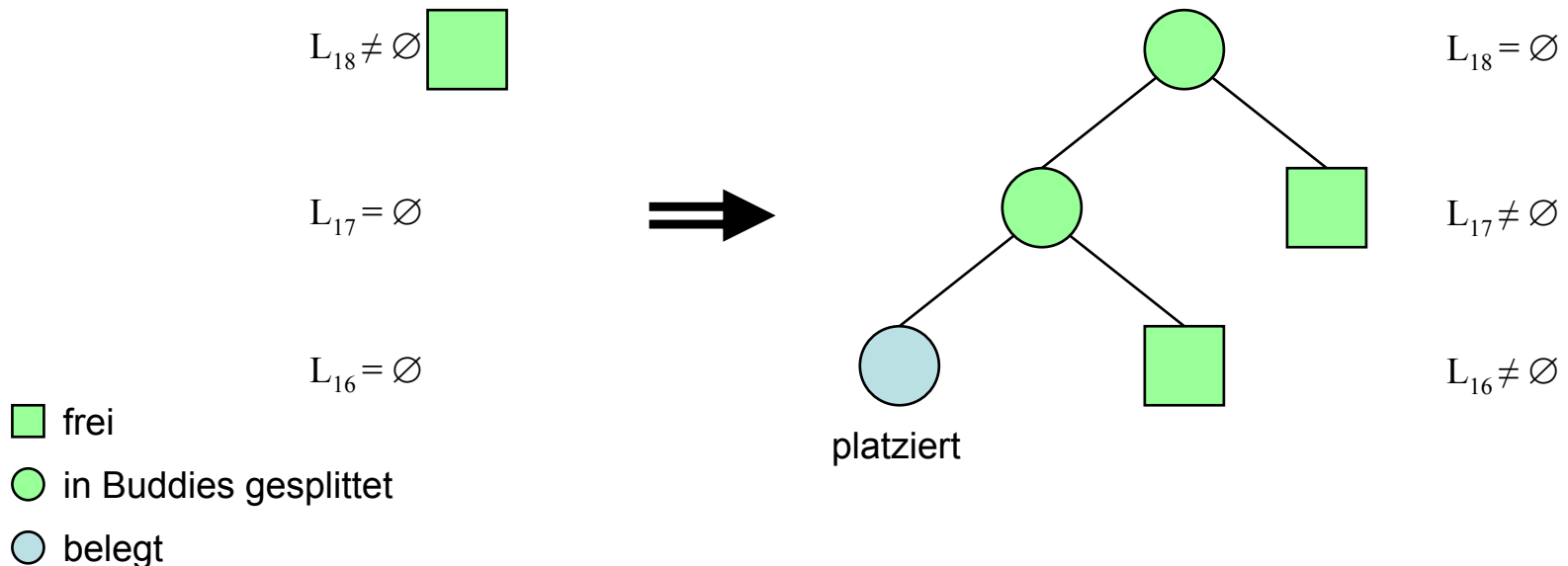
6.3 Buddy-Systeme

Freigabe von Segmenten:

- Prüfe, ob der zugehörige Buddy frei ist; wenn ja: verschmelze beide.
- Wiederhole dies solange, bis keine Buddies mehr vereinigt werden können.

Beispiel:

Anforderung von einem Segment der Größe 2^{16} ; Größe des Speichers: 2^{18}



6.3 Buddy-Systeme / Adressierung

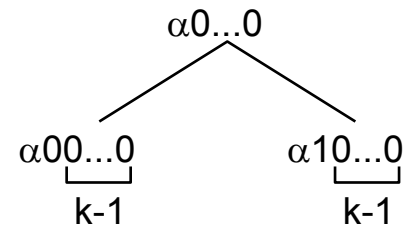
Zu adressierender Speicherbereich: $[0 : 2^{\max}-1] \rightarrow$ binär: $[0\dots0 : 1\dots1]$

Ein Speicherbereich der Größe 2^k ist durch $[\alpha \underbrace{0\dots0}_k ; \alpha \underbrace{1\dots1}_k]$ gekennzeichnet.

Die Vorsilbe α charakterisiert den Speicherbereich.

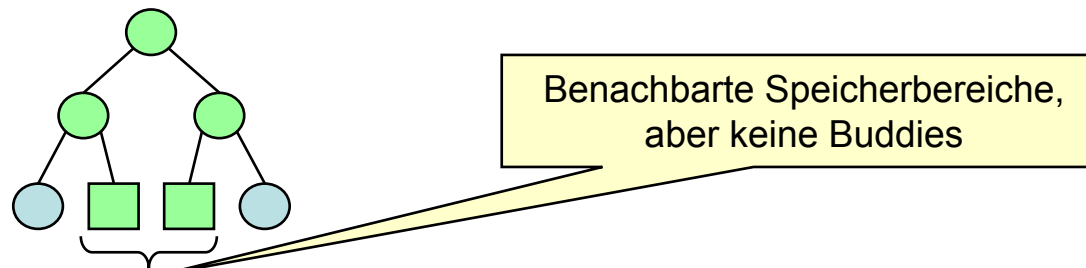
$\alpha \underbrace{0\dots0}_k$

Bei Zerlegung entstehen folgende Buddies:



→ Nachfolgende Buddies unterscheiden sich nur durch ein Bit.

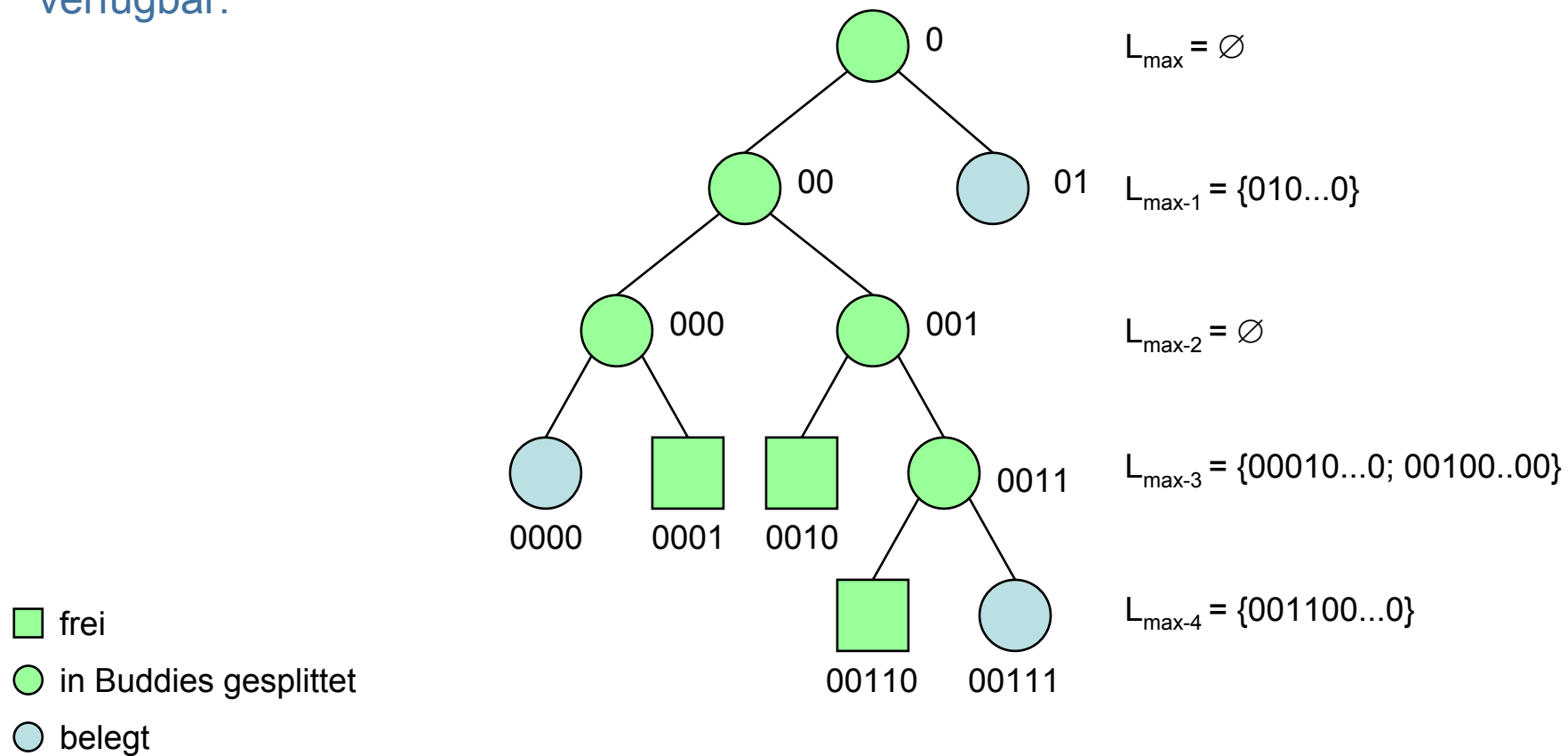
→ Unsön:



6.3 Buddy-Systeme / Adressierung

Problem:

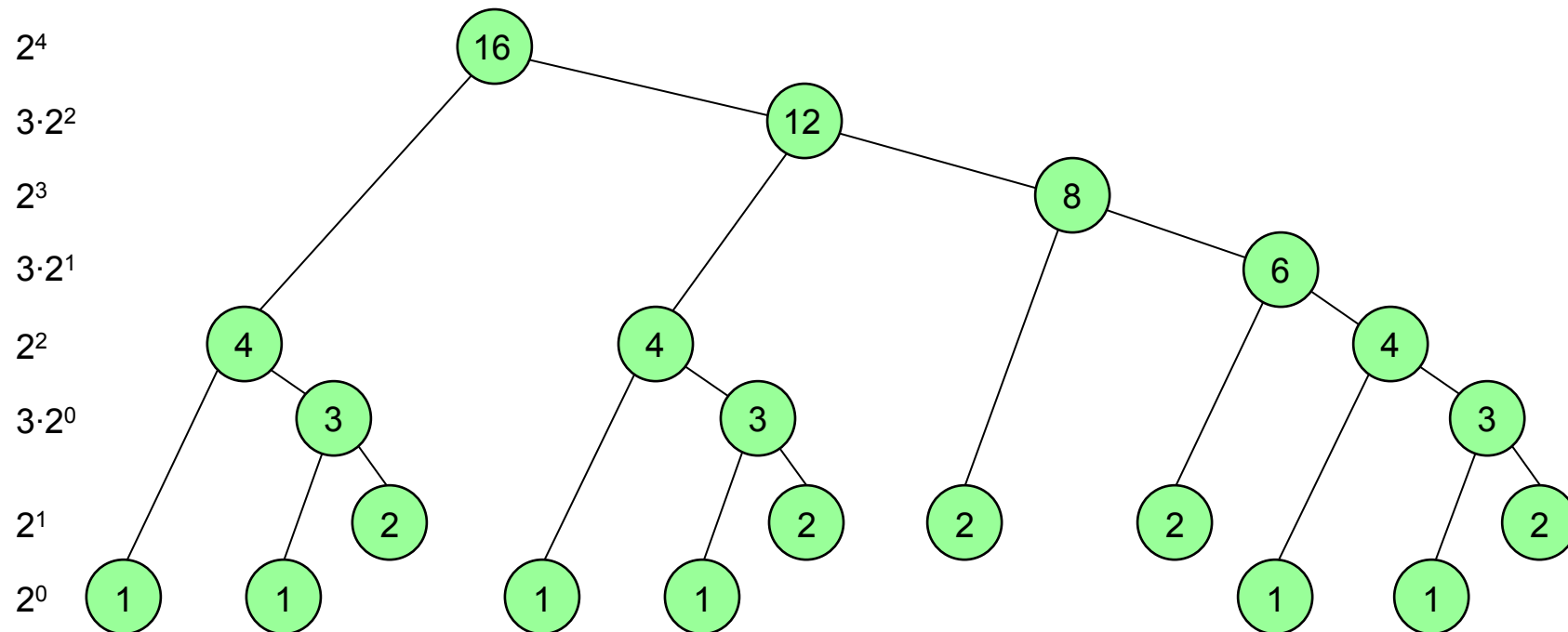
Anforderung der Größe $2^{\max-2}$ ist nicht erfüllbar, obwohl ausreichender Speicherplatz verfügbar.



6.3 Buddy-Systeme / Gewichtete Buddy-Systeme

Gewichtete Buddies erlauben eine feinere Aufteilung des Speichers.

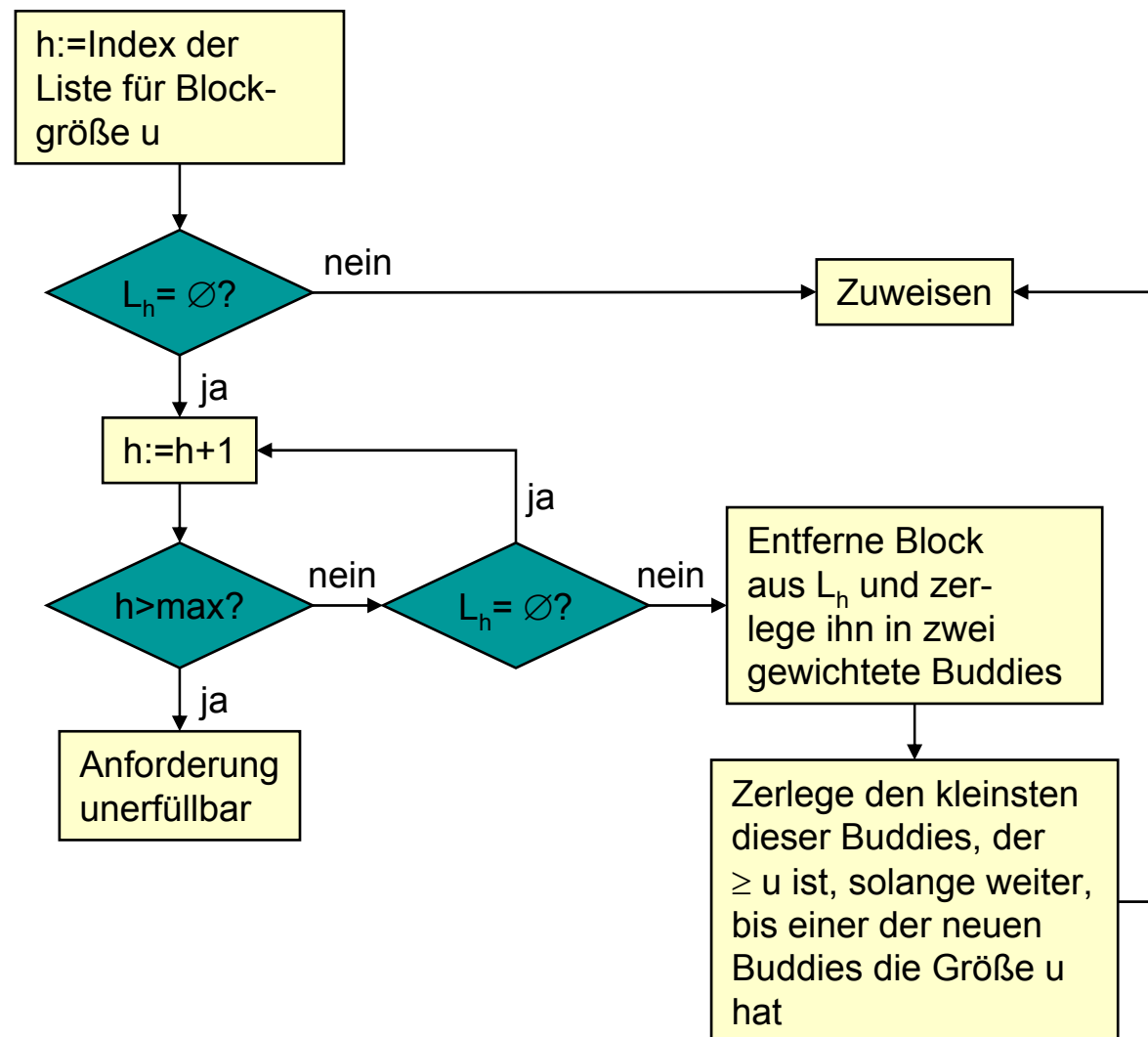
Ein Block der Größe 2^{r+2} wird im Verhältnis 1:3 in Blöcke der Größe 2^r und $3 \cdot 2^r$ zerlegt;
 $3 \cdot 2^r$ ist wiederum zerlegbar in $2 \cdot 2^r = 2^{r+1}$ und 2^r .



- Erhöhter Aufwand, insbesondere Adressberechnung komplizierter.
- Zerlegung eines Blocks kann in einen zu kleinen und einen noch zu großen Block enden.

6.3 Buddy-Systeme / Gewichtete Buddy-Systeme

Platzierungsverfahren für Segment der Größe 2^u : (auch gültig für ungewichtete Buddies)



6.4 Demand-Paging / Paging

Beim **Paging** werden ausschließlich Speicherblöcke **einheitlicher Größe** verwendet.

Der logische Adressraum ist in Seiten (**Pages**) unterteilt.

Der physikalische Speicher ist in Rahmen (**Frames**) unterteilt, die genau eine Seite aufnehmen können.

Im Allgemeinen: Anzahl der Seiten \gg Anzahl der Rahmen

Vorteil: keine Speicherzerstückelung (**externe Fragmentierung**)

Nachteil: Verschnitt innerhalb einer Seite (**interne Fragmentierung**)

Zu entscheiden:

- Wann Seitentausch?
- Welche Seite laden?
- Welche Seite dafür verdrängen?

6.4 Demand-Paging / Paging-Strategien

Mögliche Strategien:

➤ Demand-Paging

Seitenaustausch nur im Falle eines Seitenfehlers.

➤ Demand-Prepaging

Wie Demand-Paging, jedoch ist Austausch mehrerer Seiten bei Seitenfehler möglich.

➤ Look-Ahead

Es können auch Seiten ausgetauscht werden, ohne dass ein Seitenfehler vorliegt.

➔ Demand-Paging ist optimal bezüglich der Seitenfehlerzahl.

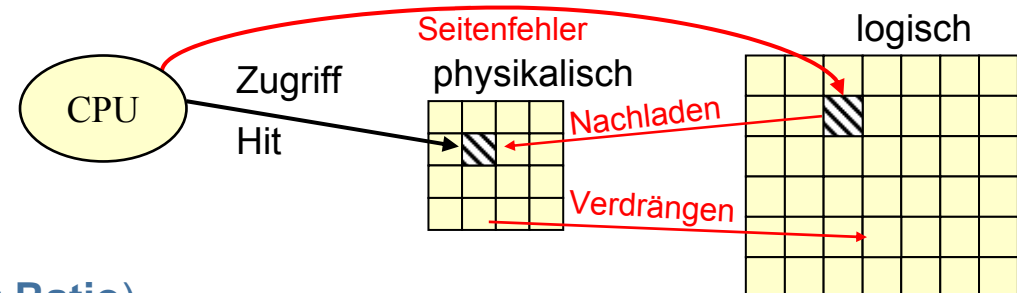
➔ Bei anderen Zielgrößen können auch Demand-Prepaging oder Look-Ahead optimal sein.

Frage bei Demand-Paging:

Welche Seite soll verdrängt werden?

Ziel:

Paging-Strategie mit hoher Trefferquote (**Hit Ratio**)



Seitenfehler: Lade genau die fehlende Seite nach und verdränge genau eine andere

6.4 Demand-Paging / Paging-Strategien

Sei

- $M = \{0, \dots, m-1\}$ der physikalische Adressraum, $n \gg m$ (Frame-Nummern)
- $N = \{0, \dots, n-1\}$ der logische Adressraum (Seitennummern)
- $\omega = r_1 r_2 \dots r_T \in N^T$ eine Folge von Seitenzugriffen → **Reference String**
- $S_t = \{i \mid i \in N \wedge i \text{ belegt Seitenrahmen in } M\}$ der Speicherzustand nach t Zugriffen

Der **Seitenaustauschalgorithmus** A ist ein endlicher Automat ohne Ausgabe:

$A = (N, \{S_t\} \times Q, q_0, g_A)$, mit

N : Eingabemenge,

Q : Menge der Kontrollzustände → Anordnung der gespeicherten Seiten,

$\{S_t\} \times Q$: Speicherzustand und Kontrollzustand,

q_0 : Startzustand und

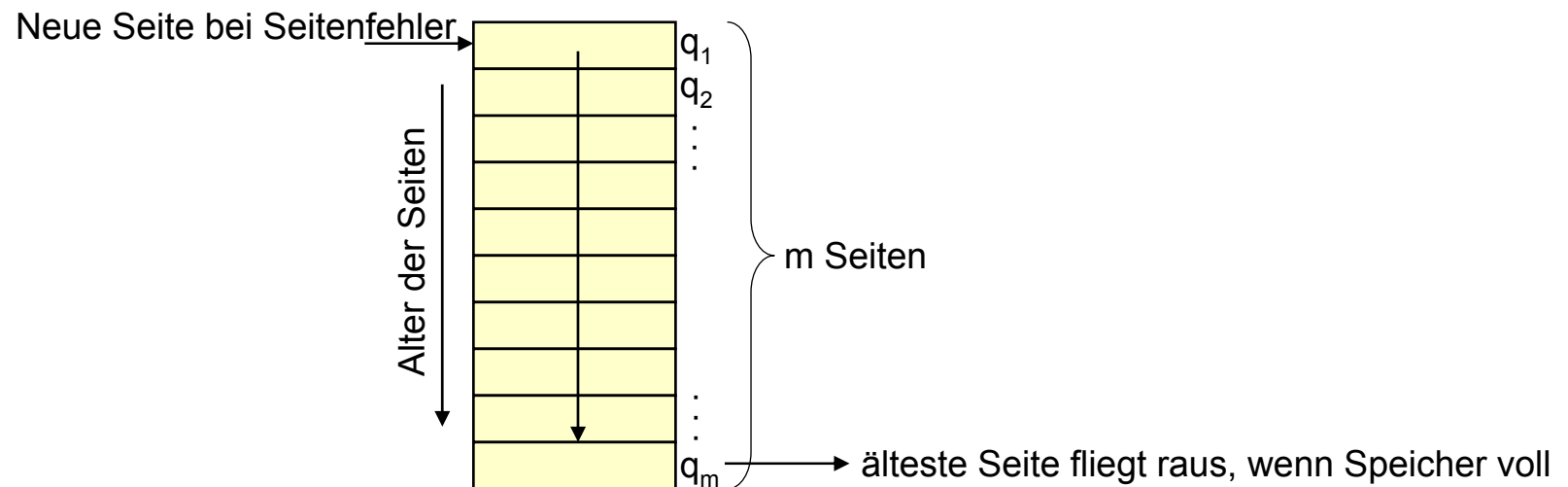
$g_A: N \times (\{S_t\} \times Q) \rightarrow \{S_t\} \times Q$ Überföhrungsfunktion.

- $g_A(r_i, S, q) = (S', q')$ besagt, dass wenn sich der Automat im Zustand (S, q) befindet und ein

Zugriff auf die Seite r_i erfolgt, in den Zustand (S', q') gewechselt wird.

6.4 Demand-Paging / Paging-Strategien

Die **FIFO**-Strategie ordnet die Seiten im Hauptspeicher nach ihrem „Alter“, d.h. nach ihrer Verweilzeit im Hauptspeicher. Im Bedarfsfall wird die älteste Seite ersetzt.



Angenommen der Hauptspeicher fasst m Rahmen. Sei der Kontrollzustand

$$\square q = (q_1, \dots, q_m) \text{ mit } q_i \in \{0, \dots, n-1\},$$

wobei q_1 die Nummer der jüngsten und q_m die Nummer der ältesten Seite im Hauptspeicher ist.

6.4 Demand-Paging / Paging-Strategien

Die Überföhrungsfunktion g_{FIFO} lautet:

$$g_{\text{FIFO}} : N \times (\{S_i\} \times Q) \rightarrow \{S_i\} \times Q$$

$$(r_i, S, q) \mapsto (S', q')$$

derart, dass

- $S' = S, q' = q,$
falls $r_i \in S$

Kein Seitenfehler

- $S' = S \cup \{r_i\}; q' = (r_i, q_1, \dots, q_k),$
falls $r_i \notin S \wedge |S| < m \text{ und } S = (q_1, \dots, q_k)$

Seitenfehler, aber
Speicher noch nicht voll

- $S' = (S \cup \{r_i\}) \setminus \{q_m\}, q' = (r_i, q_1, \dots, q_{m-1}),$
falls $r_i \notin S \wedge |S| = m$

Seitenfehler
und Speicher voll

6.4 Demand-Paging / Paging-Strategien

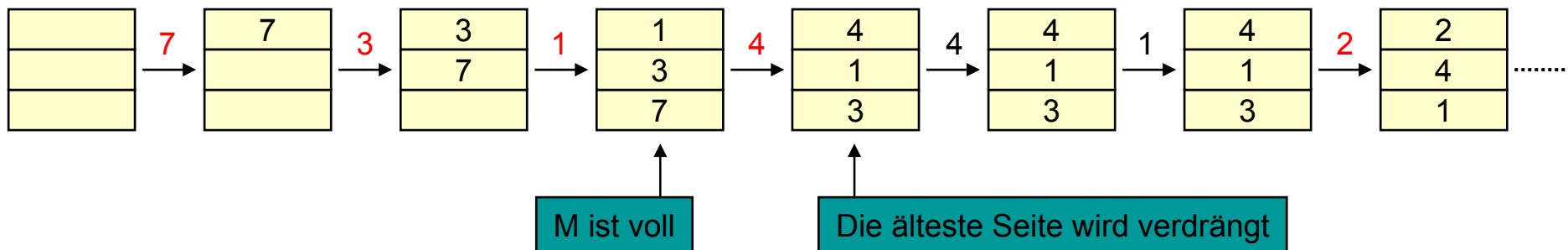
Beispiel:

$N = \{0, \dots, 9\}$

$M = \{0, 1, 2\}$

$\omega = 7, 3, 1, 4, 4, 1, 2, \dots$

Strategie: FIFO

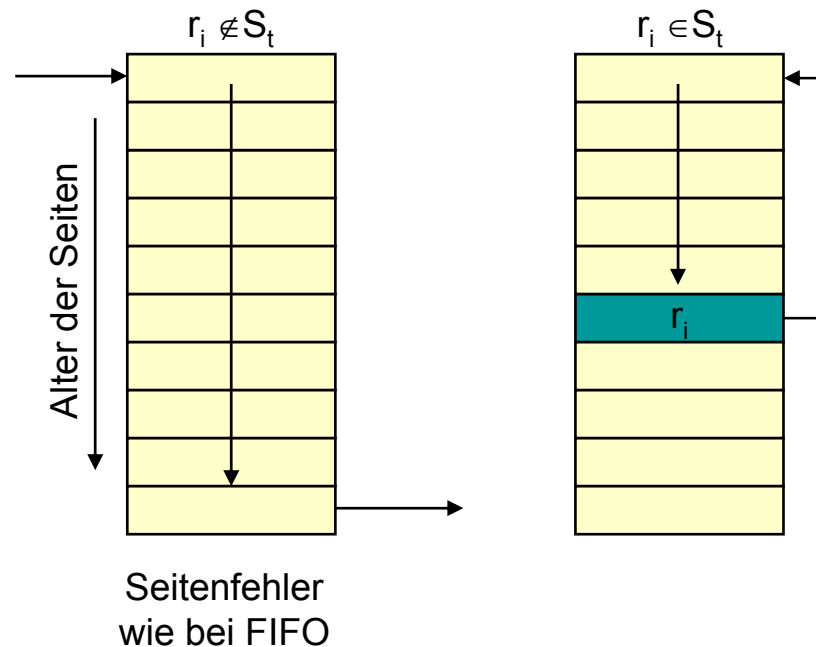


6.4 Demand-Paging / Paging-Strategien

Least-Recently-Used (LRU)

Wie FIFO, jedoch wird bei jedem Zugriff auf eine Seite, die sich bereits im Hauptspeicher befindet, diese „verjüngt“. Die am längsten nicht mehr benutzte Seite wird ausgetauscht.

Tauschkriterium: Maximale Rückwärtsdistanz.

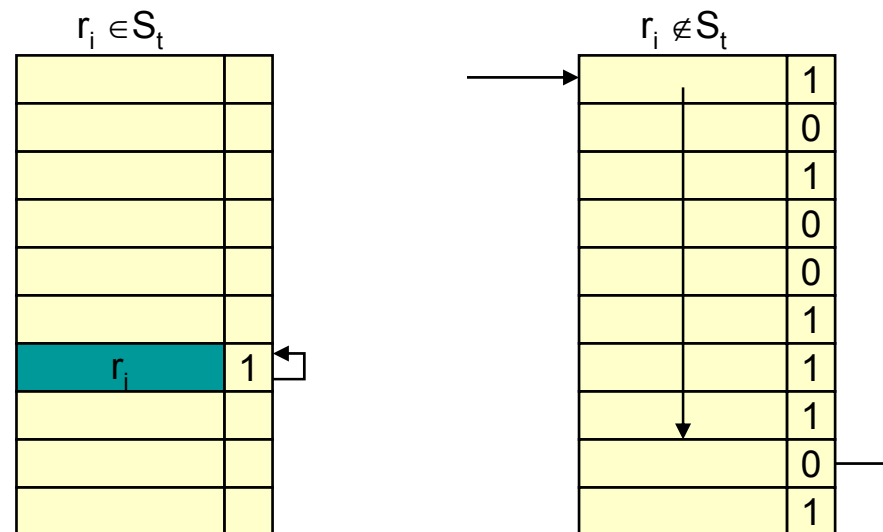


LRU ist besser als FIFO, aber höherer Verwaltungsaufwand.

6.4 Demand-Paging / Paging-Strategien

Second-Chance

Kompromiss zwischen FIFO und LRU. Arbeitet wie FIFO, jedoch existiert ein zusätzliches **Use-Bit** pro Seite, das bei einer zweiten Anforderung der Seite gesetzt wird. Bei einem Seitenfehler wird die älteste Seite mit nicht gesetztem Use-Bit ausgetauscht. Die Use-Bits werden gelöscht, wenn alle Use-Bits gesetzt sind und ein Seitenfehler auftritt.



- Variante:

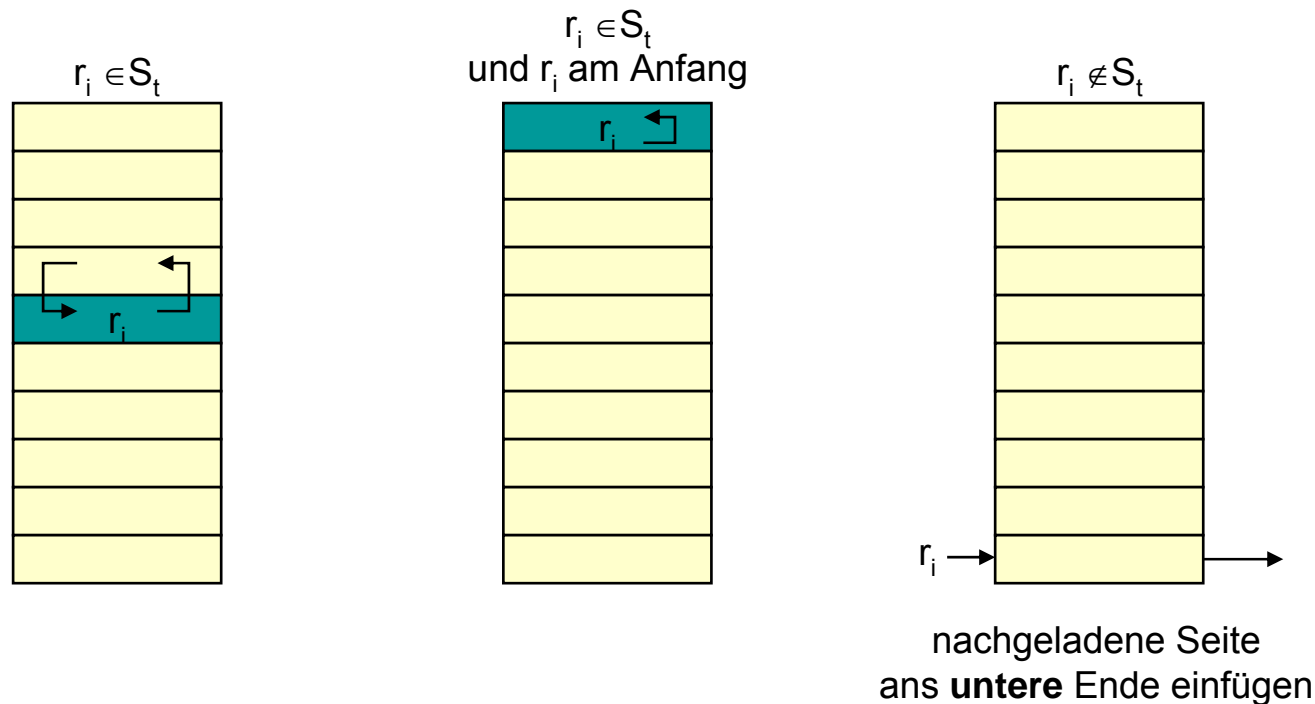
Zusätzliches **Modify-Bit**, das eine Änderung der Seite anzeigt. Beim Verdrängen der Seite in den Hintergrundspeicher muss sie also zurückgeschrieben werden.

Jede Kombination (Use-Bit, Modify-Bit) hat eine bestimmte Priorität: (0,0) (0,1) (1,0) (1,1)

6.4 Demand-Paging / Paging-Strategien

CLIMB → Aufstieg bei Bewährung

Ist die Seite bei einem Zugriff bereits im Speicher vorhanden, steigt sie eine Position höher.



6.4 Demand-Paging / Paging-Strategien

Least-Frequently-Used (LFU)

- Austausch nach Nutzungshäufigkeit (bei gleicher Häufigkeit: die älteste davon)
- Varianten:
 - seit Beginn des Referenzstrings ω
 - innerhalb der letzten h Zugriffe
 - seit dem letzten Seitenfehler

Random

- Auszutauschende Seite wird zufällig bestimmt.

OPT (Optimalstrategie)

- Seite wird nach dem größten Vorwärtsabstand ausgetauscht, d.h. die Seite, die am längsten nicht mehr gebraucht werden wird.
- Schwierig zu realisieren, da die zukünftigen Zugriffe oft nicht bekannt sind.
Aber: Approximation durch geschätzten Vorwärtsabstand.
- Optimal bzgl. der Seitenfehleranzahl → jeder andere Demand-Paging-Algorithmus verursacht mindestens genauso viele Seitenfehler wie OPT ($\forall n, \forall \omega$).
- Dient oft als Vergleich für andere Verfahren.

6.5 Nicht-Demand-Paging

Nicht-Demand-Paging-Strategien nutzen die Tatsache, dass Programme meist „geographisch lokal“ arbeiten:

- Sequenzielle Schleifendurchläufe
- Ausführung von einfachen, linearen Programmcodes
- geographisch geordnete Daten, z.B. Matrizen

➔ Vorausschauender Austausch mehrerer Seiten kann zweckmäßig sein, wenn dies effizienter ist als eine Folge von Einzeltauschoperationen.

Bekanntester Vertreter dieser Seitenersetzungsstrategien ist der **One-Block-Look-Ahead (OBL)**-Algorithmus. OBL arbeitet, solange keine Seitenfehler auftreten, im Wesentlichen wie LRU.

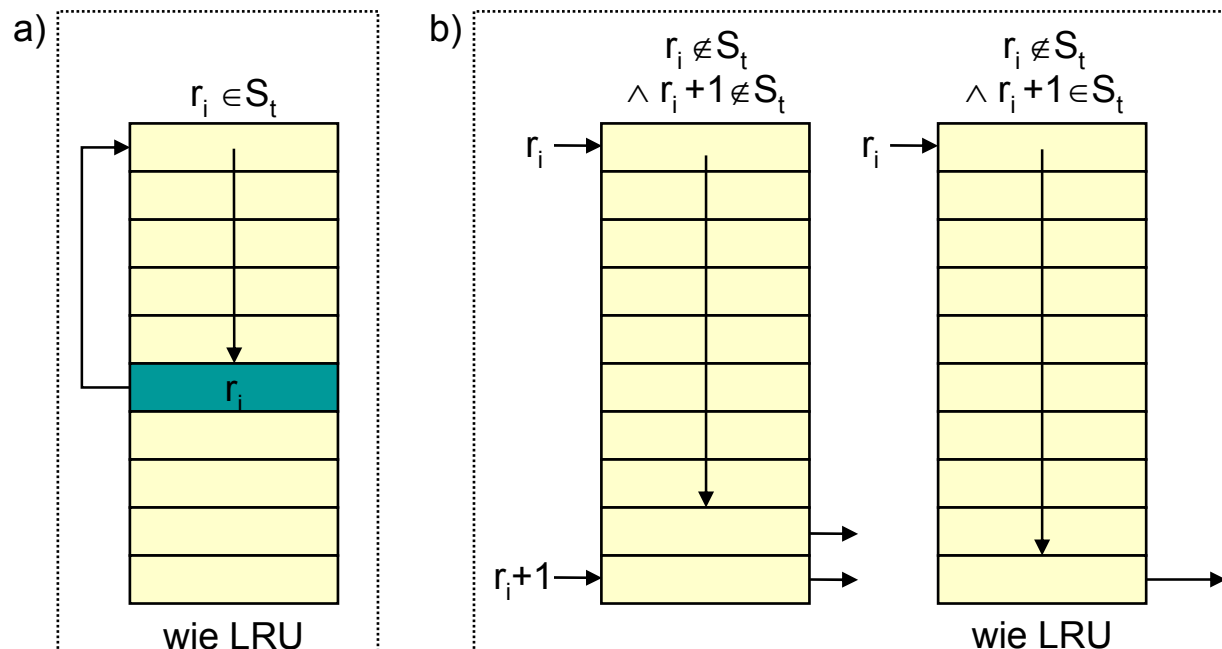
Es können verschiedene OBL-Varianten unterschieden werden, z.B.:

- OBL als Demand-Prepaging-Variante
- OBL als Look-Ahead-Variante

6.5 Nicht-Demand-Paging / One-Block-Look-Ahead

Demand-Prepaging-Variante:

Austausch nur bei Seitenfehler, aber dann ggf. mehrere Seiten auf einmal.



a) r_i bereits im Speicher: Analog zu LRU.

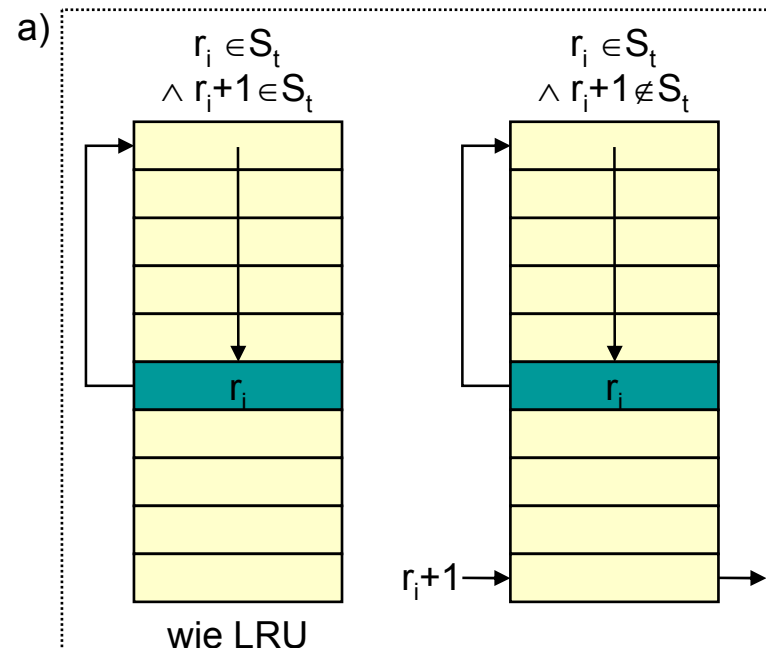
b) Seitenfehler bei r_i : Ist die auf r_i folgende Seite r_{i+1} bereits im Speicher vorhanden?

- Nein: r_{i+1} an letzte Stelle laden.
- Ja: Vorgehensweise wie bei LRU.

6.5 Nicht-Demand-Paging / One-Block-Look-Ahead

Die **Look-Ahead**-Variante unterscheidet sich von der Demand-Prepaging-Variante, falls die angeforderte Seite r_i bereits im Speicher ist.

- r_{i+1} bereits im Speicher: Verhalten wie LRU-Strategie
- r_{i+1} nicht im Speicher: r_{i+1} wird zusätzlich an die letzte Position geladen



➔ Look-Ahead: Austausch evtl. auch in Situationen, in denen kein SF vorliegt.

6.6 Diskussion der Paging-Algorithmen

Kosten von Paging-Algorithmen:

- Kosten entsprechen hier dem Aufwand für eine Speicherzustandsänderung
Aufwand für Bearbeitung eines Seitenfehlers: Laden, Verdrängen (nur bei Inhaltsänderung)
- Annahme: **Verdrängungskosten** proportional zu **Ladekosten**
- Alle diesbezüglichen Kosten sind auf eins normiert:
 - Seitenfehler ist aufgetreten \leftrightarrow Kosten = 1
 - Kein Seitenfehler \leftrightarrow Kosten = 0
- Umspeichern, z.B. bei LRU, CLIMB sei kostenfrei \rightarrow benachteiligt FIFO

- Sei

- X_t : die Menge der beim t-ten Datenzugriff neu geladenen Seiten
- Y_t : die Menge der beim t-ten Datenzugriff verdrängten Seiten
- S_t : der Speicherzustand zum Zeitpunkt t, d.h. die Menge der Seiten, die nach dem t-ten Datenzugriff im Hauptspeicher sind ($S_t = (S_{t-1} \cup X_t) \setminus Y_t$)

6.6 Diskussion der Paging-Algorithmen

Bei Anwendung eines Paging-Algorithmus A auf den Referenzstring $\omega = r_1 r_2 \dots r_T$ unter der Verwendung von m Hauptspeicherseiten ergeben sich folgende Kosten:

a) Gesamtzahl der Seitentransporte, also der Seitenfehler:

$$\alpha(A, m, \omega) := \sum_{t=1}^T |X_t|$$

b) Kosten für das Laden von i Seiten =: h(i) mit

$$h: N_0 \rightarrow N_0 \text{ mit } h(0) := 0$$

$$h(1) := 1 \text{ (normiert)}$$

$$h(i) \geq h(i-1)$$

Vernünftig wäre es, anzunehmen, dass $h(k) \leq k$, d.h. durch Mehrfachnachladen spart man etwas.

c) Gesamtkosten C für den Referenzstring ω :

$$C(A, m, \omega) := \sum_{t=1}^T h(|X_t|)$$

Die folgenden Sätze gelten leider nur für $h(k) \geq k$.

6.6 Diskussion der Paging-Algorithmen

Zu jedem Nicht-Demand-Paging-Algorithmus (NDPA) kann ein Demand-Paging-Algorithmus (DPA) konstruiert werden, der höchstens ebenso viele Seitenfehler macht wie der NDPA.

Wie geht das?

- Merken, welche Seiten NDPA vorausschauend lädt bzw. vorausschauend verdrängt
- Bei Seitenfehler wird DPA die entsprechende Seite laden und irgendeine der Seiten, die NDPA bereits verdrängt hat, verdrängen

DPA macht weniger Fehler als NDPA, wenn eine Seite von NDPA geladen und vor dem entsprechenden Seitenfehler verdrängt wird.

Vorteil von Nicht-Demand-Paging:

-In Abhängigkeit vom Referenzstring ω und Speichergröße m können die Kosten niedriger sein als beim Demand-Paging (wenn $h(k) < k \cdot h(1) = k$).

6.6 Diskussion der Paging-Algorithmen

Es gilt:

Zu jedem Algorithmus A existiert ein DPA A^* mit:

- a) $\alpha(A^*, m, \omega) \leq \alpha(A, m, \omega) \quad \forall m, \omega$
→ DPA ist optimal bezüglich der Zahl der Seitentransporte.
- b) $C(A^*, m, \omega) \leq C(A, m, \omega) \quad \forall m, \omega$
→ DPA ist optimal bezüglich der Kosten.

Die Aussage b) gilt jedoch nur dann, wenn das gemeinsame Nachladen von k Seiten mindestens ebensoviel kostet wie das einzelne Laden von k Seiten, also für $h(k) \geq k \cdot h(1)$, $k \geq 1$.

(„The rationale behind the paging concept“ besagt aber, dass größere Einheiten rationeller zu transportieren sind als kleine).

6.6 Diskussion der Paging-Algorithmen / FIFO-Anomalie

Vermutung:

Je größer der Speicher, desto weniger Seitenfehler, d.h. $\alpha(A, m, \omega) \geq \alpha(A, m+1, \omega)$, und geringere Kosten, wenn $h(k) \geq k$.

Falsch:

Die Anzahl der Seitenfehler nimmt nicht unbedingt mit wachsendem m ab, sondern kann bei bestimmten Strategien (FIFO, SECOND-CHANCE, CLIMB) sogar steigen. Andere Strategien (etwa LRU, OPT, LIFO) sind von diesem Phänomen nicht betroffen.

→ Es gibt m, ω mit: $C(\text{FIFO}, m+1, \omega) > C(\text{FIFO}, m, \omega)$

und $\alpha(\text{FIFO}, m+1, \omega) > \alpha(\text{FIFO}, m, \omega)$.

6.6 Diskussion der Paging-Algorithmen / FIFO-Anomalie

Beispiel: Sei $m = 3$; $\omega_1 = 2\ 3\ 0\ 1$; $\omega_2 = 2\ 0\ 3\ 1\ 4\ 2\ 5\ 3\ 0\ 4\ 1\ 5$; $\omega = \omega_1 \cdot \omega_2^k$

ω	2 3 0 1	2 0 3 1 4 2 5 3 0 4 1 5	2 0 3 1 4 2 5 3 ...	
3 Seiten	2 3 0 1	2 3 4 5 0 1	2 3 4 ...	Speicherzustand + Kontrollzustand
	2 3 0	1 2 3 4 5 0	1 2 3 ...	
	2 3	0 1 2 3 4 5	0 1 2 ...	
Seitenfehler	x x x x	x x x x x x	x x x ...	
4 Seiten	2 3 0 1	4 2 5 3 0 4 1 5	2 0 3 1 4 2 5 ...	Speicherzustand + Kontrollzustand
	2 3 0	1 4 2 5 3 0 4 1	5 2 0 3 1 4 2 ...	
	2 3	0 1 4 2 5 3 0 4	1 5 2 0 3 1 4 ...	
	2	3 0 1 4 2 5 3 0	4 1 5 2 0 3 1 ...	
Seitenfehler	x x x x	x x x x x x x x	x x x x x x x ...	

Die Vergrößerung des Speichers führt bei diesem speziellen ω zu einer Verdopplung der Seitenfehler und somit der Kosten.

Grund: Bei FIFO altern die Seiten unabhängig von ihrer Aktualität.

$$\frac{\alpha(\text{FIFO}, 3, \omega = \omega_1 \cdot \omega_2^k)}{\alpha(\text{FIFO}, 4, \omega = \omega_1 \cdot \omega_2^k)} \xrightarrow{k \rightarrow \infty} \frac{1}{2}$$

Vermutung: $\frac{\alpha(\text{FIFO}, m, \omega)}{\alpha(\text{FIFO}, m+1, \omega)} \geq \frac{1}{2}; \quad \forall m, \omega$

6.6 Diskussion der Paging-Algorithmen / Stack-Algorithmen

Die FIFO-Anomalie tritt bei bestimmten Algorithmen nicht auf: LRU, OPT, LIFO.

Diese Algorithmen nennt man **Stack-Algorithmen**.

Sei $S(m, \omega)$ die Menge der Seitennummern, die am Ende der Abarbeitung eines Referenzstrings ω durch einen DPA unter Verwendung von m Rahmen im Speicher stehen. Ein Algorithmus heißt **Stack-Algorithmus** genau dann, wenn

$$S(m, \omega) \subseteq S(m+1, \omega) \quad \forall m, \omega$$

gilt.

- Folgerung:

Ein **Hit** bei m Seiten liefert auch einen Hit bei $m+1$ Seiten.

→ Seitenfehler kann nicht mit m wachsen

- Für jeden Stack-Algorithmus gilt:

-Mit wachsender Speichergröße sinken die Fehlerrate und die Nachladekosten.

→ FIFO ist kein Stack-Algorithmus. Das Gleiche gilt für CLIMB und einige andere mehr.

6.6 Diskussion der Paging-Algorithmen / Prioritätsalgorithmen

Ein Demand-Paging-Algorithmus heißt **Prioritätsalgorithmus** genau dann, wenn es für alle ω (**unabhängig von m**) eine Folge $\pi_1, \pi_2, \dots, \pi_{T-1}$ von so genannten Prioritätslisten gibt, für die gilt:

- π_i ist eine geordnete Liste der in r_1, r_2, \dots, r_i vorkommenden Seitennummern
- ist $\omega = r_1 r_2 \dots r_t$ und $|S(m, \omega)| = m$ und $r_{t+1} \notin S(m, \omega)$, d.h. es muss eine Seite ersetzt werden, dann bestimmt sich die zu verdrängende Seite durch die Liste π_t mittels:

$$S(m, \omega \cdot r_{t+1}) = S(m, \omega) \cup \{r_{t+1}\} \setminus \min_{\pi_t} S(m, \omega)$$

- $\min_{\pi_t} S(m, \omega)$: Das Element niedrigster Priorität in $S(m, \omega)$ bzgl. der durch π_t gegebenen Anordnung, d.h. Tausch der gemäß Prioritätsliste unwichtigsten Seite.

-Zusammenfassung:

Prioritätsalgorithmen bestimmen das Tauschverhalten unabhängig von m .

6.6 Diskussion der Paging-Algorithmen / Prioritätsalgorithmen

Diverse Prioritätsalgorithmen und die verwendeten **Prioritätslistenanordnungen**:

Algorithmus	Prioritätslistenanordnung
LRU	wachsende Rückwärtsdistanz
OPT	wachsende Vorwärtsdistanz
LIFO	wachsende Zeit des Eintritts in den Hauptspeicher
LFU	abnehmende Häufigkeit der Benutzung

Diese Algorithmen
sind
Stack-Algorithmen!

Weiterhin gilt:

- A ist Prioritätsalgorithmus, wenn es eine von m unabhängige Anordnung der Seiten gibt, die den Austausch regelt.
- A ist Prioritätsalgorithmus \Rightarrow A ist Stack-Algorithmus
- Zu jedem Stack-Algorithmus A existiert eine Folge von Prioritätslisten, d. h.:
- A ist Stack-Algorithmus \Rightarrow A ist Prioritätsalgorithmus

Vorsicht:

FIFO ist kein Prioritätsalgorithmus!

Sei die Ordnung der Seiten nach ihrem Alter im Hauptspeicher die Prioritätsliste. Also:

FIFO = Prioritätsalgorithmus (?) FIFO = Stack-Algorithmus (?) Anomalie unmöglich ?

Denkfehler: Die vorgeschlagene Prioritätsliste ist nicht unabhängig von m!

Inhaltsverzeichnis

7.1 Einleitung

- Motivation
- Wahl des Multiprogramminggrads
- Hauptspeicheraufteilung
- Lifetime-Funktion

7.2 Working-Set

- Working-Set-Strategie

7.3 Einstellung der Fenstergröße h

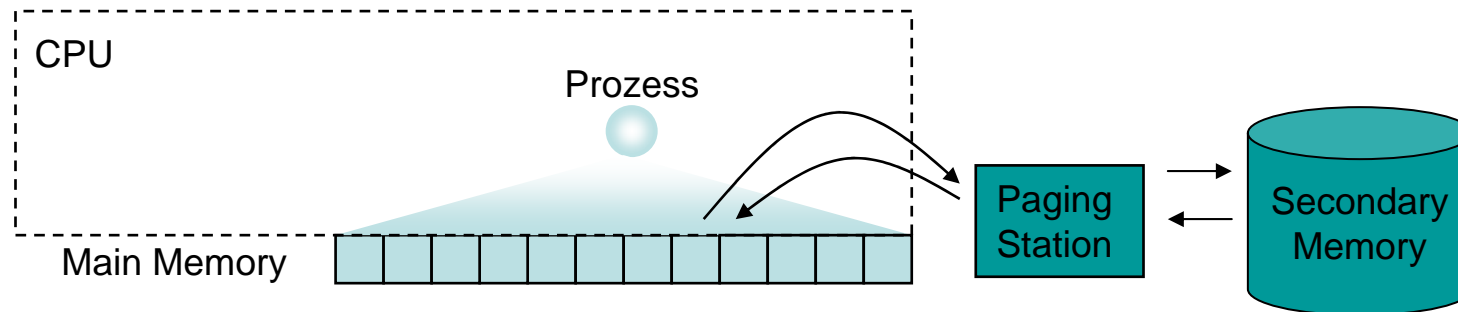
- Das Knie-Kriterium
- Das $L=S$ -Kriterium
- Das 50%-Kriterium
- Exkurs Warteschlangensysteme
- Probleme der Working-Set-Strategie

7.4 Optimale Strategie VOPT

- VOPT vs. WS
- Kostenbetrachtung

7.1 Einleitung / Motivation

Bisherige Betrachtung: Speicherzuteilung bei Einprogrammbetrieb



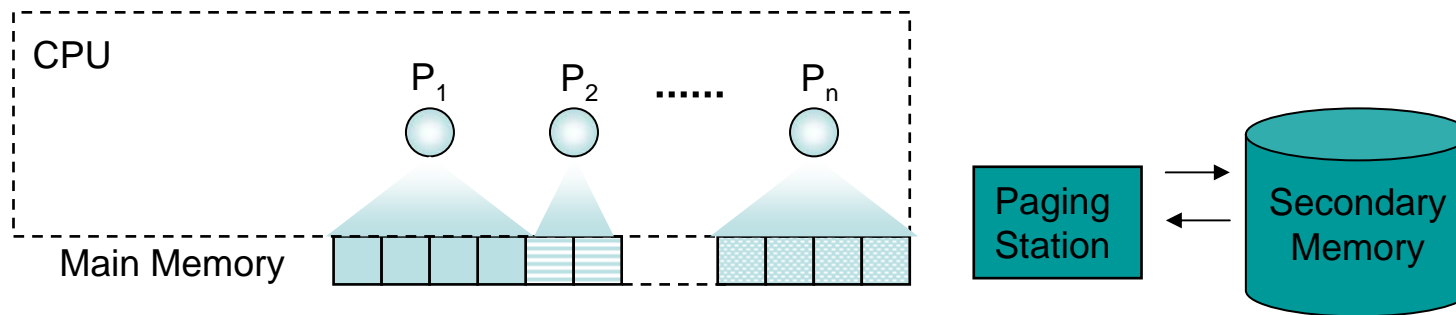
Aber:

Geschwindigkeitsdiskrepanz zwischen CPU und Endgeräten

⇒ Einprogrammbetrieb ist ineffizient!

7.1 Einleitung / Motivation

Speicherzuteilung bei Multiprogramming



Wie soll der gemeinsame Hauptspeicher verwaltet werden?

- Wie viele Rahmen erhält jeder einzelne Prozess?
- Wie viele Prozesse können gleichzeitig ausgeführt werden?
 - ➔ Wahl des Multiprogramminggrads n ?

7.1 Einleitung / Wahl des Multiprogrammingsgrads

Multiprogrammingsgrad n zu klein gewählt

⇒ Verschwendung von Systemressourcen

Multiprogrammingsgrad n zu groß gewählt

⇒ Zahl der Seitenfehler senkt den Systemdurchsatz
(Prozesse behindern sich gegenseitig, weil Speicher zu klein ist)

Extrembeispiel: „Thrashing-Effekt“

$\omega = 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, \dots$

m : Anzahl Seiten

FIFO:

$m \leq 4 \Rightarrow$ ein Seitenfehler pro Zugriff

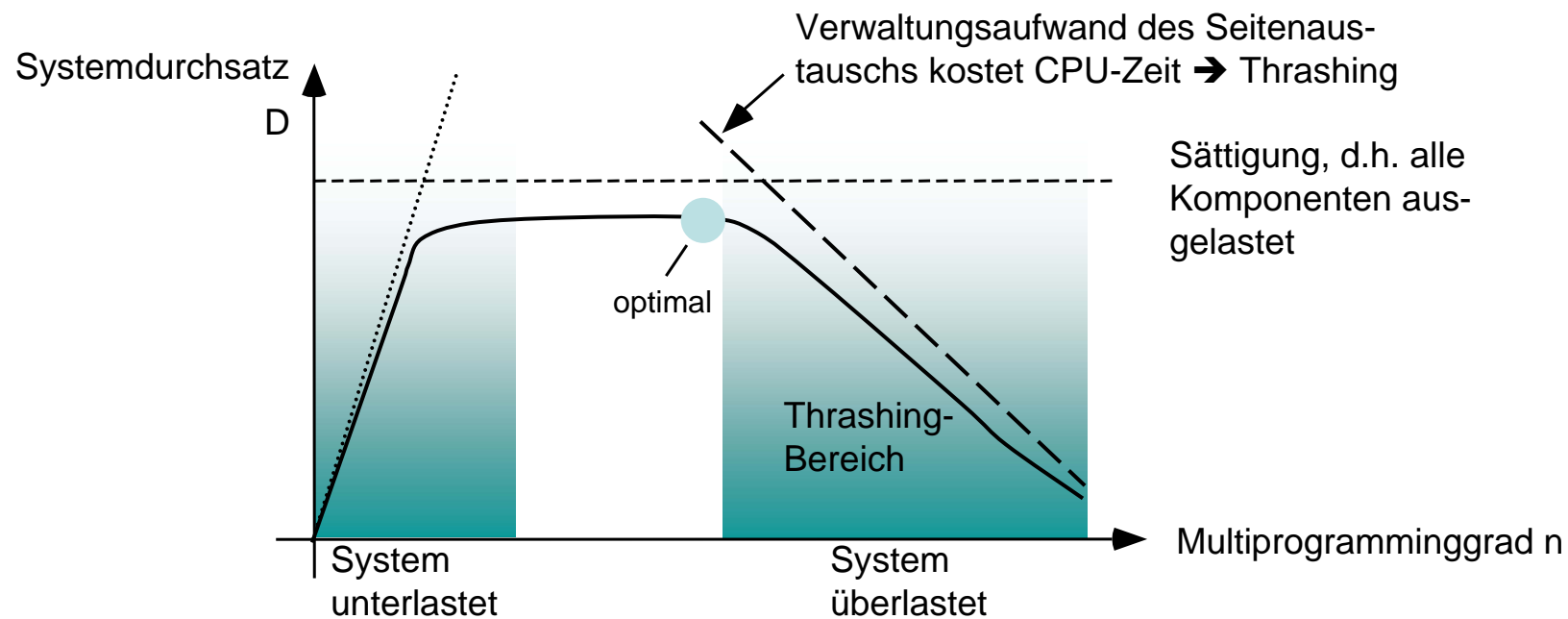
$m \geq 5 \Rightarrow$ keinerlei Seitenfehler (nach erstem Laden)

Typisch:

Ab einer bestimmten Untergrenze steigt die Seitenfehlerzahl drastisch an.

7.1 Einleitung / Wahl des Multiprogrammingsgrads

Thrashing durch steigenden Multiprogrammingsgrad



- Schranke gegeben durch Anforderungen der Prozesse
- - - - Schranke gegeben durch Systemleistung
- - - - Schranke gegeben durch gegenseitige Behinderung der Prozesse

7.1 Einleitung / Hauptspeicheraufteilung

Aktiver Rahmen:

Ein Rahmen heißt **aktiv**, wenn die Wahrscheinlichkeit hoch ist, dass er in naher Zukunft benötigt wird.

Detailfragen:

- **Wie viele Rahmen** sind **pro Prozess** zu verwenden?

Anzahl der momentan aktiven Rahmen.

- Wann soll ein **neuer Prozess aufgenommen** werden?

Starten, wenn genügend Speicher für seine aktiven Rahmen zur Verfügung steht.

- Wann soll ein **aktiver Prozess stillgelegt** werden?

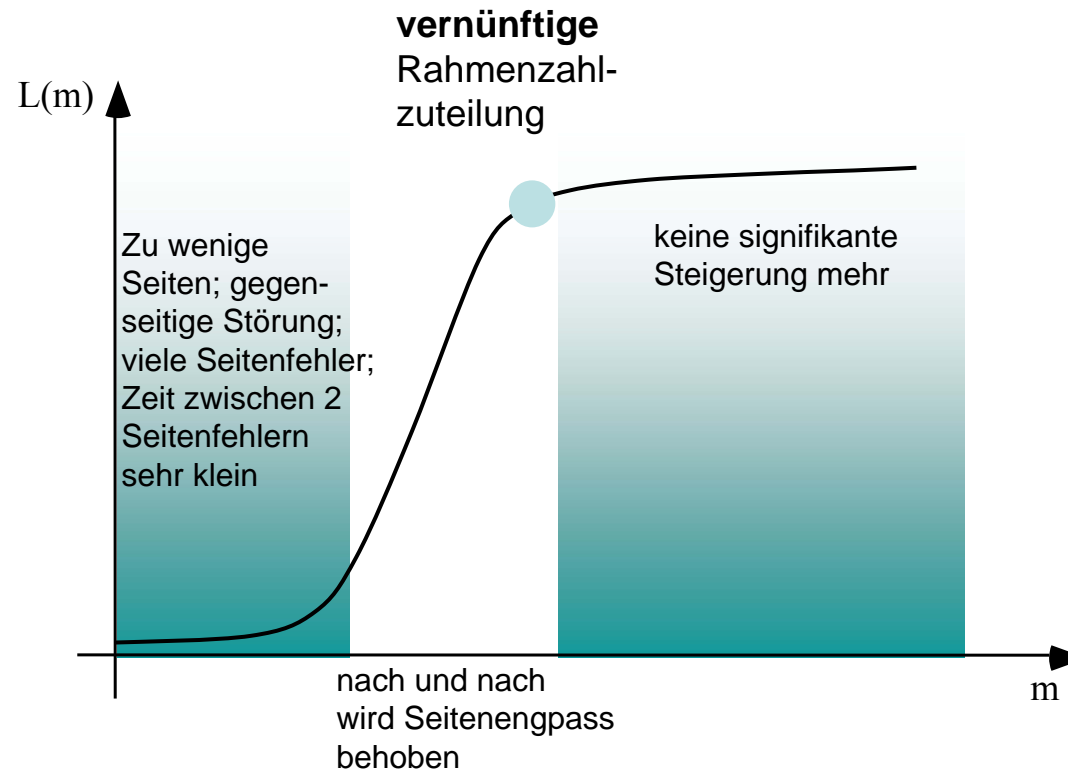
Lege Prozess still, wenn alle Tauschkandidaten aktiv sind.

- Wann ändert sich der Speicherbereich?

Der Speicherbereich ändert sich, wenn die Zahl der aktiven Rahmen bei freiem Speicher steigt.

7.1 Einleitung / Lifetime-Funktion

Die **Lifetime-Funktion $L(m)$** gibt die mittlere Zeit zwischen aufeinanderfolgenden Seitenfehlern in Abhängigkeit von der zugeordneten Rahmenzahl m an.



(Unberücksichtigt: FIFO Anomalien)

7.2 Working-Set

Ziel:

Schätzung von $L(m)$ und optimalem m

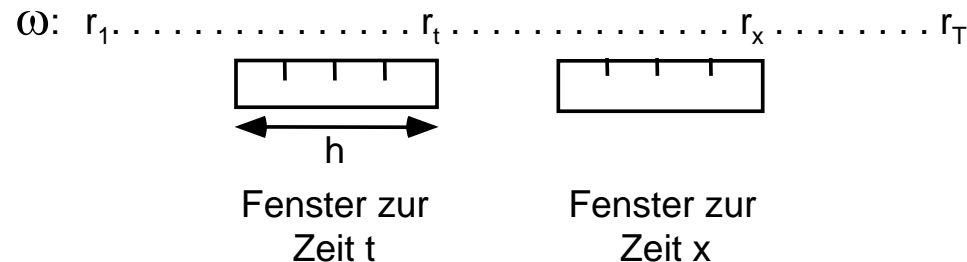
Annahmen:

- Die jüngere Vergangenheit korreliert mit der unmittelbaren Zukunft
- Die Anzahl der aktiven Seiten pro Prozess ändert sich selten
- Die Prozesse verhalten sich „lokal“

Lösung:

Betrachte ein Rückwärtsfenster der Größe h , d.h. die Menge der in der unmittelbaren

Vergangenheit benötigten Seiten → Working-Set



7.2 Working-Set

Definition:

Sei $\omega = r_1 r_2 r_3 \dots r_t \dots r_T$ ein Referenz-String eines Prozesses. Der **Working-Set** $W(t, h)$ dieses Prozesses zur Zeit t unter einem Rückwärtsfenster der Größe h ist definiert als

$$W(t, h) := \bigcup_{i=t-h+1}^t \{r_i\}.$$

$W(t, h)$ ist also die Menge der Seiten, die bei den letzten h Zugriffen mindestens einmal referenziert wurden.

7.2 Working-Set

Sei $w(t,h)$ die Mächtigkeit von $W(t,h)$, d.h.: $w(t,h) = |W(t,h)|$

Es gilt: $h \leq h' \Rightarrow w(t,h) \leq w(t,h')$

Ferner gilt offenbar:

Bei „lokalen“ Prozessen ist $w(t,h)$ klein.

Bei „nicht-lokalen“ Prozessen ist $w(t,h)$ groß.

Beobachtung:

Wird h **zu klein** gewählt

\Rightarrow nicht alle aktiven Seiten sind im Working-Set

Wird h **zu groß** gewählt

\Rightarrow viele inaktive Seiten sind im Working-Set

7.2 Working-Set / Working-Set-Strategie

- Die **Working-Set-Strategie**:

1. Fenstergröße **h** *gut* einstellen.
2. Den Prozessen die Rahmen der aktuellen Working-Sets zuteilen.
Speicher frei → ggf. neuen Prozess aktivieren.
3. Verwendete Tauschkandidaten:
Verwende die Seiten, die in keinem Working-Set sind.
4. Kein Tauschkandidat:
Lege einen Prozess still.

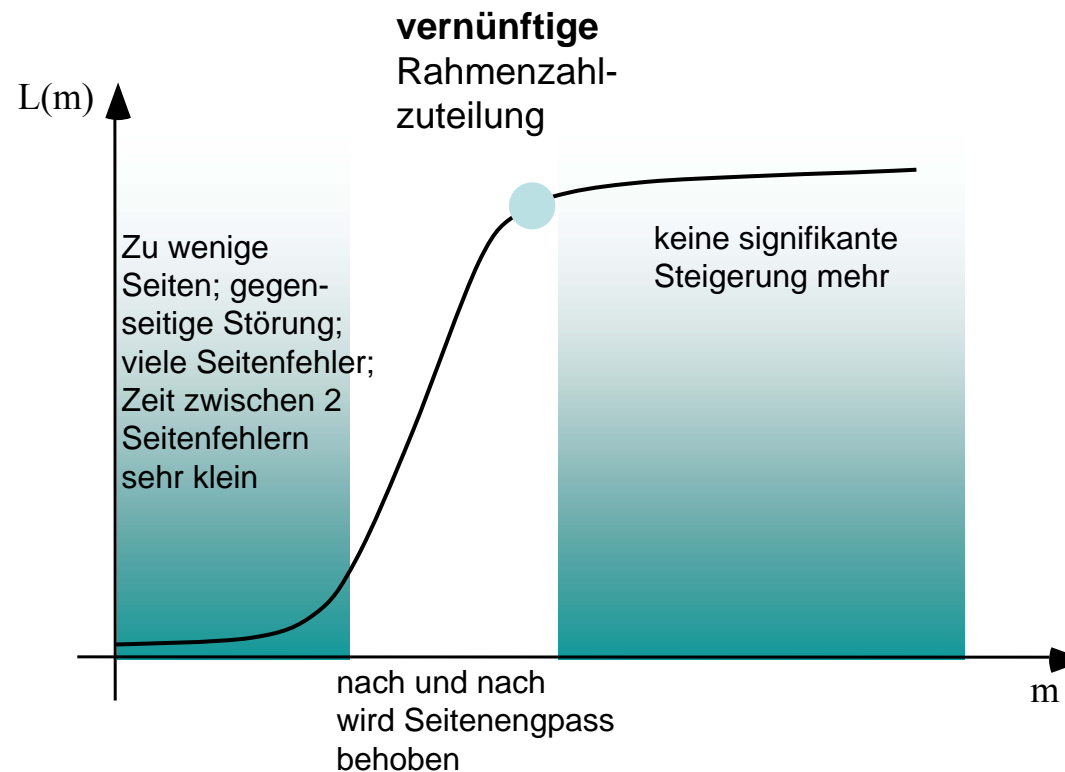
- Problem:

Wahl einer geeigneten Fenstergröße **h**

- Methoden zur Bestimmung einer geeigneten Fenstergröße **h**:
 - Knie-Kriterium
 - L=S-Kriterium
 - 50%-Kriterium

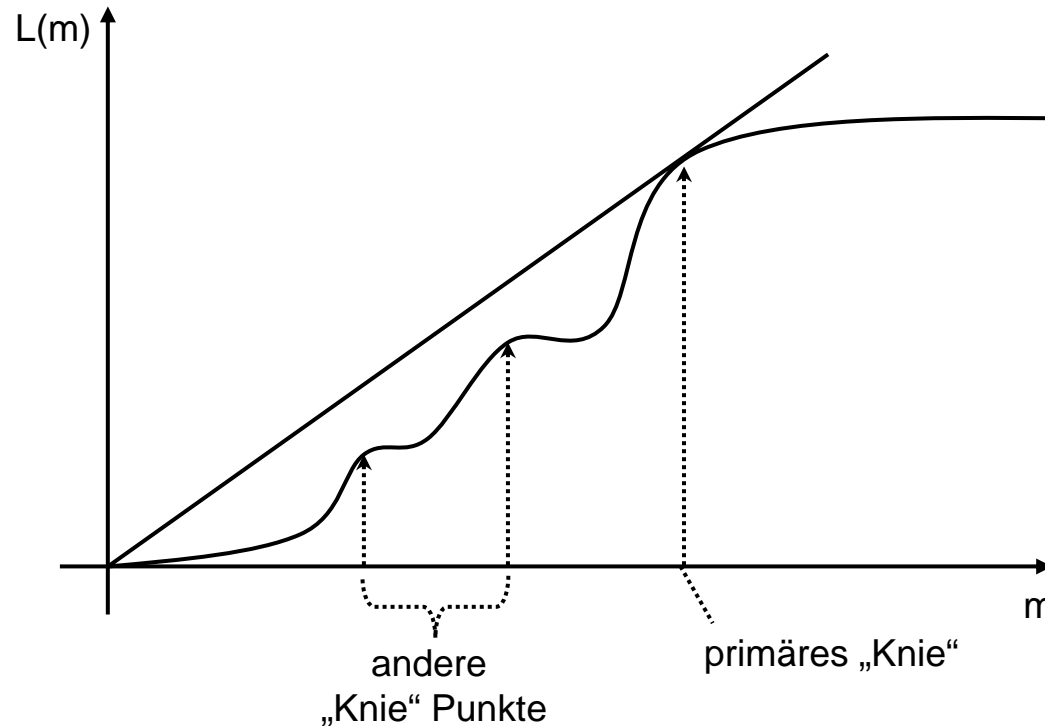
7.3 Einstellung der Fenstergröße h / Knie- Kriterium

Erinnerung: Lifetime-Funktion



7.3 Einstellung der Fenstergröße h / Knie-Kriterium

Knie: Tangente vom Nullpunkt aus an $L(m)$



Bestimmung von h :

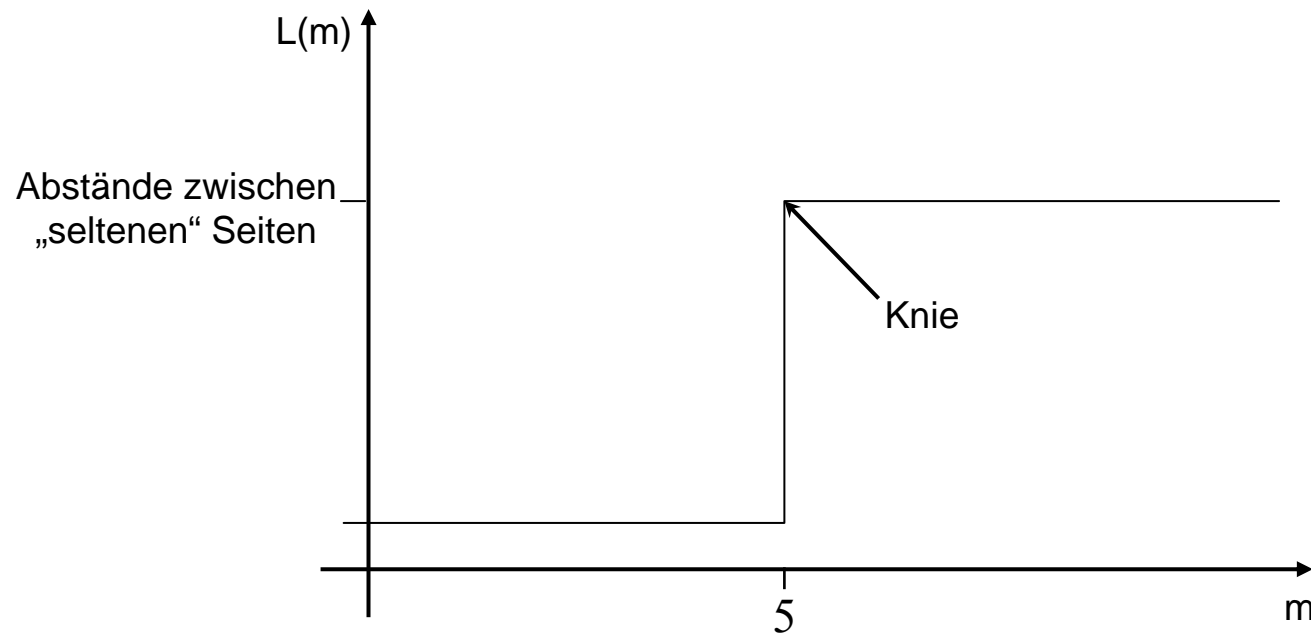
Wähle h so, dass die durchschnittliche Größe eines Working-Sets etwa der Lage des primären Knies entspricht.

7.3 Einstellung der Fenstergröße h / Knie-Kriterium

Extremfall: Lineare Schleife mit spurious pages

$\omega = 1,2,3,4,5,1,2,3,4,5,\dots$

Seiten 6,7,8.... sind selten



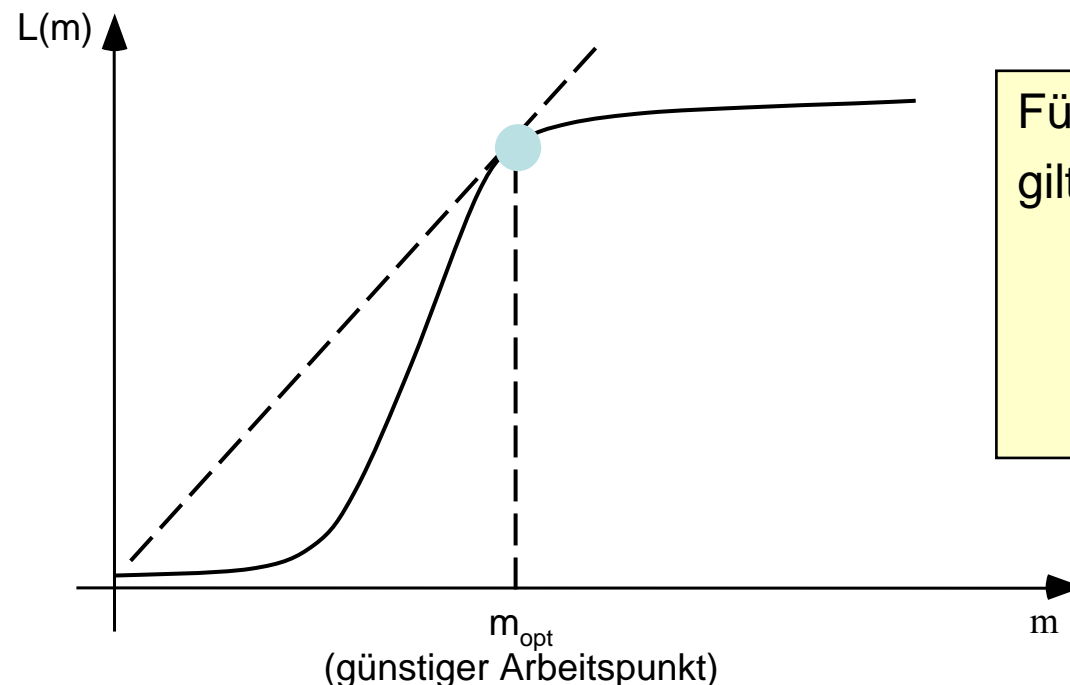
7.3 Einstellung der Fenstergröße h / Knie-Kriterium

Wie schätzt man die Lage des primären Knies?

Das Knie-Kriterium:

Wähle h so, dass $w(t, h) \approx m_{\text{opt}}$ gilt!

Dabei ist m_{opt} die zum **primären Knie** gehörige Seitenzahl.



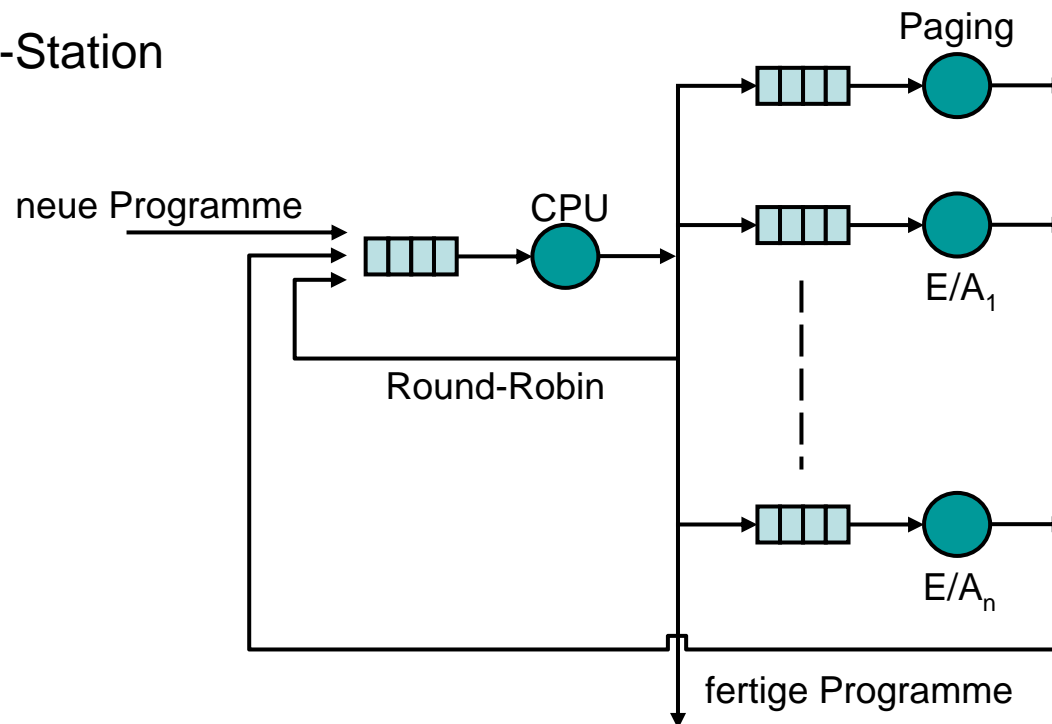
Für die optimale Fenstergröße h
gilt somit:

$$\frac{L(h)}{h} \geq \frac{L(h^*)}{h^*}, h^* \in \mathbb{N}$$

7.3 Einstellung der Fenstergröße h / L=S-Kriterium

Im **Central-Server-Model** alterniert ein Programm zwischen verschiedenen Phasen:

- Rechenphasen
- E/A-Phasen
- Paging-Station



7.3 Einstellung der Fenstergröße h / $L=S$ -Kriterium

Central-Server-Model

⇒ Systemleistung wird durch Engpässe (Bottlenecks) begrenzt

⇒ E/A-Bound: eine oder mehrere E/As sind Engpässe

Untersuche Engpass bei Paging-Station!

Idee:

Systemdurchsatz D ist beschränkt durch:

Mittlere Jobrate $1/T$, wobei T die mittlere Rechenzeit pro Job ist

Kapazität der Paging-Station

Seien

$a(m)$: **Seitenfehlerrate** eines Jobs bei m Seiten

b : **Bedienrate** der Paging-Station, d.h. $1/b$ ist die mittlere Bediendauer eines Paging-Request

T : **Mittlere Jobdauer**

D : **Systemdurchsatz**

7.3 Einstellung der Fenstergröße h / L=S-Kriterium

Für den Durchsatz D gilt:

$$D \leq \frac{1}{T} \cdot \min \left(1, \frac{b}{a(m)} \right)$$

Mit $a(m) = \frac{1}{L(m)} = \frac{1}{\text{Lifetime}}$

und $b = \frac{1}{S} = \frac{1}{\text{Swapttime}} \left(= \frac{1}{\text{Bediendauer}} \right)$

S ist die benötigte Zeit zur
Bedienung eines Seitenfehlers

ergibt sich:

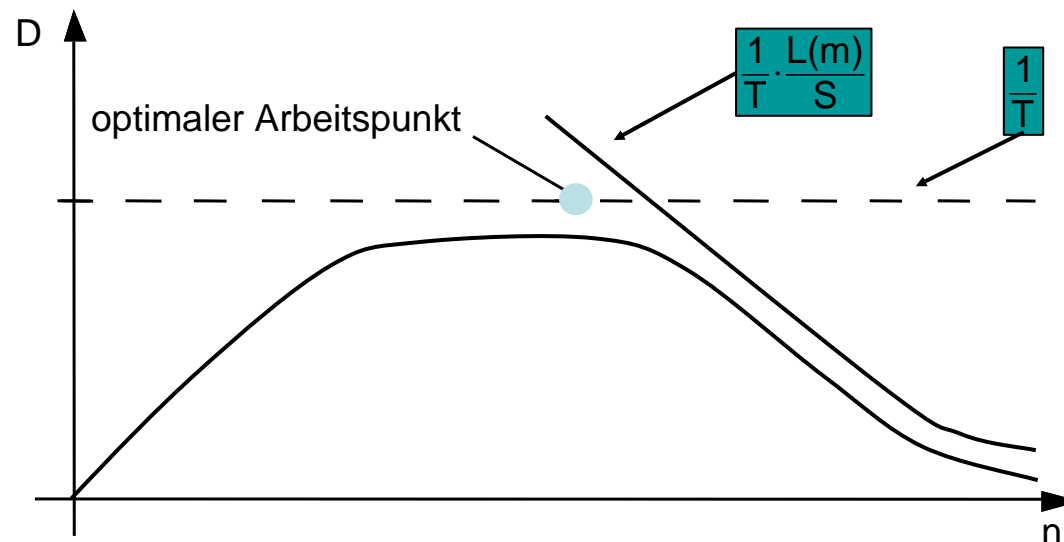
$$D \leq \frac{1}{T} \cdot \min \left(1, \frac{L(m)}{S} \right) \implies \text{Gut erfüllt, wenn } L(m) \approx S \text{ (Lifetime = Swapttime)}$$

7.3 Einstellung der Fenstergröße h / $L=S$ -Kriterium

Da insgesamt eine feste Seitenzahl M zur Verfügung steht, ist die Zahl m der einem einzigen Programm zur Verfügung gestellten Seiten, umgekehrt proportional zum Multiprogramminggrad n , d.h.

$$L(m) \approx L(M/n)$$

Mit wachsendem n wird also m und damit $L(m)$ bzw. $L(m)/S$ für $S=\text{const}$ kleiner.



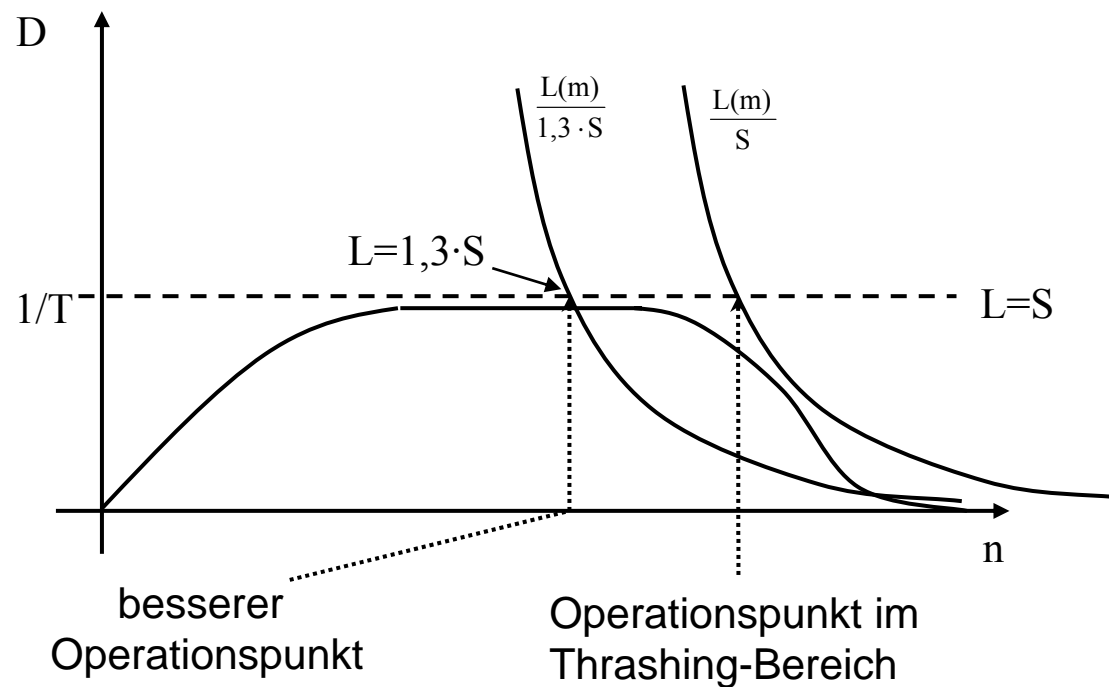
7.3 Einstellung der Fenstergröße h / $L=S$ -Kriterium

Verbesserung:

$L(m) = c \cdot S$, wobei c ein Erfahrungswert ist.

$c=1,3$ verschiebt Arbeitspunkt etwas nach links.

➔ kleinerer, weniger riskanter Multiprogrammgrad.



7.3 Einstellung der Fenstergröße h / 50%-Kriterium

Idee des 50%-Kriteriums:

Wähle Fenstergröße h so, dass Paging-Station zu 50% ausgelastet ist.

Bemerkung:

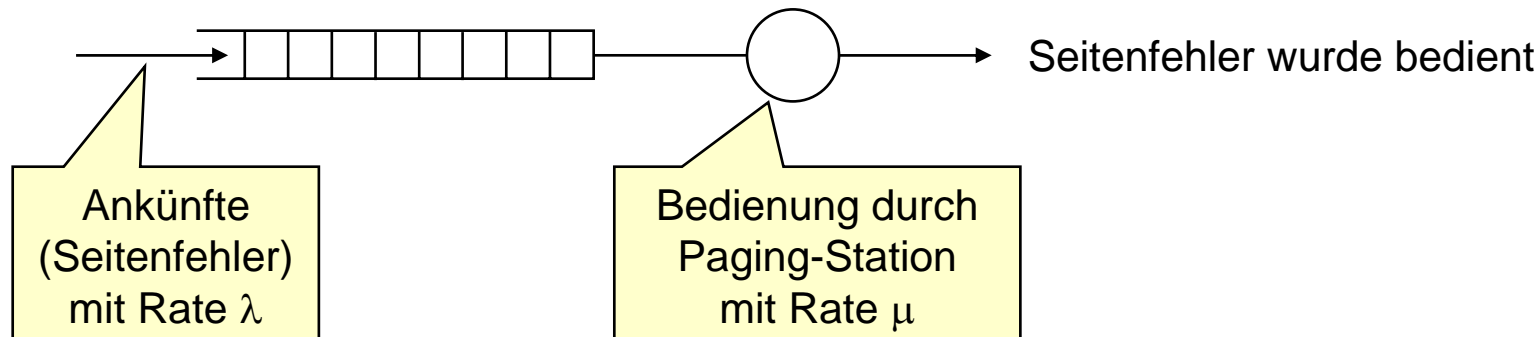
Kriterium ist fast identisch zum „L=S“-Kriterium.

Nachweis:

mittels Warteschlangensysteme

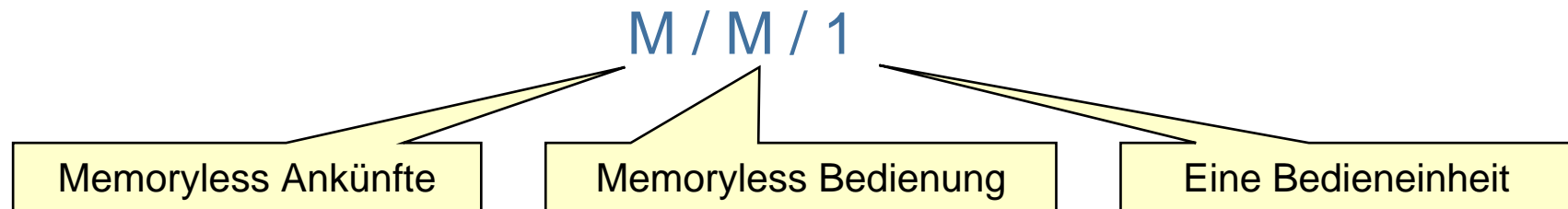
7.3 Einstellung der Fenstergröße h / 50%-Kriterium

Paging-Station ist ein **Warteschlangensystem**:



Wie sehen Ankünfte und Bedienung aus?

Einfachstes Modell:



7.3 Einstellung der Fenstergröße h / 50%-Kriterium „Memoryless“:

$$P[\text{Zeit zwischen 2 Ankünften} \leq t] = 1 - e^{-\lambda t} \quad (\text{exponentialverteilt mit Par. } \lambda)$$

$$P[\text{Bediendauer} \leq x] = 1 - e^{-\mu x} \quad (\text{exp.vert. mit Par. } \mu; \text{ Mittelwert } 1/\mu)$$

Für **M/M/1** gilt:

\bar{N} = mittlere Zahl von „Kunden“ im System (wartend bzw. in Bedienstation)

$$= \rho / (1 - \rho), \text{ wobei}$$

$$\rho = \lambda / \mu \text{ (Systemlast)}$$

Die Paging-Station soll im Mittel immer etwas zu tun haben, aber nicht zu viel:

$$\bar{N} = 1 \Leftrightarrow \rho = 1/2$$

D.h. wenn die Last ρ an der Paging-Station 50% ist, dann ist sie im Mittel immer am Arbeiten (→ guter Arbeitspunkt).

7.3 Einstellung der Fenstergröße h / 50%-Kriterium

Verbesserung wie bei Übergang von „ $L=S$ “ zu „ $L=1,3 \cdot S$ “:

Ankünfte (Seitenfehler) seien weiterhin zufällig (exp.vert. Abstände)

Aber: Bediendauer eines Seitenfehlers sei konstant = $1/\mu$

→ Übergang von M/M/1 zu M/D/1 (D: deterministische Bediendauer)

Für **M/D/1** gilt:

$$\bar{N} = \rho / (1 - \rho) - \rho^2 / 2(1 - \rho)$$

D.h.

$$\bar{N} = 1 \Leftrightarrow \rho = 2 - \sqrt{2} \approx 0,58 \approx 0,6$$

Also: Besserer Arbeitspunkt: Last der Paging-Station = 60%

7.3 Einstellung der Fenstergröße h / Probleme der Working-Set-Strategie

Problem:

Obwohl die Berechnung des Working-Sets sehr aufwendig ist, muss sie regelmäßig durchgeführt werden.

Vereinfachung:

Zuteilung der Rahmen pro Prozess nach „**Page-Fault-Frequency**“ (PFF), d.h. der Seitenfehlerhäufigkeit eines Prozesses:

Gib dem Prozess eine gewisse Anzahl an Seiten und

- gib ihm mehr, falls er zu viele Seitenfehler macht
- gib Seiten wieder frei, falls er zu wenige Seitenfehler macht

```
if (PFF > upperLimit) then
    if (Seiten frei) then
        Prozess erhält mehr Seiten
    else
        suspendiere Prozess

if (PFF < lowerLimit) then
    gib Seite frei und erhöhe ggf. Multiprogramminggrad
```

7.3 Einstellung der Fenstergröße h / Probleme der Working-Set-Strategie

Problem:

Die Größe des Working-Sets ist beschränkt.

→ Welche freie Seite ist zu ersetzen?

Mögliche Lösungen:

- Variable Working-Set-Strategie (V-WS, V-Random)

Wähle Seite beliebig aus den aktuell nicht in einem WS befindlichen Seiten

- Variable LRU-Strategie (V-LRU)

Seitenfehler nach LRU ersetzen, jedoch kann sich die Anzahl der zugeteilten Seiten erhöhen oder verringern.

Erfahrung:

Variable Strategien besser als feste Zuteilungen (z.B. Fixed OPT)

7.4 Optimale Strategie VOPT

Optimal bedeutet:

Strategie minimiert Kosten bei gegebener mittlerer Zahl zugeteilter Seiten.

Variable OPT (VOPT) benutzt ein Vorwärtsfenster der Größe h

$$VF(t,h) = \bigcup_{j=t+1}^{t+h} r_j$$

Strategie:

```
if (  $r_t \in VF(t,h)$  ) then
    halte  $r_t$ 
else
    verdränge  $r_t$ 
```

7.4 Optimale Strategie VOPT

Folgerungen:

1. VOPT ist i.A. schwer realisierbar
2. $VF(t,h) = RF(t+h,h)$
3. VOPT macht dieselben Seitenfehler wie WS.

ABER: WS hält überflüssige Seiten noch h Zeiteinheiten länger als VOPT

Satz:

VOPT macht für einen gegebenen Referenz-String ω dieselben Seitenfehler wie WS.

Beweis:

Seien r_t und r_{t+u} ($u \geq 1$) aufeinander folgende Zugriffe auf dieselbe Seite.

- Frage: Tritt ein Seitenfehler bei r_{t+u} auf?

Fallunterscheidung:

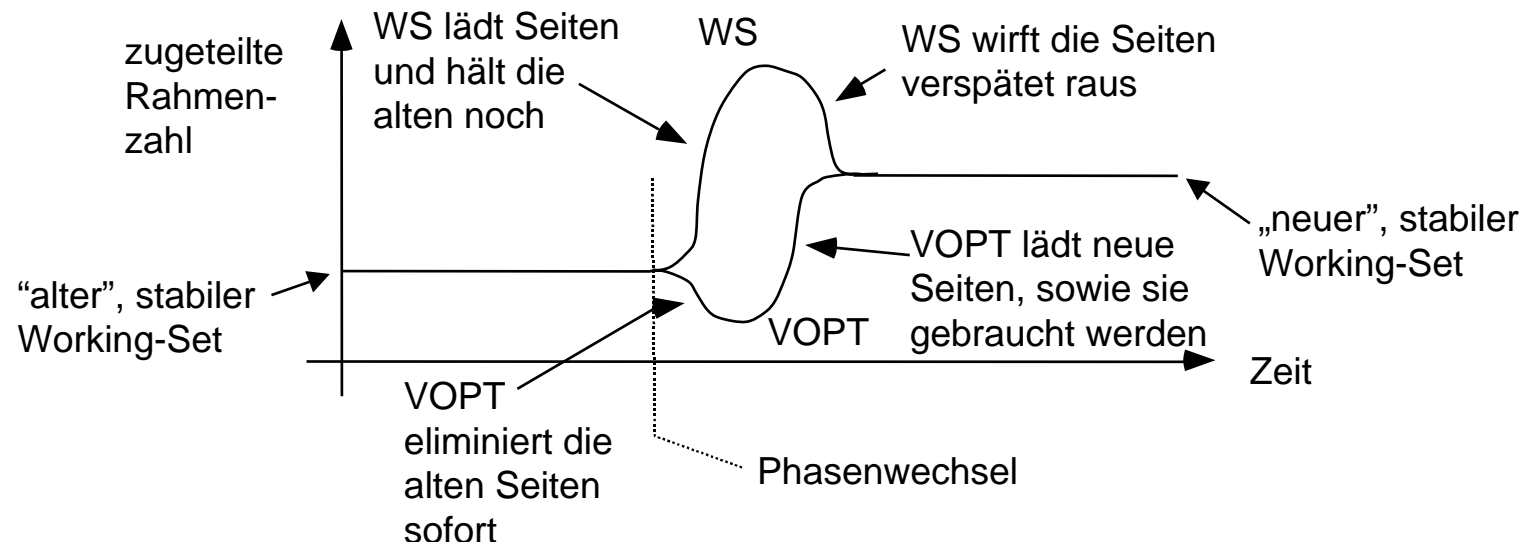
1. $u > h$:
 - VOPT wirft Seite zum Zeitpunkt $t+1$ raus, d.h. Seitenfehler zur Zeit $t+u$.
 - WS wirft Seite zum Zeitpunkt $t+h$ raus, d.h. Seitenfehler zur Zeit $t+u$.
2. $u \leq h$:
 - VOPT und WS halten Seite bis zum nächsten Zugriff, d.h. kein Seitenfehler. \square

7.4 Optimale Strategie VOPT / VOPT vs. WS

Unterschiede zwischen VOPT and WS:

Die mittlere zugeteilte Seitenzahl ist bei VOPT geringer als bei WS, weil VOPT unwichtige Seiten frühzeitig rauswirft!

Wird deutlich beim **Phasenwechsel** → Zeit, zu der ein Prozess seinen Kontext wechselt.



→ Zahl der Seitenfehler ist gleich.

→ WS hat einen höheren, d.h. „schlechteren“ Wert bzgl. „Zeit · Seitenzahl“.

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Neue Kostendefinition:

Kosten = Seitenfehlerkosten + Seitenhaltekosten

Seitenfehler kosten Zeit; das Halten einer Seite im Speicher kostet Betriebsmittel, da es z.B. verhindert, dass diese Seite von einem anderen Prozess genutzt wird.

Satz:

Falls die Fenstergröße $h = \frac{R}{U}$ gewählt wird, ist VOPT der optimale Paging-Algorithmus bezüglich dieser Kostenfunktion.

R: Kosten für die Bearbeitung eines Seitenfehlers

U: Kosten für das Halten einer Seite pro Zeiteinheit

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Beweis:

Sei $\omega = r_1 r_2 \dots r_t$ ein beliebiger Referenz-String, der insgesamt M verschiedene Seiten beinhaltet.

Die Kosten $C(\omega)$ dieses Referenz-Strings ergeben sich zu:

$$C(\omega) = M \cdot R + \sum_{i=1}^t c_i$$

Kosten, die durch den
Erstzugriff entstehen

Kosten, die nach Zugriff r_i
bis zum nächsten Zugriff
auf diese Seite entstehen

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Berechnung von c_i :

Seien r_i und r_j zwei aufeinanderfolgende Zugriffe auf dieselbe Seite.

Definiert man $x_i := j - i$, so ergeben sich zwei Fälle:

a) r_i verbleibt im gesamten Zeitraum im Speicher

$$a \quad \Rightarrow \quad c_i = x_i \cdot U$$

b) r_i wird zum Zeitpunkt $i+z_i$ ($0 \leq z_i < x_i$) rausgeworfen und zum Zeitpunkt j wieder geladen

$$\Rightarrow c_i = z_i \cdot U + R$$

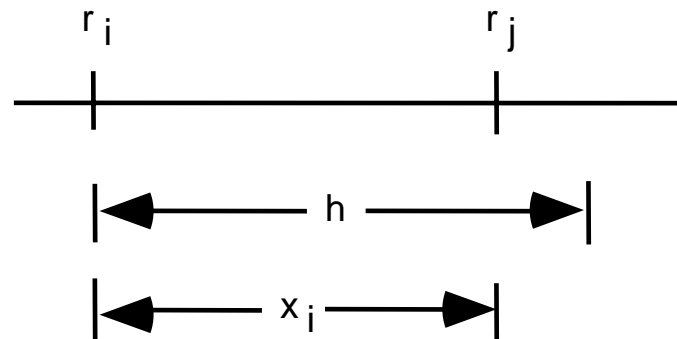
R: Kosten für die Bearbeitung eines Seitenfehlers

U: Kosten für das Halten einer Seite pro Zeiteinheit

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Fall a) Durchgängiges Halten

Damit VOPT die Seite im Speicher behält muss $h = \frac{R}{U} \geq x_i$ gelten.



Die Kosten von VOPT betragen in diesem Fall

$$\Rightarrow c_i(\text{VOPT}) = U \cdot x_i$$

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Vergleicht man VOPT mit einer beliebigen anderen Strategie A, so gilt:

Fall a1) A hält die Seite ebenfalls:

$$c_i(A) = U \cdot x_i = c_i(\text{VOPT})$$

\Rightarrow A ist nicht besser als VOPT

Fall a2) A wirft die Seite raus und lädt sie wieder ein

$$c_i(A) \geq R \text{ und wegen } h = \frac{R}{U} \geq x_i \text{ wäre dann } R \geq U \cdot x_i$$

$$c_i(A) \geq R \geq U \cdot x_i = c_i(\text{VOPT})$$

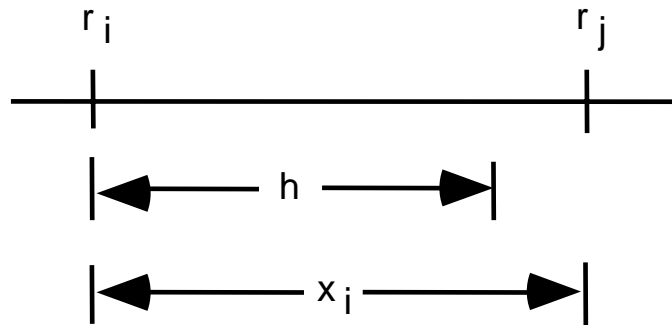
\Rightarrow VOPT verursacht höchstens genauso hohe Kosten.

\rightarrow VOPT ist im Fall a) mindestens genauso gut wie jede andere Strategie A.

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Fall b) Rauswerfen und wieder laden

Damit VOPT eine Seite frei gibt muss $h = \frac{R}{U} < x_i$ gelten.



VOPT wirft r_i (sofort) raus und lädt sie erst bei Bedarf wieder nach

$$\Rightarrow c_i(\text{VOPT}) = U \cdot 0 + R = R$$

7.4 Optimale Strategie VOPT / Kostenbetrachtung

Vergleicht man VOPT mit einer beliebigen anderen Strategie A, dann ergibt sich:

Fall b1) A wirft die Seite raus und lädt sie wieder ein

$$c_i(A) \geq R = c_i(\text{VOPT})$$

\Rightarrow A ist nicht besser als VOPT

Fall b2) A hält die Seite

$$c_i(A) = U \cdot x_i$$

wegen $h = \frac{R}{U} < x_i$ gilt dann aber

$$c_i(A) = U \cdot x_i > R = c_i(\text{VOPT})$$

\Rightarrow VOPT ist hier besser

\rightarrow VOPT ist im Fall b) mindestens genauso gut wie jede andere Strategie A. \square

Inhaltsverzeichnis

8.1 Allgemeines zum Datei-Konzept

8.2 Verzeichnisstruktur

- Single-Level-Verzeichnis
- Two-Level-Verzeichnis
- Verzeichnisbäume
- Verzeichnisse mit azyklischen und allgemeinen Graphen

8.3 Implementierung von Dateisystemen

8.4 Belegungsstrategien

- Zusammenhängende Belegung
- Verkettete Belegung
- Indizierte Belegung

8.5 Speicherplatzverwaltung

8.6 Implementierung von Verzeichnissen

8.1 Allgemeines zum Datei-Konzept / Motivation

Prozesse arbeiten auf dem Hauptspeicher:

- Direkter Zugriff auf die Daten im Hauptspeicher
- Größe vom Hauptspeicher begrenzt
- Daten sind verloren, sobald der Prozess beendet wird

➔ Daten müssen permanent gespeichert werden

- Hintergrundspeicher mit sehr großer Kapazität

Betriebssystem als auch Benutzer besitzen oft hunderte von Dateien

➔ Flexible Organisation und effizienter Zugriff auf die Dateien nötig

➔ Bei Multiuser Systemen müssen Dateien vor dem Zugriff Dritter geschützt werden

Hintergrundspeicher:

- Magnetband (lesend/schreibend)
- Festplatte (lesend/schreibend)
- CD-ROM (lesend), CD-RW (lesend/schreibend)
- DVD

8.1 Allgemeines zum Datei-Konzept

Datei

Eine mit Namen versehene Sammlung zusammengehöriger Informationen, die auf einem Hintergrundspeicher liegt.

➔ Quellcode, Objektprogramme, Texte, Bilder, Audio, Video, ...

Dateistruktur ist abhängig vom Typ

- Textdatei: Sequenz von Zeichen, organisiert als Zeilen und Absätze
- Quellcode: Menge von Funktionen mit eigener Struktur
- Bitmap: Menge von bits
- Binärdatei: Sequenz von Bytes
- Programmdatei: Menge von Code-Abschnitten

8.1 Allgemeines zum Datei-Konzept

Dateiattribute

- Dateiname: Für Menschen lesbare Bezeichnung der Datei.
 - Länge und erlaubte Zeichen variieren oft
 - Windows 2000: Maximal 215 Zeichen, nicht erlaubte Zeichen: \ / : * ? " < > |
 - Unix: Maximal 256 Zeichen
- Datei-Identifikator: Für Menschen nicht-lesbare Bezeichnung der Datei
- Typinformationen, z.B. **program.c**, **filesys.doc**, **brief.txt**, **bild.gif**
 - Einige Systeme unterscheiden zwischen Dateitypen andere nicht
- Größe
- Position im Dateisystem bzw. Hintergrundspeicher
- Zugriffsrechte: Wer darf was mit der Datei machen?
- Uhrzeit, Datum der Dateierstellung bzw. -änderung
- Benutzeridentitäten

8.1 Allgemeines zum Datei-Konzept

Basisoperationen auf Dateien (betriebssystemabhängig)

- Erzeugen von Dateien
 - Finde genügend freien Speicher und informiere das System
- Schreiben auf eine Datei
 - Finde Datei im Verzeichnis und verwalte Schreibposition
- Lesen von einer Datei
 - Finde Datei im Verzeichnis und verwalte Leseposition
- Löschen einer Datei
 - Finde Datei im Verzeichnis, gib belegten Speicher frei und lösche Eintrag aus Verzeichnis
- Umpositionierung von Zeigern

Höhere Operationen auf Dateien: können durch Basisoperationen durchgeführt werden

- Umbenennen von Dateien
- Kopieren von Dateien
- Anfügen an eine Datei

8.1 Allgemeines zum Datei-Konzept

Operationen erfordern das Auffinden einer Datei im Verzeichnis

- Open-File Table (**OFT**): enthält die Liste aller geöffneten Dateien
- open: fügt Datei in OFT ein
- close: entfernt Datei aus OFT
- Bei Multiuser/Multiprozess Systemen existiert eine lokale OFT und eine globale OFT

Informationen zu geöffneten Dateien im OFT

- Zeiger auf aktuelle Position in der Datei (pro Prozess)
- Zugreifende Prozesse Zähler: Datei wird aus OFT erst entfernt, wenn dieser Zähler Null ist
- Position der Datei auf Hintergrundspeicher
- Zugriffsrechte (im lokalen OFT, pro Prozess)
- Informationen zu gesperrten Bereichen

8.1 Allgemeines zum Datei-Konzept / Design Fragen

Dateitypen

- Betriebssystem kennt definierte Dateitypen
 - Bessere Unterstützung der Benutzer
 - Einschränkung der Benutzer auf die definierten Typen

Dateistruktur

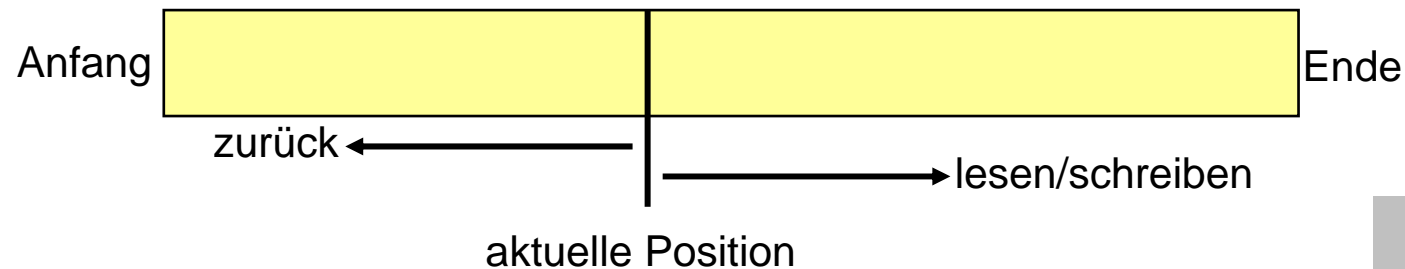
- Besteht eine Datei aus einer Aneinanderreihung von Bytes oder gibt es eine Strukturierung
- Zu wenig Strukturen: Programmierung umständlich
- Zu viele Strukturen: Aufwändige Programmierung

8.1 Allgemeines zum Datei-Konzept / Zugriffsarten

Zugriffarten

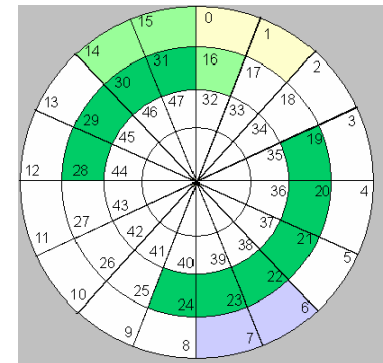
➤ Sequenzieller Zugriff

- Bandmodell
- `reset`, `read next`, `write next`, `skip(n)`



➤ Direkter Zugriff (Direct Access)

- Plattenmodell
- Beliebiger Zugriff auf erforderliche Daten
- Andere Zugriffsarten können einfach durch DA simuliert werden



Zugriff ggf. nur auf vorher geöffnete Dateien (**open-file-table**)

8.2 Verzeichnisstruktur / Organisation

Auf einem Dateisystem sind oft Millionen von Dateien

→ Organisation der Dateien durch:

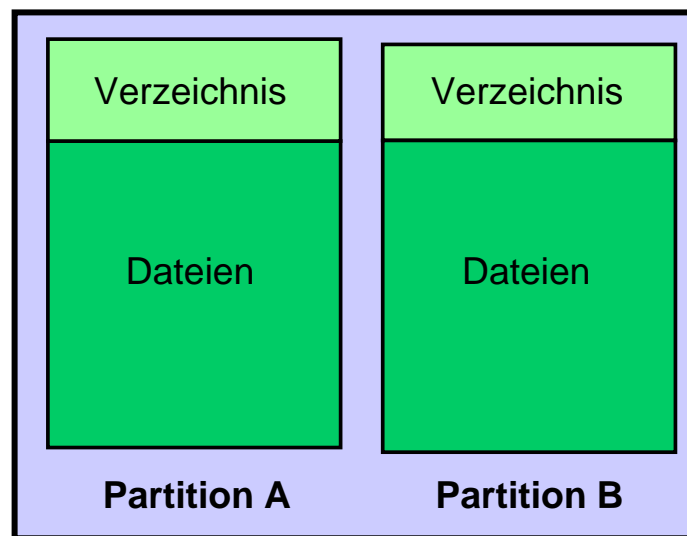
- Partitionen
- Verzeichnisse

1. Möglichkeit → eine Festplatte mit mehreren Partitionen

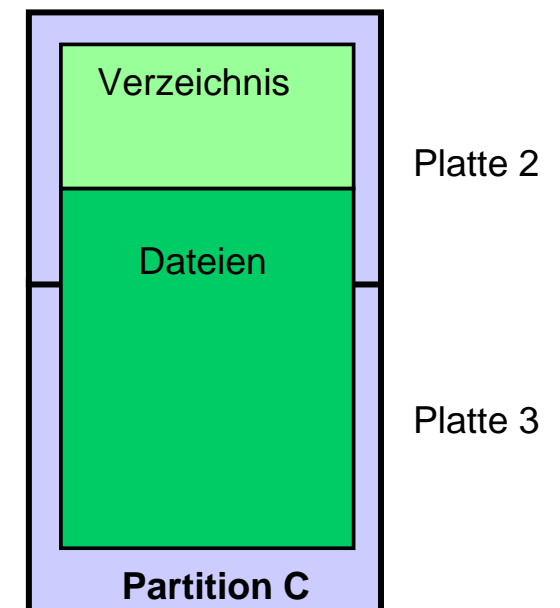
2. Möglichkeit → eine Partition über mehrere Festplatten

Partition

- Virtuelle Platte
- Eine Platte mit mehreren **Partitionen**
- Eine **Partition** über mehrere Platten



Platte 1



8.2 Verzeichnisstruktur

Verzeichnis (Directory)

- Ein Verzeichnis umfasst die Namen und die übrigen Attribute aller seiner Dateien.

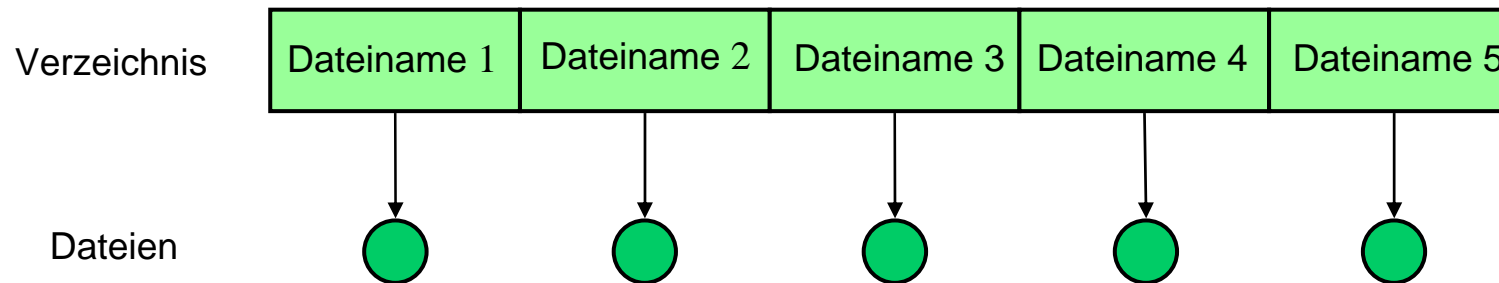
Operationen auf Verzeichnissen

- Suchen nach einer Datei
- Erzeugen einer Datei
- Löschen einer Datei
- Umbenennen einer Datei
- Auflisten von Dateien
- Durchlaufen von Verzeichnissen oder des gesamten Verzeichnisses

➔ Aufbau des Verzeichnisses beeinflusst die Effizienz des Dateisystems

8.2 Verzeichnisstruktur / Single-Level-Verzeichnis

Alle Dateien in **einem** Verzeichnis

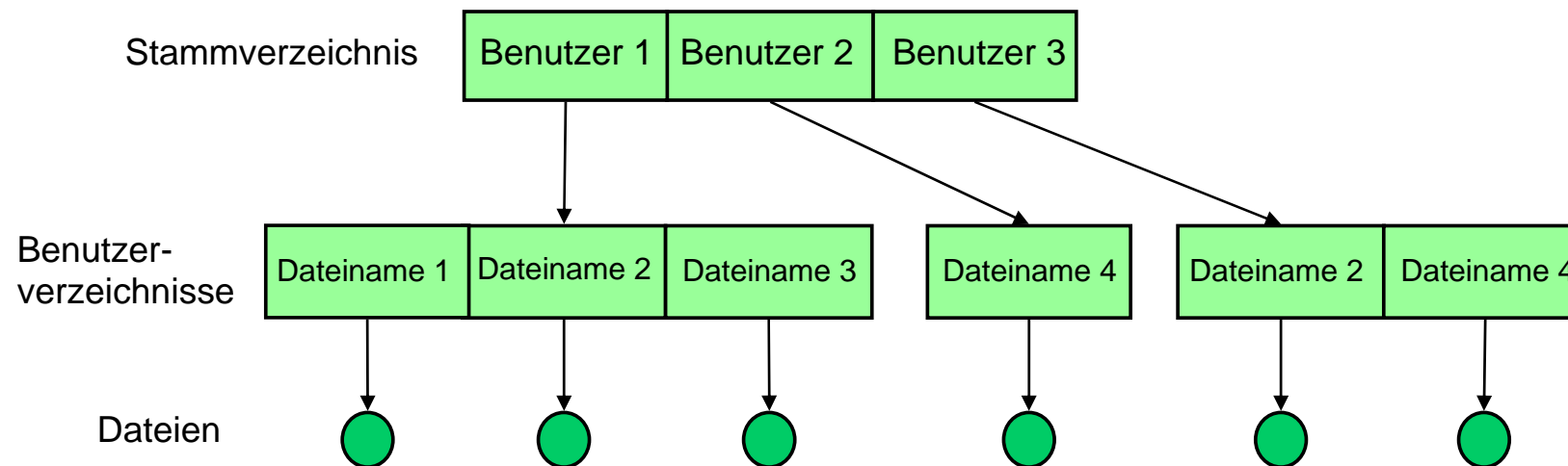


Eigenschaften:

- +einfache Implementierung
- unübersichtlich bei mehreren Benutzern oder vielen Dateien
- alle Dateien müssen unterschiedliche Namen besitzen
- keine Organisationsmöglichkeit

8.2 Verzeichnisstruktur / Two-Level-Verzeichnis

Verzeichnis mit **zwei Ebenen**, wobei jeder Benutzer ein eigenes Verzeichnis besitzt.

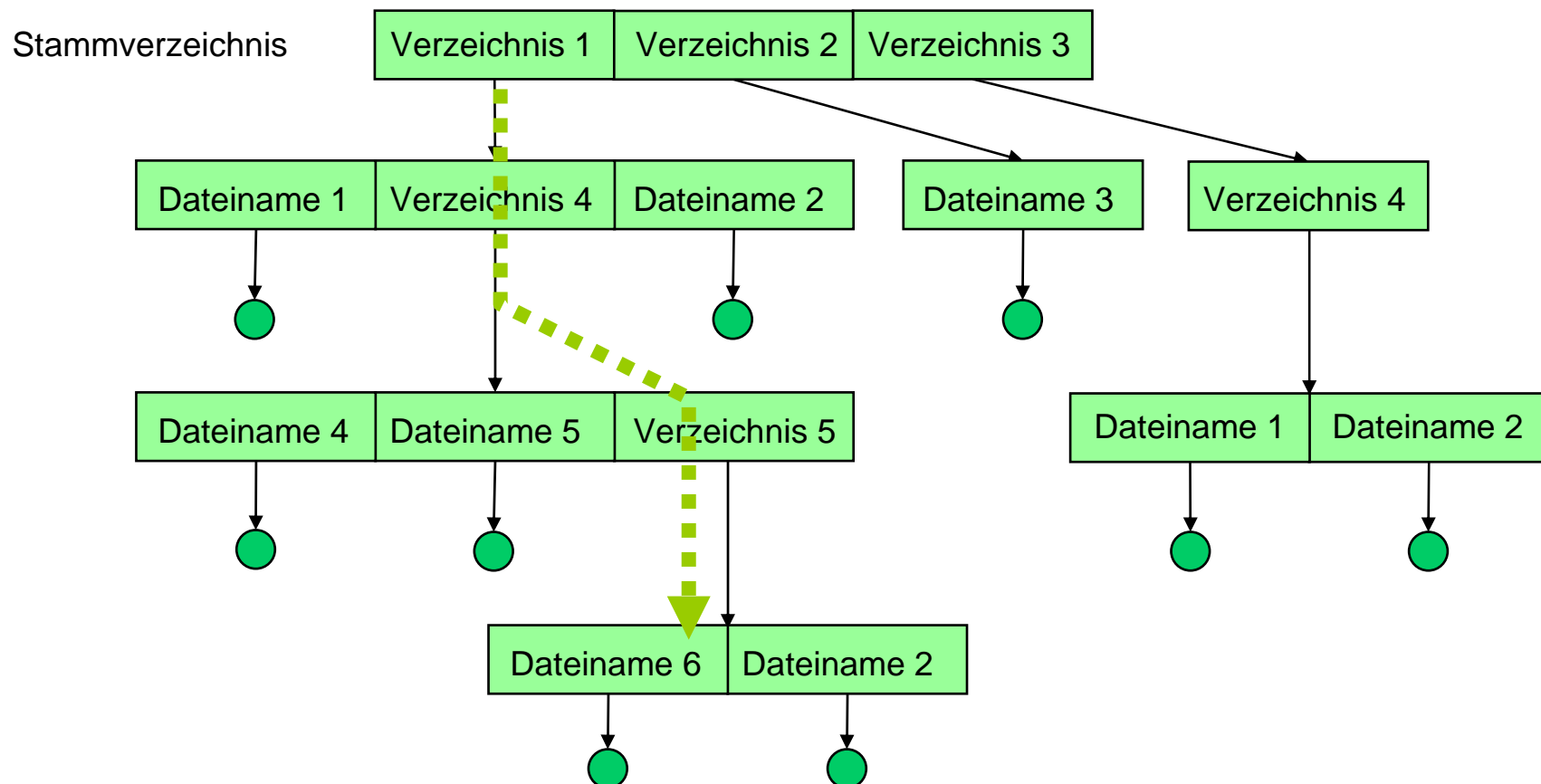


Eigenschaften

- + verschiedene Benutzer können Dateien unter gleichem Namen abspeichern
- + Zugriff auf eigene Dateien über Dateiname,
- + Zugriff auf fremde Dateien über Pfad → Benutzer- und Dateiname, z.B.:
`\Benutzer 3\Dateiname 2`
- Zugriff schwer, wenn Benutzer zusammen arbeiten wollen
- Zugriff auf Systemdateien? → Suchpfad, z.B. durch Variable **PATH**

Verallgemeinerung: **Verzeichnisbäume** mit beliebigen Unterverzeichnissen.

Dateiname entspricht dem eindeutigen Pfad vom Wurzelverzeichnis (root-directory) durch alle Unterverzeichnisse bis zur eigentlichen Datei.



8.2 Verzeichnisstruktur / Verzeichnisbäume

Eigenschaften von Verzeichnisbäumen

- Einfacher Dateiname wird nur im aktuellen Verzeichnis gesucht.
- Datei außerhalb des aktuellen Verzeichnisses muss über Pfad angegeben werden.
- Absoluter Pfad: Von der Wurzel bis zur Datei
- Relativer Pfad: Von der aktuellen Position zur Datei

Beispiel:

- aktuelles Verzeichnis: Verzeichnis 4
- absoluter Pfad: \Verzeichnis1\Verzeichnis4\Verzeichnis5\Dateiname6
- relativer Pfad: Verzeichnis5\Dateiname6

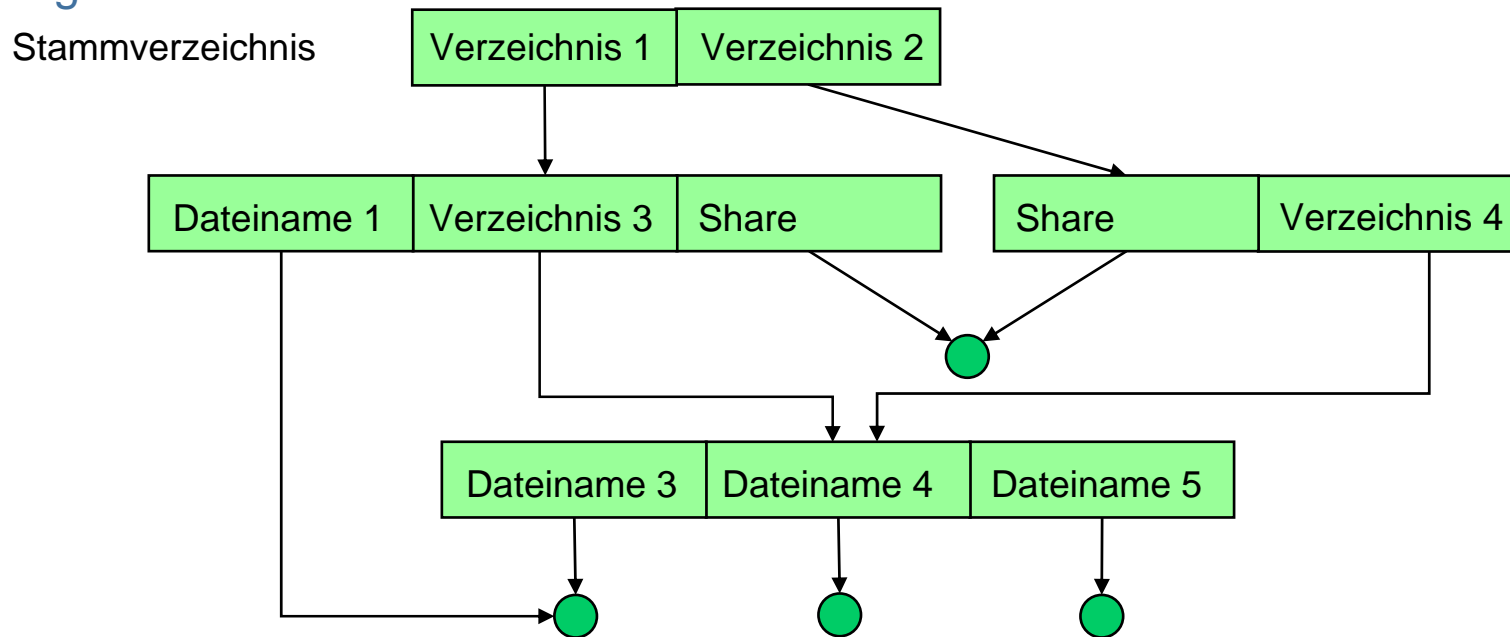
Löschen eines Verzeichnisses

- entweder erst leeren, dann löschen → erheblicher Aufwand
- oder mit allen Dateien und Unterverzeichnissen löschen → gefährlich

8.2 Verzeichnisstruktur / Verzeichnisse mit azyklischen und allgemeinen Graphen

Unterverzeichnis oder Datei erscheint im Dateisystem an zwei oder mehr Stellen, ist jedoch physikalisch nur einmal vorhanden.

→ Änderungen sind an allen Stellen sofort ersichtlich.



+ flexibel

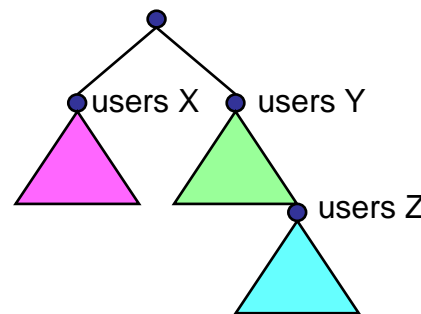
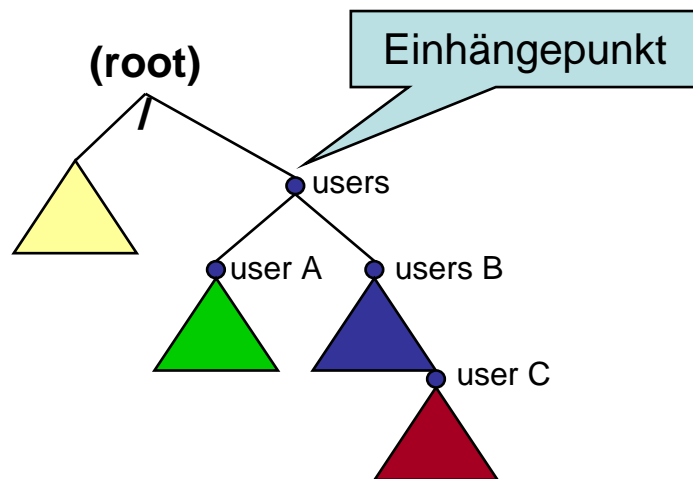
- Löschen von Dateien oder Unterverzeichnissen kompliziert
 - Was passiert mit Link nach dem Löschen des Ziels?
- Freiheit von Zyklen sicherstellen

8.2 Verzeichnisstruktur / Einhängen von Dateisystemen

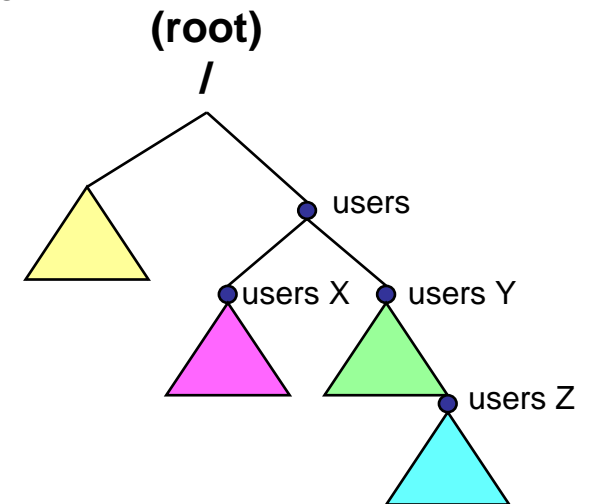
Dateisystem muss vor der Verwendung beim System angemeldet werden →
Einhängen eines Dateisystems (mount)

Ein Dateisystem wird an ein Einhängepunkt eingehängt

- Spezielle Einhängepunkte oder beliebig im Verzeichnis
- Einhängen an leere/volle Verzeichnisse
- Mehrmaliges Einhängen des gleichen Dateisystems an unterschiedliche Stellen
- Zeit des Einhängens: beim Hochfahren des Systems, zu beliebigen Zeiten



Ungemountetes
Dateisystem



Dateisystem nach
dem Einhängen

8.3 Implementierung von Dateisystemen

Dateisysteme werden auf Hintergrundspeicher implementiert

- Gebräuchlichster Hintergrundspeicher: Festplatte
 - Lesen und Schreiben auf die gleiche Position
 - Direkter Zugriff auf beliebige Daten

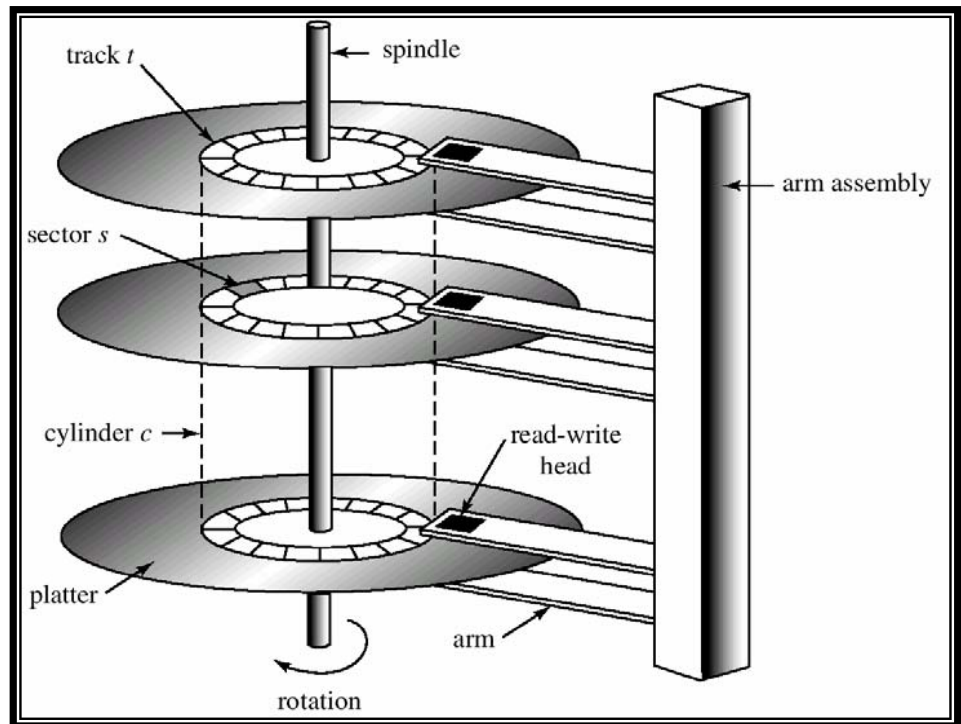
Fragen bei der Implementierung von Dateisystemen

- Strukturierung von Dateien
- Verwaltung des Speicherplatzes
- Implementierung von Verzeichnissen

8.3 Implementierung von Dateisystemen / Festplatten

Festplatte:

- Daten werden magnetisch gespeichert
- Eine Festplatte besteht aus Scheiben und Lese/Schreibköpfen (beidseitig)
- Scheiben bestehen aus Spuren (*Tracks*), die in Sektoren aufgeteilt sind
- Zylinder: Menge aller Spuren, die bei einer bestimmten Position der L/S-köpfe erreicht werden
- Zugriff wird durch Gerätekontroller (*disk controller*) gesteuert
- Um den Durchsatz zu erhöhen, werden die Daten Blockweise übertragen. Ein Block = Mehrere Sektoren
- Aktuelle Festplatten haben Kapazitäten von mehreren Gigabyte



$$\begin{aligned}
 1 \text{ GByte} &= 1000 \text{ MByte} \\
 &= 1000\,000 \text{ kByte} \\
 &= 1000\,000\,000 \text{ Byte} \\
 &= 8000\,000\,000 \text{ bit}
 \end{aligned}$$

8.3 Implementierung von Dateisystemen

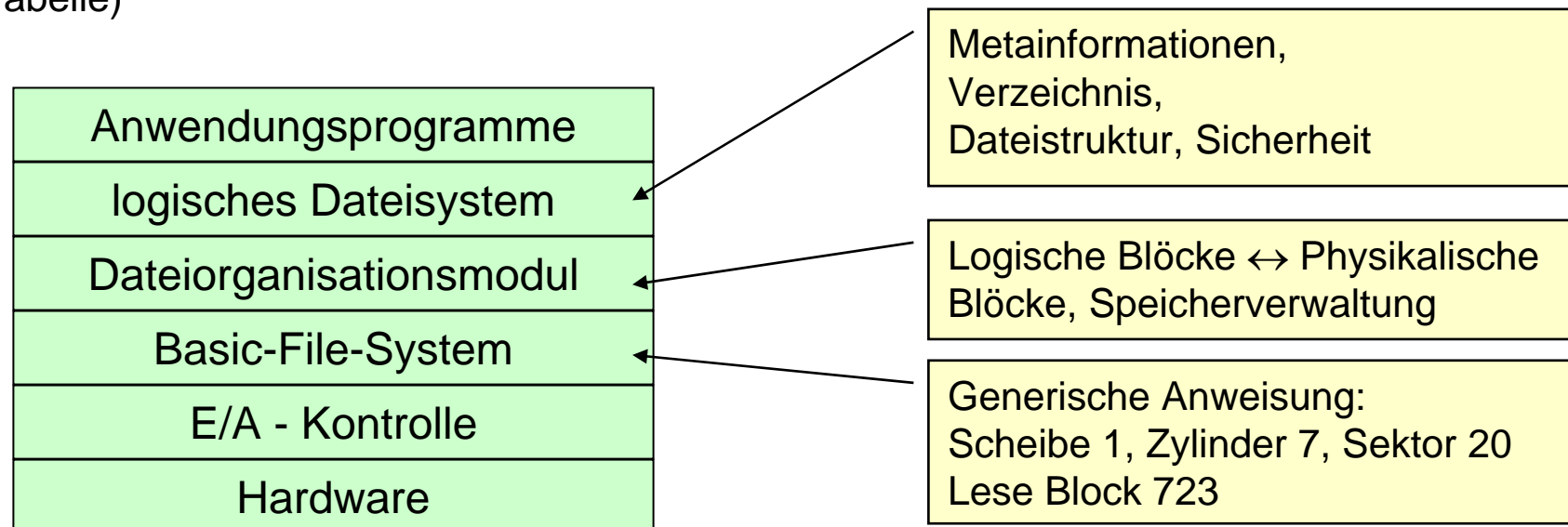
FAT-Dateisystem (File-Allocation-Table)

FAT16 (DOS, Windows 95/ NT)		
Partitionsgröße	Clustergröße	Sektoren /Cluster
16-127 MB	2 KB	4
128-255 MB	4 KB	8
256-511 MB	8 KB	16
512-1023 MB	16 KB	32
1024-2047 MB	32 KB	64
2048-4096 MB	64 KB	128 nur Win NT
FAT 32 (DOS, Windows 98, 98 SE, 2000)		
Partitionsgröße	Clustergröße	Sektoren /Cluster
256 MB – 8,01 GB	4 KB	8
8,02 GB – 16,02 GB	8 KB	16
16,03 GB – 32,04 GB	16 KB	32
> 32,04 GB	32 KB	64

8.3 Implementierung von Dateisystemen

Um Komplexität zu verringern sind Dateisysteme als Schichten realisiert

- **E/A-Kontrolle:** Treiber für Dateiübertragung, Hauptspeicher ↔ Plattensystem
- **Basic-File-System:** veranlasst Treiber, physikalische Blöcke auf Festplatte zu schreiben bzw. von dort zu lesen
- **Dateiorganisationsmodul:** Abbildung logische ↔ physikalische Blockadressen
- **Logisches Dateisystem:** erledigt Neueintragung/Löschung einer Datei sowie E/A-Zugriffe auf eine Datei und versorgt das Dateiorganisationsmodul mit Informationen zu einem (symbolischen) Dateinamen unter Verwendung der Verzeichnisstruktur (oft: Open-File-Tabelle)



8.3 Implementierung von Dateisystemen / VFS

Wozu ein Virtuelles Dateisystem (Virtual File System)?

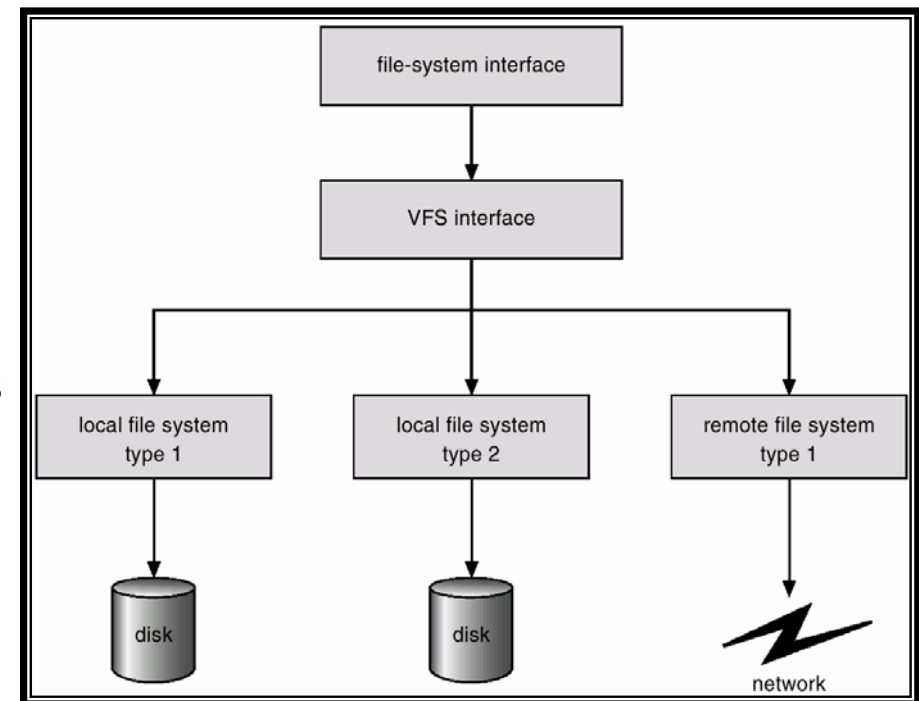
- Moderne Rechnersysteme unterstützen mehrere Dateisysteme, z.B. CD-ROM, DVD, FAT16, FAT32, NTFS, ext2fs, HPFS
- Wie können unterschiedliche Dateisysteme in einen Verzeichnis integriert werden?
- Wie können Benutzer zwischen unterschiedlichen Dateisystemen wechseln und navigieren?

Lösung 1:

- Implementierung aller Zugriffsroutinen für jedes Dateisystem

Lösung 2:

- Virtuelles Dateisystem, welches dem Benutzer eine transparente Sicht erlaubt



8.3 Implementierung von Dateisystemen

Belegungsstrategien:

- Wie soll der Plattenspeicher auf die Dateien aufgeteilt werden?

Ziele:

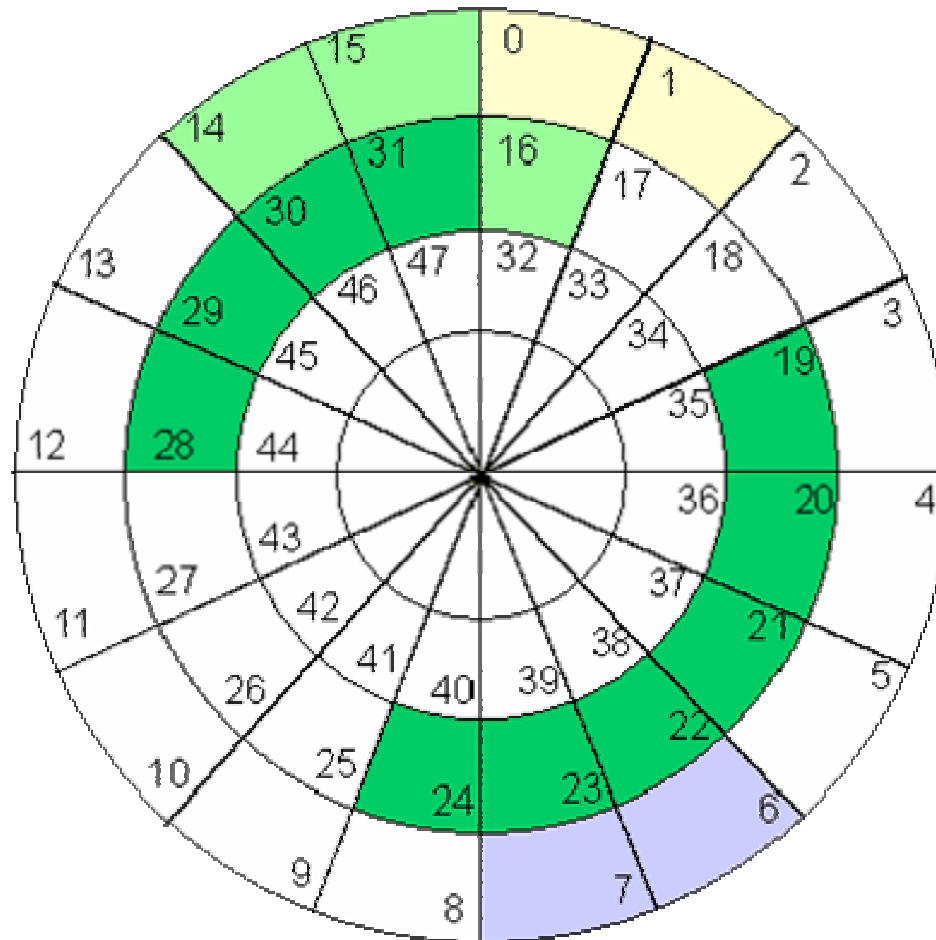
- schneller Zugriff auf Dateien
- effiziente Nutzung des Plattenspeichers

Strategien:

- zusammenhängende Belegung
- verkettete Belegung
- indizierte Belegung

8.4 Belegungsstrategien / Zusammenhängende Belegung

Eine Datei belegt eine Reihe zusammenhängender Blöcke auf der Festplatte



Verzeichnis

Datei	Start	Länge
Name1	0	2
Name2	14	3
Name3	19	6
Name4	28	4
Name5	6	2

Ähnlich wie „Segmentierung“ mit
allen Vor- und Nachteilen

8.4 Belegungsstrategien / Zusammenhängende Belegung

Vorteile:

- + Sequenzieller und direkter Zugriff auf Datei
 - sequenziell: zuletzt referenzierte Blockadresse + 1
 - direkt: Startblock der Datei ist b (physikalisch), physikalische Adresse des logischen Blocks i der Datei ist $b + i$, $i = 0, 1, \dots$

Nachteile:

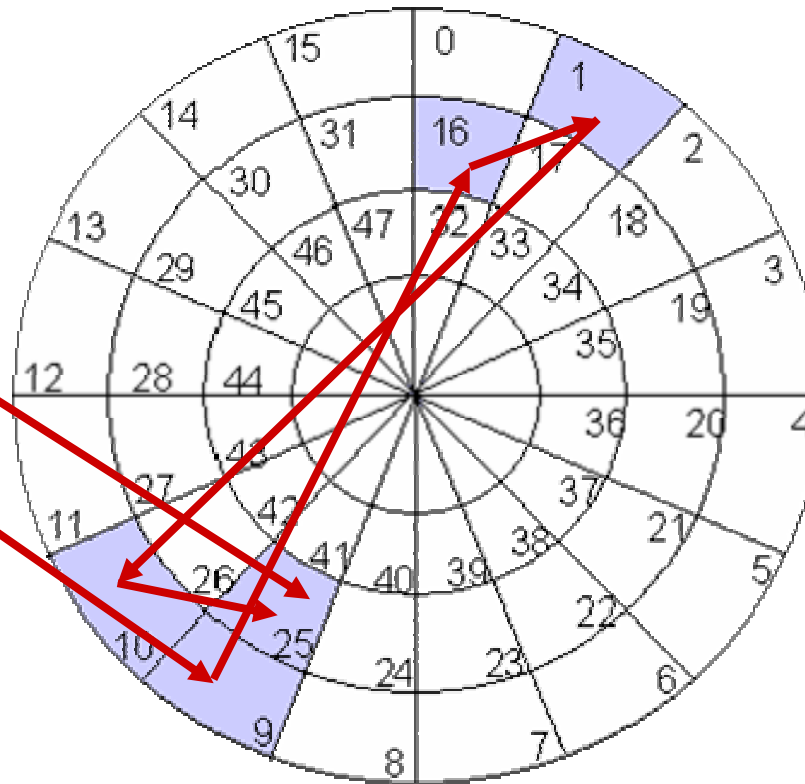
- Speicherplatz für neue Dateien finden: **first-fit, best-fit, rotating-first-fit**
freier Speicherplatz wird allmählich in kleine Stücke zerlegt
⇒ **externe Fragmentierung**
- Größe einer Datei muss bei Erzeugung angegeben werden
 - falls nicht bekannt, möglicherweise deutlich überschätzt
 - falls bekannt, aber Datei über lange Zeit anwächst, bleibt viel Speicherplatz lange ungenutzt ⇒ **interne Fragmentierung**
- **Abhilfe:**
 - Umkopieren (mit Geschwindigkeitsverlust), Garbage-Collection (Defragmentierung)

8.4 Belegungsstrategien / Verkettete Belegung

Eine Datei ist als verkettete Liste von Blöcken realisiert.

Ein Verzeichniseintrag enthält Zeiger auf den ersten und letzten Block einer Datei und deren Länge. Jeder Block enthält einen Zeiger auf den nächsten Block der Datei.

Verzeichnis			
Datei	Start	Ende	Länge
Name1	9	25	5



8.4 Belegungsstrategien / Verkettete Belegung

Vorteile

- Keine Fragmentierung, da Zeiger verwendet werden
- Größe einer Datei muss im Voraus nicht bekannt sein

Nachteile

1. Nur effizient für sequenziellen Zugriff
2. Zugriff auf Zeiger kostet zusätzliche Zeit
3. Zeiger verbrauchen zusätzlichen Speicher
4. Was machen, wenn ein Block kaputt ist?

Lösungen:

Für 1., 2. und 3. Lesen von Clustern, Cluster = Mehrere Blöcke

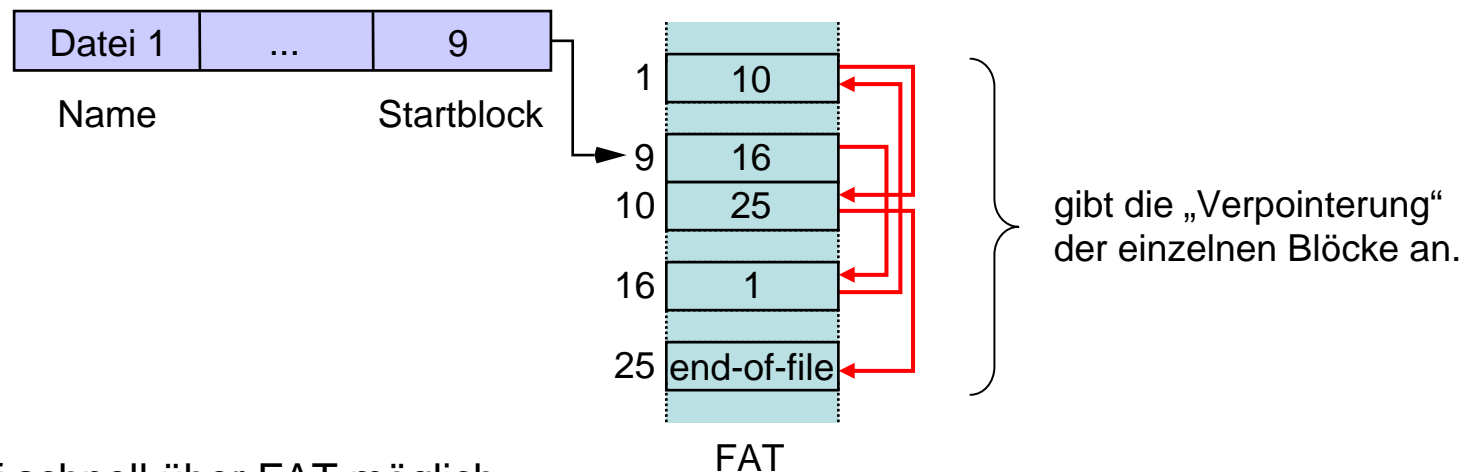
Für 4. Verwendung von doppelt-verketteten Listen

8.4 Belegungsstrategien / Verkettete Belegung

Variante: **FAT** (File-Allocation-Table, Dateibelegungstabelle)

- Am Anfang jeder Partition ist eine Tabelle mit je einem Feld für jeden Block.
- Im Verzeichnis steht die Nummer m des ersten physikalischen Blocks der Datei.
- An m -ter Stelle der Tabelle steht die Nummer des folgenden Blocks.
- Verkettung bis zum letzten Block der Datei mit speziellem EOF-Wert in der Tabelle.
- Freie Blöcke in der Tabelle sind mit einer 0 gekennzeichnet.

Verzeichniseintrag:

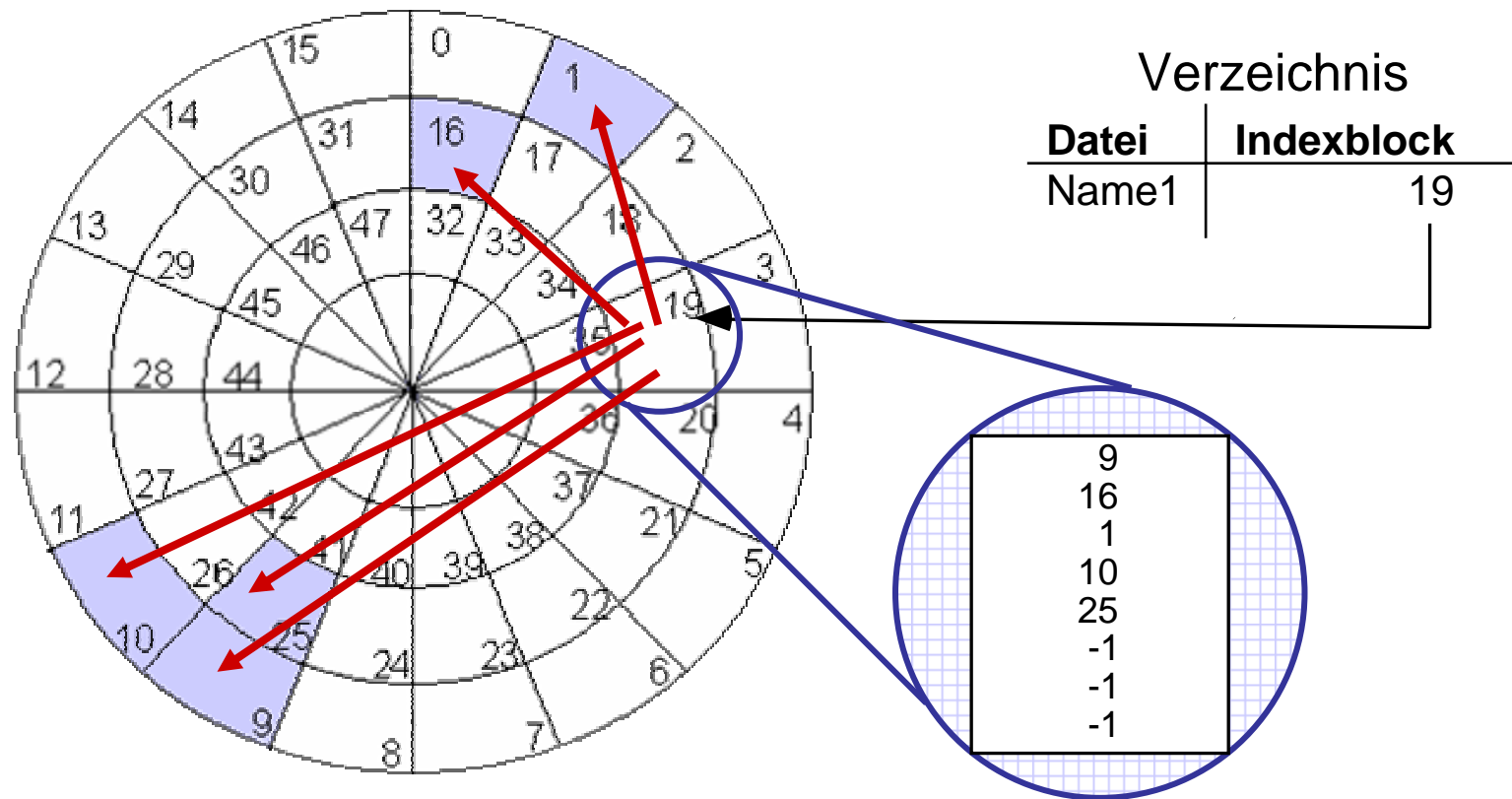


- + Direkter Zugriff schnell über FAT möglich
- Erhöhter Aufwand durch ständiges Springen der Schreib-/Lese Köpfe zwischen FAT und Blöcken
- Verlust der Tabelle

8.4 Belegungsstrategien / Indizierte Belegung

Ein **Indexblock** enthält alle Zeiger einer Datei → Zeiger = Blockadresse

Der Verzeichniseintrag für eine Datei enthält nur noch die Adresse des Indexblocks; die Adresse des i -ten Datenblocks der Datei steht an der i -ten Stelle des Indexblocks.



8.4 Belegungsstrategien / Indizierte Belegung

Vorteil:

- + direkter Zugriff ohne externe Fragmentierung

Nachteil:

- Indexblock belegt Speicherplatz

„Richtige“ Größe des Indexblocks?

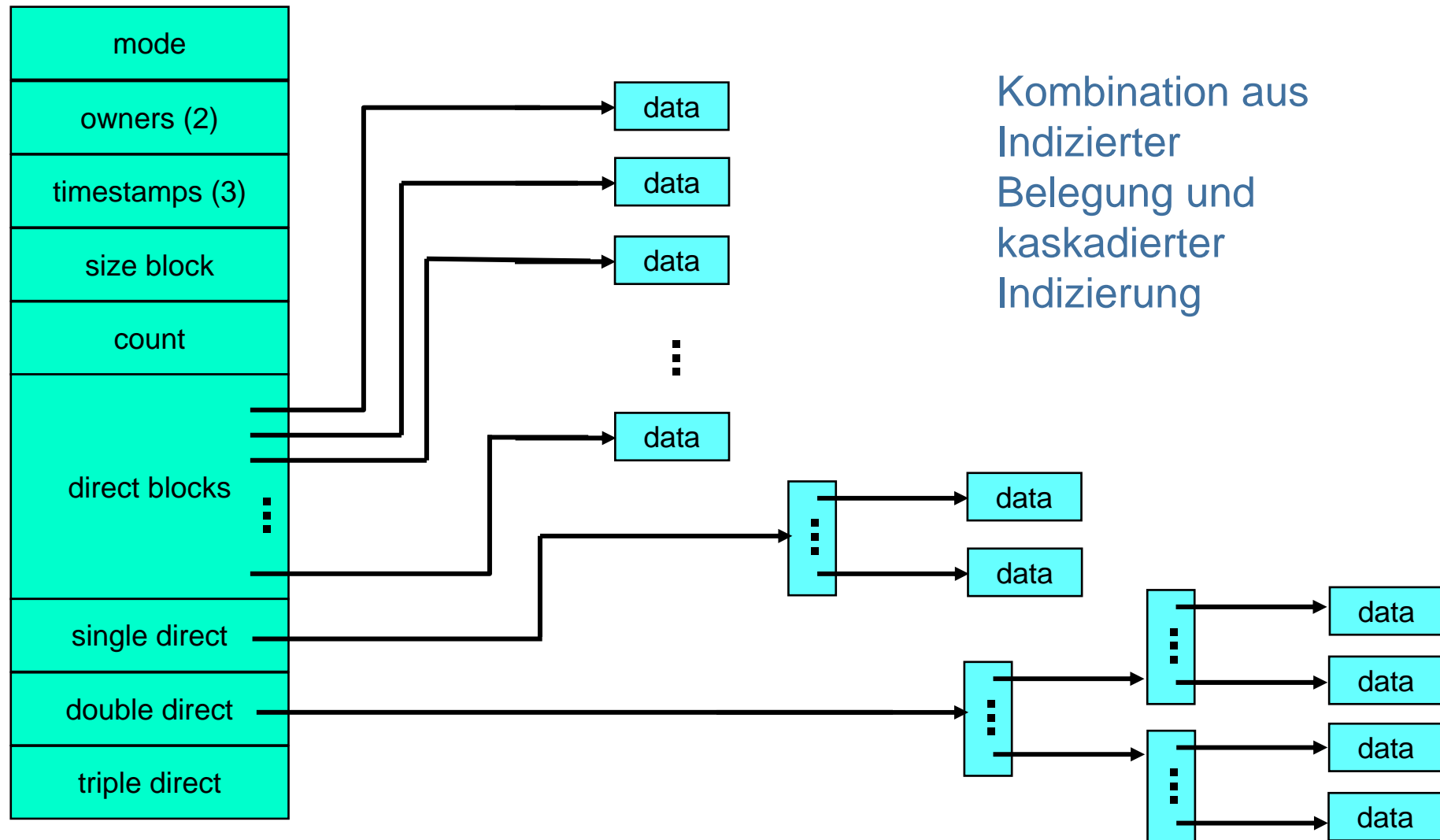
Zu groß \Rightarrow Speicherplatz wird verschwendet

Zu klein \Rightarrow Speicherplatz des Indexblocks reicht nicht für alle Zeiger

Große Dateien

- Verkettung von Indexblöcken
- Hierarchie von Indexblöcken \rightarrow Indexblock mit Zeiger auf Indexblöcke

8.4 Belegungsstrategien / Indizierte Belegung UNIX inode



8.5 Speicherplatzverwaltung

Die sogenannte **free-space-list** ist eine Tabelle, in der alle nicht belegten Speicherblöcke verzeichnet sind. Sie wird beim Erzeugen und Löschen einer Datei modifiziert.

Mögliche Implementierungen:

- Liste
- Bit-Vektor → ein Bit pro physikalischem Block
- Verkettete Liste
- Gruppierung freier Blöcke

Im ersten freien Block sind n Adressen, davon $n-1$ Zeiger auf freie Blöcke und ein Zeiger auf die nächste Blockadresse mit wiederum n Adressen.

→ Vorteil:

- + Adressen vieler freier Blöcke können schnell gefunden werden.
- + Bei vielen aufeinanderfolgenden freien Blöcken muss nur die Adresse des ersten sowie die Anzahl abgespeichert zu werden.

8.6 Implementierung von Verzeichnissen

Lineare Liste

- Neu erstellte Dateien hinten anhängen
- Löschen einer Datei:
 - als „frei“ markieren
 - oder
 - Eintrag in Liste freier Verzeichnisplätze
 - oder
 - letzten Verzeichniseintrag an die freigewordene Stelle verschieben

Vorteil:

- + einfach

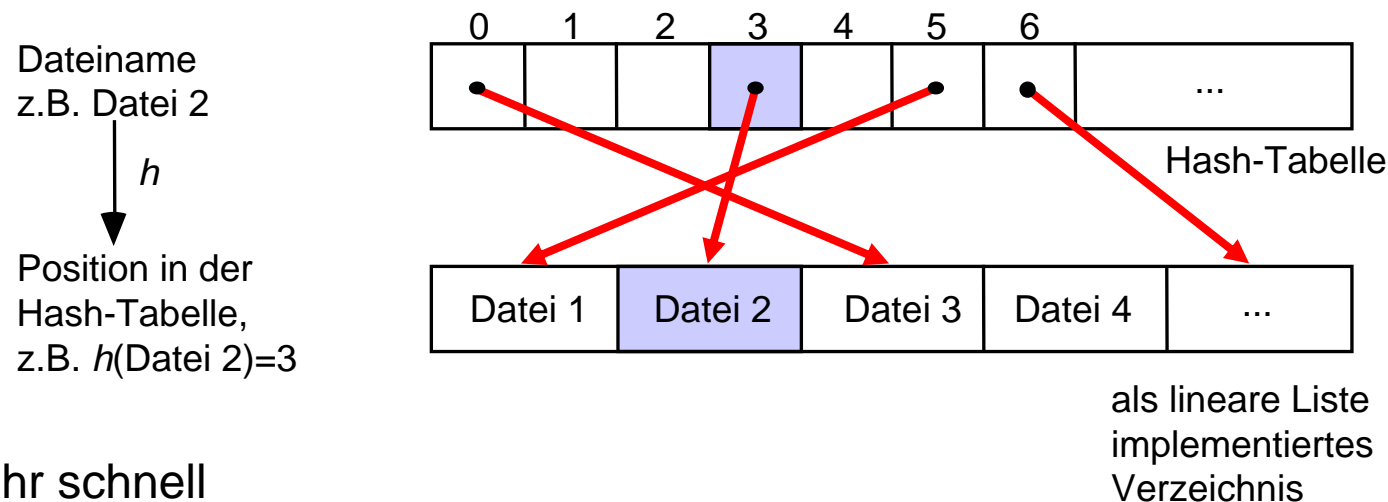
Nachteil:

- sequenzielle Suche, zeitaufwendig
 - ➔ Abhilfe: Sortieren oder **Cache**

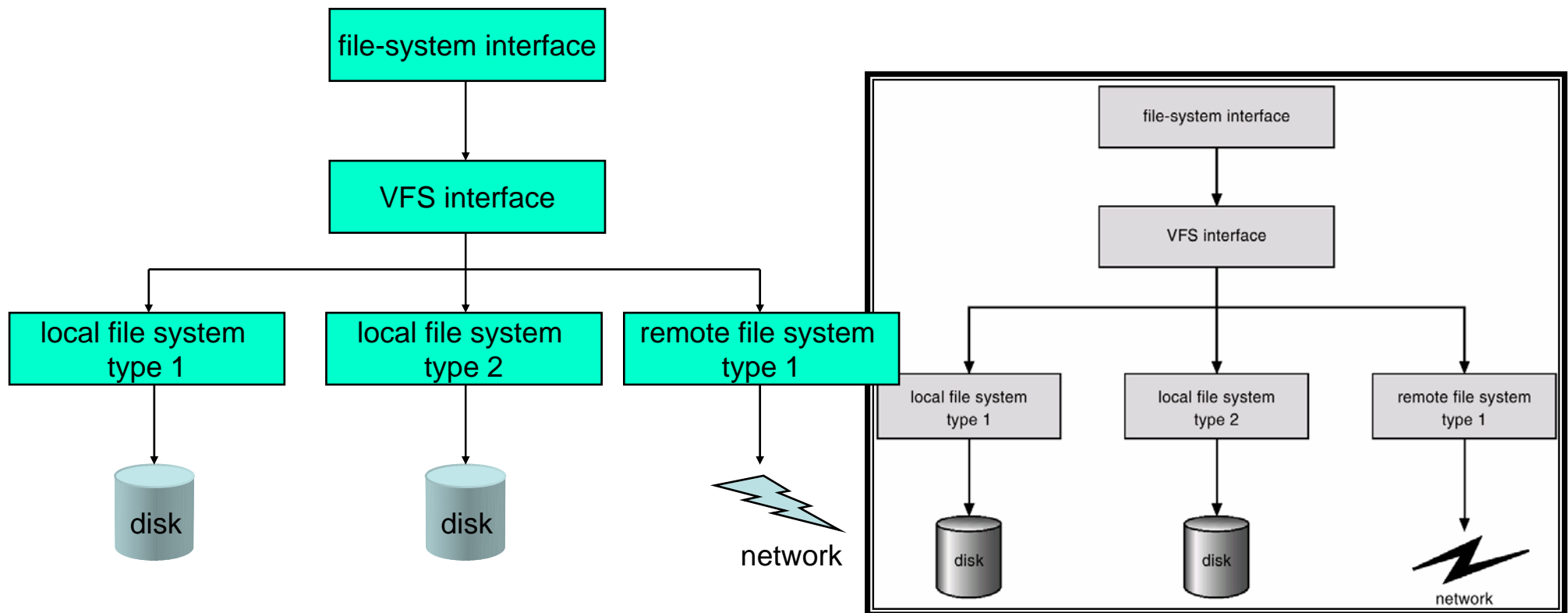
8.6 Implementierung von Verzeichnissen

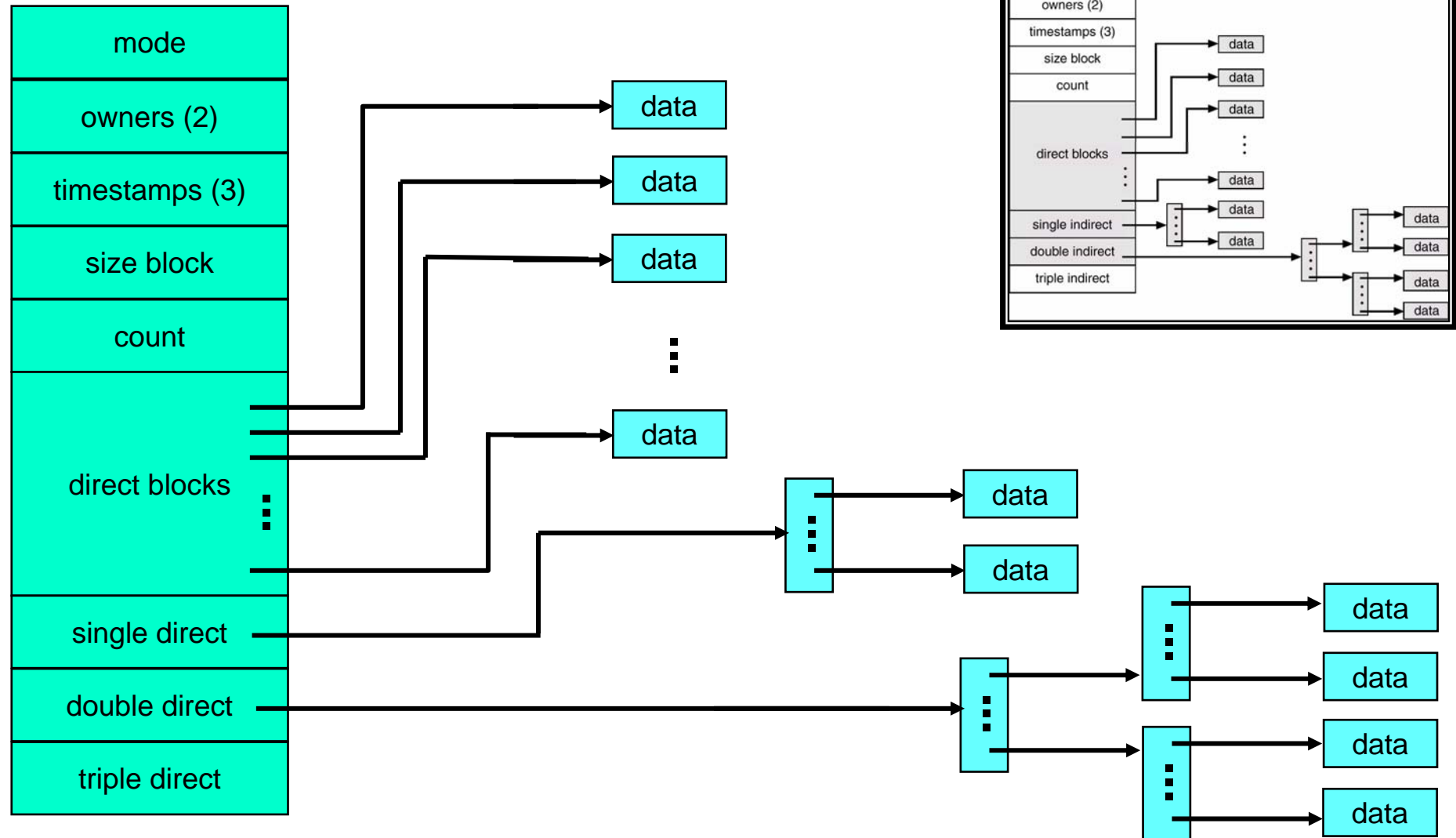
Hashtabelle:

- $h: \{\text{Schlüssel}\} \rightarrow \{\text{Speicherpositionen in Hash-Tabelle}\}$
- einfach zu berechnen
- h soll verfügbaren Speicherraum gleichmäßig ausnutzen
- h nicht notwendig injektiv \rightarrow Menge der Schlüssel kann auf wesentlich kleinere Hash-Tabellen reduziert werden.



- + Suche sehr schnell
- + Einfügen/Löschen einer Datei
- Vorkehrungen für Kollisionen notwendig \rightarrow Überlauftabelle
- Änderung der Hash-Tabellengröße erfordert neue Hash-Funktion





Inhaltsverzeichnis

9.1 Aufgaben von Linker und Lader

9.2 Der Linker

- Der Linkage-Editor
- Beispielaufbau eines Moduls
- Beispiel eines Bindevorgangs

9.3 Der Lader

9.1 Aufgaben von Linker und Lader

Programme sind im Allgemeinen modular aufgebaut → sollten es jedenfalls.

Die Zusammenfassung der Module zu einem lauffähigen Programm ist Aufgabe des **Linkers**, auch **Binder** genannt, d.h.:

- Anordnen der Module
- Auflösen und Konkretisieren von symbolischen Bezügen

Der **Lader** bringt das Programm in den Hauptspeicher.

9.2 Der Linker

Wann ist das Binden günstig?

Programmphase	Binden günstig?
Während der Programmierung	Nein, da die Module dann nicht mehr unabhängig sind
Fertige Quellcodes	Nein, da gleiche Module bei Wiederverwendung mehrfach übersetzt werden müssten.
Übersetzung	Nein, aus demselben Grund.
Wenn Module übersetzt vorliegen (Linkage-Editor)	Günstig, da die Möglichkeiten zur Zusammensetzung flexibel bleiben und jetzt genügend Zeit vorhanden ist.
Beim Laden (Linkage-Loader)	Gebräuchlich, jedoch mit dem Nachteil, dass man vor dem Start eines Programms das Binden abwarten muss.
Ausführung	Nein, obwohl sehr flexibel würde die Geschwindigkeit der Programme zu sehr leiden.

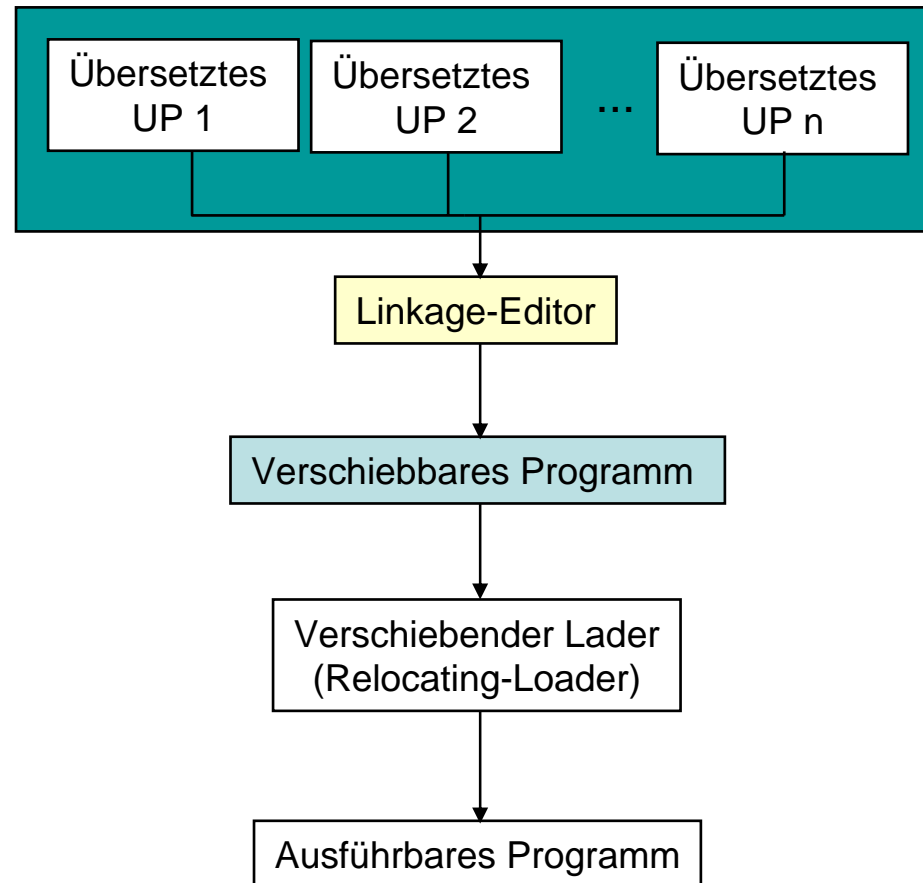
9.2 Der Linker / Linkage-Editor

Eingabe:

- Übersetzte Objektmodule UP_i
- evtl. Steuerkommandos

Ausgabe:

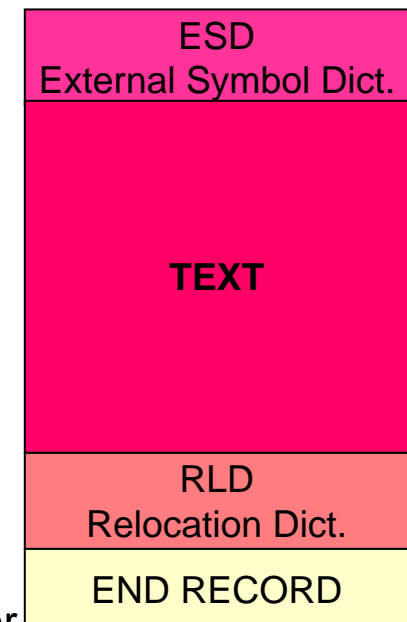
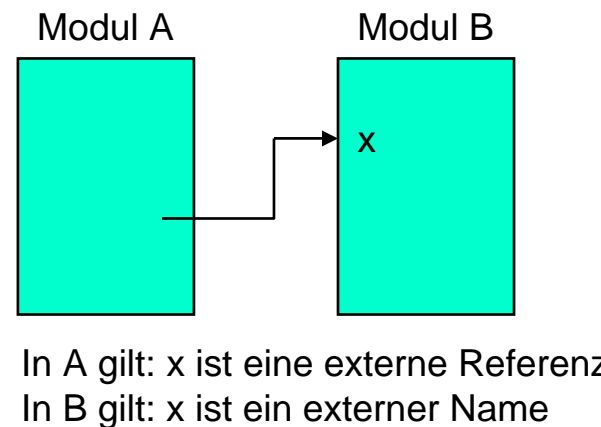
- Verschiebbares Programm



9.2 Der Linker / Beispielaufbau eines Moduls

Der Aufbau eines Objektmoduls (angelehnt an einen IBM Großrechner):

Das External-Symbol-Dictionary (ESD) ist eine Liste der im **Text-Teil** vorkommenden externen Symbole.



Ein externes Symbol ist entweder

- ein **externer Name**, unter dem das Modul von außen erreichbar ist, oder
- eine **externe Referenz**, d.h. ein Name, der einen Aufruf nach außen bewirkt.

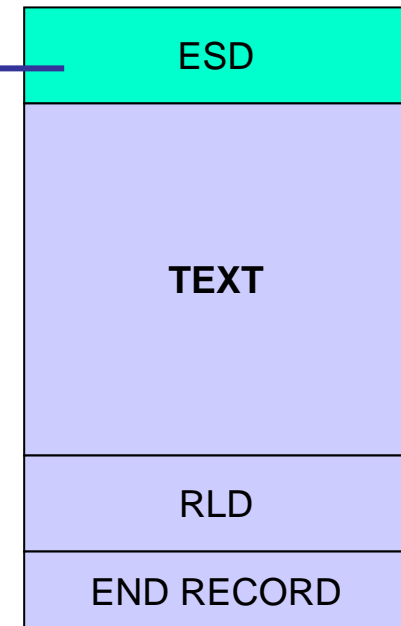
Das Relocation Dictionary (RD) enthält alle Namen, die noch speziell behandelt werden müssen, d.h. die Adressen müssen an die aktuelle Laufzeitumgebung angepasst werden

9.2 Beispielaufbau eines Moduls / ESD

Allgemein:

NAME	TYP	DATA	DATA
------	-----	------	------

Objektmodul



Das ESD kann folgende Typen von Einträgen enthalten:

Section-Definition (SD):

Name	SD	Startadresse	Länge
------	----	--------------	-------

➤ direkter Verweis auf ein Unterprogramm.

Label-Definition (LD):

Name	LD	Anfangsadresse	Pointer
------	----	----------------	---------

- verweist auf einen Entry-Point
- die Anfangsadresse ist relativ zum Modulanfang gegeben
- der Pointer verweist auf die Zeile derjenigen Control-Section (i.a. Unterprogramm), die den betreffenden Namen enthält

External-Referenz (ER):

Name	ER	leer	leer
------	----	------	------

➤ kennzeichnet, dass der verwendete Name auf einen extern definierten Namen Bezug nimmt

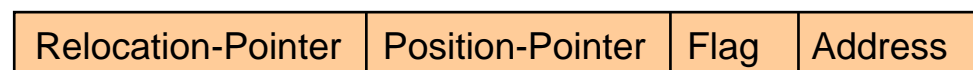
9.2 Beispielaufbau eines Moduls / RLD

Im Relocation-Dictionary (RLD) stehen diejenigen Bereiche im Text, die besonders behandelt werden müssen.

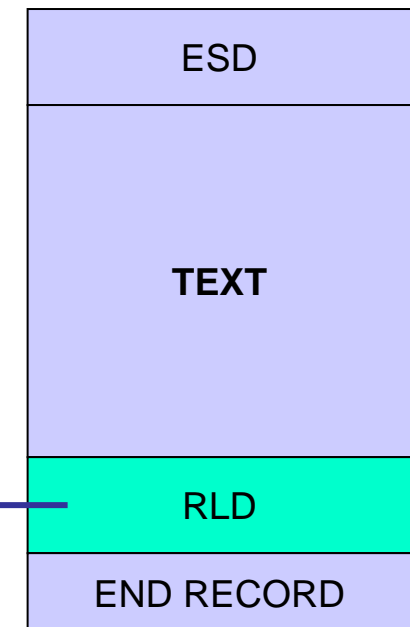
- enthält alle Adressen, bei denen beim Binden bzw. Laden noch Anpassungen vorgenommen werden müssen. (z.B. Adressbefehle)

Ein RLD-Eintrag hat den folgenden Aufbau:

- **Relocation-Pointer**: zeigt auf den entsprechenden ESD-Eintrag
- **Position-Pointer**: verweist auf den ESD-Eintrag der Control-Section, in der die betreffende Konstante definiert ist.
- **Flag**: zeigt die Art eines Befehls (s.u.) an
- **Address**: enthält den relativen Abstand der Konstanten zum Programmbeginn



Objektmodul

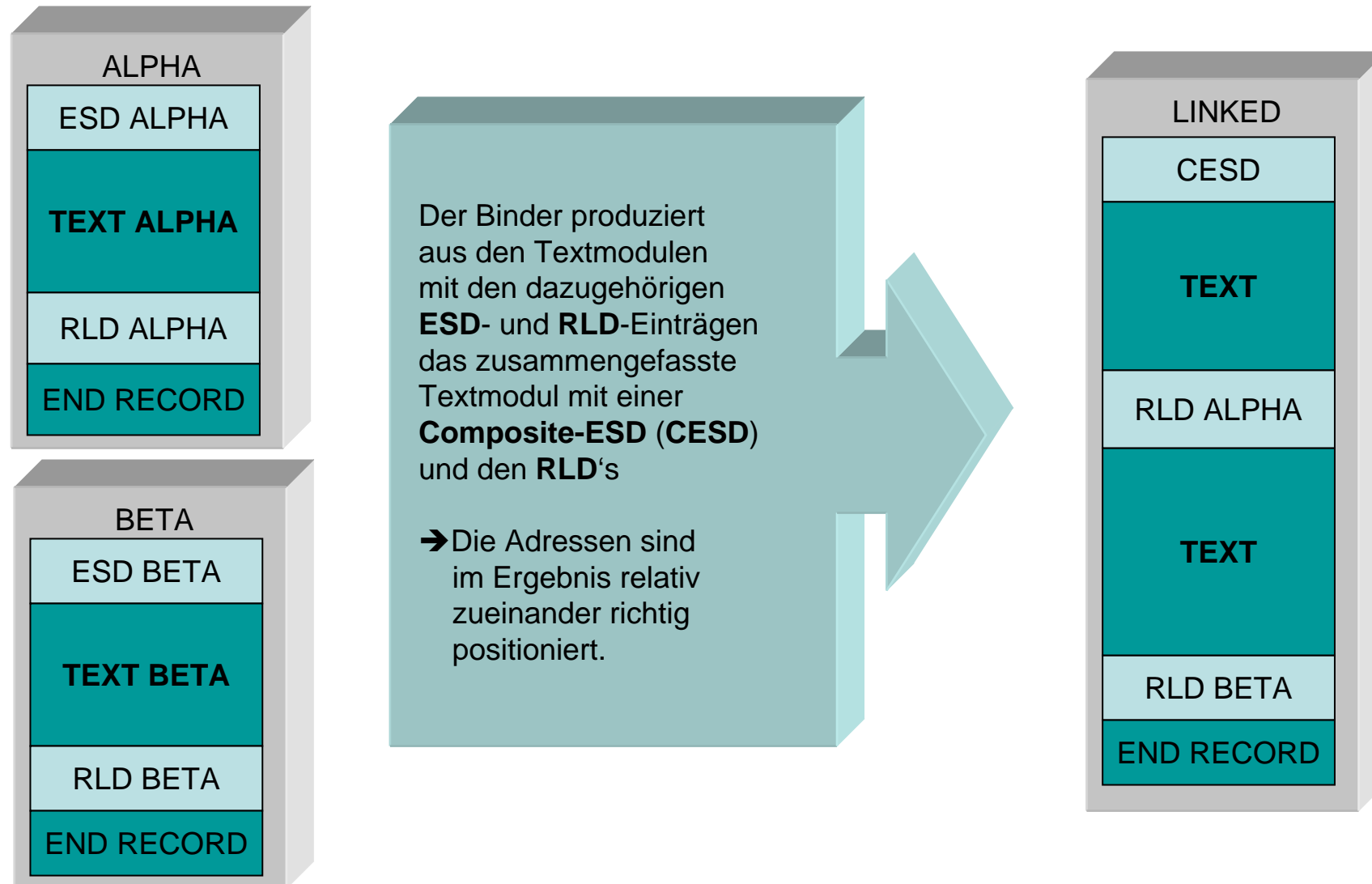


9.2 Beispielaufbau eines Moduls / RLD

Beispiele für Befehle, die eine Verschiebung erfordern und deshalb im **RLD** verzeichnet sind:

Befehl	Bedeutung	Wirkung
DC A(X)	Define-Constant-Address	Inhalt dieser Zelle ist zur Ausführungszeit die aktuelle Adresse von X, wobei X ein lokaler Name ist.
DC V(X)	Define-Constant-Variable	Inhalt ist die aktuelle Adresse von X, jedoch ist X hier ein externes Symbol.

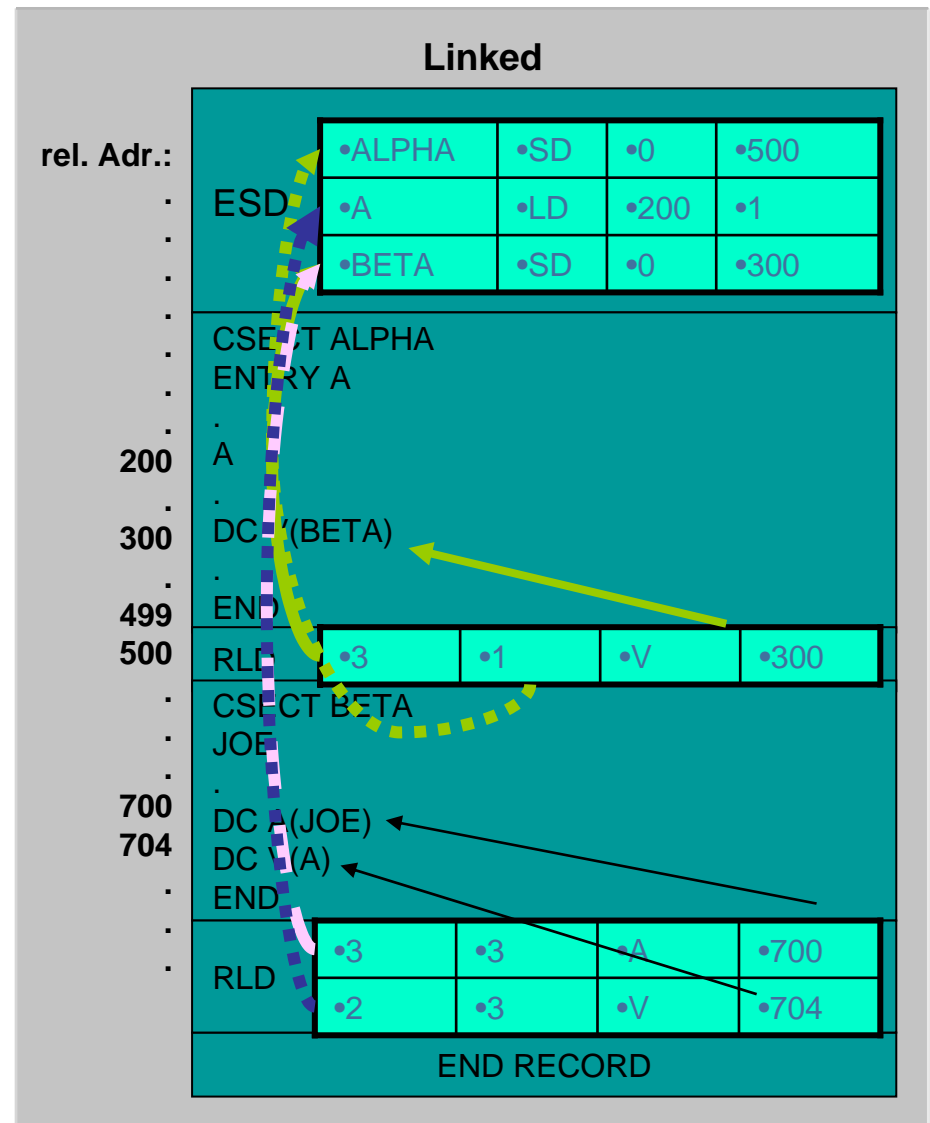
9.2 Beispiel eines Bindevorgangs



9.2 Beispiel eines Bindevorgangs

Das gebundene Modul hat folgendes Aussehen:

Es enthält nur relative Adressen und ist deshalb (immer noch) verschiebbar.



9.3 Der Lader

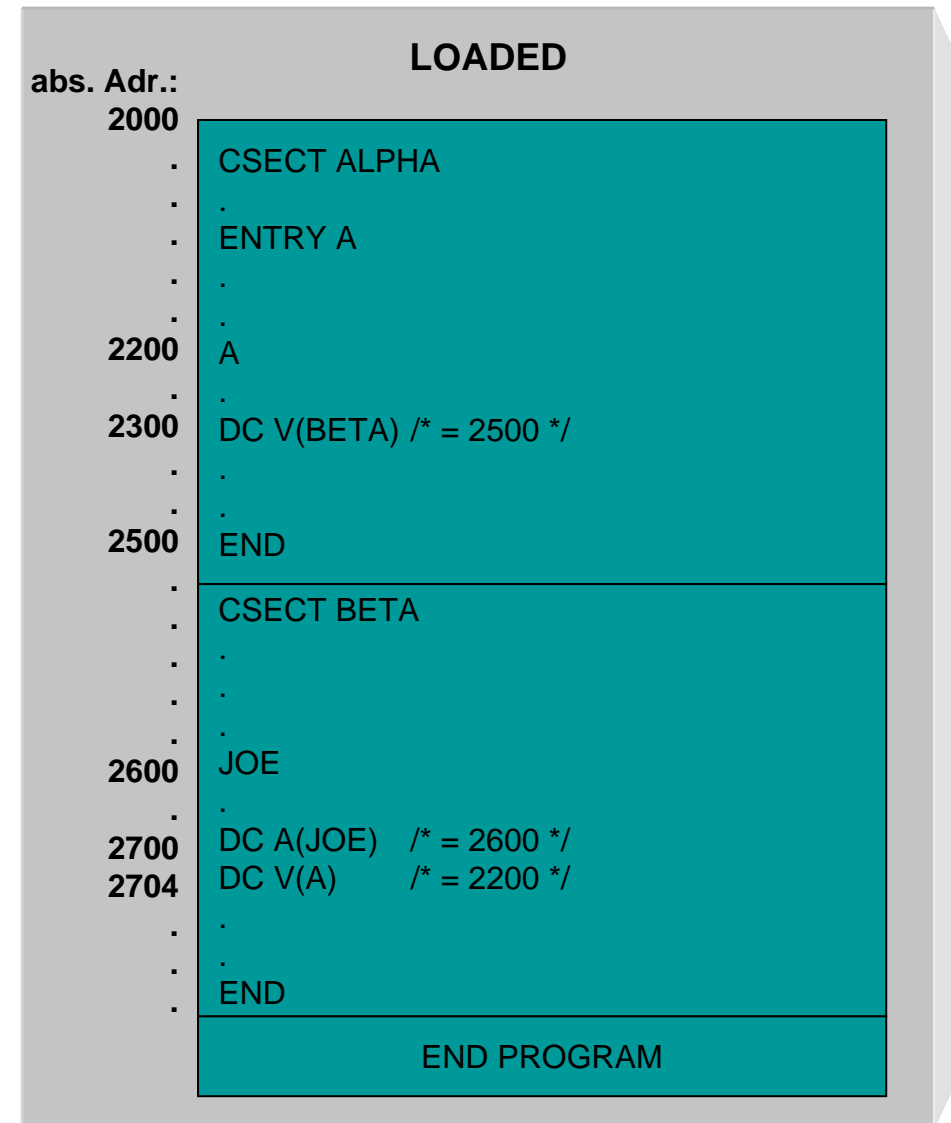
Beim Ladevorgang sind die relativen Adressen in die endgültigen absoluten Adressen umzurechnen. Der verschiebende Lader (**Relocating-Loader**) bestimmt zunächst einen ausreichend großen freien Speicher und platziert das Programm an diese Stelle. Die Berechnung der Adressen im Programm:

Basisadresse + relative Adresse

Im Beispiel:

- zusammenhängenden Speicherbereich von mindestens 800 Byte suchen:
- ➔ gefunden ab Speicherstelle 2000.

- das gebundene Modul dorthin laden:
- ➔ alle Bezüge zwischen den Modulen, die noch offen waren, sind aufgelöst.



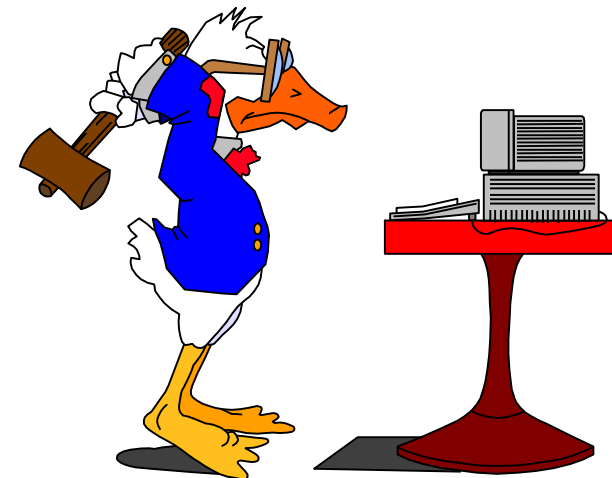
Inhaltsverzeichnis

10.1 Schutzmechanismen

- Schutzbereiche
- Zugriffsmatrizen
- Implementierung von Zugriffsmatrizen
- Entzug von Zugriffsrechten

10.2 Das Problem der Sicherheit

- Authentifizierung
- Viren und ähnliches Gewürm
- Kryptographie



10.1 Schutzmechanismen

Was wird vor was geschützt?

Betrachte den Computer abstrakt als Sammlung von Objekten, die von Prozessen genutzt werden können:

- CPU
- Geräte (Festplatte, Drucker, Schnittstellen, ...)
- Speicher
- Prozesse (User-Programme, Treiber, ...)
- Dateien
- ...

Jedem Objekt sind zugeordnet:

- Name und
- mögliche Operationen

10.1 Schutzmechanismen

Ziel:

Schutz der **Objekte** vor unberechtigten **Operationen** der **Prozesse**.

Was darf Prozess X?

Wer welche Rechte bekommt, hängt von der Politik des Schutzes ab.

- Für jeden **Prozess** wird festgelegt, zu welchen Operationen auf welchen Objekten er berechtigt ist.
- Diese Berechtigungen können auch mit einem **Benutzer** verknüpft werden.

Bei der Authentifizierung des Benutzers werden seine Berechtigungen aktiv, indem sie den entsprechenden Prozessen zugewiesen werden.



10.1 Schutzmechanismen / Schutzbereiche

Verschiedene Prozesse haben oft die gleiche Kombination von Rechten auf Objekte

→ Definition so genannter Schutzbereiche

Ein **Schutzbereich** besteht aus einer Menge von **Zugriffsrechten**.
Ein Zugriffsrecht ist ein geordnetes Paar:
<Objekt, Menge von Operationsrechten>

Beispiel:

$SB_D = \langle \text{Datei } D, \{\text{lesen, schreiben}\} \rangle$

Ein Prozess in SB_D darf von der Datei D lesen und schreiben, jedoch keine anderen Operationen ausführen.

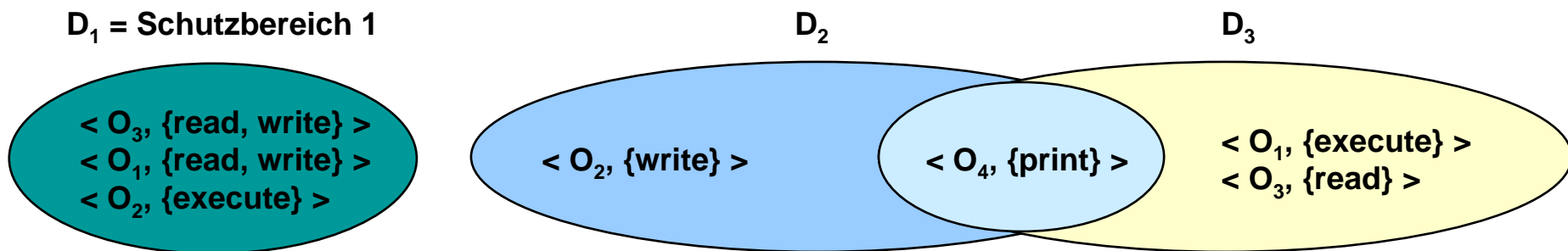
10.1 Schutzmechanismen / Schutzbereiche

Jedem Prozess wird nun statt einer eigenen Berechtigungsdefinition ein Schutzbereich zugeordnet.

Schutzbereiche können von mehreren Prozessen genutzt werden

⇒ Reduktion der Verwaltungsdaten.

Weitere Reduktion der Daten ist durch Überlappen von Schutzbereichen möglich:



10.1 Schutzmechanismen / Schutzbereiche

Die Zuordnung der Schutzbereiche kann **statisch** oder **dynamisch** erfolgen.

Statisch:

- Schutzbereich gilt für die Lebensdauer des Prozesses
- einfach zu implementieren
- unflexibel

Dynamisch:

- Berechtigungen sind während der Laufzeit frei veränderbar
- flexibel
- aufwendige Implementierung

10.1 Schutzmechanismen / Schutzbereiche

Schutzbereiche können mit unterschiedliche Granularität realisiert werden:

➤ Benutzerebene:

- Jeder Benutzer stellt einen Schutzbereich dar
- Wechsel des Schutzbereichs geschieht durch Benutzerwechsel, d.h. Login eines andere Benutzers

➤ Prozessebene:

- Jeder Prozess stellt einen Schutzbereich dar
- Wechsel des Schutzbereichs geschieht durch Aufruf anderer Prozesse
- Gewöhnlich Dualmode = {User, Kernel}, Prozess im Kernelmode darf privilegierte Befehle ausführen, Prozesse im Usermode haben beschränkten Zugriff

➤ Funktionsebene:

- Jede Funktion in einem Prozess stellt einen Schutzbereich dar
- Wechsel des Schutzbereichs geschieht durch Funktionsaufruf

10.1 Schutzmechanismen / Zugriffsmatrizen

Mehrere Schutzbereiche lassen sich durch **Zugriffsmatrizen** definieren:

		Objekt			
		Datei1	Datei2	Datei3	Drucker
Schutzbereich	S1	lesen		lesen	drucken
	S2		ausführen	lesen	drucken
	S3	lesen, schreiben		lesen	
	S4		ausführen	lesen, schreiben	drucken

- für statische und dynamisch Zuweisung geeignet
- die Zugriffsmatrix kann sich selbst als Objekt enthalten

- Beispiel:

➤ Prozess in S1 darf Datei1 und Datei3 lesen und auf dem Drucker drucken!

10.1 Schutzmechanismen / Zugriffsmatrizen

Zur Modellierung von Rechten für das Schalten zwischen
Schutzbereichen werden diese als Objekte an die Zugriffsmatrix angehängt:

		Objekt					
		...	Drucker	S1	S2	S3	S4
Schutzbereich	S1	...	drucken				
	S2	...	drucken	switch		switch	
	S3	...			switch		
	S4	...	drucken				

Erlaubtes „Switching“:
S3 → S2
S2 → S1
S2 → S3

- Typische Einträge in Zugriffsmatrizen zur Modifikation derselben:
 - Kopierrechte (move, copy): das Recht, einzelne Rechte für ein bestimmtes Objekt zu kopieren
 - Eigentümer: Kontrolle der Rechte eines Objektes, d.h. Hinzufügen und Entfernen von Rechten (Spalte)
 - Kontrolle: Kontrolle der Rechte eines Schutzbereiches (Zeile)

10.1 Schutzmechanismen / Zugriffsmatrizen

		Objekt			
		Datei1	Datei2	Datei3	Drucker
Schutzbereich	S1	lesen			drucken
	S2		ausführen		drucken
	S3	lesen, schreiben		lesen*	
	S4		ausführen		drucken

Kopierrecht:

das Recht, einzelne Rechte für ein bestimmtes Objekt zukopieren

		Objekt			
		Datei1	Datei2	Datei3	Drucker
Schutzbereich	S1	lesen			drucken
	S2		ausführen		drucken
	S3	lesen, schreiben		lesen*	
	S4		ausführen	lesen	drucken

10.1 Schutzmechanismen / Zugriffsmatrizen

		Objekt			
		Datei1	Datei2	Datei3	Drucker
Schutzbereich	S1	lesen, Eigentümer			drucken
	S2		ausführen		drucken
	S3	lesen, schreiben		lesen	
	S4		ausführen		drucken

Eigentümer:

Kontrolle der Rechte eines Objektes, d.h. Hinzufügen und Entfernen von Rechten

		Objekt			
		Datei1	Datei2	Datei3	Drucker
Schutzbereich	S1	lesen			drucken
	S2		ausführen		drucken
	S3	lesen, schreiben		lesen	
	S4	lesen	ausführen		drucken

10.1 Schutzmechanismen / Zugriffsmatrizen

		Objekt					
		...	Drucker	S1	S2	S3	S4
Schutzbereich	S1	...	drucken				
	S2	...	drucken	switch		switch control	
	S3	...			switch		
	S4	...	drucken				

Kontrolle:

Kontrolle der
Rechte eines
Schutzbereiches
(Zeile)

		Objekt					
		...	Drucker	S1	S2	S3	S4
Schutzbereich	S1	...	drucken				
	S2	...	drucken	switch		switch control	
	S3	...	drucken		switch		
	S4	...	drucken				

10.1 Schutzmechanismen / Implementierung von Zugriffsmatrizen

Möglichkeiten:

- globale Tabelle
- Zugriffslisten für Objekte
- Capability-Listen
- Schlüssel-Schloss Mechanismus

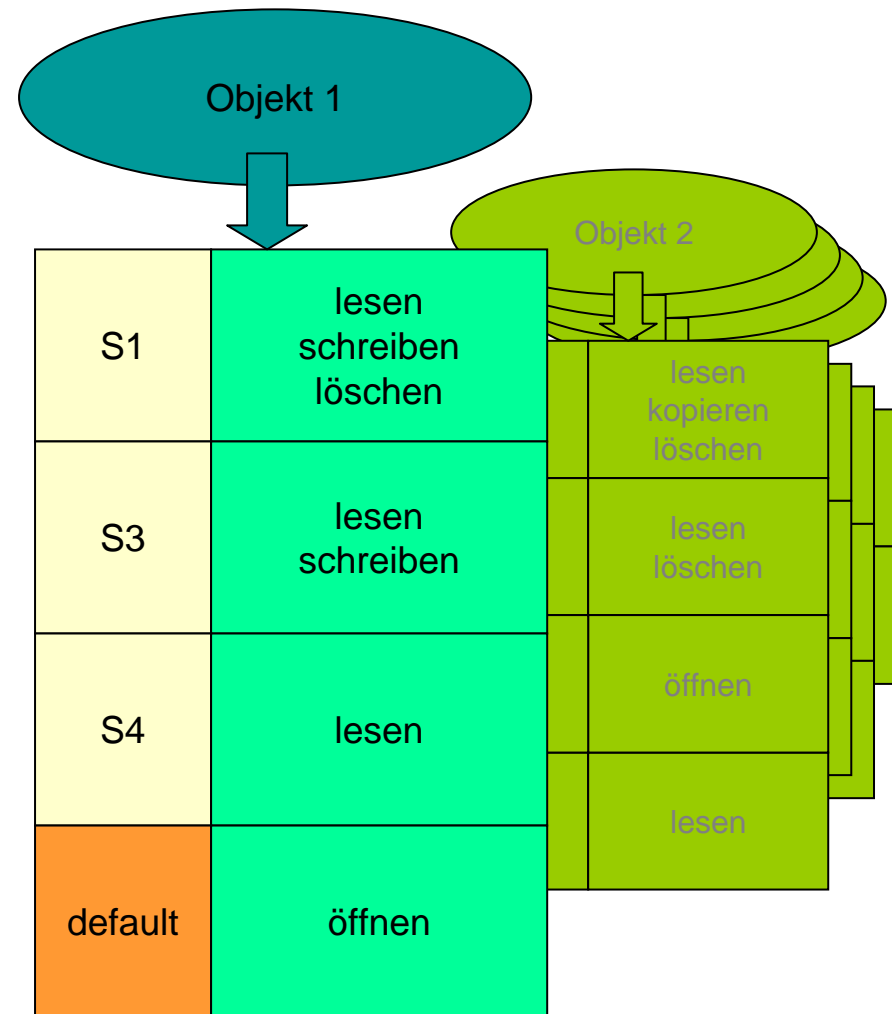
Mittels globaler Tabelle:

- Tabelle besteht aus geordneten Einträgen der Form (Schutzbereich, Objekt, Rechte)
- Bei Aufruf einer Operation m auf ein Objekt O aus dem Schutzbereich S wird überprüft, ob ein Eintrag (S, O, R) existiert, so dass $m \in R$ gilt.

10.1 Schutzmechanismen / Implementierung von Zugriffsmatrizen

Mittels Zugriffslisten für Objekte:

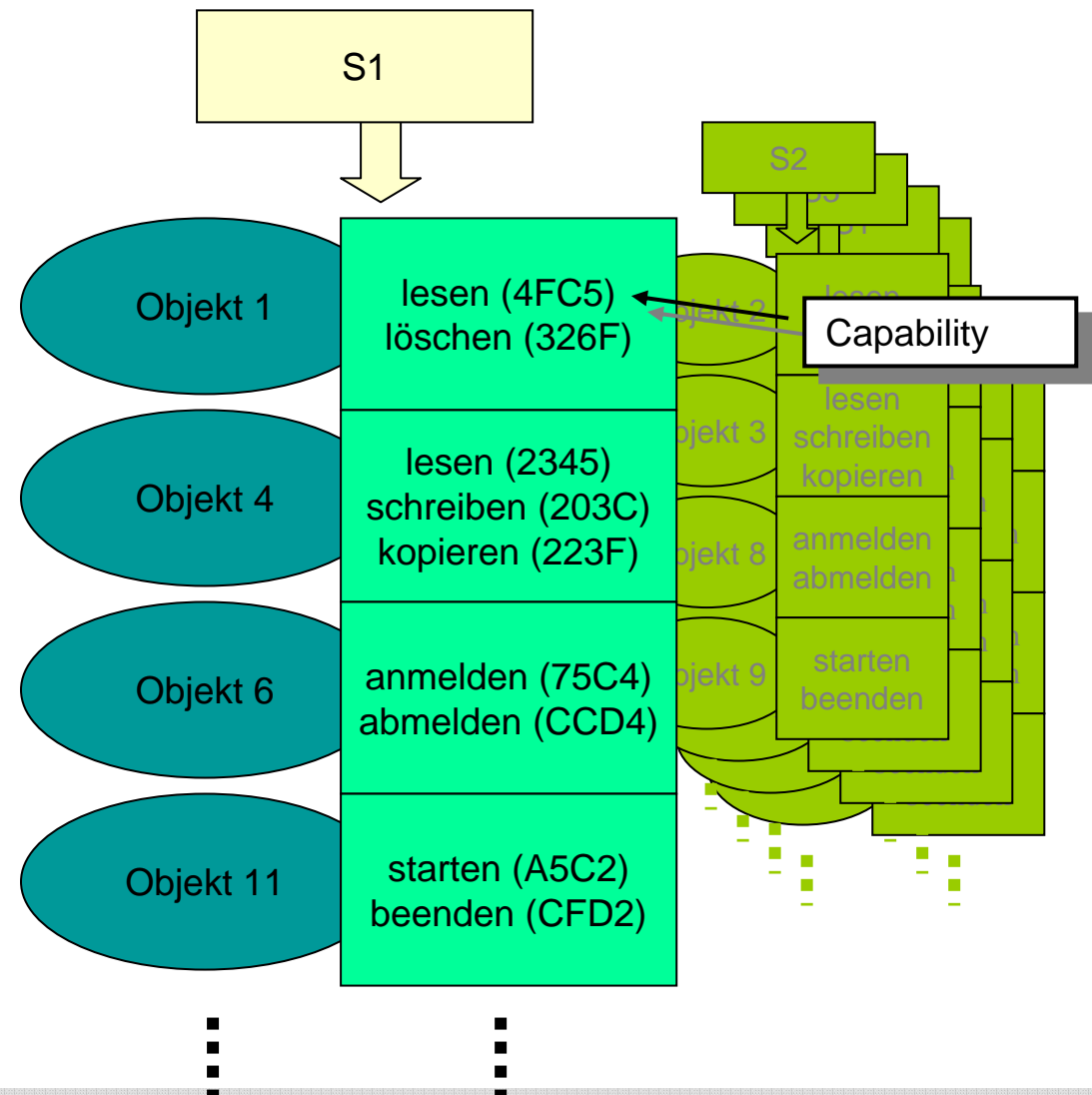
- eine (Schutzbereich, Rechte)-Liste für jedes Objekt
- Schutzbereiche ohne Rechte tauchen nicht auf
- zusätzliches Element für „default“-Rechte reduziert die Daten weiter



10.1 Schutzmechanismen / Implementierung von Zugriffsmatrizen

Mittels Capability-Listen:

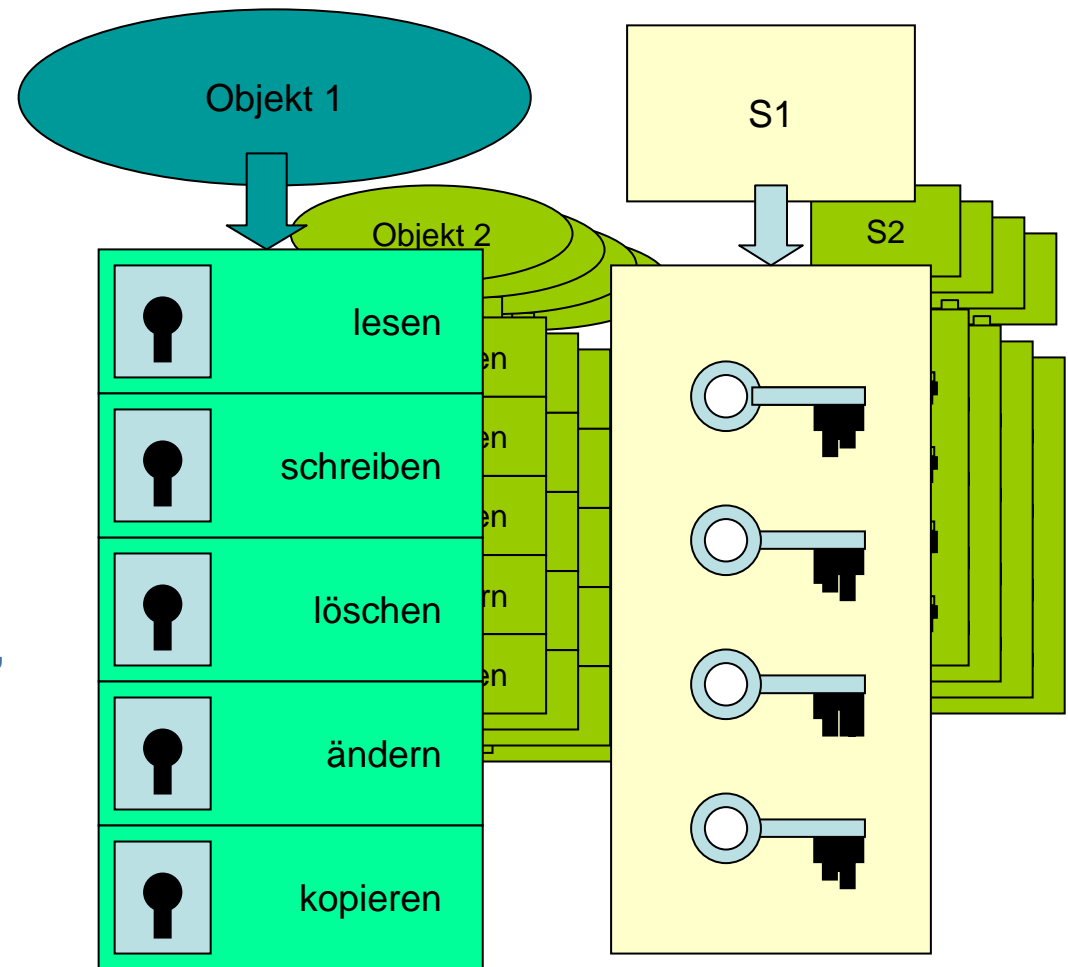
- eine (Objekt, Rechte)-Liste für jeden Schutzbereich
- Objekte ohne Rechte tauchen nicht auf
- **Capability**: physikalischer Name bzw. Adresse
- Zugriff erfolgt mittels der Capability, ohne diese könnte die Operation gar nicht aufgerufen werden



10.1 Schutzmechanismen / Implementierung von Zugriffsmatrizen

Mittels Schlüssel-Schloss-Mechanismus:

- eine (Schloss, Recht)-Liste für jedes Objekt
- eine Schlüssel-Liste für jeden Schutzbereich
- hat ein Prozess in seinem Schutzbereich den passenden Schlüssel zum Schloss einer Operation, ist der Zugriff erlaubt
- Schlüssel und Schlösser sind Bitmuster



10.2 Das Problem der Sicherheit

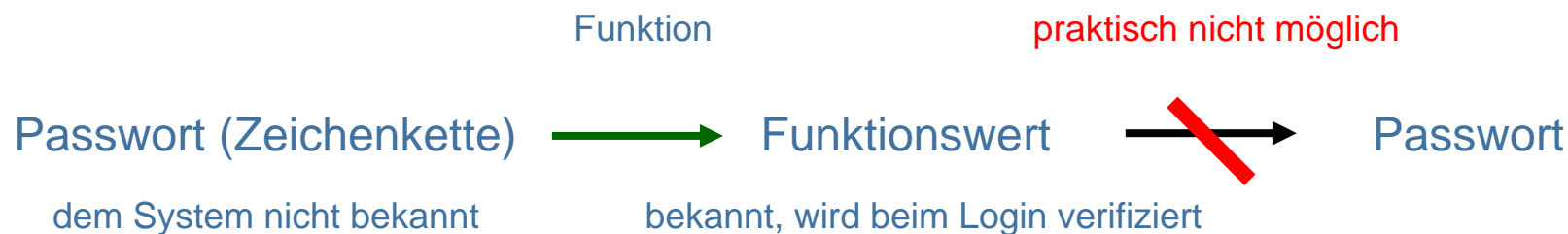
- Die oben genannten Mechanismen erlauben den Schutz von Objekten vor unerlaubten Zugriffen durch **Prozesse**
- Um das System vor unerlaubten Zugriffen durch **Personen** zu schützen, müssen Schutzbereiche mit Personen verknüpft werden
- zur Identifizierung der am Rechner arbeitenden Person wird eine **Authentifizierung** durchgeführt

10.2 Das Problem der Sicherheit / Authentifizierung

Mögliche Verfahren:

- Passwort (sorgfältige Wahl/Handhabung wichtig!)
- Fingerabdruck
- Chip-/Magnetkarte
- ...

Authentifizierung mit Passwort:



In Netzwerken wegen Übertragung des Passworts problematisch

⇒ Kryptographie nötig



10.2 Das Problem der Sicherheit / Viren und ähnliches Gewürm

Trap-Doors

- in Betriebssystemen „eingebaut“
- Trojanische Pferde
 - in gemeinsam benutzten Programmen, z.B. Compiler
- Worms
 - Programme die sich vermehren und sich im Netzwerk verbreiten
- Viren
 - keine eigenständigen Programme, sondern Teil eines infizierten Programms
 - verbreiten sich durch „Infektion“ weiterer Programme

Gegenmaßnahmen:

➔ sind nie 100prozentig!

- Monitoring
- Firewall
- Vorsichtige Vorgehensweise bei Softwareanschaffungen
- Virens Scanner

10.2 Das Problem der Sicherheit / Kryptographie

Verschlüsselung von Informationen

Klartext → chiffrierter Text

Die **chiffrierten (kodierten)** Daten sind für unberechtigte Leser nicht zu entziffern.

Ein zum Lesen berechtigter Benutzer besitzt einen **Schlüssel**, mit dem der chiffrierte Text **dechiffriert (dekodiert)** werden kann.

chiffrierter Text → Klartext

10.2 Das Problem der Sicherheit / Kryptographie

Im Allgemeinen werden für den hier beschriebenen Vorgang drei Dinge benötigt:

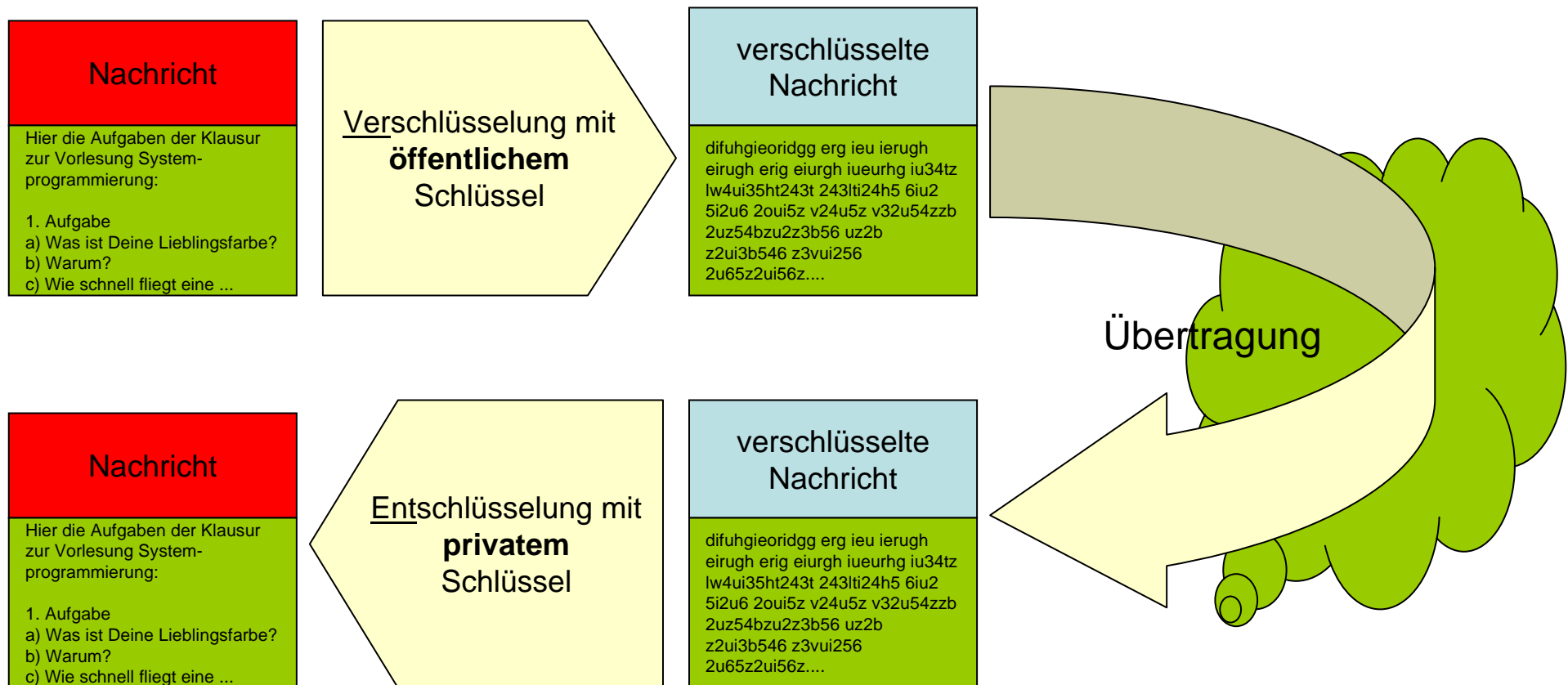
- Kodieralgorithmus K
- Dekodieralgorithmus D
- geheimer Schlüssel $s \rightarrow$ bei asymmetrischen Verfahren zwei Schlüssel

Wird die Nachricht m übertragen, müssen die folgenden Bedingungen gelten:

- $D(s, K(s, m)) = m$
- D und K sind effizient berechenbar
- Um die Sicherheit zu gewährleisten, muss nur der Schlüssel s geheim sein
(die Algorithmen D und K dürfen bekannt sein)

10.2 Das Problem der Sicherheit / Kryptographie

In unsicheren Netzwerken benutzt man für die Übertragung der Daten häufig **asymmetrische** Verfahren mit **öffentlichen** und **privaten** Schlüsseln:



10.2 Das Problem der Sicherheit / Kryptographie

Bekanntestes **asymmetrisches** Verfahren: **RSA** (Rivest, Shamir, Adleman)

Basiert auf der Schwierigkeit, große Zahlen (z.B. 100 Dezimalstellen) in ihre Primfaktoren zu zerlegen

Ablauf:

- Wähle große Primzahlen p und q
- Berechne $n = p \cdot q$, $\phi(n) = (p-1) \cdot (q-1)$
- Wähle e relativ prim (teilerfremd) zu $\phi(n)$
- Berechne d mit $d \cdot e = 1 \bmod \phi(n)$
- Öffentlicher Schlüssel: (e, n)
- Privater Schlüssel: (d, n)
- Verschlüsselung von m mit $c = m^e \bmod n$
- Entschlüsselung von c mit $m = c^d \bmod n$

10.2 Das Problem der Sicherheit / Kryptographie

Beispiel:

$$p = 17,$$

$$q = 19$$

$$\Rightarrow n = 17 \cdot 19 = 323$$

$$\Rightarrow \phi(n) = 16 \cdot 18 = 288$$

Schlüssel:

wähle $e = 43$ (relativ prim zu $\phi(n) = 288$)

$$\Rightarrow d = 67, \text{ da } d \cdot e = 1 \pmod{288}$$

e und n sind öffentlich, d geheim

Verschlüsselung von $m = 219$:

$$c = 219^{43} \pmod{323} = 281$$

$$m = 281^{67} \pmod{323} = 219$$

Inhaltsverzeichnis

11.1 Einführung in verteilte Systeme

11.2 Kommunikationsmechanismen

- Das Client/Server-Modell
- Der Remote Procedure Call

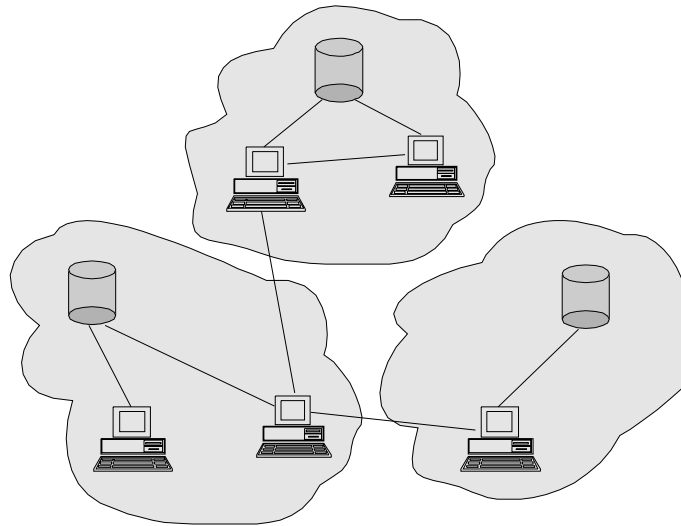
11.3 Kooperation und Koordination in verteilten Systemen

- Uhrensynchronisation
- Wechselseitiger Ausschluss
- Atomare Transaktionsverarbeitung
- Concurrency Control
- Deadlock-Behandlung
- Abstimmung eines „wahren“ Wertes

11.4 Verteilungsplattformen

11.1 Einführung in verteilte Systeme

Ein **Verteiltes System** ist ein System mit räumlich verteilten Komponenten, die keinen gemeinsamen Speicher benutzen und einer dezentralen Administration unterstellt sind. Zur Ausführung gemeinsamer Ziele ist die Kooperation der Komponenten notwendig.



Oder auch:

„Ein System, mit dem man nicht arbeiten kann, weil ein Rechner ausgefallen ist, von dem man noch nie etwas gehört hat.“ (Leslie Lamport)

11.1 Einführung in verteilte Systeme

Verteilte Systeme: Relativ junges Konzept

Die Entwicklung Verteilter Systeme wurde begünstigt durch

- Hardware: Leistungsexplosion bei Halbleiterchips
 - Stetig wachsende Leistung bei schrumpfenden Preisen und Abmessungen
 - Ausführung komplexerer Software auf mehr Rechnern
- Kommunikation: Entwicklung schneller lokaler Datennetze
 - Senkung von Zugriffszeiten
 - Wegbereiter Ethernet
- Softwaretechnik: Modularisierung, Schnittstellen, Objektorientierung
 - Remote Procedure Call, objektorientierte Modellierung
- Autonomie der Rechnerorganisation: Dezentralisierung
 - Abkehr von streng hierarchischen Organisationsformen in Unternehmen

11.1 Einführung in verteilte Systeme

Vorteile:

- Stetige Kapazitätsanpassung
 - Anpassung der Größe eines Systems
- Integrierbarkeit bestehender Lösungen
 - Nutzung existierender Systeme durch neue Systemkomponenten
(keine Neuentwicklung eines Systems gleicher Funktionalität)
- Risikominimierung
 - Minimierung des Risikos der Überlastung einzelner Systemkomponenten durch sukzessive Systemerweiterung
- Flexibilität und Anpassbarkeit
 - Kostengünstige Realisierungen durch überschaubare, organisatorische Verwaltung
- Autonomie
 - Einzelne Fehler / Ausfälle können von anderen Komponenten toleriert werden
(Überbrückung von Störfällen)

11.1 Einführung in verteilte Systeme

Nachteile und Probleme:

- Informationsverarbeitung
 - Zunahme der Komplexität durch Verteilung und Heterogenität
- Komplexe Netzinfrastrukturen
 - Verwaltung der Gesamtstruktur
- Softwaredefizit
 - Unausgereiftheit der Produkte
- Sicherheitsbedenken
 - Zusätzliche Fehlermöglichkeiten durch neue Netzwerkkomponenten
 - Datenschutz: generell einfacherer Angriff als bei separater Datenhaltung
- Parallelität der Ereignisse
 - Ordnung von Ereignissen auf verschiedenen Rechnern
- Konsistenzprobleme
 - Zugriff auf verteilt gehaltene Daten

11.1 Einführung in verteilte Systeme

Transparenz: Verbergen von Implementierungsdetails zur Nutzungserleichterung

Wichtig für verteilte Systeme:

Verteilungstransparenz → Verbergen der Komplexität des verteilten Systems

- Erleichtert den Umgang mit verteilten Anwendungen
- Interne Vorgänge sind vor dem Benutzer verborgen
- Der Anwendungsprogrammierer wird entlastet

Ausprägungen:

- Zugriffstransparenz
- Ortstransparenz
- Replikationstransparenz
- Abarbeitungstransparenz
- Migrationstransparenz
- Ausfalltransparenz
- Ressourcentransparenz
- Gruppentransparenz

11.2 Kommunikationsmechanismen

Problem in verteilten Systemen:

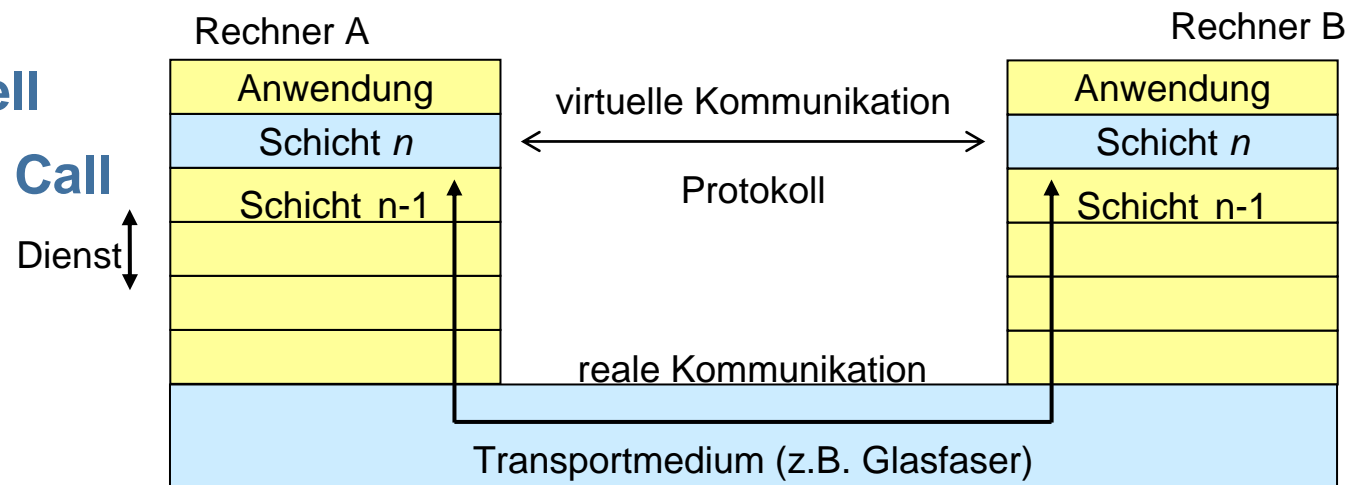
- Hoher Verwaltungsaufwand durch OSI-Modell
- Zur effizienten Kommunikation wird einfacheres Modell benötigt

Idee:

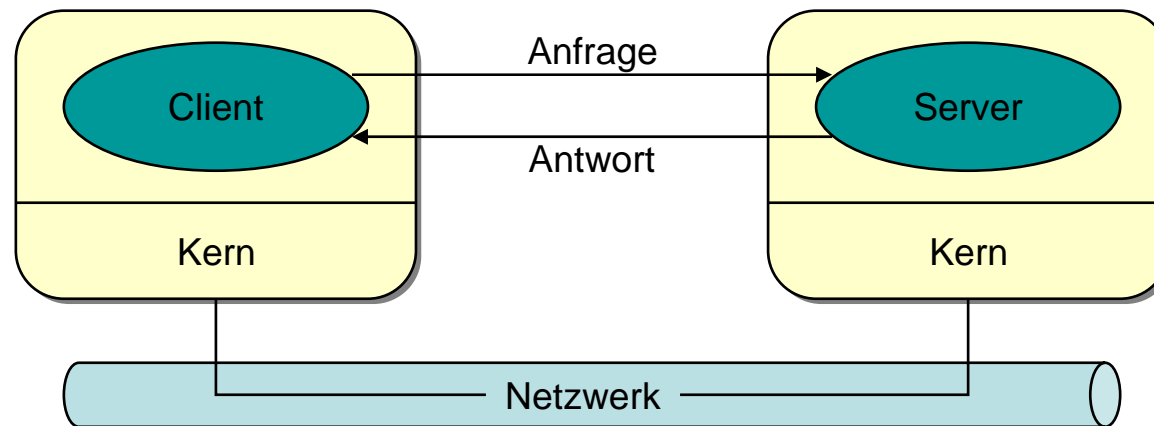
- Strukturierung des Betriebssystems als Menge kooperierender Prozesse (**Server**).
- Server stellen Nutzern (**Clients**) Dienste bereit
- Kommunikation durch einfache Primitive

➔ Client/Server-Modell

➔ Remote Procedure Call



11.2 Kommunikationsmechanismen / Das Client/Server-Modell



Kommunikationsmodell mit einfachem Verwaltungsaufwand

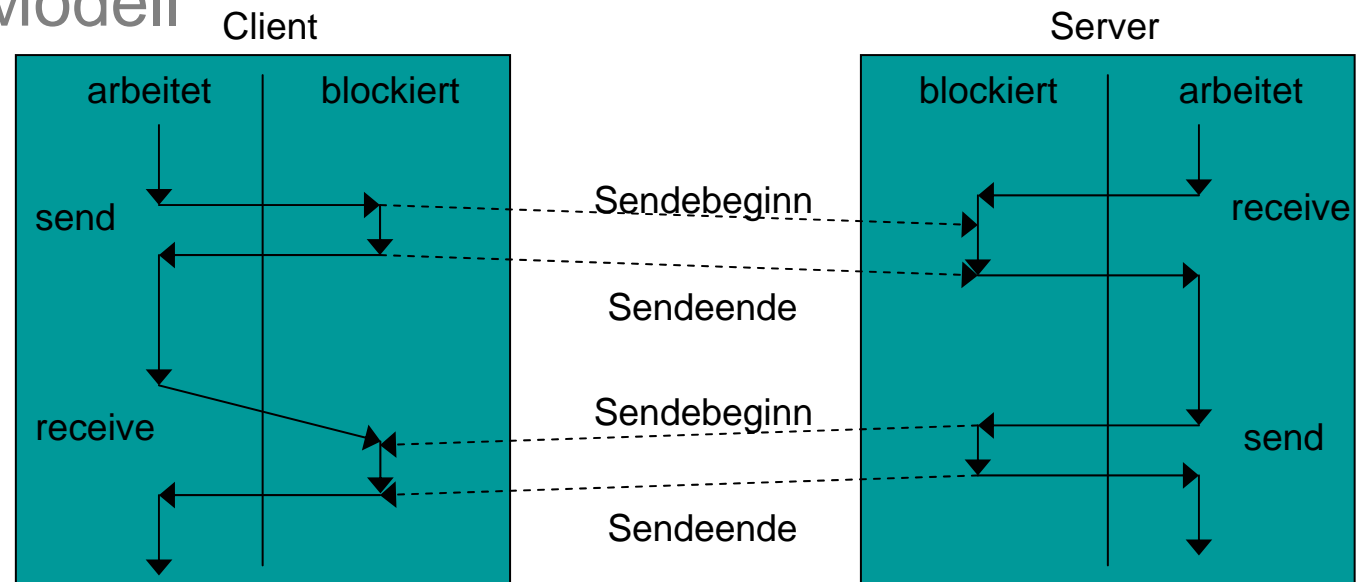
Verbindungsloses Anfrage-Antwort-Protokoll (Schichten 1, 2 und 7)

Direkte Adressierung des Servers durch den Client:

- Adresse eines Servers ist dem Client im Normalfall bekannt (Konstante)
- Einfachste Adressierung: machine.process
- Vereinbarungssache: z.B. Prozess 4102 auf IP-Adresse 137.226.12.142 durch 137.226.12.142.4102 oder 4102@137.226.12.142

11.2 Kommunikationsmechanismen / Das Client/Server-Modell

Ablauf der Kommunikation:



Die Kommunikation wird durch zwei Systemaufrufe geregelt:

`send(a, &mp)` verschickt eine durch `mp` referenzierte Nachricht an Prozess `a`.

Der Aufrufer wird blockiert, bis die Nachricht versendet ist.

`receive(a, &mp)` schreibt eine von `a` empfangene Nachricht in den durch `mp` referenzierten Puffer. Der Aufrufer wird blockiert, bis die Nachricht vollständig empfangen wurde.

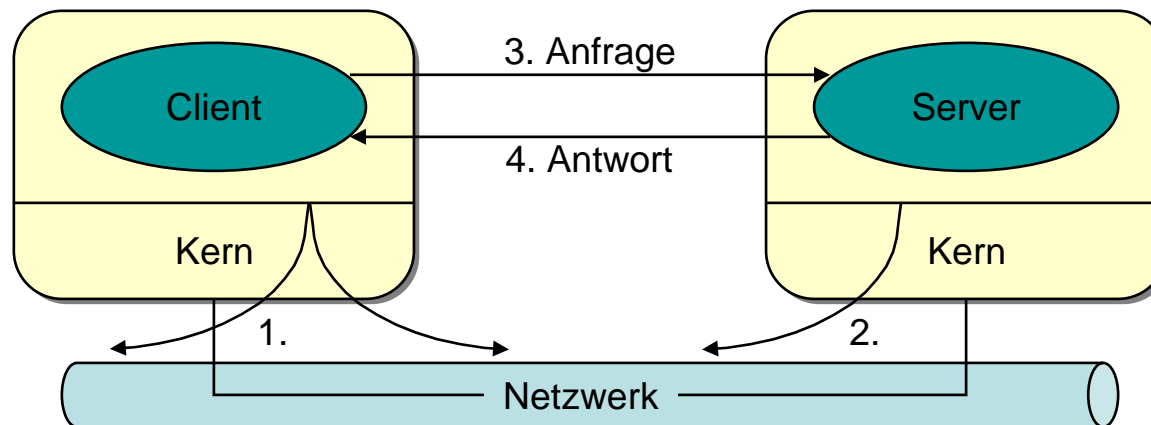
Benötigt wird eine Synchronisation von `send` und `receive`.

11.2 Kommunikationsmechanismen / Das Client/Server-Modell

Nachteil der Adressierung: keine Ortstransparenz, der Client muss Zielrechner kennen.

Lösungsansatz: Verwendung von **Lokalisierungspaketen**

1. Broadcasten eines Lokalisierungspakets für den Zielprozess durch den Client
2. „Ich bin hier“-Antwort des entsprechenden Servers
3. Anfrage an den Server
4. Antwort des Servers

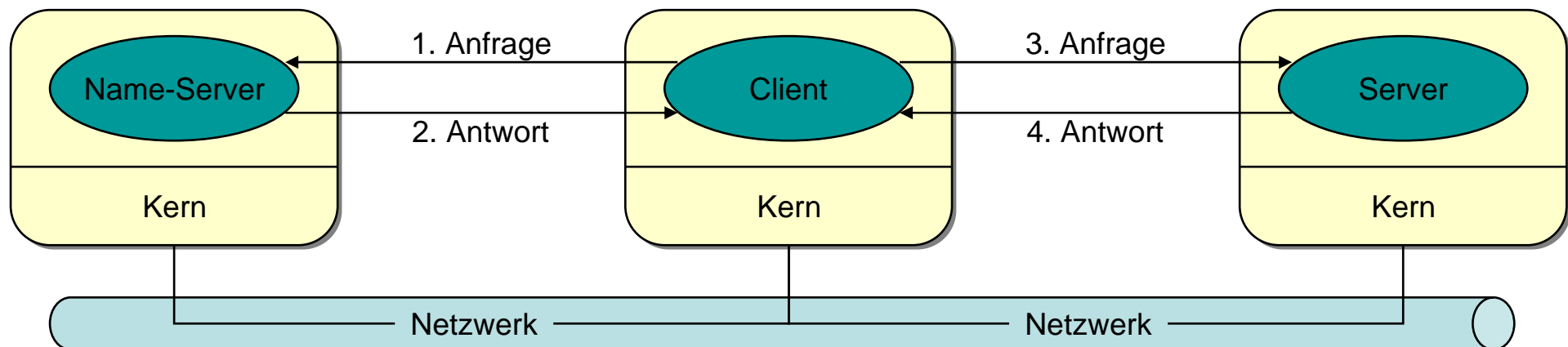


Nachteil: zusätzliche Last durch Broadcast-Aufruf

11.2 Kommunikationsmechanismen / Das Client/Server-Modell

Erweiterung des Client/Server-Modells (gängige Methode):
Verwendung eines **Name-Servers**

1. Anfrage des Clients nach gewünschter Adresse an einen Name-Server
2. Übermittlung der Server-Adresse an den Client
3. Anfrage an den Server
4. Antwort des Servers



11.2 Kommunikationsmechanismen / Das Client/Server-Modell

Entsprechend den Mechanismen unterscheidet man zwischen **blockierenden** und **nicht-blockierenden** Primitiven. Der Systementwickler kann zwischen diesen wählen.

Blockierende Primitive

- Blockierung eines Prozesses während des Sendens einer Nachricht
- Nachfolgende Anweisungen werden erst nach vollständiger Versendung weiter abgearbeitet
- Analog: Empfangen einer Nachricht

Nicht-blockierende Primitive

- Kopieren der zu sendenden Nachricht in einen Puffer des Betriebssystems
- Entblockierung des Prozesses
- Geschwindigkeitsvorteil durch parallele Nachrichtenversendung und weitere Abarbeitung
Nachteil: Der Sender weiß nicht, wann die Versendung beendet ist und der Puffer wieder freigegeben wird.

11.2 Kommunikationsmechanismen / Das Client/Server-Modell

Weiteres Unterscheidungsmerkmal: **Pufferung**

Nicht-puffernde Primitive

- Bei `receive(a, &mp)` wird der Kern informiert, dass der aufrufende Prozess die Adresse `a` abhören will und einen Speicherbereich bei `&mp` bereitstellt.
- Probleme:
 - Nachrichtenverlust bei verspätetem `receive`
Der Kern weiß nicht, wohin die Nachricht kopiert werden kann
 - Verwendung der gleichen Adresse durch verschiedene Prozesse

Puffernde Primitive

- Der Kern speichert die empfangene Nachricht eine bestimmte Zeit zwischen
- Problem: Der Kern muss einen Puffer bereitstellen und verwalten

11.2 Kommunikationsmechanismen / Der Remote Procedure Call (RPC)

- Basisparadigma jeglicher Kommunikation: **Senden / Empfangen** von Daten
- Geschieht durch expliziten Aufruf der Kommunikationsprimitiven `send` und `receive`
- Geeigneterer Mechanismus: Verteilte Berechnungen erscheinen wie lokale
- Lösung (Birell und Nelson, 1984): Ein Programm ruft ein Unterprogramm auf, das sich auf einem anderen Rechner befindet.

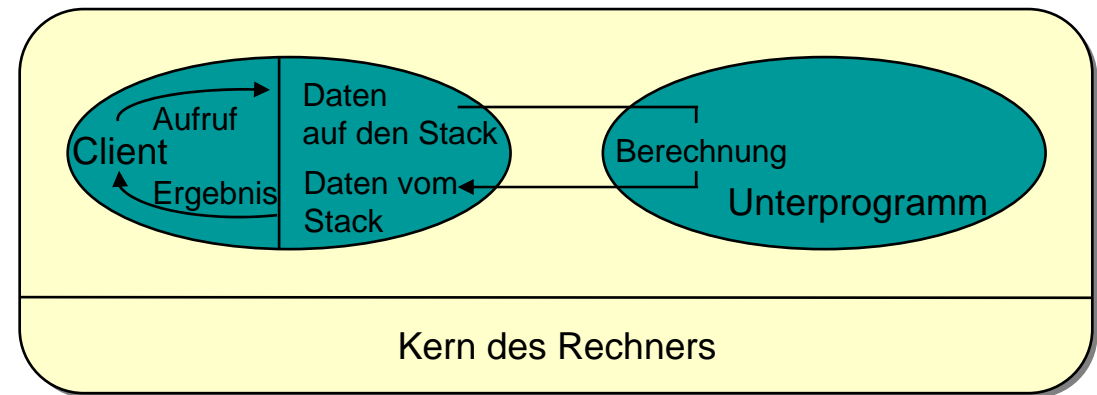
→ **Entfernter Unterprogrammaufruf (Remote Procedure Call, RPC).**

Funktionsweise:

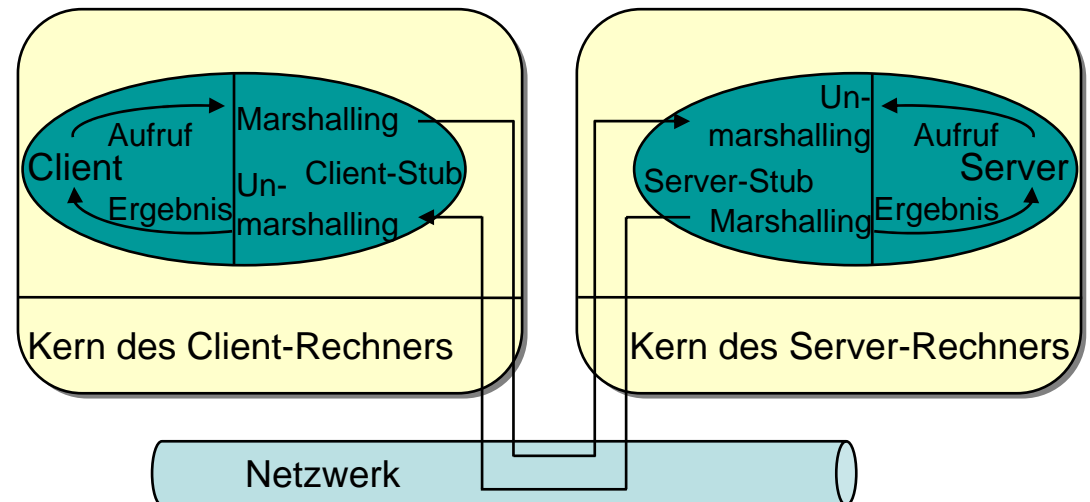
Ruft ein Programm auf Rechner A ein Unterprogramm auf Rechner B auf, wird der aufrufende Prozess auf A suspendiert und die Ausführung des Unterprogramms erfolgt auf B. Der Austausch von Parametern und Nachrichten bleibt für den Benutzer unsichtbar.

11.2 Kommunikationsmechanismen / Der Remote Procedure Call (RPC)

a) Lokaler Unterprogrammaufruf



b) Remote Procedure Call



Für den aufrufenden Client sind beide Mechanismen transparent.

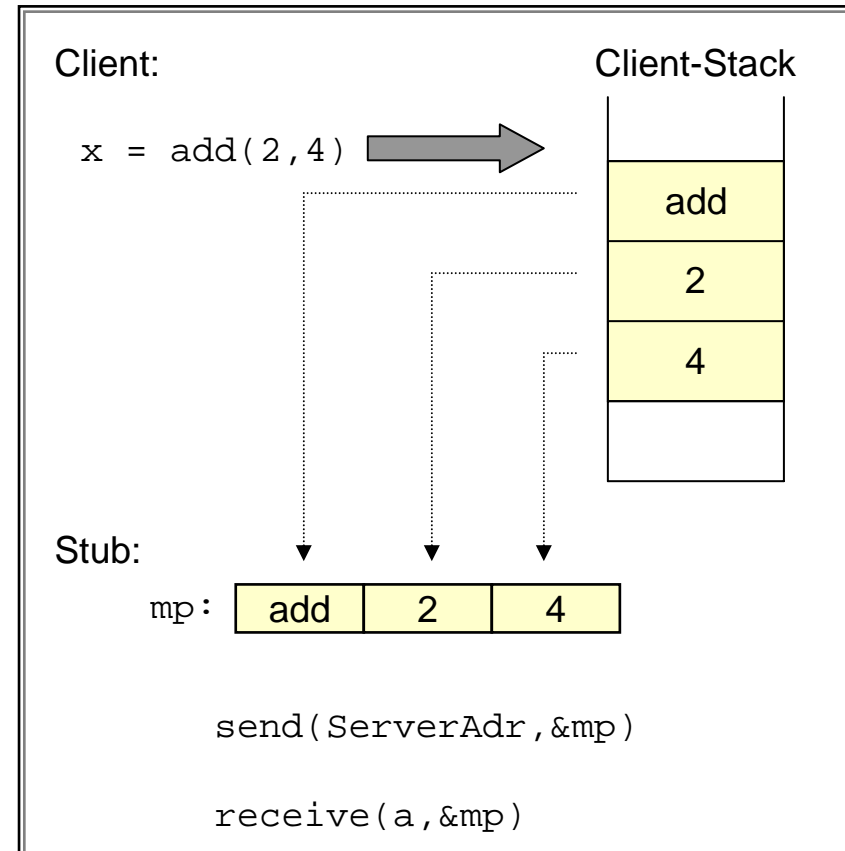
11.2 Kommunikationsmechanismen / Der Remote Procedure Call (RPC)

Client:

- Aufruf der Prozedur `add(2, 4)`
- Library des Clients enthält Referenz auf **Client-Stub** (Stellvertreterprozedur)
- Weiterleitung des Aufrufs an den Stub

Client-Stub:

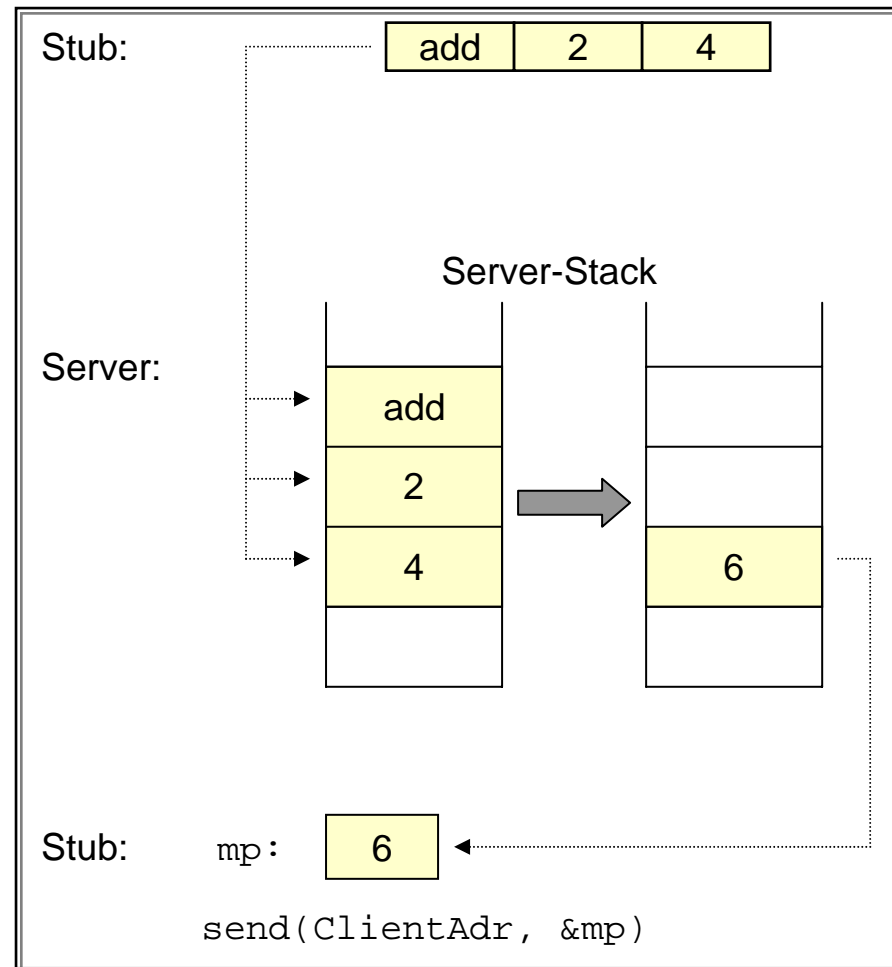
- Generiert aus der Anfrage eine Nachricht, die an den Server geschickt wird (Marshalling)
- durch `send-Primitive` wird der Kern zur Versendung veranlasst
- Stub ruft `receive` auf und blockiert



11.2 Kommunikationsmechanismen / Der Remote Procedure Call (RPC)

Server-Seite:

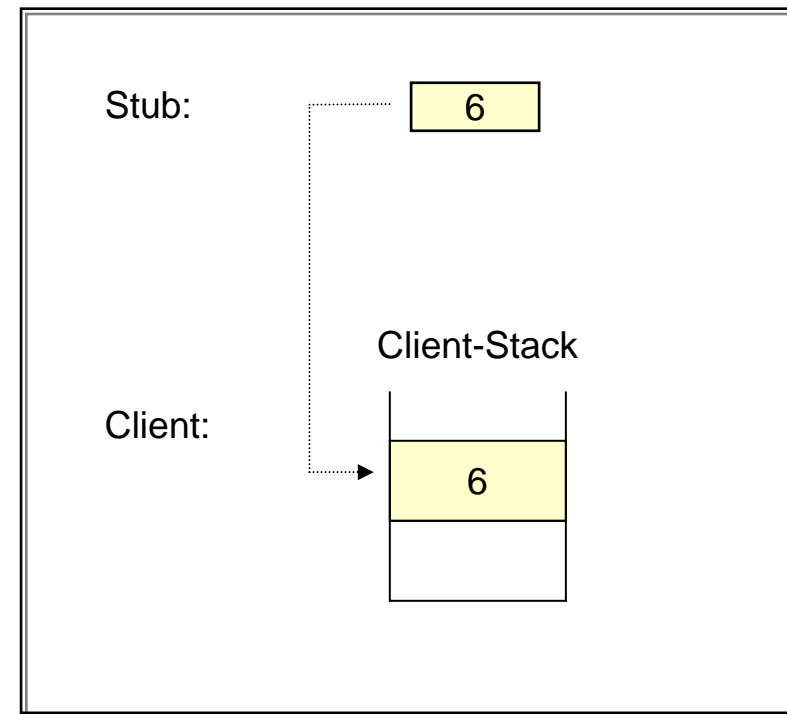
- Der Kern übergibt die empfangene Nachricht an den Server-Stub
- Der Server-Stub packt die Nachricht aus (Unmarshalling) und ruft lokal das Unterprogramm auf
- Nach Berechnung wird der Server-Stub wieder aufgerufen und verpackt das Ergebnis in eine Nachricht, die an den Client zurückgeschickt wird.



11.2 Kommunikationsmechanismen / Der Remote Procedure Call (RPC)

Client-Seite:

- Die Nachricht wird in einen Puffer geschrieben
- Der Client-Stub wird entblockiert
- Der Stub packt die Nachricht aus und legt das Ergebnis auf den Stack
- Der Client erhält die Kontrolle zurück und kann – wie bei lokalen Aufrufen – das Ergebnis vom Stack nehmen



Vorteile:

- Die verteilte Ausführung erfolgt **ohne expliziten Aufruf** der Kommunikationsprimitive.
- Details werden durch Stubs verborgen.

11.3 Kooperation und Koordination in verteilten Systemen

Probleme der Synchronisation und Kooperation treten auch in verteilten Systemen auf.

- Beispiele sind die Sicherstellung der Atomarität, Deadlockfreiheit, Konsistenzerhaltung bei Transaktionsbearbeitung, usw ...
- Verglichen mit den Problemen im Einprozessorfall müssen die verschiedenen Aspekte im Allgemeinen in verschärfter Form sichergestellt werden.

- Problem:

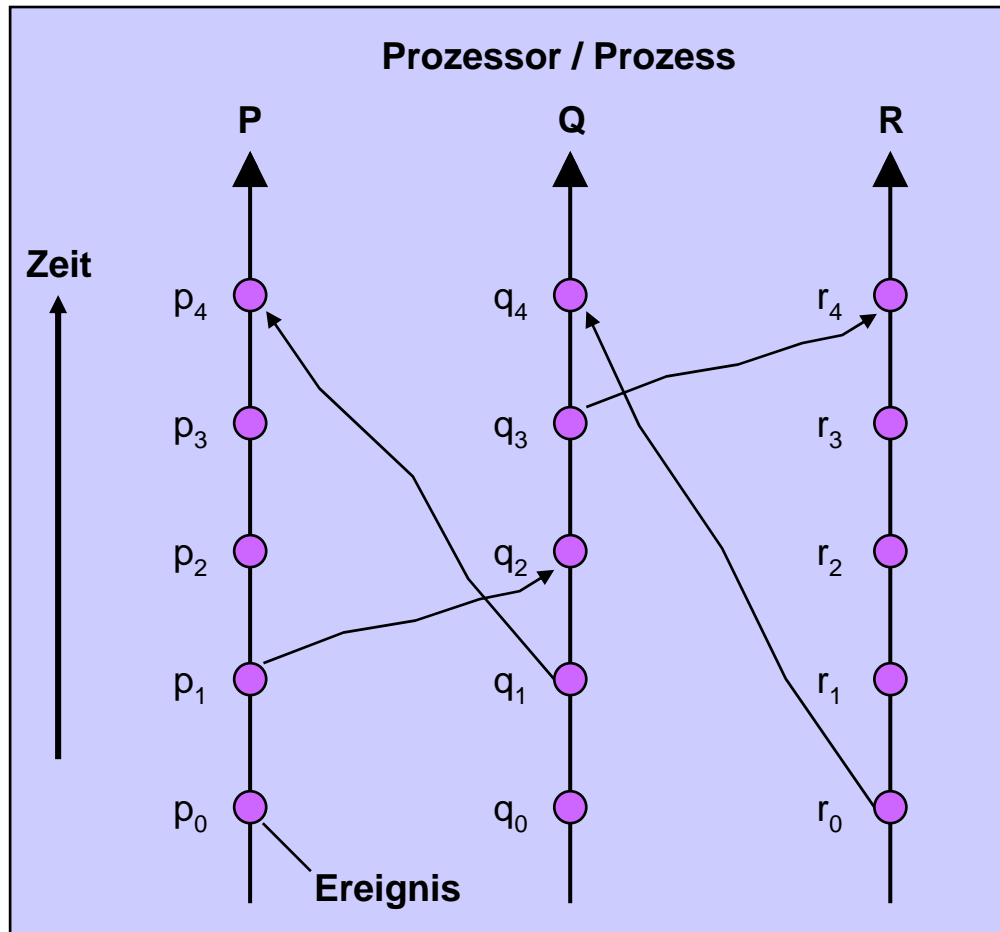
- Voneinander unabhängige und ungenau gehende Uhren in den einzelnen Komponenten eines verteilten Systems.
- ➔ Uhrensynchronisation notwendig

Wir nehmen an, dass innerhalb einer einzelnen Komponente die Reihenfolge der Ereignisse eindeutig ist.

$x \rightarrow y$ bedeute: Ereignis x **früher als Ereignis y**

11.3 Kooperation und Koordination in verteilten Systemen

Beispiel zur Reihenfolge von Ereignissen:



Beispiele der " \rightarrow "-Relation:

$$p_1 \rightarrow q_2$$

$$r_0 \rightarrow q_4$$

$$q_3 \rightarrow r_4$$

$$p_1 \rightarrow q_4, \text{ da } p_1 \rightarrow q_2 \text{ und } q_2 \rightarrow q_4$$

Ereignisse, die gleichzeitig ablaufen können:

$$q_0 \text{ und } p_2$$

$$r_0 \text{ und } q_3$$

$$r_0 \text{ und } p_3$$

$$q_3 \text{ und } p_3$$

$$\text{usw ...}$$

11.3 Kooperation und Koordination in verteilten Systemen

Implementierung einer " \rightarrow "-Relation ohne Voraussetzung genau gehender Uhren ist z.B. wie folgt durch Timestamps möglich:

- Ordne jedem Prozess P_i eine logische Uhr (**logical clock**) LC_i zu. Der Wert von LC_i ist der aktuelle **Timestamp** von P_i .
- LC_i kann eine ungenau gehende physikalische Uhr sein; Bedingung ist, dass die Werte von LC_i monoton wachsend sind.

Gegenseitige Anpassung der Uhren:

- P_k habe den aktuellen Wert LC_k und empfange eine Nachricht von P_i mit Wert LC_i .
- Ist $LC_k \leq LC_i$, dann geht die Uhr von P_k zu langsam (verglichen mit P_i), denn die Nachricht von P_i wurde früher losgeschickt und müsste einen kleineren Timestamp haben (wegen der stets positiven Laufzeit der Nachricht zwischen P_i und P_k).

Konsequenz:

- P_k erhöht den Zählerstand seiner Uhr auf $LC_{k_neu} = LC_i + 1$.

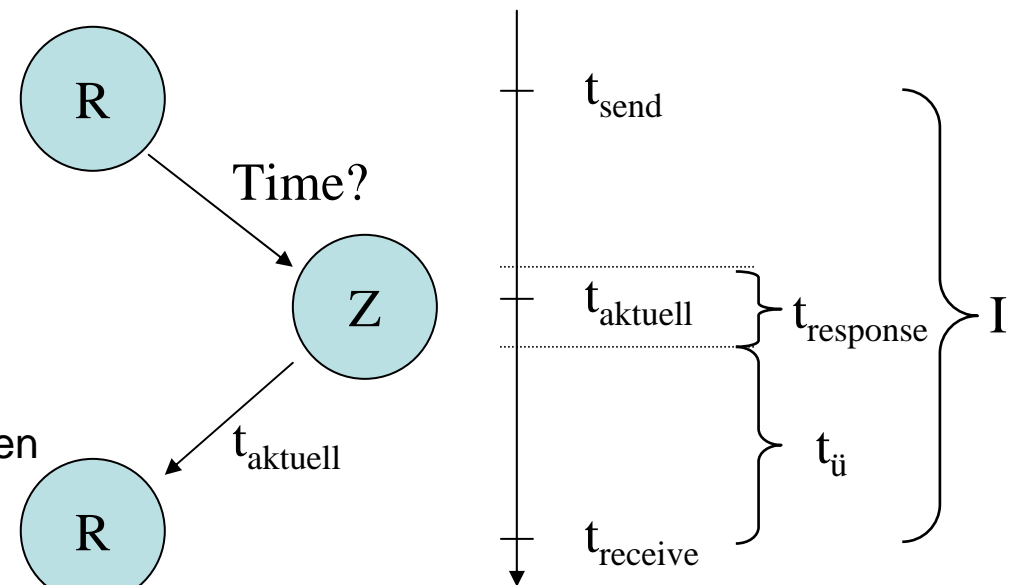
11.3 Kooperation und Koordination in verteilten Systemen / Uhrensynchronisation

Uhrensynchronisation nach Christian

Gegeben: **Zeitserver Z** mit genauer Zeit
Anfrage: Rechner R erfragt bei Z die Uhrzeit

Berechnung der aktuellen Uhrzeit durch R:

1. Merken der lokalen Uhrzeit t_{send} beim Absenden der Anfrage
2. Ermittlung des Zeitintervalls I bis zum Eintreffen der Antwort t_{aktuell} von Z durch $I = t_{\text{receive}} - t_{\text{send}}$
3. Subtraktion der Bearbeitungszeit von Z
4. Division durch 2
→ Übertragungszeit $t_{\text{ü}} = (I - t_{\text{response}})/2$
5. Synchronisierte Zeit: $t_{\text{aktuell}} + t_{\text{ü}}$



$$t_{\text{synchron}} = t_{\text{aktuell}} + \underbrace{\frac{t_{\text{receive}} - t_{\text{send}} - t_{\text{response}}}{2}}_{t_{\text{ü}}}$$

11.3 Kooperation und Koordination in verteilten Systemen / Wechselseitiger Ausschluss

Auch in verteilten Systemen tritt das wechselseitige Ausschlussproblem auf.

Lösung 1: Zentral gesteuertes Verfahren

- Einer der Prozesse des verteilten Systems spielt Koordinator C.

Wenn P_i in eine kritische Region eintreten will, sendet P_i eine **request Message** an C

- Falls KB frei, sendet C eine **reply Message** und P_i kann in den kritischen Bereich eintreten; andernfalls muss P_i warten.
- nach Verlassen des KB sendet P_i eine **release Message** an C. Danach kann C, z.B. nach FIFO, einen der wartenden Prozesse **aufwecken**.

Problem: Ausfall des Koordinators C.

- In diesem Fall muss ein neuer Koordinator gewählt werden.

11.3 Kooperation und Koordination in verteilten Systemen / Wechselseitiger Ausschluss

Lösung 2: Ohne Koordinator, dezentralisiert

Prozess P_i will in den **kritischen Bereich** eintreten:

- P_i sendet **request** (P_i , TS_i) an alle anderen. Dabei ist TS_i der aktuelle Timestamp von P_i .
- Ein Prozess P_k , der **request** (P_i , TS_i) erhält:
 - Sendet entweder eine **reply Message** an P_i zurück, wenn er nichts gegen den KB-Eintritt von P_i hat oder
 - er verschiebt das Senden der **reply Message** an P_i , z.B. dann, wenn er selbst im KB ist oder wenn er bereits länger wartet (siehe unten).
- Wenn reply Messages von allen anderen Stationen erhalten wurden, darf ein Prozess in den kritischen Bereich.

11.3 Kooperation und Koordination in verteilten Systemen / Wechselseitiger Ausschluss

Prozess P_i will in den **kritischen Bereich** eintreten (Fortsetzung):

- Nach Verlassen des KB werden **reply-Messages** an alle Prozesse versandt, für welche diese Nachricht bisher verschoben wurde.
- Wenn P_k eine Nachricht request (P_i, TS_i) erhält und selbst in KB will, vergleicht er seinen eigenen **request-Timestamp** TS_k mit TS_i . Wenn $TS_i < TS_k$, sendet er reply an P_i zurück, denn der Request von P_i ist **älter**. Andernfalls verschiebt er die reply Message.

11.3 Kooperation und Koordination in verteilten Systemen / Wechselseitiger Ausschluss

Dieses Verfahren

- +ist korrekt (bzgl. der Anforderungen des wechselseitigen Ausschlussproblems)
- +ist deadlockfrei
- +vermeidet **Starvation**, weil Timestamps monoton wachsen
- /benötigt $2(n-1)$ Nachrichten pro **KB-Eintritt**; dies ist die geringstmögliche Zahl von Nachrichten bei voll dezentralisierten Lösungen
- setzt voraus, dass jeder Prozess alle anderen **kennt**, d.h. hat Probleme, wenn Prozesse neu entstehen oder verschwinden
- funktioniert nur dann, wenn alle Prozesse korrekt arbeiten (ein besonders schwerwiegender Nachteil) und auch das Kommunikationsnetz funktioniert, d.h. alle Nachrichten müssen ankommen.
- hat hohen Overhead auch für Prozesse, die an KB unbeteiligt sind, d.h. diese müssen viele für sie nutzlose reply Messages erzeugen und senden

Konsequenz:

- Das Verfahren ist nur für eine kleine Anzahl von beteiligten Prozessen geeignet, und dann auch nur, wenn sich die Anzahl und Natur der Prozesse sehr selten ändert.

11.3 Kooperation und Koordination in verteilten Systemen / Wechselseitiger Ausschluss

Lösung 3: **Token-basierter Ansatz**

Ein Token ist ein spezielles Bitmuster, das sonst nicht vorkommt.

Methode:

- Vereinbare einen logischen Ring $P_{i1} \rightarrow P_{i2} \rightarrow P_{i3} \rightarrow P_{i4} \rightarrow \dots \rightarrow P_{in} \rightarrow P_{i1}$ und gebe die Berechtigung zum KB-Eintritt in dieser Reihenfolge weiter.

Sonderprobleme:

- Bei Tokenverlust muss ein neues Token erzeugt werden
- Tokenduplikate müssen bemerkt und vernichtet werden
- Bei Ausfall von Prozessen muss der logische Ring neu konfiguriert werden.
- Ein neuer Prozess muss eine faire Chance erhalten, sich in den logischen Ring einzuklinken.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

Aspekte der (atomaren) Transaktionsverarbeitung in verteilten Systemen:

- Datenbestände können in verschiedenen Komponenten eines verteilten Systems gehalten werden.
- Einzelne Daten können
 - nur an einer einzigen Stelle vorliegen
 - an verschiedenen Orten repliziert vorliegen
 - ➔ der Zugriff wird vereinfacht, aber die Konsistenzerhaltung erschwert
- Die Bearbeitung einer Transaktion kann Teilaufgaben beinhalten, die auf unterschiedlichen Komponenten des verteilten Systems ausgeführt werden.
 - Eine Komponente ist Initiator und verantwortlich für das Einsammeln und Koordinieren von Ergebnissen. Sie ist außerdem zuständig für die Bereitstellung von Informationen, ob die Transaktion „committed“ oder „aborted“ wird.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

Jede Komponente des verteilten Systems hat einen lokalen Transaktions-Koordinator, der:

- alle von dieser Stelle ausgehenden TA's startet
- die TA's in Sub-TA's aufteilt und den zuständigen anderen Komponenten zuteilt
- den Abschluss einer TA koordiniert und
 - im erfolgreichen Fall eine **commit-Message** an alle Beteiligten versendet
 - im negativen Fall eine **abort-Message** verschickt.

Jede Systemkomponente führt ein Logfile für ggf. notwendige Recovery-Operationen. Wir setzen der Einfachheit halber voraus, dass alle Daten auf stabilem Speicher gehalten sind, d.h. garantiert wieder zugänglich sind.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

Das **Two-Phase Commit-Protokoll (2PC)**:

Es wird garantiert, dass eine Transaktion T bei allen beteiligten Stellen entweder gemeinsam **committed** oder gemeinsam **aborted** wird.

Seien

- T eine von S_i initiierte Transaktion und
- C_i der lokale Koordinator von S_i .

Beim **2PC** Protokoll wartet C_i auf die Antworten aller unterbeauftragten Komponenten; sofern eine Antwort allzu lange ausbleibt (Timeout), wird die Transaktion abgebrochen (abort).

- Timeout deutet auf ein Problem beim Kommunikationsnetz oder der Zielstation hin.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

2PC - Phase 1:

C_i erzeugt eine `<prepare T>`-Nachricht und sendet sie an alle Beteiligten.

Wenn ein Prozess `<prepare T>` erhält

- und er nicht einverstanden ist, sendet er `<abort T>` an S_i zurück,
- andernfalls sendet er `<ready T>` an S_i zurück.

2PC - Phase 2:

Wenn C_i innerhalb des Timeout-Intervalls `<ready T>` von allen Beteiligten erhält, kann ein commit erfolgen → C_i sendet `<commit T>` an alle.

Andernfalls sendet C_i eine `<abort T>`-Nachricht an alle.

Bei manchen Implementierungen wird im **Commit-Fall** eine positive Quittung von den Beteiligten erwartet. Aber auch durch noch so viele Quittungen kann keine absolute Zuverlässigkeit erhalten werden, da die Übertragungsstrecke potenziell unzuverlässig ist. Auf die letzte Quittung kann man sich nicht unbedingt verlassen: Wäre dem so, müsste es auch mit einer Quittung weniger gehen, dann könnte man aber auch die vorletzte weglassen usw.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

Fehlerbehandlung im 2PC-Protokoll:

- Ein naheliegender Schwachpunkt ist die entscheidende Funktion von C_i bzgl. der Frage, ob T committed werden kann oder aborted werden muss. Ein Ausfall von C_i kann also zum Systemstillstand führen.

1. Ausfall eines Unterauftragnehmers S_k : Sei T eine TA, an der S_k beteiligt war.

Wenn S_k wieder on-the-air ist, wird das Logfile von T überprüft:

- a) Wenn das File `<commit T>` enthält:

➔ S_k führt `redo(T)` aus.

- b) Wenn das File `<abort T>` enthält:

➔ S_k führt `undo(T)` aus.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

c) Wenn das File `<ready T>` enthält:

- S_k fragt C_i bzgl. der Entscheidung über T .
 - C_i antwortet → Entscheidung übernehmen
 - C_i antwortet nicht → andere Beteiligte werden gefragt, indem eine `<query-status T>`-Nachricht an alle gesendet wird; an alle, weil man nicht weiß, welche anderen Stationen beteiligt waren.

d) Wenn das File weder `<commit T>` noch `<abort T>` noch `<ready T>` enthält, ist S_k ausgefallen, bevor eine positive Antwort an C_i gegeben werden konnte. Das Ergebnis von C_i musste also `<abort T>` gewesen sein.

→ S_k führt `undo(T)` aus

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

2. Ausfall des Koordinators C_i : In diesem Fall müssen die beteiligten Stationen, nach Timeout-Ablauf, über den Ausgang der Transaktion T entscheiden:

- Wenn eine Station S_k `<commit T>` in ihrem Logfile hat, kann sie T committen und das Ergebnis an alle weitermelden.
- Wenn eine Station S_k `<abort T>` in ihrem Logfile hat, muss sie T aborten und das Ergebnis an alle weitermelden.
- Wenn eine beteiligte Station S_k kein `<ready T>` in ihrem Logfile hat, kann C_i die Transaktionen T nicht committed haben, weil ihr mindestens eine der notwendigen Rückmeldungen fehlte. Konsequenz ist ein `<abort T>`.
- Wenn keiner der bisherigen Fälle zutrifft, haben alle beteiligten Stationen `<ready T>` gegeben, aber keine Antwort `<commit T>` oder `<abort T>` erhalten. Es muss darauf gewartet werden, bis C_i wieder on-the-air ist - was lange dauern und unangenehme Konsequenzen haben kann → Blockierung über unvorhersehbare Zeiträume.

11.3 Kooperation und Koordination in verteilten Systemen / Atomare Transaktionsverarbeitung

3. Ausfall von Bestandteilen des Kommunikationsnetzes:

Solange alle an der TA beteiligten Komponenten noch physikalisch miteinander verbunden sind:

→ Entscheidung kann wie bisher erfolgen

Wenn mindestens eine Komponente nicht mehr erreichbar ist:

→ `<abort T>` ausführen.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

Bisherige Concurrency-Control-Protokolle können für die Verwendung in einem verteilten System zuverlässig modifiziert werden

- Das sind Sperrprotokolle (**Locking Protocols**) wie z.B. **Two-Phase-Locking** etc.

Sperrprotokolle

- Unterscheidung zwischen Sperrmethoden bei **nicht-replizierten** und **replizierten** Daten.
- Unterscheidung zwischen **shared-locks** und **exclusive-locks**. Wir beschränken uns hier auf den Fall der exclusive-locks; shared-locks bieten zusätzliche Möglichkeiten bzw. eine Reihe von Erleichterungen.

Sperrmethoden bei nicht-replizierten Daten

Jede Station hat einen Lock Manager, der die Sperren zuteilt. Verfahren wie bisher.

Problem ist die komplizierte Deadlock-Behandlung.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

Sperrmethoden bei replizierten Daten

1. Sperren durch **zentralen Koordinator**:

Der zentrale Koordinator ist Bestandteil einer einzigen Station, z.B. von Station S_i .

Alle Sperranforderungen werden von dort koordiniert.

- Wenn die Sperre gewährt werden kann, erfolgt entsprechende Rückmeldung auf die Sperranfrage.
- Andernfalls: warten!

Bei Gewährung einer Sperre:

- Der Besitzer dieser Sperre darf die betreffenden Daten von irgendeiner Station lesen, auf der sie gespeichert sind.
- Ein schreibender Zugriff muss bei allen Stationen durchgeführt werden, bei denen Replikate dieses Datums vorliegen.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

Eigenschaften des **Zentraler-Koordinator-Ansatzes**:

- + einfach zu realisieren; 2 Nachrichten zum Sperren; 1 Nachricht zum Entsperren.
- + Deadlockbehandlung wie im Ein-Prozessorfall.
- die Station S_i des zentralen Koordinators kann zum Engpass (**Bottleneck**) werden.
- Ausfall von S_i hat schwerwiegende Folgen.

Kompromiss:

- Nicht ein einziger, sondern mehrere Koordinatoren, die sich gegenseitig abstimmen und bei Bedarf für ausfallende **Kollegen** einspringen.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

2. Majoritätsentscheidungen:

Wenn ein Datum gesperrt werden soll, das auf n Stationen repliziert vorliegt, müssen Sperrberechtigungen von mehr als $n/2$ dieser Stationen eingeholt werden (vom jeweiligen Sperrmanager dieser Stationen).

+ Dezentralisierter Ansatz ohne das Bottleneck-Risiko

➤ Zahlreiche auszutauschende Nachrichten:

$$2 \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \quad \text{Nachrichten für Sperranforderung}$$

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \quad \text{Nachrichten für Sperrfreigabe.}$$

➤ Schwierigere Deadlockbehandlung: Es kann sogar vorkommen, dass ein Deadlock bei nur einer einzigen Sperranforderung entsteht. Dies war bisher nicht möglich.
Beispiel: Ein Datum Q ist geradzahlig oft repliziert und zwei Prozesse haben jeweils $n/2$ Sperrzusagen für Q eingesammelt.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

Primäre und sekundäre Datenhaltungsorte:

- Im Fall replizierter Daten kann man primäre und sekundäre Datenhaltungsorte kennzeichnen und Sperrwünsche auf die primäre Kopie beschränken (das Datum dann aber von irgendwoher lesen).
 - ➔ Einfacher als bisher, aber sehr unangenehm bei Ausfall des primären Standorts

Timestamp Ordering für die Serialisierung von Transaktionen:

- Timestamps müssen eindeutig sein.

Möglichkeiten zur Erzeugung eindeutiger Timestamps:

1. Timestamp-Erhalt von einer zentralen Stelle
2. Timestamp lokal eindeutig vergeben und als „Tie-Break“ die eindeutige Stationsnummer anhängen (nicht voranstellen, wegen der Fairness!)

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

Wenn bei einer Station die lokale Uhr zu schnell oder zu langsam geht und diese Station sich dadurch Vorteile verschaffen würde, können die lokalen Uhren wie früher beschrieben synchronisiert werden:

- Wenn ein Timestamp x empfangen wird, der größer als der eigene Timestamp ist, setze eigenen Timestamp auf $x+1$.

Serialisierung von Transaktionen durch Ordnung nach wachsenden Timestamps
→ siehe Einprozessorfall

Problem:

- Cascading-Rollbacks sind möglich, wenn auf Daten zugegriffen wurde, die noch nicht committet wurden.
→ Kann durch strenge und möglichst frühzeitige Anwendung von 2PC vermieden werden.

11.3 Kooperation und Koordination in verteilten Systemen / Concurrency-Control

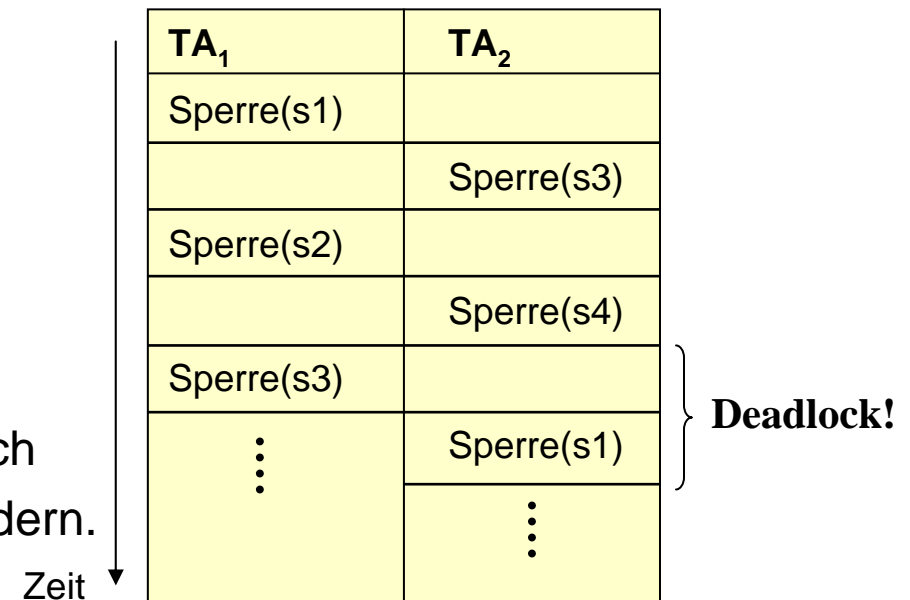
Konflikte zwischen Transaktionen:

- Bei Simple 2PL (Two Phase Locking) durch Rollback gelöst. Nachteil: viele Konflikte!
- Besser: „Wait and See“, d.h. Warte mit Lese- und Schreiboperationen solange bis feststeht, dass sie ohne Rollback ausgeführt werden können.
 - T_i muß $\text{read}(x)$ verschieben, wenn es eine TA T_k gibt, die noch $\text{write}(x)$ auszuführen hat und für die $TS(T_k) < TS(T_i)$ gilt, d.h. T_k älter als T_i ist.
 - T_i muß $\text{write}(x)$ verschieben, wenn es eine TA T_k gibt, die noch entweder $\text{write}(x)$ oder $\text{read}(x)$ auszuführen hat und für die $TS(T_k) < TS(T_i)$ gilt.
 - Eine Möglichkeit zur Feststellung dieser Wartebedingungen besteht darin, die noch ausstehenden read- oder write-Zugriffe einer Station in entsprechenden Warteschlangen zu speichern.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Deadlock Prevention:

- Konzept der BM-Hierarchie funktioniert auch in verteilten Systemen: Sperranforderungen nur **nach oben** zulässig.
- **Nach unten bzw. auf gleicher Höhe:**
zuerst Rückgabe der BM bis einschließlich zur angeforderten Höhe, dann neu anfordern.



Banker's Algorithmus ist ebenfalls auf verteilte Systeme übertragbar

- Der **Banker** wird an einer speziellen Station installiert und die Deadlockfreiheit bzw. die Sicherheit von Systemzuständen überprüft.
- Problem: Der Banker wird sehr schnell überlastet und zum Systemengpass!

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Prioritätsmethoden:

- Jedem Prozess wird eine eindeutige Prioritätsnummer zugeteilt.

Regelung:

- Prozesse mit höherer Priorität dürfen warten, wenn die BM von Prozessen niedriger Priorität belegt sind.

Andernfalls:

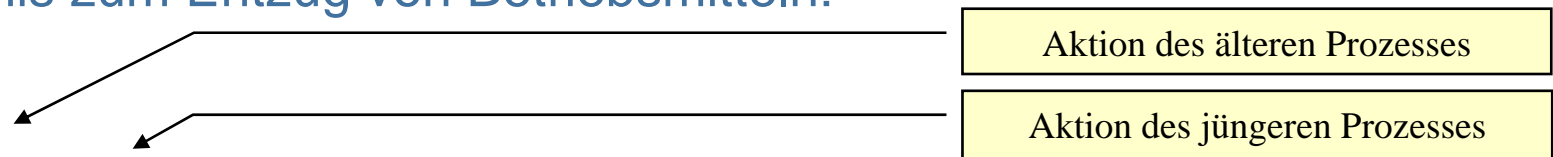
- Rollback des betreffenden Prozesses. Wegen eindeutiger Prioritätsnummer: keine Zyklen möglich, also auch kein Deadlock.

Problem:

- Dauernde Benachteiligung und Starvation von Prozessen niedriger Priorität ist möglich. Wird durch die im Folgenden angegebenen auf Timestamps basierenden Methoden verhindert!

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Verbesserte Methoden: **Deadlock Prevention mit Timestamp-Ordering** und
ggf. der Erlaubnis zum Entzug von Betriebsmitteln:



Methode 1: **Wait-Die**; non-preemptive:

- Wenn P_i ein aktuell durch P_k belegtes BM anfordert, darf P_i genau dann warten, wenn
 $TS(P_i) < TS(P_k)$ gilt, d.h. wenn P_i älter ist als P_k .
- Andernfalls wird P_i zurückgesetzt (P_i **dies**).

Der ältere Prozess wartet auf die BM-Freigabe, wenn jüngere Prozesse die BM halten! Jüngerer Prozess „stirbt“; kann „mehrfach“ sterben, wenn älterer Prozess immer noch nicht fertig ist.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Methode 2: **Wound-Wait**; preemptive:

- Wenn P_i ein aktuell durch P_k belegtes BM anfordert, darf P_i genau dann warten, wenn $TS(P_i) > TS(P_k)$ gilt, d.h. wenn P_i jünger ist als P_k .
- Andernfalls wird P_k zurückgesetzt (**P_k is wounded by P_i**) und die bisher von P_k gehaltenen BM werden ihm entzogen.

Der ältere Prozess wartet nie, sondern entzieht dem jüngeren die Betriebsmittel sofort. Wenn P_k durch P_i zurückgesetzt wurde, darf P_k nach einem Neustart auf das jetzt von P_i gehaltene Betriebsmittel warten.

Konsequenz: Geringere Rollbackanzahl bei **Wound-Wait** als bei **Wait-Die**.

Wenn der Timestamp einer Transaktion auch bei einem Rollback nicht erneuert wird, ist Starvation unmöglich, da ein gegebener Timestamp irgendwann zum aktuell ältesten wird und die Transaktion kein Rollback mehr durchführen muss.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Methode 3: **Dezentrale Wahl eines Koordinators**

Die Bestimmung eines Koordinators basiert auf der Existenz einer eindeutigen Nummer für jeden aktiven Prozess.

→ Höhere Nummer = höhere Priorität.

Wahlverfahren 1: (für vollvermaschte Systeme, d.h. jeder kann direkt an jeden senden)

- Der Ausfall des bisherigen Koordinators wird dadurch bemerkt, dass dieser keine Aktivität während eines hinreichend langen Timeout-Intervalls gezeigt hat.
- Die Reservekandidaten, z.B. P_i , bemerken dies und bewerben sich um die Stelle des Koordinators.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

P_i sendet zu diesem Zweck eine `<election- P_i >`-Nachricht an alle Prozesse mit höherer Nummer.

- Falls keine Antwort innerhalb eines Timeouts kommt, nimmt P_i an, dass alle Prozesse mit höherer Nummer ausgefallen sind oder dass keine solchen Prozesse existieren.
 - ➔ P_i erklärt sich zum neuen Koordinator, indem eine entsprechende Nachricht an alle Prozesse mit niedrigerer Nummer gesendet wird.
- Falls eine Antwort eintrifft, wartet P_i , ob sich ein Prozess mit höherer Nummer zum Koordinator erklärt. Wenn dies nicht erfolgt, geht P_i davon aus, dass dieser Prozess inzwischen ausgefallen ist und der Wahlversuch wird wiederholt.

Bei P_i können im Laufe des Wahlvorgangs Nachrichten eintreffen von:

- P_k (mit $k > i$), die besagen, dass P_k neuer Koordinator ist.
- P_r (mit $r < i$), die besagen, dass P_r neuer Koordinator werden will; in diesem Fall teilt P_i der Station P_r mit, dass P_i , also eine Station mit höherer Priorität, dasselbe vor hat.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Der Wahlvorgang wird in jedem Fall wiederholt, wenn ein zwischenzeitlich ausgefallener Prozess wieder einsatzbereit ist:

- Auch dann, wenn ein funktionierender Koordinator existiert.
- Auch dann, wenn der ausgefallene Prozess eine niedrige Priorität hat. Die höherpriorären könnten ja ausgefallen sein.

➔ Viel überflüssiger Aufwand!

Wahlverfahren für logische oder physikalische Ringe:

- a) Die Reservekandidaten entdecken den Ausfall des Koordinators.
- b) Alle Stationen P_i senden eine `<elect- P_i >`-Nachricht an ihre rechten Ringnachbarn.
- c) Empfangene Nachrichten mit höherer Priorität werden durchgelassen und weiter gesendet, solche mit kleinerer Nummer werden vernichtet. Wer seine eigene Nachricht zurückerhält, ist neuer Koordinator.

11.3 Kooperation und Koordination in verteilten Systemen / Deadlock-Behandlung

Etwas verfeinertes Verfahren: Es wird nicht vorausgesetzt, dass alle den Ausfall des Koordinators bemerken.

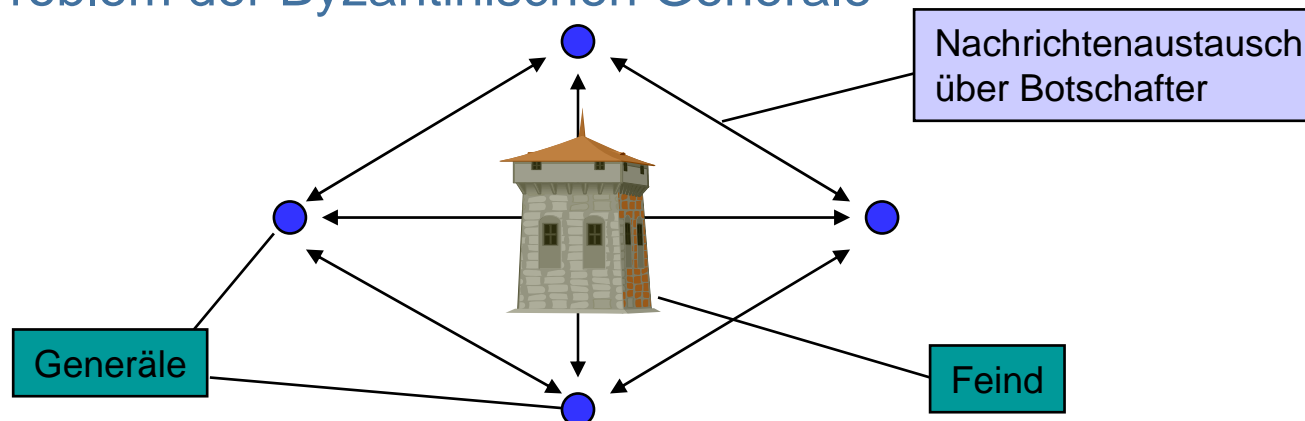
- a) P_r bemerkt den Ausfall und sendet $\langle \text{elect}-P_r \rangle$ an seinen rechten Nachbarn.
Wenn P_i solch eine Nachricht erstmalig empfängt, sendet P_i nacheinander $\langle \text{elect}-P_r \rangle$ und $\langle \text{elect}-P_i \rangle$.
- b) Empfangene Nachrichten $\langle \text{elect}-P_k \rangle$ mit bisher unbekanntem P_k werden in die Liste potenzieller Kandidaten aufgenommen.
- c) Empfang der eigenen Nachricht $\langle \text{elect}-P_r \rangle$ besagt, dass jetzt alle aktiven Stationen bekannt sind; daraus kann der aktuelle Koordinator berechnet werden.

11.3 Kooperation und Koordination in verteilten Systemen / Abstimmung bzgl. eines **wahren** Werts

Es sind Mechanismen notwendig, mit denen Prozesse sich über den **wahren Wert** eines Datums verständigen können. Dies ist ein nicht-trivialer Vorgang,

- weil das Kommunikationssystem inhärent unzuverlässig ist, d.h. Nachrichten können nicht oder fehlerhaft ankommen.
- weil Prozesse fehlerhaft arbeiten können. Eventuell können sogar Prozesse böswillig zusammenarbeiten, um ein Ergebnis gemeinsam zu verfälschen.

Beispiel: Problem der Byzantinischen Generale



11.3 Kooperation und Koordination in verteilten Systemen / Abstimmung bzgl. eines **wahren** Werts

Problem der Byzantinischen Generale:

- Mehrere Byzantinische Generäle umzingeln ein feindliches Lager und überlegen, ob sie es angreifen sollen oder nicht. Ein Angriff kann nur erfolgreich sein, wenn er gemeinsam ausgeführt wird.
- Nachrichten zwischen den Generälen werden über unzuverlässige Botschafter ausgetauscht.
- Botschafter können abgefangen werden. Dies entspricht einem Nachrichtenverlust aufgrund eines unzuverlässigen Kommunikationssystems.
- Einige Generäle können Verräter sein. Dies entspricht fehlerhaft arbeitenden Prozessen.

Unzuverlässige Kommunikationskanäle:

- Die potenzielle Ungewissheit kann letztendlich durch keine noch so aufwendige Folge von Quittungsnachrichten beseitigt werden!

Fehlerhaft arbeitende Prozesse:

- Was tun, wenn einige Prozesse fehlerhaft arbeiten, aber wenigstens das Kommunikationssystem zuverlässig ist?

11.3 Kooperation und Koordination in verteilten Systemen / Abstimmung bzgl. eines **wahren** Werts

Gegeben seien n Prozesse, unter diesen sollen maximal m fehlerhaft arbeiten. Von einem gegebenen Datum habe jeder Prozess P_i einen **privaten** Wert V_i .

Ziel: Jeder fehlerfreie Prozess P_i soll folgenden Vektor X_i bestimmen: $X_i = (X_{i1}, X_{i2}, \dots, X_{in})$ mit der Eigenschaft:

- Wenn P_r nicht fehlerhaft ist, dann ist $X_{ir} = V_r$.
- Wenn P_r und P_k nicht fehlerhaft sind, gilt $X_{ir} = X_{ik}$. (für die korrekten Komponenten)

Alle bekannten Lösungen für dieses Problem - zum Teil Algorithmen, die sehr „tricky“ sind - haben die folgenden Eigenschaften:

- 1) Damit ein korrekter Algorithmus existiert, muss gelten: $n \geq 3m + 1$, d.h. die Anzahl der Verräter muss kleiner als ein Drittel der Gesamtzahl sein.
- 2) Im **worst case** müssen $m+1$ Nachrichtenlaufzeiten abgewartet werden bis das Agreement hergestellt ist.
- 3) Die Gesamtzahl der insgesamt auszutauschenden Nachrichten ist sehr groß. Jeder Prozess muss alle Nachrichten sammeln und den Algorithmus für sich allein ausführen, da er keinem anderen trauen kann.

11.3 Kooperation und Koordination in verteilten Systemen / Abstimmung bzgl. eines **wahren** Werts

Einfachstes Beispiel: $n = 4$, $m = 1$, d.h. ein einziger Verräter.

Algorithmus besteht aus zwei **Runden**:

- Runde 1: Sende den eigenen Wert zu allen drei anderen Prozessen.
- Runde 2: Sende die gesammelten Informationen der Runde 1 an alle anderen Prozesse.

Die **Information** des Verräters kann ausbleiben oder falsch sein; sie kann - wenn sie ausgeblieben ist - durch einen zufälligen Wert ersetzt werden.

Ein Nichtbetrüger P_i kann jetzt wie folgt seinen Vektor $X_i = (X_{i1}, X_{i2}, X_{i3}, X_{i4})$ berechnen:

- $X_{ii} = V_i$
- X_{ir} ($r \neq i$): Mindestens zwei der drei gemeldeten Werte müssen übereinstimmen, da zwei Nichtbetrüger dabei sind. Setze X_{ir} auf diesen Wert ➔ **majority decision**

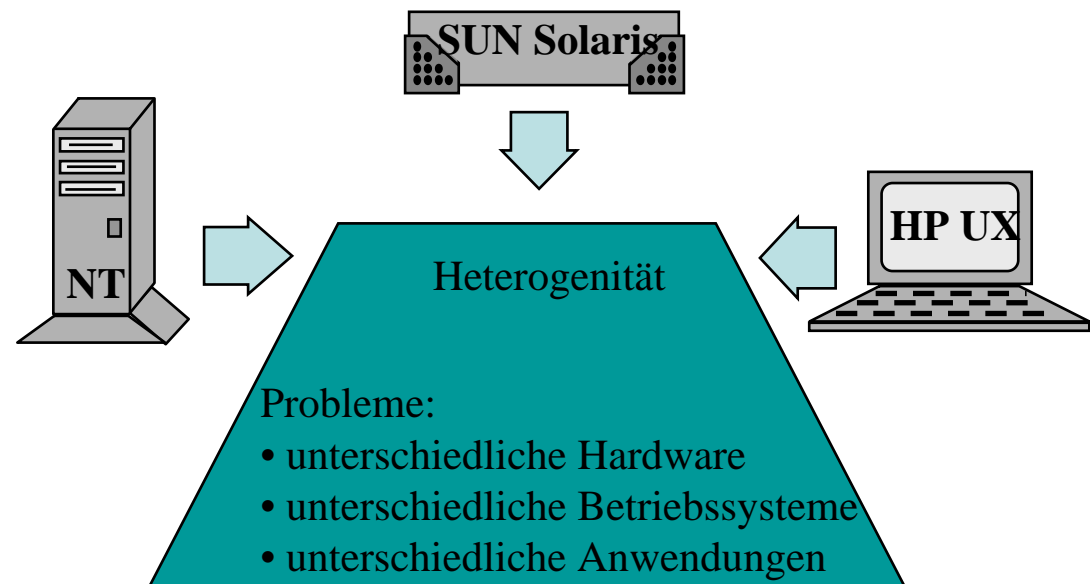
11.4 Verteilungsplattformen

Vorteile verteilter Systeme:

- Einfaches Hinzufügen neuer Komponenten
- Integration existierender Lösungen
- Autonomie einzelner Komponenten
- Flexibilität und Anpassbarkeit

Anforderungen an verteilte Systeme:

- Offenheit, Integrierbarkeit
- Modularität, Föderation
- Flexibilität, Sicherheit
- Verwaltbarkeit, Transparenz



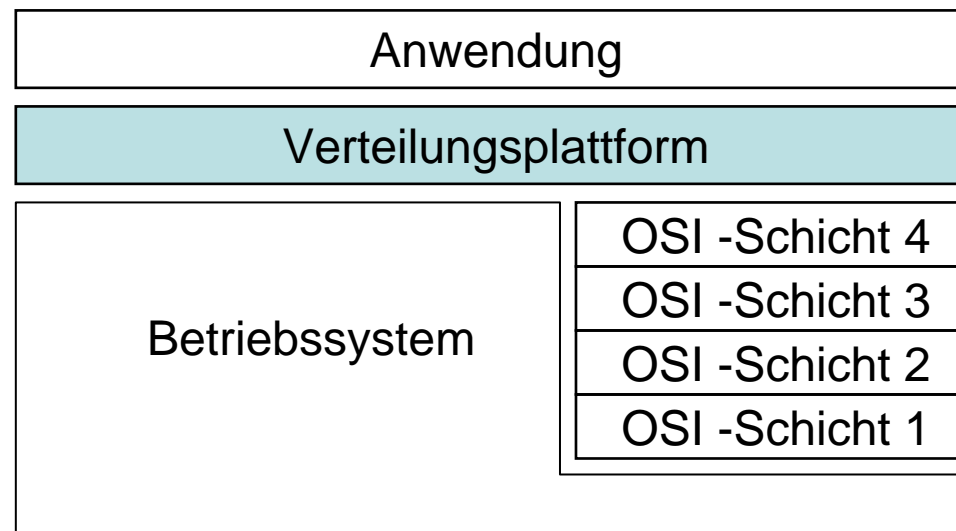
Lösung:

Verteilungsplattformen als Softwareinfrastruktur zur Überbrückung von Verteilung

11.4 Verteilungsplattformen

Verteilungsplattform (auch Netzbetriebssystem, Middleware):

- Realisierung verteilter Zugriffe durch geeignete Softwareinfrastruktur
- Verbergen der Komplexität des verteilten Systems vor dem Programmierer
- Unterstützung der Interaktion zwischen potenziell auf heterogenen Systemen ablaufenden Anwendungskomponenten
- Trennung von Schnittstelle und Implementierung
- Wird lokalem Betriebssystem hinzugefügt oder übernimmt dessen Aufgaben



11.4 Verteilungsplattformen

Verteilungsplattformen unterstützen

- die Kommunikation zwischen Anwendungskomponenten (oft RPC-basiert),
- die Erstellung modularer Anwendungen,
- dynamisches Binden an Server,
- Zusatzdienste, z.B.
 - Name-Server
 - Uhrensynchronisation
 - Transaktionsverwaltung

Wichtige Vertreter:

- Distributed Computing Environment (DCE)
- ANSAware
- Common Object Request Broker Architecture (CORBA)

