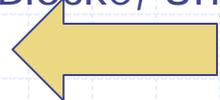


# Betriebssysteme      Vorlesung 11

Verteiltes Prozeßmanagement  
Literatur: Stallings Kapitel 14

# Stand der Vorlesung

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



HEUTE !

# Übersicht

## ◆ Prozeßmigration

- Mechanismen, Verhandlung, Abschiebung

## ◆ Verteilter globaler Zustand

- verteilter Schnappschuß und zugehöriger Algorithmus

## ◆ Verteilter wechselseitiger Ausschluß

- Konzepte, Reihenfolgen von Ereignissen, verteilte Schlangen

## ◆ Verteilter Deadlock

- bei der Ressourcenbelegung
- bei der Nachrichtenübermittlung

# Prozeßmigration

Grundidee: ein Prozeß wird auf einem Rechner unterbrochen, auf anderem fortgesetzt

Mögliche Vorteile:

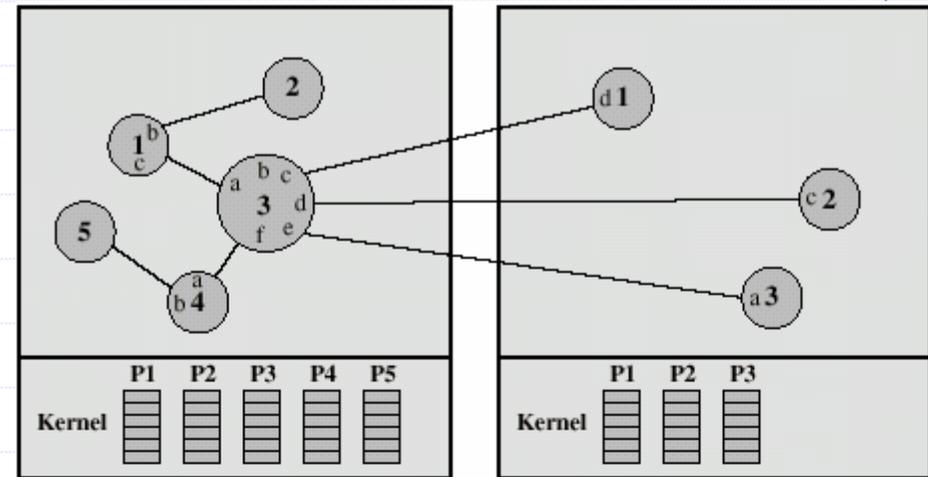
- Lastverteilung, Prozesse wandern von überlasteten zu freien Rechnern
- Kommunikationszeiten, kommunizierende Prozesse werden zusammengelegt
- Verfügbarkeit, bei erwarteten Downzeiten können langlaufende Prozesse verlagert werden.
- Nutzung spezieller Hardware eines Rechners

Aspekte:

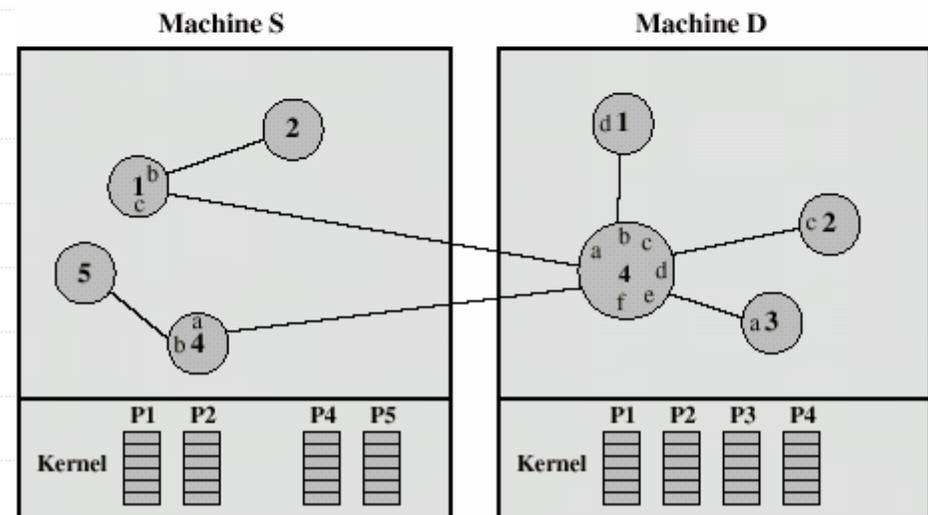
Wer initiiert Migration ?

Welche Prozeßteile migrieren?

Was ist mit Nachrichten/Signalen?



(a) Before migration



(b) After migration

## Prozeßmigration, welche Teile müssen migrieren?

- ◆ Prozeßimage
- ◆ Update der Interprozesskommunikation, Links
- ◆ Update bzgl Ressourcen, z.B. geöffnete Dateien

### Varianten zur Übertragung des Adreßraums

1. Eager (komplett): unmittelbare Übertragung aller Seiten
2. Precopy: noch während der Bearbeitung des Prozesses werden Seiten übertragen, ggfs nach Modifikation auch wiederholt!
3. Eager (dirty pages): nur modifizierte Seiten werden übertragen, andere nur auf Anfrage. Erfordert Seiten/Segmenttabelle an beiden Knoten.
4. Copy-on-reference: nur referenzierte Seiten übertragen (Variante von 3)
5. Flushing: Prozeßimage wird in Datei auf der Festplatte abgelegt, von der Festplatte werden Seiten dann nach Bedarf geladen.

### ähnliche Aspekte bei geöffneten Dateien

- zusätzliches Problem: Cache Konsistenz bzgl Disk Caches wenn mehrere Prozesse auf 1 Datei zugreifen und 1 Prozess migriert

bei Messages / Signalen: zwischenpuffern und weiterleiten

## Migration auf IBM Aix (verteiltes UNIX System)

Prozesse migrieren aus eigener Entscheidung

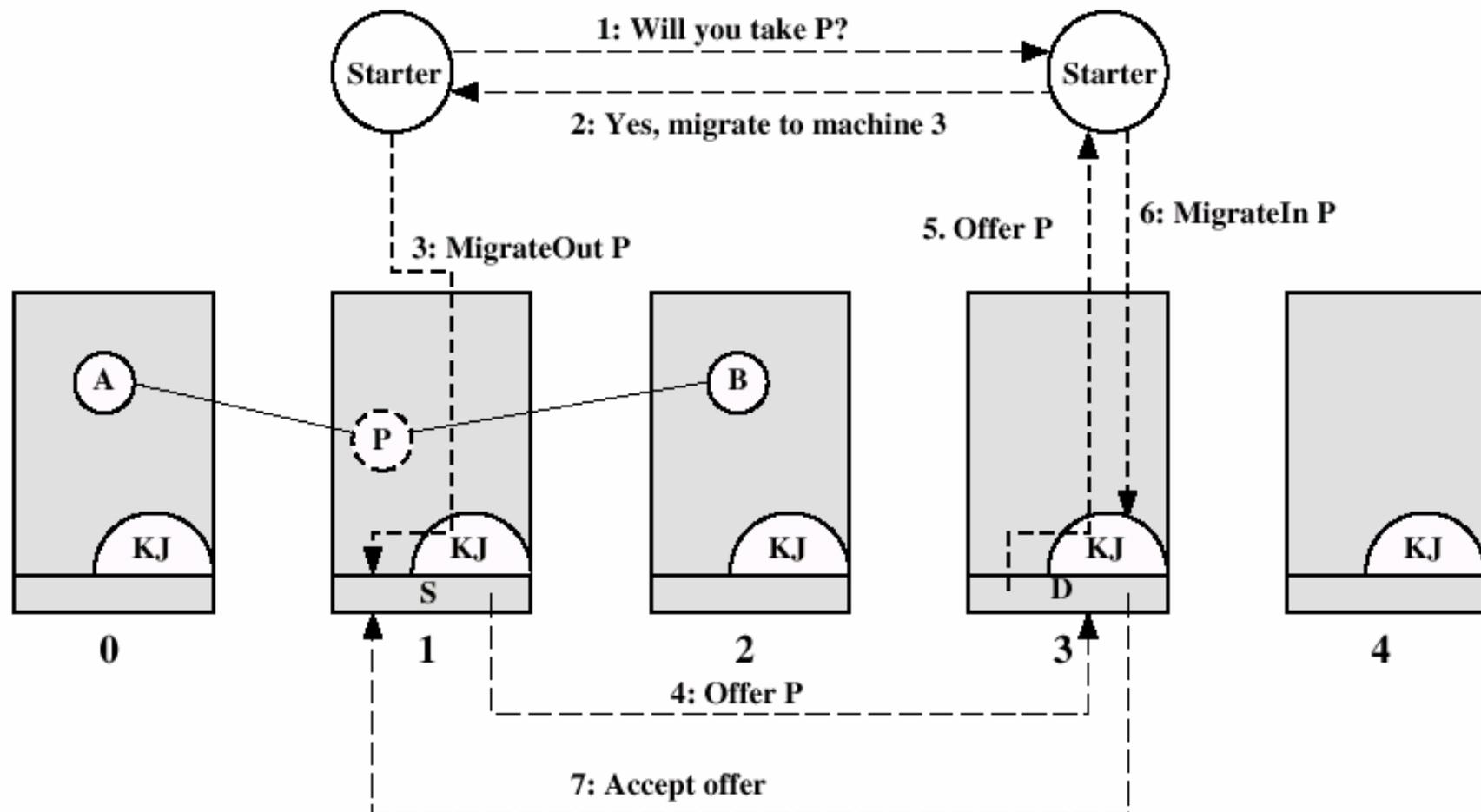
1. p sucht neuen Rechner, sendet „Remote Tasking Message“ mit Teil des Images, Info bzgl offener Dateien
2. Kernel Prozeß des Zielrechners erzeugt Prozeß (fork) und übergibt Image & Dateiinformationen
3. Der neue Prozeß  $p'$  beschafft von p nach Bedarf Daten, Umgebungsinformationen, Stack etc. , modifizierte Seiten werden vom Rechner, unmodifizierte Seiten von einem globalen Dateisystem geladen.
4. p erhält Signal, wenn die Migration abgeschlossen ist, p sendet dann eine „erledigt“ Nachricht an  $p'$ , terminiert.

### Alternative

- Prozesse werden durch Verwaltung, Lastbalancierer verschoben
- erfordert Verhandlungsprotokoll

# Verhandlung bei der Prozeßmigration

2 Verwaltungsprozesse (Starter) beschliessen, verhandeln Migration



Neben Migration auch Ausweisung,  
Rückführung „fremder“ Prozesse auf den Ursprungsrechner

# Übersicht

## ◆ Prozeßmigration

- Mechanismen, Verhandlung, Abschiebung

## ◆ Verteilter globaler Zustand

- verteilter Schnappschuß und zugehöriger Algorithmus

## ◆ Verteilter wechselseitiger Ausschluß

- Konzepte, Reihenfolgen von Ereignissen, verteilte Schlangen

## ◆ Verteilter Deadlock

- bei der Ressourcenbelegung
- bei der Nachrichtenübermittlung

## Globaler Zustand in einem verteilten System

- ◆ Grundproblem: in verteilten nebenläufigen Systemen mit nicht synchronisierten Uhren hat jede Komponente nur eine lokale Sicht auf den Zustand anderer Prozesse, Ressourcenzuordnung etc.
- ◆ Hemmnisse
  - es gibt keinen gemeinsamen Speicher
  - Informationen werden durch Nachrichten übermittelt
  - Übermittlungszeiten sind nicht bekannt
  - die lokale Systemzeit muß nicht mit der globalen Zeit übereinstimmen (Uhren sind nicht synchronisiert)
- ◆ Analogie aus der Astronomie:
  - wg der Übertragungszeiten für Licht „sehen“ wir stets ein veraltetes Abbild des Universums, wobei der Gesamteindruck aus Einzelinformationen entsteht, die zu unterschiedlichen Zeitpunkten entstanden sind.

# Begriffe

## ◆ Kanal

- unidirektionale Verbindung zwischen Rechnern zur Nachrichtenübermittlung
- typische Anforderungen: verlustfrei, reihenfolgetreu

## ◆ Zustand

- Zustand eines Prozesses, Festlegung hängt von der jeweiligen Anwendung ab, z.B:  
hier: Folge aller kommunizierten Nachrichten zwischen Zustandserfassung

## ◆ globaler Zustand

- Kombination aller (lokalen) Prozeßzustände
- konsistent: Zustand ist konsistent, wenn kein lokaler Prozeßzustand den Empfang einer Nachricht ausweist, die im Zustand des zugehörigen Senders noch nicht gesendet wurde.

## ◆ verteilter Schnappschuß

- bestimmt einen möglichen globalen Zustand aus
  - ◆ dem lokalen Zustand jedes Prozesses und
  - ◆ der Nachrichten die bereits gesendet, aber noch nicht empfangen wurden
  - ◆ möglicher Zustand: aus möglichen neuen lokalen Zuständen, entstehen nach Eintreffen der bereits abgesandten Nachrichten auf den Eingangskanälen
  - ◆ Zustand muß konsistent sein!

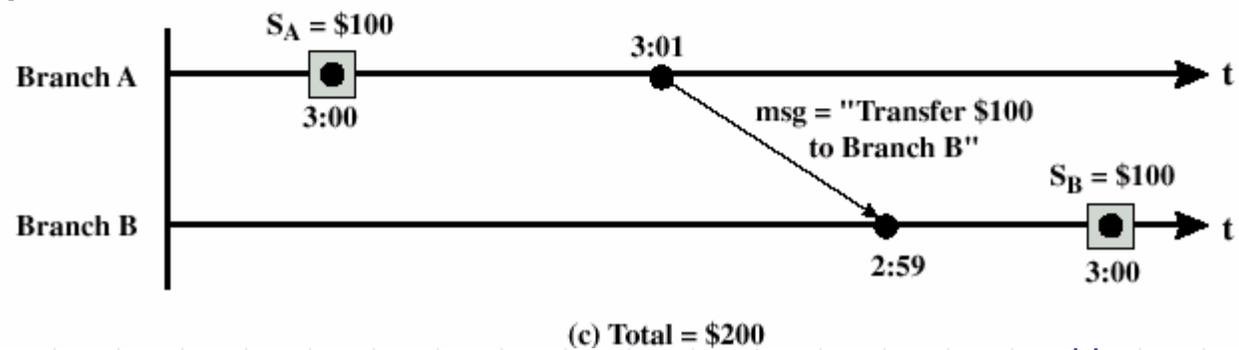
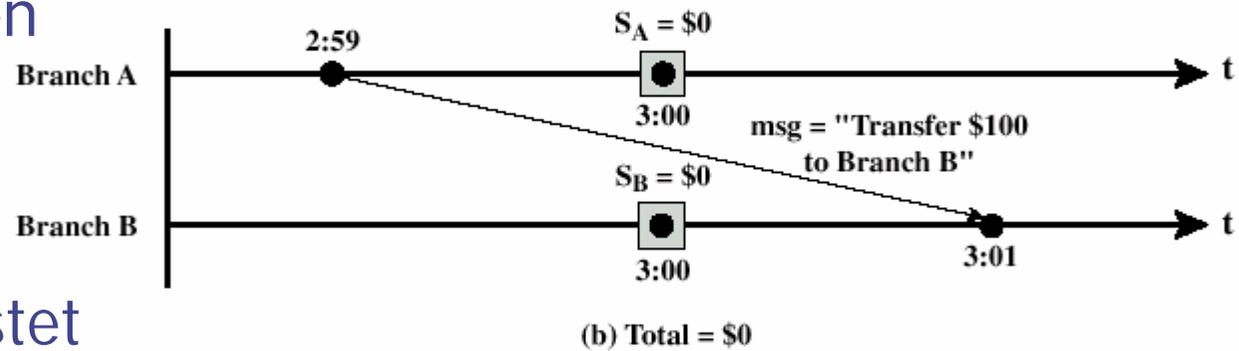
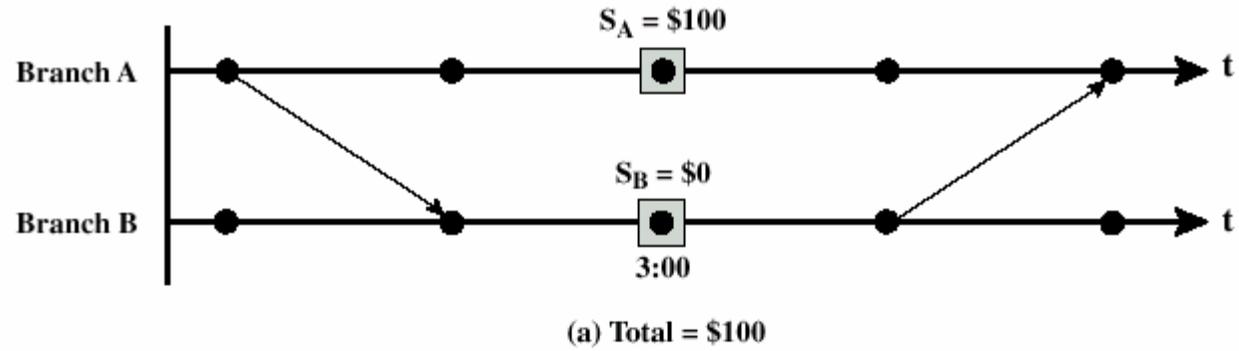
# Verteilter Globaler Zustand: Wo ist das Problem ?

Zustand zum Zeitpunkt 3:00 bei verteilter Datenhaltung.

Datum ist die Summe der lokalen Daten.

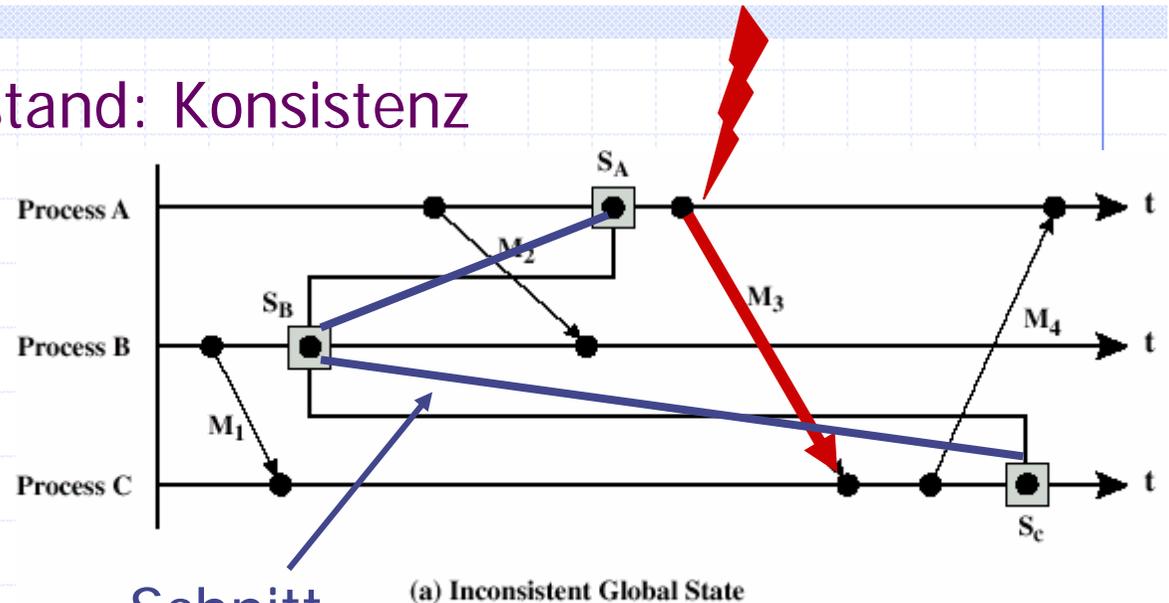
Bsp. Bankkonto Problem

- Übermittlung kostet Zeit, in b) fehlt der Betrag „in transit“
- Uhren nicht synchron, c) zählt doppelt



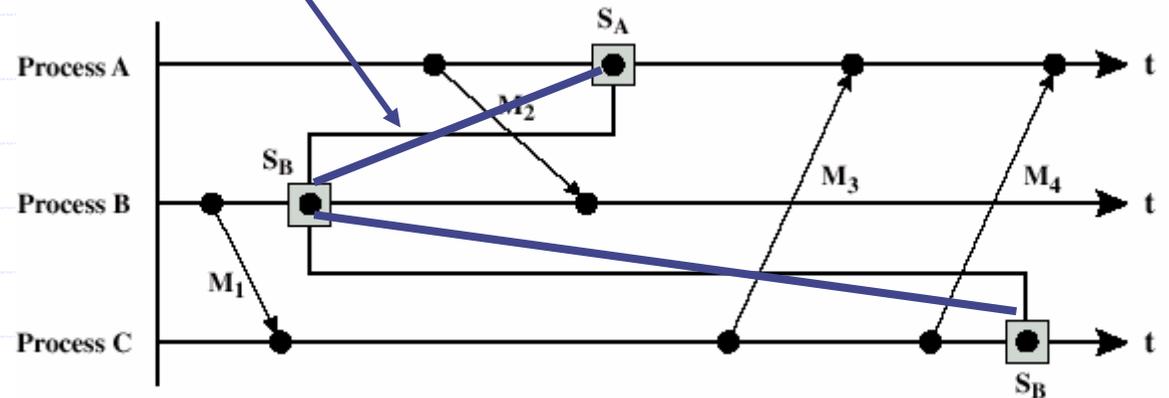
## Verteilter globaler Zustand: Konsistenz

- Idee:  
Kollektion von lokalen Schnappschüssen
- Nachrichten, die abgesendet, aber noch nicht empfangen wurden werden korrekt erfaßt
  - Problem in a) wg nicht synchronisierter Uhren, A schickt Nachricht nach dem Schnappschuß, die vor dem Schnappschuß in C ankommt => Zustand nicht konsistent!



(a) Inconsistent Global State

Schnitt



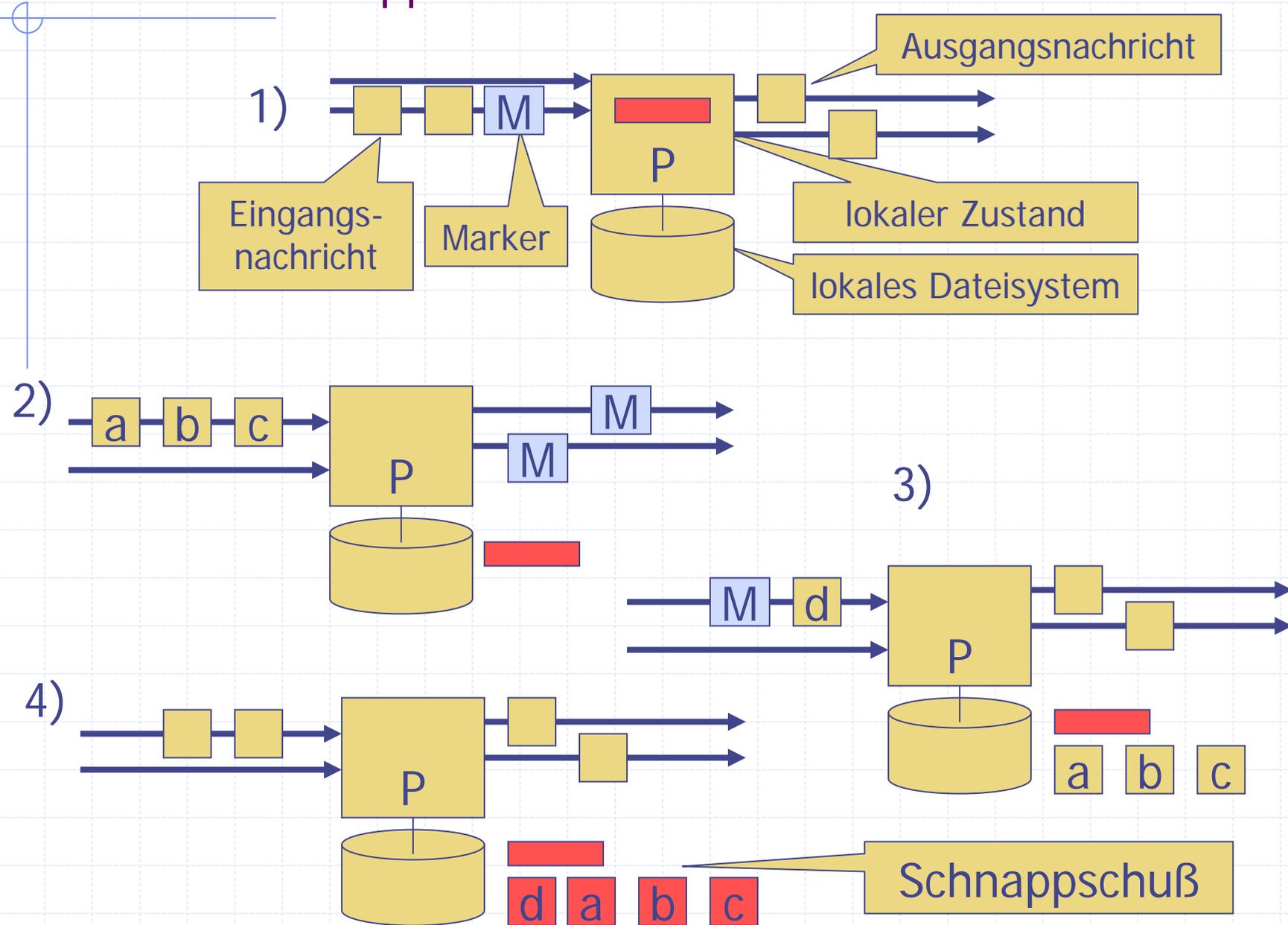
(b) Consistent Global State

Zustand ist konsistent, wenn jede eingetroffene Nachricht auch als gesendet protokolliert wird.

## Algorithmus für einen verteilten Schnappschuß

- ◆ Voraussetzung: Kanäle übermitteln verlustfrei und reihenfolgetreu
- ◆ jeder Prozess  $q$  kann Schnappschuß initiieren,
  - ermittelt eigenen Zustand
  - sendet spezielle „Markierungsnachricht“ über Ausgangskanäle
- ◆ jeder Prozess  $p$ , der eine Markierungsnachricht erhält
  - nur bei der 1. Markierungsnachricht, z.B. von  $p'$ 
    - ◆ ermittle den lokalen Zustand
    - ◆ bestimme den Kanal  $(p', p)$  als leer
    - ◆ reiche die Markierungsnachricht über alle Ausgangskanäle weiterSchritte müssen als Transaktion atomar durchgeführt werden
  - bei weiteren Nachrichten über andere Kanäle, z.B. von  $r$ 
    - ◆ ermittle den Zustand auf Kanal  $(r, p)$  als Folge aller Nachrichten, die seit dem lokalen Schnappschuß an  $p$  aufgetreten sind
- ◆ Ablauf terminiert an jedem Knoten, wenn auf allen Eingangskanälen Nachrichten eingetroffen sind

# Verteilter Schnappschuß



# Verteilter Schnappschuß

## ◆ Eigenschaften

- jeder Prozeß kann Markierungsnachricht senden, (Randproblem: mehrere Berechnungen simultan erfordern Markierungsids)
- Algorithmus terminiert in endlicher Zeit falls Nachrichten in endlicher Zeit übertragen werden
- Verteilter Algorithmus: jeder Prozeß muß nur eigenen Zustand und Eingangsnachrichten ermitteln
- Globaler Zustand kann nach Terminierung durch Abfragen aller lokalen Schnappschußinformationen (Zustand+Nachrichten) ermittelt werden (entweder durch alle oder durch 1 Prozeß)
- Schnappschußalgorithmus behindert andere Algorithmen nicht!

## ◆ Konsistenz:

- nachprüfen, Schnappschuß liefert konsistenten Zustand

## ◆ Anwendung:

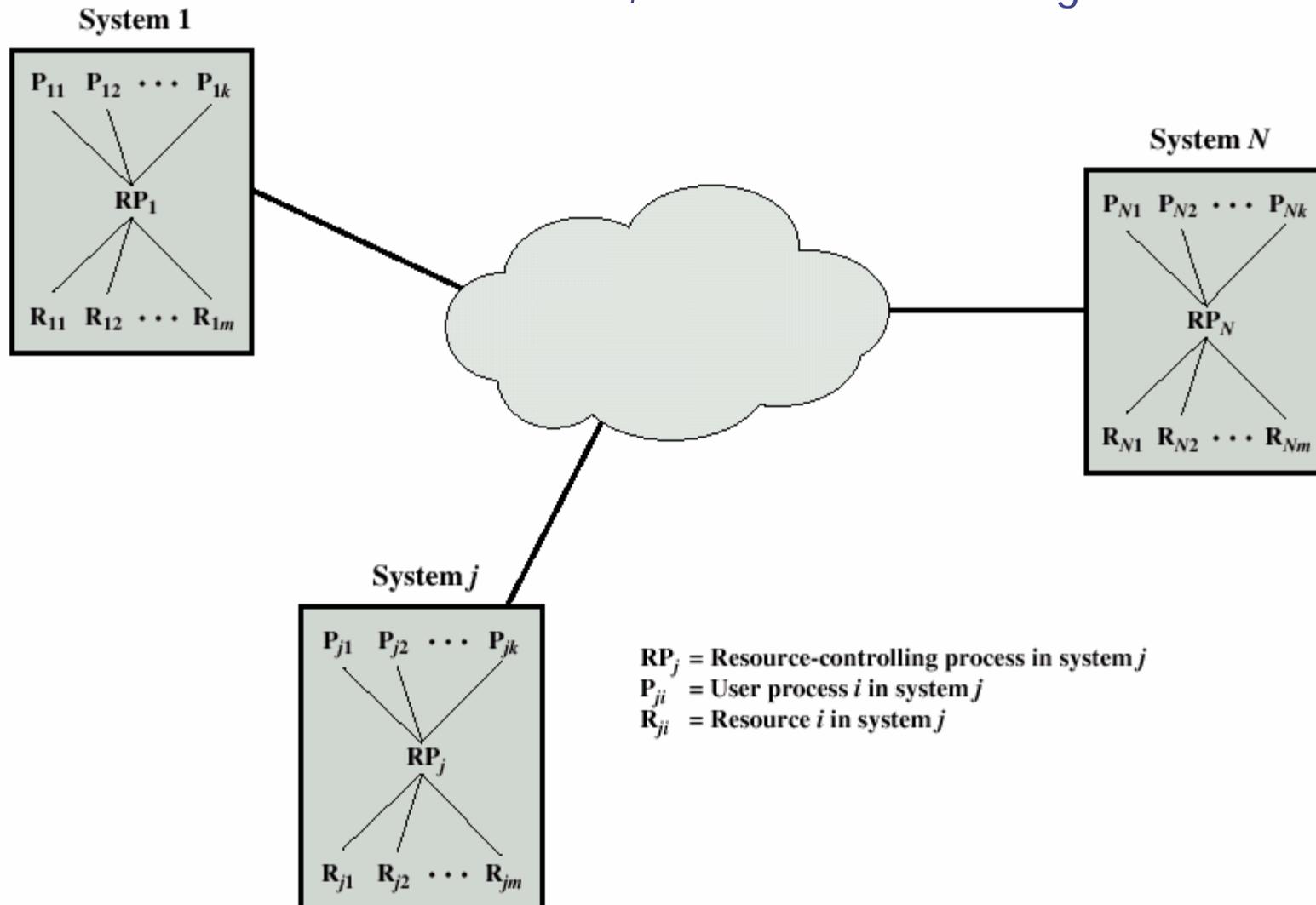
- Verteilter Schnappschuß erlaubt die Anpassung von Algorithmen mit zentraler Steuerung an eine verteilte Umgebung, z.B. Terminierungserkennung, Deadlockerkennung, Checkpointing wg möglichem Rollback

## Verteilter Wechselseitiger Ausschluß

- ◆ wechselseitiger Ausschluß beim Zugriff auf Ressourcen bereits besprochen
  - Maßnahmen: Algorithmen, Semaphore, Monitor
  - Hauptproblem: Testen & Reservieren eines Objektes bei geteiltem Speicher ist keine atomare Operation
- ◆ jetzt: geteilter Speicher -> verteilter Speicher
  - daher: Verfahren mit Nachrichtenübermittlung erforderlich
- ◆ Anforderungen (zur Erinnerung)
  - wechselseitiger Ausschluß für kritische Regionen: nur 1 Prozeß darf zu einem Zeitpunkt in seinem kritischen Abschnitt sein
  - Prozesse dürfen außerhalb des kritischen Abschnitts terminieren ohne andere Prozesse zu beeinträchtigen
  - kein Verhungern, kein Deadlock
  - falls Ressource nicht belegt, kann 1 Prozeß ohne Verzögerung fortschreiten
  - keine Annahmen bzgl der Laufzeiten von Prozessen
  - kein Prozeß bleibt beliebig lange / terminiert im krit. Abschnitt

# Verteilter Wechselseitiger Ausschluß

Szenario: je Rechner, 1 Steuerungsprozeß für Ressourcenanforderungen  
Variante a) zentralisiertes Verfahren, 1 System hat zentralen Steuerungsprozeß mit lokalen Stellvertretern, die in seinem Auftrag handeln



## Wechselseitiger Ausschluß

### ◆ Verfahren mit zentraler Steuerung

- auf 1 System existiert zentraler Steuerungsprozeß ZP
- Anfragen werden von lokalen Steuerungsprozessen an ZP weitergeleitet, nur ZP führt Ressourcenvergabe durch
- Eigenschaften
  - ◆ einfach zu realisieren
  - ◆ offensichtliche Probleme: Performancebottleneck, Ausfallsicherheit

### ◆ verteilte Verfahren

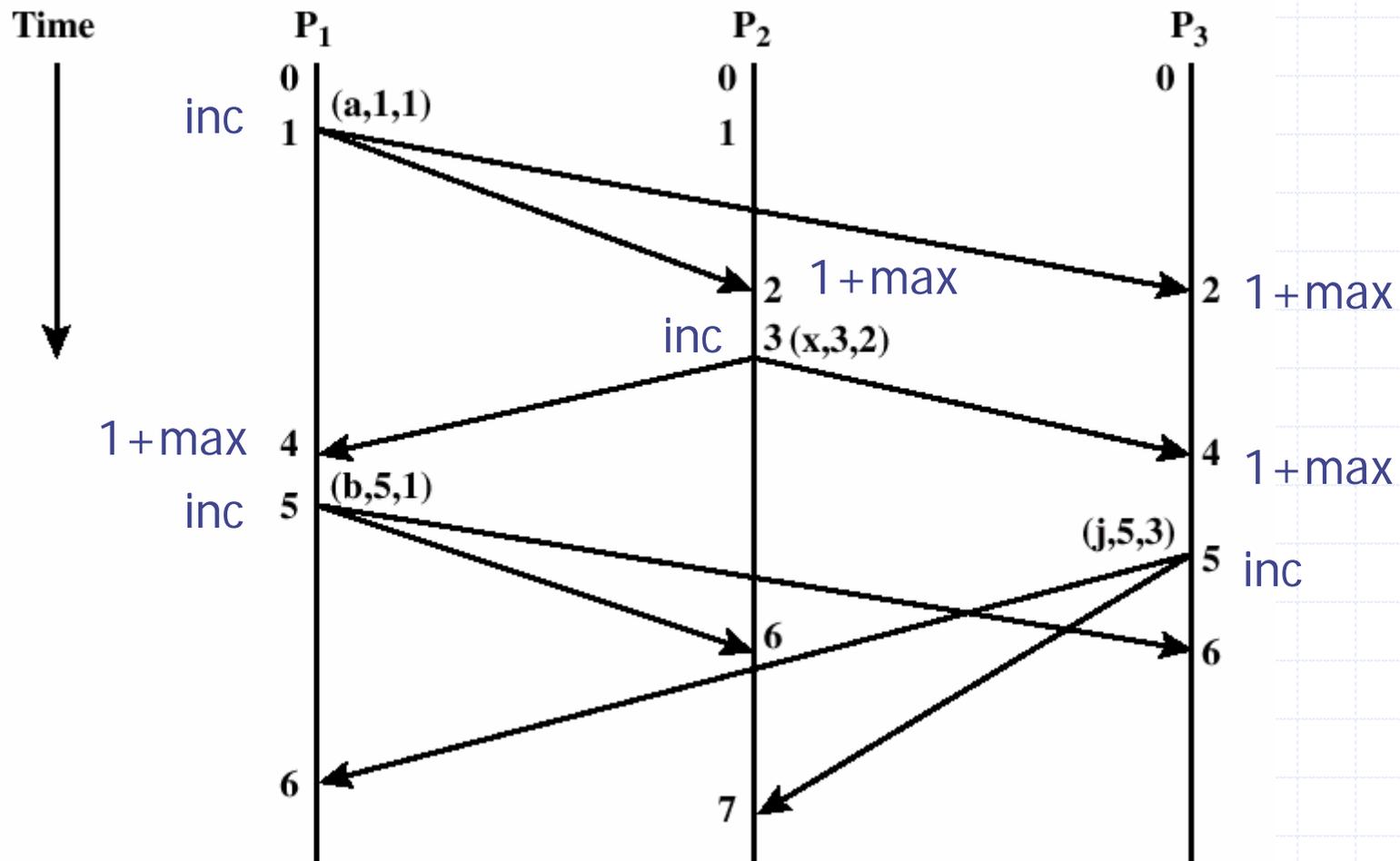
- alle Knoten haben in etwa gleichviel Information
- jeder Knoten hat nur lokale (unvollständige) Information
- alle Knoten gleich verantwortlich für die Gesamtentscheidung
- alle Knoten leisten in etwa gleichviel Arbeit
- Ausfall eines Knotens gefährdet den Ablauf nicht
- eine systemweite einheitliche Uhr existiert nicht

### ◆ Bzgl der nicht-synchronisierten lokalen Uhren existiert allg. Verfahren mit Zeitstempeln

# Zeitstempel bewirken totale Ordnung von Ereignissen

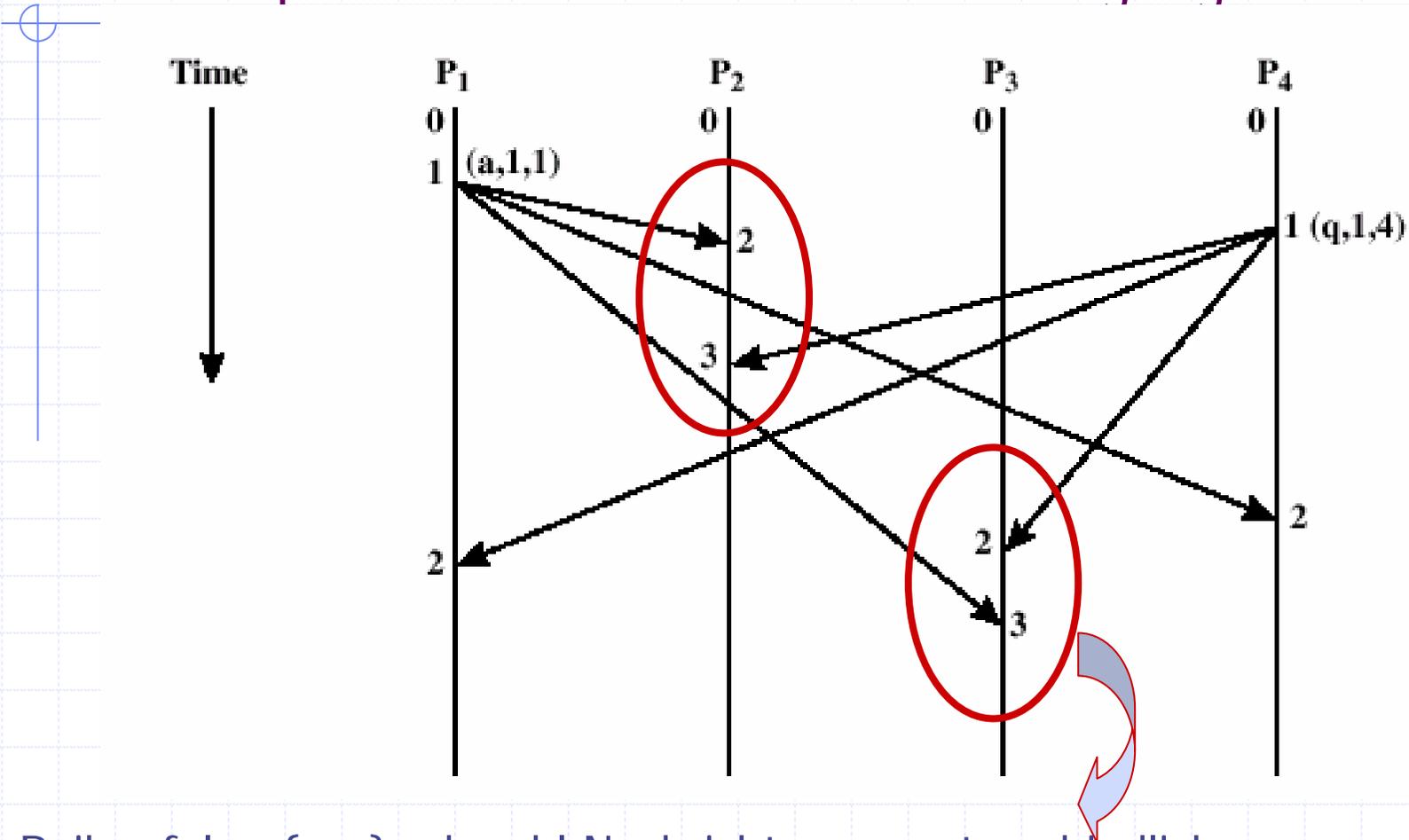
- ◆ Ziel: konsistente Ordnung aller kommunizierten Nachrichten an allen Knoten
  - alle Ereignisse auf verschiedenen Knoten lassen sich miteinander vergleichen und eine vorher/nachher Reihenfolge feststellen
  - lokale Ordnungen an Knoten im Netz sind miteinander verträglich
  - Ereignis eines Prozesses: Senden einer Nachricht (Empfangen zählt nicht)
- ◆ Zeitstempel als Wert eines lokalen Zählers
  - jeder Knoten hat lokalen Zähler  $C_i$  (als „Uhr“)
  - Senden einer Nachricht bewirkt vorher Inkrement des Zählers,  $C_i := C_i + 1$   
Nachricht  $(m, T_i, i)$  mit Inhalt  $m$ , Zeitstempel  $T_i := C_i$ , von Knoten  $i$
  - Empfangen einer Nachricht  $(m, T_i, i)$  an Knoten  $j$  bewirkt  
 $C_j = 1 + \max(C_j, T_i)$
- ◆ Reihenfolge aller Ereignisse im Netz anhand der Regel
  - $(m, T_i, i)$  vor  $(m, T_j, j)$   
falls  $T_i < T_j$  oder  
falls  $T_i = T_j$  und  $i < j$
- ◆ Vorteil:
  - unabhängig von lokalen Uhren und Übertragungszeiten

## Beispiel für den Zeitstempel Algorithmus



Reihenfolge der Ereignisse:  $\{a,x,b,j\}$  an allen Prozessen gleich und anhand jeweils lokaler Information entscheidbar!

## Zeitstempel bei unterschiedlichen Übertragungszeiten



Reihenfolge {a,q}, obwohl Nachrichten zu unterschiedlichen Zeitpunkten / in unterschiedlichen Reihenfolgen an P<sub>2</sub>, P<sub>3</sub> ankommen

Wichtig ist nicht die reale Ankunftsreihenfolge, sondern dass alle Prozesse dieselbe Reihenfolge feststellen.

## Verfahren für eine Verteilte Warteschlange

- ◆ liefert dann wechselseitigen Ausschluß, weil der 1. Prozeß in der Schlange, Zugriff auf die Ressource erhält
- ◆ Voraussetzung für Verfahren
  - N Knoten mit jeweils 1 Prozeß, der Zugriff auf kritischen Abschnitt anfordert, ggfs als lokaler Koordinator für mehrere lokale Prozesse
  - Nachrichtenübertragung ist reihenfolgetreu, verlustfrei
  - Netzwerk ist vollständig verbunden, Übertragung ist direkt
- ◆ Zur Vereinfachung: nur 1 Ressource je Knoten
- ◆ Zeitstempel liefern erforderliche Reihenfolge in den Anforderungen
- ◆ jeder Knoten hat eigene Queue, alle sollten bzgl Ordnung gleich sein
- ◆ Problem bei unterschiedlichen Übertragungszeiten
  - auf vorheriger Folie: P2 und P3 sind zeitweise unterschiedlicher Meinung bzgl des 1. Elementes in der Reihenfolge, weil noch Nachrichten unterwegs sind
- ◆ Abhilfe
  - Entscheidungen bzgl wechselseitigem Ausschluß nur dann, wenn Nachrichten anderer Knoten mit höherem Zeitstempel als 1. Element vorliegen.

# Algorithmus

## ◆ Datenstruktur an jedem Knoten

- Array der Länge N mit  $q[j]$  mit letzter Nachricht von j,
- initial  $q[j] = (\text{Freigabe}, 0, j)$  für alle  $j = 1, 2, \dots, N$

## ◆ 3 Arten von Nachrichten

- (Anfrage,  $T_i, i$ ):  $P_i$  möchte auf die Ressource zugreifen
- (Zusage,  $T_j, j$ ):  $P_j$  erlaubt Zugriff
- (Freigabe,  $T_k, k$ ):  $P_k$  gibt belegte Ressource frei

## ◆ Ablauf

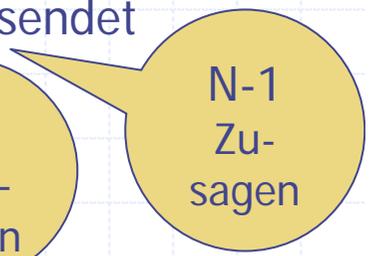
- $P_i$  fragt Ressource mit (Anfrage,  $T_i, i$ ) bei allen Prozessen an, aktualisiert  $q[i] := (\text{Anfrage}, T_i, i)$
- $P_j$  erhält (Anfrage,  $T_i, i$ ), aktualisiert  $q[i] := (\text{Anfrage}, T_i, i)$ , sendet (Zusage,  $T_j, j$ ) falls  $q[j]$  keine Anfrage enthält.
- $P_i$  kann fortschreiten, falls
  - ◆  $q[i]$  ist die früheste Anfragenachricht im Array von i
  - ◆ alle anderen Nachrichten in q sind älter
- $P_i$  gibt mit (Freigabe,  $T_i', i$ ) an alle Prozesse eine Ressource frei, aktualisiert  $q[i] = (\text{Freigabe}, T_i', i)$
- Wenn  $P_j$  (Freigabe,  $T_i', i$ ) oder (Zusage,  $T_j, j$ ) erhält wird damit das Feld i bzw j aktualisiert



N-1  
Nach-  
richten



N-1  
Nach-  
richten



N-1  
Zu-  
sagen

# Algorithmus

## ◆ sichert wechselseitigen Ausschluß

- Anfragen werden nach Reihenfolge gemäß Zeitstempel geordnet
- wenn  $P_i$  in den kritischen Abschnitt eintreten kann, kann keine andere Anfragenachricht von einem früheren Zeitpunkt sein, da alle Zusage- und Freigabenachrichten älter sein müssen
- Voraussetzung: Übertragung ist reihenfolgetreu

## ◆ ist fair

- gleiche Chancen für alle Prozesse wg Ordnung nach Zeitstempeln

## ◆ keine Deadlocks

- ebenfalls wg der Ordnung nach Zeitstempeln

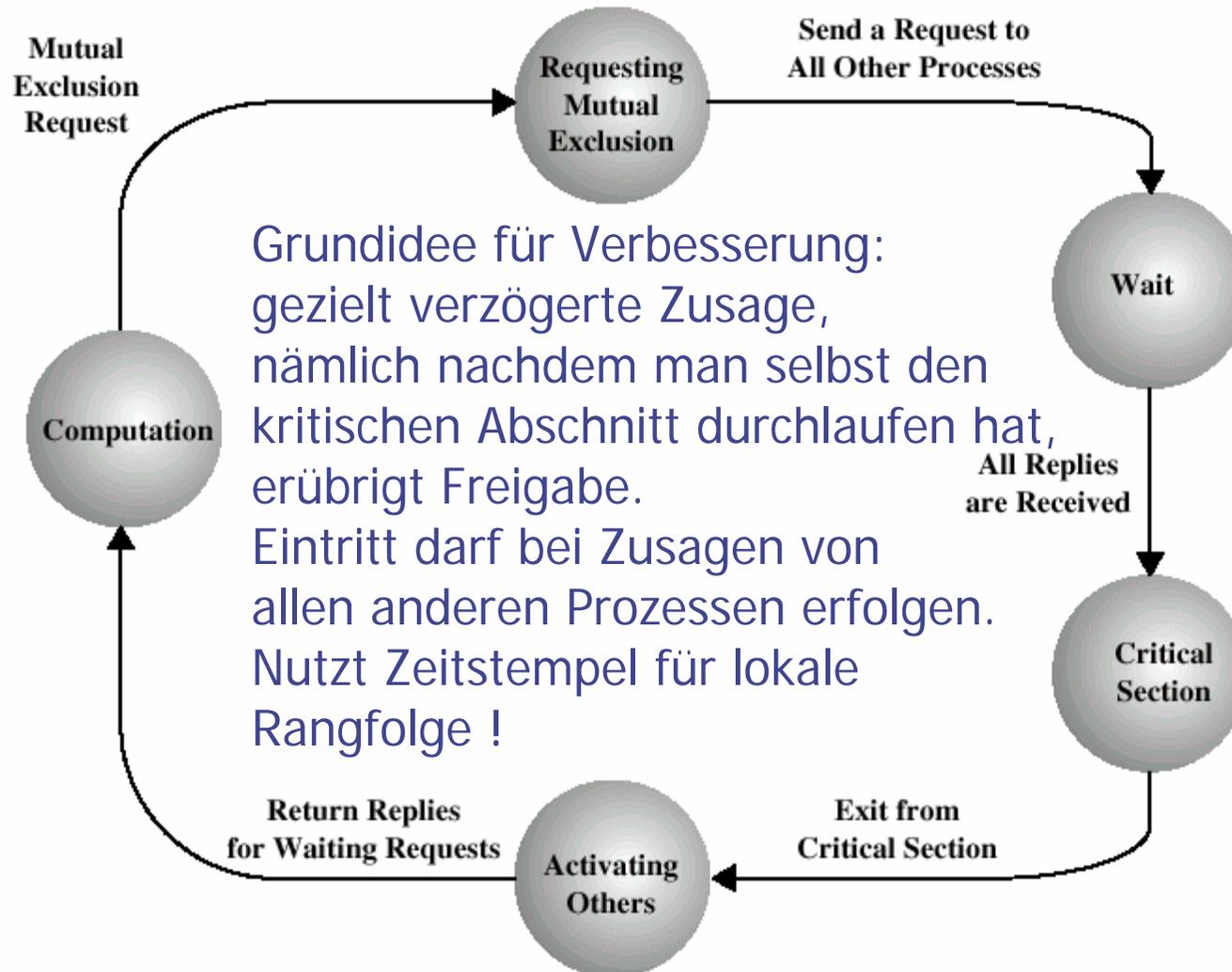
## ◆ kein Verhungern

- bei Verlassen des kritischen Abschnitts durch  $P_i$  wird mit der Freigabenachricht in jedem Array die Anfrage von  $i$  überschrieben, daher kann anderer Prozeß dann eintreten.
- faire Behandlung des Reihenfolgeproblems

## ◆ Aufwand: 3 (N-1) Nachrichten

- bessere Algorithmen bekannt, verzichten z.B. auf Freigabenachr.

# Verbesserung des verteilten wechselseitigen Ausschlusses



Grundidee für Verbesserung:  
gezielt verzögerte Zusage,  
nämlich nachdem man selbst den  
kritischen Abschnitt durchlaufen hat,  
erübrigt Freigabe.  
Eintritt darf bei Zusagen von  
allen anderen Prozessen erfolgen.  
Nutzt Zeitstempel für lokale  
Rangfolge !

Aufwand: nur noch  $2(N-1)$  Nachrichten

## Alternative Variante: Wechselseitiger Ausschluß

### ◆ Weiterreichen einer Berechtigungsmarke

- Token-Passing
- z.B. auch in Netzwerken: Tokenring als Alternative zu Ethernet

### ◆ Datenstrukturen

- **Token**: Array mit N Einträgen, letzter **Besitz des Tokens**
- je Prozeß: Array mit N Einträgen, letzte **Anforderung** jedes Prozesses

### ◆ Grundidee:

- da nur ein Token existiert, kann der jeweilige Besitzer exklusiv den kritischen Abschnitt betreten
- bei Verlassen des Abschnittes wird das Token an den nächsten weitergereicht
- ein Prozeß sendet ggfs eine Token-Anforderung an alle anderen
- Auswahl des Nächsten reihum im Tokenarray, beginnend beim aktuellen Prozess, Auswahl: **Anforderung[k] > Token[k]**
- Aufwand: N Nachrichten, (N-1 für Anfrage, 1 für Tokenvergabe)

## Zwischenstand

### ◆ Prozeßmigration

- Mechanismen, Verhandlung, Abschiebung

### ◆ Verteilter globaler Zustand

- verteilter Schnappschuß und zugehöriger Algorithmus

### ◆ Verteilter wechselseitiger Ausschluß

- Zeitstempel und Reihenfolge von Ereignissen
- verteilte Warteschlange

### ◆ Verteilter Deadlock

- bei der Ressourcenbelegung
- bei der Nachrichtenübermittlung

## Wiederholung: Deadlock = Verklemmung

Dauerhafte Blockierung einer (Teil-) Menge von Prozessen durch Warten auf Ressourcen oder Nachrichten

Deadlocks sind unerwünscht

-> Anforderung an OS Deadlocks bei Prozessen / Threads Deadlocks

zu verhindern:

- Bedingungen an die Ressourcenvergabe sind so gestaltet, dass es grundsätzlich nicht zu Deadlocks kommen kann.

oder zu vermeiden:

- Ressourcen werden durch das Betriebssystem situativ nur so vergeben, dass keine Deadlocks auftreten.

oder zu entdecken und beseitigen:

- falls Deadlocks auftreten, diese zu entdecken und zu beseitigen.

Achtung: gängige Praxis in Betriebssystemen ist jedoch **Ignorieren** d.h. keine automatische Lösung umgesetzt

## Wiederholung: Bedingungen für Deadlocks

### 1. Mutual exclusion

- maximal 1 Prozeß darf eine Ressource zu einem Zeitpunkt nutzen

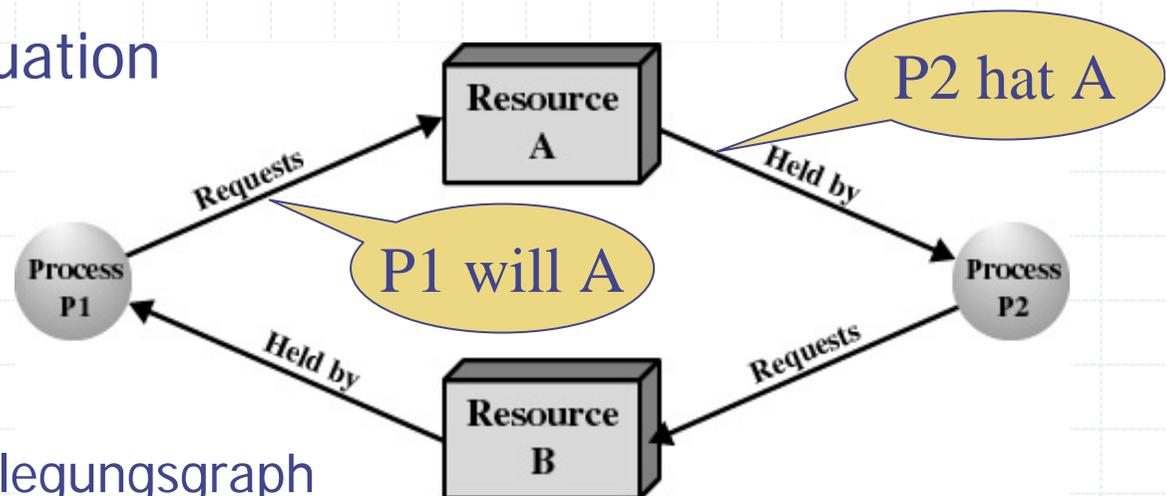
### 2. Halten-und-Warten

- ein Prozeß allokiert nacheinander Ressourcen, ggfs wartet er auf Ressourcen, jedoch ohne bereits allokierte Ressourcen freizugeben

### 3. Keine Unterbrechung und zwangsweise Ressourcenfreigabe bei einem Prozeß

### 4. Zyklische Wartesituation

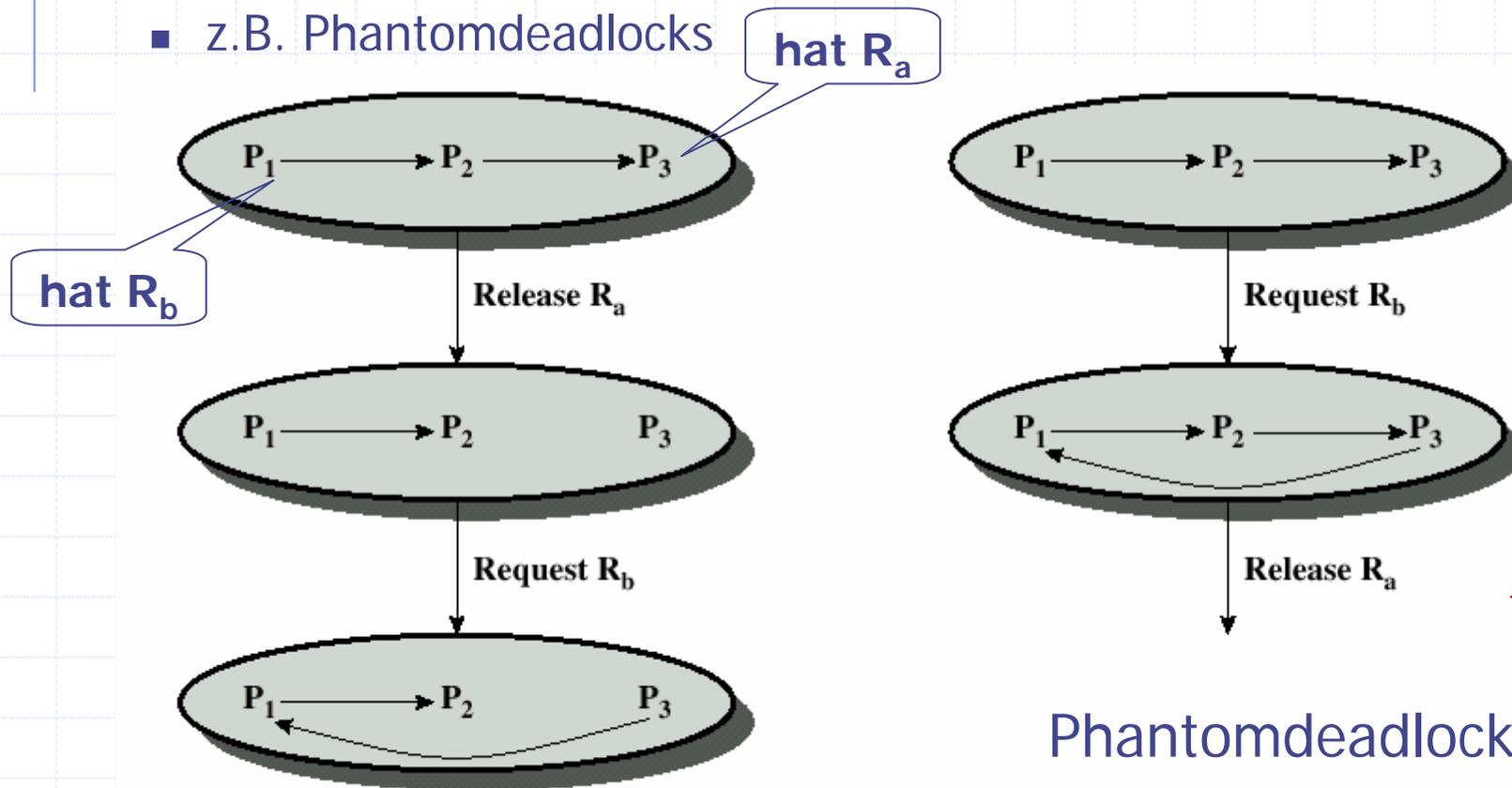
Bedingungen 1-4 sind notwendig & hinreichend!



Ressourcen-Belegungsgraph

## Wozu bei verteilten Systemen erneut betrachten?

- ◆ Ressourcen sind verteilt
- ◆ Steuerung ist verteilt und hat jeweils nur lokale Informationen, kennt aktuellen globalen Zustand nicht
- ◆ Schwierigkeiten
  - z.B. Phantomdeadlocks



(a) Release arrives before request

(b) Request arrives before release

# Varianten

## ◆ Deadlockverhinderung

- mittels linearer Ordnung
  - mittels erzwungener Freigabe bei inkrementellen Anforderungen
- beide Varianten sind übermäßig restriktiv und erfordern Bestimmung des Ressourcenbedarfs im voraus
- Alternativen aus Datenbankbereich: wait-die und wound-wait

## ◆ Deadlockvermeidung

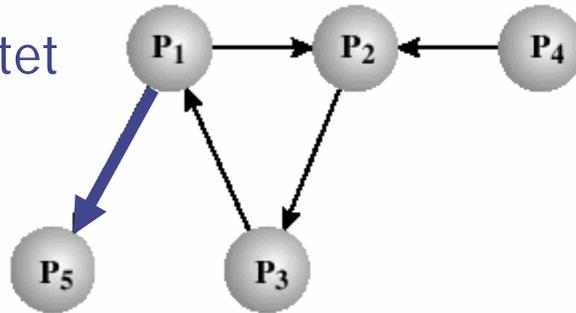
- nicht möglich, Erfassung des globalen Zustands durch jeden Knoten und Bestimmung sicherer globaler Zustände im wechselseitigen Ausschluß zu aufwendig

## ◆ Deadlockerkennung und -behandlung

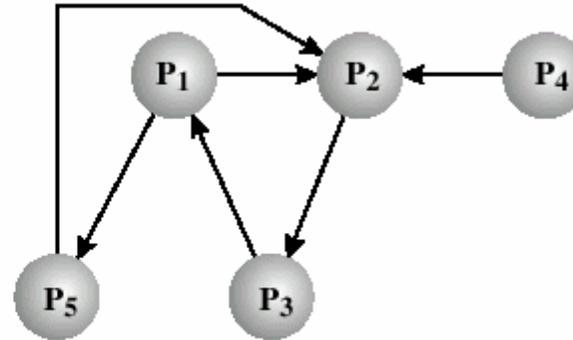
- mit zentraler Kontrolle, 1 Knoten hat globale Sicht, erkennt Deadlocks (evtl. auch Phantomdeadlocks), evtl Bottleneck
- mit hierarchischer Kontrolle, baumartige Verfeinerung der zentralen Kontrolle
- mit verteilter Kontrolle, schwierig und aufwendig, Deadlocks können mehrfach erkannt werden

# Deadlocks bei Nachrichtenübermittlung

P1 wartet auf P5



(a) No deadlock



(b) Deadlock

Alle Nachfolger von S sind in S enthalten.

Deadlock in Prozessmenge S wenn

- 1) alle Prozesse in S auf Nachrichten warten
- 2) S beinhaltet die Abhängigkeitsmenge aller Prozesse aus S
- 3) es sind keine Nachrichten mehr unterwegs

Abhängigkeitsmenge von S: Menge aller Prozesse, auf deren Nachrichten mindestens ein Prozeß aus S wartet

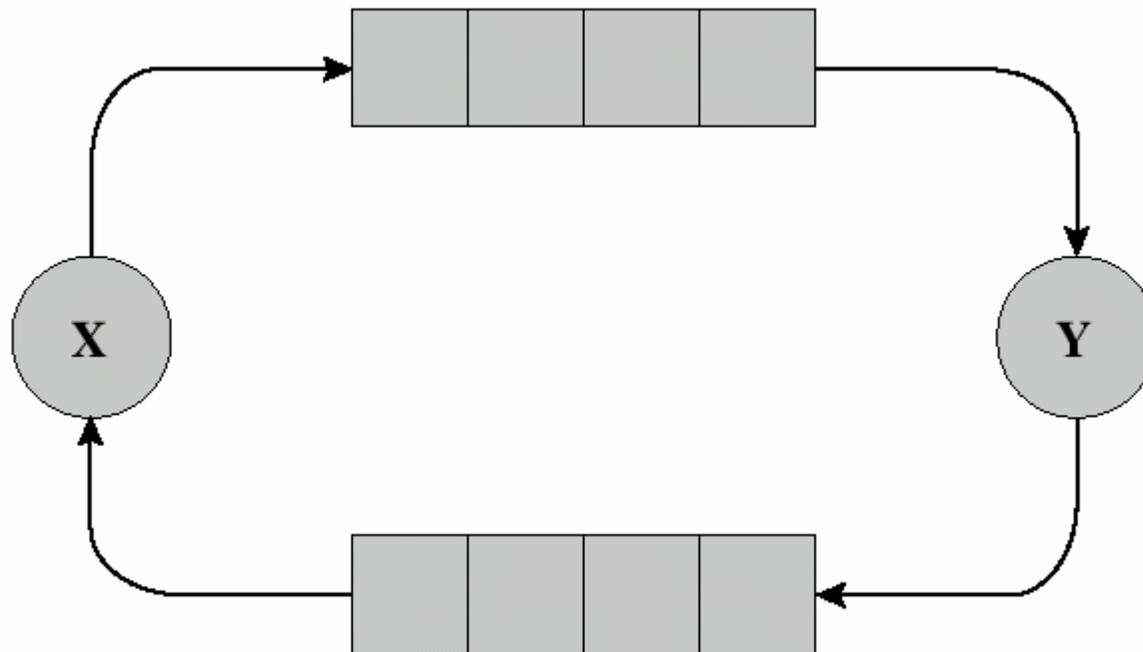
Diese Art von Deadlock kann mit Verhinderung oder Erkennung behandelt werden.

## leider noch weitere

Nachrichtenübermittlung

mit endlichen Puffergrößen bei der Interprozeßkommunikation,

- wenn Pufferplätze aus gemeinsamem Pool stammen oder
- wenn gefüllter Puffer Blockierung für den Sender auslöst.



# Zusammenfassung

## ◆ Prozeßmigration

- Mechanismen, Verhandlung, Abschiebung

## ◆ Verteilter globaler Zustand

- verteilter Schnappschuß und zugehöriger Algorithmus

## ◆ Verteilter wechselseitiger Ausschluß

- Zeitstempel und Reihenfolge von Ereignissen
- verteilte Warteschlange
- Token Passing

## ◆ Verteilter Deadlock

- bei der Ressourcenbelegung
- bei der Nachrichtenübermittlung