Wechselseitiger Ausschluss
(mutual exclusion)

Wechselseitiger Ausschluss *(mutual exclusion)*

- Ziel:
  - mehrere Prozessoren nutzen
  - mehrere gemeinsame Betriebsmittel
- Beispiel:
  - Philosophenproblem
- entscheidende Situation: **critical section** CS

Lösung im gemeinsamen Speicher

Semaphoren schützen kritischen Abschnitt

= spezielle gemeinsame Variablen

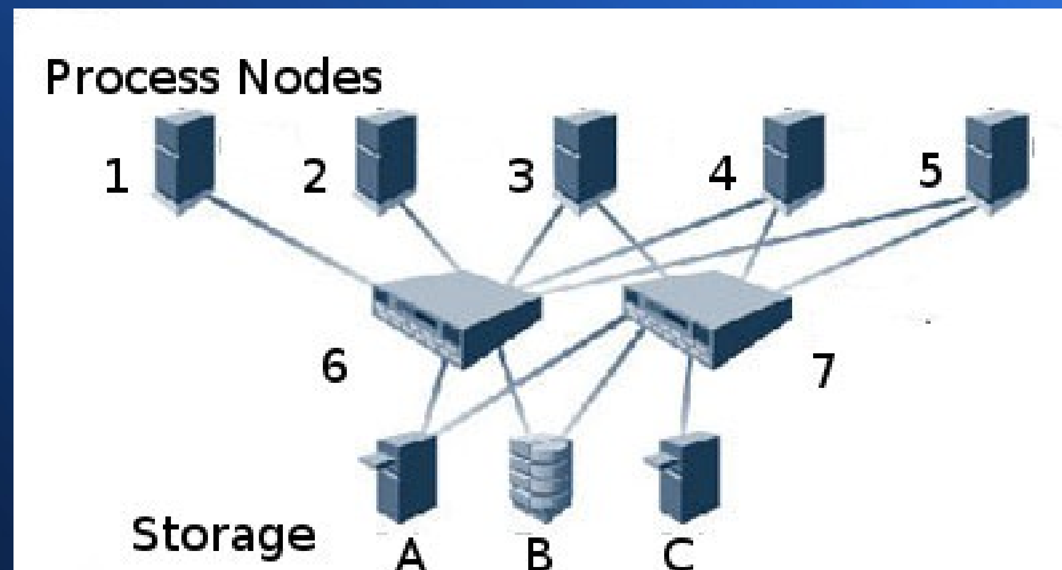
mit atomaren Operationen (P,V)

bzw **wait, acquire, down** für **P** (holl. prolaag)

signal, release, post, up für **V** (holl. vrijgave)

Ziel

- Verteilter wechselseitiger Ausschluss



- ohne gemeinsamen Speicher = keine Semaphoren
- d.h. nur mittels Nachrichten

Erweiterung des Atommodells Eigenschaften eines Prozesses

- Aktiv / Inaktiv

- Wartend
  - blockiert
  - kein Versenden

Atommodell bezieht sich auf Anwendungsnachrichten -> Kontrollnachrichten erlaubt

Mutual-Exclusion-Modell

- Prozess ist
  - CS anfragend (=requesting), wartend und blockiert
  - CS ausführend (=executing), aktiv
  - Passiv (=idle), inaktiv

- Requests → Queue

CS = critical section

Gefahr durch die ...

vier apokalyptischen Reiter verteilter Systeme

- Unfairness
- Starvation
- Deadlock
- No Progress

Safety, Liveness und Fairness bezogen auf critical section

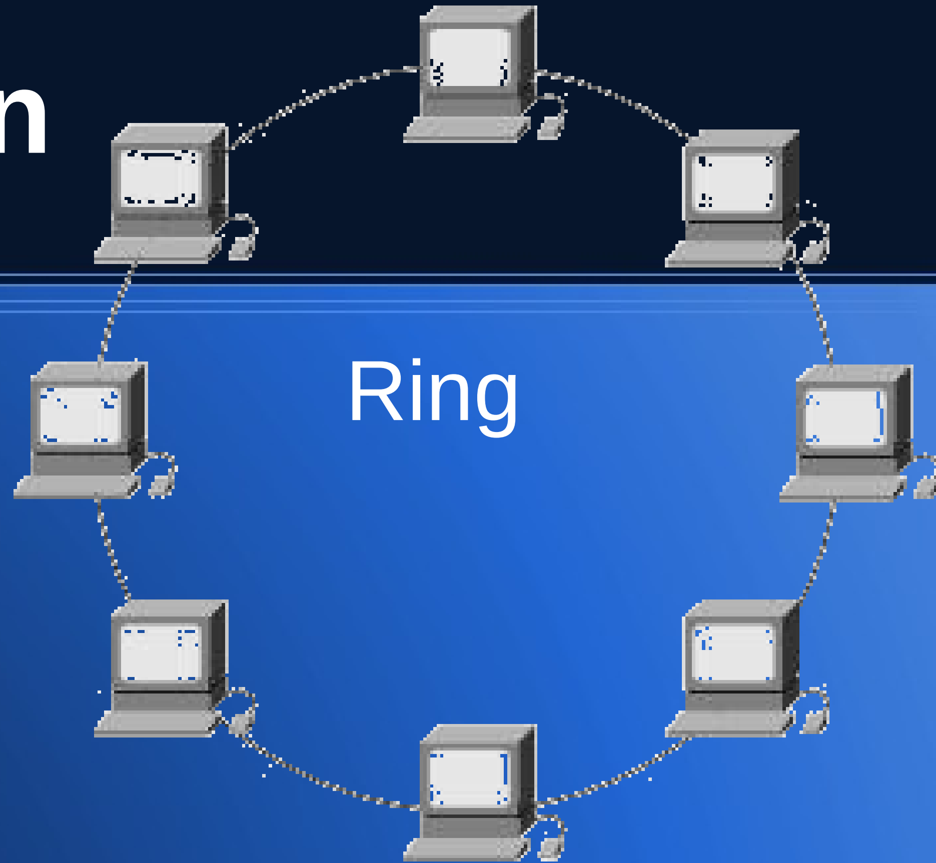
- Safety: nur ein Prozess in CS
- Liveness: jeder Prozess irgendwann in CS

  (vermeidet Deadlock oder Starvation)
- Fairness: Requests lt. Reihenfolge bedient

Ansätze

- Token
- Non-Token
  - (zentraler Server)
  - Requests/Responses
  - Quorum (Mehrheitsmeinung)

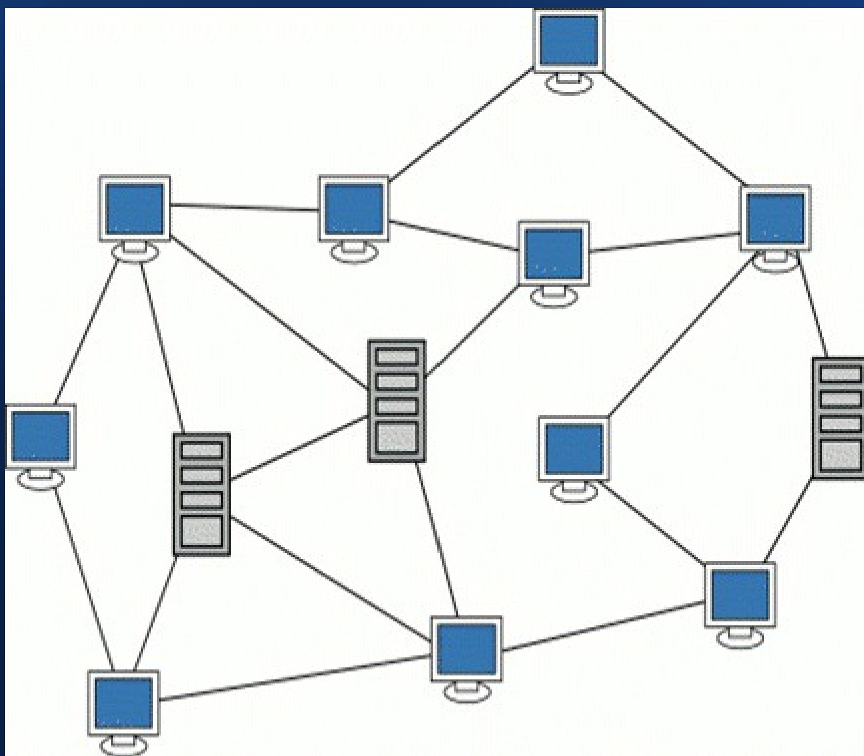
Token

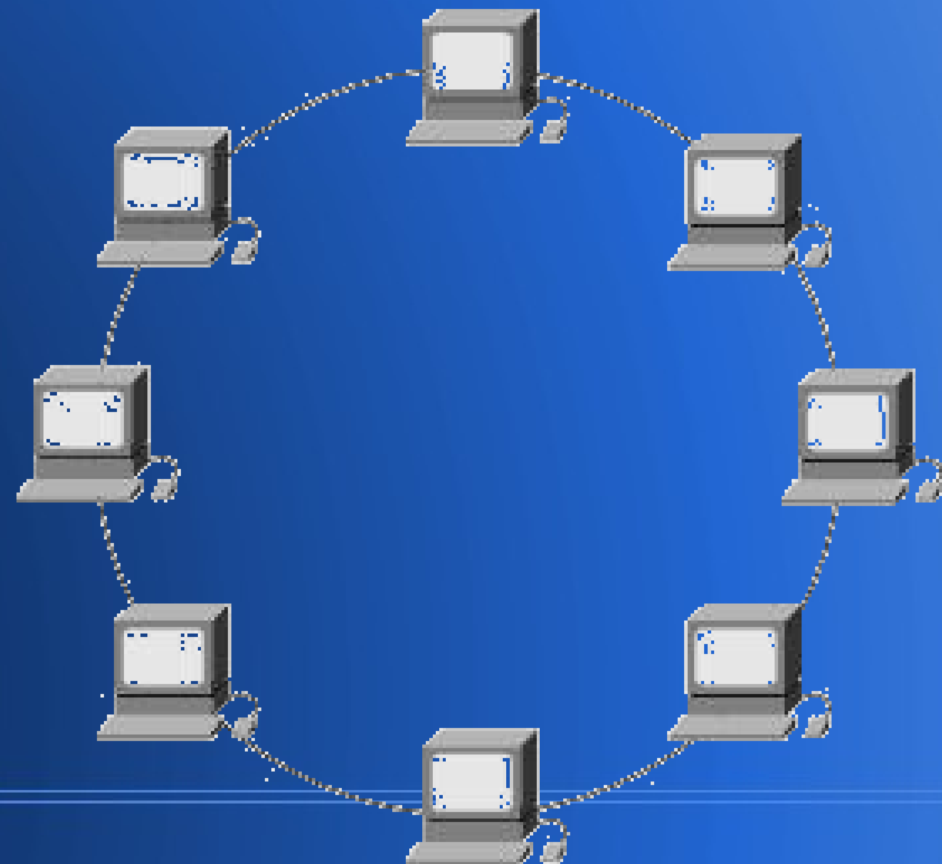


- sende Token in logischem Ring
- wer das Token hat, hat den Zugriff
- jede Ressource ein eigenes Token

Token

Aufbau des logischen Rings?





Beschleunigung (Suzuki/Kasami)

Idee für Anforderung CS:

Broadcast-Request für Token

Besitzer des Tokens:

- halte Token, solange in CS
- sende Token an den, der Request sendete

Non-Token: Zentraler Server

- Benutze Leader-Election Algorithmus, bestimme zentralen Server
- Wer den kritischen Abschnitt betritt, fragt vorher den zentralen Server
- einfache Implementierung, single-point of failure

Lamport-Algorithmus, erste verteilte Lösung (1978)

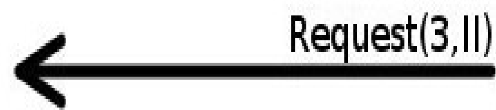
Basiert auf Lamport-Zeit (timestamp TS):

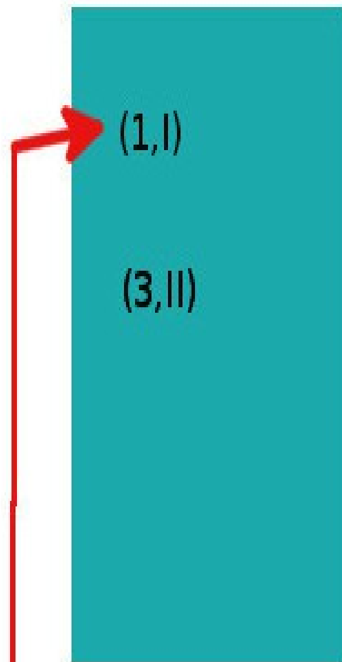
- RequestQueue (Sortierkriterium [TS,ID])
- Broadcast eigene Requests (mit TS)
- ACK von allen (größerer TS)
- Eigener Request QueueAnfang (kleinster TS) und von allen ACK erhalten Kritischer Abschnitt

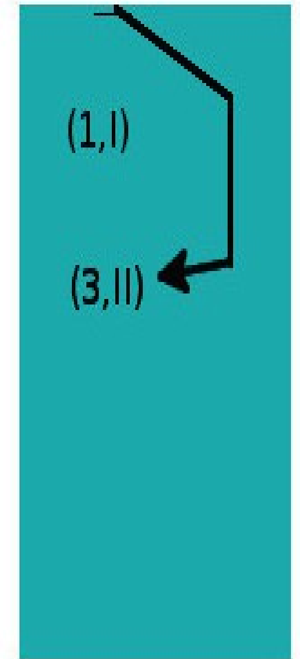
  Nach Beendigung:

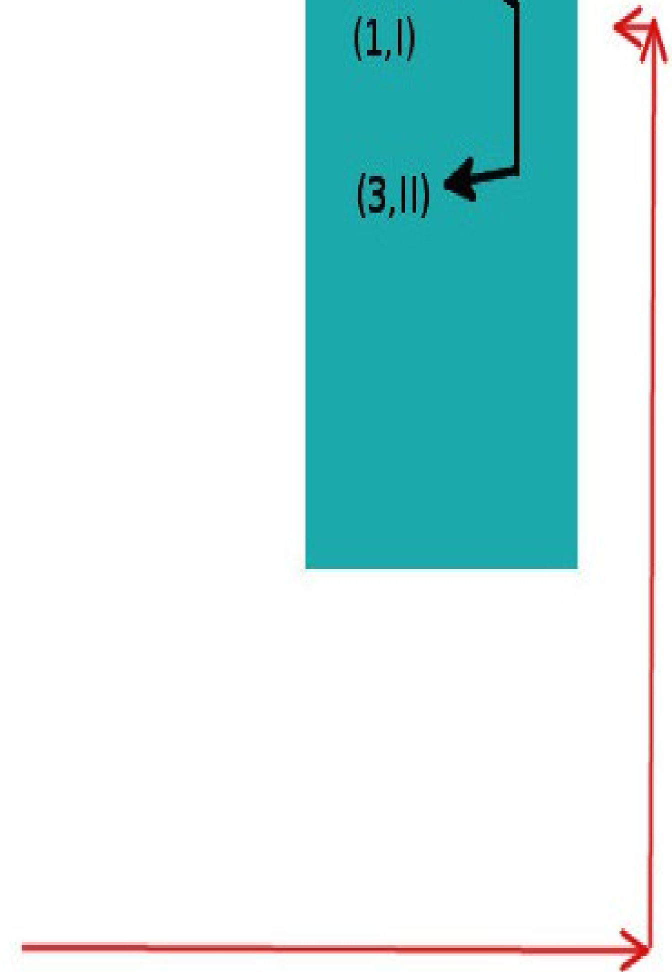
  - Broadcast Release-Message, alle entfernen [TS,ID] aus ihrer Queue

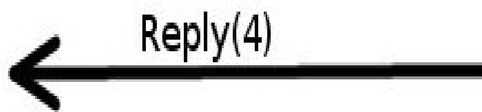
Queue I Process I Process II Queue II











first on queue and all replies received

Protocol for P_i:

To request critical section:

- Send a timestamped *request* message to every other process.
- Put that message on a local request queue.

When *request* message is received:

- Put it on local request queue.
- Send back a timestamped *acknowledgement*.

To exit critical section:

- Remove P_i's *request* message from local request queue.
- Send a *release* message to all other processes.

When *release* message is received:

- Remove corresponding *request* message from local request queue.

P_i enters its critical section if both of the following hold:

- Its own *request* has the lowest timestamp (according to $\Rightarrow$) among all *requests* in its queue.
- P_i has received a message from every other process timestamped later than its own *request*.

http://www.cs.unc.edu/~anderson/papers/lamport.pdf

Optimierungen / Vereinfachungen

- Lamport: 3(n-1) messages
- Ricart-Agrawala: 2(n-1) messages,
  verzögere ACK für nachrangige Requests
  fasse ACK mit ReleaseMessage zusammen
  Kosten: 1 Request-Deferred Array pro Prozess
- Singhal: CS-häufige und CS-seltene Prozesse, im Mittel n-1 Messages
- Raymond: aufspannenden Baum berechnen

Deadlocks

Deadlocks

The difficulty of finding data races and deadlocks is well known. Detecting such errors with testing is hard since they often depend on intricate sequences of low-probability events. This makes them sensitive to timing dependencies, workloads, the presence or absence of print statements, compiler options, or slight differences in memory models. This sensitivity increases the risk that errors will elude in-house regression tests yet make grand entrances when software is released to thousands of users.

Engler, Ashcraft (2003), RacerX, Detection of Race Conditions and Deadlocks

Deadlock-Bedingungen

- **Exclusive use**:
  nur 1 Prozess kann die Ressource benutzen

UND

- **Hold and wait**:
  Prozess kann Ressourcen festhalten,
  während er auf weitere Ressourcen wartet

UND

- **No preemption**:
  kein Prozess kann bei anderen die Nutzung
  der Ressource unterbrechen

UND

- **Cyclical wait**: wartender Ring von Prozessen

Kriterien zur Unterscheidung von Deadlock-Algorithmen

- Anzahl zusätzlicher Nachrichten
- Dauer des Deadlocks
- Speicher
- Berechnung (Zykel finden)
- Aufwand des Auflösens

Deadlocks

- Typen:

  Ressource Deadlock

  Communication Deadlock

- Ziel?
  - Deadlocks vermeiden, oder
  - Deadlocks zulassen und erkennen

- Deadlocks auflösen leicht
  (auf Kosten eines Prozesses)