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

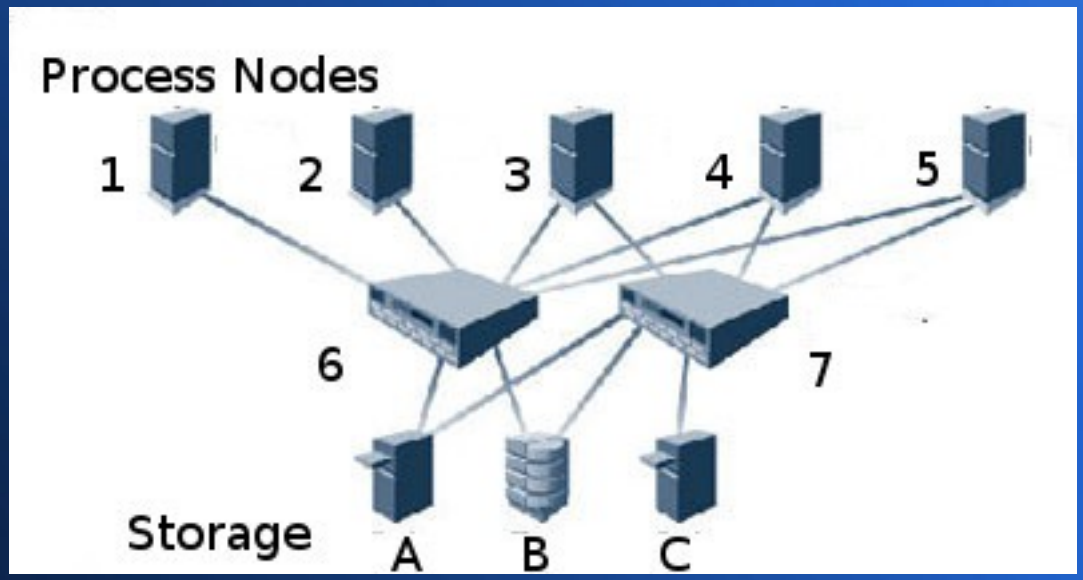
mit atomaren Operationen (P,V)

bzw **wait, acquire, down** für **P**

und **signal, release, post, up** für **V**

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

Mutual-Exclusion-Modell
CS = critical section

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

Vermeidung von...

- Starvation
- Deadlock
- Unfairness

Safety, Liveness und Fairness

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

  (kein endloses Warten wg.
  Deadlock oder Starvation)

- Fairness: Requests lt. Reihenfolge bedient

Ansätze

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

Token

- sende Token in (evtl. logischem) Ring

  Algorithmus, um logischen Ring zu erzeugen?

- wer das Token hat, hat den Zugriff

Beschleunigung (Suzuki/Kasami)

- Anforderung CS: Broadcast-Request für Token
- Besitzer des Tokens:
  - halte Token zurück, solange in CS
  - sende Token an den, der Request sendete

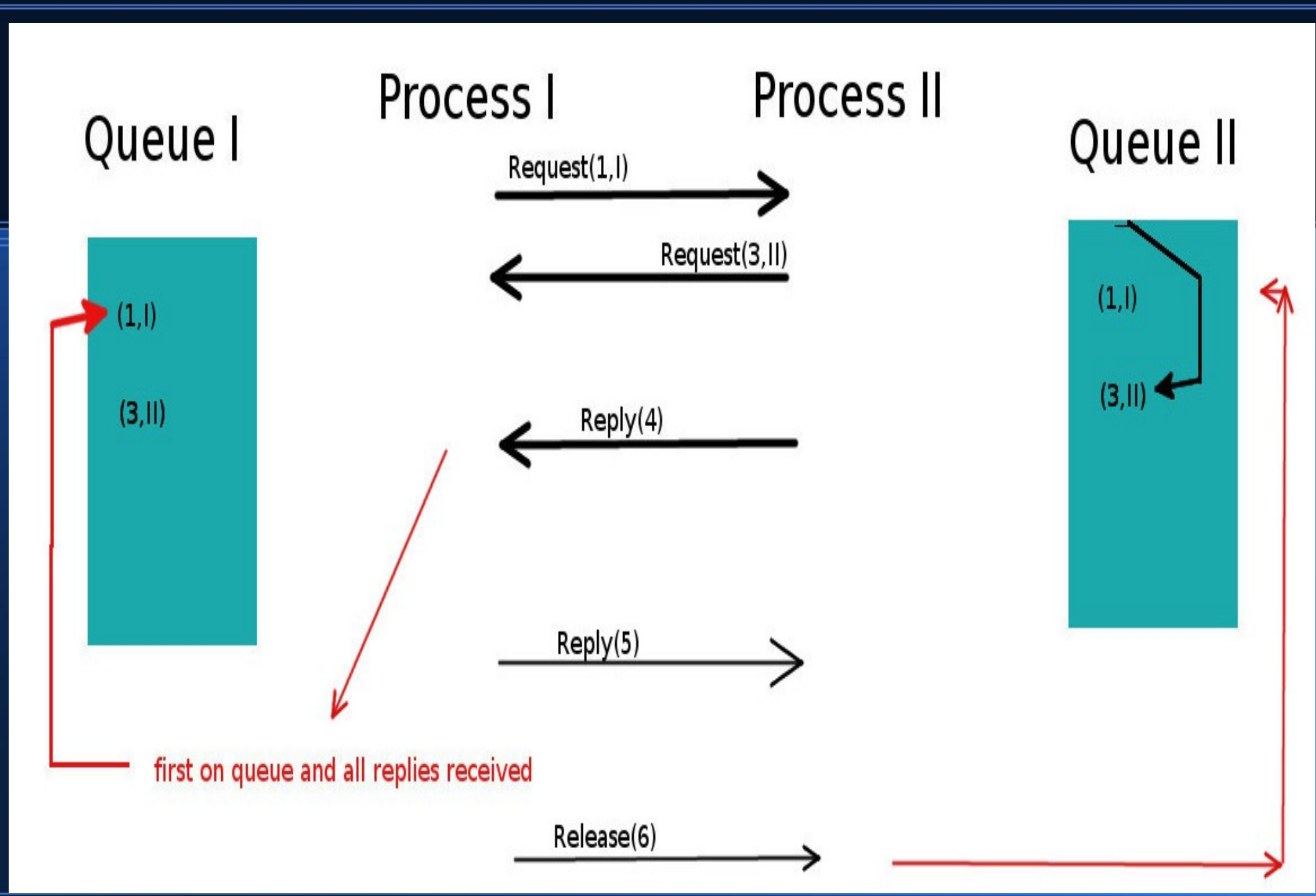
Non-Token: Zentraler Server

- 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


Process I
Process II
Queue I
Queue II
Request(1,I)
Request(3,II)
(1,I)
(3,II)
Reply(4)
Reply(5)
(1,I)
(3,II)
first on queue and all replies received
Release(6)

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