# Deadlock-Prinzip:
## Zulassen und Auflösen anstatt vermeiden

# Deadlock-Bedingungen vermeiden

- **Exclusive use**:
  wahrscheinlich systemimmanent, Ursache des Problems
- **Hold and wait**:
  P soll alle benötigten Ressourcen gleichzeitig nutzen,
  P soll Ressourcen einzeln und nacheinander nutzen
- **No preemption**:
  Freigabe von Ressourcen, falls Warten auf andere,
  Unterbrechen eines anderen wartenden P
  ( mit realen Zeitstempeln siehe wait/die, wound/wait )
- **Cyclical wait**:
  nummeriere Ressourcen, nur in aufsteigender ID anfragen

eine dieser vier Bedingungen muss ausgeschlossen werden

# Deadlocks: Simulationsstrategie, Prioritäten

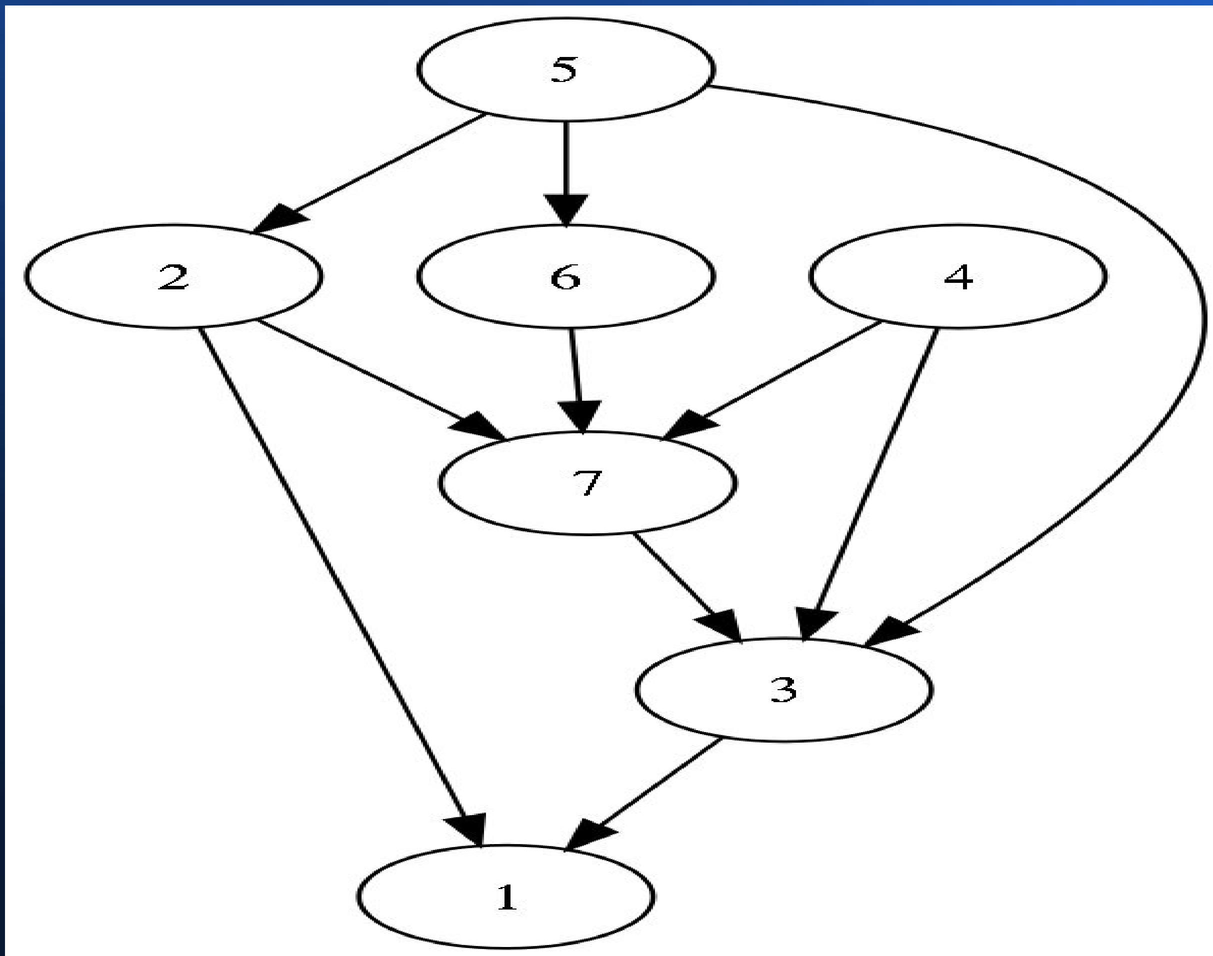
- Simulationsstrategie:
  nur Deadlock-freie Requests erfüllen
  - simuliere Allokieren der Ressource
  - informiere alle anderen Prozesse
  - skaliert nicht, ineffizient
- Prozesse mit Prioritäten
  - höhere Priorität schlägt niedrige,
    Starvation möglich
  - Problem: neu eingefügte Prozesse, neue
    Prioritäten

# Deadlocks erkennen

Kriterium:

Cyclical Wait erkennen (stabiles Prädikat)

# Transaction-Wait-for-Graph (WFG)

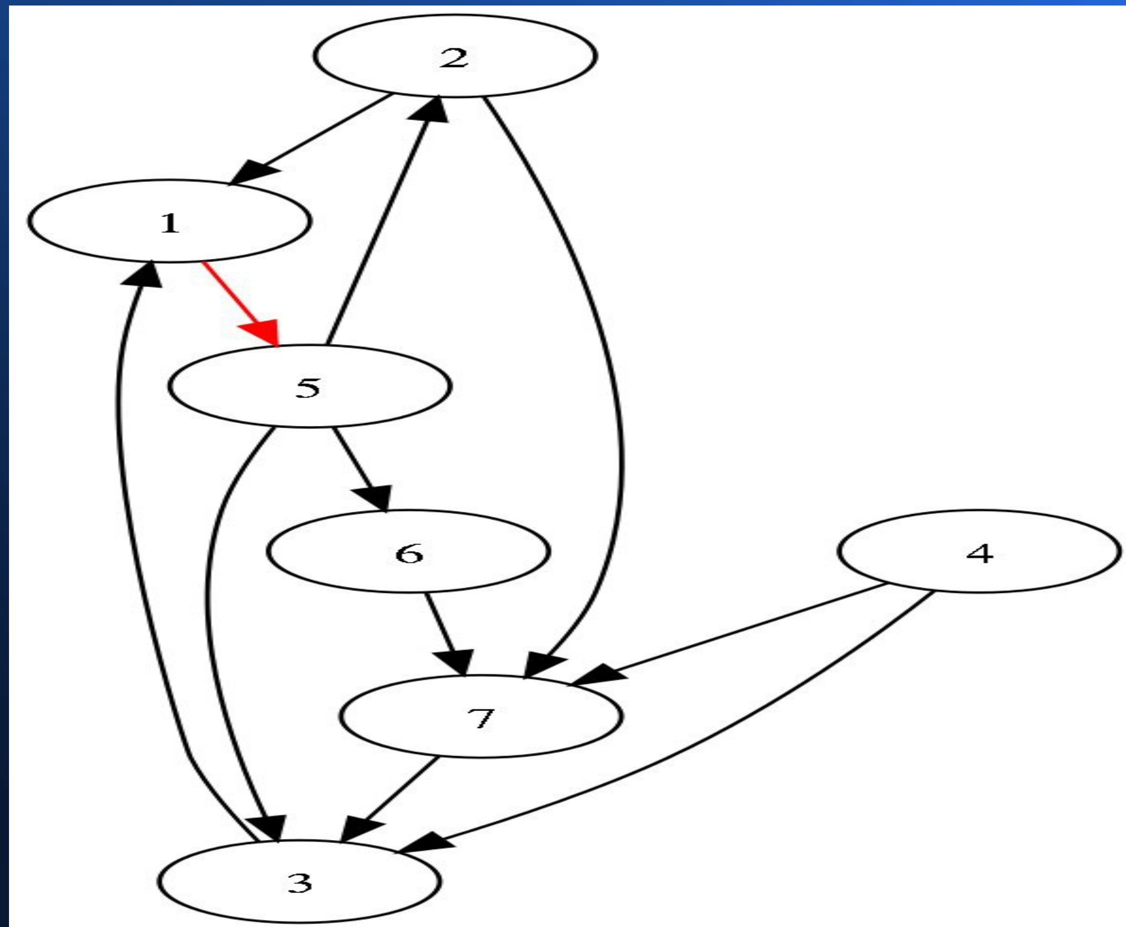



Knoten 5

wartet auf

Knoten 2

# Wait-for-Graph mit Deadlock




Beachte: auch blockierte Prozesse außerhalb des Zykels vorhanden

# Liveness und Safety

- Liveness

  jeder Deadlock wird schließlich entdeckt

- Safety

  jeder Deadlock-Meldung entspricht

  tatsächlich ein Deadlockzustand

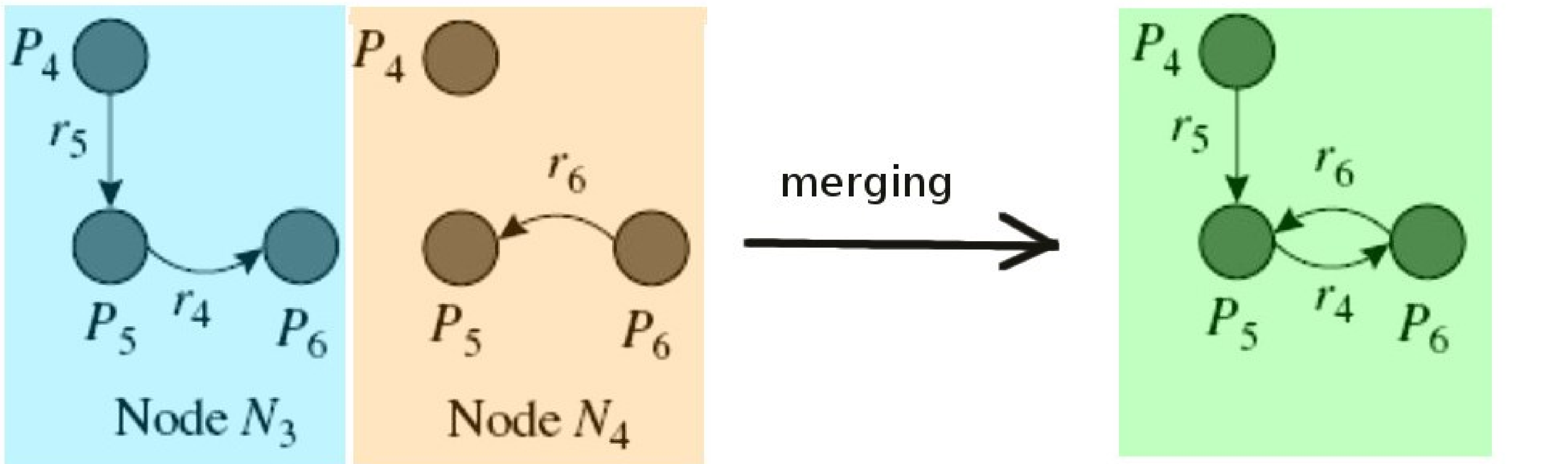
# Schnappschuss-Algorithmus?

- globalen Zustand erkennen
- Prozesse haben lokalen WFG im Zustand
- Sammeln aller lokalen Zustände
- Setze aus lokalen WFGs globalen WFG zusammen (merge)

# Wait-for-Graph evaluieren
# Zykel finden

- Zykel vorhanden?
- Graphentheorie:
  - Azyklische Graphen topologisch sortierbar
    - Finde Knoten mit Eingangsgrad 0 ... etc
  - Depth-First-Search findet Rückwärtskanten
    - Rückwärtskante ist Teil eines Zykels
- Zykel finden?    Pfad in Graph finden

# Merging von WFGs




Kann funktionieren, Gefahr: Safety-Eigenschaft nicht erfüllt

# Klassifikation Deadlock-Erkennung

(Knapp, 1987)

- Global-State-Detection (siehe vorher)
- Path-Pushing
- Edge-Chasing
- Diffusion-Computation

# Path-Pushing

- falls blockiert, WFG an Nachbarn

- WFG empfangen:
  - update eigenen WFG
  - Resultat weitersenden

- Knoten lokal:
  - Zykelfindung periodisch
  - Zykelerkennung: Benachrichtigung Nachbarn

# Edge-Chasing

- falls blockiert:
  - Prüfnachricht X über WF-Kante(n) senden
    - Prozess nicht-blockiert : X verwerfen
    - Prozess blockiert: X weiter an WF-Nachbarn
  - X zum Sender zurückgekehrt: Deadlock

- Prüfnachricht X kurz
- lokaler Algorithmus betroffen

# Diffusion Computation

- Diffusion: Verbreitung von query-messages

  auf dem angenommenen Wait-For-Graph

- ECHO-Typ Algorithmus

- Reply-Nachrichten auf WFG
  (der WFG stellt die Vater-Beziehungen dar)

# Goldman: Edge-Chasing (1)

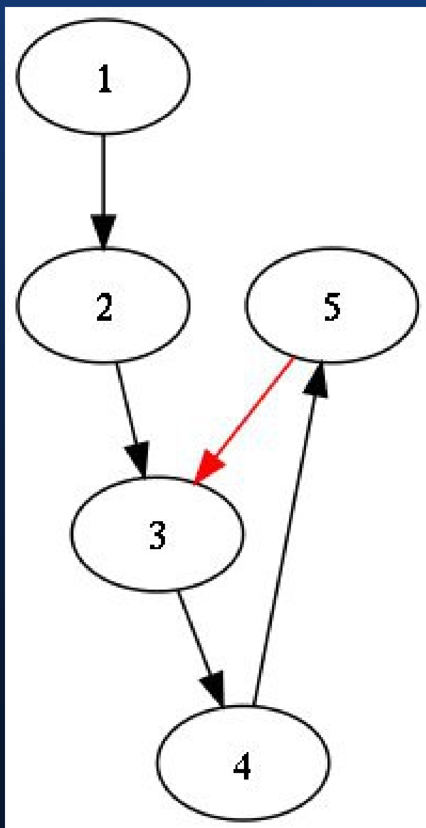
- Voraussetzungen:
  - blockierte P können Kontrollnachrichten senden
  - OBPL = ordered blocked process list

- falls blockiert,
  - sende erweiterte OBPL oder
  - inititiiere OBPL

  an WFG-Nachbarn

# Goldman: Edge-Chasing (2)

- Deadlock : erkenne in OBPL einen WFG-Nachbar

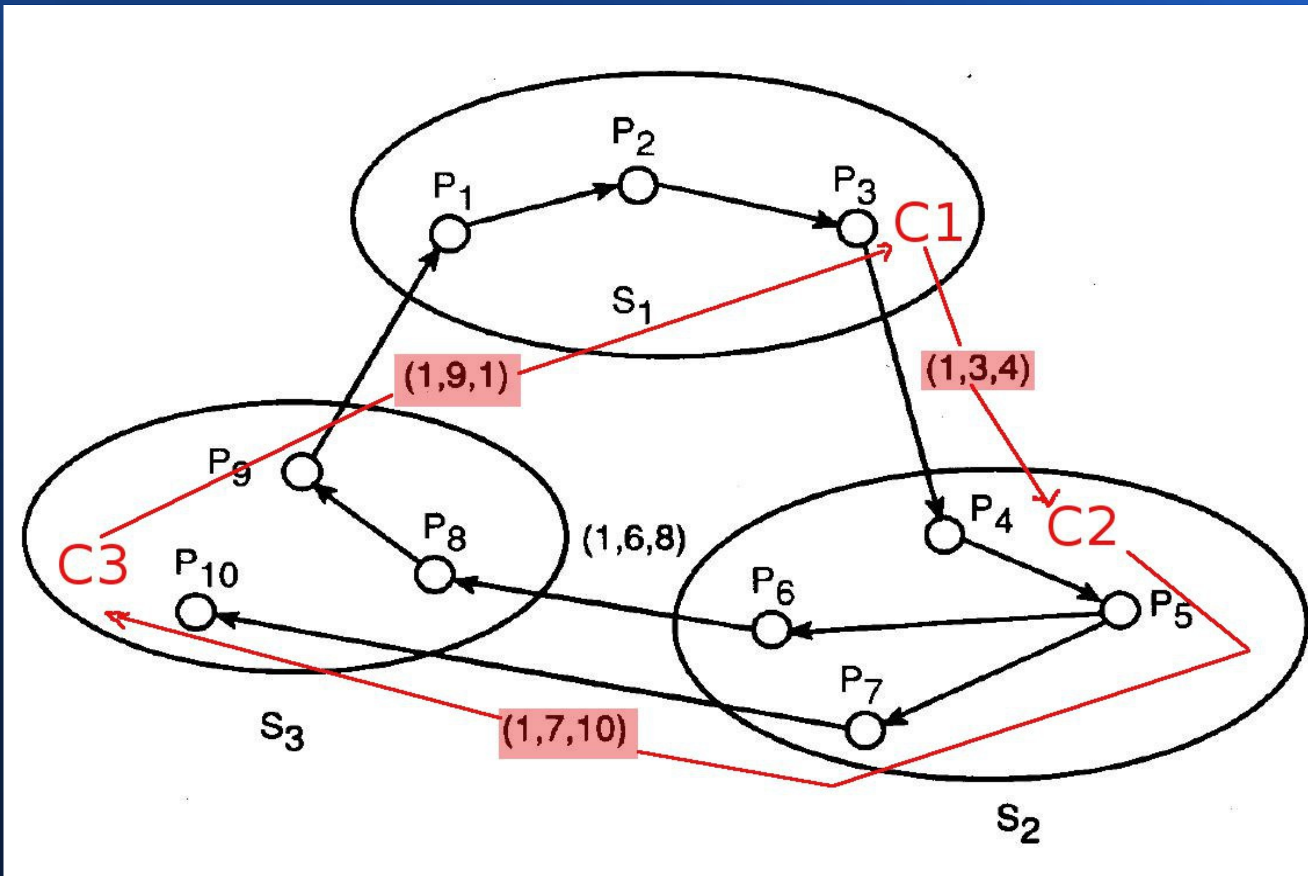



1 -> 2 : OBPL = {1}
OBPL = {1,2}
OBPL = {1,2,3}
OBPL = {1,2,3,4}
OBPL = {1,2,3,4,5}

Prozess 5 erkennt Deadlock:

Prozess 3 ist in OBPL

# Chandy-Misra-Haas: Edge-Chasing für Gruppen lokaler Prozesse




Message (i,j,k):

Prozess i

wartet, weil

(lokales) j auf k

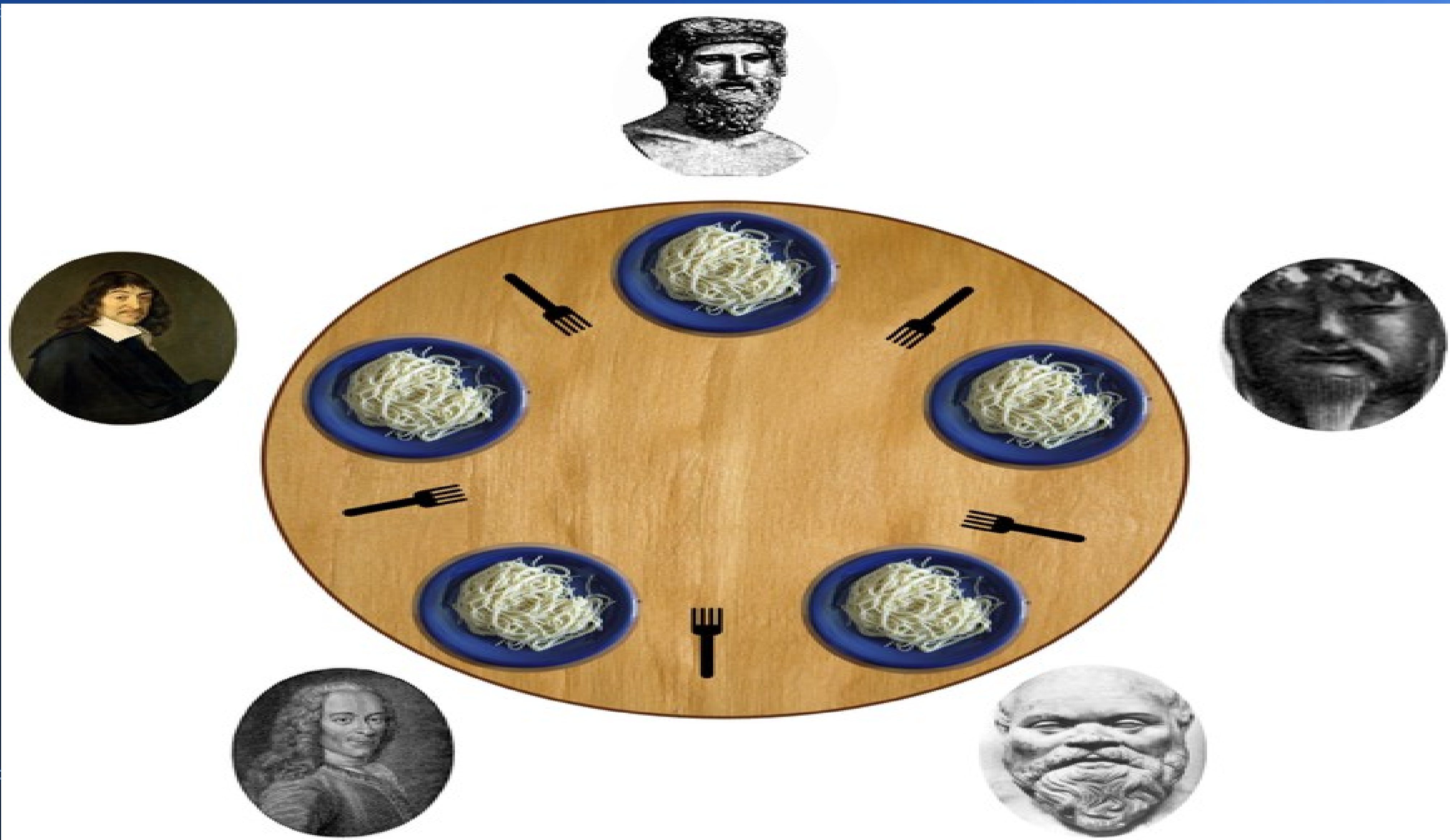
warten muss

C1, C2, C3 Kontrollprozesse für Systeme S1, S2, S3 senden (i,j,k)

# Weitere Modelle (andere Algorithmen)

- AND-Modell
  Prozess wird nur entblockt,
  wenn Request für Ressource R **und** R' erfüllt
- OR-Modell (gleiches Modell mit „**oder**“)
- P-out-of-Q Modell
  Prozess entblockt, wenn P von Q Requests erfüllt

# Philosophen



# Problemstellung

- 2 Zustände: Denken und Essen
- Denken/Essen mit zufälliger Zeitdauer
- zum Essen beide Gabeln nötig (links/rechts)
- nach dem Essen beide Gabeln ablegen
- keiner soll verhungern
- kein Deadlock

# Lösungen

- Semaphore belegt zwei Gabeln gleichzeitig

```
int  semop(int semid, struct sembuf *array, size_t nops);
```

  benötigt gemeinsamen Hauptspeicher,
  *evtl. Starvationproblem*
- Tanenbaum-Lösung: Mutex + 5 Semaphoren
  -> Betriebssysteme-Vorlesung 4. Semester BA
- benötigen beide gemeinsamen Speicher

# Deadlock per Design vermieden: Chandy-Misra (1)

A HYGIENIC SOLUTION TO THE DINERS PROBLEM

- create forks:
  - 1 fork for each phil,
  - phil with lower ID gets fork
  - forks are dirty or clean, initally dirty
  - send clean forks on request
- if want to eat: request missing forks from nb

http://en.wikipedia.org/wiki/Dining_philosophers_problem#Chandy_.2F_Misra_solution

# Chandy-Misra (2)

- on receipt of request for a fork
  - if fork clean, keep fork
  - if fork dirty, clean it, send it
- after eating, both forks are dirty
  on a pending request, a fork is then
  cleaned and sent

http://en.wikipedia.org/wiki/Dining_philosophers_problem#Chandy_.2F_Misra_solution

# Chandy-Misra-Philosophenlösung Eigenschaften

- skaliert
- keine Starvation (wg. dirty/clean)
- beweisbar frei von Deadlocks

# Zusammenfassung

- Falls Deadlock im Design vermeidbar
  → beste Lösung

- Falls Deadlock-Situationen drohen:
  - Zulassen, Erkennen und Auflösen ist besser...
  - als Vermeidungsstrategien zur Laufzeit