Principper for Samtidighed og Styresystemer

Deadlocks og deadlockhåndtering

René Rydhof Hansen

Marts 2008

Husk!

- Forelæsning og øvelser 18. marts er flyttet til 15. maj.

The Great OS Shoot-Out

- Foreløbig dato: 01 APR 2008
- Hver gruppe præsenterer et OS
  - Mac OS X
  - Win
  - Linux
  - BSD
  - ...
  - Skal kunne præsenteres on-line (hint: VMWare)
- Valg af OS koordineres og godkendes af underviser
  - Send mail med tre OS forslag senest tirsdag 18 MAR 2008
- En opponentgruppe pr. OS udpeges
- Præsentation skal relatere til kursus-indhold
  - Filsystem(er)
  - Memory management
  - Processer, tråde, gensidig udelukkelse
  - Deadlocks, deadlockhåndtering
  - Wow-effekt

Opgaver

- Opgave 1: Implementation af gensidig udelukkelse
- Opgave 2: Memory mapping

Mål

- At kunne definere deadlock-begrebet
- At kunne forklare og anvende følgende løsningsstrategier:
  - Prevention
  - Avoidance
  - Detection and Recovery
- At kunne definere livelock og starvation
- At kunne definere priority inversion
- At kunne anvende dining philosophers som forklaringsmodel

Deadlock

Definition

En mængde tråde P er i en deadlock-tilstand hvis hver tråd i P venter på en hændelse som kun kan genereres af en anden tråd i P.

```
void producer()
{
  while(true)
  {
    mutex.wait();
    free.wait();
    buffer[next_free] = data;
    next_free = (next_free+1)%n;
    used.signal();
    mutex.signal();
  }
}
```

```
void consumer()
{
  while(true)
  {
    mutex.wait();
    used.wait();
    data = buffer[next_used];
    next_used = (next_used+1)%n;
    free.signal();
    mutex.signal();
  }
}
```

Deadlock

Definition

En mængde tråde P er i en deadlock-tilstand hvis hver tråd i P venter på en hændelse som kun kan genereres af en anden tråd i P.

```
void producer()
{
  while(true)
  {
    mutex.wait();
    free.wait();
    buffer[next_free] = data;
    next_free = (next_free+1)%n;
    used.signal();
    mutex.signal();
  }
}
```

```
void consumer()
{
  while(true)
  {
    mutex.wait();
    used.wait();
    data = buffer[next_used];
    next_used = (next_used+1)%n;
    free.signal();
    mutex.signal();
  }
}
```

Ressourcetildelingsgraf

Definition (Ressourcetildelingsgraf)

- Graf (V, E) med tråde P og ressourcer R som knuder: $V = P \cup R$
- En kant $(p, r) \in E$ hvis tråden $p \in P$ ønsker adgang til ressourcen $r \in R$
- En kant $(r, p) \in E$ hvis tråden $p \in P$ har adgang til ressourcen $r \in R$

- Cykler i ressourcetildelingsgrafen
  - Trådene er i en deadlock-tilstand hvis og kun hvis der er en cykel i ressourcetildelingsgrafen[1]

[1] Forudsat, at der kun er een instans af hver ressource

Nødvendige og tilstrækkelige betingelser for deadlock (Coffman's betingelser)

- Gensidig udelukkelse
  - Ressourcer kan ikke deles
- Ingen preemption
  - Ressourcer kan ikke fratages en tråd
- Hold og vent (stykvis allokering)
  - Tråde kan altid prøve at tage flere ressourcer
- Cirkulær venten
  - Der er en cirkulær afhængighed af ressourcer

- Konsekvenser?

Løsningsstrategier

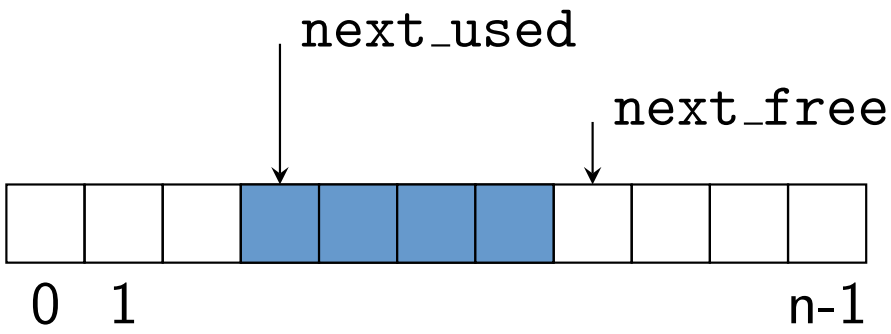
- Prevention
  - Design så deadlock er umuligt
  - Forbyd en (eller flere) af de nødvendige betingelser for deadlock
- Avoidance
  - Begrænset og kontrolleret "udlån" af ressourcer
  - Blokér en tråd med potentielt farlige allokeringsønsker
- Detection and Recovery
  - "Lad os se hvor galt det går"

Deadlock prevention

- Gensidig udelukkelse
  - Gør ressourcer delelige
  - Kritiske regioner er altid udelelige ressourcer
- Preemption
  - Tillad at ressourcer kan fjernes ved tvang
  - Kun muligt hvis ressourcers tilstand kan genskabes
- Hold og vent
  - Alle ressourcer skal tages på een gang
  - Spild af ressourcer
  - Potentielt lang ventetid
- Cirkulær venten
  - Ressourcer allokeres i en specifik rækkefølge
  - Kræver at alle ressourcer er kendt på forhånd

Eksempel: Bounded buffer and producer/consumer

```
thread INIT
{
 used = new Semaphore(0);
 free = new Semaphore(n);
 mutex = new Semaphore(1);
 buffer = new int[n];
 next_used = 0;
 next_free = 0;
}
```

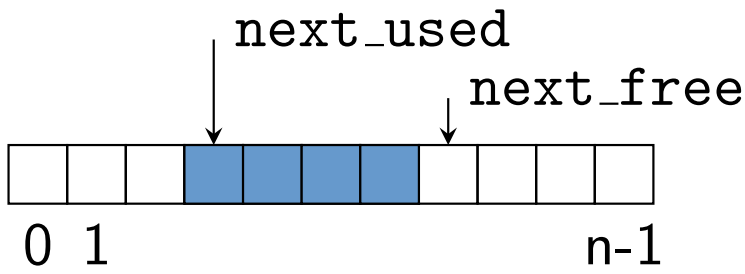



```
thread PROD
{
 while(true)
 {
  free.wait();
  mutex.wait();
  buffer[next_free] = data;
  next_free = (next_free + 1) % n;
  mutex.signal();
  used.signal();
 }
}
```

```
thread CONSUMER
{
 while(true)
 {
  used.wait();
  mutex.wait();
  data = buffer[next_used];
  next_used = (next_used +1 ) % n;
  mutex.signal();
  free.signal();
 }
}
```

Eksempel: Producer/Consumer

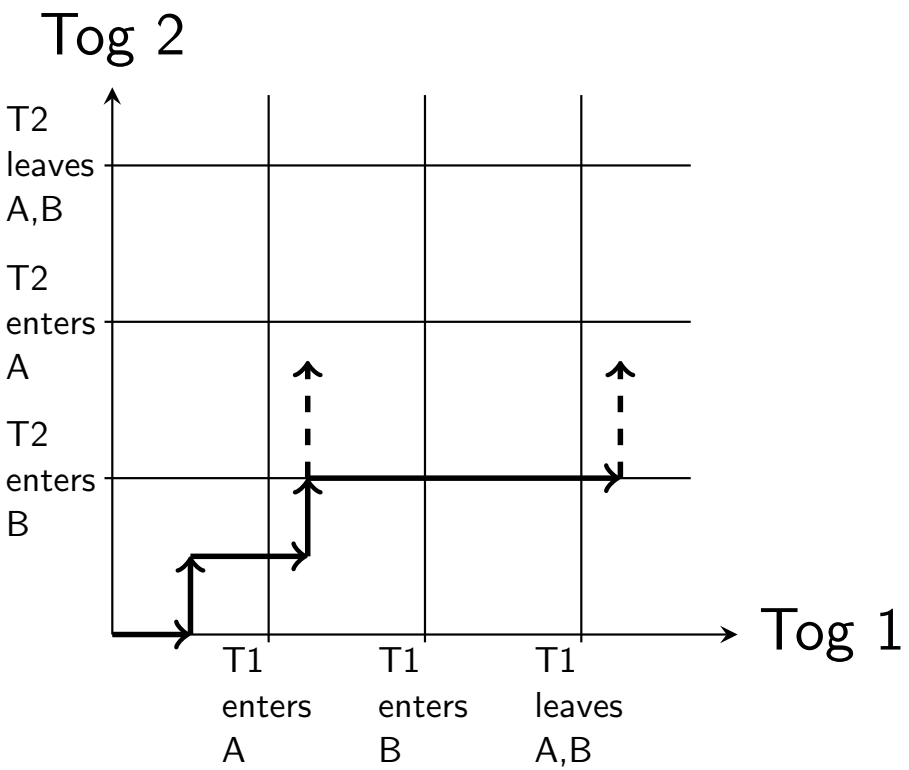
- Gensidig udelukkelse
  - Celler kan kun bruges af én tråd af gangen
  - Den kritiske region skal beskyttes
- Preemption
  - Variablene i en kritisk region kunne være i en inkonsistent tilstand hvis den ufrivilligt "sparkes ud" af den kritiske region
- Alloker alle ressourcer på en gang
  - Udfør `wait()` på begge semaforer i en operation
- Prædefineret allokeringsrækkefølge
  - Netop den valgte strategi for eksemplet



Deadlock avoidance

Definition (Sikker tilstand)

En tilstand hvori der findes mindst én allokeringssekvens så alle tråde gøres færdige uden at skabe en deadlock.

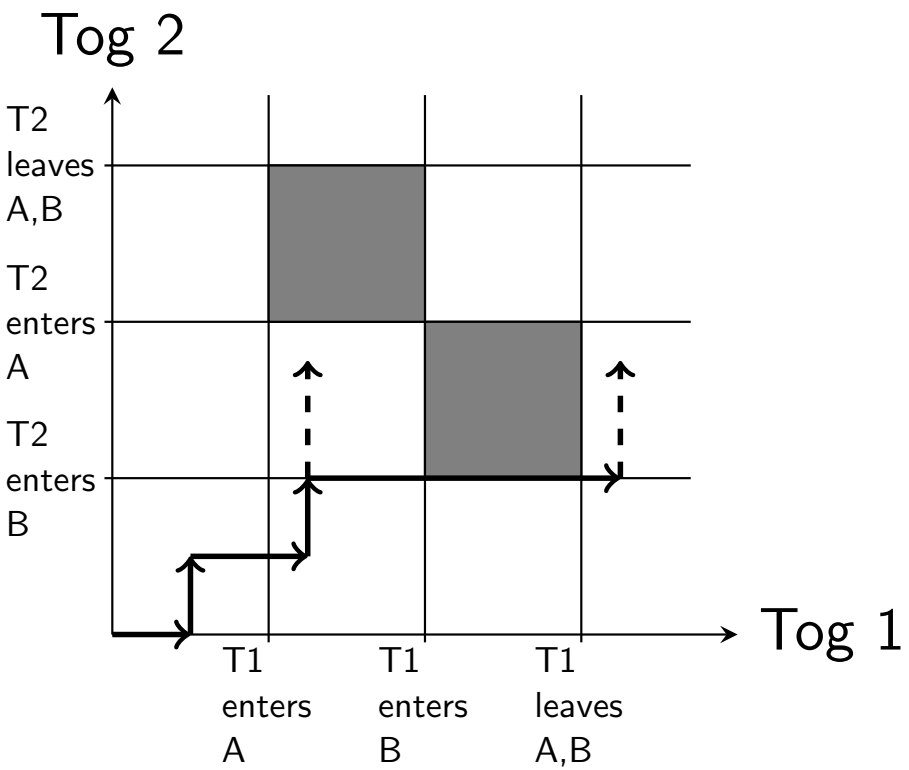


- Deadlock?
- Unreachable?
- Danger Zone?

Deadlock avoidance

Definition (Sikker tilstand)

En tilstand hvori der findes mindst én allokeringssekvens så alle tråde gøres færdige uden at skabe en deadlock.

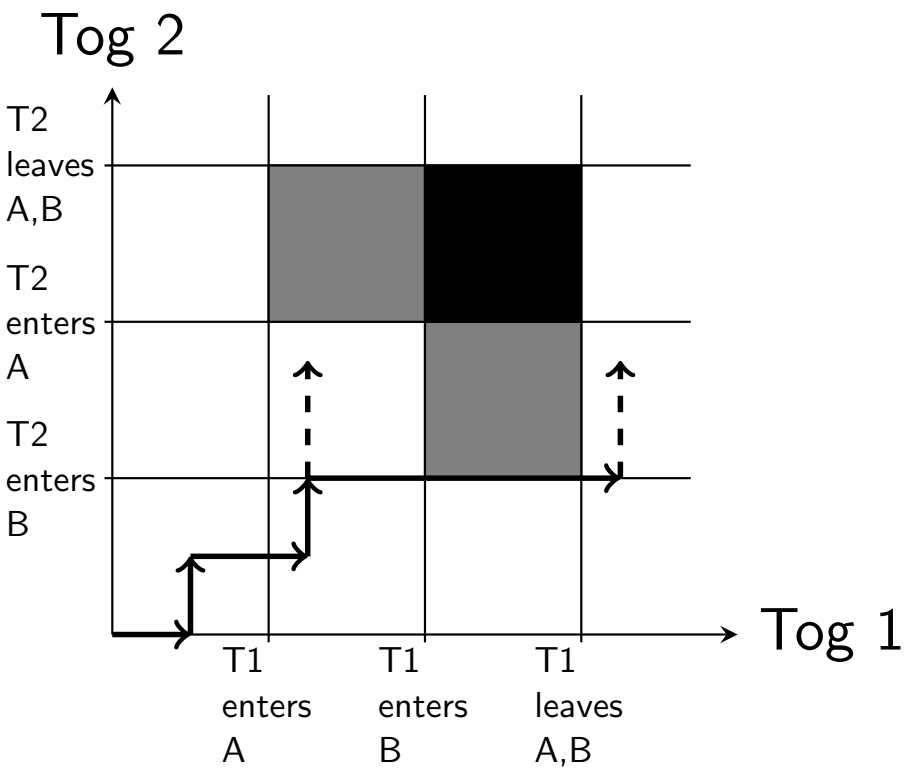


- Deadlock?
- Unreachable?
- Danger Zone?

Deadlock avoidance

Definition (Sikker tilstand)

En tilstand hvori der findes mindst én allokeringssekvens så alle tråde gøres færdige uden at skabe en deadlock.

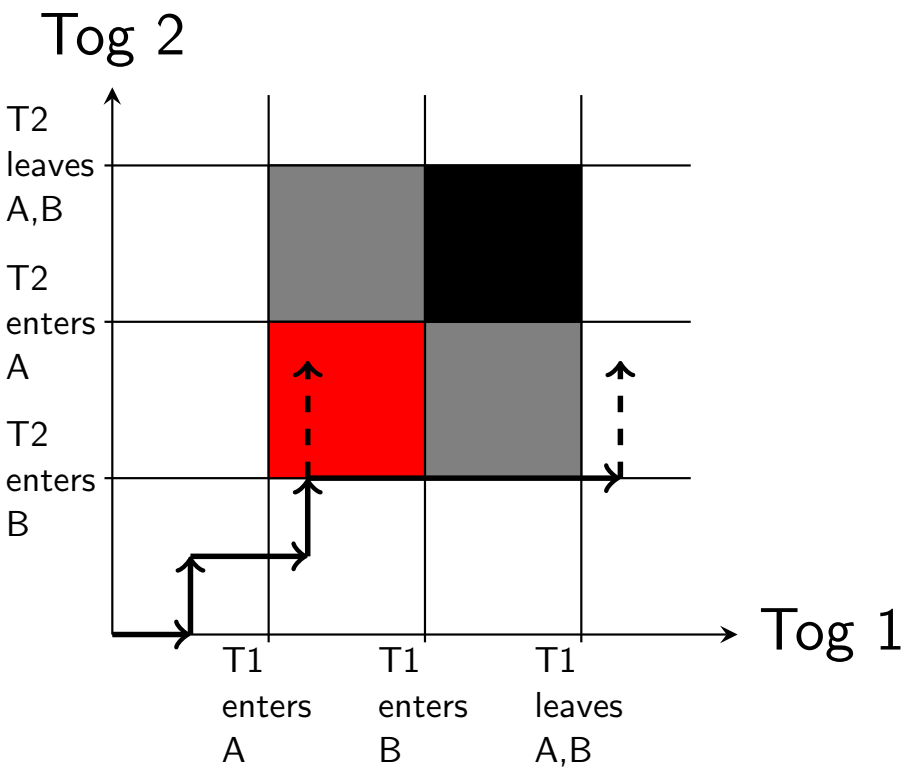


- Deadlock?
- Unreachable?
- Danger Zone?

Deadlock avoidance

Definition (Sikker tilstand)

En tilstand hvori der findes mindst én allokeringssekvens så alle tråde gøres færdige uden at skabe en deadlock.



- Deadlock?
- Unreachable?
- Danger Zone?

Sikker tilstand

Definition

Et system er i en sikker tilstand, hvis der findes en ordning $P_1, P_2, \ldots, P_n$ af systemes tråde således at

$$\forall i: \quad \text{free} + \sum_{j=1}^{i} \text{alloc}(P_j) \;\geq\; \max(P_i)$$

dvs. at en tråd P_i kan køre færdig med de nuværende frie ressourcer samt ressourcer holdt af trådene til venstre for P_i.

- Betyder, at alle trådes maksimale ressourceforbrug skal kendes på forhånd

Bankers Algorithm

```
while (P != ∅) {
  Psafe = { p ∈ P | max(p) - alloc(p) ≤ free }
  if (Psafe ==  ∅) then
    FAIL
  else {
    forall p in Psafe {
      execute p
      free = free + alloc(p)
      P = P \ {p}
    }
  }
```

Bankers Algorithm

```
while (P != ∅) {
  Psafe = { p ∈ P | max(p) - alloc(p) ≤ free }
  if (Psafe ==  ∅) then
    FAIL
  else {
    forall p in Psafe {
      execute p
      free = free + alloc(p)
      P = P \ {p}
    }
  }
```

- Opgave: mangler kode til allokering

Deadlock avoidance

- I en sikker tilstand har styresystemet altid en udvej der undgår deadlock
- Vil der altid ske deadlocks i en usikker tilstand?

Detection and recovery

- Antag at deadlocks er sjældne
- Vent på at deadlock opstår
- Deadlock-tilstande skal opdages
- Normaltilstanden skal genetableres
- Strudsealgoritmen (Ostrich algorithm)
  - Stik hovedet i sandet og håb at problemet forsvinder af sig selv, dvs. overlad det til brugeren

Opdagelse af deadlock

```
deadlock_detection()
{
  Q = { p | alloc(p) > 0 }
  while (Q != ∅) {
    remove some p from Q s.t. request(p) ≤ Free
      unless no such p exist, then deadlock found!
    Free = Free + request(p)
  }
}
```

- Kører som separat tråd

Genskabelse af normaltilstand

- Aborter en af processerne i deadlock-tilstanden
  - Acceptabelt hvis processerne kan genstartes uden at skabe problemer
  - LaTeX kan genstartes, men fx en netværksprotokol kan ikke altid "genstartes"
  - Hvilken process skal termineres?
- Check points
  - Tillader systemet at gentage afbrudte operationer
  - Eller at vende tilbage til en konsistent tilstand fra før operationen blev udført
  - Databasesystemer bruger en transaktionsmodel
  - Recovery points
  - Problematisk hvis processen interagerer med andre processer

Andre fænomener

- Livelock
  - Deadlock-lignende tilstand hvor der ikke sker fremskridt uden at der dog er cyklisk venten på ressourcer
- Starvation
  - Situation hvor en tråd aldrig får tildelt en ressource fordi andre processer altid får den først
- Prioriteter: processer, tråde, ressourcer

Priority Inversion

- En situation hvor en tråd blokerer en anden tråd med højere prioritet
- Opstår når en tråd skal bruge en ressource som en tråd med lavere prioritet holder
- Eksempel: Mars Pathfinder

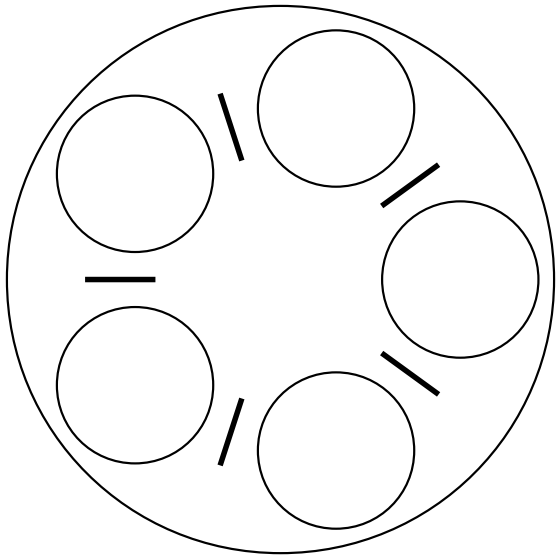
Example

Givet tråde A, B, C så $pri(A) \leq pri(B) \leq pri(C)$. Hvis A holder en ressource som C skal bruge, så kan B køre før C. Hvis C busy-waiter er der livelock.

Priority inversion: Løsninger

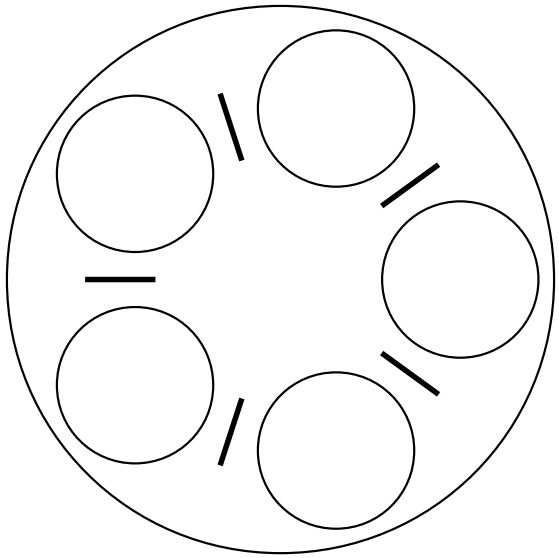
- Interrupt disabling
  - Lad interrupt maskering være den eneste måde at opnå mutex på
  - Reelt kun to prioriteter: preemptible og non-interruptable
- Priority ceiling
  - Ressourcen (mutex'en) har en højere prioritet end alle tråde
  - Tråden tildeles midlertidigt samme (høje) prioritet
- Priority inheritance
  - En lavprioritetstråd T nedarver prioriteten fra højprioritetstråde der blokerer på ressourcer T holder
- Immediate Ceiling Priority Protocol (ICPP)
  - Tildel prioriteter til ressourcer
  - Hæv midlertidigt tråd-prioritet til ressource-prioritet
  - Tråde kan kun bede om ressourcer med højere prioritet

Dining Philosophers [Dijkstra]



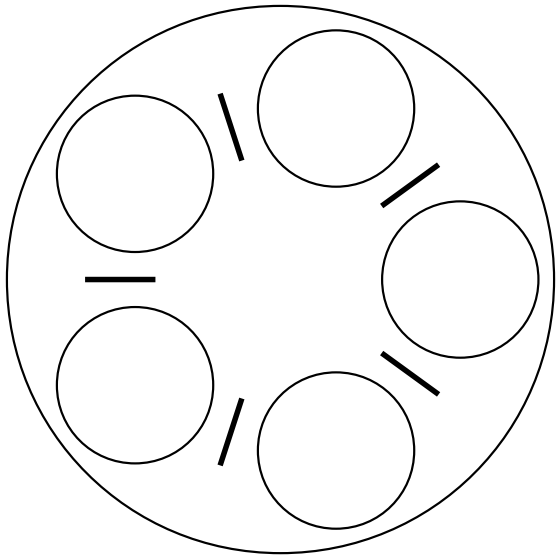
- If each philosopher picks up left chostick, the result is deadlock
  - deny hold-and-wait: put chopstick down if the other is unavailable
  - this results in livelock: pick up, put down, pick up, put down
  - fastest philosopher will eventually break the livelock

Dining Philosophers [Dijkstra]



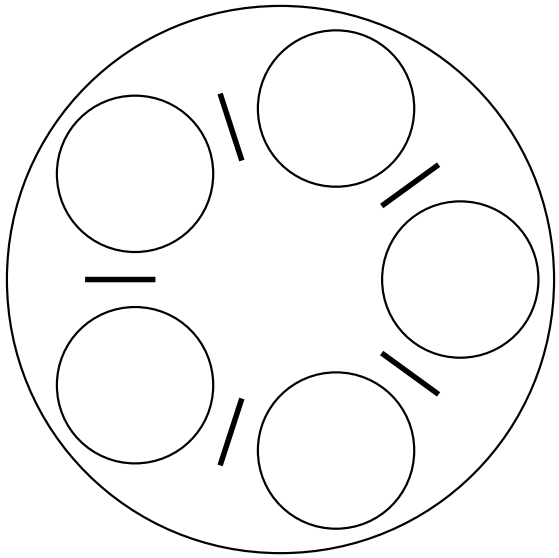
- Allow pre-emption: each philosopher grabs right-hand neighbour's chopstick
  - livelock again: get right chopstick, lose left chopstick, ...
  - strongest philosopher will eventually break the livelock

Dining Philosophers [Dijkstra]



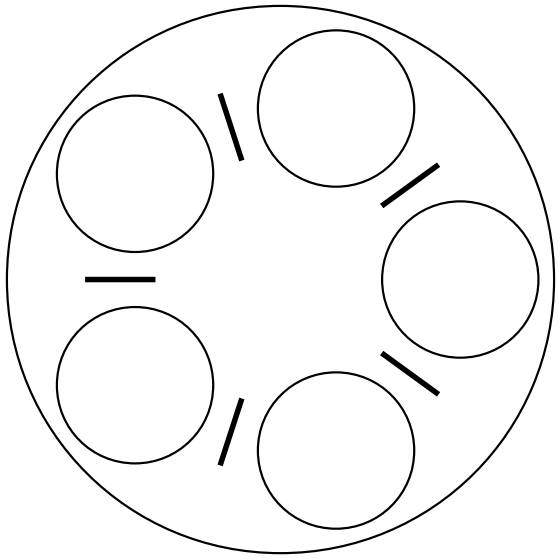
- Allow sharing
  - how to share chopsticks?

Dining Philosophers [Dijkstra]



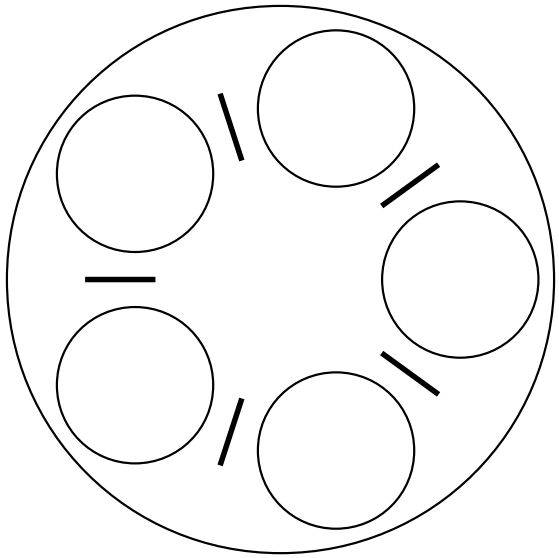
- Avoid circularities
  - add one extra chopstick so both neighbouring philosophers have one chopstick that isn't shared

Dining Philosophers [Dijkstra]



- Add a mutex to arbitrate resource acquisition
  - only the philosopher with the (one and only) soy sauce bottle can pick up chopsticks

Dining Philosophers [Dijkstra]



- Require both chopsticks are available before picking up either: Two philosophers can starve one seated between them

Opsummering: samtidighed

- Trådes relative hastigheder er uforudsigelige
- Identificer delte ressourcer
- Identificer kritiske regioner
- Sørg for gensidig udelukkelse i kritiske regioner
- Brug ikke scheduleringsmekanismer (prioriteter eller frivillig tidsdeling) istedet for mutex
- Antag ikke en ordning i hvilken tråde vækkes
- Undgå busy waiting
- Undgå delte flag-variable til at skabe synkronisering
- Undersøg om et bibliotek er trådsikkert inden det bruges
- Undersøg om der er mulighed for: race conditions, deadlocks, livelocks, starvation, priority inversion

Opsummering og næste gang

- Deadlocks
- Løsningsstrategier for deadlocks
- Livelock, starvation, priority inversion
- Dining Philosophers
- Næste gang: Memory management