Realtidssystem

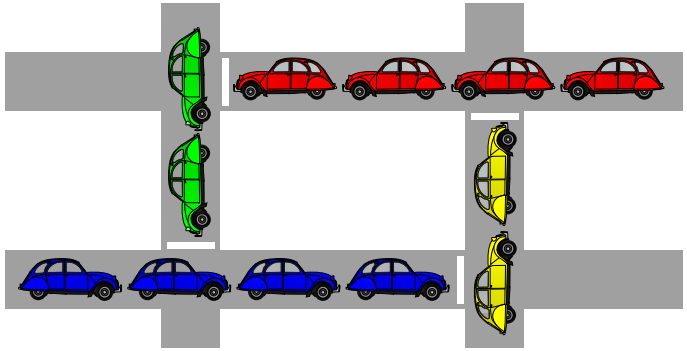
- Dödläge -

EDAF85 - Realtidssystem (Helsingborg)
Roger Henriksson

Föreläsning 5

Kursens innehåll motsvarar tidigare omgångar under beteckning EDA698
Föreläsningsbilder av Elin A. Topp
Stora delar baserad på: Föreläsningsmaterial EDA040 (Klas Nilsson, Mathias Haage) samt EDA698 (Mats Lilja)

Deadlock - dödläge



Om det blir en cirkel i villkoren, eller snarare väntandet på att villkoren uppfylls, hamnar systemet i ett dödläge (deadlock)

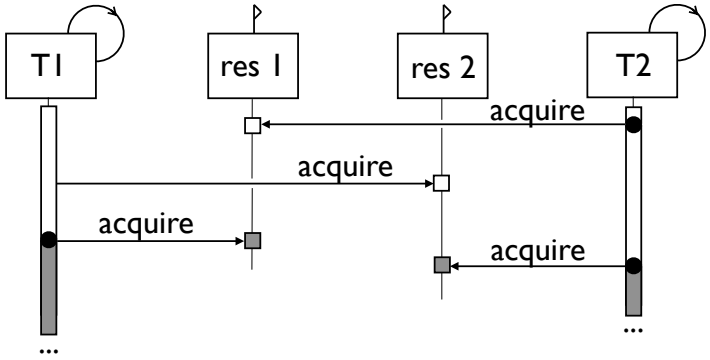
Om det är teoretiskt möjligt att aktörer (trådar, bilar, personer) måste vänta på varandra i en cirkel, finns det en risk för dödläge (deadlock risk)

Dödläge - definition och bakgrund

- Ömsesidig uteslutning betyder att en tråd blir blockerad
  - En tråd får inte komma in i en kritisk region om en annan tråd hindrar det
  - För att systemet (och all data) ska vara konsistent, för att beräkningar och resultat ska vara förutsägbara och för effektivitetens skull, kan tillgång till en sådan resurs inte bli avbruten, eller utsättas för en sk *roll-back*.
- Vi har alltså ingen resource preemption, dvs ingen tar mot vår vilja ifrån oss en delad resurs om vi fått tillgång till den.
- Om blockeringen aldrig tar slut, kan man konstatera ett **dödläge** (deadlock).

Dödläge med semaforer

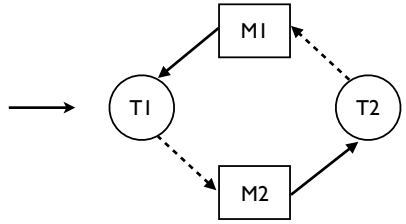
Två trådar vill komma åt två resurser och hamnar i ett speciellt läge (som kan bli dödläge), nämligen i ett ***hold - wait state***



Dödläge med semaforer, kod-exempel

```
      T1                T2
      ...               ...
S2.acquire();     S1.acquire();
S1.acquire();     S2.acquire();
      ...               ...
S1.release();     S2.release();
S2.release();     S1.release();
```

```
      T1                T2
      ...               ...
M1 m1;            M2 m2;
m1.op1();         m2.op1();
      ...               ...
```




```
class M1 {
  M2 m2;
  synchronized void op1() {
    m2.op2();
  }

  synchronized void op2() {
    …
  }
}
```

```
class M2 {
  M1 m1;
  synchronized void op1() {
    m1.op2();
  }

  synchronized void op2() {
    …
  }
}
```


Krav för att dödläge ska uppstå

Följande punkter måste vara "uppfyllda" för att det ska bli dödläge:

1. Ömsesidig uteslutning - enbart en tråd åt gången kan komma åt en resurs.
2. Hold and Wait (håll och vänta) - en tråd kan "reservera" (ta) en resurs och väntar sedan på en annan
3. Ingen resource-preemption (avbrott under resursnyttjande) - en tråd kan inte tvingas att lämna en resurs när den väl har fått den
4. Cirkulärt vänteläge - när det finns en cirkel i tråd-resurs-beroenden

För Java - monitorer gäller följande:

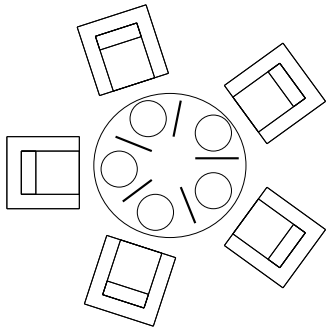
1. Synchronized - enbart en tråd kan komma in, ömsesidig uteslutning är uppfylld
2. Anrop av en metod i en annan monitor inifrån en monitor-metod är möjligt
3. En monitor friges enbart om en tråd lämnar den tillfälligt (wait()) eller lämnar efter avslutat jobb, ingen resource-preemption – än så länge alla villkor uppfyllda…
4. "Cirkulärt vänteläge" är det som man kan åtgärda själv, allt annat KAN man INTE påverka

Dining philosophers - en klassiker...

Philosopher

```
while( true) {
  think();
  preProto();
  eat();
  postProto();
}
```

```
preProto() = take fork(s)
postProto() = give fork(s)
```



Förslag 1 – finns risk för dödläge?

```
Semaphore[] fork = new Semaphore[5];
for(int i=0;i<5;i++) fork[i] = new Semaphore(1);
…
class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while( true) {
      think();
      fork[i].acquire();
      fork[(i+1)%5].acquire();
      eat();
      fork[i].release();
      fork[(i+1)%5].release();
    }
  }
}
```


Fungerar lösning 1?

Nej!

Om alla vill börja äta samtidigt kan dödläge uppstå. Alla tar höger gaffel och väntar sedan på att få tillgång till den vänstra..

Förslag 2 – en vänsterhänt filosof

```
Semaphore[] fork = new Semaphore[5];
for(int i=0;i<5;i++) fork[i] = new Semaphore(1);
…
class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while( true) {
      think();
      fork[i].acquire();
      fork[i+1].acquire();
      eat();
      fork[i].release();
      fork[i+1].release();
    }
  }
}
```

```
class LeftPhilosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while( true) {
      think();
      fork[0].acquire();
      fork[4].acquire();
      eat();
      fork[0].release();
      fork[4].release();
    }
  }
}
```


Fungerar lösning 2?

Ja!

Cirkulär väntan kan aldrig uppstå! Vi har brutit det av de fyra villkoren för dödläge som vi faktiskt kan påverka.

Förslag 3 – bara fyra stolar

```
Semaphore[] fork = new Semaphore[5];
for(int i=0;i<5;i++) fork[i] = new Semaphore(1);
Semaphore chairs = new Semaphore(4);
…
class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while( true) {
      think();
      chairs.acquire();
      fork[i].acquire();
      fork[(i+1)%5].acquire();
      eat();
      fork[i].release();
      fork[(i+1)%5].release();
      chairs.release();
    }
  }
}
```


Fungerar lösning 3?

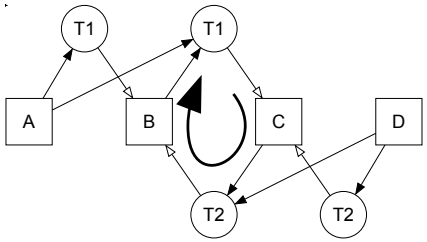
Ja!

Vi kan aldrig forma en komplett cirkel!

Resursallokeringsgrafer

Ett verktyg för att upptäcka cirkulära "hold-wait" situationer, och för att förstå, vilka trådar och resurser kan vara inblandade i ett dödläge,

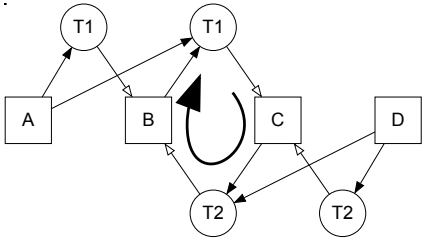
1. Rita resurserna (monitorer, semaforer ...)
2. Rita alla möjliga hold-wait situationer. Pil från en "upptagen" resurs till en markör för tråden som håller i resursen (kan bli flera markörer för en tråd, en för varje hold-wait) och pil från trådmarkör till resursen den väntar på.
3. Cirkel i grafen? Då är det risk för dödläge. "Hold-wait" förbindelserna i cirkeln visar hur många och vilka trådar kan orsaka ett dödläge.



Resursallokeringsgrafer, exempel

```
T1                T2

...               ...
A.acquire();      D.acquire();
B.acquire();      C.acquire();
C.acquire();      B.acquire();
...               ...
C.release();      B.release();
B.release();      C.release();
A.release();      D.release();
```



Ett dödläge kan uppstå om T1 lyckas komma till B.acquire() och blir sen avbruten av T2, som då lyckas komma till C.acquire() och börjar vänta på B (som är upptagen av T1). När då T1 får exekvera igen, väntar den i sin tur på C (som är upptagen av T2).

Resursallokeringsgrafer, exempel 2

```
      T1                T2

      ...               ...
A.acquire();      C.acquire();
B.acquire();      A.acquire();
      ...               ...
A.release();      A.release();
C.acquire();      C.release();
      ...
C.release();
B.release();
```

Är deadlock möjligt om det bara finns en T1-tråd och en T2-tråd?

Hur garanterar vi att deadlock aldrig uppkommer?

Sammanfattning

- Dödläge – villkor för dödläge
- Resource allocation graphs

- Man borde kunna rita RAGs och analysera ett system för att hitta risk för dödläge
- Man borde kunna skriva kod (sekvenser) när man har gjort ett system "riskfritt" med hjälp av en RAG.

- Lästips:
  - e-bok: Kap 4, "Deadlock" (s 79-101)
  - Artikel "Resource Allocation Graphs", Roger Henriksson (länk på kursens materialsida)