

# TDIU16 – Deadlocks

Detektera och undvik

Filip Strömbäck, Klas Arvidsson

# Planering

Vecka	Fö/Se	Lab
13	Fö: C + Syscall (+påsk)	C <sup>1</sup> , intro
14	-	systemanrop
15	Fö + Se: Semaforen	Processhantering
16	Fö: Lås, cond	Processhantering
17	Fö: Deadlock	Processhantering
18	-	Synkronisering
19	Fö: Låsimplementation	Synkronisering, säkerhet
20	Se: Deadlock + tenta	Säkerhet
21	-	Säkerhet

<sup>1</sup>lämpligt att demonstrera första passet

- 1 Vad är problemet?
- 2 Vad är deadlock?
- 3 Banker's Algorithm
- 4 Ytterligare exempel
- 5 Framåt i kursen

## Enkla exempel

- En fyrvägskorsning
- Fyra vägkorsningar
- Två lås, A och B
  - P: Lock A, Lock B .. Rel. A, Rel. B
  - Q: Lock B, Lock A .. Rel. B, Rel. A

Vad motsvarar resurser? Vad motsvarar trådar?

## Banköverföring - tidigare lösning

```
bool transfer(int amount, int from, int to) {
    lock_acquire(&account[from].lock);
    bool t = account[from].balance >= amount;
    if (t)
        account[from].balance -= amount;
    lock_release(&account[from].lock);
    lock_acquire(&account[to].lock);
    if (t)
        account[to].balance += amount;
    lock_release(&account[to].lock);
    return t;
}
```

## Banköverföring - varför inte så här? (FEL)

```
bool transfer(int amount, int from, int to) {
    lock_acquire(&account[from].lock);
    lock_acquire(&account[to].lock);
    bool t = account[from].balance >= amount;
    if (t) {
        account[from].balance -= amount;
        account[to].balance += amount;
    }
    lock_release(&account[to].lock);
    lock_release(&account[from].lock);
    return t;
}
```

## Dining philosophers

- Ett runt bord med obegränsad mängd spaghetti.
- Fem tallrikar (numrerade 0–4).
- Fem gafflar, en mellan varje tallrik.
- Fem filosofer sitter runt bordet (numrerade 0–4). De antingen äter eller tänker. När de tänker blir de hungriga, när de ätit förtsätter de tänka.
- Varje filosof behöver både gaffeln till höger och till vänster om sin tallrik för att kunna äta.

## När en filosof blir hungrig

1. Försök plocka upp höger gaffel
  - Vänta om den är upptagen
2. Försök plocka upp vänster gaffel
  - Vänta om den är upptagen
3. Ät tills du är mätt
4. Lägg ner vänster gaffel
5. Lägg ner höger gaffel
6. Tänk tills du är hungrig igen



# Implementation

```
while (true) {  
    acquire(&right_fork);  
    acquire(&left_fork);  
    eat();  
    release(&left_fork);  
    release(&right_fork);  
    think();  
}
```

## Steg 1: Vad händer? (ibland)

- Filosof 0 tar gaffel 4      trådbyte
- Filosof 1 tar gaffel 0      trådbyte
- Filosof 2 tar gaffel 1      trådbyte
- Filosof 3 tar gaffel 2      trådbyte
- Filosof 4 tar gaffel 3      trådbyte

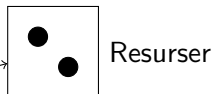
## Resursallokeringsgraf

```
struct semaphore s = 2;
```

```
void p1() {  
    sema_down(&s);  
}
```

```
void p2() {  
    sema_down(&s);  
}
```

```
void p3() {  
    sema_down(&s);  
}
```



p1

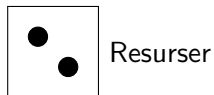
p2

p3

Trådar

## Resursallokeringsgraf

```
struct semaphore s = 2;  
  
void p1() {  
    sema_down(&s);  
}  
void p2() {  
    sema_down(&s);  
}  
void p3() {  
    sema_down(&s);  
}
```

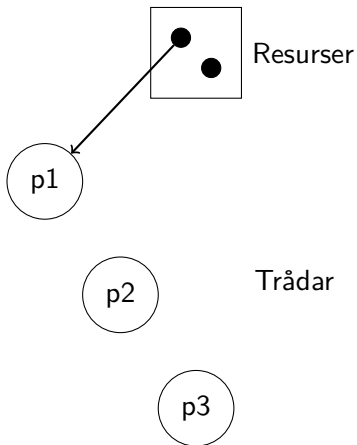


Trådar



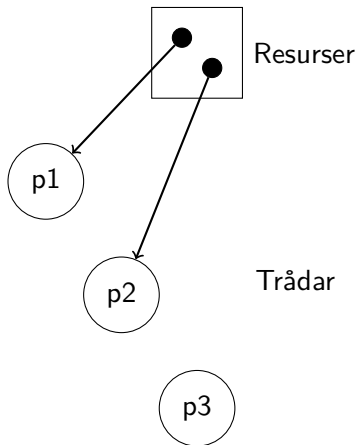
## Resursallokeringsgraf

```
struct semaphore s = 2;  
  
void p1() {  
    sema_down(&s);  
}  
void p2() {  
    sema_down(&s);  
}  
void p3() {  
    sema_down(&s);  
}
```



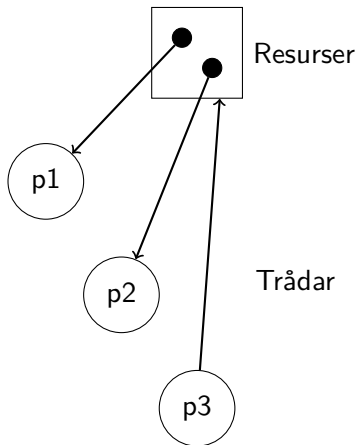
## Resursallokeringsgraf

```
struct semaphore s = 2;  
  
void p1() {  
    sema_down(&s);  
}  
void p2() {  
    sema_down(&s);  
}  
void p3() {  
    sema_down(&s);  
}
```



## Resursallokeringsgraf

```
struct semaphore s = 2;  
  
void p1() {  
    sema_down(&s);  
}  
void p2() {  
    sema_down(&s);  
}  
void p3() {  
    sema_down(&s);  
}
```



## Steg 2: Vad händer? (ibland)

- Filosof 0 tar gaffel 0      trådbyte
- Filosof 1 tar gaffel 1      trådbyte
- Filosof 2 tar gaffel 2      trådbyte
- Filosof 3 tar gaffel 3      trådbyte
- Filosof 4 tar gaffel 4      trådbyte



- 1 Vad är problemet?
- 2 Vad är deadlock?
- 3 Banker's Algorithm
- 4 Ytterligare exempel
- 5 Framåt i kursen

## Deadlock: Tillräckligt villkor

Tillräckligt villkor: *circular wait*

- Det finns en kedja av allokeringar sådan att:
  - Tråd A håller resurs 1 och väntar på resurs 2
  - Tråd B håller resurs 2 och väntar på resurs 3
  - ...
  - Tråd X håller i resurs N och väntar på resurs 1

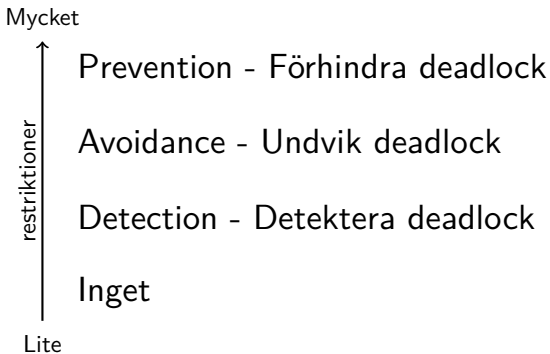
Är *circular wait* uppfyllt finns ett deadlock, och de tre nödvändiga villkoren är automatiskt uppfyllda.

## Deadlock: Nödvändiga villkor

- Mutual exclusion
  - Det finns resurser i systemet som bara får användas av en tråd i taget.
  - Dvs. det finns kritiska sektioner eller lås i systemet.
- Hold and wait
  - Det finns trådar som håller en resurs reserverad och väntar på en annan resurs.
- No preemption *of resources*
  - En *resurs* kan *endast* ges tillbaka frivilligt av den tråd som använder den. Trådar kan inte tvingas ge upp resurser, eller stjäla resurser från någon annan.

Bryts något villkor kan inte deadlock uppstå!

## Vad kan vi göra åt saken?



## Prevention

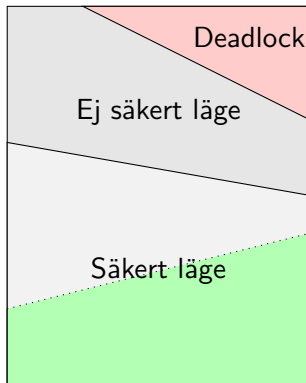
Idé: Vi bryter något av de fyra villkoren för deadlock!

- Mutual exclusion  
Kan vi undvika lås helt?
- Hold and wait  
Trådar får bara hålla en resurs i taget.
- No preemption *of resources*  
Är det möjligt att avbryta och starta om kritiska sektioner? (Transactional memory)
- Circular wait  
Numrera alla resurser och ta dem i samma ordning.

Kan vi använda detta i Philosophers-problemet?

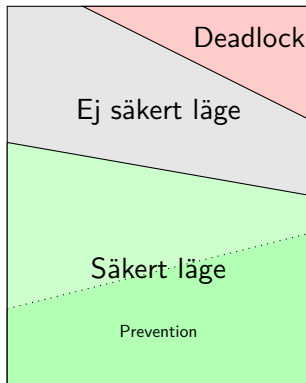
# Prevention

- Bryt något av de fyra villkoren för deadlock!



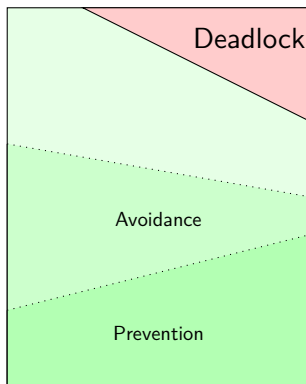
# Avoidance

- Vid resursallokering:  
kontrollera om risk för  
deadlock!
- Banker's algorithm



## Detection

- Kontrollera regelbundet om deadlock har inträffat
- Vi kan använda variant av Banker's
- Eller helt enkelt hitta cykler i resursallokeringsgraf
- Vad gör vi om vi har hittat deadlock?





- 1 Vad är problemet?
- 2 Vad är deadlock?
- 3 **Banker's Algorithm**
- 4 Ytterligare exempel
- 5 Framåt i kursen

# Banker's Algorithm

Givet:

- Alla resurser i systemet
- Alla processer i systemet
- Maximalt resursbehov för varje process
- Nuvarande resursanvändning för varje process

Bestäm:

- Är systemet i ett *säkert läge*?
- I vilken ordning kan vi låta processerna köra klart utan risk för deadlock?

Om systemet inte är i säkert läge **kan** deadlock ha uppstått, men det behöver inte ha hänt. Exakt vad innebär det?

## Banker's Algorithm för avoidance

Idé: Vi vill hålla systemet i ett säkert läge!

När en process frågar efter en resurs:

1. Vi antar att systemet är i säkert läge innan förfrågan.
2. Är systemet fortfarande i säkert läge om vi skulle tillåta förfrågan?

Ja Tillåt förfrågan

Nej Neka förfrågan - låt processen vänta tills någon annan har släppt resurser

## Banker's Algorithm: Idé

1. Håll reda på vilka resurser som är allokerade och av vilken process i en matris.
2. Hitta en process som garanterat kan terminera med de resurser som finns tillgängliga nu. Finns ingen är vi **inte** i ett **säkert läge**.
  - En process kanske behöver reservera sitt maximala antal resurser innan den blir klar.
  - Vi räknar med värsta fallet: antag att den behöver alla resurser innan den avslutar.
3. Antag att den processen kör klart, och därmed frigör sina resurser.
4. Om alla processer har kört klart är systemet i ett **säkert läge**, annars gå till steg 2.

## Banker's Algorithm: Exempel

Maximalt behov:

	A	B	C
P	2	0	3
Q	3	2	5
R	1	5	1

Totalt:

A	B	C
4	5	5

Nuvarande allokeringar:

	A	B	C
P	2	0	1
Q	2	2	0
R	0	1	0

Ledigt:

A	B	C
0	2	4

## Banker's Algorithm: Exempel

1. Är systemet i ett säkert läge?
2. P försöker allokeras en resurs av typ A.  
Ska det tillåtas?
3. R försöker allokeras en resurs av typ C.  
Ska det tillåtas?
4. Q försöker allokeras en resurs av typ C.  
Ska det tillåtas?

## Säkert läge

Säkert läge  $\iff$  det finns ett sätt för alla processer att köra klart även om alla behöver alla sina resurser

Alltså:

Säkert läge  $\implies$  Inga deadlock

Ej säkert läge  $\implies$  Vi hittade inget sätt att köra klart processerna om alla behöver alla resurser. Har vi tur går det ändå, men har vi otur så kan det bli deadlock.

## Banker's Algorithm: Nackdelar

- Förutsätter ett givet antal resurser
  - Ta bort en USB-disk så bryts antagandet
- Utgår från att alla resurser en process/tråd behöver är kända på förhand
  - Kan bero på ex.vis indata från användare...
- Komplexitet  $\mathcal{O}(rp^2)$ ,  $r$  = antal resurser,  $p$  antal processer/trådar



## Banker's för detektering av deadlock

- I stället för "återstående resursbehov" används information om vilka resurser alla processer väntar på just nu.
- I övrigt samma beräkningar. Finns ingen sekvens så finns ett deadlock.

## På tentan

Notera: Banker's algorithm är borttaget ifrån kursmålen och är numera inte längre en del av tentan. Detta ger möjlighet att fokusera mer på synkroniseringsdelen.

- 1 Vad är problemet?
- 2 Vad är deadlock?
- 3 Banker's Algorithm
- 4 Ytterligare exempel
- 5 Framåt i kursen

## Rita resursallokeringsgraf

Ett system har kört ett tag utan att vi har tittat på det, så vi känner bara till de fyra senaste händelserna:

0. (okända händelser)
1. P1 reserverar resurs A (ok)
2. P2 försöker reservera resurs A (väntar)
3. P1 reserverar resurs B (ok)
4. P1 försöker reservera resurs C (väntar)

1. Vilka villkor för deadlock är uppfyllda?
2. Kan deadlock ha uppstått?
3. Kan resurs B ingå i ett eventuellt deadlock?

## Lösning

1. Rita resursallokeringsgraf och se efter!
2. Om P2 sedan tidigare reserverat resurs C så gör de givna händelserna att deadlock har uppstått givet att det bara finns en resurs av A och C.
3. B kan inte ingå i deadlock. Eftersom B var ledig så kan ingen ha väntat på den tidigare. Alltså uppfylls inte *hold and wait* för resurs B, och den kan då inte ingå i *circular wait*.

- 1 Vad är problemet?
- 2 Vad är deadlock?
- 3 Banker's Algorithm
- 4 Ytterligare exempel
- 5 Framåt i kursen

## Framöver i kursen

- Deadline lab 3
- (Första maj)
- Nästa föreläsning: Hur implementeras lås?

Filip Strömbäck, Klas Arvidsson

[www.liu.se](http://www.liu.se)