TDIU16 – Conditions och Deadlocks

Detektera och undvik

Filip Strömbäck, Klas Arvidsson

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Från förra föreläsningen

```
int result;
struct semaphore done;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;

  sema_up(&done);
}
```

```
int main(void) {
  sema_init(&done, 0);
  thread_new(&thread_fn);

  sema_down(&done);
  printf("result=%d\n", result);
  return 0;
}
```

Kan vi lösa detta med lås?

Försök 1 (FEL)

```c
int result;
struct lock done;

void thread_fn() {
  lock_acquire(&done);
  for (int i = 0; i < 100000; i++)
    result += 2;
  lock_release(&done);
}
```

```c
int main(void) {
  lock_init(&done);
  thread_new(&thread_fn);

  lock_acquire(&done);
  printf("result=%d\n", result);
  return 0;
}
```

Försök 2 (FEL)

```
int result;
struct lock done;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;
  lock_release(&done);
}
```

```
int main(void) {
  lock_init(&done);
  lock_acquire(&done);

  thread_new(&thread_fn);

  lock_acquire(&done);
  printf("result=%d\n", result);
  return 0;
}
```

Olika typer av väntande

Vänta på att något ska hända

Exempel: Vänta på att tråd blir klar

Notera: Kräver att någon annan tråd gör något

Lås fungerar *inte*.

Vänta på att resurs blir ledig
(för att få *mutual exclusion*)

Exempel: Vänta tills ingen annan modifierar ett konto.

Kräver *inte* att annan tråd gör något. Vi väntar på att andra trådar är passiva.

Lås passar bra. Felhanteringen hjälper oss!

Condition variable

Har följande operationer (`c = condition`, `l = lock`):

`cond_init(c)` Initiera

`cond_wait(c, l)` Släpp låset, vänta, ta låset igen

`cond_signal(c, l)` Väck en tråd som väntar

`cond_broadcast(c, l)` Väck alla trådar som väntar

Är i princip en **väntekö**. Måste paras ihop med ett lås.

Vänta på ett villkor

Börja med en "dålig" lösning som väntar med *busy-wait*.

1. Identifiera villkoret som koden väntar på, kom ihåg vilka variabler som används
2. Identifiera kritiska sektionener för variablerna, skydda med ett lås
3. Identifiera busy-wait, anropa `cond_wait` i loopen
4. Identifiera ställen där variabler ändras, anropa `cond_signal` eller `cond_broadcast` efteråt

Exempel: Vänta på ett villkor

Steg 0: Skriv dålig lösning

```
int result;
bool done = false;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;

  done = true;
}
```

```
int main(void) {
  thread_new(&thread_fn);

  while (!done)
    ;
  printf("result=%d\n", result);
  return 0;
}
```

Exempel: Vänta på ett villkor

Steg 1: Identifiera villkor

```
int result;
bool done = false;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;

  done = true;
}
```

```
int main(void) {
  thread_new(&thread_fn);

  while (!done)
    ;
  printf("result=%d\n", result);
  return 0;
}
```

Exempel: Vänta på ett villkor

Steg 2: Identifiera kritiska sektioner

```
int result;
bool done = false;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;

  done = true;
}
```

```
int main(void) {
  thread_new(&thread_fn);

  while (!done)
    ;
  printf("result=%d\n", result);
  return 0;
}
```

Exempel: Vänta på ett villkor

Steg 2: Skydda kritiska sektioner

```
int result;
bool done = false;
struct lock done_lock;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;
  lock_acquire(&done_lock);
  done = true;
  lock_release(&done_lock);
}
```

```
int main(void) {
  thread_new(&thread_fn);
  lock_init(&done_lock);
  thread_new(&thread_fn);
  lock_acquire(&done_lock);
  while (!done)
    ;
  lock_release(&done_lock);
  printf("result=%d\n", result);
  return 0;
}
```

Exempel: Vänta på ett villkor

Steg 3: Lägg till cond_wait

```
int result;
bool done = false;
struct lock done_lock;
struct condition cond;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;
  lock_acquire(&done_lock);
  done = true;
  lock_release(&done_lock);
}
```

```
int main(void) {
  thread_new(&thread_fn);
  lock_init(&done_lock);
  cond_init(&cond);
  thread_new(&thread_fn);
  lock_acquire(&done_lock);
  while (!done)
    cond_wait(&cond, &done_lock);
  lock_release(&done_lock);
  printf("result=%d\n", result);
  return 0;
}
```

Exempel: Vänta på ett villkor

Steg 4: Lägg till cond_signal

```
int result;
bool done = false;
struct lock done_lock;
struct condition cond;

void thread_fn() {
  for (int i = 0; i < 100000; i++)
    result += 2;
  lock_acquire(&done_lock);
  done = true;
  cond_signal(&cond, &done_lock);
  lock_release(&done_lock);
}
```

```
int main(void) {
  thread_new(&thread_fn);
  lock_init(&done_lock);
  cond_init(&cond);
  thread_new(&thread_fn);
  lock_acquire(&done_lock);
  while (!done)
    cond_wait(&cond, &done_lock);
  lock_release(&done_lock);
  printf("result=%d\n", result);
  return 0;
}
```

Sammanfattning: Vänta på ett villkor

Väntar på att ett villkor ska bli uppfyllt

Består av tre delar:

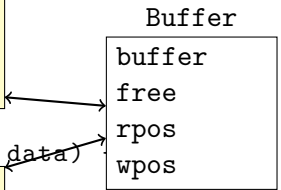
1. Villkoret (ex. `!done`)
   (inkluderar en eller flear *delade variabler*)
2. Lås som skyddar de delade variablerna
   (eller: den kritiska sektionen till vilken variablerna hör)
3. *Condition variable* för att vänta effektivt

Bounded Buffer, analys av osynkroniserad lösning

```
void get(struct Buffer *b) {
  while (b->free == SIZE)
    ; /* Vänta */
  ++b->free;
  return b->buffer[b->rpos++];
}
void put(struct Buffer *b, int data) {
  while (b->free == 0)
    ; /* Vänta */
  --b->free;
  b->buffer[b->wpos++] = data;
}
```

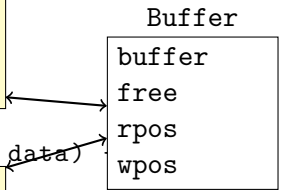
Bounded Buffer, analys av osynkroniserad lösning

```
void get(struct Buffer *b) {
  while (b->free == SIZE)
    ; /* Vänta */
  ++b->free;
  return b->buffer[b->rpos++];
}
void put(struct Buffer *b, int data) {
  while (b->free == 0)
    ; /* Vänta */
  --b->free;
  b->buffer[b->wpos++] = data;
}
```



Bounded Buffer, analys av osynkroniserad lösning

```
void get(struct Buffer *b) {
  while (b->free == SIZE)
    ; /* Vänta */
  ++b->free;
  return b->buffer[b->rpos++];
}
void put(struct Buffer *b, int data)
  while (b->free == 0)
    ; /* Vänta */
  --b->free;
  b->buffer[b->wpos++] = data;
}
```



Bounded Buffer (steg 1)

```
int get(struct Buffer *b) {
  lock_acquire(&b->lock);
  while (b->free == SIZE)
    ; /* Vänta */

  ++b->free;
  int r = b->buffer[b->rpos++];
  lock_release(&b->lock);
  return r;
}
```

Bounded Buffer (steg 2)

```
int get(struct Buffer *b) {
  lock_acquire(&b->lock);
  while (b->free == SIZE)
    cond_wait(&b->cond, &b->lock);

  ++b->free;
  int r = b->buffer[b->rpos++];
  lock_release(&b->lock);
  return r;
}
```

`cond_wait` släpper låset $\Rightarrow$ måste kolla villkoret igen!

Bounded Buffer (steg 3)

```
int get(struct Buffer *b) {
  lock_acquire(&b->lock);
  while (b->free == SIZE)
    cond_wait(&b->cond, &b->lock);

  ++b->free;
  cond_broadcast(&b->cond, &b->lock);

  int r = b->buffer[b->rpos++];
  lock_release(&b->lock);
  return r;
}
```

Bounded Buffer (steg 3)

```
void put(struct Buffer *b, int data) {
  lock_acquire(&b->lock);
  while (b->free == 0)
    cond_wait(&b->cond, &b->lock);

  --b->free;
  cond_broadcast(&b->cond, &b->lock);

  b->buffer[b->wpos++] = data;
  lock_release(&b->lock);
}
```

Bounded Buffer (två villkor)

```
int get(struct Buffer *b) {
  lock_acquire(&b->lock);
  while (b->free == SIZE)
    cond_wait(&b->not_empty, &b->lock);

  ++b->free;
  cond_signal(&b->not_full, &b->lock);

  int r = b->buffer[b->rpos++];
  lock_release(&b->lock);
  return r;
}
```

Bounded Buffer (två villkor)

```
void put(struct Buffer *b, int data) {
  lock_acquire(&b->lock);
  while (b->free == 0)
    cond_wait(&b->not_empty, &b->lock);

  --b->free;
  cond_signal(&b->not_full, &b->lock);

  b->buffer[b->wpos++] = data;
  lock_release(&b->lock);
}
```

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

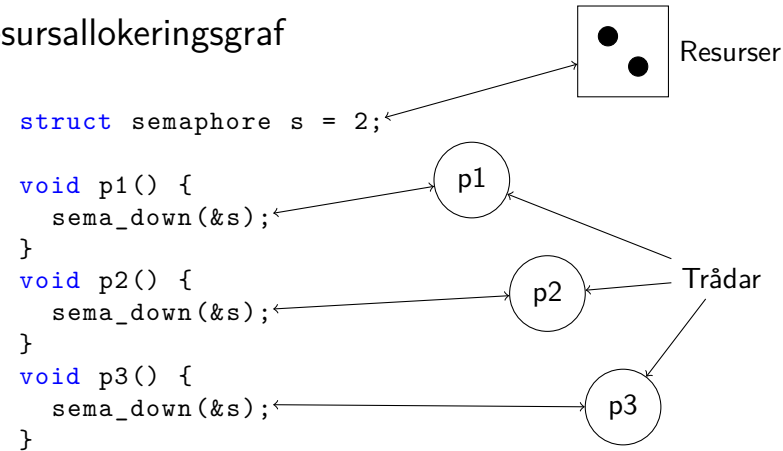
Implementation

```
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



Resursallokeringsgraf

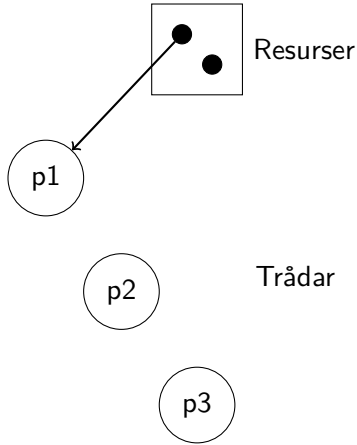
```
struct semaphore s = 2;

void p1() {
  sema_down(&s);
}
void p2() {
  sema_down(&s);
}
void p3() {
  sema_down(&s);
}
```

Resurser

Trådar

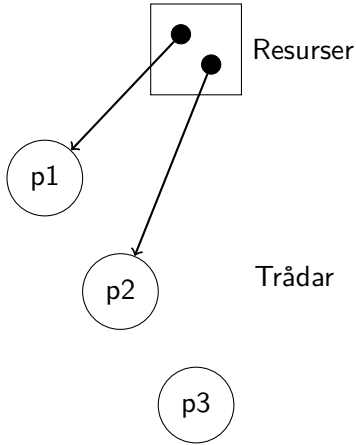
Resursallokeringsgraf

```
struct semaphore s = 2;

void p1() {
  sema_down(&s);
}
void p2() {
  sema_down(&s);
}
void p3() {
  sema_down(&s);
}
```



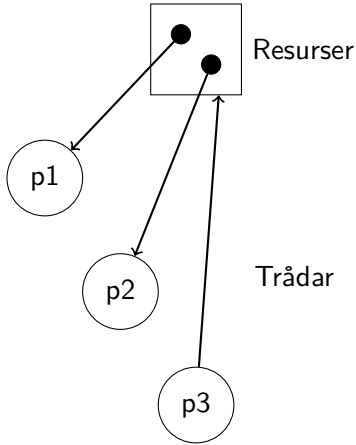
Resursallokeringsgraf

```
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

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

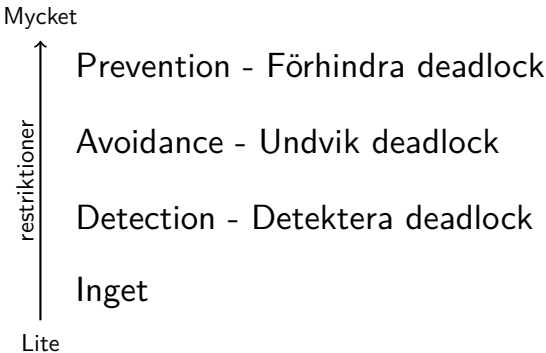
Är *circular wait* uppfyllt finns ett deadlock, och de tre nödvändiga villkoren är automatiskt uppfyllda.

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

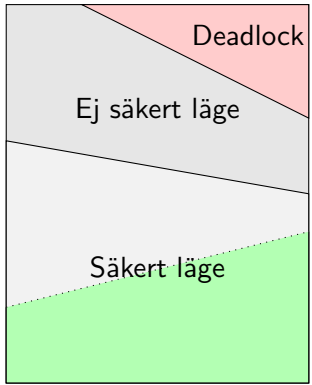
Idé: Vi bryter något av de fyra villkoren för deadlock!

- 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?

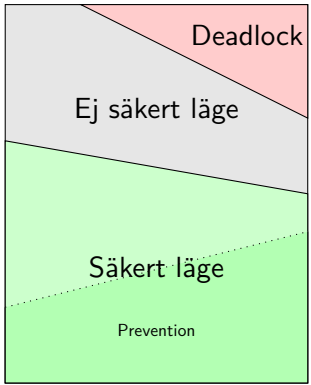
Prevention

- Bryt något av de fyra villkoren för deadlock!



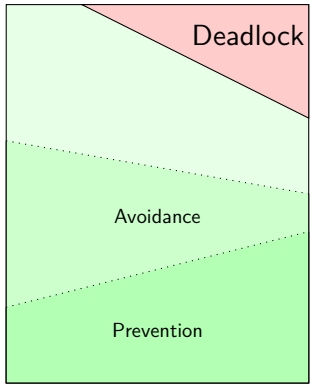
Avoidance

- 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?



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*.

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 allokera en resurs av typ A.
   Ska det tillåtas?
3. R försöker allokera en resurs av typ C.
   Ska det tillåtas?
4. Q försöker allokera 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.

Filip Strömbäck, Klas Arvidsson

www.liu.se

LINKÖPING UNIVERSITY