

# TDIU16 – Synkronisering

Ordning och reda

Filip Strömbäck, Klas Arvidsson

# Planering

Vecka	Fö/Se	Lab
13	Fö: C + Syscall (påsk)	C <sup>1</sup> , halt, exit, console
14	-	console, filhantering
15	Fö: Sema, Se: Process	main, första processen
16	Fö: Lås, cond.	exec, exit
17	Fö: Implementation	exec, exit, wait
18	Fö: Deadlock	Synkronisering
19	-	Synkronisering, accesskontroll
20	Se: Deadlock + tenta	Synkronisering, accesskontroll
21	-	Synkronisering, accesskontroll

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

- 1 Varför synkronisera?
- 2 Kritiska sektioner
- 3 Bounded Buffer
- 4 Condition variables
- 5 Parallellism

# Banköverföring

```
struct account {  
    int balance;  
    char *owner;  
};  
  
struct account account[NUM_ACCOUNTS];  
  
bool transfer(int amount, int from, int to);
```

## Banköverföring

```
bool transfer(int amount, int from, int to) {
    if (account[from].balance >= amount) {
        account[from].balance -= amount;
        account[to].balance += amount;
        return true;
    } else {
        printf("Från-kontot saknar täckning.\n");
        return false;
    }
}
```

## Vad händer? När?

I vilken ordning exekeverar instruktionerna?

- Sekventiellt inom en tråd (som vanligt)
- Odefinierat mellan trådar
  - Trådbyte kan ske när som helst
  - Slumpen avgör
  - Kanske exakt samtidigt om flera CPU finns

Det finns områden i koden där trådbyte *kommer att* orsaka fel

⇒ När gemensamma data används

# Föreläsning 1

- CPU-register sparas undan vid trådbyte
- $\Rightarrow$  Varje tråd har sin egen uppsättning register
- $\Rightarrow$  Olika `eax` för de båda trådarna
- $\Rightarrow$  Två "virtuella" CPU:er

## Vad händer?

Tråd 1:

```
transfer(10, 1, 2)
```

```
if (a[1] >= 10) {
```

```
    a[1] -= 10;
```

```
    a[2] += 10;
```

```
}
```

a[1]:

Tråd 2:

```
transfer(20, 1, 2)
```

```
if (a[1] >= 20) {
```

```
    a[1] -= 20;
```

```
    a[2] += 20;
```

```
}
```

a[2]:



## Vad händer?

Tråd 1:

```
transfer(10, 1, 2)
```

```
if (a[1] >= 10) {
```

```
    a[1] -= 10;
```

```
    a[2] += 10;
```

```
}
```

a[1]: 25→5

Tråd 2:

```
transfer(20, 1, 2)
```

```
if (a[1] >= 20) {
```

```
    a[1] -= 20;
```

```
    a[2] += 20;
```

```
}
```

a[2]: 15

## Vad händer?

Tråd 1:

```
transfer(10, 1, 2)
```

```
if (a[1] >= 10) {
```

```
    a[1] -= 10;
```

```
    a[2] += 10;
```

```
}
```

a[1]: 5 → -5

Tråd 2:

```
transfer(20, 1, 2)
```

```
if (a[1] >= 20) {
```

```
    a[1] -= 20;
```

```
    a[2] += 20;
```

```
}
```

a[2]: 15

## Vad händer?

Tråd 1:

```
a[2] += 10;
```



```
movl a[2], %eax1
```

```
addl $10, %eax1
```

```
movl %eax1, a[2]
```

a[1]:

Tråd 2:

```
a[2] += 20;
```



```
movl a[2], %eax2
```

```
addl $20, %eax2
```

```
movl %eax2, a[2]
```

a[2]:

## Vad händer?

Tråd 1:

```
a[2] += 10;
```



```
movl a[2], %eax1
```

```
addl $10, %eax1
```

```
movl %eax1, a[2]
```

a[1]:

Tråd 2:

```
a[2] += 20;
```



```
movl a[2], %eax2
```

```
addl $20, %eax2
```

```
movl %eax2, a[2]
```

a[2]:

## Vad är rätt?

1. `transfer(10, 1, 2)` först:
  - $a[1] = 15$
  - $a[2] = 25$
2. `transfer(20, 1, 2)` först:
  - $a[1] = 5$
  - $a[2] = 35$
3. "Samtidigt":
  - $a[1] = -5$
  - $a[2] = 25$

## Race condition

Typiska tecken på att ett *race condition* finns:

- Resultatet beror på i vilken ordning trådar råkar exekevera och läsa eller skriva till delad data
- Trådarna "tävlar" om vilket resultat det ska bli
- Systemet blir inte deterministiskt, ett program med *race conditions* kan ge olika resultat vid varje körning trots att indata är desamma

Notera: Race condition  $\Rightarrow$  fel

- 1 Varför synkronisera?
- 2 **Kritiska sektioner**
- 3 Bounded Buffer
- 4 Condition variables
- 5 Parallellism

## Kritisk sektion

En eller flera kodsektioner där ett *race condition* kan inträffa så att det blir "fel".

- En kodsektion där delade resurser (ofta variabler) används
- Kan därmed **inte** exekveras samtidigt som andra (eller samma) kodsektioner som använder samma variabler



## Delade resurser: Exempel

- Globala variabler
- Globala datastrukturer
- Variabler som delas via pekare
- Datastrukturer som delas via pekare
- Hårdvaruresurser (skärm, tangentbord, disk, nätverk, etc.)
  - en mening som skrivs till skärmen får inte avbrytas av en annan
  - ett paket som skall skickas över nätverket får inte blandas med andra paket

## Trådsäkra resurser: Exempel

Om en funktion (eller variabel) är trådsäker menar vi att den kan anropas (användas) samtidigt från flera trådar utan minsta risk för synkroniseringsfel eller race conditions.

- Variabler som enbart är åtkomliga från en viss tråd
- Variabler som är lokala för en funktion, inklusive kopierade parametrar
  - Se upp för referens- och pekarparametrar
- Variabler som enbart läses (t.ex. konstanter)

## Lösning

- Alla operationer inuti en kritisk ska se tillsammans, som en enhet
- ⇒ Ingen ska se delresultat av beräkningar inuti den kritiska sektionen
- ⇒ Allt inuti den kritiska sektionen ser utåt sett som en helhet, som en *atomär operation*
- ⇒ En kritisk sektion får bli avbruten, men bara av kod utanför den kritiska sektionen (dvs. kod som inte använder samma delade resurser)

## Lösning

- Alla operationer inuti en kritisk ska se tillsammans, som en enhet
- ⇒ Ingen ska se delresultat av beräkningar inuti den kritiska sektionen
- ⇒ Allt inuti den kritiska sektionen ser utåt sett som en helhet, som en *atomär operation*
- ⇒ En kritisk sektion får bli avbruten, men bara av kod utanför den kritiska sektionen (dvs. kod som inte använder samma delade resurser)

**Lösning:** Mutual exclusion - Släpp bara in en tråd åt gången i en kritisk sektion!

## Mutual exclusion (SV: Ömsesidigt uteslutande)

Släpp bara in en tråd åt gången i varje kritiska sektion.

### **Vi behöver följande:**

- Endst en tråd åt gången får komma åt den delade resursen (vårt ansvar som programmerare)
- Trådar som vill in i en kritisk sektion måste garanteras tillträde inom en begränsad tid
- Om ingen tråd exekeverar i den kritiska sektionen tillåts omedelbart tillträde
- En tråd får inte exekevera obegränsat länge i en kritisk sektion
- Ska fungera oavsett antal CPU och deras hastighet

# Lås

En mekanism som erbjuder mutual exclusion då den används korrekt

- Skyddar en delad resurs eller en *kritisk sektion*
- Alla relaterade kritiska kodsektioner måste använda samma lås
- Liknar en binär semafor initierad till 1 (räknar "exekeveringsplatser")

## Lås

Har följande operationer:

acquire (lås) : Försök gå in i en kritisk sektion, förbruka resursen (sema\_down)

release (lås upp) : Gå ut ur en kritisk sektion, ge tillbaka resursen, (sema\_up)

Till skillnad från en semafor har lås felhantering:

- Kan bara låsas upp av den tråd som låste
- Kan inte låsas igen från den tråd som låste låset

## Banköverföring (lösning 1)

```
struct account {
    int balance;
    char *owner;
};

struct account account[NUM_ACCOUNTS];
struct lock account_lock;

bool transfer(int amount, int from, int to);
```



## Banköverföring (lösning 1)

```
bool transfer(int amount, int from, int to) {
    lock_acquire(&account_lock);
    if (account[from].balance >= amount) {
        account[from].balance -= amount;
        account[ to ].balance += amount;
        lock_release(&account_lock);
        return true;
    } else {
        lock_release(&account_lock);
        printf("Från-kontot saknar täckning.\n");
        return false;
    }
}
```

# Lösning 1

- Ett lås för hela arrayen gör att två olika överföringar mellan fyra olika konton måste vänta på varandra även om de skulle kunna pågå samtidigt!
- Låset tas på ett ställe och släpps på två ställen på olika indenteringsnivå  $\Rightarrow$  Otydligt! Vi vill ha en `acquire` för varje `release`.

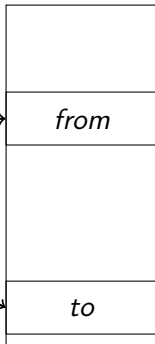
## Kritiska sektioner: Ytterligare analys

```
void transfer(...) {
```

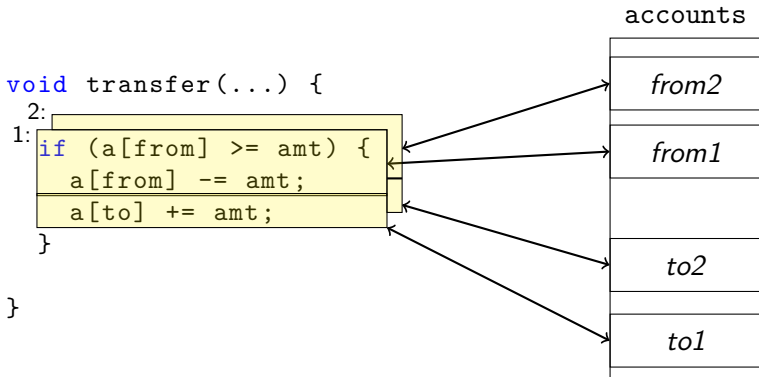
```
    if (a[from] >= amt) {  
        a[from] -= amt;  
        a[to] += amt;  
    }
```

```
}
```

accounts



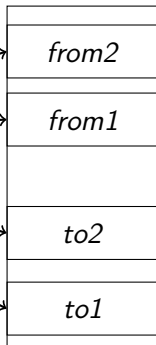
## Kritiska sektioner: Ytterligare analys



## Kritiska sektioner: Ytterligare analys

```
void transfer(...) {  
  2:   
  1:   
    if (a[from] >= amt) {  
      a[from] -= amt;  
      a[to] += amt;  
    }  
}
```

accounts



2 transaktioner, 4 kritiska sektioner!  
En för varje konto!

## Banköverföring (lösning 2)

```
struct account {
    int balance;
    char *owner;
    struct lock lock;
};

struct account account[NUM_ACCOUNTS];

bool transfer(int amount, int from, int to);
```

## Banköverföring (lösning 2)

```
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;
}
```

## Lösning 2

- Varje konto har ett eget lås. Överföringar kan oftast ske samtidigt
- Låsen tas och släpps på samma indenteringsnivå och varje acquire motsvaras av en release
- Fler lås  $\implies$  mer administration och högre minnesanvändning. Är den ökade parallellismen värt det?



## Lösning 2

- Varje konto har ett eget lås. Överföringar kan oftast ske samtidigt
- Låsen tas och släpps på samma indenteringsnivå och varje acquire motsvaras av en release
- Fler lås  $\implies$  mer administration och högre minnesanvändning. Är den ökade parallellismen värt det?
  - Inte alltid lätt att avgöra: **Mät** om du är osäker!
  - Om möjligt: Lägg låset tillsammans med den variabel som synkroniseras
  - I kursen: Fungerande lösning med högre parallellism ger alltid högre poäng. Motivation till varför det eventuellt inte är värt det ökar poäng ytterligare!

- 1 Varför synkronisera?
- 2 Kritiska sektioner
- 3 **Bounded Buffer**
- 4 Condition variables
- 5 Parallellism

## Bounded Buffer (igen)

```
typedef unsigned char byte;
const int SIZE = 256;

struct Buffer {
    int buffer[SIZE];
    byte rpos = 0, wpos = 0;
    struct semaphore free = SIZE;
    struct semaphore filled = 0;
};

int get(struct Buffer *b);
void put(struct Buffer *b, int data);
```

## Bounded Buffer (igen)

```
void get(struct Buffer *b) {
    sema_down(&b->filled);
    int r = b->buffer[b->rpos++];
    sema_up(&b->free);
    return r;
}

void put(struct Buffer *b, int data) {
    sema_down(&b->free);
    b->buffer[b->wpos++] = data;
    sema_up(&b->filled);
}
```

## Bounded Buffer (synkroniserad)

```
typedef unsigned char byte;
const int SIZE = 256;

struct Buffer {
    int buffer[SIZE];
    byte rpos = 0, wpos = 0;
    struct lock rlock, wlock;
    struct semaphore free = SIZE;
    struct semaphore filled = 0;
};

int get(struct Buffer *b);
void put(struct Buffer *b, int data);
```

## Bounded Buffer (synkroniserad)

```
void get(struct Buffer *b) {
    sema_down(&b->filled);
    lock_acquire(&b->rlock);
    int r = b->buffer[b->rpos++];
    lock_release(&b->rlock);
    sema_up(&b->free);
    return r;
}
```

## Bounded Buffer (synkroniserad)

```
void put(struct Buffer *b, int data) {
    sema_down(&b->free);
    lock_acquire(&b->wlock);
    b->buffer[b->wpos++] = data;
    lock_release(&b->wlock);
    sema_up(&b->filled);
}
```

- 1 Varför synkronisera?
- 2 Kritiska sektioner
- 3 Bounded Buffer
- 4 **Condition variables**
- 5 Parallellism



## Condition variable

Väntar på att ett villkor ska bli uppfyllt

Består av tre delar:

1. Villkoret (ex. `b->free != SIZE`)  
(inkluderar en eller flera *delade variabler*)
2. Lås som skyddar de delade variablerna  
(eller: den kritiska sektionen till vilken variablerna hör)
3. Condition variable för att vänta effektivt

⇒ Är i princip en **väntekö**

## Condition variables

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

## Effektivisera en dålig lösning med condition variables

1. Identifiera den kritiska sektionen och lås den
2. Identifiera busy wait och lägg till `cond_wait`
3. Identifiera tilldelningar som påverkar villkoret och lägg till `cond_signal` eller `cond_broadcast`

## 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;
```

```
}
```

Buffer

buffer

free

rpos

wpos

## Bounded Buffer, analys av osynkroniserad lösning

```
void get(struct Buffer *b) {
```

```
    while (b->free == SIZE)
        ; /* Vänta */
    ++b->free;
    return b->buffer[b->rpos++];
```

```
}
```

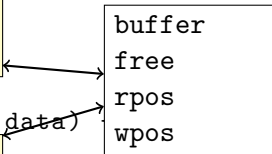
```
void put(struct Buffer *b, int data)
```

```
    while (b->free == 0)
        ; /* Vänta */
    --b->free;
    b->buffer[b->wpos++] = data;
```

```
}
```

Buffer

```
buffer
free
rpos
wpos
```



## 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);
}
```

- 1 Varför synkronisera?
- 2 Kritiska sektioner
- 3 Bounded Buffer
- 4 Condition variables
- 5 **Parallellism**

## Parallellism

- Målet med flera trådar är att åstadkomma parallellism, så att vi kan använda flera CPU.
- Målet med synkronisering är att åstadkomma sekventiell exekevering där samtidig exekevering skulle orsaka obestämda resultat.

Vad är problemet med resonemanget:

*Jag lägger alla looparna inom ett lås för att vara på den säkra sidan!*

## Vilken är bäst om vi har flera CPU?

Är den sekvensiella lösningen här bäst, eller någon av de två lösningarna på nästa sida om vi har 2 CPU?

```
void sum(int *array, struct lock *lock) {  
    for (int i = 0; i < 2_000_000; i++)  
        sum += array[i];  
}
```

## Vilken är bäst om vi har flera CPU?

```
void sum_a(int *array, struct lock *lock) {
    lock_acquire(lock);
    for (int i = 0; i < 1_000_000; i++)
        sum += array[i];
    lock_release(lock);
}

void sum_b(int *array, struct lock *lock) {
    for (int i = 0; i < 1_000_000; i++) {
        lock_acquire(&sum_lock);
        sum += array[i + 1_000_000];
        lock_release(&sum_lock);
    }
}
```



## Att fundera på

Din processlista behöver synkroniseras

- Ska du ha ett lås per index i din tabell?
- Ska du ha ett globalt lås för hela listan?
- Vilken variant ger bäst parallellism?
- Vilken variant ger minst overhead (exekeveringstid, minnesanvändning)?

## Nästa vecka

Hur implementeras synkroniseringsprimitiverna vi använder?

Filip Strömbäck, Klas Arvidsson

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