TDIU16 – Synkronisering

Ordning och reda

Filip Strömbäck, Klas Arvidsson

LINKÖPING UNIVERSITY

Planering

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

[1] lämpligt att demonstrera första passet

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

$\Rightarrow$ När delad 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]: 25

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]: 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]: -5

Tråd 2:

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

```
movl a[2], %eax2
addl $20, %eax2
movl %eax2, a[2]
```

a[2]: 15

Vad händer?

Tråd 1:

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

```
movl a[2], %eax1


addl $10, %eax1
movl %eax1, a[2]
```

a[1]: -5

Tråd 2:

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

```
movl a[2], %eax2
addl $20, %eax2
movl %eax2, a[2]
```

a[2]: 15→25

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 $\not\Longrightarrow$ fel

Formalia

- Flera trådar använder samma minne, minst en skriver
  $\Rightarrow$ *konflikt (conflict)*
- Om användning i *konflikt* sker samtidigt
  $\Rightarrow$ *data race*
- I många språk: *data race* = fel

Alltså:

- Vi **måste** *synkronisera* åtkomst till delad data

Mer läsning:
`https://queue.acm.org/detail.cfm?id=2088916`

Vad är delad data?

Data eller andra resurser som används av fler än en tråd eller process

Exempel att vara vaksam på:

- Globala variabler/datastrukturer
- Variabler/datastrukturer som delas via pekare
- Hårdvaruresurser (skärm, tangentbord, disk, nätverk, etc.)
  - en rad som skrivs ut till skärmen ska inte avbrytas av en annan
  - ett paket som skickas över nätverket får inte avbrytas av andra paket

Vad kan vi anta *inte* är delat?

En *trådsäker* funktion eller variabel kan användas samtidigt från flera trådar utan problem

Exempel på trådsäker data:

- Lokala variabler, inklusive kopierade parametrar
  - Lokala variabler kan delas, vilket kräver extra försiktighet
  - Se upp för referens- och pekarparametrar
- Variabler som endast är åtkomliga från en viss tråd
- Variabler som endast läses

Hur undviker vi *data races*?

Vi skapar *mutual exclusion* (SV: ömsesidigt uteslutande):

$\Rightarrow$ Vi ser till att maximalt en tråd kör kod som använder delad data samtidigt

Till vår hjälp har vi *synkroniseringsprimitiver*:

- Semaforer
- Lås
- ...

Lås

Mekanism för att skapa *mutual exclusion*

Ser till att maximalt en tråd *håller* låset, andra trådar får vänta.

Har följande operationer:

- `lock_init(<lock>)`: Initiera låset
- `lock_acquire(<lock>)`: Försök att ta låset, vänta om en annan tråd håller låset
- `lock_release(<lock>)`: Släpp låset, låt andra trådar ta låset

Lås – implementation

Lås kan implementeras med en semafor:

- `lock_init(x)` $\Rightarrow$ `sema_init(x, 1)`
- `lock_acquire(x)` $\Rightarrow$ `sema_down(x)`
- `lock_release(x)` $\Rightarrow$ `sema_up(x)`

Låset kan göra mer felhantering:

- Endast tråden som håller låset får anropa `release`
- En tråd som håller låset får inte anropa `acquire` igen

Undvika *data races* med lås (FEL)

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

    lock_acquire(&lock);
    account[from].balance -= amount;
    lock_release(&lock);

    lock_acquire(&lock);
    account[ to ].balance += amount;
    lock_release(&lock);
    return true;
  } else {
    lock release(&lock):
```

Kritisk sektion

En eller flera sekvenser av satser där ett *race condition* kan orsaka ett fel:

- Delade resurser (ofta delad data) används
- Koden uppdaterar delad data i flera steg
- Andra trådar måste observera operationen som antingen inte påbörjad, eller helt klar (den ska vara atomär)

Notera: Vi är intresserade av hur koden *exekeveras*, man måste alltså se förbi if-satser och loopar.

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);
```

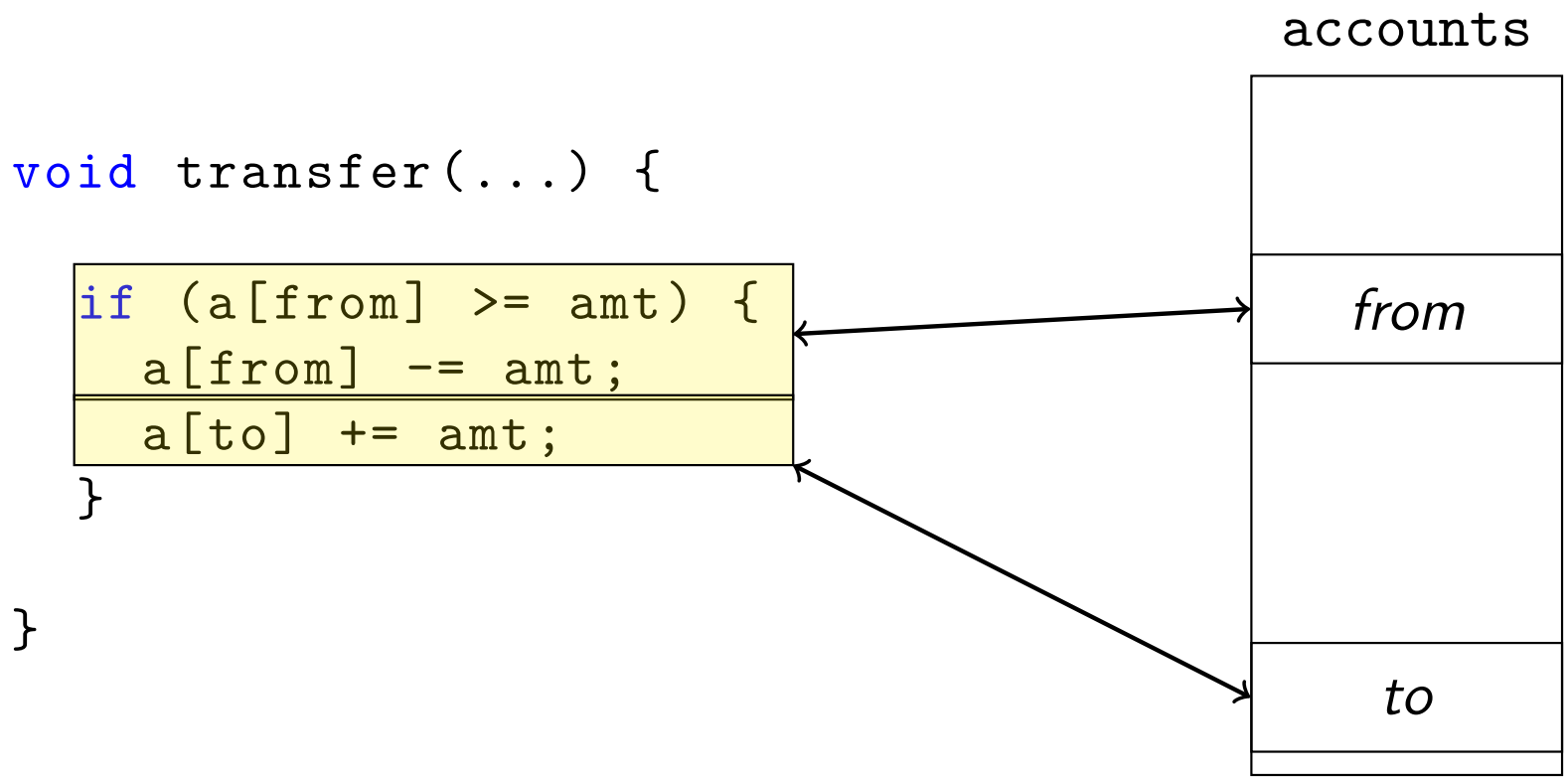
Banköverföring (lösning 1) (funkar, inte bra)

```
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;
  }
}
```

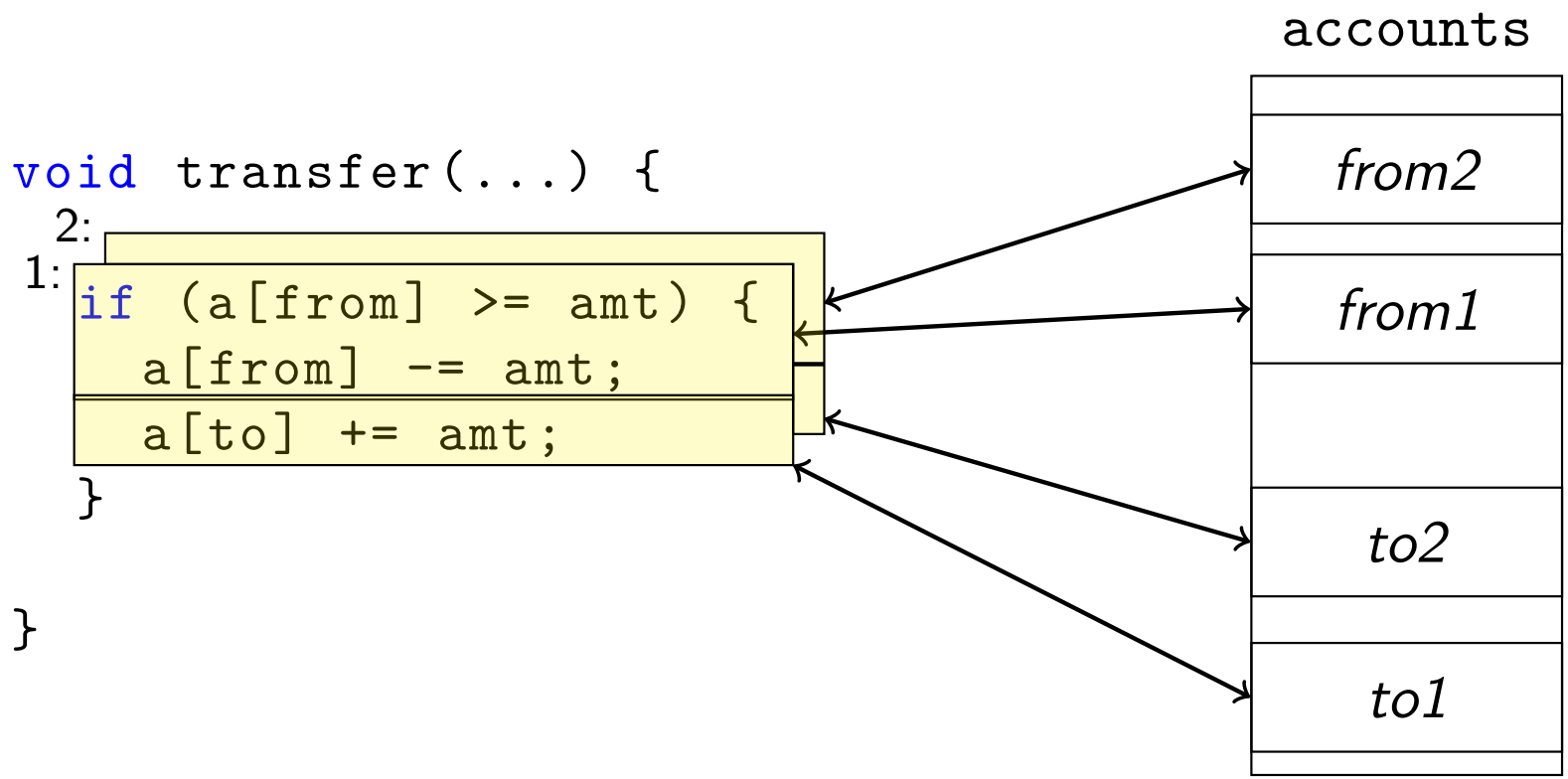
Lösning 1

- 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`.

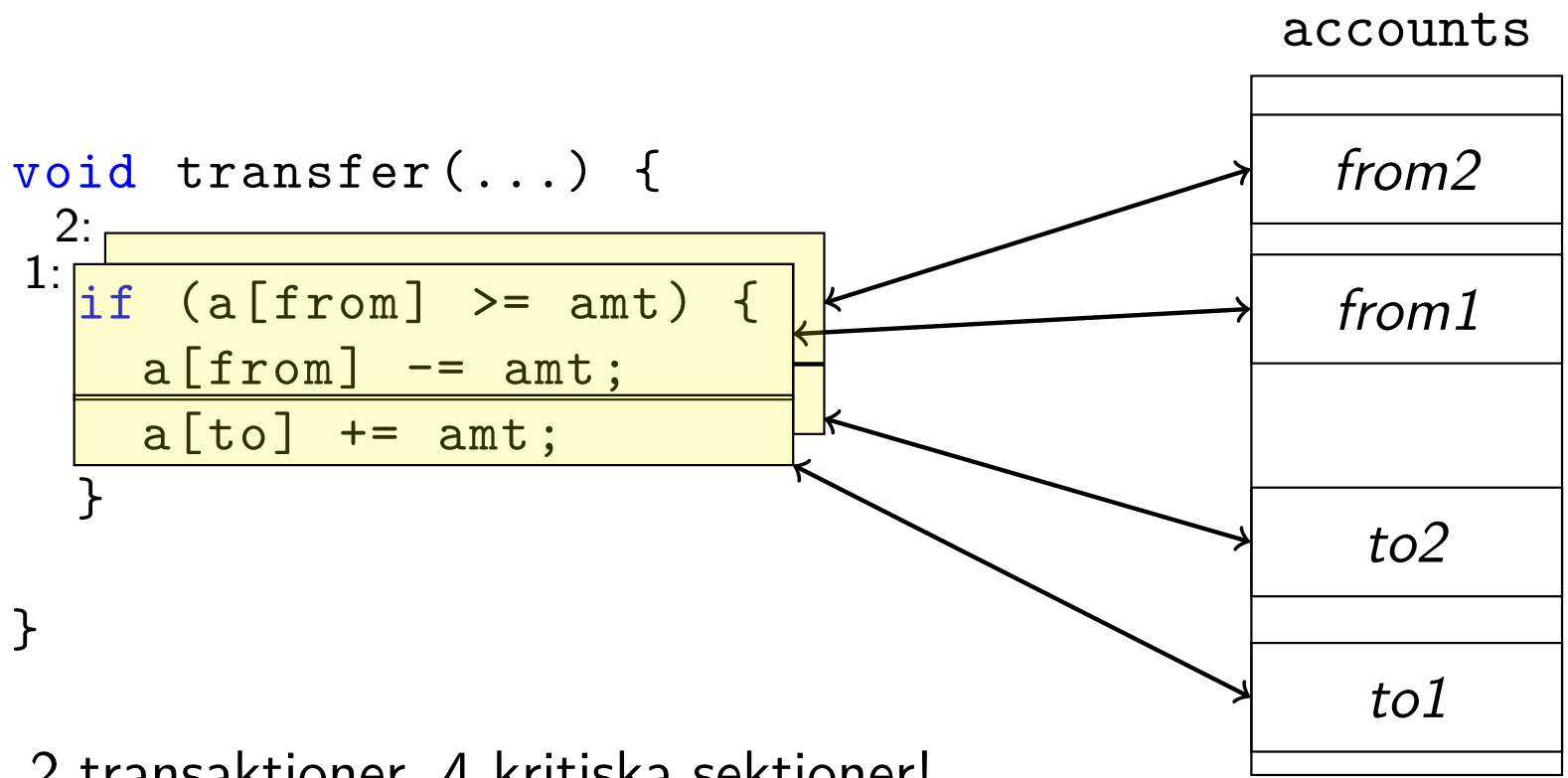
Kritiska sektioner: Ytterligare analys



Kritiska sektioner: Ytterligare analys



Kritiska sektioner: Ytterligare analys



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) (OK)

```
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?

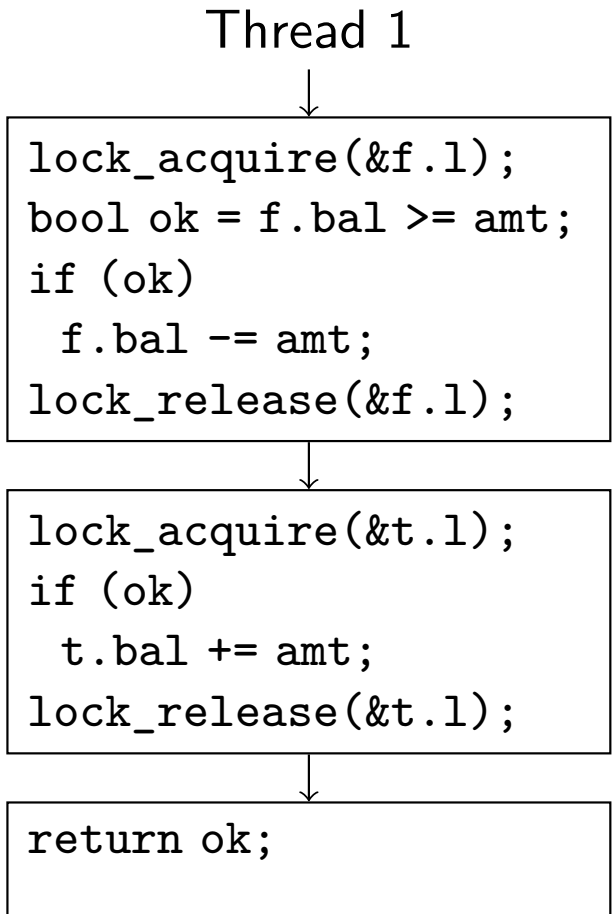
Lösning 2

- 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: Fungrerande lösning med högre parallellism ger alltid högre poäng. Motivation till varför det eventuellt inte är värt det ökar poäng ytterligare!

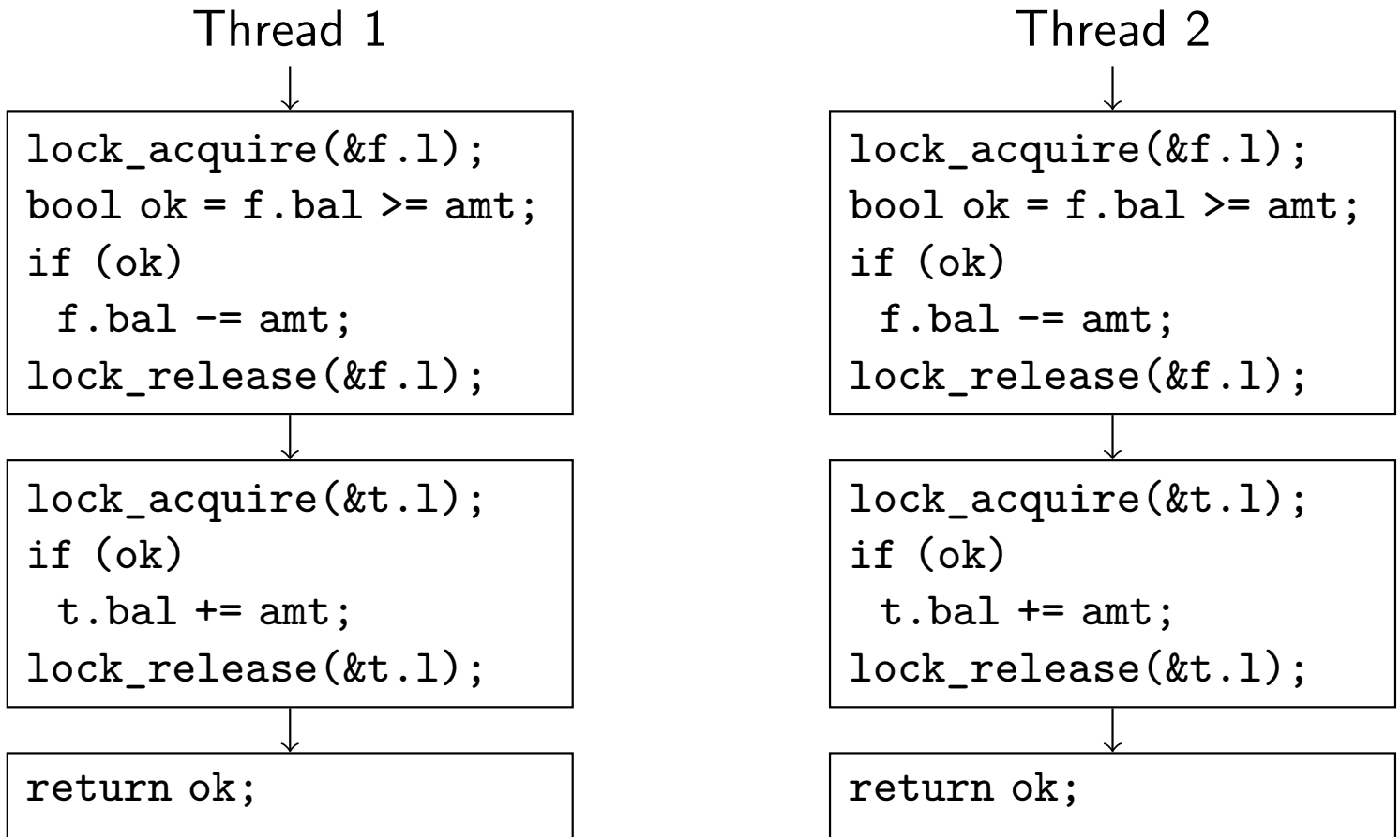
Formalia – Hur tänker kompilatorn?

Minnesmodellen i C och C++:

- Kod exekeveras som "block"
- Delas av synkronisering
- Väldefinierat så länge *data race* ej kan ske



Formalia – Hur tänker kompilatorn?



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);
}
```

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

$\Longrightarrow$ 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

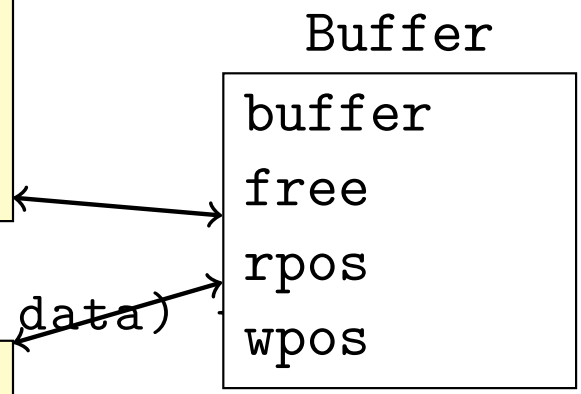
1. Identifiera den *kritiska sektionen* och lås den
2. Identifiera *busy wait* och lägg till `cond_wait`
3. Identifiera tilldelningar som *påverkar villkoret* och lägg till `cond_signal` eller `cond_broadcast`

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;
}
```

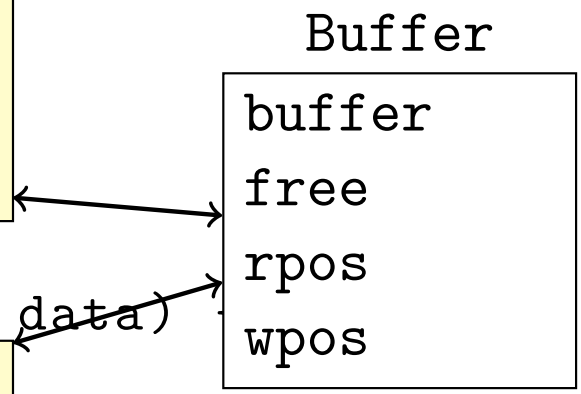
Bounded Buffer, analys av osynkroniserad lösning

```
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;
}
```



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);
}
```

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