Synkronisering

Ordning och reda

Banköverföring

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

struct account account[ NUM_ACCOUNTS ];

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

Banköverföring

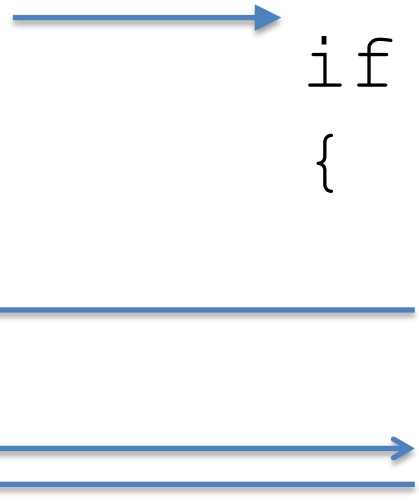
```
bool transfer(int amount, uint from, uint to)
{
  if (account[from].balance >= amount) {
    account[from].balance -= amount;
    account[to].balance += amount;
    return true;
  }
  else {
    printf(”Från-kontot saknar täckning.\n”);
    return false;
  }
}
```

Vad händer? När?

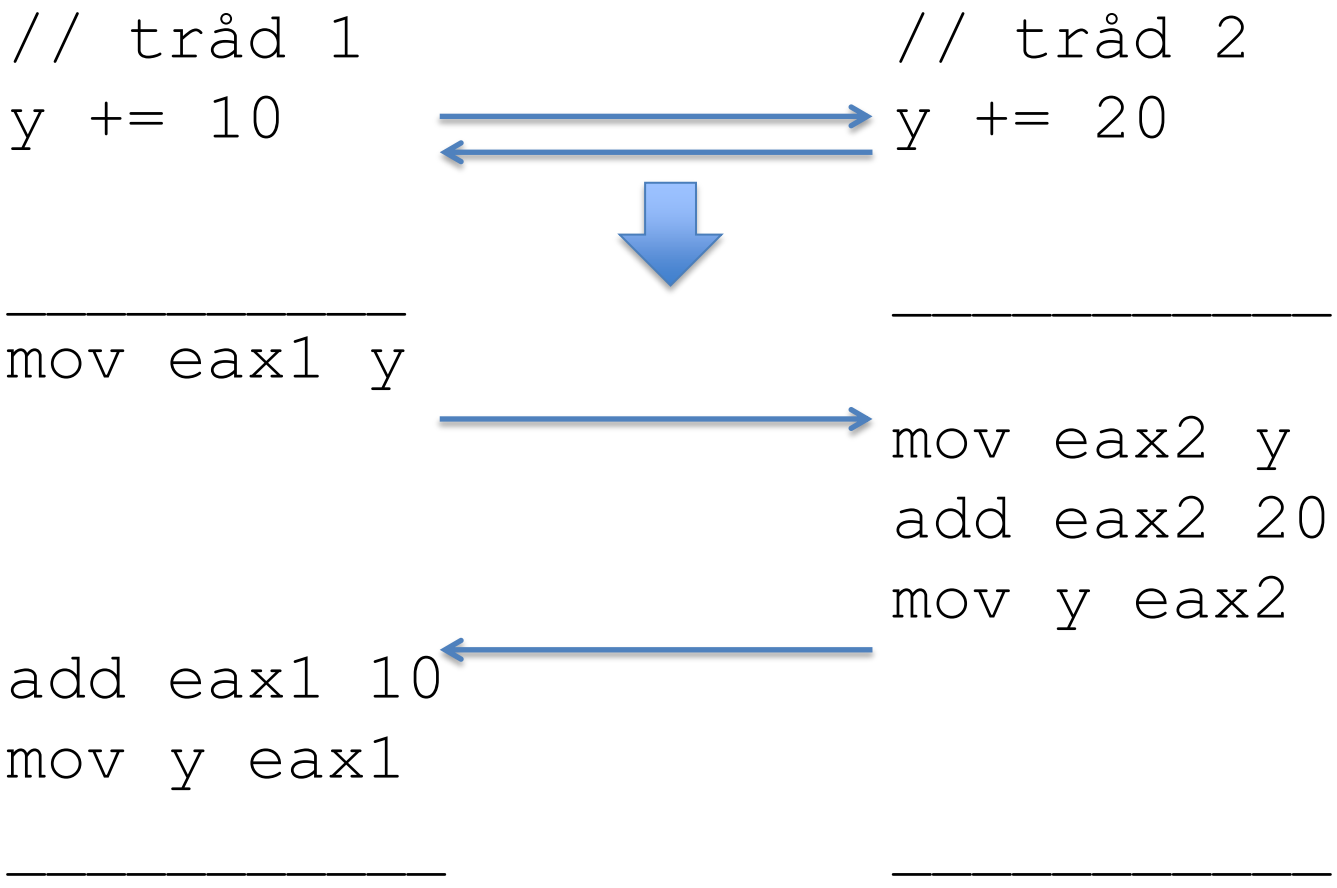
- I vilken ordning exekverar instruktionerna?
  - Sekventiellt inom en tråd (som vanligt)
  - Odefinierat mellan trådar
    - Trådbyte kan ske närsomhelst
    - Slumpen avgör
    - Undersök för exemplen ovan vad som händer!
  - Kanske exakt samtidigt om flera CPU finns
- Det finns vissa kritiska områden i koden där trådbyte *kommer att* orsaka fel
  - När gemensam data används

Vad händer? x == 25

```
// tråd 1                    // tråd 2
// konto x och y             // konto x och y
if (x >= 10)
{                                if (x >= 20)
                                 {
                                     x -= 20
    x -= 10
    y += 10                      y += 20
}                                }
```



Vad händer? y == 15



Vad händer?

- Kom ihåg att CPU-register sparas undan i tråden vid trådbyte, och återställs när tråden fortsätter.
- Varje tråd har sin egen uppsättning CPU-regiser. I exemplet är alltså eax olika för tråd 1 och för tråd 2. Detta är markerat med trådens nummer (eax1 vs eax2).
- x blir -15, borde blivit 5
- y blir 25, borde blivit 45

Vad är rätt? p = 12

```
// tråd 1
if ( p > 10 )
  p -= 10
```

```
// tråd 2
if ( p > 5 )
  p -= 5
```

Ibland är det svårt att avgöra vad som faktiskt är rätt resultat. Du måste alltid börja med att definiera vad som är rätt. Antag att p = 12 då båda trådarna startas. Vad tror ni programmeraren avsåg vara rätt resultat?

Tråd 1 först: 2
Tråd 2 först: 7
”Samtidigt”: -3 (?)

Exempel: Gemensam data

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

Används någon av ovan resurser i olika trådar som exekverar samtidigt behövs synkronisering för att undvika ”race conditions”, d.v.s. mutual exclusion behövs.

Kritisk sektion

- En kodsektion där en tråd använder delade resurser (ofta variabler) och inte kan exekveras samtidigt som andra kodsektioner som använder samma resurser
- Om relaterade kritiska sektioner (olika kodavsnitt som använder samma resurser) skulle råka exekvera samtidigt uppstår ett ”race condition”:
  - resultatet beror på i vilken ordning trådarna råkar exekvera och läsa eller skriva den delade resursen
  - trådarna ”tävlar” om vilket resultat det skall bli
  - systemet blir inte deterministiskt, ett program med race conditions kan ge olika resultat vid varje körning trots att all indata är densamma

Mutual exclusion

- Översättning: ”Ömsesidigt uteslutande”
- Kravet att när en tråd exekverar i en kritisk kodsektion får ingen annan tråd exekvera i en relaterad kritisk kodsektion
  - Alla operationer som sker i en kritisk sektion sker tillsammans, ”odelbart”. Ingen skall se delresultat av beräkningar som sker inuti den kritiska sektionen. Allt inom den kritiska sektionen sker utåt sett som en helhet, detta kallas en ”atomic operation”. En kritisk sektion får bli avbruten, *men bara av kod som inte på något sätt använder samma resurser*.
  - Att exekvera i en kritisk sektion måste därför utesluta att någon annan under tiden exekverar i en relaterad kritisk sektion.

Krav: Mutual exclusion

- Endast en tråd åt gången kan komma åt (läsa/skriva) den delade resursen. Måste genomdrivas av programmeraren.
- Trådar som väntar på något utanför en kritisk sektion påverkar inte andra trådar.
- Trådar som vill in i en kritisk sektion måste garanteras tillträde inom begränsad tid.
- Om ingen tråd exekverar inom en kritisk sektion tillåts omedelbart tillträde till den som behöver.
- Mutual exclusion och dessa krav upprätthålls oavsett antal CPU eller trådars/CPU:ers relativa hastighet.
- En tråd får inte exekvera obegränsat länge i en kritisk sektion.

Exempel: Trådsäkra data

- Med att en funktion (eller variabel) är trådsäker menar vi att den kan anropas (användas) samtidigt (concurrently) från flera trådar utan minsta risk för just synkroniseringsfel eller race conditions.
- Exempel på trådsäkra data:
  - Variabler som enbart är åtkomliga från en viss tråd.
  - Variabler som är lokala för en funktion, inklusive kopierade parametrar (varning för referens- och pekarparametrar).
  - Variabler som enbart läses, t.ex. konstanter.

Lås

- En mekanism som vid korrekt användning erbjuder mutual exclusion.
- Skyddar en delad resurs
- Som en binär semafor initierad till en resurs (en ledig ”exekveringsplats” finns bland de relaterade kritiska sektionerna från start)
- Alla relaterade kritiska sektioner måste använda samma lås
- Låsning (acquire): förbrukar resursen eller väntar
- Upplåsning (release): ger tillbaka resursen och signalerar
- Till skillnad från binär semafor har lås felhantering:
  - Kan endast låsas upp av den tråd som låste
  - Kan inte låsas igen medan en tråd har låst (”håller”) låset

Banköverföring (lösning 1)

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

struct lock lock_account_array;
struct account account[ NUM_ACCOUNTS ];

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

Banköverföring (lösning 1)

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

Lösning 1

- Globalt lås för hela arrayen gör att två olika överföringar mellan fyra helt olika konton måste vänta på varandra trots att de skulle kunna pågå samtidigt. Ineffektivt.
- Låset ”tas” på ett ställe och ”släpps” på två ställen i koden. Dessutom sker detta på olika indenteringsnivå. Tydlig kod har exakt en ”release” för varje ”acquire”, och på samma indenteringsnivå. Det gör det enklare att se att låsningen är korrekt.

Banköverföring (lösning 2)

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

struct account account[ NUM_ACCOUNTS ];

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

Banköverföring (lösning 2)

```
bool transfer(int amount, uint from, uint to) {
  lock_acquire(&account[from].lock_account);
    bool transfer = account[from].balance >= amount;
    if (transfer)
      account[from].balance -= amount;
  lock_release(&account[from].lock_account);
  lock_acquire(&account[to].lock_account);
    if (transfer)
      account[to].balance += amount;
  lock_release(&account[to].lock_account);
  if (!transfer)
    printf(”Från-kontot saknar täckning.\n”);
  return transfer;
}
```

Lösning 2

- Varje konto har ett eget lås. Överföringar kan i de flesta fall ske samtidigt utan att vänta på varandra. Bara när samma konto förekommer i två överföringar behöver de vänta.
- Fler lås innebär mer administration och minnesanvändning. Är den ökade parallellismen värt detta?
  - Inte alltid lätt att avgöra.
  - Om möjligt och om det bedöms ge fördel: Lägg låset tillsammans med variabeln som synkroniseras för bästa parallelism.
  - I kursen: Fungerande låsning med högre parallelism ger alltid högre poäng. Kommentar som motiverar varför det eventuellt inte är värt det ökar poängen ytterligare.
- Låsen tas och släpps på samma indenteringsnivå och varje låsning har exakt en upplåsning. Lätt att se att ingen låsning/upplåsning glömts bort.

Bounded Buffer

```
typedef unsigned char byte;
const byte SIZE = 256;
struct BB {
   int buffer[SIZE];
   byte rpos, wpos = 0;
   struct semaphore free = SIZE;
   struct semaphore filled = 0;
}
int  get(struct BB* b);
void put(struct BB* b, int data);
```

Bounded Buffer (get)

```
// several threads simultaneously
// use get on the buffer
int get(struct BB* b)
{
   sema_down(&b->filled);
   int r = b->buffer[b->rpos++];
   sema_up(&b->free);
   return r;
}
```

Bounded Buffer (put)

```
// several threads simultaneously
// use put on the buffer
void put(struct BB* b, int data)
{
   sema_down(&b->free);
   b->buffer[b->wpos++] = data;
   sema_up(&b->filled);
}
```

Bounded Buffer

```
typedef unsigned char byte;
const byte SIZE = 256;
struct BB {
  int buffer[SIZE];
  byte rpos, wpos = 0;
  struct semaphore free = SIZE;
  struct semaphore filled = 0;
  struct lock buffer_mutex;
}
int  get(struct BB* b);
void put(struct BB* b, int data);
```

Bounded Buffer (get)

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

Bounded Buffer (put)

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

Bounded buffer igen

- Nu med lås och conditions istället för att använda semafor vid väntan på att buffern skall bli ”inte full” eller ”inte tom”.
- Notera likheten med den ursprungliga lösningen som inte var korrekt synkroniserad.
- Condition behövs för att om vi håller låset medan vi väntar på att buffern skall bli ”icke full” eller ”icke tom” så kan inte räknaren för detta uppdateras. Låset är ju upptaget.

Bounded Buffer

```
typedef unsigned char byte;
const byte SIZE = 256;
struct BB {
  int buffer[SIZE];
  byte rpos = 0, wpos = 0; int free = 0;
  struct condition not_empty;
  struct condition not_full;
  struct lock buffer_mutex;
}
int  get(struct BB* b);
void put(struct BB* b, int data);
```

Bounded Buffer (get)

```
int get(struct BB* b)
{
  lock_acquire(&b->buffer_mutex);

  while (b->free == SIZE)
    // wait until buffer is not empty
    ;

  ++b->free;
  int r = b->buffer[b->rpos++];



  lock_release(&b->buffer_mutex);
  return r;
}
```

Bounded Buffer (put)

```
void put(struct BB* b, int data)
{
  lock_acquire(&b->buffer_mutex);

  while (b->free == 0)
    // wait until buffer is not full
    ;

  --b->free;
  b->buffer[b->wpos++] = data;



  lock_release(&b->buffer_mutex);
}
```

Condition

- Enligt kraven på mutual exclusion får en tråd inte stoppa och vänta inuti en kritisk sektion.
- Men om tråden behöver stoppa och vänta på att en gemensam resurs uppdateras?
  1. Släpp låset
  2. Använd t.ex. semafor för att vänta
  3. Ta tillbaka låset
  4. Kontrollera alltid om väntevillkoret (fortfarande) är uppfyllt
  5. Signalera varje gång gemensamma data uppdateras i andra trådar för att väcka den som väntar
- Ovan logik implementeras i en condition variabel
  - condition wait(): utför punkt 1-3
  - condition signal(): till att göra punkt 5
  - OBS! punkt 4 måste du som programmerare lösa själv när du använder conditions, det är du som bestämmer villkoret för att vänta

Bounded Buffer (get)

```
int get(struct BB* b)
{
  lock_acquire(&b->buffer_mutex);

  while (b->free == SIZE)
    // wait until buffer is not empty
    cond_wait(&b->not_empty, &b->buffer_mutex);

  ++b->free;
  int r = b->buffer[b->rpos++];
  // buffer can not be full now, inform others
  cond_signal(&b->not_full, &b->buffer_mutex);

  lock_release(&b->buffer_mutex);
  return r;
}
```

Bounded Buffer (put)

```
void put(struct BB* b, int data)
{
  lock_acquire(&b->buffer_mutex);

  while (b->free == 0)
    // wait until buffer is not full
    cond_wait(&b->not_full, &b->buffer_mutex);

  --b->free;
  b->buffer[b->wpos++] = data;
  // buffer cannot be empty now, inform others
  cond_signal(&b->not_empty, &b->buffer_mutex);

  lock_release(&b->buffer_mutex);
}
```

Parallellism

- Målet med att använda flera trådar är att åstadkomma parallellism. Målet med det är att göra delar av beräkningen samtidigt på flera CPU.
- Målet med synkronisering är att åstadkomma sekventiell exekvering där samtidig (concurrent) exekvering skulle orsaka obestämt (non-determinisic) resultat.
- Det finns de som för t.ex. summeringen av fält resonerar att “jag lägger alla looparna inom ett lås för att vara på säkra sidan”. Vad är problemet med ett sådant resonemang?

Summering av fält

```
function sum_low is
begin
   for i in 0..999_999_999
      sum += array[i]
end sum_low

function sum_high is
begin
   for i in 1_000_000_000..1_999_999_999
      sum += array[i]
end sum_high
```

Summering av fält

```
struct semaphore wait_for_two
int sum = 0; // delad variabel!

// ingen tråd färdig från start...
sema_init(&wait_for_two, 0)

run_in_thread(sum_low)
run_in_thread(sum_high)

for i in 1..2
  sema_down(&wait_for_two)  // räkna ned eller vänta

print(sum); // totalsumman?
```

Otrådad summa

- Vilken lösning är snabbast givet 16 st CPU?
  Är det den sekventiella nedan, eller den trådade på nästa sida?

```
function sequential_main is
begin
   for i in 1 .. array_size
      sum += array[i]
   end for
end sequential_main
```

Trådad summa (trådfunktion)

```
function sum_array(from, to) is
begin
   lock_acquire(sum_lock)
   for i in from .. to
      sum += array[i]
   end for
   lock_release(sum_lock)
end sum_array
```

Trådad summa (huvudprogram)

```
function threaded_main is
begin
   chunk = array_size / 16
   for i in 1 .. 16
     thread_create(sum_array,
                   chunk*i,
                   chunk*(i+1))
   end for
end threaded_main
```

Mönster

- I’ll do it for you
- Pass the baton

http://www.cs.rit.edu/~kar/papers/dpfs/patterns.pdf

- Studera detta synkroniseringsproblem. Försök lösa själv innan du lär dig lösningarna.

Mer exempel

- Dining philosophers
- Barber shop

Googla och studera dessa problem.

Att fundera på

Din processlista du har i laborationerna behöver synkroniseras.

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