Synkronisering - Semaforen

Om att vänta – men inte i onödan

Dörrvakten

- Har order uppifrån pga brandregler:
  - Släpp in max 40 personer
- Garanterar att det aldrig är fler insläppta än order angivit
- Kommer fler måste de vänta tills det finns plats (någon går eller order ändras)
- Måste hålla koll på både hur många som kommer och hur många som går

Semaforen

- Har order från programmeraren pga resurs-begränsning eller krävd händelseordning:
  - Släpp in max N trådar
- Garanterar att fler aldrig släpps förbi
- Ankommer fler trådar måste de vänta tills fler får släppas in (någon går eller order ändras)
- Anropas både vid insläpp (down) och utsläpp (up)

Ett posthämtningsbryderi

- Bonden Lena Bertilsson hämtar posten
  - Posten har dragit ned på servicen till gården, så det är 3 kilometer till brevlådan
  - Posten brukar komma runt kl 13
  - Lena kommer till lådan kl 13:25
  - Lådan visar sig vara tom.
  - Har brevbäraren passerat? Hur skall Lena göra för att slippa gå i onödan?

Summering av fält 1.1

Vi har en stor array fylld med ettor:

```
int array[2_000_000_000]={1,1,...}
```

Vi har variabler för att lagra summor och delsummor av arrayen. Dessa initieras med noll:

```
int sum, suml, sumh = 0
```

Vi har två funktioner som summerar var sin del av arrayen.

Summering av fält 1.2

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

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

Summering av fält 1.3

Vi har ett huvudprogram som startar två trådar för att sköta summeringen (varför inte bara göra *en* loop?)

```
run_in_thread(sum_low)
run_in_thread(sum_high)
sum = suml + sumh
```

Vad blir resultatet i sum?

Summering av fält 2.1 - 2.2

```
bool low_not_done, high_not_done = true

function sum_low is
begin
  for i in 0..999_999_999
    suml += array[i]
  low_not_done = false
end sum_low

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

Summering av fält 2.3 (INTE GK)

```
run_in_thread(sum_low)
run_in_thread(sum_high)

while (low_not_done)
  noop
while (high_not_done)
  noop
sum = suml + sumh
```

- Hur mycket CPU-tid behövs?
  - Busy wait!
  - EJ godkänd lösning!

Bounded Buffer

- Vi har t.ex. en applikation som skall skyffla datapaket mellan två (eller fler) nätverkskort.
  - En tråd tar emot data och lägger till i en kö.
  - En tråd läser data från kön och skickar vidare.

- Kön (buffer) har begränsad (bounded) storlek.
  - Vad gör vi om kön är full när vi skall lägga till?
  - Vad gör vi om kön är tom när vi skall läsa?

Bounded Buffer 1.1

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

Bounded Buffer 1.2 (get)

```
int get(struct BB* b)
{
  while (b->free == SIZE)
    ;/* busy wait INTE OK */
  ++b->free; // signal
  return b->buffer[b->rpos++];
}
```

OBS! Ordningen på två sista raderna, finns någon ordning som alltid fungerar?

Bounded Buffer 1.3 (put)

```
void put(struct BB* b, int data)
{
  while (b->free == 0)
    ;/* busy wait INTE OK */
  --b->free; // signal
  b->buffer[b->wpos++] = data;
}
```

OBS! Ordningen på två sista raderna, finns någon ordning som alltid fungerar?

Ett järnvägsproblem

- SJ X2000 möter Green Cargo
  - En järnvägsknut med två spår österut, ett västerut
  - X2000 kommer västerifrån och fortsätter mot sydost. Tidtabellen anger att passagen sker 16:10.
  - Green Cargos godståg kommer från nordost och fortsätter västerut. Klockan är 16:15 när det når järnvägsknuten.
  - Du är lokförare på godståget. Vad gör du när du kommer till järnvägsknuten? Kör? Väntar? Hur länge?

Binär semafor

- Har två lägen (d.v.s. den är binär)
- Talar om ifall en händelse inträffat eller inte
  - X2000 har passerat
  - Brevbäraren har passerat
- Talar om ifall en resurs är tillgänglig eller inte
  - Spåret västerut är fritt
  - Posten finns i lådan
- Kan starta på vilket som av de båda lägena

Binär semafor

- Avgör om man skall vänta...
  - Resursen inte tillgänglig ännu
  - Händelsen inte inträffat ännu
- eller fortsätta direkt
  - Händelsen har inträffat redan
  - Resursen finns tillgänglig
- Avgör när man kan sluta vänta
  - Semaforen ändrar läge då det vi väntar på inträffar

Bron mellan bergen

- Mellan två bergstoppar går en gångbro
  - Bron är byggd av spända rep och brädor.
  - Bron är inte särskilt stadig och mycket lång.
  - Bron är utsatt för väder och vind.
  - Bron är på en sådan höjd att den ofta helt eller delvis är inbäddad i moln (dimma). I detta exempel är sikten lika med noll.
  - Bron kan bära max 5 personer åt gången.
  - Går du ut på bron? Väntar du? Hur länge?

Räknande semafor

- Räknar *antalet* TILLGÄNGLIGA resurser
- Avgör när en brukare av resursen måste vänta
  - När antalet är NOLL
- Avgör när brukaren kan sluta vänta
  - När antalet blir ett eller mer
- Kan användas som binär semafor
  - Låt räknaren vara endast 1 eller 0 (två lägen)

Räknande semafor

- Vi väntar bara då vi försöker räkna ned en semafor som redan är 0, d.v.s. använda en resurs som inte är tillgänglig
- En räknande semafor kan alltså INTE användas för att räkna upptagna resurser:
  - Skulle räkna upp när en resurs förbrukas
    - D.v.s. sluta vänta när en resurs förbrukas????
  - Skulle räkna ned när en resurs blir ledig
    - D.v.s. eventuellt (om räknaren 0) vänta på något som precis blev ledigt????

Semaforer i Pintos

```
#include ”threads/synch.h”

struct semaphore sema;

// initiera räknaren till N
sema_init(&sema, N);

// räkna ned eller vänta
sema_down(&sema);

// räkna upp och väck upp
sema_up(&sema);
```

Semaforer – andra funktionsnamn

- För att räkna ned eller vänta
  - `P()  // Proberen, ursprungligt`
  - `Wait()  // Mer beskrivande...`
  - `Down()  // Pintos, ännu bättre (?)`
- För att räkna upp och signalera
  - `V()  // Verhogen, ursprungligt`
  - `Signal()// Mer beskrivande`
  - `Up()  // Pintos, ännu bättre (?)`

Semaforen väntar själv

- Semaforen väntar automatiskt på effektivaste sätt när det behövs:

```
// vet ej om resursen är tillgänglig
// eller händelsen inträffat
sema_down(...) // väntar om det behövs
// har väntat klart och vet säkert att
// resursen finns tillgänglig eller att
// händelsen inträffat (sema_up måste ge
// en signal för varje sema_down)
```

Summering av fält 3.2

```
extern struct semaphore low, high

function sum_low is
begin
  for i in 0..999_999_999
    suml += array[i]
  sema_up(&low) // signal done
end sum_low

function sum_high is
begin
  for i in 1_000_000_000..1_999_999_999
    sumh += array[i]
  sema_up(&high) // signal done
end sum_high
```

Summering av fält 3.1, 3.3 (OK)

```
struct semaphore low, high

sema_init(&low, 0)  // ingen summa finns från start...
sema_init(&high, 0) // ...så initiera med 0 resurser

run_in_thread(sum_low)
run_in_thread(sum_high)

sema_down(&low)  // räkna ned eller vänta
sema_down(&high) // räkna ned eller vänta

sum = suml + sumh
```

- Hur mycket CPU-tid behövs nu?
  - Hur är semaforen implementerad? Senare Fö!
  - Vi antar en effektiv implementation där väntande tråd läggs på väntekö!

Summering av fält 4.x (OK)

```
struct semaphore wait_for_two

// ingen summa finns 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

sum = suml + sumh
```

```
// För att fungera med ovan kod ändras sema_up i
// vardera tråd till följande anrop:
   sema_up(&wait_for_two)
```

Summering av fält 5.x (FEL)

```
struct semaphore wait_for_two

// FEL: Semaforer ska initieras med antalet
// FEL: just nu tillgängliga resurser (0).
sema_init(&wait_for_two, 2)

run_in_thread(sum_low)
run_in_thread(sum_high)

// FEL: Semaforen väntar endast när den inte har någon
// FEL: resurs att ”förbruka”. Felet här är alltså att
// FEL: den aldrig kommer vänta eftersom 2 resurser
// FEL: finns från start och endast en förbrukas här.
sema_down(&wait_for_two)

sum = suml + sumh // summeringstrådarna är ej färdiga?!
```

Bounded Buffer 2.1

```
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 2.2 (get)

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

Bounded Buffer 2.3 (put)

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

Bounded Buffer med fler trådar

- Fortfarande inte OK om mer än en tråd läser eller mer än en tråd skriver. Varför?
  - Flera olika trådar som kör samtidigt
  - Gemensam data som används samtidigt
  - Nästa föreläsning
- Vad händer när olika trådar samtidigt modifierar gemensam data i en sekvens om flera maskininstruktioner?
  - Fundera på i++
  - Fundera igen på ordningen av två sista raderna i ”Bounded Buffer 1.x (put/get)”
  - Nästa föreläsning

Summering av fält igen (nästa Fö)

```
function sum_low is
begin
   for i in 0..999_999
      sum += array[i]
end sum_low

function sum_high is
begin
   for i in 1_000_000..1_999_999
      sum += array[i]
end sum_high
```