Obligatorisk oppgave 6: Synkronisering

Start oppgave

Forfall Fredag av 23.59 **Poeng** 0 **Må leveres** en filopplasting **Filtyper** c og pdf

I denne obligatoriske oppgaven skal vi fortsette med C-programmering med *pthreads* i Linux. Vi skal se på hvordan tråder kan samarbeide om å løse en oppgave. Dette krever at trådene *synkroniseres* — de må kunne vente på hverandre, kommunisere og dele ressurser. Spesielt må vi unngå at trådene "ødelegger" for hverandre når de leser og skriver i de samme delene av minnet

Synkronisering av tråder er kanskje noe av det vanskeligste å mestre innenfor programmering. Vi skal holde oss til relativt enkle eksempler, og bare se på noen grunnleggende mekanismer for synkronisering.

Oppgaveteksten her består av to deler:

1. I den **første delen** gis det en innføring i synkronisering av tråder i *pthreads*. Her vil du lære det som trengs for å løse oppgavene i del 2. Det er også noen enkle øvinger i del 1, som det **ikke** skal leveres noen besvarelse på.
2. Den **andre delen** inneholder programmeringsoppgaver. Det skal leveres besvarelser på oppgavene i form av et PDF-dokument med tekst, og filer som inneholder C-kode.

1 Synkronisering i *pthreads*

I den obligatoriske oppgaven om trådprogrammering har du lært hvordan vi oppretter og avslutter en tråd i *pthreads*, ved å bruke *pthread_create()* og *pthread_exit()*. Vi har også sett hvordan en tråd/prosess kan vente på at en annen tråd blir *ferdig*, med *pthread_join()*. For å lage mer generelle programmer som utnytter tråder bedre, trenger vi også at trådene både kan "snakke" sammen og stoppe/starte hverandre mens de fortsatt eksekverer.

I flere av eksemplene vi har sett tidligere, bruker vi tråder som bare starter og kjører ferdig en funksjon. Det vil da ofte være bedre (og enklere) å bare kalle funksjonen på vanlig måte, i stedet for å lage en ny tråd for å kjøre den.

Eksemplet med matrisemultiplikasjon, der hver tråd fikk "ansvaret" for å beregne "sin del" av matriseproduktet, er et bedre eksempel på utnyttelse av tråder. Hvis maskinen har flere CPU'er som kjører hver sin tråd parallelt, vil kjøretiden her kunne reduseres betydelig for store matriser. Det er allikevel et relativt enkelt eksempel, siden trådene arbeider helt uavhengig av hverandre i adskilte deler av minnet, og ikke trenger å kommunisere eller synkroniseres.

Når vi bruker tråder til å håndtere større og mer kompliserte problemer, må vi først finne en oppdeling av problemet i mindre delproblemer som hver tråd kan løse. Resultatene fra hver tråd settes da til slutt sammen til en fullstendig løsning. For mer generelle problemer enn f.eks. matrisemultiplikasjon vil det oftest være slik at:

- Det er avhengighet mellom de ulike delproblemene som hver tråd løser. Det betyr at trådene må kunne kommunisere med hverandre underveis, mens de eksekverer.
- Trådene vil dele data, dvs. at de leser og skriver til de samme delene av minnet. Dette kan føre til race conditions som gir feil/uønsket oppførsel av programmet. Vi må da synkronisere trådene slik at de bruker det delte minnet i en sikker og korrekt rekkefølge.
- Trådene må noen ganger stoppe for å vente på å få et delresultat fra en annen tråd. De kan da starte videre eksekvering først når delresultatet er klart.

I resten av denne oppgaven skal vi se eksempler på hvordan vi kan få til trådsynkronisering i *pthreads* ved å bruke de to mekanismene mutexer og betingede variable. For en bedre og mer detaljert beskrivelse av ulike problemer og løsninger knyttet til synkronisering, se læreboken og læringsmodulen om interprosesskommunikasjon i Canvas.

1.1 Race conditions

En såkalt "race condition" oppstår når:

- To eller flere tråder skal lese eller skrive til/fra samme delte sted i minnet eller samme fil, og:
- Det endelige resultatet, når trådene er ferdige, avhenger av i hvilken rekkefølge de kjøres.

Race conditions bør unngås fordi de kan gjøre programvaren uforutsigbar og gi feil resultater. Debugging kan også være svært vanskelig hvis feilene skyldes at trådene deler minne.

Et svært enkelt eksempel på et C-program med en race condition er gitt nedenfor:

```
#include <pthread.h>
#include <stdio.h>

#define N_THREADS 10

int tall = 0, n;

void *skriv_sum()
{
   int i;
   for (i = 1; i <= n; i++)
     tall++;
   printf("%d ", tall);
}


int main()
{
   pthread_t threads[NUMBER_OF_THREADS];
   int i;
```

```
    printf("n? ");
    scanf("%d", &n);

    for (i = 0; i < N_THREADS; i++)
      pthread_create(&threads[i], NULL, skriv_sum, NULL);

  for (i = 0; i < N_THREADS; i++)
    pthread_join(threads[i], NULL);

    printf("\n");
}
```

Dette programmet oppretter ti tråder som kjøres parallelt. Hver tråd starter funksjonen *skriv_sum()*, som kjører en løkke der en global delt variabel *tall* "telles opp" fra 1 til *n*. Verdien på *n*, som også er en delt variabel, leses fra bruker.

Merk at alle de ti trådene vil prøve å oppdatere *tall* "samtidig" — vi får en race condition på denne variabelen! Dvs. at trådene her kan ødelegge for hverandre, slik at ingen av dem produserer "riktig" resultat.

Kompiler og kjør dette programmet selv, først for små verdier av *n* (f.eks. *n*=100). Du vil da se at alle summene blir "riktige". Det er fordi hver tråd får kjøre helt ferdig, før den har brukt opp sin "timeslice" på noen tusendels sekunder i CPU og blir byttet ut med neste tråd som skal kjøre (vi skal forklare timeslices og timesharing nøyere senere i kurset).

Kjør deretter programmet med f.eks. verdien 1000000 (en million) for *n*. Du vil da se at verdiene som skrives ut ser helt tilfeldige ut, og vil variere mellom hver kjøring. Dette skjer fordi trådene nå må bytte på å bruke CPU'en(e) før de er ferdige. Hver gang en tråd skriver en verdi til *tall* ødelegger den for de andre trådene som nå ikke lenger har riktig resultat. Vi har en race condition som gjør programmet både uforutsigbart og feil.

1.2 Kritiske områder

Et såkalt kritisk område i koden for et trådprogram, er den delen av koden hvor en race condition oppstår. Som oftest er dette kode som kan gjøre at flere tråder prøver å lese eller skrive samme del av minnet samtidig.

I C-eksemplet ovenfor ligger det kritiske området i koden inne i funksjonen *skriv_sum()*, der trådene oppdaterer og skriver ut verdien av den delte variabelen *tall*. Det er kun her at trådene jobber mot samme minne, i resten av programmet er det ingen race conditions og trådene kan trygt kjøre parallelt.

Noe av "kunsten" med å programmere med tråder er å kunne identifisere de delene av koden som er kritiske områder, og prøve å gjøre disse små. Deretter må vi håndtere dem riktig for å unngå at det oppstår race conditions.

1.3 Gjensidig utelukkelse

For å unngå race conditions i trådprogrammering kan man bruke såkalt "gjensidig utelukkelse" ("mutual exclusion"):

- Kun én tråd om gangen får lov til å kjøre koden som ligger i et kritisk område.
- Hvis en tråd *A* kjører i kritisk område, må alle andre tråder som ønsker å kjøre samme kode vente til *A* er ferdig med å eksekvere den kritiske koden — trådene utelukker hverandre fra kritisk område.

Hvis de kritiske områdene i et program er store og krever mye kjøretid, vil effektiviseringen vi får av å bruke flere tråder være dårlig. Dette fordi den kritiske koden kjøres sekvensielt (en tråd om gangen) når vi bruker gjensidig utelukkelse. Trådene vil gå mye på "tomgang" (være "idle") mens de venter på å få gå inn i det kritiske området.

Det finnes ulike verktøy i forskjellige OS og programmeringsspråk for å unngå race conditions. Vi skal i denne oppgaven bare se på et par mekanismer i *pthreads* som kan brukes til gjensidig utelukkelse. Se læreboken og læringsmodulene i Canvas for (mye) mer om hvordan man kan få til "mutual exclusion".

1.4 Bruk av *mutex* i *pthreads*

Trådbiblioteket *pthreads* tilbyr en egen type "låsvariabel", en såkalt mutex, for å få til "**mut**ual **ex**clusion". En mutex kan brukes til å blokkere/stoppe tråder som er på vei inn i et kritisk område, og vekke/starte dem igjen når kritisk område ikke lenger er opptatt av andre tråder. Mutexer er implementert i OS på en slik måte at det ikke kan bli en race condition på selve mutex-variabelen.

En mutex er en C-variabel av datatypen *pthread_mutex_t*. Her deklarerer/oppretter vi en mutex som får variabelnavnet *lock*:

```
pthread_mutex_t lock;
```

Vi bør alltid initialisere mutexer før bruk. Det er en egen metode for dette i *pthreads* som heter *pthread_mutex_init()*. Vi kommer til å bruke funksjonen på denne måten for f.eks. å initialisere mutexen *lock*:

```
pthread_mutex_init(&lock, NULL);
```

Tråder som skal samarbeide om å unngå race conditions i et kritisk område, vil dele på en og samme mutexvariabel. Vi bruker vanligvis en mutex på denne måten:

- Alle trådene kaller funksjonen *pthread_mutex_lock()* med samme låsvariabel som parameter, rett før de går inn i det kritiske området:

  ```
  pthread_mutex_lock(&lock);
  ```

- Den første tråden som kaller *pthread_mutex_lock()* vil få begynne å eksekvere koden i kritisk område.

- Alle andre tråder som kaller *pthread_mutex_lock()* med den samme mutexen som parameter, vil da bli blokkert/stoppet — mutexen fungerer som en lås som hindrer trådene i å gå videre.
- Når en tråd er ferdig med å eksekvere koden i sitt kritiske område, kan den kalle funksjonen *pthread_mutex_unlock()* med samme mutex-variabel som parameter:

  ```
  pthread_mutex_unlock(&lock);
  ```

  En av trådene som er blokkert på denne mutexen, fordi den tidligere har kalt *pthread_mutex_lock()*, vil da starte opp igjen. Systemet velger selv hvilken tråd som starter hvis det er flere som er blokkert på samme mutex.
- Når vi er ferdig med å bruke en mutex, kan vi fjerne den igjen ved å bruke:

  ```
  pthread_mutex_destroy(&lock);
  ```

I avsnitt 1.1. ovenfor så vi et enkelt eksempel på en race condition som ga uforutsigbar og feil oppførsel av programmet. Nedenfor er det samme eksemplet, men her brukes det en mutex til å sørge for at bare én tråd av gangen får kjøre den kritiske koden:

```
#include <pthread.h>
#include <stdio.h>

#define N_THREADS 10

int tall = 0, n;

pthread_mutex_t lock;

void *skriv_sum()
{
   int i;

   pthread_mutex_lock(&lock);

   for (i = 0; i < n; i++)
     tall++;
   printf("%d ", tall);

   pthread_mutex_unlock(&lock);

}

int main()
{
   pthread_t threads[N_THREADS];
   int i;

   printf("n? ");
   scanf("%d", &n);

   pthread_mutex_init(&lock, NULL);

   for (i = 0; i < N_THREADS; i++)
     pthread_create(&threads[i], NULL, skriv_sum, NULL);

  for (i = 0; i < N_THREADS; i++)
    pthread_join(threads[i], NULL);
```

```
  pthread_mutex_destroy(&lock);

  printf("\n");
}
```

I koden ovenfor ser vi at mutex-variabelen *lock* deklarereres globalt, slik at den er tilgjengelig for alle trådene. Hovedprogrammet *main()* kaller *pthread_mutex_init()* for å initialisere mutexen, og *pthread_mutex_destroy()* etter at trådene er ferdige.

Hver tråd vil kalle *pthread_mutex_lock()* i funksjonen *skriv_sum()*, før eksekveringen av kritisk kode starter. Når trådene forlater det kritiske området, kaller de *pthread_mutex_unlock()*.

Kjør dette programmet selv og se hvordan det virker. Du vil da se at trådene ikke lenger "ødelegger" hverandres resultater. I Canvas-modulen for interprosesskommunikasjon vil du finne flere eksempler på bruk av mutex for å løse mer kompliserte problemer.

1.5 Betingede variabler

I forrige avsnitt brukte vi en mutex for å sikre at bare én tråd om gangen får kjøre i et kritisk område, slik at vi unngår race conditions. For å få tråder til å samarbeide om å løse problemer trenger vi mer generelle mekanismer for synkronisering. En slik mekanisme i *pthreads* er betingede variabler, som kan brukes for å sikre at en tråd venter med å kjøre videre inntil en bestemt betingelse er oppfylt.

En betinget variabel ("conditional variable") tilhører datatypen *pthread_cond_t* som er definert i *pthreads.h*. Opprettelse av en betinget variabel med navn *cond_var* gjøres slik i C:

```
pthread_cond_t cond_var;
```

Betingede variabler skal initialiseres før de brukes, med et kall på *pthread_cond_init()*:

```
pthread_cond_init(&cond_var, NULL);
```

Når vi er ferdig med å bruke en betinget variabel, kan vi fjerne den igjen ved å bruke:

```
pthread_cond_destroy(&cond_var);
```

En tråd kan "blokkere på en betinget variabel" (stoppe eksekvering) ved selv å kalle funksjonen *pthread_cond_wait()*. Tråden kan vekkes opp/startes igjen ved at en annen tråd kaller funksjonen *pthread_cond_signal()*.

Her er et lite eksempel som bruker to tråder, en mutex og en betinget variabel:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_cond_t cond;
pthread_mutex_t lock;
```

```
void *func_1(void *t_id)
{
   pthread_mutex_lock(&lock);
   printf("Tråd %ld: Venter på betinget variabel\n", (long) t_id);

   pthread_cond_wait(&cond, &lock);

   printf("Tråd %ld: Ferdig\n", (long) t_id);
   pthread_mutex_unlock(&lock);
}

void *func_2(void *t_id)
{
   pthread_mutex_lock(&lock);
   printf("Tråd %ld: Sender signal til betinget variabel\n", (long) t_id);

   pthread_cond_signal(&cond);

   printf("Tråd %ld: Ferdig\n", (long) t_id);
   pthread_mutex_unlock(&lock);
}

int main()
{
   pthread_t t_1, t_2;
   long i;

   pthread_mutex_init(&lock, NULL);
   pthread_cond_init(&cond, NULL);

   i = 1;
   pthread_create(&t_1, NULL, func_1, (void *) i);

   usleep(100000);

   i = 2;
   pthread_create(&t_2, NULL, func_2, (void *) i);

   pthread_join(t_1, NULL);
   pthread_join(t_2, NULL);

   pthread_mutex_destroy(&lock);
   pthread_cond_destroy(&cond);
}
```

Programmet ovenfor fungerer på følgende måte:

- Den betingede variabelen *cond* og mutexen *lock* deklareres globalt, først i programmet, slik at de er tilgjengelige for alle funksjonene. Merk at en betinget variabel alltid skal brukes sammen med en mutex for å sikre korrekt håndtering av variabelen.
- I hovedprogrammet *main()* initialiseres først både mutexen og den betingede variabelen, med kall på *pthread_mutex_init()* og *pthread_cond_init()*.
- Deretter opprettes første tråd, som starter med å eksekvere funksjonen *func_1()*. Hovedprogrammet vil deretter "sove" i et tidels sekund (se *man 3 usleep*), for å gi første tråd tid til å blokkere på den betingede variabelen før neste tråd starter.
- Første tråd, som kjører *func_1()*, låser på mutex, skriver ut en melding om at den skal vente, og kaller deretter *pthread_cond_wait()*. Tråden vil da være blokkert på variabelen *cond* inntil det sendes

et signal til variabelen fra en annen tråd.

- Den andre tråden som opprettes i *main()* starter med å eksekvere funksjonen *func_2()*. Her låses også mutexen, og det skrives ut en melding om signalering. Deretter kalles *pthread_cond_signal()* som sender et "oppvåkningssignal" til den betingede variabelen. Til slutt vil andre tråd skrive ut en melding om at den er ferdig, låse opp mutex igjen og deretter terminere.
- Når den betingede variabelen mottar signalet som er sendt fra den andre tråden, vil første tråd starte igjen. Den skriver også ut en melding om at den er ferdig, låser opp mutex og terminerer.

Kjør programmet selv og se hvordan det fungerer.

Nedenfor er det gitt mer utfyllende beskrivelser av hvordan de to funksjonene *pthread_cond_wait()* og *pthread_cond_signal()* virker. For enda mer informasjon om metodene, se f.eks. *man pthread_cond_wait* og *man pthread_cond_signal* på *itstud.hiof.no*.

1.5.1 Blokkere på en betinget variabel: *pthread_cond_wait()*

For at en betinget variabel skal fungere sikkert og pålitelig i *pthreads*, er det viktig at variabelen alltid brukes sammen med en mutex. Mutexen brukes til å beskytte håndtering av den betingede variabelen, slik at vi ikke får en race condition.

Anta at *cond_var* er en betinget variabel og at *lock* er en mutex. For å få en tråd til å blokkere (vente på den betingede variabelen), brukes følgende kall:

```
pthread_cond_wait(&cond_var, &lock);
```

Før dette kallet skal mutexen *lock* låses med et kall på *pthread_mutex_lock()*.

Et kall på *pthread_cond_wait()* fungerer slik:

- Tråden som kaller *pthread_cond_wait()* blokkeres. Den vil da være stoppet inntil den "vekkes" av en annen tråd som sender et signal til den betingede variabelen *cond_var*, med et kall på *pthread_cond_signal()*.
- Kallet på *pthread_cond_wait()* låser opp mutexen *lock* før tråden blokkeres. Mutexen låses igjen før tråden vekkes opp.

Flere tråder kan vente på samme betingede variabel, med ulike betingelser for å gå videre. Det kan hende at en tråd frigis fra en blokkering på en betinget variabel uten at betingelsen den venter på faktisk er oppfylt (se virkemåten for *pthread_cond_signal()* beskrevet nedenfor). Det er derfor vanlig (men ikke nødvendig) å legge kallet på *pthread_cond_wait()* i en løkke som sjekker om tråden faktisk kan gå videre fordi betingelsen den venter på er oppfylt.

Standard kode for å blokkere på en betinget variabel ser omtrent slik ut:

```
pthread_mutex_lock(&lock);              // Lås mutex
while (!betingelse)                     // Så lenge tråden ikke kan gå videre:
```

```
    pthread_cond_wait(&cond_var, &lock); // Blokker tråden på betinget variabel
pthread_mutex_unlock(&lock);            // Lås opp mutex igjen
```

Merk at denne måten å bruke en betinget variabel på henger nøye sammen med hvordan tråden vekkes opp igjen med *pthread_cond_signal()*.

1.5.2 Vekke opp en tråd som er blokkert på en betinget variabel: *pthread_cond_signal()*

For å vekke opp (starte igjen) en tråd som er blokkert på en betinget variabel *cond_var*, kan en annen tråd kalle funksjonen *pthread_cond_signal()*:

```
pthread_cond_signal(&cond_var);
```

Kallet på *pthread_cond_signal()* sender et "oppvåknings-signal" til den betingede variabelen. Dette fungerer slik:

- En tråd som venter på variabelen *cond_var*, fordi den har kalt *pthread_cond_wait()*, vekkes opp igjen.
- Hvis ingen tråder venter på variabelen, ignoreres signalet.
- Hvis det er flere tråder som venter på samme betingede variabel, velger systemet selv hvilken av dem som vekkes opp. Vi har her ingen kontroll på hvilken tråd som faktisk vekkes opp. Dette er forklaringen på hvorfor det er vanlig å legge kallet på *pthread_cond_wait()* i en løkke, som vist ovenfor — det er ikke sikkert at tråden kan gå videre selv om den vekkes.

  Grunnen til at betingede variable virker på denne måten, er at de vanligvis er implementert som en kø av tråder som er stoppet. Tråder som stoppes med *pthread_cond_wait()* legges bakerst i køen. Et kall på *pthread_cond_signal()* vil ta ut første tråd fra køen igjen og starte denne (mer om køer og andre datastrukturer vil dere få lære i emnet "Algoritmer og datastrukturer").

Funksjonen *pthread_cond_signal()* brukes vanligvis slik:

```
pthread_mutex_lock(&lock);      // Lås mutex
pthread_cond_signal(&cond_var); // Send signal til variabel
pthread_mutex_unlock(&lock);    // Lås opp mutex igjen
```

Det er igjen viktig å låse koden med en mutex før kallet på *pthread_cond_signal()*. Hvis dette ikke gjøres, kan vi få en race condition og risikerer å miste signalet.

1.5.3 Forenklet producer/consumer

I læreboken og i læringsmodulen i Canvas brukes "producer/consumer"-problemet for å illustrere bruk av mutexer og betingende variable. Nedenfor er en enklere variant av problemet, der produsenten bare produserer ett "item" om gangen, og konsumenten alltid forbruker dette før neste kan produseres:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <pthread.h>

#define MAX_WAIT 1000000

pthread_mutex_t mutex;
pthread_cond_t cond;
int count = 0, buffer = 0;

void *consumer()
{
   while(1)
   {
      pthread_mutex_lock(&mutex);

      while (buffer == 0)
         pthread_cond_wait(&cond, &mutex);

      usleep(rand() % MAX_WAIT + 1);
      buffer--;
      printf("Consumed item #%d\n", count);

      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
   }
}

void *producer()
{
   while(1)
   {
      pthread_mutex_lock(&mutex);

      while(buffer == 1)
         pthread_cond_wait(&cond, &mutex);

      usleep(rand() % MAX_WAIT + 1);
      buffer++;
      count++;
      printf("Produced item #%d\n", count);;

      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
  }
}

void main()
{
   pthread_t t1, t2;

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&cond, NULL);

   pthread_create(&t1, NULL, producer, NULL);
   pthread_create(&t2, NULL, consumer, NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond);
}
```

Programmet ovenfor bruker bare en enkel variabel *buffer*, som enten vil være lik null eller en, til å "lagre" et produsert "item". Systemet som programmet implementerer har følgende regler:

- Systemet har én produsent og én konsument.
- Det er plass til bare ett produsert item på lager.
- Produsenten kan produsere bare ett item om gangen.
- Konsumenten kan konsumere bare ett item om gangen.
- Så lenge lageret er tomt (*buffer* er lik null) må konsumenten vente.
- Så lenge lageret er fullt (*buffer* er lik en) må produsenten vente.

Merk også at det er lagt inn et tilfeldig "tidsforbruk" for hver gang det produseres eller konsumeres, får å få litt forsinkelse og dynamikk i systemet.

En utskrift fra dette programmet, der nummeret på siste item skrives ut, ser slik ut:

```
Produced item #1
Consumed item #1
Produced item #2
Consumed item #2
Produced item #3
Consumed item #3
Produced item #4
Consumed item #4
Produced item #5
Consumed item #5
Produced item #6
Consumed item #6 ...
```

Kjør producer/consumer programmet selv. Legg merke til hvordan de to trådene bruker den betingede variabelen til hele tiden å stoppe og starte hverandre.

2 Oppgaver

Nedenfor er det gitt tre oppgaver som skal besvares med både vanlig tekst og C-kode.

Oppgave 1

Følgende C-program bruker to tråder som begge "teller opp" og "teller ned" samme delte variabel *count*. Første tråd legger til 1 på variabelen 100 millioner ganger, andre tråd trekker 1 fra variabelen 100 millioner ganger:

```
#include <stdio.h>
#include <pthread.h>

int count = 0;

void *increase()
{
   int i;
   for(i = 0; i < 1e8; ++i)
      count++;
}
```

```
void *decrease()
{
   int i;
   for(i = 0; i < 1e8; ++i)
      count--;
}

int main()
{
   pthread_t thread1, thread2;

   pthread_create(&thread1, NULL, increase, NULL);
   pthread_create(&thread2, NULL, decrease, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

   printf("count = %d\n", count);
}
```

a. Kjør dette programmet selv, flere ganger. Du vil se at verdien av *count* er forskjellig hver gang. Selv om den ene tråden gjør "++" like mange ganger som den andre gjør "--", blir aldri variabelen lik 0! Gi en forklaring på hvorfor dette skjer.

b. Bruk en mutex til å "reparere" programmet slik at *count* alltid blir lik 0. Du skal ikke skrive om noe av den eksisterende koden, men bare legge til bruk av en mutex.

Oppgave 2

Programmet nedenfor bruker to mutexer:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1, mutex2;

void* print1()
{
   pthread_mutex_lock(&mutex1);
   usleep(100000);
   pthread_mutex_lock(&mutex2);

   printf("Her er funksjonen \"print_1\"!\n");

   pthread_mutex_unlock(&mutex2);
   pthread_mutex_unlock(&mutex1);
}

void* print2()
{
   pthread_mutex_lock(&mutex2);
   usleep(100000);
   pthread_mutex_lock(&mutex1);

   printf("Her er funksjonen \"print_2\"!\n");

   pthread_mutex_unlock(&mutex1);
   pthread_mutex_unlock(&mutex2);
}
```

```
int main()
{
   pthread_t t1, t2;

   pthread_mutex_init(&mutex1, NULL);
   pthread_mutex_init(&mutex2, NULL);

   pthread_create(&t1, NULL, print1, NULL);
   pthread_create(&t2, NULL, print2, NULL);

   while (1)
   {
     putchar('.');
     fflush(stdout);
     usleep(100000);
   }
}
```

Hovedprogrammet *main()* initaliserer her de to mutexene, og oppretter så to tråder. Første tråd starter med å kjøre funksjonen *print1()*, andre tråd kjører *print2()*. Deretter går *main()* inn i en evig løkke som i hver iterasjon skriver ut et punktum og deretter "sover" i et tidels sekund.

Kjør dette programmet selv. Du vil da se at ingen av de to trådene skriver ut noen meldinger, selv om begge kaller *printf()*!

Spørsmål: Gi en forklaring på hvorfor ingen av trådene får eksekvere *printf()*.

Oppgave 3

Programmet nedenfor leser først antall tråder som skal opprettes, *n*, fra bruker. Deretter startes *n* tråder som alle kjører funksjonen *terningkast()*:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define MAX_WAIT 2000000

pthread_mutex_t mutex;

void *terningkast(void *nr)
{
   while(1)
   {
      int terning = rand()%6 + 1;

      pthread_mutex_lock(&mutex);
      printf("Tråd %3ld -> %d\n", (long) nr, terning);
      pthread_mutex_unlock(&mutex);

      usleep(rand() % MAX_WAIT + 1);
   }
}

void main()
{
   int n;
   long i;
   pthread_t t;
```

```
   printf("n? ");
   scanf("%d", &n);

   pthread_mutex_init(&mutex, NULL);

   for (i = 0; i < n; i++)
     pthread_create(&t, NULL, terningkast, (void *) i);

   pthread_join(t, NULL);
}
```

Hver av de *n* trådene kjører en evig løkke. I hvert gjennomløp av løkken "kastes en terning" ved å velge tilfeldig ett av tallene 1, 2, 3, 4, 5 eller 6. "Terningkastet" skrives deretter ut og tråden "sover" en liten stud. Merk at utskriften beskyttes med en mutex, slik at trådene ikke overskriver hverandres output.

Når du kjører dette programmet får du en utskrift som f.eks. kan se slik ut med *n*=5 tråder:

```
n? 5
Tråd   0 -> 2
Tråd   1 -> 4
Tråd   2 -> 6
Tråd   3 -> 5
Tråd   4 -> 4
Tråd   2 -> 3
Tråd   1 -> 3
Tråd   2 -> 6
Tråd   0 -> 1
Tråd   1 -> 5
Tråd   0 -> 6
Tråd   4 -> 4
Tråd   3 -> 3
Tråd   2 -> 3
Tråd   4 -> 2
Tråd   1 -> 6
Tråd   3 -> 1
Tråd   0 -> 6
Tråd   4 -> 2
Tråd   2 -> 2 ...
```

a. Bruk en betinget variabel, og skriv om programmet ovenfor slik at:

- Hver gang en tråd får terningkast 1, skal tråden blokkere på den betingede variabelen.
- Hver gang en tråd får terningkast 6, skal tråden sende et signal til den betingede variabelen for å evt. å vekke opp en annen tråd som er blokkert.
- Når en tråd får terningkast 1 eller 6, skal den også skrive ut hvor mange tråder som nå er blokkert.

Utskriften fra programmet ditt kan f.eks. se slik ut med *n*=5 tråder:

```
n? 5
Tråd   0 -> 2
Tråd   1 -> 5
Tråd   2 -> 6 (0)
Tråd   3 -> 2
Tråd   4 -> 4
Tråd   1 -> 3
```

```
Tråd   1 -> 3
Tråd   4 -> 6 (0)
Tråd   1 -> 1 (1)
Tråd   0 -> 1 (2)
Tråd   3 -> 5
Tråd   2 -> 6 (1)
Tråd   1 -> 4
Tråd   1 -> 3
Tråd   4 -> 2
Tråd   3 -> 2
Tråd   2 -> 1 (2)
Tråd   1 -> 1 (3)
Tråd   4 -> 1 (4)
Tråd   3 -> 6 (3)
Tråd   3 -> 5
Tråd   0 -> 5
Tråd   3 -> 4
Tråd   0 -> 6 (2)
Tråd   0 -> 3
Tråd   3 -> 6 (1) ...
```

b. Kjør først programmet ditt med et lite antall tråder, f.eks. *n*=5. Du vil da se at programmet stopper etter en kort stund. Gi deretter programmet et stort antall tråder, f.eks. *n*=50, og la det kjøre lenge. Prøv å gi en forklaring på hva som skjer.

Hva skal du levere?

Svarene på oppgave 1a, oppgave 2 og oppgave 3b skal leveres sammen i et PDF-dokument med filnavn *oblig_6.pdf*. C-koden fra oppgave 1b og 3a skal leveres på to filer som heter *oppgave_1.c* og og *oppgave_3.c*. PDF-dokumentet og C-filene skal lastes opp i Canvas som svar på oppgaven.

Mye av innholdet i denne oppgaven ble opprinnelig laget av førsteamanuensis Nga Dinh ved HiØ til tidligere OS-kurs ved HiØ. Dette er oversatt fra engelsk og noe omarbeidet.