Wiki »

Exercise 4

Exercise 1 Precursor: Using the synchronization primitives

Printout from two threads

Den globale variabel mutex, som vi kalder m bliver lavet og initialisere den her:

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Nedenunder er lock og unlock tilføjet til funktionen printID()

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
//funktion til at printe thread ID og counter
void * printID(void * arg)
{
    int counter = 0;

    //caster void pointer til int pointer
    int * i = static_cast<int*>(arg);

    while(counter < 10)
    {
        pthread_mutex_lock(&m);
        std::cout <<"# "<< counter << " from thread "<< *i << std::endl;
        pthread_mutex_unlock(&m);
        ++counter;
        sleep(1);
    }
    return 0;
}
```

Det ses, at threads 1 og 2 udskriver fra 0 til 9, som ønsket. Dog bides der mærke i, at Thread 1 og 2 udskriver ikke i samme prioriteringsrækkefølge hele tiden.

```
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/1.1$ g++ -o TwoThreads TwoThreads.cpp -lpthread
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/1.1$ ./TwoThreads
# 0 from thread 1
# 0 from thread 2
# 1 from thread 1
# 1 from thread 2
# 2 from thread 1
# 2 from thread 2
# 3 from thread 2
# 3 from thread 1
# 4 from thread 1
# 4 from thread 2
# 5 from thread 1
# 5 from thread 2
# 6 from thread 1
# 6 from thread 2
# 7 from thread 1
# 7 from thread 2
# 8 from thread 2
# 8 from thread 1
# 9 from thread 1
# 9 from thread 2
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/1.1$ s
```

What does it mean to create a critical section?

critical section hentyder til et område i ens kode, hvor det er muligt at data kan blive behandlet af flere threads.

Which methods are to be used and where exactly do you place them?

For at lave en *critical section* uden om *printout* anvendes funktionskladense *pthread_mutex_lock(&m)* og *pthread_mutex_unlock(&m)*. *&m* er en refernece til den globale mutex *m*. *pthread_mutex_lock(&m)* indsættes lige inden *setAndTest()* kaldes, og der tjekkes for om *setAndTest()* returnerer *false*. *pthread_mutex_unlock(&m)* indsættes lige efter at *printf()* er blevet kaldt. Grunden til at de to funktioner sættes der, er at vi sikrer os det data som skal udsrikves ikke behandles.

What would you need to change in order to use semaphores instead?

Den grundlæggende tilgang ville være den samme med semaphores. Dvs. at der skulle laves en *critical section* udenom det samme kode, som med mutex. Specifikt så skulle der dog oprettes og initialiseres en semaphore istedet for on mutex. Denne sempahore skulle initialiseres til have en værdi på 1. Ydermere skulle

sem_wait() kaldes istedet for *mutex_lock()*. Og i stedet for *mutex_unlock()* skulle *sem_post()* kaldes.

At anvende semaphore istedet for mutex i implementeringen, ville ikke have nogen synlig betydning, da det data som behandles af de to threads, ville være "låst" i begge tilfælde.

Exercise 2 Fixing vector

Med mutex

For at løse problemet med mutex, så bruges kommandoerne unlock() og lock() ligesom i sidste opgave, samt initialisering og opretning af mutex som en global variable. Dette er smart, fordi det gør den tilgængelig over alt.
Opretter en mutex, samt initialiserer den

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Opretter et globalt vector objekt

```
Vector obj;
```

Dertil kan write funktionen skrives

```
void* writer(void *arg)
{
    while(1)
    {
        int *id = static_cast<int*>(arg);

        //locker mutex'en
        pthread_mutex_lock(&m);

        //tjek om alle objektets pladser er ens
        if (obj.setAndTest(*id)==false)
        {
            std::cout << "setAndTest() returned false" <<std::endl;
            sleep(1);
        }
        //unlocker mutex'en
        pthread_mutex_unlock(&m);
```

```
    }
    return NULL;
}
```

Fuld dokumention for implementeringen med semaphore kan ses her source:exercise4/2.0/VectorMutex

Dertil compileres det og programmet køres, hvor værdien 99 angiver antallet af threads der oprettes.

```
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex$ make
g++ -MTbuild/host/Vector_mutex.d -MM -I. Vector_mutex.cpp > build/host/Vector_mutex.d
g++ -MTbuild/host/Vector_mutex.o -MM -I. Vector_mutex.cpp >> build/host/Vector_mutex.d
g++ -c Vector_mutex.cpp -o build/host/Vector_mutex.o -I.
g++ -I. build/host/Vector_mutex.o -o bin/host/Vector_mutex -lpthread
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex$ ls
bin  build  Makefile  Vector_mutex.cpp  Vector_mutex.hpp
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex$ cd bin/host
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex/bin/host$ ls
Vector_mutex
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex/bin/host$ ./Vector_mutex
99
^C
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/2.0/VectorMutex/bin/host$
```

Med semaphore

Oprettelse af global semaphore variabel

```
sem_t sem;
```

Initiering af sem

```
sem_init(&sem, 0, 1);
```

sem_init() initialiserer en semaphore, og informerer systemet om hvorvidt den initialiserede semaphore vil blive brugte af flere processer, eller om den vil blive brugt af flere threads, som tilhører en process.

- At parameter nr. 2 er *0*, betyder at den initialiserede semaphore, bliver brugt af flere threads, som tilhører en enkelt process.
- At parameter nr. 3 er *1*, betyder at den initialiserede sempahore, sættes til at have en værdi lig med *1*.

```
//Vector_semaphore.hpp
bool setAndTest(int n, sem_t* sem)
      {
        //dekrementer sem(lås)
        sem_wait(sem);
        set(n);
        bool res = test(n);
        //inkrementer sem(lås op)
        sem_post(sem);
        return res;
      }
```

sem_wait() dekrementerer værdien på den semaphore, som der i parameteren bliver referreret til. Hvis værdien af semaphore'n er større end 0, mindskes værdien med 1, og sem_wait() returnerer. Hvis værdien af semaphore'n er lig med 0, så venter funktionen indtil værdien er større 0, mindsker værdien med 1, og returnerer. At funktionen venter på at værdien bliver større end nul, betyder at den blokere for at de efterfølgende unstruktioner kan udføres.

sem_post() inkrementerer værdien på den semaphore, som der i parameteren bliver referreret til. Det betyder at hvis en *sem_wait()* venter på at kunne dekrmentere værdien, så vil den kunne gøre det efter at *sem_post()* har returneret, og en anden *thread* kan fortsætte.

Fuld dokumention for implementeringen med semaphore kan ses her source:exercise4/2.0/VectorSemaphore

Nedestående billede viser eksekvering af programmet *Vector_semaphore*, med 100 threads

```
stud@stud-virtual-machine:~/isu_work/exercise4/2.0/VectorSemaphore/bin/host$ ./Vector_semaphore
100
^C
stud@stud-virtual-machine:~/isu_work/exercise4/2.0/VectorSemaphore/bin/host$
```

Som der kan ses på billedet udskrives der ingen tekst, som ville indikere at *setAndTest()* havde returneret false, og at der altså var sket fejl.

Does it matter, which of the two you use in the scenario? Why, why not?

Ja det giver en forskel, for det første så skal man skrive meget mere kode til semaphore end til mutex, så der er implementerings tid. Udover det så løser begge løsninger problemet, så derfor har det i dette senarie næsten ingen betydning.

Where have you placed the mutex/semaphore and why?

Vi har valgt at placere mutex'en som global variabel. Dette gør vi, så den kan tilgås alle steder i programmet. Hvis vi havde lagt den i en klasse, som initialiserede mutex'en, så ville vi få en mutex, til hver klasse der blev oprettede, og derfor vil de ikke låse for hinanden.

I løsningen med semaphores, oprettes semphoren *sem* som en global variabel, men bruges kun i klassen *Vector*. Et stærkt argument for ikke at gøre det sådan, er at man ville kunne bruge den globale semphore i flere klasser. I sådan et scenarie ville der kunne opstå situatuioner, hvor data ville være låst for tråde, som slet ikke behøvede at være låst. Det ville altså kunne have konsekvenser for et en process' effektivitet. Grunden til at vi implementerede løsningen sådan, skyldes intuition. Hivs vi dog skulle løse et lignende problem med semaphores ville vi forsøge os med en implementering hvor semphores var klasse medlemmer, hvis klassen kun blev brugt i forbindelse med *threads*.

Exercise 3 Ensuring proper unlocking

For at kunne teste om programmet virker, så udskrives der, at "mutex is locked" lige efter den låses i contructoren. Og der udskrives at "mutex is unlocked" i destructoren lige før den låses op. Dette vil medføre at vi kan sikre os at mutex'en rent faktisk tvinges til at blive låst og låst op, og ikke blot at Vector objektet bliver behandlet korrekt af de oprettede threads. Derudover initialiseres attrubten via member initialiser, og kommandoerne pthread_mutex_lock() og pthread_mutex_unlock()

```
#include <pthread.h>
#include <iostream>
using namespace std;
class scoped_locking
{
public:
   scoped_locking(pthread_mutex_t *m)
   : mtx(m)  // mutex pointer mtx peger på mutex m
      {
         // mutex mtx låses
         pthread_mutex_lock(mtx);
         cout<<"mutex is locked" << endl;

      }
   ~scoped_locking()
      {
         // mutex mtx låses op
         cout<<"mutex is unlocked" << endl;
         pthread_mutex_unlock(mtx);
      }

private:
    pthread_mutex_t *mtx;
};
```

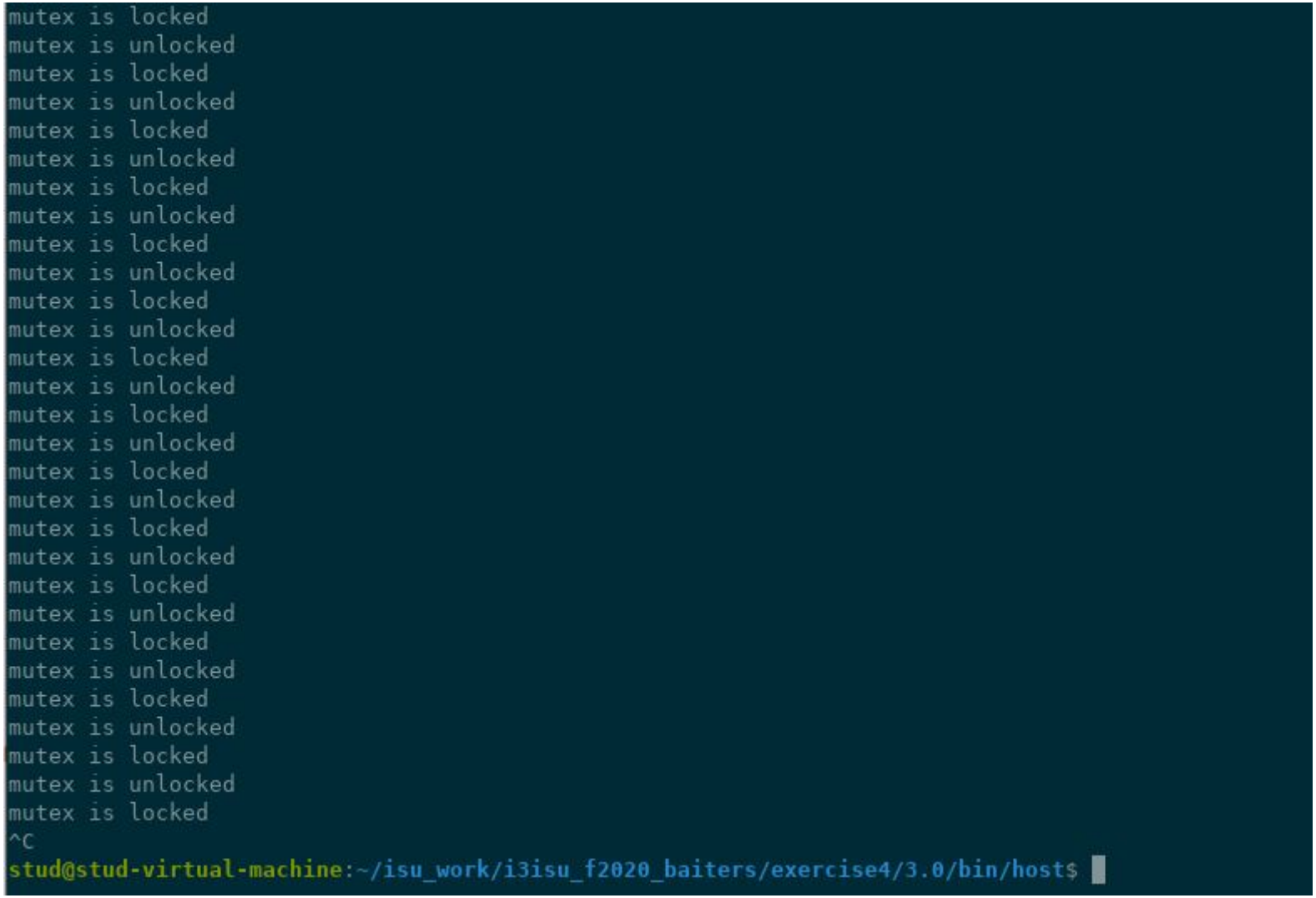
I funktionen setAndTest(), oprettes der et objekt fra klassen scoped_locking, for at den kan udføre operationen, at åbne og låse for mutex'erne samt sørge for, at de ikke bliver låst fast i "locked". Som der kan ses i nedestående code-snippet, så bliver mutex *m* videregivet som reference. Dette er afgørende for at opnå den korrekte funktionalitet, da de threads som behandler samme objekt skal være afhængige af samme mutex. Hvis en mutex i denne implementering ville blive givet med *by value*, ville *scoped_locking* klassen behandle en ny mutex hvergang der blev oprettet et objekt af denne klasse.

```
bool setAndTest(int n)
     {
        // Opretter scoped_locking objet og initialiserer det med
        // en reference til mutex m
        scoped_locking scope(&m);
        set(n);
        return test(n);
     }
```

Filen laves og eksekveres, hvori antal elementer der skal køres igennem er 99.

```
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/3.0$ make
g++ -MTbuild/host/Vector_mutex2.d -MM -I. Vector_mutex2.cpp > build/host/Vector_mutex2.d
g++ -MTbuild/host/Vector_mutex2.o -MM -I. Vector_mutex2.cpp >> build/host/Vector_mutex2.d
g++ -c Vector_mutex2.cpp -o build/host/Vector_mutex2.o -I.
g++ -I. build/host/Vector_mutex2.o -o bin/host/Vector_mutex2 -lpthread
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/3.0$ ls
bin  build  Makefile  scoped_lock.hpp  Vector_mutex2.cpp  Vector_mutex2.hpp
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/3.0$ cd bin/host
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/3.0/bin/host$ ls
Vector_mutex2
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/3.0/bin/host$ ./Vector_mutex2
99
```

Idet funtionen setAndTest er i en uendelig while-loop, så vil programmet også testes uendelig mange gange, og derfor aldrig blive færdigt. Derfor bruges kommandoen ctrl c for at stoppe programmet. Hvortil det ses at programmet ikke opnår at låse sig selv på noget tidspunkt



Exercise 4 On target

Her skal filerne cross compiles, overføres samt ekskeveres på Rpi'en.

```
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0$ make
arm-rpizw-g++ -MTbuild/host/Vector_mutex3.d -MM -I. Vector_mutex3.cpp > build/host/Vector_mutex3.d
arm-rpizw-g++ -MTbuild/host/Vector_mutex3.o -MM -I. Vector_mutex3.cpp >> build/host/Vector_mutex3.
d
arm-rpizw-g++ -c Vector_mutex3.cpp -o build/host/Vector_mutex3.o -I.
arm-rpizw-g++ -I. build/host/Vector_mutex3.o -o bin/host/Vector_mutex3 -lpthread
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0$ ls
bin  build  Makefile  scoped_lock.hpp  Vector_mutex3.cpp  Vector_mutex3.hpp
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0$ cd bin/host
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0/bin/host$ ls
Vector_mutex3
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0/bin/host$ scp Vector_mutex3
 root@10.9.8.2:
Vector_mutex3                                 100%   17KB   2.1MB/s   00:00
stud@stud-virtual-machine:~/isu_work/i3isu_f2020_baiters/exercise4/4.0/bin/host$
```

Og filen eksekveres:

```
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
mutex is locked
mutex is unlocked
^C
root@raspberrypi0-wifi:~#
```

Dertil kan det konkluderes, at der var ingen forskel mellem at eksekvere filen på computeren eller på Raspberry'en, da det giver samme resultat, idet der ikke opstår fejl meddelelser på Rpi'en. Og det ses også i sekvensen ovenover, at mutex'en først bliver "locked", og så "unlocked", som så fortsætter i et uendeligt while - loop.

Tilføj en kommentar

1.1.JPG (38,8 KB) Thomas Gammelby Lodberg, 2020-02-24 11:31
2.0.JPG (24,2 KB) Thomas Gammelby Lodberg, 2020-02-24 13:34
resultat.JPG (52,8 KB) Thomas Gammelby Lodberg, 2020-03-01 12:33
resultat.JPG (51 KB) Thomas Gammelby Lodberg, 2020-03-01 12:52
resultat_mutex_ex2.JPG (51,2 KB) Thomas Gammelby Lodberg, 2020-03-01 13:11
crosscompile.JPG (54,9 KB) Thomas Gammelby Lodberg, 2020-03-01 13:48
resultat4.0.JPG (17,6 KB) Thomas Gammelby Lodberg, 2020-03-01 13:49
Vector_sempahore_test.PNG (8,65 KB) Lasse Andresen, 2020-03-01 14:21
vectorsemaphore_test.PNG (11,1 KB) Lasse Andresen, 2020-03-01 14:22
resultat_mutex_ex22.JPG (39,1 KB) Thomas Gammelby Lodberg, 2020-03-01 14:30
resultat_mutex_ex22.JPG (39,1 KB) Thomas Gammelby Lodberg, 2020-03-01 14:31
resultat.JPG (46,9 KB) Thomas Gammelby Lodberg, 2020-03-01 14:33
resultat4.1.JPG (29,5 KB) Thomas Gammelby Lodberg, 2020-03-01 21:12