Exercise 4 - Thread Synchronization I

Exercise 1 Precursor: Using the synchronization primitives

Exercise 1.1 Printout from two threads...

- **What does it mean to create a critical section ?**
  Det er en måde, så man kan sikre sig på at en og kun en tråd eksekverer et stykke kode uden at blive "forstyret". Man kan altså bruge det til at synkronisere tråde i den samme proces.
- **Which methods are to be used and where exactly do you place them?**
  Der findes to metoder, Mutex og Semaphore. Førstnævnte er en låsemekanisme, hvor Semaphore er mere et slags signal. De bliver begge erklæret inde i processen.

Nedenstående kode viser hvordan der er blevet tilføjet en global mutex til programmet fra en tidligere opgave, som udskriver fra to threads:

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

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void* threadfunc(void *arg)
{
    int tid=*static_cast<int*> (arg);

    for(int count=1;count<=10;count++)
    {
        pthread_mutex_lock(&mtx);
        std::cout<<"Hello #"<<count<<" from thread "<<tid<<std::endl;
        pthread_mutex_unlock(&mtx);
        sleep(1);
    }

    return nullptr;
}

int main(void)
{

    pthread_t tid1;
    pthread_t tid2;
    int no1=1;
    int no2=2;
    void* res;
    pthread_create(&tid1,NULL,threadfunc,&no1);

    pthread_create(&tid2,NULL,threadfunc,&no2);
    pthread_join(tid1,&res);
    pthread_join(tid2,&res);
```

```
    return 0;
}
```

Det ses her hvordan mutex bliver initialiseret globalt, og hvordan den inde i processen først låser, og efterfølgende låser den op igen.

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe4/exe1_1$ ./bin/host/prog
Hello #1 from thread 1
Hello #1 from thread 2
Hello #2 from thread 1
Hello #2 from thread 2
Hello #3 from thread 1
Hello #3 from thread 2
Hello #4 from thread 1
Hello #4 from thread 2
Hello #5 from thread 1
Hello #5 from thread 2
Hello #6 from thread 1
Hello #6 from thread 2
Hello #7 from thread 1
Hello #7 from thread 2
Hello #8 from thread 1
Hello #8 from thread 2
Hello #9 from thread 1
Hello #9 from thread 2
Hello #10 from thread 1
Hello #10 from thread 2
```

Exercise 1.2 Mutexes & Semaphores

Når lock/unkock bruges omkring en funktion ville det være bedst at locke/unlocke i selve funktionen, da dette sikrer at det bliver gjort. Hvis der lockes/unlockes som en wrapper omkring funktionskaldet, ville en anden programmør selv skulle gøre det hvilket vi ikke kan sikre os, dermed er det ikke sikkert at funktionen virker. Det ville ikke være smart, at have den som en global variabel, hvis den eksklusivt bruges i en klasse. Den kan dermed blive tilgået uden for klassen, hvilket giver en usikkerhed i, hvem der kan bruge den hvornår.
Semaphores virker i princippet på samme måde som en Mutex, ved at kunne gemme og låse ønskede elementer i ens kode. Forskellen på de to metoder er at Mutex skifter fra at være mellem 0 og 1, hvor Semaphores er låst når den er 0 og elles kan være alle positive tal.
Nedenstående kode har samme funktion som i forrige opgave, denne gang er det initaliseret med Semaphores istedet for Mutexes:

```
#include <iostream>
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <semaphore.h>

static sem_t sem1;

void* threadfunc(void *arg)
{
    int tid=*static_cast<int*> (arg);

    for(int count=1;count<=10;count++)
    {
```

```
        sem_wait(&sem1);
        std::cout<<"Hello #"<<count<<" from thread "<<tid<<std::endl;
        sem_post(&sem1);
        sleep(1);
    }

    return nullptr;
}

int main(void)
{
    sem_init(&sem1, 0, 1);
    pthread_t tid1;
    pthread_t tid2;
    int no1=1;
    int no2=2;
    void* res;
    pthread_create(&tid1,NULL,threadfunc,&no1);
    pthread_create(&tid2,NULL,threadfunc,&no2);
    pthread_join(tid1,&res);
    pthread_join(tid2,&res);
    sem_close(&sem1);

    return 0;
}
```

Eksekvering af koden:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe4/exe1_2$ ./bin/host/prog
Hello #1 from thread 1
Hello #1 from thread 2
Hello #2 from thread 2
Hello #2 from thread 1
Hello #3 from thread 1
Hello #3 from thread 2
Hello #4 from thread 1
Hello #4 from thread 2
Hello #5 from thread 1
Hello #5 from thread 2
Hello #6 from thread 1
Hello #6 from thread 2
Hello #7 from thread 2
Hello #7 from thread 1
Hello #8 from thread 2
Hello #8 from thread 1
Hello #9 from thread 2
Hello #9 from thread 1
Hello #10 from thread 2
Hello #10 from thread 1
```

Ovenstående billede viser, at der i dette tilfælde ikke er nogen forskel på at bruge Mutex eller Semaphores.

Exercise 2 Fixing vector

I dette afsnit følges op på vektor-programmet fra sidste sidste opgave. I denne omgang udvides programmet med henholdsvis mutex og semaphorer. Implementeringen af henholdsvis symaphore versionen og mutex

versionen minder meget om hinanden. Derfor gennemgås mutex- eksemplet, hvorefter foreskelle i semaphore programmet gennemgås. Man kan argumentere for at bruge begge typer synchronization primitives, mutex er i sin implementation en lås, der giver en enkelt tråd adgang af gangen. Mens en semaphor også kan benyttes som en lås, hvilket dog kræver lidt større omtanke ift. hvordan den initialiseres.

I begge implementationer, oprettes en mutex/semaphore og initieres i Vector-klassens constructer. Denne slettes derefter i Vector-klassens destructor. I setAndTest() låses mutexen som det første i funktionen, og låses op som det sidste.

```
#ifndef VECTOR_HPP_
#define VECTOR_HPP_
#include <pthread.h>
//=========================================================
// Class: Vector
// contains a size_-size vector of integers.
// Use the function setAndTest to set all elements
// of the vector to a certain value and then test that
// the value is indeed correctly set
//=========================================================
class Vector
{
public:
   Vector(unsigned int size = 10000) : size_(size)
      {

         mut_= new pthread_mutex_t;
         pthread_mutex_init(mut_,nullptr);
         vector_ = new int[size_];
         set(0);

      }
   ~Vector()
      {
         delete[] vector_;
         pthread_mutex_destroy(mut_);
         delete mut_;
      }
   bool setAndTest(int n)
      {
         pthread_mutex_lock(mut_);
         set(n);
         bool res=test(n);
         pthread_mutex_unlock(mut_);
         return res;

      }
```

Fordelen ved at implementere mutexen/semaphoren inde i Vector-klassen, er at det er nemt for andre at benytte klassen, uden at have kendskab til semaphorer eller mutexer. Desuden kunne en ydre implementation af mutex/semaphorer ødelægge klassens funktionalitet, da man i sådant tilfælde ikke ville have kendskab til implementationen af funktionerne i klassen. Et argument mod at implementere mutex/semaphore inde i klassen, er at klassen jo ikke nødvendigvis skal benyttes sammen med tråde. Dog fungerer klassen stadig i en enkel tråd, dog en lille smule mindre effektivt, da funktionskaldende på mutex/semaphore giver lidt spildtid. En anden mulgihed der nævnes, er at oprette en global mutex/semaphore, og benytte denne inden i klassen. Dette er dog uhensigtsmæssigt, da det kræver at en programmør der skal benytte klassen ved hvilken

mutex/semaphore der skal benyttes, samt at klassen er afhængig af at denne benyttes korrekt, det kunne f.eks. give implikationer, hvis semaphoren/mutexen også blev benyttet i andre sammenhænge.
Derfor er det sikrest og mest genanvendeligt at implementere inden i klassen, dog unødvendigt at implementere, hvis man ved at klassen kun skal bruges i en enkelt tråd.

For neden ses implementationen med semaphorer. I dette tilfælde initieres semaphoren i konstrukteren, med en værdi på 1. I hvert kald af setAndTest() dekrementeres semaphoren før med en (til 0) hvilket betyder at andre kald af funktionen ikke kan tilgå semaphoren. I slutningen af funktionen inkrementeres semaphoren, så et andet kald af funktionen kan få adgang.

```
class Vector
{
public:
   Vector(unsigned int size = 10000) : size_(size), sem(nullptr)
      {

         sem=new sem_t;
         sem_init(sem,0,1);
         vector_ = new int[size_];
         set(0);

      }
   ~Vector()
      {
         delete[] vector_;
         sem_close(sem);
      }
   bool setAndTest(int n)
      {
         sem_wait(sem);
         set(n);
         bool res=test(n);
         sem_post(sem);
         return res;

      }
```

Forneden ses et skærmudklip af en kørsel af programmet, det kan ses at der nu ikke kommer nogen fejlmældinger.

```
stud@stud-virtual-machine:~/Desktop/Courses/i3isu_f2020_beany_business/exe4/exe2_2$ ./bin/host/prog
Indtast antal threads(1-100): 100
```

Exercise 3 Ensuring proper unlocking

Et af de større problemer ved måden vi tidligere har implementeret Mutexs og Semaphores er, at programmet ikke nødvendigvis behøver at låse dem op igen inden en funktionen går ud af scope. Derfor hvis man er uheldig, kan man ende med at have låst sin Mutex eller Semaphore for altid og har ikke mulighed for at tilgå den.
En løsning til dette problem er at bruge en *scope locker*, den skal sørge for at en Mutex eller Semaphore altid bliver låst op når en funktion er færdig. Princippet bag en *scope locker* er, at i constructoren låser mutex'en eller semaphore'en, også bliver de låst den op igen i destructoren. En variabel oprette i funktionen vil derfor ikke kunne bruges andre steder, end inde i funktionen selv, og er derfor bestykket.

Nedenstående kode viser implementeringen af ScopedLocker.hpp med både constructor og destructor. Constructoren gemmer mutex referencen i en midlertidig variabel og herefter låser mutex'en. For at skabe mere overblik printer til terminalen, så det er nemmere at følge med. Destructoren gør ikke så meget andet end at låse op for mutex'en igen, og af samme grund som før printes det til terminalen.

```
#include "Vector.hpp"
#include <pthread.h>
#include <iostream>
using namespace std;

class ScopedLocker
{

public:
  ScopedLocker(pthread_mutex_t *mutex)
  {
    tempMutex = mutex;
    pthread_mutex_lock(tempMutex);
    cout << "Mutex is locked." << endl;
  }

  ~ScopedLocker()
  {
    pthread_mutex_unlock(tempMutex);
    cout << "Mutex is unlocked." << endl;
  }

private:
  pthread_mutex_t *tempMutex;
};
```

Med ScopeLock-klassen er det muligt at beskytte et nyoprettet objekt inde i funktioen. Tilføjelse til Vector.hpp i funktionen setAndTest(), hvor ScopedLocker oprettes i. Klassen besidder allerede en privat attribut mut_, som bliver konstrueret når et objekt af denne klasse oprettes. Med dette er det derfor muligt at undlade låse-funktionen *pthread_mutex_lock(mut)_* og funktionen til at låse op igen *pthread_mutex_unlock(mut)_*, og disse står derfor udkommenteret:

```
bool setAndTest(int n)
     {
        ScopedLocker temp(mut_);
        //pthread_mutex_lock(mut_);
        set(n);
        bool res=test(n);
        //pthread_mutex_unlock(mut_);
        return res;

     }
```

Derudover er der også blevet tilføjet lidt kode til vectorthread.cpp, så det ikke kun er muligt at se hvornår det *ikke* lykkedes, men også når det fungerede:

```
if(!err)
   {
       std::cout<<"Error detected in thread with ID: "<<tArg->tid<<"\n";
   }
       std::cout<<"It worked thread ID: "<<tArg->tid<<"\n";
```

```
        sleep(1);
    }
```

Det ses her at vi har ingen problemer med at bygge og eksekvere filerne, og samtidig ingen fejl får. Dette er kun et udklip af terminalen, da outputtet af så langt:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe4/exe3$ make
mkdir -p ./build/host
g++ -lpthread -c vectorthread.cpp -o build/host/vectorthread.o
mkdir -p ./bin/host
g++ ./build/host/vectorthread.o -o bin/host/prog -lpthread
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe4/exe3$ ./bin/host/prog
Indtast antal threads(1-100): 2
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 2
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 2
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 2
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
```

Exercise 4 On target

For at teste forskellen fra eksekvere programmet på computer og Rpi, skal det først kompileres til at kunne køre på Rpi. Herefter kopires det til Rpi og eksekveres det, og resultatet ser ud på følgende måde:

```
root@raspberrypi0-wifi:~# ./prog
Indtast antal threads(1-100): 2
Mutex is locked.
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is unlocked.
It worked thread ID: 2
Mutex is locked.
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 2
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 2
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
Mutex is locked.
Mutex is locked.
Mutex is unlocked.
It worked thread ID: 1
```

Det ses her at eksekveringen af programmet på Rpi giver det samme resultat som på host.

- **Tilføjet af Mathias Holm Brændgaard for 3 måneder siden**

Læringsmål

- Læringsmål #1: Grundlæggende om mutexes

Gruppen forklarer tilfredsstillende de grundlæggende egenskaber ved både mutex og semaphores samt forskelle mellem disse. Der uddybes dog ikke, hvad der menes med "hvor Semaphores er låst når den er 0 og elles kan være alle positive tal.", hvilket ville give bedre indsigt i, hvorfor man kunne ønske af bruges semaphores i stedet for mutexes. Resten er dog tilstrækkeligt forklaret til, at læringsmålet betragtes som opfyldt. Godkendt. ✔

- Læringsmål #2: Two threads

Gruppen får her implementeret programmet "two threads" på hensigtsmæssig vis med både mutex og semaphores. Der bliver tilfredsstillende forklaret, hvorfor gruppen har valgt den pågældende implementeringsmetode. Alt i orden. Godkendt ✔

- Læringsmål #3: Fixing Vector

Gruppen formår at implementere passende låse i programmet, så det forløber uden fejl. Der argumenteres for, hvorfor netop den pågældende metode (ændring i selve Vector.hpp) blev valgt, og gruppen implementerer desuden programmet både med mutexes og semaphores. Alt i orden. Godkendt ✔

- Læringsmål #4: Scoped Locking

Gruppen får først og fremmest implementeret en løsning, som virker på både host-system og RPi. Gruppen forklarer først, hvilket problem Scoped Locking skal løse, og efterfølgende hvordan dette løses. Dette gøres ved at tilføje en privat mutex i hvert objekt af klassen Vector, således at flere objekter af klassen kan oprettes i samme program uden problemer, hvilket viser god forståelse for stoffet. Alt i orden. Godkendt ✔

Feedback

Fine implementeringer og dokumentation. Nogle forklaringer er dog en smule upræcise, hvis man ikke har source-koden ved hånden, imens man læser.

Must haves

Relevante filer ✔

Makefiles ✔

Konklusion

Gruppen har udvist stor omhyggelighed i forhold til afvikling af øvelsen og har følgeligt opfyldt alle fire læringsmål. Alt i orden, intet at bemærke. Godkendt.

lab4_exe3_withworked.png (59,6 KB) Magnus Nygaard Lund, 2020-02-29 14:12

lab4_exe1_2.png (42 KB) Magnus Nygaard Lund, 2020-03-01 12:06

lab4_exe1_1.png (42 KB) Magnus Nygaard Lund, 2020-03-01 12:06

lab4_exe4.png (20,3 KB) Magnus Nygaard Lund, 2020-03-02 09:30

exe4_screen.png (7,12 KB) Jens Nørby Kristensen, 2020-03-02 11:42