

[Wiki »](#)

Exercise4 - Thread synchronization I

Exercise 1 - Precursor: Using the synchronization primitives ¶

Exercise 1.1 Printout from two threads...

På billedet herunder ses koden for programmet hvor der er indsat mutex.

```
#include <iostream>
#include <mutex>
#include <semaphore.h>

using namespace std;
pthread_mutex_t mut;

void* hello1(void* arg)
{
    for(int i=0;i<10;i++){
        pthread_mutex_lock(&mut);

        cout << "Hello #" << i << "from thread " << pthread_self() << endl;

        pthread_mutex_unlock(&mut);

        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t0,t1;

    pthread_mutex_init(&mut,NULL);
```

- What does it mean to create a *critical section* ?

En kritisk sektion er et område med en restriktion på antal af tråde som kan køre i området. Hvis det kritiske område er lavet med en mutex kan der kun være en tråd, men med semaphores kan det brugerdefineres hvor mange tråde, der er tilladt.

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

som nævnt i spørgsmålet over kan der gøres brug af mutexes og semaphores. De placeres i det scope, so skal restrikeres. Der låses inden det begrænsede område og låses op efter det begrænsede område.

Exercise 1.2 Mutexes & Semaphores

What would you need to change in order to use semaphores instead? Would it matter? For each of the two there are 2 main characteristics that hold true. Specify these 2 for both

En semaphore virker nogenlunde på samme måde som en mutex, men i stedet for mutex_lock og mutex_unlock bruges sem_wait og sem_post. Derudover skal der også laves en sem_destroy i main, mutexen skulle ikke nedlægges efter brug.

Herunder ses samme program som opgave 1 men lavet med en semaphore istedet for en mutex.

```
using namespace std;
sem_t mute;

void* hello1(void* arg)
{
    for(int i=0;i<10;i++){
        sem_wait(&mute);

        cout << "Hello #" << i << "from thread " << pthread_self() << endl;

        sem_post(&mute);

        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t0,t1;

    sem_init(&mute,0,1);

    pthread_create(&t0,NULL,hello1,NULL);
    pthread_create(&t1,NULL,hello1,NULL);

    pthread_join(t0,NULL);
    pthread_join(t1,NULL);

    sem_destroy(&mute);
}
```

Exercise 2 Fixing vector

Questions to answer:

- **Does it matter, which of the two you use in this scenario (Mutex/Semaphore), why/not?**

Det betyder ikke noget hvilke af de to vi bruger, men en mutex er bedst idet den er designet til at der kun er en tråd som skal tilgå det afgrænsede område, som det er i denne opgave.

- **Where have you placed the mutex/semaphore and why and ponder what the consequences are for your particular design solution**

Vi har placeret mutexen rundt om kaldene i "Writer" funktionen. På den måde er der ikke andre tråde, som kan tilgå de pågældende kald imens de er i brug af en tråd. Dette kan ses på billedet herunder:

```
using namespace std;

pthread_mutex_t mut;

void* writer(void* arg)
{
    pthread_mutex_lock(&mut);
    if ((*(Vector*)arg).setAndTest(*(int*)pthread_self()) == false){
        cout << "error" << endl;
    }else{
        cout << "korrekt" << endl;
    }
    pthread_mutex_unlock(&mut);
    return NULL;
}

int main(int argc, char *argv[])
{
    int input;
    Vector v1;

    pthread_mutex_init(&mut, NULL);

    cout << "enter a number between 1 and 100" << endl;
    cin >> input;

    pthread_t t[input];

    for(int i=0; i<input; i++){
        pthread_create(&t[i], NULL, writer, &v1);
    }
}
```

Exercise 3 Ensuring proper unlocking

The idea behind the Scoped Locking idiom is that you create a class ScopedLocker which is passed a mutex.

How is it passed a mutex? by value or by reference and why is this important?

Der gøres brug af "pass by reference" på den måde er vi sikre på, at det er den rigtige mutex der bliver låst og låst op igen.

scopelock klassen:

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

#ifndef SCOPELOCK_HPP_
#define SCOPELOCK_HPP_

class scopelock{
public:
    scopelock(pthread_mutex_t *mut ){
        mutex = mut;
        pthread_mutex_lock(mutex);
    }

    ~scopelock(){
        pthread_mutex_unlock(mutex);
    }
private:
    pthread_mutex_t *mutex;
};

#endif
```

Måden hvorpå klassen scopelock virker er, at der i constructoren refereres til en global mutex som bliver kaldt som en reference og som så bliver låst. Når destructoren bliver kaldt låser den op for den specifikke globale mutex. Klassen bruges ved at oprette et objekt inde i det scope der skal låses og når scopet har kørt igennem koden kalder den derved objektets destructor. I dette tilfælde er objektet oprettet i setandtest for klassen vector:

```
class Vector
{
public:
    Vector(unsigned int size = 10000) : size_(size)
    {
        pthread_mutex_init(&mm, NULL);
        vector_ = new int[size_];
        set(0);
    }

    ~Vector()
    {
        delete[] vector_;
    }

    bool setAndTest(int n)
    {
        scopelock ll(&mm);
        set(n);
        return test(n);
    }
}
```

Ud over det initieres mutexen i constructoren til klassen vector.

Grunden til at der i denne opgave er taget et valg om at kalde scopelock inde i vector klassen istedet for writer funktionen er, at da vi kørte den med scopelock inde i writerfunktionen kom der fejl.

Kørsel af programmet:

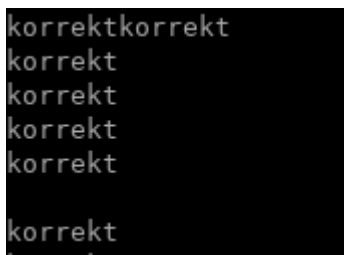
```
stud@stud-virtual-machine:~/Desktop/ISU/lektion4/exe3$ ./exe
enter a number between 1 and 100
100
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
```

Exercise 4 On target

I denne øvelse recompilerer vi koden til RPIen i øvelse 3 og tester at programmet virker på vores RPI. Dette gøres ved at lave om i makefilen.

Kørsel af programmet:

```
root@raspberrypi0-wifi:~/ISU/lektion4# ./exe
enter a number between 1 and 100
100
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
korrekt
```



```
korrektkorrekt
korrekt
korrekt
korrekt
korrekt
korrekt
```

Som det kan ses sker der en lille fejl på RPlen ved at den nogle gange udskriver "korrekt" på samme linje og nogle gange springer den en linje over, men der kommer ingen fejl.

- Tilføjet af Mathias Holsko Jespersen for 25 dage siden
-

Review:

Exercise 1:

Det ser ud til at spørgsmålet om hvilke metoder/API'er der er brugt, måske er misforstået, men opgaven er løst og der ligger virkende kode i Repo.

Jeg tror spørgsmålet hentyder til at i har brugt mutex_lock / unlock osv.

Exercise 1.2:

Fint at I har lavet opgaven med en semaphore selvom ex1.2 var ment som en tekst-besvarelse. Jeg synes lidt andel del af spørgsmålet er udeladt, i forhold til deres egenskaber.

Exercise2:

Her mangler besvarelse af nogle spørgsmål, i hvert fald efter hvordan jeg har forstået opgaven, Og hertil også noget besvarelse til hvad konsekvenserne kunne være til de givende løsninger. Koden ser dog ud til at virke hvilket er fint.

Exercise3:

Meget bedre besvarelse, savner lidt at det var gjort på denne måde i alle opgaverne, men fint i har noget mere tekst og dokumentation her!

De opgaver i har lavet virker til at være ok, og der ligger kode + makefiles i jeres repo.

Jeg savner en del tekst, og nogle af opgaverne er jeg ikke helt sikker på er besvarede (eksempelvis nederst i Ex. 2), men det mest essentielle virker til at være forstået og jeg giver derfor en lidt haltende ok.

[1.png](#) (41,4 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:12

[1-2.png](#) (46,3 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:14

[2.png](#) (52,2 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:19

[scopelock.png](#) (28,9 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:23

[vector.png](#) (26,7 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:23

[3.png](#) (9,18 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:24

[4.png](#) (7,14 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:26

[4-1.png](#) (1,17 KB) Emil Christian Foged Klemmensen, 2019-03-05 12:26