Wiki »

Exercise 4 - Thread Syncronization I

Introduction

In this exercise we will work with thread synchronization mechanisms. We will re-use the printout Exercise and ensure that the printouts work properly using mutex. Then we will use a Vector class to get deeper understanding of using mutexes and semaphores. Finally we will create a ScopeLocker class using RAII idiom to ensure that locks are always relinquished.

Exercise 1 Precursor: Using the synchronization primitives

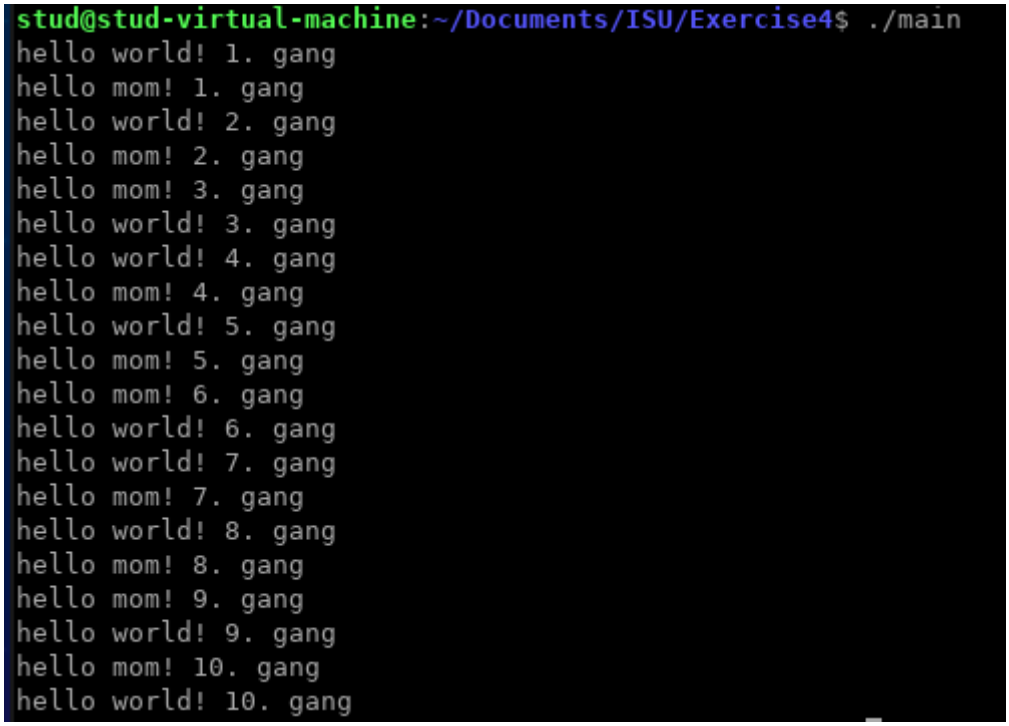
Exercise 1.1 Printout from two threads...

In this exercise we will create two threads in a program, and pass an ID to these, which they will print out every second along with the number of times the thread hans printed. When the threads has run through 10 print-outs they will terminate.

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

using namespace std;

pthread_mutex_t mut;

void* thrFunc(void *ptr)
{
    for( int i = 0; i<10; i++)
    {
        string *s = (string*) ptr;
        pthread_mutex_lock(&mut);
        cout << *s << " " << (i+1) << ". gang" << endl;
        pthread_mutex_unlock(&mut);
        sleep(1);
    }
    return nullptr;
}

int main(int argc, char* argv[])
{

    pthread_mutex_init(&mut, nullptr); // Here the mutex is initialized!
    pthread_t thr1;
    pthread_t thr2;
```

```
    string s1 = "hello world!";
    string s2 = "hello mom!";
    void *sptr;

    pthread_create(&thr1, nullptr, thrFunc, &s1);
    pthread_create(&thr2, nullptr, thrFunc, &s2);
    pthread_join(thr1, &sptr);
    pthread_join(thr2, &sptr);

    return 0;
}
```

We can observe the output on the picture below:



• What does it mean to create a critical section ?

It is a section of code, in which both threads are trying to access / edit variables etc.

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

We use the Mutex, which we initialize in main like this: pthread_mutex_init(&mut, nullptr);, and declare outside like this: pthread_mutex_t mut; . We use the lock: pthread_mutex_lock(&mut); and unlock pthread_mutex_unlock(&mut);
These are used within our for-loop, where we lock right before the print-out, and unlock immediately after.

Exercise 1.2 Mutexes & Semaphores

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

We would have to declare a semaphore instead of a mutex:

```
sem_t my_semaphore
```

We must then use sem_wait(&my_semaphore) instead of pthread_mutex_lock(&mtx).
And also sem_post(&my_semaphore) instead of pthread_mutex_unlock(&mtx).

Would it matter?

No not in this instance. Because it's initialized with the parameter 0, which only allows for a single thread to access data at the same time.

For each of the two there are 2 main characteristics that hold true. Specify these 2 for both:

Semaphores are unique in the way that 2 or more threads can be active at the same time.
Mutex' are unique in the way that only the thread who locked can unlock. Semaphores can allow multiple threads to release.

Exercise 2 Fixing vector

In this Exercise we will fix the Vector class, first by using mutex and secondly by using semaphores. Below you can see the code for Fixing vector, using a mutex:

```
#include <pthread.h>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <unistd.h>
#include "vector.hpp"

using namespace std;

Vector v1;
long delayt;                        // Variabel delayt initieres
pthread_mutex_t mut;

void * writer(void *ptr)
{
    bool err;
    while(1)
    {
        pthread_mutex_lock(&mut);
        err = v1.setAndTest((long)ptr);
        pthread_mutex_unlock(&mut);
        if (err == false)
        {
            cout << "FEJL I " << (long)ptr << endl;          // pthread_self()
        }
        usleep(delayt);         // Ændring til usleep(delayt)
    }
   return nullptr;
}

int main(int argc, char* argv[])
{
   int input;
    cout << "Indtast antal threads: " <<endl;
    cin >> input;

    cout << "Skriv delay i mikrosekunder" << endl;
    cin >> delayt;

    pthread_t wr[input];
    pthread_mutex_init(&mut, nullptr);

    for(int i=0;i<input;i++)
```

```
    {
        pthread_create(&wr[i], nullptr, writer, (void*)(intptr_t)i);
        cout << "Creating thread: " << i << endl;
    }
    for(int i=0;i<input;i++)
    {
       pthread_join(wr[i], nullptr);
    }
    return 0;
}
```

Below you can see the code for Fixing vector, using semapthore:

```
#include <pthread.h>
#include <iostream>
#include <stdlib.h>
#include <string>
#include <unistd.h>
#include "vector.hpp"
#include <semaphore.h>

using namespace std;

Vector v1;
long delayt;                        // Variabel delayt initieres

sem_t sm;

void * writer(void *ptr)
{
    bool err;
    while(1)
    {
        sem_wait(&sm);
        err = v1.setAndTest((long)ptr);
        sem_post(&sm);
        if (err == false)
        {
            cout << "FEJL I " << (long)ptr << endl;          // pthread_self()
        }
        usleep(delayt);         // Ændring til usleep(delayt)
    }
   return nullptr;
}

int main(int argc, char* argv[])
{
    int input;
    cout << "Indtast antal threads: " <<endl;
    cin >> input;

    cout << "Skriv delay i mikrosekunder" << endl;
    cin >> delayt;

    pthread_t wr[input];
    sem_init(&sm,0,1);
```

```
    sem_destroy(&sm);
    for(int i=0;i<input;i++)
    {
        pthread_create(&wr[i], nullptr, writer, (void*)(intptr_t)i);
        cout << "Creating thread: " << i << endl;
    }

    for(int i=0;i<input;i++)
    {
       pthread_join(wr[i], nullptr);
    }

    return 0;
}
```

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

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

We placed it in main - this means that we can easily can edit the critical section of the code and add more lines of code in the semaphore.

– Inside the class - as a member variable?

In this case the constructor initializes a semaphore, whenever a object is created. This way the user does not have to worry about the use of semaphores when mutithreading.
However this limits the Vector-class, to be unable to work without semaphors, even when multithreading is not desired, and you cannot initialize irrelevant semaphores.

– Outside the class - as a global variable, but solely used within the class?

This is not safe, and makes it possible for the variable to be changed.

– In your main cpp file used as a wrapper around calls to the vector class?

This is what we did. Again this makes it easy for us to edit the critical section of the code. Having considered these questions did however make us realize that it would be more user-friendly with in-class semaphore.
It would also make our code follow the principles of object-based programming better.

The code was made into an executable and run in the terminal. The results are seen below, and the conclusion is that it works:

```
stud@stud-virtual-machine:~/i3isu/lecture4/exercise2$ ./main
Indtast antal threads:
10
Skriv delay i mikrosekunder
1000
Creating thread: 0
Creating thread: 1
Creating thread: 2
Creating thread: 3
Creating thread: 4
Creating thread: 5
Creating thread: 6
Creating thread: 7
Creating thread: 8
Creating thread: 9
```

Exercise 3 Ensuring proper unlocking

The risk of unintentionally leaving a mutex or a semaphore in a locked state can be rectified by using the Scoped Locking idiom, which is a way to use the con- and destructor in the class to hold and release respectively.

The idea behind the Scoped Locking idiom2 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?

By reference to ensure that it's the correct Mutex being unlocked.

For this exercise the class scopLock was implemented and used in vector.hpp.

Below is the header-file scopLock.hpp:

```
#ifndef SCOPLOCK_HPP_
#define SCOPLOCK_HPP_
#include <pthread.h>

class scopLock
{
public:
    scopLock(pthread_mutex_t *mut)
    {
        mut_ = mut;
        pthread_mutex_lock(mut_);
    }

    ~scopLock()
    {
        pthread_mutex_unlock(mut_);
    }

private:
    pthread_mutex_t *mut_;

};

#endif
```

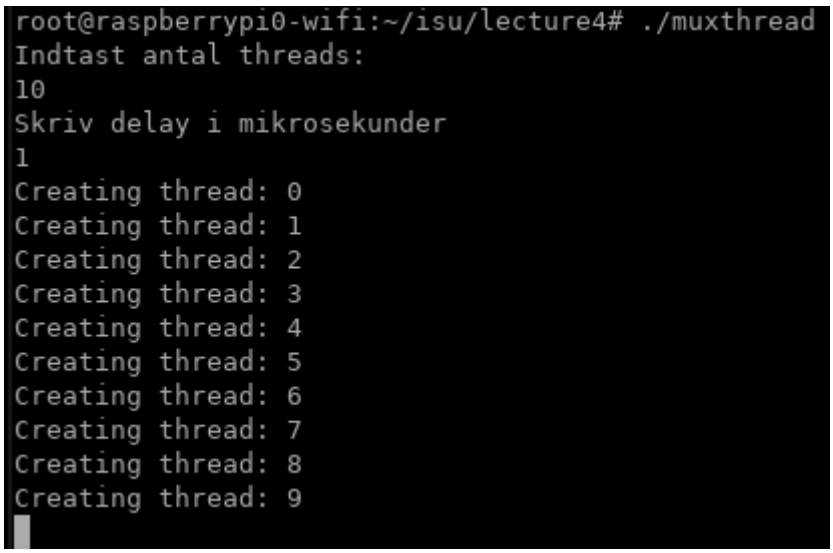
The vector.hpp file using the scopLock.hpp is shown below:

```
#ifndef VECTOR_HPP_
#define VECTOR_HPP_
#include "scopLock.hpp"         // scopLock.h must be included

//=========================================================
// 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)
      {
          pthread_mutex_init(&mut_, nullptr);          // mutex is initiated when v

          vector_ = new int[size_];
          set(0);
      }

   ~Vector()
      {
          delete[] vector_;
      }

   bool setAndTest(int n)
      {
          scopLock sl(&mut_);          // scopLock object, sl, is constructed using
          set(n);
          return test(n);
      }

private:
   void set(int n)
      {
         for(unsigned int i=0; i<size_; i++) vector_[i] = n;
      }

   bool test(int n)
      {
         for(unsigned int i=0; i<size_; i++) if(vector_[i] != n) return false;
         return true;
      }

   int*          vector_;
   unsigned int  size_;
   pthread_mutex_t mut_;
};

#endif
```

Exercise 4 On target

In this exercise we only have to scp the executable file to target and run it there. Below you can see a screendump of it running on target.



- **Tilføjet af Sahand Matten for 23 dage siden**

  Da der ikke igen er nogle Learnings goals har vi valgt at review ud fra vores egne learning goal

  ## Learning goals

  #Understanding the basics of thread synchronization.
  #Learning to utilize mutex and semaphores to ensure thread synchronization.
  #Understanding the Scoped Locking Idiom.
  #Being able to use thread synchronization on target.

  ## Understanding the basics of thread synchronization (/)

  Vi kan se basisk forståelse for synchronization.

  ## Learning to utilize mutex and semaphores to ensure thread synchronization (/)

  Vi kan se at der er en forståelse for mutex samt semaphore, der er dog nogle spørgsmål der ikke er blevet svaret.

  ## Understanding the Scoped Locking Idiom (/)

  Lidt tynd besvarelse, men overall fint nok. Lidt mere kød på besvarelser af spørgsmål.

  ## Being able to use thread synchronization on target (/)

  Vi kan se at den kører på target.

  ## Must haves:

  relevant files in repository ✔

  Makefile in repository ✔

  ## Konklusion

  En fin besvarelse, dog lidt tyndt i nogle af besvarelserne af spørgsmål især i opgave 2 og 3. Som lille sidekommentar kunne der være dependency generation i jeres makefile - Så er i sikre på at jeres filer bliver rigtigt compilet hvis i ændrer noget i .hpp filer. OK besvarelse 4/4.

Opg1_terminal.PNG (24,6 KB) Mikkel Mahler, 2019-02-27 11:24

host_ex2_terminal.png (17,2 KB) Andreas Ventzel, 2019-03-05 14:42

RPi_muxthread_terminal.png (9,5 KB) Andreas Ventzel, 2019-03-05 14:54