Thread synchronization I

Læringsmål for øvelsen:

1. Opnå kendskab til og implementere Mutexes & Semaphores
2. Implementere og teste for thread synchronization
3. Implementere Mutexes i en klasse (Scoped Locking idiom)
4. Teste applikaition med thread synchronization på target(Rpi)

Exercise 1 - Precursor: Using the synchronization primitives

Exercise 1.1 Printout from two threads...

Indledning

Write a program that creates two threads. When created, the threads must be passed an ID which they will print to stdout every second along with the number of times the thread has printed to stdout. When the threads have written to stdout 10 times each, they shall terminate. The main() function must wait for the two threads to terminate before continuing.

Old Solution
Løsning fra sidste øvelse, afsamme opgave indsættes.
Besvarelsen fra tidligere øvelse:
https://redweb.ece.au.dk/sw3isu-e21/projects/isu_gruppe_33/wiki/Exercise3

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

static void *threadFunc(void *arg)
{
    for (int i = 1; i <= 10; i++)
    {
        printf("Thread: %ld  - Thread ID %i  -  Print Nr: %i\n",pthread_self(), *(int*)arg ,i);
        sleep(1);
    }
    return arg;
}

int main()
{
    pthread_t t1, t2;
    int thread1, thread2;

    int ID1 = 1;
    int ID2 = 2;

    thread1 = pthread_create(&t1, NULL, threadFunc, &ID1); //default attributes NULL
    if (thread1!=0){ //return 0 on succes
        printf("pthread_create failed\n");
    }

    thread2 = pthread_create(&t2, NULL, threadFunc, &ID2); //default attributes NULL
    if (thread2!=0){ //return 0 on succes
        printf("pthread_create failed\n");
    }

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;

}
```

1. Extend the program by creating a global mutex
2. Remember to initialise the mutex
3. Create a critical section around your printout

Som vist med implementeringen og resultatet fra sidste øvelse kan der opstå problemer. Problemerne forsøges ny løst ved brug af en global mutex.

New Solution

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

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; //Global mutex init

static void *threadFunc(void *arg)
{
  int s;
  for (int i = 1; i <= 10; i++)
  {
    s = pthread_mutex_lock(&mtx);  //lock section
    if (s != 0)
        printf("Error: pthread_mutex_lock");

    printf("Thread: %ld  - Thread ID %i  -  Print Nr: %i\n", pthread_self(),
           *(int *)arg, i);

    s = pthread_mutex_unlock(&mtx);  //unlock section
    if (s != 0)
        printf("Error: pthread_mutex_unlock");

    sleep(1);
  }

  return arg;
}

int main()
{
  pthread_t t1, t2;
  int       thread1, thread2;

  int ID1 = 1;
  int ID2 = 2;

  thread1 =
      pthread_create(&t1, NULL, threadFunc, &ID1); // default attributes NULL
  if (thread1 != 0)
  { // return 0 on succes
    printf("pthread_create failed\n");
  }

  thread2 =
      pthread_create(&t2, NULL, threadFunc, &ID2); // default attributes NULL
  if (thread2 != 0)
  { // return 0 on succes
    printf("pthread_create failed\n");
  }

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
```

Result

```
tud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1$ ./bin/x86-64/prog
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 1
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 1
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 2
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 2
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 3
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 3
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 4
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 4
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 5
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 5
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 6
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 6
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 7
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 7
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 8
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 8
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 9
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 9
Thread: 140325959185984  - Thread ID 2  -  Print Nr: 10
Thread: 140325967578688  - Thread ID 1  -  Print Nr: 10
```

Questions

- **What does it mean to create a critical section ?**
  Man locker mutex, inden sektionen hvor en shared variabel ændres eller der udskrives et resultat af en shared variabel mm.. Når operationerne er færdige unlockes mutex igen så en ny thread kan lock og udføre sine operationer uden en tredje thread ændrer i en sharedd variabel på samme tid.
- **Which methods are to be used and where exactly do you place them?**
  For mutex benyttes lock/unlock så der kan arbejdes på en shared variabel uden en anden thread kan ændre variablen samtidig. Placeringen af lock/unlock er vist her:

  ```
  static void *threadFunc(void *arg)
  {
    int s;
    for (int i = 1; i <= 10; i++)
    {
      s = pthread_mutex_lock(&mtx);  //lock section
      if (s != 0)
          printf("Error: pthread_mutex_lock");

      printf("Thread: %ld  - Thread ID %i  -  Print Nr: %i\n", pthread_self(),
             *(int *)arg, i);

      s = pthread_mutex_unlock(&mtx);  //unlock section
      if (s != 0)
          printf("Error: pthread_mutex_unlock");

      sleep(1);
    }
  ```

Exercise 1.2 Mutexes & Semaphores

Questions

- **What would you need to change in order to use semaphores instead?**
  I stedet for lock/unlock af mutex, ændres der til take/release eller eventuelt wait/post, som også er muligt med semaphores.

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

Mutex:

1. Mutex ejes/kontrolleres af en thread af gangen. Den thread som "lock" skal også "unlock"
2. *"Mutexes are used to enforce MUTual EXclusion"* meaning:
   "A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource."
   https://www.techopedia.com/definition/25629/mutual-exclusion-mutex
3. Operationer:

- lock(mutex)
- unlcok(mutex)

Semaphores:

1. Semaphores ejes/kontrolleres **IKKE** af en thread af gangen. Alle kan derfor release.
2. Semaphores bruges til at signalere. Og kan på den måde bruges til MUTual EXclusion som beskrevet ovenfor.
3. Operationer:

- take(semaphores) - get(semaphores) , pend(semaphores , wait(semaphores)
- release(semaphores) - give(semaphores) , post(semaphores) , signale(semaphores)

Exercise 2 - Fixing vector

Indledning

Fix the Vector problem twice, once using a mutex and secondly using a semaphore

Solution - Mutex
Først løses problemet med Mutex ved:
Kodeudsnit, hele filen kan findes i Repository -> exercise4

```
 while (test)
  {
    s = pthread_mutex_lock(&mtx); /*Lock section*/
    if (s != 0)
        printf("Error: pthread_mutex_lock");

    test = shared.setAndTest(*(int *)arg);

    s = pthread_mutex_unlock(&mtx); /*Unlock section*/
    if (s != 0)
        printf("Error: pthread_mutex_unlock");

    usleep(wait);
  }
```

Result - Mutex
Resultatet ved 40 threads og 100ms wait time mellem setAndTest.

```
stud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1$ ./bin/x86-64/prog
Enter number of new Threads (0-100)
40
Enter number of ms between setAndTest
100
New thread: 1
New thread: 2
New thread: 3
New thread: 4
New thread: 5
New thread: 6
New thread: 7
New thread: 8
New thread: 9
New thread: 10
.
.
.
New thread: 38
New thread: 39
New thread: 40
```

Denne implementeringen laver ingen fejl hvor der i tidligere øvelse med 100ms wait time mellem setAndTest opstod mange.

Solution - Semaphore
En løsning ved brug af Semaphore:
Kodeudsnit, hele filen kan findes i Repository -> exercise4

```
while (test)
  {
    s = sem_wait(&s_); /*Take*/
    if (s != 0)
        printf("Error: pthread_mutex_lock");

    test = shared.setAndTest(*(int *)arg);

    s = sem_post(&s_); /*Release*/
    if (s != 0)
        printf("Error: pthread_mutex_unlock");

    usleep(wait);
  }
```

Result - Semaphore
Resultatet ved 20threads og 100ms wait time mellem setAndTest.

```
stud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1$ ./bin/x86-64/prog
Enter number of new Threads (0-100)
20
Enter number of ms between setAndTest
100
New thread: 1
New thread: 2
New thread: 3
New thread: 4
New thread: 5
.
.
.
New thread: 18
New thread: 19
New thread: 20
```

Denne implementeringen laver heller ingen fejl hvor der i tidligere øvelse med 100ms wait time mellem setAndTest opstod mange.

Questions

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

I dette program er resultatet ens. Implementeringen er simpel i begge tilfælde og der arbejdes kun med en funktion med en lock(m)/wait(s) og unlock(m)/post(s).

- **Where have you placed the mutex/semaphore and why and ponder what the consequences are for your particular design solution?**
  - Inside the class - as a member variable?
  - Outside the class - as a global variable, but solely used within the class?
  - **In your main cpp file used as a wrapper around calls to the vector class?**

I begge tilfælde er det implementeret i main.cpp filen og ikke i Vector klassen. Implementeringen i en klasse vil hjælpe med at huske release, men for dette program er der hurtigtsimpelt at implementere i main.cpp med 1 lock og 1 release. Implementeringen i en klasse ville være nemmere genanvendelig.

Exercise 3 - Ensuring proper unlocking

Indledning

Implement the class ScopedLocker and use it in class Vector to protect the resource. Verify that this improvement works. You only need to make it work with a mutex.

Solution
ScopedLocker udarbejdes således:

```
#ifndef ScopedLocker_HPP_
#define ScopedLocker_HPP_

#include<pthread.h>
//=======================================================
// Class: ScopedLocker
//=======================================================
class ScopedLocker
{
public:
    ScopedLocker(pthread_mutex_t &m):mtx(&m)
    {
        pthread_mutex_lock(mtx);
    }

   ~ScopedLocker()
      {
         pthread_mutex_unlock(mtx);
      }

private:
    pthread_mutex_t *mtx

   };

#endif
```

I klassens constructor bliver mutex mtx"lock" og i destructoren "unlcok".

Dette implementeres nu i Vector klassen:

```
bool setAndTest(int n, pthread_mutex_t &m)

      {

         ScopedLocker SLocker(m);

         set(n);

         return test(n);

      }
```

Result
Implementeringen testes nu, og resultatet ved at ændre til ScopedLocker implementeringen i Vector fremfor mutex implementeringen benyttet tidligere i main.cpp filen er :

```
stud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1$ ./bin/x86-64/prog
Enter number of new Threads (0-100)
20
```

```
Enter number of ms between setAndTest
100
New thread: 1
New thread: 2
New thread: 3
New thread: 4
New thread: 5
New thread: 6
New thread: 7
New thread: 8
New thread: 9
New thread: 10
New thread: 11
New thread: 12
New thread: 13
New thread: 14
New thread: 15
New thread: 16
New thread: 17
New thread: 18
New thread: 19
New thread: 20
```

Implementeringen synes at virke. Ved denne implementeringen vil destructoren unlock mutex, hvorfor det ikke kan forglemmes, som ved implementeringen tidligere.

Exercise 4 - On target

Finally recompile your solution for Exercise 3 for target and verify that it actually works here as well.

Applikationen bygges:

```
stud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1$ make ARCH=arm
mkdir -p build/arm
mkdir -p bin/arm
arm-rpizw-g++ -I. -o build/arm/ScopedLocker_mutex.o -c ScopedLocker_mutex.cpp
arm-rpizw-g++ -I. -o bin/arm/prog build/arm/ScopedLocker_mutex.o -lpthread -lrt
```

Applikationen kopieres til target:

```
stud@stud-virtual-machine:~/isu_work/exercise4_thread_sync1/bin/arm$ scp /home/stud/isu_work/exerc
ise4_thread_sync1/bin/arm/prog root@Rpi:
prog                                          100%   17KB   1.9MB/s   00:00
```

Applikation testes:

```
root@raspberrypi0-wifi:~# ./prog
Enter number of new Threads (0-100)
20
Enter number of ms between setAndTest
100
New thread: 1
New thread: 2
New thread: 3
New thread: 4
New thread: 5
New thread: 6
New thread: 7
New thread: 8
New thread: 9
New thread: 10
New thread: 11
New thread: 12
New thread: 13
New thread: 14
New thread: 15
New thread: 16
New thread: 17
New thread: 18
New thread: 19
New thread: 20
```

Hvor der i tidligere øvelse opstod mange fejl i applikationen på target, er der ved denne implementering ingen opståede fejl under test.