

Lecture 3 - Exercises

In this exercise we will try to fix the shared data problem we experienced in the exercises from Lecture 2 (Posix Threads). The problem will be fixed by implementing a mutex and a semaphore solution.

We will also write the ScopeLocker class that utilizes the RAII idiom to ensure that locks are always relinquished.

Exercise 1 - Using the synchronization primitives

In this exercise we will fix the former vector-problem. First by using mutex then by using a semaphore.

Implementation /Mutex

First we implement the Mutex:

This is done by first initializing mutex:

```
//Int mutex
pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;
```

Then we implement the lock and unlock. This is done in the writers for-loop:

```
void* writer(void* argument)
{
    data *dataPtr = static_cast<data *>(argument);
    for (;;)

        //Mutex Lock
        hest = pthread_mutex_lock(&mutx);

        bool test = (*dataPtr).vectorPtr_>setAndTest((*dataPtr).id_);
        if (!test)
        {
            cout << "Failed test at thread: " << (*dataPtr).id_ << endl;
        }
        else
        {
            cout << "PASSED" << endl;
        }

        //Mutex unlock
        hest = pthread_mutex_unlock(&mutx);
        //
}
```

This will make sure that only one instance of setAndTest at a time.

Implementation / Semaphores

The semaphore first gets declared:

```
//Declerate semaphore
static sem_t Sema;
```

Remember to #include <semaphore.h>

In the writer, we implement a wait and a post:

```
void* writer(void* argument)
{
    data *dataPtr = static_cast<data *>(argument);
    for (;;)
    {
        //Start semaphore
        sem_wait(&Sema);

        bool test = (*dataPtr).vectorPtr_>setAndTest((*dataPtr).id_);
        if (!test)
        {
            cout << "Failed test at thread: " << (*dataPtr).id_ << endl;
        }
        else
        {
            cout << "PASSED" << endl;
        }

        //End semaphore
        sem_post(&Sema);
    }
}
```

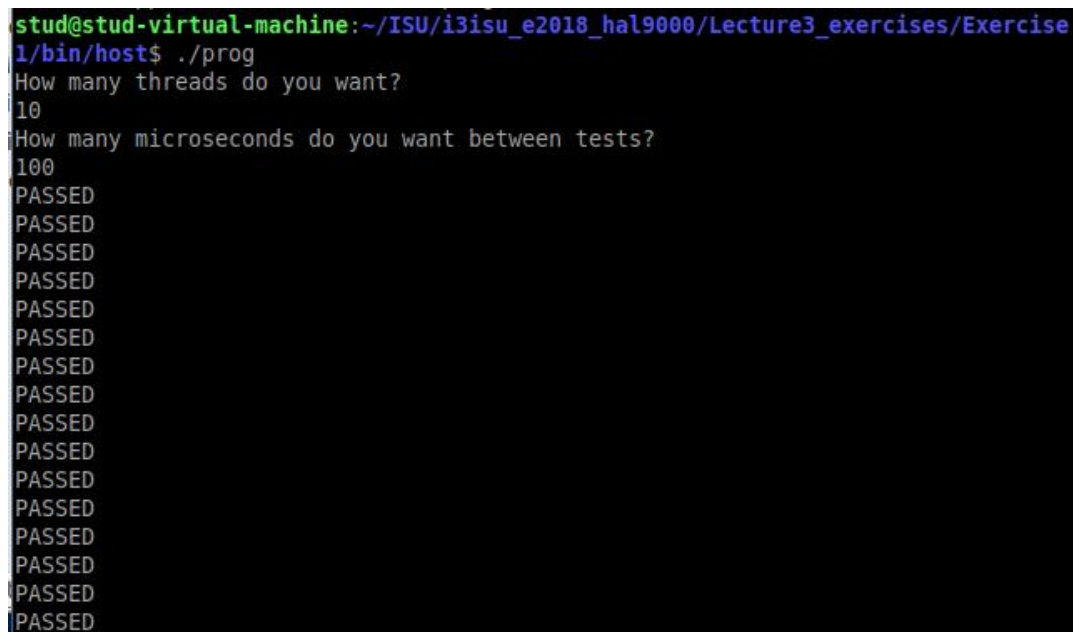
We also initialize the semaphore in main:

```
//Initialize the semaphore
sem_init(&Sema, 0, 1);
```

Sem_init takes three inputs. The first is the name of the semaphore. The first number is 0, so the semaphore is shared between the threads of the process. The second number is 1 (the value), so that only one decrementation from wait is needed to unlock it.

Test /Mutex

We use our makefile to make an executable and we test it:



```
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
1/bin/host$ ./prog
How many threads do you want?
10
How many microseconds do you want between tests?
100
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
```

All instances get 'PASSED', even with almost no sleep-time. Compared to the code from hand-in 2, where even two threads with a second of sleep-time gave an output of almost 50% failures, a mutex gives a way more secure process.

Test /Semaphores

Same as with mutex:

```
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
1 Semaphore/bin/host$ ./prog
How many threads do you want?
2
How many microseconds do you want between tests?
100000
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
PASSED
```

Discussion

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

In this case it doesn't matter, since there's only one resource being accessed.

A semaphore is more efficient/fast than a mutex, but when there's no need to lock multiple resources, these benefits are invalidated, which is the case for this instance.

In general the mutex is far more simplistic and safe to use, which is why one should only use a semaphore when "needed".

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

Both the mutex and the semaphore is placed in a never ending for-loop. Another option would be for it to be included in the classes; however this is not recommended, since only the given class would be able to access the mutex/semaphore.

However, one could argue that placing the mutex/semaphore in the main would be more beneficial, since the main is a thread itself.

Exercise 2 - Mutexes & Semaphores

We have now used both mutexes and semaphores.

For each of the two, specify the two main characteristics that hold true.

Mutexes

- Only one thread can access the data at once.
- The only thread able to unlock the mutex, is the one which locked it.

Semaphores

- A predefined number of thread can access the data from a semaphore.
- It is not necessarily the same thread that locked the semaphore, that have to unlock it.

Exercise 3 - Ensuring proper unlocking

The method used in exercise 1 has the disadvantage that the programmer isn't forced to release the mutex or semaphore after updating the shared data. This may result in a mutex or semaphore left in a locked state. To prevent this, a class ScopedLocker is created and passed a mutex on construction.

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

A mutex should always be passed by reference, since the mutex exist a place in memory. So when it comes time to unlock the mutex you need to make sure it's the original and not a copy.

Implementation

The class ScopedLocker takes a mutex object in its constructor and holds it until its destruction:

```
class ScopedLocker
{
public:
    ScopedLocker(pthread_mutex_t *mutex)
    {
        mutex_ = mutex;
        pthread_mutex_lock(mutex_);
    }
    ~ScopedLocker()
    {
        pthread_mutex_unlock(mutex_);
    }
private:
    pthread_mutex_t *mutex_;
};
```

The class is implemented in the class Vector (in Vector.hpp). Here it protects the setAndTest-function:

```
bool setAndTest(int n)
{
    //call constructor
    ScopedLocker lockedScope(&mutex_);
    set(n);
    return test(n);
    //auto-destructor when leaving scope
}
```

Also, the mutex is initiated when the Vector is initiated:

```
//init mutex
mutex_ = PTHREAD_MUTEX_INITIALIZER;
```

Test

```
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
3/bin/host$ ./prog
How many threads do you want?
2
How many microseconds do you want between tests?
50000
Failed test at thread: 1
PASSED
PASSED
PASSED
PASSED
PASSED
^Z
[2]+  Stopped                  ./prog
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
3/bin/host$
```

Exercise 4 - On target

Test

The executable made in exercise 3 is copied to the Raspberry Pi

```
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
3/bin/target$ scp prog root@10.9.8.2:/ISU/
prog                                100% 23KB 1.0MB/s 00:00
stud@stud-virtual-machine:~/ISU/i3isu_e2018_hal9000/Lecture3_exercises/Exercise
```

Here it is tested:

[illegible]

Discussion

The executable seems to work just as well on the target as on the host.

Filer				
Mutex_test.png	29,1 KB	2018-09-24		Brian Nymann
seme_test.png	32,3 KB	2018-09-24		Brian Nymann
mutex2_test.png	32,6 KB	2018-09-24		Brian Nymann
scp_prog.png	15,6 KB	2018-09-24		Brian Nymann
test_target.png	11,4 KB	2018-09-24		Brian Nymann