Søren Hansen <sha@ase.au.dk>
V1.6

Thread synchronization I

Introduction

In this exercise you will get some routine using thread synchronization mechanisms. First, you will rectify the shared data problem you experienced in Exercise *Posix Threads*, by creating a mutex and a semaphore solution. Then, you will write the `ScopeLocker` class that utilizes the RAII idiom to ensure that locks are always relinquished.

Prerequisites

In order to complete this exercise, you must:

- have completed Exercise *Posix Threads*

The problem in Exercises *Sharing data between threads* and *Sharing a Vector class between threads* from *Posix Threads* is that all threads share and utilize a resource and that resource is not protected. This is illustrated in the fact that a thread could *not* necessarily complete its read or write operation uninterrupted. For write operations the consequence could be inconsistent data in the shared resource, whereas a reader operation could return inconsistent data due to either an "in-between update" or a write being interrupted as mentioned above resulting in an error.

This problem can be rectified using a `mutex`/`semaphore`.

Exercise 1 Using the synchronization primitives

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

Questions to answer:

- 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 (*do note that the answer to the below questions do actullay require some thought!*) ?
  - 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?

Exercise 2 Mutexes & Semaphores

At this point you have used both `mutexes` and `semaphores` and you have been introduced to their merits.

For each of the two there are **2** main characteristics that hold true. Specify these **2** for both[1].

[1] Its **NOT** an explanation but merely a short statement where their properties are described. Whether this is a single statement or 2 points for each is up to you.

Exercise 3 Ensuring proper unlocking

The method for data protection in Exercise 1 has one problem namely that the programmer is not *forced* to release the `mutex/semaphore` after he updates the shared data. This scenario poses a risc since a `mutex` or a `semaphore` can unintentionally be left in a locked state. This can be rectified by using the *Scoped Locking idiom*.

The idea behind the *Scoped Locking idiom*[2] 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?) on construction. The `ScopedLocker` *takes* the `mutex` object in its constructor and *holds* it until its destruction - thus, it holds the `mutex` as long as it is in scope.

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`.

Exercise 4 On target

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

[2] This is a specialization of the *RAII - Resource Acquisition Is Initialization idiom.* This idiom is extremely simple but one of the most important you will learn, which is why it will be the focal point of a later lecture.