Thread synchronization I

shan@iha.dk
V1.0

Introduction

In this exercise you will get some routine in using thread synchronization mechanisms. First, you will rectify the shared data problem you experienced in Exercise *Posix Threads*, using a mutex or a semaphore. Then, you will create a generic (template) class that ensures atomic access to the data it protects.

Prerequisites

In order to complete this exercise, you must:

- have completed Exercise *Posix Threads*

Goal

- To make you able to differentiate between different kinds of a mutex and semaphores and decide which one is the right one for synchronization issues
- To give you routine in defining and using thread synchronization mechanisms under Linux
- To make you understand how a template class can automate thread synchronization issues

---

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 necessarily complete its read or write operation uninterrupted. When a thread was interrupted in the read or write of the shared resource, the shared resource could be left in an inconsistent state and an error would be reported. This problem can be rectified using a mutex/semaphore.

Exercise 1 Mutex and/or Semaphore

Does it matter, in both scenarios described above, which of the two is used? Present your arguments.

Exercise 2 Using the synchronization primitive

Fix the problem for both scenarios. Verify that your solution works for the second scenario[1].

Exercise 3 Ensuring proper unlocking

The method for data protection in Exercise 2 has one problem. The programmer is not *forced* to release the mutex/semaphore after he updates the shared data. Using the *Scoped Locking idiom* can enforce this.

[1] If you experienced problems with the first scenario verify it as well otherwise just fix it.

The idea behind the *Scoped Locking idiom*[1] is that you create a class `ScopedLocker` which is passed a `Mutex` 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.

Exercise 4 On target

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

[1]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