ENCE360 Lab 7: Synchronisation

Objectives

The goal of this lab is to have you write code that uses mutexes and semaphores to coordinate multiple threads and processes.

This lab should:

1. Introduce you to using mutexes in code to prevent data races.
2. Get you using semaphores in code to create thread-safe data structures.

Preparation

Download and extract your Lab 7 files from `lab7Synchronisation.zip` on Learn. This contains the source files `mutex.c` and `semaphore.c`.

Once you've completed the tasks below, make sure you also complete the quiz on the quiz server.

Remember, because we're using the threading library, we have to link `pthread`. This time we've supplied a Makefile, so you can compile both programs by simply running `make`.

Program `mutex.c`

This program computes the sum of a large array using a single thread. Your job is to convert it into a multi-threaded program.

Start out by turning the call to the function `runSummation()` into a call to spawn a thread using `pthread_create()`, similar to last week. The thread you spawn should run `runSummation()`. Ignore the mutexes completely for this step; get it running with threads first.

After converting it to a multi-threaded program, run it several times and observe the output. You should notice the final answer it generates is different from before. What are the reasons for this?

Fix the issues by making sure all threads finish before we print the final total, and by protecting the shared variables with a mutex. Check that the fixed program returns the same result as the original single-threaded program.

Program `semaphore.c`

In this program, child threads communicate to the main thread using a semaphore-based `Channel`. Your task is to implement the operations `channel_read()` and `channel_write()`, and initialise the `Channel` correctly in `channel_init()`.

First read the `producer()` and `main()` functions, and understand how the `Channel` is being used to communicate between threads. Then try to implement the operation.

At any one point in time the channel is either full or empty; it has a queue length of one.

If the channel is empty:

- The read semaphore has a value of 0 (must wait to read).
- The write semaphore has a value of 1 (allowed to write immediately).

If the channel is full:

- The read semaphore has a value of 1 (allowed to read immediately).
- The write semaphore has a value of 0 (must wait to read).

The read and write operations are therefore symmetrical, and proceed like this:

1. Acquire access to read or write via `sem_wait()`.
2. Perform read or write activity.
3. Signal to opposite read or write that they can now proceed with `sem_post()`.

The channel should be initialised to an empty state. Comments give hints here, but make sure you understand what would happen if the semaphores were initialised differently.

Stretch Goal: Expand `semaphore.c`

If you want to go the extra mile, try expanding `Channel` so that it has a queue length of 10. The number of semaphores and mutexes can remain the same, but your operations and initialisation will need to change, as well as the data structure itself.