Søren Hansen <sha@ase.au.dk>
V1.4

Posix Threads

Introduction

In this exercise you will gain experience in creating and handling threads in Linux using Posix threads. You will also experience some of the problems involved in multiprogramming, particularly the shared data problem.

Prerequisites

Have access to Kubuntu on the VMware Golden Image and the *Raspberry Pi Zero W* target

---

This is a rather large exercise, but it serves to exhibit important fundamentals about threading and shared data in particular. Make sure to complete and understand it.

Exercise 1 Creating Posix Threads

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 (hint: Look up `pthread_join()`).

Listing 1.1: A possible output from running the program is

```
$ ./lab
Main: Creating threads
Main: Waiting for threads to finish
Hello #0 from thread 0
Hello #0 from thread 1
Hello #1 from thread 0
Hello #1 from thread 1
...
Hello #9 from thread 0
Hello #9 from thread 1
Thread 0 terminates
Thread 1 terminates
Main: Exiting
$
```

Questions to consider:

- What happens if function `main()` returns immediately after creating the threads? Why?
- The seemingly easy task of passing the ID to the thread may present a challenge; In your chosen solution what have you done? Have you used a pointer or a scalar?

Exercise 2 Sharing data between threads

Create a program that creates two threads, *incrementer* and *reader*. The two threads share an `unsigned integer` variable named `shared` which is initially 0. *incrementer* increments `shared` every second while *reader* reads it every second and outputs it to `stdout`.

Are there any problems in this program? Do you see any? Why (not)?

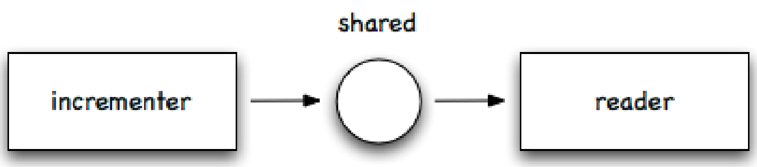

Figure 2.1: *incrementer* and *reader* thread utilizing the shared variable `shared`

Exercise 3 Sharing a Vector class between threads

The supplied class `Vector`[1] holds 10.000 elements per default, which at all times must have the same value. `Vector::setAndTest()` sets the value of all the elements and then immediately checks that the `Vector` object is consistent (all elements hold the expected value).

Create a thread function *writer* that uses `Vector::setAndTest()` to set and test the value of a shared `Vector` object. Then create a `main()` function that creates a user-defined number of *writer* threads (between 1 and 100), each with their own unique ID. Let each *writer* set and test the shared `Vector` object to its ID *every second*. If a *writer* detects an inconsistency in the shared `Vector` object (i.e. `setAndTest()` returns `false`), it should write an error message.

Run the program with different number of threads. Do your writers detect any problems? Are there any problems in this program? Do you see them? Why do you (not) see them?

Exercise 4 Tweaking parameters

Modify your program from exercise 3 so that the writers loop time is no longer one second but a user-defined number of microseconds. Experiment with the number of writers created and shorter loop time as well as the number of elements in the `Vector` - do you see any problems? Explain when and why they start to occur, and why you did not see them in exercise 3.

Exercise 5 Testing on target

Recompile the solution from exercise 3 and test it on target following the same line of thinking as in exercise 4. Compare your findings with those in that of exercise 4.

Are the parameters that you found to present problems on the host the same that yield problems on the target?

Why do you experience what you do?

[1]You will find the class in the file `Vector.hpp`, in the same place you found this document.

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING