CS4023 Week11 Lab Exercise

Lab Objective: In this week's lab we will look at how threads are implemented in Linux. This is an alternative to the OpenMP library from the last lab but can be used in more diverse situations. However, the price for this flexibility and generality is that there is more of an onus on us now to manage the flexibility. In our very simplified case, we will just look at firing off a collection of threads that print to the screen and we will look at what happens if we don't synchronise this output. We will then fix this using a mutex.

Threads, while very appealing, can disintegrate into chaos very quickly.

Summary

1 Create a directory for this week's work and download the sample program from Sulis.

2 Compile the file and observe it running.

3 Add a critical section to avoid the race condition in the code.

In Detail

1 You can create this week's lab directory with

```
mkdir ~/cs4023/labs/week11
```

Download the files threads.c, out_50.txt and out_50_correct.txt from Sulis and copy them into this week's folder.

2 To create a proper executable this week you will need to link your code with the pthread library, which is Linux's implementation of the POSIX threads standard. Here's how I compiled mine and you should do the same:

```
gcc threads.c -o threads -lpthread
```

This single command actually triggers a separate compilation step (of threads.c) followed by linking together the resulting "object" code (in threads.o) with pthread (the threads library) into an executable program that we're calling 'threads'.

Take a look at the code in threads.c for a few minutes. The gist of it is that *num_threads* threads are created, each of which are going to call the function *PrintHello()* with their respective thread number. The exact value of *num_threads* can be read as the first command-line argument, **otherwise** it defaults to DEFAULT_NUM_THREADS.

In the main() function a thread is created with the function call *pthread_create()*. When the thread is created, we need to tell it what function to execute and we pass the thread number to that function. Note how we check the **result** of the function call, printing an error message if it errors. Always important.

All that happens in the thread is a call to *PrintHello()* which will print out two similar lines:

```
Hello World! It's me, thread #45!
  Part 2 of the message from thread #45!
```

and then it exits the thread. Simple! You can run the program just by giving the command

```
./threads 50
```

I have provided you with sample output that I made earlier in the *out_50.txt* file. Looking at that, you can begin to see the trouble that unsynchronised threads cause. The first three threads are created initially by the main program and then thread 0 gets a go on the CPU and prints its 2-line message. But then look! It is thread 2 that goes next and it isn't even allowed finish it's two lines before it gets interrupted by thread 1. If this was a web server where potentially hundreds of threads were writing to a single file, this would be chaotic. We want to bring some order to this process.

3 The solution we will adopt is to treat the two *printf* statements in *PrintHello()* as a ***critical section*** and ensure that only one thread can "access" these lines of code at a time.

The easiest way to achieve this in Linux is with a *mutex*, a lock that only one thread may hold at any one time. A mutex lock variable is declared with:

```
    pthread_mutex_t lock;
```

It's name is lock and to initialise this mutex variable we use the function

```
    int pthread_mutex_init(&lock, NULL)
```

All threads can then request the lock with

```
    pthread_mutex_lock(&lock);
```

but it will be given to **only one** of the requesting threads at a time, leaving the others waiting.

After finishing, you can release the lock for another thread to use it with

```
    pthread_mutex_unlock(&lock);
```

It is always a good idea and a sign of a hygienic programmer to clean up after you by giving back and destroying the lock after use. This can be done with

```
    pthread_mutex_destroy(&lock);
```

See here for further information on the thread synchronisation problem and mutexes.

This is the final lab! Best of luck with the final exam – Shane, Jim & Josh.

- Submit the following files: threads.c and your executable.
- Submission is due next week: **Thursday 1st December 2022 at 18:00** (7 Days)