BCIT

COMP 4736 **Lab 07: Pthreads Sychronization (Part A)**

In this lab, you will learn the basics of multi-threaded programming, and synchronizing multiple threads using locks and condition variables. You will use the `pthreads` (POSIX threads) API in this assignment.

Before You Begin

Please familiarize yourself with the `pthreads` API thoroughly. Many helpful tutorials and sample programs are available online. Practice writing simple programs with multiple threads, using locks and condition variables for synchronization across threads. Remember that you must include the header file `<pthread.h>` and compile code using the **`-pthread`** flag when using this API.

Warm-up Exercises

You may write the following simple programs before you get started with this lab.

1. Write a program that has a counter as a global variable. Spawn 10 threads in the program, and let each thread increment the counter 1000 times in a loop. Print the final value of the counter after all the threads finish—the expected value of the counter is 10000. Run this program first without using locking across threads, and observe the incorrect updation of the counter due to race conditions (the final value will be slightly less than 10000). Next, use locks when accessing the shared counter and verify that the counter is now updated correctly.

2. Write a program where the main default thread spawns N threads. Thread i should print the message "I am thread i" to screen and exit. The main thread should wait for all N threads to finish, then print the message "I am the main thread", and exit.

3. Write a program where the main default thread spawns N threads. When started, thread i should sleep for a random interval between 1 and 10 seconds, print the message "I am thread i" to screen, and exit. Without any synchronization between the threads, the threads will print their messages in any order. Add suitable synchronization using condition variables such that the threads print their messages in the order 1, 2, ..., N. You may want to start with $N = 2$ and then move on to larger values of N.

4. Write a program with N threads. Thread i must print number i in a continuous loop. Without any synchronization between the threads, the threads will print their numbers in any order. Now, add synchronization to your code such that the numbers are printed in the order 1, 2, ..., N, 1, 2, ..., N, and so on. You may want to start with $N = 2$ and then move on to larger values of N.

Part A: Master-Worker Thread Pool

In this part of the lab, you will implement a simple master and worker thread pool, a pattern that occurs in many real-life applications. The master threads produce numbers continuously and place them in a buffer, and worker threads will consume them, i.e., print these numbers to the screen. This simple program is an example of a master-worker thread pool pattern that is commonly found

in several real-life application servers. For example, in the commonly used multi-threaded architecture for web servers, the server has one or more master threads and a pool of worker threads. When a new connection arrives from a web client, the master accepts the request and hands it over to one of the workers. The worker then reads the web request from the network socket and writes a response back to the client. Your simple master-worker program is similar in structure to such applications, albeit with much simpler request processing logic at the application layer.

You are given a skeleton program `master-worker-skeleton.c`. This program takes 4 command line arguments: how many numbers to "produce" (N), the maximum size of the buffer in which the produced numbers should be stored (B), the number of worker threads to consume these numbers (W), and the number of master threads to produce numbers (M). The skeleton code spawns M master threads, that produce the specified number of integers from 0 to $N - 1$ into a shared buffer array. The main program waits for these threads to join, and then terminates. The skeleton code given to you does not have any logic for synchronization.

You must write your solution in the file `master-worker.c`. You must modify this skeleton code in the following ways. You must add code to spawn the required number of worker threads, and write the function to be run by these threads. This function will remove/consume items from the shared buffer and print them to screen. Further, you must add logic to correctly synchronize the producer and consumer threads in such a way that every number is produced and consumed exactly once. Further, producers must not try to produce when the buffer is full, and consumers should not consume from an empty buffer. While you need to ensure that all W workers are involved in consuming the integers, it is not necessary to ensure perfect load balancing between the workers. Once all N integers (from 0 to $N - 1$) have been produced and consumed, all threads must exit. The main thread must call `pthread_join` on the master and worker threads, and terminate itself once all threads have joined. Your solution must only use `pthreads` condition variables for waiting and signaling: busy waiting is not allowed.

Your code can be compiled as shown below.

```
gcc master-worker.c -o master-worker -pthread -Wall
```

If your code is written correctly, every integer from 0 to $N - 1$ will be produced exactly once by the master producer thread, and consumed exactly once by the worker consumer threads. We have provided you with a simple testing script (`test-master-worker.sh` which invokes the script `check.awk`) that checks this above invariant on the output produced by your program. The script relies on the two print functions that must be invoked by the producer and consumer threads: the master thread must call the function `print_produced` when it produces an integer into the buffer, and the worker threads must call the function `print_consumed` when it removes an integer from the buffer to consume. You must invoke these functions suitably in your solution. Please do not modify these print functions, as their output will be parsed by the testing script.

The testing script can be invoked as shown below.

```
./test-master-worker N B W M
```

where N, B, W, and M are number of items to be produced, size of buffer, number of workers and number of masters, respectively. Note that the script will perform the above compilation for you.

Please ensure that you test your case carefully, as tricky race conditions can pop up unexpectedly. You must test with up to a few million items produced, and with a few hundreds of master/worker threads. Test for various corner cases like a single master or a single worker thread or for a very small buffer size as well. Also test for cases when the number of items produced is not a multiple of the buffer size or the number of master/worker threads, as such cases can uncover some tricky bugs. Increasing the number of threads to large values beyond a few hundred will cause your system to slow down considerably, so exercise caution.

Submission Instructions
You must submit a single code file as `master-worker.c`.

— End of Lab 07 —