CS2106 Operating Systems

Lab 3 – Threads

1. INTRODUCTION

In this lab we will revise inter-process communications using pipes. We will also be looking at threads.

Although memory usage in threads is much lighter than processes, they're equivalent in terms of scheduling. In fact, LINUX schedules only threads and not processes. Any process that does not spawn child threads is viewed to be a process with a single thread.

This lab should be done on a machine or virtual machine running Ubuntu 16.04. The correctness of this lab is not guaranteed on any other OS, and you are responsible for any issues arising from using any OS other than Ubuntu 16.04.

2. TEAM FORMATION

Please work in teams of 2 or 3. Single-person submissions are not accepted.

3. DEADLINE AND SUBMISSION INSTRUCTIONS

Decide who amongst you is the team leader. Name your report <student number>.docx or <student number>.pdf, where <student number> is the Student Number of the team leader as shown on his or her student card. If your team leader's student ID is a0344567b, then the report should be called a0344567b.docx, etc.

Please ensure that the names and student numbers of every team member is included in the report.

Please upload your report to your group's folder for Lab 3 on IVLE by 2359 hrs on Sunday, 11 March 2018.

4. PROCESS CREATION CHALLENGE

We will now create a program that generates 16384 random integers, and return the number of prime numbers amongst the integers. Our strategy would be to split the list into two, with the parent counting prime numbers in the 0 to 8191 sublist, and the child counting in the 8192 to 16383 sublist. The parent finally prints out the number of prime numbers in the array.

Use the skeleton given below to start with. Call your program lab3p1.c. Note you must compile your program with "gcc lab3p1.c –lm –o lab3p1". The "-lm" is important as we need to bring in the math library where the sqrt function resides.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define NUMELTS 16384

// IMPORTANT: Compile using "gcc lab3p1.c .lm -o lab3p1".
// The "-lm" is important as it brings in the Math library.
```

```
// Implements the naive primality test.
// Returns TRUE if n is a prime number
int prime(int n)
{
    int ret=1, i;

    for(i=2; i<=(int) sqrt(n) && ret; i++)
        ret=n % i;

    return ret;
}

int main()
{
    int data[NUMELTS];

    // Declare other variables here.

    // Create the random number list.
    srand(time(NULL));

    for(i=0; i<NUMELTS; i++)
        data[i]=(int) (((double) rand() / (double) RAND_MAX) * 10000);

    // Now create a parent and child process.
            //PARENT:
                // Check the 0 to 8191 sub-list
                // Then wait for the prime number count from the child.
                // Parent should then print out the number of primes
    // found by it, number of primes found by the child,
                // And the total number of primes found.
           // CHILD:
                // Check the 8192 to 16383 sub-list.
                // Send # of primes found to the parent.
}
```

Question 1. (5 marks)

Cut and paste your completed code into your answer book.

5. INTRODUCTION TO POSIX THREADS

POSIX threads (henceforth called "pthreads") is an API specification for threads on Unix based systems. Pthreads packages are available for non-Unix systems as well. The basic thread creation, joining and destruction calls are:

Call	Description
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg)	Creates a new thread. Returns 0 if successful. Arguments: thread - A data structure which will contain information about the created thread. attr - Thread attributes. Can be NULL. start_routine - Pointer to the thread's starting function. Must be declared as void *fun(void *) arg - Argument passed to the starting routine.
void pthread_exit(void *value_ptr)	Exits from a thread. The value in value_ptr is passed to another thread that joins with this exiting thread.

void pthread_join(pthread_t thread, void **value_ptr)	Suspends execution until the thread specified by "thread" completes execution. If value_ptr is not NULL, it will point to a location containing the value passed by "thread" when it exits using pthread_exit. pthread_join returns 0 if successful.

You need to `#include <pthread.h>` to use these. We will now look at an example of how to use threads. Type out the program below on Ubuntu and call it lab3p2.c. Compile it using "gcc lab3p2.c -o lab3p2", and execute using "./lab3p2". Run this program several times (at least 8-10 times) and observe the output.

```
#include <stdio.h>
#include <pthread.h>

// Global variable.
int ctr=0;
pthread_t thread[10];

void *child(void *t)
{
      // Print out the parameter passed in, and the current value of ctr.
      printf("I am child %d. Ctr=%d\n", t, ctr);
      // Then increment ctr
      ctr++;
      pthread_exit(NULL);
}

int main()
{
```

```
	int i;

	// Initialize ctr
	ctr=0;

	// Create the threads
	for(i=0; i<10; i++)
		pthread_create(&thread[i], NULL, child, (void *) i);

	// And print out ctr
	printf("Value of ctr=%d\n", ctr);
	return 0;
}
```

Question 2 (2 marks)

Do the threads print out in order? I.e. does it go "I am child 1.", "I am child 2.", etc? Or are the thread outputs mixed up? In either case explain why.

Question 3 (2 marks)

Based on your observation on the values of ctr printed by each thread, do threads share memory or do they each have their own portions of memory? Explain your answer, with reference to ctr.

Question 4 (3 marks)

Are the values of ctr as printed out by the child threads correct? Explain why or why not.

Question 5 (2 marks)

The variable "i" in main is effectively the index number of the child thread. Explain why it must be cast to (void *) before passing to the child thread, and why the child thread can successfully print out "i" without recasting it back to an int.

You will notice that the "printf("Value of ctr=%d\n", ctr);" statement in main sometimes executes even before the last thread completes. We will now use "pthread_join" to ensure that all threads complete before this statement executes. To do this, add the following statement to just before the printf:

```
    for(i=0; i<10; i++)
      pthread_join(thread[i], null);
```

Our main would therefore now look like this, with the added lines in **bold.**

```
int main()
{

	...

	for(i=0; i<10; i++)
		pthread_join(thread[i], NULL);
	// And print out ctr
	printf("Value of ctr=%d\n", ctr);
	return 0;
}
```

Compile and run the program, and verify that the printf no longer executes until the 10th thread completes. However you will notice that the threads still do no execute in order.

6. USING THREADS IN OUR WEB SERVER

Our old friend, the web server from Lab 2, is back again! Open up lab3p3.c and make the following modifications:

Question 6. **(3 marks)**

Modify startServer to spawn a new thread each time a new connection comes in. Note that you don't want to do a pthread_join with your new thread because this will cause your web server to wait for the new connection to end before receiving a new connection, defeating the purpose of multithreading.

Instead immediately after the pthread_create, call pthread_detach. Research into pthread_detach on your own.

Cut and paste your code to your answer book, and explain what pthread_detach does.

Question 7. (3 marks)

Create a new thread called "loggerThread" that:

- Polls the logReady variable continuously.

- If logReady is set:

    Write the contents of logBuffer to the logging file pointed to by logfptr.

    Call fflush(logfptr) to flush the buffer to ensure the contents are written to the file.

    Clear logReady.

Cut, paste and explain your code in your answer book.