UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS3010A
Operating Systems

Lab Concurrency

Semester 2, 2022

Instructors

COMS3010A Lecturer:
Branden Ingram branden.ingram@wits.ac.za

COMS3010A Tutors:
William Hill 2115261
Andrew Boyley 2090244
Oluwatimileyin Obagbuwa 2134111
Sheslin Naidoo 2094701
Derrin Naidoo 2127039
Phindulo Makhado 1832463
Mohammed Gathoo 2089236
Sedzani Ramathaga 2083519
Ryan Alexander 1827474
Talion Naidoo 1448771
Jonathan Nunes 2087190
Phoenix Krinsky 2233063
Christopher Walley 1846871
Alice Govender 1847313

Consultation Times

Questions should be firstly posted on Discord, for the Lecturer and the Tutors to answer. If further explanation is required consultation times can be organised.

1 Introduction

In today's lab you are going to be getting hands on with using threads and tackling the problems that concurrency brings with it. The examples that you will implement will hopefully help you understand the usage and need for our concurrency control techniques. You have been given some partially completed code which will aid you in completing aspects of this lab. Please download the "lab4Files" zip, extract and make sure to work within this folder.

2 Threads

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a

process. The multiple threads of a given process may be executed concurrently via multi-threading capabilities. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time.

In order to create threads we need to include a special library namely "pthread.h". To familirise yourself with its functionality and capabilities you can look here https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html.

2.1 Our First Thread

Create a file called "1Thread.c" as seen in the example below.

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

#include "common.h"
#include "common_threads.h"

int max;
volatile int counter = 0;

int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: main-first <loopcount>\n");
                exit(1);
        }
        max = atoi(argv[1]);

        return 0;
}
```

We are going to create a program which can count in half the time, i.e. we are going to get two threads to perform the same counting operation on the same shared variable. To do this we need to define two threads lets call them "p1" and "p2".

```
pthread_t p1, p2;
```

This simply defines two "pthread_t" objects to actually create our threads we use the following function:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

As you can see this function requires a few input parameters to be passed to it. The first you may be able to correctly guess is going to be "p1" or "p2" our "pthread_t" objects. Remember this is expecting a pointer therefore you will need to give the address of your objects. "attr" specifies which attributes are associated with the thread, for this example you simply use "NULL" which indicates the default parameters should be used. "void *(*start_routine)(void*)" may look daunting but it simply a reference to the function you want this thread to perform. Since a thread is just another sequence of instructions we can define this function to perform any task we want. For example in your "1Thread.c" define the following function to perform our counting

```
void *mythread(void *arg) {
        char *letter = arg;
        int i;
        printf("%s: begin [addr of i: %p]\n", letter, &i);
        for (i = 0; i < max; i++) {
                counter = counter + 1;
        }
        printf("%s: done\n", letter);
        return NULL;
}
```

The last input parameter for "pthread_create" is "void *arg" which are simply the additional arguments which you can pass to your thread. In this example you should pass a unique name to each created thread i.e. "A" and "B". So with this information you must now create two threads using p1 and p2 which both call the mythread function in your main method.

After creating a thread they essentially act as separate processes which undergo the same scheduling routines as the original process. Now all that remains is to wait for them to finish, this is handled similar to fork/join which you have previously studied. To wait for a particular thread to complete we call:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Here "pthread_t thread" is the thread of which we are waiting to complete and "void **value_ptr" is any return arguments we expect back from the exiting thread. Since "mythread" does not return anything we can simply use NULL for "void **value_ptr". Given this information on how join works call "pthread_join" for "p1" then "p2". Display the value of counter after joining both threads, we will use this to check if the program is working as intended.

Compile your code as you would an ordinary "c" file and run the generated output file. You will note an error when compiling this is due to the usage of the "pthread" libraries, make the required fix to how you compile.

2.2 Our First Argument :(

In this second example we will look to return values and send values to threads dynamically rather than using static global variables. One form of parallelism you might be familiar with is data parallelism and that is only possible if we are able transfer data between threads. For example if you are doing array addition you could use one thread to add the first half of two arrays while the other does the second by passing different indexes to each thread. In our example we are going to simply get a thread to perform a job and then return the result based off of the information we send to the thread. For this example the file is provided for you called "2ThreadsArgs.c". It is very important to read through and understand how it performs. In this program we look to perform some operation on two 2D points defined by a structure as you can see here:

```
typedef struct {
        int a;
        int b;
} myarg_t;
```

Unfortunately we can only pass a single object to the thread so how do we get around this issue? Well since you pass in a void pointer, it can point to anything, including a structure or a structure of structures ect. Complete the code for "2ThreadsArgs.c" by defining an additional struct called "data_t" which wraps the data of our two points "args1" and "args2". Finally complete the correct thread creation with the relevant input arguments, remember mythread expects a void pointer so the address of the data you are passing to the thread is required. The locations where you need to perform these tasks has been indicated in the file.

3 Concurrency

So for the most part we have looked at independently running threads where it does not really matter that other threads even exist. In these situation concurrency is not really even an issue as you can divide the job and simply wait until all threads are done. However, this is not always the case, there are situations where the processing of threads is very much dependant on the order in which events take place. For these cases we need to be smarter in how we operate.

3.1 The Humble SpinLock

In this example we will look into the most basic way to control operations that being with the concept of a spinlock. Create a file called "spin.c" and add in the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

volatile int done = 0;

void *child(void *arg) {
        printf("child: begin\n");
        sleep(5);
        printf("child: end\n");
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        printf("parent: begin\n");
        Pthread_create(&p, NULL, child, NULL);
        printf("parent: end\n");
        return 0;
}
```

Compile and run "spin.c" what do you notice? Now in order to ensure consistency you must implement a spinlock using the shared variable "done". This is achieved by setting the value of done in the child to a non zero value just before it is going to return. That is not all, inside the main method you must also add a while loop such that it infinitely loops while done is equal to zero. This loop must be added after the thread is created. In essence you now have the parent thread waiting until its' child changes the done variable. Compile and run again, now what can you notice?

There are multiple types of lock implementations such as TestAndSet, CompareAndSwap all of which seek to achive the same goal.

3.2 Establishing some control

The spin lock technique is effective although it is not very efficient as it required our main thread spinning in a loop. This means that this process is actually still executing, as mentioned in the lectures it would better to swap out this process while it was waiting.

For this next example you can use "spin_with_control.c" as the base program. Inside this file you will quite a few things have been added to our original "spin.c", however, it is still the same thing. All we want to achieve is for our parent thread to wait for the child to complete some desired task. Note that we use both a lock and a conditional variable in this example.

```
pthread_cond_t  c = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

The idea here is that we use "Cond_wait(&c, &m);" to wait for the child thread to signal us when they are finished. Compile and run this file what do you notice when you look at the sequence of events that occur? Is it what we expect to happen?

This is indeed what we want to see, however, this situation is not always going to occur. In other words this implementation is not safe. To demonstrate this we are going to force a situation where the child finishes first. To do this add the following line, on the first line inside the while loop in the main function:

```
sleep(2);
```

Now compile and run the same program what happens?

As you can see we have not got a perfect solution just yet. In order to correct this we can use the lock to surround another critical region forcing us to wait for the parent thread to go to sleep first. This can be achieved by using the following lines

```
Mutex_lock(&m);
Mutex_unlock(&m);
```

In order to determine which is the critical region you should think about which regions involve accessing shared variables.

4 Bugs!

As you can see even when dealing with one additional thread concurrency can be quite a difficult issue to handle. There are multiple different ways in which bugs can arise. For this final example we will investigate a deadlock. Including in the lab files is a file called "deadlock.c". Compile and run this file. Run it again does it complete? In this example we have two threads which compete to acquire and release two separate locks in tandem. It is your job to determine if it is thread safe or not, and if not then in which situation does it fail.

Academic Integrity

There is a zero-tolerance policy regarding plagiarism in the School. Refer to the General Undergraduate Course Outline for Computer Science for more information. Failure to adhere to this policy will have severe repercussions.

During assessments:

- You may not use any materials that aren't explicitly allowed, including the Internet and your own/other people's source code.
- You may not access anyone else's Sakai, Moodle or MSL account.
- You may not use any device other than the lab machines.
- You may not edit your submissions using any other device either inside or outside the designated venue.

Offenders will receive 0 for that component, may receive FCM for the course, and/or may be taken to the legal office.