CMPS 12M: Lab 4
Multithreading

Prof. Darrell Long

May 31, 2017

1 Introduction

With the advent of technology, processors with multiple cores are now widespread. The phone in front of you probably possesses two or more cores. Each of these cores is capable of executing instructions at "the same time" (technically, the instructions are executed in overlapping intervals of time). All the code you have written until now is termed `single-threaded` and has not made use of this enormous computing power. This lab introduces you to the basic idea of parallel computing (fancy word for multithreading), and how to write a simple C program which makes use of the C multithreading interface: `pthreads`.

Parallel Computing is the present and the future. Almost every industrial application implements multithreading to speed up execution of their programs. It is a growing field of computing and is as such far too broad a topic to cover in its entirety by this lab. Hence, we teach you the core fundamentals and otherwise refer you to the vast breadth of knowledge available freely online.

Parallel computing requires a rather different mindset for writing good code. It introduces several advantages over single-threaded code, however, it also introduces several disadvantages. Let's understand some terminology associated with multithreading and see what these advantages and disadvantages are.

2 Terminology

2.1 Process

A process is defined as a program which is currently under execution by a processor. It is made up of several objects, namely `instructions, resources and threads`. These collectively define the working of a process.

2.1.1 Instructions

These are the lines of code a processor executes. Remember, all a processor does is execute instructions sequentially, one after the other in a logical sequence. The C code we write gets converted into instructions the processor understands. In the case of the timeshare, and most modern laptops, these are `x86_64` (assembly) instructions.

2.1.2 Resources

The instructions in a process usually operate on some `resources`, either owned or shared by the process. These resources could be something as simple as a variable stored in memory or a complex graphics card attached to the PCIe Bus. As you will learn later, these resources need to be protected in order to ensure successful execution of a program.

2.1.3 Thread

A thread is the part of a process which actually executes the instructions. By default, each UNIX process spawns with a single thread and thereafter more threads can be spawned by the process if required. Each thread maintains its own stack as the stack keeps the state of the execution for each thread. Global variables and the heap are shared among different threads. This means that when one thread changes a global variable, the change will be reflected for all other threads in the process. In fact, one of the biggest problems associated with multithreading is the synchronicity of shared resources like global and dynamically allocated variables by different threads.

Another common problem associated with threads is the order of execution. At no point of time do we have a guarantee that one thread will execute before another. This scheduling of threads creates interesting problems for Computer Scientists to solve.

2.2 Context Switch

A context switch is defined as the concept of halting (or pausing) the execution of a certain thread (for whatever reason) by the operating system to execute some other thread.

There are a lot of threads active in an operating system at any given time, however, most computers only possess about four cores. This means that this computer can only execute four threads in concurrence. Hence, the operating system consistently needs to switch out threads and allow other threads to execute so that each thread is allowed to finish its task. A paused thread will be switched back into execution by the operating system after some time to finish its task.

2.3 Critical Section

A critical section of a process is the piece of code during which a shared resource is accessed. This section needs to be protected such that upon execution of this section, other parts of the code do not break. When a critical section is not properly protected, we get `race conditions`.

2.4 Race Condition

A race condition is a type of a nasty bug associated with multithreaded applications. It occurs when multiple threads enter a critical section of the code at "the same time".

Let's consider an example. We have a global variable `counter = 1` and two threads running in parallel: thread A and thread B. Both the threads want to increment the value of `counter` by 1 (Remember again that we do not know which thread will execute first). Suppose thread A runs first and reads in the value of `counter` as 1. Right before it increments this value, the operating system decides to perform a context switch and swaps outs thread A for thread B. Thread B comes in and reads the value of `counter` as 1. It goes on ahead to increment this value to 2 and writes it back to memory. Now, the operating system performs another context switch and this time it swaps out thread B for thread A. Thread A resumes execution from where it left off. The last step it had performed was reading in the value of `counter` and the value it read was 1. It now goes on ahead to increment this value by 1 (making it 2), and then writes this value back to memory. So even after two increments, the variable `counter` still holds the value 2 and not 3.

It is important to realize that the above case happens only 5% of the time. The other 95% of the time, the operating system will not context switch thread A out for thread B at such a critical point of time. Thus, we encounter bugs which do not replicate themselves on successive executions of the program. This is one of the main reasons why writing multithreaded code is tricky as you can successfully execute code thousands of times and still have a bug which has not surfaced yet.

Hence, we need a way to prevent race conditions. Perhaps a mechanism that enforces that only a single thread at any point of time is allowed to access a variable and any other thread which wants to access the same variable needs to wait until the first thread is done with it. This mechanism is what we call a `mutex`.

2.5 Mutex: Mutual Exclusion

A `mutex` is a tool by which we protect a critical section of code such that only a single thread is allowed to enter and execute the instructions of the critical section.

Protecting a critical section is straightforward. Any thread which wishes to enter the critical section needs to `acquire` a mutex. Once acquired, it is free to execute the instructions of the critical section. After it executes all the instructions, it `releases` the mutex so that other threads can acquire it. This concept works because we enforce that at any point of time, only a single

thread is able to acquire a mutex. If another thread tries to acquire the mutex, then that thread needs to wait until the first thread finishes execution.

A function which correctly employs a mutex to protect a shared resource is known as a `thread safe function`. Something interesting to note is that a mutex is itself a shared resource which is in turn used to protect other shared resources.

Consider some pseudo-code.

```
1  mutex m;
2
3  // Thread Safe. It correctly acquires and releases the mutex
       for the thread.
4  function someFunction():
5      acquire_mutex(m);
6
7      critical_section....begin
8      .
9      // Some important code here - world saving types
10     .
11     critical_section...end
12
13     release_mutex(m);
14
15 // NOT thread safe. It requires that the user acquire and
       release the mutex outside the function.
16 function anotherFunction():
17     critical_section....begin
18     .
19     // Some important code here - world saving types
20     .
21     critical_section...end
```

In the following code snippet, `someFunction` is thread safe as it does not assume that the programmer will correctly acquire and release the mutex for the execution of the critical section. On the other hand, `anotherFunction` is NOT thread safe as it requires that the programmer acquire and release the mutex themselves, before and after calling the function respectively.

There are cases when you would want to write code like `anotherFunction`, however, those cases are rare and in general you should write thread safe functions like `someFunction`.

Quite naturally, none of this is a problem if you only have a single thread running in the process and is again a reason why writing multithreaded code is hard to write. This concludes the terminology required for understanding basics of multithreading. Lets get to some threading!

3 A host of threads....watch out for the race condition!

The code you need to write for this lab is rather simple. You need to have a shared variable `counter` (you may make it a global variable) and you need to spawn `N` threads, each of which increments `counter` by 1 a thousand times through a thread-safe function. We enforce that you increment `counter` only once per the thread safe function call. So each thread will need to call this thread safe function a thousand times. Hence, for five threads, this function will be called an overall 5000 times.

Your program should be called `increment` and your source file must be called `increment.c`. You should accept a single flag `-n` which allows you to enter the number of threads you want to spawn. A skeleton of a basic multithreaded application is provided at the end of the PDF for reference.

1. It should have a thread safe function which acquires a mutex, increments `counter` by one and then releases the mutex (Try doing this without the mutex to see a race condition in action).
2. It should have a function which is the entry point for each thread that is spawned. This function should accept a variable which tells the thread how many times it needs to call the thread safe function (for this lab, this variable will always be 1000).
3. Inside main, you will spawn the required number of threads. Then the main thread (the thread which spawned all the other threads) will wait until all the other threads have finished execution. It should print only a single integer value followed by a newline and this value should end up being `N * 1000`.

3.1 Ehh....some help might be nice!

This might seem like a difficult lab (and it is), but all the code you need to write is about 75 lines of code. Here are some helpful things.

3.1.1 pthreads.h

This header files declares all the functions and type necessary to write multithreaded programs using the pthreads interface.

3.1.2 pthread_t

This variable type is used to hold a thread. Internally, it is a structure which contains a lot of fields, however, the pthread interface allows us to use this without having to learn how the internals work.

3.1.3 pthread_mutex_t

This variable type is used to hold a mutex. The usual operations we perform on a mutex include acquiring and releasing a mutex.

3.1.4 pthread_mutex_init

This function is used to initialize a mutex before it can be used. Usually, we make the mutex a global variable and then initialize it in `main()` before we use it.

3.1.5 pthread_mutex_destroy

This function is used to destroy a mutex once we are done using it. The function `pthread_mutex_init` makes some system calls and allocates memory on the heap for some fields of a mutex and hence we need to destroy the mutex once we are done using it lest we cause a memory leak.

3.1.6 pthread_mutex_lock

This function is used to acquire a mutex by a thread. Only one thread at a time can acquire a mutex.

3.1.7 pthread_mutex_unlock

This function is used to release a mutex which has been acquired by a thread. This function also "wakes up" any threads which have been waiting to acquire the mutex. In case more than a single thread is waiting, then one thread at random is selected and this thread is allowed to go on to acquire the mutex.

3.1.8 pthread_create

This function is used to spawn a new thread. Each thread gets its own stack and hence it can execute functions with its own local variables. Each thread needs a starting point to begin execution at, and this starting point is provided in `pthread_create` using a function pointer (Read the function pointers tutorial posted on Piazza).

3.1.9 pthread_join

This function is used to wait on a thread. In other words, this function blocks the execution of the waiting thread until the thread it is waiting for finishes execution.

4 Deliverables

For this lab, you should use `lab4` as your working directory:

4.1 README

Your README should clearly explain the functioning of your program and any odd design choices you might have made. We also require that you write the meaning of a `mutex` and a `race condition`. This definition should be what you understand about these terms and not any online definition. It should also not be the definition provided in this PDF.

4.2 Makefile

This should compile your program. It should support `make all`, `make increment`, and `make clean`. NOTE: You will need to link with the pthreads library. You do this by adding `-lpthread` at the end of your compile command.

4.3 increment.c

The C source which implements a multithreaded program and is successfully able to increment a shared variable without encountering race conditions.

5 Submission

You will submit *only* the following files to your git repository as explained in lab 0:

1. `lab4/README`
2. `lab4/Makefile`
3. `lab4/increment.c`

After pushing your code, use `git log` to find your commit ID and submit it using the Google form.

6 Skeleton

```c
int someSharedVariable;
pthread_mutex_t mutex;

struct threadArgs {
    int data;
}

void* threadFunc(void *myArgs)
{
    struct threadArgs *args = (struct threadArgs*)myArgs;
    int data = args->data;

    someWork();
}

void someWork()
{
    pthread_mutex_lock(&mutex);

    // Perform stuff on someSharedVariable

    pthread_mutex_unlock(&mutex);
}

int main(int argc, char *argv[])
{
    pthread_t thisThread;
    struct threadArgs args;

    // Init Mutex
    pthread_mutex_init(&mutex);

    // Set Arguments and create thread
    args.data = 10;
    pthread_create(&thisThread, NULL, threadFunc, &args);

    // Wait for thread to finish executing
    pthread_join(thisThread, NULL);

    // Destry Mutex
    pthread_mutex_destroy(&mutex);
}
```