**Operating Systems - Project**

**Overview**

We've now taken a look at threads which are used to allow shared memory between different executions. In this lab you will implement a **Work Queue** which draws on a thread pool to perform arbitrary work. This pattern shows up often in all manner of multithreaded applications. One example is a server handling arbitrary requests for webpages from various clients. Having a pool of threads present eliminates the need to construct and tear down threads and is ideal for many short multithreaded tasks.

We are covering condition variables and mutexes which are needed to control threads. Arbitrary shared memory is a dangerous thing. Pay close attention to your implementation. The framework in which the program is written is likely to exercise many possible interleavings but take care. Unlike in single-threaded execution, no amount of testing can reveal every concurrency bug.

The two most common concurrency bugs are race conditions and deadlocks. Be alert for race conditions where one access to a variable is not controlled to be before or after another access but instead allowed to occur at any relative time. These non-deterministic results are usually errors. Deadlocks are much less common and occur when two threads are trying acquire mutexes that the other holds. The threads will be blocked indefinitely.

Avoid busy-waiting when a thread has no task to accomplish. As an example fix, a thread can give up its spot in the scheduler by using the wait function of a condition variable. This is much better than spinning in a loop while endlessly checking a variable.

**Work Queue**

The work queue performs a number of tasks. These tasks are provided in order of priority. They are assumed to be independent and completable without a strict order. There are three tasks are performed (a, b, c). The work queue is made up of producers and consumers along with a main thread that manages them.

Limited queue is the data structure used to hold tasks as they are produced before they are consumed. It has a fixed size so all operations on it must halt when it fills up. It is provided to you in the skeleton fully functional. However, it is unprotected and in need of safe concurrency control to be used in a multithreaded environment.

The producers sleep for 2 seconds before doing anything to represent input delay. Don't change this. They then take a list of jobs from a file named work. This file can be modified to include any number of characters (a, b, c, s). Multiple producers should all read independently from the same file and produce the same schedule of work. We've included a few extra variants of the file to help you test your program. You can rename them to work or set the Java program to open them by name.

The producers take jobs (a, b, c) and place them on the queue when there is room. If a producer receives an (s) representing a significant delay in input, it sleeps for five seconds. Maintain this behavior in your implementation. When a producer has finished reading its file, it should alert the main thread. At this point whether you want to have producers terminate immediatly or by signal from the main thread is up to you.

The consumers take jobs off the queue and perform them. The jobs are modeled by varying lengths of sleep(). The consumers should do nothing if there is no work present (but not terminate or busy wait).

WorkQueue will take two arguments that tells it the number of threads to use for producers and consumers respectively. The runtime will launch the specified amount of producers which will each put the amount of work in the file onto the queue. The specified amount of consumers will all compete to do this work.

All consumers should eagerly complete tasks as they arrive. All producers should eagerly put maximum work in queue when it is not full, sleeping only when instructed by a delay. You should strive to write the program with the minimum amount of time spent excluding other threads from shared resources. Here is a simple example of WorkQueue:

jmevans@sam$ java WorkQueue 1 3  
Thread 1 Starting Job A (#1)  
Thread 1 Ending Job A (#1)  
Thread 3 Starting Job B (#2)  
Thread 2 Starting Job C (#3)  
Thread 3 Ending Job B (#2)  
Thread 2 Ending Job C (#3)  
3 Jobs Done

Implement proper multi-threaded program. All consumers and producers should be active at all times that there is some work to do.

When consumer threads have no work to do, they should be suspended waiting for more work. When all work is exhausted, the main thread should terminate them.