Nathaniel Basara
CECS 526
Mini-Project

Simple Muli-Thread programming without Synchronization

Design

Shared Variables:

- SharedValue – shared resource that each thread is trying to change and access
- lock – mutexlock used for synchronization and controlling access to shared variable
- barrier – barrier used to make every thread wait till all others are finished.

Main
Variables:

- integer-i – iterator for FOR loops
- threadsToMake – integer that stores the amount of threads requested in program arguments.
- Error – checks if there is an error while trying to create multiple threads
- arguementLength – holds the string length of the second arguement
- thread_array – an array of pthreads to launch to desired function
- thread_nums – stores the corresponding thread number to thread in thread_array

Process:

First thing in the main function is that we check if the correct number of arguments were given when the program was started. Two are required because the first is the program binary file, the second is the number of threads to make. Main will then determine if the second argument is indeed a positive integer. It will walk through each character in the second argument verifying it is a digit. If not the program will tell the user that the argument given did not match requirements.

After verifying the second argument is a digit we will convert its string representation and store as an integer in threads to make. We then use that number of threads initialize the size of the thread and integer array. As well as, initializing the number of threads our barrier is meant to wait for before letting all threads advance.

Main then begins the thread launching process by establishing the thread number to corresponding thread being sent off. We then create a thread using pthread_create and store it's return value in error. We send the thread to SimpleThread function with a void reference to the value stored in the thread_nums array. The reason for sending the thread number array instead of iterator itself is because the iterator will continue to change throughout the thread process creation. Therefore, it is required to create another location that stores the thread_num value.

After all threads have been created then thread will wait for each thread to finish before ending the program.

Simple Multi-Thread Programming without Synchronization
Arguments:

- Which – void type that holds the address number that stores the integer representation of thread number

Variables:

- num – iterator for FOR loop
- val – stores the shared value reading and then altering shared value
- thread_num – stores the integer representation of the which argument

Process:

First SimpleThread will initialize thread_num to the thread number of the driving thread. Then a 20 iteration loop will execute. Inside the loop the body the thread generates a random number. If that random is greater than half the random max value, the thread will sleep for 500 milliseconds. The thread then stores what it sees in SharedValue and prints the number and the value stored in val. The thread will then change the SharedValue by storing val + 1, thus ending the loop body.

Once the loop has ended the thread will look one last time at SharedValue and print one last time before the thread finishes the function.

Explanation:

When number of threads running is one. The output is similar to that if main was to call SimpleThread itself. If there is more than one thread running at a time, the output will not look like what we might expect. For instance, if there are two threads running the output will mostly consist of two different threads seeing the same value. Possibly a thread sees a value that was seen by another two iterations ago, and more than likely, threads will finish out of order and before another thread has started its 20^{th} iteration.

Simple Multi-Thread Programming with Proper Synchronization

Arguments:

- Which – void type that holds the address number that stores the integer representation of thread number

Variables:

- num – iterator for FOR loop
- val – stores the shared value reading and then altering shared value
- thread_num – stores the integer representation of the which argument

Process:

Prior to this iteration of SimpleThread there were two unused variables, lock and barrier. Lock is now used as a mutex lock. Only one thread has access to critical region at a time and a thread cannot move forward until the acquire the lock. This addition will now control access to SharedValue. Thus, every thread will see a unique number for each iteration and SharedValue will always increment by 1. After doing so the thread that has the lock will then drop the lock allowing every other thread an opportunity into the critical region.

After each threads completion of the loop is will reach the barrier. Once it reaches the barrier, it will wait until the barrier condition is met. The program barrier condition is initialized to the amount of threads. So, once all threads have finished they will access the SharedValue for the last time and print its value. Therefore, every thread sees the same final value.

Video:

https://youtu.be/iGU55QxA-dg