**Writeup:**

**Comments:**

**Code:**

**CS 4414 Operating Systems – Fall 2017**

**Homework 2: Synchronization**

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly.

1. **Problem Description:**

The objective of this assignment was to learn synchronization techniques and apply them to a basic problem. The program had to find the maximum value in a list of *N* numbers, which were given via an input file, by using *N/2* threads. In order to avoid synchronization problems, a barrier had to be written using POSIX semaphores.

1. **Approach:**

The program accepts a list of numbers, each on their own line, via standard in. The inputs are parsed into numbers using the *sscanf* method and the values are stored in a linked list. After all inputs have been read, the linked list is flattened into an array and the length of the array is stored as a value *N*. Each input list is assumed to have a length that is a power of two.

The maximum array value is found by running the method *find\_max.* This method generates *N/2* threads and has each thread run through *log(N)* rounds of comparisons. Each thread is given an *Argstruct* which contains a pointer to the array being operated on, the length of the array *N*, the index of the thread, and a pointer to a *Barrier* structure shared by all the threads. The threads are created in a for-loop running from 0 to *N/2*, and the index stored in the *Argstruct* is taken to be twice the value of the counter variable for a given thread. For example, the thread created at *i = 1* has index 2. The goal of index is to signify which spot in the array the thread is responsible for.

For each round, the threads compare the value stored at their own *index* in the array with a value stored at *index+offset*. The higher of the two values is stored in the array spot at *index*. Here, *offset* is initialized at 1 for round 1 and doubled for each subsequent round. Some threads are inactive in later rounds. For example, in round 2, thread 0 compares index 0 with index 2. Index 2 will not be compared with index 4, so the thread responsible for index 2 will be inactive in that round. Similarly, the thread responsible for index 4 will be inactive for round 3 since thread 0 will compare index 0 and index 4. A thread can determine that it should be inactive for a given round if its index is not evenly divisible by *2^r*, where *r* is the round number which starts at 1 and goes to *log(N)*. If a thread is active, it performs the comparison and stores the higher value in its slot in the array. Whether the thread is active or inactive, at the end of each round, it doubles its offset value and has the *Barrier* call wait.

The *Barrier* is implemented as a struct containing integers *N* and *count* as well as pointers to three binary semaphores: *waitq*, *mutex*, and *throttle*. It can be passed as an argument to two functions, *bar\_init(Barrier\* bar, int n)* and *bar\_wait(Barrier\* bar)*. Upon calling *bar\_init*, the Barrier is initialized with *count* set to 0 and *N* set to the integer argument of the function. The three semaphores are initialized using *sem\_init* from the POSIX semaphore library. *Waitq* and *throttle* are initialized with a value of 0 and *mutex* is initialized to 1 so that the first thread to access the barrier can access the section protected by *mutex*.

The goal of *bar\_wait­* is to force the first *N-1* processes calling it to block. The *N*th call of *bar\_wait* will result in all blocking processes to be resumed. This is done by keeping track of the number of processes who have called it in the variable *count*. First, the function waits on *mutex*, which protects against simultaneous accesses of the variable *count*. Next, *count* is incremented. If it is equal to *N*, *waitq* is signaled and *throttle* is waited on *N-1* times. This is done in a for-loop, and the loop will not progress at each iteration until the *wait* call on *throttle* completes. Because the semaphores are being used only as binary semaphores, without calling *wait* on *throttle*, there is no way to ensure that the signaled process actually resumed. After the loop completes, *count* is reset to 0 and *mutex* is signaled using *sem\_post*.

If *count* is not yet equal to *N*, then *mutex* is signaled and the process calls *wait* on *waitq*. This will cause the process to block until it is signaled by the *N*th process calling *bar\_wait*. After the *wait* call, the process signals *throttle* to allow the for-loop that the *N*th process is undergoing to continue.

In each round, the threads perform their comparisons if necessary, store the maximum of the two values in the array spots designated by their *index* values, and then call *bar­\_wait*. The *bar\_wait* call ensures that each process is operating on the same round so that the correct values are being compared. At the end of all of the rounds, the maximum value is stored in index 0 of the array, since thread 0 is always active based on the criteria established earlier. The threads then exit and the *find\_max* method returns the value stored in index 0 of the array.

It is important to note that no additional memory is allocated outside of that required for the *Argstruct* for each thread and the array itself. Each thread stores the maximum of the two values it compares in a place that other threads will access in later rounds. This also ensures that the maximum is placed in position 0 and the main thread needs to do no comparisons.

1. **Problems Encountered:**

The implementation of the *Barrier* and multithreaded algorithm was very straightforward. The most difficult part was managing the input and ensuring that edge cases would not cause any errors. The in-class discussions of implementations of synchronization structures using binary semaphores greatly simplified the threaded aspect of this problem, which at first seemed like it would be the most troublesome.

1. **Testing:**

The program was tested by generated a large number of files containing randomized lists of number and piping them into the executable. The result was compared to that of a separate program which found the maximum value by iterating sequentially through the array and storing the maximum value. The number files were generated by a Python program and the test was managed by a Bash script. There have been no input files for which the multi-threaded max-finding program gives a different result than that of the sequential max-finding program.

1. **Conclusion:**

This assignment was successful in teaching the use of semaphores and related structures in solving classic synchronization problems. I have learned the correct implementation of barriers, counting semaphores, and events as a result of this problem and the related in-class discussions. The multithreaded max-finding program utilizing barriers works as expected and runs quickly.

1. **Pledge**

On my honor as a student I have neither given nor received aid on this assignment.