Guthrie Alexander
CS 4414 (SP2016) Operating Systems
Project 2 writeup
Due: March 29 2016

Design of solution:

In designing a solution for this project, I ended up creating a struct **barrier_t** which served as the barrier, but also it contained the mutex in order to handle synchronization of access to the barrier, but also the condition variable so that threads could be held and released as needed. Essentially, the program reads the file twice (the first time to get the number of items in the file, the second time to populate the input[] array with all the values), and then it creates a thread for each comparison needed. The results are stored temporarily in a second array, output[], and then each thread calls the **barrier_wait()** function in order to make sure they are all finishing and moving on to the next round at the same time. Once the round completes, output is copied into input, and the next iteration of the while loop in the main function begins, until there is only 1 item left in input. This is the largest value.

Barrier_t

The barrier_t struct has three items in it: count, which is an int that contains the number of active threads in that round; pthread_mutex_t m, which is a mutex that synchronizes the threads access to the count variable in the barrier; pthread_cond_t cv, which is a condition variable that calls wait() on a thread when it's not the last thread in the round. The barrier is initialized once per while loop iteration, the only thing getting passed being the barrier_t global variable (so there's just one barrier that's initialized and destroyed every while loop iteration), and the number of threads in that round plus 1, for the main thread. Otherwise, the main thread would continue to execute while the others would be waiting, which would cause problems.

The barrier_wait() function is called for each thread after they complete the comparison, and store the value in the output[] array. All the wait function does is check to see if the thread calling the function is the last thread in the round (meaning that count has been decremented to or is equal to 1), and if it is, it will call **pthread_cond_broadcast()** in order to release all the threads. If it is **not** the last thread in the round, it will call the **pthread_cond_wait()** function in order to put the thread on hold.

Initially, I implemented and tested the barrier as just the **count** variable and had global variables for the mutex and condition variable, but that was causing problems. So instead, I created the struct and had it include all 3 of these variables, since each thread needed to use the mutex and condition variable every time it accessed the barrier_t anyway.

The first testing I did to make sure the barrier was working was just on a single round. Once it was clear it was synchronizing access to the barrier properly (i.e. multiple threads were attempting to access the count variable, but it was never displaying the incorrect value after numerous run attempts), and that all the threads were successfully updating the output array and then being released, I moved on to a run with the while loop (i.e. multiple rounds, not just a single round).

The issue I had here was somewhat interesting; it seemed to work about 80% of the time, but when I would run the program soon after initially compiling (i.e. the first time running make after saving project2.c) or running the program immediately after I just had done so, I would get a Seg Fault. After some troubleshooting (just printf() statements to the console), I realized it was the barrier, so that's when I implemented the **barrier_destroy()** function. This fixed the problem.

Main()

The main thread is organized into a few sections. First, it gets the number of values held in the file, and stores it in the **size** variable (this is necessary in order to malloc the correct size for input, the int array). Then, it calls the while loop which iterates through the rounds of comparisons. Each round needs a number of threads equal to the **threads** variable (which is just half the number of values in input, or size / 2). The main function does not participate in comparisons, but leaves that to the **compare()** function. The main thread does, however, need to call the barrier_wait() function, as otherwise it would continue on while the threads doing the comparisons would be waiting, which could lead to inconsistencies and problems in later rounds.

Input and output arrays

The way I thought about this was the input array holds the values to be used in the round of comparisons, and the output array holds the resulting values. The output array is the size of the number of comparison threads (which is half the size of the input array, since each comparison is done on an individual thread), and stores the value of that comparison (i.e. whichever number is higher gets stored in the output[] array at index "thread number"). Once the output array value is set, the thread waits/is released at the barrier, and then the main function continues. The output array is then copied to the input array (after it has been reallocated to be half it's original size), and then used in the next iteration of the while loop.

Perhaps this was unnecessary, and a single array would have been suitable, but I wanted to have separate arrays so that threads wouldn't be reading and writing to and from the same memory addresses, which would certainly lead to synchronization issues. Instead, threads for a single round are all reading from (but not writing to) **input**, and the same threads are writing to (but not reading from) **output**, which is a separate space of addresses, thanks to both arrays being malloc'd their own space each iteration of the while loop.