Write up
Comments
Code

Operating Systems Homework #2

Maurice Wong, mzw7af@virginia.edu

The assignment was successfully completed.

1. Problem:

The goal of this assignment was to implement a parallel, binary reduction using multiple threads to find the maximum in a list of numbers. Given a list of *N* numbers, the problem specifies that *N*/2 threads should be used to repeatedly perform parallel comparisons. In order to reuse the threads, a barrier primitive must be implemented to wait for all threads to finish before proceeding with the next round. The program terminates by printing the maximum number to *stdout*.

2. Approach

The main data structure implemented was the barrier, a synchronization primitive that takes in an integer *M* and requires *M* threads to wait on the barrier before releasing all waiting threads. The implementation can be found in the *barrier.h* and *barrier.c* source files. The barrier is implemented as a struct barrier, which has 5 attributes:

1. **int *size*** – the integer *M*, the number of threads the barrier must wait for before release.
2. **int *counter*** – a counter to keep track of how many threads have arrived at the barrier.
3. **sem_t *waitq*** – a semaphore to block threads while waiting for barrier release.
4. **sem_t *waitq2*** – another semaphore to block threads after barrier release while waiting for the barrier to reset before it can be reused in the next round of comparisons
5. **sem_t *mutex*** – a semaphore to give controlled access to the *counter* variable

Two methods were implemented along with the barrier struct: *initBarrier()* and *waitBarrier()*. Because C does not have classes, these barrier functions simulate object functions by taking in the barrier itself as a first argument, similar to how many programming languages implement object-oriented programming. *initBarrier()* initializes a barrier's *size* value *M*, initializes the *counter* to 0, and initializes the semaphores with the appropriate values. *Waitq* is initialized to a value of 0 because the barrier must be locked at first. *Waitq2* and *mutex* are both initialized to 1 because threads should not be blocked on the first *wait* invocation on either semaphore in the *waitBarrier()* method.

The *waitBarrier()* method is the main component of the barrier, which is called to force a thread to wait for the barrier's release. This method was implemented using the **two-phase barrier** approach outlined in *Little Book of Semaphores* under the "Reusable barrier solution" (Downey 41). It incorporates the use of two "turnstiles" which are the two semaphores in the barrier struct *waitq* and *waitq2*. A turnstile is the consists of a semaphore performing a wait and then immediately followed with a signal, which causes a thread, upon release from the "wait" instruction, to “signal” and unlock the semaphore for another thread to pass through, emulating a turnstile at a gate. The first of these turnstiles is used to cause threads to wait for the barrier’s release. When the barrier collects *M* threads, the second turnstile is locked and the threads are released from the first turnstile. However, before finishing the *waitBarrier()* function, the barrier must fully be reset, or else a thread can arrive again at the beginning of the barrier in the middle of resetting, causing unpredictable behavior. The second turnstile collects threads while waiting for the *counter* variable to reset to zero. Once it does, the first turnstile is locked again to finish resetting the barrier, and the threads are released from the function.

The second data structure created is the *argstruct* struct, which holds arguments to each individual thread, including the thread number, the total number of threads, the numbers array, and a pointer to the barrier. It also stores the result from the thread function.

To implement the actual max-finding algorithm, all the *N* numbers given are first read into an array *nums* in the *readNums*() function. *N*/2 argstructs are initialized accordingly, and then *N*/2 threads are created to run the *max()* thread function. The main thread does not participate in comparisons. The *max()* function consists of a while loop that runs once on each round of comparisons, and it reads and writes results to the *nums* array in each round. The function has 2 phases in its implementation:

The first is where each thread reads two values specific to that thread from the *nums* array and computes the maximum of the two, storing the result in a local variable. Each thread *tid* reads from array indices (*tid* * 2) and (*tid* * 2 + 1) as its “input”. This completes the first phase of the *max()* function and *waitBarrier()* is called to ensure all threads have finished reading the current values from *nums* before continuing.

The second phase is where threads determine if they will be rendered inactive on the next round, which dictates where threads should write their results back to the *nums* array. In each round, the lower half of threads will be kept active, and the upper half will be made inactive. Thus the threads that remain active will write their result to (*tid* * 2) and the threads that will soon be made inactive write to (2 * *tid* - *activeThreadCount* + 1), resulting in all the “winners” from this round being stored in the lower half of *nums* for the lower half of threads to access. *waitBarrier()* is called once again after this process to wait for all threads to finish writing to *nums* before proceeding with the next round.

This parallel process continues until the number of active threads is 0, at which point the first thread at index 0 is the last active thread and has written its result to its argstruct. The result from this argstruct is extracted and printed to *stdout* to finish the program.

3: Problems Encountered

The first major problem encountered was simply getting the program to compile. Attempts to compile the program resulted in an error message of “reference not found” for functions related to semaphores. Initially the problem seemed to be with the Makefile or not including the correct libraries in the right places for header and source files, but eventually it was realized that the compiler needed the flag “*-pthread”* to compile correctly. More careful reading of the documentation will be made in the future.

The second problem encountered involved setting up two turnstiles in the *waitBarrier* function. It was initially unclear what values each semaphore in the barrier struct should be initialized to, so it took carefully walking through the code and tracing values of semaphores to understand what was happening and what values they should start with. The particular problem encountered was that the second semaphore *waitq2* was initialized originally to 0, but this prevented the first thread from performing the initial locking of the second turnstile and passing through, causing the threads to hang.

4: Testing

The barrier implementation was tested independently to ensure functionality before integrating it with the actual max-finding algorithm. To test it, a simple program was written that initialized a barrier of size K, and then K threads were created that each printed out their thread ID number 3 times. However between each printing, each thread waited on the barrier to release, and the barrier printed to stdout upon release. The output was checked to make sure that each different thread ID's were printed and followed by the barrier statement, and that this output is repeated 3 times total.

To test the overall *max()* function, simple test cases were created:

- 2 numbers of different values
- 2 numbers of the same values
- 8 numbers with a mix of positive and negative
- Larger test case of 4096 numbers ranging positive and negative

5: Conclusion

Synchronization and race conditions are extremely difficult to think through and debug. The *waitBarrier()* function was challenging to implement since it took careful tracing from the perspective of multiple concurrent threads to determine where to wait and signal the semaphores and release threads. Ultimately, this was a useful exercise in learning to think about multiple concurrent processes and synchronizing them to avoid deadlocks and unintended side effects.