Ansun Sujoe

CS 4348

Exercise 3 Report

**Multithreads** **Problem**

Whenever the program tl.c runs on a single thread, everything seems fine and the program works as expected. However, when the program runs on two threads, the program does run faster but with errors as the number of keys missing is greater than 0. This is due to the fact that inside the put operation that the thread runs, the insert method that uses the shared resource called “table” is not atomic.

For example, in the insert statement, if thread 1 executes part of the insert statement, but there is a context switch to thread 2 before thread 1 gets to execute the line \*p = e, then the insert operation is for naught and one key will be missing when role is taken.

**Solution: Locks**

If we use a lock and lock the mutex before the insert operation in the put function, we see the key mismatch error go away in the case of multiple threads. We do not need locking in the get function because at that point in time when the gets are executed, there is no operation that is actually modifying the shared resource.

With the locks change, the performance of “put” is hit slightly (from about 0.005 seconds to about 0.0075 seconds), but the correctness of the code is maintained. Now, if the lock is on the get method, then the performance of “get” is worse by a factor of 2. It’s completely unnecessary because get can perform parallelly, but in this case, the two-threaded version is even slower than the one-threaded version (about 0.53 seconds total completion time for 1 thread versus about 0.66 seconds for 2 threads).

**Exploiting Parallelism**

As discussed earlier, we can run the “gets” in parallel and remove the locks altogether. This sped performance up (0.29 seconds for 2 threads) and this made the 2-threaded version slightly greater than half of the 1 thread version.

We can go further than this and exploit parallelism (not fully) in the put method. We can do this by having a lock for every bucket. When this is performed, the performance improves for “put” slightly (from 0.0075 seconds to about 0.0067 seconds).

**Increasing the Number of Threads**

2 threads: 0.270670 seconds

4 threads: 0.137961 seconds

8 threads: 0.080412 seconds

10 threads: 0.89042 seconds

16 threads: 0.100577 seconds

Notice that when there are 2, 4, or even 8 threads, the time performance is about 2, 4, and 8 times (respectively) faster. However, when there are 10 or more threads, it seems that the program runs slower. This may be because the threads are constantly locking and waiting for each other most of the time, so having more threads will be inefficient because the majority of the time, the threads will be waiting for a mutex to unlock.