Megan Sullivan
HW1 – Multithreaded Programming

Performance

The first workload I made (caps_bal_fine) was designed to favor the fine-grained locking implementation. First I created a (roughly) balanced binary tree, based off the list of countries and capitols provided to us in the initial project folder. Then I created two separate files with database commands. One file reads from one half of the tree (in this case, all the nodes on one side of the kuwait node) while the other file writes to the other half of the tree. Since the reads and writes are happening on two separate halves of the tree, they should be able to happen simultaneously in the fine-grained implementation, since nodes are locked on an individual basis rather than locking the entire tree at once. In both the coarse-grained and the read-write implementations, threads will have to wait until the entire tree is unlocked to continue. When I tested this implementation, I found that the fine-grained implementation (which took approximately 1.0 seconds) ran faster than the read-write implementation (which took 1.2 seconds), but the coarse-grained implementation was still the fastest (at 0.5 seconds). This is probably due to the fact that the coarse-grained locking strategy has the least overhead, since there is only one lock that needs to be acquired to proceed. While in theory fine-grained should work the fastest with this workload, in practice the overhead of acquiring and releasing all the locks for every node accessed outweighs the benefits.

Next, I created a workload (read_heavy) which favored the read-write locking implementation. I constructed a tree with over 8000 nodes and then ran eight threads who were all querying the database hundreds of times, with no add or remove commands (after the initial creation of the database). The read-write implementation performed the best (approximately 0.09 seconds), because once the database is locked for reading the first time, none of the other readers ever have to wait on the db_mutex. For the coarse-grained implementation (which ran at approximately 0.13 seconds), each thread still needs to wait for the db_mutex so that it can have exclusive access to the database, even though in this case none of the eight threads were changing the database structure at all. The fine-grained performance had the greatest variation, and the running times drifted between 0.12 and 0.17 seconds. This is probably due to the particular order in which the read commands were processed, since the performance of the fine-grained implementation is largely dependent upon the order in which nodes are accessed (and locked).

I created another test (linkedTree) that uses a database with over 16,000 nodes. All of the nodes are added to the database in numerical order, so the tree should essentially be a linked list of nodes. Then eight threads are run simultaneously, each of which begins deleting nodes from the bottom of the tree and working its way up. I found that with this particular workload, the coarse-grained implementation was consistently the most efficient, running at approximately 7 seconds, while read-write ran around 9 seconds. Fine-grained performed particularly poorly, probably because the amount of overhead due to that many nodes being locked and unlocked outweighed any benefits of the more involved locking strategy.

Locking strategy is more important on multi-processor machines because there will actually be multiple threads running on it at once. A uniprocessor system gives the illusion that multiple things are happening simultaneously, because the CPU will switch contexts to give multiple threads a chance to run, but there is really only ever one thread running at a time. In a multi-processor machine, there are multiple CPUs, therefore multiple threads can be executing (and potentially accessing the same memory) at once, so it's important that any shared data be locked properly to prevent race conditions.