Megan Sullivan
HW1 – Multithreaded Programming

Fine-Grained Locking

To implement a fine-grained locking system, I took the read-write implementation described in rw.pdf and integrated it into the node stucture. That is, I changed the node_t struct in node.h to include a num_readers variable, as well as locks request_mutex, num_readers_mutex, and node_mutex (analogous to the db_mutex in rw.pdf). These new elements had to be initialized in the node_create() method and destroyed in the node_destroy() method. I also wrote new methods read_lock(), read_unlock(), write_lock(), and write_unlock(), to make it easier to understand the changes I made to other methods.

There were four methods I changed to get appropriate database-locking behavior.

The first method I changed was the search() function. The search needed to behave slightly differently based on whether the method calling it was a read or write command. Luckily, for query (the read function), the parentpp parameter passed in is null, while in both add and xremove (the write functions), the parentpp parameter has a value which is not null. Therefore, I could use the value of parentpp to distinguish between whether the function who called search() was a reader or a writer.

If a reader performs the search, we don't need to keep track of the parent of the next node (the node we're looking at). So we can unlock the parent node parameter as soon as we have locked the next child to investigate. (It's important to always lock the child before unlocking the parent node, just in case there's a context switch between the two commands.) For the two outcomes of search(), if the target node is found in the database, it will be read-locked when the search returns, and if the target node is not in the database, none of the nodes will have been read-locked by the search.

If a writer performs the search, then we do need to keep track of the parent node. When the search returns, if the target node was in the database, then both the target node and its parent will be write-locked. If the target node wasn't in the database, then only the parent (of where the target node should have been) will be write-locked upon return from the search. So, the only case in which you'd want to write-unlock the parent node is if you're going to recurse and perform the search again. (If we know we're going to recurse, then we know that the current parent node can't be the parent of the target node, so it's safe to unlock it.)

The next command I modified was the query() function. Because search() assumes that the parent node will already be read- or write-locked, I had to acquire a read-lock for the head node (the first parent). After you perform the search, the target will either be a read-locked target node or null. If the node is found, you have to remember to unlock it after you perform your read and return.

The third command I changed was add(), a write function. This time, I had to write-lock the head node before performing the search. As previously mentioned, when you return from search, if the target node was found, both it and its parent node will be write-locked, otherwise only the parent node will be write-locked. In the event that the target node (the node you are trying to add to the tree) already exists in the database, you cannot add it again, so you have to unlock both the target node and the parent node before you return. If the target node was not found in the database, then you are free to add it. After you create a new node and attach it to one of the child pointers of the parent node, you can unlock the parent node and return. You don't need to worry about write-locking the new node, since no one else in the database will be able to tell the new node exists until you unlock the parent.

The final (and most complicated) method I had to modify was the xremove() function. As above, I had to write-lock the head node before calling search(). If the target node was not in the database, then you cannot remove it, so unlock the parent node and return. Otherwise if you did find the target node, figure out which type of remove you will have to perform (based on the number of children the target node has). If the target has zero or one child(ren), then all you have to do is remember to unlock the parent node after you destroy the target. If the target has two children, you can unlock the parent node, since this kind of remove does not require changing the parent node. Then you acquire the write-lock for the right child of the target. As you work your way down the chain of left children, you can unlock the nodes above as you leapfrog down the tree. (Since you already have the target node locked, no other threads will be able to get between the target node and the current write-locked node in your chain.) Once you reach the bottom of your chain and swap pointers, you can destroy the node at the bottom of the tree and then unlock the target node (which has a new key and value) above it.