UNIVERSITY OF SOUTHERN CALIFORNIA

CS402 Assignment 01 – Read/Write Locking

Michael Gerow

February 22, 2013

1 Fine Grained Locking

Fine grained locking is really the first locking style that differs significantly from coarse grained locking. With fine grained locking each node has its own pthread_rwlock_t, and we modify the search function in order to properly lock nodes as we traverse the tree.

From a high level, we modify search to expect the head node to be locked upon entry. Then as we traverse the tree we either grab and drop read locks or write locks depending upon the presence of the parentpp value. If it is present then we are either doing an add or a delete, so we should try to end the search with a write lock on both the target and its parent (or just its parent if the taget doesn't exist). If the parentpp value is not present then we try to end the search with a read lock on the target (but not the parent). If there is no target, of course, we exit with a read lock on nothing.

This allows us to essentially leave most of the query, add, and xremove functions unmodified save for the requirement to lock a node with the correct type of lock before calling the search function.

Within the query function we simply get a read lock on the head before we call search and then make sure we unlock the target before returning. Though, if we return without a target there is nothing we need to unlock.

With the add function need to make sure to get a write lock on the head before calling search. Then, if we find a node already exists at the location, we simply unlock the write lock on both the parent and the target we discover. Otherwise, we create a new node, put it on the correct side of the parent, and then unlock the parent.

With the xremove function we have a special case for when the node we want to delete has two children. In this case we simply collect and release write locks down the left side of the right subtree until we find a node with a left child that we can put in the removed node's

place, all the while maintaining our write lock on the original node. Once this is completed we realease our lock on the node we deleted, the parent of the node we deleted, and the node that we just moved into its place.

1.1 Testing

We test the correctness of the read write lock similarly to the read writer lock, using the test that adds a lot of elements to the database and then removes them, which should leave us with an empty database. This can, like with read write, be done simply by running:

```
./server_rw < test/250_add_remove | tail -n 900 | less
```

Again, this can be run and verified using the test.py script.
Of course, we should also expect an increase in speed for fine grained locking. It turns out that this is only true for certain types of accesses of the database. Performance testing will be covered in the performance document.