6.033
COMPUTER SYSTEMS ENGINEERING
HANDS ON 3: VALGRIND

JOHN WANG

1. Using Valgrind to Find Race Condition

(1) While *put* is looping through the entries in the table, it doesn't acquire any locks. Therefore, when it finds an open entry and attempts to set the open entry's key and value to the key and value inputs to put, there could be a race condition. Suppose that two threads have looped over the hash table and have found that entry i is open. If thread one starts setting the key the value given by it's put function, thread two still sees entry i as in use. Therefore, thread two will also try setting the the key and value in entry i. In the end, thread one writes to entry i, then thread two overwrites what thread one wrote.

2. Fixing the Error

(2) I created a new lock, called *lock2*, which every put method must hold in order to change an entry in the hash table. The put methods will read through the table looking for an unused table entry. If it finds an unused table entry, the put method will acquire the lock using the *pthread_mutex_lock* method. Once the method acquires the lock, it makes sure that the table entry is still unused (because it may have changed status before the lock was acquired). If the entry is still unused, the key and value in the hash entry are changed, the lock is released, and the function returns. If the entry is used, however, the lock is simply released. This is correct because nothing can write the the table unless it holds the lock and that entry to the table is unused. Thus, there is no way that the table entry could change in the time that a method holds *lock2*.

(3) Yes, the two-threaded version is faster. The two threaded version runs in an average of 1.61 seconds (with 5 trials), while the single threaded version runs in an average of 2.79 seconds (with 5 trials).

(4) I did achieve a speedup because I was smart about how I used my locks. The naive implementation I used the first time I implemented my locking was much slower. The naive implementation is to lock whenever a puts method reads from the hash table. This is slow because you need to acquire a lock on every read (and hence is slower than just a single thread reading the entire hash table without having to acquire and release a lock). The optimization is to not lock on a read, but only on a write. The reads in my implementation don't require a lock, but after the lock is acquired, my implementation rechecks for whether the entry is unused and puts a value if it is. This means that locks don't have to be acquired unless the program is reasonably sure it is going to write to the table.

3. Living Dangerously

(5) Now, the threads don't have to wait to acquire a lock before reading through their tables. Since locks take time to both acquire and release, the time decreases quite dramatically once the locks are removed. Now, instead of obtaining a lock every time a get is called, there is no lock. Moreover, now concurrency actually provides and advantage instead of a disadvantage. This is because multiple threads can be reading the table at the same time.

(6) There are no errors because *get()* does not modify the table. This means that no lock is required to read the table. Thus, even if two threads are reading a variable at the same time (or very close times), neither of the reads will be affected by the other read. This means that *get()* does not really require a lock and can still be thread safe even without a lock.

4. Challenge

(7) I implemented a change that has a speedup in both phases using my approach to locking only on writes. See the discussion in question 2.