6.033 Hands-on Valgrind

Kevin Cho
R5 Peter Szolovits 1PM

Due February 21th 2016

I Using valgrind to find race condition

1. If we look at the possible data race section, we could see that the locks held when we used the put function was none all the time. This is a problem because with multiple threads, we could be writing into an entry multiple times through multiple threads without knowing it. For example, we have two threads, thread a and thread b. Both threads are writing into entries in a hash table. However, both threads a and b land at entry *i*. What happens here is that without locks, thread a writes in the entry *i* first and then thread two overwrites *i* without the user knowing.

2. The change to the code was a simple change. In the code, we notice that the put function does 3 things. It first looks through the list of entries to find an empty or unused entry. Once this entry is found, we fill in the necessary slots and then return. Now, our problem is how to insert the locks in order that we eliminate the missing keys.

   The way we insert the lock is quite simple. First, we create the lock named something other than *lock*. Now, in the put function, after we find an empty entry (after the if statement), we obtain the lock/ we set the lock up. However, there could be a problem that the entry was already filled before the lock was obtained, so we add another if statement to check if the current entry is unused. We have two cases now:

   (a) The entry is still unused is the first case. Then, we do what we have to by setting the key and value for the hash entry. After we set the value and key, we would unlock the lock and then return.

   (b) The second case is that the entry is used now. For this case, we would just unlock the lock and continue.

   This is method is correct because we cannot set any key or value without obtaining the lock first. Thus, this corrects our error from before.

3. For the single-threaded process, I got an average time of 3.423 seconds after five runs. For the dual-threaded process, I got an average time of 1.738 after five runs. Thus, we can conclude that the two-threaded process is, indeed, faster than the single-threaded process.

4. There actually was a speed-up for my changes. This is because instead of the usual lock method of implementing the lock whenever we read from the table, I implemented the lock so that the lock only happenes when we find an unused entry. This speeds things up because both threads can work seperately, in a sense, to check different entries, but for the usual implementation, each thread must wait for the other to read. The way I locked the function only has the threads wait on each other to write if needed. Thus, there would be a significant speed up.

5. Without the locks in get, each thread does not need to wait for each other to acquire and release the lock before having their respective get functions fully run. Thus, all threads are running concurrently at all times and not waiting for each other to release. This causes the time to decrease by a whole lot because multiple threads can read the hash table at the same time.

6. No errors are read because our *get*() function requires no locks. This is because the *get*() function only reads our table and does not write. Thus, we don't need to stagger the threads to read what is on the table. What this means is that if two different threads are trying to read a certain entry at the same time, we do not need for one thread to wait for the other to finish.

7. The way I set up the locks in question 2 actually speeds up both the phases when we remove the locks in *get*().