Sam Mishra
N12140743
Lab #4

Part 1 - Under what circumstances can entries get lost?

Before adding locks to the *insert* function there was a race condition in which threads could potentially overwrite the same portion of memory.

For example if you had 5 buckets and 10 threads then there will be two threads writing to a bucket. So a thread could be doing its work, enter the *insert* function, execute the entire **e->next = table[i]** line, and then the scheduler could switch to another thread. Now when the next thread begins running it will run the **e->next = table[i]** line. At this point both threads are in the process of creating two different entries at the same memory location. Let's say the second thread finished executing the *insert* function and then the scheduler runs the first thread again. It will start running from the **e->key = key** line and therefore rewrite the two values of the second thread .

At this point we have lost one entry.

This race condition can be solved by using a mutex lock on the critical section of code, which is the part of the *insert* function where the linked list is expanded, the key and value are written to memory and the table is updated.

Part 2 - Does retrieving an item from the hash table require a lock?

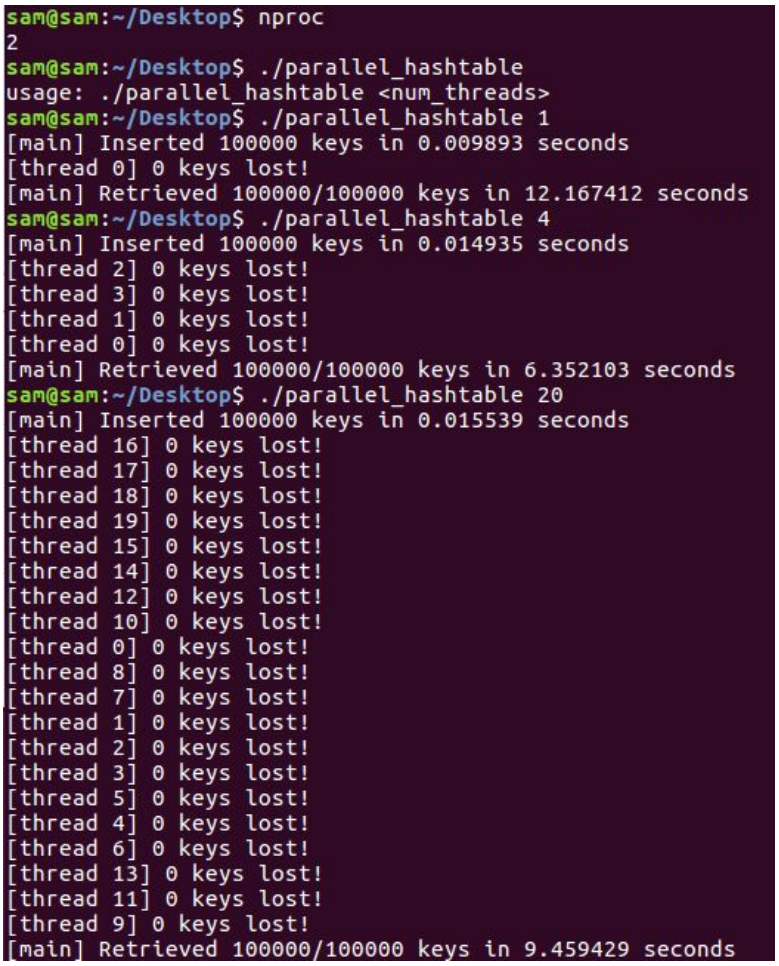
Retrieving an item does not require a lock because a barrier was created in which all the writing threads are to end before any more of the code is executed.

Since the read comes after the barrier there is no possibility for a piece of memory to be overwritten while being read. Also each thread has a specific set of keys it is checking for, therefore there is no conflict between reading threads since the keys, buckets and tables are not shared resources.

Part 3 - If two inserts are being done on *different* buckets in parallel, is a lock needed? What are the shared resources that need to be protected by mutexes?

Threads only write into the buckets they hash to. Only threads that hash to the same value share a resource (the bucket). So it is not a problem to have multiple threads writing into the table at one time but it is a problem to have multiple threads writing into one bucket at a time. To solve this we need a lock for each bucket instead of one lock for the entire table.

Output:



This screenshot shows:
that the machine has two processors,
usage of the parallel_hashtable function,
running the script with only 1 thread,
running the script with 4 threads,
running the script with 20 threads.

As seen in the screenshot threading provides some efficiencies and with 20 threads we were able to safely insert and retrieve all the keys without running into a race condition.