Jaime Riley

CIS 3207

Lab Section 4

Networked Spell-Checker

**Introduction**

This lab is an example of the producer-consumer problem we have studied in class. This solution focuses on correctness and efficiency through locks and condition variables to provide mutual exclusion of resources. Since the high-level overview of this project is included in README3.md of my github, I will assume that the knowledge of what the program does is known. Therefore, I will move on to the design and implementation of my solution.

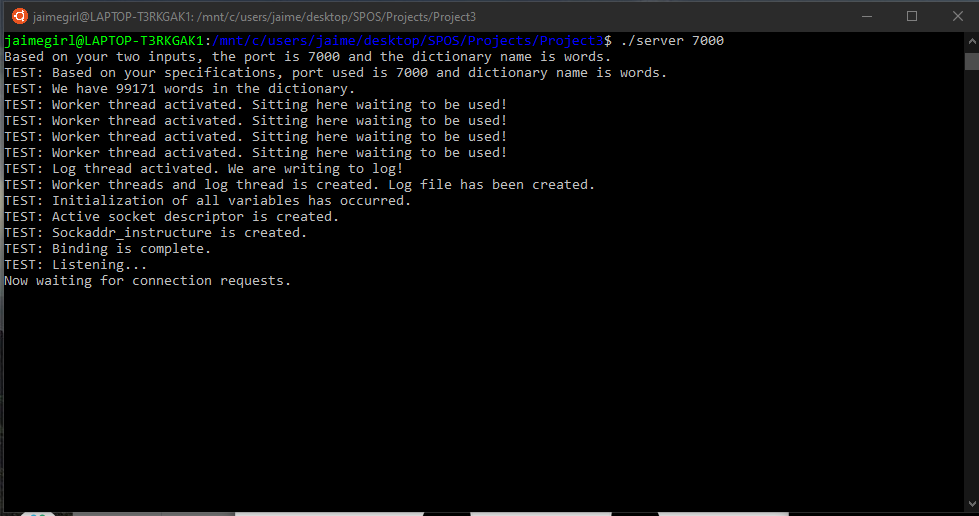
**Design & Implementation**

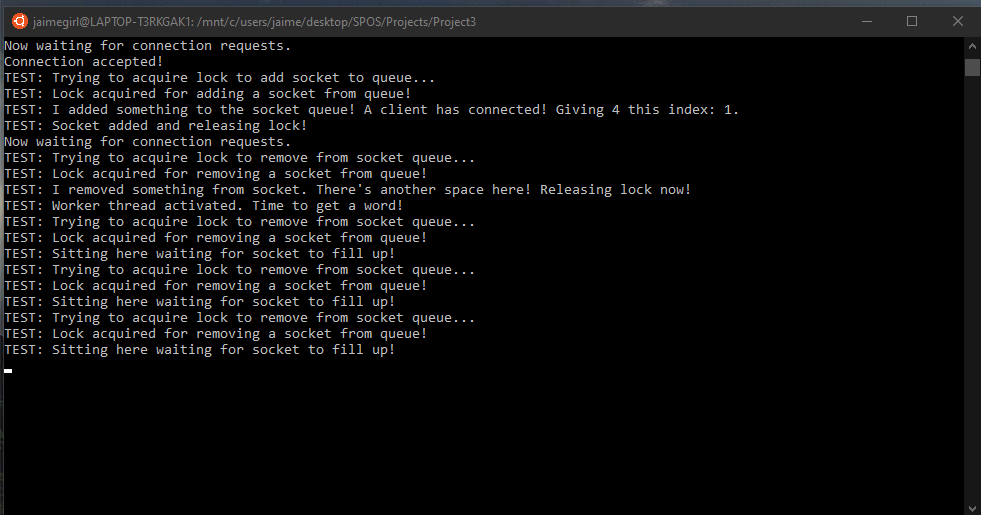
In my design for this project, I relied heavily on chapters of the textbook and the pseudocode provided in lab. In comparison to past assignments, the conceptualization was not as difficult and therefore the design of my project did not take long. I have commented heavily throughout my code, and the comments were laid out primarily before I implemented my solution. My goal was to have a few helper functions and functions dedicated to each thread, which includes the main thread, log thread, and worker threads. For my helper functions, I have two separate functions that add to the socket queue and log queue, two separate functions that remove from the socket and log queue, and a check dictionary function that will determine whether a word is in the stored dictionary. Loading the dictionary was implemented in the main function, which simply loads a predetermined string array that is globally accessible. A major help in my design was having two structures dedicated to all the variables that would be need mutual exclusion for logging and for sockets. These are only accessed when either the add or remove function is called for a queue. This helped to make mutual exclusion straightforward. In the main thread I initialized all of my variables and established the connection to the server. Then, the main function is on a constant loop waiting for connection requests. When one is received, the socket descriptor is added to the socket queue using the function addToSocketQueue. Each time the mutual exclusion variables are accessed, the lock is acquired by the thread. If we are adding and the queue is full, the thread waits for a signal that the queue has an empty spot. It then adds to the queue, releases the lock and signals that something was added to the queue in case other threads were waiting to remove from said queue. If we are removing from the queue and the queue is empty, the thread waits for a signal that something was added to the queue, then it removes from the queue and signals that a spot is now empty incase a thread is waiting to add to the queue. This is how mutual exclusion works in my program. I was able to implement this through the use of pthread structures provided through the pthread.h header.

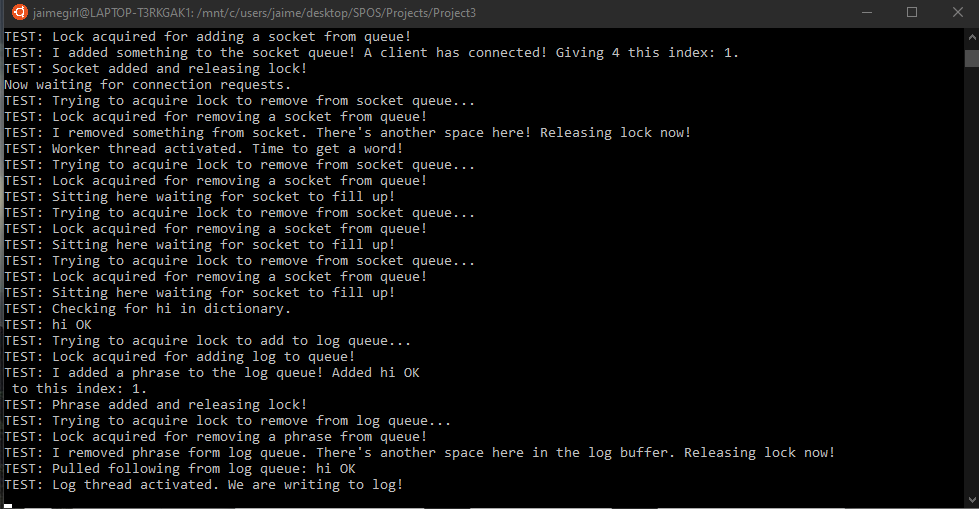
**Testing**

When I first started testing my server, I was not able to use the loopback address to connect. However, I was able to use several instances of Ubuntu to achieve testing. I have included screenshots of my tests below, when I had my testing condition on.

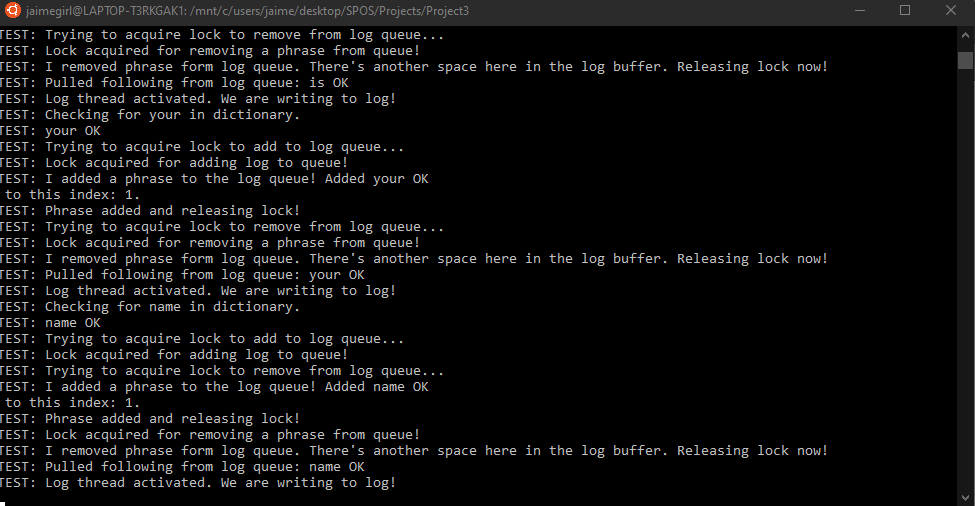
This is what happened when I run my server with the port 7000.



This is what happens when I connected a client.

This is what happens when I have a client input one word.

And finally, this is what happens when I have multiple clients input multiple words all around the same time.



Although this testing has a lot going on at once, it is able to handle multiple clients giving input on a server at once without getting rid of any of the inputs or having issues with the log file.