Developing a Network Spellchecker

Author: Carlos Chavez

OVERVIEW

This project is a simple implementation of a Network Spellchecker in C. Essentially, there is a server thread that communicates with clients. There is also a certain amount of worker threads that handle the spell checking. Lastly, there is a logger thread that prints all network activities to the main terminal. Throughout this document, I will describe how I programmed the Network Spellchecker and how I tested it.

In order to begin this program, it is important to understand that queues are the backbone of this program. I began this project by creating queues that accept and return void pointers. I decided to use void pointers because we need two different types of queues. One queue that would handle integers for the client descriptors, and another that would track words for the log. At the time I did not think to screenshot the testing of these data structures, but I have a screenshot of my first commit to github where I tested my queue.

```c
#include <stdio.h>
#include <stdlib.h>
#define DEFEAULT_DICT "dictionary.txt"
#define DEFAULT_TEST_DICT "testdict.txt"
#include "queue.h"

int main(int argc, char **argv){
    Queue *test;
    test = init();
    char *t = "ltest";
    enqueue(test, t);
    printf("%d", test->length);
}
```

After making sure that my queues worked properly, I set up the logic to handle the program arguments. If nothing is passed with the file, then the default port and dictionary are set to values which are defined at the top of the server.c file. I then began messing around with threads, which I luckily screenshotted. Nothing fancy here, I just had the thread functions print out their type.

```
(base) carloschavez@Carloss-MBP-2 spellchecker % ./test
Worker created created!
Logger created!
Worker created created!
Worker created created!
```

Next, using the open_listenfd function given to us in the Computer Systems textbook, Blake helped me figure out the networking logic that creates and binds socket descriptors to the specified port.

After locking and adding the client socket to the client queue, I began to work on the management of socket descriptors by the worker threads. At first I thought the program worked perfectly, but I was having a deadlock issue. I began testing the program with a number of workers and clients. Whenever I tried to add more clients to the queue than the amount of workers available, the program would get stuck. I wish I took a screenshot because that image would look perfect here. However, I added an if statement to check whether the queue was full or not, and if it is filled then the program shall wait until there is room on the queue. This allows the program to successfully and concurrently access the client queue, while handling each client properly while not corrupting the queue (along with the help of locks, of course).

After building the worker thread function, I noticed a problem with my spell checker function as I was testing multiple words. Instead of using strcmp, I was using strstr which would always return true if any letter (or sequence of letters) was passed by a user. After removing the newline character with strtok, my spell checker function began working properly.

```
Send the escape key to close the connection.
>>> test
test OK
>>> te
te MISSPELLED
```

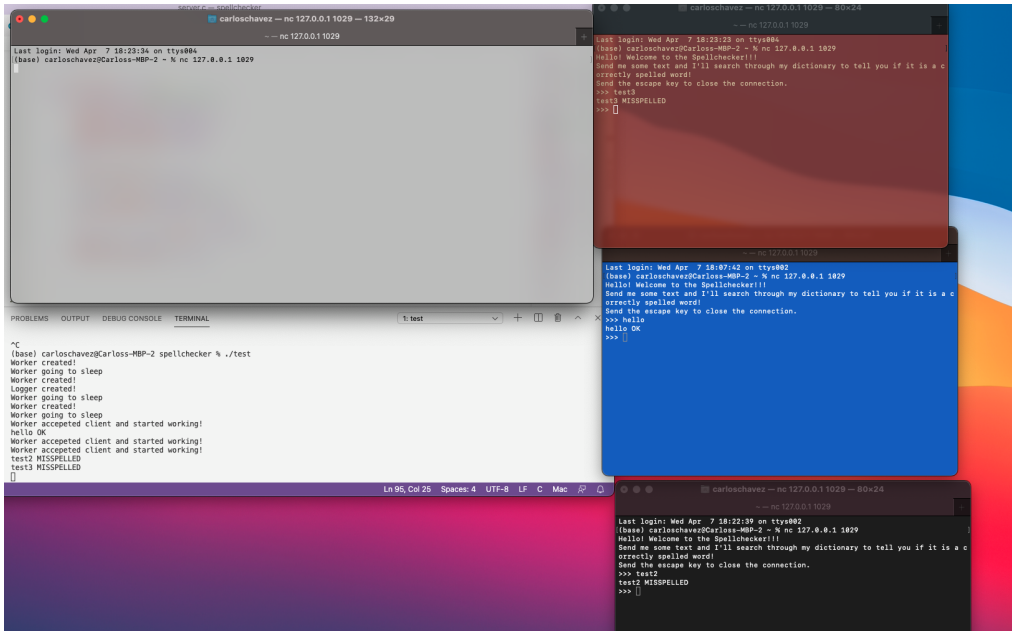
One of the last things I did was create the log function that monitors the log queue and processes entries. The logic for the log function is quite simple, it simply removes the processes entries by dequeuing from the log queue in a safe manner. Again, shoutout to locks for allowing us to properly access the queue.

```
(base) carloschavez@Carloss-MBP-2 spellchecker % make
gcc -Wall -Werror server.c queue.c -o test
(base) carloschavez@Carloss-MBP-2 spellchecker % ./test
Worker created!
Worker going to sleep
Worker created!
Worker going to sleep
Worker created!
Worker going to sleep
Logger created!
```

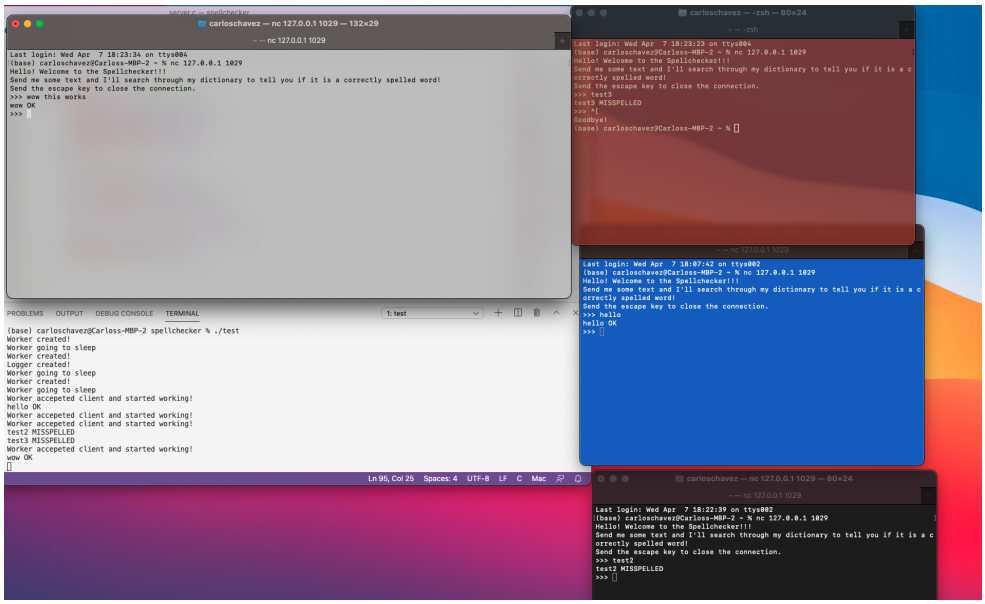
Although I tested and fixed the deadlock problem when I was messing around with the number of workers and clients, I will run and take screenshots of the working program to conclude the tests that are needed for this project.

More below.

As you can see in this screenshot, the number of workers available was only 3. When I try to call the netcat program in the grey terminal screen, it seems as though nothing is happening. However, the worker threads are really just busy servicing other clients.

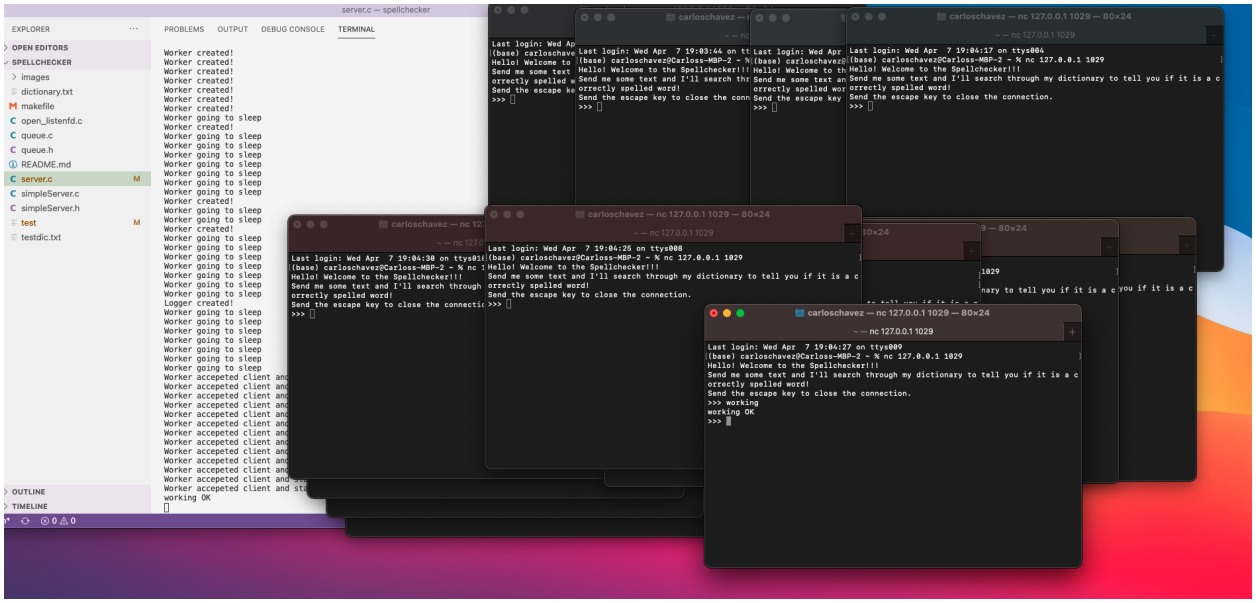


As soon as a worker thread becomes available, the program interacts with the expected client.



Just to make sure the program truly works, I decided to mess around with the number of workers. I first made the number of workers 10 and everything worked as expected. I even bumped the number of workers up to 30 and everything is still working perfectly (as seen

below).



Finally, I noticed that I spelled accepted wrong and fixed it. The project is officially complete.