To solve the producer/consumer problem, I created two mutexes/condition variables to protect the two concurrent resources. When the main thread sees an incoming connection, it puts the file descriptor into the queue of incoming connections. This queue is protected so that only one thread can read or write from it at a time. When the worker threads take the connection out of the queue, they only need access to it for a brief time, after which the connection can be stored as a local variable. By releasing mutual exclusion immediately after reading from the queue, other threads are free to read from it without waiting for the current thread to finish working. When writing to the log queue, the threads generate the message they are storing before putting it in the queue, so that other threads don’t have to wait for it.

To test this program, first I made sure my socket code was working properly. I wrote a program that simply echoed the message it receives. Then I added the multithreading support and ensured that multiple clients could connect at the same time. Once I knew that everything was working properly, I added the code to check the dictionary for proper spelling to complete the program. I tested it locally and on the Linux servers to again ensure that everything was working properly. I used several simultaneous clients to see that it can handle concurrent connections.

For the spellchecking component, I elected to have the program simply build an array before starting any worker threads. When the worker threads check a word, they simply search the dictionary until they find it. I considered implementing a hashmap to check spelling more rapidly, but I elected not to keep the program simple and easy to test.

For my queues, I used generic code I have written previously that I know works.