Include

* pthread.h
* stdio.h
* stdlib.h
* unistd.h
* sys/socket.h
* netinet/in.h → IDK
* string.h

Initiate the pthread\_cond\_t cond

Initiate the pthread\_mutex\_t lock

Critical Section

* The queue of socketfd that need to be pulled and bound to the socket
* The output file that writes all the information that comes

1. Server - Main Thread - purpose of the main thread is to open up the server for connections and continuously process the clients that are accepted
   1. Load in the Dictionary
   2. Create the phrase Buffer and the Client Buffer
      1. Phrase Buffer
         1. char\* phrase\_buffer[] = (char\*)malloc(sizeof(char)\*100);
         2. pthread\_mutex\_t phrase\_mutex
         3. pthread\_cond\_t phrase\_empty, phrase\_full
      2. Client Buffer
         1. int\* client\_buffer[] = (int\*)malloc(sizeof(int)\*10)
         2. pthread\_mutex\_t client\_mutex
         3. pthread\_cond\_t client\_full, client\_empty
   3. Create a Pool of Threads
   4. Set up Network Connection
      1. Use the struct sockaddr\_in to create the variables “client” and “server”
      2. Get the socket descriptor using socket()
         1. server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)
      3. Set the sockaddr\_in server fields
         1. **server.sin\_family = AF\_INET**
         2. **server.sin\_addr.s\_addr = INADD\_ANY**
         3. **server.sin\_port\_htons = htons(portNumber)** → port number as defined by me, will be a global variable most likely
      4. Use bind() to connect server socket address to the socket descriptor
         1. **bind(socket file descriptor, sockaddr \*addr, socket length)**
            1. **\*addr = (sockaddr\*)&server**
            2. **Length = sizeof(server)**
      5. Use listen() to convert active socket to a listening socket
         1. **listen(int socket\_file\_descriptor, backlog)**
            1. Backlog is the size of the queue that holds the outstanding connections - usually 3
      6. Initiate a continuous loop
         1. Accept a new socket
            1. int client\_sd
            2. if((client\_sd = accept()) > 0))
            3. Add the cliet\_sd to the client buffer ← client buffer is a critical section and will be checked within the thread
         2. Check if socket acceptance worked
         3. If it did work → PROCESS the connection with the worker threads
2. Server - Worker Thread - Spell Checker Function → **spell\_check(void\*)**  - must get the client socket descriptor from the queue of clients I NEED TO MAKE A QUEUE OF INTS THAT REPRESENT CLIENT SOCKETFD’S, receive a word from the client and check it, and send a message to the client about the word
   1. Use **pthread\_mutex\_lock(&lock)**
   2. Initialize while loop with condition: while the work queue (the queue of clients waiting to connect) is not empty
   3. Remove a socket from the queue (the work queue will be a queue of ints)
   4. Signal that there is now an empty spot in the queue
      1. Use **pthread\_cond\_wait(condition variable, mutex)** → releases a lock on a specified mutex and waits on a condition variable
         1. When using pthread wait the thread it is used in will go to sleep until another thread uses pthread signal on the condition variable
      2. Use **pthread\_cond\_signal(\*condition)** → wakes up threads waiting for the conditional variable to be unlocked
   5. Perform the spell check with appropriate I/O
   6. Close the socket
3. Spell Checker Function
   1. Take word from the client as a parameter
   2. Initialize while loop that will go through the dictionary with condition: while the client word does not match the dictionary word
   3. Return whether or not the word is spelled correctly according to the dictionary
4. Log Writing Thread
5. Log Writing Function

<https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/>

<https://www.geeksforgeeks.org/socket-programming-cc/>

<https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>