Jonathan Chu
CSCI-402
February 22, 2013

Part A: Commands

- **e-command**

  For the whole server, the way I handled creating threads was to malloc a space to hold an array for a max number of threads from the beginning. I set the max number of threads to be 1000. I don't really expect there to be that many threads because normal computers cannot handle more than about 100 threads. Each index in this array corresponds to a spawned thread's id number. Also, to keep track of the status of a thread, I have separate array of the same max size. Each index corresponds to that thread's status as either not in use, running, or waiting to be joined.

  I created a new function that will be called by the pthread_create which is in charge of the threads running and destruction. First, I call the client_create function to create a new client wit ha window. In that method, I use the global mutex to have exclusive to change the thread's status in the status array to change it to signify that the thread is in use. This allows for shared data with the main thread, which will be checking on the status of the thread under certain conditions.

  After the client_create is done, I call the pthread_create to create a new thread under which this client will run. The client will run on its own, and when it finishes, the thread will terminate, and wait to be joined with the main thread. I again lock the global mutex to change the thread status to waiting to be joined. The main thread joins any waiting threads at the end of its execution lock (you have to input another command to see that a thread has terminated and joined).

  As a side note, my system only takes in one command at each line, so something like a string of commands like. "eeeee" will only be interpreted as a single "e".

- **E-command**

  The process for doing the E command is pretty much the same as the e command. However, the main difference is that instead of calling client_create, I use the client_create_no_window function. Furthermore, to get files to input and output, I have a separate prompt for them. That means, to create a client with no window, you do the following:

  >> E
  >> <input file name>
  >> <output file name>

  This is a very important distinction, since I do not read in all those fields in one line. IF you do not want to have an output file, and instead want the client to output to the command line, just leave the <output file name> entry blank, and press <ENTER>. There are some issues reading in file names since there seems to be a bug in the given code.

- **s-command**

  To deal with stopping threads, I use a mutex, condition variable, and flag to denote that "s" has been pressed. When I receive the s-command, I use my flag to denote that the threads should stop. At the start of my client_runner function, as well at the top

of the while loop in the function to handle commands, I lock my mutex (separate from the one used in the above commands), and I check to see if the flag is raised. If it is, I call pthread_cond_wait to wait for my condition variable to be broadcasted in "g" to signal to all the clients to start back up again.

The reason why I check the flag at two places in the function is so I can stop threads depending on why I need them to be stopped. The top flag check allows me to create a bunch of threads that are stopped from the very beginning, so I can run them concurrently. On the other hand, the check in the while loop is to pause threads that are currently running when I input the "s" command.

- **g-command**

  The "g" command is the opposite of the "s" command. When I receive the "g" command, I lock the mutex used in the "s" command, and change the flag to show that threads do not need to stop. This allows for not currently blocked threads to run. Then, it broadcasts my condition variable. By doing this, all the threads that were blocked because of the "s" from above will receive this broadcast, and will start back up again.

- **w-command**

  The main job of the "w" command is to properly wait for all the threads to finish executing. I do this by using the status array that I created for the "e" and "E" commands. I lock my access mutex, and check each index of the status array for an index that denotes that the corresponding thread is still in use. When I find the first occurrence of this, unlocks the mutex (so the joining thread can access the status array), and then I call join on that thread. I have access to that thread because they are stored in the client array, and has the matching index as the status array. This blocks the main thread as it waits for the other thread to terminate so it can join. This method joins threads in order of increasing thread id number. While this is not as efficient as other methods which would join threads in order of their termination, I reason that this allows for the least amount of busy waiting and having to constantly loop through the status array looking for one thread's status to change.

- **Ctrl-d**

  The way that I handled all of the above commands was by using a while loop that kept getting getline arguments as long as "Ctrl-d" was not inputted. When this signal is inputted into the command line, I exit out of the while loop. Outside of the loop, I do the same thing as the "w" command, joining any terminated threads, and waiting for other executing threads to complete and joining them. Afterwards, I exit out of the program.