Megan Sullivan
HW1 – Multithreaded Programming

Commands

'e' – Create an interactive client in a window

> The 'e' command creates a new client using the client_create() method provided in the server.c file. If a new client is created successfully, the program calls pthread_create() and starts a new thread to run the new client. (The attribute that is passed in to pthread_create() ensures that the new thread will be joinable later on, when it is going to be killed by the terminator thread.)

'E' – Create a non-interactive client that reads commands from one file and writes results to another

> The 'E' command is implemented in a similar manner as the 'e' command. The only difference is that the new client is created via client_create_no_window(), which was included for us in the server.c file. After the command in the server window is received via getline(), the input is parsed and checked to make sure that the proper number of arguments (i.e. filenames) are included.

's' – Stop processing commands from clients

> The 's' command sets a paused flag. This flag is protected by paused_mutex, since it needs to be accessed by all the running client threads. In the client_run() method, before handling a command, the client checks to see if the system is paused or not. In case acquiring the lock for the paused variable takes a large amount of time (and we assume that the system will be unpaused the majority of the time), we first check the paused variable without acquiring the lock. Then if the system is paused, the client acquires the lock and double-checks that the system is still actually paused (since there may have been a context switch since the first check and the system may have unpaused already). If it is, then the client waits on a condition variable which will be signaled when the user calls the 'g' command. Then the client will wake up and handle commands again.

'g' – Continue processing commands from clients

> The 'g' command resets the paused flag. First it has to acquire the lock to the paused variable, then it sets it back to 0 and broadcasts all the threads waiting on the paused condition variable. Then it releases the paused_mutex so that other threads can acquire it.

'w' – Stop processing server commands until currently running clients have terminated.

> The first thing the 'w' command does is to implicitly call the 'g' command, to make sure that threads are actually running before you start waiting for them to finish. (If we didn't do this, then if a user called 's' and then immediately called 'w', the program would never be able to progress, since it would be waiting for threads to

finish but the threads wouldn't be running and therefore would never finish, and the user would be stuck.) Then the program waits on a condition variable for the running variable, which keeps track of the number of threads that are currently running at any time. (This running variable is also protected by a running_mutex, since it will be accessed by multiple threads.) The program will wait until running reaches zero, which means all the previously running threads have finished. In order to actually terminate threads, we have a separate thread which runs the terminator() method and kills clients (and joins their respective threads) who are finished running. Each time the terminator kills another client, it decrements the running variable and signals the waiting thread (in our case, the only thread waiting on the running variable is the main thread, so we can use signal instead of broadcast). If the thread sees that running has reached zero, then the server will continue processing commands input by the user. (Note: if the user continues to enter lines of input while the server is still waiting for running threads to terminate, when the main thread wakes up again it will process all the commands, i.e. lines of input, entered by the user.)