Saatvik Mohan

CS 3281

5/1/2019

Final Assignment

This report outlines the approach that I took to successfully fulfill the Final Assignment requirements. I start by providing some background, followed by my general approach. The bulk of the report is running through the process of sending commands from the client and the results acquired (provided along with screenshots).

Background

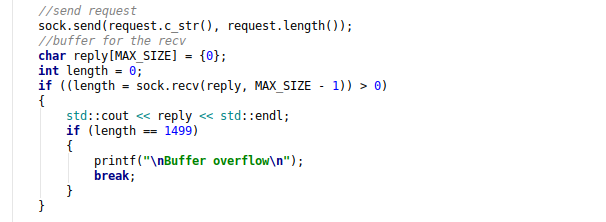
My code relies heavily on the Practical Socket code, both that specific class as well as the TCP Server and Client classes, and Assignment 2 code. The Practical Socket code provided the functionality of the socket, providing the skeleton for the connection between the server and client. Assignment 2 provided the functionality to put a buffer of chars in a shell argument[] and execute a command.

General Approach

I outline my general approach in both my client and server.

**Client:**

In the client main function, there is a test first to check that the number of arguments is two. The first is the server address, and the second is the server port. This is the server that the client is trying to connect to. The client can only connect to one server at a time. If the server connects successfully with the username and password (this is merely a formality because that information is provided to the user), the client enters a terminal loop. In the loop, getline() is used to get the request by the user. The user has the option to disconnect or Ctrl C, which kills an executing child process in the server. Additionally, if the length of the request is greater than 99, which is the MAX\_SIZE - 1 of the server’s buffer, the request is ignored. This is a preemptive check so that the server is guaranteed to have a command of accurate length. After the request is sent to the server, it must then wait to receive a message from the server. There is no way to read byte by byte since you don’t know what content the client needs. I implemented the recv by specifying a max size of the buffer and if that max size is violated, the buffer has overflowed, and the terminal exits.



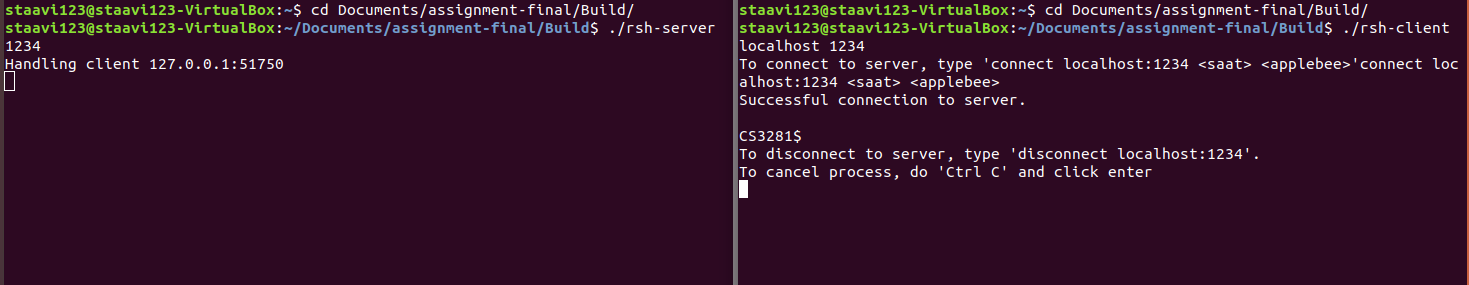
One another piece of functionality in the client is the SIGINT handler, but I’ll go into more detail later.

**Server:**

The server’s main function specifies argument 1 as the local port, and then the server enters a terminal loop that calls handleconnection() if a client connects to it. In the handleconnection(), after some exception checking, a fork occurs. Within the child, the server enters a while loop to receive messages from the client. This message is deposited in an array of chars. Then there are checks to see if the client sent over a SIGINT or a built-in command. If not, this means a regular command has been entered. If that’s the case, using the code from Assignment 2, the buffer’s contents are placed in a shell\_arg[]. Then the execute function is called, which forks and calls execvp on the shell arguments in the child. Before that, however, dup2 is used so that the output from execvp goes to the socket and therefore the client. Unlike previous times we’ve forked, the parent does not do a waitpid() in execute(). Instead, it exits and goes back to receiving messages from the client. The reason for this is so that the parent can see if a SIGINT signal comes so it can kill the running child process. After the while loop ends, the socket is deleted.

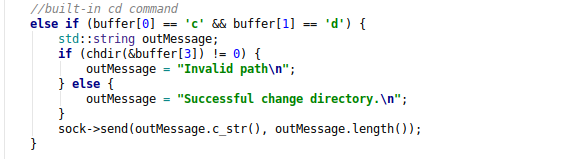
Running through Specific Commands

1. Connection between Client and Server

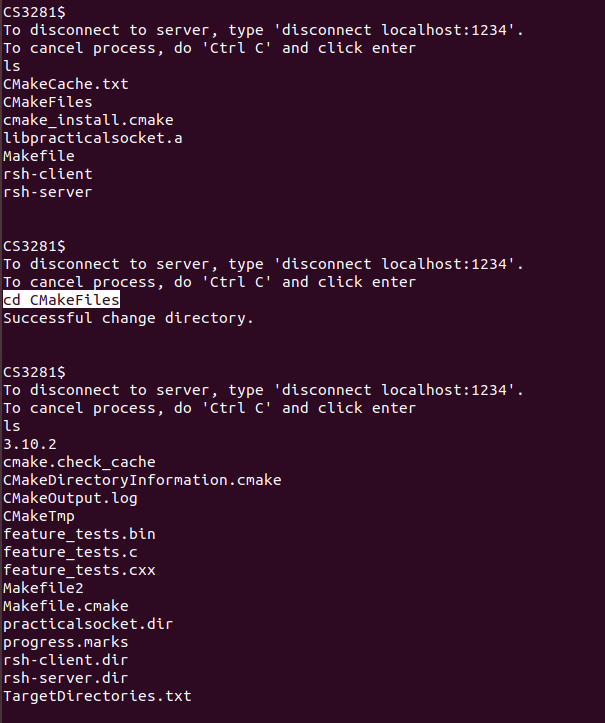


1. Built in cd command

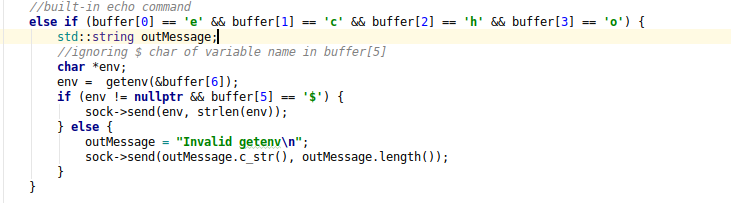
When a cd command is sent to the server and parsed into the buffer, the server sees that the first two chars correspond to the command. After the path is put into a string, chdir changes the directory. Then a message is sent to the client indicating success. c\_str() was useful in converting strings to a pointer to an array of chars.

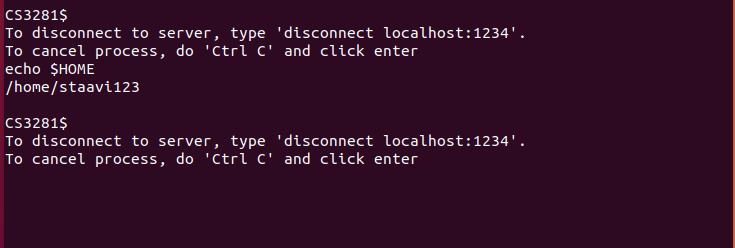


Below is the terminal output

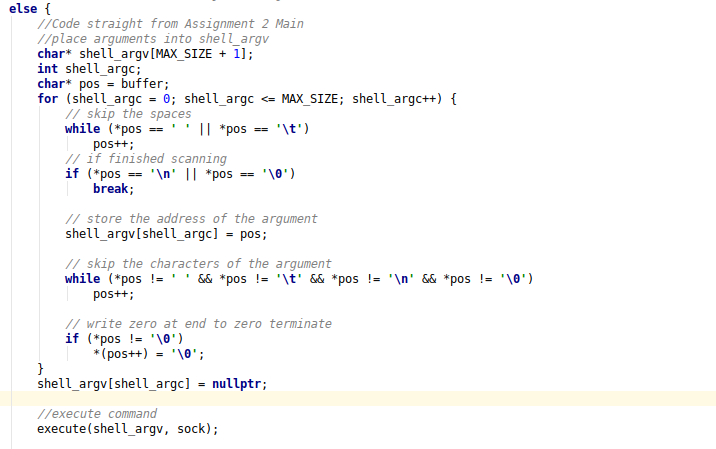


1. Built in echo command

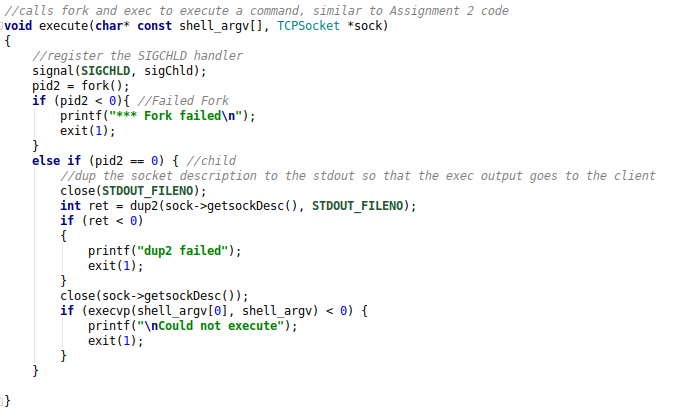
The echo command is done almost the same way as the cd command. It calls the getenv() function, passing in a pointer to the environment. Note that since the $ simply indicates that there is a VARIABLE\_NAME, the environment starts at the char after it.



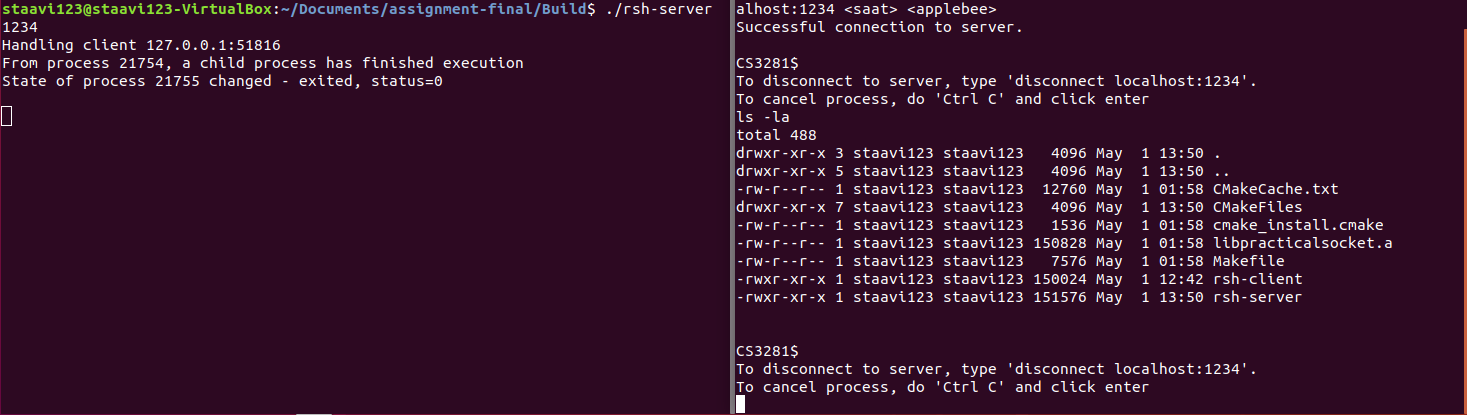
1. Regular Command

When a regular command is sent to the server and parsed there, it is placed into the shell arguments. This code is from Assignment 2. Execute() is then called.

In execute(), first the signal handler for SIGCHLD is registered (more about this later). In the child, the output is closed, then dup2 is called so that the exec output goes to the client. It is important to note that the parent immediately leaves the function and goes back to the beginning of the while loop to receive messages from the client. This is in case there’s a SIGINT signal.

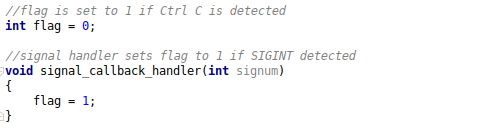


Once the child finishes executing, it allows other processes to execute. Below is terminal code for a command. More about the server output in the next section.

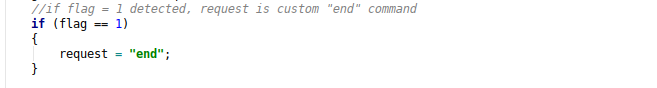


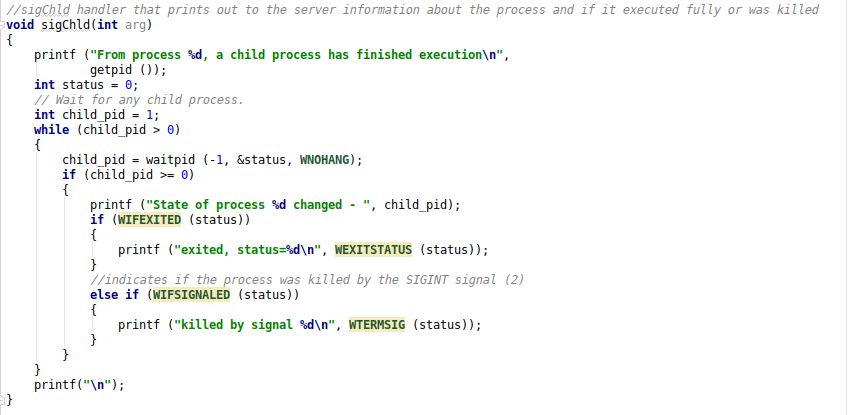
1. SIGINT

The SIGINT signal, if applied through a Ctrl C, should end any child process still running in the server and then end the client terminal. To capture the Ctrl C in the client, a signal handler is needed to set a global variable flag to 1. The signal is registered outside of the loop with signal(SIGINT, signal\_callback\_handler).

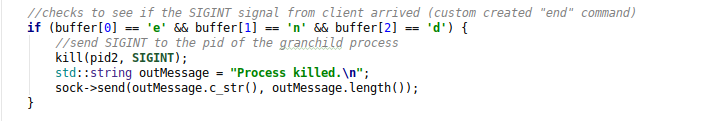


Then within the loop, if the flag is 1, the request is set to the string “end”, so the server can recognize it as a SIGINT. Important to note that with my design, the getline() is still a requirement, so after Ctrl C, an enter must still be pressed.

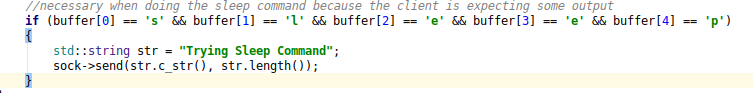


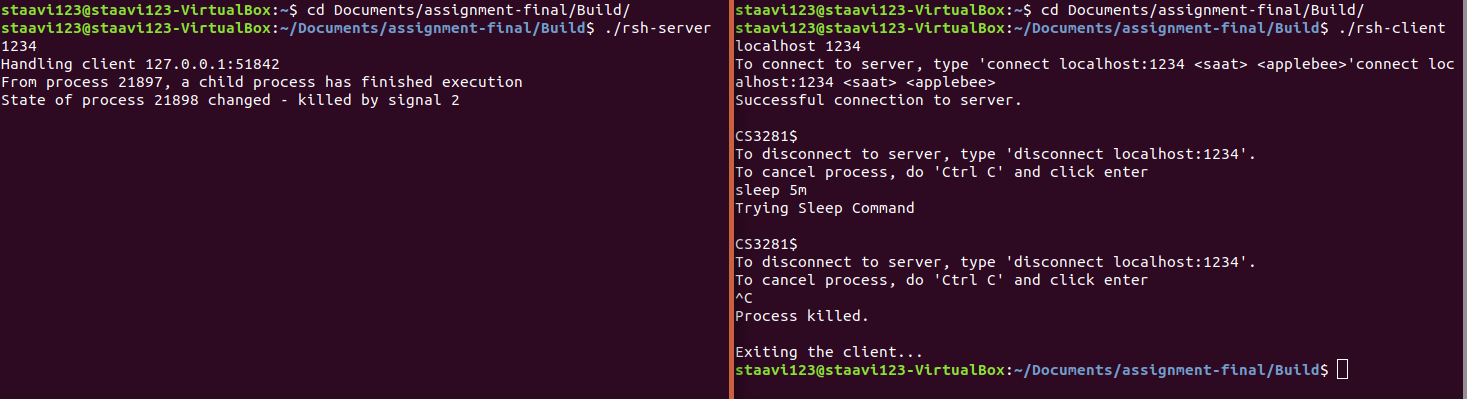
In the server, another signal handler must be registered for the SIGCHLD. 

Whenever a child process returns, information is given about that process. When a process gets killed, it indicates that as well. When the parent of a currently running process encounters a SIGINT, it sends the signal using kill().



Below is terminal output from a SIGINT on a sleep command. One thing to preface that is to say that there was no way to have the sleep command return without having it output back to the client first (so it looks a bit weird).





Conclusion:

Overall, this assignment was a good learning experience in sockets and required using all the knowledge we gained during the semester.