Purpose:

This work is split into two programs: a client and a server. The client program is used to connect to a specified server created by the server program. The client then acts as a limited remote shell, allowing the user to run some commands, such as “pwd”, “ls”, etc. The server is concurrent, meaning that one server can run commands for multiple clients simultaneously.

Client:

The client starts by validating the arguments it is called with. Once it validates the arguments, then it tries to connect with the server specified. If it can not connect with the server, then it will error out and terminate. If the client can connect with the server, it will run the session() call, which performs the shell function.

In the session() call, the client waits for a new command (an input from stdin) from the user. When it sees a command, it checks if the command is invalid. If it is, then it tells the user and then waits for a new command. If the command is empty, it simply waits for a new command. If the command is “exit” then the session ends and the program terminates successfully. If the command is anything else, it will send it to the server and then wait for a response. It will continue to attempt to read from the socket until it receives “done sending”. When it receives this message, it will then wait for a new command. If at any point during this loop the program encounters an error, it will generate an informative error message and then terminate with an error.

Server:

The server begins by validating the commands it is called with. Once it validates the arguments, then it listens for new connections on the specified port. When a new connection is made and a new socket is generated via accept(), the server forks a process to handle this.

In the parent, the server closes the socket generated and waits for a new connection, in which case it will perform a fork the same as above. This means that multiple clients may connect with the server at the same time. It also has the added benefit that if one of the children crashes or otherwise becomes overwhelmed with performing one command, it will not affect other processes nor the parent.

In the child, the child gets its process id and passes the process id and socket file descriptor into the server version of the session() function. In the session() call, the child waits for a new command. When a command is received, it checks if the amount read by the read() call is

0, in which case it cleans up its memory and closes the connection because the client has ended prematurely, probably by using the interrupt (^C) signal.

Otherwise, the child has received a “valid” command and will thus use popen() to fork a process to handle that command. It will use the pipe returned by the popen() call to read data from. As long as it does not read 0 bytes from the pipe created by popen(), then it will continue to read from the pipe 1024 bytes at a time, and then send these 1024 bytes to the client. Once it reads 0 bytes, then it will send the “done sending” message after a brief 250,000 nanosecond delay. The purpose of this delay is to ensure that the previous data is completely read by the client before the client stops reading from it and waits for a new command. If at any point during this process the child encounters an error, then it will clean up its memory, exit from session() and terminate with an informative error message about what went wrong. If there was no error, then after sending the “done sending” message, it will close the pipe, reset the buffer used for fread(), and then wait for a new command.

If the command received is ever “exit”, then the child cleans up all of its memory, exits session() successfully, and then terminates successfully.