Benjamin Wyss 2898025

Jeff Kissick 2414018

10/18/19

Quash Report

The quash shell project was successful. We implemented quash in C++, using only one file called main.cpp. We found that this was a sufficient solution to the problem of parsing each and every command properly. To aid in understanding how we implemented all features, learning the general structure of the program will be useful. The entire program is as mentioned on one file, within this file we first display the file path to show exactly what directory the user is exploring. Then the program waits for user input. The input is parsed via string stream and we handle the user input by taking in commands one at a time via the string stream. The entire process is in a while loop that will repeat once the user input has been properly handled. Every feature required was implemented fully and completely. The features will now be explained in order they appear on the grading policy.

Running executables without arguments: This feature was very simple to implement and was one of the first features added. Once we parse the string stream using if and if else conditional statements to catch commands, if nothing is caught then we know that the user did not enter a specific command. Thus they are looking for execution to take place. We first put the arguments into a vector then fork and use execvpe on the child process to execute the command. If the command is invalid, we display a message stating as such. Once the child process is complete, control of the program will be given back to the user and they are free to enter more commands as they see fit.

Running executables with arguments: For sake of brevity I will simply state that we handle the executables with arguments in the exact same way as we handle executables without arguments. The vector will simply take more arguments and when we fork and execvpe then those arguments are executed from the vector in that call.

Set Home and Path: When parsing specific commands out of the string stream, we look for the string “set” as the first string in the command. We then read in the next string from the string stream, looking specifically for HOME or $HOME or PATH or $PATH. If we find one of these key words we use setenv to set the home or path respectively. If we don’t get any of the 4 commands after set then this will do nothing.

Exit and Quit: Similar to set, when we parse the input from the string stream we look for specific commands, in this case either exit or quit. Instead of executing a command, we simply return 0 and the int main function ends as expected. In case we fail to catch this case, the while loop will end when either quit or exit is the input, that is the condition to end the loop. The program will then simply end but we should never have this happen as the return 0 will catch the quit or exit command for us.

CD (change directory): Again similarly, we parse the input from the string stream and look for cd as the first command. If we see a cd we check if there are any other parameters along with it. If there aren’t then we execute chdir with the users home directory to change their directory to the home directory. If there is another command we execute chdir with the new directory, subsequently changing the directory to the specified location. If either directory, for one reason or another, doesn’t exist we return to the user that we couldn’t find the directory and move on.

PATH: As stated within the set command explanation, if we see the set command we then look for PATH and handle that case accordingly. We execute setenv with PATH.

Child Processes inherit the environment: This is similar to running executables. When we run execvpe the child process takes control of the environment. The parent must wait in a do while loop until the waitpid call returns 0.

Allow background/foreground execution: Foreground execution has been covered already at least twice. For background execution we check if the & symbol is present at the end of the string stream. If it is, we flip a bool to true and continue to do the exact same process as foreground execution. When the process gets executed through a conditional that checks for this bool to be true, the background execution information is printed to the screen. Other than that, the child process calls execvpe and the parent does not have a waitpid so we are immediately taken back to the top of the while loop and the user can enter further commands. The child is run without need to wait for the parent and thus we achieve background execution.

Printing/reporting of job processes (jobs): The background execution will print out as explained above. The jobs command is parsed out from the string stream. This will print out any jobs running the in the background as expected in the format [JOBID] PID COMMAND. Any jobs that start or finish are reported as such.

Allow file redirection: This is handled similarly to background processes. Before execution of a process if we see a ‘>’ or ‘<’ in the string stream we flip a bool to true. From here we pipe the input or output using pipes and dup2 calls accordingly.

Allow pipe: Similar to file redirection we check if the ‘|’ character is seen in the input. We push the command onto a vector to keep track of the number of pipes. We then create forks and pipe the inputs and outputs of the desired commands.

Support reading commands from prompt and from file: We allow for input from commands simple enough, since the program wouldn’t run without this, and with file redirection we allow reading from files as well.

Support multiple pipes in one command: The pipe implementation comes built in with multiple pipes. We simply put the commands in a vector using a for loop, when it comes time to pipe the commands we create forks and dup2 calls to redirect input and output accordingly.

The kill command delivers signals to background processes: Again, we parse the input and if the first string is kill we read in the next string from the input. That should be a kill signal number, if it is we continue to read in from input. We should then see the job number which the user wants to send the kill signal to. If all of this is successful, then we send the kill signal using kill. Should any part fail, we prompt the user as such.

Testing: We tested quash primarily by executing each command over and over again with each change. This makes sense as a lot of the program is parsing apart the commands. For the more robust commands, such as background execution, pipes, and executing with arguments we wrote simple programs that would either continue to run or had an expected output. For instance, to keep a background process running we made an infinite loop program. This way we could test if the job was properly displayed after running and after the jobs command. For pipes we attempted first to pipe one command to another like piping ls to more. After working out kinks we moved on. During the general test cases we simply kept a small list of commands that were working. After realizing about everything was working this testing was abandoned for more stress test related activities. Of course, we ran the program with valgrind and while its not the most efficient, it does clean up all the allocated memory properly.