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 a string stream and we handle user input by reading in commands one at a time via the string stream. This 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 the order that 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 else if conditional statements to catch designated 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 push the string stream's arguments into a vector, then convert the vector into a two-dimensional character array, and lastly fork and have the child process call execvpe to execute the command. If the command is invalid, the execvpe call fails and we display an error message. Once the child process terminates, 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: Executables with arguments are handled in the exact same way as executables without arguments. The argument vector will simply have additional arguments pushed onto it before converting itself into a two-dimensional character array. After we fork, the converted argument array is passed as a parameter to the child process' execvpe call, which executes the command with arguments.

Set Home and Path: When parsing specific commands out of the string stream, we look for the string "set" as the first command. We then read in the next string from the string stream, looking specifically for "HOME=", "$HOME=", "PATH=", or "$PATH=". If we find one of these key words we read in the new environment variable string and call setenv to set the HOME or PATH environment variable respectively. Setting the PATH relative to the current PATH also works, since we parse the new PATH string and look for "PATH" or "$PATH" at the beginning and end of the new PATH string. If we don't read in any of the 4 commands after set, then the set command 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 read as the input. The program will then simply end, but this case should not 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 arguments along with it. If there aren't then we execute chdir with the environment variable HOME directory to change the current working directory to the home directory. If there is an additional argument, 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, we can alter the PATH environment variable with the set command. We also allow printing the current PATH by parsing the commands "echo PATH" and "echo $PATH", which both simply print the stored PATH environment variable. When forking, we have the child process execute commands with execvpe so that we can specify the PATH and HOME environment variables of the child process. Additionally, execvpe searches the specified PATH for executables when a command is not given in the absolute path format.

Child Processes Inherit the Environment: Since the child process executes commands with execvpe, we specify the HOME and PATH environment variables of the child process by passing this information to execvpe as a two-dimensional character array.

Allow Background/Foreground Execution: Foreground execution is achieved by default by having parent processes call waitpid with WNOHANG within a do-while loop after fork is called. For background execution we check if the & symbol is present at the end of the string stream or if a command within the string stream contains an & symbol. If it is, we flip a bool to true and call fork and execvpe as normal, only without having the parent ever call waitpid. This allows the parent process to continue executing quash as the background process is run simultaneously. When a process gets executed in the background, the background execution information is printed to the screen, and when the background process terminates, a SIGCHLD signal handler executes a function that prints out termination information.

Printing/Reporting of Job Processes (Jobs): When a process is run in the background, it is assigned a unique job id, and information about that process is stored in an active jobs list. The jobs command is parsed as a specified command from the input string stream. This command will print out the contents of the active job list, which contains any jobs currently running in the background, as expected in the format [JOBID] PID COMMAND. Whenever a background process finishes, its SIGCHLD signal handler additionally removes its entry from the active jobs list.

Allow File Redirection: This is handled similarly to background processes. Before execution of a process if we read a '>' or '<' in the input string stream, we flip a bool to true and read in the next string stream argument as a file name. From here we call open to create a file descriptor for the specified file and after we fork, we redirect the standard input and/or output of the child process using dup2 calls before the child calls execvpe.

Support One or More Pipes: Similar to file redirection we check if the '|' character is seen in the arguments string stream. If it is, we push pipe information onto a new vector for every '|' character in the argument string stream to keep track of the number of pipes. We then create file descriptors for every pipe needed using the pipe call, and call fork within a for loop that creates child processes for each command that will be executed. Within this for loop, we have each child processes call dup2 to redirect either its standard input, output, or both according to which command it will execute. After all child processes are created and have their inputs and outputs redirected, unused file descriptors are closed and the child processes execute their execvpe calls.

Support Reading Commands From Prompt and From File: We allow for input from interactive commands through a command line interface by default, but quash can also be executed with its input redirected from a file containing a list of commands. When running quash with a commands file, the commands file must be terminated with an “exit” or “quit” command, or quash will not terminate.

The Kill Command Delivers Signals to Background Processes: Again, we parse the string stream input and if the first command is “kill” we read in the next string from the input--which should be a signal number. We then read in one more argument which contains the job number of the background process to send the designated signal to. If all of this parsing is successful, then we send the specified signal using the kill call. 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, input/output redirection, pipes, and executing commands with arguments, we wrote simple test programs that would either continue to run or had expected inputs and outputs to help us test quash. 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 that the jobs command works as intended. For pipes, we attempted first to pipe one command to another like piping ls to more. After working out kinks we moved on to more generalized tests. During the general test cases we simply kept a small list of commands that weren’t working and debugged accordingly. After our general test cases succeeded, we utilized larger scale stress-testing to see interactions between different aspects of quash, such as running a multiple-piped command in the background with output redirection. Finally, we ran the program with valgrind to verify that quash does indeed clean up all dynamically allocated memory properly.