Joseph Pennington (2912079) & Megan Rajagopal (2915076)

EECS 678 Project 1 Report

For this project, we implemented Quite a Shell (quash) using various Unix system calls. We developed the project using C. To test our build, we manually inputted commands into the command line and read commands in from various text files. To debug our quash, we used GDB and Valgrind to see if there were any bugs or memory leaks and used print statements to verify any output or further understand where exactly in the code an error was occurring.

Listed below are each of the features and a brief explanation of how they were implemented:

Running Commands from the Command Line

To run a command on quash, simply type the command into the command line. From there, the input is parsed using the parseInput() function. Then, depending on the command, the main() function calls the appropriate helper function to execute the command.

Redirection

The shell can redirect both the standard input and standard output. The standard input is redirected using the "<" symbol. The standard output is redirected using the ">". However, only one redirection is allowed per command. After the command has been parsed, the redirection symbol is saved and either the input or output is redirected. The standard input will now read in commands from a file, and the standard output will now print the results of the command to a file. For example, the command "ls > a" will print the result of "ls" to file "a".

Running Commands from File

Commands can also be read in from a file. To do this, in the command line type "./quash < filename.txt". The redirect() function handles this case by redirecting the standard input to read in the commands from the file one line at a time. Each line is then sent to the runFromFile() function that calls the appropriate helper function to execute the command.

Running Executables with Arguments

Quash can run executables that require parameters by sending the input to the runCommand() function. This function creates a fork. The child process then calls the execvp() function to handle the multiple parameters. When running executables, ensure that the "./" character is present before the executable name.

Running Executables without Arguments

Similar to running executables with parameters, the initial input is sent the runCommand() function. This function then creates a fork where the child process calls the execlp() function to run the executable without parameters. When running executables, ensure that the "./" character is present before the executable name.

Set HOME & Set PATH
The shell sets the HOME and PATH environment variables by utilizing the setNewPath() helper function that calls the setenv() command on the parameter newPath. If the new path cannot be set, an error message is displayed instead. The path variable can also contain two directories if the directories were separated by ":" in the initial argument. Additionally, if an executable is not specified in the absolute path format, the shell will search the directories in the original path variable unless the path variable has been set.

Changing Directories with & without Parameters
After parsing the input and seeing the "cd" command, main() calls the changeDir() function. In this helper function, if there are no additional arguments given, then the directory is changed to the HOME directory. If there is an additional argument, then the directory will be changed to that directory using the chdir() and getenv() commands. If the directory is not found, then an error message will be printed instead.

Child Processes Inherit Environment Variables
All child processes inherit the environment variables that they were created from. To test this, change an environment variable using the "set" command and printing the environment variable to the screen.

Foreground & Background Execution
In the shell, the process structure keeps track of each process, including each ID, process ID, and its command. These processes are then stored in the array m_process[]. If a command is prefixed with the "&" symbol, then the shell will see this when the command is parsed and run the command as a background process. To do this, the runBackgroundProcess() helper function is called. This helper function creates a child process using fork and then calls the runCommand() helper function.

Printing Job Status
Quash will print the status of each job when the "jobs" command is inputted into the command line. When the "jobs" command is inputted, the printJobs() helper function is called. This function prints the contents of the m_process[] array for the currently running processes. The job status will also print when a command is run in the background. Lastly, the job status will print when a background command finishes.

Single Pipe Command
The single pipe command takes the output from one command and feeds it to another command as the input. This is accomplished by redirecting the standard output of the first command to the standard input of the second command. To run the pipe command, the input, such as "cat myprog.c | more", is parsed. This function sees the "|" character and calls the makePipe() helper function to create the pipe. This helper function parses the first and second commands independently. This function then calls the runCommand() function on the two separate child process that are connected through the pipe.

Quitting & Exiting Quash
To end the execution of quash, type either "quit" or "exit" into the command line or input file. Once the input is parsed, the program exits.