Project 1 Report-Quash

Nicole Maneth and Megan Teahan

7 March 2015

**Introduction:**

The assignment for this project was to implement a shell called Quash--Quite a Shell--that could run executable files, handle background and foreground processes, implement pipes and redirect I/O functions. This report explains our implementation of the project and overall functionality of our completed program.

**Implementation:**

The main part of our program is the parsing function, called **readCommand**. This is where the input string is tokenized and broken into an array of commands. We then check the string for the special characters and specific commands we made our own functions to perform (**<**, **>**, **&**, **|**, **cd**, **set**, **quit**, **exit**, and **jobs**).

File reading (**<**) was implemented by making a copy of the original input and tokenizing it to separate the filename to read from. The string was then processed to remove new line characters. The process was forked, and an input stream was made and the file read in. The stream was closed after reading was complete.

File writing (**>**) was implemented by making a copy of the original input and again tokenizeing it to separate the filename to write to. If that file does not already exist, then it creates a new file, else it opens the already existing file to write to. The string was processed to remove new line characters. The process was forked, and an output stream was opened to the file. The original input command was run, the outcome written to the file, and then the output stream was closed; the process exited.

Background Processes (**&**) work by adding a job object to the list of job objects. The process is forked, placing it in the background. The command is finally run in the background. This allows for the user to continue to run commands in the foreground. This background job is also stored in the jobs array, to keep track of if the job is running or not for the user. When the process has finished, it alerts the user through the console.

Pipes( **|** ) are implemented by tokenizing the **inputCopy** string to separate the different commands to run. Two process ids are made and forked. Each process has the whitespace removed before running it through the **readCommand** function (which is the command that parses the input). Then with readCommand, those different processes are run in the correct order, reading from left to right. This recursive-like setup allows multiple pipes to be fed into the program.

**Cd** is a function that can either take in a string argument or take no argument to change the directory the shell is working in. If a string argument is given, it is the path of the directory to change to. If no argument is given, it changes the directory to the HOME path. There are error handlers to check if the directory is a valid pathname. If it is an invalid pathname, then Quash warns the user of this.

**Set** takes in a string argument that contains the variable to change as well as its arguments (ex: PATHTOSET=/whatever/path/user/wants). It first tokenizes the string to break up the variable (environment name) from its arguments. It then sets the environment variable using the setenv function by passing in the environment name and its arguments. This function also contains error handling, for cases like if the path does not exist. Quash warns the user of this invalid path.

**Quit** and **Exit** aren't implemented as their own functions; the command line input is parsed, and there is a compare to see if it contains either “quit” or “exit”. If either of these are true, exit(0) is called.

**Jobs** contains a loop that searches through the jobs array that stores all jobs. Each spot is first checked to see if the job is alive. If the job is alive, it is printed out in a formatted manner. If no jobs are in the array, it prints a message to the console that there are no jobs.

**Executables** are taken care if none of the other special cases are matched. These executables have the ability to run with or without arguments, depending on what the executables require/what the user inputs. We added in a special case for printing out the path because we used it when debugging our program. We decided to leave it in the program in case the user wants to call that command.

**Kill** works by taking in two arguments of the job the user wants to kill: job id and signal number. It searches for it in the jobs array and then calls the built in kill function on it. If the job couldn't be found, it alerts the user.

The **main** function prints out a formatted prompt that contains the current working directory. Because we used the readline library in the prompt, users are able to use tab completion when typing in their commands. This makes changing directories much quicker.

**Testing:**

We used an iterative approach to this project. The first functions we made were the **runCommand** (which parsed the user input) and **cd** functions. **Set**, **exit**, **quit**, and **jobs** were done next. Ending with file io, pipes and background jobs. As each function was created, we tested it with a variety of input to make sure it had the correct functionality. This was how we discovered that our pipe implementation worked for multiple pipes.

**Conclusion:**

The software functions as it should. This project took more time than anticipated because we spent a lot of time searching definitions of functions and deciding which versions (as in the case of executables) we should use. Both of us, also did not have a lot of experience programming in C. There was a bit of a learning curve when figuring out libraries and the semantics of C. We were very lucky to find the GNU library which gave many details and information on the functionality of a shell. We also referred to our previous labs to look at examples of pipes for running commands. We split the design of the software into different functions for each command to keep the program organized. The shell is fully functional and we are very happy with the progress we made