Nick North & Dawson Frick
EECS 678 Project 1: Quash
Report

The Quash shell that we designed was created using C. We tested Quash by inputting commands manually to Quash's command line and through reading file input. To debug the program, we used the GDB debugger and print statements.

Listed below are the features we implemented.

Running executables with and without arguments
The shell has the ability to run executables with and without arguments. To do this, we send the input to our exe function in which we create a fork. Inside the child we call execvp for programs with multiple arguments and execlp for programs with no arguments.

Set HOME and PATH
Quash can also set the HOME and PATH environment variables. This is done using our setPath function that takes in the string *path* and calls setenv to set the path. If unsuccessful, it will print an error.

Exit and Quit
Quash implements *exit* and *quit* by using the exitQuash function to check if the input reads as "exit" or "quit". If this is true, then Quash will exit.

Change Directory
Our shell implements *cd* by first verifying that the input contains "cd" with the use of the checkChangeDirectory function. If it is true, then the function changeDirectory is called. This takes the name of the directory as the input string. If there is no directory (or NULL) then chdir(getenv("HOME)); will be called. Otherwise, the function will call chdir for the directory if it exists, or also change the directory to HOME if the directory is "~" or "HOME".

Path
The PATH variable functions as intended for all processes. The default path is used to search for executables unless the set command is used to change the path. An error message is displayed when an executable can't be found.

Child inherits environment
Child processes inherit the environment they are spawned from. This can be checked by changing an environment variable using set then executing a program that displays that variable to the terminal.

Background processes and jobs

Quash allows background and foreground execution of processes. The struct *Process* is used to keep track of a process. Each *Process* has an id, pid, and commandString. For Quash, the processes are contained in an array called myJobs. The command *jobs* allows us to see current processes. If an input command contains an '&', then it records that it was found with ampersandFound and sets the ampersandIndex. Then the runBackground function will be called. This takes the command that was input and forks a child process to run it in by calling exe(cmd). Before and after running the command, the process status is printed. In the parent, the new process information is added into the struct. When the background processes finish, they will be removed from the myJobs array.

Redirection

Quash implements redirection for both stdin and stdout. Stdin is redirected using the "<" symbol. Stdout uses the ">" symbol. Only one of these symbols is supported per command line. The redirect function is called after the command line has been parsed for a redirect symbol. Stdin opens a file for reading and stdout opens a file for writing. The files are both opened inside of a child process which calls the exe function to execute the command to the left of the redirect symbol.

Single Pipe

The use of a single pipe between two commands is supported by the shell. This is done by redirecting stdout of the left command to stdin of the right command. After the command line is parsed for the "|" symbol, the function makePipe is called. The left and right commands are parsed independently inside of makePipe and fed to the exe function in two separate child processes that are connected through a pipe.

Reading Commands from File

Commands from file are executed by typing the following: ./quash < file.txt. Inside of the redirect function's first case for redirecting stdin, there is a while loop that reads 1 line of the file per iteration. This line is sent to the runCmdFromFile function which decides what to do with it based on the type of command it is.

Testing

Quash was tested both manually and by using the files that were provided on Blackboard. Errors in code were found by printing variables and using GDB to locate where segfaults occurred.