CSCE 313
PA2 Report
Jeffrey Xu
527008162
09/20/20

1 Shell Structure

Everything for the shell could've been coded in the main function, however due to the size of this project, it was decided to modularize everything. Processes like parsing, executing commands and checking for background processes were put into functions that allowed for more streamline code.

Within the shell, a *while* loop continously asks for user input until the input equals the exit token. The shell itself checks for a couple of special cases of input such as an empty input (user simply pressed enter before entering anything), the exit case and the case that the input is a background process. After checking for these special cases, the shell then parses the input using *splitOnQuotes* and *parsePipeProcesses*. These two functions result in a double-layered vector that contains at each index the tokens for each pipe process in the input. This double-layered vector is then passed into *runCommands* where the tokens are evaluated and executed.

After all of the parsing, execution and background process checking, the shell updates the previous path, current path and the date and time. The shell then prompts the user again for input and the loop continues.

2 Parsing

The majority of the parsing is done through two functions, *splitOnQuotes* and *parsePipeProcesses*. *splitOnQuotes* does normal parsing but doesn't destroy the structure of quoted strings. It accomplishes this by having flags that detect whether a quoted string is currently open and being parsed. *singleQuote* and *doubleQuotes* are the flags that detect whether this is occurring. Both cannot be true at once. During the event that the current token being parsed isn't a quoted string, the parser parses it normally, splitting at spaces and the pipe delimiter.

parsePipeProcesses converts the vector containing all the tokens from *splitOnQuotes* (includes quoted strings, other tokens and pipe delimiters) by splitting at each pipe delimiter. It accomplishes this by having a double-layered vector which is what will be returned and a buffer vector for storing the current pipe tokens. If a pipe delimiter is hit or a pipe delimiter is contained within the token,

then the buffer is processed and pushed onto the returning vector and the buffer is cleared.

3 Executing Commands

The majority of the shell depends on *runCommands* which properly processes each pipe command and executes them. The function has the same structure as the skeleton code with more additions to account for special cases.

The function starts by checking if the current command is a call to change the directory. If so, the function calls the function *split* and uses the system call *chdir* to either change the the requested path or goes to the previous path (passed into the function).

After checking for changing directories, the function then parses the case that both IO input and IO output redirection is present in the current pipe process. Since the input needs to be before the output redirect, the function can easily parse the process into the command, input file and output file. It then executes the function.

After completing all of these checks, the function then performs piping. Piping was a tricky component to implement as it needs to account for single IO redirection and sometimes background processes. After calling *fork*, the child process checks for single IO redirect and properly redirects the file descriptors if needed. Then, the function gets rid of quotes and executes the current pipe command. If the execution fails, the process id is pushed onto the background processes vector and the function exits. In the parent process, the function only calls *wait* on the last pipe process if it isn't a background process. All other processes have their pid pushed onto the background processes vector and the pipe and IO redirection file descriptors are closed. The standard input and output is also redirected back to the original.

YouTube Link: https://youtu.be/7l_HdNqhcWk