System Programming Lab

2021315385

이건 / Gun Daniel Lee

PA 2 Report

In this report, I will briefly explain the details of my pa2 source code.

Disabling termination on Signal

To start off, I used the *signal()* function to prevent *SIGINT* and *SIGTSTP* from terminating the program. If the program detects any of the two signals, they will be redirected to a signal_handler function, a void function which only returns. This will allow the code to run despite receiving the signals.

Separating the Commands:

Afterwards, the main section of the mini shell code is within an infinite while loop so that it will continually ask for inputs, similar to an actual shell. Within the while loop, the program will request for one line of input. Once the input is received, then the code focuses and splitting the input to retrieve all the commands and files. To achieve this, the input will be divided and stored into a 2D character array named *cmd*, each row index storing a command. One command can be found using a string token on the pipeline character (‘ | ’). Every time the input contains a pipeline, the code will split the two commands and separately store them in the *cmd* array.

Retrieving files

Once the commands are separated, the code focuses on finding possible files. Generally, this can be separated into two cases. The first case is when there is only one command. In this case, the code will check the single command string to find for input and output redirection. The second case is when there is a pipeline, in other words, there are two or more commands. In this case, the code will only check the first command for input redirection and the last command for output redirection. If input or output redirection is found, the name will be stored as *file1* and *file 2,* respectively. Additionally, if either are found, a flag variable called *file1exist* and *file2exist* will be set. In the case of *file2exist,* it will be set to 1 if the redirection is the ‘>’. If it is ‘>>’, it will be set to 2.

Validating Commands

Once all the required information has been retrieved, the code will check if all the

commands are allowed. If not, the code will exit with an error message. The method of checking all the commands is by using string compare. After making 2D arrays containing the possible commands, the code will compare the command array with the possible commands. If string compare fails for any of the commands from our input, the code will print an error and ask for another input. Otherwise, the code continues.

Executing Commands

The executing section was separated into two case, one command only and two or more commands. If there only is one command, the file will take the only command string existing. The code will then parse this string by the spaces due to the options and arguments possible. Once the string has been parsed, the code will use forking and execv to run the command. In the case of file redirection, if file 1 exists, then the code will use file 1 as the input using a pipe. In the case of file 2, if it exists, the code will use file 2 as the output. If file 2 is the rewriting case, the file 2 will be created and opened using the flag O_TRUNC, which is to rewrite a file. If file 2 is the append case, the O_APPEND will replace O_TRUNC. After the redirection, the code will execute the command.

In the case of two or more commands, the code used a different method which may not be consistent. Instead of directly connecting one command's output to another command's input, the code goes through a file in between. Basically, the code outputs a command's result to a random file, then the next command will use the random file as its input. This will continue if there are several commands using a for loop.

All the executing commands that had to be self-made were based off of the pdf and the given functions that had to be used.

This summarizes the main source code for my PA2. Hopefully this report helps with understanding the code.