Course: ECE 40800

Name: Kefil Tonouewa and Ravyn Dickinson

HMW: #2

Date: 02/20/2020

**Introduction**

The objective of this homework is to show us the various ways processes can be issued and handled in the terminal. In order to attain this objective, we were tasked to design a C program to serve as a shell that accepts user commands and then executes each command in separate processes. This homework also stretches on the use of some key functions such as fork(), exec(), wait(), dup2() and pipe(). After many lines of code and some problem solving along the way, we resulted with a program that runs similarly to a Unix shell. Some challenges that we face were mainly due to the fact that our child process was not returning properly as well as understanding how to manipulate the argument arrays.

**Methodology**

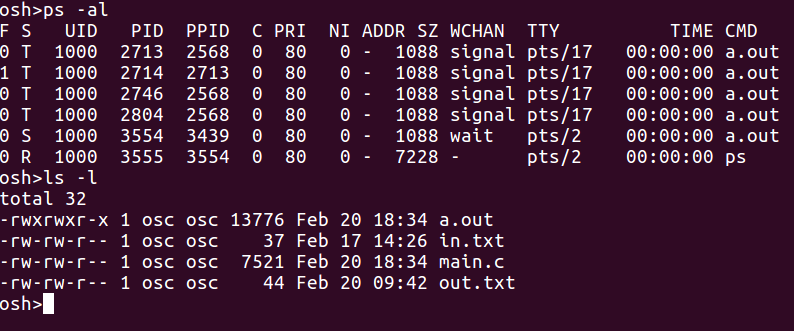
We accomplished this assignment by using a divide and conquer method. Since building the shell was an enormous task by itself, with the help of the given directions, we started small with a simpler program and built on it with each new feature. First, we wrote a C program that just takes user input and yields the expected results. To accomplish that step we took user inputs and tokenized them in order to extract the arguments. Then, a child process was created and it executes the command entered by the user using the execvp function. This means that those arguments were passed to the execvp() function to get the desired results (Figure 1). The method of tokenizing was utilized for all features in the project where user input was considered. Also, there were other smaller features like the exit command.

Next, we created a history feature that allows the user to run the command that he/she previously ran. In order to complete this task, we saved the previous command given by the user into a separate variable. The command was tested to see if it contained “!!” and if so, the saved, previous command was executed using the execvp() function (Figure 2). This feature also takes care of the case where there are no commands in the history.

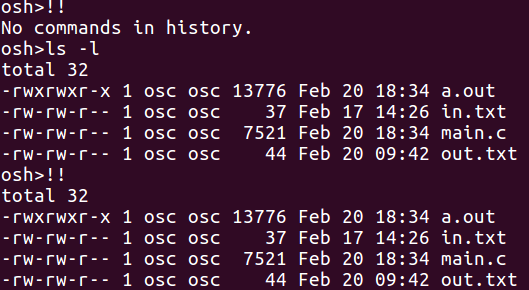
After that, we created a feature that allows the user to redirect input and output. To do that, we parsed the user command to verify if there was a “>”(output) or a “<”(input). For example, if the user entered an “>” in the command, there would be a process to write standard output to the said file. A big part of this feature was the dup2() function, which took in two integers and allowed for the duplication of file descriptors, including standard output and standard input. Therefore, based on the result issued from our parsing process, we opened a file and used it as an input to read or as an output to write using the execvp() function to execute that command (Figure 3).

Last but not least, we implemented communication via piping. To do so, we had to pass the output issued from a command as an input to another command. To accomplish this task, we first verified if the command entered contained a pipe (|). Upon successful verification, a pipe was created via the pipe() function. Much data manipulation was done to get the arguments on either sides of the pipe into their own separate argument arrays so they would not be confused. After that we executed the command located before “|” and wrote it’s result to the pipe. Closing the pipe was a very important step. Next, we used the output in the pipe as input for the command located after “|” (Figure 4).

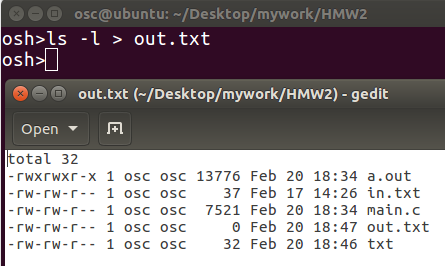
**Results**



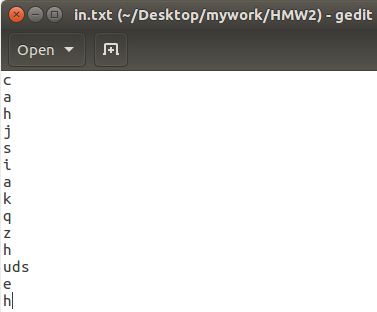
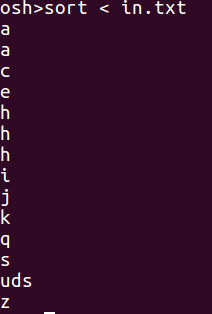
The above figure illustrates the result issued from executing a regular command in a child process.



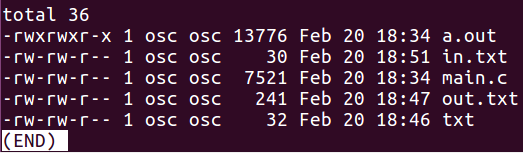
The above figure illustrates the result issued from using the history feature. Notice that when there is no previous command entered by the user, “No commands in history.” is echoed to the terminal.



The above figure illustrates the result issued from redirecting output.

The first figure shows the input file fed to the command that supports input redirecting and the second figure shows the output issued from using redirecting input.



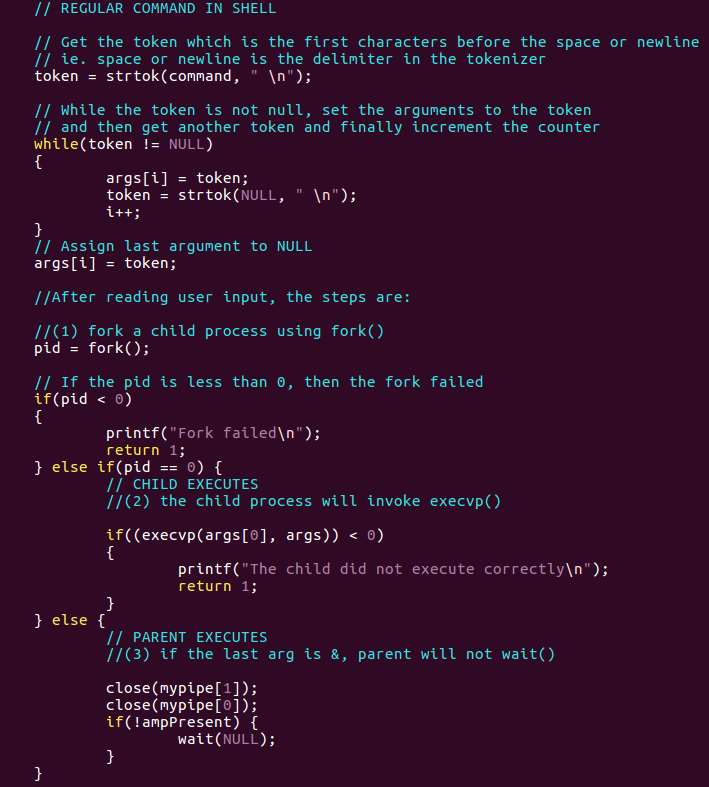
The above figure illustrates the result issued from using communication via a Pipe.

An analysis of the results includes the fact that there was more difficulty with the last piping part. The printf statements for debugging worked well for actually seeing what was going on with the code. The features work in terms of the various parts - executing a command, providing a history feature, redirecting input and output and communication via a pipe.

**Summary**

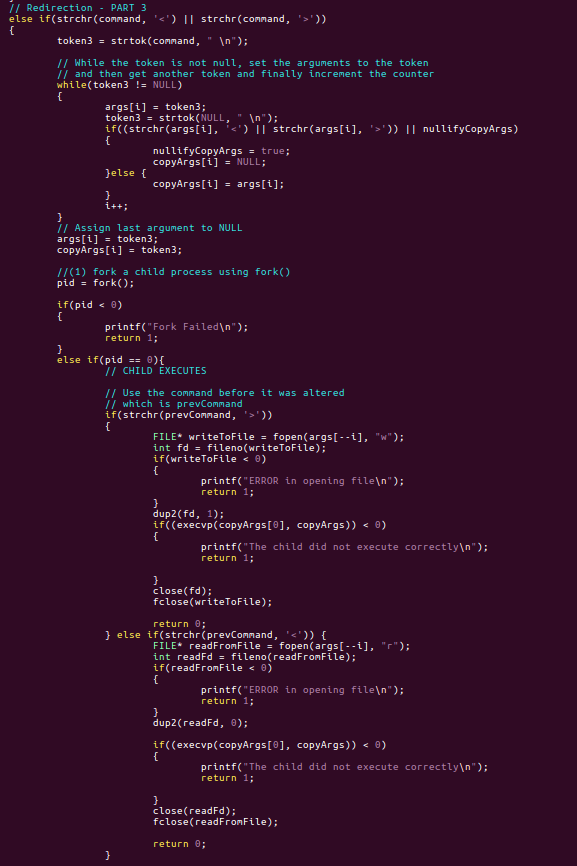
Overall, this was a decent project in which students learned to create child processes using the fork() function. Also, students learned how to use the exec() function to have a process execute a command line command. Another important lesson was the dup2() function as it was very useful in redirection. Finally, the pipe() function was useful for communication via a pipe. If more time was given, improvements would include making code more modular.

**Appendix**

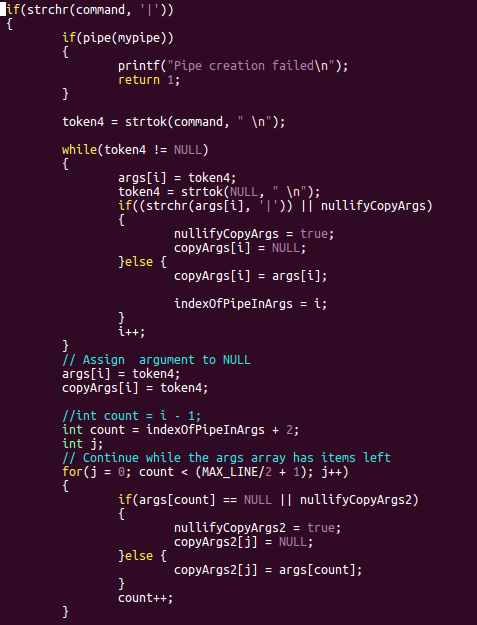
**Figure 1**: Executing Command in Child Process.



**Figure 2**: Creating History Feature



**Figure 3**: Redirecting Input and Output



**Figure 4**: Communication via Pipe