Arjun Bhagat ID: 917129686
Github: smeerj CSC415 Operating Systems

Assignment 3 – Simple Shell with Pipes

Description:

This assignment was designed to help us understand the use of fork(), exec(), wait(), pipe(), and dup2() functions, read and tokenize user input, use an array of pointers, and redirect the standard input and output with pipes.

This program is a simple shell that has piping implemented.

Approach:

For starters, I took a new approach to my write up and my code commenting for this assignment. I have lost points in both of these categories for both of the previous assignments. I asked around among my classmates to find out who was getting a better score for these categories and asked them for help. I got a lot of good pointers and I realized my understanding of the write up and code comments was pretty flawed in general.

Now for my approach to this assignment, I'm going to break it into two parts: gathering the user input, and making the shell.

User Input

First I set up a buffer for handling the input. I set it to the specified size as requested in the readme of 159. I recognized that the last byte of this buffer would have to be the null terminator meaning I would generally have 158 bytes to use.

I then set up some global constants of my maximum values for the buffer size, the maximum amount of commands, and the maximum amount of pipes. The maximum amount of commands considering every other character could be a space is the maximum buffer size divided by 2, and minus one to leave space for the null terminator. Then the maximum number of pipes would be the maximum amount of arguments divided by 2 considering there could be a pipe after every command.

To get the input from the user I decided to use fgets() which allows me to take a specific amount of bytes from the standard input (stdin), and adds a null terminator to the 159th byte of the buffer. There are two edge cases I realized I needed to consider: the 'end of file' condition and a general reading error. I conditionally check if fgets() returns NULL to check for these cases, and we can proceed if not. In the case of EOF, we want to 'exit gracefully without reporting an error' as specified in the readme. I used feof() to check if it's at EOF, and if not that means it's a real error and I print an error message.

Arjun Bhagat ID: 917129686
Github: smeerj CSC415 Operating Systems

Making the shell

Next I went on to tokenize the commands. I decided to tokenize by pipes instead of spaces first which would allow me to break down each command and store them, and I did this using strtok() with the '|' character as the delimiter. I initialize an array to store the tokenized command strings, and a counter to hold the amount of commands that were parsed.

I then initialize an array of file descriptors for the pipes using pipe() for each command. This array is the size of the amount of commands minus one because that is the amount of pipes there would be, multiplied by two since there is a file descriptor for both ends of the pipes. If pipe() returns a negative value, the pipe creation failed and the command can't be executed so we print an error and stop.

I now need to create a process for each command in the command array and we can do this using fork(). I handle an error immediately if process ID returns as -1 indicating the child process was never made. I then begin rerouting the input and output of each of the processes through the pipes to account for multiple commands and pipes and we can do this with dup2(). I rerouted the standard out (stdout) of the process to the write end of the pipe, and then rerouted the read end of the pipe to the standard in (stdin) of the next process.

There are some special cases when piping commands, specifically the first and last command. The standard in of the first command doesn't need to be rerouted to a pipe, and the same goes for the standard out of the last command. This also accounts for use of single commands not needing to be piped. This was a simple fix: I just checked the argument count to confirm whether we are currently on the first or last command within the loop and don't reroute the input or output respectively (don't call dup2()).

Now all the pipes in the child processes are closed to free up the file descriptors, and we can do this with close().

To actually run the commands, they then have to be tokenized by spaces to separate arguments of each command to hand to execvp(). I initialize another array for the commands and tokenize the command and its arguments with strtok() using space as a delimiter. Because this is an array that's being handed to execvp() it needs to be null terminated so that the end of the array is recognized. I add the null terminator and hand the array to execvp().

We now need to close the pipes of all the parent processes as well as they aren't needed anymore and this is also done in a loop and using close().

Lastly for each process we need to wait for it to complete and print out the process ID and return status, and we do this in another loop using wait().

Arjun Bhagat ID: 917129686
Github: smeerj CSC415 Operating Systems

Issues and Resolutions:

I struggled a lot in understanding a lot of the functions that I needed to use and that were specified to be used for this assignment. I was constantly referencing the manpages for wait(), fork(), execvp(), dup2(), and pipe(). This helped me understand how to use these functions, what library they are contained in, and how they behave in general.

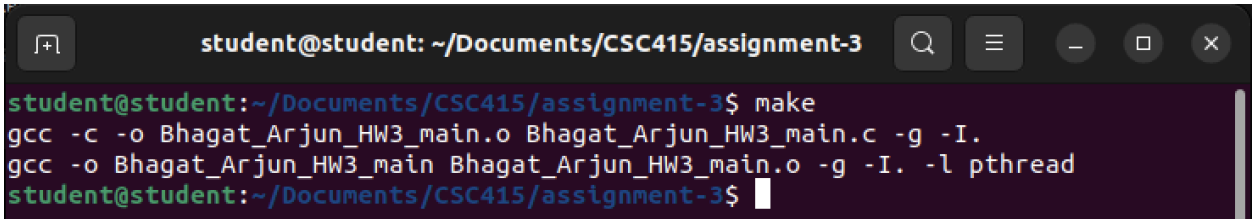
The first main issue I ran into was that every time I tried to run a single command the shell would print the command couldn't be found. I double checked all handling of edge cases in my piping and I didn't find the issue there and I was at a loss. I asked a classmate to see if he could help me figure out where the issue was and he explained to me that with the use of fgets() to get the user input, it would add a new line ("\n") to the end of the input. I then remembered that the command needs to be null terminated in the command array when being handed to execvp() and this was where the issue was coming from. I fixed this by replacing the last new line with a null if it exists before handing it to my executeCommands() function. This solved the issue and single commands were now functional.

The next issue I had was that after completing the program, I realized I had forgotten to check for the exit command. I had assumed it was a linux command and didn't know it was built in until testing in my shell once I saw that it was specified that the shell needed to account for this command as well and found that it wasn't working. This was an issue because I would need to pass that the shell needs to be exited from the executeCommand() function and I wasn't sure how to do this. I could have added a return value to my executeCommand() function for this specific case but it felt like a large change for a small and specific case, and I knew there had to be a simpler way. I realized that in the way I replaced the new line at the end of a command with null, I could similarly check if the command is 'exit' within my main(). I did this using strncmp() to compare the first 4 bites of the command with the string 'exit' and if it came back true, we exit the loop.

Arjun Bhagat
Github: smeerj

ID: 917129686

Screenshot of compilation:



Screen shot(s) of the execution of the program: