CS433: Assignment 2 Report

Rusty Dillard

Submitted Files:

- prog.cpp

How to Compile/Run the Program:

To compile, command used: make

To run, command used:

- *./prog*

1 Overview of Program

1.1 Results of all tests when compiled in GradeScope:

My program ran prog.cpp with no errors and passed all tests for each of the components required by the assignment prompt. Dozens of runs were made and any bugs found were handled. When turned in to GradeScope, the program ran perfectly well.

1.2 Choice of parsing algorithm:

The algorithm I chose ran the command through a for loop and used a switch statement to determine each case. The four cases that ended up working best for my algorithm were a whitespace character ‘ ’, a tab ‘\t’, a new line ‘\n’, and the default case. To keep track of the address of the beginning of the next argument, an integer is initialized to -1 – this value is used as a flag – and updated to the current value of i in the for loop.

The switch cases take any whitespaces and tabs and stores the C null char ‘\0’ in their place, if no argument has been found before the whitespace, the for loop just continues to the next index in the command and resets the found index to -1. However, if the found index is anything other than -1, the token stored at the address starting from the current found index in the command is stored into the argument array of char pointers. The new line char ‘\n’ does much the same, except when it's finished, it doesn’t need to reset the found index.

The default case updates the found index to the current value of i and checks for the wait flag character ‘&’. If the ‘&’ is found, the boolean to tell the program whether the parent process should wait for the child process to finish is changed to false, so that the parent will just execute. At the end of the parsing algorithm, the index associated with the number of total arguments found is set to NULL, to tell execvp() where the end of the arguments array is and the counted number of arguments is its return value.

1.3 The Unix shell program:

The Unix shell runs on a while loop that can be exited once the user types in "exit" anywhere in the command line. At its simplest, the shell parses a user's entered commands into tokens, stores them into an array, forks a new process, and executes the commands. The results of the command will be displayed in the console. For this assignment, however, that would be too simple and could be completed in under an hour with proper research and knowledge of C and Unix functions. We were tasked to add the following functionalities: detection of a "don't wait" flag '&', detection of a history command "!!", detection of input and output redirection flags '<' and '>', and detection of a pipe flag '|'.

The wait flag was simple enough and was handled, as previously mentioned, in the parsing algorithm. The only part of it that touched the main function was the "should_wait" boolean, which, when true, executed the wait() function. Something that I noticed and was intrigued/annoyed by was the "osh>" printout that happens at the top of the loop wouldn't print out until the user's next command was entered, but that's just a byproduct of the parent running straight through without waiting for the child process to complete itself.

For the history command "!!" to work, it was a simple case of storing the current command from each run into a history buffer. The functionality needed to have a failsafe in it that accounted for there being no previous commands. This was achieved by a variable initialized to 0 in the main function that would keep track of the number of commands entered. The only time that number is updated is after a successful command has been executed and the loop has reached its final line of code before returning to the top of the loop. If the conditions are met that there has been a successful execution before and that the "!!" command was entered, the program will transfer the previous command from the history buffer into the command to be parsed and execute whatever that command was.

The redirection flags were a bit more complicated, but not too bad. I added a function to dive into the arguments array after it had been parsed and check for either of the redirection characters. If one the input redirection character '<' was found, the appropriate file would be opened as a read-only file, the standard input would be saved from dup() and dup2() would become the new stream. With output redirection, the thought was the same, but with different flags in the open() function to create the file if it didn't already exist, make it write-only, and truncate the output. The commands were then executed just like a normal command, only the input/output was redirected from or to an external file.

Piping… Now piping was a bear to handle. To pipe, there needed to be two new char pointer arrays: one for the left side of the pipe and one for the right side of the pipe. After the fork had been made, the pipe() command is called with an array of two data streams and another fork is called to create a process grandchild. The standard output is gathered from dup(), and the new output stream is one of the slots in the piping array. The command left of the pipe is then executed and the program moves into the second phase

where it saves the standard input and uses dup2() with the other slot of the piping array to create the new input stream and finally executes the right side of the pipe.

Once any of these cases is completed, the loop can finally reach the end of its instructions and reset the standard input and output, if they were changed. That only happens in the cases of redirection and piping. Something that was encountered in GradeScope but didn't have any effect on my end when running the program was that if I didn't enclose the data stream resets into an if statement executing under the conditions that redirection or piping had occurred, GradeScope would throw an error stating that I had a "Bad file descriptor" in the history function.

1.4 Files and What they Do:

1.4.1 **prog.cpp**: This is the only file in this program. It contains global variables, the main function with the while loop to drive the mock Unix shell, and the following few functions to tidy up the code a little bit.

1.4.1.1 int num_commands = 0:

- Command counter.

1.4.1.2 int file_index:

- Index of filename in args.

1.4.1.3 int in, saved_stdin, out, saved_stdout:

- Ints for file operations.

1.4.1.4 int pipe_index:

- Index of pipe in args.

1.4.1.5 bool should_run = true:

- While loop condition.

1.4.1.6 bool input_redirection_flag = false:

- Input redirector (<) found.

1.4.1.7 bool output_redirection_flag = false:

- Output redirector (>) found.

1.4.1.8 bool pipe_flag = false:

- Pipe (|) found.

1.4.1.9 bool empty_command = false:

- Command entered was empty.

1.4.1.10 bool check_hist = false:

- Check for history.

1.4.1.11 bool should_wait = true:

- Wait (&) found.

1.4.1.12 int main():

- This function contained the majority of the code for the program. A set of char pointers and char pointer arrays for commands and arguments, an int to keep track of the number of arguments, a pid_t to store the process ID, and a couple states to help keep track of

which piped instruction was executed. All of this aided in the while loop's stable execution until the user entered the command "exit".

1.4.1.13 int parse_command(char*, char):**

- This function was described in detail earlier in section **1.2**.

1.4.1.14 void reset_flags():

- This function restores the initial values of the booleans should_wait, input_redirection_flag, output_redirection_flag, pipe_flag, empty_command, and check_hist.

1.4.1.15 bool exit_condition(char):**

- This function parses the arguments array sent into it and checks for an "exit" command entered anywhere in the command line by the user. If "exit" is discovered, the function sets the value of should_run to false and returns a value of true.

1.4.1.16 void swap_command(char*, char*, char*):

- This function is used to copy what's in the history buffer into the command to be parsed.

1.4.1.17 void check_redirect(char):**

- This function checked for redirection commands '<' and '>' that had been parsed into the arguments array. If either are found, the appropriate boolean flag is set to true, so the program can execute the proper redirection feature.

1.4.1.18 void check_piping(char):**

- This function searches for an argument that consists of the piping character '|' with a for loop. If this is found, the pipe_index global variable is updated to the index it was found at and the pipe_flag boolean is set to true.

1.4.1.19 void boot_up():

- This is just meant to be a nod to the movie 2001; A Space Odyssey and is left commented out in the main function for expediting the testing, since it uses the sleep() and usleep() functions to simulate a computer booting up.

Lessons Learned/Re-Learned:

I took the opportunity to sharpen my skills with Unix commands and learned how piping and redirection worked in a simple shell environment. A lot of research was needed and I'm glad we were given three weeks for this assignment because I needed to do a lot of technical googling.

2 Final Words and Findings:

2.1 References/Resources:

When I began the assignment, I mostly just followed examples in the zyBooks readings in Chapter 3 section 3.

- Section 3.3 - CS 433: Operating Systems | zyBooks

In order to gain knowledge of how to execute the piping and redirection, I needed to learn how to use the dup() and dup2() functions, which I picked up mostly reading GeeksforGeeks.

- dup() and dup2() Linux system call - GeeksforGeeks

2.1 Conclusions

In conclusion, the programming assignment on creating a Unix shell program helped me to better understand the inner workings of operating systems and how they manage parent and child processes, piping, input and output redirection, and history features. This assignment also allowed me to gain practical experience in implementing these concepts, as well as honing my programming skills. Overall, this assignment was a valuable learning experience that gave me a deeper understanding of the critical role that commands like dup(), dup2(), execvp(), and pipe() play in the realm of a command line Unix operating system shell.