CIS 415 Operating Systems

Project 1 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Josh Jilot

UO ID - jjilot

951863225

Report

Introduction

In this project I was tasked with creating a "pseudo-shell"–essentially a simpler version of the Unix shell. The Unix shell takes command-line input from the user and can do simple tasks like move files, display the contents of a file, and list the contents of the current directory. While it is very simple to use, this simplicity comes with the caveat that the user must understand exactly how the shell functions–this isn't Python! Because there is now hand-holding, the shell will often spit out errors with little insight into what was improperly inputted because it is expected that the user already knows the correct syntax and use-cases.

Because the shell is so uncomplicated it makes a perfect first project for this class. My job for the last three weeks was to implement a simpler, single-threaded version of the shell using lower-level C commands. While this project may seem simple at first, (the shell is so uncomplicated, how hard can it be!) it required much more knowledge of how Unix commands, C file I/O, and unfamiliar functions work than I ever expected. I had never gone to such a deep level, using the man page for Unix system calls to learn how the Unix commands I assumed were already low-level actually use even more commands I had never heard of to get information and display it to the user. This project tested my knowledge of C and Unix in a new way and forced me to learn many new techniques and processes that I had never even thought about before.

Background

As I will detail in the Implementation section, I chose to make my entire program function with file mode and interactive mode without requiring me to rewrite any functions for one specific mode. Because of this, my implementations of string parsing and some functions had to be altered to what I originally planned. As well, I was very used to doing file I/O with specific functions like fprintf and fputs that were not allowed in the command.c file. This forced me to learn new methods of reading from and writing to a file–namely file descriptors and the write and open functions.

As well, I had never coded unix commands in C before, so I had to study up with much help from the Unix man page provided in the project description. I found out that most commands were actually daily straightforward, with the exceptions being commands that needed to print something. These required me to handle reading and writing to files and the command line with newly learned functions and tools (such as file descriptors). I also had never attempted to traverse directories in C, and this project taught me about tools like the access and stat functions which allowed me to check if files or directories were accessible and the correct format for what the function was expecting.

Implementation

The major "something nifty" that I used in my implementation was the previously mentioned goal of not having to implement anything twice (once for each mode). I achieved this by writing all command line input to a temporary file and then creating a function called input that would handle all string parsing. This function assumed that the input was to be parsed from a file and to avoid redoing any logic I passed my temporary file as the input. This allowed me to do all string parsing through the same logic, regardless of if the shell was in interactive or file mode which greatly reduced the number of functions and code required. The loop that runs for every interactive mode prompt is shown below, as it is what enables the entire program to work with minimal redundant coding. I needed these file pointers to be accessible across files so I defined them with the extern keyword in a header file and linked main.c and command.c together in my makefile.

```
while (repeat != 2) {
    // Create tmp file
    f_in = fopen("tmp", "w+");

    printf(">>> ");
    fgets(inp, sizeof(inp), stdin);
    fprintf(f_in, "%s", inp);
    rewind(f_in);
    repeat = input(comm_arr);

    // close tmp file
    fclose(f_in);
    remove("tmp");
}
```

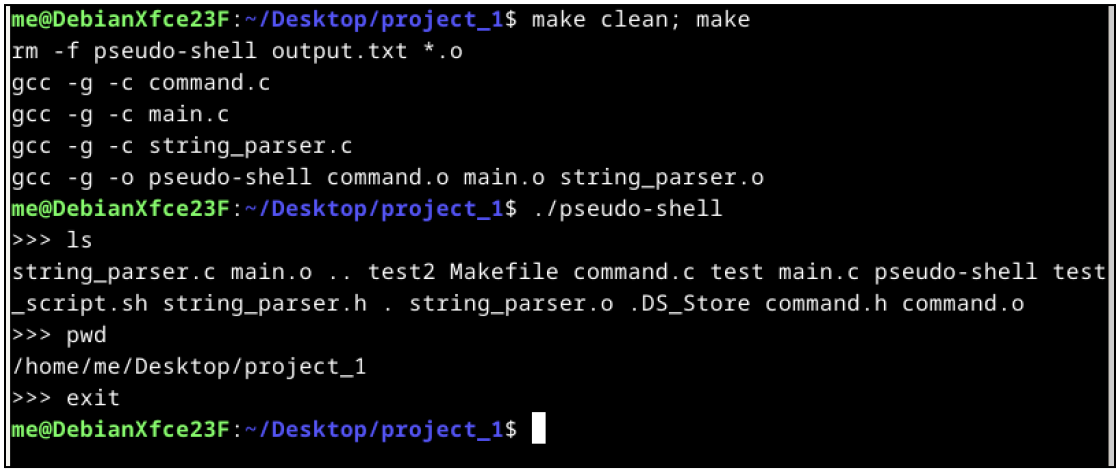
However, because I used a file pointer for all input and output this meant that I needed to convert these pointers into file descriptors to be used in command.c. I accomplished this as shown in the picture below by turning the file pointer into a file descriptor. To handle cases where the executable may have been moved and the file pointer may be pointing to the wrong stream, I added a line that sets the output file descriptor to STDOUT_FILENO if the program is being run in interactive mode. I used a while loop to write to either stdout or the output file with the newly acquired file descriptor as shown below.

```
int f_outd = fileno(f_out);
int f_ind;
f_ind = open(filename, O_RDONLY);
if (f_ind == -1) { error = 1; return; }

if (f_out == stdout) {
f_outd = STDOUT_FILENO;
}

/* Read in each line using read() */
char buffer[1024];
size_t bytes;

while((bytes = read(f_ind, buffer, sizeof(buffer))) != 0) { write(f_outd, buffer, bytes); }
```

The logic of my program was daily simple due to the fact that I avoided creating separate functions for file and interactive mode. I would take user input either from a file or from the command line to be put into a file. Then I ran that input file into the string parser function that we created in lab. Every time this function parsed a command it would store each token in an array of strings and then call a new function with this array as an input. This function checked the value of the first token (the command) and would call the appropriate command from command.c, while checking if the given arguments are legal. The only unique condition was if the exit command is called, in which case it cleanly frees all memory, closes all files, and terminates the program.

Performance Results and Discussion

My project runs perfectly in interactive and file mode. It compiles with no errors and has no memory leaks. It passes all test cases from the test script in the VM besides three. The first two are pwd and cat which it shows printing the correct results. The other is cd because the cd test depends on the pwd test passing. I have spoken to my GE and confirmed that my project is working correctly; he believes that I should be given full points for these test cases. Furthermore, these test cases pass with the correct output when run on other machines. Below is an example of the program compiling and running in interactive mode.



Conclusion

This project taught me much about how lower-level C commands work and how Unix commands function. I also explored file linking, external variables, and C file I/O on a level that I had not previously. Through testing my code I also got much more experience finding memory leaks with Valgrind and locating where the fixes should be implemented. I feel like this project has given me much more experience coding with C beyond the functions like printf and fputs that I was familiar with and forced me to code outside my comfort zone with new, more complicated functions.